FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Continuous Delivery Strategy for Unikernel-Based Cloud Services

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Mathias Mahlknecht, Bsc.
Matrikelnummer 1115808

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dipl.-Ing. Clemens Lachner

Wien, 1. Dezember 2019

Mathias Mahlknecht | Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

A Continuous Delivery Strategy for Unikernel-Based Cloud Services

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Mathias Mahlknecht, Bsc.
Registration Number 1115808

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Dipl.-Ing. Clemens Lachner

Vienna, 1st December, 2019

Mathias Mahlknecht	Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Mathias Mahlknecht, Bsc.
Sechsschimmelgasse 18/11, 1090 Wien

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

Wien, 1. Dezember 2019

Mathias Mahlknecht

Acknowledgements

I would like to thank Prof. Schahram Dustdar for agreeing to supervise this diploma thesis.

I thank my parents for enabling me to pursue this study and for their patience and support throughout my whole school and university career.

A special thanks goes to Alexia, for her support and patience and for encouraging me to finish this study.

Finally, I would like to thank Clemens for his valuable input, feedback and encouragement, throughout the writing of this thesis.

Kurzfassung

In den letzten Jahren hat sich die Art wie Anwendungen entwickelt, deployed und gehostet werden stark verändert. Vor zehn Jahren waren die meisten Anwendungen noch monolithische Systeme. Sie liefen lokal oder auf selbst gehostet Servern und die Entwicklung umfasste teils Monate dauernde Release Zyklen. Moderne Anwendungen hingegen werden heute nach agilen Prinzipien entwickelt, wobei die Funktionalität in kleine unabhängige Services gekapselt wird, welche auf einer Cloud Infrastruktur ausgeführt werden. Automatisierung und Virtualisierung unterstützen die Entwicklung und das Deployment solcher Anwendungen. Container, z. B. *Docker*, bieten eine leichtgewichtige Alternative zu klassischen virtuellen Maschinen und sind heute die meist genutzte Technologie für Cloud basierte Softwarelösungen. Ihre Kompaktheit und kurze Startdauer sind ideal in Kombination mit modernen Automatisierungs- und Orchestrierungstools, zum hosten von verteilter Software, in der Cloud. In den letzten Jahren haben sich *Unikernel* als Alternative zu Containern und klassischen virtuellen Maschinen erwiesen. Diese basieren auf dem Konzept der Library-Betriebssysteme, kombiniert mit modernen Methoden der Virtualisierung, d. h. Unikernels vereinen die Vorteile von Containern und klassischen virtuellen Maschinen, durch starke Isolation gepaart mit einer leichtgewichtigen Implementierung. Während Container von Continuous Delivery Tools bereits gut unterstützt werden, ist das für Unikernels noch nicht der Fall. Es ist mit erheblich mehr Aufwand verbunden eine kohärente Continuous Delivery Pipeline zu erstellen, die Cloud Services als Unikernels liefert.

Diese Arbeit präsentiert eine Continuous Delivery Strategie für Cloud Services basierend auf Unikernel. Als Erstes werden Unikernels mit anderen Virtualisierungstechniken verglichen. Dabei werden Performance- und Sicherheitsfeatures, unter Zuhilfenahme von existierenden Publikationen, untersucht. Die Ergebnisse zeigen die Vorteile von Unikernels für Cloud Services und das Verlangen nach entsprechenden Tools, die den Übergang für Entwickler erleichtern, die diese leichtgewichtige und sichere Alternative nutzen möchten. Darauf folgend wird einen Continuous Delivery Pipeline, für auf Unikernels basierenden Cloud Services, ausgearbeitet. Dabei werden sowohl existierende, also auch neuentwickelte Tools verwendet. Abschließend wird die Verlässlichkeit von Unikernels im Vergleich zu Containern evaluiert, basierend auf zur Zeit zur Verfügung stehenden Tools zum deployen und betreiben von Cloud Services in Produktion.

Abstract

Over the last decade the development, deployment, and hosting process of applications changed drastically. Ten years ago most applications where monolithic systems, i.e., running locally or on premise, developed in long release cycles over months. Today, modern applications are developed in an agile fashion, providing functionality encapsulated in small services that run in the cloud. Automation and virtualization are the key enablers for the development and deployment of such applications. Containers, e.g., *Docker*, offer a light weight alternative to classical virtual machines and became the standard virtualization technique for cloud-based services. Small size and fast startup times, combined with modern automation and orchestration tools, outweigh drawbacks like laxer isolation. In recent years, a third alternative to containers and classical virtual machines, called *unikernels*, emerged. Based on the concept of library OS, combined with modern technological features of virtualization and hypervisors, unikernels offer the advantages of both, containers and classical virtual machines, by providing advanced isolation and a lightweight implementation. While continuous delivery tools provide good support to deploy cloud services as containers, the support for unikernels is still sparse and it is not trivial to build a coherent continuous delivery pipeline for cloud services packaged as unikernels.

This thesis presents a continuous delivery strategy for unikernel-based cloud services. First, a thoroughly comparison of unikernels and other virtualization techniques, mainly containers, is provided. Performance and security properties are examined, based on existing research and publications. This will show the advantages of unikernels for cloud services and the need for good tooling to ease the transition for developers, that want to take use of this lightweight and secure alternative. Second, a continuous delivery pipeline for unikernel-based cloud services is proposed and developed, by taking use of existing tools and the implementation of new ones. Finally, the feasibility of unikernels in contrast to containers is evaluated, based on the tools that are today available to reliable deploy and run cloud services in a production-like environment.

Contents

CHAPTER 1

Introduction

This thesis evaluates the unikernel concept in the context of cloud services, and provides a continuous delivery strategy for cloud services packaged as unikernels. In this chapter the motivation and problem statement will be described, fallowed by the methodology and the description of the used approach.

1.1 Motivation and Problem Statement

Continuous delivery is an important concept in agile projects, where a system rapidly changes and those changes have to be frequently deployed. The goal is to automate as much as possible of the delivery process. With a minimum of manual interference, the project gets build, tested, and shipped. This makes the process faster and replicable. Releasing a new version should be as easy as pressing a button [1].

In such projects, teams may rely heavily on virtualization together with advanced continuous delivery tools, e.g., Jenkins[1], to build, spin-up, take down, scale, and distribute their applications. Virtualization in general means, there is an additional software layer running on the host machine, which allows to run multiple guest operating systems (OS) on top of it. This additional layer is often called hypervisor. A hypervisor provides a set of virtual hardware, so that the guest OS, also called virtual machine (VM), only needs to have drivers for this specific set of virtual hardware. The VMs are isolated from each other and act as if they have exclusive access to the hardware. In terms of continuous delivery, this brings advantages, since it allows to spin-up a clean environment for each build and also makes this environments portable.

Containers gained increased industrial popularity lately, being a lightweight alternative to full fledged VMs. The key difference between containers and full-virtualization

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

hypervisors, is the level where isolation is implemented. Traditional hypervisors isolate on the hardware abstraction layer, whereas containers isolate on the system call/ABI layer. This leads to a trade-off between isolation and performance. While full-virtualization provides a better isolation between the VMs and makes it virtually impossible for the VMs to be aware of each others presence or even to interfere directly with each other, there is a significant overhead that impacts performance. Containers, on the other hand, have less strict isolation, but applications perform almost equally as on a physical machine [2].

Docker[2] is a well established, popular containerization tool and is supported by the majority of cloud providers. It has a well defined API and different types of configuration files, that makes it easy for a developer to define a new image, spin-up a container or configure a cluster of communicating containers, without deep knowledge of the underlying virtualization technology. It became the de facto standard for the industry, it is integrated into all the major cloud providers and other cloud orchestration solutions are build on top of it.

However, containers are not the only alternative to full fledged virtual machines. *Unikernels* are another upcoming technology, which gained increased popularity, although mostly in academic environments. The idea is based on *library OS*, which was already developed in the 1990ies [3, 4]. In recent years, major problems of library OS and its derivatives were solved, and this “second generation“ library OS was dubbed unikernel [5]. Various tools aim to make it as easy to build an application as a unikernel, as it is to containerize an application. But most of them are still in an early phase or support just a specific type of unikernel.

Although containers and unikernel share similarities, they differ in other aspects, and they are definitely not completely interchangeable. Containers still hold a complete operating system like Linux, even if it is stripped down. The application, deployed into a container, runs as one process alongside the other system processes inside the container. A unikernel, on the other hand, is operating system and application at the same time. This means applications, build as unikernel, can run without an operating system directly on the hypervisor (or even on bare metal) [5].

This brings advantages, e.g., on embedded systems, where the firmware can be implemented lightweight and updated by simply replacing the whole software as one. Another advantage, especially in the context of cloud services, is enhanced security. First of all a running unikernel can not be changed because everything up to configuration is done at build time. This means injection of malicious code, is virtually impossible. Additionally, only libraries that are needed for the applications, are also included into the build. This reduces the attack surface significantly [6]. However, there are also disadvantages, e.g., it is not possible to deploy two closely coupled applications together, like it is with containers [5].

One of the biggest advantages containers have over unikernels is, that it is currently still easier to build, ship, and run an application as a container than it is as a unikernel. This

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

is mainly due to the Docker project and all the other tools that build on top of it or make use of it. A developer does not need a deep understanding of operating systems and virtualization to run their application in a container. As previously mentioned, this is achieved by providing simple configuration files and taking use of well specified continuous integration (CI) tools and extensions for those tools.

Regarding unikernels, the integration of CI into the development process is still not a simple task. Deeper understanding of how unikernels actually work is needed, combined with a lot of additional manual steps that have to be taken to build and run an application. However, there are a ongoing projects that aim to facilitate this CI implementation process. One of the most promising ones is Unik [7].

Unik is a framework that builds applications into different kind of unikernel implementations and runs them on different platforms. The developer simply provides a configuration file and runs a command, specifying language, type of unikernel and targeted provider platform. Little knowledge is required about how the unikernel is build. The framework is open source and can easily be extended for new languages, unikernels and platforms. The project is still in a relative early phase, but the aim is to make the configuration and commands as similar to *Docker* as possible, so that the transition from container to unikernels is easier [7].

To make continuous delivery of unikernels as simple as it is for containers, a proper support of CI tools is needed. To the best of our knowledge there is currently no CI tool that provides support for unikernels, neither directly built in, nor via plugins. To build and ship unikernels with various CI tools, manual steps are necessary, resulting in time and maintenance overhead.

The purpose of this thesis is to compare unikernels against containerization and other virtualization techniques, in particular in the context of cloud-based services. Additionally to the analysis of existing research in the area, this will be achieved by proposing and evaluating a continuous delivery strategy for cloud services as unikernels.

1.2 Methodology and Approach

This thesis can be divided into four main parts. In the first part, the essentials are provided, which the reader needs to follow the rest of the thesis. A review of existing literature will show the different concepts and techniques used by an OS with respect to the various types of virtualization, containers and unikernels especially. The concepts of continuous delivery and microservices are described briefly. Additionally a presentation of the state of the art and of related work is provided.

In the second part a thorough comparison between unikernels and other virtualization techniques will be made. Since this is a broad field with a lot of different use cases, the focus will be put on the usage in the cloud. Existing literature provides results of experiments, where the different technologies are analyzed according to different metrics. Common metrics in this area are size, build time, startup time, security and performance. There is research that compares different container solutions with each other, container solutions with virtual machines and bare metal, but also container solutions with unikernels and different unikernel solutions with each other. This different results will be put into context and additional observations and conclusions will be made. The results will be divided into two chapters. The first one sets the focus on performance metrics, while the second mainly covers security concerns in a cloud environment. This evaluation will show the advantages of unikernels for cloud services and the need for good tooling to ease the transition for developers, that want to take use of this lightweight and secure alternative.

The third part will deal with the main contributions of the thesis. A continuous delivery strategy will be presented and a Jenkins plugin that eases the build and deployment of unikernels will be implemented and discussed. Obstacles or major risks encountered during the development process will be documented as well.

The development and evaluation of the results will be performed in three steps. First, existing tools for continuous delivery and building of unikernels are evaluated and selected. In order to integrate this tools with each other, extensions and customization will be developed. This will result in *JUnik*, a general *Java* client library for the *Unik* framework and *Unik Builder*, a *Jenkins* plugin that takes use of this library, to add simple build steps for unikernels to *Jenkins*.

Second, a continuous delivery strategy for cloud services as unikernels is proposed. This strategy will be used to develop a continuous delivery pipeline, by taking use of the previously selected and developed tools.

Finally, obstacles encountered during the development are described and the current state of unikernels and their tools evaluated. This will lead to an assessment of unikernels in contrast to other virtualization techniques, in the face of today's need of highly scalable and lightweight cloud services.

In the last section, the provided work is discussed and areas for future work identified. A conclusion will finalize the work and provide a summary.

CHAPTER 2

Fundamentals

Over the past years, server virtualization gained an increasing importance in deployment and hosting of applications on the web. Especially in distributed systems, virtualization techniques gained increased popularity. With the establishment of microservice based applications, an increasing number of companies are switching to agile development methods, where frequent and automated deployment is crucial. Tools to support and simplify this automated deployment on virtualized hosts are continuously developed and released.

In the following chapter fundamentals which are essential for the rest of this thesis are provided. First, various aspects of operating systems (OS) are introduced. Secondly, virtualization and respective tools are described. Third, the basic concepts of continuous integration and deployment are presented.

2.1 Operating Systems

Andrew S. Tanenbaum describes the purpose of Operating Systems (OS), from two views: The OS can be seen as "*an Extended Machine*" and its major task is "*to hide the hardware and present programs (and their programmers) with nice, clean, elegant, consistent, abstractions to work with instead*". But it can also be seen as "*a Resource Manager*", then its primary task would be "*to keep track of which programs are using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users*" [8].

However, the purpose of an OS is a combination of these and much more. In his book, *Modern Operating Systems* [8], Tanenbaum provides a detailed view on operating systems, their role and tasks and how they are implemented. Since this is beyond the scope of this thesis, only short descriptions of the specific features of operating systems are provided, which are necessary to understand the concepts of Virtualization and Containers as well as Unikernels.

2.1.1 Kernel

A kernel is the core of every OS. It is a program running in a privileged mode and all other programs rely on this program and its functionality for system critical task. The kernel manages the communication with the hardware and the communication between the different user processes. This way an abstraction layer for the user processes is provided and some protection of the system from malicious and faulty instructions is ensured.

One important task is the distribution of CPU time among the processes, i.e., it has to make sure every process gets access to the CPU eventually and no idle process blocks the CPU, without actually needing it. Another important task is the management of memory. The kernel provides functionality to allocate memory space for the processes and frees up memory if it is not needed any more, so a new process can use that space. In general, all system-critical operations, that potentially effect other parts of the system, need to be delegated to the kernel process. This is done via a special instruction set called system calls.

Depending on the type of kernel, also the drivers for different I/O devices are located here. Most modern OSs use one of the following two kernel types: a monolithic kernel or a micro kernel. Monolithic kernels include a wide range of functionality and perform all of them in privileged mode, which is a special permission level, reserved for security sensitive operations. Micro kernels on the other hand, try to externalize as much functionality as possible into the user space. Monolithic kernels consist of significantly more lines of code and are naturally more complex. This implies that they are also more error prone, e.g., a bug in a driver, which is included in the kernel and executed in privileged kernel mode, can cause a failure of the whole system. Micro kernels are build from a smaller code base since they contain only the functionality absolutely essential to the system, like process and memory management. The above mentioned bug would only cause the process of the driver to fail rather than the whole system [8].

Process

A process is a running instance of a program. Every time a program is executed at least one new process is started. It holds all the relevant information needed for the program to be executed. Every process operates in its own address space, i.e., a list of memory locations on which the process can read and write. Apart from the binary, being the executable program, the process contains the program's data and its stack. Additionally, associated with each process are a set of resources, e.g., registers, pointers, open files, and related processes.

Not all processes have the same privilege level. There are user processes and system processes. The system processes can only be started by the kernel and are allowed to perform system critical instructions, e.g., interacting directly with the file system and other I/O devices. The user processes have limited access to the resources and every time a process needs to communicate with another process or the hardware, it needs to call on the kernel, which in return delegates this operation to a system process [8].

Driver

Drivers are programs that control devices which are attached to the system. For every device and OS there is a specific driver. Drivers provide a software interface that enables the different programs of a system to interact with the hardware, without the need to know the different protocols to communicate with different devices. The program simply sends it requests to the software interface of the driver and the driver handles the communication with the hardware and returns the response to the calling program.

Drivers are usually not included directly in the kernel, since they commonly come from an external provider, like the manufacturer of the hardware. However, in many OSs, they are executed in privileged mode, which allows them to interact closer with the hardware. The downside of course, is the risk that an error in a driver could crash the whole system.

System Calls

System calls are a set of privileged instructions, that are only allowed in kernel mode. A normal user process is not allowed to execute this instruction directly, but it passes the control over to the kernel, which performs this task for it. This is because significant security issues arise, if user processes is allowed to directly access I/O devices or the file system.

Every time a user process wants to perform a privileged instruction, a system call is performed. This is done by placing the system call number in a dedicated register and executing a TRAP instruction, i.e., a special instruction that causes the process to stop and the kernel to take over control. The kernel will then look up the system call number and dispatch the requested system call handler. The system call handler is basically a specific procedure that performs a privileged task, like reading from a file. After the handler is finished the kernel passes the result to the process and returns control so that the process can continue with its next instruction [8].

For UNIX like systems, a standard called POSIX [1] exists. It defines interfaces for about a hundred systems calls and has the purpose to simplify the portability of applications between different systems.

CPU Rings

Most of modern CPU architectures comprise different privilege levels and only allow certain instructions on specific levels. By doing so, they make sure that a process can only

[1] https://standards.ieee.org/develop/wg/POSIX.html

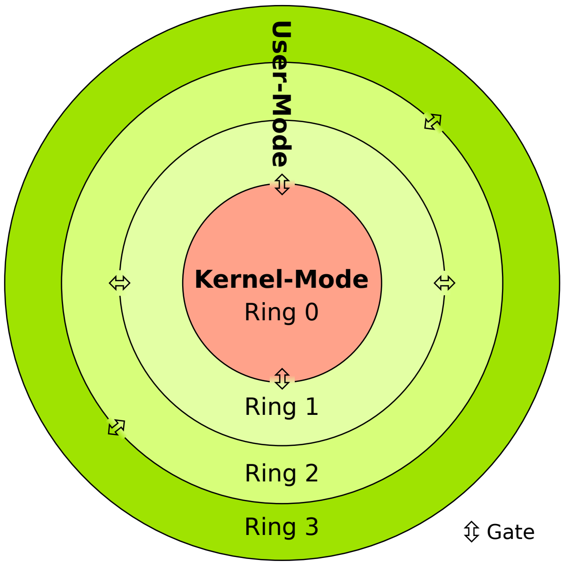


Figure 2.1: Scheme of CPU Rings of the x86 architecture [10]

run instructions on its own up to its designated privilege level. For all other instructions a process must delegate a task to another process with higher privilege level. Figure 2.1 illustrates the privilege scheme of the x86 architecture [9], a common CPU architectures. It consists of four privilege levels: Ring 0 to Ring 3, whereas Ring 0 is the most privileged level and Ring 3 the least one.

Most kernels only use two of these Rings: Ring 0 for kernel operations and Ring 3 for user application operations. User applications can only access the address space assigned to them directly, for everything else they have to call on the kernel to perform the task for them. For instance to open a file, print to the screen or allocate memory. This is because that could have an effect on an other application or the rest of the system. If an application tries to run a protected instruction outside of Ring 0, a general-protection exception is thrown [11].

2.2 Unikernel

The term *Unikernel* was first introduced by Madhavapeddy et al., where they define unikernels as "*single-purpose appliances that are compile-time specialized into standalone kernels, and sealed against modification when deployed to a cloud platform*" [5].

Although the term is new, the concept, also known as *Library OS*, is around since the late 1990s. Their first representatives were *Exokernel* [3] and *Nemesis* [4]. This two implementations had slightly different goals, than today's unikernels.

The *Exokernel* project wanted to provide an alternative to general purpose, monolithic operation systems. They argued that fixed high-level applications are hurting the application performance, because their is no single way to abstract physical resources that is best for all applications. General operation systems also hide information, so it is not possible for applications to implement their own resource management. New ideas or different approaches only get slowly integrated into this abstraction, which also limits functionality that would be possible otherwise. To overcome this issues the *Exokernel* project implemented a new minimalistic kernel, that manly focused on secure bindings, visible revocation and abort protocols. Due to this few simple functionalities this kernel can be made efficient and low-level multiplexing of resources can be provided with a minimum of overhead. Everything else, for instance traditional abstractions like page tables and events, are implemented as libraries that can be used in the libraryOS running on top of this kernel. This way every application can use the implementation that is best suited for their needs. In their publication Engler, Kaashoek and O'Toole Jr. [3], compared their *Exokernel* with a monolithic Unix operation system and were able to show significant performance increases in there benchmark tests.

The *Nemesis* project on the other hand, aimed to provide a specialized operation system for multimedia streams. They ended up with a libraryOS implementation, that had significant performance increases, for this specific tasks, compared with a general purpose operation system [4].

