Pós-Graduação em Ciência da Computação

Vinícius Matos da Silveira Fraga

A comparison between OS^v unikernels and Docker containers as building blocks for an Internet of Things platform

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
`www.cin.ufpe.br/~posgraduacao`

Recife
2019

Vinícius Matos da Silveira Fraga

A comparison between OS^v unikernels and Docker containers as building blocks for an Internet of Things platform

A M.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Concentration Area: Computer Networks
Advisor: Djamel Fawzi Hadj Sadok

Recife
2019

Catalogação na fonte
Bibliotecária Mariana de Souza Alves CRB4-2106

F811c Fraga, Vinícius Matos da Silveira
A comparison between OSv unikernels and Docker containers as building blocks for an Internet of Things platform – 2019.
64f.: il., fig., tab.

Orientador: Djamel Hadj Fawzi Sadok
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da computação. Recife, 2019.
Inclui referências.

1. Redes de Computadores. 2. Unikernels. 3. Internet das Coisas. 4. Computação em Nuvem. I. Sadok, Djamel Hadj Fawzi (orientador). II. Título.

004.6 CDD (22. ed.) UFPE-MEI 2019-145

Vinícius Matos da Silveira Fraga

"A COMPARISON BETWEEN OSV UNIKERNELS AND DOCKER CONTAINERS AS BUILDING BLOCKS FOR AN INTERNET OF THINGS PLATFORM"

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 22 de agosto de 2019.

I dedicate this dissertation to all my family.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Prof. Dr. Djamel Sadok, for guiding me during this research, as well as for being understanding and supportive since 2012, when first started working together. I would also like to thank Thiago Oliveira, who helped me in my LaTeX journey to write this work. Cleber Morais, Daniel Bezerra, and Carolina Cani, for all the tips they provided me, and Michaela Carmo, who helped me a lot with this thesis images and kept me going during the toughest days of work.

ABSTRACT

The Internet of Things (IoT) growth has been stimulating the development of novel technology to better fulfil its requirements. Due to its scale, IoT is powered by distributed, horizontally scalable systems, such as service oriented architectures and cloud computing. In this context, a potentially cheaper, safer and more efficient approach to virtualisation in the cloud could be the unikernel model. A unikernel is a single-process binary made of a kernel and an application built together, therefore fitting into microservice architectures, and capable of lowering computational costs in terms of time and space per service. The objective of this work is to analyse how viable it would be to develop an IoT platform meant to run on unikernels, as well as evaluate and compare unikernels' performance to containers. In order to do so, a microservice IoT platform was proposed and deployed to OS^{v} unikernel and Docker containers to work as a benchmark. Results show that it is possible to deploy modern solutions to unikernels, while highlighting open challenges and issues. Also, the expected performance gains of unikernels cannot be yet generalised, as in many cases they are still surpassed by containers.

Keywords: Unikernels. Performance. Internet of Things. Cloud Computing. Containers.

RESUMO

O crescimento da Internet das Coisas (IoT) é um fenômeno que impulsiona diversas áreas da computação para melhor atender a seus requisitos. Devido a sua grande escala, a IoT se sustenta em tecnologias distribuídas e horizontalmente escaláveis, como arquiteturas orientadas a serviço e computação em nuvem. Nesse contexto, uma alternativa potencialmente mais barata, eficiente e segura do que as atuais técnicas de virtualização utilizadas na nuvem é o modelo de unikernel. Um unikernel executa apenas uma aplicação por vez, adequando-se ao modelo de microsserviços, enquanto economiza processamento e espaço utilizados por instancia da aplicação. O objetivo deste trabalho é analisar a viabilidade de implementação de uma plataforma de serviços para IoT utilizando unikernels, assim como comparar seu desempenho com o de containers. Para tanto, foi proposta e implementada uma arquitetura que atende a um conjunto mínimo de requisitos de IoT, baseada na literatura e em soluções comercias, para servir de benchmark de comparação entre essas plataformas de virtualização. Os resultados apontam a viabilidade de se utilizar unikernels para entregar serviços de IoT, considerando algumas dificuldades encontradas, porém demonstram que os ganhos de desempenho não podem ser generalizados.

Palavras-chave: Unikernels. Desempenho. Internet das Coisas. Computação em Nuvem. Containers.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AWS	Amazon Web Services
BLE	Bluetooth Low Energy
cgroups	control groups
CLI	Command Line Interface
DNS	Domain Name System
HTTP	HyperText Transfer Protocol
IFTTT	If This, Then That
IoT	Internet of Things
IP	Internet Protocol
KVM	Kernel-based Virtual Machine
MQTT	Message Queue Telemetry Transport
NIST	National Institute of Standards and Technology
OS	Operating System
PAN	Personal Area Network
SD	Standard Deviation
SOA	Service-oriented Architecture
UUID	Universally Unique Identifier
VM	Virtual Machine

CONTENTS

1 INTRODUCTION

1.1 MOTIVATION

Since the creation of the web in the early 90's, the number of users connected to the internet has grown exponentially. Soon enough, various businesses (e.g. Amazon in 1995) have started harnessing the World Wide Web due to its long reach and cost efficiency when compared to affording physical stores and traditional propaganda. People were not just chatting online, they were also shopping. But it did not stop there; this phenomenon got to a stage where much more than just users are connected: all things are. It is the IoT era. This means that every single object might be connected to the internet, generating insightful data and accepting remote or even fully automated control. Empowered by the IoT, users, companies, and governments can now benefit from wireless home automation, smart healthcare, smart cities, autonomous vehicles, and the list goes on.

By its very definition, IoT encompasses a set of well-known requirements, such as scalability, availability, flexibility, security, performance, and privacy (Morabito *et al.*, 2018). Nevertheless, deployments of smart city solutions, or even hospitals and factories, can involve thousands to billions of connected devices. To meet these requirements in scenarios with such dimensions, the upfront cost of ownership could be prohibitive, despite maintenance and further upgrades. Fortunately, there is already an alternative to optimise this kind of situation: cloud computing. Based on a proven principle of focusing on your expertise and leaving everything else to third parties, cloud providers offer on-demand services at many levels, from virtual infrastructure to serverless applications. Thus allowing for clients to worry only about their own business while drastically reducing costs with a *pay-as-you-go* model.

Despite providing flexible and scalable infrastructure, cloud computing alone is not enough to meet IoT requirements. An IoT system needs an architectural style capable of exploring the cloud features. Monolithic architectures, for instance, would produce brittle IoT solutions, as they would only scale only by replicating the entire system. A better option would be the microservices architecture, which conceives multiple independent, fine granular services with a single responsibility each. Thus allowing for precise scaling and maintenance, fitting into dynamic IoT scenarios. However, cloud computing started based on VMs, emulating computers from top to bottom. It turned out that in many cases that was not the best fit for clients demands, and microservices are an example of that, as full Operating System (OS)s on top of VMs are too bloated to run a microservice alone. Containers then appeared to be a sufficiently isolated solution with better cost-benefit. Containers are light weight and encapsulate apps in their own environment, while sharing the kernel of a host operating system OS and therefore eliminating the need of a full VM to run a specific service or appliance. Cloud providers then started offering container-based provisioning, making it transparent to the client how the underlying OS functions are managed.

Considering that IoT solutions require granular architectures such as microservices, on top of cloud infrastructure, and that cloud computing is evolving towards light virtualisation to better fulfil such requirements, a new alternative has emerged: unikernels. Unikernels are based on the not so recent concept of library operating systems (Engler *et al.*, 1995). A unikernel goes against the concept of multitasking OSs, because it is a whole system compiled to run a single process application. Notice that this is a match to microservices architecture. Furthermore, it has no separation between user and kernel space, an nothing else besides the required libraries to run the desired application. As a consequence, some unikernels boot really fast (milliseconds magnitude), have a reduced memory footprint and can execute faster, as there are much less tasks sharing the CPU. Besides, they are also safer, as the attack surface is limited by the application requirements. However, unikernels are not being widely used in production yet, and no cloud provider offers any unikernel-based solution so far. The available publications and experiments do not cover enough ground for one to be confident in deploying unikernels to production. While containers are the status quo of light virtualisation, comparative studies based on realistic scenarios and architectures deployed to unikernels can lead cloud and IoT to the next stage.

1.2 RESEARCH QUESTIONS

Given the context, this work was developed aiming to answer the following question: *How unikernels performance compare to containers in microservice-based, cloud-oriented IoT solutions?* However, in order to answer that question, another one arises: *can* unikernels be used to deploy such architectures? This dissertation will answer both of them, highlighting difficulties and challenges encountered in the process.

1.3 OBJECTIVES

The main objective of this dissertation is to evaluate unikernels performance and behaviour when compared to containers in an IoT platform context. Furthermore, there are some specific objectives to be accomplished:

- Define an archetype of IoT architectures based on literature and industry solutions, to work as an IoT benchmark;
- Identify the most suitable unikernel for implementing the benchmark architecture;
- Provide a software implementation of the benchmark and the tools to deploy it to both containers and unikernels;
- Provide a report of issues and open challenges found during the process of deploying the benchmark architecture to unikernels;
- Validate literature results.

1.4 METHODOLOGY

The methodology applied to this study can be separated into the following phases: literature review, architecture definition, implementation and deployment, experimentation, and analysis.

More than just familiarising with the state of the art, the literature review was performed with the premise of filtering which metrics to observe during the experiments, and to find an architecture template for IoT platforms. While the performance evaluation portion of the selected works is exposed in Chapter 3, the remainder is used as reference in Chapter 2.

For the related work part, we searched for the string *"performance evaluation of unikernels"* in Google Scholar engine, which returned around five hundred results at the time. Theoretical evaluations, mathematical models, framework proposals, security analysis, and comparisons of light virtualisation that did not include unikernels in any experiment, were all discarded. The selected ones are discussed in Chapter 3.

To start experimenting, the first step was to find a valid IoT architecture template. When searching *"Internet of Things architecture"* in Google Scholar, the amount of results was astonishing. Therefore, only surveys were considered by using the string *"Internet of Things architecture surveys"*, and only the ones from 2018 upwards. As the results would still be in the order of tens of thousands, the selected pieces were an outcome of a cherry-picking process, based mainly on title and abstract filtering.

Once we defined the architecture (detailed in Chapter 4), the implementation followed a unikernel-first manner. In other words, it was permeated by partial testing on OS$^{\text{v}}$ unikernel to make sure everything would work at the end. This precaution was taken due to unikernels immaturity, and that proved to be a wise decision as we faced many OS-related issues during development (those issues will also be discussed in Chapter 4).

After the implementation was finished and tested on unikernels and containers, the real experimentation began. A set of different request types was defined in order to explore all the features provided by the architecture (and thus put all the microservices to work). Every experiment was repeated 30 times. Finally, a discussion was conducted to put unikernels and containers results into perspective.

2 BACKGROUND

This chapter introduces basic concepts for the understanding of this work. Starting with the Internet of Things, which provides context for the experiments, to a thorough explanation of unikernels, which are on focus in this research. Related technologies in the dependency stack are also included, such as microservices architecture, cloud computing, and containers.

2.1 INTERNET OF THINGS

In 1926, Nikola Tesla made the following prediction: *"When wireless is perfectly applied the whole earth will be converted into a huge brain, which in fact it is, all things being particles of a real and rhythmic whole."* (Kennedy, 1926). His statement unveils the main idea behind IoT: to connect *everything* to the Internet.

The expression *Internet of Things* was used for the first time by Kevin Ashton, in 1999, during a presentation at Procter & Gamble (Ashton, 2009). However, it was only after years of debate and research that IoT achieved maturity in literature. Many authors surveyed the Internet of Things and were capable of identifying dominant characteristics. In 2015, researchers from IEEE provided a formal definition:

"Internet of Things envisions a self configuring, adaptive, complex network that interconnects 'things' to the Internet through the use of standard communication protocols. The interconnected things have physical or virtual representation in the digital world, sensing/actuation capability, a programmability feature and are uniquely identifiable. The representation contains information including the thing's identity, status, location or any other business, social or privately relevant information. The things offer services, with or without human intervention, through the exploitation of unique identification, data capture and communication, and actuation capability. The service is exploited through the use of intelligent interfaces and is made available anywhere, anytime, and for anything taking security into consideration." (Chebudie *et al.*, 2015).

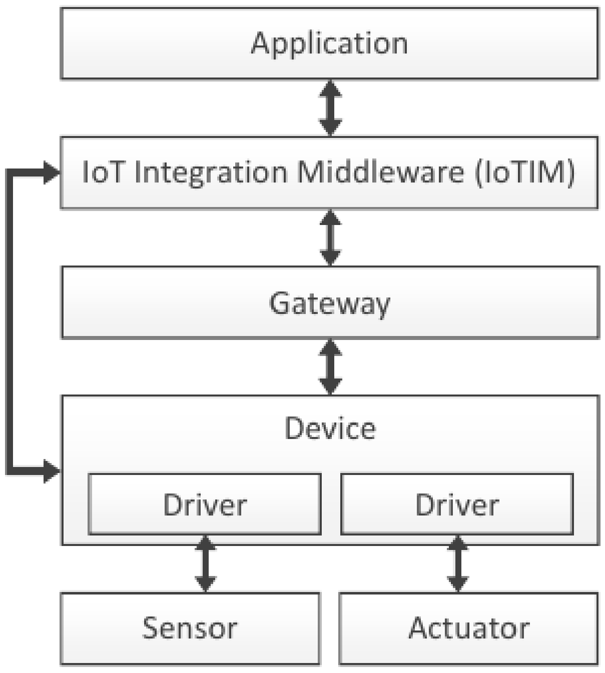
Given this definition, it is clear that IoT solutions can be used in a multitude of scenarios. Connected devices can turn common environments into smart environments, capable of obtaining and applying knowledge autonomously. These environments can be smart homes, offices, hospitals, parking lots, or even entire cities (Ahmed *et al.*, 2016). However, along with the benefits of having smart things generating insightful data and actuating on its own decisions, comes a series of requirements to be met. Since the number of devices can reach billions (Shah & Yaqoob, 2016), Big Data comes as a natural consequence, bringing scalability issues. All things can be different and often are, meaning heterogeneity is also a concern when making it all communicate. Last but not least, privacy and security are major issues (Saadeh *et al.*, 2016), especially considering that IoT devices can carry personal and sensitive information. In addition, these devices are in many cases running on power constrained platforms, incapable of heavy cryptography computation.

As aforementioned, heterogeneity in IoT systems became a challenge to overcome. Such a wide range of use cases has pushed different companies, researchers, and manufacturers to invest (the economic impact is expected to reach trillions of dollars by 2025(Al-Fuqaha *et al.*, 2015).) and produce new IoT solutions and protocols. Nonetheless, despite its idiosyncrasies, it was observed that many architectures have much in common. It is expected for an IoT solution to have a *middleware* at its core (Al-Fuqaha *et al.*, 2015) (Kraijak & Tuwanut, 2015), and that this middleware can manage different types of services, such as data storage and analytics (Ngu *et al.*, 2017). In fact, almost every reference cited in this work relies on a service-based architecture or has a layer dedicated to services. These services are commonly provided to the application layer through RESTful APIs (Ngu *et al.*, 2017) (Lea & Blackstock, 2014).

Not all devices in IoT are capable of connecting directly to the internet. In many cases, they have simple hardware and connect to each other through Personal Area Network (PAN)s. The solution of choice is to have a *gateway* sitting in between the middleware and such devices. A gateway is often a more powerful device which can communicate using different protocols, such as Zigbee and Bluetooth Low Energy (BLE) (Ahmed *et al.*, 2016). A common practice to communicate with gateways is through message brokers, based on publish/subscribe protocols such as Message Queue Telemetry Transport (MQTT) (Al-Fuqaha *et al.*, 2015) (Lee *et al.*, 2017).

To help summarise the Internet of Things concepts, Guth proposed an IoT architecture model (Guth *et al.*, 2018). It successfully maps different open source projects and private IoT solutions from big cloud companies (Amazon, Microsoft, IBM) into a six-layer architecture. This architecture was used as a reference to design the Jung platform (detailed in Chapter 4), proposed and evaluated in this dissertation' experiments. From top to bottom, the first layer is the application layer, which sits on top of an IoT middleware. Between the middleware and the devices, there is a gateway. As aforementioned, not every device needs a gateway to connect to the middleware; Internet Protocol (IP)-capable devices can connect directly. Finally, each device can have sensors and actuators, and thus it must have proper drivers. Figure 1 depicts the architecture.

Figure 1: IoT Reference Architecture.



Source: (Guth *et al.*, 2018)

Despite having Guth's architecture template to guide the IoT platform development, a few more questions still need to be addressed to meet real-world requirements. Where to run the middleware? How to scale the system? How to distribute gateways? The answer to all these questions is often the same: *cloud computing*. IoT benefits from cloud "virtually unlimited" resources, horizontal scalability and other established services, such as real-time analytics and data-oriented models (Gil *et al.*, 2016).

At this point, we have a definition for IoT and its requirements, along with solutions of choice such as cloud computing and service-based architectures, as well as a reference architecture. The next section explores the specifics of *microservices* architectural style, which is an important premise for unikernels and containers to shine.

