

An infrastructure agnostic application deployment framework for the Internet of Things

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Peter Eder

Matrikelnummer 0926758

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar

Mitwirkung: Dr. Michael Vögler

Wien, 18. April 2017

Peter Eder

Schahram Dustdar

An infrastructure agnostic application deployment framework for the Internet of Things

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Peter Eder

Registration Number 0926758

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar

Assistance: Dr. Michael Vögler

Vienna, 18th April, 2017

Peter Eder

Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Peter Eder
Weimarer Strasse 3/3, 1180 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. April 2017

Peter Eder

Acknowledgements

I would like to thank my advisor Prof. Schahram Dustdar who gave me the opportunity to write this thesis at the Distributed Systems Group. Many thanks also to my co-advisor Dr. Michael Vögler who helped me in the course of writing this thesis. He always provided very constructive and fast feedback, and continued supervising even after he finished his PhD. Last but not least, I want to thank my parents, who made it possible for me to study Computer Science in Vienna and supported me throughout the past years.

Abstract

In the last years terms like *Smart Grid*, *Smart City* or *Smart Vehicle*, became increasingly popular. They all fall under the conception of the so called *Internet of Things (IoT)*. It is estimated that in 2020 will be up to 34 billion devices connected with the Internet.

However, the huge potentials of IoT come with various challenges. Among issues like privacy, security, scalability, etc., the infrastructure in IoT is a significant problem. The huge amount of different devices leads to a highly heterogeneous environment. Thus, developing applications that respect these heterogeneous infrastructures is exceedingly complex and in further consequence very error prone. A possible solution to this problem is to abstract the hardware layer through virtualization.

In the course of this thesis we investigate how operating-system-level virtualization can be used to cope with the heterogeneous environment. Therefore, commonly used IoT devices are presented and an in-depth explanation of how operating-system-level virtualization is implemented on different operating systems (e.g., Linux, FreeBSD, Solaris, Windows) is given. Furthermore, currently available container engines with focus on Linux are introduced and compared. We also explain the concept of continuous delivery, and why this development approach facilitates agile development.

To show the feasibility of operating-system-level virtualization in the context of IoT, a prototype of an application deployment framework is developed. This framework provides distribution and deployment of applications within one click. Furthermore, it allows to deploy on different CPU-architectures (e.g., ARM, x86) transparently to the user. Additionally, it can easily be integrated within a continuous delivery pipeline through a REST API. The features of the framework are demonstrated via the deployment of a Building Managament System (BMS). In the last part of the thesis, the deployment time of different applications on various devices is evaluated and discussed, which shows both, feasibility and applicability of the proposed approach.

Kurzfassung

In den letzten Jahren wurden Begriffe wie *Smart Grid*, *Smart City*, oder *Smart Vehicles* immer populärer. Sie fallen alle unter das Konzept des *Internet der Dinge (IoT)*. Schätzungen zufolge werden im Jahr 2020 bis zu 34 Milliarden Geräte mit dem Internet verbunden sein.

Das große Potenzial von IoT ist mit vielen Herausforderungen verbunden. Neben Aspekten wie Privatsphäre, Sicherheit, Skalierbarkeit, etc., ist die Infrastruktur des IoT ein erhebliches Problem. Die große Anzahl an unterschiedlichen Geräten führt zu einem sehr heterogenen Umfeld. Die Entwicklung von Applikationen, welche diese heterogene Infrastruktur berücksichtigt, wird dadurch äußerst komplex und fehleranfällig. Eine mögliche Lösung für dieses Problem ist die Abstrahierung der Hardwaresebene durch Virtualisierung.

Im Zuge dieser Diplomarbeit wird die Verwendung von Betriebssystemvirtualisierung in Verbindung mit dieser heterogenen Umgebung untersucht. In diesem Zusammenhang werden häufig verwendete IoT Geräte vorgestellt. Zudem erfolgt eine detaillierte Beschreibung der Virtualisierung in verschiedenen Betriebssystemen (z.B.: Linux, FreeBSD, Solaris, Windows). Neben der Vorstellung von aktuellen Container Engines mit Fokus auf Linux, wird auch das Konzept der Continuous Delivery erläutert.

Zur Demonstrierung der Anwendbarkeit der Betriebssystemvirtualisierung im Kontext von IoT, wird ein Application Deployment Framework entwickelt. Dieses Framework bietet die Möglichkeit Applikationen mit einem Klick zu verteilen und einzusetzen. Zusätzlich erlaubt es das Einsetzen auf verschiedenen CPU-Architekturen (z.B.: ARM, x86) und vereinfacht die Integration in eine bereits vorhandene Continuous Delivery Pipeline durch eine REST API. Die Funktionen des Frameworks werden durch den Einsatz eines Gebäudemanagementsystems veranschaulicht. Im letzten Teil der Diplomarbeit, wird die Verwendung von verschiedenen Applikationen auf unterschiedlichen Geräten hinsichtlich der Realisierbarkeit und Anwendbarkeit des vorgestellten Ansatzes evaluiert und diskutiert.

Contents

Abstract	ix
Kurzfassung	xi
Contents	xiii
1 Introduction	1
1.1 Problem Statement	1
1.2 Aim of the Work	2
1.3 Methodological Approach	2
1.4 Structure of the Work	3
2 Background	5
2.1 Internet of Things	5
2.2 Virtualization	9
2.3 Continuous Delivery	13
3 State of the Art	19
3.1 IoT Hardware	19
3.2 IoT Communication Standards	22
3.3 Operating-System-Level Virtualization	27
3.4 Container Deployment Frameworks	42
4 Design & Implementation	49
4.1 Features	49
4.2 Design	50
4.3 Implementation	58
5 Demonstration	63
5.1 Use Case Definition	63
5.2 Application Deployment and Management	63
5.3 Summary	67
6 Evaluation	69
	xiii

6.1	Scenarios	69
6.2	Benchmark	70
6.3	Discussion	77
7	Conclusion	79
7.1	Summary	79
7.2	Future Work	80
	List of Figures	83
	List of Tables	84
	Bibliography	85
A	Measurements	95
A.1	Notebook	95
A.2	Hosted Server	97
A.3	Raspberry Pi	99

Introduction

In the last years terms like *Smart Grid*, *Smart City* or *Smart Vehicle*, became increasingly popular. They fall under the conception of the so called *Internet of Things (IoT)*. Basically, the idea behind IoT is to integrate the virtual world of information technology seamlessly with the real world of things, as stated by Uckelmann et al. [114]. This means that things that surround us in our everyday life are interconnected with each other and can be used to gain information that is collected via the Internet.

The following numbers give some insights about the potential IoT has in the next years. The Business Insider [84] estimates that in 2020 will be up to 34 billion devices connected with the Internet. Where about ten billion devices will be out of the traditional computing domain (PC, smartphone, tablet, etc.) and 24 billion devices will be real IoT devices. Considering the world population estimation for 2020 [115] with 7.7 billion people living on the earth, there will be about 4.4 connected devices per person.

A typical IoT application would be *Assisted driving* [21], where interconnected cars and various sensors increase safety and efficiency through information exchange of current traffic, road conditions, etc. In context of healthcare a possible application is *Sensing* [21], which allows to monitor health conditions of patients remotely in real time, whereas they are not restricted in their mobility. These are only two examples, but the possible scenarios where IoT leads to a more intelligent environment are nearly endless.

1.1 Problem Statement

However, the huge potentials of IoT come with various challenges. Among issues like privacy, security, scalability, etc., the infrastructure in IoT is a significant problem. The huge amount of different devices leads to a highly heterogeneous environment. Thus, developing applications that respect these heterogeneous infrastructures is exceedingly complex and in further consequence very error prone. A possible solution to this problem

is to abstract the hardware layer by so called *Virtual machines* as described by Chen and Noble [32].

Beside infrastructure, another challenge is maintaining and managing IoT software. IoT devices are naturally highly distributed, thus, applications must be managed remotely. This means that a standardized interface to each node is needed to distribute and deploy applications automatically. Since modern software development embraces short development cycles with low risk releases this comes also in hand to establish a Continuous Deployment Pipeline as described by Humble and Farley [64]. In addition to deploying, it is also needed to monitor running applications. IoT applications are often deployed in a massive scale and therefore hardware failures are the norm, not the exception. When it comes to software bugs, tools are needed to overwatch running applications. Thus, developers need easy access to logging output for debugging purposes.

1.2 Aim of the Work

The aim of this work is to investigate how the concept of virtualization can be applied in the area of IoT to decrease the complexity of application deployment on highly heterogeneous environments. Therefore, we will discuss different kind of virtualization techniques that are currently available. We will also have a look on different frameworks to manage applications on distributed devices.

The gained insights will be used to develop a prototype that allows to deploy and manage distributed applications. There will also be an interface that makes it possible to integrate the deployment framework in an existing continuous delivery pipeline. Furthermore, we will evaluate the performance and feasibility of the implemented prototype in different scenarios.

1.3 Methodological Approach

The methodological approach for this thesis consists of the following steps:

- The first step is to analyze different implementations of operating-system-level virtualization. The differences will be evaluated in terms of applicability and feasibility for IoT.
- Second, currently available deployment frameworks are analyzed. This will give an overview of commonly supported features. The outcome will help in designing an application deployment framework for IoT.
- In the next step, the architecture of the application deployment framework will be designed and a prototype developed. The framework will follow a microservice architecture [96] and offer a web interface.

- After development, the performance of the prototype will be evaluated. These benchmarks will give insights about distribution and deployment time in various real world scenarios. The test scenarios will consider different image sizes and infrastructure types.
- Finally, the gained insights will be discussed in terms of applicability, performance, etc. in the thesis. Furthermore, an outlook on future work is given.

1.4 Structure of the Work

The remainder of this thesis is structured the following way:

- Chapter 2 provides a detailed description of the topics and basic concepts covered in this thesis. It introduces and defines IoT, explains different virtualization concepts in context of IoT and describes the concepts behind continuous delivery.
- Chapter 3 shows various IoT hardware and explains commonly used communication standards. Furthermore, it gives an overview of state of the art virtualization engines and explains different application deployment frameworks.
- Chapter 4 describes the design and implementation of the deployment framework prototype.
- Chapter 5 presents a typical use case and shows how the introduced framework can be used to efficiently deploy and manage applications in that context.
- In Chapter 6 an evaluation of the prototype is made in terms of performance. Therefore, deployment time of several images is measured in context of different infrastructures.
- Chapter 7 presents the outcome of this thesis and gives an outlook on future work.

Background

In this chapter we discuss several topics which are fundamental for the remainder of this thesis. First, we will introduce the Internet of Things and explain applications and important challenges. Second, we explain the concept of virtualization and give an overview about the different types of virtualization. Finally, the last part describes continuous delivery, a modern software development approach. It should give insights why application deployment frameworks are essential for modern software development.

2.1 Internet of Things

It is considered that the term *Internet of Things (IoT)* first came up in a speech hold by Peter T. Lewis in 1985 [109]. Later, in 1999, Kevin Ashton describe the shift from humans who enter data into machines to machines, which can observe and identify the world by their own [20]. In the last couple of years IoT gained more and more interest because of increased miniaturization of computers and ubiquitous Internet access. In 2008 the U.S. National Intelligence Council named the IoT one of the six *Disruptive Civil Technologies* for the future. The EU is investing 192 million Euro in the IoT research in the years from 2014 to 2017. Also China is investing enormous amounts of money to become a global leader in deployment of the Internet of Things.

2.1.1 Definition

The term IoT is widely used, but it is not easy to give a clear explanation what the IoT really means. Today, several different definitions are used depending on the stakeholders perspective. Atzorri et al. [21] describe three different kind of visions that converge in the IoT paradigm: the things-oriented vision, the Internet-oriented vision, and the semantic-oriented vision.

The things-oriented vision comes from a perspective with simple objects like Radio-Frequency IDentification (RFID) tags. Therefore, it is mainly focused on traceability of objects and current properties like location, state, etc. On the other hand the Internet-oriented vision is focused on the Internet Protocol for connecting objects and reusing of web standards to connect surrounding objects with the Internet. The idea behind the semantic-oriented vision is that the enormous amount of connected things and therefore the large amount of collected data will be challenging to analyze and process. Thus, semantic technologies could solve this issue with smart organization of the data. The full potential of the Internet of Things can only be reached as a confluence of this different kind of perspectives.

A commonly used definition for the Internet of Things is explained by Bassi and Horn. The authors define the IoT as "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" [22]. Another, more general definition was stated by Gubbi et al. They describe the Internet of Things as an "Interconnection of sensing and actuating devices providing the ability to share information across platforms through a unified framework, developing a common operating picture for enabling innovative applications. This is achieved by seamless large scale sensing, data analytics and information representation using cutting edge ubiquitous sensing and cloud computing." [57].

2.1.2 Applications

Although the number of possible applications for the IoT is large, they can be grouped in the following domains [21]: *Transportation and logistics*, *Healthcare*, *Smart environment*, and *Personal and social*.

Transportation and Logistics

A currently very popular application is assisted driving. Sensors and actuators used in vehicles and the road itself lead to an enormous gain on information. This data can be used to optimize routing depending on real time traffic conditions. Thus, assisted driving leads to more efficient and safer transportation.

When it comes to logistics, the IoT brings a significant improvement of organization. Since transported goods can be easily equipped with low cost RFID tags, it is possible to track down the current position of each object in the supply chain. This helps identifying bottlenecks and increases throughput.

Healthcare

A very important application is remote monitoring of patients. Sensors that track vital functions, can provide real time information about the current conditions when used in context of the IoT. Real time monitoring provides the possibility of faster life-saving action in case of emergency. Also the patient is not bound locally, which offers a huge

increase in quality of life for certain groups of patients. Furthermore, it also decreases costs since patients do not have to be on-site all the time and therefore capacities in hospitals are freed-up.

As for logistics, another applications is tracking of patients. This information can be used to optimize organizational processes inside of hospitals and improve waiting time and in further consequence customer satisfaction.

Smart Environment

The purpose of smart environments is to increase comfort and efficiency through intelligence of involved objects. An application in this domain that is nowadays widely covered in the media is the smart home. Smart homes allow intelligent regulation of home facilities. Some examples are: light regulation according to daytime; heat regulation in context of weather conditions; automatic power off of unused electronic devices; etc.

Personal and Social

A very simple, but useful application in the personal domain is searching for lost objects. For this purpose, the last tracked position of a tagged object can be used. When it comes to social networking a use-case is automatic updating of current activities. Attending of events for example can be automatically updated on location based services. This is of course a highly controversial topic since it raises significant privacy issues that offer the possibility of total supervision.

2.1.3 Current State

According to Uckelmann et al. [114] the evolution of the IoT follows a phased approach. Starting from the Intranet of Things, evolving to the Extranet of Things and finally becoming the actual Internet of Things. The first phase can be characterized by many small intranets of things, where lots of different technologies are used and a lack of standardization is prevalent. These networks cannot be accessed globally. The Extranet of Things offers limited access for external individuals, but is still lacking the global scope. Bassi et al. [126] describe the last phase, the true Internet of Things, as "a globally integrated system". This implies that each device can be accessed universally. Uckelmann et al. [114] state that currently most implementations are still in the first or second phase.

2.1.4 Architecture of the IoT

Want et al. [121] describe two different architectures for the IoT: a centralized IoT service and a peer-to-peer approach. Using a centralized approach, all devices are connected with the Internet and the services are communicating directly with the devices. In general, centralized services are easier to manage and have better scalability compared to a distributed approach. However, while some of the devices have the capability to connect to the Internet directly, a big part of devices are ultra low power sensors and

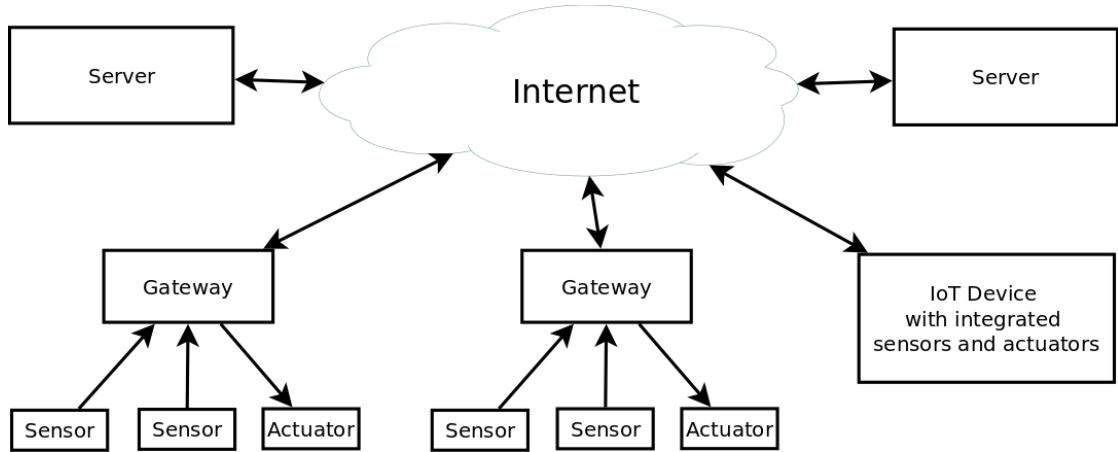


Figure 2.1: Example of a network architecture for the IoT

actuators with no or only less computing possibilities. RFID tags, for example, are typically passive and cannot communicate directly with the Internet. Therefore, often some kind of bridge is needed to make this devices universally accessible. A peer-to-peer architecture with IoT gateways is used exactly for this purpose. Figure 2.1 shows an example of an IoT architecture, where gateways are used to access actuators and sensors through the Internet. As for RFID tags, the RFID readers can be seen as their gateway to the Internet [41]. Other actuators and sensors can make use of standards designed for communication of low power devices like ZigBee or 6LoWPAN to communicate with each other and the gateway [53].

Enabling Technologies

Putting the vision of the IoT into the real world requires enabling technologies from several different areas. Atzorri et al. [21] describe two major categories: *Identification, sensing and communication technologies* and *Middleware*. As mentioned before, a key technology for IoT will be RFID systems. RFID tags are of very low cost and do not require batteries, which means they can be used over a very long period of time. The problem of RFID tags is that they offer no processing power and no advanced sensing capabilities. Basically, RFID tags can only be used to check the presence of things. Another important technology will be wireless sensor networks (WSN). WSNs are networks with usually a high number of sensing nodes, which communicate with each other in a peer to peer fashion. WSNs have more advanced processing and sensing capabilities compared to RFID systems, but require a battery. RFID sensor networks (RFNs) are the combination of both technologies, RFID systems and WSNs. RFNs support sensing and processing, but are still passive devices, which harvest their energy during communication. The disadvantage of this technology is the low range it offers, which is usually only a few meters. Middleware makes the low power devices universally accessible. Furthermore, it helps to abstract the different technologies and thus, simplifies

software development.

According to Gubbi et al. [57], the ability to identify and address IoT devices by a unique ID is a further key technology to establish the IoT. However, the address space of IPv4 is almost exhausted. Its successor, IPv6 [38], allows theoretically about $3.4 * 10^{38}$ different addresses. Although IPv6 was proposed a couple of years ago by the Internet Engineering Task Force (IETF), it still has a low degree of prevalence. In order to be able to address every IoT device, establishing IPv6 is needed.

Important Challenges

Beside technologies that are needed to spread usage of the IoT we also have to address important open research issues. Although there are many problems that still are not solved we will focus here on issues concerning standardization, privacy, and security.

As mentioned earlier, the lack of standardization is one of the biggest problems in the IoT. Standardized interfaces for sensors and actuators would help to decrease costs of application development. Currently, many proprietary solutions are available that lead to a highly heterogeneous environment.

When it comes to security the IoT is at high risk of attacks, because of several reasons as described by Atzorri et al. [21]. First, IoT devices are typically deployed at a massive scale, thus they can not be overseen directly most of the time. This makes it possible for attackers to gain direct access physically. Second, IoT devices are mostly using wireless connections to communicate with each other. And third, they are usually of very low processing capability (e.g., RFID tags). These characteristics require new security concepts to build common trust in the new technologies. Although, significant effort in terms of research in this area has been taken, the proposed solutions often require sophisticated cryptographic encoding and are not feasible for IoT devices with low processing power.

Another important challenge of the IoT is privacy. This is a very sensitive topic, since IoT devices are everyday things and therefore offer access to very private data (e.g., vital function, localization data, etc.). This could lead to universal supervision in a way that has never been possible before.

2.2 Virtualization

To understand what virtualization in context of computer systems means, we first have to distinguish between emulation and virtualization. The difference has been explained by Lowe in [92]. Using emulation the complete hardware is simulated with software. This allows, for example, to run software that is only compatible for system A to run on another system B, that is emulating system A. The architecture of system A and B can differ in every aspect, as long as an emulation for system A is available. Since old architectures can be simulated on new systems, it leads to a higher compatibility, but comes with cost of performance. Although virtualization also uses some kind of scheduler

to access hardware resources, these accesses are done directly. Thus, it has a significant better performance compared to emulation. However, it is limited to software that is built for the underlying hardware. Virtualization implementations often also use some form of emulation for specific hardware resources to increase compatibility.

Nanda and Chiueh [33] define virtualization as follows: "Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others".

Virtualization was first used in the 60s by IBM, in the area of mainframes [33]. It was developed to increase utilization of the expensive systems via concurrent access. Over the last decades significant effort in terms of research and optimization was taken, to improve virtualization technologies. Today modern servers are virtualized, because of several inherent benefits [124]:

- **Server consolidation:** Several under-utilized servers can be virtualized on one physical machine. This increases efficiency and, thus, decreases hardware expenses. Furthermore, maintenance costs are reduced when less physical machines are needed.
- **Fault tolerance:** Untrusted programs can easily be isolated to protect the physical resources. This can also be done to create secure environments for end users.
- **Intrusion detection:** Virtualization can be used to set up an environment as a so called *honeypot* to catch intruders.
- **System migration:** Virtualized environments, are a software implementation of a real machine, and therefore can easily be transferred to different physical machines. This leads to lower down time in case of hardware failure.
- **Virtual appliance:** Already set-up applications, packed within their needed environment (operating system, database, etc.) allows costumers fast and easy deployment of new software. Since the environment is standardized and known by the vendor, updating and bug fixing of their applications is less error prone.
- **Debugging and testing:** Virtualization offers the possibility to clone production systems and create identical test systems within no time. This makes developing and testing in production environments very easy and leads to better software quality.

Virtualization is basically implemented with an added virtualization layer somewhere between hardware and application. The control instance at the virtualization layer is called a Virtual Machine Monitor (VMM), or sometimes referred to as Hypervisor. There are different virtualization concepts available, depending on where the layer is put and how resources are virtualized. Each approach has its specific use case and is always a trade off between efficiency, isolation, reliability, and compatibility [111, 123].

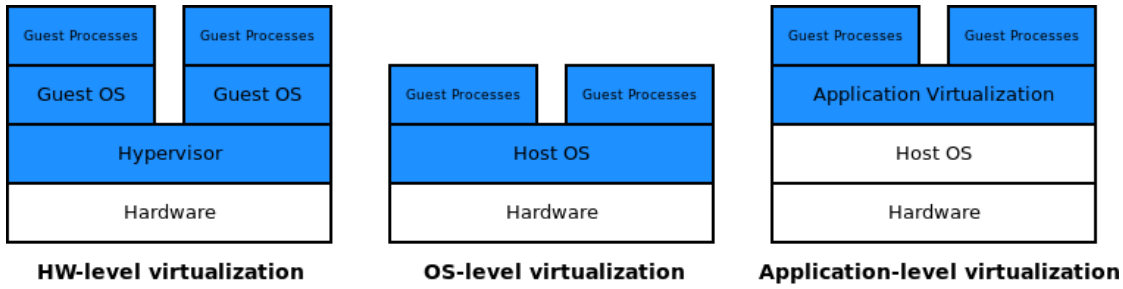


Figure 2.2: Different levels of virtualization

Goldberg and Popek [99] described three fundamental properties that are important for virtualization: *efficiency*, *resource control*, and *equivalence*. Efficiency in this context means all safe instructions are executed directly by the hardware without use of the virtualization layer. Resource control is defined as complete control of the resources by the Hypervisor. Thus, guest operating systems cannot allocate resources through bypassing the virtualization layer. This also means that every program executed inside a virtual environment has the same access to privileged instructions as executed without virtualization. Although these requirements were defined in 1974, and computer systems and architecture changed since then, they are still valid [97].

