



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Supporting Unikernel-based Microservices at the Edge

Patrick Sabanic





DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Supporting Unikernel-based Microservices at the Edge

Unterstützung von Unikernel-basierten Microservices auf Edge-Computing-Knoten

Author: Patrick Sabanic
Supervisor: Prof. Dr.-Ing. Jörg Ott
Advisor: Dr. Nitinder Mohan, Giovanni Bartolomeo
Submission Date: 15.12.2022



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.12.2022

Patrick Sabanic

Acknowledgments

I want to thank the Chair of Connected Mobility that offered me the possibility to undertake this thesis work. In particular, I want to thank my supervisors and advisors, Prof. Dr.-Ing. Jörg Ott, Dr. Nitinder Mohan, and Giovanni Bartolomeo, for the tremendous support they gave me during the past months. I also want to thank my friends and colleagues who supported and helped me in the past. Finally, I would like to thank my parents, who always gave me their support.

Abstract

With interest in Unikernels for different use cases growing in many deployment environments, like cloud and edge, a multitude of challenges in this arrives. With Unikernels presenting a different kind of deployment, they, nonetheless, should be able to be used within the same platform and interact with the already existing Container based services. In an edge environment, the nodes are most of the time heterogeneous regarding their capabilities, architecture and network connectivity. However, such an edge environment has the benefit of a shorter distance, thus lower latency to the end users of services scheduling on the edge. This thesis presents a way of scheduling Unikernels on a heterogeneous system. This includes awareness of Unikernels throughout the system and further introduces a runtime environment for Unikernel deployment and explores Unikernels with the help of said runtime. Furthermore, this includes the ability to communicate with services of different runtimes, such as the Container runtime. The implementation of this runtime sees an increase in deployment time in some instances compared to Containers but, at the same time, can improve the runtime behavior of some services running as a Unikernel.

Kurzfassung

Mit dem wachsenden Interesse an Unikernels für verschiedene Anwendungsfälle in vielen Einsatzumgebungen wie Cloud- und Edge-Umgebungen entstehen eine Vielzahl von Herausforderungen für diesen Anwendungsfall. Obwohl Unikernels eine andere Art des Deployments darstellen, sollten sie dennoch in der Lage sein, innerhalb der gleichen Plattform verwendet zu werden und mit den bereits bestehenden Container-basierten Diensten zu interagieren. In einer Edge-Umgebung sind die Knoten meist heterogen in Anbetracht auf ihre Rechenleistung, Systemarchitektur und Netzwerkkonnektivität. Eine solche Edge-Umgebung hat jedoch unter anderem den Vorteil einer kürzeren Entfernung und damit einer geringeren Latenz zu den Endnutzern von Diensten, die auf Edge-Knoten ausgeführt werden zu haben. In dieser Arbeit wird ein Verfahren für das Scheduling von Unikernels auf einem heterogenen Cluster vorgestellt. Dies beinhaltet das Wissen über Unikernels im gesamten System und stellt im weiteren eine Laufzeitumgebung für den Unikernel Einsatz vor und untersucht die Unikernels mithilfe der besagten Laufzeitumgebung. Darüber hinaus umfasst dies die Fähigkeit, mit Diensten verschiedener Laufzeitumgebungen zu kommunizieren, wie zum Beispiel der Container-Laufzeitumgebung. Die Implementierung dieser Laufzeitumgebung zeigt in einigen Fällen eine Erhöhung der Deployment-Zeit im Vergleich zu Containern auf, kann aber in anderen Fällen, wenn das Laufzeitverhalten der Dienste betrachtet wird, eine Verbesserung dieser in einigen Diensten feststellen, wenn sie als Unikernel ausgeführt werden.

Contents

Acknowledgments	iii
Abstract	iv
Kurzfassung	v
1. Introduction	1
1.1. Problem Statement	1
1.2. Research Questions	2
1.3. Contribution	2
1.4. Thesis Structure	2
2. Background	4
2.1. Virtualization	4
2.1.1. OS-level Virtualization	4
2.1.2. Hardware Virtualization	6
2.1.3. QEMU	6
2.2. Unikernels	8
2.2.1. Language specific Unikernel	9
2.2.2. Non-Language specific Unikernel	9
2.2.3. Unikraft	9
2.3. Orchestration Platforms	12
2.3.1. Kubernetes	12
2.3.2. Oakestra	13
3. Unikernel Support	18
3.1. Unikernels on the Edge	18
3.1.1. Limitations	19
3.2. Unikernel Application Porting	19
3.2.1. General Process	19
3.2.2. Simple Applications	20
3.3. Porting of Containerized Services to Unikraft	22
3.3.1. Video Aggregation	23
3.3.2. Porting Process	23
4. Oakestra Integration	28
4.1. Requirements	28

4.2. Oakestra	30
4.2.1. Worker Node	31
4.2.2. NetManager	37
4.2.3. Cluster Orchestrator	39
4.2.4. Root Orchestrator	40
4.2.5. Deployment Descriptor	44
5. Evaluation	47
5.1. Test Setup	47
5.2. Evaluation Results	47
5.2.1. Unikernel Measuring	47
5.2.2. Aggregation Comparison	48
5.2.3. Resource Utilization	49
5.2.4. Throughput	51
5.2.5. File Size	54
5.2.6. Deployment	54
6. Conclusion	56
6.1. Conclusion	56
6.2. Limitations and Future Work	56
A. Deployment	58
B. Additional Measurements	68
List of Figures	74
List of Tables	76
Bibliography	77

1. Introduction

1.1. Problem Statement

The decentralized nature of edge computing moves the execution closer to the recipient and allows the deployed services to work together to deliver a result back to the end user more quickly. Edge networks can consist of a far more heterogeneous hardware base than typical cloud computing clusters and the deployment of services on the edge is an active research topic. The Nodes can be vastly different regarding computational power and network connectivity. With services being deployed on edge Nodes, the first-mile latency can be reduced with a thorough service placement relative to the clients [1]. Further, multiple improvements can be made to enhance the overall deployment of service in an edge environment. One of the is energy improvement when deploying services [2].

While edge clusters have many advantages, there are also different challenges to overcome when it comes to the layout of them. This manifests itself with the Nodes in an edge cluster. There they can be vastly different in terms of performance. Further, the network capabilities might be drastically different between them. The architecture of such Nodes might also differ significantly, with the most prevalent being AMD64 and AArch64 in today's landscape. While those architectures are the most prevalent, this does not exclude the possibility of others being used, e.g., RISC-V or x86, without 64-bit comparability. These differences on the edge increase the complexity when it comes to orchestration and management tasks for the cluster.

With the emergence of Unikernel frameworks, e.g., Unikraft [3], the interest in deploying Unikernels has increased [4]. Unikernels offer a different approach to containers, with them being self-contained operation systems running on the Nodes. Unikernels, with their minimal design, show increased performance in many workloads [5, 6]. With this increased performance while running on the same hardware, they are a matching fit for edge Nodes. This is due to the edge Node's hardware constraints which might be better utilized with a Unikernel instead of a Container based deployment. While Unikernels might be able to increase the performance of some services, not all might benefit, which is why this needs to be looked at in more detail to gain additional insight into how to best schedule Unikernels on edge clusters and also other clusters in general.

However, Unikernels add further problems when regarding the Node's heterogeneity. Unikernels, like all Virtual Machines, rely on the host's feature set to be executed efficiently. These feature sets are not required for containerized deployment, which leads to further constraints on the management aspect of the cluster when it comes to Unikernels. The cluster must know about certain fundamentals for both the services which are to be scheduled as well as the Nodes. These fundamentals define if a Node is capable of running Unikernels.

1.2. Research Questions

R1 Design of a extensible Unikernel runtime for the edge

Most orchestration platforms are heavily focused on containerized services. Therefore, to support Unikernel based deployment, extensions have to be made that further allow both runtimes to interoperate. Thus, this thesis proposes a Unikernel runtime based on the orchestration platform Oakestra, detailed in subsection 2.3.2.

R2 Strategy for the usage of Unikernels

With the extension of orchestration platforms to support Unikernels, the question arises for what kind of service Unikernels might be beneficial. While it has already been shown that Unikernels can increase the performance of a service [5, 6], a more focused consideration has to be made for the usage of them on the very different structure of edge environments.

R3 Scheduling strategy for Unikernels on edge clusters

The differences between containerized services and Unikernels might benefit from different kinds of host systems. This is why a better-suited strategy to distribute Unikernels onto Nodes within a cluster might be required to achieve the best results.

1.3. Contribution

This thesis proposes an extension to the Oakestra framework, described in Figure 2.10, to enable the scheduling of Unikernels to Nodes within the edge clusters. Further, it proposes a way to execute the Unikernels on the Nodes as part of Oakestra and introduces a test service to compare the behavior of Unikernels compared to Containers.

The Unikernel runtime uses QEMU [7] to enable the deployment of Unikernels on the edge Nodes within Oakestra. However, with the optional support of Unikernel for the Nodes and the support for network communication between Container and Unikernel services, the complexity of the scheduling of services is increased. While this is the case, the changes are transparent to the end user. From the perspective of Container deployment, nothing has changed, making the additional configuration only necessary in the case of Unikernel deployment.

With the ability to deploy Unikernels, the question of the usefulness of their deployment arises. Because of this, parts of a multi-service application have been transitioned to a Unikernel platform, enabling the direct comparison between both runtimes. With this, it was possible to evaluate the Unikernel runtime proposed in this thesis and to evaluate what services could benefit from being a Unikernel

1.4. Thesis Structure

The thesis introduces the different virtualization technologies and their most common implementations in chapter 2. In this chapter, Unikernels in general and Unikraft, in particular, are

explored, and further, it goes over possible orchestration platforms, including a high-level view of Oakestra's architecture. In chapter 3, Unikernels are further explored with a focus on the Unikraft [3] framework. The chapter goes over the development process for a Unikernel environment. Also, it introduces the service, which was ported to Unikraft, to in later chapters compare its behavior and performance with that of a Container service. In chapter 4, a list of requirements for a Unikernel extension in Oakestra is explored. Further, the changes to Oakestra required to enable Unikernel support and how they affect the rest of the Oakestra system are laid out in detail. With chapter 5, the proposed changes are compared in a multitude of ways indicating both benefits and downsides to the deployment of Unikernels. The thesis concludes with a description of the advantages and downsides of the deployment of Unikernels in general and possible future work to overcome those downsides.

2. Background

2.1. Virtualization

There are multiple distinct kinds of virtualization both having their benefits and downsides. Figure 2.1 shows the general difference between OS-level Virtualization used by container based applications and virtualization used by Unikernels.

2.1.1. OS-level Virtualization

OS-level virtualization is a form of isolation for which no direct hardware support is required and only the OS needs to implement necessary support. The virtual machines on the level of virtualization are called containers and examples of such a kind virtualization are Docker and LXC [8, 9]. These containers are isolated user-space instances which run with the same kernel as the host system and are managed by a container engine. The features that allow modern container engines to function are cgroups and namespaces [10, 11]. Cgroups, or Control groups, are a feature of the Linux kernel that allows monitoring and restricting the usage of system resources. The cgroup feature is provided to the system through a pseudo filesystem by the kernel. The cgroups are structured in a hierarchy made up of different subsystems. Subsystems control the resources through cgroups, which enables the control of different resources, including CPU time and memory limits. A single cgroup represents a collection of processes bound to a set of limits defined by this filesystem. While cgroups are used to limit resources and access to them, namespaces are responsible for directly separating processes from the host system. Each container runs in its own namespace and has its own set of resources, which is only visible for processes running within the same namespace. There

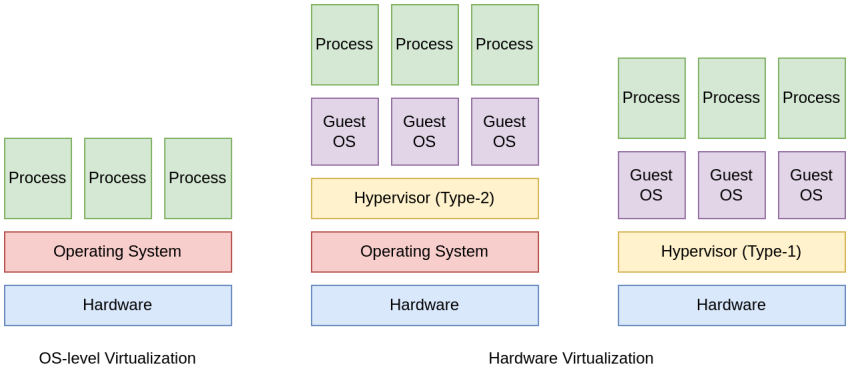


Figure 2.1.: Difference between Virtualization Design

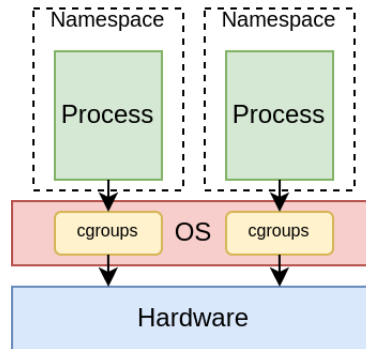


Figure 2.2.: Cgroups and Namespaces

are different kinds of namespaces, including mount and network namespaces separating mount points and networking devices.

In terms of networking, Containers can have different ways to communicate over the network. One way is to have the container directly be part of the same network namespace as the host, which gives the container access to the entire network, which the host system would have access to. Another way is for the container to use its own network namespace. This separates the host and container network and adds better isolation. The container can also be given access via veth to a bridge on the global network namespace. Further, there are different overlay network layers to enable communication between multiple containers running on multiple hosts.

The most common implementation, with a large user base, are OCI [12] based containers. Many different implementations use the specification provided by it, such as Docker and podman [8, 13]. This enables multiple different container engines to be interoperable. Thus, container images from one platform can be used and run on other platforms using the same standard. The OCI specifies three main elements of containers. Those are the Runtime Specification, the Image Specification and the Distribution Specification [14, 15, 16]. The Runtime Specification, describes how a container should be configured on a system, how the configuration of such a container should be formatted, how the execution environment should be set up, and what the container lifecycle should be. The Image Specification describes how container images are laid out and what information they contain. Container images consist of layers, and each modification adds a layer on top of the layers within an image. It further includes information about what host system the container is made for. In cases where the container is not made for the host system, emulation or virtualization might be needed. Moreover, it also contains information on how the image should be run. The last specification, the Distribution Specification, describes how the containers can be distributed. It specifies how container registries and clients should communicate and in which way the container images are transferred. This enables different container engines to access, for example, Docker Hub [17].

Because of the lightweight virtualization layer, which isolates the container from the rest of the host system and only has minimal impact on the application's performance, its simple

usage without the need for special hardware support and the standardization in OCI which enables multiple different implementations to share the same container images, have container become widespread from cloud computing to local single instance deployment.

2.1.2. Hardware Virtualization

Unlike virtualization on the OS level, hardware virtualization has additional layers, namely the Hypervisor and the guest OS running within the virtual machine. The hypervisor is what isolates the resources of the host OS and the virtual machines and manages resources between them [18]. There are two types of hypervisors. Type-1 which are running directly on the hardware, and as such, no additional Operating System is in between the hypervisor and hardware. On the other hand, Type-2 which are hypervisor running on top of an already existing Operating System. On Linux, the most widespread hypervisor is KVM [19] as part of the Linux kernel. Transforming the Linux host instance into the hypervisor, thus being a Type-1 hypervisor. It enables the use of hardware-assisted virtualization, using hardware features like Intel VT [20] on x86, which accelerates the execution of virtual machines to near-native performance.

There are many different tools compatible with KVM, including Firecracker [21], Solo5 [22] and QEMU [7]. These tools enable the usage of KVM and implement the necessary userspace implementation for the paravirtualization of devices for the virtual machines. These devices are provided by virtio [23] and include device drivers for networking, block and balloon devices depending on the implementation and the support of the guest operating system. Because of the increased isolation virtual machines provide, microVMs emerged. MicroVMs are a type of virtual machine introduced by Firecracker. They consist of a minimal platform, only including the necessary components to run the guest system and remove all unnecessary device support. These removed feature sets include elements such as the lack of PCI support for the guests. This mode of operation is supported by Linux and some Unikernels [21]. Besides Firecracker's microVM, QEMU has also introduced a similar machine type to Firecracker's microVM [24]. While these machines can decrease boot time [3], they also come with the downside of lower compatibility for guest systems.