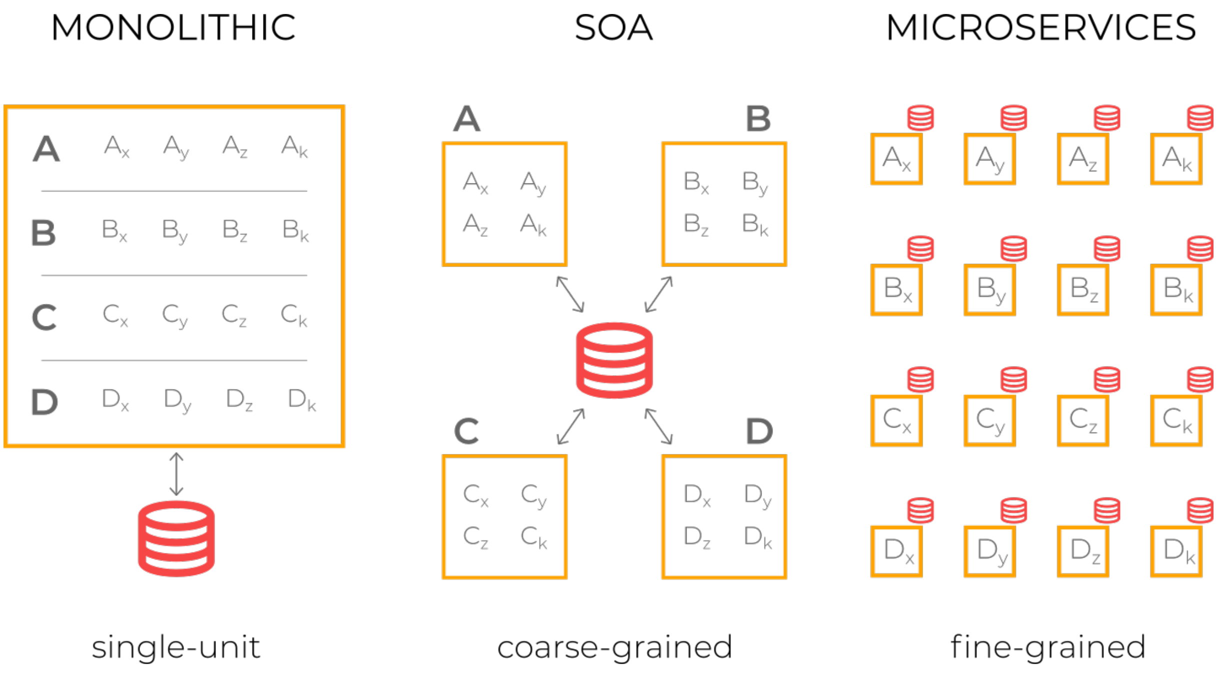
2.2 MICROSERVICES

First of all, please note that *service-based* architecture does not mean Service-oriented Architecture (SOA). Both SOA and microservices are service-based architectures, but they are not the same.

SOA moved from monoliths towards distributable modular systems, where each module would provide a service; each service would encompass a variety of tasks related to a given domain, such as orders from an online shop. An Order service would provide different information to other modules of the system, such as logistics, financial, and customer services. All this information could be stored in a single database as well. In the SOA world, it is acceptable.

Microservices, on the other hand, would separate each sub-domain of an order into an independent service. There would be a microservice to provide customers with its customer-related information of an order, another one for financial-related information, and yet another for logistics. Moreover, each one would ideally have its own database. While SOA is a coarse-grained, *share-as-much-as-possible* (e.g., use a single database) architecture, microservices are a fine-grained, *share-as-little-as-possible*, and context-bound (Richards, 2016). Figure 2 illustrates those differences.

Figure 2: Granularity differences among Monolithic, SOA, and Microservices architectures.


MONOLITHIC
SOA
MICROSERVICES
A A_x A_y A_z A_k
B B_x B_y B_z B_k
C C_x C_y C_z C_k
D D_x D_y D_z D_k
single-unit
coarse-grained
fine-grained

Source: (Fraga, 2019)

It is clear that microservices introduce challenges: all these moving parts must somehow work together to provide consistent outputs. What are the benefits of having such a sparse architecture? The answer is that *it mitigates complexity.* By separating concerns into very specific microservices, one can avoid bloated implementations which are hard to maintain and costly to deploy. It gives freedom to develop each service using the most appropriate tools, even using different languages and frameworks, despite not enforcing it. Given the dynamic range of IoT scenarios, those are desirable features for a system to have, as every requirement can vary depending on each feature and involved devices. In fact, microservice-based IoT solutions have been proposed and tested in literature (Vresk & Čavrak, 2016) (Bak *et al.*, 2015) (Wanigasekara, 2015).

Organisation, communication, and deployment of microservices are well studied. One of the most prominent books on the subject, *Building Microservices* (Newman, 2015), advocates for choreography over orchestration to organise the architecture. It means that every microservice should be smart enough to know its own responsibilities and course of action. In orchestration, a central unit of control would be required to command every service, tending for brittle designs. In choreography, they would be talking to each other. Communication can be synchronous or asynchronous, translating in most cases to request/reply and event-driven models. The first one is simpler, while the latter is highly decoupled, thus more flexible.

The cloud is the most natural place for microservices to be, as they are designed to be distributed, and highly scalable. Due to their small size and context-boundaries, they are often encapsulated into minimalist VMs, or in containers, which are an appealing deployment platform in this case (Chris Richardson, 2016).

In a nutshell, microservices are an architectural pattern to develop systems as a composed set of small, single-responsibility services. Each microservice should be as independent and decoupled as possible. Microservices architectures are flexible and dynamic by design, and are often deployed using small VMs or containers. The next section is an overview of virtualisation, presenting fundamental concepts of containers and unikernels.

2.3 VIRTUALISATION

Previous sections mentioned the relation of IoT and cloud computing multiple times. Cloud is formally defined by the U.S. National Institute of Standards and Technology (NIST) as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* (Mell & Grance, 2011). The cornerstone for its flexibility and convenience is virtualisation.

Virtualisation is an abstraction technique to create a given resource, e.g. a VM or a network device, on top of a physical layer. Differently from what one might think, virtualisation is not a synonym to emulation nor simulation. Emulation would be a software reproduction of hardware features, while virtualisation can access the hardware itself, thus achieving superior performance. Simulation, for its turn, is based on modelling and mimicking instead of reproducing any infrastructure behaviour; it is just an enclosed representation of an environment, such as flight simulators for pilots.

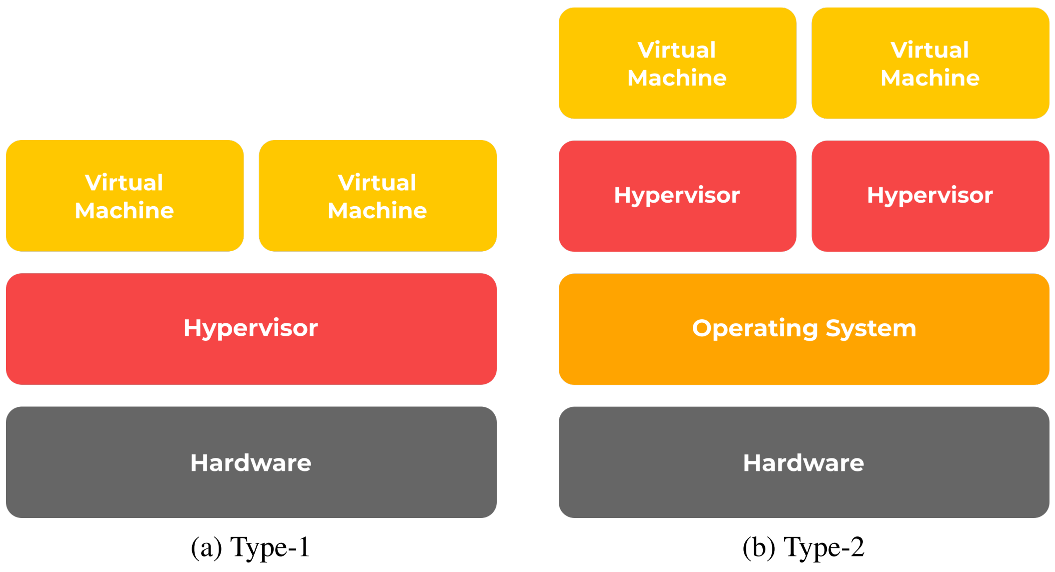
2.3.1 Hypervisors

VMs access to hardware is crucial for cloud computing to be a cost-effective paradigm. But how do these virtual machines compete for resources? How are they managed and isolated from each another? The answer is that there is a software component in between the hardware and the VMs, called hypervisor (also known as Virtual Machine Monitor). According to its definition, a hypervisor must exhibit three properties: equivalence, efficiency, and resource control (Popek & Goldberg, 1974).

- **Equivalence** states that any program running on a VMs performs the same as if it were running on hardware, except for resource availability and timing. These exceptions are explained by the possibility of multiple VMs to be running on top of one physical CPU, creating the need for scheduling and limiting execution time.
- **Efficiency** states that a statistically dominant subset of a virtual processor's instructions run directly on a real processor, without intervention from the hypervisor.
- **Resource control** states that the hypervisor manages all hardware, preventing any program running on a VM to affect resources availability to other VMs.

Hypervisors can be classified into two different types: Type-1, which sits on top of hardware and manages resources by itself; and Type-2, which runs on top of a host OS that manages resources. Both can be seen in Figure 3. Note that it is possible to run multiple Type-2 hypervisors at the same time, while Type-1 has a performance advantage due to its direct contact with hardware. Popular examples of type-1 hypervisors are Xen, VMWare ESXi, and Microsoft Hyper-V. As for type-2 there are VirtualBox, QEMU, and VMWare Player.

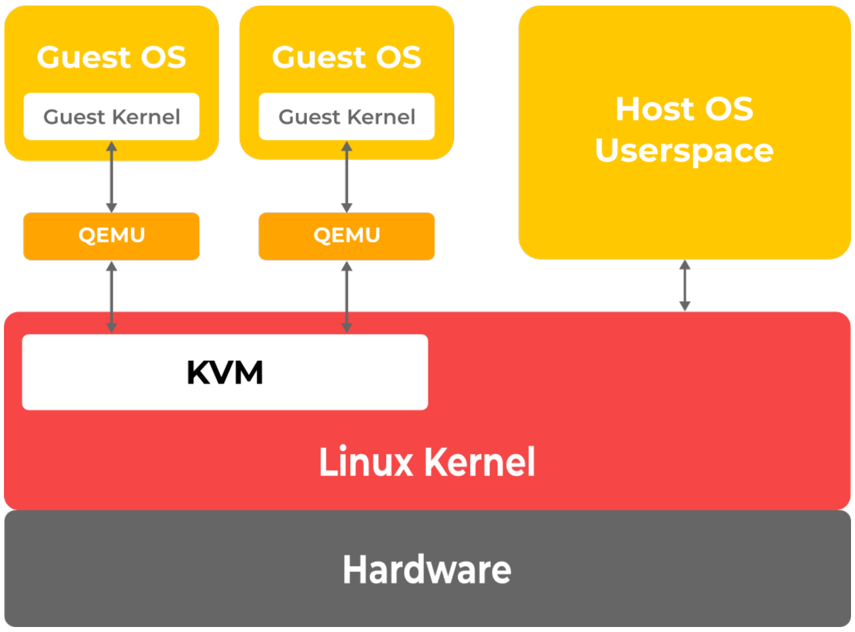
Figure 3: Hypervisor architectures.



Source: (Fraga, 2019)

Nevertheless, there is a "hypervisor" which deserves special attention: the Kernel-based Virtual Machine (KVM). KVM, which architecture is illustrated in Figure 4, is a module of the Linux kernel which provides hardware-assisted virtualisation features to upper layers, meaning it turns the OS into a type-1 hypervisor (KVM, 2016). KVM does not create virtual machines by itself, but it can be used to bridge QEMU (which is type-2 when used alone) access to hardware, powering it with type-1 performance, and providing full hardware-assisted virtualisation. It was the hypervisor of choice for this dissertation experiments, and it is the core of Amazon Web Services (AWS) [1] virtualisation (Sharwood, 2017).

Figure 4: KVM architecture.



Source: (Fraga, 2019)

2.3.2 Containers

The previous subsection explained how virtualisation of full OSs works. However, considering a microservices scenario, something smaller than an entire OS would be preferable to deploy each of those independent, isolated, often tiny microservices. Such a lightweight deployment is possible through the use of containers. Containerisation is a technique to run software as isolated as possible from a host OS while sharing its kernel functionality. As the containerisation tool used in this dissertation was Docker, and it is the dominant container technology (Pahl *et al.*, 2017), this section provides a quick Docker-oriented timeline to explain containers principles.

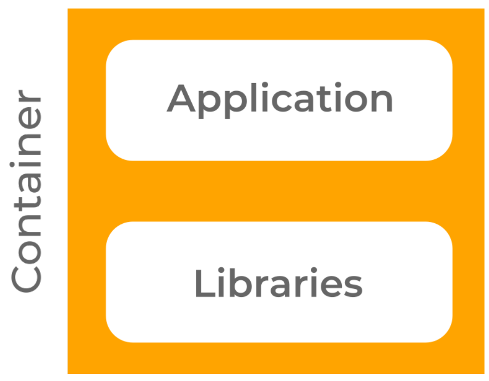
One of the basic ideas behind containerisation dates from 1979, when the *chroot* system call was added to Unix V7 (Marquez, 2018). The chroot command changes the root directory to an arbitrary path and then executes a given command, thus allowing for encapsulating a process separately from the real filesystem of the host. Fast forwarding to 2008, *control groups*

[1] Leading cloud service at writing time

(cgroups) were merged into Linux 2.6.24; cgroups enable the creation of groups of processes, allowing for resource management (memory, CPU, and I/O) in a per-group fashion. However, those processes would still be able to "see" every other process running on the OS, living in thin isolation. Then Linux *namespaces* come in: a set of system calls to isolate cgroups, network devices, filesystem (replacing chroot with pivot_root), user and groups ids, process ids, and interprocess communication queues. Namespaces were added to Linux in 2002, but only 11 years later it reached the "container ready" state, providing all these features in kernel 3.8 (Kerrisk, 2013).

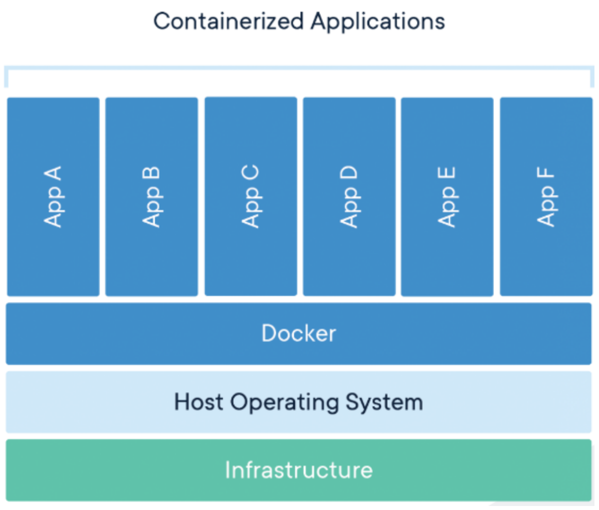
Docker project started in 2013, built on top of namespaces and cgroups and designed to run only one application per container. Docker has a runtime to manage containers and a remote repository called Docker Hub (Docker, 2019a), where Docker images can be stored and shared. A Docker image is a file containing the needed information to run code in a container. It has multiple layers, making it possible to create new images on top of existing ones. For example, a user could take an image containing Python3 interpreter and add Django [2] to it, saving it as a Django image which could be later used to create a Django app. To assemble an image, a user can write a Dockerfile, which is a text document containing all the commands needed to build the image from Command Line Interface (CLI)[3]. Figure 5 demonstrates the internals of a regular container and an overview of Docker deployment architecture.

Figure 5: Container and Docker architectures.



(a) Container architecture

Source: (Fraga, 2019)



(b) Docker containerisation architecture.

Source: (Docker, 2019b)

When compared to VMs, containers stand out by requiring less disk space, less memory, and "booting" faster (as they are just processes starting), thus being considered lightweight. On

[2]Django is a popular web framework for Python
[3]Image building details can be found at https://docs.docker.com/engine/reference/builder/#usage

the other hand, despite their multiple software isolation levels, they still less isolated than VMs, which have hardware-level isolation. It makes them more vulnerable, and depending on the security requirements they might not be a suitable approach. The next section presents unikernels, which are lightweight compared to VMs while keeping the same isolation, and conceptually very distinct from containers at the same time.