The classification of the different virtualization concepts is not clearly defined in the literature. We will follow the taxonomy as described by Nanda and Chiueh [33], where the concepts are grouped by the level of the abstraction layer. Furthermore, we will focus on *hardware-level virtualization*, *operating-system-level virtualization*, and *application-level virtualization*. Figure 2.2 gives an comparison of the different levels. The name *guest operating system* stands for the operating system that runs in a virtualized environment, the *host operating system*, on the other hand, is the operating system that hosts the virtual machines.

2.2.1 Hardware-Level Virtualization

In hardware-level virtualization, the virtualization layer sits right on top of the hardware layer. The virtualized environments with their isolated operating systems are often called *Virtual machines*.

There are several different kinds of virtualization techniques available on the hardware-level. The two most commonly used are *Native virtualization* and *Paravirtualization*.

Native Virtualization

Native virtualization, or sometimes called full virtualization, allows to run operating systems and software as if they would be installed on original hardware [123]. This means that the guest operating system in the virtual machine is not aware of the underlying virtualization. Native virtualization is achieved through a compromise of

emulation and direct hardware access. This technique leads to a very high compatibility of operating systems since there is no adaptation needed. However, this comes with possible performance issues, in cases where instructions mainly have to be emulated instead of directly processing by the hardware.

Paravirtualization

Gribble et al. [122] introduced Paravirtualization as a lightweight alternative to native virtualization that offers better scalability, performance, and simplicity. Basically, in Paravirtualization guest operating systems are aware of the virtualization of the hardware and thus, can give more information about current state to the Hypervisor. The Hypervisor uses this information for more intelligent assignment of resources to each virtual machine. For example, if the operating system is idle, normally the guest operating system would still waste resources in an idle loop, but since the guest operating system can inform the Hypervisor, resources can be assigned to another virtual machine. The main disadvantage of Paravirtualization is that the guest operating system has to be adapted for usage.

2.2.2 Operating-System-Level Virtualization

As described by Chan et al. [119] operating-system(OS)-level virtualization uses a modified operating system that allows running multiple isolated instances. These instances are often called *containers* or *jails*. Containers share the kernel of the host system and thus, are bound to its kernel version. Although, all containers share the same kernel, each instance can be booted, shut down and rebooted the same way as a normal operating system [111]. Isolation of containers is achieved through various OS-kernel features depending on the implementation. OS-level virtualization has the advantage of very little virtualization overhead [44]. Thus, performance is close to the native approach without virtualization. Furthermore, little resources like memory and CPU are wasted. The containers are also very lightweight when it comes to disk space, because it is not necessary to have a whole clone of the operating system for each container. The main drawback of this method is that since all containers are using the same kernel it is not possible to run operating systems with different kernels. Another problem is that the kernel is a single point of failure, since a crash of it causes a crash of all containers.

2.2.3 Application-Level Virtualization

Application-level virtualization puts the virtualization layer between host operating system and the application itself [107]. The application layer is usually an application that runs on top of the host operating system. Virtualized applications are then executed by this virtualization layer. Each application is executed in its own virtual environment. The abstraction of the underlying system can offer high compatibility where the virtualized application can be executed on different operating systems without any modifications. Furthermore, it can also be used as a sandbox to execute untrusted applications in a safe

environment. Since only the application itself is virtualized, it uses significantly less disk space compared to a container or virtual machine. Another advantage is that multiple versions of the same application can be run simultaneously since they are isolated from each other and no installation conflicts can occur. On the other hand, it only allows to isolate single programs but not a whole operating system instance. Typical example for an application-level virtualization implementation is the well known Java Virtual Machine (JVM).

2.3 Continuous Delivery

Modern software development requires fast adaptation on changing requirements, bug fixing, customer feedback, etc. Extreme programming, Scrum, DSDM, Crystal, Feature-Driven Development are some example techniques to achieve such an agile environment. In 2001 *The Agile Alliance*, a group of leading software development experts, defined the *Agile Manifesto* [24]. The Agile Manifesto is composed of four values and twelve principles that lead to agile software development. The first principle is defined as: "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software." Thus, continuous delivery of software is an integral part of an agile culture. Fowler [49] describes continuous delivery as "a software development discipline where you build software in such a way that the software can be released to production at any time."

Farley and Humble explain the principles and methodologies of continuous delivery in [64]. They state that the main goal is to "deliver high-quality, valuable software in an efficient, fast, and reliable manner."

2.3.1 Why Continuous Delivery?

Releasing new software is often a manual process. Since modern software projects tend to be very complex with numerous dependencies, every step in this procedure is very error prone. Doing it manually means that every release is very risky because of potential human mistakes. Therefore, a release is very stressful for everybody who is involved. Furthermore, because deploying new software into production takes so much effort, software teams are likely to make only a few, but big releases. The problem with big and infrequent releases is that feedback takes a long time. Thus wrong assumptions are fixed in a late stage of development, which means a lot of effort compared to early stages. Another common problem software projects often face is that production environments are configured manually by the operations team. This leads to test systems that differ from production and in further consequence, tests are not reliable anymore.

Continuous delivery can solve these problems. In detail, this can be achieved through frequent, automated releases. Automation is needed to make the whole process from building to releasing, repeatable and standardized. As a further consequence automation reduces errors since reviews and improvements can be done transparently and every release

is done in the exact same way. A high level of automation can only be reached through extensive test automation, which is also an inherent part of continuous delivery. Frequent releases, on the other hand keep the delta between releases low. Small, incremental releases have the benefit that the risk of releasing software decreases significantly. Furthermore, if new functionality or changes are released frequently, we get quick feedback from users, and thus, can adjust fast if the product is not meeting their expectations. Summarized, continuous delivery brings several benefits for software development [63]:

- Low risk releases: As described, automation removes potential human errors and makes releasing standardized and repeatable.
- Faster time to market: The ability to make frequent releases requires a fast way to bring new software into production. Therefore, development teams are forced to improve the whole development process from building to releasing. Continuous delivery makes bottlenecks visible since releasing is way more transparent compared to traditional approaches. Ultimately, new functionality is deployed faster, because of constant improvements.
- Higher quality: Reliable automated tests give fast feedback about correct implementation of functional requirements. This allows development teams to focus on high level testing of, for example, usability, performance and security, which ensures a higher quality of the software.
- Lower costs: Establishing continuous delivery takes some effort at the beginning since a high degree of automation and test coverage is required. But in the long run it pays off, because the fixed costs of releasing are significantly lower compared to manual releasing.
- Better products: Small, incremental releases allow fast customer feedback, which helps to meet their needs.
- Happier teams: Since releasing becomes an effortless everyday task, the typical stressful days before a deadline vanish. Reducing or even removing painful and error prone tasks that had to be done by developers in order to release new software helps to keep them motivated since they can focus on the interesting parts like developing of new features.

2.3.2 Establishing Continuous Delivery

Basically, three concepts build the foundation for establishing continuous delivery [63]: *configuration management*, *continuous integration*, and *continuous testing*.

Configuration Management

Farley and Humble [64] define configuration management as follows: "Configuration management refers to the process by which all artifacts relevant to your project, and the relationships between them, are stored, retrieved, uniquely identified, and modified."

The key element of configuration management is universal version control. We need to version control everything that is required for the software project. This means source code, test and deployment scripts, infrastructure configuration, application configuration, project dependencies, project documentation, etc. are kept in a central version control system, that is accessible by the whole team. The two main goals of configuration management are reproducibility and traceability. Reproducibility means it should be possible to automatically provision an environment and get identical copies if needed at anytime. Traceability refers to the ability to determine the version of every part, which was used to create the environment. Furthermore, it should be possible to easily compare different versions of the environment.

Continuous Integration

The main idea of continuous integration [48, 42] is that every developer integrates his work to the main branch frequently. Frequently means that developers merge with the mainline at least daily. While integrating regularly, it has to be ensured that the main branch is in a working state all the time. If the mainline breaks, it is fixed immediately. Continuous integration was first brought up by Kent Beck as a part of XP (extreme programming) [23]. It guarantees that developer branches are not drifting away from trunk. Since small changes are integrated into the main branch frequently instead of big changes rarely, merging with trunk becomes less error prone. However, establishing continuous integration comes with some challenges that have to be addressed. For example, since only code that can be built and is tested, is allowed to be merged into trunk, developers tend to commit very infrequently or stick to a low degree of test coverage. In general, continuous integration requires a cultural shift in the development team. There are several practices, which are needed to establish effective continuous integration [64, 48]:

- Check in regularly: Committing frequently is the fundamental practice of continuous integration. It has to be ensured that all developers are using the main branch and not separate feature branches, which would undermine the whole concept.
- Automated testing: Since testing has to be done after each commit, there is a high degree of automation needed. Manual tests would slow down the building and testing and is not feasible. To make sure that the main branch is in a working state, a high degree of test coverage is needed.
- Keep the build fast: Building is done after every commit. Thus, building and testing of the code needs to be fast so that the developers get rapid feedback about

their code. This can be quite challenging but it is a very important aspect when it comes to continuous integration.

- Never check in if building fails: Developers have to synchronize with the trunk branch and test their build locally or on a integration machine before they merge with the master. If the build fails, they are not allowed to merge.
- Fix immediately: If building of the mainline fails, it has to be fixed immediately. In case finding a solution to the problem takes longer than ten minutes the mainline should be reverted to its last successful build.
- Test in an environment that is identical to production: To get confidence in the tests, it is essential to test in a clone of the production environment. Otherwise there is always a significant risk that a successful build breaks in production.

Continuous Testing

Humble explains the idea of continuous testing as follows: "Our goal is to run many different types of tests — both manual and automated — continually throughout the delivery process." [62]. The different types of tests include automated tests like unit tests, component tests, system tests, functional and nonfunctional acceptance tests, etc. and manual tests like usability testing or nonfunctional acceptance tests that cannot be automated. Thus, continuous testing is not only about test automation, it is about implementing a comprehensive testing strategy. However, the automation of tests should be as high as possible. Since fast feedback is needed the first tests should run really fast. Therefore, unit tests should be run first and give feedback within a couple of minutes. If this tests pass, more complex automated tests can be run. Only if all automated tests are successfully executed, time-consuming manual tests are carried out. The goal is to achieve a high test coverage and variety of different kind of tests so that developers have confidence that the application can safely be deployed to production. This is essential for frequent deployment.

The Deployment Pipeline

As described above, automation of building and testing is fundamental for continuous delivery to get fast feedback. The problem is that extensive tests can take a long time to complete and some tests cannot be automated. Modeling the delivery process as a *Deployment pipeline* helps to overcome this problem by splitting up the whole process from compiling to deployment in several stages as stated by Fowler [50]. Figure 2.3 shows an example of a typical deployment pipeline. It was first mentioned by Humble et al. [65] in 2006. The deployment pipeline consists of several steps where each step can be seen as a barrier that has to be taken so that the application can proceed to the next stage. The feedback takes longer the more stages are passed. Therefore, the first stage, often called commit stage, checks if the application can be build successfully and runs only fast tests. Humble [61] states that it should not take longer than 10 minutes to pass the

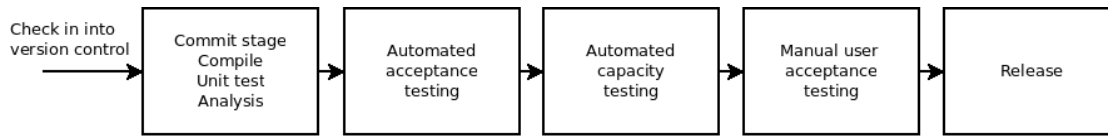


Figure 2.3: Deployment pipeline, adapted from [64]

commit stage. The next stage carries out more sophisticated tests, as integration tests, performance tests, etc. To make processing faster, some stages can be parallelized in a way that complex tests are executed simultaneously.

Maturity of Continuous Delivery

To get clear insights about how well continuous delivery is established in the development process, some metrics are needed. It is important to make the improvements and the progress during implementation of a deployment pipeline visible. An often used metric is the *cycle time*. Farley and Humble [64] define the cycle time as "the time from deciding that you need to make a change to having it in production". Thus, the goal is to keep the cycle time as low as possible, while keeping a high test coverage which ensures that quality requirements are met.

2.3.3 Continuous Delivery vs. Continuous Deployment

A concept that is often used interchangeably with continuous delivery is continuous deployment [64]. Although, many people use them as synonyms there is an important difference between them. While the goal of continuous deployment is to bring every change that passes the deployment pipeline into production, continuous delivery only guarantees that deployment can be done whenever wanted. This means that the last step of the deployment pipeline, deploying the software into production, is done manually in continuous delivery and automatically when continuous deployment is implemented. Thus, continuous deployment relies on a well established continuous delivery process and goes one step further. It can be seen as the evolution of continuous delivery. However, for some projects it is not feasible to release new software several times a day. Therefore, deciding which software development approach fits best always depends on the software project.

State of the Art

This chapter gives an overview about currently available IoT hardware and modern communication standards that are used in the IoT domain. Furthermore, it shows how operating systems support virtualization in detail and describes the features of popular container engines. The last section presents state of the art application deployment frameworks.

3.1 IoT Hardware

In the last years many different IoT hardware platforms were introduced. We will present some common devices that are used in context of IoT development in this section. Therefore, we focus on three different types of devices: sensors, gateways, and development platforms.

3.1.1 Sensors

There are many different sensors for all kind of applications available. Some examples are temperature sensors, humidity sensors, light meters, accelerometers, magnet sensors, motion sensors, etc. These sensors are mostly sold as microchips and come without any further computational power or higher level connectivity. Thus, a sensor node is needed, which is directly connected to the sensor or in a low distance when RFID is used. A sensor node allows to access the sensors through various interfaces like WLAN, ZigBee, etc. An example for such a node is the Arduino Uno [15]. Table 3.1 gives an overview of its specifications. Although the Arduino is mostly used for development purposes, its specifications should give some insights about the typical capabilities of sensor nodes.

Arduino Uno Revision 3	
Microcontroller	ATmega328
Clock Speed	16MHz
Digital I/O Pins	14
Analog Input Pins	6
Flash Memory	32KB
SRAM	2KB
EEPROM	1KB
Price	approx. 8\$

Table 3.1: Specifications of the Arduino Uno Revision 3 [15]

Dell Edge Gateway 5000	
Processor	up to 1.75GHz
Architecture	x86 64bit
RAM	up to 8GB
Storage	up to 256GB
Supported OS	Ubuntu Core 16, Snappy Ubuntu 15.04, Intel Wind River Linux 3.1, Windows 10 IoT, ...
Supported Interfaces	WLAN, Ethernet, LTE, CAN Bus, ZigBee, HDMI, ...
Price	starting from approx. 900\$

Table 3.2: Specifications of the Dell Edge Gateway 5000 [39]

3.1.2 Gateways

Typically, sensor nodes are cheap, but have very limited computing power, hence, they cannot communicate with the Internet in a sufficient manner. Therefore, so called gateways are used as a bridge between sensor nodes and the Internet. When IoT gateways were introduced, they were mostly used as plain bridges without any further processing. However, nowadays many gateways allow complex processing of collected sensor data. An example for a commercial gateway is the Dell Edge Gateway 5000 [39]. Table 3.2 gives an overview of its specifications. Gateways often run a full OS and, hence, allow to run custom applications as explained by Vögler et al. [118]. Furthermore, it is also possible to connect sensors directly to the gateways through input/output pins.

3.1.3 IoT Development Platforms

IoT development boards became popular for prototyping and are often used in IoT hobby projects. Two well known IoT development platforms are the Raspberry Pi [47], and the UDOO NEO [82]. While these boards come with powerful processors, they also allow to connect sensors directly through input/output pins and support a wide range of different

Raspberry Pi 3 Model B	
Processor	1.2GHz quad-core
Architecture	ARMv8
RAM	1GB
Storage	MicroSD-Card
Supported OS	Raspbian, Snappy Ubuntu, Ubuntu Mate, Windows 10 IoT, ...
Supported Interfaces	WLAN, Ethernet, Bluetooth Low Energy, HDMI, I2C, UART, ...
Input/Output Pins	24
Price	approx. 30\$

Table 3.3: Specifications of the Raspberry Pi 3 Model B [47]

UDOO Neo Full	
Processor	1GHz
Architecture	ARMv7
RAM	1GB
Storage	MicroSD-Card
Supported OS	Android, UDOObuntu2, ...
Supported Interfaces	WLAN, Ethernet, Bluetooth Low Energy, HDMI, I2C, UART, CAN Bus, ...
Input/Output Pins	32
Price	approx. 60\$

Table 3.4: Specifications of the UDOO Neo Full [82]

interfaces out of the box. Both boards support Linux and other operating systems. These aspects and the cheap price make them very well suited for prototyping and even for use in production environments. Table 3.3 and Table 3.4 show some specification details of these boards.

The UDOO Neo comes out of the box with integrated accelerometer, magnetometer, and digital gyroscope. Since both boards have powerful processors they can be used for many different purposes. The Raspberry Pi, for example, is often deployed as a gateway in hobby projects. Moreover, they can be easily extended with additional functionality through standardized interfaces.

3.1.4 CPU Architectures

As shown above, many IoT devices use processors with a different architecture, compared to traditional x86 CPUs. The so called ARM architecture is a Reduced Instruction Set Computing (RISC) architecture, while x86 processors use the Complex Instruction Set

Computing (CISC) architecture. The difference between this two designs is described as follows by Afuah [14]:

"In the design of CISC processors, a primary goal in instruction set design was to have so-called semantically rich instructions—instructions that get the hardware of the CPU to do as much as possible per instruction, moving as much of the burden of programming—of closing the semantic gap between human and computer—as possible from software to hardware. RISC technology calls for the opposite—simple instructions that get the hardware to do less per instruction thereby moving the programming burden from hardware back to software. With their simpler instructions, RISC microprocessors take up less chip real estate, *ceteris paribus*."

These fundamental design differences lead to incompatibility of software between the two architectures. This means, software that is compiled for the x86 architecture cannot be run on an ARM processor. Therefore, applications have to be recompiled for the specific platform.

Java, on the other hand, allows portability since it uses the JVM to abstract from the underlying architecture. However, platform independence breaks when native libraries or the Java Native Interface (JNI) are used.

3.2 IoT Communication Standards

IoT devices, especially sensors and sensor nodes, are typically distributed devices with low computing power. Furthermore, they are often operated by batteries, and thus, need to be optimized for low energy consumption. In this section we will describe various communication standards that take these characteristics into account and hence, are well suited for IoT applications. Figure 3.1 shows an overview of IoT standards and their corresponding layers in the OSI model [58].

3.2.1 Datalink and Physical Layer

IEEE 802.15.4

The IEEE 802.15.4 [13] standard defines the physical layer and the datalink layer for Wireless Personal Area Networks (WPANs). The protocol focuses on low energy consumption, secure communication, and low cost of hardware. The data rate can reach up to 250kbits/s, depending on the frequency band used. The transmission range is usually between 10 and 200 meters and the peak current consumption is <15 mA [113].

There are two different node types: full-function devices (FFD) and reduced-function devices (RFD). FFDs can be used as common nodes or coordinators. They are able to talk to all other devices and can relay messages. RFDs are simple devices with very limited computing power. They can only be used as common nodes and are only able to communicate with coordinators.

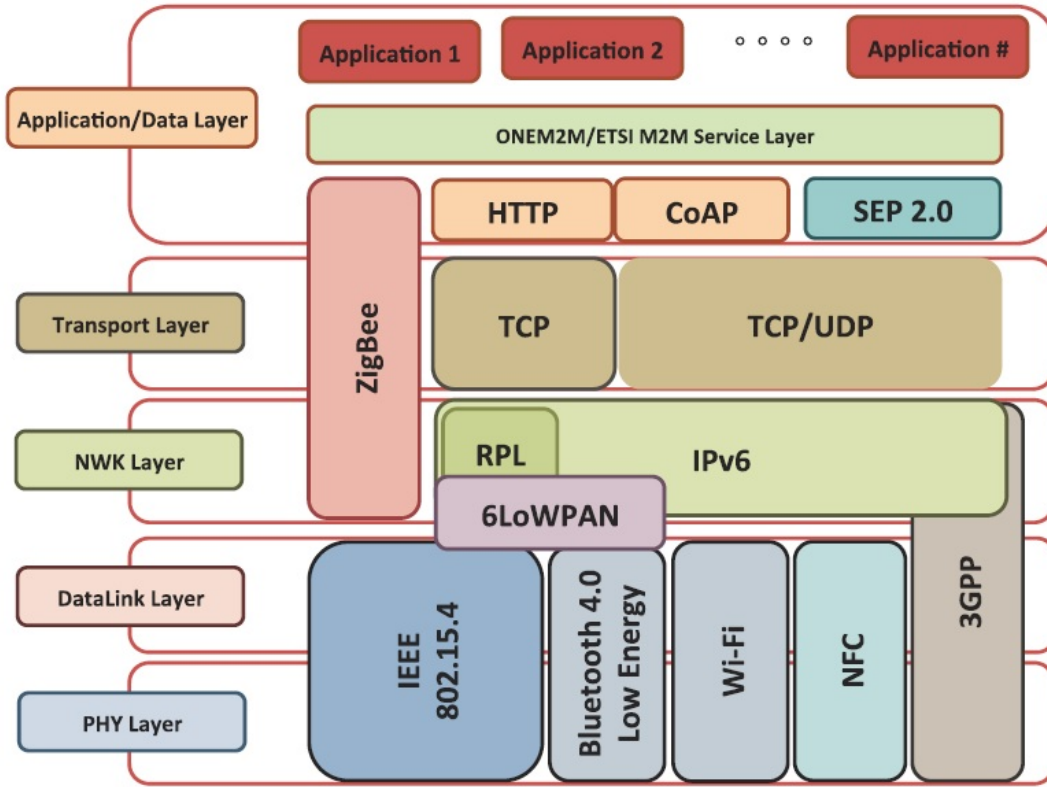


Figure 3.1: Standards used in IoT applications [58]

IEEE 802.15.4 supports peer-to-peer and star network topologies. Each network needs at least one coordinator that organizes communication.

Bluetooth Low Energy

Bluetooth Low Energy was introduced by Nokia in 2006 as Wibree. It was added to the Bluetooth specification in 2010 and renamed to Bluetooth Low Energy (BLE) [25]. The main idea behind BLE is that it only sends data and connects when required. This means that there is no constant connection, which lowers the power consumption tremendously. This suits very well for IoT applications, where sensors only need to transfer their state infrequently. For example, a temperature sensor, may only update the current temperature once every minute or less and is most of the time in idle mode.

The supported data rate of BLE v4.2 is up to 1 Mbits/s, the range can be up to 100 meters and the peak current consumption is <15 mA [113]. The network topology in BLE is called star-bus or tree network. A star-bus network consists of master and slave nodes, where each slave is connected to a master using a star topology and each master is connected with the other masters through a bus.