2.1.3. QEMU

QEMU is a general purpose emulator and virtualization platform for Linux. QEMU is capable of emulating various different hardware architectures in both user space emulation and system emulation via KVM. User space emulation is the process of running binaries for different architectures directly on a target host system. This involves emulation on the part of QEMU. System emulation is the process of QEMU emulating not only a single binary but instead emulating an entire system with all required and additional hardware. This allows the use of an OS within the emulated environment. Both of these modes rely on TCG [25] in modern QEMU versions. TCG is a JIT-capable code generator capable of translating the target architecture, which is the architecture QEMU emulates, to the host's architecture, where it can then be executed. While this enables the execution of binaries on incompatible architectures,

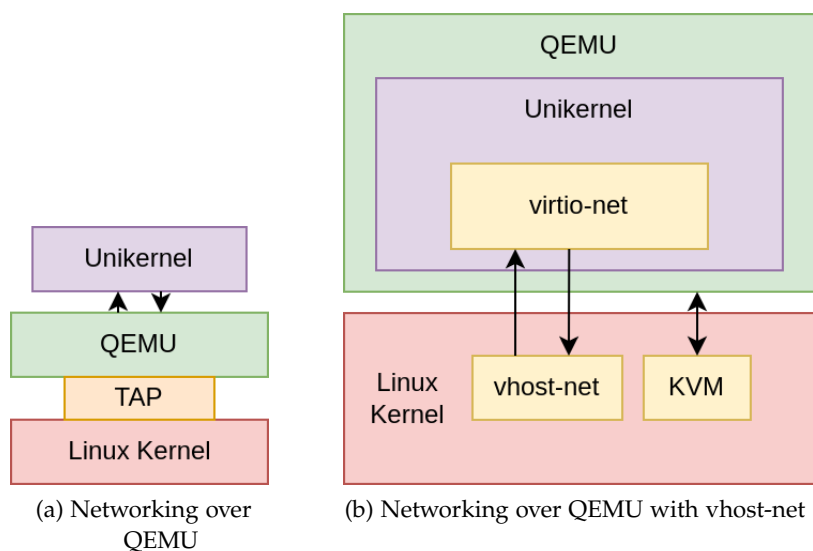


Figure 2.3.: Virtual Machine Networking

TCG still introduces a processing overhead. With the addition of KVM in the Linux kernel, QEMU gained additional support for KVM. With KVM enabled, the guest system can run code directly on the hardware without translation steps involved. This significantly increases the performance of the guest system. While the system is running directly on hardware via KVM, QEMU is still responsible for emulating the required hardware. This hardware can be completely emulated or paravirtualized depending on guest compatibility and requirements. If the hardware is to support paravirtualization, the guest needs to implement the support. One of these paravirtualized devices, defined by virtio and implemented by QEMU, is a NIC called virtio-net in QEMU. This type of NIC, like every paravirtualized device, removes the need to emulate actual hardware and increases performance in this way. In general, the networking in QEMU is done with a TAP device that QEMU is using to transfer the traffic of the guest system, which is, in this thesis, a Unikernel in most cases, to the host networking stack. A visualization of this is depicted in 2.3a. The overall networking does not change even for fully virtualized NIC, but the overhead in the required emulation of the devices is reduced, and thus throughput increases with virtio.

Furthermore, there is the possibility of a further change in the working that can be made to increase the throughput further and lessen the CPU load with QEMU. For this, the vhost-net kernel module is required. Vhost-net allows the circumvention of QEMU in the network communication. The interactions on the data plane are depicted in Figure 2.3b. While the control plane for the NIC is still handled by QEMU, where it is responsible for the communication of virtio and the hypervisor, the data plane is able to communicate directly with vhost-net. This removes QEMU from the exchange of packets and reduces latency and throughput.

2.2. Unikernels

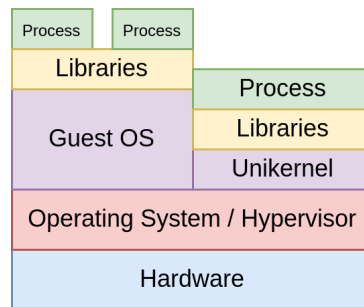


Figure 2.4.: Virtual Machines and Unikernels

Unikernels are single-purpose build kernel images based on the principles of library operating systems, with a single address space limiting the execution to a single process. They, unlike traditional virtual machines, limit the capability of the guest OS to a minimum, removing all unnecessary components which are not needed to run a required application. This can include network or threading support or even go as far as removing the standard libraries of the language the Unikernel is written in. The virtualization aspect of the Unikernel also further isolates it from the host system. Unikernels do not use the traditional concepts of operating systems like the separation of user and kernel space. Thus, a Unikernel application runs in a single address space and has unrestricted access to the resources within the virtual environment. This leads to increased performance [5, 6].

But there are also downsides with only one address space. It is not possible, as already mentioned, to execute multiple processes within the same Unikernel since the address space could not be mapped to a different virtual address space for each application because the Unikernel does not support execution outside of the kernel memory space. This address space division is depicted in Figure 2.5. In a more general operating system with a monolithic kernel like Linux, the kernel space is accessed to do I/O operations and other system functions. This includes allocating memory or opening a network socket. Most of the system calls done in a program are wrapped in a library function which then executes this system call, which in turn causes a context switch from user to kernel space and is costly compared to a normal function call. This is where the Unikernel operation differs. Instead of requiring a context switch, the process is already in kernel space and the library functions do not need to do system calls. Instead, they call a function within the same address space as the application is running in.

Further, this kind of miniaturization leads also to small file sizes compared to a traditional virtual machine and even container images [26] and other possible advantages depending on the Unikernel implementation. Further, Unikernel can have a lower initialization time than traditional virtual machines and container based applications [27]. These benefits can make Unikernels a viable alternative to container based applications. On the other hand Unikernels can have security vulnerabilities due to the toolchains being different and each Unikernel

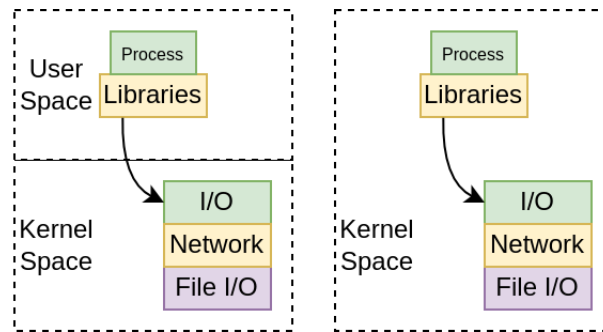


Figure 2.5.: Address Space

implementations using different approaches [28].

2.2.1. Language specific Unikernel

There are many different Unikernels built with different goals in mind. One of them being *Mirage OS* [29], which is in active development. *Mirage OS* is a Unikernel designed for high-performance network applications and their applications and libraries are written in OCaml. This limits the applications which can be ported without significant changes. Other language specific Unikernels include LING [30], HaLVM [31], includeOS [32] and runtime.js [33].

2.2.2. Non-Language specific Unikernel

On the other hand, there are also Unikernel projects which are designed to be usable with a multitude of languages like *Unikraft* and *OSv* [34]. *Unikraft* describes itself as a "Unikernel Development Kit" [35].

2.2.3. Unikraft

The fundamental aspects of the *Unikraft* design are the following. A single address space which means a single application in a Unikernel and no ability to create further processes within it.

A fully modular system which means that all components of the system are exchangeable. This goes down to the very core of the system and includes everything from drives to system functionality, e.g., scheduling.

A single protection level which goes hand in hand with the single address space. There is no separation between the user and kernel space to avoid costly context switches.

POSIX support which increases compatibility with already existing applications while also allowing for specialization under the different POSIX APIs.

Platform abstractions which enable the Unikernel to support multiple different hypervisor and hardware architectures seamlessly.

With these requirements, Unikraft started with a primarily original code base, with a few exceptions [36].

The Unikraft kernel consists of so-called micro-libraries, which make up most of the Operating System. Such libraries have minimal dependencies and can be rather small in size. Libraries for the same purpose implement the same API and can be interchanged, e.g. memory allocators [3]. The modularity of Unikraft can be seen in Figure 2.6. The Figure shows a simplified version of the Unikraft dependencies in a typical kernel build [37]. On the top layer is the actual application code which is dependent on multiple underlying components. In the case of a C application, it directly uses one of the C library implementation in Unikraft. Unikraft support multiple different implementations. Those implementations are *nolibc*, which is a minimal c library used for small applications, *newlib*, which is an embedded C library, and *musl*, which is a fully compliant c library.

These libraries, in turn, can depend on internal ones. These internal libraries in the next layer represent the POSIX compatibility within Unikraft. In the example, the application uses the BSD socket API provided by the *posix-socket* library. The library itself is only thin a layer enabling the use of underlying functionality. Further, libraries on this layer can include threading support and other POSIX related functionality. Since Unikraft is designed as a modular library operation system, there is no need to include all aspects of the POSIX functionality for all kernel builds if there is no element requiring the library.

The OS layer consists of Unikraft's internal micro-libraries. These libraries implement an API layer which can then be used for the underlying system to implement the features or use already existing solutions and fit them to the APIs. This is, for example, the case with the allocator under the *ukalloc* library. Each library implementing an allocator is interchangeable but might have different requirements. The default allocator, built into the kernel, is a buddy-based allocator, which might not be the optimal allocator in all cases. Because of these different possible requirements there are a multitude of allocators available, including a port of *mimalloc* [38]. Another exchangeable element is the scheduler. The default scheduler in Unikraft is non-preemptive, implementing a round-robin scheduling cycle. A non-preemptive, or cooperative, scheduler is based on the threads giving up control of the execution for the scheduling decision to happen. This means, unlike a preemptive scheduling system, the programmer needs to be aware of other threads in the execution and give up control regularly. Otherwise, some threads might starve. While preemptive scheduling can be useful for specifically developed applications, it might hinder porting applications with preemptive scheduling as a prerequisite. Thus it might help to change it to the desired scheduling once they are added as libraries.

While kernel micro-libraries provide an API for the underlying implementations, there are also fundamental function which are not as replaceable. However, instead, the different libraries use them to provide the functionality. Those include the block device interface *ukblockdev* which, provides fundamentals that can be used to implement different filesystems on top of it. Those filesystems include 9P [39] provided as a device via QEMU and a ram disk filesystem loaded at startup into memory. Further, this includes the network API. The default networking library is, as depicted, the *lwip* [40] library, which is a standalone embedded socket

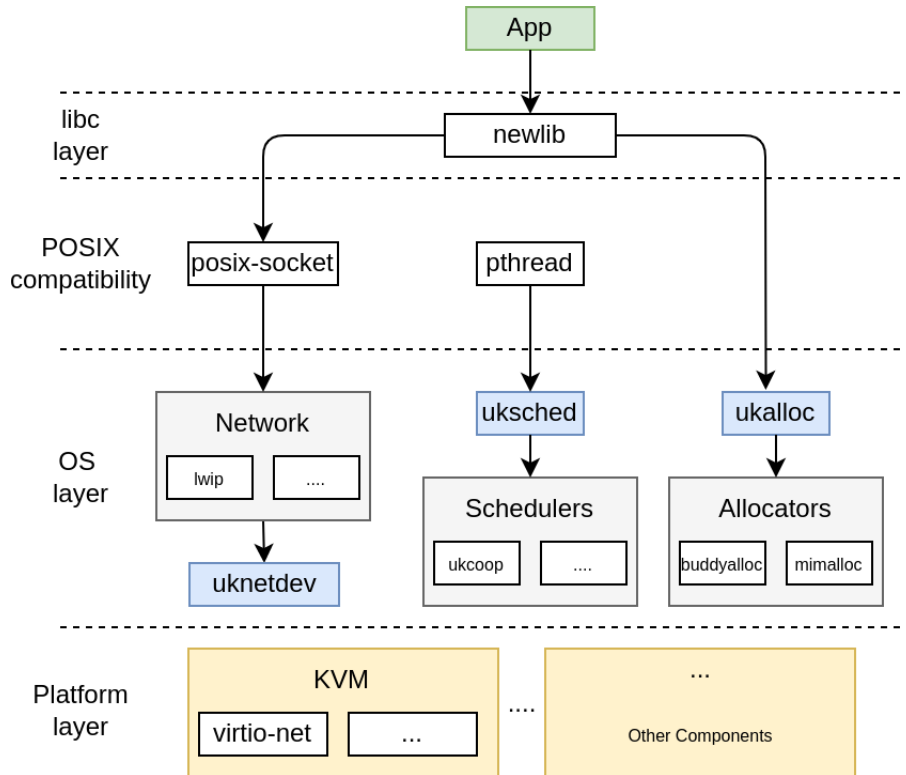


Figure 2.6.: Unikraft Architecture

implementation. It was modified to use the Unikraft library *uknetdev* to enable networking for general applications within the Unikernel. The Unikraft library can also be used directly to achieve higher performance by removing the abstraction done by *lwip*.

The lowest layer of the Unikraft dependencies is the platform layer. This layer is responsible for compatibility with the target platform, virtualized or bare metal. Unikraft supports multiple targets, including KVM, Xen, Linux userspace and bare metal [19, 41]. The most relevant version for this thesis is KVM since it will be used for the deployment of Unikernels and, unlike Xen, does not require to have a special system setup. The layer implements the drivers for the environment, in which the Unikernel is executed. This includes drivers to support virtio devices, such as *virtio-net*, which enables network support.

Further, Unikraft includes an additional abstraction layer to enable the execution of Linux binaries. For this to work, a system call layer called *syscall-shim* has been introduced. This layer is responsible for catching system calls in Unikraft. This is important because of the nature of system calls, which would, in a regular system, cause a context switch that would not work for a Unikernel.

This makes Unikraft highly configurable and extendable. Although currently Unikraft only supports `x86_64` and ARM architectures, RISC-V support is in development.

2.3. Orchestration Platforms

For the use of any kind of application in the cloud or on the edge, an orchestration platform is required to enable scheduling and networking between components. These platforms abstract the hardware away from the user, only presenting necessary information and enabling the automatic scaling and migration of services based on pre-agreed terms of an SLA.

2.3.1. Kubernetes

Kubernetes is an open-source system that can be used to automate the deployment, scaling and management of containerized applications [42]. Kubernetes is generally run in a cloud configuration. Such a configuration consists of at least one worker node and the control plane, which can also consist of multiple nodes. The control plane makes global decisions and manages the worker nodes and the Pods. Pods are the components of an application in Kubernetes and are the smallest unit of work. The control plane is split into multiple components. These components include a key value store, a scheduler, and multiple controllers responsible for different parts of the system. The functionality of the Kubernetes control plane can be extended further by the use of addons. The node components include kubelet, the agent running on each node and orchestrating the containers for each Pod, a network proxy and a container runtime.

The deployment can be initialized via the API server. This results in the generation of a new Pod, which will be scheduled to a Node within the cluster based on set scheduling configurations. These can include Node affinities and taints. Node affinity effect to which node a Pod is being scheduled, while taints are set to nodes and prevent the scheduling of Pods to them [43]. While Kubernetes has traditionally been used for container deployment but can also be used for VM and Unikernel deployment.

Further, networking in Kubernetes is done in multiple ways. One of them is Pod to Service communication. A service is an abstraction of multiple Pods in a single service. Such a Service enables other Pods to address the Pods within the Service via the Cluster IP. The Cluster IP uniquely identifies the Service and the Pods within it. Whenever the Cluster IP is used, Kubernetes takes care of the necessary load balancing, which can use different algorithms like round-robin or consistent hash and network requirements. A service can have multiple configurations and be made publicly available as well. While Pods can be indirectly addressed via the Service, each Pod still has its own address [44].

Another way of communication is the Pod to Pod communication. This kind of communication is realized in Kubernetes by the CNI plugins. These plugins are responsible for enabling the exchange between Pods on a local machine or on different machines via, e.g., bridged and tunneled networks [45].

Kubernetes does not stand alone and there are a multitude of distribution compatible with it and extending it in some way or another. One of these distributions is K3s [46]. K3s describes itself as a lightweight Kubernetes. It packages most of the functionality of Kubernetes in a single binary which can be used for deployment. The project includes many of the dependencies and plugins of Kubernetes, including the container runtime and the CNI

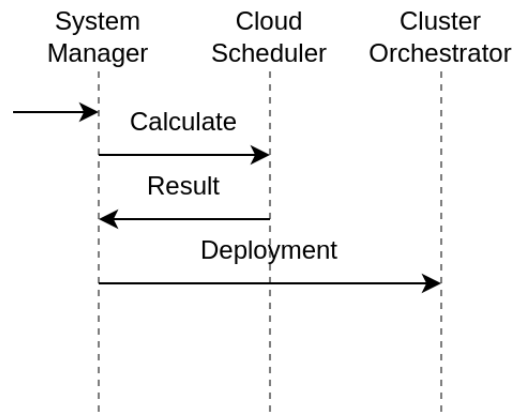


Figure 2.7.: Root Scheduling and Forwarding to Cluster

plugin for Pod to Pod communication. A k3s cluster consists of a k3s server node acting as the control plane the same way it does in Kubernetes and one or multiple k3s agents, representing the Nodes responsible for executing the Pods. With the exception of multiple elements already being integrated, the functions of k3s are very similar to Kubernetes.

2.3.2. Oakestra

Oakestra [47] is a lightweight orchestration framework. It is structured hierarchically in a three layer design, capable of deployment on a cluster of heterogeneous resources. Worker Nodes are able to receive container based applications. Oakestra is composed of the Root Orchestrator at the highest level, followed by one or more Cluster Orchestrators, and on the lowest level, the Worker Nodes. All Worker Nodes are registered with exactly one Cluster Orchestrator and are the ones onto which the workload is distributed. The structure of the hierarchy and the communication between the components is depicted in Figure 2.10.

While Oakestra’s design allows for the possible scheduling of services that are not container-based, this is not currently the case. This thesis extends Oakestra’s framework to support Unikernels, which under some circumstances, can lead to improved performance and offer better isolation compared to container-based deployment. The following describes the components as they were before the proposed Unikernel runtime changes were added.

Root Orchestrator

The Root Orchestrator is the component at the top of the hierarchy. It is responsible for scheduling services for the clusters, and enabling user interaction, taking deployment requests and new services via a Restful API. It is comprised of the System Manager and Cloud Scheduler.

Figure 2.7, shows a successful deployment request from the perspective of the Root Orchestrator. The deployment is initialized via the Restful API. The Root Orchestrator consists of two distinct components the System Manager and the Cloud Scheduler. The System

Manager is responsible for the overall working of the system. This includes communicating with the clusters, taking in external API requests and keeping track of the cluster resources.

The Root Orchestrator considers all clusters as pools of resources without knowledge about the individual Nodes. Those resource pools are the aggregation that the Cluster Orchestrators collect and send, in aggregated form, to the Root Orchestrator.

The deployment of a microservice starts with a request, e.g., from a user wanting to deploy said microservice, which starts a scheduling process to find a fitting cluster and forwards it to the Cluster Orchestrator of that cluster.