The first Library OSes had to deal with two major problems, namely resource isolation and device drivers. It is not a trivial task to run multiple applications side by side and still provide a strong resource isolation, to prevent communication between different instances. To keep up with the fast development of PC hardware and to provide a wide range of up-to-date drivers, is especially challenging for scientific projects, with limited resources. The rise of hypervisors and virtualization helped to overcome these challenges. Isolation can be ensured, simply by letting the hypervisor spawn a new VM for each application. Since hypervisors provide an abstraction layer to the hardware, by providing a set of supported virtual devices, the Library OSes only have to implement drivers for these virtual devices. [12].

Unikernels are structured differently than a conventional OS. A conventional OS, like Linux or Windows, is designed to support a wide range of tasks while running independently and allowing the parallel execution of different applications. A substantial set of drivers and services are integrated into the OS and might even run in the background, no matter if current applications need them or not. In a unikernel, on the other hand, all services are packed as libraries. These libraries are linked directly into an application at compile time and can not be modified afterwards. Therefore, after compilation, the result is a single purpose application, consisting of one process, that can run directly on a hypervisor, just like a regular OS. Its benefits are much less computational overhead and reduced security risks due to these specific reductions of unused services. It reduces the attack surface, by simply not including interfaces that are not needed by the application. By only including what the application needs, the computational overhead is reduced, since there are no unnecessary background process running.

Configuration of applications, on a traditional OS, is many times done via ad-hoc text

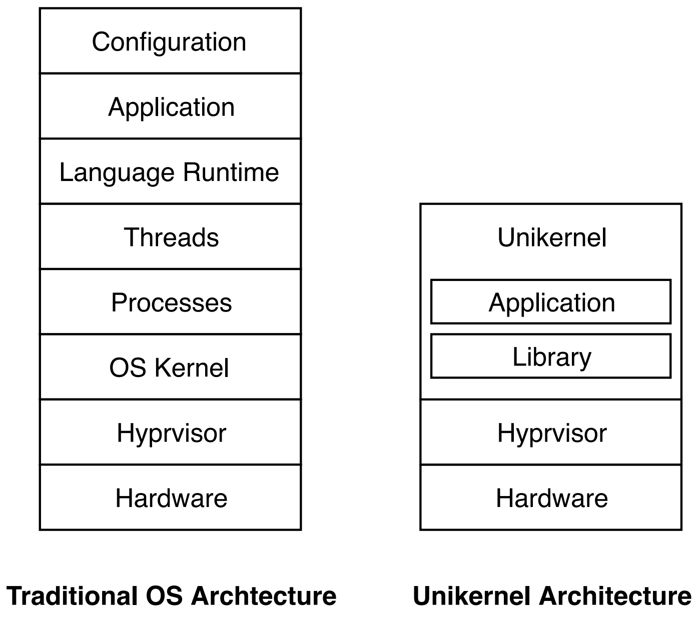


Figure 2.2: A unikernel architecture is able to reduce its size by removing many layers of the software stack, compared to a traditional OS. [13]

files, that are stored on the file system of the OS and need to be carefully maintained by operations personal. Regarding unikernels, all the configuration is done by the developer and can not be changed after compilation. Such configurations are implemented against specific dynamic or static APIs which are checked during build time.

Since a traditional OS, does not know which applications will be executed on it, it must therefore consider all executed code as potentially malicious. Concepts like userspaces and privilege levels are introduced to protect the core system from the users and the different processes from each other. Unikernels run just one application by design, with a designated purpose, in only one process. There is no need to distinguish between user commands and system commands, there are no different privilege levels and no costly context switches are required. Since application and OS are one, in the unikernel concept, there is no reason to protect the kernel from a failure of the application. If the application fails, the whole unikernel can fail. This supports unikernels in increasing performance and reducing their size. Figure 2.2 shows the differences between the software layers of a traditional OS and those of a unikernel.

Today many different unikernel implementations, with different scopes, exist.

The first group of unikernels focus on reusability and support of already existing applications. This is realized by implementing the POSIX standard, which is also implemented by the Linux kernel. An application implemented for a standard Linux OS, by taking use of the POSIX interface, can be executed on a unikernel. Members of this group are *OSv* or *Rumprun*. The second group focuses more on type safety by using a high level language. This unikernels reimplement all standard protocols in this same high level language, optimize those protocols for unikernels and add additional special APIs. They justify this sacrification of backwards compatibility and portability with performance and safety improvements. This unikernels usually also require their applications to be implemented in the same type save language. A member of this group is *MirageOS*.

The following sections will introduce some well established unikernel implementations.

2.2.1 MirageOS

MirageOS [5] is the first of this new generation of Library OSes and defined the term *Unikernel* as *a Library OS that is specifically designed to run in the cloud*. It is fully implemented in the type save, high level language *OCaml*[2] and applications must be developed specifically for this unikernel in the same language.

The selection of *OCaml* as the only supported language of course has some draw backs. It is still a relatively esoteric language and it imposed a significant engineering effort to reimplement the system components like the network or storage stack. But the *OCaml* runtime's fast performance for sequential executions, makes it well suited for unikernels, since unikernels by design run in a single process on one vCPU. *OCaml* is also a type-save language which adds to security.

MirageOS currently only runs on *Xen* hypervisor, but the support for other platforms is planed. Core elements of *Xen* are also implemented in *OCaml* which made the integration of *MirageOS* unikernels for this hypervisor easier.

The scheduling and thread logic is fully contained in application libraries, which allows the developer of an application to modify it as it fits best for their application. Also the device drivers are fully implemented in *OCaml*. Here we find an example where MirageOS does not follow the POSIX standard. The POSIX API does not support zero-copy sockets, this means there is always a second copy from the VM's kernel to the userspace process. Since *MirageOS* does not distinguish between kernel and user space, this is not necessary and received pages can be passed directly to the application without copying. This gives MirageOS a performance advantage.

Since *MirageOS*, as a unikernel, runs only one application and thus does not have to distinguish between kernel and user space, it gives the developer much more control over lower level OS functionality, than for example a standard Linux VM would. Which has significant performance benefits. For instance, filesystem and caching are provided as *OCaml* libraries, which gives control to the application over the caching policy instead of just providing one default implementation.

In the benchmarks Madhavapeddy ed al. show that MirageOS has the same boot time as a minimal Linux kernel and slightly less than half of the boot time of the Debian Linux. This is on an unmodified *Xen* hypervisor. After modifying the *Xen* boot stack, to allow parallel domain construction, they reach a minimal boot time of under 50 milliseconds.

[2] `https://ocaml.org/`

2.2.2 OSv

OSv [14, 15, 16] is a unikernel implementation by a company named *Cloudius Systems*.

Similar to *MirageOS* it is especially developed to run on the cloud, but it provides support for multiple platforms, such as *Xen*, *VMWare* or *Virtual Box* as well as for cloud providers like *Amazon EC2* and *Google GCE*.

Originally it was implemented to support the execution of existing *Java* applications as unikernels in the cloud, so it naturally provides a good support for this applications. By now support was added for other languages like *Golang*, *Python* or *Nodejs* as well as for *Linux* executables in general.

OSv is fully implemented in *C++*. Some OS components were reimplemented from scratch others are taken from open source projects like *FreeBSD*. In general *OSv* focuses on reusability, so that existing applications can easily be ported to run as a unikernel in the cloud. Additionally a non POSIX API was added, that *OSv* aware applications can use to increase their performance, for example a zero-copy, lock-free API for packet processing. All JVM languages benefit from this API natively, because *OSv*'s JVM implementation takes already use of it.

OSv aims to perform better than a standard *Linux*. Because of that, parts of the OS were reimplemented instead of fully relying on existing projects like *FreeBSD*. To achieve this, for instance system calls are linked to normal functions, since in a single address space environment no special handling is needed for them. Another improvement is to get rid of all spin-locks, for example by using lock-free algorithms, per-cpu run queues and threading.

On of the biggest modifications is a high quality TCP/IP stack, since network communication is important for cloud native applications. This is done by using Van Jacobson's net channel ideas [17]. Here for each TCP or UDP connection a channel is created, which is a single producer/single consumer queue. This significantly reduces the amount of locks needed and in return increases performance.

Apart from the kernel it self, *OSv* provides additional tools, that aim to make it easier to build, operate and maintain *OSv* kernels in the cloud.

With *Capstan* a build tools is provided, where the unikernel can be configured, similar to a docker container, with a simple configuration file. This makes it much easier and less error-prone, than writing a Makefile and build scripts directly.

Similar to the `Capstanfile` a `cloud-init` file can be added, that allows to define a set of task that are performed on startup of the unikernel.

An other feature is a REST API, that can be added to the kernel on build time, which allows remote monitoring of the unikernel but also to send commands, like restarting the application. This api can be accessed directly, via command line, or via a web client.

OSv, with its ecosystem, is one of the most advanced unikernel projects. It allows to execute every application as a unikernel, that can be executed on a standard *Linux* and on the same time provides a better performance. With its tools it is also relatively easy to build, run and scale unikernels as a swarm in the cloud.

2.2.3 Rumprun

Rumprun unikernel [18, 19] is a unikernel implementation based on *rump kernels*. It aims to run all applications developed for a POSIX environment as a unikernel.

The first purpose for the *rump kernels* was it to make it easier to develop and test drivers for the *NetBSD* kernel in userspace. This means the rump kernels bring along everything required to run a kernel driver. From there it went to not only make kernel driver work in userspace and in the *NetBSD* kernel, but everywhere. This is resolved by the *Anykernel* and the *Hypercalls*.

The *Anykernel* is a an architectural concept that ensures there is no direct reference where there should not be one. For example it should be trivial to exchange the TCP/IP stack with out any dependencies to the rest of the kernel.

The *Hypercalls* is a layer that provides an interface for the drivers to back-end resources such as memory and I/O functions.

With this modifications it is possible to run drivers in any configurations, monolithic, microkernel or unikernel.

Rumprun is simply a wrapper to run the *rump kernels* as unikernels. It takes existing drivers from *rump kernels*, ads a libc and an application environment, and provides a toolchain which builds existing applications as a *rumprun* unikernel.

2.3 Virtualization

VMware, one of the leading companies providing solutions for virtualization, describes the technique as a "*separation of a service request from the underlying physical delivery of that service*" [20]. This means, that there is an additional software layer between the host machine and the guest OS. This software layer is called Hypervisor or Virtual Machine Monitor (VMM, see 2.3.2). The hypervisor is able to virtualize the hardware layer in a way, so that the different OSes, running on top of it (guest OS), can act like they have exclusive access to the hardware. In fact, they are not aware of the other guests. This allows to share specific hardware capacities and features between different services to make of use their full potential, but at the same time isolate the services from each other.

The first server virtualization comparable to modern techniques was developed by *IBM* in the 1970s as a way of time-sharing for their mainframe computers, specifically for their *System/370*. This virtual machines simulated the *System/370* architecture and worked exclusively on this mainframe [21].

At about the same time Popek and Goldberg defined formal requirements for virtual machines and formulated three essential characteristics, what they believed a virtual machine architecture must fulfill [22]:

1. "*Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly.*"
2. "*A statistically dominant subset of the virtual processor's instructions are executed directly by the real processor.*"
3. "*The VMM is in complete control of system resources.*"

Sticking to this characteristics proved to be difficult later on, because they were simply unsuited to certain tasks, or required expensive operations like additional context switches. Nowadays most existing hypervisors do not abide to those formal requirements that strictly anymore [23].

In the late 70s the rise of the personal computer led to a decrease of interest in virtualization techniques. Companies could now provide multiple PCs to their employees instead of one big mainframe they all had to share. Another reason for the stagnation was, that the most popular processor architectures were not designed to run multiple operating systems at once. An example of this is Intels *x86-32* architecture [9]. It was then believed to be impossible running virtualization software on this architectures. 1999 *VMware* released its product *VMware Virtual Platform*, which was able to run Virtual Machines on Intels *x86-32* architecture based processors [20].

Today many different hypervisors exist, both commercial and open source. Some examples are: *VMware*[3], *the Xen Project*[4] or *Denali*[5]. In addition most of modern processors provide hardware support for virtualization, which helps developers of different hypervisors to increase the performance of their products [24]. However, not all hypervisors work the same way and the virtualization techniques can be divided in different types.

2.3.1 Types of Virtualization

Virtualization techniques can be roughly divided into four types, although in practice many hypervisors use a mixture of different techniques to maximize their performance.

Operating System-level virtualization The virtualization is performed at operating system-level. The host OS has a modified kernel that allows the execution of multiple isolated Containers (see 2.4). There are many different kinds of this type of virtualization and their implementations are widely used. This technique shows a low performance overhead, but it does not support multiple kernels [24].

[3]https://www.vmware.com/
[4]https://www.xenproject.org/
[5]https://www.denaliai.com/

Para-virtualization For this technique the guest OS needs to be modified. A special set of instructions (Hypercalls) are added to replace the real machine's instruction sets. The advantages are a low performance overhead and most of modern processors implement hardware support for this technique, that allows the hypervisor to run between hardware level and guest OS level. [24].

Binary translation This technique is used to emulate a processor architecture which differs from the actual architecture, alongside with an emulated eligible instruction set through translation of code. The advantages here are the portability of OSes and application code, but the emulation causes a significant performance overhead. So in practice a combination of emulation and direct execution is used. Only a small set of instructions, those that need privileged execution, are emulated the rest are executed directly on the host CPU. This leads to a lower overhead, but limits the possible guest OSes to those that run on the host CPU [24].

Hardware assisted virtualization Here the hardware provides extensions that support virtualization. More specific, an additional Ring -1 (root mode) is introduced, in which the hypervisor can run, because of this the unmodified guest OS can run in Ring 0 and is still on a higher level than the hypervisor. This increases performance since a trap-and-emulate model can be performed on hardware level instead of software level [24]. The trap-and-emulate model describes a technique, where the processor stops and returns the control to the hypervisor, if the guest OS attempts to execute privileged instructions. This way the hypervisor can intercept and emulate the execution of this special instructions.

The last two types, *Binary translation* and *Hardware assisted virtualization*, can be counted as Full-Virtualization techniques, as defined by Popek and Goldberg, since the gust OS does not need modification and is not aware of the underlining hypervisor.

2.3.2 Hypervisor

The Hypervisor, also called Virtual Machine Monitor (VMM), is described by Whitaker et al., where they present their hypervisor *Denali*, as "*a software layer that virtualizes all of the resources of a physical machine, thereby defining and supporting the execution of multiple virtual machines (VMs)*" [25].

This software layer provides an interface between hardware and software, while virtualizing CPU, memory and I/O devices. This way multiple VMs, typically called guests, can run on the same machine, called host. The VMs are fully isolated from each other and behave like a physical machine. It can be distinguished between two types of hypervisors. Type I runs directly on hardware on the Ring of highest privilege and controls all VMs. Type II runs within an OS on the same Ring as the OS, and it relies on the OS for a specific task, so every VM is a process within the OS [24].

2.4 Container

Containers are a type of *Operating System-level virtualization* and gained much popularity in the last years. They can be counted as type II hypervisors, because they do not emulate any of the hardware, but instead talk to the underlying OS, that than performs calls to the hardware.

The key difference between containers and full-virtualization hypervisors, is the level where isolation is implemented. Traditional hypervisors isolate on the hardware abstraction layer, whereas containers isolate on the system call/ABI layer [6]. This leads to a trade-off between isolation and performance. While full-virtualization provides a better isolation between the VMs and makes it virtually impossible for the VMs to be aware of each others presence or to interfere directly with each other, there is a significant overhead that impacts performance. Containers, on the other hand, have less strict isolation, but applications perform almost equally as on a physical machine [2].

Since containers relay on the host OS to talk to the hardware, they normally support only one kernel, even if there are approaches for multi-kernel containers [7]. In order to run containers on a specific OS, modifications to the kernel are required. In the past this was seen as a disadvantage, but today it is not an issue any more, since several modifications are integrated into the official Linux kernel now and therefore all major Linux distributions support containerization out of the box [27].

Windows and *macOS* didn't have this support for containers for a long time. *Docker* started to add it, at least for the developer machines, by transparently spinning up a Linux VM and running the containers inside this VM. While this was sufficient for developers to test their containers, this was never thought to be used in production. Also, the containers were still Linux containers and no other OS image could be used inside the containers [28].

This changed a few years ago, when *Microsoft* started a partnership with *Docker* to develop containers that can run natively on *Windows*. Now there exist two types of Windows containers: First the standard Windows Server Container, that shares the Kernel with the Host OS, like it is also done on Linux. This has the disadvantage that the container OS and the Host OS have to be of the exact same version. The second solution are Windows Server Containers with *Hyber-V* isolation. *Hyber-V* is a hypervisor on which Windows Containers can run fully virtualized, which means they don't share the Kernel with the Host and thus can have different Kernel versions [29].

MacOS still does not have a native support for containers and since there is also no server solution from Apple, it is not a real issue.

Another feature provided by containers is explicit resource sharing between different VMs. While this violates the strict isolation rules for full-virtualization, it proves useful

[6]The ABI (Application Binary Interface) defines the low-level interface for a piece of software on a specific hardware. For instance how to interact with the kernel or a system library [26].

[7]`https://github.com/cloudfoundry-attic/warden`

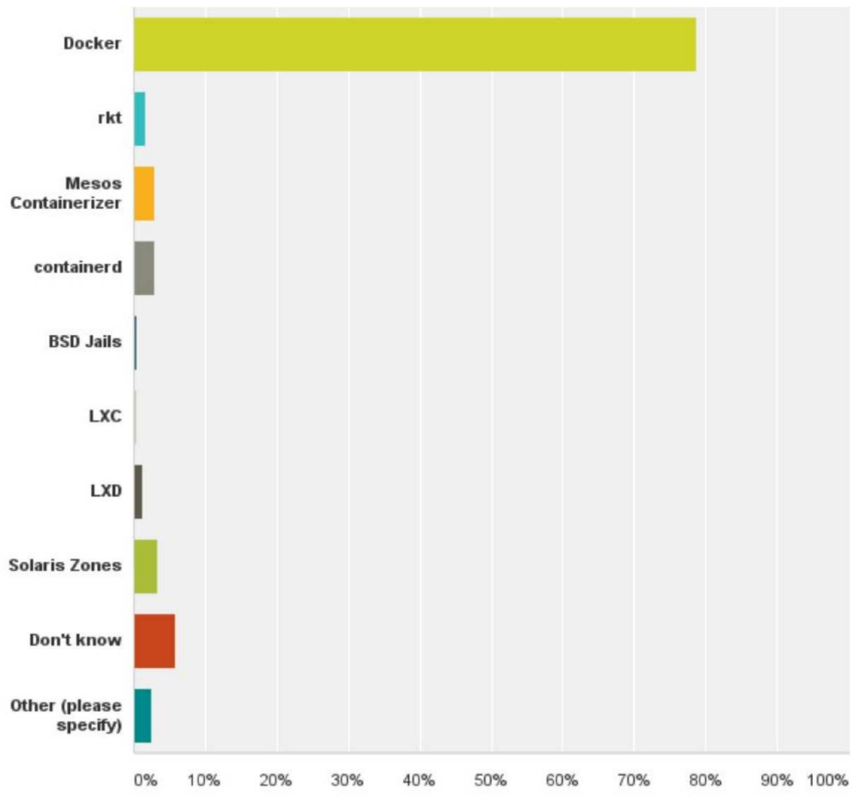


Figure 2.3: The market share of containerization technologies, based on a survey by Portworx from 2017 [30]

for certain tasks. For instance two applications that need to share resources can do so while still running each in their own VM. This way if one application causes a system failure in its VM the other one is only marginally affected by it.

The two most popular container implementations are *Linux Containers (LXC/LXD)* and *Docker*, while the latter builds on top of *LXC/LXD* and provides an abstraction-layer, which makes it easier to use, also without in-depth knowledge about how containers work under the hood. Especially *Docker* contributed to the popularity of containers, because it made them much easier to use. *Docker* is today the most used containerization technology as shown in figure 2.3, which shows the market share of different containerization technologies, based on a survey performed by Portworx in 2017.

2.4.1 Containers on Linux

Most container solutions, like *Docker*, run mainly on Linux. This is simply because the Linux kernel provides already all the tools the containers need to isolate their processes from each other. The two most important tools are *control groups* and *namespaces*.

While *control groups* limit how much resources a process can use, *namespaces* limit what a process can see. In the following we will describe the two concepts in more depth.

Control Groups

The control groups (cgroups) provide a mechanism to manage the distribution of the overall system resources transparently between different groups of processes. It forms a tree structure and every process in the system belongs to one node in a hierarchy, but multiple hierarchies are supported. The concept of cgroups is completely generic and every subsystem can hook into the cgroups to organize their processes [31].

Currently there exists a cgroups implementation for memory, CPU, block I/O and network. For each of this resources exists a hierarchy and for each node in the hierarchy can be specified how much of the resource can be used. Initially every process belongs to the root node, which gives it access to the whole resource, the further down in the hierarchy the process gets moved, the more restricted is his access. Additionally if a process gets forked from an other process, it automatically belongs to the same cgroups as its parent [32].

Cgroups prevents one or multiple processes to block all resources and affect thereby the performance of others, which is a key requirement for virtualization.

Namespaces

Namespaces are another way to ensure process isolation. It is a technique that wraps the global system resources in an abstraction, so that it appears to all processes that they have their own isolated instance of the system. In fact they only see the part of the system that is within there namespace [33, 32].