The next version, Bluetooth 5, was unveiled in June 2016. Among other things it increases the range significantly and doubles the speed compared to the old version [68].

RFID and NFC

Radio-Frequency Identification (RFID) is a term that describes technologies that use electromagnetic fields to identify and localize objects [120]. RFID uses so called tags that are attached to objects and readers that can read out information stored in the tags. There are two different types of tags available: passive and active. Passive tags harvest the energy during communication and therefore, do not need any battery. However, this limits the maximum distance between reader and tag, which is typically between 1 and 12 m. Active tags, on the other hand, have a power supply (e.g. battery) and thus, can communicate up to 200 m depending on the implementation. There are several standards specified depending on the application. For example the ISO/IEC 14223 specifies a standard for animal identification and the ISO/IEC 18000 defines specifications for item management. RFID devices mainly use three different frequency ranges: Low Frequency (LF) (125 - 134 kHz), High Frequency (HF) (13.56 MHz), and Ultra High Frequency (UHF) (856 MHz - 960 MHz).

The Near Field Communication (NFC) [85] standard is based on existing RFID protocols and extends them further. It operates within the High Frequency range of RFID, but requires devices to be in a very close proximity. Typically, NFC devices must be in a range of about 10 cm to each other to be able to communicate. The low range was a design decision that was made to enable NFC for sensitive applications that require explicit user actions (e.g., contact-free payments). NFC devices can operate in three different modes [45]: reader/writer, peer-to-peer, and card-emulation. Reader/writer mode allows the NFC device to read from tags and write data to them. Peer-to-peer enables NFC devices to communicate interactively. Card-emulation mode lets a NFC device act as a passive smartcard and therefore, allows other readers to read out data.

Others

Common WiFi standards (e.g., 802.11a, 802.11n, etc.) are widely established, but they are not well suited for IoT applications because of low range and high power consumption. The WiFi Alliance announced a new standard for IoT communication called WiFi HaLow in early 2016 [16]. It allows a range of up to 1 km and is highly optimized regarding power consumption. The standard is not yet fully approved, according to the official IEEE Working Group Project Timelines [19].

The 3rd Generation Partnership Project (3GPP) is an association of telecommunication institutions. The goal of the collaboration is to provide world wide standards for telecommunication. The 3GPP defined the specifications for GSM, LTE, LTE-Advanced, UMTS, etc.

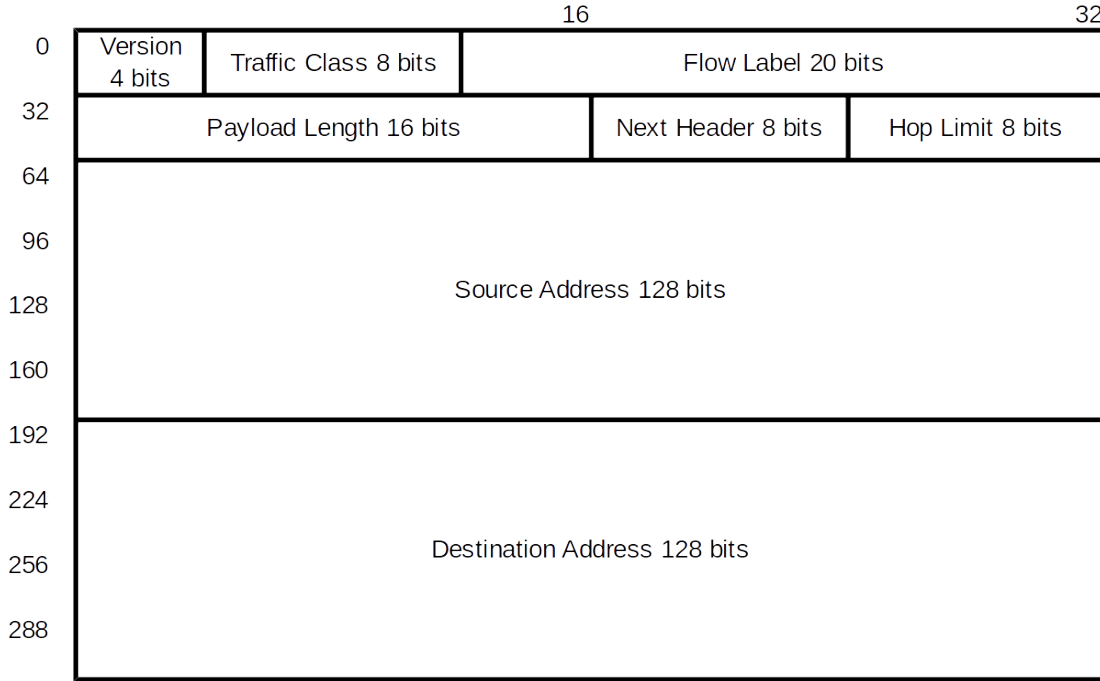


Figure 3.2: The IPv6 header (adapted from [38])

3.2.2 Higher Layers

IPv6 and 6LoWPAN

IPv6 [38] was introduced in 1998 because of the upcoming scarcity of IPv4 addresses. As explained in Chapter 2, IPv6 allows to address more than 3.4×10^{38} different devices. An IPv6 address is represented as eight groups of four hexadecimal digits, where the groups are divided by colons (e.g., 900f:01bd:4233:2222:0000:8a2e:d3ba:1124).

An IPv6 packet consists of the header and the payload. Figure 3.2 shows the composition of an IPv6 header. It has a minimum length of 320 bits and can be extended if needed. The payload field in the header has a length of 16 bits and thus, allows a maximum payload of 65535 bytes. There also exists the possibility of larger payloads, but this feature is rarely used.

A problem that comes up when IPv6 is used for IoT applications is that it defines a Maximum Transfer Unit (MTU) of 1280 bytes, while for example IEEE 802.15.4 defines a MTU of 127 bytes on the physical layer. This means that there is the need of fragmentation of IPv6 packets to fit larger packets in the IEEE 802.15.4 frame. Another issue is that since 40 bytes are needed by the IPv6 header there is not much space left for the payload. In the worst case the payload on the application layer is only 33 bytes when the overhead of each layer is taken into account.

IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) [94] was defined to handle these issues. 6LoWPAN is an adaption layer between the datalink layer and the networking layer. Beside support of fragmentation, it is able to compress the IPv6 header down to 7 bytes or to 2 bytes in special scenarios. This means that the IPv6 + UDP header can be reduced to 6 bytes, which leads to a maximum payload of 75 bytes in the best case.

ZigBee

ZigBee is a proprietary standard developed by the ZigBee Alliance [17]. Main focus of the ZigBee Alliance was to define a standard that allows to use cheap and energy efficient hardware. It builds up on top of the IEEE 802.15.4 standard and includes network layer, transport layer, and parts of the application layer. ZigBee adds routing, encryption, and application services to the communication stack. The networks typically consist of three different types of nodes: coordinators, routers, and end devices. Each network has exactly one coordinator. Routers transmit the information of the end devices to the coordinator. End devices can only communicate with routers or coordinators directly. ZigBee supports star, tree, and mesh topologies.

3.2.3 Application/Data Layer Protocols

The Constrained Application Protocol (CoAP) [110] is a synchronous protocol that is interoperable with HTTP. It was defined by the IETF in 2014. Compared to HTTP it uses much smaller packets and thus, is better suited for devices with low resources. Furthermore, it uses UDP instead of TCP. It applies a client/server architecture, where the client makes GET, PUT, POST and DELETE requests to the server.

Message Queue Telemetry Transport (MQTT) [86] follows a lightweight publish/subscribe messaging approach. Basically it has three main components: publishers, subscribers, and a broker. The sensors are publishers and send their data to the broker. The broker groups the sensor data in topics and sends the data to all applications that are subscribed to certain topics. MQTT uses TCP for reliable connections. MQTT-SN [66] extends the MQTT protocol with focus on sensor networks. MQTT-SN does not rely on TCP/IP and can be used with other transport layer protocols (e.g., UDP).

The Advanced Message Queuing Protocol (AMQP) [117] is an asynchronous messaging protocol that was proposed by the IEEE in 2006. It follows a publish/subscribe architecture similar to MQTT and uses TCP for reliable connections. Compared to MQTT it provides way more complex message scenarios and features. AMQP supports transactions and advanced security features. Furthermore, it uses a different broker architecture. The broker consists of two main parts: the exchange component and queues. The exchange component receives all published messages and distributes them to the queues according to certain pre defined rules.

Data Distribution Service (DDS) [55] is a standard that was developed with focus on time critical embedded systems. It offers very low latency, high performance, advanced

security, high scalability and reliability. DDS uses a publish/subscribe pattern with a peer-to-peer architecture. This means there is no central broker required. Moreover, it offers automatic discovery of publishers and subscribers.

Extensible Messaging and Presence Protocol (XMPP) [108] is a messaging protocol defined by the IETF that supports both publish/subscribe and request/response architectures. The basic message format is XML, which creates additional overhead compared to binary formats.

Which protocol to choose depends on the special use case. DDS, for example, fits very well for time critical, reliable applications, while CoAP offers high interoperability with the Internet and low resource usage.

3.3 Operating-System-Level Virtualization

Operating-system-level virtualization is widely used these days. We will give an overview about how virtualization is implemented in Linux, Windows, and FreeBSD. The main focus will be on Linux and Docker, since Docker is currently by far the most popular container engine [40].

3.3.1 Linux

In context of Linux, OS-level virtualization is realized through so called containers. Containers are using several Linux kernel features. The most important ones are: namespaces, control groups, and the chroot system call. Furthermore, often some security modules are used to harden container security.

Namespaces

The official Linux man pages describe namespaces the following way: "A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes." [6]. In other words, namespaces allow to run processes isolated and in their own namespace.

The first namespace was added for mount points in 2002 in the kernel version 2.4.19 [5]. Since then several new namespaces were implemented. Currently there are seven different namespaces supported: mount, PID, network, IPC, users, UTS, and cgroup.

- Mount (since kernel version 2.4.19 [88]): The mount namespace of a process is basically the mounted filesystem a process sees. Usually there is only one global mount namespace and a child process has a reference to the parents mount namespace. If a new namespace for a child process is created, the child process gets

a copy of the parents namespace instead, and can modify it without reflecting of the changes back to the parent process.

- PID (since kernel version 2.6.24 [28]): Traditionally, processes have a unique process id (PID) within the whole operating system. PID namespaces allow different processes to have the same PID as long as they have different PID namespaces [89]. This makes containers independent of the host system since there cannot be any PID conflicts after migration. Basically a process has two PIDs if a separate namespace is used. One unique global PID in the global namespace and another PID that is used inside its own PID namespace as described by Kerrisk [88].
- Network (since kernel version 2.6.24 [28]): Every network namespace has its own network stack with unique network devices, IP addresses, port numbers, etc. This allows for example to bind several web servers on the same port as long as they are in separate network namespaces. Internally this is achieved through routing of network packets by the host system [88].
- IPC (full support since kernel version 2.6.30 [54]): The Inter-Process Communication (IPC) namespace allows to isolate the childs System V IPC objects (semaphores, message queues, and shared memory) and POSIX message queues from the parent process [6].
- Users (since kernel version 3.8 [29]): This feature offers the ability to map user and group ids for each namespace differently. This means, for example, that a standard Linux user in the parent scope can be mapped to the root user in the namespace of the child process [88].
- UTS (since kernel version 2.6.19 [88]): UTS namespaces allow the definition of a different hostname and domainname.
- cgroup (since kernel version 4.6 [30]): cgroup namespaces allow to isolate cgroup hierarchies. A process that creates a new namespace sees its own cgroup as the root cgroup [1]. Thus, the ancestor cgroups are invisible to the process.

The usage of separate namespaces can be enabled at the creation of a child process with flags that are used as parameters for the *clone* system call. Listing 3.1 shows the prototype of the clone function in C and an example call where new namespaces for the network stack and PID are used.

Listing 3.1: The clone system call for C in Linux

```
1  /* prototype */
2  int clone(int (*fn)(void *), void *child_stack, int flags, void *arg)
3
4  /* example */
5  int pid=clone(childFunc, stackTop,
6               CLONE_NEWPID | CLONE_NEWNET, argv[1]);
```

Another way to enable a separate namespaces for a running process is through the system call *unshare*. The only difference compared to the clone call is that if it is used to create a new PID namespace, only the child processes are placed in the new namespace but not the caller itself. Listing 3.2 shows the prototype of the unshare function in C and demonstrates how a process can create a new namespace for mount points, and user and group IDs.

Listing 3.2: The unshare system call for C in Linux

```
1 /* prototype */
2 int unshare(int flags);
3
4 /* example */
5 int error=unshare(CLONE_NEWNS | CLONE_NEWUSER);
```

cgroups

The official Linux man pages describe cgroups as follows: "Control groups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored." [2]. This means cgroups allow to control and monitor resource usage in a fine-grained manner.

cgroups version 1 was initially released with the Linux kernel 2.6.24 [28]. The current version 2 was officially released with the kernel 4.5 [90]. Since a lot of virtualization engines still use the old version we will first explain how cgroups were implemented in version 1. Afterwards we will describe the changes in version 2.

Basically, a cgroup is a set of tasks that can be bound to several limits or parameters. A task can be a process or a thread. cgroups are ordered hierarchically. There can be several different hierarchies while each hierarchy is bound to one or more subsystems (resource controller). Figure 3.3 shows an example hierarchy that is bound to the CPU subsystem. The root cgroup `cpu_cg` is split up in two cgroups called `cg1` and `cg2`. The `cg1` cgroup is then divided again in `cg1_1` and `cg1_2`. In this example `cg1` gets a 60% share of the CPU, while `cg2` receives a share of 40%. All tasks in `cg1_1` get 75% of the `cg1` share, which results in a 45% share of the CPU overall.

According to the official Red Hat Customer Manual [100], there are several important rules for cgroup hierarchies. A hierarchy can have one or more resource controllers attached. When a new hierarchy is created every task of the system is automatically a member of the default cgroup in this hierarchy (the root cgroup). Furthermore, each task can only be member of exactly one cgroup inside a hierarchy. When a task creates a child task, the child task inherits the cgroup membership of its parent. After forking, the child task is handled separate from the parent and can become member of every cgroup independent of the parents membership.

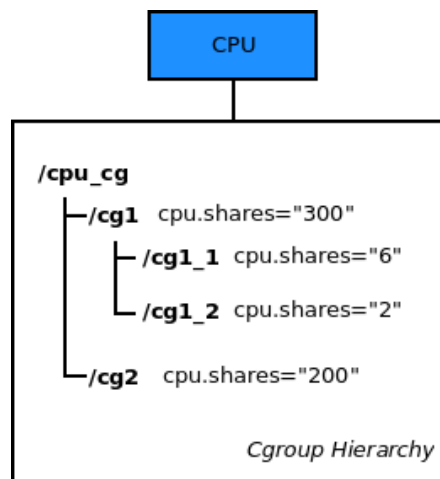


Figure 3.3: Example of a cgroup hierarchy attached to the CPU subsystem (adapted from [100])

There are 12 different controllers available in cgroup v1 [106]. We will focus on the most important ones:

- CPU: The CPU resource controller allows to define CPU shares for each cgroup. The resource restriction is a soft limit, which means that a cgroup can have a higher CPU usage as long as the CPU is not fully utilized.
- PIDs: This subsystem controls the number of tasks that can be created within a cgroup.
- memory: The memory controller permits limiting of memory available to a cgroup.
- freezer: The freezer subsystem allows to suspend and resume all tasks in a cgroup. If the parent of a process is frozen all of its children are frozen too.
- devices: Allows to define access control per device.
- blkio: The blkio subsystem allows to limit the IO rate of each device.
- perf_event: The perf_event subsystem offers the ability to monitor all processes with the *perf* tool in a hierarchy it is attached to.
- net_prio: This controller allows to set priorities per network interface for each cgroup.

Due to many inconsistencies among the cgroup controllers, the new cgroup version 2 has some major changes compared to the initial release [59]. There is only one unified hierarchy in cgroup v2. Furthermore, a process can only belong to one cgroup and

processes can only be attached to leaves and no internal nodes. Assigning of tasks was restricted to processes instead of both, processes and threads.

Currently there are three resource controller implemented in cgroup v2 [59]. These are the memory, io, and pids controllers. It is possible to mount different controllers under version 1 and version 2 hierarchies. But it is not allowed to mount the same controller on a version 1 and version 2 hierarchy at the same time.

chroot

Chroot is an abbreviation for change root. The official Linux man pages describe the chroot system call the following way: "chroot() changes the root directory of the calling process to that specified in path. This directory will be used for pathnames beginning with /. The root directory is inherited by all children of the calling process." [4]. Listing 3.3 shows the prototype of the chroot system call and an example usage where the root directory is changed from "/" to "/tmp/". To make the new chroot environment practically usable we would also have to hardlink or copy various binaries to the "/tmp/" folder since the process can only see what exists inside its chroot environment.

Listing 3.3: The chroot system call for C in Linux

```
1 /* prototype */
2 int chroot(const char *path);
3
4 /* example */
5 int error=chroot("/tmp/");
```

The chroot system call was first introduced in 1979 during the development of Unix V7 [80]. Later on it was added to BSD in 1982 [3]. Using a separate root environment for some processes can increase security slightly, but does not guarantee that it cannot be escaped since there are several ways to break out of the environment. This is because it was never intended to be a security feature and allows the root user to get out the environment by design [26]. Therefore, chroot should not be used for processes that are run by root.

AppArmor, SELinux, and seccomp

AppArmor, SELinux, and seccomp are Linux kernel security modules that add another layer of security to the system.

AppArmor and SELinux enable mandatory access control (MAC) on Linux systems. AppArmor stands for Application Armor and allows to specify rules that define which resources can be accessed by an application [11]. These rules are called *profiles*. AppArmor can be used in a *default allow* or *default deny* behaviour. The former one means that if no rule is defined for a resource, it can be generally accessed, the latter one follows a whitelist approach, which is way more secure.

SELinux is an abbreviation for Security Enhanced Linux and was initially developed by the NSA. Like AppArmor, it allows to set rules that define which objects can access resources. These rules are bundled in so called policies. SELinux has 3 different modes [67]: enforcing, permissive, and disabled. In enforcing mode access is denied strictly if the policy does not allow it. When permissive mode is used, access is not denied, but actions that would be denied in enforcing mode are logged. SELinux is considered to be more secure than AppArmor, but this comes with the cost of higher complexity compared to AppArmor.

Seccomp is short for Secure Computing. As described by Edge [43], it allows to define which system calls can be invoked by a process. If a process makes a system call that is not allowed, the default action is to kill that process. Many applications only need a subset of the system calls available by the Linux kernel, thus restricting the set of available system calls can lead to a significant improvement of security, in cases where intruders gain control over an application. On the other hand, it is often hard to predict which system calls are used by an application, thus filtering of them can be cumbersome.

3.3.2 FreeBSD

The FreeBSD operating system is a successor of the Berkley Software Distribution (BSD). Among other BSD based operating systems like OpenBSD and NetBSD, it is currently the most popular one. FreeBSD was the first OS that offered OS-level virtualization support out of the box through so called *jails* [80].

Jails

Jails were implemented by Poul-Henning Kamp in 1999. Watson and Kamp describe the concepts behind jails in detail in [87]. Jails are often called *chroot on steroids*, because they build up on the *chroot* system call. While *chroot* was not planned as a security feature, jails were designed with focus on security. Thus, they have more sophisticated features in terms of security and feasibility compared to *chroot*. *Chroot* only isolates the view on the filesystem, a jail also isolates processes from each other. Therefore, jailed processes can only see the processes in the same jail. An unjailed process, on the other hand, has a view on all processes on the machine. Basically, this allows to watch processes inside a jail while they cannot see the watcher. Furthermore, jails have a separate set of users and a separate root account. This means that, a jail root account has only admin rights inside the jail and cannot do changes to the system outside its jail. Therefore, it is by design no root escape possible. Each jail also has a separate hostname and IP address. This isolation features are comparable to the namespaces in Linux as described earlier. Control of resource usage of jails can be done through the program *rctl* [8]. It allows to limit memory usage, cpu usage, etc.

Listing 3.4: The jail system call in the shell in FreeBSD

```
1 //synopsis
2 jail [-i] [-l -u username | -U username] path hostname ip-number
   command ...
3
4 //example usage
5 jail /tmp/jails/exampleJail exampleHost.org 10.0.0.12 /bin/sh
```

Listing 3.4 shows the usage of the jail system call and gives an example invocation. The example uses `/tmp/jails/exampleJail` as the root directory, `exampleHost.org` as the hostname and `10.0.0.12` as the ip address for the jail. `/bin/sh` is the command that is run after the jail is created. The jail call can only work if the folder where the jail is placed contains required resources (e.g., binaries, shared files, etc.). All jails are further configured in the `jail.conf` configuration file.

3.3.3 Microsoft Windows

In the last decade open source operating-system-level virtualization was mainly available and used on Unix-like operating systems like FreeBSD and Linux. But recently Microsoft opened up and joined the market of open source lightweight virtualization solutions. The virtualized partitions follow the established naming conventions and are also called containers on Windows.

Containers

Since Microsoft Windows Server is closed source it is naturally hard to find information about how the kernel supports virtualization. According to a talk by John Starks at `dockercon 16` [75], the kernel offers features comparable to cgroups and namespaces in Linux. These features are not accessible through a public interface. This means that it is currently not possible for other vendors to offer operating-system-level virtualization solutions for Windows, which make use of these features. Microsoft focused on supporting Docker and thus, ported the Docker Engine so that it can be used on the Windows platform. All changes they made were contributed to the open source project. Since the containers share the Windows kernel it is not possible to run Linux containers on Windows. Currently there are two types of base images supported: `NanoServer` and `WindowsServerCore`. Both are based on Windows Server 2016, but while the first one is a minimal system with focus on performance, the latter one is optimized for compatibility and needs about 10 times the disk space of the `NanoServer` image. Since the normal Docker Engine can be run on Windows, also all additional management tools offered by Docker can be used. Another feature that is offered by Docker on Windows are Hyper-V containers. Hyper-V containers are virtual machines where each VM runs one Docker container. This means that every container has its own environment, and therefore, its own kernel. This approach offers a better isolation compared to the normal

operating-system-level virtualization approach, but comes with a performance overhead since it is a type of hardware virtualization. Hyper-V runs a Windows Server 2016 VM, which has full support for Docker out of the box. Although Windows also offers an Ubuntu subsystem that allows to run Linux applications on Windows without any adaptations, this subsystem does not allow to run Linux containers. It still uses the Windows kernel, and thus, does not support namespaces and cgroups as needed by Linux containers.

3.3.4 Solaris

Operating-system-level virtualization on Solaris was introduced in Solaris 10 in 2004 [102]. Solaris containers consist of two parts: the Solaris Zone and the resource manager. Zones offer a mechanism for isolation similar to Linux namespaces. There are two type of Zones in a container: the global Zone and non-global Zones [37]. The global Zone is the traditional operating system environment and has control over the non-global Zones. Non-global Zones are the container specific environments. Usually non-global Zones are simply referred to as Zones if not stated otherwise. Non-global Zones are isolated from each other in terms of processes, file system, and network interface [35]. They also have their own hostname and IP address. The resource manager allows to control and limit the usage of CPU, memory, number of processes, etc. Non-global and global Zones can be configured as so called immutable Zones, which allows to define read-only access to folders of the container [36]. This adds another layer of security.

