Containerization of telco cloud applications

Samu Toimela

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 22.05.2017

Thesis supervisor:

Prof. Jukka Manner

Thesis advisor:

M.Sc. Tero Venetjoki

Aalto University
School of Electrical Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Samu Toimela

Title: Containerization of telco cloud applications

Date: 22.05.2017 Language: English Number of pages: 7+56

Department of Communications and Networking

Professorship: Communications Engineering

Supervisor: Prof. Jukka Manner

Advisor: M.Sc. Tero Venetjoki

Mobile service providers and manufacturers have moved towards virtualized network functions, because the amount of mobile data traffic has increased a lot during the past few years. Virtual machines offer high flexibility and easier management. They also enable flexible scaling, which makes it easier to respond to the varying traffic patterns during the day.

However, the traditional virtual machines contain overhead and have reduced performance in most of the operations. One high performing alternative to a virtual machine is a Linux container. Linux containers do not contain additional operating system or any unnecessary services. Containers are isolated user spaces which share host computer's kernel. This makes processes inside them perform almost as well as if they would be running directly on host. Also, the startup time of containers is extremely fast compared to virtual machines.

This thesis studies, if Linux containers are suitable for telco applications. The research is conducted via proof-of-concept where parts of an existing telco application are moved to containers. First, the container technology and related tools are discussed. Benefits and requirements of the Linux containers are then studied based on the proof-of-concept.

In this thesis, it was found out that containers are suitable for running small parts of the application. For example, the software update and scaling are a much more efficient processes with containers than with virtual machines. However, the isolation is weaker in containers than in virtual machines, and at the moment they are not suitable for applications or environments where strict isolation is a necessity.

Keywords: Virtualization, Docker, Container, Radio Access Network, Cloud

Tekijä: Samu Toimela

Työn nimi: Ohjelmistokonttien hyödyntäminen pilvipohjaisen mobiiliverkon sovelluksissa

Päivämäärä: 22.05.2017 Kieli: Englanti Sivumäärä: 7+56

Tietoliikenne- ja Tietoverkkotekniikan laitos

Professuuri: Tietoliikennetekniikka

Työn valvoja: Prof. Jukka Manner

Työn ohjaaja: M.Sc. Tero Venetjoki

Mobiilidatan määrä on kasvanut voimakkaasti muutaman viime vuoden aikana. Tämän johdosta mobiiliverkon palveluntarjoajat ja laitevalmistajat ovat alkaneet virtualisoimaan mobiiliverkon laitteita. Virtualisointi tarjoaa joustavuutta ja helpottaa laitteiden hallintaa. Virtualisoinnin avulla mobiiliverkon laitteita voidaan skaalata verkon liikennemäärien mukaan.
Virtuaalikoneet sisältävät ohjelmien suorituksen kannalta epäolennaisia palveluita ja niiden suorituskyky on usein heikompi verrattuna tavallisiin tietokoneisiin. Linux-kontit tarjoavat kevyemmän ja suorituskyvyltään tehokkaamman vaihtoehdon virtuaalikoneille. Ne eivät sisällä ylimääräistä käyttöjärjestelmää tai ylimääräisiä palveluita. Kontit ovat eristettyjä alueita käyttöjärjestelmän sisällä ja ne myös jakavat käyttöjärjestelmän ytimen. Tämän ansiosta prosessien suorituskyky kontin sisällä on lähes identtinen kuin ilman kontteja. Konttien käynnistymisaika on myös huomattavasti lyhyempi kuin virtuaalikoneiden.
Tässä diplomityössä tutkitaan, soveltuvatko Linux-kontit mobiiliverkon sovellusten suorittamiseen. Tutkimus suoritetaan käytännön esimerkin avulla, jossa erään mobiiliverkon sovelluksen osia suoritetaan konteissa. Aluksi tutkitaan Linux-kontteja, niiden teknologista taustaa sekä niihin liittyviä työkaluja. Tämän jälkeen konttien hyötyjä ja niiden vaatimuksia tutkitaan edellä mainitun käytännön esimerkin avulla.
Tässä työssä saatiin selville, että kontit soveltuvat pienien sovelluksen osien suorittamiseen. Esimerkiksi sovelluksen päivitys ja skaalaus on tehokkaampaa kontteja käytettäessä. Konttien eristys on kuitenkin heikompaa kuin virtuaalikoneiden ja tällä hetkellä ne eivät sovellu sovelluksille tai ympäristöihin, joissa vaaditaan vahvaa eristystä.

Avainsanat: Virtualisointi, Docker, Linux-kontti, Radioverkko

Preface

This thesis was conducted at Nokia Oyj in Espoo, Finland. I would like to thank Markku Niiranen for giving me the opportunity to work with this interesting topic.

I would like to thank professor Jukka Manner for supervising my thesis and giving me valuable feedback. I would also like to thank my thesis advisor Tero Venetjoki from Nokia for his guidance and support during the thesis.

In addition, I would like to thank all my colleagues at Nokia, especially Jan Zizka for always being interested in my topic and for providing great feedback. I am also grateful for all the other thesis workers in my team for their peer support and proofreading.

Finally, I would like to thank my family for supporting me in my studies and all my friends in Otaniemi for making the past six years unforgettable.

Espoo, 22.05.2017

Samu A. K. Toimela

Contents

Abbreviations

API	Application programming interface
AuFS	Another union filesystem
BBU	Baseband unit
BSC	Base station controller
CAPEX	Capital expenditures
cgroup	Control group
CPU	Central processing unit
C-RAN	Cloud radio access network
I/O	Input/Output
IPC	Inter-process communications
LXC	Linux containers
MME	Mobile management entity
OPEX	Operational expenses
OS	Operating system
PID	Process identifier
RAN	Radio access network
RNC	Radio network controller
RRH	Remote radio head
seccomp	Security computing mode
VM	Virtual machine
VMM	Virtual machine monitor
VNF	Virtual network function

1 Introduction

Over the past few years, mobile data traffic has increased greatly. Current forecasts are showing that mobile data traffic will continue to grow rapidly in the future as well. According to Cisco's forecast [1], mobile data traffic will increase eightfold during 2015-2020. One challenge with mobile data is that the traffic is varying by its nature. Mobile phones are mostly used during the daytime, so the mobile network must be able to respond efficiently to the alternating traffic peaks. Increased data usage and varying traffic have caused mobile network providers to change their traditional approach from fixed network devices into more flexible ones.

As a solution, mobile network providers and network device manufacturers have begun to move towards virtualized, software-based architecture instead of traditional dedicated hardware. With virtualized software, new instances can be easily launched whenever they are needed to support high traffic peaks. However, with continuously increasing mobile data usage, even the traditional virtualization seems to be lacking effectiveness that is needed. One of the reasons is that virtualization of a software usually comes with significant overhead. With traditional virtualization techniques, an operating system needs to be virtualized first and software can be then added on top of it. When multiple virtualized instances are running on the same hardware, overhead caused by multiple operating systems will significantly waste computing resources and thus reduce the performance.

Linux containers, often referred simply as containers, offer a solution for eliminating overhead caused by traditional virtualization. Container is an isolated area in host operating system's kernel, that is created using kernel namespaces and control groups. Containers are like lightweight virtual machines, but they do not have an operating system running inside them. Instead, containers utilize host operating system's kernel to offer basic operating system services to processes inside them. From application's perspective a container acts similarly to a traditional virtual machine, but it performs faster. By eliminating the additional operating system, the overhead of virtualization becomes so minimal that the performance is as good as native.

Containers offer other advantages as well. They will for example make the software distribution and management easier for the mobile network providers. By running applications in containers, software manufacturers can offer direct updates to the specific parts of the software instead of delivering a new version of the whole application. Traditionally the unit of delivery has been a large virtual machine image, which contains an additional operating system and all the services and binaries required by the applications. Updating a distributed application with new virtual machine image will require recreation of all the virtual machines. In contrast, software updates for containerized applications do not usually require restarting of the whole machine. Containers can be recreated using the updated image without restarting other services.

1.1 Problem statement

As mentioned in the previous section, increased mobile data traffic and varying traffic patterns require flexibility and efficient computing resource usage. Virtual machines contain unnecessary overhead and suffer from reduced performance in most of the operations. Also, the scaling and the software update of virtual machine based application is a slow operation, because the startup time of a virtual machine can be even several minutes [2]. Linux containers could be one potential solution for improving the performance and flexibility of telco applications.

This thesis aims to find out what are the key advantages and disadvantages of using containers in radio cloud infrastructure. This thesis also aims to find out what kind of mechanisms are needed to operate and manage containers efficiently and what kind of services are required from the platform. Because containers are introduced to an existing application, other aim is also to find out what kind of architectural changes are needed to the current solution. Finally, the thesis also aims to provide general guidelines that should be followed when developing distributed applications that are running in containers.

1.2 Scope and Methodology

This thesis is only focusing on telco applications, which normally consist of multiple instances and are traditionally delivered as virtual machine images. As almost every telco application is based on Linux, this thesis is only focusing on Linux-based solutions. Solutions based on Windows or other operating systems are not discussed in this thesis. Also, other lightweight virtualization technologies, such as unikernels, are not discussed in a scope of this thesis.

Container technology and related tools are first researched by doing literature review. Second part of the research is done by a proof-of-concept, which includes practical implementation of one telco application that utilizes containers. Advantages and requirements of containers are then studied based on the practical implementation.

1.3 Results

In this thesis, it was found out that containers are suitable for running small parts of the application. For example, the software update and scaling are much more efficient processes with containers than with virtual machines, mainly because container images are very light and they start up quickly. However, the isolation is weaker in containers than in virtual machines, and at the moment they are not suitable for applications or environments where strict isolation is a necessity. Also, the process of moving existing monolithic applications into containers requires a lot of work and it should be carefully considered if the gained benefits overcome the costs of the transformation.

1.4 Structure of the Thesis

In chapter 2, radio access network and its evolution are discussed. In the same chapter, basic requirements for radio access networks are presented. Chapter 3 focuses on virtualization. First, two main virtualization methods are introduced. Main focus of the chapter is on operating system-level virtualization and containers. Introduction of different container managers is also included in this chapter. Chapter 4 discusses distributed container-based applications, container orchestration and microservice-based architecture. Chapter 5 talks about practical implementation and requirements of container based application. In chapter 6, the evaluation is carried out based on the practical work. In the same chapter, basic containerization guidelines are also presented and future work is discussed. Chapter 7 is a summary of the thesis.

2 Mobile network infrastructure

Mobile networks are a necessity in the modern world where everything is connected. Because the number of devices and amount of mobile data traffic has increased, the mobile networks have had to evolve. This chapter presents the structure of radio access network and discusses its evolution. The requirements and properties of radio access networks are also presented.

2.1 Radio access network

Radio access network (RAN) consist of multiple mobile technologies such as 1G, 2G, 3G and 4G. Main components of radio access networks are base stations and their controllers. Base stations offer signal coverage for users in specific area, called cell. Radio access network is connected to a mobile core network, which in turn is connected to internet and landline network.

In 1G and 2G, a radio and a baseband unit (BBU) are combined into a base station, which serves one cell [3]. Base stations are then connected to a base station controller (BSC). However, this leads to high power consumption because every cell does the processing separately on the individual sites. Also, the computing resources might be wasted at the times when the cell is under lower utilization. Illustration of this base station architecture can be seen in Figure 1(a).

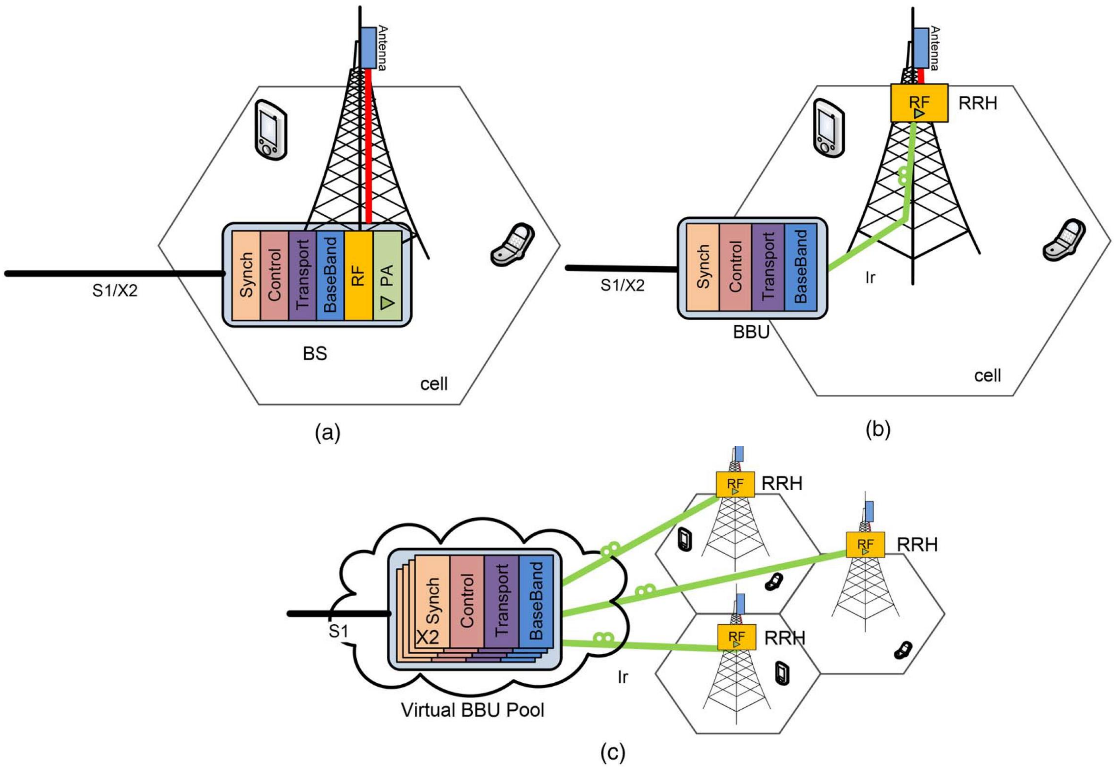

Figure 1: Base station evolution [3]

3G made the BBU management easier as it separated BBU and remote radio head (RRH). This is shown in Figure 1(b). Remote radio head in the base station takes care of the analog/digital conversions, and for example amplification and filtering. BBU can be moved away from the base stations into a more accessible location, so that it can be managed more efficiently. Distance between BBU and the base station can be a maximum of 40km, after which the processing and propagation delay becomes too high [3]. Also, the power consumption can be optimized with a separated BBU. However, occasional under-utilization of the computing resources is still a problem in this architecture.

In 4G, One BBU can control up to three RRHs [4]. On top of that, BBUs are clustered into centralized locations called BBU pools. These centralized BBU pools offer savings in operational expenses (OPEX), because previously each site had separate rental and air conditioning expenses. Running BBUs in a centralized data center also makes operation and maintenance tasks much easier, and costs of operating multiple base station sites are thus reduced.

BBU efficiency and the computing resource utilization can be improved by virtualizing the BBUs. This architecture is presented in Figure 1(c). The virtualized BBU pool can be serving multiple cells dynamically. Multiple virtualized BBUs can be run on a same hardware and that offers savings in capital expenditures (CAPEX). The architecture, where virtualized BBU pools do not require specific infrastructure is called Cloud Radio Access Network (C-RAN or cloud-RAN) [3]. Standardization of cloud-RAN is still underway.

In addition to the base stations and the BBUs, radio access network also includes base station controllers. In 2G, a single base station controller (BSC) can serve multiple base stations. BSC allocates radio frequency channels for the base stations and controls handovers between them. In 3G, same kind of base station controller is called radio network controller (RNC). RNC handles for example call processing, mobility management and handover-control. However, in 4G, the base station controller is included in the base station called eNodeB. Mobility management and handover control is then done between the eNodeBs or by using mobile management entity (MME) in mobile core network. The controllers and other mobile network elements have also been virtualized into virtual network functions (VNF) to gain OPEX and CAPEX savings from centralized management and more efficient resource usage.

2.2 Radio access network requirements

Radio access network has multiple differences compared to the other networks such as the internet. Because of the nature of the mobile traffic, radio access network has strict requirements, for example for low latencies and high availability. They also need to be able to scale according to the current load.