The process of scheduling a microservice is described in the following. After the Root Orchestrator receives a request for a new microservice instance, the scheduling process is initialized. The new service instance is registered in the database of the root and the scheduling request is sent out to the Cloud Scheduler. The Cloud Scheduler receives the request via HTTP and proceeds to process it. The first element to be checked is if there are any constraints to be considered. The constraints are not important for the deployment of Unikernels and are not further discussed here. Regardless of whether constraints are applied, the scheduling process proceeds with a list of clusters, which either are all available clusters or only the clusters that match the constraints. The scheduling process goes over all clusters and filters out the ones which do not have the required hardware resources available. After all the Nodes have been traversed, the best match is chosen as the target cluster. The best match, in this case, is the cluster with the most resources available. This matching is the default scheduling behavior which can be changed via the selection of a different scheduler. The cluster ID and the job the scheduling process corresponds to are then sent back to the System Manager via HTTP. The System Manager then sets the status of the service to `CLUSTER_SCHEDULED` and informs the network component at the root level. The last step in the scheduling process from the perspective of the Root Orchestrator is to forward the deployment request to the cluster specified in the scheduling result.

At the cluster level, a new scheduling decision is being made to select a Node with the requested requirements and then scheduled to the selected Node.

The deployment of a new service is done via deployment descriptors in JSON format, detailing the general service and the deployment information of the microservices it consists of, with an example later given in subsection 4.2.5. These microservices can each individually be given constraints and hardware requirements.

Cluster Orchestrator

The Cluster Manager is responsible for the aggregation of Node resources and scheduling services on the Nodes which are registered with it. It is comprised of the Cluster Manager and Cluster Scheduler. Figure 2.8 depicts a successful scheduling process from the Cluster Orchestrator's point of view.

In the same way, the Root Orchestrators are required to schedule the microservice so is the Cluster Orchestrator. Once the Cluster Orchestrator receives a request to deploy a service via HTTP from the Root Orchestrator, the cluster proceeds to register the new service to be scheduled in the cluster-level database. After this, the Cluster Manager sends a scheduling

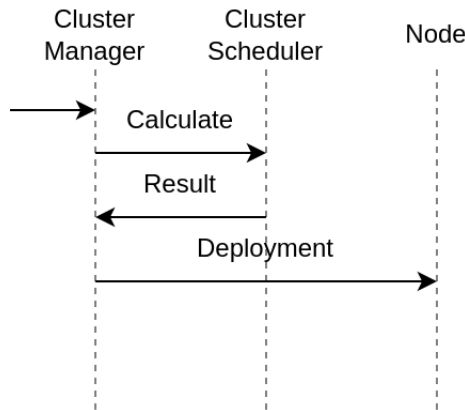


Figure 2.8.: Cluster Scheduling and Deployment

request to the Cluster Scheduler via HTTP. From here on, the Cluster Scheduler starts the scheduling process. This, in the first step, involves checking the microservice’s constraints. If no constraints were given, the scheduling goes over all Nodes. Otherwise, only the Nodes matching the constraints are selected for further consideration. The next step of the scheduling process is the examination of each Node, considering their available hardware, such as CPU load, free memory and other requirements. From all Nodes that match the service’s requirement, the best match is selected and sent back to Cluster Manger via another HTTP POST request. This POST contains the selected Node and service information in JSON format. At this point, the scheduling process is completed and the status of the service instance is set to scheduled at the Node level (NODE_SCHEDULED). Further, the network component is notified about the instance of the service. The last step of the Cluster Manager in this process is to notify the Node about the new deployment via MQTT.

Worker Nodes

The Worker Nodes are the lowest in the hierarchy of Oakestra and are responsible for executing the microservices. The information is transmitted to the Nodes via MQTT and each Node is part of one and only one of the clusters. In addition, the Nodes regularly send status updates about resource consumption and available resources to their Cluster Manager. These updates are used for further aggregation for the root level and scheduling within the cluster. In the case a Node receives a deployment request from the Cluster Manager, shown in Figure 2.9, the Node selects the matching runtime specified and proceeds to run the service. The process is highly dependent on the selected runtime, and in the case of Unikernels, the newly proposed runtime in subsection 4.2.1 is used. Further, in case the Node is registered with its local NetManager, the network overlay for inter-microservice communication is enabled. For this, the Node registers each microservices before it starts with the NetManager, which handles the communication if required.

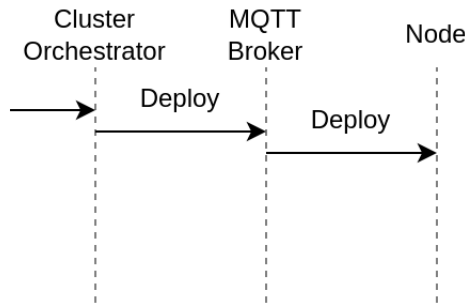


Figure 2.9.: Node Deployment

Networking

The network overlay [48] in Oakestra enables the communication between services on all Nodes within not only the same Node or cluster but the entire set of Nodes available in all clusters managed by the Root Orchestrator. This is achieved by adding components at each hierarchy level. From top to bottom, these additions are the Root Service Manager, the Cluster Service Manager and the NetManager, which can be seen in Figure 2.10.

At the root and cluster level, the Service Managers are responsible for managing the network and enabling routing between Nodes, with the Root Service Manger keeping the overview of the entire system while the Cluster Service Manager manages the networking of the services on the cluster level.

The NetManager is responsible for the network communication of the individual services as part of the Worker Node. Each Node is running its own instance of the NetManager. It is tasked with communicating with the Cluster Service Manager and configuring the network for each container deployed on the Node. The network setup and the communication with the Service Managers enables the overlay network to route traffic between microservices.

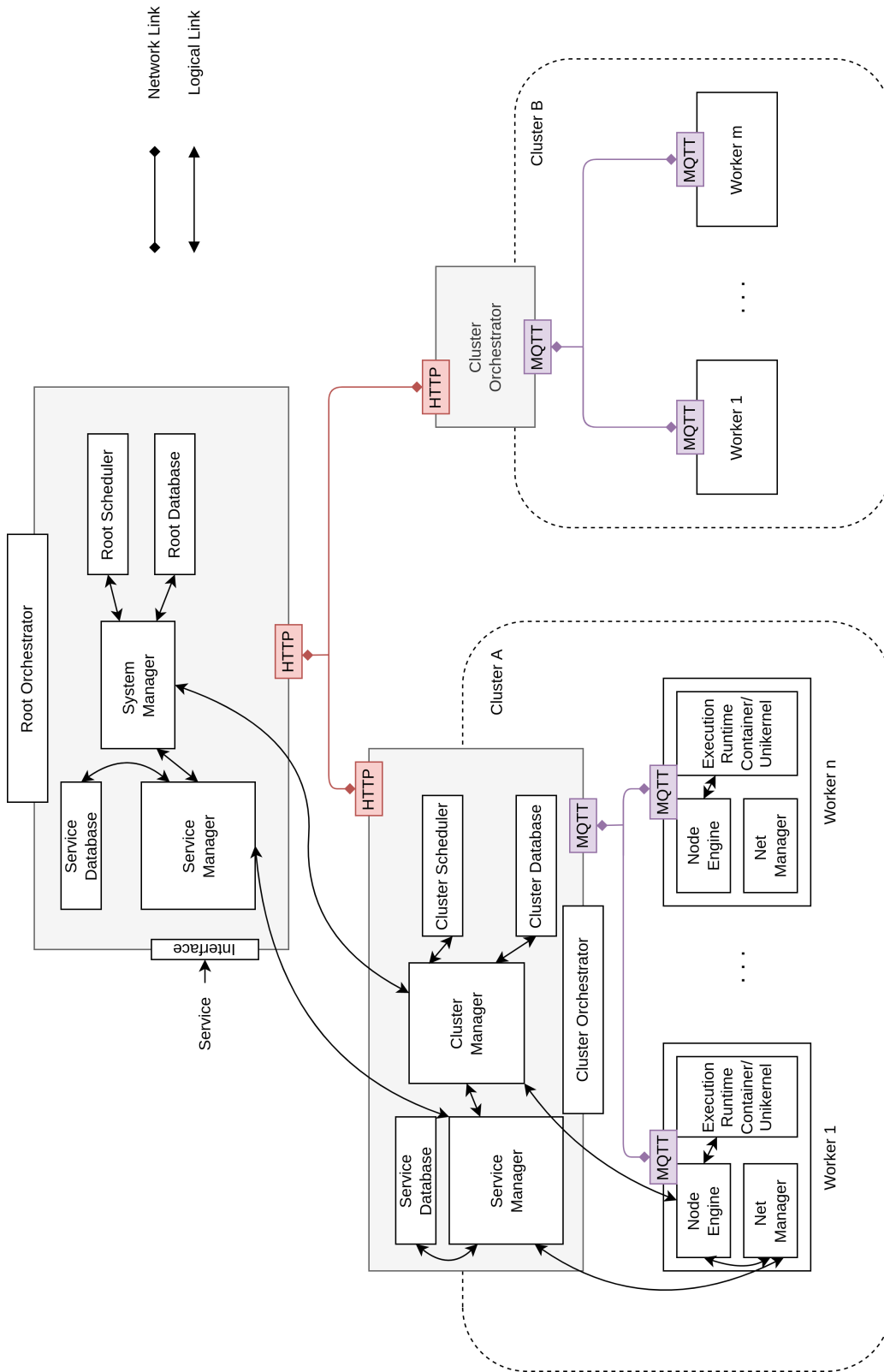


Figure 2.10.: Oakestra Architecture

3. Unikernel Support

This chapter goes over the porting process of a generic application to a Unikernel and further extends this to the example of a more complex application used in further testing at a later part, chapter 5, to evaluate the use of Unikernels for edge deployment.

3.1. Unikernels on the Edge

Unikernel deployment in an edge environment is a challenging subject due to the high heterogeneity of edge Nodes and networks. Many different apposes to the deployment of Unikernels exist [49, 50]. The Nodes in an edge cluster do not need to be of the same architecture, nor do they need to be of similar hardware capability in terms of performance. This leads to a multitude of factors that need to be considered regarding Unikernel deployment.

The first factor to consider is the scheduling of Unikernels onto Nodes of an edge cluster. At each stage of scheduling, the system needs to be aware of the capability of each Node within the cluster to schedule a Unikernel properly. These capabilities include the Node's architecture as well as if the Node is even capable of scheduling Unikernels.

The capability of a Node to schedule Unikernels is characterized by its ability to use hardware virtualization. This ability to use hardware virtualization is indicated by the CPU. On x86 CPUs, the extensions are either AMD-V, indicated by the svm flag [51], or VT-x indicated by the vmx flag [52] for AMD and Intel, respectively. On ARM64, there is no CPU flag indicating the support for hardware virtualization but most systems, if running in 64-bit mode, should be able to.

Architecture is another important aspect when it comes to Unikernel scheduling. The architecture of a Unikernel or a hardware-specific container application, for that matter, needs to match the one of the host system. Otherwise, the Unikernel is limited to software emulation, which will, in most cases, lead to a significant decline in performance. The emulation would be done via QEMU. Instead of forwarding the instruction of the architecture via KVM and executing them directly on the CPU, the TCG emulation would be required. This would lead to the Unikernel being interpreted instead of being run on the hardware, negating not only the possible advantages but adding additional overhead to the service.

While the creation process of Unikernels can be more complicated and require more work in regards to porting of libraries, one of the supposed benefits of the Unikernel deployment could potentially be greater performance due to a multitude of factors. Those factors include simplifying the runtime, with only the necessary elements being present and eliminating context switches due to system calls even without optimizing the application to be deployed in a Unikernel [3].

3.1.1. Limitations

While Unikernels can have a positive performance impact, some limitations apply. First, the usage of GPUs and other accelerators is limited to only partial or no support at all. This limits the applications that can be run as Unikernels to those that do not rely on the usage of such accelerators. Further, since Unikernels are not sharing the same kernel as a container does, the overhead for the Unikernel should increase in some circumstances with this being explored further in chapter 5. Additionally, while not applying to all Unikernels, in the case of Unikraft, there is, at the moment, no preemptive scheduler limiting the way how an application can be structured. While this is the case, the non-preemptive scheduling of Unikraft could also lead to better performance because less locking within the kernel is required.

3.2. Unikernel Application Porting

This section introduces how typical applications can be ported to a Unikernel platform and what needs to be considered in the process. The porting process is focused on the Unikraft framework introduced in subsection 2.2.3, which is at, the time of writing still, in heavy development, thus, the porting process could be subject to change in the future.

3.2.1. General Process

Most applications can be made into Unikernels with some limitations, as discussed in subsection 3.1.1. However, the process of porting and the change in performance can vary significantly from application to application. It depends on a multitude of factors. Those include the application's use case, the language used, and the dependencies the application relies on for its functionality. While more and more languages are supported by Unikraft, the standard library might not yet be fully supported, or an external library might need the additional support of, for example, Linux system calls which might not be available for Unikraft. Further, the nature of virtual machines requires an additional network stack which is introduced on top of the host network stack and adds additional delay for communications from the Unikernel to the outside Network.

In Unikraft, all elements of the application are a library, as already described in subsection 2.2.3. This includes not only the former user space application but all parts of the Unikernel, e.g., schedulers and memory allocators. This modularity allows for the ability to remove not only unnecessary elements from the kernel but also replace certain aspects of the kernel with different implementations. This is done by all kernel components with the same functionality to implement the same APIs. This kind of modularity can have many benefits. For example, if a scheduling strategy does not work well with a given application, the scheduling library can be replaced with a more beneficial implementation.

In the most general terms, the porting process is to go over all parts of an application and port all necessary dependencies as Unikraft libraries or replace the dependency with an already available Unikraft library. In the most simple case, all libraries required are already available. Therefore, only the application can be ported directly by creating a Unikraft

compliant Makefile without any additional modifications. In a more complicated scenario, the library might need to be modified. This could apply to libraries that rely on some kind of special functionality, such as accelerator support. If such requirements are in place, Unikernel deployment might not be the best choice for the service.

3.2.2. Simple Applications

The first step in the porting process is to set up and configure the Unikernel environment. There are two possible ways to set up a Unikraft environment. The first one is the tool `kraft` [53] provided by Unikraft. This python-based tools can be used to set up all necessary libraries and build the Unikernel. The second way is to manually set up the Unikraft environment and use a Makefile to build the Unikernel.

The setup process using `kraft` looks like the following:

```
kraft up -p <platform> -m <architecutre> -t <template> <applicaton-name>
```

This command will initialize all required libraries and the Unikernel itself based on the specified template. The setup can also be done manually without the `kraft` tool. In that case, the folder structure for the application needs to be generated manually. In both cases, the working directory of Unikraft is organized as shown in Figure 3.1. The Unikraft kernel libraries are contained within the `unikraft` folder. The `libs` folder contains the external libraries of the application. Those include networking support with *lwip* [40], standard C library support via *newlib* or *musl* [54, 55] and also entire languages like Python and Ruby [56, 57].

Further, the `apps` folder contains the actual Unikraft applications. The libraries in the `library` folder do not contain the actual code base of the libraries but instead contain additions that are not in the original code. This can include additional functions that a Linux userspace application might expect to be available as system calls or replace internal functions with Unikraft compatible ones. Additionally, the `patches` directory can include patch files. Those files are applied whenever the library is retrieved from the Internet. The URL for the library is contained within the "Makefile.uk" file. This file additionally defines how the library needs to be compiled to be properly usable within Unikraft. These compile configurations include the compiler flags and components to be built. The `include` directory can contain additional header files and is given as an include path to the compiler in the Unikernel build step. Finally, The "Config.uk" file contains different configuration which can be selected independently for each Unikernel application. These configurations set the behavior of the components and can include elements like the inclusion of a main function in the case of libraries like Python or the ability to disable or enable the functionality of the library further.

There is no major difference between a library and an application in Unikraft. They shared most of the configuration and additional elements apart from the `include` and `patches` folder on the library side. However, applications have an additional Makefile and "kraft.yaml" file. Both can be used to configure the Unikernel build process. The "kraft.yaml" is a YAML formatted file containing information about the required libraries. This includes the version of Unikraft, which libraries are to be used and the build targets indicating for which target the application can be built. Those build targets specify both system architecture as well

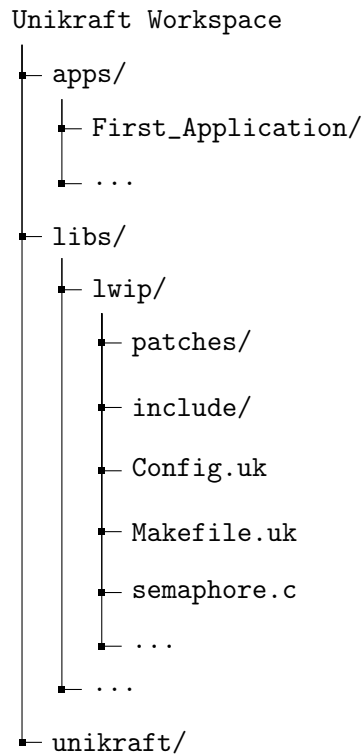


Figure 3.1.: Unikraft Workspace Directory

as the virtualization platform. Further, the libraries and the Unikernel can have further configurations attached to them, specifying the switches defined in each library's "Config.uk" file.

The YAML file is only used in combination with the kraft tool. If the Unikernel is to be built without kraft, the Makefile can be used. However, the Makefile only specifies which libraries are to be used and does not provide further customization regarding library configuration. The configuration and build process of Unikraft has the following steps.

1. `make menuconfig`

The first step is using the menuconfig tool. This tool comes from the Linux kernel and is used there to configure the kernel. In the case of Unikraft, it is used to configure the kernel libraries and the additional libraries and for what architecture and virtualization platform the kernel should be built. This process creates a ".config" file containing all libraries' configurations.

2. `make fetch`

The second step goes through all libraries and fetches the actual library data from the source defined in the "Makefile.uk" file of each library. After extracting the tars, the patches for each library are applied to the source code.

3. `make prepare`

The third step is setting up the configurations required for the build process.

4. `make`

The fourth step is to build all required components and the current application. This process generates an object file for each component, which is then used to link together the Unikernel.

In case the `kraft` tool is used, the commands are the following.

1. `kraft init`

This command sets up the `kraft` configuration for the application.