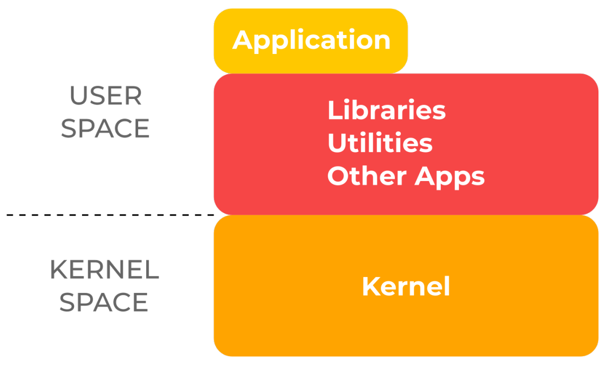
2.4 UNIKERNELS

The first paper about unikernels was published in 2013 and states the following: *"Unikernels are single-purpose appliances that are compile-time specialised into standalone kernels, and sealed against modification when deployed to a cloud platform."* (Madhavapeddy *et al.*, 2013). It is still mostly accurate, except for that last part where it restricts unikernels immutability to cloud deployments, which is not correct. They are immutable by nature. Beyond its definition, a useful way to understand unikernels is to compare them to a traditional OS. The first difference is that systems like Linux and Mac OS were built to provide multiple services to multiple users (while unikernels run a single process). This diversity led to the necessity of controlling the access of users and applications to the machine's resources, to avoid interference among them. The solution of choice was to separate code execution into different privilege levels, based on different addressing spaces: kernel space and user space. As its name suggests, the kernel space is the home for kernel code, which is responsible for filesystems, networking and I/O in general, memory management, and process management. User space is where libraries and user programs live. Every time a process needs I/O, the control has to be passed to the kernel. This is accomplished by using system calls, which rely on special CPU instructions to change the privilege. Besides, every time a process gets rescheduled from a CPU, a context switch[4] must be performed. This means CPU registers content must be moved into memory, and a switch into kernel mode is required. All these modes and context switches consume processing time, specially context switches because of memory access.

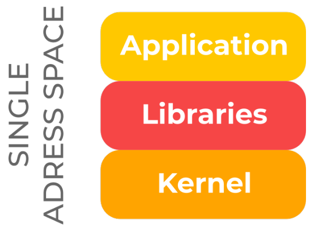
Unikernels, as can be seen in Figure 6, have a single address space. This is what the "standalone kernel" in the definition means: the application, its dependencies, and the kernel code are all executed "together" in a single process with maximum privilege. Hence they are free of mode and context switches, saving processing time.

[4]A context switch consists of storing a task state in memory for later execution

Figure 6: Unix-like vs Unikernel architectures.



(a) Unix systems.



(b) Unikernels.

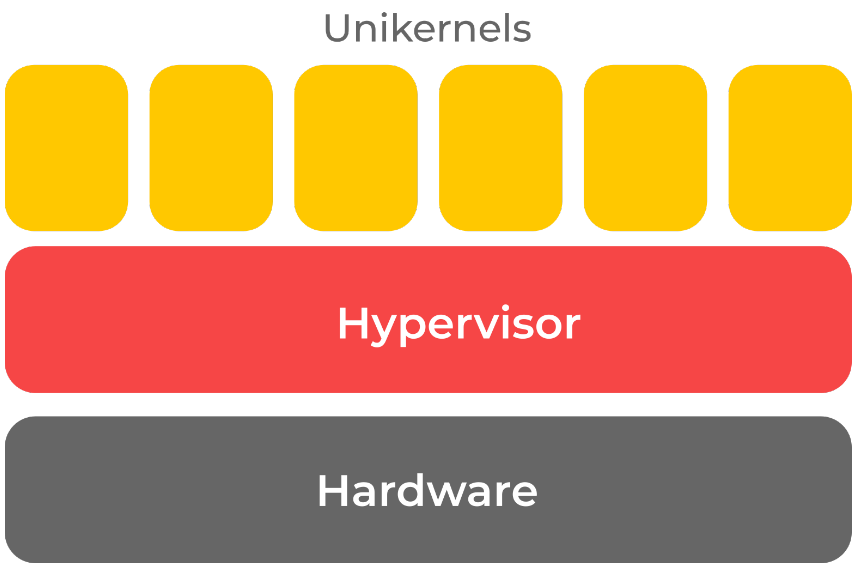
Source: (Fraga, 2019)

Also in Figure 6 it is noticeable that Unix systems and unikernels architectures have different sizes. That is a representation of unikernels minimalism, as they only ship what is strictly needed by its application. Traditional OSs, on the other hand, have tons of software such as libraries and device drivers that are never used as a whole by a single app. This characteristic can be useful for deployment to constrained devices, such as routers or battery-powered sensors, which do not usually have much memory and could benefit from unikernels to host applications smarter than a firmware and lighter than a traditional OS.

Besides a small footprint, unikernels benefit with improved security due to its reduced attack surface. By including only dependencies it avoids unwatched software laying around; commonly there is no shell to interact with, no unused drivers, nor password files or connections to external machines beyond the application needs (Duncan *et al.*, 2017)(De Lucia, 2017). It means that even in the situation of a successful invasion to a unikernel, an attacker would only have access to the application itself, being unable to harm other VMs processes of the system unless they were capable of breaking into the hypervisor itself. Besides, it is a useful characteristic for deployment to constrained devices, as it provides security while saving resources. On the other hand, it can make it difficult to monitor and debug unikernels in production environments, for example. One option would be to kill troublesome instances, replace them with new ones, and rely on logs to perform maintenance and prevent bugs. But this would require specific studies to measure how costly it can be to maintain unikernels in production environments.

In terms of deployment, there are unikernels designed to be exclusively used with a given hypervisor, such as MirageOS with Xen; others can run on different hypervisors but never on hardware, such as OS^{v} ; and others enable bare-metal deployment, such as Rumprun and HermitCore. However, even those last two need some kind of underlying hardware abstraction layer (CloudKernels, 2019) (RWTH Aachen University, 2017). Therefore we can visualise unikernels deployment stack as in Figure 7, replacing the hypervisor for a specific tool in some cases.

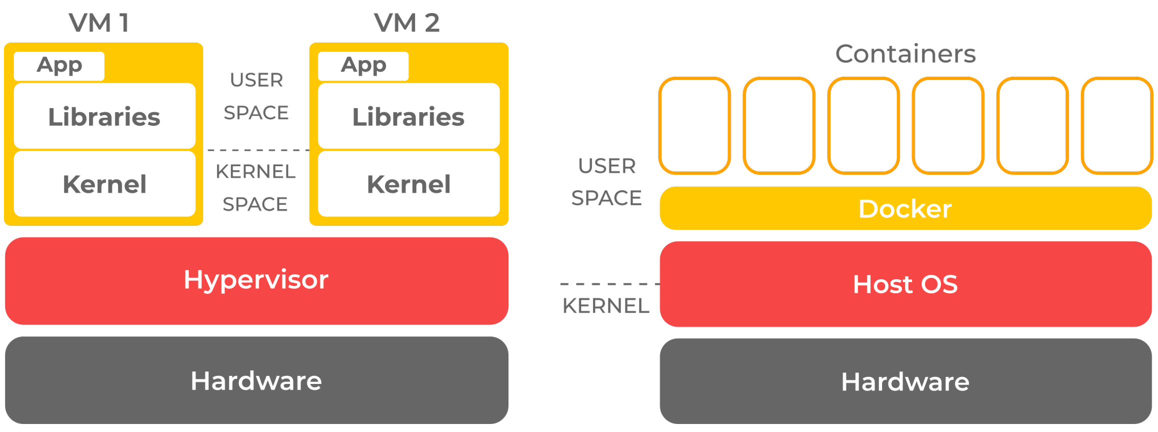
Figure 7: Unikernel deployment stack.



Source: (Fraga, 2019)

For the sake of comparison, VMs and containers deployment would look like Figure 8. Note that it is also possible to deploy containers inside virtual machines on the cloud (as offered by Google (Google, 2019) and VMWare (VMWare, 2018), for example), stacking more layers.

Figure 8: Deployment stacks of VMs and Docker containers.



(a) Virtual Machine (b) Docker native-hosted containers

Source: (Fraga, 2019)

Unikernels are still in their infancy, but they are receiving increasing attention. At the time of writing there are at least 11 unikernel projects[5] ranging from high level, language-specific

[5]A complete list can be found at http://unikernel.org/projects/

designs such as MirageOS, to POSIX-compliant alternatives such as OS^v and Rumprun. Their ecosystem is also evolving, counting with tools like Unik (Levine, 2018) to simplify building and orchestration of multiple unikernel flavours, and research on live updating (Walla, 2017) and provisioning (Madhavapeddy *et al.*, 2015) as well. Investment-wise, many companies have already contributed to unikernels development, such as IBM an Microsoft (Pavlicek, 2017). In 2016 Docker bought Unikernel Systems, a company which contributes to MirageOS and Rumprun (Blatstein, 2016). In 2017 they made their LinuxKit project open source. LinuxKit is "a toolkit for building custom minimal, immutable Linux distributions." (Docker, 2017), with clear unikernel influences.

2.4.1 OS^v

Among the multiple unikernel flavours, we have decided to use OS^v for this dissertation experiments. It is one of the most mature projects (already supports different hypervisors and cloud providers), and it was designed to run almost any POSIX program (given it is single-process), eliminating the need for learning new languages and stacks. Nevertheless, other options were tested, as will be discussed in Chapter 4.

OS^v is defined by its creators as *"a new guest operating system designed specifically for running a single application on a virtual machine in the cloud"* (Kivity *et al.*, 2014). It was created to be a general-purpose unikernel, accepting unmodified Linux programs (Cloudius Systems, 2019). There is a list of supported applications in the project's page[6] which are ready to be used. To ease building and deployment OS^v has the Capstan tool, which provides docker-like configuration files and CLI (MIKELANGELO, 2015).

Regarding its implementation, the first important aspect is that OS^v does not have drivers for real hardware, only for hypervisors. The core of OS^v is new code written in C++ version 11, including memory management, thread scheduler, virtual-hardware drivers and more. It follows a traditional Unix virtual filesystem design, having ZFS[7] as its major filesystem while counting with some alternatives, such as the in-memory ramfs.

Despite not having multiple processes, OS^v does allow for multi-threading. Its thread scheduler keeps separated run queues for each available CPU, performing load balance to preserve the fairness. Notice that for OS^v to support multithreading it needs to implement context switches between threads, but those are lightweight when compared to traditional OSs because of the single address space. OS^v also has an optimised TCP/IP stack, inspired by Van Jacobson *net channels* (Van Jacobson, 2006). Kivity *et al.* presented results of improved throughput (25%) and reduced latencies (up to 47%) when compared to Linux.

[6]https://github.com/cloudius-systems/osv-apps
[7]Learn more about ZFS at https://itsfoss.com/what-is-zfs/

3 RELATED WORK

This chapter surveys the literature in unikernels performance evaluation, comparison to other virtualisation techniques, and applications in IoT scenarios. In the following sections it is noticeable that CPU and memory usage are the most popular metrics, along with image size and request throughput. Methodologies generally consist of running a specific application or piece of software and on a given unikernel and a regular Linux or container deployment. Each of these papers has a contribution for this dissertation, either in the form of an insight or a methodology reference.

3.1 UNIKERNELS AND THE INTERNET OF THINGS

In (Morabito *et al.*, 2018), there is a discussion on the applicability of unikernels and containers in edge computing elements of IoT platforms. After highlighting IoT requirements, the authors present their architecture proposals for three different scenarios: vehicular networks, smart city, and augmented reality. For each one of the architectures, an edge layer (between the devices and the cloud) would provide extra processing, storage, and network resources to the whole system, alleviating the cloud workload. This edge layer would be constituted by devices such as Raspberries, running lightweight virtualised applications/systems.

As an outcome of their research, they provide a list of open challenges and issues. Despite covering relevant topics as standardisation and security certificates, one aspect was particularly insightful for this research: unikernels are unfit for data storage. As their nature is to be immutable and isolated, once a unikernel crashes or needs code maintenance, the rule of thumb is to kill the instance, compile and run a new one, making it costly to maintain data consistency. Also, databases commonly need to be closely monitored, while unikernels would only provide logging capabilities by design.

Although their proposals and discussion are valid, the authors did not perform any experiments or evaluation in their work, limiting their contribution to the theoretical field. This dissertation, on the other hand, proposes an IoT architecture and provides a performance evaluation when running it on containers and unikernels.

3.2 PERFORMANCE EVALUATION OF UNIKERNELS

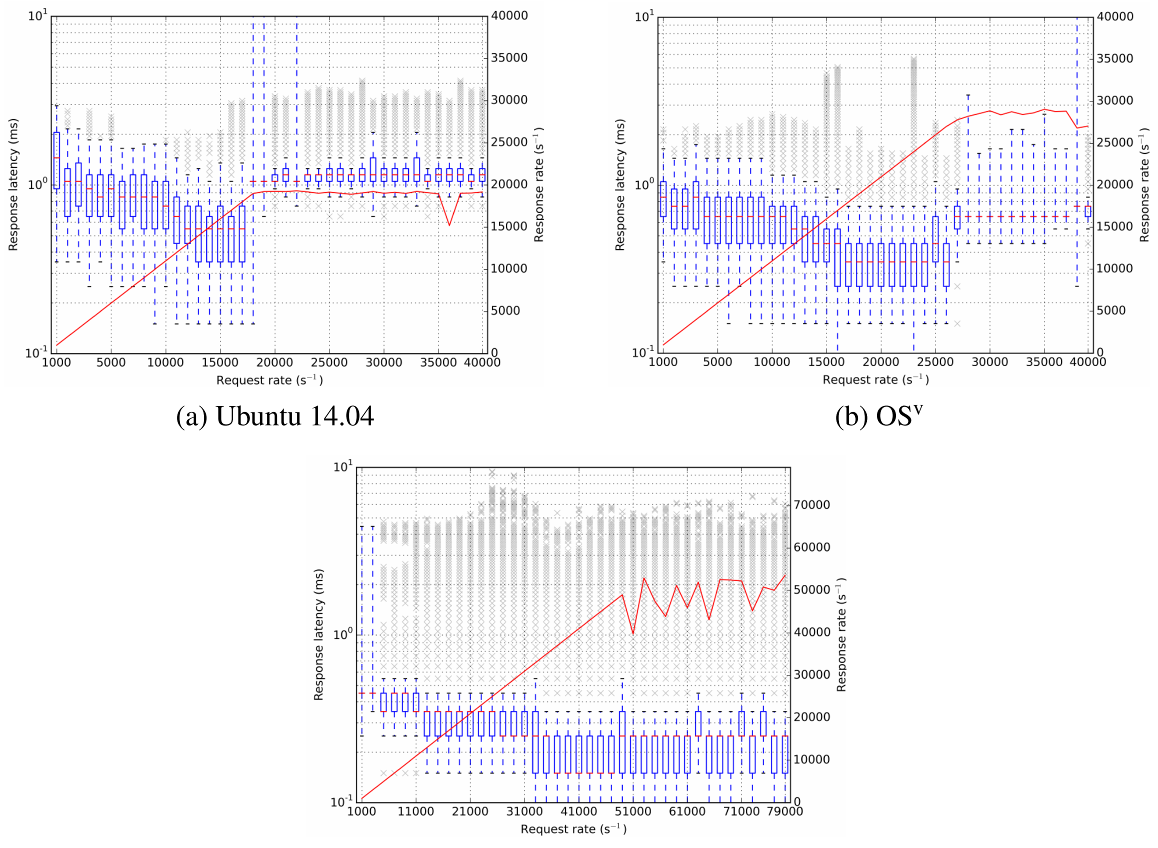
3.2.1 Mirage vs OS^v vs Linux

One of the first performance evaluation works found during literature review was presented in 2014, titled *"A Performance Evaluation of Unikernels"* (Briggs *et al.*, 2014). The authors' goal was to validate the claims about unikernels performance, that could be found in papers from their creators where they were commonly presented and tested against a specific application. In order to provide a broader evaluation, the authors picked two different unikernels, Mirage and OS^{v} , which they considered to be more stable at the time. They used macrobenchmarks based on network performance tools to compare the unikernels against a regular Linux system (Ubuntu 14.04), aiming to achieve results closer to real-world scenarios.

Despite the objective of validating unikernels performance, two things require attention in this publication: the first one is that the authors *opted not to measure memory utilisation*, claiming that running an additional process to monitor it could impact the performance. The second is that they also decided *not to measure CPU utilisation as well*. The reason would be that the tools they were using were not capable of handling this kind of monitoring in client-server scenarios. That said, the metrics they measured were response latency and request throughput from Domain Name System (DNS) queryperf (Nominum, 2012) and HyperText Transfer Protocol (HTTP) httperf(Hewlett-Packard, 2005) benchmarks.

Their DNS tests reflected that OS^{v} performed better than Linux, showing lower response latencies (box plots) and higher response rate (line chart), as can be seen in Figure 9. Mirage achieved even higher request rates, but with some response latency outliers with significantly high values, as can be seen in the grey area above the line. The authors speculate that OCaml's runtime garbage collection was the cause of it.

Figure 9: DNS server results. The box plots refer to the response latencies, while the line chart reflects the response rate.