2.2.1 Latency

Telecommunication services must offer minimal latencies. Originally, traffic in mobile networks was almost entirely voice traffic. Latencies in voice calls have to be kept consistently low, because the voice conversations would not be natural with high delays. Machine-to-machine communications also benefit from lower latencies, as machines can process the information a lot faster than humans. Lower latencies are required for precise automation and communication. In the future, very low latencies and high reliability are also requirements for virtual reality, health-care systems and even for self-driving cars [5]. According to GSA's 5G specification [6], 5G should have latencies under 1 millisecond.

To provide lower latencies, operators must deploy cells closer to the customers. Also the content, such as videos and search engine caches can be aggregated closer to customers. This does not only decrease latency, but also makes the general network performance better, because large amounts of content are not transferred through the whole network.

2.2.2 Scaling

Telecommunication traffic varies greatly between the cells during a day. Base stations near office buildings are gathering huge traffic peaks during the office hours, while base stations outside the city are at the same time on low utilization and are thus wasting computational resources [7]. Figure 2 shows the mobile network load during a day. The combined traffic amount is also a lot higher during the business hours than during the off-the-peak hours. Also, the high traffic variation in different cells can be for example caused by public events and spontaneous gatherings.

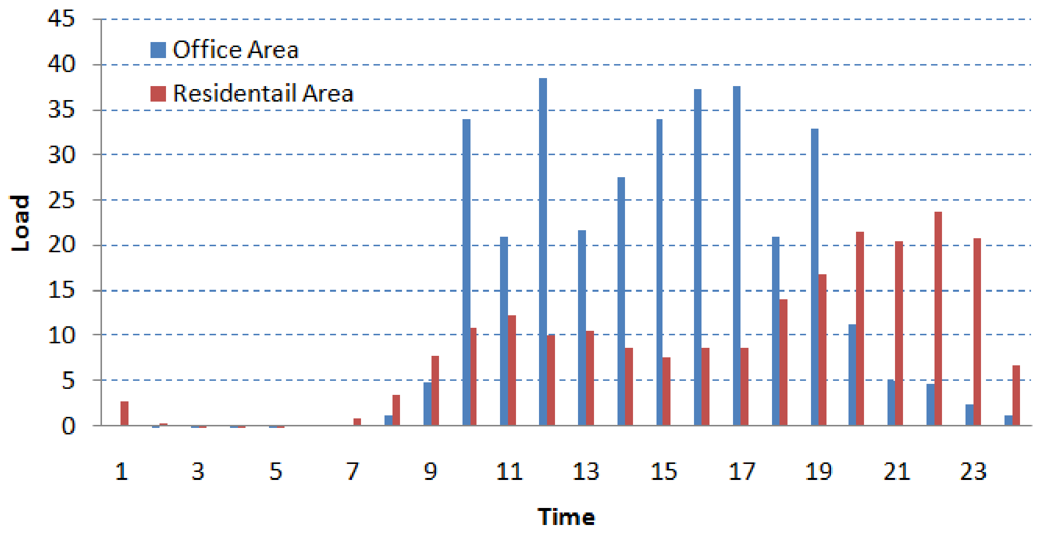

Figure 2: Mobile network load during a day [7]

Radio access network must be able to answer these varying traffic patterns by allocating resources according to the cell utilization. Clustered and virtualized RAN makes it easy to do this. The virtualized parts of RAN can be easily scaled out in a case of traffic peaks. Similarly, the elements in the cells with lower utilization can be scaled in. Scaling of traditionally virtualized network elements is sufficient when traffic pattern is known beforehand, for example in a case of the normal work related traffic variation. However, to respond to a spontaneous traffic growth in certain cells, scaling of the virtualized elements can be too slow as it can take several minutes [2]. More effective scaling methods need to be researched if optimal service levels are needed to be offered at all times. One potential solution is a Linux container (see chapter 3).

2.2.3 High availability

Telecommunication networks have to be operating at extremely high availability. For example, some telecommunication elements are providing 99,999% availability. The availability must be guaranteed in a case of element failures, as well as in normal handover situations when user moves to another cell.

This high availability can be achieved by using redundant computers and connections. Multiple redundancy schemes exist, but most common ones are N+M redundancy and 2N redundancy. N+M redundancy means that there is N computers serving the users, and M computers waiting in case of failure. The additional computers are in a cold standby state, but they can be used as hot standby for example during software updates. 2N redundancy means that there are N computers serving the users, and a second set of computers waiting in hot standby. The internal state of these computer pairs is kept synchronized, and computers can be failing without it affecting the ongoing traffic. Redundancy can be increased by distributing these replication sets on different sites. This is called geo-redundancy, and it is used to guarantee availability even if one site gets disconnected.

2.3 Summary

Radio access network consists of base stations and their controllers. The evolution of the radio access network technologies has led to a separation of remote radio heads and baseband units. Because of this, BBUs can be clustered into centralized locations to make their management easier. The clustering has also enabled a possibility to virtualize BBUs to improve computing resource utilization.

Radio access networks have very tight requirements for example for latency and high availability. Also, because of the varying traffic patterns of the mobile networks, the network should be scalable. This means that computing resources can be allocated to each cell based on their utilization.

3 Containers and virtualization

Virtualization technologies have been around for over 50 years [8]. The term "virtualization" was first used by IBM when they researched efficient time sharing methods for hardware resource usage. Virtualization became popular again around year 2000, when multiple solutions for x86 virtualization started to emerge. Main motivation for virtualization was a need for running multiple independent instances with their own operating systems (OS) and libraries on one physical hardware. The term virtual machine (VM) refers to a software based computer that acts mostly like a physical computer. VM runs an operating system and applications on top of it. The operating system inside the virtual machine is called a guest operating system (guest OS). Operating system that is running directly on top of hardware is called a host operating system (host OS).

Major advantage of virtual computing comes from the reduced costs that can be achieved by running multiple virtual instances on the same hardware, so that the hardware resource utilization can be improved. This offers direct cost savings, because fewer physical computers are required. Cost savings can be increased with more efficient virtualization, for example by running virtual machines on a server infrastructure that is specifically optimized for them [9]. For example, multiple infrastructure-as-a-service providers have large and heavily optimized server farms where customers can run their own virtual machines.

Other advantage is the isolation that virtualization offers. Applications might require different versions of their dependencies and libraries. Without virtualization, these conflicting applications cannot be run on the same host and would require a dedicated host computer. This might lead to the wasting of computational resources, because the application might only use a fraction of its dedicated hardware resources. By using virtualization, the application with its dependencies and libraries is running in an isolated space.

Most common virtualization method is called hypervisor-based virtualization. Hypervisor is a piece of software (or combination of software and hardware) that runs and manages virtual machines. Hypervisor can launch multiple virtual machines on a single host. Sensitive system instructions, such as input/output (I/O) system-calls, are trapped and translated by the hypervisor. [10]

One alternative for hypervisor-based virtualization is called operating system-level virtualization. These kinds of virtual instances do not include a guest operating system, unlike hypervisor-based virtual machines. Operating system-level virtualization uses Linux control groups and namespaces to create isolated spaces in host operating system. These isolated spaces are called containers. Processes inside the containers cannot see other processes outside them. They can also have a dedicated directory tree. From application perspective, container does not differ from traditional hypervisor-based virtual machine.

This chapter focuses on virtualization and is mostly discussing operating system-level virtualization and containers. First, hypervisor-based virtualization and its main properties are explained. Then, operating system-level virtualization is discussed and underlying technologies enabling it are presented. After that, a comparison of hypervisor-based virtualization and operating system-based virtualization is discussed. Finally, a few popular container managers are introduced.

3.1 Hypervisor-based virtualization

Hypervisor allows multiple virtual machines to run simultaneously on a same host. It offers a platform where virtual machines can run isolated from each other and the host. This kind of virtualization is called hypervisor-based virtualization [11]. Hypervisor-based virtualization is the most popular virtualization technique.

Hypervisor, also called virtual machine monitor (VMM), launches, manages and terminates virtual machines. Hypervisor offers an illusion of dedicated native hardware to the virtual machines. It is done by trapping a variety of sensitive instructions between the virtual machine and the host hardware. Instructions that would access host's resources, such as the physical disks, are considered sensitive [10]. These instructions are trapped and translated, because they could otherwise break the illusion of a guest OS not being the only one in the system. This kind of virtualization is called full virtualization. However, I/O performance of full virtualization is reduced because of the trapping and translating of the I/O calls.

Para-virtualization is a technique where the virtual computer is aware that it is running in a virtualized environment [12]. By using para-virtualization, there is no need for trapping, because both host and guest operating system can cooperate. However, para-virtualization requires a modified guest operating system.

There are two types of hypervisor-based virtualization, depending on where

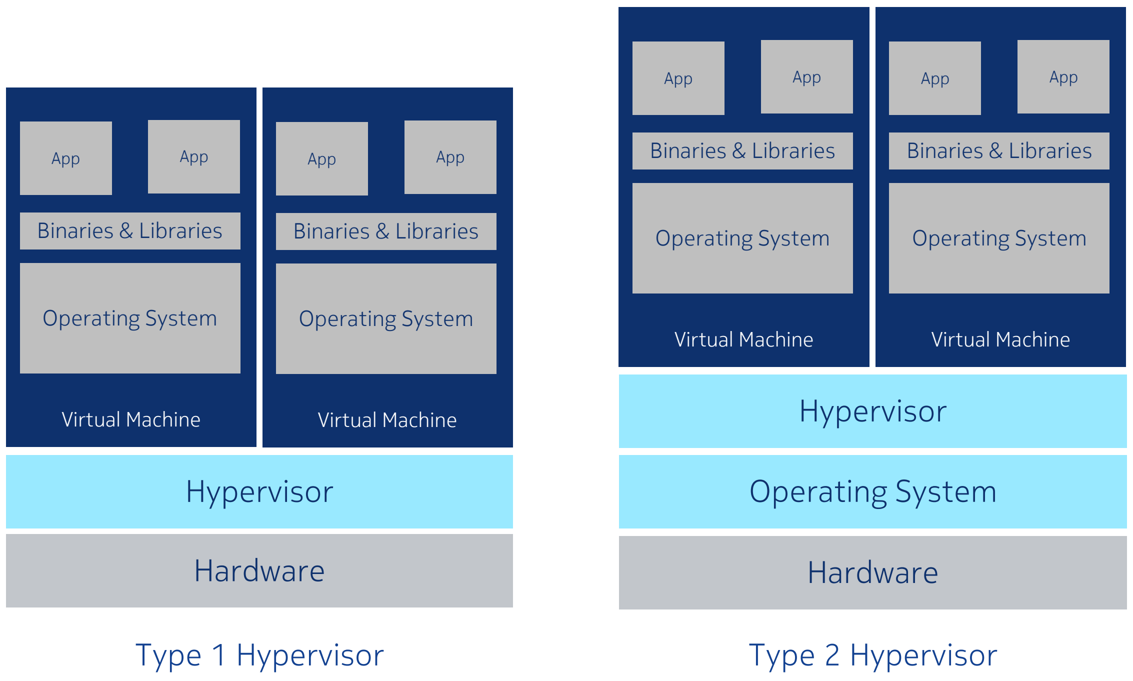

Figure 3: Comparison of the hypervisor types

the hypervisor is running [13]. These types are illustrated in Figure 3. A native hypervisor (type 1) is running directly on top of hardware, while a hosted hypervisor (type 2) is running on the host operating system.

Type 1 hypervisor consists of a small set of software that is required for virtualization. The small set of software is used to manage resources and access to I/O-devices between virtual machines and the hardware. Type 1 hypervisor provides better performance, security and availability than type 2 hypervisor [14]. Notable type 1 hypervisors are Xen [15], KVM [16] and VMWare ESX [17].

Type 2 hypervisor utilizes underlying host OS for its functions. It is often used on systems, that require support for variety of I/O devices. Examples of type 2 hypervisors are QEMU [18], VMWare Workstation [19] and Oracle VM Virtualbox [20].

Hypervisor-based virtualization also offers good isolation and security for virtual machines. Virtual machines are not aware of each other and they can access host only through the hypervisor [21]. Hypervisor can also emulate multiple different processor architectures, such as x86 and ARM. Because of this, virtual machines can run almost any operating system regardless of the host's hardware and operating system.

3.2 Operating system-level virtualization

An alternative to traditional hypervisor-based virtualization is operating system-level virtualization. Operating system-level virtualization has been around for many years already. Chroot [22], developed in 1982, is considered as the first operating system-level virtualization tool. Other similar tools, such as Linux-Vserver [23] and OpenVZ [24], have emerged after chroot. Also, a few Unix-based operating system-level virtualization tools, such as FreeBSD jails [25] and Solaris Zones [26] exist. However, their developers have not been active on integrating their tools in the Linux mainstream kernel.

Virtual instances that are created with operating system-level virtualization are often called containers. Difference between a hypervisor-based virtual machine and a container is that the container does not run a guest operating system inside it [27]. Instead, containers share parts of the host's kernel to provide an operating system-like functionality for the applications. Container gives the processes an illusion of running in a separate machine, even though in reality the process is just running in an isolated area in the host operating system, sharing the host's resources. Processes inside the containers can also run native system calls without any translating in between. However, there are access-control checks for the system-calls to enhance the security of containers [28]. Some system calls that access the kernel can be harmful and are prevented by default.

Isolation of container-based virtualization is achieved by using Linux namespaces. Namespaces can be used for example to control which part of host's file system container is allowed to see, and which processes are visible in the container's process tree. Resource management of containers is handled by Linux control groups. Control groups can be used to limit container's access to the host's hardware resources, such as CPU, memory and network.

Containers can run isolated system services, such as init, sshd, syslogd and cron. These kinds of containers are called system containers [29]. System containers are usually built with a container image that contains multiple tools and libraries. System containers are in many way similar to the traditional virtual machines. Containers can be also used to run just a single application. These containers are called application containers. Application containers are often considered as micro containers, which means that the container image only contains application binaries, libraries and other dependencies that are required by the application.

During the past few years, containers have started to gain more popularity. Developers and system administrators have just now started to realize the advantages that containers offer compared to the hypervisor-based virtualization. Container technologies are widely used and developed by the community, but they still are not as mature as the virtual machines. For example, the security of containers is still a big issue [30].

3.2.1 Chroot

Chroot (Change root) [22] was one of the first implementation in Unix kernel to allow processes to run in isolated space inside the host operating system. This isolated space is called chroot jail. Chroot was introduced in Version 7 Unix in 1979.

Chroot changes root directory for a process, so that the process cannot access or see files outside this new directory tree. With chroot, it is also possible to allow root access in an isolated space. Chroot jail is mainly used as an isolated environment where kernel development can be made without it accidentally affecting the host kernel. Chroot has also been used to create a secure and isolated space where malicious software can be safely investigated.

However, chroot has some security vulnerabilities. For example, it is relatively easy to break out of the chroot jail if a process inside it has root privileges [31]. Other applications such as FreeBSD jails, Solaris zones and LXC (Linux Containers) [32] have adopted chroot functionality and improved the security by using for example namespaces.

3.2.2 Namespaces

Namespaces are used to create multiple isolated user-spaces in a host kernel. Like chroot, namespaces allow the processes to see an arbitrary directory as their root directory. Processes can work with root privileges inside the namespace, without endangering the host system outside it. However, unlike chroot, namespaces expand similar isolation to the other parts and functions of the kernel as well. The first namespace (mount namespace) was introduced in Linux kernel version 2.4.19 in 2002 [33]. Currently, Linux has namespaces for process identifiers, inter-process communications, networking, hostname, mount points, user identifiers and cgroups [34].