Currently there exist six different namespace implementations:

pid Processes within this namespace only see other processes in this namespace. Every namespace has its own PID numbering starting at 1 and this namespace can be nested, which results in on process having multiple PIDs.

net Every network namespace has its own network stack.

mnt Every mount namespace has its own root file system and even process private mounts are possible.

uts Every uts namespace has its own set of hostnames it can see.

ipc Every ipc namespace has its own semaphores, messages queues and shared memory.

user Allows to restrict certain UIDs or GIDs to their own namespace.

So each container can have its own namespaces, which provides isolation to the processes and is crucial for virtualisation.

2.4.2 Containers in Agile Projects

In agile projects containers gained high popularity and there are multiple reasons for that. The first reason is the relatively small size of a container image compared to the images of full virtualization systems, so the images can easily be build and pushed to a repository, from where they can be deployed to different servers. Running containers can also easily and transparently be migrated to another location. Second, if an application runs correctly in the container on a developer's machine, it most likely behaves the same way on the production server, since the environment for the application inside the container never changes. Also the whole delivery pipeline can be realized with containers, so that tests run in the same environment setting as the production server. Finally, due to their small overhead, every service can run its own container, which provides higher security. But on the same time, if services need to share files or communicate directly, they can still do so, even if they run in different containers.

2.5 Microservices

The concept of modularization is a major research topic in software engineering. Dividing an application into different components facilitates the whole software development process as well as maintenance tasks [34]. The individual components are loosely coupled, each concerned with a specific task and clear boundaries. Extracting common behavior and tasks into libraries that can be reused and build upon is a core principle in the world of software development. This allows multiple teams to work on the same application, with minimal crosscutting concerns.

Services are the next step in this process of modularization and decoupling. The term *microservices* has first been introduced at a workshop in 2011 [35], where participants agreed on this term as an architectural concept. Since then, microservices gained increased popularity and its architectural principles are used world wide by large and small companies to build their applications [35, 36, 37].

Still, microservice is a broad term and various definitions exist. In general, every application that splits its functionality into small, independently running services that communicate with each other to reach a desired outcome, can be considered to correspond to the architectural principle of microservices [36, 37]. Levis and Fowler [35] define microservices as "*an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API*".

A key feature of microservices is the *ease of deployment*. If a specific service has to undergo certain changes, this single service only has to be redeployed, while functionality of all other services of the system is maintained. This allows for faster and easier system updates, in contrast to a monolithic system, where the whole system has to be rebuild, and redeployed [35, 36, 37].

The downside is increased complexity, to manage the communication between the services and to maintain a consistent state throughout the whole system.

2.6 Continuous Integration/ Continuous Delivery

Continuous integration (CI) is a principle of agile development methods. The goal of CI is to continuously integrate the codebase of a project, meaning every time a change is performed. *Continuous delivery (CD)* takes this principles a step further. After the build passed the automated tests and validations of the delivery pipeline, the changes will be packaged as a release candidate. The release candidate often gets automatically deployed to a production like environment or even directly into production [1].

Today these two terms are often used interchangeable or simply referred to by the abbreviation *CI/CD*. To successfully implement a good CI/CD strategy the use of tools and extensive automation is necessary, but good policies and discipline in the whole team is at least as important.

Traditionally most of the tasks necessary to release new version are a sequence of manual steps that are performed by multiple roles that are often organized in different teams. The developers implement an application and when they consider a new version to be done, they pass it on to the testing team. The testing team extensively tests the state of the application. This can go trough multiple cycles until both, developers and testers are confident that the new version is finished and fulfills all requirements. Only now the new version is released and the operations personal will set it up in the production environment.

This causes the release process to take several days or even weeks. Additionally it divides the different roles and enforces them to only focus and optimize for their needs instead of what is best for the whole project [1].

With CI/CD this whole process is automated. This is done by taking use of existing tools and the heavy reliance on scripting. It must be possible to create a new release, with minimal to no human interference.

This can be done by setting up a build and release pipeline that is automatically triggered every time a change to the code base is performed. This pipeline consists of a set of scripts that are held in the same code base as the application and versioned together with the rest of the code. This enables the pipeline to evolve and change together with the application and still allows to build older version.

Automated tests are one of the most important steps of this automated pipeline. By a tight collaboration of testers and developers all tests are automated. Starting from unit tests for specific parts of the code, integration tests for the whole system and regression tests to detect unwanted side effects by changes to the code base. The more tests are performed the higher the confidence that a systems behavior is correct. Only a good test coverage provides the confidence to fully automated a release without human interference [1].

Not only the build scripts, release scripts and tests should be kept in version control, but everything that is needed to build, deploy, test and release an application. This also includes database creation and update scripts, the configuration for the target environments and the technical documentation. Every change set need to have an unique identifier, which allows to identify the version for every build. Further more it is visible which build is currently deployed to a specific environment and in the worst case it allows to roll back to a previous version [1].

Operations personal and developers are encouraged work closely together to guaranty a seamless deployment and well configured application and environment in production. This close collaboration and the extension of the responsibility of the developers up to the live time of an application, are also known as the *DevOps* principals.

When a fully automated build and release pipeline is in place, and the test coverage is high enough to have a good confidence about the integrity of the system, releasing a new version becomes unspectacular. This allows to increase the release cycles up to multiple times per day. Every commit could become a new release, if it passes all automated tests.

The team can focus on producing actual value, rather than spending much of their time, with manually testing and fixing the system and manually building and deploying the application.

CHAPTER 3

State of the Art and Related Work

This chapter provides information about the state of the art of orchestration and automated builds of unikernels, as well as information about related work in the area of unikernels in a cloud environment.

3.1 State of the Art

In this section we will present state of the art tools for orchestration and automated builds of unikernels. Automated builds are crucial for the continuous delivery of an application. They make the release of a new version painless and repeatable. This is also important for unikernels to have tools that add an abstraction over the build of new images, so that the developer does not need to be concerned with the deep technical details on how the image is created. The same is true for orchestration. Only after there is a good support from orchestration tools to automatically manage unikernels, their full potential can be used and they become a real option for the enterprise world.

3.1.1 Orchestration

With more applications being developed as distributed systems, running on a cloud platform, it becomes increasingly important to have reliable tools to manage and orchestrate all the independently running services.

Kubernetes [38] is on its way to become the de facto standard for orchestration in the cloud. It was first developed by *Google* and released as open source. Now it is managed by the *Cloud Foundry Organization*, an open source organization that drives the development of tools and standards for the cloud.

Kubernetes is not the only orchestration tool for cloud services. Other tools are *Misos* from *Apache* or *Swarm* by *Docker*. All of them aim to simplify the orchestration of containers across multiple hosts and they do this in a transparent way. The high configurability and extensability of *Kubernetes* made it popular and lead to most cloud providers integrating it into their system, which gave it another boost. The other tools are still used by many developers and companies, but it seams like *Kubernetes* won the race for the cloud platform of the future.

In order to really become the main platform for the whole cloud, it is important not be limited to containers. *Kubernetes* started off as orchestrator for *Docker*, but there was quickly added support for other container runtimes and even full blown virtual machines. With the last big refactoring, the runtime was extracted from the core and replaced by a clear interface. This way everyone can easily add their own runtime for what every systems they want to orchestrate. This interface was named *Container Runtime Interface (CRI)* [39]. Among diffident container solutions, there are currently also attempts to provide a hypervisor based runtime, which takes use of this interface. This allows to add VMs and in turn also unikernels as first class citizens to *Kubernetes*, just like containers.

The *Unik* project [7] was the fist that attempted to add support for unikernels to *Kubernetes* through their framework. This was before CRI was released and it is not easy to setup and based on a couple of workarounds, so not all features of *Kubernetes* could be used, like it is possible with containers.

Integrating this two tools trough the CRI interface, benefits both tools and immediately provide a powerful orchestration support to a wide range of unikernels, which as of now does not exist.

3.1.2 Automated Builds

In order to build unikernels in an automated and reliable way, special tools are needed. This tools automatically select the needed OS libraries to run an application and package the application as bootable image.

There are currently different tools available that perform this tasks. In fact have most of the unikernel implementations scripts in place, that help to create an image. Nevertheless, at the moment there are only a few tools available that fully automate this process.

Currently the research goes in two directions. One option is to relay on the well established Linux Kernel, but create an extremely striped down version of it, to build lightweight VMs. The second option is to rely on specific library OS to build VM images. While both result in much lighter VMs, than by using of traditional OS images, only the second can be considered as real unikernel as they were defined by Madhavapeddy ed al. [5].

In the following representatives of both categories are presented.

LinuxKit [40] is one tool that is able to package an application as a lightweight bootable image. It originally comes form the *Docker* ecosystem and was designed to build lightweight images for containers. Over the time it was generalized and it is now possible to run the images not only as containers, but on different hypervisors es well as on bare metal.

The images created by this tool, resemble closely to unikernels. They have similar attributes like immutability, small size and fast boot time.

The approach how ever is different. While unikernels are general constructed by selecting a set of OS libraries that are needed by a specific application, *LinuxKit* takes the Linux OS and strips away as much as possible.

So it is arguable if *LinuxKit* can be counted as real unikernel build tool.

Another tool that falls into the same category of *LinuxKit* is *tynix*. It was developed by Filipe Manco et al. and presented in their publication about extremely lightweight VMs [41].

The tool packages applications as extremely lightweight VMs, which include only the application it self, dependencies of the application and a BusyBox to support basic functionality.

Although the resulting images resemble unikernels, they again take use of the standard Linux OS and don't use a real library OS to build the images.

Capstan [42] on the other hand, is a build tool designed to package applications as unikernels, specifically as *OSv* unikernel. Its usage is closely related to *Docker*. This is on purpose, so that the transition from containers to unikernels, or more specific, the transition from *Docker* to *Osv*, is as easy as possible.

The tool provides a CLI interface to build and run unikernels. Additionally a so called *Capstanfile*, equivalent to the *Dockerfile*, can be used to specify and persist the configuration for an image. New images can be build by using this configuration file or existing images can be pulled from *Github* or an *S3* storage.

Capstan is on of the more mature build tools, but its biggest drawback is that it only allows for building *OSv* images. If developers wants to use one of the many other unikernel solutions, they still have to rely on a partially manual process or fined other tools that fulfill their needs.

Unik [7] has the potential to be one of this tools. It takes the same approach as *Capstan*, by making CLI and configuration files similar to *Docker*. However, the configuration options are still sparse and the whole project is still in an early phase and by no means production ready. Nevertheless, it has big potential to become the leading tool for building and running unikernels.

It already supports a wide range of unikernels, programming languages and target platforms. The project is open source and the architecture is modular in a way, that quite simply support for additional unikernels or target platforms can be added, without touching the rest of the system.

The different approaches have all their advantages and disadvantages. It will be seen which of the them will break trough in the future. Be it lightweight VMs, still relaying on the well established Linux kernel, a specific unikernel implementation or a more general build tool, that allows to target many different unikernel solution.

3.2 Related Work

After the concept of unikernels was proposed and the first implementations were presented. Researchers quickly started to investigate how unikernels can be leveraged for cloud computing and how to optimize existing solutions to optimize them further for the cloud.

The following related work are examples of research that tries to optimize the unikernels them self or their infrastructure, as well as taking use of the unikernel concept to optimize cloud services.

In his thesis extended Maghsoud Chinibolagh [43] the unikernel *IncludeOS* with multicore processing functionality. In general are unikernels by design single process and singe core. But by extending an established unikernel implementation with multicore processing functionality, Chinibolagh was able to show a good performance improvement compared to a standard *Ubuntu* running on multiple cores. Additionally it was more resource efficient than running multiple single core instances in parallel.

This thesis shows an interesting approach to take the most leverage of the high-end multi core CPUs of the cloud servers, by extending and adapting the idea of how a unikernel operates.

Dan Williams and Ricardo Koller proposed to extend the minimal implementation of a unikernel, to the hypervisor the unikernel is running on. In their article ”Unikernel Monitors: Extending Minimalism Outside of the Box” [44], they present their idea of a *Unikernel Monitor*. One key benefit of unikernels is the reduction of attack surface, by only including system libraries that are needed by the application. However the hypervisor the unikernels are running on, is still general purpose. Thus it includes a lot of functionality that are not needed by the application. This functionalities unnecessarily occupy resources and are potential sources for vulnerabilities.

In Their proposal they suggest to extend the build mechanism of the unikernel, which automatically includes dependent system libraries. The extensions additionally builds the hypervisor and only includes dependent hypervisor libraries. The result is a lightweight hypervisor, which in the extreme case only has the ability to boot and destroy the unikernel. It does not even include a network interface, if the unikernel does not need one. To prevent the provider from the task off managing many different hypervisors running on one host, this unikernel monitor is be packaged and shipped together with the unikernel.

The result is a type-II hypervisor which is able to run on a standard Linux *KVM* system. The prototype *ukvm* contains only about 1000 lines of code and delivers a significant faster startup time, than a unikernel running on standard *QEMU/KVM*.

The next step in cloud computing is called *serverless* or *Functions-as-a-Service (FaaS)*. Developers define a function, executing on simple task, and an endpoint which triggers this function. They are not concerned with how the function is actually executed and the underlying infrastructure. Andreas Happe et. al. proposed a FaaS framework based on unikernels [6].

Every function is wrapped as a unikernel and for every call to this function an instance of the unikernel gets started. After the request is completed the unikernel gets immediately terminated. This is all done transparently to the user.

The small size and fast startup time of unikernels makes this possible. No resources are wasted by idle processes and between the functions is a strong isolation.

Another area where unikernels gain traction besides cloud services, is the world of IoT. Jan Amort evaluated the unikernel for software on IoT devices [45]. Based on his evaluation he proposed a pipeline to deploy software on IoT devices using unikernels and thus simplifying the process of rolling out updates to this devices.

CHAPTER 4

Performance Evaluation of Unikernels

Performance is an important topic when choosing the platform for an application. With compute and memory becoming cheaper in the last decades, the priorities shifted for most applications and resource efficient software was not number one priority any more. By simply adding more computational power, performance can often be easier achieved.

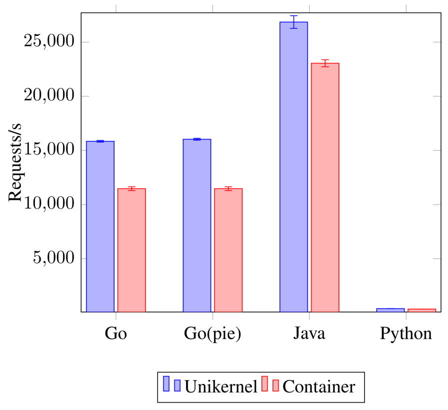
This is about to change again, with more developers and companies relying on third party cloud providers. The providers charge their customers not per rented VM, but rather based on used memory and storage as well as computation time.

Because of this, it is not feasible any more to simply rent a couple of powerful VMs. Running VMs that are idle, mean lost money. One way to reduce these costs are containers. They allow to easily scale horizontally and to add and remove instances, based on the workload. Although the lightweight containers are cheaper than bulky VMs, they are still to big for real just-in-time starting and destroying of instances, e.g. one instance per request for a webserver. Unikernels are a possible solution to this problem, which presented it self over the last couple of years.

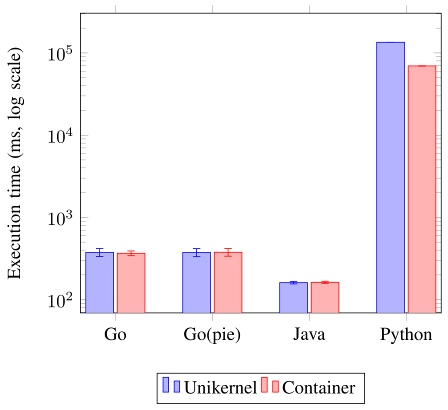
This chapter consists of a collection of studies and publications that evaluated different unikernel solutions and compared them with other virtualization technologies, like container or classical VMs, as well as bare metal. It contains observations about the general computational performance of unikernels as well as more specific areas like memory, network I/O, image size and boot time.

4.1 Computational Performance

To have a low computational overhead is especially important for cloud services. Not only does a fast computation directly result in a shorter response time, but it potentially



(a)



(b)

Figure 4.1: The results of a benchmarking study on unikernels versus containers, by Tom Goethals et. al. [46]. In (a) the REST stress test performance evaluation and in (b) the Bubble sort execution time.

saves cost as well. Especially if it is running on a public cloud provider, who charges by computation time.

Because of this many studies analyze the computational performance of different virtualization technologies. This allows us to directly compare the performance of unikernels to more established technologies like classical VMs and containers.

Tom Goetahls et. al. [46] performed an in-depth benchmarking study on unikernels versus containers.

They implemented a simple REST service in each of the languages *Go*, *Java* and *Python*. Based on this service they performed stress tests both on *OSv* unikernel and *Docker* container and measured the request per second.

By comparing the results for the different implementations in general, the highest performance is seen for the *Java* implementation, fallowed by the implementation in *Go*. Due to the fact of *Python* being an interpreted language, the implementation in this language was significantly slower.

Figure 4.1a shows the results of the single-threaded setup. All three implementation show a significant better performance while running as a unikernel, compared to running as a container. With 38% more requests per second, *Go* shows the highest improvement. *Java* and *Python* have with 16% and 15% similar performance gains.

By enabling multi-threading the containers experience a performance increase, the unikernels however show a drop in the performance and are outperformed by containers on a good portion. When single-threaded and multi-threaded results for unikernel are compared directly, it is immediately visible that only *Java* shows an increase in performance at all. While *Python* is with a small performance drop of 3% relatively unaffected by the multi-threading, *Go* looses 25% if its performance. Even *Javas* 60% performance increase is quite underwhelming for a quadrupled processor power.

In the same study a heavy load test was performed by implementing a simple bubble sort algorithm, again in *Java*, *Go* and *Python*. Figure 4.1b shows, that the advantages of unikernels over containers are not so clear in this case. While, with a 3% decrease for the *Go* unikernel and a 1% increase for the *Java* unikernel, the performance is almost equal to the containers, the execution time of the *Python* unikernel is twice as long as that of the container. This low performance of the *Python* implementation is explained with the assumption that language uses a significantly bigger amount of operations that perform badly in a fully virtualized environment.

Another paper, that not only compares unikernels and containers, but also takes the performance of classical VMs into account, is the paper for the 2015 IEEE International Conference on Cloud Engineering by Murabito et. al. [47]. In their paper they present the results of multiple benchmark tests, where they compared the performance of a classical VM (*KVM*[1]), containers (*LXD* and *Docker*) and a unikernel (*OSv*). Two of the benchmark tests focused on computational performance.

The first benchmark tool they used was *noploop*[2], this is a simple tool which measures CPU clock speed, by using an unrolled No-Operation (NOP) loop. *OSv* performs in this benchmark test slightly better than containers and classical VMs, but the minimal difference of about 0.15ms is neglectable.

The second more sophisticated benchmark tool they used is called *Linback*[3]. It measures performance by solving an algebra problem. The algorithm solves $A \times X = B$, where A is a matrix with size N and B is a vector. The benchmark performs the calculation repeatably while slowly increasing N. The results again show almost equal performance of all virtualization types. Only for a small N, shows *OSv* minor performance degradation. No explanation is given as of why this is the case, but it might hint for an implementation flaw in a system library in *OSv*.

In the paper of Avi Kivity et. al. [14], where they present the *OSv* unikernel, they as well performed multiple benchmark test, to measure the performance of their unikernel implementation. To measure the overall performance they used *Memaslap*[4], which performs request to the key-value store *Memcached*[5], and *SPECjvm2008*[6], which is a Java benchmark site.

In the benchmark tests using *Memaslap* they were able to achieve 22% higher throughput than on a standard Linux. The *SPECjvm2008* benchmark showed an average performance

[1]https://www.linux-kvm.org
[2]www.brendangregg.com/blog/2014-04-26/the-noploop-cpu-benchmark.html/
[3]people.sc.fsu.edu/~jburkardt/c_src/linpack_bench/linpack_bench.html
[4]http://docs.libmemcached.org/bin/memaslap.html
[5]https://memcached.org/
[6]https://www.spec.org/jvm2008/

gain of 0.5%. This does not seem much, but is still significant, since the standard deviation with this benchmark tool is only 0.2%.

Another interesting result is from their analysis of the context switches, which shows that switching between threads is 3 to 10 times faster on *OSv* compared to a standard Linux VM.

MirageOS is another unikernel implementation. It was presented in the article of Anil Madhavapeddy et. al. [5], where they explained the concept of unikernels. The performance of their unikernel is evaluated by using several real world applications.

The first application is a DNS server based on *MirageOS*. Its throughput is compared to those of two well established DNS servers, *Bind9*[7] and *NSD*[8]. Initially *MirageOS* performed poorly. By adding changes to the implementation, to memorize responses and prevent repeated computation, the performance improved significantly. This shows that sometimes algorithmic improvements exceed those of machine level improvements.

Another application developed to evaluate the unikernel, is a webserver implemented on top of *MirageOS*, which was compared to a standard webserver on Linux. It shows that the unikernel scales much better, with a linear scaling up to 80 sessions, before it becomes CPU bound. In contrast the Linux VM reaches its limits already with 20 sessions.