2.1 `kraft menuconfig`

The second command is the same build step as with the Makefile. This command can be replaced with the following command.

2.2 `kraft configure`

The difference in using the `configure` command is that the `configure` file takes the "`kraft.yaml`" into consideration and offers valid configurations derived from it. Thus, the "`kraft.yaml`" can be used to create preconfigured Unikraft configurations that can be used to initialize the build system and the application, which is defined by it without additional configuration overhead. Otherwise, the command fulfills the same role as the fetching and configuration steps of the Makefile by retrieving the source code for all libraries.

3. `kraft build`

The third and last step is the same as with the Makefile version. The building process is initialized and the Unikernel is created.

With this, the Unikernel has been created and can be deployed. This method applies to all Unikernels built with Unikraft from the most straightforward case to more complex applications, as can be seen in the next section.

3.3. Porting of Containerized Services to Unikraft

This section goes over the porting process of the object tracking video pipeline proposed in [58]. The application consists of four distinct elements. These four services are the Video Source, the Video Aggregation, the Object Detection and the Object Tracking service. The services work together to enable real-time object tracking in a video stream. The component that has been ported to demonstrate the abilities of Unikernels has been the Video Aggregation service. While the Tracking and Detection services are heavily reliant on larger libraries and can also gain from the availability of accelerators, the Aggregation service does not gain as much in most cases. The service is highly reliant on the CPU of the host system, making it a good choice for the portation to a Unikernel platform. Another element of why the Aggregation service was chosen is that while the original implementation relied on OpenCV as a library for video processing, this can be reduced to a subset of the libraries used by OpenCV to achieve the same functionality, leading to lower complexity of the porting process.

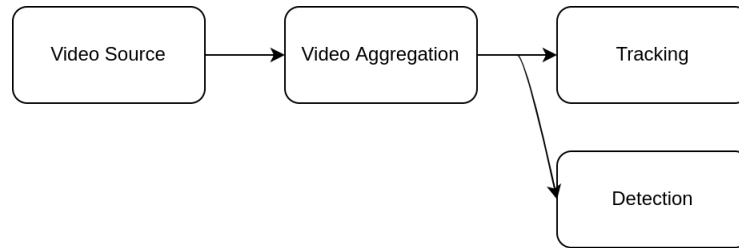


Figure 3.2.: Video Aggregation in the Video Analytics Pipeline

This is not possible with the other services, which rely heavily on the functionality of OpenCV. Thus, the Aggregation service was chosen since it has the most to gain by running as a Unikernel with the smallest implementation overhead.

3.3.1. Video Aggregation

The Video Aggregation Service receives the video from the Video Source and processes it. Further, after processing a frame, it sends it out to the Object Detection and Object Tracking services. The processing includes scaling the image to a defined resolution and converting it to a JPEG to save on bandwidth.

In detail, the Video Aggregation service consists of multiple threads. One is reading from the incoming video stream and resizing each frame it receives from the video source. Those resized frames are then added to a queue of frames to be sent out together with the time it took for the frame to be resized. The other thread is going over the queue and processing each frame. This thread keeps track of the number of frames it sends out and the time it takes to process each frame. Each frame is either sends out to the Detection service or the Tracking service based on the current frame count and a preset detection frequency of the service. If the previous frame has not been sent when a new frame is processed, the frame is dropped. Otherwise, the frame is converted into a JPEG to save bandwidth for the transmission and sent to one of the services via gRPC. The frame is sent together with a timestamp and the frame number of the frame in the video stream. Further, the service measures internal processing times for different process aspects. These aspects are the time it takes the frame to be resized, the skipped frames and the time for the frame to be sent to the next service.

The original version uses Python as language and *OpenCV* for processing the incoming video frames, resizing them and converting them to JPEG. *OpenCV* can use FFmpeg and OpenCL to add GPU acceleration to the image processing. The goal of the porting process was to enable as much of the functionality of the original implementations as the Unikernel environment was capable of while also improving upon it to increase its performance.

3.3.2. Porting Process

There are multiple elements of the original service to consider when moving to a Unikernel, in this case, Unikraft in particular. The first element of consideration is the usage of Python as the language of the application. While it is possible to run Python within a Unikernel

environment, there are still some limiting factors regarding library availability and functionality. Thus, a different approach was chosen to reduce the overhead of porting a multitude of Python dependencies. The service was re-implemented in C. This is to keep the number of dependencies low while also allowing the same capabilities as the previous version, making the new one comparable to the previous Python one in terms of functionality.

One more element to consider is the used libraries. In the original, the frame extraction and processing are handled by *OpenCV*. *OpenCV* is an extensive library with many additional features not required to achieve the task of the Aggregation service. While in the other services, *OpenCV* is used for more in-depth image processing, the tasks required in the Aggregation service can be directly done with the underlying components used within *OpenCV*. In this case, the component is the library collection FFmpeg [59]. FFmpeg is a collection of tools and libraries for video and audio processing that enables all the necessary processing for the Aggregation service.

Regarding the communication with the following services, the protocol in use needs to be considered. gRPC enables remote function calls between different endpoints and languages but has yet to be ported to Unikraft. Thus, a different way of communication was implemented to enable the services to receive frames from the Unikernel implementation without the required additional dependencies. The new implementation replaces the gRPC calls with two direct TCP connections between the Aggregation service and the Tracking in Detection services.

The original implementation uses threading and *asyncio* API [60] calls to realize the communication. While this is also possible with a Unikernel, Unikraft, at the time of writing, does not have preemptive scheduling [61]. This limits the program's ability to utilize multiple threads in execution properly. Threads must manually yield the execution to allow the scheduler to make decisions. Thus, the threading has been removed from the service in the process of porting. It is also worth mentioning that this limitation would not affect the Python threading module because the threads, in that case, are handled within a single thread of execution. This single thread is the Python interpreter managing the scheduling internally as user-level threads. Thus, not requiring actual preemptive scheduling support for this functionality.

The last element to consider is the collection of the measurement from the application. The measurements collection is achieved by sending the information in JSON format via HTTP. For this reason, the library *libcurl* [62] has been ported as part of the re-implemented version.

As already stated, the new implementation is generally comparable in terms of functionality, but this is not the case when it comes to its performance. This is explored in subsection 5.2.2.

FFmpeg

FFmpeg is a set of tools and libraries created to handle audio and video files and streams. The main components of FFmpeg are detailed in the following. *Libavcodec* is the library containing the general encoding and decoding framework and multiple different encoders and decoders. *Libavdevice* is the library responsible for accessing input and output devices. *Libavfilter* contains the filter framework and multiple filters. *Libavformat* is responsible for providing the

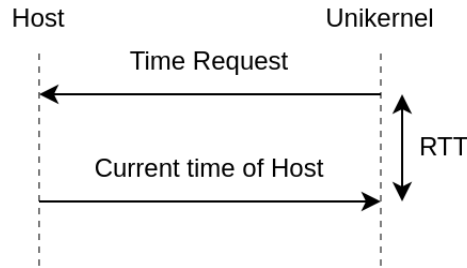


Figure 3.3.: Time Synchronization with Host and Unikernel

framework for multiplexing and demultiplexing of media formats. *Libavutil* is a general helper library providing additional functionality which is shared between components, including random number generation and string operations. Finally, *libswresample* and *libswscale* are providing the capabilities of audio and video processing, including resampling and image scaling.

For the port of FFmpeg, most of the libraries are required to process the video for the Aggregation service. For this reason, most of the functionality of FFmpeg has been ported to Unikraft. This includes the RTSP demuxer, which is required to receive the video stream in the Video Pipeline. Further, codecs or acceleration modules which require hardware support were disabled in the configuration. In the process of making the version of FFmpeg less target specific, the optimizations written in assembly were taken out in the configuration process. While this may lead to lower performance, the version of FFmpeg becomes more portable between different system architectures. The FFmpeg build process uses the GNU Autotools to build the project. Those tools generate multiple configuration headers and Makefiles and fit the build to the current system. The generated configuration, while mostly suitable for the Unikernel, still contained some architecture specific configuration in the form of inline assembly, which was removed from the build process. Further, since all components of FFmpeg are more or less independent libraries, the libraries have been individually added as Unikraft libraries.

Libcurl

Libcurl is a library that can be used as a client for data transfer via multiple different protocols. The focus of the port was to use the library to connect to an external server via HTTP. Thus, only a minimal version of *libcurl* was configured with only HTTP support. The configuration files generated by the Autotools, disabling most protocols yielded, were mostly usable for the Unikernel, except for a few functions and headers which were manually removed from it. The function not available were *fnmatch*, *getifaddrs* and *sigsetjmp* and the header was the Linux TCP header. All of these can be removed from *libcurl's* configuration without losing the required HTTP functionality.

Time Synchronization

In the process of estimating the performance of a Unikernel, it can be necessary to measure times within the Unikernel and compare them to the host's time. While at first glance, this might be achieved without any modification. However, through our repeated experimentation, we found this not to be straightforward. Specifically, we found that the time within the Unikernel is only accurate to about 1 second of the host system's time. This leads to a problem when trying to compare very small time durations. If a test requires synchronization, the time is synchronized with the host based on the RTT and timestamp from the host.

In Figure 3.3, the sequence for the time synchronization is depicted. First, the Unikernel starts measuring its current time and sends a request for the host's time to the host. The host responds with its current time, and upon receiving the answer from the host, the Unikernel measures the timestamp at which the packet arrives. From both timestamps, the Unikernel can calculate the RTT to the host system. Further, together with the timestamp contained in the packet the host send, the time delta is derived. This time delta can be used to correct the Unikernel's internal time. The accuracy of the time delta is within one RTT. Because of that, the process is repeated multiple times and the result with the best RTT is chosen to be used as a time delta.

When the Unikernel is restarted multiple times, the resulting cumulative distribution indicates a uniform distribution of the time difference from the Unikernel to the host system. The measurement of the difference throughout 1000 measurements can be seen in Figure 3.4.

Service Design

As mentioned in this chapter, the operation of the service was slightly changed in the re-implementation. The new service implementation is described in the following way. The video source and the addresses of the other services are still processed via the command line. In the case of a Unikernel, this is given as kernel parameters. The metric collection is only enabled if an address is given to the service via a parameter. If such a parameter is defined,

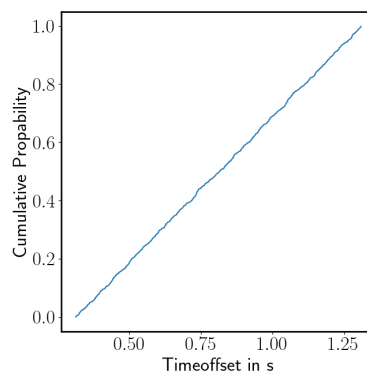


Figure 3.4.: Cumulative Distribution of the Time Difference

Listing 3.1: Frame Data Structure

```
struct __attribute__((__packed__)) jpeg_packet{
    uint32_t size; // Size of buffer
    uint64_t timestamp; // Timestamp of Current Frame
    uint64_t frame_number; // Frame Number
    uint8_t buffer[]; // JPEG encoded Frame
};
```

the initialization of the measuring data structures and *libcurl* is processed. *Libcurl* is used in the implementation to send the runtime measurements to the collection server via HTTP. Regardless of whether the collection process is initialized, the application continues with the next step, comprised of the setup process to access and manipulate the video source. Since the service uses FFmpeg, it allows both the Unikernel and the Linux user space variant to work with various different video formats and streams. First, the video source is opened with FFmpeg. Then, the decoder for the given video format and the encoder for the JPEG conversion, as well as the conversion context for rescaling the image buffer are created. In the last preparation step, the application connects to the other services via TCP. Finally, the program enters a loop, processing and sending each frame.

The frames are received by querying the video stream. If a video frame is returned, it is rescaled to the desired resolution and encoded as JPEG. This frame is then sent out to either the Tracking or Detection service, depending on the current frame number. If the frame number is a multiple of a set value, which is set to 20 per default, the frame is sent to the Detection service. Otherwise, it is sent to the Tracking service. It is sent, including the frame number and a timestamp, like in the original implementation, which used gRPC instead. However, instead, the information is sent in the form of a packed C struct, as shown in Listing 3.1. This loop is processed until the end of the file or stream is reached. In each iteration of the loop, timestamps are collected to measure different parts of the application's execution. After each frame is sent, the application collects the data in the form of a JSON document and sends it to an external server via HTTP, where it can be further processed.

4. Oakestra Integration

This chapter introduces a possible solution on how to enable Unikernel scheduling on the edge. The proposed approach is based on the already existing foundations of Oakestra's ability to schedule container-based applications. This includes changes to the scheduling behavior and aggregation behavior in both the Root Orchestrator, in subsection 4.2.4, and Cluster Orchestrator, in subsection 4.2.3. Further, it goes into detail on what is required for the Nodes to be able to support Unikernels. And finally, it goes over how to enable a Unikernel runtime, of which the design is also described, on the Worker Nodes in subsection 4.2.1. In section 4.1, the requirements the implementation needs to uphold are laid out.

4.1. Requirements

The following requirements are mandatory elements for the implementation of both the scheduling and the runtime environment of Unikernels. They guarantee that the Unikernel is executed in a suitable environment while not hindering the execution of Containers, and enabling cross-virtualization networking.

1. **Transparency**

The deployment of the Unikernel must be transparent to the end user or the network. There should be no difference between deploying a Unikernel or a container-based microservice. This requires Unikernel and Containers to be able to communicate within the Oakestra framework and support the same management API calls.

2. **Modularity**

The Capability to run a Unikernel should not be a requirement for the Node but instead, be modular. This enables the Node to be run on systems that do not support hardware virtualization. However, while they are still supported, the runtime is limited to only support container-based deployments. This also requires the scheduling to differentiate between virtualization technologies.

3. **Extensibility**

The implementation should be extendable to allow further changes regarding the requirements. This may include a change of hypervisor or how certain elements of deployment are handled, e.g., switching from QEMU to Firecracker.

4. **Network Isolation**

Unikernel deployment must use the same isolation mechanisms as the container-based applications. This includes each Unikernel instance to run within a separate namespace to isolate them from other microservices.

5. **Architecture Independent**

The Unikernel must be able to perform the task that it is given without being held back by hardware constraints. In this instance, hardware constraints mean the hardware compatibility of the Unikernel and the host system. So, for example, an AMD64 Unikernel should not be assigned a cluster or Node that only supports AArch64 based systems.

6. **Strict Resource Usage**

The resource constraints set by the deployment descriptor for any given Unikernel microservice should be upheld. The constraints in the deployment descriptor set the different resources a microservice is supposed to use and a Unikernel should also be bound by these constraints.

7. **Virtualization Requirement**

The Unikernel must be executed with virtualization support of the host. Otherwise the performance of the service would degrade. Thus it must be enforced that the Unikernel and the Node's system are binary compatible and that the host system can use hardware virtualization. The virtualization support, which enables the use of KVM, is indicated by either hardware flags in the case of AMD64 or the architecture itself in the case of AArch64.

8. **Low Initialization Overhead**

The initialization of the deployment request should only introduce minimal overhead and be executed as efficiently as possible. Furthermore, the overhead should be within the delay of Container services.

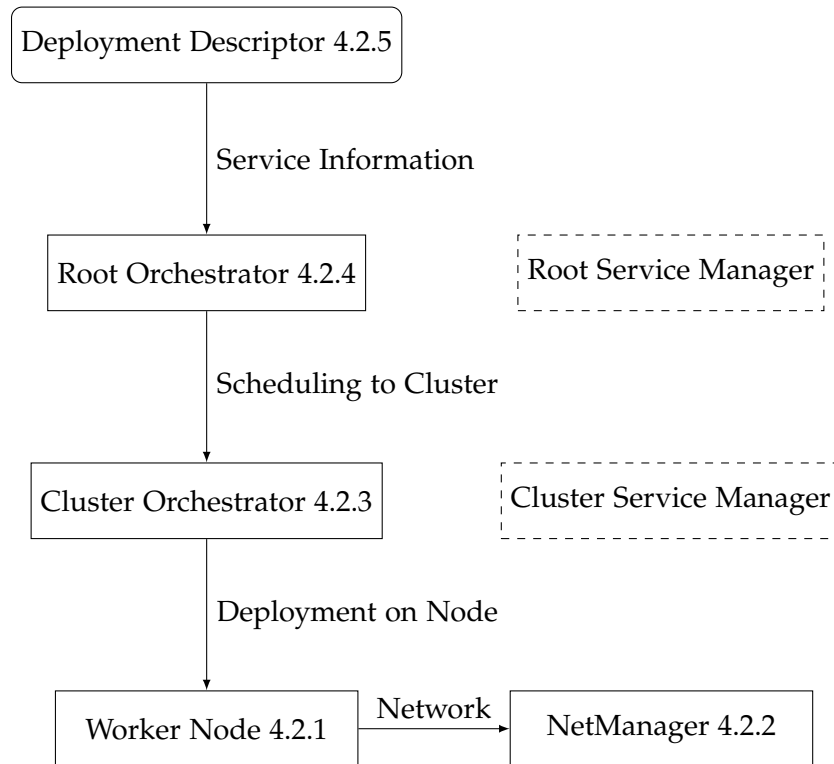


Figure 4.1.: Oakestra Components

4.2. Oakestra

The changes proposed in the following detail how a Unikernel deployment can be achieved with the proposed extension to the Oakestra framework without disrupting the previously established hierarchy. While the changes affect most elements of the scheduling process, the networking only needs to be extended at the Node level with the NetManager. At the same time, the Service Managers stay the same. The overview of Oakestra's components can be seen in Figure 4.1. As already mentioned, the network components do not require changes at the higher layers. This is due to the integration of the Unikernel in the network of the Oakestra, which makes a container and a Unikernel indistinguishable on the network. Thus the functionality for both the Service Managers is not affected as long as the NetManager can correctly configure the network environment for Unikernels. The overview starts from the top with the deployment descriptor, which specifies the services and their containing microservices to be deployed and goes then over the hierarchy from the Root Orchestrator over the Cluster Orchestrator to the Worker Nodes and its local communication with the NetManager.