3.3.5 Open Container Initiative

In the last years container virtualization became increasingly popular. While Docker is the de facto standard, also many other vendors came up with different container runtimes and container image formats [40]. To improve user experience and ensure that there is no vendor lock-in, standards had to be defined. Therefore, CoreOS announced an open specification called App Container (appc) in December 2014 [69, 98]. Half a year later, the Linux Foundation launched the Open Container Initiative (OCI) [7]. The OCI is a consortium of major IT leaders and cloud providers like Google, Docker, VMware, Microsoft, Amazon, Facebook, CoreOS, Resin.io, etc. The mission of the OCI is defined as follows: "The Open Container Initiative provides an open source technical community within which industry participants may easily contribute to building a vendor-neutral, portable and open specification and runtime that deliver on the promise of containers as a source of application portability backed by a certification program." [83]. Therefore, they defined specifications for the container runtime and the container image format. Furthermore, the goal of OCI is to provide a container runtime according to these specs. The cornerstone of these standards are the Docker runtime, runC, and the Docker image format, which were donated by the company. Also the appc specification eventually merged into the OCI and thus, it is becoming the universal standard for OS-level virtualization.

3.3.6 Important Concepts

This subsection describes some important concepts that are used by various container engines. Furthermore, we will clarify some common misunderstandings when it comes to operating-system-level virtualization.

Copy-on-write

Copy-on-write is a strategy that is used by many container engines to efficiently manage the disk usage of images and containers. The idea behind that strategy is that if a duplicate of some data is needed, the resource is not duplicated but shared in a read only manner. The data is only copied if changes are made to the shared resources. This method allows to start up new containers very quickly since already existing images can be used and no data has to be copied. Furthermore, it reduces the disk utilization of unmodified copies significantly. Unmodified copies are very typical for containers since they often have a similar root file system. The drawback of this concept is that it adds a performance overhead when resources are modified heavily.

System Container vs. App Container

There are two different purposes for which container engines are designed: app containers and system containers. App containers are mainly designed to run only a single application per container with the minimal system needed. System containers are more like virtual machines and thus, designed to be used as a full operating systems with various applications running.

Container Engine vs. Container Runtime

The terms *container engine* and *container runtime* are often used interchangeably, although their meaning differs. Sometimes it is hard to distinguish between these two terms, but basically a runtime refers to the low level software that, among other things, takes care of starting and stopping a container. A container engine uses a runtime, but adds additional features like networking, creating the container filesystem, etc.

3.3.7 Container Engines

In the last years many different container engines came up. Since the foundation of the OCI, significant effort in terms of standardization has been taken, that led to universal standards. These standards are already supported by some engines. We will describe the features of currently available container engines in detail. The focus is on open source container engines available for Linux, since they are most widely used.

Docker Engine

Docker is currently by far the most popular container engine. According to a study by ClusterHQ, 94% of people who use operating-system-level virtualization are using

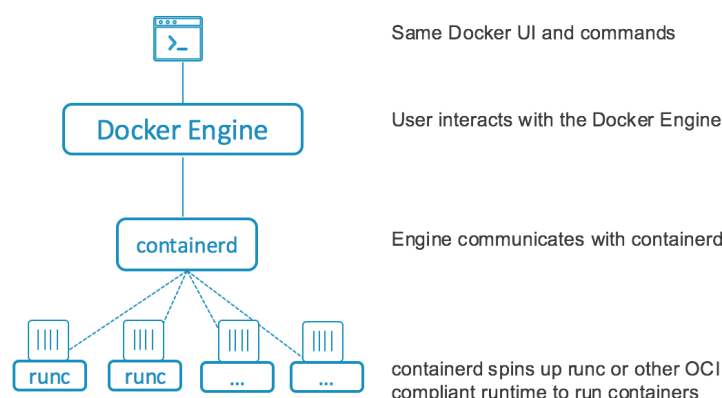


Figure 3.4: The Docker Engine architecture [101]

Docker [40].

The Docker Engine is available for Linux, Windows and MacOS. While it runs natively on Windows and Linux, MacOS needs to virtualize specific Linux kernel features that are used [77]. Support for FreeBSD is still in experimental state [52]. It is licensed under the Apache 2.0 license and the source code is available on GitHub [76]. Docker containers are designed to be used as app containers.

The Docker Engine uses a client-server architecture, where the server is called *docker daemon* and exposes its functionality via a REST API to the client, which is called *docker cli*. Figure 3.4 shows an overview of the architecture. The docker daemon uses the containerd runtime which "can manage the complete container lifecycle of its host system: image transfer and storage, container execution and supervision, low-level storage and network attachments, etc." [74]. Containerd is part of the runC project and uses runC as the container runtime. RunC is the reference implementation of the Open Container Initiative's runtime specification. It has full support of Linux namespaces and cgroups.

Docker uses a layered approach to build images and containers, as described by the official userguide [78]. Figure 3.5 shows how this is done in detail. Each layer only stores the differences to the layer underneath. Every image layer gets an ID, which is computed through cryptographic hashing. Furthermore, the layers of the image are read only. If containers are run, a new read/write layer for each container is added on top of the image layers. This approach has some important benefits. Since the base image layer stays the same, it can easily be reused for other images. Thus, the required disk space is highly optimized. Beside that, each image can be used for several containers, because they are read only. This improves the start time of a container significantly since only a thin read/write layer for each container has to be created on start up. When a container is removed, all changes that are made to the read/write layer are lost. If permanent writes are needed, a separate space has to be mounted inside the container. Docker also uses a copy-on-write strategy for images and containers, which is very similar to the layered

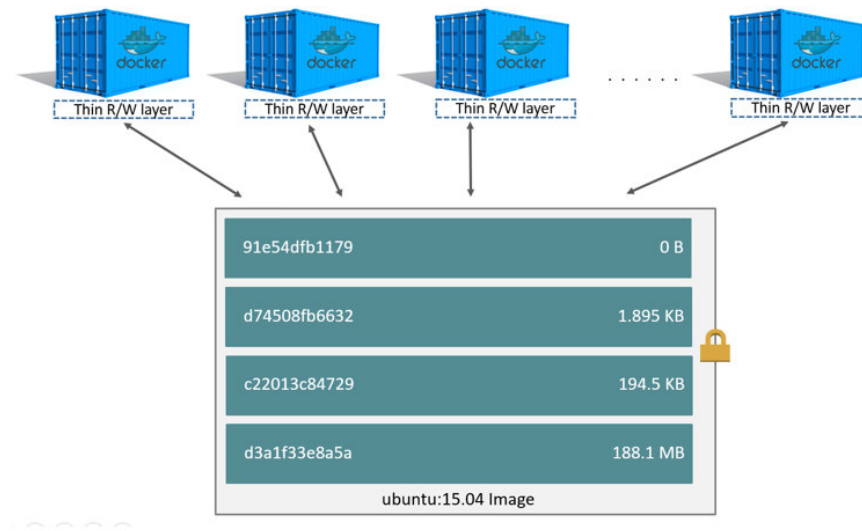


Figure 3.5: A Docker image with the container layers on top [78]

approach.

New Docker images can be created by committing all changes that are made inside a container. To make this process less error prone, this can be done automatically with a *Dockerfile*. This file allows to define how an image is built and configured. An example Dockerfile that installs and runs an apache2 server on Ubuntu 16.04 is shown in Listing 3.5. The first line defines the base image that is used, which is Ubuntu 16.04 in this case. Line 3 defines some metadata about the image. In line 5 the apache2 server is installed. The following commands set environment variables that are used for the apache user, group, and log file directory. In the last line the command that is executed on container startup is specified.

Listing 3.5: An example Dockerfile

```

1 FROM ubuntu:16.04
2
3 MAINTAINER Peter Eder version: 0.1
4
5 RUN apt-get update && apt-get install -y apache2
6
7 ENV APACHE_RUN_USER www-data
8 ENV APACHE_RUN_GROUP www-data
9 ENV APACHE_LOG_DIR /var/log/apache2
10
11 CMD ["/usr/sbin/apache2", "-D", "FOREGROUND"]

```

Docker creates three different networks during installation, according to the official user

guide [73]. Namely these are: bridge, none, and host. The bridge is part of the host network stack and is called `docker0` on the host. If no network is specified on container startup the container attaches to the bridge network by default. When the none option is selected, the container gets a network stack that is detached from the host and therefore from the outside world. Thus, only the loopback interface is enabled. If the host option is chosen, the container attaches directly to the host network without any different configuration. This means, that the same IP address, hostname, etc. are used.

Docker makes use of iptables on Linux to manage network traffic. Docker sets up a rule to masquerade the container traffic by default. While containers can communicate with the outside world, hosts from outside cannot connect to the containers by default. If container ports should be exposed, they have to be mapped to ports on the host. This is done on container startup. Internally the port mapping is again done through iptable rules.

Docker offers a basic seccomp policy that can be used when a container is started. Furthermore, containers are started with a predefined AppArmor profile by default.

Docker comes with native support for cluster management via Docker Swarm. It was merged into the Docker Engine with version 1.12 and allows to administrate Docker hosts on a large scale. The Docker Engine can also be accessed through the Remote API. This allows to easily control Docker hosts over the network.

Usually the term Docker refers to the Docker Engine, but beside the engine itself, several other useful tools for container management are available:

- **Docker Hub:** The Docker Hub is a registry service for Docker images hosted by Docker Inc. It is used by default when new images are pulled. There are official images available like Ubuntu, MySQL, etc. Searching and pulling of images can be done anonymously, but pushing of images requires registration.
- **Docker Registry:** The Docker Registry can be seen as a private Docker Hub. This means that it can be installed locally and used as a private registry. A private Docker registry enables full control over image storage and distribution.
- **Docker Compose:** Docker Compose is a tool that allows to define and run applications that consist of several containers. It introduces a file called `docker-compose.yml` to define and configure all services (containers) that are used by the application.
- **Docker Machine:** This tool can be used to provision Docker hosts. It supports provisioning of virtual machines on physical machines, on cloud platforms, etc. through various drivers.

rkt

rkt [71] is an open-source container engine that is mainly developed by CoreOS Inc. It is available for the Linux OS and licensed under the Apache 2.0 license. rkt supports the

appC standard, which was later merged into the OCI standards as already mentioned. Native OCI runtime spec support is currently under development. It is also possible to run Docker images with rkt. The main focus of rkt is on app containers.

rkt does not launch a long running daemon as Docker does with containerd, but uses the init system like systemd to manage the lifecycle of containers. Furthermore, it uses the concept of pods as the unit of execution [69]. A pod consists of one or more containers and metadata that is applied to the pod. Metadata could be resource constraints for example. All containers in a pod share the same context, which also includes networking. This means all containers inside a pod can reach each other via localhost (127.0.0.1). The image format used in rkt is called *Application Container Image* (ACI), which consists of a tarball that is containing the rootfs and an image manifest with metadata.

Container execution is split up into three different stages, according to the official rkt documentation [72]. Stage 0 is invoking of the rkt binary. This includes some initial tasks as generating a pod UUID, generating a pod manifest, creating the filesystem of the pod, setting up the directories for the following stages, etc. In stage 1 the necessary steps for launching a pod are taken. These are container isolation, creating the network, mounting of the filesystem, etc. There are several different isolation environments, called "flavors", available in rkt:

- systemd/nspawn: an isolation environment that is using cgroups and namespaces (this is the default behaviour)
- fly: a chroot only environment
- kvm: uses hardware virtualization through the kvm hypervisor

Stage 2 uses the environment as set up by stage 1 and executes the container. Figure 3.6 gives an overview of these different stages.

Images are created with a command line tool called acbuild, which has a quite similar workflow as Dockerfiles. There are bash script templates available, which can be used to automate building an image.

rkt has two different categories of network options as mentioned in the official documentation [70]: host mode, and contained mode. In host mode, the pod shares the network stack with the host machine. Thus, IP addresses, hostname, routes, iptable rules, etc. will be used as configured on the host. Contained mode uses a separate network namespace for the pod. There are three different built in networks for this purpose: default, default-restricted, and none. Similar to docker, none mode only sets up a loopback interface and isolates the container from the host. The default network consists of a virtual ethernet (veth) and loopback interface. The veth can be used as bridge, macvlan, ptp, etc. The default-restricted network is similar to the default, but with the restriction that no default route and IP masquerading is setup. Therefore, the pod can only communicate with the host. rkt uses seccomp isolators by default when the

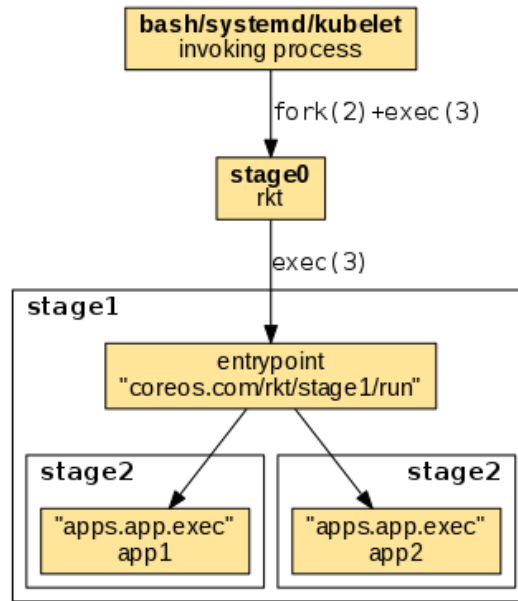


Figure 3.6: The different stages of rkt [72]

containers are started. There are several predefined filters available, which can easily be adapted.

LXD/LXC

LXD is an open source container engine that builds up on Linux containers (LXC) [93]. Basically LXD is a daemon that can be seen as a hypervisor for LXC containers. It is designed to run full system containers. Furthermore, it offers a REST interface that allows to control containers remotely. LXD is published under the Apache 2 license and the driving force behind LXD is Canonical Ltd. LXC was released in 2008 and is published under the GNU LGPL v.2.1 license. Initially Docker used LXC as container runtime, but later they switched to runC. LXD and LXC do not support any standards as defined by the OCI.

The engine itself, is based on cgroups and namespaces. It can run two different types of containers: privileged and unprivileged. Privileged containers are using the global root also inside the container. This is a potential security issue, since a root escape from a container leads to global root rights for intruders. Unprivileged containers, on the other hand, are using root users inside containers that are mapped to normal global users. They are the default when a container is launched.

There are two image types in LXD: unified images and split images. Unified images are a single tarball that consists of a rootfs directory with the root file system, metadata file and a templates directory. Split images consists of two tarballs, one with the root

file system and one with the metadata and templates directory. Split images are mostly used to if there is already an existing compressed root file system available.

LXD adds a new virtual ethernet interface during installation that is called `lxdbr0`. `lxdbr0` serves as a bridge that is connected to each container by default. LXD also allows to define new network interfaces or usage of existing physical interfaces directly. All containers in LXD are running with a `seccomp` policy and an `AppArmor` profile by default.

Others

OpenVZ is another container engine that was mainly designed to be used for system containers. It was initially released in 2005 and is available under GNU GPL. The current version, OpenVZ 7, offers HW virtualization support, live migration, thin disk provisioning, etc. [12]. The proprietary container engine Virtuozzo 7 builds up on OpenVZ 7 and adds additional features and commercial support.

"Let Me Contain That For You", or `lmctfy`, is an operating-system-level virtualization engine developed by Google Inc. It was used as Google's container stack and was later published as open-source. The core concepts were merged into the Docker project and further development was stopped in 2014.

3.3.8 Container Security

There are some security concerns, which have to be addressed when it comes to operating-system-level virtualization as explained by Mouat [95]. Since many containers run on one host, it would be fatal if intruders gain control over the host system, since they would gain control over several applications. Another problem is that all virtual environments share one kernel. This means that a kernel panic caused by malicious software takes down all containers. Compared to hardware virtualization this is a critical security risk. Moreover, a shared kernel also means that a kernel bug affects all containers. Since resources are shared between all containers, it is possible that a container can starve out other containers. This makes containers exceedingly vulnerable to Denial-of-service (DoS) attacks.

In the last years significant effort was taken to enhance security and counteract these vulnerabilities. A major improvement was established by integrating of user namespaces into container engines. As explained earlier, user namespaces allow to map unprivileged users to root users inside the container [88]. This means that normal local users are used as root inside the container. Thus, if root escape happens, intruders only have normal user rights on the host. This avoids many security risks by design.

Another way of hardening security of containers, was the integration of SELinux, Apparmor, and `seccomp`. As explained above, Docker, `rkt`, and LXC use default profiles for some of these features. This adds another layer of security.

Cgroups are an effective way to reduce the potential impact of DoS attacks. Many container engines have them integrated, and hence, allow to define limits for resource usage of containers.

Overall, container security was raised significantly over the last years. However, there are still some risks (e.g., the shared kernel) that have to be considered when operating-system-virtualization is used.

3.4 Container Deployment Frameworks

In this section we will describe some commonly used build and deployment frameworks. Furthermore, we discuss their feasibility for container deployment and how they can be applied in the domain of IoT.

3.4.1 Jenkins

Jenkins [103] is an open-source build automation server. It was initially developed by Sun Microsystems as Hudson. Due to some disagreements between the community and Oracle after they have bought Sun, Hudson was forked and renamed to Jenkins. It is released under the MIT license and the development is managed by *the Jenkins project*.

Jenkins is considered to be the market leader when it comes to build automation and continuous integration for the Java programming language [125]. This is mainly due to fact, that Jenkins is highly extensible through plugins and thus, it can easily be integrated into a continuous delivery pipeline.

Jenkins calls the workflow from source code to deployment a pipeline. A pipeline typically consists of several stages: source code checkout, build, test, and deployment. Thus, although Jenkins is mainly used for build automation and continuous integration, it can also be used for deployment to production. There are several plugins available that support automatic building of Docker images and deployment of containers. Another way to deploy docker containers is to use plain shell commands, which can be triggered after each stage. Since it is possible to execute any type of shell script, there are many different ways to use Jenkins for deployment. However, deployment through shell scripts tends to be cumbersome and is difficult to maintain. Another problem that comes up when Jenkins is used for container deployment is that there is no support for further management of containers (e.g., lifecycle management or log access).

3.4.2 Kubernetes and OpenShift

The Cloud Native Computing Foundation describes Kubernetes as follows: "Kubernetes is an open source system for managing containerized applications across multiple hosts, providing basic mechanisms for deployment, maintenance, and scaling of applications." [46]. Kubernetes is the successor of Googles internal cluster manager called Borg [116]. In 2015, Google donated the first stable version v1.0 to the Cloud Native Computing Foundation,

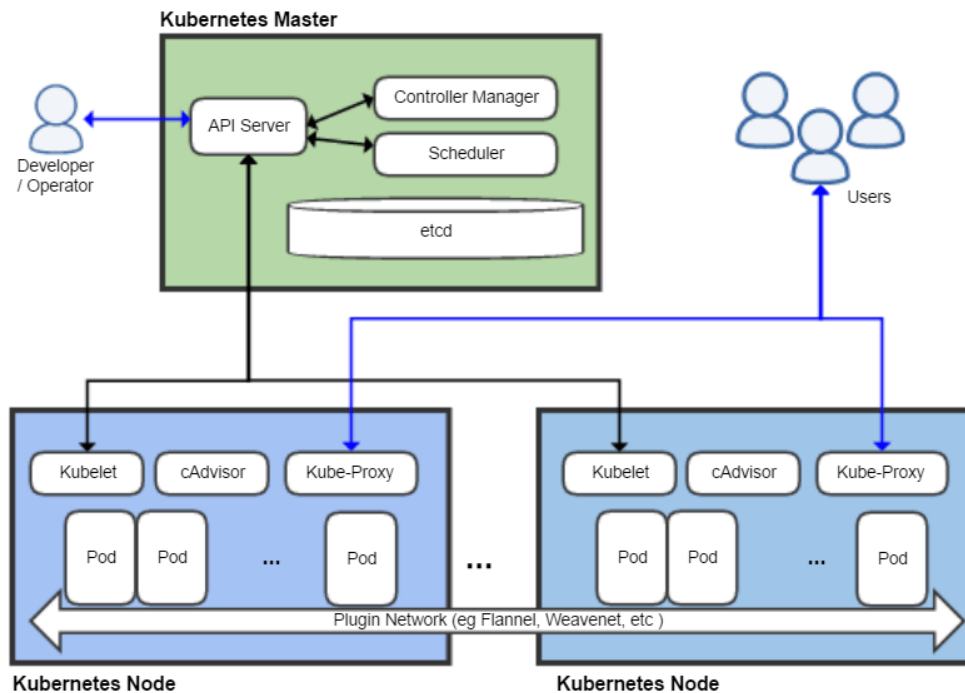


Figure 3.7: Architecture of Kubernetes [91]

which is hosting the project now. The project is released under the Apache License 2.0. Currently, Kubernetes is among the top 0.01% projects on GitHub and number 1 when it comes to activity [18].

The basic architecture of the Kubernetes system is shown in Figure 3.7. In general, a Kubernetes system consists of one or more Kubernetes masters and several nodes.

The master is the central managing component and has four different subsystems: the API server, the controller manager, the scheduler, and etcd. The API server exposes the REST API and command line tool to the user and thus, provides the frontend of the cluster. The controller manager runs several different controllers that are responsible for replication, health checking of nodes, endpoints, etc. The scheduler selects the nodes to which newly created pods are assigned. A pod in Kubernetes is conceptually the same as a pod in rkt [69]. This means that a pod is a group of one or more containers that share the same context. The pod is the smallest unit that can be scheduled and deployed in Kubernetes. Etcd is a distributed key value store and the primary datastore in Kubernetes. It uses the Raft consensus algorithm and can handle hardware failures and network partitions [34].

A node is a physical or virtual machine on which containers/pods are executed. The

most important components are the kubelet, the kube-proxy, and the container engine. The kubelet is the main agent on the machine. Among other tasks, it watches all pods, mounts volumes, runs containers, periodically checks liveness of containers, etc. The kube-proxy is responsible for networking that enables so called service endpoints. Since pods are typically deployed and destroyed frequently there is another layer needed to make them accessible through a static interface. Therefore, Kubernetes uses the concept of services. A service has a static IP and a hostname, and can be seen as the loadbalancer for the pods that are attached to it. Basically, this is done through iptables rules. The service endpoints are found through DNS.

By default Kubernetes uses containerd and therefore, runC as container runtime as described by Chanezon [31]. But it can also be used with rkt, although this is still in experimental state. Furthermore, Kubernetes offers a runtime interface that makes it easier for other vendors to integrate their container runtime [60]. However, the interface is still in alpha state.

Beside these components, there are also processes running for supervising, cluster-level logging, etc., on a Kubernetes node.

Kubernetes offers several powerful features for container/pod management:

- Auto-scaling: Kubernetes allows to automatically scale pods horizontally according to CPU utilization or some application specific metric.
- Self-healing: If a pod fails, a new instance is automatically started according to the configuration.
- Rolling updates: When updates need to be deployed without downtime, so called rolling updates can be used in Kubernetes. Basically, a rolling update means that a new node is deployed and an old one is taken down afterwards. This is done iteratively until all old pods are replaced. If an error occurs during the update, the roll out is stopped and can easily be reverted.
- Resource monitoring: Kubernetes supports very detailed resource monitoring on a container, pod, or even cluster base.
- Volume management: Kubernetes offers pod-wide volumes that can be mounted by each container inside a pod and are attached to the pods lifecycle. This allows to keep data independently of containers state.

Kubernetes can be used through a command line client, a REST API, and a Web UI. While it offers many tools to manage the deployment and lifecycle of containers, it does not support building of applications and packing them into containers. However, since all features are accessible through a REST API it can easily be integrated into a continuous deployment pipeline.

OpenShift [81] is an open source container application platform developed by Red Hat. It builds on top of Kubernetes and extends it with many features. OpenShift integrates the whole development pipeline into Kubernetes, this means it is directly connected to the source code management (SCM) and allows to build images and to deploy containers directly from source code. Thus, it has an continuous deployment pipeline integrated. Furthermore, it offers several templates for services like MySQL, PHP, NodeJS, etc. out of the box.