Finally, in their proceeding [13], Kai Yu et. al. compared the unikernel implementation *MiniOS*, which is shipped together with the *Xen* hypervisor, to a standard Ubuntu VM. They implemented a minimal http server and measured its performance by using the benchmark tool *Weighttp*[9]. This way they were able to show a 39% performance improvement of *MiniOS* compared to the *Ubuntu* VM.

As expected, there is not much difference between all the different virtualization techniques, when it comes to raw compute. This is because it mainly depends on the used CPU, which is always the same. However as soon as it depends on specific software components as well, like drivers or kernel libraries, a good performance boost is observed by using unikernels.

This is because of unikernels being highly customized for specific usage, for instance to run cloud services. An example is the optimized network stack of *OSv* [14]. Standard OS, which are used for VMs as well as a base for container images, are implemented to support a big variety of use cases and perform reasonable well in all of this cases. Unikernels on the other hand are designed to be specialized for a specific task, by adding optimized kernel libraries at build time, this gives them the advantage.

[7]https://www.isc.org/bind/
[8]https://www.nlnetlabs.nl/projects/nsd/about/
[9]https://redmine.lighttpd.net/projects/weighttp/wiki

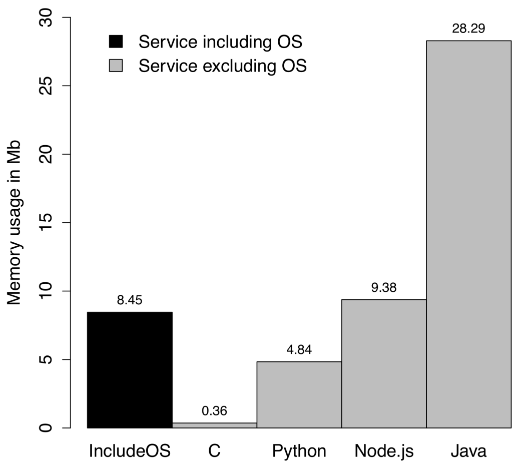


Figure 4.2: Showing the minimal memory usage of an IncludeOS virtual machine including a "Hello world" program, the necessary parts of the OS- and standard libraries, and a custom bootloader, in addition to the Qemu process itself. In comparison "Hello world"-programs in various language frameworks, excluding their operating systems, are shown [48]

4.2 Memory Management

Memory usage is another pressing topic for cloud services. Especially a small memory footprint is important, because it allows to run many instances of services on the same host, without running out of resources.

In the paper already mentioned above [47], Robert Murabito et. al. performed a memory I/O benchmark test as well. For this they used the tool *STREAM*[10] which performs simple vector kernel operations.

The results show that *KVM*, *Docker* and *LXD* all perform equally well as the native execution. Only *OSv* has approximately half of the memory throughput than the others.

The same benchmark tool was used by Alfred Bratterud et. al. in the article where their unikernel *IncludeOS* is presented [48].

IncludeOS was compared to *Ubuntu* running in a VM as well as native. While the native execution performed clearly best, the differences between *IncludeOS* and the *Ubuntu* VM are not so clear. On *Intel IncludeOS* performed slightly better than the VM, both on *AMD* it was the other way around. In both cases are the differences less than 0.5%.

The different results of this two studies show, that with unikernels it highly depends on the type of unikernel. Every implementation optimizes for different goals and it is important to always carefully choose the right solution for a specific use case.

[10]`https://www.cs.virginia.edu/stream/`

Alfred Bratterud et. al. [48] additionally compared the memory usage of a minimal hello world implementation using the *IncludeOS*, to the same implementation in different languages and on a traditional OS. While in the results for *IncludeOS* the whole unikernel is includes, the results of the other implementations, running on standard OS, include only the application it self without the OS.

The *IncludeOS* unikernel uses with 8.45 MB more than the pure applications in *C* and *Python*. But the *Nodejs* implementation was already slightly bigger and the *Java* implementation required more then three times of the memory space, always without including the OS. Figure 4.2 shows these differences.

In the study of Tom Goetahls et. al. [46], additionally to the benchmark tests mentioned before, the memory consumption of the *OSv* unikernel were compared to those of the *Docker* containers. As expected the memory usage is higher for the unikernel as for the containers. It ranges between twice as much for the *Java* implementation and *30* times as much for the *Go* implementation. Although this sound a lot, in absolute numbers it is only between 70MB and 130MB more memory required.

The differences are again explained with the fact that unikernels always have to include the kernel as well, while the containers can rely on the kernel of its host. But since unikernels are able to run directly on a hypervisor type I, no OS is needed. This saves in total a lot of memory, which in the case of the containers is used by the OS it self.

Simon Kuenzer et. al. compare in their article [49], the minimum memory allocation that is required to boot a VM. They introduce a highly optimized CDN implementation, using the unikernel *MiniOS*. Their implementation, which they call *Minicache* only needes 1.2 MB of memory. This is little, even compared to the extremely stripped down, unikernel like, Linux distribution *Tynix*, which is running the two standard tools *nginx*[11] and *lighttpd*[12] and needs 51 and 23 MB memory respectively. With 82 MB has the standard *Debian* the highest minimal memory allocation.

All of this publications show that unikernels may not provide much improvements compared to other virtualization technologies, when it comes to memory throughput, but in terms of memory usage, they have the lowest footprint by far.

This is because a small memory footprint is a key feature, for which unikernels were designed in the first place. It is achieved by getting rid of all OS features and libraries that are contained in standard OS and loaded in to the memory, even if they are not needed at all by the application running on the OS.

The small memory footprint makes unikernels cheap and allows the execution of hundreds or even thousands of instances on the same host, which is in general not possible with classical VMs and even hardly achieved by containers.

[11]`https://www.nginx.com/`
[12]`https://www.lighttpd.net/`

4.3 Network I/O

Applications that are build according to the microservice paradigm consist of multiple distributed service. This services communicate with each other and the different clients mainly over a network connection. Thus it is important that the network I/O virtualization is fast and comes close to bare metal network drivers.

Pekka Enberg [50], in his master thesis, and Roberto Morabito et. al. [47], in their paper for the 2015 IEEE International Conference on Cloud Engineering, both evaluated network I/O performance of hypervisor based virtualization (using *KVM*), to *OSv* unikernel and container based virtualization (using *Docker*).

Their experiment setups were fairly similar, the biggest difference was that Morabito et. al. used a 10 Gigabit network interface and Enberg only a 1 Gigabit network interface. This is important because in certain scenarios it lead to differences in the results.

The first scenario measures the transmission of TCP and UDP data from the client to the server and the other way around. For both cases *Docker* performs best fallowed by *OSv*. Although *OSv* performance still significantly better than a standard Linux VM. For UDP Morabito et. al. observed a considerable performance drop, for all technologies, compared to TCP. Since Enberg did not observe this differences, it suggests that the faster network interface suffers more from the virtualization than the slower one.

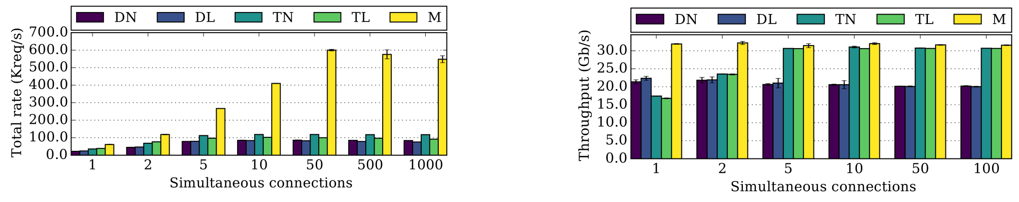
The next scenario measures the round-trip, a request is send to the server and a response to the client, both for TCP and UDP. *Docker* using bridged networking has the least overhead fallowed by *OSv* with vhost-net. *Docker* using NAT networking has even more overhead than the Linux VM with vhost-net. This means it depends here more on the network I/O virtualization technique, than the OS virtualization.

Additionally to this raw request benchmark tests, Enberg performs an experiment using *Memecached* to measure the effects of network intensive applications on different virtualization techniques. The results are mostly consistent with the raw networking benchmarks. The one exception is that *OSv* does not beat *Docker* NAT in this setup anymore. This suggests that other components of the system in *OSv* are slowing down the request/response processing, which is not visible for raw network measurings.

Tom Goetahls et. al. [46] analyzed the results of their benchmark tests also in terms of latency. The *Go* implementation running in a *OSv* unikernel has a slightly higher median response time than its container counter parts. On the other hand is its response time relative stable, while the maximum response time for the container is almost 10 times higher.

The *Java* implementation performs equally well on both, unikernel and container. Finally the *Python* implementation has as a unikernel, with a median response time of 100ms compared to 111ms, a considerable shorter latency than as container.

The highly specialized unikernel of Simon Kuenzer et. al. [49], also brings performance gains in terms of network I/O. The results in figure 4.3 show that *MiniCache* reaches with



(a) (b)

Figure 4.3: HTTP serving performance in (a) reqs/sec and (b) throughput of MiniCache compared to nginx and lighttpd on various platforms over 40 Gb/s NICs, using a single VM. From the study performed by Kuenzer et. al. [49]. In the legend, D=Debian, T=Tinyx, M=MiniCache; L=lighttpd, N=nginx.

600K request per second, for 50 simultaneous connection, by fare the best performance. Fallowed by *nginx* and *lighttpd* on *Tynix*, with 118K and 102K req/sec. The slowest performance showed *Debian* with 85K and 75K req/sec, for *nginx* and *lightpd* respectively. The same is true for the throughput, where *MiniCache* reached 32 Gb/s fallowed by *Tynix* with about 30.7 Gb/s and *Debian* again last with 20-22 GB/s.

The *OSv* unikernel was also evaluated in terms of network performance. Avi Kivity et. al. [14] used the tool *Netperf*[13] for this evaluations. It showed a reduction of 37%-47% in latency for request/response, compared to a standard Linux. The results of TCP STREAM on the other hand, which measures single-stream throughput, was 24%-25% higher.

Both unikernels and containers show an improvement in network I/O compared to standard VMs. When comparing unikernels with containers it is not always so clear and often containers are able to outperform unikernels. This is possibly because containers don't need to rely on a virtualized network interface, but can used the network driver of the host kernel directly. There are however unikernels that are able to beat containers as well, again by specializing and optimizing the network stack for their specific needs.

4.4 Boot Time and Concurrent Provisioning

On cloud platforms services get spawned and destroyed rapidly, often many instances on the same time. This means it is important that the instances don't slow each other down if they get provisioned concurrently and that they have a fast startup time.

Bruno Xavier et. al. [51] conducted an experiment where they spawned several instances (10, 20 and 30 instances) at the same time and investigated the impact on the provisioning time. This was done with a standard Linux VM on *KVM*, with an *OSv* unikernel on *KVM* and with a *Docker* container. The experiment was performed on an *OpenStack*[14]

[13]`https://hewlettpackard.github.io/netperf/`
[14]`https://www.openstack.org/`

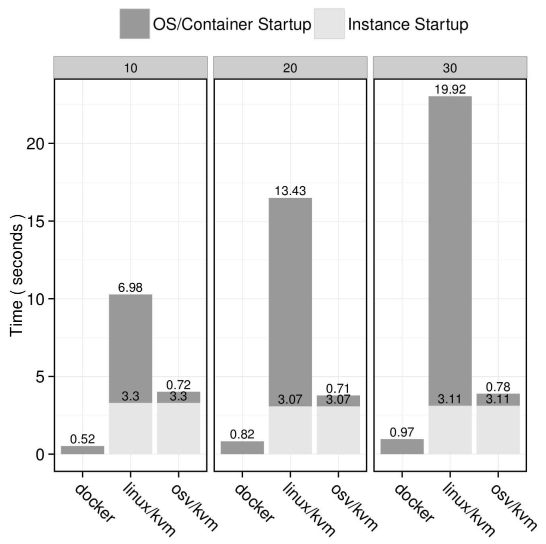


Figure 4.4: Instance and Operating System/Container Startup for 10, 20 and 30 instances, as evaluated by Bruno Xavier et. al. [51]

environment with one controller and one compute node and the results were compared with each other.

The first evaluation was regarding the startup time. Figure 4.4 shows that *OSv* and *Docker* have almost the same results for all three scenarios and always below one second. But regarding the variance, *Docker* shows an increase in the distance between the first and the last container, which is not visible in the graphic. This is due to the internal synchronization of the *Docker Engine*, which will always increase the overall time with the addition of concurrency. Since this only becomes relevant with a large number of containers, it is not a big drawback.

The next evaluation inspected the image import. This consists of the copy of the image from the repository and its preparation inside the compute node. For the *OSv* unikernels the concurrency had non effect what so ever on the import time. The *Docker* containers on the other hand, showed again a slowing down with increased concurrency. This is explained by the decomposition of the images to check its metadata, casing again a serialization by the *Docker Engine*.

The last evaluation shows the overhead generated by the platform it self. Here both *OSv* and *Docker* were impacted by the concurrency, although *Docker* shows less overhead in general.

The overall provisioning time increases for both *OSv* and *Docker* with increased concurrency. But *OSv* unikernels are significantly faster and the distance between *OSv* and *Docker* increases with more concurrent instances. This increase in the distance can be explained by the image import of the *Docker* containers.

Simon Kuenzer et. al. show in their article [49], where they created the optimized CDN server *Minicache*, how they were able to even increase the boot time of the unikernel further, by adding optimizations to the underlying *Xen* hyervisor. They managed to reduce the startup time from 200 ms on an unmodified *Xen* to on 89 ms, by applying multiple modifications. With this modifications in place, *Minicache* was compared to an other unikernel *Tynix* and to a standard *Debian* VM. *Tynix* showed a startup time of 665 ms and *Debian* took 5.4 sec to finish the startup.

In the article of Anil Madhavapeddy et. al. [5], where they presented the concept of unikernels, based on their implementation of *MirageOS*, they compared the boot time of a minimal webserver running on *MirageOS* to that of a minimal Linux kernel and that of a standard *Debian* running *Apache2*[15]. The boot time of *MirageOS* was equal to that of the minimal Linux kernel and half of the standard *Debian* VM. Again, by optimizing the underlying *Xen* hypervisor they were able to reduce the boot time of the unikernel significantly, to only 50 ms.

This studies show that another strong advantage of unikernels is their boot time. Due to their small size and the fact that there is no full sized OS that needs to boot before the application can be started, gives unikernels an advantage. Some were able to reduce the startup further time by optimizing the underlying hypervisor, instead of only modifying the tool stack of the unikernel it self. This optimizations are definitely harder to achieve as adding modified libraries to the unikernel, but it shows what is possible if a hypervisor, optimized for unikernels, is used.

4.5 Summary

Through the analysis of this different studies multiple observations about the performance of unikernels are possible.

First of all, it highly depends on the selected unikernel how well it performs for a certain task. Every unikernel focuses on different targets, some focus on portability, others on type safety. Again other unikernels make high performance their priority.

This means the results of the different studies can only be applied to other unikernels up to a certain degree. In a big fraction of the studies *OSv* is used as unikernel, this makes many results somewhat comparable, but the differences in studies where other unikernels are used, underline the differences between the unikernels.

The main reason why *OSv* is often used as a representative for unikernels, is probably its compatibility to POSIX and to possibility to run applications written in many different languages. This makes it easier to compare it to other technologies like containers and classical VMs. *OSv* is also one of the most production ready unikernels, that are currently available.

[15]`https://httpd.apache.org/`

Another obvious observation is that one of the biggest advantages of unikernels is their small size. It not only makes it easy to load images over the network, it is also the main reason for their small memory footprint and the short startup time. This three attributes share all unikernel implementations and are also one of the main reasons why unikernels are used in the first place. Also no other virtualization technique is able to beat this qualities, not even containers.

Possibly the biggest advantage of unikernels when it comes to performance, is their high customizability. This is also the reason for the first generation of library OS, which were developed in the 1990s. The design of unikernels allow to add every OS functionality at build time as a library. This makes it possible to optimize things like the filesystem or the network stack in a way that is not possible for a general purpose OS, like Linux. Since a unikernel can run only one process and thus only one service with a specific task, everything can be optimized for this task, which brings a big advantage for performance.

Attempts to not only optimize the VM it self, but also the underlying hypervisor, added another performance boost to the unikernel.

Unikernels have several advantages compared to other virtualization technologies,. When it comes to performance and resource efficiency, their small size and fast startup time, as well as their high customizability, are the most important ones. Security is another important topic, which will be covered in the next chapter.

CHAPTER 5

Security Evaluation for Cloud-Services

Microservice applications run almost exclusively in some kind of cloud environment. Sometimes it is a private cloud, only accessible from within an entities Intranet, but many times it is a public cloud, that is managed by a third party. Security is an important issue in either case. But especially in the public cloud it is essential, because if an application has access to the public Internet, there is always the chance of attacks. In a third party public cloud, with its multi-tenancy, comes the issue of trust on top. The provider and the different tenants can never trust each other fully.

This means it is important to evaluate a technology for cloud services, not only from the perspective of performance, but from a security perspective as well.

This chapter first describes in general which security implications applications face, when they run in a cloud environment. Second a more closer look will be taken at possible vulnerabilities of the *Docker* ecosystem, as a representative of containers, and directly compared with unikernels. This way the performance of unikernels in a cloud environment from a security perspective, will be visible.

5.1 Security Implication in a Cloud Environment

In a cloud environment it is especially important to evaluate the used technologies from a security perspective. For this reason a look at the different security risks in the cloud is necessary.

5.1.1 Security Threats

There are already many publications that address the general security treads in cloud computing and to its virtual infrastructure. One comprehensive list is by Tariqul Islam et. al., they analyzed the vulnerabilities and security threats linked to cloud computing and grouped them into six categories [52].

Network security contains all vulnerabilities and threads associated with network communication and configuration. Part of this category are, among others, malware injection attacks, Insecure APIs of the cloud services, Cross Site Scripting or SQL injection.

Virtualization and Hypervisor security addresses all vulnerabilities in the virtualization and resource sharing. This can be vulnerabilities in the hypervisor, that can be used to gain full access to all VMs running on it, or the single point of failure of a hypervisor. Other threats in this category are flaws in the OS running inside the VM, that allow attackers to escape the VM and break the isolation boundaries between the Guests, or out dated packages used in the VMs that can expose vulnerabilities.

Identity and Access Management of the cloud services is another important point, that can pose a security threat, if not done properly. It could allow attackers unauthorized access to information or event help to gain full control of VMs. This is an important task that up to some extend lies in the hands of the cloud providers.

Data and Storage Security becomes especially important in a cloud environment where storage is outsourced to a third party. This party is not necessarily trusted, which brings new challenges for data integrity, availability and encryption, but also the sanitization of data not needed anymore.

Governance or better the loss of it, by giving the providers control over critical issues like policies, procedures, and security mechanisms of deployed services. This brings the risk of data leaks or vendor lock-in.

Legal and Compliance Issues summarize all issues that come from the responsibility of an organization to comply with laws, regulations or standards. This can be regulations about the geographical location of data or differences in security and privacy regulations in different countries.

This categories are broad and some of this security threats are not technical, like *Governance* or *Legal and Compliance Issues*. For all the other categories lies the responsibility always to some extend at the provider and to some extend at the developer who implements the services in a way that they are secure. But also the used technologies play a big role in securing a system and unikernels can help to provide a better isolation, hardening of the system and reduction of the attack surface.

5.1.2 The OWASP Top 10

The OWASP (Open Web Application Security Project) is a community based organization, who's aim is to educate about security risks of web application and how to protect those applications. Among many others projects, OWASP releases regularly a list of the Top 10 security risks for web applications. The latest is the OWASP Top 10 - 2017 list and is displayed in table 5.1.

Code	Threat
A1:2017	Injection
A2:2017	Broken Authentication
A3:2017	Sensitive Data Exposure
A4:2017	XML External Entities (XXE)
A5:2017	Broken Access Control
A6:2017	Security Misconfiguration
A7:2017	Cross-Site Scripting (XSS)
A8:2017	Insecure Deserialization
A9:2017	Using Components with Known Vulnerabilities
A10:2017	Insufficient Logging&Monitoring

Table 5.1: The OWASP Top 10 - 2017 [53]

Some of them are entirely on application level and can not be solved by the platform, like A2, A5 or A7. For others unikernels can help to reduce the risks.

An important one is A6. Misconfiguration is a big security risk and it is common, also because the general use of ad-hoc configuration, e.g. in text files. Unikernels can help here, because all their configuration is programmable and added at build time. This way it allows for build time checks by the compiler and after the build the configuration can not be changed anymore [5].

Another example where unikernels can help reduce the attack surface is A9. Using outdated libraries with known vulnerabilities is a big issue. This is of course also possible with unikernels. But since unikernels link only those components that are essential for the correct execution, they contain much less unnecessary code, which in turn reduces the included components with potential vulnerabilities [5].

Also code injection (A1) becomes much harder, because unikernels are immutable and do not allow the execution of code that was not there add build time [5].

Finally, unikernels can also indirectly help to prevent security risks, e.g. thread A10. Since unikernels are like one standalone application, a developer can not simple log into the system to see what is happening, like it is possible with VMs or containers. This forces the developer to add more logging and monitoring mechanisms.

5.2 Vulnerability Comparison

It is essential for web based applications to be resilient towards all kind of attacks. How secure an application is depends not only on the implementation of the application it self, but as well highly on the platform to application runs on. The more security a platform provides out of the box the better, since it helps the developers to make their applications more secure.