Listing 4.1: Node Service Struct

```
1 type Service struct {
2     JobID string `json:"_id"`
3     Sname string `json:"job_name"`
4     Instance int `json:"instance_number"`
5     Image string `json:"image"`
6     Commands []string `json:"cmd"`
7     Env []string `json:"environment"`
8     Ports string `json:"port"`
9     Status string `json:"status"`
10    Runtime string `json:"virtualization"`
11    StatusDetail string `json:"status_detail"`
12    Vtpus int `json:"vtpus"`
13    Vgpus int `json:"vgpus"`
14    Vcpus int `json:"vcpus"`
15    Memory int `json:"memory"`
16    Architectures []string `json:"arch"`
17    Pid int
18 }
```

4.2.1. Worker Node

Starting from the lowest element in the hierarchy, the following goes over the workings of the Worker Node and explains the integration of Unikernel support into it.

As detailed in subsection 2.3.2, the Nodes are responsible for providing the actual hardware integration on which microservices run in Oakestra. When the Node receives a deployment request from the Orchestrator via MQTT, the information sent is unwrapped into a struct with the structure seen in Listing 4.1. The struct contains the information about the microservice to be deployed. This includes the requirements of the service regarding memory, CPU cores and required accelerators. Further, it specifies the network configuration in the form of ports to be forwarded and IP addresses to be used by the microservice. Further, general information about the process is included, such as the service's ID, name and instance number. The instance number differentiates the service from already deployed services with the same *Sname* and *JobID*. Also included in the struct is the command which should be run in the container environment and a link to a container registry containing the container to be executed.

In the next step, the runtime is selected based on the information and the deployment is initialized. Each runtime implements an interface that is used to deploy and undeploy services. When looking at the implementation before the extensions made by this Thesis, the only available runtime was the container runtime.

The suggested changes add a new kind of deployment. In this case Unikernels deploy-

Listing 4.2: Runtime Interface

```
type RuntimeInterface interface {
    Deploy(service model.Service,
           statusChangeNotificationHandler func(service model.Service)) error
    Undeploy(sname string, instance int) error
}
```

ment to the Nodes. This is achieved by adding the ability to advertise supported runtime environments for Unikernels. This happens at the start of the Node application. Before the Node registers itself with the cluster, it goes over its virtualization technologies and adds them to a list of supported technologies. The Container runtime is, in all cases, required and the Unikernel runtime can be manually disabled by a parameter at startup or is not used in case the required software is not installed on the target system. Otherwise, both technologies are appended to the list. In the next step, the Node registers with the Cluster Orchestrator and advertises its hardware and supported virtualization technologies. From here on out, the Node waits for deployment or undeployed requests via MQTT and sends regular updates about its available resources.

If a Unikernel is supposed to be deployed, the Unikernel runtime is selected by the runtime informant given in Listing 4.1 and the deploy method is called for the Unikernel runtime. From here, the runtime registers the service in the list of currently existing services on the Node. Then, it starts setting up the Unikernel environment for the service in a different thread. The first thing the creation process does is generate the name of the service, called *taskID*, in the Container Runtime. This name is used to identify the service throughout its lifetime and consists of multiple elements of the deployment descriptor seen in Listing 4.9. This is done, so the identifier for the Unikernel is unique on the Node and no overlap between different deployments can happen. Most of the elements of the name are already part of the *Sname* contained in the struct, as can be seen in Listing 4.1. The *Sname* is made up of the application name to which the service belongs, the namespace of that application, the name of the service and the namespace of the service, all separated by dots. This already identifies the microservice in general but does not prevent overlap with different instances of the same service. Thus the name used for identification further appends the instance number to the string. This uniquely identifies the microservice across the Oakestra system.

Unikernel Image

The next step in the creation process is acquiring the Unikernel image. The Unikernel image is specified in the *Image* field of the struct. In the case of a Container, this would point to a Container registry, based on the OCI specification, from which the Container can be acquired. In the case of a Unikernel, there is no standardized way to deliver them.

In the following, the current way the implementation retrieves the Unikernel images is detailed. For the purpose of identifying the image which can be used for the current Node,

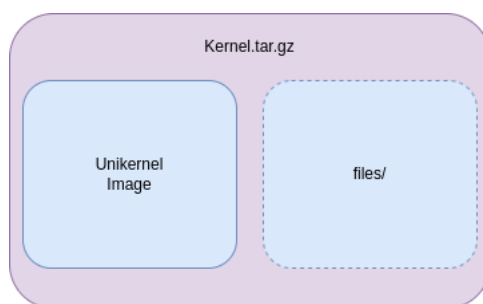


Figure 4.2.: Unikernel Tar structure

the Architecture field has been added to the service struct, seen in Listing 4.1, and the deployment descriptor in subsection 4.2.5. All Unikernel images for the microservice are comma separated in a string of URLs pointing to the Unikernel images via HTTP. The Node separates the URLs and goes over all the available architectures listed in the Architecture field. The number of given architectures and Unikernel images must always be the same. Otherwise, the deployment might fail. The Node should always receive a matching architecture in the list of architectures for the service. This is guaranteed by the scheduling process. Once a match is found, the Node proceeds to download the Unikernel image from the given URL. The Unikernel in the remote location must be formatted correctly to be further processed. The correct format is shown in Figure 4.2. The Unikernel is wrapped in a tar archive containing the Unikernel itself as the kernel file. Further, an optional files folder can be included, which will be mounted inside the virtual machine as the root directory if included. The archive is unpacked in a folder named after the *Sname* of the service.

Network Overlay

If the network overlay is used, the Node sends a request to the NetManager (subsection 4.2.2). It is responsible for creating the Unikernel's namespace and networking devices. The request type is special to Unikernels and significantly differs from how Containers are added to the network. This is necessary because Unikernels require a different network setup than Containers. Unlike Containers, the Unikernel networking needs to be set up before the actual runtime creation. With Containers, the manipulation of the network configuration is still viable at runtime, which is not the case for Unikernels. The named namespace the NetManager creates is based on the names and namespaces defined in the Deployment Descriptor laid out in subsection 4.2.5. This ensures that no overlapping will occur if multiple different services or instances of the same service are deployed on the same Node. For example, the namespace generated for the first instance of the microservice in Listing 4.9 would be *aggr.aggr.testaggr.aggr.instance.0*. The usage of the namespace is described in subsection 4.2.2.

While it enables the Unikernel to have the same layout as the Containers in regards to being encapsulated in a namespace, it also adds overhead to the routing for the Unikernel because of the reliance on additional network elements, which are also described in subsection 4.2.2.

Listing 4.3: QEMU Command

```
1 qemu-system-x86_64 \  
2     -kernel <kernel_path> \  
3     -nographic -nodefaults -no-user-config -cpu host -enable-kvm \  
4     -netdev tap,id=tap0,ifname=tap0,script=no,downscript=no,br=virbr0,vhost=on \  
5     -device virtio-net,netdev=tap0 \  
6     -append "netdev.ipv4_addr=192.168.1.2 netdev.ipv4_gw_addr=192.168.1.1  
7     netdev.ipv4_subnet_mask=255.255.255.252 -- <args>" \  
8     -fsdev local,id=fsid,path=files,security_model=none \  
9     -device virtio-9p-pci,fsdev=fsid,mount_tag=fs0,disable-modern=on \  
10    -qmp unix:<qmp_path>,server,nowait \  
11    -m <Memory> -smp <cpu cores>
```

Listing 4.4: Execution in Network Namespace

```
ip netns exec <Unique Name> <QEMU Command>
```

QEMU Command

Regardless of the network overlay being present, the next step is the preparation of the command used to execute QEMU. QEMU was chosen because of its high compatibility and widespread availability in the default repositories of most Linux distributions. The Node uses `golang's exec` package to run the QEMU command directly. The command consists either of only the QEMU command or, in case the network overlay is used, the command is prefixed with a call to run it inside of the namespace previously prepared for it by the NetManager. For this reason QEMU front ends like `libvirt` were not used in the resulting implementation. While `libvirt` eases the management and configuration overhead for QEMU, it does not currently support the usage of network namespaces. Thus, the decision was made to use QEMU directly. This also reduces the overhead that a further abstraction would have introduced, e.g., the configuration of `libvirt` uses XML documents which would first need to be created and then be parsed by `libvirt` again before the actual Unikernel could be started. With the reasons for the use of QEMU clarified, an example of a command which is created in this proposed runtime can be seen in Listing 4.3. Moreover, with the overlay enabled, the command is prefixed with the `ip` command to run QEMU within the named namespace, which can be seen in Listing 4.4. From the top down, the command first specifies which kernel image to use. The kernel image, in the case of a Unikernel, is the entire system. The Unikernel image in the command is shared between all local instances of the microservice. This reduces the required disk space because the image can be reused and does not require its own file mount as a Container would. In case another instance of the same microservice is requested, the previous download step detects that the Unikernel is already present on the Node and, in that case, only checks if the control process of the microservice with the

same instance number is still running. Such an overlap could only happen if the Unikernel has unexpectedly stopped its execution. In this case, Oakestra might already try to schedule the instance again while the cleanup of the previous run of the instance is still ongoing. Regardless whether the instance is new or rescheduled, the reuse of the Unikernel limits the amount of disk space needed for each instance. However, the files directory, if present, needs to be reproduced for each instance to avoid simultaneous access to the files contained.

Further, in line 3, multiple elements are configured. Most important is the enabling of KVM. Otherwise, QEMU might run the Unikernel in interpreter mode, which would significantly impact the performance of the Unikernel, as described in subsection 2.1.3. The other parts of the line set the CPU type QEMU should report to the Unikernel as the current host CPU, that no default configurations should be applied and that no graphical user interface is required for the run. While QEMU can be compiled without the need for a GUI, the default configuration and the one provided by most distributions include it. The default configuration would include multiple devices, which are not required for the Unikernel. These devices include serial and parallel ports and others.

Lines 4 and 5 of Listing 4.3 are only present if the network overlay is used. They specify the tap device which is always called tap0 and register it as a virtio-net device for the Unikernel. The virtio-net device is a paravirtualized network device communicating with the guest over the virtualized PCI bus. Additionally, the "vhost=on" option instructs QEMU to enable the usage of the vhost kernel module, which improves the network performance of the virtual machine [63].

The sixth and seventh line specifies the arguments that will be passed by QEMU to the Unikernel. The first half before the "--" configures the network configuration of the running guest. This step is specific to Unikraft. Other Unikernels might need to be pre-configured with the correct address, subnet and gateway. The second half specifies the actual command for the Unikernel. These are taken from the `Commands` contained in the service struct depicted in Listing 4.1.

The following two lines, the eighth and ninth line, are only present if the files directory is present for the current microservice instance. In the download process, if the files directory is contained within the archive, the folder is copied to a predefined location within a folder with the unique name of the instance. The filesystem used is 9P with the virtio-9p-pci device, which uses PCI to communicate with the Unikernel. This enables the mounting of the folder directly into the virtual machine.

The second to last line configures the QMP, which stands for "QEMU Machine Protocol" and is used to communicate and control the QEMU process. The QMP connection is later used to end the execution of QEMU.

The last line configures the memory and the number of CPU cores the Unikernel has access to. The values for both are taken from the service description. In the case of memory, it specifies the amount the Unikernel has access to from the time of creation. QEMU will reserve it from the host, unlike the case of Containers in which the Containers do not need to use the entire memory from the start of creation. The number of CPU cores the Unikernel has does not actually need to be within the limits of physical cores. However, the scheduling

Listing 4.5: Unikernel Information

```
type qemuDomain struct {  
    Name string  
    Sname string  
    Instance int  
    qemuProcess *os.Process  
}
```

process done by the Root and Cluster Orchestra prevents oversubscribing of cores to the virtual machine. The requested number of threads is passed through the Unikernel and can all be used. Unlike the memory case, there is no reservation of the CPU cores. Thus, multiple systems can run in parallel, even on the same cores.

QEMU Execution

The next step in the Node's execution is the actual execution of the QEMU command. After the execution is initialized, the Node process attempts to connect to the QEMU process via QMP. At this step, the creation process waits for the QMP interface to come up or until an error occurs. If no QMP connection is possible, the QEMU process is considered dead and the deployment of the Unikernel is considered to have failed. This should never happen with Oakestra unless the Unikernel image is faulty, e.g., incorrect architecture or built for different virtualization. If the QMP connection is successful, the process stores the important information of the Unikernel in a struct, which can be seen in Listing 4.5. The information includes the unique name of the service instance, the *Sname* of the service and a reference to the process of the QEMU execution. The reference points to the necessary information, such as the process PID. From here on out, the Unikernel continues execution until an undeploy request is received via MQTT, the Node's execution is stopped or the Unikernel finishes its execution.

Undeploy Request

In case an undeploy request is received, the Unikernel deployment is stopped in the following way. The undeploy is accompanied with by same struct (Figure 4.1) as the deployment request was. If the Unikernel runtime is used for the service, the Undeploy method of the runtime is executed. This method accesses the map of active Unikernels and sends a signal to the thread responsible for the Unikernel. The thread sets the Unikernel to the status `SERVICE_DEAD`, publishing it via MQTT and proceeds to stop the QEMU process. This is accomplished by using the QMP connection setup in the initialization process. QMP takes JSON formatted instructions. In this simple case of closing QEMU, the instruction can be seen in Listing 4.6. The `quit` instructs QEMU to stop execution as quickly as possible. The documentation [64] specifies that QEMU might return with an acknowledgment but the QEMU process might

Listing 4.6: Unikernel Exit Instruction

```
{"execute": "quit"}
```

stop before this. Thus failure to receive anything back from QEMU is considered a successful operation by the Node. Further, if the network overlay is used, a delete request is sent to the NetManager. The Unikernel environment, in Listing 4.5, is deleted from the global map of Unikernel services, and the folder which contained the QMP connection file is deleted. This folder also includes the directory mounted by QEMU in case additional files were provided with the files folder. In the last step, the responsible thread sends back a signal if the undeploy was successful and in the Undeploy method, the service is removed from the list of running services.

Resource Aggregation

The Nodes collect information about all locally running services. This also includes Unikernels. The resource monitoring periodically goes over all Nodes and reports their status via MQTT to the cluster. Since QEMU is used, the resources the Unikernel uses can be directly acquired by looking up the QEMU processes PID. This PID is saved for all running Unikernels in the struct shown in Listing 4.5. This allows the Unikernel runtime to report the same resources as the container counterpart.

4.2.2. NetManager

The NetManager, as part of the Worker Node, also needs to be modified to accommodate Unikernel deployment. The changes enable the NetManager to create the required environment for Unikernels to communicate and make the communication between them and Containers possible. The communication between Unikernels and Containers is entirely interchangeable, meaning a service does not know if it communicates with a Unikernel or a Container.

In the case of Containers, the host bridge is connected to the namespace of the container by a virtual Ethernet connection while the Container is already deployed. This, however, is not sufficient for a unikernel-based deployment since a Unikernel does not operate within the same kernel. This difference results in the Unikernel requiring its own network stack, thus requiring additional setup.

In general, the NetManager receives requests about deployment and undeployment via HTTP POST. The information the deployment requests contains is the name and instance number of the service as well as the port mapping that is requested. After the NetManager receives a request for deployment containing the necessary information, it creates the network namespace depicted in Figure 4.3 in the following way. At first, a new namespace is created. This step is necessary because the QEMU instance the Unikernel runs in can only be started once the network setup has been completed. Otherwise, QEMU would not be able to set up

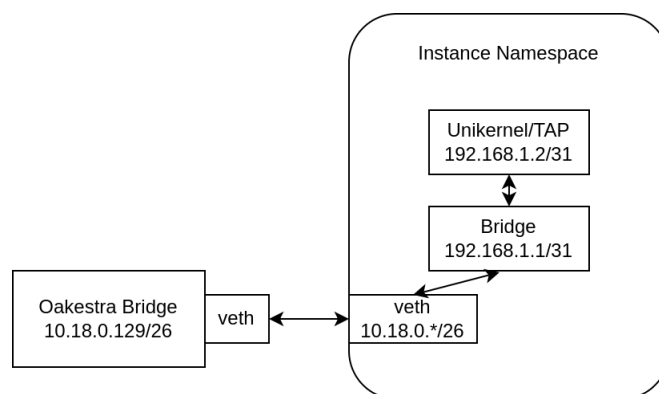


Figure 4.3.: Namespace Networking

the required configuration to enable the Unikernels to interact with the network afterward. Thus, the namespace must be created by the Net Manager before the Unikernel process has started. Unlike the Container counterpart, where the container engine handles the namespace configuration of the container. This namespace is unique for each Unikernel running on any Node and is known by the Node managing the Unikernel. The namespace creation is followed up by the creation of a virtual Ethernet pair to connect the namespace to the Oakestra bridge. Up to this point, the process is quite similar to the one for Containers, with the exception of the manual creation of the namespace. However, the Unikernel requires additional elements in the chain. The devices that are required are a bridge and a TAP device within the namespace of the Unikernel. The TAP device is attached to the bridge within the namespace. The bridge is required to enable the commutation over the TAP device with the rest of the namespace. This is the TAP device given to QEMU in the initiation command to enable networking for the Unikernel (Listing 4.3). At this point, all necessary network components have been created. The next step is the routing configuration. The virtual Ethernet is set up to be the default gateway and NAT is set up for the subnet of the bridge within the namespace. This subnet is always set to 192.168.1.1/24 and the Unikernels address is always set to 192.168.1.2. With this configuration, the requirement for any kind of DHCP interaction that might delay execution is removed. However, this also requires the network configuration to be set up in the Unikernel. Otherwise, if a Unikernel with a different configuration were used, QEMU would start normally, but no network connectivity would be possible on the side of the Unikernel. The NAT configuration is set to forward everything to the TAP device QEMU will use. One part the Unikernel and the Container networking have in common is the port forwarding. Since the Unikernel is executed within a namespace, the port forwarding can be configured in the same way. Once a connection on one of the ports matching with the forwarding is incoming, the packets would be redirected into the namespace. Here, the packets would be affected by the NAT configuration. The NAT is set up to forward any packets to the Unikernel. In the other direction, the routing is configured to NAT the packets coming from the Unikernel to fit the assigned IP address from the NetManager. Thus, enabling the Unikernel to communicate with other microservices.