There are several different versions of OpenShift available. The open-source project is called OpenShift Origin [104]. Furthermore, an enterprise version with commercial support and some added functionality is provided.

Both, Kubernetes and OpenShift are very powerful tools for container management in clusters. However, they are both only limited applicable for IoT applications since there is a significant overhead of resources for managing containers. Furthermore, IoT applications differ significantly from applications that are operated in a cluster. For example, networking capabilities between IoT nodes are restricted. Moreover, many features are not useful for the IoT domain. For example, if an IoT gateway fails, starting a new one somewhere else is usually not possible or not feasible, because many IoT devices will not be able to reach the new gateway anymore.

3.4.3 Resin.io

Resin.io [9] is an application deployment service with focus on IoT. It uses Docker to distribute and deploy software to IoT devices. Resin.io is provided as Software as a Service (SaaS).

Deploying a new container on an IoT device is very simple as described by resin.io [105]. If devices are already setup correctly, the whole deployment process comes down to a git push into the resin.io repository. This triggers an internal workflow that builds the application, packs it into a container, transfers the image to the configured device and deploys the container. The whole deployment process from source to deployment is illustrated in Figure 3.8.

All devices are connected through a virtual private network (VPN) to the resin.io server. Transferring the Docker images is achieved through a Docker registry: The image is deployed to the registry and then pulled by the device.

Figure 3.9 shows the architecture of a resin.io device. Each device runs a container with the resin.io agent. This agent manages the user applications on the device. It connects to the resin.io server, pulls the newest application, and monitors resources as well as the application itself. Resin.io uses Yocto Linux [10] as host operating system on the devices. Yocto Linux is a customizable distribution with focus on embedded products.

Provisioning of IoT devices is done through a standardized image that needs to be installed on the device. On startup the device automatically connects to the resin.io server and registers on the users dashboard. After that, the resin.io agent is started and

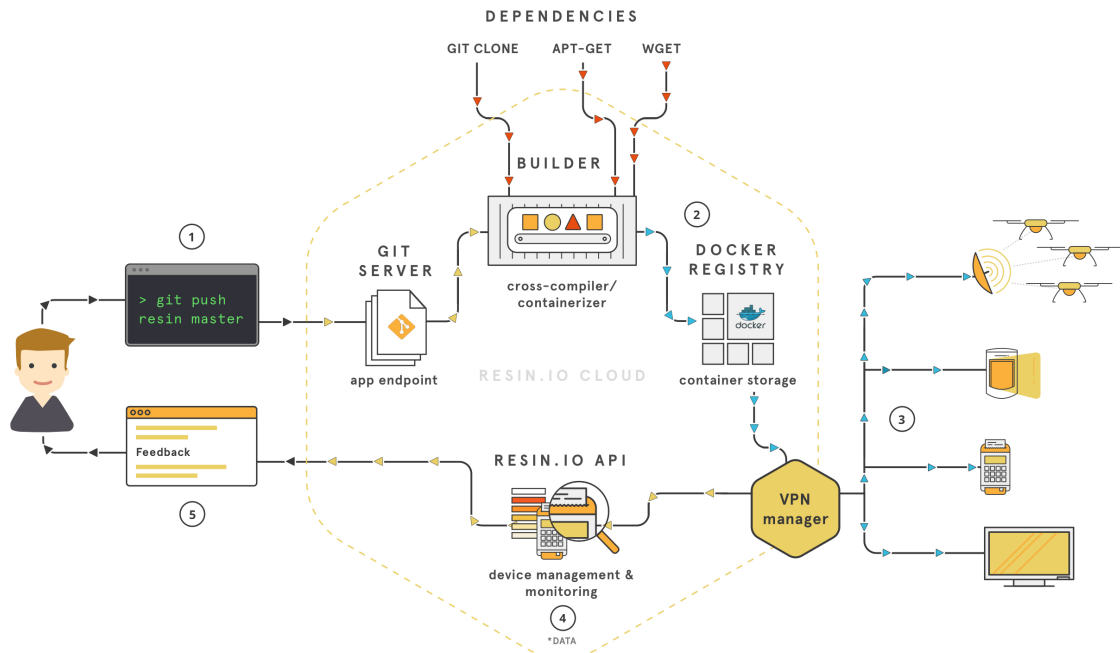


Figure 3.8: Application deployment with resin.io [105]

registers on the VPN to receive a unique API key. Afterwards, the device shows up as online on the dashboard and is ready for deployment.

Resin.io uses a Linux OS that has the minimum features to be able to run Docker. This allows to run it on many devices with constrained resources. Furthermore, the application container runs in privileged mode, which enables full control over the hardware.

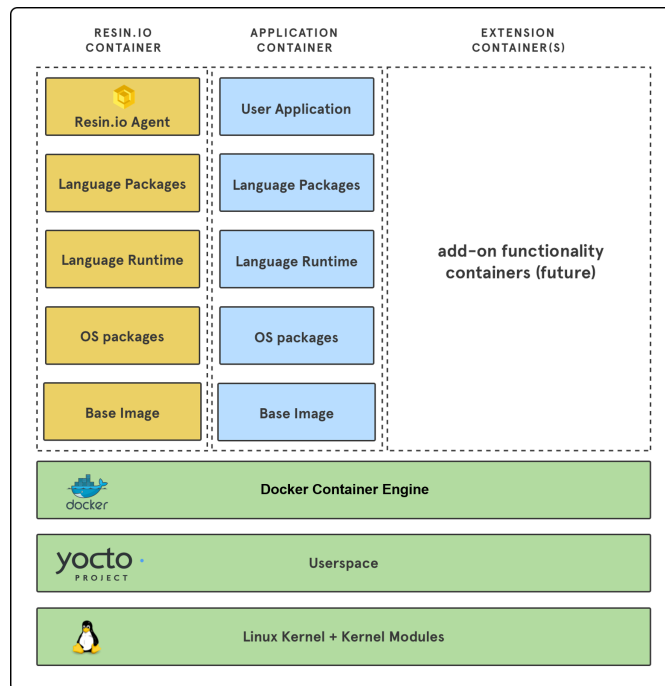


Figure 3.9: IoT device architecture in resin.io [105]

Design & Implementation

This chapter explains the design and implementation of the prototype. First, we will describe the main features of the framework. Afterwards, the design decisions are discussed, and the architecture is presented. The last part explains the implementation in detail.

4.1 Features

The main goal of the application deployment framework we are implementing is to provide operating-system-level virtualization in the IoT domain. Thus, the framework needs to support creation and distribution of images, and full remote control over the applications. The particular features that need to be implemented are explained in this section.

Containerization

Since we want to provide a showcase for OS-level virtualization the framework needs to be able to pack the application into an image. We call this procedure containerization, because this term is often used in this context, although in fact we create an image. The whole containerization process should be completely transparent to the user.

Image Distribution

Created images need to be distributed. Therefore, the framework must be able to transfer the image to the corresponding device, depending on the selection of the user.

Application Management

The user should be able to control the application on the device remotely. Hence, a user interface is needed that allows to run, stop, pause, and remove an application. This should be realized through a web UI.

Maintenance

In order to be able to maintain a distributed application, it is necessary to detect and debug failures. Therefore, the framework should support access to the logging output. Furthermore, it should be able to get an overview of the utilization of the resources of each container on a device. Both should be presented through a web UI.

One Click Deployment

Deployment of an application should be as easy as possible. Therefore, a deployment feature should be offered, that triggers containerization and distribution of an application within one click.

Device Management

In order to manage the deployment of the applications an overview of all devices is needed. This overview should show all devices that are registered. Furthermore, it should give information about the current state of the device (online or offline).

4.2 Design

This section explains the main design decisions that were made during development of the framework. Furthermore, the architecture is described in detail. The last part illustrates the deployment process of an application from upload by the user to deployment on the corresponding device.

4.2.1 Design Decisions

During the implementation of the prototype three important design decisions were made. These were: the container engine, the architecture of the framework, and the if there should be an agent used on the devices.

Docker Engine

We decided to use the Docker Engine as the underlying container engine. There are several reasons why Docker fits well for our purpose:

- **Docker Engine API:** Docker offers a powerful remote API. This allows to control the engine remotely on each device without relying on any other service.
- **Docker Registry:** As explained earlier, Docker comes with a private registry. The registry simplifies the distribution of images, since it is well integrated within the Docker platform. Thus, we do not need to implement our own mechanism to transfer images to the devices.

- The community: As mentioned before, Docker has by far the highest market share of all container engines. Hence, there is a huge community behind the project. As a result, many libraries are available. Furthermore, many recurring problems are already discussed and solved in public forums. Another advantage is that numerous official images are offered on the public registry (e.g., debian, java, nginx, mysql).
- OCI compliant: Currently, Docker is the only container engine that is fully compliant to the OCI standard. Both, the container runtime and the image format fulfill the OCI specification. This minimizes the risk of a vendor compared to other engines that are not fulfilling any standards.

Microservice Architecture

We decided to implement the framework with a microservice architecture. The term *microservice* coined up a couple of years ago and is currently one of most popular topics in software development. Sam Newman defines microservices as follows: "Microservices are small, autonomous services that work together." [96]. Microservices are small services that focus on doing one thing and doing it well. This is similar to a key concept of the UNIX philosophy. Microservices are autonomous in terms that they can be deployed independently. Communication between services is done through network interfaces.

As outlined by Sam Newman [96] and Martin Fowler [51] this architectural style has many benefits compared to a monolithic architecture:

- Partial deployment: Since all microservices are deployed independently, deployment of new versions does not imply deployment of the whole system. This limits the risk to the updated service, instead of the whole application. In further consequence, new features can be delivered faster and with small incremental changes.
- Availability: Compared to a single monolithic application where a failure often crashes the whole application, a failure in a microservice based application is often limited to a single service. Hence, all other service can still operate and the overall system is still available.
- Well defined interfaces: A microservice style enforces developers to use well defined interfaces between the services, because it is the only way how the services can communicate between each other.
- Loose coupling: Microservices are loosely coupled by design, because each services operates independent of the other services.
- Scalability: A large monolithic service can only scale as a whole. Microservices allow to scale only the real bottlenecks in a more fine grained manner.
- Technology heterogeneity: Each service can use a different underlying technology, as long as the communication pipe between the services stays the same (e.g., HTTP

requests). This allows to use the technology that fits best for a certain task (e.g., relational database vs NoSQL database, functional language vs object oriented language)

However, there is no such thing as a free lunch and thus, these benefits come with some trade-offs:

- **Complexity:** A microservice architecture adds significant complexity to a system. For example, orchestration of all services needs to be handled. Since there can be hundreds of services, this is a very crucial task. Moreover, it comes with all the complexities and pitfalls of a distributed system.
- **Inter service refactoring:** Refactoring or adding new features that cover several services can get cumbersome. There is the need for deployment strategies for rolling out such features. For example, if the API needs to change, it is possible to deploy two API versions at the same time. The old version can be marked as deprecated, but is still available for other services that are not updated yet. Although, this can be handled, compared to a monolithic application, it requires way more effort.
- **Communication overhead:** While normal method calls in a monolithic application are cheap and reliable, this is not true for microservices. Using the network interface to communicate between services adds an overhead in terms of network latency and computing resources. Furthermore, it can not be guaranteed that the messages are delivered correctly.

Since we want to provide the deployment framework as a web service, a microservice architecture makes sense, despite the drawbacks listed above. The major advantage for our purpose is the possibility of fine grained scaling. For example, containerization of applications is a potential bottleneck and implemented as a microservice it can easily be scaled out.

Agentless

Since Docker offers the Docker Engine API we decided to implement the framework without any agent on the devices. The main advantage of this approach is that we save resources and reduce complexity. Since IoT devices typically have limited computing power and storage capabilities, the available resources should not be wasted. However, this is a trade-off in terms of flexibility and capability since we are limited to the interface Docker provides.

4.2.2 Architecture

To give an overview of the architecture of the deployment framework we follow an approach described by Brown [27], where different layers of abstraction are used to

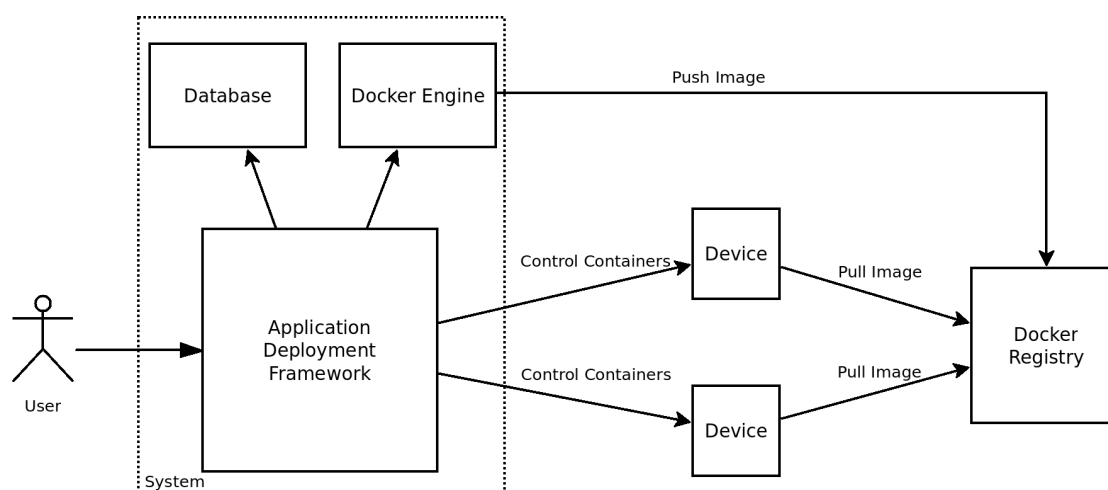


Figure 4.1: Container diagram of the framework

describe a system. First, we will describe it with a high level point of view, and then we give a more fine grained overview.

High Level Architecture

Figure 4.1 shows the container diagram of the prototype. Basically, the application deployment framework creates an image with the uploaded application and pushes the image to the Docker Registry. Afterwards, the framework triggers the pull command on the device and the device downloads the image from the Docker Registry. Controlling of the deployed containers is done through the web interface of the framework, which communicates directly with the devices through the Docker Engine API.

Component Diagram

Figure 4.2 presents all components of the framework and their dependencies. The Web Service used as an API gateway that forwards all requests to the corresponding microservice. Both, the Machine Service and the Authentication Service run their own database instance to store information about device and the users. The Containerization Service uses the local Docker Engine to build Docker images with the uploaded application. The Audit Service, Logging Service, and Deployment Service communicate directly with the device through the Docker Engine API.

Deployment Diagram

Figure 4.3 illustrates the deployment of the framework. Since we developed the framework with a microservice architecture, each service can be deployed on a different server. In

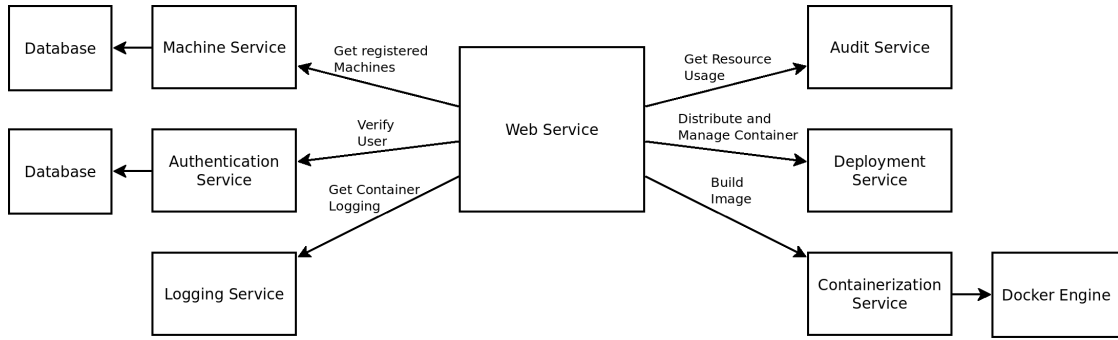


Figure 4.2: Component diagram of the framework

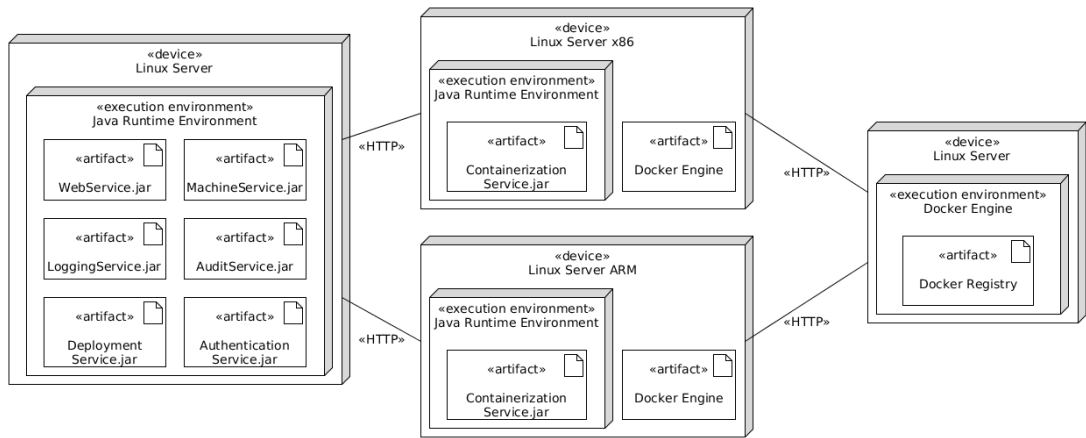


Figure 4.3: Deployment diagram of the framework

order to support devices with ARM and x86 architecture, we deploy two Containerization Services on two different servers. This allows to build Docker images for each platform. Containerization requires most computing power compared to the other services, and thus, the two microservices are deployed on a dedicated server. All other microservices are deployed on the same server to ensure fast communication between them. The Docker Registry should also be running on a separate server for security reasons, because it needs to be accessible by user devices. Furthermore, it needs a significant amount of disk space to store all Docker images.

The Containerization Services require a running Docker Engine to build the Docker images. This also applies for the Docker Registry, which is provided as a Docker image and thus, needs the Docker Engine as execution environment.

In order to deploy several instances of a service a load balancer is necessary. To duplicate the Authentication Service and the Machine Service it is necessary to switch to a separate Database instead of the embedded one.

4.2.3 Microservices

The framework consists of seven different microservices. In the following we will describe each of these services.

Web Service

The Web Service delivers the static web content to the user and serves all user requests through a REST API. Thus, it is used as an API gateway to all microservices. The web page is a single page application where all requests are done through AJAX calls to the REST API. All requests are then forwarded to the corresponding microservice. The Web Service also stores the session for currently logged in users. Moreover, it offers a REST interface to integrate the framework in a continuous delivery pipeline.

Containerization Service

The Containerization Service is one of the main components of the framework. It generates a Docker image with the user application according to the Dockerfile that was uploaded by the user. The resulting image is then pushed to the internal Docker Registry. In more detail, this works as follows:

- The service gets the configuration data for the Docker image (e.g., base image, application name, RUN commands, port mapping) and generates a Dockerfile.
- The uploaded application is downloaded from the Web Service.
- The local Docker Engine is called through the Docker Engine API to build the image according to the generated Dockerfile. The registry URL and a timestamp is added to the image name so that we can guarantee that the image has a unique ID.
- After the image is created, the image is pushed to the internal Docker Registry.

Deployment Service

The Deployment Service manages all states of the containers and distributes the Docker images to the devices. A container can have the following states: running, paused, and stopped. The Deployment Service exposes a REST API to switch containers to each of these states. It also allows to retrieve information about the current state and to remove containers and images from the device. The service communicates directly with the device through the Docker Engine API. The distribution of the image is handled through triggering of the pull command on the device, so that it pulls the image from the internal Docker Registry.

Machine Service

The Machine Service gives access to all registered devices for each user. It also offers an interface to ping registered devices in order to check if they are online. The database

stores the URL of each device, the port number through which the Docker Engine can be accessed, and the corresponding user id.

Authentication Service

The Authentication Service allows to add new users and authenticate them on login. It uses the *pbkdf2* algorithm with the *HmacSHA256* hashing function to generate and validate the passwords. Furthermore, we use 10000 iterations for the hash computation. The Authentication Service has its own database instance where it stores the username, the password hash, the salt, and the number of hashing iterations.

Logging Service

The Logging Service gives access to the logging output of each container. It uses the Docker Engine API to attach to a container on a certain host and delivers the whole logging output.

Audit Service

The Audit Service allows to monitor the resource utilization on a device. This is done through the Docker Engine API, which returns the CPU and RAM utilization for each container. To get the utilization of the whole device, we add up the resource usage of each container. The problem with this approach is that we only get the resource usage of the Docker containers without the processes outside of the containers. Docker does not support access to the system utilization.

4.2.4 Deployment process

Deployment of a new container involves several microservices and other applications. Figure 4.4 shows the sequence diagram of the deployment process starting with the upload of a new application. The following describes each step involved in this process in more detail:

- Step 1: The user uses the web UI to configure the container (e.g., port mapping, container name, or commands for the Dockerfile) and uploads the application.
- Steps 2 & 3: The Web Service requests the machine data from the Machine Service (e.g., URL or state)
- Step 4: The Web Service triggers the build process of the image on the Containerization Service.
- Steps 5 & 6: The Containerization Service downloads the application from the Web Service.

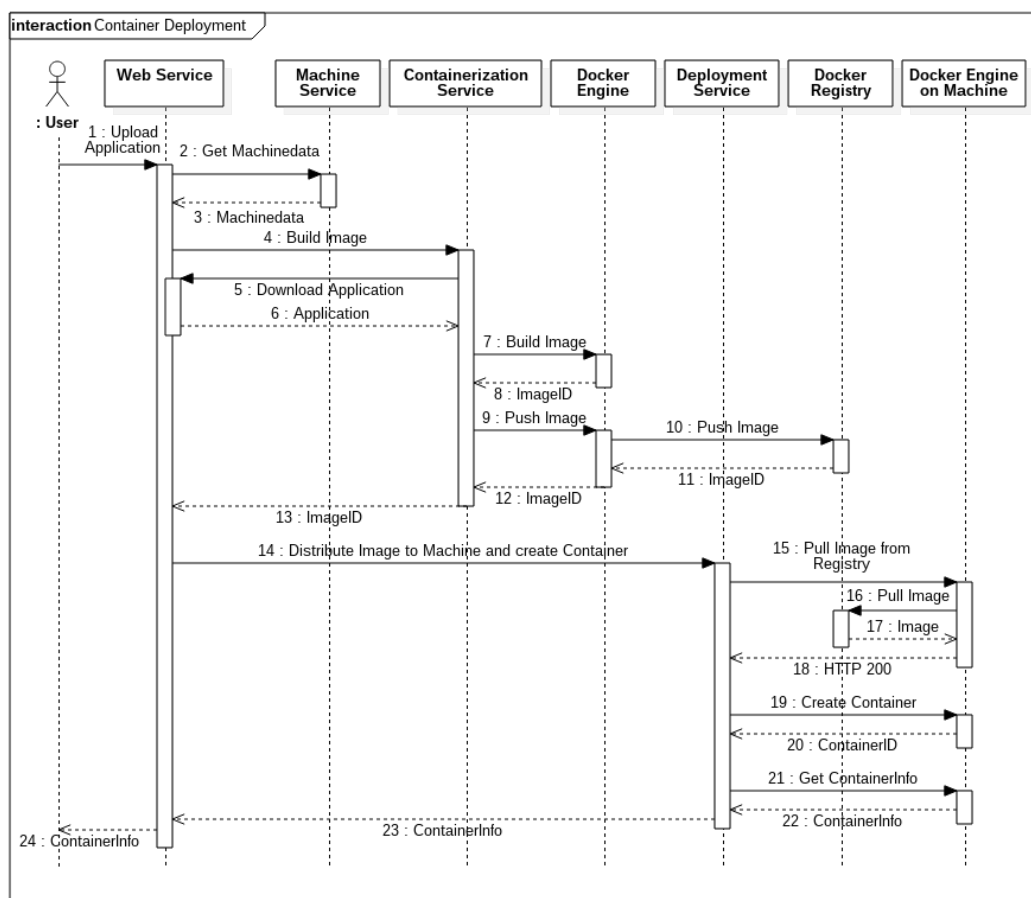


Figure 4.4: Sequence diagram of the deployment process

- Steps 7 & 8: The Containerization Service builds a Dockerfile according to the user configuration and executes the build command on the Docker Engine on the same host.
- Steps 9 - 13: The Containerization Service calls the push command on the local Docker Engine so that the image is pushed to the private Docker Registry.
- Step 14: The Web Service triggers the distribution of the image and the container creation for the selected machine on the Deployment Service.
- Steps 15 - 18: The Deployment Service triggers pulling of the image on the machine. The machine downloads the image from the Docker registry and responds with HTTP code 200 if there was no error.
- Steps 19 - 20: The Deployment Service triggers creation of the container on the machine.