Traditionally, Linux system can only maintain a single process tree. Process tree consists of parent and child processes. Different processes can have different privileges,

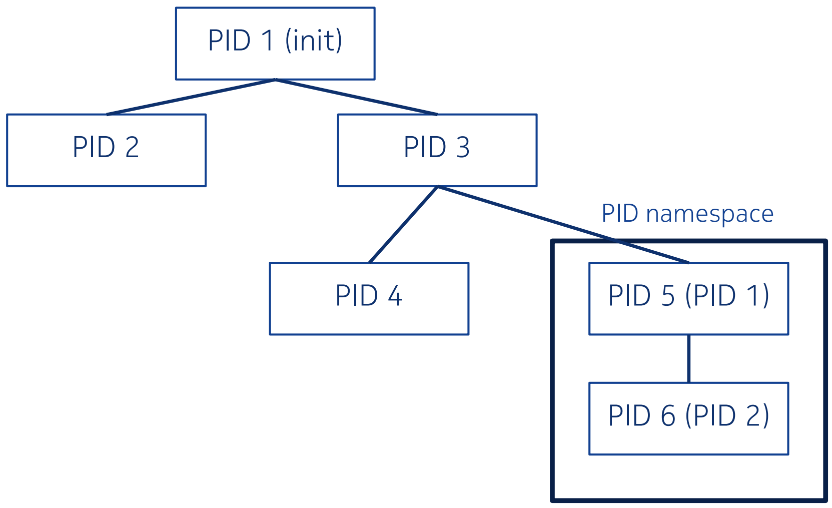

Figure 4: A process tree with a PID namespace

and they can also inspect other processes of the process tree. Each process has an unique process identifier (PID). When a Linux system starts, the first process gets PID 1. This process is considered as an init process of the system. The init process starts and maintains all the other necessary daemons and services [35]. Common init services in Linux are for example systemd [36] and upstart [37].

Process identifier namespaces (PID-namespaces) are used to virtualize process trees. Example of a PID-namespace is illustrated in Figure 4. Processes inside the PID-namespace are isolated from the host processes as well as from the processes in different PID-namespaces. PID-namespaces are hierarchical, which means that processes are also able to see all the processes inside its children namespaces. For example, the host can see the processes of every namespace in its original process tree. Processes inside the PID-namespace have isolated process identifier number space, and they can have their own init-process as PID 1. This means that processes can have multiple PIDs assigned to them, one for the host process tree and one that is visible inside the namespace.

Processes running inside a namespace might need to communicate with other processes to function correctly. Inter-process communication namespaces (IPC-namespaces) allow communications only between the processes belonging to the same IPC-namespace. This prevents the processes from interfering with other processes in other namespaces.

By using networking namespaces, different processes can see different networking interfaces. Network namespaces can provide new network interfaces and virtual addresses for its processes to see. Host processes and the processes in the other namespaces, can communicate with each other using these virtual interfaces, just like they would communicate with external hosts.

Mount namespaces are useful as it is possible to determine which mount points can be seen inside the namespaces. A process can initially see the same mount points as the host. However, with mount namespaces, processes can mount or unmount endpoints without it affecting the host's mount points.

User namespaces are used to run the processes with a different user ID or group ID. The user namespace also allows the process to have a root privileges in the new namespace without it having the root privileges outside the namespace.

3.2.3 Control Groups

Control groups (cgroups) [38] can manage the hardware resource usage, such as CPU, memory, disk and network, of specific process-groups. This is done by assigning a set of processes into hierarchical groups that have specific rules for the behaviour, for example to limit how much CPU they can consume.

Cgroup project was started by two Google engineers, Paul Menage and Rohit Seth, in 2006. Initially cgroups were named process containers [39]. In 2007, the functionality was merged to the Linux kernel and the name was changed to control groups. However, some time after that, the Linux community thought that concept of cgroups should be reworked. Tejun Heo took over the development of cgroups and an improved version, cgroups v2, was finally merged into the Linux kernel version 4.5 in 2016 [40].

With cgroups, it is possible to ensure that certain process groups are not able to consume too much computing resources such as CPU time. Cgroups can be also used to guarantee that a specific process group is able to get enough computing resources. Cgroups become really useful when they are combined with namespaces in a form of containers. When cgroups are combined with namespaces, it is possible to determine how much computing resources each container is allowed to use [27]. This brings operating system-level virtualization closer to hypervisor-based virtualization, where computing resource allocations can be assigned to separate virtual machines. Cgroups also provide metrics for the computing resource usage. These metrics can be also used for other purposes, such as billing.

3.3 Comparison of container and hypervisor-based virtualization

From the application perspective, both hypervisor-based virtual machines and containers act similarly [41]. Containers can be rebooted and they can have for example their own user accounts, processes and file system. However, there are actually multiple differences that affect the performance and the isolation. Figure 5 illustrates main architectural differences between hypervisor-based virtualization and operating system-level virtualization.

Each hypervisor-based virtual machine contains an operating system and all the binaries and libraries that are needed for running the operating system and the applications. This causes a lot of overhead and leads to the wasting of computing resources, especially when multiple virtual machines are run on the same host. In contrast, containers are isolated areas in the host operating system and they do not include an additional guest operating system inside them. Instead, to enable

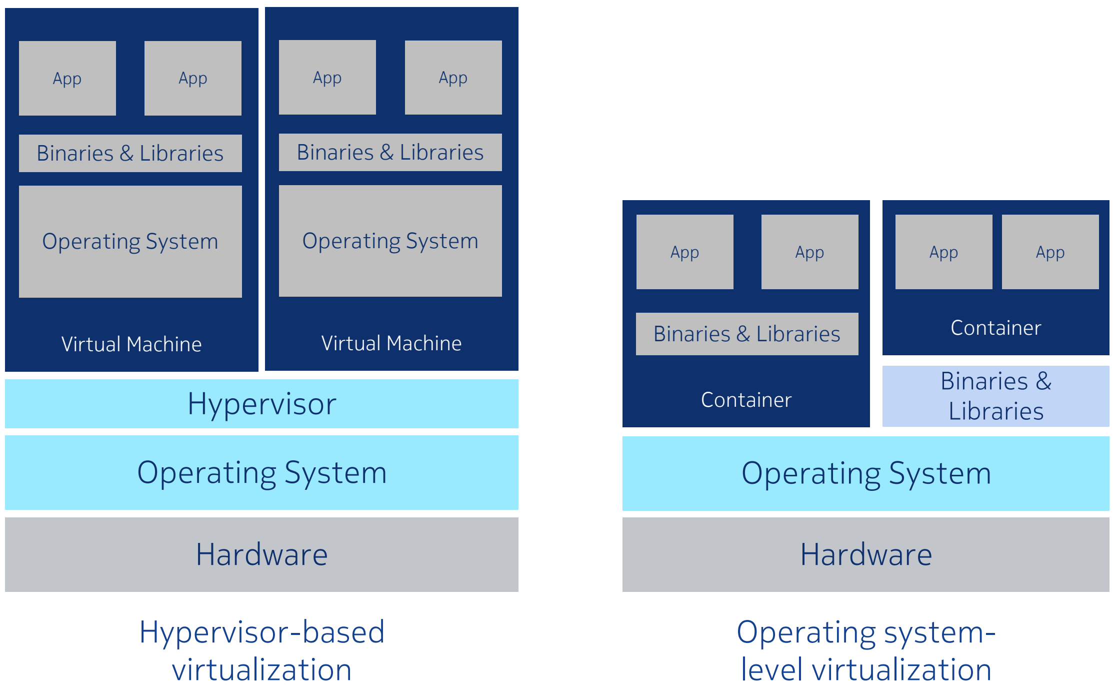

Figure 5: Comparison of hypervisor-based virtualization and operating system-level virtualization

operating system-like functionality, they share parts of the host kernel as well as the host's libraries and binaries where appropriate [42]. Containers can also include application specific binaries and libraries.

In hypervisor-based virtualization, the computing resource control is applied to the virtual machines by a hypervisor. The hypervisor controls how many CPUs and how much memory is allocated to a certain virtual machine. Virtual machines are unaware of the underlying computing resources and can only see the resources, such as the amount of CPUs and the memory, which are allocated to them. In operating system-level virtualization, the resource control is implemented by using cgroups. Cgroups can limit the computing resources that a process group is able to use, but the processes inside the containers are still able to see the full amounts of memory and CPU [43]. This might lead to some problems with containerized applications, that are not aware of cgroup restrictions and are assuming that all the visible resources are available for use.

Because containers are sharing the host's kernel, it is not possible to run container images of different operating systems. For example, it is not possible to run Windows-based container image on a Linux host. However, hypervisor-based virtual machines can run almost any operating system inside them.

Containers are more suitable for the applications, that require a good performance and faster scaling. Containers are also beneficial for the applications, that require faster startup times and more flexible deployment options. In contrast, virtual machines are more suitable for applications that require stronger security and isolation.

Virtual machines are also the only option for running multiple different operating systems. Virtual machines are also often used in environments where different users are sharing the same infrastructure, for example in public clouds [44].

Container images are smaller and lighter than traditional VM-images. Container images do not contain an operating system. On top of that, some binaries and libraries can be shared directly from the host computer and do not have to be included in the container image. Also, the structure of a container image is layered and modifications can be downloaded just by fetching the missing layers. Old layers can be mounted for example from other images.

3.3.1 Performance

An operating system, that is running inside a hypervisor-based virtual machine, requires additional computing resources, which negatively affects the overall performance of the host system. Adding multiple virtual machines inside the same host decreases the host's performance even more [45].

Reduced performance of hypervisor-based virtualization is not only caused by the overhead from the additional operating system. Hypervisor is also trapping and translating all the sensitive instructions from the virtual machine. This will require additional computing and cause additional latency.

In operating system-level virtualization, the isolation of the user space is achieved with two lightweight tools, cgroups and namespaces. System calls are not translated between a container and the host kernel. There is no overhead if multiple containers are run on same node, because only the application processes inside a container use the computing resources. Containers are also fast and easy to create and destroy. Their startup time is fast because unnecessary services are not launched. Shutting down a container merely terminates the processes that are running inside it.

Resource utilization of containers is effective. If a container is not running anything, no resources are used. Namespaces and cgroups are just building a frame. This is different in hypervisor-based virtualization, where a guest OS, that is running in an otherwise idle VM, still uses computing resources.

According to Xavier's study of a virtualization performance [27], hypervisor-based virtualization suffers 4.3% reduction in CPU-intensive processes. Memory usage overhead was 31%. In contrast, operating system-level virtualization did not affect on CPU or memory overhead and the processes practically ran with equal performance as without virtualization.

According to the same study, read and write speeds of the disk were also worse on hypervisor-based virtualization (65% read and 50% write). In operating system-level virtualization, disk operations were not practically affected by the virtualization. However, depending on the chosen container manager, some I/O performance overhead might occur in operating system-level virtualization as well [46].

Also, the network performance of hypervisor-based virtualization was significantly worse compared to native. Average bandwidth was 41% worse than on native hardware. Network performance of operating system-level virtualization was almost equal to bare metal performance.

3.3.2 Isolation

A definition of process isolation is that one workload executing on the system cannot interfere with other workloads executing on the same system. In other words, process isolation means that one process group cannot learn anything or affect other processes by any means [29].

Hypervisor-based virtual machines are more secure than operating system-level virtual instances [47]. In hypervisor-based virtualization, a virtual machine does not have direct access to host kernel, but everything goes via the hypervisor. In operating system-level virtualization, the host kernel is shared between all containers in the system. System-calls from containers to the host kernel are also run natively without any translating, unlike in hypervisor-based virtualization. This can cause serious security issues for containers. Because of this, many container managers use a Linux kernel feature called security computing mode (seccomp) [28]. Seccomp can be used to block all the risky system calls, which could harm the host system.

The isolation of containers can be improved by not sharing anything extra from the host. Inter-container communications can be also disabled when launching containers, in order to isolate them from each other. However, the kernel is still shared which might cause some security hazards. If a malicious user is somehow able to break out from a container, for example by utilizing some kernel exploit, he will then have access to the host and all the other containers as well. Also, if an application inside a container causes a kernel panic, the whole host will go down with all the containers on it.

3.3.3 Failure handling

Failure handling means detecting and correcting possible failures in the software. In virtual machines, a supervising process, such as systemd, is able to detect the failures and restart the required processes. A similar approach can be also used with containers by including a supervising process, for example supervisord [48], in the container. It will similarly handle the failure detection and the restarting of other processes inside the container.

However, containers do not necessary include supervising process. In these situations, only the failures in the initial process of the container can be detected. When the initial process of a container dies, the container is automatically killed. The container manager can then detect the terminated containers and relaunch them.

Other part of the failure handling is notifying user about the failures, for example by logging. On hypervisor-based virtualization, the logging services are implemented inside a virtual machine. Logging services, such as syslog [49], can be easily run inside a virtual machine, same way as they would be running in a host. All the logs are stored inside a virtual machine by default. This approach is suitable for hypervisor-based virtualization, because virtual machines are able to store data, such as logs, that persist over the machine reboot.

In contrast, containers are not usually rebooted even though that is possible. Normally the containers are killed and new containers are launched right when they are needed. Data inside a container does not persist when the container is killed. This means that containers are not suitable for storing persistent data. Because of this, the traditional logging approach is not suitable for containers. With containers, it is possible to redirect logs from the containers to a host computer's persistent storage. It is also possible to stream the logs into the standard output of the container, and have another component in the system that collects and stores them appropriately.

3.4 Container managers

Creating an isolated user space using operating system-level virtualization is a very complex process and it includes multiple steps, such as setting up the namespaces and the cgroups for the specific process groups. Container managers, such as LXC, systemd-nspawn and Docker have been developed to simplify this process. They offer tools for basic container operations, such as for creating and deleting containers.

3.4.1 LXC

LXC [32] is considered to be the first real containerization method that was included in the mainstream Linux kernel. There have also been other applications that take advantage of operating system-level virtualization, for example Linux-Vserver and OpenVZ. However, these applications have not been active on integrating their solutions to the mainstream Linux kernel.

LXC uses the same kind of file system isolation as chroot. However, LXC uses Linux namespaces to expand this isolation, for example to the processes and networks. LXC also uses control groups for resource management and can utilize multiple security features, such as seccomp policices and kernel capabilities, that are offered by the Linux kernel [32].

LXC offers logically complete isolation for the processes running in containers. Containers are isolated from the host, as well as from other containers. LXC containers share the underlying host kernel so an additional operating system is not required inside the containers. By using LXC, it is also possible to share host's devices and files with the containers.

LXC offers a set of tools that can be used to manage containers. These include for example tools for creating, destroying, starting and stopping containers. LXC also supports taking of snapshots, cloning and monitoring of the containers. However, LXC is quite low-level technology and not so user friendly as other container managers. Other container managers such as Docker have been previously using LXC as their container driver. LXC does not have as big community support as for example Docker. Also, the support for LXC is mostly focused on Ubuntu.

3.4.2 Systemd-nspawn

Systemd-nspawn [50] is a tool that can be used to create lightweight namespace containers. Systemd-nspawn isolates the file system hierarchy, same way as chroot does. However, the improvement over chroot is that with systemd-nspawn, the isolation is stronger and it is expanded for example to the process tree.

One of the biggest advantages of systemd-nspawn is that it is simple and it natively supports multiple processes in a single container. Systemd-nspawn comes with the native support for systemd inside a container, and because of that the systemd-nspawn containers are even capable of running whole operating systems. However, this is not recommended as it contradicts the generally accepted containerization principles. Other advantages of systemd-nspawn are that it uses simple command-line interface commands, and that it does not require any additional daemons if systemd already exists in the system. As the security might be an issue when using containers, systemd-nspawn has also taken some actions to eliminate some of the common security issues. Systemd-nspawn for example disables writing on some of the inner interfaces, such as /sys, /proc/sys and /sys/fs/selinux [50].

However, systemd-nspawn is only providing the isolation for the containers. Resource control, for example by using cgroups, is not supported. Systemd-nspawn is also difficult to manage on a bigger scale, because it does not have a good application programming interface (API).

3.4.3 Docker

Docker is one of the most popular container managers and it has a large community support. According to a ClusterHQ's survey [51], over 92% of the respondents are using or planning to use Docker to run their containers. One big success factor of Docker has been its simple container management and a public container image hub, where the users can easily download and upload container images. Docker provides a simple toolset and API for managing the kernel-level technologies that enable containers. [45]

Docker has previously used LXC as its main container driver. It is still possible to use LXC when launching containers, but nowadays Docker uses libcontainer by default. Libcontainer for example supports wider range of isolation technologies. Docker uses Libvirt as its virtualization API.

Similar to the other container technologies, a guest OS is not needed, because Docker containers share the underlying host kernel. Docker utilizes a full set of Linux namespaces and cgroups, so it can for example mount host's devices and create separate network devices for the containers.

Docker containers are started by using so called entrypoint. A container entrypoint is a binary or a script, that is executed when a container starts. If an entrypoint process dies, the container is also killed. Because of this, Docker containers are best suited for running only a single process, unlike for example systemd-nspawn containers. However, it is also possible to run multiple processes in a single Docker container. To manage multiple processes in a single container, a supervising process, such as supervisord, must be used as an entrypoint [48, 52]. Supervisor will then launch other processes and handle for example their failures and restarts.