A. Martin et. al. [54] analyzed the vulnerabilities of the docker ecosystem. They divided the vulnerabilities into five categories:

a) Insecure configuration

b) Vulnerabilities inside the images

c) Vulnerabilities directly linked to the runtime

d) Vulnerabilities in the kernel

e) Vulnerabilities in the image distribution, verification, decompression and storage process

In the following they will be describe more closely and suggestions will be made how this vulnerabilities apply to unikernels. This allows for a directly comparison between containers and unikernels from a security perspective.

5.2.1 Insecure Configuration

Configuration is an important part for every system. It allows the adjustment of the system to the special needs of a specific use case. For most systems, including container, most of this configuration is done via ad-hoc configuration, either via special commands or text files. Most systems and application provide a default configuration and many users rely even in production on this default configuration. This can lead to a number of weaknesses, because many of this default configurations focus more on usability than security [55].

The default configuration of *Docker* is relative secure. But there are many options available that allow to configure a container, with almost full access to the host. Although this possibility can be useful at times, e.g. a privileged container that manages other container, this can provide some attach surface both for the host it self and for other containers running on it. According to A. Martin et. al. this can especially become a problem when the containers are not used in their recommended way, which is a single purpose instance that runs only one process, but as a full fledged VM with multiple processes. Ubuntu and CentOS are the by fare most used base images, which means they package a full OS with all its capabilities inside the container. [54].

Unikernel are much more restrictive. First of all they simply allow only one running process and include only the libraries that are needed to run this process. Since they run on a hypervisor and not on a host OS, resource sharing at a scale that is possible with containers is here not an option. All the configuration is added as libraries during build time, which allows them to be explicit decisions that are programmable and can not be changed after the build. This prevents an instance from getting more rights as it was intended. Especially the explicit programming of every configuration prevents the developers to some extend from the inconsiderate application of some configuration settings [5].

5.2.2 Vulnerabilities Inside the Images

Into this category fall all vulnerabilities that come from the application code base it self or the code base of the OS image used.

Often vulnerabilities in this category come from out-date dependencies, this is on one side due to the fact that the multiplication of image builds in the automated processes, leads to outdated versions still a available at the registry, while the fast updated and deployment cycles focus most of the time only at the `latest` image. On the other site, due to the layered fashion docker constructs images, a vulnerability in a base image effects all its child images. Relevant in this context are attacks that come from outside, e.g. code injection [54].

Also unikernels can contain vulnerabilities due to out-dated dependencies. The advantage here is, that the code base is usually much smaller and only liberties that are essential for the application are included into the build. This limits the attach surface tremendously.

Bratterud et. al. call code that is included in an image but not needed *bloat*. They show that Linux-based VMs can easily be 2000% bloated, especially if they are used as a single purpose machine. We argue that the same is true for containers, as long as full sized OS are used as base images. *IncludeOS*, as a representative of a unikernel, is 50/50 software and OS, with only a few percent bloat [56].

Unikernels contain usually no shell, this comes naturally with its single process architecture, since a shell is a process with the purpose of running additional processes. Additionally has a well designed unikernel no address space that is executable and writable at the same time and every address space that neither need to be executable nor writable is read-only. With this two attributes we have an immutable system and even if an attacker manages to inject code, there is no way to execute it [56].

This way virtually the only way to inject malicious code into an unikernel, is before the build, for instance when code gets pulled from a remote repository. But to secure this, lies in the responsibility of the provider of the build environment.

5.2.3 Vulnerabilities Directly Linked to the Runtime

The runtime environment is an other source for vulnerabilities. This can be vulnerabilities in the physical or virtual host, its OS or other malicious guests that run in the same environment. This is especially an important category in the public cloud. There we have multiple tenants and the provider, and all of them can not trust each other fully, there is always a chance at least one of the actors has malicious intends. This means vulnerabilities in this categories mostly concern isolation.

With *Docker* sources of vulnerabilities in this categories lie mostly in the framework it self or the libcontainer. Container are isolated with their own namespaces and cgroups, and additionally protected with technologies like SELinux, Apparmor and Seccomp. This provides in general a good isolation of the containers from the host system. But in the end those techniques are all just advanced process isolation techniques. By default containers use the same groups and security profiles, which means the isolation between the containers is not so strict. This can be changed, but it is not done often, because it comes to the cost of resource sharing, which means the communication between the containers is much more restricted. Additionally since containers often run with root privileges, they immediately have access to the whole host system and other containers if they escape [54].

Unikernels are executed like any other VM, which means they have by definition a stronger isolation and also at a lower level. Depending on the hypervisor used, it is fully virtualized and isolation is enforced at the instruction level with hardware or paravirtualized, with parts of the isolation on software level and some minimal resource sharing. In any cases it is much stringer isolated than containers. A malicious unikernel can still gain control to the underlying hypervisor by exploiting some vulnerabilities of said hypervisor and thus indirectly also effect the other unikernel running on it. But a hypervisor is not a full OS which again limits its attach surface [56].

5.2.4 Vulnerabilities in the Kernel

The Linux kernel is the base of most docker containers. It is quite complex and its code base is over 19.5 million lines of code and it is growing. Projects of this size are seldom without errors and in the last decade there were about 1500 detected vulnerabilities in the kernel code alone [57]. Even minimal Linux distributions are shipped with additional software on top of the kernel, which further increases the attack surface.

Since container use the same kernel as the host, they are vulnerable to kernel exploits. Which allows an attacker to break out of the container and gain control of the host [54].

Unikernels on the other hand have their own kernel, this means there is virtually no way that an attacker can gain control of other guest or even the host, trough such a vulnerability. The fact that only those parts of the OS libraries that are really needed are linked to the final image, which further reduces the attach surface. Unikernels often re-implement big parts of the kernel, either to introduce type safety or to increase performance. Although this brings advantages, with the type safety also from a security perspective, the reimplementation can introduce new bugs and vulnerabilities, that are not present in the well established Linux kernel [5].

5.2.5 Vulnerabilities in the Image Distribution, Verification, Decompression and Storage Process

Martin et. al. name the vulnerabilities in this section as the highest, especially if the automated build process is performed as from *Docker* recommended. This is because there are multiple external services involved that all have full access to the code base and could potentially add changes with malicious intend. On the other hand it is also the only section where there is almost no difference to unikernels. Every time an automated build is performed, using third party services and over the public Internet the security lies in the hands of this services and secure connections [54].

5.3 Summary

Containers are good, light-weight alternative to traditional VMs. This shows already their popularity and increased usage over the last years. If they are used correctly they are relative secure and their bigger flexibility at isolation brings definitely advantages compared to other virtualization techniques. This is especially useful in a private cloud, where in general runs only trusted code and security requirements are often not so strict, because it is not reachable from the public Internet and entities who own it have full control what is deployed onto it.

In a public cloud, with multiple tenants, the security requirements are much higher. Nevertheless, rely provider more and more on container solutions over traditional VMs. They have a smaller resource footprint than VMs and with their fast boot time they can rapidly be spawned and destroyed, often on a per request basis. For the user they are easier to setup. If they are used and configured correctly they are definitely a good alternative to classical VMs. But from a pure security perspective they are far from the best solution and pose many vulnerabilities and security risks.

Since unikernels are becoming more mature and the first implementation become ready for industrial usage, they pose an interesting alternative to containers. The previous chapter showed how they perform equally well or even better as containers and have an even smaller resource footprint. From a security perspective they provide much better isolation than containers do, in fact there isolation is equal to the one of traditional VMs. Additionally, due to their small code size and the sealing at build time, the attack surface stays as small as possible.

CHAPTER 6

A Continuous Delivery Strategy for Unikernels

In this chapter a continuous delivery strategy is proposed, that is specifically designed for cloud services which are deployed as unikernels.

First, a set of chosen tools is presented, that can support the automation process. Secondly, the necessary changes and contributions to this tools, to fit them to the needs of the deployment of services as unikernels, are explained. Subsequently, a continuous delivery pipeline for services as unikernels is developed, by taking use of said tools. Finally, the reliability and usability of such a pipeline is analyzed by developing a small microservice application and continuously delivering one of its services via the pipeline developed before. The results are compared to a similar pipeline using *Docker* containers instead of unikernels, which is currently the most popular technology to release and deploy microservices.

6.1 Tools for a Continuous Delivery of Unikernels

In order to set up an automated build and deployment pipeline, appropriate tools are needed. They need to work well together in order to perform the different tasks that are necessary for a CI/CD process specified later.

6.1.1 CI/CD Server

The first and most important tool is the one that performs and manages the automated deployment process. There are many different so called CI/CD servers on the market. They all have the goal to ease and automate the build and deployment process. Their features range from simply executing predefined scripts in a certain order, to extended visualization of success and failure rate, monitoring of the build processes, security features like authentication or secure stores for API keys and passwords, and notification features. Additionally most of them support direct integration to other tools like git and artifact repositories, testing frameworks and a wide range of hosting providers and orchestration tools.

Some of these are open source and free to use, others are proprietary. The wide range of available CI server solution shows how much of an important part CI/CD has become for the development and release cycle. *Jenkins*[1], *Hudson*[2], *Tarvis*[3], *GitLab CI*[4], *CruseControl*[5], *Codeship*[6], *TeamCity*[7],*CircleCI*[8] and *Bamboo*[9] are some of the most used ones. Most of them can be installed on premise and some are additionally offered as a service. Since they all serve mainly the same, specific purpose, their features vary just slightly.

For the continuous delivery setup proposed in this thesis *Jenkins* was chosen. It is well established and widely popular, used by large enterprises and individual developer teams. Jenkins is free to use, open source, and maintained by a large and active community. The modular structure makes it easily extendable making thousands of plugins available for many different tasks.

6.1.2 Build Tool for Unikernels

In order to reliably build a service as a unikernel, a build tool is important. It needs to wrap the often complex steps that are necessary to create a unikernel image and to execute it on a hypervisor.

Currently there are many different unikernel solutions in development, most of them are not production ready yet, and the building of an image is a highly manual process that requires a lot of knowledge in operating systems and virtualization. The average developer does not necessarily want to be occupied with this manual process, which can easily lead to errors. They want a tool that, with a few commands, builds and ships their application as an unikernel and provides an abstraction to various kinds of OS layer operations.

The following are a selection of tools that are currently available and meet those requirements. *Unik*, an open source project that is capable to build applications of different languages, using different unikernel solutions and targeting different platforms [7]. *Capstan* is a tool to build applications specifically using the *OSv* unikernel. It is one of the most advanced unikernel solutions. It is commercially developed and can already be used

[1]`https://jenkins.io`
[2]`http://hudson-ci.org`
[3]`https://travis-ci.org`
[4]`https://docs.gitlab.com`
[5]`http://cruisecontrol.sourceforge.net`
[6]`https://codeship.com`
[7]`https://www.jetbrains.com/teamcity`
[8]`https://circleci.com/`
[9]`https://de.atlassian.com/software/bamboo`

to run applications in production [42]. The last one is *tynix*, a build tool that packages and runs applications with *LighVM*. Both were developed together and manly used for research purposes [41].

As build tool for the toolchain presented in this theses, *Unik* is selected, although it is still under development. It already supports multiple unikernels, including *OSv*. Its modular architecture makes it possible to add support for additional languages, unikernel solutions and target platforms, in a plug and play manner. Additionally, its open source nature makes it easy to extend and change functionality if needed. The support for multiple unikernel solutions and target platforms prevents vendor lock-in and makes a transition to another unikernel solution or target platform really simple.

With this tools, it is already possible to build a continuous delivery pipeline. Although *Unik* only provides a command line interface at the moment, it is in *Jenkins* fairly easy to use shell commands, so it is possible to call the command line interface from within a Jenkins job. A drawback is that it requires the host that runs Jenkins to have *Unik* installed locally and a user of *Jenkins* might not always have the rights to install system tools as they like. However, it is best practice in *Jenkins* to use a plugin that provides the desired functionality. Since to the best of our knowledge, at the time of writing this thesis, there exists no such plugin for *Unik* or for unikernels in general, a plugin was developed and also released to the *Jenkins* community so that others might take use of it as they see fit.

6.1.3 JUnik, the Java Library

Since *Jenkins* is developed in *Java*, it made sense to wrap the functionality to interact with *Unik* into a library. This brings the advantage that this implementation might show useful not only to the *Jenkins* community, but to everyone that wants to integrate a *Java* application with *Unik*. This library was subsequently used to create the *Jenkins* plugin.

Before implementing the library, different options on how to interact with the *Unik* framework were investigated.

The first approach is to simply run the shell commands from within the client library. The obvious disadvantages with this approach are, that it only works on Linux environments and the framework must be installed locally.

Another approach is to use the daemons REST API, which the command-line client uses as well to send commands to the daemon. This way we can directly communicate with the daemon, independently of the location of the targeted daemon. It is independent of the OS and the framework does not necessarily have to be installed locally.

Because of the early stage of the framework, its documentation is sometimes sparse. One missing part was the API documentation, so the source code was analyzed and the documentation provided back to the *Unik* project.

After extracting this information the java client library was implemented, that wraps the REST calls to the unik daemon. The library was dubbed *JUnik* and released on Github[10].

The library is divided into different types like instances, images, and volumes. For each type there are multiple commands like creation, deleting, listing, description and others. These different types are provided by a single client class, that can be instantiated with a specific URL. This way any *Unik* daemon can be targeted, which is available over th network.

Furthermore, the library provides an API for every functionality that *Unik* offers, although some of them are not directly necessary for the implemented *Jenkins* plugin, like the description of an image, or the listing of all available compilers.

Due to the fact that the Unik framework is still in an early state, it does not provide a sophisticated error handling. It only sends 400 or 500 http errors, with out providing additional information on what exactly caused the error. Because of this as much as possible is handled at client side and the parameters are thoroughly validated, before they are send to the server.

In the case there is still an error received from the daemon, it is wrapped in an exception and left to the caller to handle it.

6.1.4 Unik Builder, a Jenkins Plugin

Using the *JUnik* library, a plugin for *Jenkins* to build and run unikernels was implemented, named *Unik Builder*.

The aim of this plugin is to provide a set of simple commands that can be used in a *Jenkins* job to build, ship, and deploy applications as unikernels.

For the implementation the *docker-build-step-plugin*[11] was used as reference and template for the structure and usage of the *Unik Builder* plugin. Since the aim of the *Unik* project is to make the configuration and commands as similar to *Docker* as possible, so that the transition from container to unikernels is easier, it maid sense to also make the usage of the two plugins as similar as possible. The plugin was released open source as well and added to the official *Jenkins* plugin repository[12]

Jenkins defines extension points, which are interfaces or abstract classes that model specific behavior. Those extension points are either provided via the *Jenkins* core or via a plugin. A plugin that targets a specific behavior simply has to implement one of these extension points and annotate it with `@Extension` [58].

For the Unik Builder plugin multiple extension points were used. The most important one is the `Builder`. It provides functionality to implement a build step. By implementing

[10] `https://github.com/mathiasmah/junik`
[11] `https://wiki.jenkins.io/display/JENKINS/Docker+build+step+plugin`
[12] `https://github.com/jenkinsci/unik-builder-plugin`

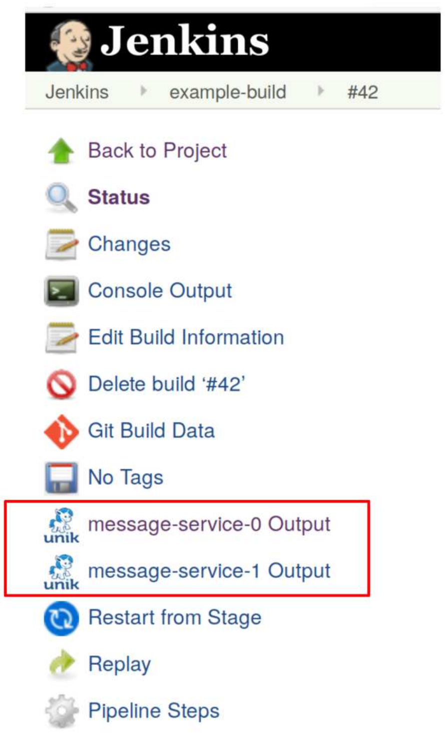


Figure 6.1: The two links in the red box are custom actions, added by the Unik Builder plugin, to view the logs of executed unikernels.

it as a `SimpleBuildStep`, it is automatically usable in freestyle and in pipeline jobs. The `UnikBuilder` lets you define a `UnikCommand`, which is an extension point of its own, but a custom one. Each implementation of the `UnikCommand` defines one specific unik command.

Another special type of extension point is `Action`. Every page in *Jenkins* has a list of navigation and command links on the left site. An action is the implementation of one of the links. If the link is only for navigation, the action simple performance the redirect to the desired target page. Alternatively an action can execute a specified task. Often it is a combination of both. In general, they are added manually by calling `addAction(...)` on an `Actionable` [58].

In this case the action was added to `Run`, which is the object representation of one job execution. This means the action will appear as a link on the view of a specific execution. The action implemented is a `TaskAction`, which allows the definition of an asynchronous task. This allows to add an action for every unikernel that is started, to get and list its internal logs. The red box in figure 6.1 shows these actions.

All extension points are dynamically mapped to URLs using *Stapler*[13]. *Stapler* uses reflection to determine how to process an URL. An example from this project is the validation methods. For instance the class `CreateImageCommand` has a validation method `doCheckImageName(...)`, *Stapler* dynamically maps it to `io.jenkins.plugins.unik.cmd.CreateImageCommand/checkImageName`.

Most of the extension points also have a view that need to be rendered. The technology used for this is *Jelly*[14]. Views are organized according to classes, using *Stapler*. Stabler identifies the correct view for an extension point by naming conventions [58].

Finally, we want to make the plugin more usable for pipeline projects also. This is achieved by using the `@Symbol` annotation. By defining these symbols for the extension points, the amount of text needed to call an extension point from a pipeline script, reduces significantly [58].

For example if there were no symbols, the `unik start` command would look like this:

```
step([$class: 'UnikBuilder',
       command: [$class: 'StartInstanceCommand',
                  instanceName: 'myInstance']
      ])
```

With symbols on the other hand it is much shorter:

```
unik start('myInstance')
```

As seen above, in a pipeline job, configuration is only a matter of a few commands, while as in a free style job, configuration looks like depicted in figure 6.2. This example pulls an existing image from the hub end executes it on the globally configured *Unik* host.

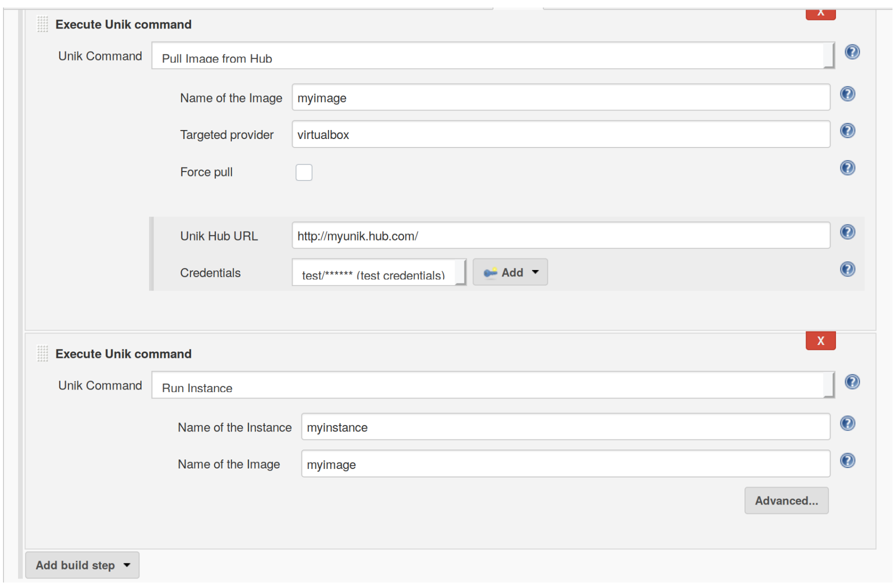
6.2 Continuous Delivery Pipeline

Combining the selected tools and the provided *Unik Builder* plugin, a pipeline for automated delivery and deployment can be established. Figure 6.3 shows the graphical representation of this pipeline.

A continuous delivery pipeline automatically detects changes to the code base and start a build and delivery process. Every time a developer pushes changes to a version control system, the CI/CD tool, in this case Jenkins, detects this changes and triggers a new build.

During the build process all the needed dependencies for the service itself as well as for the unikernel base, are pulled from external sources. The service gets built, by taking use

[13] `http://stapler.kohsuke.org/`
[14] `http://commons.apache.org/proper/commons-jelly/`


Execute Unik command

Unik Command: Pull Image from Hub

Name of the Image: myimage

Targeted provider: virtualbox

Force pull: ☐

Unik Hub URL: http://myunik.hub.com/

Credentials: test/****** (test credentials)

Execute Unik command

Unik Command: Run Instance

Name of the Instance: myinstance

Name of the Image: myimage

Figure 6.2: The configuration consist of two steps, in the first one an existing image is pulled from a repository. In the second step the pulled image gets executed.

of the *Unik Builder* plugin, which relies on a *Unik* daemon to perform the operations. This daemon can be installed locally on the same host as *Jenkins* itself, as well as on a dedicated server on a remote location.