(a) Ubuntu 14.04

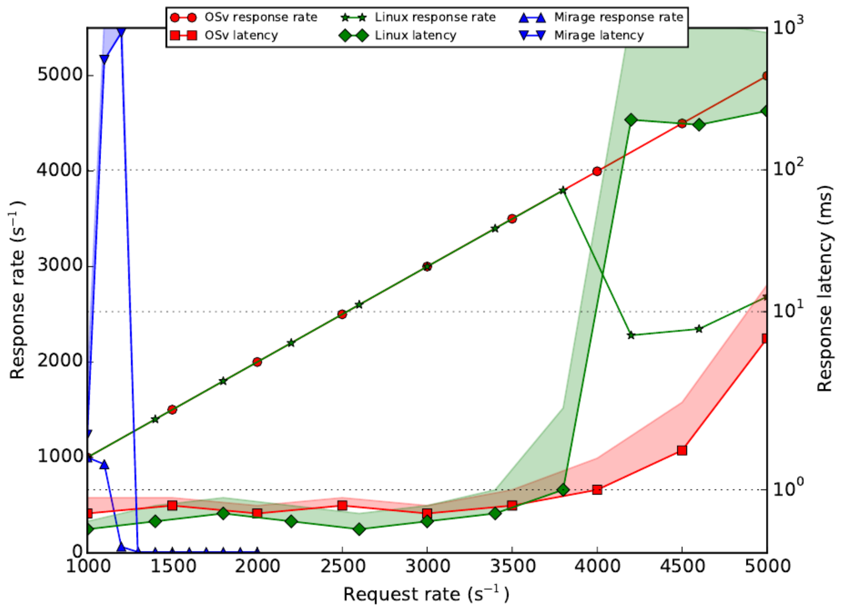
(b) OS^{V}

(c) Mirage OS

Source: (Briggs *et al.*, 2014)

When looking at the HTTP server results, depicted in Figure 10, it is noticeable that Mirage's numbers are quite odd. It starts with high latency and then its response rate abruptly drops. The authors concluded that it was caused by a memory leak every time a TCP connection was opened. This bug severely compromised its performance. On the other hand, OS^{V} surpassed Linux, maintaining consistent performance over 5,000 requests per second.

Figure 10: HTTP server results.



Source: (Briggs *et al.*, 2014)

Ultimately, *"A Performance Evaluation of Unikernels"* confirms unikernels potential, especially with OS^v , while it also uncovers bugs and paints a picture of immaturity for production. Their scope was very limited metric-wise, as memory and CPU were ignored, and thus the authors include such observations in their future work. Among other improvements, they suggest comparing unikernels to containers, as well as using a more dynamic testbed instead of static content servers, all of which were done in this dissertation.

3.2.2 Rumprun vs Debian

In (Elphinstone *et al.*, 2017) the authors objective was to measure how difficult it would be to run a unikernel on top the seL4[1] OS, besides evaluating its performance. They also compare Rumprun unikernel performance to a regular Linux, thus providing relevant information for this dissertation.

During their tests, the authors used Iperf tool to generate TCP traffic loads, and then measured the CPU utilisation of different deployments of Rumprun on top of seL4, along with a bare metal deployment and a Debian Linux. At the end of the day, Linux achieved the lowest CPU utilisation in all experiments. However, authors claimed that the Linux machine was running a different network card driver and thus it was difficult to draw direct comparisons. In spite of that driver difference, their results reflect values five times lower for Linux, even when compared to bare metal Rumprun. Such a difference is at least a signal that regular Linux can

[1]seL4 is a microkernel focused on security which has a formal proof of its implementation, guaranteeing its correctness.

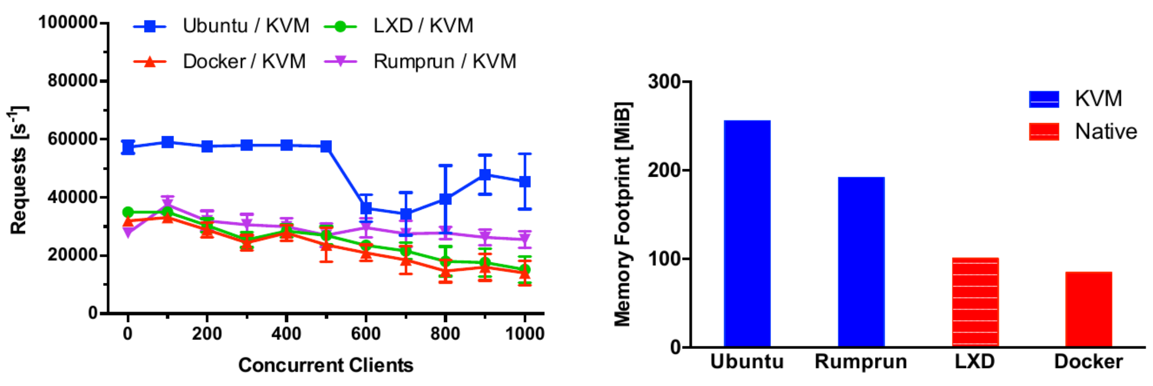
be more CPU efficient than the Rumprun unikernel. Despite that insight, the authors did not perform any other performance comparison, such as memory, image size, or boot time.

3.2.3 Rumprun vs OS^v vs Ubuntu vs Docker

As an effort to draw a comprehensive comparison of lightweight virtualisation techniques, the authors of (Plauth *et al.*, 2017) conducted experiments with two different unikernels (OS^v and Rumprun), containers (Docker and LXD), regular virtual machines (Ubuntu), as well as bare metal deployments of all the prior except OS^v . Their experiments were directed by two major research questions: how fast these virtualisation techniques could be when running different workloads, and which one would be best for on-demand provisioning scenarios. In order to answer those questions, they have used cloud-based application workloads, testing an HTTP server and a key-value store. Each experiment was repeated 30 times.

For the first case, the authors picked Nginx HTTP server 1.8, and configured it to run concurrent requests. Then they used the *weighttp* benchmark tool to collect performance metrics, running it on a dedicated client machine sending requests over local network to a server. This client-server approach can also be observed in (Goethals *et al.*, 2018). As for the results, they unveiled a similar performance between containers and Rumprun. However, unikernels do not have context switches, thus they are supposed to perform better in I/O intensive workloads. The authors attribute this result to Rumprun network stack, based on NetBSD, claimed to be inferior to Linux network stack on top of which the containers were running. The highest throughput was obtained by a regular Ubuntu VM, while Docker consumed the least amount of memory. Rumprun consumed twice as much memory as Docker, and OS^v results were not presented. Figure 11a depicts the throughput for virtualised environments and Figure 11b depicts memory footprints.

Figure 11: Throughput (a) and memory footprint (b) of Nginx 1.8 HTTP Server.

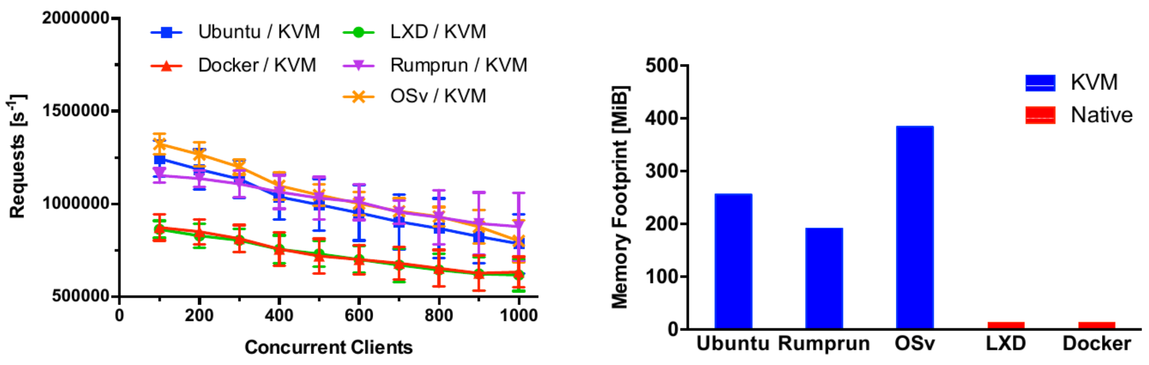


(a) (b)

Source: (Plauth *et al.*, 2017)

In regard to the key-value store experiments, the authors used Redis, an in-memory database, to perform their experiments. Redis throughput was higher when running on unikernels, demonstrating some potential. On the other hand, memory consumption was still kept at minimum by containers. While Rumprun was able to stay below Ubuntu consumption, OS^{V} demanded around 40% more. The authors argue that a memory leak bug in OS^{V} -port of Redis could have caused this higher consumption. However, no reference or evidence is provided to confirm this suspicion.

Figure 12: Throughput (a) and memory footprint (b) of Redis 3.0.1.



(a) (b)

Source: (Plauth *et al.*, 2017)

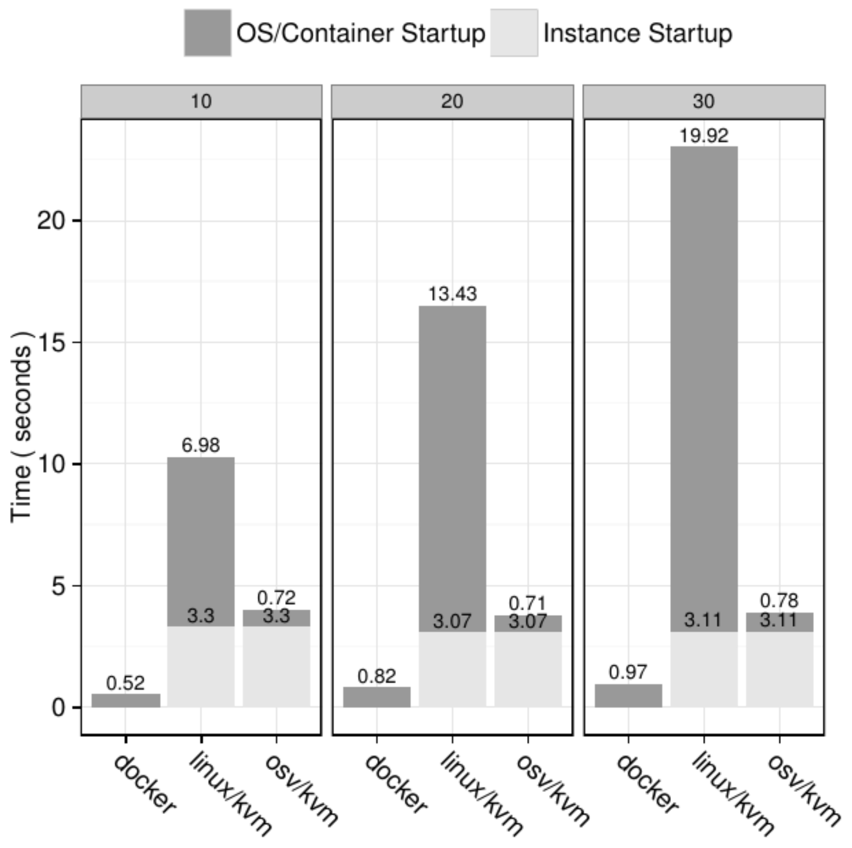
Looking at the bigger picture, it becomes noticeable that unikernels do not outperform more traditional virtualisation techniques out of the box. However, the experiments were limited to Nginx HTTP requests and Redis in-memory reads and writes, leaving room for further investigations on different programming languages and application contexts.

3.2.4 Time Provisioning: OS^{V} vs Docker vs Linux

A relevant aspect to consider when planning cloud-based systems is the required time to provide running instances. This is directly related to availability and reliability of a system. In (Xavier *et al.*, 2016), the authors conducted a study on time provisioning of unikernels in a cloud platform, comparing OS^{V} to Docker and KVM virtual machines on top of Open Stack. OS^{V} was chosen due to its flexibility when it comes to hypervisors to run (KVM, Xen, VirtualBox, VMWare), as well as its acceptance for different runtimes and programming languages. Notice that in this study OS^{V} was deployed on top of KVM, just as its full virtual machine counterparts.

The first evaluation they made was on startup time, which can be seen in Figure 13.

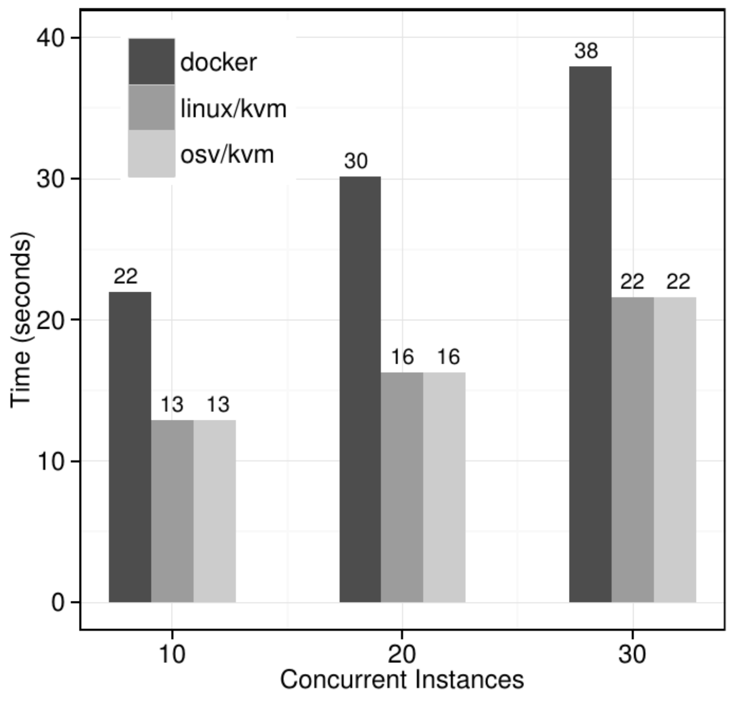
Figure 13: Docker, OS^{v} and KVM startup time for 10, 20, and 30 concurrent instances.



Source: (Xavier *et al.*, 2016)

It is noticeable that the overall time for an OS^{v} instance to be up and running, despite its clear advantage over regular virtual machines, is many times higher than a Docker container. It is explainable by the fact that, even though it is a light one, there is a virtual machine instance to be created by the hypervisor for OS^{v} before it can run, while Docker does not suffer from that delay. On the other hand, they also concluded that when running on Open Stack, and thus considering its overheads, the provisioning time for the total workload of instances to become ready was better for OS^{v} , as can be seen in Figure 14.

Figure 14: Open Stack full workload times.



Source: (Xavier *et al.*, 2016)

The main contributions found in this work were the references for unikernels technology (OSv) and its comparison counterparts, as well as the insight that unikernels do not surpass containers' performance in every aspect, e.g., startup time in a cloud environment.

3.2.5 OS^v vs Docker

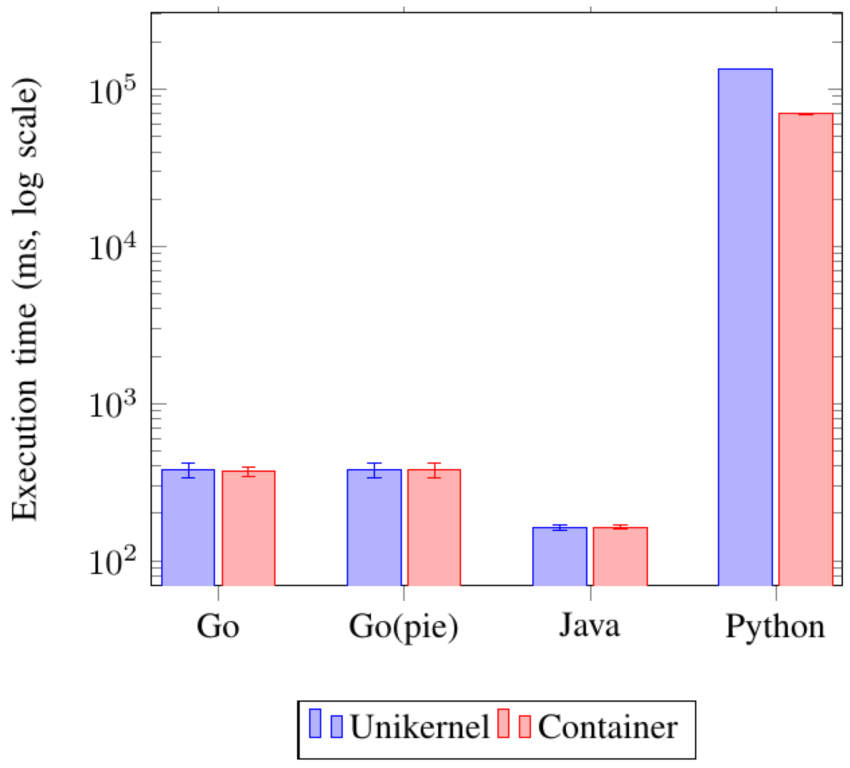
One of the most relevant publications found in literature in the context of this work is (Goethals *et al.*, 2018). It is a performance study of microservices deployed both to unikernels and containers. More specifically, during the experiments the authors utilised the OS^v unikernel and Docker.

The study was motivated by similar reasons to this dissertation, sharing an interest in understanding how unikernels behave and would compare to current production technologies. In order to define which unikernel to use, the authors tested three different options: Rumprun, UniK, and OS^v . The outcome of their testing led to the conclusion that OS^v was the only viable option, due to its counterparts instability (they reported crashes at run time) and steep learning curve. Also, the hypervisor of choice to run OS^v was Xen.

The authors decided to test three different programming languages, comparing the results of two different implementations: one focused on a heavy workload, in the form of a simple bubble sort, and a more realistic one to the context of distributed systems, which was a RESTful API. The languages were Go, Java, and Python. More specifically, they utilised Python 2.7, claiming that at the time of their writing there was no easy way to use Python 3, as many needed system calls were missing in OSᵛ .