Docker has multiple storage-driver options such as AuFS, OverlayFS and Device Mapper [53]. Each option supports copy-on-write model for modifying the filesystem. AuFS (Advanced Multi-layered Unification File System, originally Another Union File System) was the first storage driver in Docker. AuFS layers multiple file systems

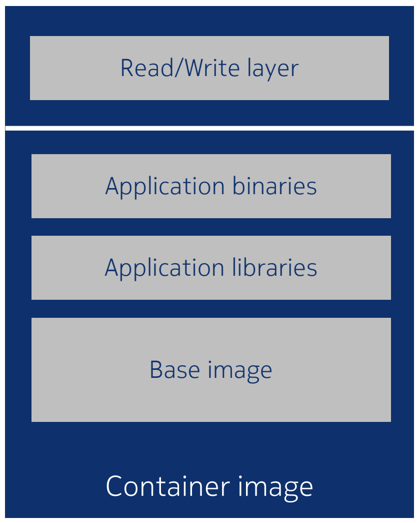

Figure 6: An example of a container image

on top of each other and provides an unified view for the new file system. Docker uses AuFS for its container images in order to make them more flexible and easier to modify. OverlayFS is another option for a Docker storage driver. It is similar to AuFS, because of its layered design. However, OverlayFS is simpler, and it exists in the Linux mainline kernel, unlike AuFS. When Device Mapper is used as a storage driver, the file system consists of multiple small virtual block devices, that each contain a part of the whole file system. Device Mapper also supports copy-on-write, but unlike in AuFS and OverlayFS, where the copy-on-write operations are executed in a file-level, Device Mapper operates in a block-level and all the operations are executed for the blocks instead [53].

Each Docker image consists of layers, regardless of the storage-driver that is used [54]. Figure 6 illustrates the structure of a Docker image. Each Docker image has a base image, and the modifications can be committed as layers on top of it. On top of the image layers, there is a read and write layer, where the local modifications are stored. The changes in the read and write layer can be saved to the image just by saving the layer. By layering multiple changes, images can be indefinitely customized while the information on previous changes is still stored. The layers can be also removed to roll back the changes. The benefit of this kind of structure is the re-usability of the layers. Multiple different container images can use the same underlying layers, and thus they only need to be stored once on the disk. Also, when downloading a new image, only the layers that do not already exist on the system are downloaded. This offers a great advantage compared to the hypervisor-based virtualization, where the VM-images are handled as a whole, and each version of the image contains everything from the OS to the application services.

Modified container images can be then pushed to a repository called Docker registry [55]. Docker registry is a public or private image repository, where Docker images can be easily pushed and pulled. Public Docker registry, called Docker Hub, allows users to upload and share their own images. Developers can download each other's images and modify them for their purposes. This also makes the collaboration between the developers easier.

When operating in corporate environment, it is important that confidential information, such as custom images, is not leaked to the public. In corporate environment, it is common to use a private Docker image registry. Docker offers a possibility to run local Docker repositories for example on the company intranet. This enables the same advantages as public repository, such as easy downloading of the images, but the access to it is limited to the intranet. Private registry is also an optimal way of delivering container images to the container hosts, so that the images are not needed to be rebuilt on each host separately.

Compared to the other container managers, Docker has a lot more features, but most of them might be unnecessary for the small and simple applications. Also, the additional features such as Docker Hub make the attack surface larger. Docker also suffers from the same limitations as Linux containers in general. Managing a distributed software requires additional work because of the increased complexity. For example, managing inter-container communications can be a complex task, especially when the application consists of tens of containers. Other drawback is that the management of persistent data is more difficult than in the traditional applications. Docker containers cannot store persistent data, and all application specific data inside a container is lost when the container is destroyed. If an application requires storing of some data, it should be saved to a persistent data volume. Finally, the weaker isolation of containers makes them unsuitable for certain applications, that require strong isolation and security.

3.5 Summary

In this chapter, two different virtualization methods, hypervisor-based virtualization and operating system-level virtualization, were introduced and compared. Both of these methods are used to optimize the computing resource usage by running multiple instances on a same physical computer. From application point-of-view, hypervisor-based virtual machines and operating system-level virtual instances act similarly. Major differences are that the hypervisor-based virtualization causes some overhead, because it contains an additional operating system. Hypervisor-based virtualization also causes some performance loss in most of the operations.

Operating system-level virtualization is more lightweight and its performance is almost native. On the other hand, operating system-level virtual instances, containers, can only run applications that are compatible with the host operating system. In comparison, traditional virtual machines can run almost any operating system. Also, in the operating system-level virtualization, the host kernel is shared between the virtual instances so the isolation is weaker.

Multiple container managers have been developed to make the operation of containers easier. They offer simple tools for basic container operations, such as for creating and deleting containers. Docker is one of the most popular container managers.

4 Distributed container-based applications

Containers are usually used to manage a large, distributed application because of their flexibility and isolation. Because the overhead of additional operating systems does not exist in operating system-level virtualization, multiple containers can be deployed on a single node. As Docker documentation suggests [56], containers should only contain a small logical part of the application. Distributed applications can then consist of multiple small services, which are running in separate containers.

System with multiple containers can be managed by using a container orchestrator, which centrally manages the containers. This chapter explains container orchestration and presents a few popular container orchestrators. After this, microservice-based architecture is presented and it is compared to the traditional monolithic architecture. Also, the advantages and the disadvantages of both the approaches are explained.

4.1 Container orchestration

Container is a relatively small unit in large distributed applications. These applications can consist of tens or even hundreds of containers, which are scattered across multiple nodes. Managing containers one by one is not feasible and centralized management of containers is required.

Container orchestrator is a centralized management unit, that can operate containers on multiple hosts. Containers can be clustered and thus multiple containers can be managed as a single entity. For example launching, deleting, updating and scaling of the various container clusters can be done with an orchestrator. User actions are not necessarily required when using orchestrator, because the operations can be automatically executed using predefined policies. A cluster of containers can be automatically managed by the orchestrator, and the failures in the containers or in the host nodes can be also detected by the orchestrator. For example, containers can be moved to a new node in a case of a host failure. With an orchestrator, replication rules can be also set for the containers, and parts of the application can be automatically scaled, for example based on a current load [57].

Multiple orchestrator solutions have emerged as the use of containers has become more popular. Popular orchestrators are for example Kubernetes, Mesosphere Marathon and Docker Swarm. Orchestrators can be also built-in to the custom operating systems, that are purely designed to run containers. Examples of these are CoreOS and RancherOS. In this thesis, I am only focusing on the orchestrator applications that can be run separately.

4.1.1 Kubernetes

Kubernetes is an open-source container orchestrator developed by Google. It was released in 2014. For over 10 years, Google has been managing containers at scale by using their own tool called Borg [58]. Once other developers became more interested in Linux containers, Google decided to create an open-source container orchestrator Kubernetes, that was based on Borg. Motivation for this was to get valuable feedback and improvements from multiple skilled engineers in the field [59].

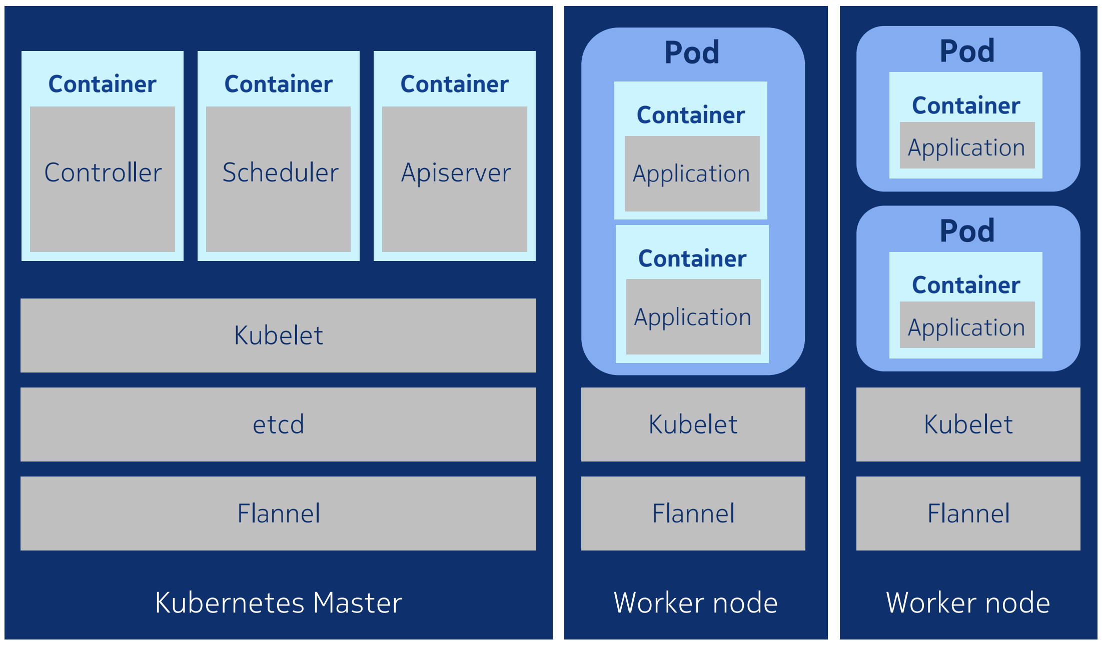

Figure 7: Basic architecture of a Kubernetes cluster

Kubernetes manages for example deployment, updating, monitoring, resource sharing and scaling of the containers in a distributed environment with multiple hosts. Kubernetes can monitor a state of the containers in the system, and the containers can be automatically re-created in a case of a failure. Kubernetes uses a key-value storage etcd [60] to store the configuration data and the state of the nodes.

Kubernetes organizes one or multiple containers into small groups called pods. Pod is the smallest managed unit in Kubernetes and it usually contains a small logical part of the whole application. All the containers in a pod are guaranteed to be located on the same host. Pod can also specify shared volumes that are automatically shared with all the containers in it. Kubernetes automatically chooses a host, where the pods are launched. It can do that for example by checking a host's load. It is also possible to label the hosts and deploy the pods only to the hosts with the correct label.

Basic architecture of Kubernetes cluster can be seen in Figure 7. The cluster consists of a Kubernetes master node and multiple worker nodes. Worker nodes are running a node agent Kubelet, which is communicating with the master node to receive for example workloads. One worker node can contain multiple pods. Master node can also act as worker node in smaller deployments. Kubernetes uses external networking component, such as Flannel [61], to provide an overlay network for the pods. Networking component is included on every Kubernetes node.

Kubernetes is one of the most popular orchestrators. It is widely used in the industry and it is supported by multiple big organizations. Kubernetes is also promoted by Cloud Native Computing Foundation, which aims to advance and promote container technologies. Another big company, Red Hat, has selected Kubernetes as the orchestrator for their OpenShift platform [62]. Also, CoreOS recently abandoned their orchestrator fleet and switched to Kubernetes [63].

4.1.2 Docker Swarm

Docker Swarm is an orchestrator offered by Docker. Docker Swarm is already included in the Docker package from version 1.12 onwards. It offers native clustering for the Docker containers. It is also using the same API as Docker, which means that the same tools can be used with Docker and Docker Swarm.

Similarly to Kubernetes, Docker Swarm also enables deployment, updating and scaling of the containers. It also offers networking, service discovery and load balancing [64]. Docker Swarm creates a "swarm" of hosts that can run Docker containers. Each host is running a swarm agent which is then controlled by a swarm manager.

Docker Swarm is simpler and easier to use than Kubernetes. Major advantage of Docker Swarm is that it ships with the Docker package and basically works out-of-the-box. However, Docker Swarm is usable only for the Docker containers. Also, because the Docker API is used, functionalities of Docker Swarm are restricted to that. Other orchestrators could be needed for the most complex applications [65].

4.2 Microservice-based architecture

Microservice-based architecture means that an application consists of multiple independent services that perform a single function and communicate with other services using well-defined interfaces [66]. These services can be developed and deployed independently. Microservices can be implemented using different technologies and programming languages.

Opposite of microservice-based architecture is called monolithic architecture. Monolithic applications are huge and every functionality is built-in to a single application. Each component of a monolithic application is required to be able to develop and deploy the application. For that reason, development of a monolithic application can be very difficult, especially when the application is large and developers change during the application life cycle.

Figure 8 illustrates the differences between monolithic and microservice-based architecture. Scaling of a monolithic application requires that each component is scaled. In practice, this means that the whole application is duplicated. Single component of a monolithic application cannot be scaled independently. In contrast, if an application is built with microservices, each service is isolated and independently scalable. It is possible to scale just the microservice that is required, without duplicating any unnecessary services.

One issue with monolithic applications is that it is very hard or practically impossible to change the technologies used in them. Technology stack of a monolithic application is very inflexible and it is really hard to change underlying technologies without a complete rework of the application. Even a gradual adoption of newer

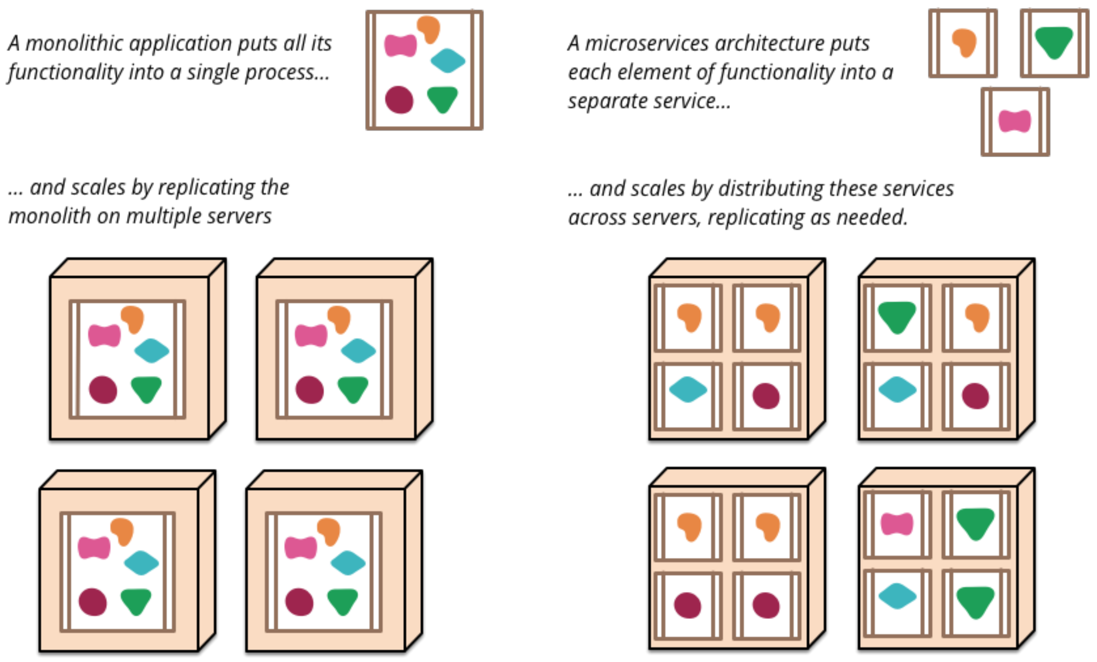
A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

Figure 8: Monolithic and microservice-based architecture [67]

technologies is very hard. All components in monolithic applications usually stick with the technology that was chosen in the beginning of the development.

In microservice-based approach, the services are independent and isolated, so the technologies they are using are irrelevant. Most suitable technologies and programming languages can be selected separately for each microservice. Adopting a new technology for a microservice might require some rework to the service, but the required work is significantly smaller than with the monolithic applications.

Microservice-based architecture also solves so called "dependency hell" that is often encountered with monolithic applications [45]. Modern applications are often built from existing components which require certain dependencies. However, these components might require different versions of the dependencies, which will cause conflicts. Monolithic applications have to solve these conflicts, and it might easily lead to non-optimal solutions. Microservice-based approach does not have this issue, because each component can have its conflicting dependencies inside its isolated space.

Microservice-based application can be distributed on multiple hosts. All the services are able to find each other by using a service discovery protocol, so that their physical location does not affect the functionality. Parts of the application can be for example deployed into a public cloud [68]. The distributed architecture can also make debugging of the application easier, as inter-service communications can be debugged with existing tools such as tcpdump. However, in monolithic applications, specific tracing capabilities have to be built-in to the application to be able to debug the application execution.