- Steps 21 - 22: The Deployment Service requests container information.
- Steps 23 - 24: The container information is passed to the user, who sees the resulting container in the web UI.

4.3 Implementation

This section describes the technologies and libraries that were used for the implementation of the framework in detail. The last part covers *Lessons learned*, where some problems that came up during development and how we solved them are explained.

4.3.1 Technology Stack

The framework bases on three main technologies: Java 8, the Docker Engine, and the HSQLDB database.

Java

Java is an object-orientated programming language with static and strong typing. It was initially developed by Sun Microsystems which was acquired by Oracle Corporation in 2010. As explained in Chapter 2, Java uses the concept of application virtualization that enables platform independence. This is realized through the Java Virtual Machine (JVM) which executes the Java applications. Currently Java is available in version 8.

Today, Java is one of the most popular programming languages for web applications. This means that there are many frameworks and libraries available. Moreover, it is widely supported by build tools. For build automation we used Apache Maven. Although Maven supports various programming languages it is mainly used for Java projects.

Docker Engine

We used the Docker Engine in version 1.12.6. The following technical details are explained in this section: how images are built, how to configure the Docker Engine API, how the Docker Registry is deployed and integrated, and how networking works with Docker.

To build new images we generate a Dockerfile. Listing 4.1 shows the basic schema of this file. Line 1 defines the base image according to the user input. If the user configured extra commands for the image they are added in line 2. In line 3 the uploaded application is copied inside the container. The ENTRYPOINT command in line 4 ensures that the user application is executed on startup.

Listing 4.1: The generated Dockerfile

```
1 FROM <selected base image>
2 [USER COMMANDS]
3 ADD <user_application> <user_application>
4 ENTRYPOINT ["java", "-jar", "<user_application>"]
```

The Docker Engine API is a HTTP API that is mostly REST. It is used by the Docker CLI to communicate with the Docker daemon and offers remote access. Thus, everything that can be done with the local Docker CLI, can also be done through the API. To enable remote access, it is necessary to bind the Docker daemon on a TCP socket. This is done in the global config file which is stored in `/etc/docker/daemon.json` on Linux. The option `"hosts":["127.0.0.1:2375", "unix:///var/run/docker.sock"]` binds the daemon to the port 2375 on localhost, and the default UNIX socket `docker.sock`.

The Docker Registry can easily be deployed on a host. Listing 4.2 presents the command used to download the Docker Registry image and run the container attached to port 5000. Each client needs to set the Docker option `insecure-registry` with the IP and port where the registry is listening, to enable access. This is also done in the global config file. It is only needed to use an insecure registry, because we do not use TLS for the prototype. In order to use the private registry, images have to be tagged with the hostname or IP and port where it is reachable (e.g., `192.168.0.20:5000/ubuntu`), otherwise the default registry is used which is typically the official Docker Hub.

Listing 4.2: Deploying of the Docker Registry

```
1 docker run -d -p 5000:5000 --name registry registry:2
```

By default, Docker uses a bridge network called `docker0`. The Docker Engine automatically creates a subnet and a gateway to the network. Each container gets an IP address in this subnet through which it can be accessed. Users can define a port mapping of host ports to container ports in the web UI of the framework. The mapping automatically forwards all packets from the host to the container. This is done through iptable NAT rules. The port mapping needs to be set when the container is created from the Docker image, which is done through a Docker Engine API call. By default containers can connect to the outside world, but are not reachable from the outside. The implemented prototype allows to use both, the host network or a bridge network.

HSQldb

HSQldb [56] is an open source relational database management system which is available under a license that bases on the BSD license. We use HSQldb as the database for the framework, because of two reasons: First, it is written in Java and hence, easy to integrate in a Java project. Second, the Spring Framework supports auto-configuration for HSQldb. Basically, adding the HSQldb dependency in Maven is sufficient to get it

up and running. We use it in embedded mode, which means that it runs in the same JVM as the application and is invisible to the user.

4.3.2 Used Frameworks & Libraries

Several frameworks and libraries are used for the implementation of the framework. The most important ones are the Spring Framework, AngularJS, and the Spotify Docker Client.

Spring Framework

The Spring Framework is a framework for enterprise applications and a *Inversion of Control* container for Java. It provides several different modules, where each fits a certain use case. A commonly used module is Spring Boot. It allows to create stand-alone Spring applications with minimal effort in terms of configuration. There are also so called *starter* artifacts available that bundle several modules which are often used in combination. We used version 1.5.2 of the framework.

All microservices include the `spring-boot-starter-parent` and the `spring-boot-starter-web` artifacts. The `spring-boot-starter-parent` enables some default configurations that are typically used within Spring, which reduces the configuration effort even more. The `spring-boot-starter-web` artifact comprises a servlet container (Apache Tomcat), `spring-webmvc`, `jackson-databind`, etc., hence, it fits well for web applications.

To enable Spring Boot it is sufficient to add the dependencies in Maven and configure the main-class as shown in Listing 4.3.

Listing 4.3: Spring Boot configuration for the Authentication Service

```
1 @SpringBootApplication
2 public class AuthApp{
3     public static void main( String[] args ){
4         SpringApplication.run(AuthApp.class,args);
5     }
6 }
```

The Web Service uses the `spring-boot-starter-security`, which makes it very simple to secure a web application. The whole configuration is done in the `WebSecurityConfig` class, where basic HTTP authentication is enabled and the authentication provider is registered.

Both, the Machine Service and the Authentication Service use the `spring-boot-starter-data-jpa` artifact to access the HSQLDB database. Spring Data extends the Java Persistence API (JPA) with useful features that avoid boiler plate code. For example, it allows to use so called `Repositories` that have out-of-the-box support for basic operation like save, find, count or delete.

The REST API of each microservice makes use of the Spring `RestController` annotation. This annotation allows to configure the API through annotation of the methods. Listing 4.4 shows the implemented method to get the machine information by machine id. Line 2 defines the path of the method (e.g., `/machine/1`) and the allowed HTTP request method. The returned DTO is automatically converted into a JSON object by the Spring Framework.

Listing 4.4: Annotated method of Machine Service REST API

```
1 @RequestMapping
2 (value="/machine/{machineId}", method= RequestMethod.GET)
3 public MachineDTO getMachineById(@RequestParam Long machineId) {
4     return machineService.getMachineById(machineId);
5 }
```

AngularJS

AngularJS [79] is a front-end web application framework for single-page applications. We used version 1.5 in combination with the Twitter Bootstrap CSS framework to implement the front-end of the deployment framework.

The main page of the application is `index.html`. Navigation is done through loading of other HTML templates into this page. Furthermore, we defined so called directives that can be used for recurring elements. These directives are used for the list elements of the user machines and the list elements in the container overview.

The web application consists of four main pages: login, dashboard, container overview, and container logging. The login page is a simple form that allows the user to enter the username and password, and to login. The dashboard gives an overview about the registered devices, and supports removing and adding of new devices. The container overview shows all containers that are available on a particular device. Furthermore, it allows to control each container (e.g., start, stop, pause) and to upload a new application. The logging page shows the logging output of a selected container.

AngularJS follows the Model-View-Controller (MVC) pattern. Hence, we defined four different controllers, one for each main page. The controllers define all available javascript methods of the corresponding page. For example, when the add button on the dashboard is clicked, attached method in the dashboard controller is called.

After the application is loaded the first time, all further communication with the Web Service is done through AJAX calls on the REST interface.

Spotify Docker Client

The Spotify Docker Client [112] is an open source Docker client developed in Java. It can be seen as a wrapper for the HTTP calls to the Docker Engine API. Using this library makes it very simple to control the Docker Engine remotely.

Listing 4.5 shows how the `dockerClient` object is created and some example methods that are available. Line 2 configures the IP address and port number where the Docker Engine API is reachable and line 3 defines the timeout limit of the connection. Line 8 demonstrates an example call that pulls the `test_image` from the official Docker Registry to the device defined in line 2.

Listing 4.5: Spotify Docker Client usage

```
1 final DockerClient dockerClient = DefaultDockerClient.builder()
2     .uri(URI.create("http://192.168.0.20:2375"))
3     .connectTimeoutMillis(3000)
4     .build();
5
6 String response=dockerClient.ping();
7 Info info=dockerClient.info();
8 dockerClient.pull("test_image");
9
10 dockerClient.close();
```

4.3.3 Lessons Learned and Pitfalls

During development we encountered several pitfalls and learned some important lessons.

One thing we experienced was that the commands in a Dockerfile can be quite misleading and have to be read carefully. For example, the `EXPOSE` command does not expose the port to the host, instead it just exposes the port to other Docker containers. The `VOLUME` command does not support to mount a host directory inside the container, this has to be done within starting of a container.

Concerning Spring, a lesson that we learned was to check implicit dependencies carefully. On the one hand the framework comes out of the box without any need of complicated configuration, which makes it straightforward getting a new project up and running. On the other hand this means that many settings are hidden from the developer and it is hard to understand what really happens under the hood. The starter artifact, for example, implicitly defines Java 6 as the compiler level, which had to be overwritten in order to use Java 8.

When it comes to development itself, we learned that before new libraries are integrated, release notes and dependencies should be checked precisely. We wanted to use the Spring Eureka library for automatic service discovery. This worked well for the Web Service and the Machine Service. When we tried to integrate it in the Containerization Service, we ended up with a library conflict. The problem was that Eureka still used Jersey version 1, while the Spotify Client relied on version 2. In the end we had to get rid of Spring Eureka because of that.

Demonstration

This chapter describes a typical IoT application and shows how the developed application deployment framework simplifies the deployment and maintenance.

5.1 Use Case Definition

In order to demonstrate the feasibility of the introduced application deployment framework for the IoT domain, we define a typical use case in this section. We will implement the following Building Management System (BMS):

Let's assume we have a building with three floors (basement, 1. floor, and 2. floor) where we want to deploy a BMS. Each floor consists of a Raspberry PI that is connected to the WiFi of the building. There are several sensors located in the building: temperature sensors, humidity sensors, smoke detectors, motion detection sensors, contact sensor for the entrance door, etc. The Raspberry PIs are the central communication devices for these sensors on the corresponding floor. The sensors send their collected data through Bluetooth Low Energy to their assigned Raspberry PI. The data is then exposed in the network through a REST API. One Raspberry PI is also used as the central web server, and therefore, it aggregates the sensor data and makes it accessible via the Internet. Furthermore, the web server offers a GUI that shows an overview of the current conditions of the building.

5.2 Application Deployment and Management

The BMS consists of two different applications. One application is called `sensor_collector` and is used on every Raspberry Pi to collect sensor data. The second application is called `web_server` and needs to be deployed on one central Raspberry Pi to serve as web server.

5.2.1 Preconditions

We have to define some preconditions for the devices in order to be able to deploy the introduced BMS with the implemented framework. In particular the following requirements have to be fulfilled:

- The Raspberry PIs are reachable through the network (WLAN or LAN).
- Each device is provisioned with Linux and the Docker Engine that exposes the Docker Engine API on the network.
- Every Raspberry PI allows access to the insecure Docker Registry.

5.2.2 Application Deployment

Using the Web UI

The deployment of the BMS is done in several steps:

Machines

State	IP	Port
■	192.168.0.20	2375

■ Online ■ Offline ■ Unknown

[Add new Machine](#)

IP-Address

192.168.0.29

Port

2375

[Add](#)

Figure 5.1: Registration of a new device.

Step 1: Login and register all Raspberry PIs with the IP addresses and the ports where the Docker Engine API can be accessed. Figure 5.1 shows how to add a new device through the Web UI.

Step 2: Upload the application that collects the sensor data and configure the following parameters: container name, Docker base image, and the network type. Figure 5.2 shows the form to create and distribute the container.

Step 3: Start the container after uploading as presented in Figure 5.3. Figure 5.4 demonstrates the overview of all containers, the device information (e.g., architecture, operating system), and of the current resource utilization.

Step 4: Watch the logging output to make sure that the application starts up without any error. Repeat Step 2 - 4 for each application that collects sensor data and provides the REST API.

Add new Container

Containername

Application
 sensorcolle...APSHOT.war



Image
☒ Predefined Image ☐ Custom Image

Run Commands

Network
☒ Host ☐ Bridge

Figure 5.2: Upload of the application.

Containers at 192.168.0.29

Created	Name	CPU(%)	Memory(%)	Portmapping (External->Internal)	
 Apr 9, 2017 3:08:19 AM	epic_liskov	0.07%	15.85%	Host Network (ALL PORTS)	<input type="button" value="Logging"/> <input type="button" value="Action"/>
 Apr 10, 2017 1:10:59 AM	sensor_collector	--	--	Host Network (ALL PORTS)	<input type="button" value="Logging"/> <input type="button" value="Action"/>




 Running  Paused  Stopped

Figure 5.3: Start of the deployed container.

Step 5: Upload the web server application with the following parameters: container name, Docker base image, and network type.

Step 6: Check the logging output of the web server to ensure that it started up correctly.

Using the REST API

Since we implemented services with a REST API we can integrate the framework in a continuous delivery pipeline. Listing 5.1 shows the command that is necessary to deploy a new application to the Raspberry PI. This command produces the same container as described before through the Web interface. We used the username test and the password 1234 for authentication. The web server is hosted at 192.168.0.20 and the Raspberry PI is reachable through 192.168.0.29. In order to deploy the application as illustrated in Listing 5.1, the user has to register the device first.

5. DEMONSTRATION

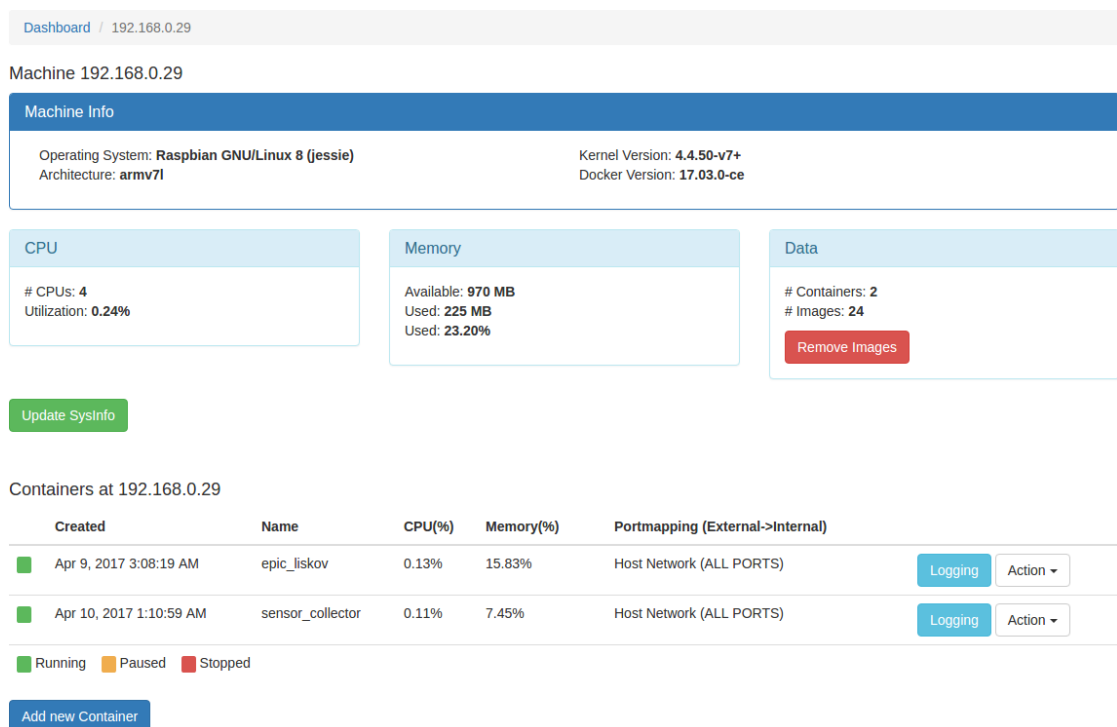


Figure 5.4: Running container with resource utilization.

Listing 5.1: Command to deploy an application through the REST API.

```
1 #upload the application to the web server (192.168.0.20:8080) and
2 #deploy it to the raspberry pi (192.168.0.29)
3 curl --user test:1234 \
4     -F "ip=192.168.0.29" \
5     -F "networkConfig=host" \
6     -F "containerName=sensor_collector" \
7     -F "imageName=dordoka/rpi-java8" \
8     -F "file=@sensorcollector-1.0-SNAPSHOT.war" \
9     192.168.0.20:8080/upload
```

5.2.3 Application Management

After deploying all applications, the resource utilization for each device and every container can be overseen through the Web UI, and thus, performance problems are encountered fast.

Let us consider that we want to role out a new Web Server GUI. In order to do that, we upload the new application to the corresponding Raspberry PI. If the same base image is used for the update, only the changed application has to be downloaded, which makes

Update

[illegible]

the distribution very fast. To be able to start the new container, first we have to stop the old web server to avoid port conflicts. Afterwards we can start the new container.

As presented in this chapter, deployment of an application is done within a few clicks using the introduced deployment framework, since configuration effort is reduced to a minimum. The only configuration that is required is the container name and the base image. All other parameters are automatically set by the introduced framework. If needed, fine grained control of the Dockerfile commands is possible, which allows to set up the resulting Docker image according to the users needs.

Furthermore, we described how the framework can be integrated within a continuous delivery pipeline through the REST API.

Evaluation

In this chapter we evaluate the deployment time of different Docker images on various platforms.

6.1 Scenarios

In order to evaluate the deployment time we define three different deployment scenarios:

- Scenario 1: A new application is deployed on a device. The base image was never used before and thus, is not stored in the local cache or Docker Registry.
- Scenario 2: An application is deployed on a device that has already been deployed before. Therefore, the resulting Docker image is already stored in the local cache and Docker Registry.
- Scenario 3: A modified application is deployed on a device. The base image and the Dockerfile commands have already been used before, only the uploaded application changed.

Furthermore, we use two different Dockerfiles for each scenario. One Dockerfile without any user specific commands (only the standard commands to add the uploaded application and execute it on startup), and another Dockerfile with a larger base image and commands to run apt-get update and install the apache2 server. The first Dockerfile (without run commands) is identified with a small a (e.g., 1a) and the second one with a small b (e.g., 1b).

Deployment in the context of this evaluation means that we create the Docker container on the device. We don't start the container since the start-up time is mostly depending on the application that gets started inside the container and not on Docker itself.

Device	Image from Dockerfile a	Image from Dockerfile b
Notebook & hosted server	138MB	677MB
Raspberry Pi	319MB	460MB

Table 6.1: Different image sizes used

OS	Ubuntu 14.04 LTS
Kernel	Kernel 4.4.0-72
Processor	Intel i5-4200U 2x 1.60GHz (up to 2.60)
Architecture	x86 64bit
RAM	12GB DDR3
Harddisk	SSD Samsung EVO 840 250GB
Docker	17.04.0-ce
Network	1000MBit

Table 6.2: Specifications of the Notebook

The base images used for the x86 architecture (Notebook and hosted server) are called `anapsix/alpine-java` (a) and `java:8` (b). For the ARM architecture (Raspberry Pi) we used `jsurf/rpi-java` (a) and `dordoka/rpi-java8` (b). The uploaded user application has 14.1MB.

6.2 Benchmark

6.2.1 Setup

We use three different devices for deployment: a Notebook, a Raspberry Pi 3 Model B, and a hosted server. The detailed specifications for the Notebook and the hosted server are listed in Table 6.2 and Table 6.3. The basic specifications of the Raspberry Pi are described in Chapter 3. It has the Raspbian operating system installed, which is based on Debian 8.

The application deployment framework is deployed on a PC in the local network. Table 6.4 shows its specifications. In order to be able to build a Docker image for the Raspberry Pi (armv8/armv7l architecture), we use another Raspberry Pi where the Containerization Service is executed. Thus, we have two Containerization Services, one for the ARM architecture and one for the x86 architecture. Furthermore, the Docker Registry is also running on the PC. This means that all microservices except one of the Containerization Services communicate via the Linux loopback device with each other.

Each scenario was tested ten times with an automated script. The registry and local cache are purged after each deployment for scenario 1. The error bar diagrams show the average, as well as the min and max of the measured times. The measured values can be

OS	Ubuntu 16.04 LTS
Kernel	Kernel 4.4.0-72
Processor	Intel Xeon E5-2650L 2x 2.3 GHz
Architecture	x86 64bit
RAM	2GB DDR3
Harddisk	40GB SSD
Docker	17.03.1-ce
Vendor	DigitalOcean
Location	Frankfurt
Round-trip time	20ms - 30ms

Table 6.3: Specifications of the hosted server

OS	Ubuntu 16.04 LTS
Kernel	Kernel 4.4.0-72
Processor	Intel i5-3570k 4x 3.40GHz
Architecture	x86 64bit
RAM	8GB DDR3
Harddisk	SSD Samsung EVO 850 250GB
Network	1000MBit
Docker	1.12.6
Internet	75MBit Download 7.5MBit Upload (measured)

Table 6.4: Specifications of the used PC

found in the Appendix A of the thesis.

6.2.2 Notebook

As we can see in Figure 6.1 the deployment time is quite high for new applications, but decreases significantly if the Docker image is already cached on the device (e.g., scenario 2). Figure 6.2 shows that the high variation of the deployment time of scenario 1b comes mostly from the image building process. Furthermore, Figure 6.3 and Figure 6.4 illustrate that pushing to the Docker Registry takes nearly twice the time than pulling. In Figure 6.5 we can see that container creation is very fast, but also varies heavily.