In the bubble sort scenario, both Go and Java implementations showed very similar results for OSᵛ and Docker (Go was 3% slower and Java was 1% faster). However, the Python version took twice as long to finish on OSᵛ compared to its Docker deployment. The authors' thought on this result is that it might be caused by the python interpreter implementation, which relies on operations that run slower in a virtual environment, such as array and variable access. This result seems to put unikernels' promised performance in check. It can be observed in Figure 15.

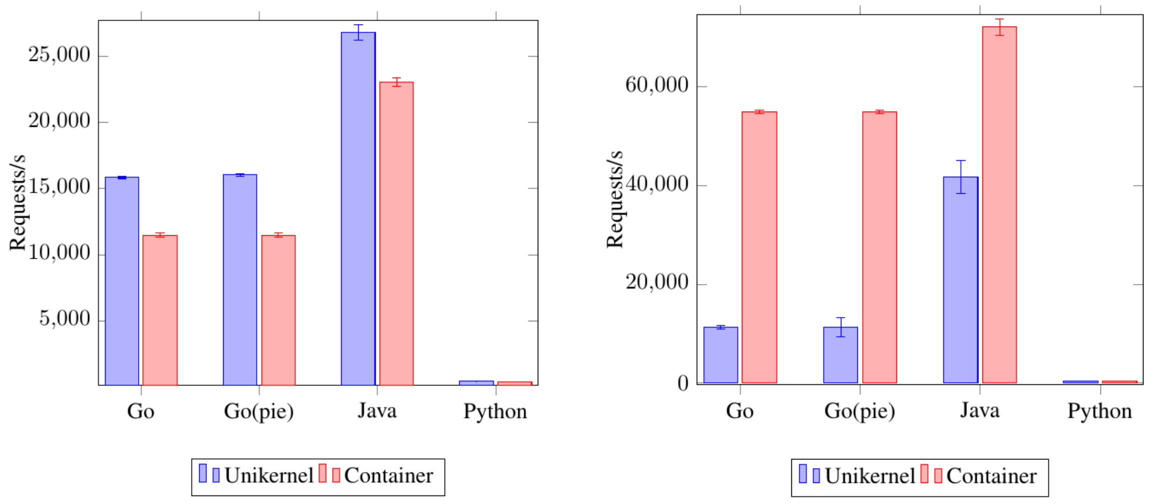
Figure 15: Bubble sort execution time for OSᵛ and Docker.



Source: (Goethals *et al.*, 2018)

Their methodology for testing the API consisted of using a dedicated server to host the unikernels and containers (not at the same time) and a client machine to send requests. They used 40 threads firing 50,000 requests each, and measured throughput, response time, and memory consumption. When running the API in single threaded mode, results reflected a 15% improvement in throughput for Python when using OSᵛ , which was similar to Java's 16%, and around 38% for Go. However, multithreaded results were worse then the previous in OSᵛ . Go only managed to reach 75% of its original performance, while Python stayed 3% below its single-threaded throughput. Java showed an improvement of 60%, but at the cost of using 4 times more CPU. These results, depicted in Figure 16, indicate that OSᵛ is not capable of handling multithreading consistently.

Figure 16: REST service performance of OS^{V} and Docker.

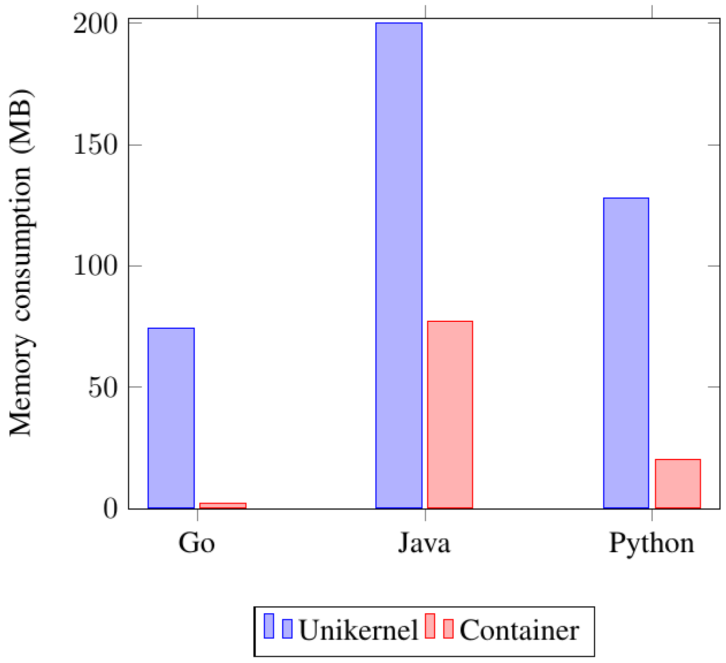


(a) Single-threaded (b) Multi-threaded

Source: (Goethals *et al.*, 2018)

In regard to the remainder metrics, latency was on average 10% better on OS^{V} for Python, slightly better for Go, and stayed the same for Java. The memory footprint, as can be seen in Figure 17, was way higher on unikernels: Java consumed twice as much memory on OS^{V} than on Docker; Python consumed six times more memory on OS^{V} than in Docker, while Go consumed 30 times more. It can be explained by the fact that unikernels have a kernel implementation after all, and containers rely on its host OS kernel.

Figure 17: Memory consumption of OS^{V} and Docker.



Source: (Goethals *et al.*, 2018)

As a quick summary, this work presented the following insights: Python performance seems to be much less affected in OS^{V} than other lower level languages such as Java and Go. OS^{V} was pointed as the most mature unikernel platform to perform the experiments (which was also concluded by authors of other papers, including this dissertation). Unikernels are expected to consume more memory than containers, but can deliver better processing performance depending on the type of workload.

3.3 CONTRIBUTION SUMMARY

In Table 1 we can see a summary of the contributions found in literature in perspective to this dissertation's contributions. Notice that this work was he only one to use an IoT benchmark to perform its experiments, with the most complete metric set, including image size, which was not observed by the others.

Table 1: Comparison to related work

Author	**Platform**	**Testbed**	**CPU**	**Mem**	**Image Size**	**Latency**	**Resp. Time**	**Prov. Time**
Briggs	OSv, Mirage	HTTP				X	X	
Plauth	OSv, Ubuntu, Rumprun, Docker	Memory I/O		X			X	
Goethals	OSv, Docker	HTTP, Bubblesort	X	X				
Xavier	OSv, Docker, Linux	OpenStack						X
Fraga	OSv, Docker	IoT benchmark	X	X	X	X	X	

Source: (Fraga, 2019)

4 BENCHMARK DESIGN AND IMPLEMENTATION

This chapter describes the development process of the IoT benchmark used to obtain the results presented in Chapter 5. It details the chosen technology stack and design decisions, including issues and challenges encountered along the way and how they were solved. The experiments methodology is documented here as well, providing a visualisation of how the results were obtained and how to replicate the tests.

4.1 THE JUNG PROJECT

Three major elements were needed to fulfil this dissertation main objective: an IoT benchmark, a unikernel platform, and a container platform. This Section addresses the first. A suitable IoT benchmark would be microservice-based to meet scalability and flexibility requirements, provide IoT features such as device management, and log performance metrics. The Jung[1] project was then conceived, an archetype of IoT platforms to be used as a benchmark. As the goal was the comparison between unikernels and containers, and not to have a production-ready solution, the design was kept minimalist.

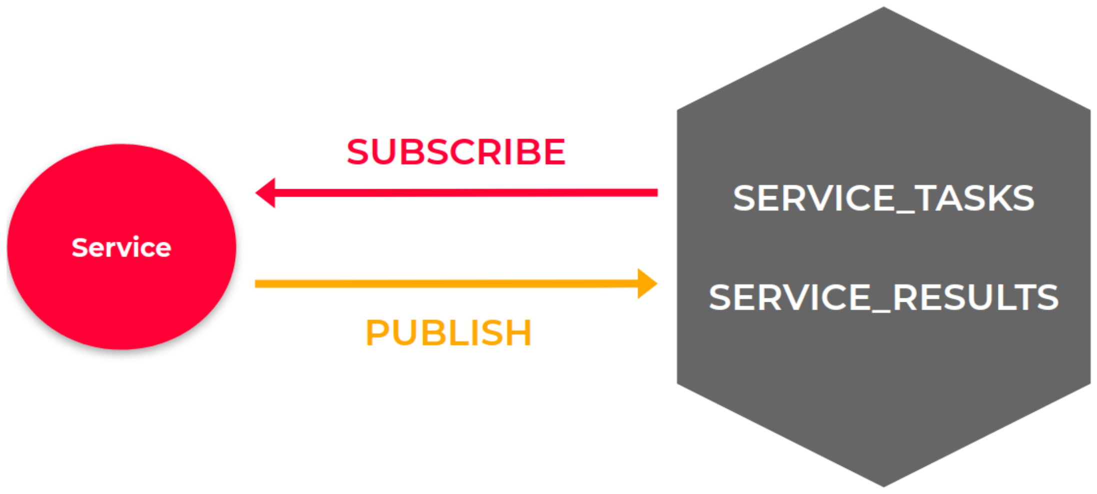
Jung's architecture is portrayed in Figure 19. It is based on the template mentioned in Chapter 2. Therefore, it has a device gateway at the bottom, which is a simulator of multiple IoT devices generating readings and receiving commands. Such a decision was made because device management at gateway level is out of the scope of the research. The layer on top of the device gateway has a microservice-based middleware. The middleware is composed by six microservices: auth manager, user register, device register, device monitor, device commander, and rule engine. They provide authentication, authorisation, user account and device registration, monitoring of device readings, device controlling, and the creation of logic rules to trigger actions automatically. These features were based on literature and commercial solutions (such AWS IoT and Azure IoT) review, found in Guth *et al.* (2018). They address security, management, and minimal human interaction requirements. On top of the middleware there is an API gateway, which exposes a RESTful interface for client applications to consume the services provided by Jung.

As discussed in Section 2.2 in the background chapter, microservices can be orchestrated or choreographed. Jung follows Newman's advice and implements choreography, meaning all of its microservices are aware of how to complete their tasks. Communication relies on a broker-based publish/subscribe[2] pattern, allowing for services to collaborate with each other without knowing anything about their counterparts. This is particularly useful in distributed

[1]The project's name was inspired in Carl Jung, who was a Swiss psychiatrist and psychoanalyst, founder of analytical psychology. One of his main contributions were the 12 Jungian archetypes, which are models used to help shaping human personality.

[2]Publish/subscribe consists of producers creating messages on a given topic, and subscribed consumers receiving those messages. A broker is a known entity which keeps messages and topics.

Figure 18: Jung’s publish/subscribe communication scheme.



Source: (Fraga, 2019)

architectures such as microservices, specially in systems designed to work in scenarios as big as IoT.

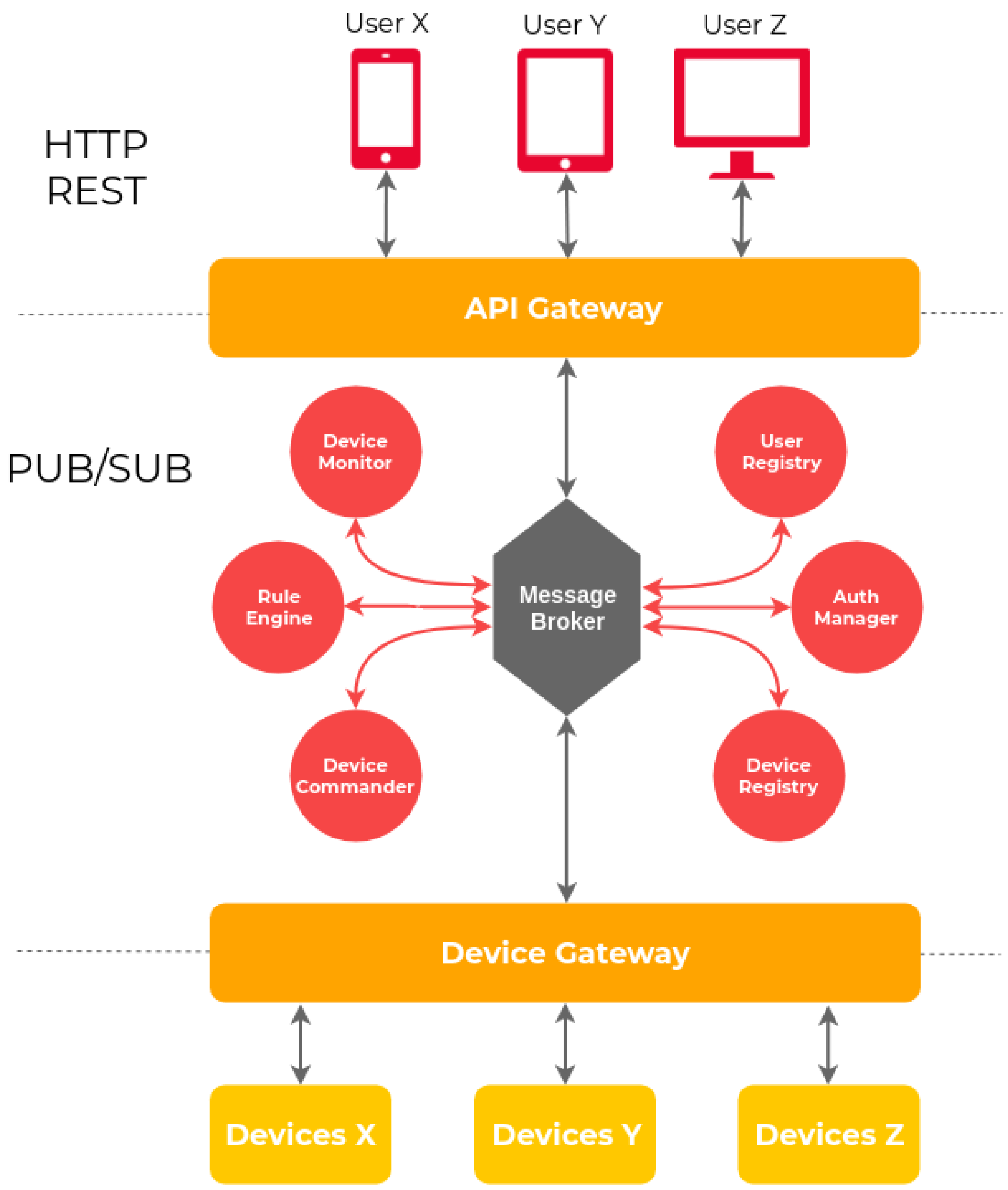
Each microservice uses at least two topics, depicted in Figure 18: one for incoming tasks and another to publish results. Those topics are in the format *microservice_tasks* and *microservice_results*. A service can publish a task into another service’s topic, and subscribe to its result topic to get a response. For simplicity reasons, communication among services is synchronous. It means that after publishing a task, a service waits for its result instead of doing other activities. Every task has a Universally Unique Identifier (UUID), which allows consumers to filter responses.

The element at the centre of Jung’s architecture is the broker responsible for hosting topics and messages. Despite seeming like a centralised approach, it was conceived to be a distributable cluster of brokers instead of a single node. The broker must have such properties to meet IoT scalability requirements. It was implemented with Kafka framework, which provides these features, as will be described in Section 4.2.

4.1.1 API Gateway

An API gateway is often applied to microservices architectures. It consists of a single entrypoint to route application requests to the correspondent services and collect results. Jung’s API gateway translates HTTP REST requests into task messages and publishes them to the proper topics, returning the output when the job is done.

Figure 19: Jung Architecture.



Source: (Fraga, 2019)

4.1.2 Auth Manager

Auth manager is responsible for authorisation and authentication of every request coming into the middleware. It replies to the AUTH task, created by the API Gateway with a username, password, and a device ID depending on the request. It does not use any complex mechanism: for authorisation it fetches user data from user registry and matches the password. When the task needs authentication, i.e. there is a device involved, it fetches the device data from device registry to perform the validation.

4.1.3 User Registry

User registry is a simple in-memory data storage that holds user information, including a list with its devices IDs. It responds to the following tasks: CREATE a user, GET a user, ADD_DEVICE to a user, and ADD_RULE to a user. The ADD_DEVICE task is generated by the device registry when creating a new device record. All the others come from client applications through the API Gateway.

4.1.4 Device Registry

The device registry is analogous to user registry, holding device information, including a reference to the owner of each device. It can CREATE a device, and GET a device. Jung expects a device to be introduced to the system containing the username of owner; device registry will take the username which comes along with CREATE task, and publish an ADD_DEVICE task in the user_tasks topic. An instance of user registry will then update the proper user with the ID of its new device.

4.1.5 Device Monitor

Device monitor subscribes to the "reading_tasks" topic. From this topic it receives readings from multiple devices, forwarded by the Device Gateway inside WRITE tasks. A WRITE task contains the ID of the origin device, the reading value, and a timestamp of when it was "measured". The monitor then can reply to READ tasks, returning paginated results of readings from specific devices.

4.1.6 Device Commander

Device commander replies to COMMAND tasks. To send a command properly, it needs to know which device gateway is responsible for the target device. To obtain this information, the commander creates a GET task for the device registry. Once it receives the response, it adds an entry to an internal cache to avoid further queries.