In case a service is to be undeployed, the Unikernel is expected to be already stopped, which is guaranteed by the Node, and the previously created namespace and veth pair are deleted. This deletes all other network devices and configurations done to them within the namespace. Further, the port configuration is again done in the same way it is done with Containers since the required configurations are shared between them.

While this approach enables the Unikernels to have a similar network setup to the Containers, the overall complexity increases due to the need for additional virtual network devices. Further, bridging and the configuration of NAT within the namespace increase the routing complexity for the kernel and could lower overall throughput for the Unikernel while increasing CPU load.

4.2.3. Cluster Orchestrator

The following details how changes were made to accommodate Unikernels in the scheduling decision described in subsection 2.3.2. The general process does not change for Unikernel deployment. The changes regarding scheduling are contained within the Cluster Schedulers selection process. At the point of selection, after the constraints have already been applied, instead of only comparing hardware availability, the requested virtualization technology is matched, which is made possible by the changes to the Worker Node in subsection 4.2.1, enabling the advertising of Unikernel support. Only if a Node fulfills the requirement of being able to deploy Unikernels is that Node considered for deployment. Further, if the Node supports Unikernels, the available architectures of the microservice are matched against the Node's architecture. At that point, the process of scheduling and deploying the microservice is the same for Containers and Unikernels. This means the scheduling process selects one of the Nodes based on the previous decision and sends the results back to the Cluster Manager. The compatibility selection process for the initial scheduling decision can be seen in Figure 4.4.

Another area of change to the Cluster Orchestrator is the aggregation of Node resources. The aggregation is responsible for abstracting the Nodes away from the Root Orchestrator, which will only be presented with the overview of available resources available in a cluster. The aggregated information contains the amount of memory, CPU cores, and GPUs the cluster has available. Further, the aggregation includes the supported virtualization within the cluster. The aggregated information is based on the updates received by the Nodes via MQTT. The information is aggregated and sent to the Root Orchestrator via HTTP at regular intervals.

This way of aggregation has a few problems when it comes to Unikernel deployment. While the cluster may support the scheduling of Unikernels, the aggregated resources only show the overview of all resources within the cluster. The aggregation does not indicate which resources are capable of deploying Unikernel nor do they indicate what the target architecture is. This limited overview of the resources could lead to Unikernels being deployed onto clusters in which the resource requirements are not met or the available Nodes do not support Unikernel deployment, leading to undeployed services within a cluster and the overall system.

The following details what changes were made to ensure the correct scheduling of Unikernels. The collection of Node information via MQTT is not affected by the change as the Nodes

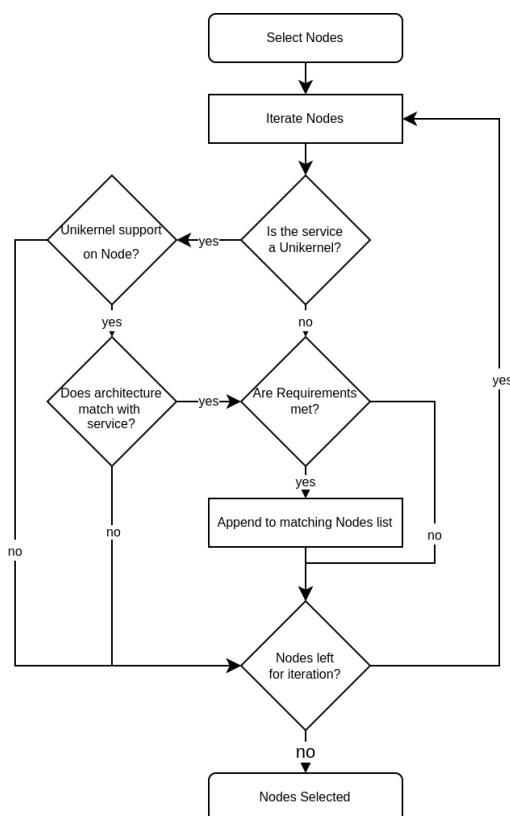


Figure 4.4.: Cluster Node Selection

already provide the necessary information with the changes made to them in subsection 4.2.1. The aggregation process has been extended to not only include the overview of all resources but also aggregate based on architecture provided by the Nodes. This aggregation process goes over all Nodes and collects all available architectures and resources available for them, resulting in a JSON document that can look like what can be seen in Listing 4.7. This new document is sent inside the general aggregation of resources to the Root Orchestrator via HTTP. Additionally, the general overview of the cluster has been extended with a list of containing architectures.

The combined changes enable the Cluster Orchestrator to schedule the incoming requests from the Root Orchestrator as well as properly inform the Root Orchestrator about the required information to enable the cluster scheduling process.

4.2.4. Root Orchestrator

The problems with Unikernel deployment on the Root Orchestrator are similar to the ones for the Cluster Orchestrator in regard to the scheduling decisions to be made. The scheduling process does not take into account how the resources are distributed in a cluster. While the cluster can have the ability to run Unikernels, the aggregated information does not indicate

Listing 4.7: Aggregation Format Per Architecture

```
{
  "amd64":{
    "cpu_percent":"<CPU usage in Percent>",
    "cpu_cores":"<CPU cores>",
    "memory":"<Available Memory>",
    "memory_in_mb":"<Available Memory in MB>"
  },
  "arm64":{
    "cpu_percent":"<CPU usage in Percent>",
    "cpu_cores":"<CPU cores>",
    "memory":"<Available Memory>",
    "memory_in_mb":"<Available Memory in MB>"
  }
}
```

what the actual resource distribution is. This can lead to a scheduling decision in which the Unikernel is forwarded to a cluster where the available hardware is insufficient to run the requested Unikernel, due to the cluster only sending the information that at least one of the systems within it supports Unikernels. Another problem is the architecture of the system. The cluster abstracts away the hardware from the Root Orchestrator in such a way that there is no indication to which architecture the resources belong. In the case of the architecture not being handled, the Unikernel might be forwarded to a cluster without the required hardware to run the Unikernel. While it would still be possible to execute the Unikernel on such a system, the emulation overhead would lead to a significant performance decrease. Thus, only matching architectures should be considered when scheduling the clusters.

The following examines the changes made to the Root Orchestrator to enable the scheduling and processing of Unikernels. The scheduling needs to be aware of the architectures available in a cluster. The required information has already been extended to the aggregation result with the modification made to the Cluster Orchestrator in subsection 4.2.3. The following Listing 4.8 shows a possible cluster aggregation as it is received by the Root Orchestrator. The JSON document received by the Root Orchestrator from each cluster contains the general information about the cluster hardware from lines 2 to 16. However, this has been extended to include the architecture information as well. In the case of the example, from lines 14 to 24, there is only one Node in the cluster. This Node also supports both Unikernels and Containers indicated by the virtualization entries. Since the cluster only contains one Node, the aggregation of the overall Nodes and the one for the architecture sorted one is the same. This would differ, for example, if another Node were added into the cluster with the same architecture but without Unikernel support. Then the resulting information for the architecture would only include the initial Node's hardware. If a Node with a different architecture with Unikernel support were added, the information of the per architecture

Listing 4.8: Aggregation Information from Cluster

```
1 {
2     "cpu_percent": 0.9,
3     "memory_percent": 38.766353153520356,
4     "cpu_cores": 12,
5     "cumulative_memory_in_mb": 7627,
6     "gpu_cores": 0,
7     "gpu_percent": 0,
8     "number_of_nodes": 1,
9     "jobs": [],
10    "virtualization": [
11    "docker",
12    "unikernel"
13    ],
14    "arch": [
15    "amd64"
16    ],
17    "aggregation_per_architecture": {
18        "amd64": {
19            "cpu_percent": 0.9,
20            "cpu_cores": 12,
21            "memory": 38.766353153520356,
22            "memory_in_mb": 7627
23        }
24    },
25    "more": 0
26 }
```

aggregation would be extended to show the resource for the newly available hardware. With this, the Root Orchestrator has the ability to differentiate between different architectures without needing to know the exact topology.

Furthermore, this ability to distinguish the architectures is used to extend the matching process to find fitting clusters and thus support Unikernels in the following way. The process of the Cloud Scheduler still goes over all clusters and checks for the hardware requirements in case of Containers. In case of a Unikernel being scheduled, a different branch is taken. At first, the architectures the microservice supports are extracted. The matching process then goes over each cluster's architectures and extracts the information about the cluster resources for the architecture, which were send because of the previously mentioned modifications. After each extraction the hardware of the orchestration is matched with the requirements of the microservice. If the requirements are met, the cluster is added to the candidates for the following scheduling process. The process goes only over the available architectures as

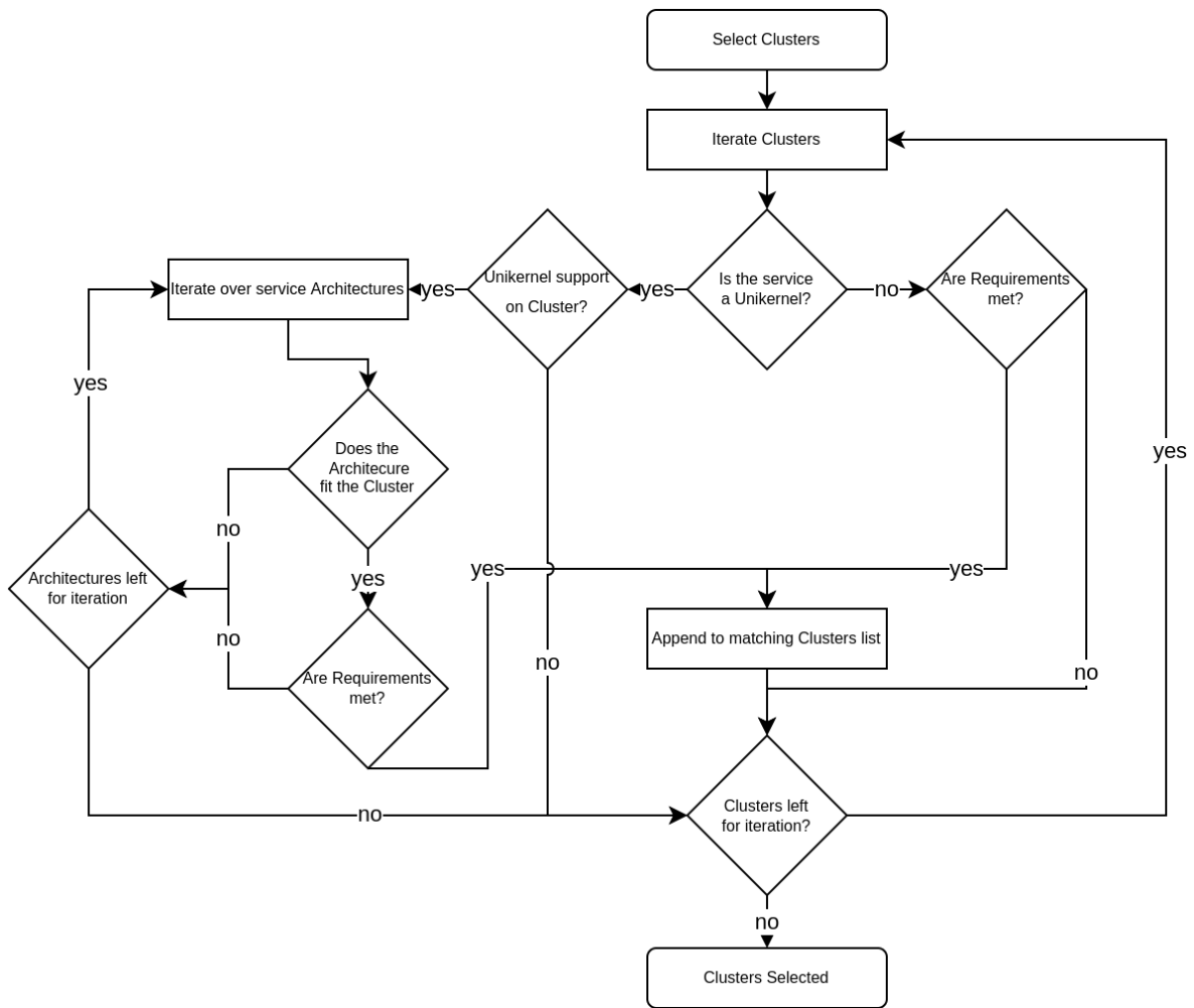


Figure 4.5.: Root Cluster Selection

long as no fit is found. Otherwise the cluster is directly appended to the candidates for the deployment. If no match is found, the cluster is not further considered for the scheduling process for the current scheduling decision. It is also to note that the aggregation on the basis of architecture does only occur in case the Nodes on the cluster side does support Unikernels. Otherwise, the cluster is skipped in the scheduling process. This guarantees that all the resources for any given architecture in the more specialized aggregation can deploy Unikernels. The decision process of the selection can be seen in Figure 4.5. The last step in the scheduling process before sending the result back to the System Manager and forwarding it to the chosen cluster is to get the best match between clusters and required hardware. For this, the cluster can not just be iterated through like it is done in the case of Containers but the microservice’s requirements need to be considered. This is achieved by iteration over all clusters and matching the cluster’s architecture specific resources with the microservices requirements to find the best match. These changes combined enable the Root Orchestrator

to correctly schedule Unikernels for the clusters.

4.2.5. Deployment Descriptor

The deployment descriptor is the basis for the registration of services with Oakestra. Listing 4.9 shows a deployment descriptor for the simple case of the Aggregation service from section 3.3 within Oakestra. The descriptor can contain multiple services, each containing what is called microservices representing the application's actual component. The microservices are the parts that are actually scheduled to the Nodes individually. Each application has a fully qualified name which in Oakestra means the application name, the application ID and the application namespace in lines 7, 6 and 8, respectively. In turn, each microservice has a fully qualified service name which consists of the name of the microservice and the microservice's namespace, which are depicted in lines 13 and 14. The ID in line 6 on the initial upload is left empty and filled by Oakestra. This ID can be used to update the services later on. The microservice contains further information about the deployment requirement and configurations. In it, the hardware resources required by the service to be deployed are laid out. In the case of the example, the container requires 100 MB of memory and one core to run based on lines 20 and 21. Further, the network configuration is supplied. This contains the port forwarding in line 29 and internal IP address to be accessible by other microservices in line 31. The actual container is specified in the code segment of each microservice. For example, line 27 specifies a container in the docker registry. In each microservice's deployment process, the container is executed with the command specified in the *cmd* array seen in line 16 and following.

The deployment descriptor already has a field for the virtualization, which is used to differentiate between different types of virtualization. This field, which can be seen in line 15, can be used for the initial support of Unikernels. While this allowed to specify the virtualization, it does not contain the necessary information to schedule or deploy a Unikernel because it does not contain the information needed to make the required decisions. The elements required are the architecture of the Unikernel, which is defined by the microservice, and its external location. External location here means where to acquire the Unikernel from, which in the case of the current implementation is from an HTTP server. Unlike Containers, Unikernels do not have a standardized way in terms of structure or delivery.

Thus, it is required to extend the deployment descriptor to support Unikernels. The supported architectures are known for all microservices beforehand. This means every microservice can be configured to specify which architecture the Unikernels targets. The microservice definition has been extended with an *arch* field containing an array of architectures for each microservice in case the virtualization platform is the Unikernel one. This array contains all architectures the microservice supports. An example of this can be seen in Listing 4.10. With this array, the scheduling is able to know how the Unikernel microservice is supposed to be scheduled. The number of architectures in this array is not limited, but for each entry, a valid Unikernel must be available. Since there is no standardized way to access Unikernels, the current method to access them is in the format described in subsection 4.2.1. The Unikernel URLs are stored in the same entry as the container registry would be stored.

Listing 4.9: Deployment Descriptor

```
1 {
2   "sla_version": "v2.0",
3   "customerID": "Admin",
4   "applications": [
5     {
6       "applicationID": "",
7       "application_name": "aggr",
8       "application_namespace": "aggr",
9       "application_desc": "Video Aggregation",
10      "microservices": [
11        {
12          "microserviceID": "",
13          "microservice_name": "testaggr",
14          "microservice_namespace": "aggr",
15          "virtualization": "container",
16          "cmd": [ "./aggr", "--video-source", "mot20-01.mp4",
17            "--detector-address", "192.168.1.173",
18            "--tracker-address", "192.168.1.173"
19          ],
20          "memory": 100,
21          "vcpus": 1,
22          "vgpus": 0,
23          "vtpus": 0,
24          "bandwidth_in": 0,
25          "bandwidth_out": 0,
26          "storage": 0,
27          "code": "docker.io/psabanic/caggregation:arm",
28          "state": "",
29          "port": "",
30          "addresses": {
31            "rr_ip": "10.30.30.30"
32          }
33          "added_files": []
34        }
35      ]
36    }
37  ]
38 }
```

Listing 4.10: Architecture Array

```
"arch": [  
    "amd64",  
    "arm64"  
],
```

Listing 4.11: Unikernel URL

```
"code": "<URL for amd64>,<URL for arm64>",
```

This is possible because the field is otherwise unused for Unikernels. The code contains one URL for each architecture specified in the `arch` array in a comma-separated string, as seen in Listing 4.11. The `cmd` array, in line 16 and following, for each microservice can also be reused for the Unikernel. Unlike a Container, the application does not need to be specified since, in the case of a Unikernel, it is itself the application. Other than that, it can be supplied with the same format of arguments as a Container microservice. Another way to deal with the difference in configuration would be for the developer to ignore the first argument in the `cmd` array. The first argument is the name of the application to be run in the Container, which is not needed for the Unikernel. These changes enable the specification of Unikernel microservices in the deployment descriptor with all required information for the Root and Cluster Orchestrator to schedule and for the Node to deploy the Unikernel.