Microservice-based architecture enables independent development of a single service. One microservice can be separately changed or updated, without it affecting other microservices in the system. Because of this, smaller development teams can be created so that each of them would be responsible of specific microservices. This also makes it easier for the new developers to start working with the application. They do not have to have deep knowledge of every single component of the application, and can just focus on specific services.

A small, one line change in one microservice does not require the compilation of the whole application. Deployment of a changed service is also faster and more robust. By using microservice-based approach, smaller releases can be frequently published during a year. Because of the more frequent and smaller patches, developers can more easily fix for example security vulnerabilities. In addition to these, normal bug fixes can be also delivered faster.

Changes to a monolithic application are usually delivered as large releases with relatively large time intervals. Because of this, even the smallest fixes could be delayed by weeks or even by months. Other problem with large software releases is that the releases are often packed with as many new features as possible. This makes the update process very risky, because many features and functionalities change during one update. Massive testing is also required and the update process is much slower. Software update of a monolithic application is a very resource-intensive process. Also, the monolithic application needs to be always restarted when applying updates.

Microservice-based architecture enables continuous delivery and deployment [69]. Changes to the services can be done more efficiently because of the smaller dedicated development teams. On top of that, delivering and deploying the changes to a production environment will become faster, because only the changed microservices have to be updated. Continuous monitoring of the microservices will also give important feedback to the developers.

However, microservice-based approach has its own disadvantages. Microservice-based architecture introduces additional complexity, because the components of the application are distributed. Application also needs to handle different service types. Microservice-based application development and usage will require a lot of automation to be effective. An effective monitoring system must be also included to be able to handle all the services efficiently.

Another disadvantage is that there is a need for inter-service communication between different services. Communication is happening via remote-calls, which will generate a networking overhead. The bandwidth and latency requirements must be taken into account when planning microservices [70]. For example, telco applications have strict requirements for the latency and bandwidths.

There are also some misconceptions about microservices. Some developers think that a microservice should be as small as possible. However, this will often lead to the non-optimal solutions with too much communication overhead between the services. A microservice should just contain one logical entity. Another misconception is that microservices can be developed however the developers want. However, a microservice has to be planned so that the interactions with other microservices are taken into account. For example, the interfaces of a microservice have to be well defined and logical [71]. Otherwise, a microservice-based application can end up as a big monolith, where single services cannot be easily changed without it requiring a lot of rework to other services.

Security of microservices must be taken into account in planning. Some services are designed to be externally accessible, while others have an absolute requirement of isolation. Firewalls can be implemented to prevent external access to the delicate parts of the application. Intelligent network planning can also enhance the security of the application.

4.3 Summary

This chapter focused on distributed applications that are utilizing containers. To manage multiple containers efficiently, a container orchestrator is required. A container orchestrator can centrally manage the containers across multiple hosts. Containers are most suitable for running small entities, such as microservices. Microservices are small independent services that perform a single function and communicate with each other using well-defined interfaces. Microservices can be also developed and deployed separately. Distributed applications can then consist of multiple microservices.

5 Running telco application in containers

To study the benefits and requirements of containers, a practical proof-of-concept was conducted. In the proof-of-concept, a generic telco application was used as an example, and plan was to move the application functionality into containers. In this chapter, the practical implementation is explained. First, the example application and the containerization approach are discussed. Then, the process of moving the application services into containers is presented. Finally, the supporting platform services are explained.

5.1 Containerization approach

To study containers in practice, a generic telco application was used as an example. The example application consist of four virtual machines: application node, database node, load balancer node and UI-node. The architectural structure can be seen in Figure 9. Each of these virtual machines contains an operating system, libraries, platform services and application services. The application services provide the actual functionality of the nodes. The platform services contain for example monitoring, logging and troubleshooting services. All of the virtual machines are managed by OpenStack [72], which is an open-source platform for virtualization and cloud computing. OpenStack also offers other services, such as virtual disks for permanent data storage.

Application node provides the actual functionality of the example application. Depending on the deployment, the application node can be scaled out to increase capacity. Database node runs database processes, which manage and store the data

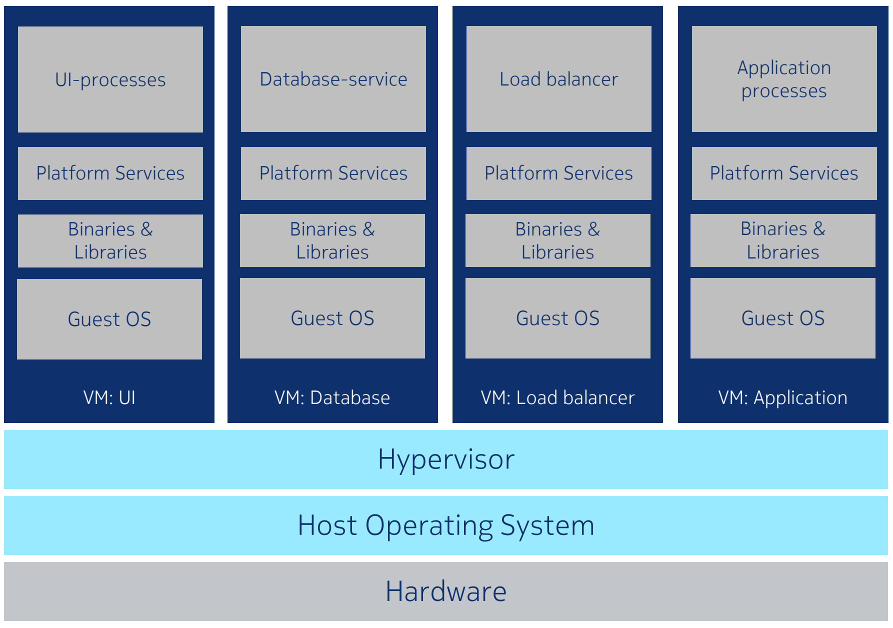

Figure 9: Architecture overview of the example application

that is required for the application functionality. The actual data is stored on a virtual disk that is offered by OpenStack and is mounted to the database node. Load balancer forwards the traffic from the clients to the application nodes. It divides the traffic load evenly. The UI-node is used for the operation and maintenance. It also contains a graphical user interface. All the services on each node are managed by a supervising process systemd.

The example application has a monolithic structure, and the deployment of the application must include all of the components. Also, the development of a single component requires all the other components to be present. The application can be deployed with varying amounts of application nodes, but adding and deleting them afterwards requires restarting of the other virtual machines. All the components are always expected to be found from the same location, because the application does not have a service discovery framework.

Before introducing containers to the application, the monolithic structure has to be broken down into smaller logical pieces, which can be then moved into containers. It is important to proceed in small logical steps, so that the migration is not done simultaneously on all parts of the monolithic application. Other possibility for the migration is to start again from the scratch and build the same application by using microservices. However, this approach would have required a lot of work, and was basically out of the question in a context of this thesis.

In this thesis, I decided to begin with the old virtual machine based structure and to proceed in logical steps towards the container based architecture. My plan was to first identify different functionalities of the application and their dependencies on other components. Then, I isolated these smaller pieces by using containers. The containers were still running on top of the virtual machines.

By taking this approach, only a few modifications were required to the actual application source code. Also, the identification of the functional pieces was relatively simple as the virtual machine structure was followed. Because the containers were still run on top of the original virtual machines, the physical location of the services did not need to change, and the migration could be done service by service. Functionality of the whole application was always easy to test, because the application was never completely broken.

Goal of this practical experiment was to introduce containers into the application, so that the benefits, issues and requirements of containers could be easily observed. Dividing the whole application and all of the platform services into microservices turned out to be an unrealistic task in the context of this thesis, because of the time constraints. Because of this, I decided to focus only on the application services rather than the platform services. Figure 10 shows the architectural plan for the practical implementation, where most of the application services are run in containers. Same approach can be later used for all the other components of the application. In the future, all parts of the application, including all of the platform services, could be running in containers, and they would not have any ties to the underlying host architecture.

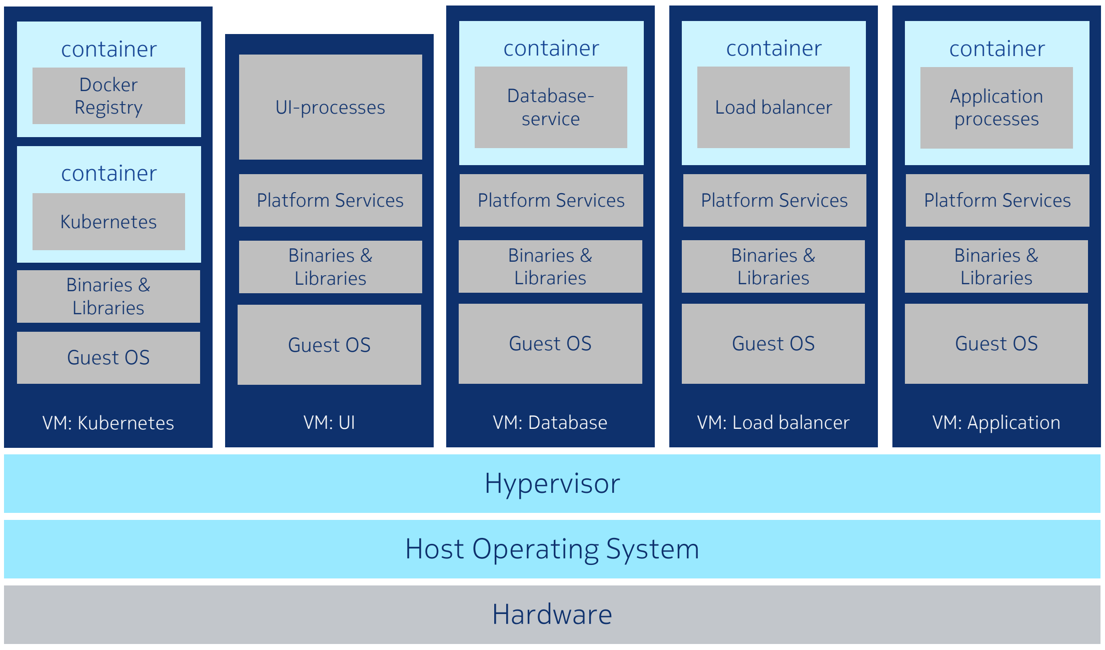

Figure 10: Application architecture with containers

5.2 Software setup

In this proof-of-concept, I decided to use Docker as a container manager. Docker was selected because it is easy to use and it fits the purpose of this proof-of-concept very well. Docker is also the most popular solution for containerization, and it has been already successfully used in multiple applications in the IT world.

Because the example application was already distributed on multiple hosts, a container orchestrator was needed. Kubernetes was selected as the container orchestrator, because it offers a wide range of features, such as pods, rolling update and rollback. Kubernetes is also one of the most popular orchestrators and for example cloud native computing foundation promotes the use of it [73]. Kubernetes was also used in another proof-of-concept that was related on the container orchestration, so it was logical to use it in this thesis as well.

Each container image that was used in this exercise was built with a Dockerfile [74]. Every container image had Fedora Linux base image [75] and generic company specific libraries. In addition to these, depending on the service, the container image also included the application binaries and their dependencies, such as libraries and other supporting components, such as python and perl. Using Fedora as a base image resulted in a small increase in size of the container image. However, it made the first phase of the integration much easier, as all the generic libraries and tools were already available.

5.3 Application architecture changes

The functionality of the example application is provided on multiple application services, that are divided on four nodes. They consist of a database service, load balancer and multiple application processes. In order to study containerization, these were first moved into containers node by node.

5.3.1 Load balancer

The example application uses an open-source load balancer service HAProxy [76] with custom configuration, to enable the traffic division to each application node. The HAProxy service is managed by systemd and is started automatically with the host. HAProxy processes are launched by using a custom start-up script that is tied to the systemd service. The startup script will create the required directory structure if it does not already exist. It will also fetch the required certificate-files from LDAP, which is a directory service that provides access to files, devices and other data in a distributed system.

HAProxy binaries and the startup script were first moved inside a container. After this, missing libraries and dependencies were found out by using Linux debug tools, such as ldd [77] and strace [78]. Ldd is used to print the library dependencies of some binary. Strace is printing out all the system calls during a binary execution, so that for example missing files can be detected.

In the first phase, the host's network was used to guarantee connectivity between the load balancer and the application nodes. The original HAProxy service had to be also killed before launching the container. This had to be done, because when the host's network is used, both services will try to bind themselves on the same port with the same IP address, resulting in a conflict.

HAProxy configuration did not need any changes as all the application services were still using the network of the host. If a separate container network was used instead, the current load balancer configuration would have to be changed to look up the application service locations for example from a service discovery framework.

5.3.2 Database

The example application uses an open source database service PostgreSQL [79]. The database service consists of a watchdog process that launches and monitors the actual postgres processes and keeps them running. In the original architecture, the watchdog process has all the postgres processes as child processes and the watchdog process itself is managed by systemd. Initializing the system and variables for the database service is done by a startup script.

The approach for containerizing the database service was similar to the load balancer node. First, the libraries, binaries and a startup script of the postgres service were moved into container. After this, the dependencies were resolved and moved inside the container. When the database service was put into a container, process hierarchy remained almost the same as before containerization. However, after the containerization the postgres processes were running inside a PID-namespace, so they were not able to see any other processes on the system.

As mentioned earlier, containers are designed to be launched and deleted, and they are not usually restarted. When a container is terminated, all the data inside it is lost. Because of this, the database container requires additional persistent storage. Persistent data can be stored either to the host's disks or to a Docker data-volume. I decided to use a virtual disk from the host as a database storage. The virtual disk was provided by OpenStack and it was mounted to the database node. To enable persistent storage for the containerized database processes as well, the virtual disk had to be mounted to the container. The actual virtual disk was still mounted to the host node, and the mountpoint was just shared with the container.

I also decided to use host's network for the database container. This was done, because other services, such as application processes in the application node, are expecting that the database is always located in the database node. By using the network of the host, the IP address and the port of the database remained the same. This approach is required until a service discovery framework is introduced to the system. Even then, the database container cannot be deployed anywhere, because the host has to have the virtual disk mounted.

One issue that I encountered during the containerization process was that the database service status was heavily tied to the host's init process systemd, which manages and monitors all the services in the system. Also the alarms, which are user notifications about system errors, were tied to the database service status in systemd. When the service was moved to a container, the information about the database service status was lost and the alarms about database unavailability were set. However, this issue was mainly cosmetic and the database still functioned correctly regardless of the active alarms.

5.3.3 Application node

The actual functionality of the example application is done by multiple processes in application node. In the example application, all the application processes are managed by systemd. There is also a process that is specifically monitoring the application processes that provide the actual application functionality. If a failure is detected on any of these processes, all of them are restarted. This is done so that the startup order of the processes can be always guaranteed.

Because the application functionality consists of multiple services, I had to test multiple approaches for the containerization process. First, I tried out an approach where every application process was running in a separate container. The set of containers were put into one Kubernetes pod and were managed as a group. The containerization process was again similar to the previous nodes. Service binaries, libraries and their dependencies were found and moved into containers. The container image was based on Fedora and it also contained some basic libraries and binaries.

However, this approach had multiple issues. The first problem that I encountered was that Kubernetes cannot guarantee a starting order of the containers inside the pod. Some of the application processes required strict startup order to work properly.

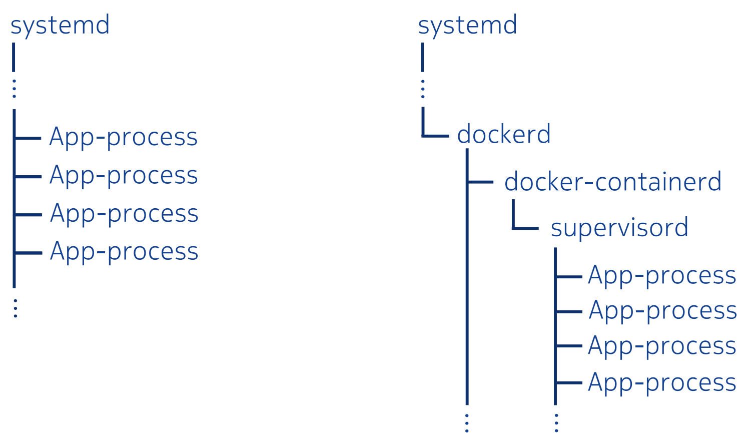

Figure 11: Application processes on a host and in a container

For example, one process required an access to a file that was created by another application process. This problem could have been solved by implementing so called "launch-and-wait" principle to the processes. The processes could then be launched in an arbitrary order and the they would be able to wait for the other dependent services to become available. Another problem with this approach was that some of the processes were accessing shared files in the host. If the required files were copied to each container separately, they could not be managed easily. On the other hand, if the shared files on the host were mounted to the containers, the issue would not exist. However, this solution would require that the files exist locally on the container hosts, which would set unnecessary requirements for the container hosts. Both of these changes would have required a lot of re-designing to the application processes so I decided to focus on other approaches instead.