4.1.7 Rule Engine

The rule engine is a simple If This, Then That (IFTTT) mechanism. It supports equal (==), not equal (!=), greater than (>), and less than (<) operators. A user can then define conditions to be checked within devices' readings, and if the current reading does not comply with the condition, a command is triggered. It implies that the rule engine must check the readings of all the devices in its internal list. This verification is performed every second by creating READ tasks for the device monitor microservice. When a rule is triggered, it creates a COMMAND task for the device commander to process.

When the rule engine receives a CREATE task, it expects to receive a rule definition and a target device. If the device is not in its cache yet, it publishes a GET task to the device registry. It then stores the device in a list of devices with their respective rules. It also responds to GET tasks, returning the rules of a requested device.

4.1.8 Device Gateway

The device gateway simulates a set of devices sending readings and accepting commands. It sends readings to Jung every second, publishing to the "reading_tasks" topic; those readings are then consumed by the device monitor microservice. The gateway keeps a dictionary of devices with their last state. The state value is source of the readings being sent to the middleware, and for the experiments of this dissertation it was just a hardcoded value. By keeping it hardcoded, it was possible to have total control of how many times the rules would be triggered, as the only way to change the readings was by sending commands to the devices.

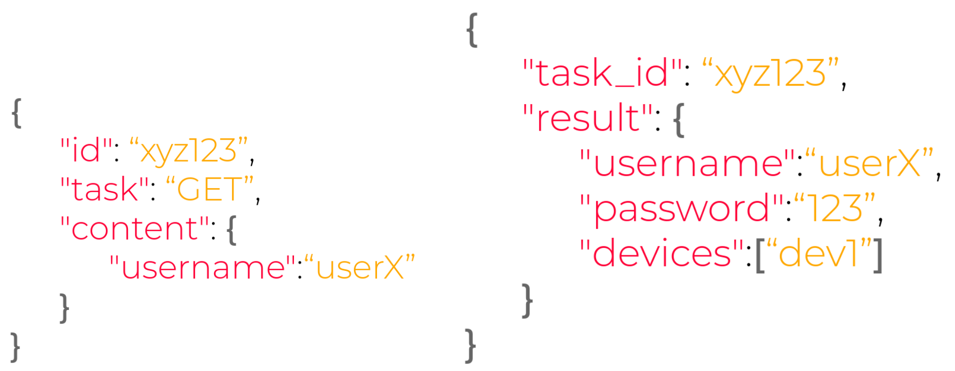
4.2 IMPLEMENTATION

Jung was developed in Python 3, which is a mature, flexible and popular language. It was elected language of the year in 2018 (TIOBE, 2019). Its expressiveness allows for fast development and maintenance, which are desirable characteristics for research endeavours such as this dissertation. Besides, we have seen related work in Chapter 3 that presented results in Python 2 rather than Python 3, leaving room for tests with this implementation that is the present and future[3] of the language.

As aforementioned, Jung's microservices communicate by creating and resolving tasks using specific topics in the message broker. Both tasks and results were kept at an abstract level until now, but they need a serialisation format. The chosen one was the popular JavaScript Object Notation (JSON), due to its large adoption an easy manipulation. Figure 20 provides an example of how a task to retrieve information about a *userX* would look like, as well as a response to that task.

[3]Versions 2 and 3 are based on different specifications; Python 2 will not be maintained past January 1, 2020.

Figure 20: Jung tasks and results serialisation format.



```
{
    "id": "xyz123",
    "task": "GET",
    "content": {
        "username":"userX"
    }
}
```

(a) Task to retrieve *userX*.

```
{
    "task_id": "xyz123",
    "result": {
        "username":"userX",
        "password":"123",
        "devices":["dev1"]
    }
}
```

(b) Response containing *userX* data.

Source: (Fraga, 2019)

The next step was to decide how to implement the message broker itself. The selected technology was Kafka, a data streaming framework created at LinkedIn and open sourced in 2011 under Apache. The reason is that Kafka is designed to be highly distributable[4] and to provide horizontal scalability (Apache Software Foundation, 2017). Moreover, Kafka's topics are load balanced by groups. It means that multiple instances of microservices could share huge loads of tasks by subscribing to a topic under the same group name. Any service inside the group would never receive replicated messages. Services from different domains can use different group names, as all the groups receive all the messages.

Once the middleware communication was resolved, the remainder decision was how to provide the RESTful API at the gateway. Among the many web frameworks[5] available for Python, we chose Falcon due to its microservice focus, minimalist design, and superior performance compared to its counterparts (Falcon Contributors, 2019).

As many of the microservices shared common task-related needs such creating, adding UUID, publishing, and getting results of a specific task, we developed a library called PyJung. It is composed by the JungTasker class, which is widely used on Jung to perform the aforementioned activities, and the JungRegistry class, which provides in-memory storage features to registry services and rule engine.

There is a part of the Jung that was not mentioned yet: the experimentation tools. They consist of auxiliary scripts to help run Jung to process a batch of requests, while collecting performance metrics. The details will be discussed in Section 5.1.2. For now it is worth mentioning what libraries we used to collect those metrics. For CPU and memory, we used psutil library on the machine running Jung. Docker provides a library that could be used to measure containers data, but to keep them under the same methods applied to OS[v] , we just used psutil as

[4]Kafka runs on top of ZooKeeper, a high-performance coordination service for distributed applications. See more at https://zookeeper.apache.org/doc/r3.5.5/

[5]A thorough list is available at https://wiki.python.org/moin/WebFrameworks

well. That was done by keeping track of each process PID when starting Jung.

The source is available under Apache-2.0 license. All the microservices, PyJung library, and experiment tools are submodules of the Jung project. They can be downloaded together from https://gitlab.com/vinicius-masters/jung.

4.3 ISSUES AND CHALLENGES

This Section is dedicated to reporting issues, challenging bugs, and workarounds applied to overcome them. Unikernel technology is recent (six years old at the time of writing), especially considering its goal to substitute traditional OSs for cloud deployments; therefore, problems were expected to be faced.

The experimental portion of this research started with unikernel prospection. Theoretical review pointed out that the most mature option would be MirageOS, but at the cost of learning a new stack as it works exclusively with OCaml language. Besides, it only works on Xen hypervisor, reducing its reach even more. We aimed at more generalist alternatives which could accept legacy software and achieve better adoption. With this goal in mind, the language-specific flavours were put aside. The most promising options then were POSIX-compliant unikernels, such as Rumprun and OS^{v} .

When comparing OS^{v} to Rumprun, we noticed that the first had approximately six times more results when searched in Google Scholar (430 vs 77), meaning more research resources. Its community were also more active, counting with five times more contributors (94 x 18) on GitHub and three times more commits (7,7k x 2,3k). Despite these indicators, we tried both of them. Rumprun did not compile at the beginning and required manual efforts to configure networking. We tried to build it using UniK tool, but it crashed every time we ran it. OS^{v} , on the other hand, worked roughly as its tutorial stated it would. Considering the odds, we moved forward with OS^{v} from that point on.

Foreseeing that OS^{v} would be the main source of bugs, we performed tests step-by-step along with the implementation. The first one was to run a Kafka client for Python inside OS^{v} . The first choice was the popular confluent-kafka-python library, provided by Confluent[6]. It crashed as soon as imported, reporting a missing ELF[7] tag. There was no simple solution available at the time. The next option was kafka-python library. This option could be imported, but its default clients (both KafkaConsumer and KafkaProducer) would crash on creation due to a socket assertion failure. The workaround was to use kafka-python deprecated clients, SimpleConsumer and SimpleProducer. These two required more coding effort to listen for messages, retry connections and setting message offsets, though.

[6]Confluent is company founded by Kafka's creators.

[7]Executable and Linkable Format. It is a binary format for executable files. See more at https://www.linuxjournal.com/article/1059

5 EXPERIMENTAL EVALUATION

This chapter thoroughly explains how the experiments were designed, implemented, and performed. The next section starts by describing the scope of the tests, the observed metrics and setup configuration. Then it details the execution steps to obtain the results, which are presented thereafter.

5.1 SCENARIO

The most studied IoT scenarios (with real-world experimentation) are smart home, smart healthcare, and smart city. Such scenarios were recurrently tested with 23 to 28 devices, mostly capable of sensing temperature, acceleration, light, and humidity (Morais *et al.*, 2019). Combining this piece of information with the experiment numbers from Chapter 3, which shows results based on tens of thousand requests, we defined our testbed as follows: a set of 30 users owning 30 devices each, reflecting the number of devices used in real testbeds in literature. Each device has 1 conditional rule which triggers a command, to ensure every aspect of the platform will be seen during the tests, as the rule engine will constantly monitor device readings and generate commands based on its rules. Each user would then send 500 requests concurrently to the API gateway, summing 15k requests per iteration. The request dataset is organised in a per-user fashion, as described in Table 2. We assembled the request dataset according to the aforementioned IoT scenarios, which are expected to be monitoring-intensive as well as actuation-heavy, considering the context of smart environments managing themselves and accepting human interaction.

5.1.1 Metrics

We observed the following metrics:

- **CPU usage** - Percentage of time a process spent using CPU resourcer during its execution. Measured using *psutil* library for Python3.

Table 2: Requests per user

Endpoint	**Verb**	**Per User**	**Total**	**%**
/users/<username>	GET	25	750	5
/users/<username>/devices/<device_id>	GET	25	750	5
/users/<username>/devices/<device_id>/readings	GET	225	6750	45
/users/<username>/devices/<device_id>/commands	POST	225	6750	45
TOTAL		**500**	**15000**	**100**

Source: (Fraga, 2019)

- **RAM usage** - measured using *psutil* library for Python3
- **Latencies** - measured with Apache JMeter
- **Throughput** - measured with Apache JMeter
- **Response Time** - measured with Apache JMeter
- **Binary size** - collected from server with *du* command

5.1.2 Execution

To achieve statistically significant results, the experiments were configured to run multiple times. The default value was set to 30, meaning that for each platform the experiment was repeated 30 times with 15k concurrent requests each. In total, 900000 requests were analysed.

Two components are responsible for the experiments execution: a controller script and a runner script. Each of these scripts is running on a dedicated machine; the goal was to save as much resources as possible for the Jung host to process the requests. Both controller and runner communicate to each other using Kafka topics named *controller* and *runner*.

The runner script is responsible for processing two types of commands sent by the controller: RUN and STOP. When it receives a RUN command, it also comes with parameters for the execution ahead. Those parameters include the platform (OS^v or Docker) and the list of microservices to start (usually all the six of them), amount of users, devices per user, and readings and rules per device. Once the platform is initialised and ready to start, the runner sends a READY message to the controller. At this point Jung is running in separate processes, and the runner process is just listening for more commands. The next command is expected to be STOP, which makes to runner collect metrics data from Jung's execution, kill Jung (which is composed by multiple processes, one per microservice plus Device Gateway and API Gateway) and send the results to the controller.

The controller script has a configuration file which defines what platforms to test (OS^v and Docker), how many iterations should be executed (30), and all the aforementioned parameters needed by the runner. For each platform the controller runs a given amount of iterations. After sending the RUN command to the runner, it waits for a READY message. Once received, the controller starts the JMeter tool as an external process (JMeter uses a configuration file, which path is in the controller configuration file as well). JMeter sends the HTTP requests described in subsection 5.1, while collecting metrics such as latency and throughput. These results are stored in a JTL[1] file. After the requests are over, the controller sends a STOP command and expects to receive CPU and memory data from Jung's execution. It then stores the iteration results in a list, which will be used to produce a summary at the of the platform's iterations.

[1]Output file used to automatically generate a report dashboard with JMeter.

Table 3: CPU Usage

Microservice	OS^v (%)	Docker (%)
User Registry	21,279 $\pm 0,515$	6,925 $\pm 0,060$
Auth Manager	24,633 $\pm 0,679$	9,568 $\pm 0,094$
Rule Engine	58,326 $\pm 1,375$	33,404 $\pm 0,157$
Device Registry	15,411 $\pm 0,379$	3,150 $\pm 0,027$
Device Monitor	60,374 $\pm 1,320$	34,509 $\pm 0,146$
Device Commander	38,328 $\pm 1,288$	24,699 $\pm 0,233$

Source: (Fraga, 2019)

The final result of the experiments is a JSON file for each platform containing CPU and memory data summarised from all the iterations, as well as a JTL file with accumulated requests data. Those are the files used to produce the charts of Section 5.2. The entire experiment life-cycle is demonstrated in Figure 21.

5.2 RESULTS

This section presents results of each one of the metrics mentioned in Section 5.1.1, followed by a reasoning about the numbers of OS^v and Docker. Just as a reminder, every result is based on the mean and standard deviation values of 30 iterations. During each iteration, Jung processed 15000 concurrent requests.

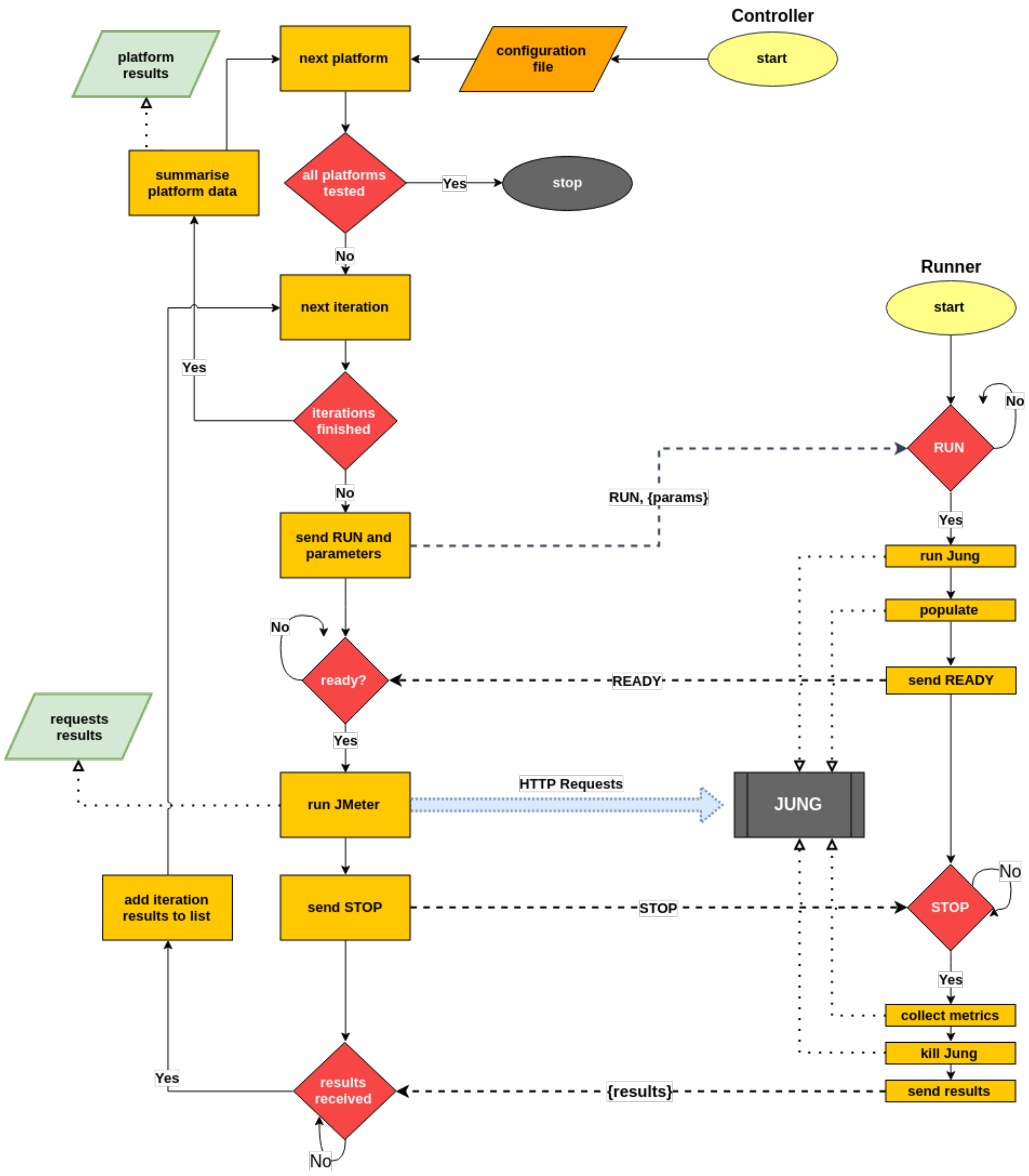
5.2.1 CPU Usage

The first observed metric was the CPU usage. The presented values reflect the amount of time that each microservice spent using the available CPU resources during the execution. OS^v finished the experiment in 132 minutes, while Docker version finished in 110 minutes. Therefore, it is visible in Figure 22 that containers completed their tasks more efficiently, consuming less CPU and yet finishing the tasks in less time. Standard Deviation (SD) was too small to be seen in the chart, but the values can be checked in Table 3.

The diversity among microservices results can be explained by the characteristics of the requests. As only 5% of them were related to get user and device information directly, both registry services were expected to require lower CPU activity. However, User Registry is a little higher than Device Registry, and Auth Manager is higher than both. The reason is simple: besides the GET tasks sent by the clients, Auth Manager creates additional requests to check user and device information. It checks for user information 100% of the time, but only 95% for devices.