5. Evaluation

This chapter aims to evaluate the advantages and disadvantages of Unikernel usage, in general, and in the Oakestra framework. For this purpose, the ported service from section 3.3 and others are being used to evaluate the deployment of Unikernels compared to a container service with the same functionality. Further, the capability of the container runtime presented in chapter 4 is evaluated in the test setup described in the following.

5.1. Test Setup

The test environment used in this thesis consists of 7 Nodes, 1 Router and 1 Switch. The Nodes are made up of systems with different hardware architectures and capabilities, seen in Table 5.1. All tests were run on a single cluster with one Root and one Cluster Orchestrator, both running on the system labeled PC in Table 5.1. The same node running the Orchestrators is used to collect data from the tasks run on the cluster and to execute benchmarks.

5.2. Evaluation Results

The implementation discussed in chapter 4 is now evaluated in this section. Parts of the evaluation process use the application described in section 3.3.2. The evaluation focuses on comparing the performance of Unikernels and Containers in the Oakestra framework, with the runtime proposed in section 4.2, but also, in general, to see the difference in the usage of both virtualization technologies.

5.2.1. Unikernel Measuring

There is a multitude of factors while measuring the performance of Unikernels. While most of these are comparable with ones for Containers, including CPU usage and Memory utilization,

Table 5.1.: Test System Overview

System	Architecture	CPU	Memory	OS
2x Raspberry Pi 3	AArch64	ARM Cortex-A53	1G	Raspberry Pi OS
2x Raspberry Pi 4	AArch64	ARM Cortex-A72	4G	Raspberry Pi OS
1x Fujitsu Esprimo G558	AMD64	Intel Core i5-8400T	8G	Ubuntu 20.04 LTS
1x UDOO x86	AMD64	Intel Atom x5-E8000	2G	openSUSE Tumbleweed
1x PC	AMD64	Intel Core i7-8086K	32G	Arch Linux

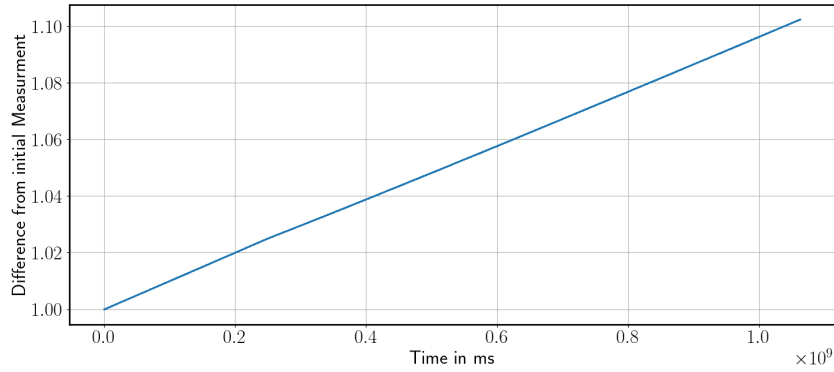
the Unikernels had a additional challenge do measure curtain aspects of the finer details. Unlike a Container or a Linux virtual machine, the Unikernel has far fewer facilitates to conduct a more fine-grained analysis of the Unikernel’s inner workings. Part of that has to do with the Unikernel running in a virtual environment but also that a Unikernel has no system calls to be measured, nor does a Unikernel come equipped with analysis tools.

Another aspect to consider regarding virtual machines is the problem of measuring time stamps with the intent to compare them with the host system. The host system and the Unikernel are not necessarily in sync. At initialization the BIOS within QEMU gives the Unikernel the current time, which only has a precise of about one second, as seen in subsection 3.3.2. This inaccuracy leads to precise measurements having large offsets close to a second in time, even if the expected result should have been in a millisecond range. Further, even if the initial time difference is accounted for, the Unikernel can still experience time drift because of various factors within the virtualized environment and the Unikernel itself. This can be seen in Figure 5.1a. For this test, a Unikraft kernel was used as a simple application to measure the time difference with the host. The y-axis shows the percentile difference between of the initial offset the Unikernel’s time and the host system’s time. The measurements were taken over a period of 17 minutes and they show an increased time drift over time. From an initial offset of $414888\mu\text{s}$ to an offset of $457369\mu\text{s}$ in the last measurement. This leads to a total shift in time for the Unikernel and Host of 42.481ms. Further, this problem is not constant, as seen in Figure 5.1b. Overall this change in the time is hard to account for when doing precise measurement involving the host and the Unikernel. However, this does not affect external measurements taken by the host without any input from the Unikernel and has only a marginal effect on internal measurements taken over short and long time frames.

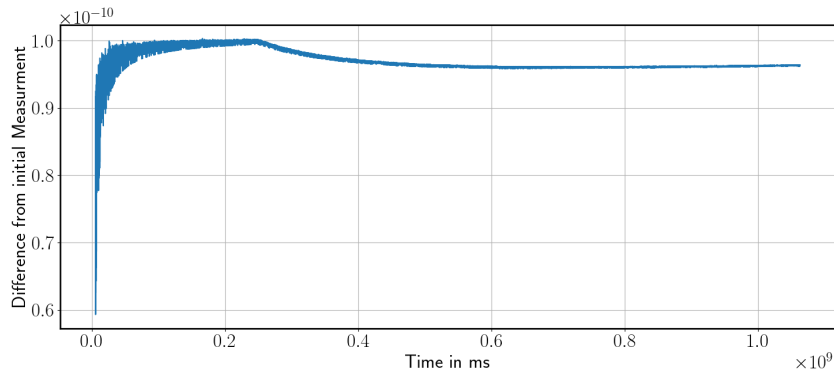
5.2.2. Aggregation Comparison

In the process of porting the Aggregation service, which is an essential part of the benchmarking and testing for Oakestra’s Unikernel support, the application changed significantly. This is due to the constraints and design decisions required for current Unikernel support. While some aspects of the original Python service and the new service are kept the same, others have changed. Those changes include the language, which leads to a significant change in the performance of the overall service. This change can be seen in Figure 5.2, which shows the difference in execution time for the encoding of a frame and the processing time. The processing time is the time the service requires to execute the send process for a frame to either of the following services in the chain. In the case of the old implementation, this does happen via gRPC calls and in the new one, over a TCP connection. This alone would reduce the required time it takes. However, the implementation of gRPC, used in the Python service, is also entirely written in Python, which leads to a more noticeable performance difference.

Moreover, the encoding time for both implementations differs greatly. The Python version uses OpenCV, which uses Ffmpeg internally, to do the video processing, while the new service uses Ffmpeg directly. This layer of abstraction could explain some of the performance increases of the C service. Further, like with the processing time, the overhead of the Python language, which is an interpreted language unlike C, could make up a part of the performance



(a) Unikraft Time Drift



(b) Time Drift Slope

Figure 5.1.: Unikernel Time Measurements

difference. Overall, both factors should affect the performance of the service leading to the new service performing better than the old one.

5.2.3. Resource Utilization

One of the essential benefits possibly gained by using Unikernel over Containers in an edge deployment is the possibility of increased performance compared to Containers. With the port of the Aggregation service, there is a comparable application enabling the comparison between both technologies. The Container application is built with the same FFmpeg base configuration without additional modification applied. This means non of the Unikraft specific changes to the headers have been made. Both are linked statically to the service's binary. As seen in subsection 5.2.2, the Python implementation is not comparable to the new service, which is why the focus of the comparison is directed at the new one. To minimize the influence from the over parts of the service, the Aggregation is tested in isolation. This is achieved by replacing the other services of the pipeline, responsible for further processing the

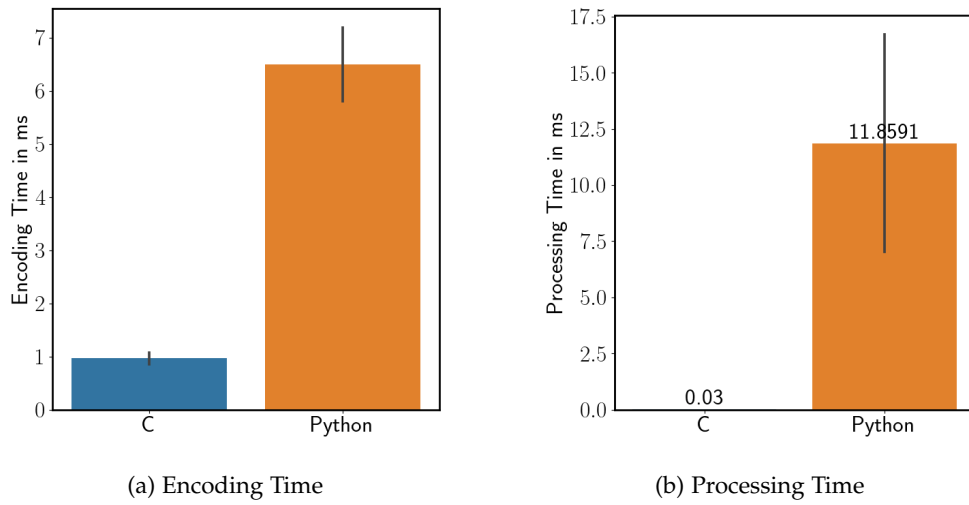


Figure 5.2.: Execution without Virtualization

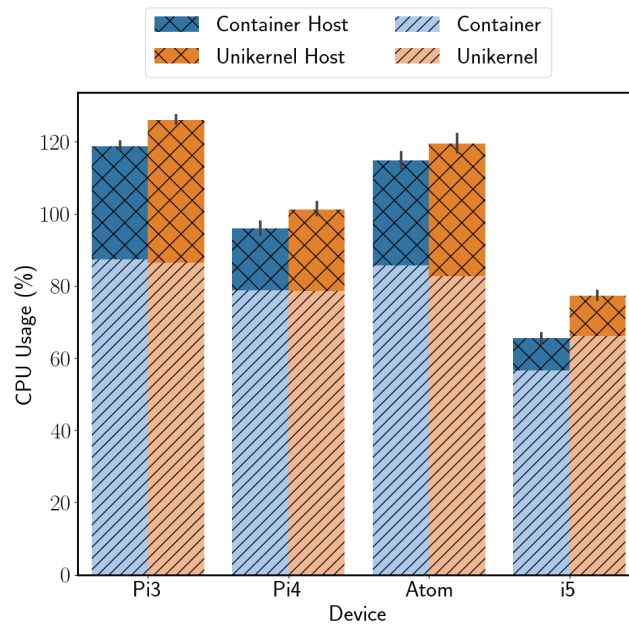


Figure 5.3.: CPU Usage over Execution Time

frames, with a simple server that does not do any further calculations on them so as not to delay the Aggregations service.

The layout of the services of the application is depicted in Figure 3.2. The video source used in all streaming tests was run on the same system and all other systems were used for testing. Running the service on all the systems separately and not at the same time yields

what can be seen in Figure 5.3. In the Figure, the top indicates the average of the total system CPU usage, while the fainter colored parts of the bars indicate the CPU usage of the actual service. This is the process running within the Container in case of the Container deployment and the entire QEMU process in case of the Unikernel deployment. The usage is scaled to the utilization of one CPU core for each system, meaning that the overall system utilization is a multiple of 100%, e.g., on a system with four threads, it would be 400% at maximum utilization.

With this average utilization, it can be seen that, while in most cases causing a lower direct CPU load, the Unikernel overall has a higher impact on the entire system's CPU usage than the Container execution. The higher CPU load can be explained by the Unikernels overhead in network communication. While the Container uses the same kernel, the Unikernel has its own separate network stack running independently of the host system. The Unikernel tasks QEMU and the host system, in general, to do additional computation and routing decisions for the Unikernel to transfer the network traffic from the Unikernel environment to the host system. Further, in the Oakestra networking structure, the Unikernel is even more impacted because of additional networking required for the network overlay to be compatible. While this might seem unfavorable to the Unikernel, it is not the only important metric for the service.

A different picture can be seen when more measurements are taken from within the processes. In Figure 5.4a and Figure 5.4b, both the encoding time and the processing time are lower for the Unikernel. This could be because of the already mentioned benefit of a Unikernel not requiring context switch which could help FFmpeg with memory allocations required for image processing. While both of the Figures indicate a lower time in the actual application part of the service, it does not show the whole picture when it comes to Unikernels. Both the Unikernel and the Container need to communicate with other services to send and receive the video frames. While the Container process itself is only concerned with the execution of the actual aggregation process, the Unikernel is required in addition to that to manage the network communication too. This additional processing should introduce a time increase from the parts of the Unikernel processing that are held back because other parts of the Unikernel need to be processed. With this consideration in Figure 5.4c, it can be seen that the execution time of both the Unikernel and the Container are essentially the same in all scenarios, with the Unikernel being slightly faster on the more powerful system and being tied for the weakest system. The execution time for this test has been measured by the server receiving the frames and is the time difference between the first and last frames.

Overall, the Aggregation service does not show significant performance gains compared to the increased CPU usage the Unikernel requires. However, the Aggregation service is one of many services which could be used in Unikernel form.

5.2.4. Throughput

In the Aggregation service evaluated in subsection 5.2.3, the execution is both network and CPU bound, limiting the capabilities of a single-threaded Unikernel execution to some extent. To further evaluate the use cases of Unikernels, it is thus required to look at services that are

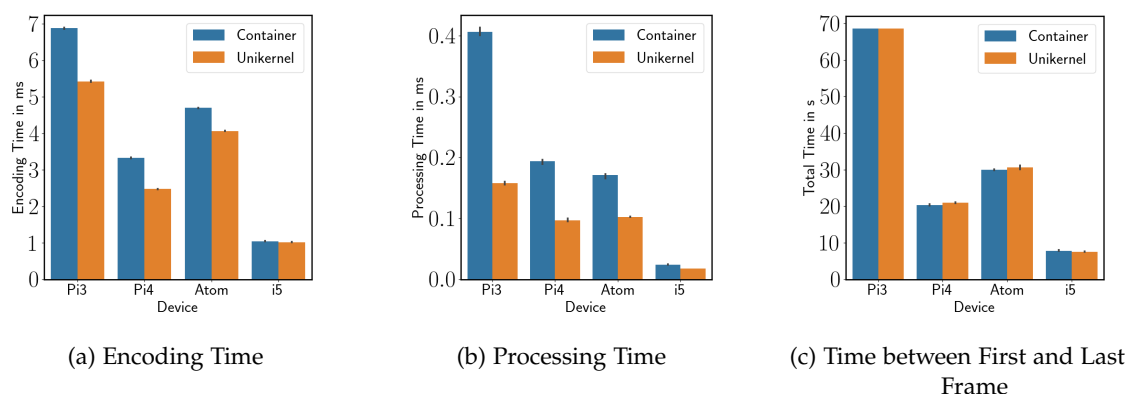


Figure 5.4.: Unikernel and Container Metrics

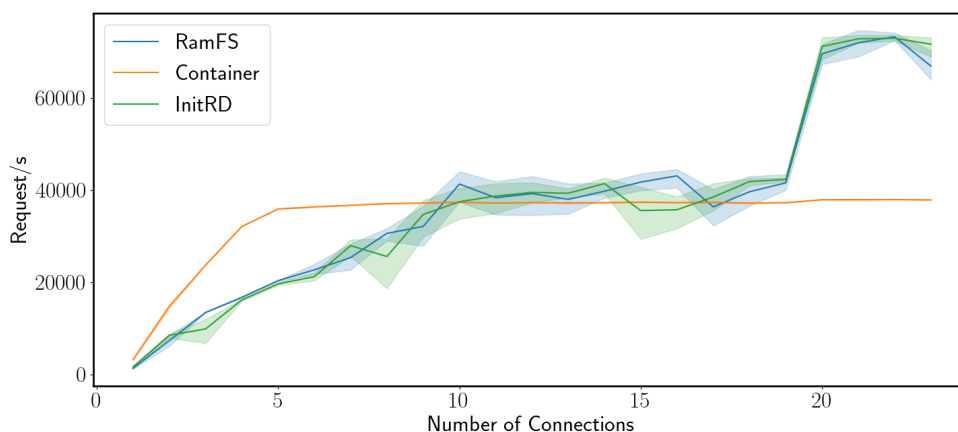


Figure 5.5.: Nginx HTTP Request/s

less CPU bound and thus might benefit more from the Unikernel deployment without the additional overhead due to networking.

When looking at the performance of an application like nginx [65], the increase in single thread performance can be seen in Figure 5.5. The Figure shows the request per second for the Container and Unikernel versions of nginx. Nginx is configured the same way in both cases with one worker thread and both containing the same document to be delivered. The measurements were taken with the wrk benchmarking tool [66]. In the case of Unikernels, the test is split in two, one is an initial RAM disk which is loaded by QEMU at startup and the other is a simple filesystem mapped by Unikraft into memory. The files of this memory filesystem are part of the executable, while in the case of initrd, they are separate and require QEMU to load it at startup. Both Unikernel versions show very similar behavior and are comparable for this test.

When comparing the performance of both the Unikernel version and the Container version,

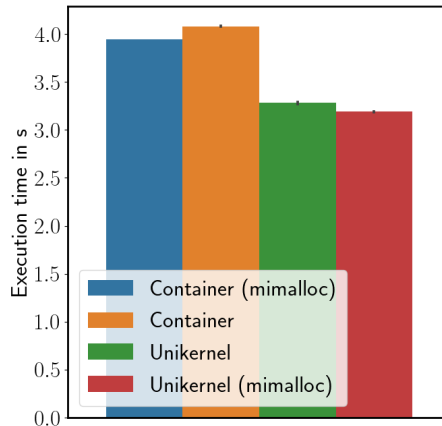


Figure 5.6.: Python Application Execution Time

it can be seen that while the Container quickly reaches its peak throughput, the Unikernel has a more linear increase and reaches the same level at ten connections. Further, the Unikernels performance drastically increases at 20 parallel connections and far outperforms the Container version. The reasons for this could be manifold. One element of it could be the network library used with Unikraft, which in this case is *lwip*. Further, it could be caused by the nature of Unikernels by mitigating system calls or, in part, due to the way scheduling with a cooperative schedule requires less locking for the kernel components.