Before, after, or in between certain steps of the build process, unit-, integration-, performance-, or any other test can be performed, as needed for the specific service that is built.

After the built and all tests are successful, the unikernel image can be released.

Unik comes with a registry, similar to *Docker*, where images can be published. It is called *Unik Hub* and in the current implementation it acts merely as a proxy to an *Amazon S3* storage. This brings the advantage that storage locations can easily be changed or multiple storage options can be added in the future.

By taking use of the *Unik Builder*, the image can be pushed to a *Unik Hub* directly, at either a globally configured or a specific location, specifying the corresponding command. In either way the authentication is handled by the plugin using the *Jenkins* credentials plugin, which allows to reference secrets stored in *Jenkins* without storing sensitive information in the code base.

At this point the release process is finished. Now it is necessary to decide, if the new version is automatically deployed to production or to a staging system, or if the pipeline waits for a manual confirmation before it continues with additional steps, like the deployment of the new release.

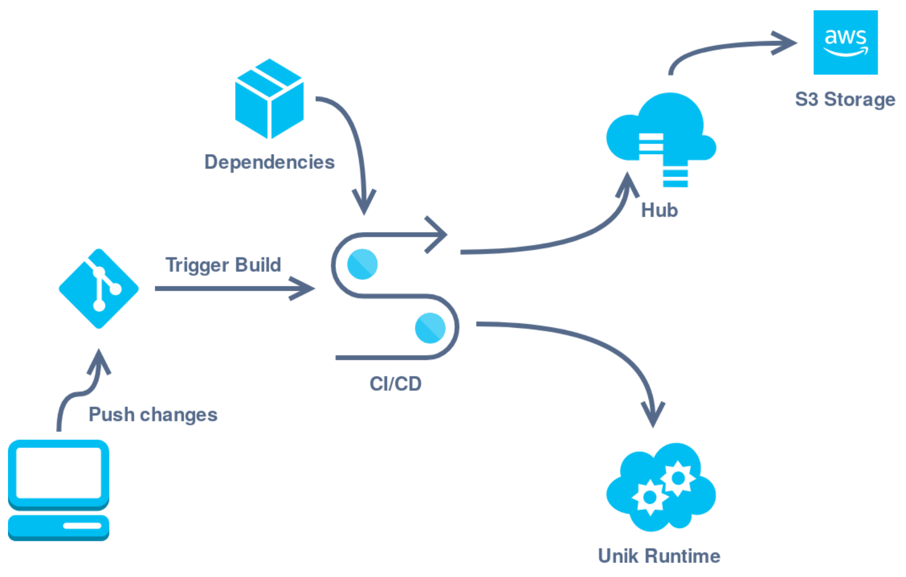


Figure 6.3: Changes to the code base get detected, which triggers a new build in the CI/CD pipeline. A new image is build, released to a repository and finally deployed. All unikernel specific tasks are executed, by a Unik Runtime available over the network.

Although the *Unik* framework manages all the unikernel instances, it differs from a full fledged orchestrator. It can be compared with the pure *Docker* daemon. Each daemon only knows and manages the instances that are deployed to its host. There are attempts to integrate it with *Kubernetes*, which would immediately bring a lot of additional potential to both frameworks, but as of now, this is not in an usable state yet. So the fully or partially automated pulling and roll-out of new versions by the *Unik* framework is currently not possible.

However, another option is to let *Jenkins* deploy the new version, again by taking use of the *Unik Builder* plugin. This will need a manual input or at least minor configuration for the different hosts (each one needs to run a *Unik* daemon) and the number of instances. With this the Unik Builder can download the new image to the hosts, stop running instances if there are any and finally start new instances with the new image.

6.3 Experimental Setup

In order to evaluate the proposed pipeline together with the selected tools, a simple microservice application was built, with one service that is continuously delivered using this pipeline. Additionally, a second pipeline was created that is constructed the same way as the one proposed above, only instead of unikernel it delivers a *Docker* container. This second pipeline was used for comparison.

Component	Description
CPU	Intel®Core^{TM}i7-6500U CPU @ 2.50GHz × 4
Memory	16 GiB
OS	Ubuntu 18.10
Kernel	Linux 4.18.0-25-generic.x86_64
Hypervisor	Virtualbox 6.0.12r133076
Container	Docker version 19.03.2, build 6a30dfc

Table 6.1: All the experiments and analysis were performed on a *Lenovo Yoga-900* Notebook with *Ubuntu 18.10* as OS.

Environment

All the experiments and analysis were performed on a *Lenovo Yoga-900* Notebook with *Ubuntu 18.10* as OS. The full configuration and specifications of the environment is listed in table 6.1.

Unik supports different unikernel implementations and target platforms that can be used. Currently not every unikernel is fully supported on every platform, because the framework is still under development. After trying different approaches, *rumprun* is used as unikernel and *Virtualbox* as virtualization platform. This combination supported *nodejs* as language and proved as most reliable.

The *Unik Hub* and the *Docker* registry were both self hosted. While the *Unik Hub* used additionally an *AWS S3* storage in the eu-central-1 region, the *Docker* registry stores all images locally.

Jenkins was running locally as a *Docker* container and for the version control a public *Github* repository was used.

6.3.1 Best Practices

An application is needed in order to properly compare unikernels with containers from the perspective of cloud services and to build a continuous delivery for both of the virtualization technologies. The aim is for the application to align with the best practices of a microservices application that is designed to be deployed to the cloud.

Hence, first existing best practices for microservices and specifically for cloud native applications, need to be discussed and evaluated. Decisions need to be made, about which best practices apply for the use case ad hand and how they are applied. Only than, the application can be implemented based on those decisions.

Microservices

Microservice is a broad term and was first introduced at a workshop in 2011 [35]. In general, every application that splits its functionality into small, independently running services that communicate with each other to reach a desired outcome, can be considered to correspond to the architectural principle of microservices [36, 37].

James Levis and Martin Fowler tried to specify the term a bit in more detail and came up with the characteristics of a microservice architecture and its development process. This characteristics are based on observations of the development process of many different microservice applications, the prevalence of this characteristics mean they are well proven and does make them good best practices [35].

In the following the characteristics of microservices, as defined by Levis and Fowler, will be discussed and the relevance for the use case at hand, will be described.

Componentization via Services Every good system architecture structures the software in multiple components with clear boundaries. While a monolithic application does this for instance in the form of libraries that are linked into the application, a microservice architecture does this in the form of services. Services are out-of-process components that run independently and communicate via web service requests or similar ways. This way, services are independently deployable and if there are changes in only one component, there is no need to redeploy the whole application.

Unikernels are well suited as platform for small services, that are independently running. This allows to leverage the advantages of unikernels, like small resource footprint, fast startup time and strong isolation. The example application developed to evaluate the proposed deployment pipeline will consist of different services that are lightweight and loosely coupled.

Organized around Business Capabilities To develop a well designed microservice application it is also necessary to organize the teams the same way. The teams are organized around business capabilities with all required skills represented. This is in contrast to the classical way, where teams are organized around skills, i.e., there is a UI design team, a database team, and so on. Conway's Law states that an organization that designs a system, will produce a system whose structure is a copy of the organization's communication structure. So according to this, an organization that is organized around business capabilities will produce a microservice application where each service represents a specific business case, which is a good structure for microservices.

Although this is an important characteristic, that allows organizations with multiple teams to work efficiently and produce clean, maintainable code, it is not relevant for this study, since the development will be done by only one team, mainly consisting of one person.

Product not Project Applications are often seen as a project, where after the software is delivered it is handed over to maintenance and the project team moves on. The better way is to see them as products and the team that builds it, is responsible for it during the whole live-time of the product. This comes close to the DevOps movement, where the teams also contain operations skills and are fully responsible for the application in production.

This is again a purely organizational characteristic, with the purpose of giving the development team more responsibility for there application, also in production, to encourages them to invest more time in maintainability and to keep the whole application live cycle in mind. This in turn reduces the time to market. Since the application developed for this study, will never become a product and is solely intended to evaluate the proposed deployment pipeline, this characteristic does not apply.

Smart endpoints and dumb pipes The applications aim to be as decoupled and as cohesive as possible. The communication is done either trough RESTish protocols or trough a lightweight message bus. Unlike with Enterprise Service Buses (ESB), which often contain complicated logic for choreography, transformation or applying business rules, with the microservice approach all the smarts lies usually with the endpoints, that is the individual service, and the communication platform is dumb.

The example application will be implemented according to the RESTfull principles and therefore applies this characteristic as well.

Decentralized Governance Independent components bring the advantage that there is no need to use the same technology stack for everything, instead we can use what ever is best suited to the specific use case of a service. The same is true with standards. Instead of enforced standards from a centralized governance entity, that all teams have to comply with, teams produce tools that solve a specific problem. Other teams can take use of this tools, or choose another approach if it fits their needs better. In-house open source is often used to encourage this behavior. So can one team take use of the battle tested code of another team, without the explicit enforcement of specific standards.

Another organizational principle, that is important for bigger organizations with multiple teams. For the project at hand it is not relevant, since it is an academic project, with no explicit governing body and only one team.

Decentralized Data Management This addresses the problem that the conceptual model of the world differs between different parts of a system. Ever service has his own view of the world and needs only to save the data it really needs. This way we can easily use different ways of data management, always the one that is best suited for the specific service. The draw back here is consistency. Traditionally are transactions used to guarantee consistency. However are distributed transitions difficult to implement, thus generally a transaction less approach is used. This prevents temporal coupling between the services, but introduces the necessity of compensation mechanisms, because of the only eventually consistency of the data, between the different services.

In order to keep the example application simple and since the data management and consistency is not related to the underlying platform, for this study only one database is used. This does not necessarily violate this characteristic, since only one service of the application handles data that needs to stored.

Infrastructure Automation The rise of infrastructure automation techniques is important for microservice applications. Manual deployment of an application always bears the risk that something breaks because one step is done differently than the last time. This risk increases with more independently deployable components of a system. With many independent services, that get continuously delivered on a daily basis, it is not feasible anymore to manage the deployment manually, this makes the heavy usage of automation necessary.

This is of course the most relevant characteristic for this study, since an automation pipeline is evaluated.

Design for Failure With multiple independently running services, that often communicate over the network, the unavailability of one or more services at any time is much more likely as with one monolithic application. At the same time multiple service can be more resilient towards failure. A failure in a monolithic application is likely to cause the full application to crash, while in a distributed system only one service will be temporarily unavailable. Thus it is important that the system can handle it transparently, if a service is unavailable. Well implemented error handling is key to enable this resiliency. Extensive monitoring is important, it shows the current state of the system, helps to identify sources of failure and allows for automated restart of failed services. To protect services to be affected, if other service are not available, different techniques are used, like asynchronous calls, caching or circuit breakers.

Although this is an important topic, for this study we are more interested in the operational aspect, like monitoring and automated restarting of failed services, and less in the logical aspects, like caching.

Evolutionary Design The microservice principles come from an evolutionary design background. The breakdown of an application into independent services allows to apply changes faster and more often. This is because it is not necessary to redeploy the whole system only because of a change in one component. Together with automation, this allows to release changes almost immediately.

The evolutionary design principle not rely applies, in the case of this study, since the example application is only developed for testing purposes and only used for a short time.

This characteristics are a guide for organizations that want to produce well designed microservice applications. They show that not only a good design, but the whole structure and the philosophy of the organization, influences the development process and the outcome of a project. The characteristics are high-level and specified for bigger teams, many of them do not directly apply to the development of the example application, developed for this study. However, it is important to evaluate all characteristics and

Nr	Factor	Description
I	Codebase	One codebase tracked in revision control, many deploys
II	Dependencies	Explicitly declare and isolate dependencies
III	Config	Store config in the environment
IV	Backing Services	Treat backing services as attached resources
V	Build, Release, Run	Strictly separate build and run stages
VI	Processes	Execute the app as one or more stateless processes
VII	Port binding	Export services via port binding
VIII	Concurrency	Scale out via the process model
IX	Disposability	Maximize robustness with fast startup and graceful shutdown
X	Dev/Prod parity	Keep development, staging, and production as similar as possible
XI	Logs	Treat logs as event streams
XII	Admin Processes	Run admin/management tasks as one-off processes

Table 6.2: The 12 Factors by Adam Wiggins are best practices for cloud services [59].

principles, before one can decide which apply for the project at hand and which are not necessary for the specific use case.

Componentization via services, smart endpoints and dumb pipes, decentralized data management, infrastructure automation, and design for failure are the characteristics that we will embrace during the following design and development of the example application, used to evaluate the proposed delivery pipeline.

12 Factors

The microservice characteristics are generic and apply to a broad area in distributed systems. Although they provide already a good guideline for the development of a microservice application, additionally a detailed specification of what are the best practices for services that are developed for the cloud is needed, in order to implement a well designed example application, that is a good representative for an arbitrary cloud service. This allows to draw conclusion from this study, that can be applied to cloud services in general. Therefore, the 12 Factors for the design of cloud services are introduced and explained.

The 12 factors are principles defined by Adam Wiggins and are seen as best practices for applications that are designed as cloud services [59]. These principles are published on a website that itself is implemented according to this principles and its sources are available at Github [15].

Table 6.2 lists the 12 factors with a short description. In the rest of this section they are described in more detail, along with a short analysis on how the different concepts of containers and unikernels help to fulfill or even enforce these factors.

[15] https://github.com/heroku/12factor

I: One codebase tracked in revision control, many deploys There is always a one-to-one correlation between an application and a codebase. If an application consists of multiple codebases it is a distributed system, not an application. On the other hand, multiple applications that share the same code base are not allowed either. The reason for this is to prevent code sharing and thus dependencies between two applications, that should be independent. One application is deployed to multiple environments, starting from various testing environments up to the production environment. the different environments don't have to contain the same version, f.i. only stable and well tested versions of the applications get deployed to production, while the version on the development environment contains the latest commits to the code base.

Containers or unikernels do not enable this factor them self, rather the usage of a version control system, like Git, and a clean code base. The fulfillment of this factor however, will ease the usage of both virtualization techniques. The clear split of each applications allows to easily add a *Dockerfile* as well as the corresponding configuration file for a unikernel implementation, to the code base. This way the deployment configuration is versioned together with the the application code, which allows the deployment of every version of the application, at any given time.

II: Explicitly declare and isolate dependencies Dependencies have to be explicitly declared in a manifest file, so it is immediately clear which dependencies are needed for the application. Additionally, the dependencies have to be isolated during execution so that no implicit dependencies can leak-in from the surrounding system. For this dependencies management exist tools for most languages. *Maven* is such a tool for *Java*, it allows to specify the dependencies and its desired version in a manifest file called *pom.xml*. At build time the dependencies get pulled from a remote location and added to the application. This factor not only covers dependencies like application libraries, but also extends to system tools. So if an application relies on an external system tool, this tool as well has to be explicitly declared and installed.

The use of a tool to manage application dependencies, like third-party libraries, is a widely accepted best practice, enabled by the many dependency management tools that are available for every language. However, the extension of this practice to system tools and the whole environment is less common. Docker enables this through its *Dockerfile*, which specifies the base image and additional system tools, that need to be installed. While this already allows for a better fulfillment of this factor, the OS at the base of the container is still general purpose and the container shares its kernel with the host OS. Unikernels provide the ability to finalize the fulfillment of this factor, with the possibility to have fine grained control over each kernel library that is used and the full isolation from the host.

III: Store configuration in the environment The configuration consists of values that are dependent on the environment the application is executed in. This values change between different deploys, f.i. credentials or connection strings for databases. This configuration has to be stored directly on each environment, by taking use of the so called environment variables. Environment variables are language and OS agnostic standards and can easily be changed between deploys.

Docker containers and unikernels alike support this functionality. At startup time values can be injected into the environment variables, which allows to use the same image for different environments.

IV: Treat backing services as attached resources A backing service is any service that the application consumes over the network. This can be a database, other internal services, or external third party services. They all have to be accessed by using an URL or any other connection string, stored in the configuration. A key factor is that they can be exchanged without the need to make changes to the code base. Additionally, there is no distinction between an internal or an external service.

This factor is mainly solved through the architecture of the system. Containers do not provide any advantages over other environments, like bare metal or classical VMs, in regard to this factor. Unikernels, however, enforce this factor through there restriction to a single process. On other environments a backing service can just as well be installed as an additional process on the same environment, which adds complexity to the environment and might lead to problems when the backing service is exchanged. By taking unikernels as deployment environment, there is no other possibility than to have a unikernel per service and due to there strong isolation the service have to communicate over the network.

V: Strictly separate build and run stages The build stage generates a bundled executable of the codebase, called a release. This release can then be executed in the run stage. Changes, directly on the release or during the run stage, are not possible because they can not be propagated back to the codebase. Hence, for every change there must be a new release. Additionally, every release needs a unique identifier.

Both, containers and unikernels provide this separation. In the build phase an image is created, which subsequently can be deployed to various systems. However, the restriction of modifications to an image in the run stage is not enforced by containers. Container images, being full fledged OS environments, allow for changes to a running instance even if it is in general considered bad practice. Unikernel images on the other hand are immutable by definition, i.e., it is simply not possible to apply changes to an image after it is built.

VI: Execute the app as one or more stateless processes Processes of a 12-factor application are stateless. Stateless means that the process is not allowed to hold information about a certain state over the span of multiple request, especially if this influences the response of a fallowing request. Every data that needs to persist across multiple requests must be stored in a backing service, like a database. This allows to run multiple instances of an application, where each request can be routed to any instance.

It is also important that a restart or a redeploy can be performed without the fear of the loss of data stored in memory.

This factor is purely architectural and can neither be supported nor its violation prevented, by the usage of a certain virtualization technique. However, if a service does not need to fulfill the 12 factors, f.i. a backing service like a database, that needs to persist data, both, containers and unikernels, support this via volumes. Volumes are a way to store data over multiple deployments. After a new deployment all data generated by the application would be lost otherwise.

VII: Export services via port binding Each service binds itself to a port and listen for incoming requests. It has to be completely self-contained, e.g., so that it is not possible to inject a webserver into the execution environment during runtime to create a web-facing service. The app should rather export the webserver as a service via port binding. This is typically implemented by adding a webserver library directly to the application. This way the developer can access the service directly via the process port, while in deployment, a routing layer handles the forwarding of requests to the correct process port.

Both, containers and unikernels isolate services from each other. Through this isolation the communication over the network comes naturally making them self-contained. *Docker* containers as well as some of the more mature unikernels, like *OSv* support also the configuration of port forwarding rules, through there respective tools.

VIII: Scale out via the process model The process model from factor VI with its statelessness almost automatically enables this. The application can be scaled by adding more instances of the same application.

For *Docker* containers exist multiple orchestration tools that manage the scaling of instances. Although unikernels support this as well, sophisticated tooling is still missing, which would allow to automate the management of larger clusters.

IX: Maximize robustness with fast startup and graceful shutdown The startup of the application needs to be fast, to provide more agility for releases and scaling. Furthermore, it has to be able to be shut down at any time without data loss and additionally be resilient, so that uncontrolled crashes of the service it self or of a resource can be handled transparently.

Th grace full shutdown and the resiliency must be handled by the application logic, the underlying environment has little possibilities, since it highly depends on the application. However do already containers provide a significant faster startup time than, f.i. traditional VMs. Unikernels are able to reduce the startup time even further, which makes them an ideal candidate for just-in-time instantiation and fast scaling of an application.

X: Keep development, staging, and production as similar as possible Traditionally, there exist huge gaps between the environments. Examples are i) time gaps: code takes up to weeks or months to go into production, ii) the personal gap: the developers who write the code, are not the same as the operators that run it in production, and iii) the tool gap: different tool stacks are used in development than in production. These gaps need to be reduced as much as possible. Code has to be deployed immediately after writing it; automation aids that. The same developers that write code has the responsability for bringing it to production and monitor its behavior in every environment. The tools need to be always be the same; even small differences in the tools and backing services potentially leads to code that runs fine locally but fails in production.

The close the tool gap between the development environment and production is one of the main reasons to use containers or unikernels, enabled by the ability to create an image that contains the application as well as its full environment. This allows to have conditions as similar to production as possible, on the local development environment as well as during automated tests and on staging environments.

XI: Treat logs as event streams Logs are aggregated, time-ordered events from the output stream of the application. In stead to be written to a file, they have to be written unbuffered to `stdout`. Applications don't have not be concerned on how the logs are handled, because this can be different for each environment. Developers are able to locally monitor the stdout in the foreground while they are working on the code, while in deployment, logs can be collected and e.g., sent to a centralized logging system.

Docker containers and unikernels provide there logs as stream. This stream can be handled trough configurations in the environment. Many log aggregation tools have good support for *Docker* containers, while unikernels are not yet support so natively. However, this only means that additional configuration for the tools is needed and with growing popularity of unikernels, many maintainers will add better support for them as well.

XII: Run admin/management tasks as one-off processes Besides the regular business of the application, developers often wish to run admin on-off administration tasks on the application, like f.i. a database migration. They need to be performed in an identical environment as the regular processes of an application and use the same codebase and configuration as the release.

Containers still contain a full OS, this means administrators can directly connect to the container via `ssh` protocol. This enables them to perform arbitrary maintenance tasks on the system. Unikernels do not allow this, since they are a closed system and do not provide the full OS functionality (e.g, a shell) which would enable this. In the case such maintenance tasks are necessary, they must be explicitly implemented, together with an API, which allows to trigger those tasks.