Next approach was to put every process into the same container. Processes were launched by using a simple script, that launched the processes in certain order. This approach solved the issue with startup order, without requiring any modifications to the processes themselves. The shared files were also moved to the container, and all the processes were able to access them when necessary. The issue with this approach was that the container manager was only able to monitor the state of the first process that was launched. In this approach, the first process to launch was the startup script, which means that if any of the application processes die, the system would not able to detect that, and the process could not be restarted.

The final approach was similar to the previous one. All the application processes were run in a same container, but the processes were launched and monitored by a supervising process called supervisord. With supervisord, the startup order of the processes can be determined. Supervisord also monitors the application processes and restarts them in a case of a failure. Figure 11 shows how the application processes are located in the process tree. On the left side, the application processes are just running on the host without any containers. On the right side, the application processes are running in the same container with supervisord launching and monitoring them.

For the purposes of this proof-of-concept, the approach with a single container and a supervisor was the most suitable. The approach offered an easy solution for both failure handling and shared memory. Also, the solution did not require any changes to the actual source code of the application. On top of that, the management of a single container was really simple from the orchestrator point-of-view. Until the processes are able to start regardless of the order, an external supervisor such as supervisord is required to manage the startup order.

5.3.4 UI-node

UI-node contains operation and maintenance tools and for example the graphical user interface of the application. It also runs rsyslog [80], which is aggregating logs from the other nodes. I did not add any containers to this node, because the scenario would have been really similar to the scenarios of the other nodes and would not have brought anything new to the thesis. For example, the graphical user interface could be containerized using the same approach as before, but the practical work was omitted in a scope of this thesis. In the future, every part of the UI-node should be containerized so it would not be dependent on the host anymore.

5.4 Platform services

The platform also required some changes to enable containerization. For example, a container orchestrator, service discovery framework and logging services enable easier management and add robustness to the system.

5.4.1 Orchestration

Kubernetes was chosen as an orchestrator for this proof-of-concept. All the management tasks of the containers are handled through Kubernetes master components, that are running in Docker containers. Kubernetes master components can be deployed anywhere in the network, as long as there is a connectivity between the Kubernetes master components and all of the worker nodes. In this proof-of-concept, I decided to deploy the Kubernetes master components in a separate node in the same stack as the example application. This was done so that networking between the master node and the worker nodes functioned without any issues.

Private Docker registry was also deployed in the same node as the Kubernetes master components. The registry contained all the required container images, including the container images for all of the application services as well as the images for all of the Kubernetes components. As with the Kubernetes master, Docker registry is also running in a Docker container, and it can be deployed anywhere, as long as the there is a connectivity between the registry and the Kubernetes worker nodes. In this proof-of-concept, the easiest way was to deploy the Docker registry in the same stack and the internal network as the worker nodes.

In the proof-of-concept, three Kubernetes pods were used; one for the database, one for the load balancer and one for the application processes in the application node. Each pod consisted of one container. This is a logical approach for example in the load balancer node, where the container runs only one process. However, in the application node, multiple processes are required for the functionality. I still decided to run all the application processes in a singe container. I also included supervisord to the container, because it offers the ability to start processes in a correct order, and it makes using of the shared files easier for the processes.

In addition to the Kubernetes master node, each virtual machine had kubelet and etcd installed, making each of them a Kubernetes worker node. When commands, for example to deploy or to delete a pod, are entered in the Kubernetes master, it will contact the kubelet in a correct worker node. Kubelet then handles the communication with the Docker itself. For example, when creating a pod, Kubelet first instructs Docker to fetch the correct images from an image repository, and then to launch the containers with flags and options that are defined in the pod configuration file. Kubernetes chooses a worker node for the new containers based on the current load on worker nodes, or by using labels. Worker nodes can be assigned with labels, and the containers can be configured so that they are only launched on the nodes with certain labels.

To guarantee functionality, every Kubernetes worker node was labeled with the host's name. This approach ensured that the containers in this proof-of-concept were always deployed on the same node as before containerization. For example, the database container configuration included a rule, that the container could only be launched on a node that is labeled as "db". This approach is necessary until a service discovery framework is present and containers can be deployed everywhere. In the future, labeling of the nodes could be used to ensure that the database containers are only deployed on worker nodes which contain fast disks, or that the application containers are only deployed on the high performance worker nodes that contain more computing resources.

Kubernetes can also offer an overlay network for containers through external networking components, such as flannel. By using the overlay network, Kubernetes can provide an individual IP address and a port for each container, so that multiple identical containers can be run with different IP addresses on a same node. This is a common practice for example for load balancing and high availability purposes.

In the practical implementation, the host's network was used for the containers. The benefit of this approach was that the containers were able to find other services without an overlay network or a service discovery framework. For example, the database container and the service inside it were still running on the original database node with the same IP address and the port as previously. However, by using the host's network, multiple identical containers cannot be launched on the same host because the services bind themselves on the same ports. If a port is already used by some other service, the service in a new container is unable to start.

5.4.2 Service discovery

If a separate network was used for the containers, the services would not necessarily have the same IP addresses all the time. Containers usually get the IP addresses allocated dynamically. In that case, a service discovery framework is required, so that the services are able to find each other. A part of the service discovery framework is a service registry, which is a database of available services and their network locations. When a new service becomes available, it registers itself to the service registry. Similarly, when the service becomes unavailable, it removes itself from the registry. Registering and de-registering a service is usually done via application programming interface (API), such as REST API. Popular service registries are Consul [81], Netflix Eureka [82] and Apache Zookeeper [83].

In this proof-of-concept, a service discovery framework was not implemented because of time constraints. Also, because the practical implementation was using a host's network for the containers, the service discovery framework would not be really useful, as the services already know where to find each other.

5.4.3 Logging

Originally, the example application collects and stores logs by using journald, which is a part of a systemd package [84]. Each host is running journald locally, and in addition to that, the logs are also sent to a centralized storage by using rsyslog. However, in containerized architecture the services are running in multiple isolated containers. Because containers should not store any persistent data, the logs should not be stored inside the container. Otherwise, a failure of the container would also destroy the logs. Also, adding an additional logging daemon in every container would result to unnecessary overhead.

Two main approaches for implementing the logging in a containerized application are to either use host's logging capabilities, or to start pushing the logs of containerized processes into a standard output, and then collect them by other means. Docker offers multiple logging drivers, for example for syslog, journal and fluentd, that make use of the host's logging capabilities [85]. The corresponding services have to be running on the host machine. All the logging calls from containers are then forwarded to the logging daemon which stores the logs on the host's disks. The logs can then be forwarded from the host to a centralized location, for example by using rsyslog. This is a clean way to implement logging in containers, but it sets requirements for the container hosts.

Another possibility is to forward the logs of containerized processes to the container's standard output. This can be done by either modifying the application to push the logs to the standard output automatically, or by including a logging service such as rsyslog in the container, and configuring it to push the logs to the standard output. There are multiple ways of collecting the logs from the standard output. With Docker, it is possible to save the standard output of the containers into a json-file. It is also possible to use some third party application, such as logspout [86], to collect the container output logs. This way, the host infrastructure does not have any requirements regarding the logging.

For this proof-of-concept, I decided to use the first approach and implement logging using host's journald. The approach was selected, because currently the example application is still running on top of virtual machines, and each host is already running a journald. Also, the logs from each node are already automatically sent to a centralized location by rsyslog, so additional aggregation of the logs was not required. However, the selected approach sets requirements for the container hosts, and the standard output based logging should be investigated and implemented when designing a fully containerized application.

5.5 Summary

This chapter discussed practical implementation of a telco application, that was utilizing containers. For this implementation, Docker was selected as a container manager and Kubernetes was selected as a container orchestrator. Basis of the practical implementation was an existing telco application, which was modified to utilize containers. Functional parts of the application were moved into containers, which were running on top of the old virtual machines. In addition, a separate virtual machine was deployed to the stack and it contained a Kubernetes and Docker registry. Orchestrator was used to make management of multiple containers easier.

6 Evaluation

This chapter discusses the advantages and considerations related to telco application containerization. The practical implementation in the previous chapter is used as an example. First, the containerization advantages such as software update, performance and scaling are discussed. Then, general guidelines for container-based application design are presented, and container security is discussed. Finally, the future work is presented.

6.1 Software update

An update of a service is a slow and heavy process in the old virtual machine based architecture. In the old method, user has to first download large virtual machine image from software manufacturer's server. Size of the image is usually from few hundred megabytes to one gigabyte, mostly because the VM-image also contains an operating system. When a modification is made to the VM-image, user needs to download the whole image again regardless of the modification's size. In addition, the application might consist of multiple different VMs, and each of them might have a different VM-image.

After downloading the image, user has to launch a new VM with it. Startup of the new VM is really slow, because the guest OS needs to be started first. After that, all the platform services and finally the application specific services are launched. VM startup times can be even several minutes [2]. While the new VM is deploying, an old VM has to be still running to avoid service interruption. This will cause some unnecessary computing overhead. After the new VM is started, all the other services in the system must be instructed to use it. Only after all the other services are communicating with the new VM, the old VM can be killed. The whole process is really slow and will introduce challenges in the environments where high availability is required.

In contrast, the update process with containers is very light. The user needs to download a new container image the same way as in older VM-based architecture. However, the container image is much smaller than a full VM-image, because it does not contain an operating system or any other unnecessary services. It only contains the functionality that is required to run a service inside a container. Another advantage of the container images is that they are layered, and only the missing layers are downloaded. Other layers can be mounted from the images that already exist in the system.

Container startup time is really fast compared to a VM startup, because only application specific services are launched. In practice, this means that the container startup times are usually only a few seconds. This also makes the update process really fast. If other services can recover from the short interruption in a service that is updated, no special precautions are needed. If the service is critical and no interruptions are accepted, a new container can be launched next to the old one like in VM update. Traffic is similarly directed to the new container before updating the original service. When containers are used, this is still a very fast process. Updating containerized services does not usually require rebooting of the underlying machine. It is often enough just to launch the updated container again.

The container orchestrator makes the update of a containerized service easier for the user. Only one command is entered and the orchestrator handles downloading, launching and terminating of all the required containers automatically. An example command for updating database containers in the database deployment would be:

kubectl set image deployment/database-deployment database-container=database-image:v2

The command will update every container named "database-container" in "database-deployment" with the new image. In a normal scenario, where the deployment only contains one database container, the old container is simply killed and a new container is launched with an updated image. If the database deployment is scaled or if it contains multiple database containers, there is an option to select an update policy. By using an update policy called *recreate*, the orchestrator will just kill and relaunch all the database containers in parallel. If the update policy is set to *RollingUpdate*, the orchestrator will keep at least one database container running at all times. It will first kill one container and relaunch it with an updated image. Only after the new container is deployed, the other containers are updated one by one.

6.2 Scaling

Scaling out means adding more components in parallel to share workload between multiple instances. In contrast, scaling in means reducing the number of the parallel components. Scaling out is a common way to increase capacity of an application.

In the example application, the application node can be scaled out to increase capacity. If the application needs to serve a large number of requests simultaneously, new application nodes can be launched to share the load. A load balancer is then responsible for dividing the traffic into each application node. In the old VM-based architecture, a smallest unit of scaling is a VM. However, scaling a whole VM causes a lot of overhead, and the scaling process is really slow.

In this proof-of-concept implementation, Kubernetes can be told to scale out an existing pod. It will then proceed to launch identical pod somewhere in the environment based on the predetermined rules. For example, it is possible to define that application pods are only launched on the nodes with a specific label, such as "application". Kubernetes will select a node for the new pod by using its scheduler-module, which for example examines the current load of each node [87]. As with the VM-based architecture, a load balancer divides the traffic evenly to the parallel services.

It was noticed that the scaling out process of the example application failed if the new container was created on a node that already had identical container or the same application processes running on the host. The scaling out process failed, because the processes in the new container were still using host's network and tried to bind themselves on the ports that were already bound to the original processes.

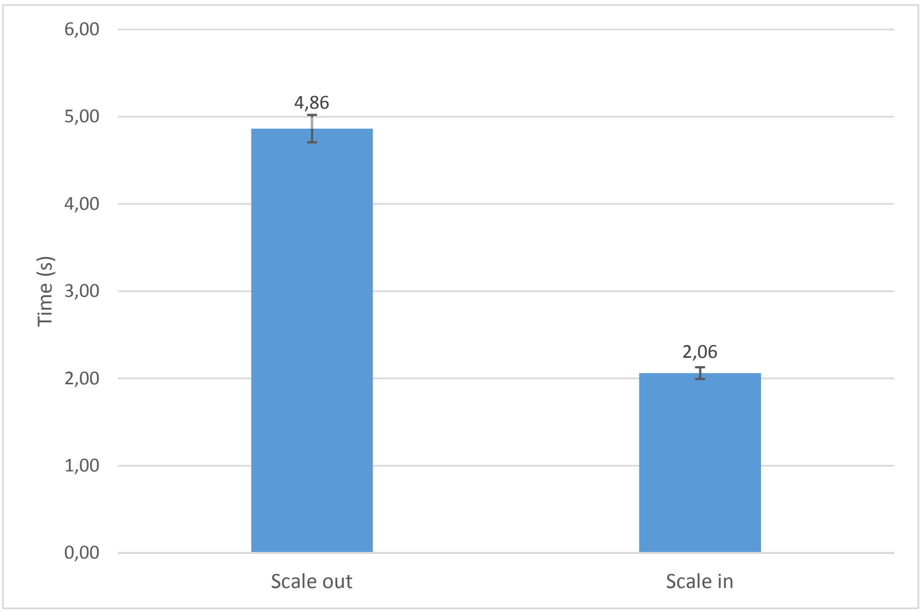

Figure 12: Application container scale-out and scale-in times

In the current solution where the host's network is used, scaling cannot work within the same node.

The problem was avoided by creating a place where Kubernetes can safely launch a scaled container. In this proof-of-concept, I decided to add one idle worker node to the stack. This idle worker node was always running, but it did not have any application services running to avoid conflicts. The new node was configured as a new Kubernetes worker node and it was labeled as "application", similar to the first application-node. Load balancer configuration was also changed to use this additional worker node as an application node. If the new application node was not running any application services, the load balancer just used the original application node. This approach simulated a situation, where application container can be freely scaled, without conflicts for example in port bindings. However, if the practical implementation had separate network for the containers with service discovery framework, the additional VM would not be required. The two application containers could be just run on the same node, but they would be using different IP addresses.

A command that was used for the scaling was:

kubectl scale deployment/application-deployment –replicas=2

where "application-deployment" was the name of the deployment. The deployment consisted of one pod, that contained one application container. Kubernetes then detected that the new worker node had no application containers running, and it launched the new pod there. Scaling times can be seen from Figure 12. Scaling out took approximately 4,9 seconds and scaling in only 2,1 seconds. Scaling out is a slower process, because all the application services have to be started inside the container. In contrast, scaling in only deletes the container and all the processes inside it.

6.3 Failure handling

To investigate recovery time of a single service in the application node, four test scenarios were created:

1. No containers in use
2. One container with multiple services and supervisord
3. All services in separate containers
4. All services in separate containers managed by Kubernetes

All the tests scenarios were carried out in the application node, that was running all the necessary services. Same service was killed in every scenario and time was measured until the service was back up. The service was killed with SIGKILL-signal and it was brought back by the infrastructure.

In the first situation, recovery is handled by host's systemd. It does not only restart the killed process, but it also restarts other relevant application processes. In the second scenario, supervisord inside a container detects the failure of the service and restarts it. It does not restart any other processes. In the third scenario, the Docker container exits, because the entrypoint process is killed. Docker then proceeds to launch the container again. In the fourth scenario, entrypoint process inside a container is killed and the container exits. Container orchestrator Kubernetes will detect that container exited and will launch a new one. Results can be found from Figure 13.