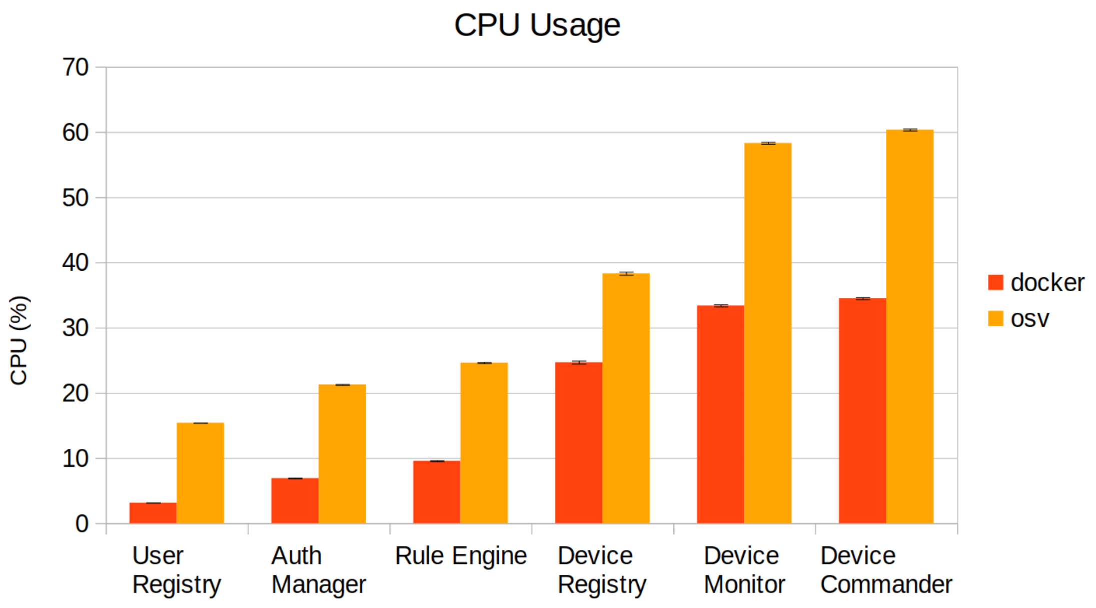
The workload was more CPU intensive for the 3 remaining microservices. 45% of the requests were to get readings, and 45% were to send commands. But there was also the fact that every device had a programmed rule, meaning Rule Engine was checking on readings every

Figure 21: Jung experiments flowchart.



Source: (Fraga, 2019)

Figure 22: CPU usage of OS^v vs Docker. Less is better.



Source: (Fraga, 2019)

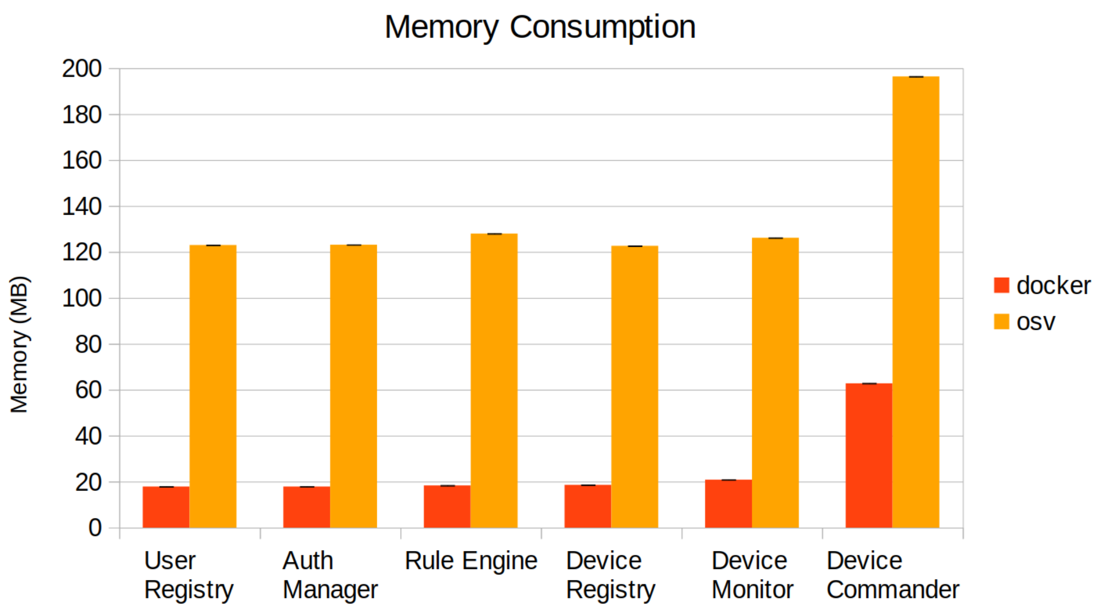
second, thus generating more tasks to Device Monitor. When the command requests start to hit Jung, every command triggers a rule, that creates an opposite command task in response. Hence, Rule Engine kept Device Monitor busy even during the command-focused requests and doubled Device Commander load.

The difference between Monitor and Commander is further explained by their inner activities. The Monitor collects readings from Device Gateway and delivers them when requests, processing pagination, while Commander just sends a single key:value message.

5.2.2 Memory Usage

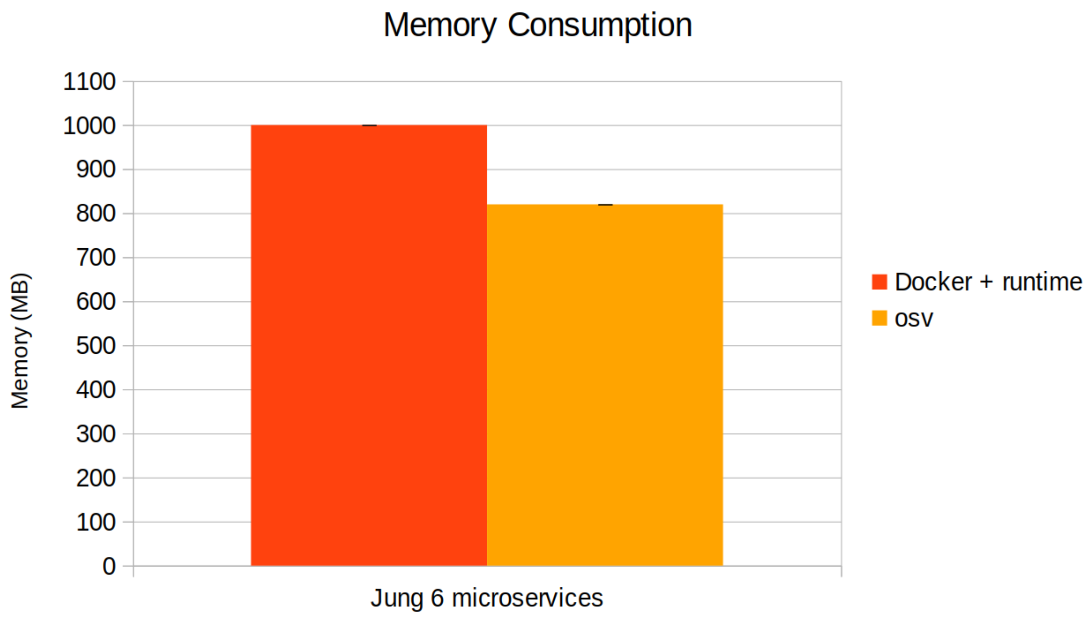
The memory results in Figure 23 resemble the Python 2 results from Section 3.2.5, as OS^v usage is approximately 6 times higher than Docker. For most of the microservices the reasons are also similar: OS^v has kernel code running, while Docker containers are limited to application code. All the drivers and resource management are running in the host OS, thus its memory usage is invisible to Docker. Just as for CPU results, SD was low. Table 4 presents the full numbers for each microservice. However, there is a crucial element for Docker to run that was not considered in the related work, and that increasingly uses memory as well: Docker runtime, which is the dockerd daemon process. As can be seen in Figure 24, when summing the memory usage of all six microservices plus the runtime, Docker numbers become higher than OS^v . The runtime alone uses an average of 888MB when running Jung, unveiling itself as a major source of memory usage.

Figure 23: Memory usage of OS^v vs Docker.



Source: (Fraga, 2019)

Figure 24: Total memory usage of OS^v vs Docker, including Docker runtime.



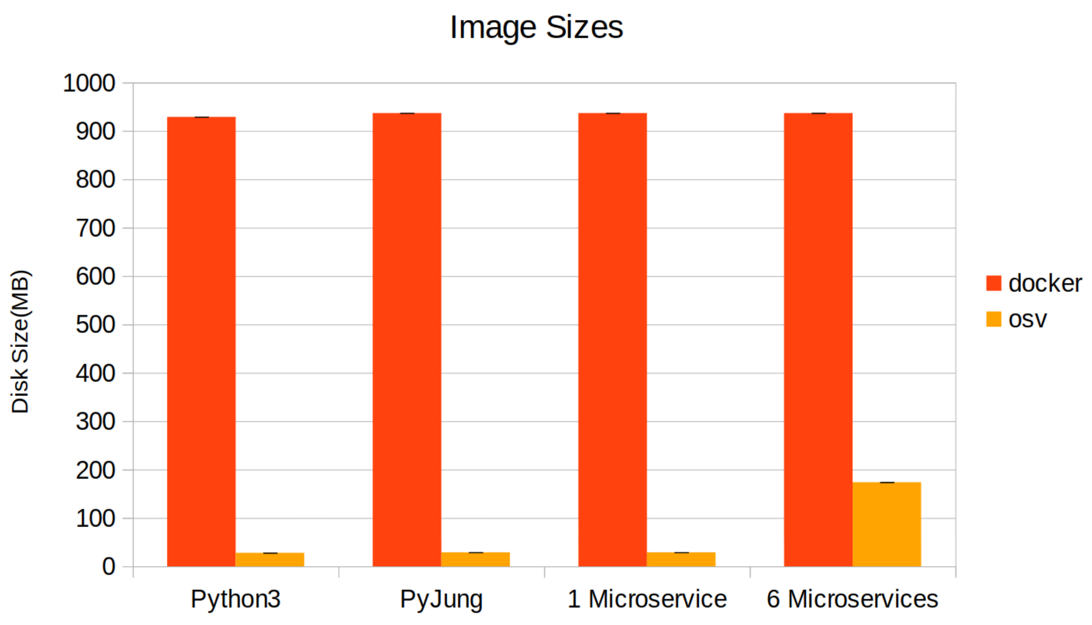
Source: (Fraga, 2019)

Table 4: Memory Usage

Microservice	**OS^v (MB)**	**Docker (MB)**
User Registry	123,015 $\pm 0,908$	17,889 $\pm 0,122$
Auth Manager	123,164 $\pm 0,841$	17,891 $\pm 0,117$
Rule Engine	128,002 $\pm 0,985$	18,373 $\pm 0,167$
Device Registry	122,662 $\pm 1,004$	18,616 $\pm 0,144$
Device Monitor	126,182 $\pm 1,14$	20,866 $\pm 0,133$
Device Commander	196,420 $\pm 1,049$	62,834 $\pm 0,156$

Source: (Fraga, 2019)

Figure 25: Images sizes on disk.



Source: (Fraga, 2019)

5.2.3 Disk Space

Small image size is a unikernel idiosyncrasy, and OS^v is not different. While Docker default[2] image for Python 3 interpreter has 937MB, OS^v version has only 28MB. Adding PyJung increases 8MB to Docker and 1MB to OS^v . The microservices are in KB magnitude (ranging from 1 to 5KB), reflecting no practical difference. However, Docker still has an advantage in that matter: the base image is shared among all microservice instances. The last bar in Figure 25 illustrates the accumulated size that all six microservices would have on disk. For OS^v , they stack. For Docker, only the KB of each microservice stacks on top of a shared base image. It indicates that there is a threshold for running multiple instances of OS^v before it starts consuming more disk space than Docker containers.

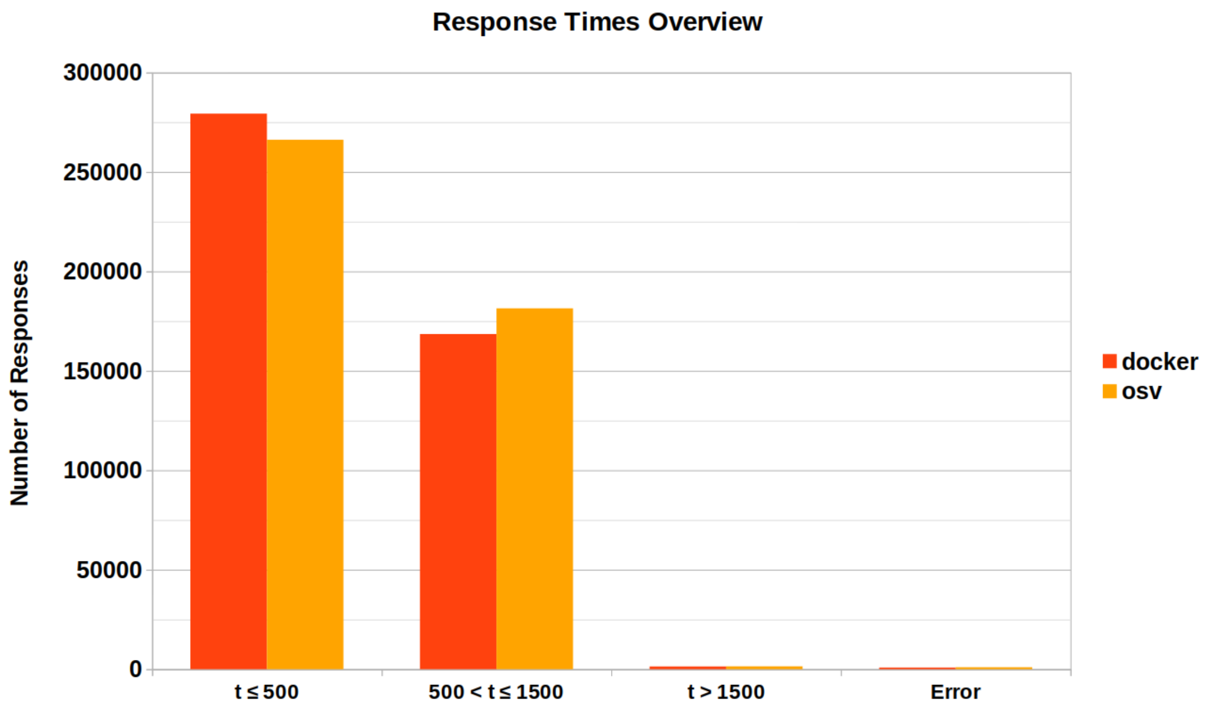
[2]There are other Docker images optimised for image size; in this research we opted to avoid optimisations in both Docker and OS^v and stick to the defaults.

Table 5: Requests Summary

Platform	Execution		Response Times			Throughput	Network	
	KO	Error %	Avg (ms)	Min (ms)	Max (ms)	Transactions (s)	Received (KB/s)	Sent (KB/s)
OS^V	957	0,21	391,67	7	29522	56,95	25,01	13,15
Docker	802	0,18	366,65	9	4583	67,95	29,84	15,69

Source: (Fraga, 2019)

Figure 26: OS^V vs Docker response times.



Source: (Fraga, 2019)

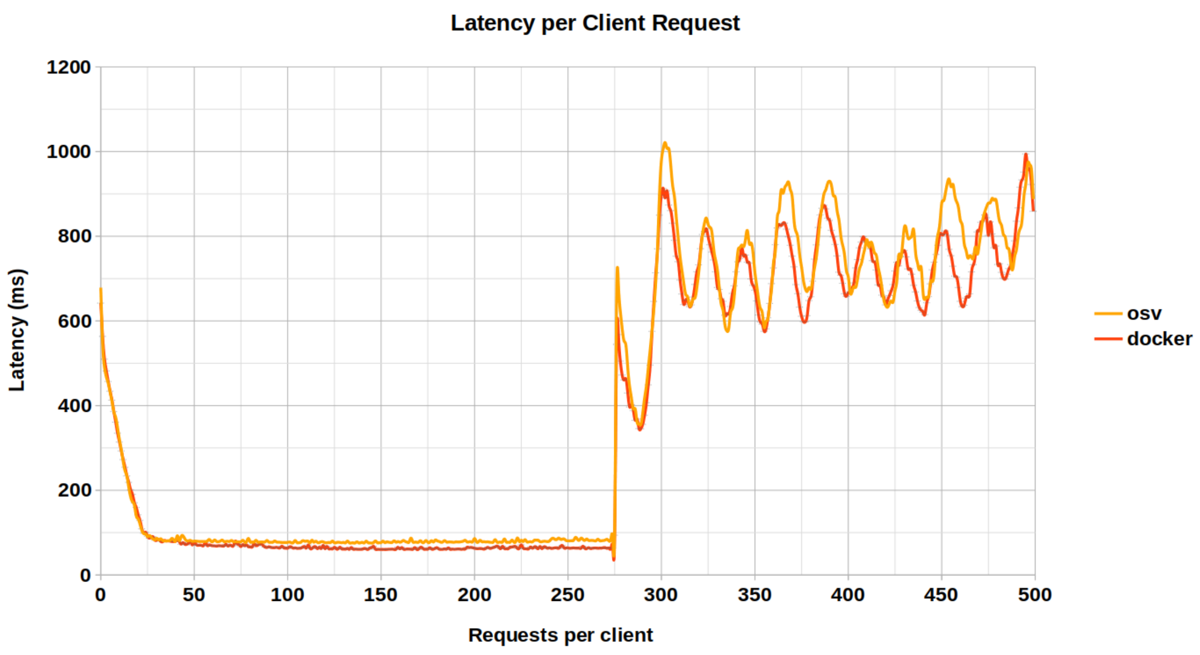
5.2.4 Networking

Table 5 is the starting point to analyse networking metrics. Similarly to CPU and memory, the network results are in favour of Docker. The error rate was only 0,03% higher on OS^V , but while its response time was on average 25ms longer, it had spikes of 29 seconds. Docker delayed 4,5 seconds at maximum. As a consequence of its quicker responses, Docker achieved almost 20% higher throughput values.