After analyzing the 12 factors and compare *Docker* containers and unikernels in their ability to host such 12 factor applications, it appears that both technologies are equally

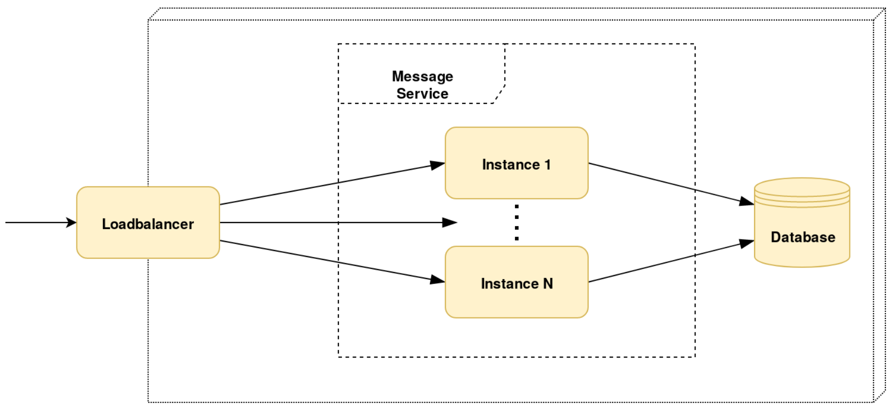


Figure 6.4: The simple microservice application consists of a scalable service, a loadbalancer and a database.

well suited for this use case. There are minor differences like the small size and better isolation of unikernels or the easier maintainability of containers. Unikernels are more restrictive and often leave no choice than to follow the 12 factors. This might be an advantage, but it also decreases flexibility. Overall, it depends more on the actual implementation which technology provides a better fit.

6.3.2 A Microservice Application

The aim is to develop a distributed application and build an automated deployment while taking in account the twelve factors and the characteristics of a microservice architecture. The application will be build as *Docker* container, as well as *rumprun* unikernel with the *Unik* framework. This allows for a thorough comparison of both technologies, containers and unikernels, in the context of cloud services.

The high level design of the microservice application is depicted in figure 6.4. The main part is a small messaging service, that can be scaled as needed. The service uses a mongodb database to store its messages, all instances of the service share the same database. A loadbalancer transparently distributes the requests between the instances of the service.

Messaging Service

The core of our microservice application is a RESTful service that receives and stores messages, providing three endpoints:

GET /api/messages To get all currently stored messages as json object, each together with timestamp and id

GET /api/messages/:id To get the message with the id specified as parameters

POST /api/messages To stores a new message

The service is implemented with *nodejs* and *express* as frameworks using *typescript* as programming language to ensure type safety. To fulfill *Factor I* the service has its own code base and is stored in a git repository. Dependency isolation is performed by using the *nodejs* dependency manager *npm*. All dependencies are defined in the file `package.json` and installed in the source tree under `node_modules`. The file `package-lock.json` makes sure that the versions stay exactly the same even if the dependencies are reinstalled on a new system. This fulfills *Factor II*.

The service automatically registers with the loadbalancer on startup. The address of the loadbalancer can be specified via an environment variable. The same is possible for the address, user, and password of the database connection as well as for the port the service is listening on. This is in alignment with *Factor III* and *Factor IV*, because all configuration is stored on the environment and the database and loadbalancer are backing services that are configured as resources.

Factors VI, VII and *VIII* are ensured through the statelessness of the service. Messages are the only data that have to be stored permanently. Since they are stored in an external database, which is shared by all instances, statelessness is ensured. This facilitates scaling of the application. Every instance listens on a different port while the loadbalancer manages forwarding of requests to the appropriate one.

A part of *Factor IX* is implemented by the service itself. Uncaught errors, as well es termination signals, are caught globally and a graceful shutdown is initiated, which allows for open requests to be finished and clean up operations to be performed, e.g., unregistering from the loadbalancer. The rest is covered by the virtualization framework that wraps the service, i.e., *Docker* and *Unik*.

In alignment with *Factor XI*, the logs are simply written to `stdout/stderr` and the virtualization framework is responsible on how they are delivered. *Factors V* and *X* are fully covered by the virtualization frameworks and the deployment pipeline. Due to the simplicity of our application, no admin tasks are necessary, hence no special handling according to *Factor XII*.

Loadbalancer

The loadbalancer has the purpose of distributing load between different services. Independently of how many instances are currently running and at which location and port they can be reached, the caller always uses the same interface, i.e., the one of the loadbalancer. The loadbalancer knows every instance of the service and its current address and port. This way, the request can be distributed transparently between all instances and adding or removing of instances is always possible.

Various loadbalancer implementations are currently available. There are well established ones like *haproxy*[16] or modern ones like *traefik*[17]. Most of them support dynamically adding and removing of services, which is an important feature for this application setup. However, this is often a premium feature which is not included in the free version, or it is only supported for specific technologies. *Docker* containers are well supported, coming with functionality like auto-discovery of new containers. For other technologies, where no direct integration exists, more configuration is needed. Sometimes even additional tools, like external key-value stores.

The same is true for orchestration frameworks like *Kubernetes* which often have loadbalancing features built-in. Many of those were build to orchestrate containers specifically. Only slowly support for other technologies is added, like *Kubernetes* with its *CRI Interface* [39]. Although it is expected that most of orchestration frameworks will support unikernels in the future, currently it is not trivial to set them up for other technologies than containers.

To keep it simple and to have the same conditions for both unikernels and containers, a simple loadbalancer was implemented, using *nodejs*. It listens on two ports. One port is used by the service instances to dynamically register and unregister itself. Requests to the other ports are forwarded to the instances in a simple round-robin way. However, a minimal health check is implemented: if an instance is not reachable by the loadbalancer, it gets removed from the list and the request is forwarded to the next instance in the list.

Database

In order to implement a microservice application in accordance to the 12 Factors and to keep the application stateless, it is necessary to store everything that needs to be shared between the instances in a dedicated backing service. In this case the messages need to be available to all instances, even after an instance is rebooted.

For this purpose, the *Mongodb* database is used, which is shared between all instances. This database was chosen because it integrates well with *nodejs*. Due to its NoSQL nature, it can be used without the prior specification of a schema.

6.4 Evaluation and Results

During the development multiple observations were made and issues occurred, both regarding unikernels in general and the *Unik* framework in particular. This allowed to take conclusions about the state of unikernels and especially their feasibility to run cloud services in production. The results are put in contrast to *Docker*, which is currently the most commonly used technology for this use case.

[16]`http://www.haproxy.org/`
[17]`https://traefik.io/`

6.4.1 Implementation and Build

The continuous delivery pipeline starts with a developer who pushes changes to the code base, which triggers a build. Before the developers can create features and push them to the code base, they need to make general decision about the project, e.g., which language and tools they will use. After those decisions are made, each developer can start to create features, using the selected language and tools.

Language selection

Docker has a vast variety of base images to choose from, built upon all kinds of different technology stacks and languages. In the case where no base image exists, that fits certain requirements, it is easy to extend an existing one with little additional tools needed.

Such images have a layered structure. Every layer adds tools or configurations to an existing image, which results in a new image. This brings maximal reusability and due to the fact that most images are based on common Linux distributions, every tool or application that can run on this platforms, can also run inside a container without modifications. Hence, there are virtually no restrictions on languages, dependencies and tools that can be used to create an application that is be able to run in a container.

With most unikernels this is different. They often support only one language. For instance *MireageOS*, which only supports *OCaml* as language or *IncludeOS* which only supports *C/C++*. Other implementations, like *OSv* or *rumprun*, however do support multiple languages. So unlike with containers, a developer has first to decide which unikernel implementation needs to be supported and selected, based on the used language.

So unless one unikernel implementation cuts out all others, which is only possible if it supports all languages, a tool like *Unik* is crucial to standardize the packaging and execution of unikernels. With this framework developers do not have to learn to build and operate a new unikernel solution every time they want to use another language. They can always rely on the same build tool and simply select a targeted unikernel for each built. This brings unikernels a step closer to compete with containers on a larger scale.

Unik officially supports multiple programming languages, target platforms, and unikernel implementations. This brings the advantage that a developer can always use the same tool to package their application as a unikernel and can still use different languages. They just have to select an appropriate unikernel implementation when building the package.

As of now the supported possibilities are somewhat limited, i.e., there is no support for every combination of language, unikernel and provider. In fact, faced with a few restrictions, e.g., which language is used, the options are reduced significantly.

For this study *typescript* was selected as language for the application, more specifically the *nodejs* framework. *Node.js* is currently one of the most used frameworks for simple microservice implementation. Given this restriction, *OSv* and *rumprun* are the two unikernels that *Unik* currently supports for this language.

Building and Packaging

Before an application can be package by either using *Docker* or a unikernel implementation, it has to be build. The build process it self is independent of the targeted virtualization technology and is fully performed by language tools like compilers.

In this specific case *Typescript* is used as language, which is a dialect of *Javascript* with type-safety build in. Although *Typescript*, just like *Javascript*, is an interpreted language, it is good practice to transpile the code to native *Javascript* for increased performance. The resulting *Node.js* application is then ready to be packaged either as *Docker* container or as unikernel.

In order to package the application as container, a so called *Dockerfile* is created. Among others, it contains information about the base image used, additional tools to be installed, and instructions on how the application is executed. The build command provides additional options, ranging from the name for the created image to specific instructions about the CPU and memory usage of the packaging process.

The process to package the application as unikernel highly depends on the unikernel implementation. However, most of the unikernels provide boilerplate base images for common use cases, so developers can use those and build their application on top of them. Additionally, tools and scripts are available which can be used to create the image. How the tools and the level of abstraction work differs between the different unikernels. *OSv* is organized similar to *Docker*. Its images are constructed as layers, that can be reused and a so called *Capstanfile* defines how the application can be added on top of a base image. *rumprun* is constructed differently, there are no layers that can be reused. Hence, every unikernel has to be build from scratch. To simplify this, there exists a repository which contains the boilerplate code to build images of many different languages and frameworks. These base images can be used to build an application. Furthermore, additional scripts are available in this repository, that help with the build process. Nevertheless, the level of expertise required, is already significantly higher than for *OSv*.

This are just two examples. Different unikernel implementations follow different approaches for the build process and thus require different levels of expertise. A framework like *Unik*, that provides a wrapper and a higher abstraction for the different unikernel implementation, has the ability to reduce the complexity for developers significantly. It makes it much simpler to build an application as unikernel and brings it much closer to the way *Docker* builds containers. Which paves the way, for developers to use unikernels the same way they use containers for their services.

As of now *Unik* only provides few configuration options and relies mainly on parameters added to the build command. There is first support for a manifest file in the code base, but the options in that file are very limited and differ between which unikernel is used. This is likely to change and the configuration will more heavily rely on a manifest file and be equal for all unikernels.

When creating the image in both cases, *Docker* as well as unikernels, it is important that the targeted platform is known. For *Docker* it depends if the platform is using an *ARM* or *x86* processor and if the host is *Linux* or *Windows*. Based on this, the appropriate base image has to be selected.

For unikernels the targeted hypervisor has to be known when building an image. The *Unik* framework simplifies this. The target platform can simply be specified as an option to the build command and the framework adds the platform specific modifications to the image. The build is performed in two steps, first the raw unikernel image is created. Based on this raw image, a bootable image for a specific provider is created. This brings the possibility to shift the provider specific modification that are performed under the hood, from build time to startup time. This way an image can be build that supports all platforms. Of course the impact on the startup time has to be considered, which is after all an important factor. Nevertheless, show resent efforts of *Docker* to create true multi platform images, that this is an important topic.

For this study, the previously build application is packaged as *OSv* as well as *rumprun* unikernel, by taking use of the *Unik* framework as a build tool. However attempts to package and run the application using *OSv* failed. The problem is related to the way the *Unik* framework is constructed. It has a highly modular architecture, which allows to plugin a modules in order to add support for another unikernel or an additional provider. A module that is responsible to package an application of a certain language, as a specific unikernel, is called compiler. The compiler for the *OSv* unikernel takes use of the *Capstan* build tool, which is the official build tool of this unikernel. Since this module was implemented, the API of the *Capstan* tool changed, but the changes were not added to the module. It causes *Capstan* to produce an error, which in turn causes the whole packaging process to fail. This leaves *rumprun* the only unikernel that is currently supported by *Unik* and is able to package a *node.js* application without errors.

This is one issue that highlights the challenges of a framework like *Unik*. It can only be successful with an active community that keeps maintaining the different modules of the framework, in a way that the integration to the different unikernel implementations and target platforms stay current.

The next issue that occurred was caused by the usage of base images. Since *Unik* is simply a wrapper for different unikernel implementations, also the base images provided by each unikernel implementation are used. This in turn means *Unik* is dependent on the different unikernel providers and communities to maintain the base images. So is currently the latest supported *node.js* version for *rumprun* with 4.3.0 quite old. Additionally is this version number only specified in the compiler module for *rumprun* and *nodejs*, but there are already preparations in the code to specify the version in the *Unik* manifest file. After the cause of the problem was identified, it was simply resolved by taking use of the transpiler, which allows to specify a targeted *node.js* version. Only one logging library had to be removed, because it was not compatible with this old *node.js* version.

While the incompatibility issues with the *OSv* compiler and the version issues with the *rumprun* base image are inconveniences, they can be explained by the early phase in which the *Unik* project and many unikernels in general are. However, they are not related to the concept of unikernels in general. The following problem that occurred with one used library, on the other hand, is a common problem with unikernels.

For the communication with the database the popular library *mongoose* was selected. Unfortunately when running the service with this library as a *rumprun* unikernel, it led to an obscure memory error located in the code of the unikernel implementation itself. This is always a possibility with unikernels, when a library relies on functionality that a traditional OS supports but not the unikernel. By changing to the native *mongodb driver* for *nodejs* as library, this error was resolved. It is expected that with increasing popularity of unikernels, maintainers of third party tools and libraries will start making their code compatible to unikernels as well and this kind of issues will become less common.

Similar issues occur with whole tool stacks, if they are packaged as unikernel. For instance, if they rely on multiple processes, they are not compatible with the unikernel paradigm. In this study this was the case with the *mongodb* database, which was not possible to run as a unikernel. Unikernel maintainers try to overcome this issue by providing custom builds of this tools, with patches that change the implementation in a way that it is compatible with unikernels. For instance instead of processes only threads are used, which is possible in unikernels.

Debugging

Debugging, as the process of error analysis, is an important part of the development cycle and many tools exist that support the developer with this task. One reason to use containers or unikernels, is the advantage that the application is always executed under the same conditions, which reduces errors caused by discrepancies between the development and the production environment.

Docker allows to connect directly to the container via ssh protocol. This is possible, because the container still holds a full OS with all functionalities, f.i., a shell. Thus, developers can debug application directly inside the containers and inspect the whole environment. Additionally delivers *Docker* all logs via stream, which makes it possible to collect them and inspect them at anytime.

Unikernels on the other hand, being a closed immutable system, do not allow the connection to an instance and the execution off arbitrary commands inside of it. This makes it notoriously difficult to debug unikernels with the classical tools and approaches, developers are used to, which is one of the major pints of criticism. Unikernel projects, try to solve this problem, by adding functionality to add a debugger directly to the unikernel or adding an API to the unikernel that delivery the logs.

Unik provides both, a flag that tells the compiler and the provider, that a debugger has to be attached and commands to tail the logs of a unikernel. Since this functionalities highly depend on the unikernel implementation, it is up to the compiler and provider modules to implement them. The retrieval of logs is generally well supported, while the debugging option is currently only supported by provider of the *QEMU* hypervisor and few unikernels.

The *rumprun*, f.i, has the logging functionality not build into the kernel, but it is up to each base image to provide this functionality. So consist the base image for *node.js* of certain bootstrap *javascript* files that perform specific operations before starting the main program. In order to provide the logging functionality is the `stdout` and `stderr` output redirect to a buffer and an endpoint is added which delivers the content of this buffer. This way the logs can be retrieved via HTTP. The *Unik* framework makes use of this endpoints, to allow the retrial of the *rumprun* logs via *Unik's* API.

During this study certain issues were detected with the way the logs are delivered. *Unik* assigns a public address to the unikernel, but only after the startup is fully completed. This leads to the problem that no logging can be retrieved, if an error occurs before the main program is fully started, because the endpoint that delivers the logs is not available yet. This made it hard to investigate such an issue. As a workaround the *Virtualbox* tools had to be used directly to start the unikernel in the foreground. In particular the `vboxmanage` command, which provides CLI functionality to manage *Virtualbox* instances. The same tools is used by the *Unik* provider for *Virtualbox* to manage the instances. The tool allows to start an instance in the foreground, this way the logs can be seen directly in the console. This can be solved by either assigning the address earlier in the startup process or by adding a debug option, to allow the start of the unikernel in the foreground, without the need to rely on third party tools.

6.4.2 Delivery and Deployment

After a new image is created and all automated tests are passed successfully, the new version can be released. This involves multiple parts. First the image must be made available for execution, only then one or multiple running instance of it can be deployed. After instances are deployed they must be operated and maintained. In all of this steps automation and abstraction becomes more important, with the amount of instances that are running in parallel.

Releasing

A released image of an application gets generally uploaded to a shared registry, from where it can be deployed to a server or made publicly available.

Docker provides this functionality through the *Docker Registry*. The registry it self is available as *Docker* image, so anyone can host such a registry for internal or public use. Alternatively exists the *Docker Hub*, which is the official public registry. It is the default registry configured in the *Docker* daemon and provides millions of images.

Unik aims to provide the same functionalities via there *Unik Hub*. The domain configured in the the daemon as standard location of the official *Unik Hub* is currently not available. In contrast to *Docker*, does *Unik* currently not provide a ready to use image of the registry either, but the full source code is available as open source at *Github*. However, the different components are not in the same state, and the whole system is not operational. Initial attempts to apply patches, to fix this issues and make the registry applicable for this setup, where quickly dismissed because it would have required a reimplementation of bigger parts of all components of the system, which would be out of scope of this thesis.

There are three parts that play together so that images can be pushed and pulled to and from a *Unik Hub*. The *Unik Hub* server, which has its own code base, the *Unik* daemon, which connects to this registry and a modified *AWS* SDK for *go*, which is used by the daemon. This three components seem to be all in a different state of development.

The SDK for *AWS* is a fork from an outdated version, which means the API for *AWS* changed in the meantime and calls to it by using this fork fail. It is fair to assume that this is only a temporary solution, because it is not sustainable to maintain a permanent fork from the SDK in the long run, only to have this small modifications. More likely is that a library will be implemented that wraps the SDK. Another possibility is that the whole communication with *AWS* is performed by the *Unik Hub* so that the daemon does not need the SDK at all.

In the daemon implementation specific parameters are hard-coded for a public *Unik Hub*. Additionally, there are currently changes in the *Unik Hub's* server API, which are not yet incorporated in the daemon.

Deployment

The daemon of *Docker* as well as the one of *Unik*, are not only responsible for building an image, but to execute and manage running instances as well. From a consumer perspective the daemon of *Unik* is quite similar to the one of *Docker*. All of this plays into the goal of making the transition from containers to unikernels as smooth as possible. Both daemons act as a REST server and executes the commands send from the clients. Both project ship a standard CLI client together with the daemon. Additionally is the REST API open, which allows for third party tools to integrate with the daemon easily.

Due to its maturity the *Docker* daemon provides many options and fine grained control over the containers it is managing. This was not always the case, when *Docker* was fist released its API was much more limited and only grew over time. The *Unik* is still in its early phase and the daemons API is quite limited.

However, the approach to supported target platforms is very different for the two platforms. *Docker* supported for a long time only Linux containers and the x86 processor architecture for the host. Only in the last couple of years support for ARM architecture and Windows containers was added. *Unik* on the other hand aims to provide support for as many unikernels and platforms as possible, right form the beginning, which adds an additional layer of complexity and is visible in the architecture of the daemon.

The architecture of the *Unik* daemon consists of three parts, the API server, the compilers and the providers. For every combination of language and unikernel exists a specific compiler and for every target platform exists a provider. The API server hands of a build command to the compiler of the specified unikernel, which constructs the raw image. After that, the raw image is passed on to the provider implementation of the specified target platform, which creates a bootable image for the specific platform.

The support for different target platforms, also makes the running and managing of instances more complex. The unikernels instances run on hypervisors which are placed on top of the host, or even at a remote location, similar to classical VMs. For every configured provider the daemon starts one special unikernel, which is called *Instance Listener* and manages all instances that are running on the same hypervisor and the daemon collects and aggregates the information from these instance listeners or forwards commands to them.

During this study multiple problems were discovered, which are caused by the instance listener, who's implementation is partially still very rudimentary.

One problem is that the instance listener manages and holds the state of the instances in a *Json* file on the host. Every time a new operation is performed, like creating a new instance or stopping one, this changes are applied to the file. This can lead to a divergence between the real state of the system and the state represented in this *Json* file. For instance, when a unikernel crashes it is sometimes not detected by the listener, which leads to an error when a new instance with the same name is started, because the listener assumes that an instance with this name is already running.