First scenario was significantly slowest: a killed service was restarted after 8,2 seconds on average. Reason for the slower restart was that the original supervising process on the host restarts all the relevant application processes if one of them dies. This is done so that the services are always launched in a correct order, and that no unknown complications are caused by an interruption in one service. However, this also means that failure in one service will make the whole application unusable until the services are restarted.

In the second scenario, recovery time was 1,0 seconds on average. Supervisor detected the failure of a service inside the container and restarted it. Other services and the container itself were not restarted. However, restarting only the failed service might cause errors in other services if they are not developed to handle these interruptions. Also, if services are relying on a specific starting order, restarting a single service might break functionality of other services.

In the third scenario, all the services were in separate containers, so that each service was an entrypoint of a container. *Restart=Always* option was defined on container launch to enable automatic restart when a container exits. In this test

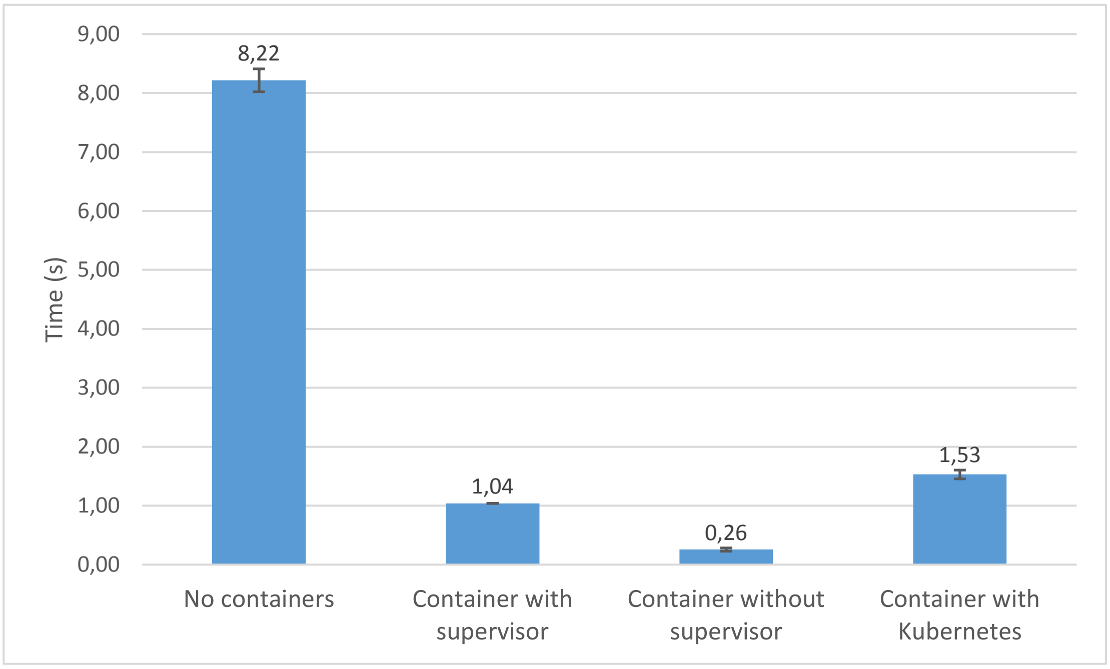

Figure 13: Recovery times on process failure

scenario, failure of the service terminated the container right away. Docker then detected that the container was terminated and proceeded to launch it again. This was the fastest scenario and restart happened on average within 0,26 seconds. However, the problem is the same as in the second scenario, because only a single service was restarted.

In the fourth scenario, the container terminated because the entrypoint service was killed. Kubernetes then detected this and launched a new container. Average recovery time of a killed service was around 1,5 seconds. With Kubernetes, it is also possible to launch containers with *Restart=Always* option selected. By using that option, restart is handled by Docker itself, same way as in the third scenario.

The test script required some modifications for the fourth scenario, because Kubernetes includes a feature called *CrashLoopBackOff*. It will double the time between each container restart if it detects frequent crashes in some container. That is why the measurements were run once every 15 minutes to prevent *CrashLoopBackOff* from increasing restart times.

6.4 Containerization practices

When designing a container based application, general guidelines should be followed. For example, it is important to design the application so that the containers are kept as logical entities, and that the application is independent of the host infrastructure.

6.4.1 Containerized applications

Containerized application should work regardless of the order in which the containers are launched. Traditionally, the processes are started by using systemd or other supervisor, and the start-up dependencies can be easily specified. However, the container environment cannot usually guarantee the start-up order of containers. For example, Kubernetes cannot guarantee the order of which containers are started inside a pod. I encountered this problem when the application processes of the example application were put in separate containers. As a workaround, I had to put all the application processes in a single container that also had a lightweight supervisor process to manage the processes and their start-up order. However, this approach is not recommended for truly containerized applications. Instead, the services in the containers should be able to wait for other services to become available. Services can still have dependencies on other services, but their startup order should not affect the deployment of the containers. Suggested solution would be to redesign services to apply continuous retry, so that a missing service-dependency would not terminate the application right away.

Containers can be used to run either a single process or multiple processes. In the proof-of-concept, the HAProxy load balancer was running a single process inside a container, whereas in the application node, multiple application processes were run inside the same container. It is recommended to run multiple processes in the same container only when the processes are heavily related, and are for example utilizing shared memory and shared files. For example, in the proof-of-concept, some of the application processes were using shared memory so it was logical to put these processes in a same container. Other solution would be to share an IPC-namespace between the containers that access shared memory. For example, Kubernetes pods automatically share an IPC-namespace between its containers. However, unnecessary coupling of processes will lower manageability and introduce unnecessary complexity to container images.

If multiple processes are run inside the same container, the container should also have a supervising init process, such as supervisord or systemd. This is not only the case with multiple different processes, but supervisor should also be used if a single process inside a container launches additional child processes. The scenario would then be similar to launching multiple processes initially. Supervisor must be configured so that failures in one process can be resolved by restarting it. Other processes in a container must be developed so that they can recover from that failure, or that the supervisor knows to restart all the required processes. In addition to process restarting, supervisor can also clear hanged zombie processes and reap orphaned processes [88].

6.4.2 Infrastructure

Containerized application should not have any expectations from host software and services. For example, containerized applications should not be expecting that the host has a systemd that would handle the dependencies. Also the other parts of the application, such as alarms, cannot rely that a status of a containerized service is tied to any host service. In addition, the containerized services should not expect that host has for example syslog or journald, that would handle logging. Logs from the containerized service should be pushed to a standard output and then collected by the infrastructure.

Also, the service inside a container should not be dependent on the host infrastructure. For example, an application should work regardless of the container orchestrator that is used. The application should be able to function on any kind of infrastructure, as long as containers are supported. The container infrastructure must take care of container re-creation in a case of container failure. Services inside containers should not crash if another container momentarily goes offline. Instead, they must wait for the required services to become available. Also, in a case of a host node failure, container orchestrator must be able to move all the containers from the broken host to a functional one. The host should then be able to recover automatically, for example by initiating a restart.

New applications should also be developed to be as stateless as possible, because persistent data storage in containers is problematic. However, in some cases, persistent storage is required and it should be then offered for example by the host. For example, the host's data volumes can be mounted to containers. In this case, containers must be only placed on the hosts that have these data volumes.

All containers should be managed in a logical and centralized way. If the application consists of multiple containers, a container orchestrator should be used. This is especially true when containers are divided to multiple hosts. By using an orchestrator, failure handling, software update and scaling can be instructed from one centralized point. Otherwise, all of these operations would have to be done separately to each container.

The containerized application can have two different delivery models based on the deployment. In an embedded deployment, all container related software components are delivered and deployed as a part of the application. Both the application and the supporting container infrastructure are delivered by the telco application provider. In a provided deployment, only the containerized application is delivered. The operator offers container management and orchestration. In the beginning of containerization process, using an embedded deployment is a safer option because the container technology is still transforming rapidly. By delivering an embedded and properly tested solution, the application is guaranteed to work as intended, regardless of the environment. In the future when container technology matures and operators start to have their own container infrastructure, delivering only the application containers is an option.

6.4.3 Networking

Containers should have a separate network, that is offered by the container manager or the orchestrator. The network is needed so that multiple containers in a same node would not have conflicting IP addresses and ports. For example, the database service cannot be scaled out within the same host, because the initial database service is still bound to the original port. However, at the initial phase of containerization, host's network can be used to maintain connectivity to the original services, that are still running on other hosts.

In fully containerized architecture, both IP addresses and IDs are assigned dynamically to the containers. Container cannot be guaranteed to launch with a specific ID. When a container is killed and launched again, its container ID changes. The application must not rely on fixed IP addresses or IDs. Services of the distributed application can be found by using a service discovery framework instead. The service discovery framework should be offered by the container infrastructure. When a service becomes available, it should register itself to the service discovery framework. Same way, a service should remove itself from the service discovery framework when it becomes unavailable. Services can locate each other by polling the service discovery framework.

6.4.4 Container images

When designing container images, the layered structure of container images should be utilized as much as possible. Existing layers should be reused between different images. For example, one layer could include a base OS and another layer could include python. The container image layers should be used as basic building blocks, so that only the new layers are needed to be downloaded between different versions. Developers should not use different base images in their container images unless absolutely necessary. Using the same base image in tens of container images will only take disk space for one base image. Because each modification adds a new layer to the image, it can be useful to combine layers when the amount becomes too high.

Overall size of the container image should be kept as small as possible. Developers should avoid adding unnecessary packages to the container images. Container image should include only the components and libraries that are required for the functionality. This will ensure that images stay compact, and that the attack surface is minimized. Image size can be also greatly reduced by deleting a packet manager cache after installing the required tools. To confirm this, I built two similar Docker images. Both images were based on Fedora and had both python and perl, which were installed using a package manager. Without clearing the package manager cache, the resulted image was 474.4 MB. By removing the package manager cache, image size was reduced to 311.6 MB.

6.5 Security

The process isolation and security is weaker in containers than in virtual machines. However, with careful planning, the security hazards can be minimized. Two major security principles were specified over 40 years ago, and they are still valid for the applications in containers. The two principles are [89]:

1. "Least privilege: Every program and every user of the system should operate using the least set of privileges necessary to complete the job."

2. "Least common mechanism: Minimize the amount of mechanism common to more than one user and depended on by all users. Every shared mechanism (especially one involving shared variables) represents a potential information path between users and must be designed with great care to be sure it does not unintentionally compromise security."

6.5.1 Container capabilities

Processes inside the containers are assigned to capability groups which in turn manage access rights and capabilities of the processes in the host system. The safest way of running containers would be just to run them without any dangerous capabilities. However, this would make the applications in containers quite minimal, as most of the useful processes require at least some capabilities that are considered dangerous. For example, a web server requires binding to a port 80, which would not be possible without extra capabilities. Docker fully supports adding and removing capabilities for the processes. It is advised to remove every capability that the service does not require. With careful capability and permission planning, risks associated with containers can be minimized.

By default, most container managers launch the containers in unprivileged mode. Unprivileged containers are running with limited capabilities to minimize the risks to the host system. An unprivileged container uses so called whitelist approach and drops all but the required Linux kernel capabilities [90]. For example, processes in the unprivileged containers cannot access host's devices by default. All mount operations are disabled as well. However, to work properly, containers still require access to certain devices of the host. To mitigate potential risks related to that, devices can be mounted as read-only.

Some host directories, such as /sys/ and /proc/sys/ are required for the container functionality. To minimize the security hazards, Docker can prevent writing on them as well. Unprivileged containers disable writing capabilities to these risky directories by default. Mount operations on these locations are also disabled by default [47]. Unprivileged containers also prevent access to the Docker daemon, because otherwise, an user could launch an additional, fully privileged container.

Privileged containers give its processes same capabilities as if the processes were running directly on host. Processes in the privileged containers are able to access all the devices on the host, use mount operations and access for example SELinux and AppArmor settings. Privileged containers should not be used in real life products, because one vulnerability in these containers could compromise the whole system. An unprivileged container is a more secure option, and it is suggested to use them whenever possible. If system admin wants to offer containers where user is able to log in or otherwise send commands to, container must be always running in unprivileged mode.

6.5.2 Using host's files in containers

Users are able to share host directories to containers on container launch. By default, processes in container can interact with the shared directory without any restrictions.

All modifications to the shared directory in container are also affecting that directory on the host, and thus the sharing must be planned so that serious harm cannot be done. For example, system critical or risky directories should not be shared to containers at all. However, if they are absolutely needed, a better approach would be to copy them to a container image, so that the possible modifications would not affect the files on host. Similarly, mounting of host's folders should be limited to separate directories that are created for that purpose.

Regardless of whether an unprivileged or a privileged container is used, processes inside a Docker container are running as root by default. This might lead to some security issues, for example if a system admin accidentally shares confidential root protected files from the host to a container. This means that any process, that is running as root inside the container, is also able to access these shared objects. To address this problem, Docker version 1.10 enabled user namespaces. By using them, it is possible to run processes inside a container as a less-privileged user of the host. Processes that are running with less-privileged user are not able to access root protected files inside the container.

Sharing of these critical directories is possible, because Docker daemon has to always run with root privileges. In practice, nothing prevents users of Docker daemon from sharing the whole root directory of the host to a container. If user is able to access Docker daemon, meaning that he is able to launch new containers, he is at the same time able to mount everything to these new containers, including for example /etc/shadow that contains encrypted passwords. He is then able to access them through the container, even though the user would not normally have root privileges on host system. In practice, this means that user accounts, which belong to the Docker group, should be also considered as root users of the system. Because of this, access to the Docker daemon should be restricted to the system admins only. Also, when designing a service with user input, careful parameter checking must performed to make sure that no malicious commands or parameters can cause creation of new containers. This is dangerous because the newly created container could be running with more privileges than initially intended, making it possible for the attackers to access the system.

6.5.3 Docker vulnerabilities

Because containers are using namespaces, they cannot see or interact with anything outside their namespace. However, primary security issue with containers are system-calls that are not tied to any namespace by their nature. These could be abused and used to break user free from the container. One example of this was published in June 2014 [91]. In this published exploit, attacker was able to access any file in the host file system. This certain exploit was found in Docker version 0.11 and is since patched in Docker version 1.0 and forward. However, new exploits for breaking out of containers have been found since [92]. This clearly shows that breaking out of a container is a serious risk that has to be taken into account when containers are used in production. Even though this particular exploit was fixed, there might still be similar exploits where vulnerabilities of a host kernel are used to break out from the container.

Extra care must be taken when downloading Docker images from Docker public repository called Docker Hub. It is possible that malicious image is uploaded to the Docker Hub or delivered to an unsuspecting system admin by other means. For example, even some of the most popular images in Docker Hub are containing major security vulnerabilities [93]. Other problem with pulling Docker images is that the pull command itself can be dangerous. Pulling an image using *docker pull* command also automatically unpacks the image and malformed packages might compromise the system [94]. Safer approach is to download the image manually and add it to Docker by using *docker load* command.

6.5.4 Enhancing security of containers

Security in container systems can be increased by using additional security features, such as AppArmor, seccomp and SELinux, that are offered by Linux kernel [47]. SELinux uses labels to enhance security. Processes inside a container are assigned a label, and can only access files that are labeled accordingly. Docker has also taken some actions to prevent malicious activities against containers. For example, API endpoint now uses Unix-sockets instead of TCP sockets bound on localhost. Because the API can also be exposed over HTTP, admins need to carefully configure it so that the API is only reachable from an internal network or for example by using virtual private network.

Many bugs related to namespaces still exist in Linux system call API. Security hazards can be reduced by white-listing some harmless system calls and denying the rest. However, the trade-off will then be the functionality of the containerized application. If specific functionality can only be gained with unsafe system-calls, those applications are not suited to run in containers and should be run in a more secure environment such as a hypervisor-based VM [47]. In general, it is advised to handle containerized applications as if they would be running without containers. Virtual machines are still the suggested approach for applications that require stronger security.

6.6 Future work

The practical implementation in this thesis focused mostly on the actual application services and their containerization. However, the example application also contains multiple platform services, that are required for the application functionality. The next logical step towards fully containerized application would be to move these platform services to containers as well. The goal would be to have everything running in containers, even if the host's network was still used to make this implementation easier.