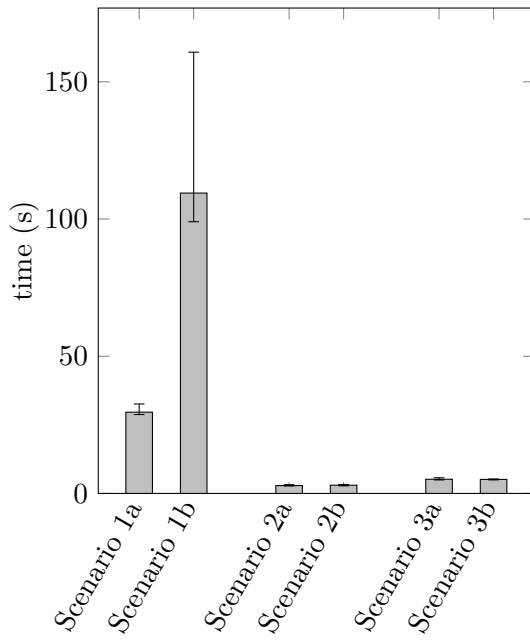


Figure 6.1: Overall deployment

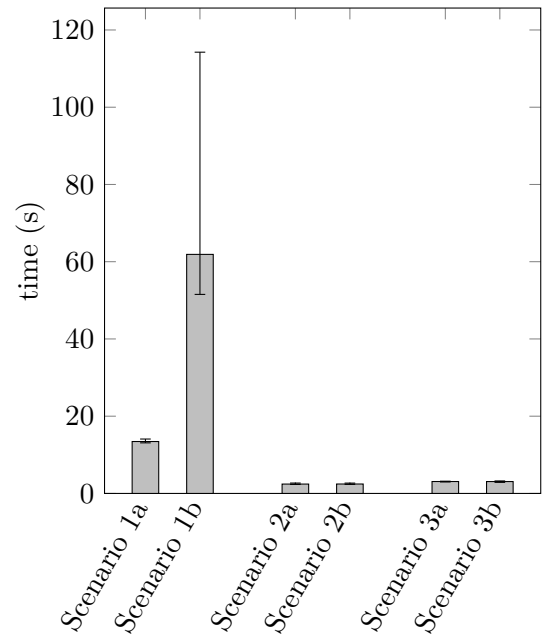


Figure 6.2: Image building

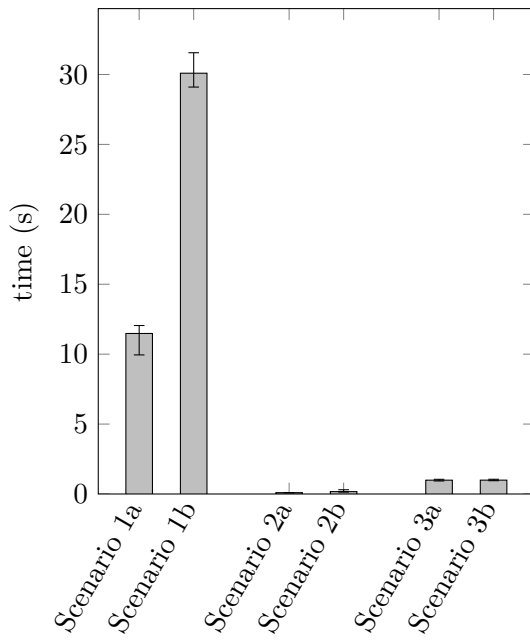


Figure 6.3: Pushing to registry

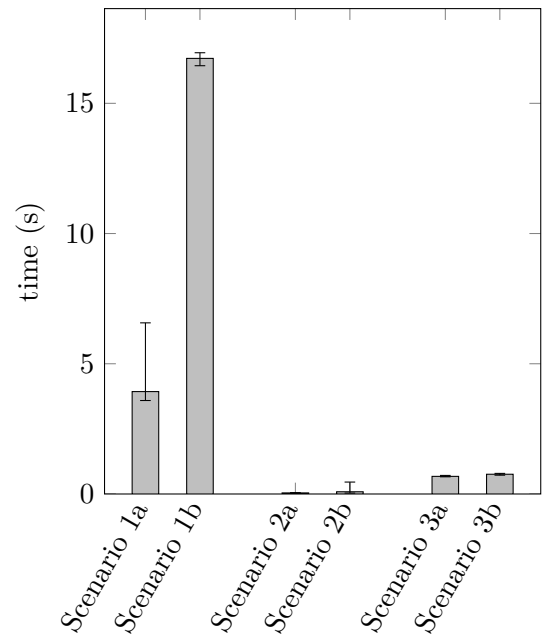


Figure 6.4: Pulling of the image

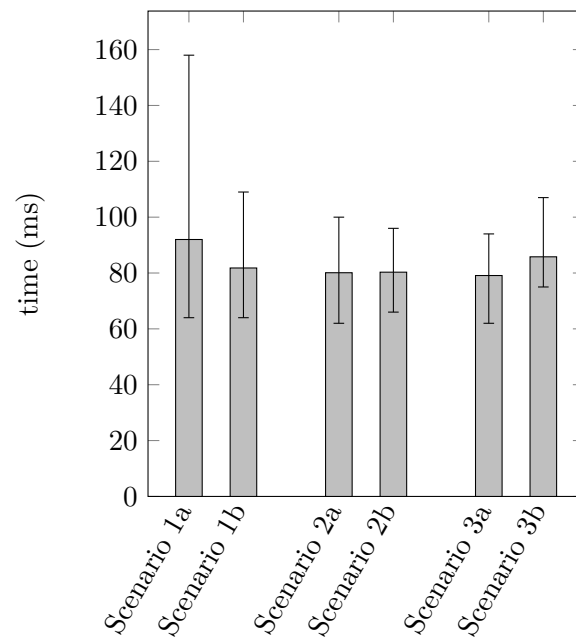


Figure 6.5: Container creation

6.2.3 Hosted Server

Figure 6.6 demonstrates a very high deployment time for new applications. As can be seen in Figure 6.7 this is not caused by the image building process, but by pulling the image to the server (Figure 6.9). This can be explained by the limited upload speed of the deployment server.

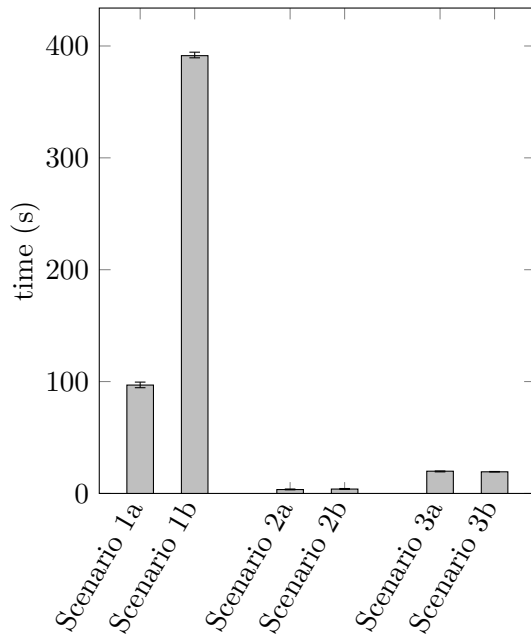


Figure 6.6: Overall deployment

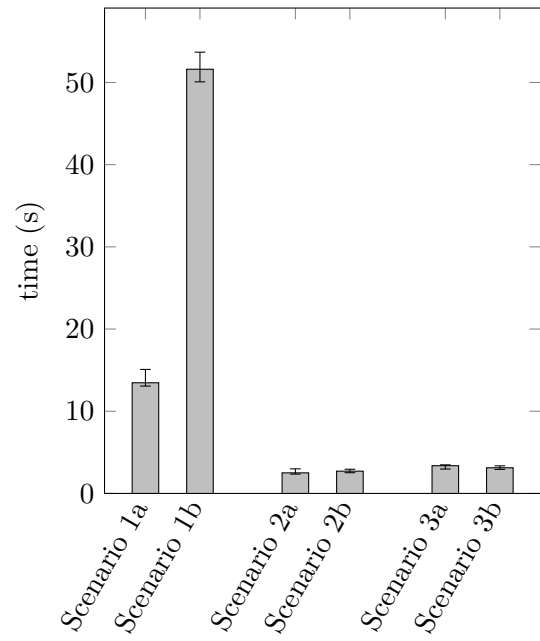


Figure 6.7: Image building

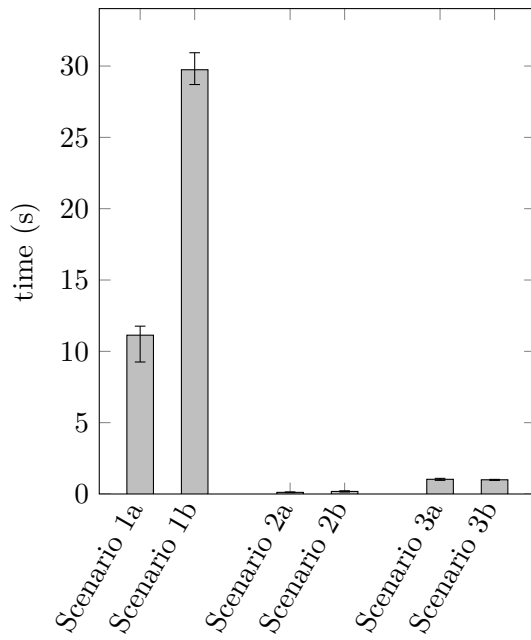


Figure 6.8: Pushing to registry

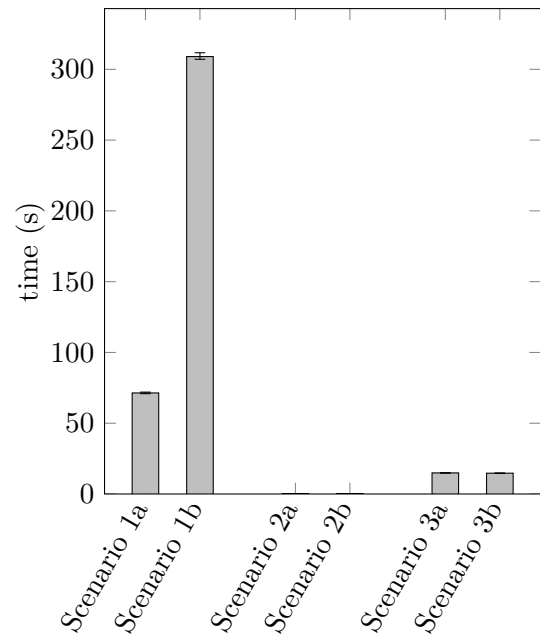


Figure 6.9: Pulling of the image

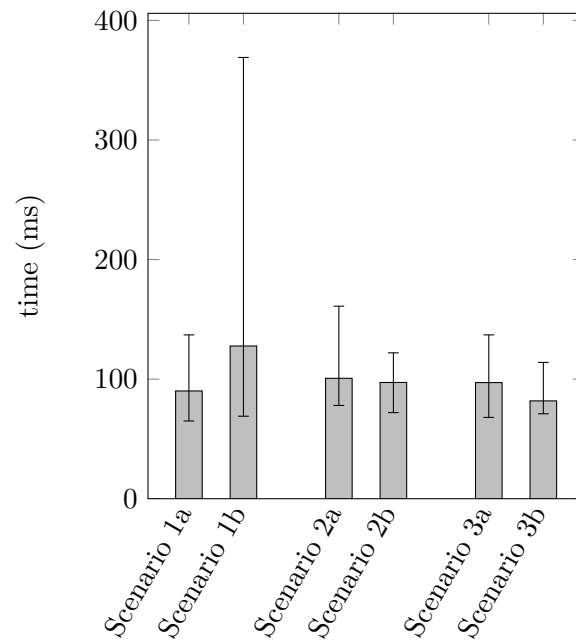


Figure 6.10: Container creation

6.2.4 Raspberry Pi

The Raspberry Pi has by far the highest deployment time compared to the other devices. As can be seen in Figure 6.12, Figure 6.13, and Figure 6.14, each deployment process takes significantly longer than on other devices and it is not due to one particular event. This is mostly caused by the bad I/O performance of the SD-card that is used in the Raspberry Pi.

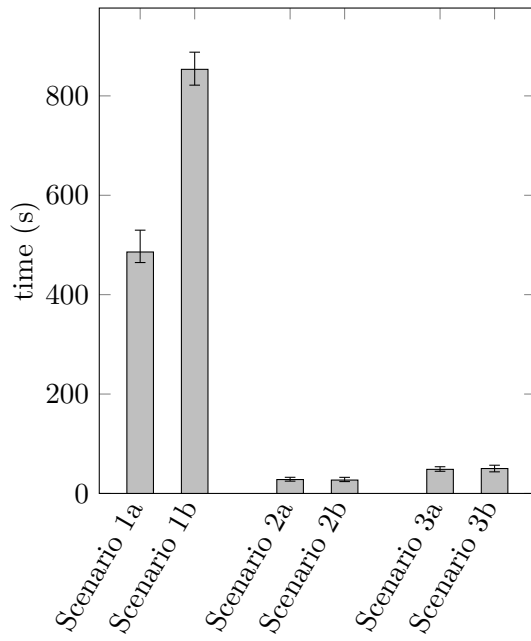


Figure 6.11: Overall deployment

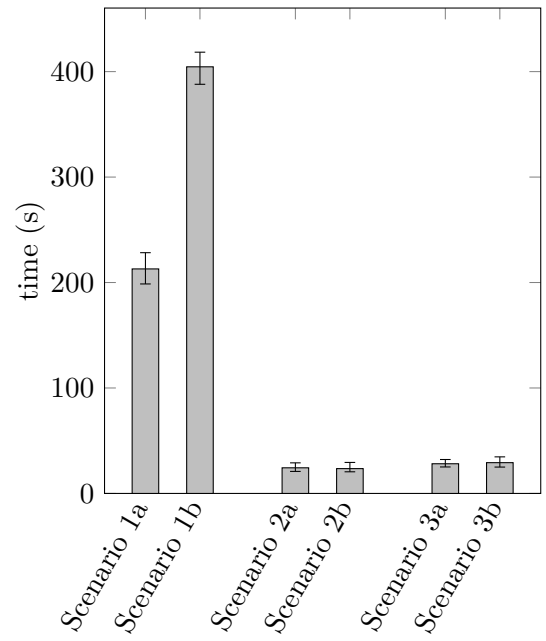


Figure 6.12: Image building

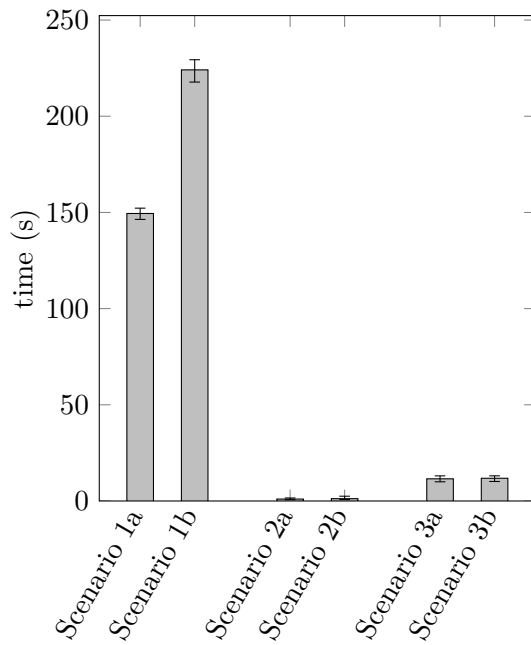


Figure 6.13: Pushing to registry

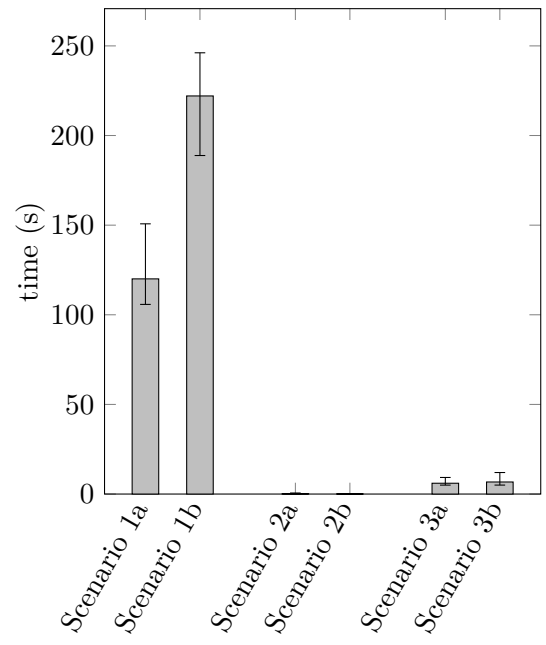


Figure 6.14: Pulling of the image

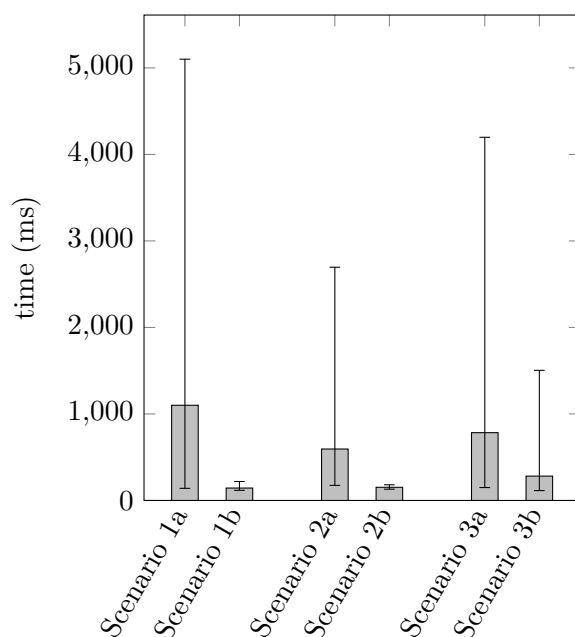


Figure 6.15: Container creation

6.3 Discussion

As we can see in Figure 6.1, Figure 6.6, and Figure 6.11, the deployment time for new applications with a new base image (scenario 1), heavily relies on the image size and whether additional Dockerfile commands are executed or not. The high variation of deployment times on the Notebook for scenario 1b is caused by the image building process as shown in Figure 6.2. This is mostly because the `apt-get update` command checks for updates on many servers, which can lead to delays if one of the servers has a high response time. The deployment time on the hosted server was mainly influenced by pulling the image to the device. This is due to the fact, that it requires uploading from the local Docker Registry to the hosted server, which is limited to 7.5Mbit. The long time for deployment on the Raspberry Pi can be explained with the bad I/O performance of the device. This is caused by the SD card, which is way slower compared to the SSDs used for the other devices. Considering that we also use a Raspberry Pi for image building this has a massive effect on the deployment time. Faster image creation could be achieved by using of an ARM server or with a Network File System (NFS) mounted within the Raspberry Pi.

When an application is deployed a second time (scenario 2), the deployment time decreases tremendously. This is caused by the intelligent caching mechanism implemented in Docker, which compares the hashes and run commands, and recognizes if an image has already been built before. It is independent of the resulting image size and whether commands in the Dockerfile are used or not.

The measurement also shows that if a changed application is deployed (scenario 3), the caching works very well. Only the changed application is transferred to the device and thus, the deployment time is very low compared to a new deployment (scenario 1).

Creating a container out of the resulting Docker image is done fast as shown in Figure 6.5, Figure 6.10, and Figure 6.15, since Docker uses the copy-on-write mechanism. The image is used for the container without copying of any data, only a writeable layer is added on top of the image. The measurement also showed that the creation time varies significantly, independent of image size or device. It is worth mentioning that the container creation on the Raspberry Pi takes longer for the smaller image. The reason for this is probably that the smaller base image has way more layers than the larger base image. This is because the image is created from another base image itself. The high time variation on the hosted server is probably also caused by the round trip time to the datacenter.

Conclusion

This chapter discusses the outcome of the thesis and gives an outlook on future work.

7.1 Summary

Application deployment is a critical topic in the domain of IoT. Since there are naturally many devices used, managing application distribution and deployment manually becomes very time consuming and error prone. Furthermore, the variety of devices lead to a heterogeneous environment with different CPUs and thus, different architectures.

In the course of this thesis, we investigated how operating-system-level virtualization can be applied to cope with these problems. When it comes to platform independence this approach offers a good way to abstract from the underlying operating system and allows to deploy the same image on different devices without any adaptations needed. However, there are some device specific features that cannot be abstracted. For example, applications that are compiled for a x86 architecture cannot be deployed on an ARM device. Moreover, also Java has some limitations when native libraries or JNI is used, as described in Chapter 3.

OS virtualization offers nearly native performance, because it is a very lightweight virtualization approach as described in Chapter 2. The only drawback compared to a native approach is that containers require a significant amount of disk space since the whole Linux userspace is packed into it. As explained in Chapter 3 and evaluated in Chapter 5, Docker handles this problem very well through smart layering of the images and applying of the so called copy-on-write concept. This allows to reuse an image for every container that builds upon it. Furthermore, the startup time is very low because no data has to be copied.

The introduced framework allows to deploy, monitor, and debug applications within a few clicks, and therefore, simplifies application deployment and management. Moreover,

since it is developed as a microservice architecture, each service can easily be distributed according to the domain specific needs. This allows, for example, to deploy the service that builds the Docker images on different architectures and thus, leads to better platform independence.

There are some limitations using operating-system-level virtualization in context of IoT. The most crucial one is that it requires an operating system with support for a container engine installed on the device. Considering that many IoT devices have limited computing power, this limits the number of usable devices. Another drawback is that it is restricted to platforms that are supported by the container engine. Currently, Docker only supports the x86 and ARM architecture.

In summary, in this thesis we showed that operating-system-level virtualization can be applied for IoT and offers platform independence to a great extent. The implemented prototype gave insights in the deployment time of different applications. The time to create and distribute images is quite high for new applications, because of the huge image sizes, but decreases significantly when images are already cached on the device.

7.2 Future Work

Since the developed prototype is a proof of concept, there are many features and changes that could be considered for future work. The most important ones are described in this section.

7.2.1 Security

During designing and developing we did not take care of security issues. For example, the Docker Engine API is not protected at all and can be used on the devices without any authentication. Obviously, this cannot be done in production. Therefore, significant effort has to be taken to secure the clients and the Docker Registry. Since this is a very sensitive topic when it comes to IoT, securing of the application has to be done with great caution and very carefully.

7.2.2 Client Agent

We decided to use an agent less approach for the prototype to save resources. As described this makes the framework highly dependable on the Docker Engine API. To offer more features and support other container engines it is required to implement an agent for the devices. Another approach would be to use ssh, which would allow full control of the device. Although this approach requires more resources, the benefits outweigh the cost.

7.2.3 Source Compilation

In order to enhance support of different platforms and programming languages it is necessary to compile the source code on the respective architecture. Therefore, an option

to upload the source code together with the build instructions instead of the application would offer more flexibility. Currently only Java is supported since it is to a great extent platform independent.

7.2.4 Image Overview

Since the deployment time of new applications is highly dependent on whether the base image is already used or not, it is necessary to show currently stored images on a device. This allows the user to reuse old images, and therefore, only the new application has to be delivered to the selected device.

7.2.5 Dockerfile Storage

A typical usecase is to deploy a new version of an application. Thus, most of the time the Dockerfile to build the image will not change. Therefore, an option to use the Dockerfile of an already created container would make the deployment configuration much faster and less error prone.