Despite the spikes of OS^V response times, when grouping the number of responses, as depicted in Figure 26, it becomes apparent that both platforms had a similar behaviour, denoting consistency between their results. The majority of the responses stayed under 500ms, followed by another big group between 500 and 1500ms. The outliers amount was almost the same: 1399 for OS^V and 1349 for Docker.

When looking at the latencies in Figure 27, what can be seen is the behaviour of each

Figure 27: Latencies of OS^{v} vs Docker.



Source: (Fraga, 2019)

request group. The X-Axis represents the data of those 500 requests dataset mentioned in Section 5.1, being concurrently transmitted by 30 clients. The first 50 requests in the chart are GET user and GET device, followed by 225 GET readings. Starting from this point, the curve goes from stable values under 100ms to oscillating hundreds that surpass 1 second at times; when considering the sparse deviation values in the background, an instability becomes clear. It is not by coincidence that is starts to happen when the requests change from getting readings to sending commands, triggering the device rules, which sends more commands. The chain effect causes increased CPU activity, as concluded in Section 5.2.1.

OS^{v} has consistently demonstrated to be more vulnerable than Docker through the experiments, and the same has happened at latencies level. However, Docker did not perform much better from this perspective.

5.3 DISCUSSION

We performed 30 iterations of a set of 30 clients running concurrently, sending 500 requests each, summing up to a total of 900000 analysed requests. While Jung was processing these requests, separate processes were monitoring its resource consumption, including network metrics. The big picture of the results unveils a very similar behaviour between both OS^{v} and Docker, which at least indicates that OS^{v} did not break during the execution. However, its numbers for CPU usage, response latencies, and throughput were outperformed by Docker containers. The memory usage, when looking at microservice level, was higher on OS^{v} ; this is explained by the fact that OS^{v} kernel is accountable in its memory usage, and the same is not true for Docker, which runs on top of the host OS kernel. However, when taking Docker runtime into account, it became notOS^{v} disk images were more than 100x smaller, as expected due to the minimalist design of unikernels.

Nevertheless, the CPU usage was expected to be smaller on OS^{v} as well, but the results showed otherwise. Our workload at the microservices level was majorly composed by memory access and Kafka communication, which executes a binary protocol on top of TCP. According to OS^{v} release paper, it outperforms Linux on TCP flows, meaning it should reduce latencies and achieve higher throughput. Both technologies were configured to run with bridge networking; both were given 2 cores to execute each microservice. Considering these conditions, further research is required to determine the reasons behind such a performance gap, going deep into implementation details. It is worth mentioning that OS^{v} has a very specific implementation of scheduling for its threads and network stack, leaving room for granular investigation and future improvements.

It is also noticeable that OS^{v} images are smaller than the Docker images used. Such small images can be transferred though the network at a cheaper cost, allowing for fast and flexible migrations, for example. Besides saving space in cloud deployments of complex systems, e.g. IoT platforms, this characteristic opens a range of scenarios in which unikernels such as OS^{v} could be valuable. For example, deploy unikernels to networking hardware, such as switches or routers, embedding it with smart software at a low space cost. Another possibility would be to deploy unikernels to edge computing devices, such as simple Raspberry Pis.

In section 3.2.5 we have seen that the results for Python 2 showed containers outperforming OS^{v} for bubble sort, and OS^{v} in slight advantage on network intensive workloads. Our results indicate that when running Python 3 OS^{v} is not capable of outperforming Docker as well. Despite having different specifications and implementations[3], both versions of the language are compiled into bytecode and then interpreted by a Python Virtual Machine, and certainly share common ground. The explanation for OS^{v} weaker performance with Python might be related to the language internal design as well, considering OS^{v} idiosyncrasies.

As we can see in Table 6, the overall results point that OS^{v} and Docker obtained advantages in different aspects of the tests. The big picture leads to see different use cases for each technology, depending on requirements and available resources; an important observation is that OS^{v} can be used to deploy IoT systems, and trusted to deliver the same results as established platforms such as Docker, given the same inputs. It then draws a baseline of what can be explored with this technology.

[3]Python language has implementations in multiple languages, e.g. C, Java, C#, JavaScript. All the mentions in this dissertation refer to CPython, which is the official and default implementation.

Table 6: Results summary.

Metric	**Best Platform**
CPU	Docker
Memory	OS^{V}
Image Size	OS^{V}
Networking	Docker

6 CONCLUSION

6.1 SUMMARY AND FINAL THOUGHTS

Unikernels started to be disseminated six years ago as a resource efficient and safe alternative to deploy software to the cloud. A perfect match to their minimalist design would be the microservices architectural pattern. These are also the same years of the Internet of Things rise, leading the industry into its 4.0 version with massive scalability and security requirements. Despite both technologies apparent affinity, there is a lack of studies correlating the two. This dissertation had the objective of filling this gap, defining a baseline on unikernels behaviour in IoT context. To achieve this goal, we performed experiments comparing unikernels to containers, using a microservice IoT platform as testbed.

The architecture design was guided by IoT surveys. We developed Jung, an archetype of an IoT platform. Jung follow the literature patterns and has a device gateway at its bottom layer, followed by a middleware, and an API gateway as the application layer at the top. The middleware has a choreographed and synchronous microservice architecture. Its message broker had to be scalable, thus we selected Kafka for the job.

Once the architecture was defined, we went through the list of available unikernel flavours and decided for OS^v . OS^v was designed to be a cloud OS, matching our goal to test cloud-oriented IoT deployments, and it is almost[1] 100% POSIX compliant, allowing to run legacy applications and develop new ones in many different languages. Moreover, it worked as its tutorials stated, in contrast to other unikernels and specific tooling such as Rumprun and UniK. For the container platform, on the other hand, we just picked the most popular one, Docker, which worked out of the box.

After analysing the results, the overall conclusion is that OS^v has a consistent behaviour in perspective to established techniques such as Docker, meaning it can be used to deploy an IoT system and will deliver the same outputs as Docker when both are fed with the same inputs. But it still needs improvements to achieve the same performance level and development stability. While it required adaptations workarounds to properly execute simple Kafka clients, Docker containers worked out-of-the-box. On the other hand, it takes advantage of being hardware isolated and exposing a much smaller attack surface, noticeable by its tiny images. Therefore, instead of granting both performance and security, OS^v offers a tradeoff between security and performance when compared to containers, at least for CPython. We then conclude that unikernels meet the requirements for IoT solutions with development restrictions, being a viable option for microservice deployments due to their small functionality set.

[1]Except for system calls such as fork() and exec(), given that unikernels are designed to be single-process.

6.2 CONTRIBUTIONS

The development of this dissertation led to contributions ranging from theoretical insights to open source tooling for further research work. The first one is the comparison between unikernels and containers in the IoT context. Jung was designed to be an archetype of an IoT platform, based on literature review and industry solutions, to work as an IoT benchmark. Prior unikernel evaluations were limited to compare raw throughput and resource consumption while performing simple tasks, such as responding to contextless HTTP requests, and running sort algorithms such as bubble sort. Jung experiments provided prime matter for reasoning, leading to a baseline understanding of how an IoT platform behaves when running o top of unikernels, and also how it compares to a Docker-based version.

Besides the experimental results, Jung itself is a contribution. It is a feasible microservices-based benchmark for IoT platforms; based on related work which identified an architectural pattern, Jung is a step forward into specifying components to be implemented, including a publish/subscribe communication scheme among the services. Its python implementation was also made open source [2] under Apache 2 license, including a repository with the tools needed to run the experiments with different configurations.

Moreover, in section 4.3 we listed issues and challenges faced during the development, as well as the applied workarounds to overcome the obstacles. It can help other researchers to deploy their own applications to OS^v faster.

6.3 FUTURE WORK

Multiple research paths still open after the conclusion of this work. We have listed a number of possibilities below:

- Use Jung and its tooling to compare different unikernels and containers, such as Rumprun, rkt[3] and Microsoft containers.
- Compare Kafka performance when deployed to OS^v and Docker.
- Test Jung with real devices and compare the results to the simulation of this dissertation.
- Improve Jung security with state-of-the-art techniques to compare pentesting results between unikernels and containers.
- Implement Jung architecture on compiled programming languages, such as C and Go, to rerun the experiments and compare the results.

[2]Available at https://gitlab.com/vinicius-masters/jung
[3]Main competitor of Docker, rkt (pronounced "rocket") is container technology from CoreOS.

- Evaluate performance of device gateways deployed as unikernels on edge devices.
- Analyse the maintenance capabilities of unikernels in perspective to containers and VMs.

REFERENCES

Ahmed, E., Yaqoob, I., Gani, A., Imran, M., & Guizani, M. (2016). Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges. *IEEE Wireless Communications*, 23(5):10–16.

Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., & Ayyash, M. (2015). Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys Tutorials*, 17(4):2347–2376.

Apache Software Foundation (2017). *Apache Kafka - a distributed streaming platform*. Accessed February 3, 2019.

Ashton, K. (2009). *That 'Internet of Things' Thing*. Accessed February 3, 2019.

Bak, P., Melamed, R., Moshkovich, D., Nardi, Y., Ship, H., & Yaeli, A. (2015). Location and context-based microservices for mobile and internet of things workloads. In *2015 IEEE International Conference on Mobile Services*, 1–8.

Blatstein, M. (2016). *Docker Acquires Unikernel Systems to Extend the Breadth of the Docker Platform*. Accessed February 3, 2019.

Briggs, I., Day, M., Guo, Y., Marheine, P., & Eide, E. (2014). A performance evaluation of unikernels. *Technical report, tech. rep, Tech. Rep.*

Chebudie, A. B., Minerva, R., & Rotondi, D. (2015). *Towards a definition of the Internet of Things (IoT)*. PhD thesis.

Chris Richardson, F. S. (2016). *Microservices: from design to deployment*. NGINX.

Cloudius Systems (2019). *OSv Linux ABI Compatibility*. Accessed June 2, 2019.

CloudKernels (2019). *Run a rumprun unikernel on a RPi3*. Accessed June 2, 2019.

De Lucia, M. (2017). A survey on security isolation of virtualization, containers, and unikernels. Technical report.

Docker (2017). *LinuxKit*. Accessed February 3, 2019.

Docker (2019a). Docker hub. Accessed May 17, 2019.

Docker (2019b). What is a container? Accessed May 16, 2019.

Duncan, B., Happe, A., & Bratterud, A. (2017). Cloud cyber security: Finding an effective approach with unikernels. In Sen, J., editor, *Advances in Security in Computing and Communications*, chapter 2. IntechOpen, Rijeka.

Elphinstone, K., Zarrabi, A., Mcleod, K., & Heiser, G. (2017). A performance evaluation of rump kernels as a multi-server os building block on sel4. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, 11:1–11:8.

Engler, D. R., Kaashoek, M. F., & O'Toole, Jr., J. (1995). Exokernel: An operating system architecture for application-level resource management. *SIGOPS Oper. Syst. Rev.*, 29(5):251–266.

Falcon Contributors (2019). *Falcon Benchmarks*. Accessed February 3, 2019.

Fraga, V. (2019). A comparison between OS^{v} unikernels and docker containers as building blocks for an internet of things platform. Master's thesis, Federal University of Pernambuco.

Gil, D., Ferrández, A., Mora, H., & Peral, J. (2016). Internet of things: A review of surveys based on context aware intelligent services. *Sensors*, 16:1069.

Goethals, T., Sebrechts, M., Atrey, A., Volckaert, B., & De Turck, F. (2018). Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications. 1–8.

Google (2019). *Deploying Containers on VMs and Managed Instance Groups*. Accessed June 2, 2019.

Guth, J., Breitenbücher, U., Falkenthal, M., Fremantle, P., Kopp, O., Leymann, F., & Reinfurt, L. (2018). A detailed analysis of iot platform architectures: Concepts, similarities, and differences. In *Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*, 81–101. Springer.

Hewlett-Packard (2005). *httperf, a tool for measuring web server performance*. Accessed May 3, 2019.

Kennedy, J. B. (1926). *An interview with Nikola Tesla*. Accessed February 3, 2019.

Kerrisk, M. (2013). Namespaces in operation, part 5: User namespaces. Accessed May 16, 2019.

Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., & Zolotarov, V. (2014). Osv—optimizing the operating system for virtual machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 61–72.

Kraijak, S. & Tuwanut, P. (2015). A survey on iot architectures, protocols, applications, security, privacy, real-world implementation and future trends. In *11th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM 2015)*, 1–6.

KVM (2016). Main page — kvm,. Accessed May 16, 2019.

Lea, R. & Blackstock, M. (2014). Smart cities: An iot-centric approach. In *Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing*, 12:1–12:2.

Lee, S., Bae, M., & Kim, H. (2017). Future of iot networks: A survey. *Applied Sciences (Switzerland)*, 7(10).

Levine, I. (2018). *UniK: Build and Run Unikernels with Ease*. Accessed February 3, 2019.

Madhavapeddy, A., Leonard, T., Skjegstad, M., Gazagnaire, T., Sheets, D., Scott, D. J., Mortier, R., Chaudhry, A., Singh, B., Ludlam, J., Crowcroft, J. A., & Leslie, I. M. (2015). Jitsu: Just-in-time summoning of unikernels. In *NSDI*.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., & Crowcroft, J. (2013). Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472.

Marquez, E. (2018). The history of container technology. Accessed May 16, 2019.

Mell, P. M. & Grance, T. (2011). Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States.

MIKELANGELO (2015). *Capstan, a tool for packaging and running your application on OSv.* Accessed February 3, 2019.

Morabito, R., Cozzolino, V., Ding, A. Y., Beijar, N., & Ott, J. (2018). Consolidate iot edge computing with lightweight virtualization. *IEEE Network*, 32(1):102–111.

Morais, C. M. d., Sadok, D., & Kelner, J. (2019). An iot sensor and scenario survey for data researchers. *Journal of the Brazilian Computer Society*, 25(1):4.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

Ngu, A. H., Gutierrez, M., Metsis, V., Nepal, S., & Sheng, Q. Z. (2017). Iot middleware: A survey on issues and enabling technologies. *IEEE Internet of Things Journal*, 4(1):1–20.

Nominum (2012). *queryperf DNS query performance testing tool*. Accessed May 3, 2019.

Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2017). Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, PP:1–1.

Pavlicek, R. (2017). Existing unikernel projects. In *Unikernels: Beyond Containers to the Next Generation of Cloud*, chapter 3, 27. O'Reilly Media.

Plauth, M., Feinbube, L., & Polze, A. (2017). A performance evaluation of lightweight approaches to virtualization.

Popek, G. J. & Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421.

Richards, M. (2016). *Microservices vs Service-Oriented Architecture*. O'Reilly.

RWTH Aachen University (2017). *HermitCore - A lightweight unikernel for a scalable and predictable runtime behavior.* Accessed June 3, 2019.

Saadeh, M., Sleit, A., Qatawneh, M., & Almobaideen, W. (2016). Authentication techniques for the internet of things: A survey. In *2016 Cybersecurity and Cyberforensics Conference (CCC)*, 28–34.

Shah, S. H. & Yaqoob, I. (2016). A survey: Internet of things (iot) technologies, applications and challenges. In *2016 IEEE Smart Energy Grid Engineering (SEGE)*, 381–385.

Sharwood, S. (2017). Aws adopts home-brewed kvm as new hypervisor. Accessed May 16, 2019.

TIOBE (2019). The python programming language. Accessed Jun 6, 2019.

Van Jacobson, B. F. (2006). *Speeding up networking*. Accessed May 3, 2019.

VMWare (2018). *Building and Deploying Single Containers to a Virtual Container Host*. Accessed June 2, 2019.

Vresk, T. & Čavrak, I. (2016). Architecture of an interoperable iot platform based on microservices. In *Proc. Electronics and Microelectronics (MIPRO) 2016 39th Int. Convention Information and Communication Technology*, 1196–1201.

Walla, A.-A. (2017). Live updating in unikernels. Master's thesis, University of Oslo.

Wanigasekara, N. (2015). A semi lazy bandit approach for intelligent service discovery in iot applications. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers*, 503–508.

Xavier, B., Ferreto, T., & Jersak, L. (2016). Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *Proc. Cloud and Grid Computing (CCGrid) 2016 16th IEEE/ACM Int. Symp. Cluster*, 277–280.