After both the application services and the platform services are running in containers, a separate network for the containers should be implemented. For this, a research on different networking tools, such as Flannel, Calico and Nuage VSP, should be conducted. When separate network is used for the containers, a service discovery framework is also required, so that the services are able to find each other. Different service discovery frameworks, such as Consul, Eureka and Zookeeper, should be researched and implemented with the networking. A separate network and a service discovery framework will make containerized application less dependent on the hosts. A separate proof-of-concept, where containers are running directly on hosts without virtual machines, could be also done after this step.

In the current implementation, all of the container images contained overhead, that was a result from using Fedora base image. Most of the tools, that are provided by the base image are not needed for running the processes. More lightweight base images, such as busybox and alpine, could be used to create smaller images. Using these lighter images might require some changes to the application processes. However, using lighter container images is a necessity for the efficient software delivery and deployment, and it should be researched.

Comprehensive performance measurements should be conducted for the containerized application before introducing containers into the production. It should be measured, how the containers affect latency of the application and how do different storage drivers affect the performance. Also, a separate research on container security must be conducted before using them in production, especially if containers would be running directly on top of host.

Finally, a big topic that was not discussed in this thesis is continuous deployment and delivery. A research on different delivery methods should be conducted. Also, the usage of different deployment models, such as testing a new version in parallel to the running deployment, needs more research.

7 Conclusions

Because mobile data amounts keep growing also in the future, more effective virtualization methods are needed for the telco applications. In this thesis, Linux containers were introduced and proposed to be one possible solution for the problem. Containerization technology was compared to the traditional virtualization and benefits and disadvantages were discussed. Also, a popular container manager Docker was introduced.

This thesis also researched an idea of distributed telco applications, which would be built using microservices that are running in containers. A practical proof-of-concept was conducted, where functional parts of an existing telco application were moved in containers. Also, the requirements for this kind of architecture, such as a container orchestrator, were studied. The practical implementation was then evaluated in regard of software update, scaling, failure handling and security.

Benefits of containers are clear, for example in the software update, when comparing to the previous method. In the old architecture, even the smallest updates to some service require compiling of new virtual machine image. On top of that, the deployment process of a large virtual machine contains a lot of overhead and the startup can even take minutes. Another benefit of containers is their fast startup time. It enables extremely fast scaling that is required in environments, where traffic amounts vary a lot during a small time window.

Failure handling of containerized services can be done in multiple different ways, such as with a container manager, container orchestrator or by a supervising process. The biggest change compared to the previous architecture is that the containerized services and processes are not managed by host supervisor, such as systemd.

Because containers are sharing the host kernel, some additional security hazards might exist. These are often related to kernel capabilities, or shared directories and devices between the host and the container. That is why careful planning is required, and only the required capabilities should be given to containers. In the past, there has been a couple incidents where user has been able to break out from the container by exploiting a kernel vulnerability. The known vulnerabilities have been fixed, but many unknown exploits might still exist. Because of this, containers are not yet suitable for applications or environments where strict isolation is required.

Transforming existing monolithic applications to microservice-based containerized applications might require huge amounts of work. This includes re-architecturing and developing, and might even require changing of organizational structures. When investigating if old monolithic application should be transformed, careful analysis must be conducted to determine if the gained benefits overcome the amount of required work. However, when new applications are developed, it should be considered that the application could be running in containers in the future. That is why a new application should be designed so that it can be run in containers as it is, or at least so that the transformation process is easy.

References

[1] *Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2015–2020 White Paper.* `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html`. Accessed: 31.08.2016.

[2] Ming Mao and Marty Humphrey. "A performance study on the vm startup time in the cloud". In: *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on.* IEEE. 2012, pp. 423–430.

[3] Aleksandra Checko et al. "Cloud RAN for mobile networks—A technology overview". In: *IEEE Communications surveys & tutorials* 17.1 (2015), pp. 405–426.

[4] Kimio Watanabe and Mamoru Machida. "Outdoor lte infrastructure equipment (enodeb)". In: *Fujitsu Sci. Tech. J* 48.1 (2012), pp. 27–32.

John S Robin and Cynthia E Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. Tech. rep. DTIC Document, 2000.

[5] Nokia. *5G use cases and requirements, White Paper 2016.* `http://resources.alcatel-lucent.com/asset/200010`. Accessed: 10.04.2017.

[6] Global mobile Suppliers Association. *The Road to 5G: Drivers, Applications, Requirements and Technical Development.* `http://www.huawei.com/minisite/5g/img/GSA_the_Road_to_5G.pdf`. Accessed: 15.12.2016.

[7] China Mobile. "C-RAN: the road towards green RAN". In: *White Paper, ver* 2 (2011).

[8] Nectarios Koziris. "Fifty years of evolution in virtualization technologies: from the first IBM machines to modern hyperconverged infrastructures". In: *Proceedings of the 19th Panhellenic Conference on Informatics.* ACM. 2015, pp. 3–4.

[9] Chee Shin Yeo et al. "Utility computing and global grids". In: *arXiv preprint cs/0605056* (2006).

[10] John S Robin and Cynthia E Irvine. *Analysis of the Intel Pentium's ability to support a secure virtual machine monitor.* Tech. rep. DTIC Document, 2000.

[11] Paul Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS Operating Systems Review.* Vol. 37. 5. ACM. 2003, pp. 164–177.

[12] Shannon Meier et al. "IBM Systems Virtualization: Servers, Storage, and Software". In: *IBM Redbooks* (2008).

[13] Peng Li. "Centralized and decentralized lab approaches based on different virtualization models". In: *Journal of Computing Sciences in Colleges* 26.2 (2010), pp. 263–269.

[14] L YamunaDevi et al. "Security in virtual machine live migration for kvm". In: *Process Automation, Control and Computing (PACC), 2011 International Conference on.* IEEE. 2011, pp. 1–6.

[15] *The Xen Project.* `https://www.xenproject.org/`. Accessed: 22.11.2016.

[16] *Kernel Virtual Machine*. http://www.linux-kvm.org/page/Main_Page. Accessed: 22.11.2016.

[17] *vSphere ESXi Bare-Metal Hypervisor*. http://www.vmware.com/products/esxi-and-esx.html. Accessed: 22.11.2016.

[18] *QEMU - Open Source Processor Emulator*. http://wiki.qemu.org/Main_Page. Accessed: 22.11.2016.

[19] *VMWare Workstation*. http://www.vmware.com/products/workstation.html. Accessed: 22.11.2016.

[20] *Oracle VM Virtualbox*. https://www.virtualbox.org/. Accessed: 22.11.2016.

[21] Stephen Soltesz et al. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 275–287.

[22] *Chroot - Linux Programmer's Manual*. http://man7.org/linux/man-pages/man2/chroot.2.html. Accessed: 02.11.2016.

[23] *Linux-VServer Homepage*. http://www.linux-vserver.org/. Accessed: 30.11.2016.

[24] *OpenVZ Homepage*. https://openvz.org/. Accessed: 30.11.2016.

[25] Poul-Henning Kamp and Robert NM Watson. "Jails: Confining the omnipotent root". In: *Proceedings of the 2nd International SANE Conference*. Vol. 43. 2000, p. 116.

[26] Daniel Price and Andrew Tucker. "Solaris Zones: Operating System Support for Consolidating Commercial Workloads." In: *LISA*. Vol. 4. 2004, pp. 241–254.

[27] Miguel G Xavier et al. "Performance evaluation of container-based virtualization for high performance computing environments". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE. 2013, pp. 233–240.

[28] *Seccomp security profiles for Docker*. https://docs.docker.com/engine/security/seccomp/. Accessed: 13.03.2017.

[29] Wes Felter et al. "An updated performance comparison of virtual machines and linux containers". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE. 2015, pp. 171–172.

[30] Theo Combe, Antony Martin, and Roberto Di Pietro. "To Docker or Not to Docker: A Security Perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[31] *How to break out of a chroot() jail*. http://www.unixwiz.net/techtips/mirror/chroot-break.html. Accessed: 23.11.2016.

[32] *LXC Homepage*. https://linuxcontainers.org/lxc/introduction/. Accessed: 16.12.2016.

[33] Rami Rosen. “Linux containers and the future cloud”. In: *Linux J* 2014.240 (2014).

[34] Eric W Biederman and Linux Networx. “Multiple instances of the global linux namespaces”. In: *Proceedings of the Linux Symposium*. Vol. 1. Citeseer. 2006, pp. 101–112.

[35] Yvan Royon and Stéphane Frénot. “A Survey of Unix Init Schemes”. In: *arXiv preprint arXiv:0706.2748* (2007).

[36] *systemd - Linux man-pages*. `http://man7.org/linux/man-pages/man1/init.1.html`. Accessed: 03.05.2017.

[37] *Upstart homepage*. `http://upstart.ubuntu.com/`. Accessed: 03.05.2017.

[38] *cgroups, Linux Control Groups*. `https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt`. Accessed: 13.12.2016.

[39] Paul B Menage. “Adding generic process containers to the linux kernel”. In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 45–57.

[40] *Cgroup v2 Documentation*. `https://github.com/torvalds/linux/blob/master/Documentation/cgroup-v2.txt`. Accessed: 23.11.2016.

[41] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. “Containers checkpointing and live migration”. In: *Proceedings of the Linux Symposium*. Vol. 2. 2008, pp. 85–90.

[42] Roberto Morabito, Jimmy Kjällman, and Miika Komu. “Hypervisors vs. lightweight virtualization: a performance comparison”. In: *Cloud Engineering (IC2E), 2015 IEEE International Conference on*. IEEE. 2015, pp. 386–393.

[43] Fabio Kung. *Memory inside Linux containers*. `https://fabiokung.com/2014/03/13/memory-inside-linux-containers/`. Accessed: 14.03.2017.

[44] Marcus K Weldon. *The future X network: a Bell Labs perspective*. Crc Press, 2016.

[45] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux Journal* 2014.239 (2014), p. 2.

[46] Moritz Raho et al. “Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing”. In: *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in*. IEEE. 2015, pp. 1–8.

[47] Thanh Bui. “Analysis of docker security”. In: *arXiv preprint arXiv:1501.02967* (2015).

[48] *Supervisor: A Process Control System*. `http://supervisord.org/`. Accessed: 14.03.2017.

[49] *syslogd - Linux man-pages*. `https://linux.die.net/man/8/syslogd`. Accessed: 06.03.2017.

[50] *Systemd-nspawn man page*. `http://man7.org/linux/man-pages/man1/systemd-nspawn.1.html`. Accessed: 16.12.2016.

[51] ClusterHQ and DevOps.com. *The current state of container usage. Identifying and eliminating barriers to adoption.* `https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2015.pdf`. Accessed: 08.03.2017.

[52] *Use Supervisor with Docker - Docker documentation.* `https://docs.docker.com/engine/admin/using_supervisord/`. Accessed: 14.03.2017.

[53] *Docker storage drivers - Docker documentation.* `https://docs.docker.com/engine/userguide/storagedriver/`. Accessed: 16.12.2016.

[54] *Docker: Understand images, containers, and storage drivers.* `https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/`. Accessed: 16.12.2016.

[55] *Docker Registry.* `https://docs.docker.com/registry/`. Accessed: 16.12.2016.

[56] *Best practices for writing Dockerfiles - Docker documentation.* `https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/`. Accessed: 03.03.2017.

[57] *Kubernetes Documentation - Horizontal Pod Autoscaling.* `https://kubernetes.io/docs/user-guide/horizontal-pod-autoscaling/`. Accessed: 27.02.2017.

[58] Brendan Burns et al. "Borg, omega, and kubernetes". In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[59] *From Google to the world: the Kubernetes origin story.* `https://cloudplatform.googleblog.com/2016/07/from-Google-to-the-world-the-Kubernetes-origin-story.html`. Accessed: 27.02.2017.

[60] *Etcd Documentation.* `https://coreos.com/etcd/docs/latest/`. Accessed: 14.03.2017.

[61] *Flannel documentation.* `https://coreos.com/flannel/docs/latest/`. Accessed: 14.03.2017.

[62] *Openshift Homepage.* `https://www.openshift.com/container-platform/kubernetes.html`. Accessed: 04.05.2017.

[63] *Container orchestration: Moving from fleet to Kubernetes - CoreOS Blog.* `https://coreos.com/blog/migrating-from-fleet-to-kubernetes.html`. Accessed: 03.03.2017.

[64] *Docker Swarm.* `https://docs.docker.com/engine/swarm/`. Accessed: 28.02.2017.

[65] Rancher OS. *Kubernetes, Mesos, and Swarm: Comparing the Rancher Orchestration Engine Options.* `http://rancher.com/comparing-rancher-orchestration-engine-options/`. Accessed: 27.02.2017.

[66] Dmitry Namiot and Manfred Sneps-Sneppe. "On micro-services architecture". In: *International Journal of Open Information Technologies* 2.9 (2014).

[67] Martin Fowler and James Lewis. *Microservices.* `http://martinfowler.com/articles/microservices.html`. Accessed: 22.02.2017.

[68] Claus Pahl. “Containerisation and the PaaS cloud”. In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[69] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture”. In: *IEEE Software* 33.3 (2016), pp. 42–52.

[70] *Introduction to microservices on Azure - Microsoft Docs.* `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices`. Accessed: 08.05.2017.

[71] Chris Richardson. *Building Microservices: Inter-Process Communication in a Microservices Architecture.* `https://www.nginx.com/blog/building-microservices-inter-process-communication/`. Accessed: 16.03.2017.

[72] *Openstack homepage.* `https://www.openstack.org/`. Accessed: 04.05.2017.

[73] *Cloud Native Computing Foundation - Projects.* `https://www.cncf.io/projects`. Accessed: 23.02.2017.

[74] *Dockerfile reference.* `https://docs.docker.com/engine/reference/builder/`. Accessed: 14.03.2017.

[75] *Fedora Linux homepage.* `https://getfedora.org/`. Accessed: 17.03.2017.

[76] *HAProxy homepage.* `http://www.haproxy.org/`. Accessed: 06.03.2017.

[77] *ldd - Linux man-pages.* `http://man7.org/linux/man-pages/man1/ldd.1.html`. Accessed: 06.03.2017.

[78] *strace - Linux man-pages.* `http://man7.org/linux/man-pages/man1/strace.1.html`. Accessed: 06.03.2017.

[79] *PostgreSQL homepage.* `https://www.postgresql.org/`. Accessed: 06.03.2017.

[80] *rsyslogd - Linux man-pages.* `http://man7.org/linux/man-pages/man8/rsyslogd.8.html`. Accessed: 07.03.2017.

[81] *Consul homepage.* `https://www.consul.io/`. Accessed: 06.04.2017.

[82] *Eureka - Github.* `https://github.com/Netflix/eureka`. Accessed: 06.04.2017.

[83] *Apache Zookeeper Homepage.* `https://zookeeper.apache.org/`. Accessed: 06.04.2017.

[84] *systemd-journald - Linux man-pages.* `http://man7.org/linux/man-pages/man8/systemd-journald.service.8.html`. Accessed: 07.03.2017.

[85] *Configure logging drivers - Docker documentation.* `https://docs.docker.com/engine/admin/logging/overview/`. Accessed: 04.05.2017.

[86] *Logspout - Github.* `https://github.com/gliderlabs/logspout`. Accessed: 07.03.2017.

[87] *The Kubernetes Scheduler.* `https://github.com/kubernetes/community/blob/master/contributors/devel/scheduler.md`. Accessed: 15.03.2017.

[88] *Docker and the PID 1 zombie reaping problem - Phusion Blog.* `https://blog.phusion.nl/2015/01/20/docker-and-the-pid-1-zombie-reaping-problem/`. Accessed: 03.05.2017.

[89] Jerome H Saltzer and Michael D Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[90] *Docker security - Docker documentation.* `https://docs.docker.com/engine/security/security/`. Accessed: 03.11.2016.

[91] J. Turnbull. *Docker container Breakout Proof-of-Concept Exploit.* `https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/`. Accessed: 02.11.2016.

[92] National Vulnerability Database. *Vulnerability Summary for CVE-2014-9357.* `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9357`. Accessed: 16.01.2017.

[93] Jayanth Gummaraju, Tarun Desikan, and Yoshio Turner. *Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities.* Tech. rep. tech. rep., BanyanOps, 2015.

[94] *Before you initiate a "docker pull" - Red Hat Blog.* `https://access.redhat.com/blogs/766093/posts/1976473`. Accessed: 06.04.2017.