A similar issue occurs when the daemon crashes or is terminated. After a restart and with a `state.json` file already existing it simple gets read and assumed to be in the correct stated, without checks for the real state of the system. This can be a vulnerability because the file can be edited manually to bring the instance listener into a wrong state. Additionally, a manual clean up is sometimes necessary, to bring the system back in a consistent state.

Another problem detected, is that the instance listener gets stuck in a deadlock after running for a while and performing multiple operations. As of writing this thesis the cause of this issue could not be found yet.

Although this are breaking issues for running *Unik* in production, the project does not claim to be production ready and for this experimental setup this problems were manageable. It is fair to say that there is still way to go for he *Unik* daemon to become as reliable to manage unikernels, as the *Docker* daemon is for managing containers. But it goes in the right direction to make the operating of unikernels on a bigger scale, as simple as it is with containers.

Storage Management

Volumes are a concept for containers, to specify storage locations that can be mounted to the container at startup. The advantage of volumes is, that data can be stored over the extend of the live time of one container.

Unik has copied this concept and has already good support for volumes. Similar to containers, it is important to have this functionality with unikernels, since data saved inside the unikernel will be no longer available if a new version of the unikernel is deployed. Currently volumes have to be created for a specific provider and can than be added to an instances on startup. Volumes can only be added to mountpoints that are specified during the build of the image and if such a mount point is specified a volume has to be added, otherwise the startup will fail.

This is much more restrictive and explicit than the volume management with *Docker* and on the same time there are little configuration options. *Docker* for example allows the specification of many different storage drivers, for all kind of underlying file systems.

Nevertheless is the current implementation of volumes quite reliable and with *Unik* gaining more maturity additional options can be added and the whole process can be made more dynamic.

Networking

Docker has a rich set of networking options which allows a fine grained control over how the containers communicate with each other and with the outside world. Trough the pluggable networking subsystem, it is not only possible to add standard network plugins from *Docker*, like `host` networks for local communication or `overlay` networks for communication between containers on different hosts. It is easily possible to use third party plugins for specialized network stacks. This rich functionality evolved over time and in the beginning also Docker only supported few networking options like `host` or `bridge` networks.

Networking is a complex topic and it requires good expertise to manually setup networks between different unikernels. *Unik* currently does not support the configuration of different networking options, but simple uses the default host network of each provider. For the state of the project this is currently good enough since it allow the communication of all unikernels on a host as well as the connection the Internet. For production use this is however not feasible. It is certainly necessary to create different closed networks for groups of unikernels as well as restricting the access to the Internet. For cluster orchestration it will be necessary to create `overlays` between unikernels on different host and providers.

Although there is nothing implemented yet, the modular structure of the unikernel framework can be a way to add support for many different network stacks. Networks can simply be added as an additional module, like compiler and provider.

Orchestration

Orchestration is an important topic when it comes to managing multiple distributed cloud services. Over the last years and with *Docker* becoming the defacto standard for containers, many orchestration tools incorporated *Docker* as a first class citizen or were developed specifically for this container solution. Starting from the minimal orchestration tool *docker-compose* over the cluster management tool *Swarm*, up to third party tools like *Apache Mesos* and *Kubernetes*.

For unikernels there is currently no native supported orchestration tool. This is likely to change in the future when framework like *Unik* becomes more mature and more stable. A tool like *docker-compose* can simply be implemented as an other client using the REST API of the daemon, the same is true for other more complex orchestration and clustering tools.

With *Kubernetes* becoming the leading orchestration tool, a good integration with a unikernel framework like *Unik* is key to make unikernels truly applicable also for bigger projects. In fact there were already initial attempts to add support for *Unik* to the orchestration tool and with the refactoring of *Kubernetes* and the construction of a clear interface for any kind of runtime implementation [39], unikernels are able to become real fist class citizens for *Kubernetes* by using the *Unik* framework.

No doubt that there is still a way to go for *Unik* to reach this goals, but it is certainly possible and only than it will become a real alternative for *Docker*, especially with bigger projects, often consisting of hundreds or thousands of services.

6.4.3 Summary

Unikernels have the ability to become an alternative to containers for hosting cloud services. The fact that in general unikernels images are smaller than container images and their startup time is even faster, is an advantage. As many studies already showed, they are also more resource efficient than containers, due to the fact that they don't have to rely on an underlying OS, but can run directly on a hypervisor and sometimes even on bare metal. The better isolation and the low attack surface makes a good case for them from the perspective of security.

What was missing until now and where there is still a lot of work and research left to do, is the automation both for build and deployment as well as for operating and orchestrating big clusters of unikernels. The fact that there are many different unikernels with different specializations adds even more complexity to this work.

In this chapter a continuous delivery pipeline was presented, that automates the build and deployment process of cloud services as unikernels. The pipeline was implemented on Jenkins with a custom plugin that takes use of the *Unik* framework [7]. The advantages of this framework are, that it provides a wrapper for many different unikernel implementations, which means the same pipeline can be used to build an application as different unikernels and for different target platforms.

In order to provide a good analysis of the proposed pipeline an example application was developed, by following the best practices of the microservice characteristics [35] and the 12 Factors [59]. This application was deployed with the proposed pipeline as a unikernel and additionally as *Docker* container for comparison.

The results show that the *Unik* framework is a good first step, by providing a common build tool for different unikernels. By making the interface similar to *Docker*, the transition is made easier for developer, even without deep knowledge of OS and unikernels. However, the implementation of *Unik* is still rudimentary and many necessary features like networking or a shared registry are missing or not in a working state. There is still a lot of work to do, in order to add support for more unikernels and target platforms as well as making the commands for the already supported ones more reliable and provide more fine grained controlling options. Also debugging is still a problem. A running unikernel is a closed box, which makes it hard to debug it, but this is necessary to investigate issues during the development. *Unik* might provided answers for this problem even tough the implementation is not finished yet.

Unik has the ability to standardize the way how unikernels are build, without limiting users to one specific unikernel. It makes it already now relatively simple to run a 12 factor application, even tough it is not stable enough for production.

The next big step that is necessary to use unikernels on a bigger scale for cloud services, is a reliable orchestration tool. Integrating a tool like *Unik* with *Kubernetes* would enable the orchestration of many different unikernels at once.

CHAPTER 7

Discussion

Since the unikernel concept was proposed by Madhavapeddy et. al. in 2013 [5], many different implementations emerged and many publications evaluated different use cases for unikernels and compared them to classical VMs and containers. Some might expect unikernels to replace containers in the near future, but just like containers didn't fully erase the need for classical VMs, will unikernel simple present an additional alternative to the existing technologies.

Many different areas exist, where unikernel might provided an interesting alternative. In this thesis cloud services were examined as one area, where unikernels might provide significant benefits. Concepts like microservices and Functions-as-a-Service (FaaS) lead to highly distributed applications, that often relay on untrusted and shared resources. The applications take use of fully automated orchestration tools, that use just-in-time instantiation and auto scaling to guaranty high availability. Traditional VMs are often to heavy for this use cases and containers, with their relative weak isolation, might impose security risks. this makes unikernels an interesting alternative that is worth to consider, since they are able to combine the advantages of both, traditional VMs and containers.

In the first parts of this thesis unikernels are compared to other virtualization technologies, like containers. For cloud services, performance and security are two important topics, hence each area has its own chapter, where existing literature is evaluated and summarized to provide a full picture about the advantages and drawbacks of unikernels, in contrast to other virtualization technologies. The decision to rely on existing literature for this evaluation, was based on the wast amount of existing research in this area, which lead to the presumption that a summary and comparison of existing research provides more value than yet another performance experiment of a specific unikernel implementation.

In the second part a continuous delivery strategy was proposed and implement, which allowed for a better evaluation of the unikernel concept for cloud services. The implementation relays heavily on an existing framework for unikernels, called *Unik* [7], in spite of its relative early state. This lead of course to some issues during the development. Nevertheless, its support for many different unikernels and target platform, provides advantages over the usage of one specific unikernel implementation and its often relative low level tooling. Due to the early phase of the whole unikernel area, there is lots of movement and it is not yet clear which unikernel implementation will persist and which will fade. It is likely that a framework will emerge, that provides an abstraction and a more developer friendly interface for unikernels, just like *Docker* for containers. *Unik* might not necessarily be the one to persist, but it is the only one currently availability that supports multiple unikernels and it provides interesting concepts on how such a framework might look like.

CHAPTER 8

Future Work

This chapter describes the future work that is still necessary, to make unikernels truly ready as a cloud platform for future applications in the industries.

In the last years extensive research has been done in the field of unikernels. Different implementations have been proposed and developed, showing different approaches to the unikernel concept. Most of this implementations are only experimental and for academic purposes, only few reached the state where they can be considered for real world use cases. Nevertheless, those implementations are an important step and provide the base for the extensive evaluation of unikernels, in terms of performance and security as well as to show the advantage of unikernels over classical VMs and containers in specific areas of cloud computing and IoT.

As important those first implementations were, the next step will be to bring some unikernel implementations into a state, where they can be considered for production environment. Some implementations, like the *OSv* unikernel are already in a good state, but new implementations will most definitely emerge that incorporate learnings from those first unikernels.

Another area where additional research and work is needed, is the building and orchestration of unikernels. Tools that abstract the complex process of unikernel building are essential to make them applicable for a broader audience of developers. These tools are also important to ease the creation of automated build and deployment pipelines. In order to manage large amount of running unikernel instances, new orchestration tools are necessary, as well as the integration into existing orchestration tools like *Kubernetes*. Only the automation provided by tools like this, will enable the real advantages of unikernels, like high scalability and just-in-time instantiation.

One promising tools is the *Unik* framework, which was also used for the evaluations in this thesis. It has the ability to enable the usage of unikernels for a bigger audience, especially in the industries, just like *Docker* did with containers. The framework is in an early state and there is still much work and research necessary to make it a full-fledged and stable build framework for unikernels. It is not clear yet, if *Unik* will succeed in its goal to become the main platform for unikernels, but it will definitely provide good learnings for future work in this area.

CHAPTER 9

Conclusion

The need for a specialized OS that can be optimized for a specific application, was already known for a long time. The first library OS implementations in the 1990s tried to tackle this problem, but they faced different problems, like the vast amount of hardware interfaces that needed to be supported and maintained. With the rise of virtualization and other technologies most of this issues could be resolved and unikernels emerged as the second generation of library OS, that are specifically developed to run in virtualized environments.

Over the last years many different unikernel implementation were developed and the concept was extensively examined and compared to different virtualization techniques like classical VMs and containers. The research showed that unikernels are able to combine the advantages of both, classical VMs and containers, while being secure and light weight.

In this thesis the advantages for cloud-services running as unikernels were evaluated and a continuous delivery strategy for unikernel-based cloud services was presented.

The thesis started with the essentials, which the reader needs in order to understand the concepts and terminology used in this thesis. A review of existing literature showed the different concepts and techniques used by an OS with respect to the various types of virtualization, containers and unikernels especially. The concepts of continuous delivery and microservices were described and a presentation of the state of the art and of related work was provided.

In the second part a thorough comparison between unikernels and other virtualization techniques was made, while keeping the focus on the usage in a cloud environment. The comparison is based on existing literature that provided results of experiments, where the different technologies were analyzed according to different metrics. Common metrics in this area are size, build time, startup time, security and performance. There is research that compares different container solutions with each other, container solutions with virtual machines and bare metal, but also container solutions with unikernels and different unikernel solutions with each other. This different results were put into context and additional observations and conclusions were provided. The results are divided into two chapters. The first one sets the focus on performance metrics, while the second mainly covers security concerns in a cloud environment. This evaluation showed the advantages of unikernels for cloud services and the need for good tooling to ease the transition for developers, that want to take use of this lightweight and secure alternative.

The third part presented the main contributions of the thesis. A continuous delivery strategy was presented and the implementation of a Jenkins plugin that eases the build and deployment of unikernels was discussed. Obstacles or major risks encountered during the development process were documented as well.

The development and evaluation of the results were performed in three steps. First, existing tools for continuous delivery and building of unikernels were evaluated and selected. In order to integrate this tools with each other, extensions and customization were developed. The results of this first step was *JUnik*, a general *Java* client library for the *Unik* framework and *Unik Builder*, a *Jenkins* plugin that takes use of this library. Second, a continuous delivery strategy for cloud services as unikernels was proposed. This strategy was used to develop a continuous delivery pipeline, by taking use of the previously selected and developed tools. Finally obstacles encountered during the development were described as well as the current state of unikernels and there tools evaluated.

This work lead to an assessment of unikernels in contrast to other virtualization techniques, in the face of today's need of highly scalable and lightweight cloud services. It showed the advantages of unikernels for the deployment of cloud services, but on the same time the flaws that are still present in the currently existing tools and unikernel implementations.

The research area of Unikernels is still in its infancy. There is a need for stable tooling that pave the way for unikernels in the industry, but it is likely that unikernels will disrupt the way services are operated in the cloud.

List of Figures

List of Tables

Bibliography

[1] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[2] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors Stephen," *SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 275–287, 2007.

[3] R. Dawson, M. Engler, F. Kaashoek, and J. O'T Oole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *SIGOPS Operating Systems Review*, vol. 1, no. 212, pp. 251–266, 1995.

[4] I. Leslie, D. McAuley, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 7, pp. 1280–1297, 1996.

[5] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library Operating Systems for the Cloud," *SIGPLAN Not.*, vol. 48, no. 4, pp. 461–472, 2013.

[6] A. Happe, B. Duncan, and A. Bratterud, "Unikernels for Cloud Architectures: How Single Responsibility can Reduce Complexity, Thus Improving Enterprise Cloud Security," in *2nd International Conference on Complexity, Future Information Systems and Risk (COMPLEXIS)*, 2017, pp. 30–41.

[7] solo.io inc., "Unik - The Unikernel Compilation and Deployment Platform," Accessed on: Feb. 23, 2019. [Online]. Available: https://github.com/solo-io/unik

[8] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 2nd ed. Pearson Education International, 2014.

[9] Intel Corporation, *iAPX 86,88 User's Manual*. Intel Corporation, 1981.

[10] Cljk, "Ring (CPU)," Accessed on: Feb. 23, 2018. [Online]. Available: https://de.wikipedia.org/wiki/Ring_(CPU)#/media/File:CPU_ring_scheme.svg

[11] G. Duarte, "CPU Rings, Privilege, and Protection," Accessed on: Mar. 10, 2019. [Online]. Available: https://manybutfinite.com/post/cpu-rings-privilege-and-protection/

[12] A. Madhavapeddy and D. J. Scott, "Unikernels: The Rise of the Virtual Library Operating System," *Communications of the ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[13] K. Yu, C. Zhang, and Y. Zhao, "Web Service Appliance Based on Unikernel," in *37th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 280–282.

[14] A. Kivity, D. Laor, G. Costa, and P. Enberg, "OSv—Optimizing the Operating System for Virtual Machines," in *USENIX Annual Technical Conference (USENIX ATC)*, 2014, pp. 61–72.

[15] Cloudius Systems, "OSv Homepage," Accessed on: Mar. 10, 2019. [Online]. Available: http://osv.io/

[16] Claudius Systems, "OSv Github Wiki," Accessed on: Mar. 10, 2019. [Online]. Available: https://github.com/cloudius-systems/osv/wiki/

[17] V. Jacobson and B. Felderman, "Speeding up Networking," Accessed on: Feb. 23, 2019. [Online]. Available: https://es.slideshare.net/networksguy/speeding-up-networking-3771560

[18] A. Kantee and J. Cormack, "Rump Kernels: No OS? No Problem!" *;Login;*, vol. 39, no. 5, pp. 11–17, 2014.

[19] Rumpkernel Community, "Rumpkernel Github Wiki," Accessed on: Feb. 23, 2018. [Online]. Available: https://github.com/rumpkernel/wiki/wiki

[20] VMware, "Understanding Full Virtualization, Paravirtualization, and Hardware Assist," Tech. Rep., 2007. [Online]. Available: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

[21] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, vol. 25, no. 5, pp. 483–490, 1981.

[22] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[23] R. Rose, "Survey of System Virtualization Techniques," Master's thesis, Oregon State University, 2004.

[24] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes, "A summary of virtualization techniques," *Procedia Technology*, vol. 3, pp. 267–272, 2012.

[25] A. Whitaker, M. Shaw, and S. S. D. Gribble, “Denali: Lightweight Virtual Machines for Distributed and Networked Applications,” in *USENIX Annual Technical Conference (USENIX ATC)*, 2002.

[26] bta and M. Gondim, “What is an application binary interface (ABI)?” Accessed on: Feb. 05, 2019. [Online]. Available: https://stackoverflow.com/a/2456882

[27] J. Pönisch, M. Kosler, and D. Schreiber, *Tagungsband: Chemnitzer Linux-Tage 2011*. Universitätsverlag Chemnitz, 2011.

[28] Docker Inc., “Docker Documentation.” [Online]. Available: https://docs.docker.com/

[29] Microsoft, “Windows Containers,” Accessed on: Feb. 23, 2018. [Online]. Available: https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/

[30] Portworx, “Portworx Annual Container Adoption Survey 2017,” Tech. Rep., 2017. [Online]. Available: https://portworx.com/wp-content/uploads/2017/04/Portworx_Annual_Container_Adoption_Survey_2017_Report.pdf

[31] P. Menage, P. Jackson, and C. Lameter, “CGROUPS,” Accessed on: Mar. 10, 2019. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

[32] J. Petazzoni, “Anatomy of a Container: Namespaces, cgroups & Some Filesystem Magic,” Accessed on: Mar. 10, 2019. [Online]. Available: https://fr.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon

[33] The Linux man-pages project, “Linux Programmer’s Manual: NAMESPACES(7),” Accessed on: Mar. 10, 2019. [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html

[34] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen, “The Structure and Value of Modularity in Software Design,” *SIGSOFT Software Engineering Notes*, vol. 26, no. 5, pp. 99–108, 2001.

[35] J. Levis and M. Fowler, “Microservices,” Accessed on: Feb. 02, 2019. [Online]. Available: https://martinfowler.com/articles/microservices.html

[36] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, today, and tomorrow,” in *Present and Ulterior Software Engineering*. Springer International Publishing, 2017, pp. 195–216.

[37] A. Balalaie, A. Heydarnoori, and P. Jamshidi, “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture,” *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.

[38] The Linux Foundation, "Kubernetes," Accessed on: Aug. 25, 2019. [Online]. Available: https://kubernetes.io/

[39] The Linux Fondation, "CRI: the Container Runtime Interface," Accessed on: Nov. 05, 2019. [Online]. Available: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md

[40] J. Cormack and R. Neugebauer, "LinuxKit - A toolkit for building secure, portable and lean operating systems for containers." [Online]. Available: https://github.com/linuxkit/linuxkit

[41] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than your Container," in *26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 218–233.

[42] Cloudius Systems, "Rapid VM builds - Capstan," Accessed on: Mar. 10, 2019. [Online]. Available: http://osv.io/capstan/

[43] M. M. Chinibolagh, "Development of Multicore Computing for a Cloud-Based Unikernel Operating System," Master's thesis, University of Oslo, 2016.

[44] D. Williams and R. Koller, "Unikernel Monitors: Extending Minimalism Outside of the Box," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[45] J. Amort, "Evaluating the Unikernel Concept for the Deployment of Software on IoT Devices," Master's thesis, Technische Universtät Wien, 2017.

[46] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," in *8th IEEE International Symposium on Cloud and Services Computing (SC2)*, 2018, pp. 1–8.

[47] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. Lightweight Virtualization: A Performance Comparison," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2015, pp. 386–393.

[48] A. Bratterud, A. A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum, "IncludeOS: A minimal, resource efficient unikernel for cloud services," in *7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016, pp. 250–257.

[49] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici, "Unikernels Everywhere," *SIGPLAN Notices*, vol. 52, no. 7, pp. 15–29, 2017.

[50] P. Enberg, "A Performance Evaluation of Hypervisor, Unikernel, and Container Network I/O Virtualization," 2016.

[51] B. Xavier, T. Ferreto, and L. Jersak, "Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 277–280.

[52] T. Islam, D. Manivannan, and S. Zeadally, "A Classification and Characterization of Security Threats in Cloud Computing," *International Journal of Next-Generation Computing*, vol. 7, no. 1, pp. 1–17, 2016.

[53] The OWASP Foundation, "OWASP Top 10 - 2017," Tech. Rep., 2017. [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

[54] A. Martin, S. Raponi, T. Combe, and R. Di Pietro, "Docker Ecosystem – Vulnerability Analysis," *Computer Communications*, vol. 122, pp. 30–43, 2018.

[55] B. Duncan, A. Bratterud, and A. Happe, "Enhancing Cloud Security and Privacy: Time for a New Approach?" in *6th International Conference on Innovative Computing Technology (INTECH)*, 2016, pp. 110–115.

[56] A. Bratterud, A. Happe, and B. Duncan, "Enhancing Cloud Security and Privacy: The Unikernel Solution," in *8th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017, pp. 1–8.

[57] M. Jimenez, M. Papadakis, and Y. L. Traon, "An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel," in *23th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 105–112.

[58] Jenkins Community, "Extend Jenkins," Accessed on: Oct. 10, 2019. [Online]. Available: https://wiki.jenkins.io/display/JENKINS/Extend+Jenkins

[59] A. Wiggins, "The Twelve-Factor App," Accessed on: Sep. 30, 2019. [Online]. Available: https://12factor.net/