7.2.6 Container Management

In terms of container management there are plenty of features that need to be implemented to offer better support. Some very interesting ones are:

- Continuous monitoring and health checking
- Automatic redeployment in case of container failures
- Automatic replication to scale out on high load
- Rolling updates

List of Figures

2.1	Example of a network architecture for the IoT	8
2.2	Different levels of virtualization	11
2.3	Deployment pipeline, adapted from [64]	17
3.1	Standards used in IoT applications [58]	23
3.2	The IPv6 header (adapted from [38])	25
3.3	Example of a cgroup hierarchy attached to the CPU subsystem (adapted from [100])	30
3.4	The Docker Engine architecture [101]	36
3.5	A Docker image with the container layers on top [78]	37
3.6	The different stages of rkt [72]	40
3.7	Architecture of Kubernetes [91]	43
3.8	Application deployment with resin.io [105]	46
3.9	IoT device architecture in resin.io [105]	47
4.1	Container diagram of the framework	53
4.2	Component diagram of the framework	54
4.3	Deployment diagram of the framework	54
4.4	Sequence diagram of the deployment process	57
5.1	Registration of a new device.	64
5.2	Upload of the application.	65
5.3	Start of the deployed container.	65
5.4	Running container with resource utilization.	66
5.5	Logging output of the selected container.	67
6.1	Overall deployment	72
6.2	Image building	72
6.3	Pushing to registry	72
6.4	Pulling of the image	72
6.5	Container creation	73
6.6	Overall deployment	74
6.7	Image building	74
6.8	Pushing to registry	74

6.9	Pulling of the image	74
6.10	Container creation	75
6.11	Overall deployment	76
6.12	Image building	76
6.13	Pushing to registry	76
6.14	Pulling of the image	76
6.15	Container creation	77

List of Tables

3.1	Specifications of the Arduino Uno Revision 3 [15]	20
3.2	Specifications of the Dell Edge Gateway 5000 [39]	20
3.3	Specifications of the Raspberry Pi 3 Model B [47]	21
3.4	Specifications of the UDOO Neo Full [82]	21
6.1	Different image sizes used	70
6.2	Specifications of the Notebook	70
6.3	Specifications of the hosted server	71
6.4	Specifications of the used PC	71
A.1	Overall deployment (ms)	95
A.2	Create image (ms)	95
A.3	Push image (ms)	96
A.4	Pull image (ms)	96
A.5	Create container (ms)	96
A.6	Overall deployment (ms)	97
A.7	Create image (ms)	97
A.8	Push image (ms)	97
A.9	Pull image (ms)	98
A.10	Create container (ms)	98
A.11	Overall deployment (ms)	99
A.12	Create image (ms)	99
A.13	Push image (ms)	99
A.14	Pull image (ms)	100
A.15	Create container (ms)	100

Bibliography

- [1] *CGROUP_NAMESPACES(7) Linux Programmer's Manual.*
- [2] *CGROUPS(7) Linux Programmer's Manual.*
- [3] *CHROOT(2) FreeBSD System Calls Manual.*
- [4] *CHROOT(2) Linux Programmer's Manual.*
- [5] *MOUNT_NAMESPACES(7) Linux Programmer's Manual.*
- [6] *NAMESPACES(7) Linux Programmer's Manual.*
- [7] Open Container Initiative. <https://www.opencontainers.org> Accessed 11/2016.
- [8] *RCTL(8) FreeBSD System Manager's Manual.*
- [9] resin.io. <https://resin.io/> Accessed: 03/2017.
- [10] Yocto Project. <https://www.yoctoproject.org/> Accessed: 03/2017.
- [11] AppArmor FAQ, 2011. <http://wiki.apparmor.net/index.php/FAQ> Accessed 01/2017.
- [12] Comparison, 2016. <https://openvz.org/Comparison> Accessed 01/2017.
- [13] Ieee standard for low-rate wireless networks. *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)*, pages 1–709, April 2016.
- [14] Allan Afuah. How much do your "co-opetitors'" capabilities matter in the face of technological change? *Strategic Management Journal*, pages 387–404, 2000.
- [15] Arduino AG. Arduino Uno SMD, 2017. <https://www.arduino.cc/en/Main/ArduinoBoardUnoSMD> Accessed 02/2017.
- [16] Wi-Fi Alliance. Wi-Fi Alliance introduces low power, long range Wi-Fi HaLow. <http://www.wi-fi.org/news-events/newsroom/wi-fi-alliance-introduces-low-power-long-range-wi-fi-halow> Accessed: 02/2017.

- [17] ZigBee Alliance et al. ZigBee Specification, 2006.
- [18] Matt Asay. Why Kubernetes is winning the container war. <http://www.infoworld.com/article/3118345/cloud-computing/why-kubernetes-is-winning-the-container-war.html> Accessed: 03/2017.
- [19] Alex Ashley and Stephen McCann. Official IEEE 802.11 Working Group Project Timelines - 2017-01-26. http://grouper.ieee.org/groups/802/11/Reports/802.11_Timelines.htm Accessed: 02/2017.
- [20] Kevin Ashton. That 'Internet of Things' Thing, 2009. <http://www.rfidjournal.com/articles/view?4986> Accessed 09/2016.
- [21] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
- [22] Alessandro Bassi and Geir Horn. Internet of things in 2020: A roadmap for the future. *European Commission: Information Society and Media*, 2008.
- [23] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [24] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
- [25] SIG Bluetooth. Covered Core Package, Version: 4.2;. *Specification of the Bluetooth System*, 2014.
- [26] Joshua Bressers. Is chroot a security feature?, 2013. <https://access.redhat.com/blogs/766093/posts/1975883> Accessed 12/2016.
- [27] Simon Brown. Agile software architecture sketches and nouml, 2013. <https://www.infoq.com/articles/agile-software-architecture-sketches-NoUML> Accessed: 03/2017.
- [28] Diego Calleja. Linux 2.6.24, 2010. https://kernelnewbies.org/Linux_2_6_24 Accessed 11/2016.
- [29] Diego Calleja. Linux 3.8, 2013. https://kernelnewbies.org/Linux_3.8 Accessed 11/2016.
- [30] Diego Calleja. Linux 4.6, 2016. https://kernelnewbies.org/Linux_4.6 Accessed 12/2016.
- [31] Patrick Chanezon. Docker containerd Kubernetes sig node. <https://www.slideshare.net/chanezon/docker-containerd-kubernetes-sig-node> Accessed: 03/2017.

- [32] Peter M Chen and Brian D Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138. IEEE, 2001.
- [33] Susanta Nanda Tzi-cker Chiueh. A survey on virtualization technologies. *RPE Report*, pages 1–42, 2005.
- [34] CoreOS. etcd. <https://github.com/coreos/etcd> Accessed: 03/2017.
- [35] Oracle Corporation. Features Provided by Non-Global Zones, 2010. <https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-9/index.html> Accessed 01/2017.
- [36] Oracle Corporation. Immutable Zones, 2014. http://docs.oracle.com/cd/E36784_01/html/E36848/glglv.html Accessed 01/2017.
- [37] Oracle Corporation. Zones Overview, 2014. http://docs.oracle.com/cd/E36784_01/html/E36848/zones.intro-2.html Accessed 01/2017.
- [38] S Deering and R Hinden. Internet protocol, version 6 (ipv6) specification. 1998.
- [39] Dell Technologies Inc. *Edge Gateway 5000 Series Spec Sheet*, 5 2016.
- [40] DevOps.com, ClusterHQ. Container market adoption, 2016. <https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf> Accessed 11/2016.
- [41] Sandra Dominikus and Jörn-Marc Schmidt. Connecting passive rfid tags to the internet of things. In *Interconnecting Smart Objects with the Internet Workshop, Prague*, 2011.
- [42] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.
- [43] Jake Edge. A seccomp overview, 2015. <https://lwn.net/Articles/656307/> Accessed 01/2017.
- [44] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.
- [45] NFC Forum. What are the operating modes of NFC devices? <http://nfc-forum.org/resources/what-are-the-operating-modes-of-nfc-devices/> Accessed: 02/2017.
- [46] Cloud Native Computing Foundation. Kubernetes, 2017. <https://github.com/kubernetes/kubernetes> Accessed 03/2017.

- [47] Raspberry Pi Foundation. Raspberry Pi 3 Model B, 2017. <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/> Accessed 02/2017.
- [48] Martin Fowler. Continuous Integration, 2006. <http://martinfowler.com/articles/continuousIntegration.html> Accessed 11/2016.
- [49] Martin Fowler. Continuous Delivery, 2013. <http://martinfowler.com/bliki/ContinuousDelivery.html> Accessed 10/2016.
- [50] Martin Fowler. DeploymentPipeline, 2013. <http://martinfowler.com/bliki/DeploymentPipeline.html> Accessed 11/2016.
- [51] Martin Fowler. Microservice trade-offs, 2015. <https://martinfowler.com/articles/microservice-trade-offs.html> Accessed: 03/2017.
- [52] FreeBSD. Docker on FreeBSD, 2016. <https://wiki.freebsd.org/Docker> Accessed 01/2017.
- [53] Vangelis Gazis, Manuel Görtz, Marco Huber, Alessandro Leonardi, Kostas Mathioudakis, Alexander Wiesmaier, Florian Zeiger, and Emmanouil Vasilomanolakis. A survey of technologies for the internet of things. In *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1090–1095. IEEE, 2015.
- [54] Fred Grosshans and Diego Calleja. Linux 2.6.30, 2009. https://kernelnewbies.org/Linux_2_6_30 Accessed 11/2016.
- [55] OMG Object Management Group. Data Distribution Service, V1.4, 2017. <http://www.omg.org/spec/DDS/1.4/> Accessed 03/2017.
- [56] The HSQL Development Group. Hypersql, 2017. <http://hsqldb.org/> Accessed: 03/2017.
- [57] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, 2013.
- [58] Patrick Guillemin, Friedbert Berens, Marco Carugi, Marilyn Arndt, Latif Ladid, George Percivall, Bart De Lathouwer, Steve Liang, Arne Bröring, and Pascal Thubert. Internet of things standardisation—status, requirements, initiatives and organisations. *RIVER PUBLISHERS SERIES IN COMMUNICATIONS*, page 259, 2013.
- [59] Tejun Heo. Control Group v2, 2015. <https://www.kernel.org/doc/Documentation/cgroup-v2.txt> Accessed 12/2016.

- [60] Yu-Ju Hong. Introducing Container Runtime Interface (CRI) in Kubernetes. <http://blog.kubernetes.io/2016/12/container-runtime-interface-cri-in-kubernetes.html> Accessed: 03/2017.
- [61] Jez Humble. Deployment pipeline anti-patterns, 2010. <https://continuousdelivery.com/2010/09/deployment-pipeline-anti-patterns/> Accessed 11/2016.
- [62] Jez Humble. Continuous Testing, 2013. <https://continuousdelivery.com/foundations/test-automation> Accessed 11/2016.
- [63] Jez Humble. Continuous Delivery, 2016. <https://continuousdelivery.com> Accessed 10/2016.
- [64] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
- [65] Jez Humble, Chris Read, and Dan North. The deployment production line. In *AGILE*, volume 6, pages 113–118, 2006.
- [66] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. Mqtt-s—a publish/-subscribe protocol for wireless sensor networks. In *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*, pages 791–798. IEEE, 2008.
- [67] Sadequl Hussain. An Introduction to SELinux on CentOS 7 – Part 1: Basic Concepts, 2014. <https://www.digitalocean.com/community/tutorials/an-introduction-to-selinux-on-centos-7-part-1-basic-concepts> Accessed 01/2017.
- [68] Bluetooth SIG Inc. Bluetooth 5: What it’s all about, 2017. <https://www.bluetooth.com/specifications/bluetooth-core-specification/bluetooth5> Accessed 03/2017.
- [69] CoreOS Inc. App Container basics. <https://coreos.com/rkt/docs/latest/app-container.html> Accessed 11/2016.
- [70] CoreOS Inc. Networking. <https://coreos.com/rkt/docs/latest/networking/overview.html> Accessed 02/2017.
- [71] CoreOS Inc. rkt - the pod-native container engine, 2017. <https://github.com/coreos/rkt/> Accessed 01/2017.
- [72] CoreOS Inc. rkt architecture, 2017. <https://coreos.com/rkt/docs/latest/devel/architecture.html> Accessed 01/2017.
- [73] Docker Inc. Docker container networking. <https://docs.docker.com/engine/userguide/networking/> Accessed 02/2017.

- [74] Docker Inc. containerd, 2016. <https://containerd.io/> Accessed 01/2017.
- [75] Docker Inc. Windows Server and Docker - The Internals Behind Bringing Docker and Containers to Windows by Taylor Brown and John Starks, 2016. <http://de.slideshare.net/Docker/windows-server-and-docker-the-internals-behind-bringing-docker-and-containers-to-windows-by-taylor-brown-and-john-starks> Accessed 02/2017.
- [76] Docker Inc. Docker, 2017. <https://github.com/docker/docker> Accessed 01/2017.
- [77] Docker Inc. Install Docker on macOS, 2017. <https://docs.docker.com/engine/installation/mac/> Accessed 01/2017.
- [78] Docker Inc. Understand images, containers, and storage drivers, 2017. <https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/> Accessed 01/2017.
- [79] Google Inc. Angularjs, 2017. <https://angularjs.org/> Accessed: 03/2017.
- [80] Pivotal Software Inc. Moments in container history, 2016. <https://pivotal.io/platform/infographic/moments-in-container-history> Accessed 12/2016.
- [81] Red Hat Inc. OpenShift. <https://www.openshift.com/> Accessed: 03/2017.
- [82] SECO USA Inc. UDOO NEO, 2017. <http://www.udoo.org/udoo-neo/> Accessed 02/2017.
- [83] Open Container Initiative. Open Container Initiative Charter. <https://www.opencontainers.org/about/governance> Accessed 11/2016.
- [84] Business Insider. How the 'Internet of Things' will impact consumers, businesses, and governments in 2016 and beyond, 2016. <http://www.businessinsider.de/how-the-internet-of-things-market-will-grow-2014-10?r=US&IR=T> Accessed 07/2016.
- [85] Information technology – Telecommunications and information exchange between systems – Near Field Communication – Interface and Protocol (NFCIP-1). Standard, International Organization for Standardization, Geneva, CH, March 2013.
- [86] Information technology – message queuing telemetry transport (mqtt) v3.1.1. Standard, International Organization for Standardization, Geneva, CH, June 2016.
- [87] Poul-Henning Kamp and Robert NM Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International SANE Conference*, volume 43, page 116, 2000.

- [88] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview, 2013. <https://lwn.net/Articles/531114/> Accessed 11/2016.
- [89] Michael Kerrisk. Namespaces in operation, part 3: PID namespaces, 2013. <https://lwn.net/Articles/531419/> Accessed 11/2016.
- [90] Michael Kerrisk. Linux 4.5, 2016. https://kernelnewbies.org/Linux_4.5 Accessed 11/2016.
- [91] Khtan66. File:Kubernetes.png. <https://commons.wikimedia.org/wiki/File:Kubernetes.png> Accessed: 03/2017.
- [92] Scott D. Lowe. What is the difference between emulation vs. virtualization?, 2013. <http://www.virtualizationadmin.com/blogs/lowe/news/what-difference-between-emulation-vs-virtualization.html> Accessed 10/2016.
- [93] Canonical Ltd. What's LXD? <https://linuxcontainers.org/lxd/introduction/> Accessed 02/2017.
- [94] Gabriel Montenegro, Nandakishore Kushalnagar, Jonathan Hui, and David Culler. Transmission of IPv6 packets over IEEE 802.15. 4 networks. Technical report, 2007.
- [95] Adrian Mouat. 5 security concerns when using Docker. <https://www.oreilly.com/ideas/five-security-concerns-when-using-docker> Accessed: 02/2017.
- [96] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., February 2015.
- [97] A Peter and José Fortes. Resource virtualization renaissance. 2005.
- [98] Alex Polvi. App Container and the Open Container Project, 2015. <https://coreos.com/blog/app-container-and-the-open-container-project/> Accessed 11/2016.
- [99] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.
- [100] Red Hat Customer Portal. Relationships Between Subsystems, Hierarchies, Control Groups and Tasks. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-Relationships_Between_Subsystems_Hierarchies_Control_Groups_and_Tasks.html Accessed 12/2016.
- [101] Arnaud Porterie. Docker 1.11: The First Runtime Built On Containerd And Based On OCI Technology, 2016. <https://blog.docker.com/2016/04/docker-engine-1-11-runc/> Accessed 01/2017.

- [102] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *LISA*, volume 4, pages 241–254, 2004.
- [103] The Jenkins Project. Jenkins. <https://jenkins.io/> Accessed: 03/2017.
- [104] Red Hat Inc. OpenShift Origin. <https://www.openshift.org/> Accessed: 03/2017.
- [105] resin.io. How does resin.io work?, 2017. <https://resin.io/how-it-works/> Accessed 03/2017.
- [106] Rami Rosen. Understanding the new control groups API, 2016. <https://lwn.net/Articles/679786/> Accessed 12/2016.
- [107] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226, April 2010.
- [108] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, RFC Editor, March 2011.
- [109] Chetan Sharma. Correction of IoT History. http://www.chetansharma.com/IoT_History.htm Accessed 11/2016.
- [110] Z Shelby, K Hartke, and C Bormann. The constrained application protocol (coap). Technical report, 2014.
- [111] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.
- [112] Spotify. docker-client - a simple docker client for the jvm, 2017. <https://github.com/spotify/docker-client> Accessed: 03/2017.
- [113] Rohan Tabish, Adel Ben Mnaouer, Farid Touati, and Abdulaziz M Ghaleb. A comparative analysis of BLE and 6LoWPAN for U-HealthCare applications. In *GCC Conference and Exhibition (GCC), 2013 7th IEEE*, pages 286–291. IEEE, 2013.
- [114] Dieter Uckelmann, Mark Harrison, and Florian Michahelles. An architectural approach towards the future internet of things. In *Architecting the internet of things*, pages 1–24. Springer, 2011.
- [115] United Nations, Department of Economic and Social Affairs, Population Division. World population prospects: The 2015 revision, 2015.

- [116] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.
- [117] Steve Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, 10(6), 2006.
- [118] Michael Vögler, Johannes M Schleicher, Christian Inzinger, and Schahram Dustdar. A Scalable Framework for Provisioning Large-scale IoT Deployments. *ACM Transactions on Internet Technology*, 16(2):11:1–11:20, March 2016.
- [119] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr, and Steve Gallo. A comparison of virtualization technologies for hpc. In *22nd International Conference on Advanced Information Networking and Applications (aina 2008)*, pages 861–868. IEEE, 2008.
- [120] Roy Want. An introduction to rfid technology. *IEEE pervasive computing*, 5(1):25–33, 2006.
- [121] Roy Want, Bill N Schilit, and Scott Jenson. Enabling the internet of things. *IEEE Computer*, 48(1):28–35, 2015.
- [122] Andrew Whitaker, Marianne Shaw, Steven D Gribble, et al. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, Technical Report 02-02-01, University of Washington, 2002.
- [123] Joshua White and Adam Pilbeam. A survey of virtualization technologies with performance testing. *arXiv preprint arXiv:1010.3233*, 2010.
- [124] Yang Yu. *Os-level virtualization and its applications*. ProQuest, 2007.
- [125] ZeroTurnaround. Java Tools and Technologies Landscape Report 2016. <https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/> Accessed: 03/2017.
- [126] Michele Zorzi, Alexander Gluhak, Sebastian Lange, and Alessandro Bassi. From today’s intranet of things to a future internet of things: a wireless-and mobility-related view. *IEEE Wireless Communications*, 17(6):44–51, 2010.

Measurements

A.1 Notebook

	Average	Max	Min
Scenario 1a	29606.3	32600	28744
Scenario 1b	109446	160792	99018
Scenario 2a	2864.2	3187	2730
Scenario 2b	2995.1	3266	2771
Scenario 3a	5169	5703	4919
Scenario 3b	5117	5268	4934

Table A.1: Overall deployment (ms)

	Average	Max	Min
Scenario 1a	13441.4	14086	13098
Scenario 1b	61894.8	114251	51535
Scenario 2a	2424.7	2688	2330
Scenario 2b	2440	2677	2340
Scenario 3a	3084.4	3168	2972
Scenario 3b	3074	3234	2866

Table A.2: Create image (ms)

	Average	Max	Min
Scenario 1a	11479.1	12048	9947
Scenario 1b	30090.9	31555	29100
Scenario 2a	94.5	103	89
Scenario 2b	170.2	303	133
Scenario 3a	988.3	1050	931
Scenario 3b	990.7	1054	950

Table A.3: Push image (ms)

	Average	Max	Min
Scenario 1a	3932	6571	3589
Scenario 1b	16725.2	16942	16444
Scenario 2a	41.9	49	35
Scenario 2b	85.5	458	35
Scenario 3a	676.9	713	660
Scenario 3b	758.4	791	722

Table A.4: Pull image (ms)

	Average	Max	Min
Scenario 1a	92	158	64
Scenario 1b	81.8	109	64
Scenario 2a	80.1	100	62
Scenario 2b	80.3	96	66
Scenario 3a	79.1	94	62
Scenario 3b	85.8	107	75

Table A.5: Create container (ms)

A.2 Hosted Server

	Average	Max	Min
Scenario 1a	96879	99528	94519
Scenario 1b	391361.4	394467	389427
Scenario 2a	3461.1	3970	3264
Scenario 2b	3914.1	4412	3542
Scenario 3a	19886.7	20074	19273
Scenario 3b	19332.8	19645	19174

Table A.6: Overall deployment (ms)

	Average	Max	Min
Scenario 1a	13462.9	15089	13063
Scenario 1b	51609.8	53685	50077
Scenario 2a	2503.7	2992	2354
Scenario 2b	2712	2944	2522
Scenario 3a	3372.4	3478	2967
Scenario 3b	3123	3358	2923

Table A.7: Create image (ms)

	Average	Max	Min
Scenario 1a	11132.6	11765	9253
Scenario 1b	29740.7	30933	28704
Scenario 2a	114.1	146	96
Scenario 2b	180.2	225	139
Scenario 3a	1029.7	1096	956
Scenario 3b	996.5	1025	951

Table A.8: Push image (ms)

	Average	Max	Min
Scenario 1a	71388.2	71966	70863
Scenario 1b	308995.3	311716	307096
Scenario 2a	297	321	273
Scenario 2b	294.5	310	270
Scenario 3a	14894.4	15000	14757
Scenario 3b	14702.1	14792	14628

Table A.9: Pull image (ms)

	Average	Max	Min
Scenario 1a	90.1	137	65
Scenario 1b	127.7	369	69
Scenario 2a	100.7	161	78
Scenario 2b	97.2	122	72
Scenario 3a	97.1	137	68
Scenario 3b	81.8	114	71

Table A.10: Create container (ms)

A.3 Raspberry Pi

	Average	Max	Min
Scenario 1a	485861.9	529779	464612
Scenario 1b	853426.9	887962	821605
Scenario 2a	27948.4	32482	24717
Scenario 2b	26907.9	32381	23827
Scenario 3a	48462.5	53717	44569
Scenario 3b	50118.6	56770	43455

Table A.11: Overall deployment (ms)

	Average	Max	Min
Scenario 1a	212901.8	228299	198653
Scenario 1b	404557.7	418438	387992
Scenario 2a	24301.3	28999	20878
Scenario 2b	23534.3	29406	20517
Scenario 3a	28143.9	32181	25108
Scenario 3b	29183	34681	24995

Table A.12: Create image (ms)

	Average	Max	Min
Scenario 1a	149451.2	152219	146370
Scenario 1b	224107.5	229373	217764
Scenario 2a	1004	681	1570
Scenario 2b	1212.9	2489	757
Scenario 3a	11531.1	13060	9972
Scenario 3b	11819.2	13084	10136

Table A.13: Push image (ms)

	Average	Max	Min
Scenario 1a	120030.7	150743	105790
Scenario 1b	222079.1	246153	188846
Scenario 2a	142.3	586	85
Scenario 2b	89.1	106	77
Scenario 3a	6030.3	9253	4983
Scenario 3b	6729.6	12005	4998

Table A.14: Pull image (ms)

	Average	Max	Min
Scenario 1a	1100.2	5101	140
Scenario 1b	143.3	219	117
Scenario 2a	594.5	2696	174
Scenario 2b	152	181	130
Scenario 3a	783.4	4198	149
Scenario 3b	281.4	1504	114

Table A.15: Create container (ms)