While *nginx* is a service strongly reliant on networking, another possibility is that a service contains a more CPU-bound task. When looking at such a task, in this case, a Python application which does numerous calculation with dictionaries, the results indicate an improvement for such applications which can be seen in Figure 5.6. The application consists of the initialization part and the calculation part. The initialization part is filling two matrices, each 100 by 100 in size, with random values. The calculation part is the part measured in the test and repeats the dot product over the matrices 1000 times resulting in the times. This process is repeated multiple times leading to the results seen in Figure 5.6. In addition to the difference between the two deployment types, the memory allocator was also changed for both. While different allocator change the runtime behavior of the application overall, the Unikernel is still faster when it comes to this part. Taking the best versions for both cases, which is, in this test, the version using *mimalloc*, leads to a visible performance improvement of about 20% for the Unikernel. This might be because of the single address space of the Unikernel. Moreover, this performance increase can be more clearly tied to the design of Unikernels because of the lack of networking in the python application.

This section shows that there is a use case for Unikernels in which they outperform their Container counterparts for certain tasks.

5.2.5. File Size

On the topic of file size, Unikernels should be able to produce smaller sized compared to Containers because a Unikernel only contains the minimal viable library content to run a service. The size comparisons of the compressed images for the Unikernel can be seen in Table 5.2, in which a clear difference between the file sizes of both compressed formats can be seen. The Unikernel is in the format the Unikernel runtime described in chapter 4 is expecting and the Container is a standard container based on the OCI [12] specifications.

The Unikernel is generally always smaller than the Container counterpart, even in cases in which additional data is required, e.g., in the case of Nginx and Python. Both require additional files in the form of configurations and files to be delivered and in the case of Python, the standard library which is required for most of Python's functionality. This is primarily due to the Container requiring additional libraries and system dependence to function, while the Unikernel either removes those dependencies by the difference in the Unikernels runtime or only has the minimum required functionality included in the binary itself.

5.2.6. Deployment

Not only is the performance of a running Unikernel important, but also the overhead the Unikernel creation causes for the deployment in Oakestra is essential. In Figure 5.7, it can be seen that in the case of no network being used for the Unikernel, that the Unikernel can be created quickly. In such a case, the network overlay is not used, reducing the overall deployment time for both runtimes. This difference comes mainly from a lower overhead when creating the virtual machine compared to a container runtime. The QEMU process can directly load the Unikernel, while the container engine needs more preprocessing in the creation process.

On the other hand, when looking at the enabled networking overlay, a significant difference can be seen. Here, both runtimes require more time due to the additional communication with the NetManager. However, the Unikernel takes longer for the initialization process. The lower part of the bars, which are fainter in color, shows how long the Node's communication with the NetManager, with the additions made in subsection 4.2.2, takes. This time includes the request for the network, the creation of all necessary components for either Container or Unikernel and the answer back to the Node. Both times are on a very similar scale for both virtualization technologies. This indicates that the additional steps required for the Unikernel network setup only add minimal overhead to the creation of the network.

Table 5.2.: Unikernel and Container Size Comparison

Platform \ Service	Unikernel	Container	%
Nginx	0.75MB	7.38MB	10.1
Aggregation Service	7.8MB	13.17MB	59.2
Python	5.3MB	32.27MB	16.4

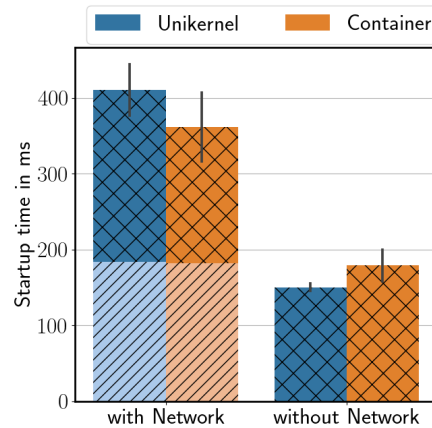


Figure 5.7.: Startup Time

Considering the time for the networking shows that the base Container creation time has stayed the same in the process. This can be explained by the fact that the Container is still running in the same environment. From the Container's perspective, only an additional network interface has been attached while the NetManager modified it. Otherwise, the Container stays the same introducing no additional overhead to the initialization apart from the NetManager. However, on the Unikernel side, in addition to the increase of the initialization delay because of the NetManager, the general initialization time has also increased. This is primarily on because of the initialization process of QEMU. Whenever the network overlay is active, the Node adds a network device of the type virtio-net to the Unikernel. This inclusion requires additional initialization for the paravirtualization and the virtual environment QEMU emulates, thus, increasing the time it takes for the QEMU process to start the Unikernel.

6. Conclusion

6.1. Conclusion

The scheduling behavior and the virtualization runtime for Unikernels, which were introduced in this thesis, enable the usage and deployment of Unikernel in a heterogeneous edge environment. By letting the Nodes advertise their architecture, the scheduling is able to operate on an arbitrary amount of different architectures. Moreover, schedule them correctly based on the information aggregated from the clusters. While also not interfering with the scheduling or execution of services for other runtimes, such as the Container runtime. Moreover, allowing for the scheduling to stay as extensible as possible and enable future changes to it. Further, advertising the supported runtime allows the Nodes to opt out of Unikernel support if they do not have the runtime requirement to deploy Unikernels efficiently. With the extension to the NetManager, seamless communication between all kinds of services is possible, regardless of runtime. Meaning that both Unikernel and Containers can communicate freely within the overlay network. Further, with the port of a preexisting container application and other applications to Unikraft, a Unikernel platform currently under heavy development, a comparison was made. With the help of these services advantages and disadvantages of Unikernel deployment were explored. Finally, possible reasons were discussed and improvements in performance with the deployment of Unikernels have been explored. This has shown that while a Unikernel can have benefits, it depends heavily on the kind of application. In some cases, the Unikernel performs worse than the Container version of the application and far outperforms the Container in others. With Unikernels performing bettering with more focused services, seen in subsection 5.2.4. Such services can be either be network reliant, e.g. nginx, or be more reliant on the CPU and does less networking. With both requirements at the same time, as seen with the Aggregations service in subsection 5.2.3, the benefits of Unikernels diminish. Further, based on the Aggregation service, it can be seen that the host system's architecture does not seem to impact the Unikernel significantly, with the Unikernel, in some cases, slightly performing better than the containerized version on more capable hardware.

6.2. Limitations and Future Work

With the tests described in the evaluation chapter 5 and their analysis, we have found limitations in the Unikernel runtime and, generally, in Unikernels.

The current runtime version has a less user-friendly service specification in regard to the imaging of Unikernels. The images are single architecture, and each one needs to be specified

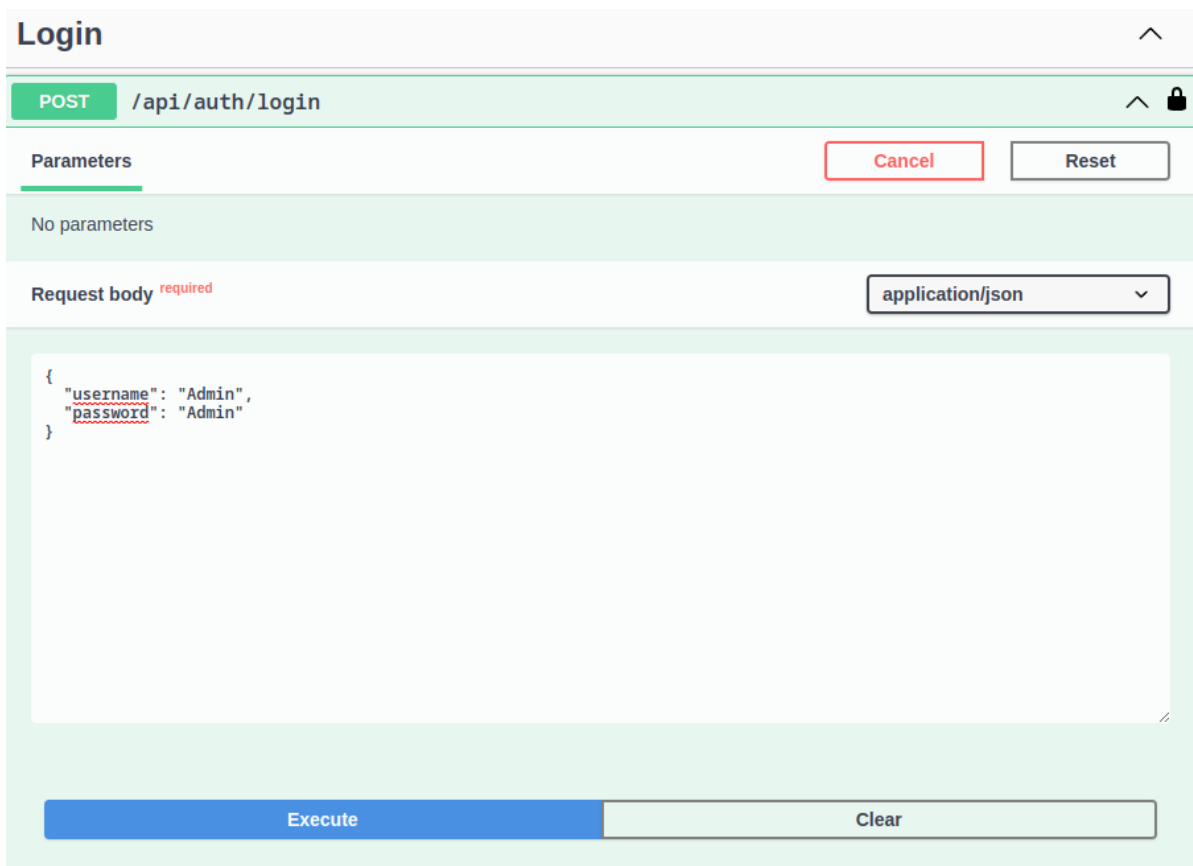
separately, unlike Containers, where this is handled by the image format specified by OCI. This could, in the future, be resolved by using the same format as a basis and using the layers and its metadata to store the Unikernel, additional files and the necessary meta information to run the Unikernel. This could, in the future, be resolved by using the same format as a basis and using the layers and its metadata to store the Unikernel, additional files, and the necessary meta information to run the Unikernel. With this extension, the configuration overhead for the end user would, to some degree, be lessened, allowing the user to specify a simple link to a registry that contains all specified architectures. Also, allowing Oakestra to possibly automatically detect the supported architectures for the service.

Another limitation is the high startup time compared to the Container deployment. While this only affects the virtual machines in the case the Network overlay is used, the network is still highly relevant for most services that need to communicate to the network in any way. This in the future could be mitigated by different virtualization optimization or the possible change away from QEMU to a supposedly faster system, which might lead to a change in the deployment time.

Regarding Unikraft, there are still a lot of questions open about why some elements are improved, or some see diminished results. For example, the reason for why nginx shows such a different behavior, as seen in subsection 5.2.4, needs to be further evaluated. Because of the lack of tooling for Unikernels, the measuring of internals is a difficult task. This is leading to testing currently primarily relying on measuring external behavior. Since Unikraft is a platform still in a state of active and heavy development, the ability to evaluate the kernel in more detail is also still in development. Most likely, it will be improved upon in the future, which should make it possible to examine further the behavior of Unikernel on edge Nodes and form more general rules for their use cases and scheduling.

A. Deployment

To deploy a Unikernel service with the web interface of the Root Orchestrator, the following steps need to be taken. First, before the web interface can be used, the user needs to log in.



The screenshot shows a REST client interface for a login endpoint. The title is "Login". The method is "POST" and the path is "/api/auth/login". There are "Parameters", "Request body", and "Execute" sections. The "Request body" is set to "application/json" and contains a JSON object with "username": "Admin" and "password": "Admin".

```
POST /api/auth/login
```

Parameters Cancel Reset

No parameters

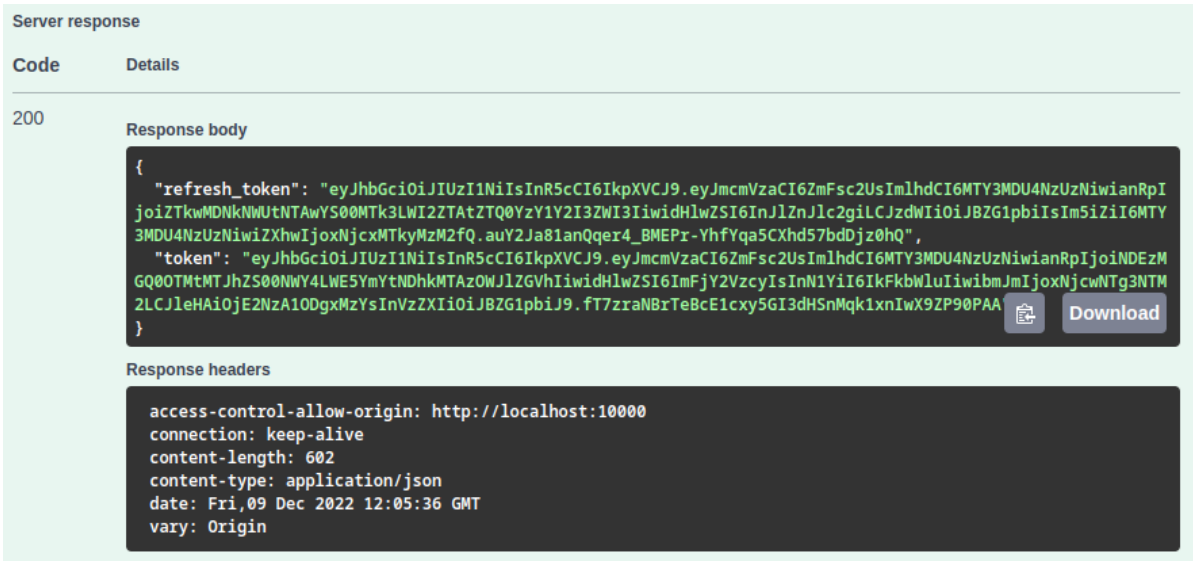
Request body required application/json

```
{  
  "username": "Admin",  
  "password": "Admin"  
}
```

Execute Clear

Figure A.1.: Service Registration

A. Deployment

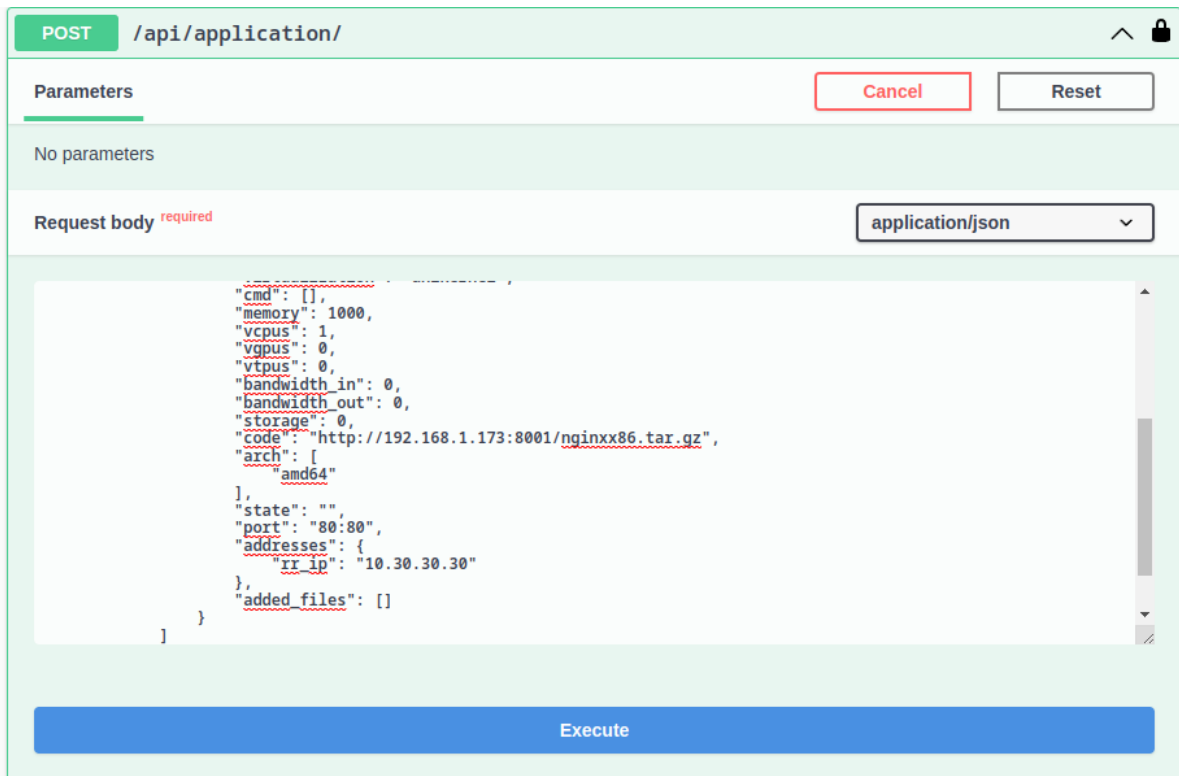


Server response

Code	Details
200	<p>Response body</p> <pre>{ "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImhhdCI6MTY3MDU4NzUzNiwiYWVzIjoiZTkwMDNkNWU0NTAwYS00MTk3LWl2ZTA4ZTQ0YzY1Y2I3ZWl3IiwiaWF0IjoiYjZG1pbiIsIm5iZiI6MTY3MDU4NzUzNiwiZXhwIjoxNjc0MTkyMzY2fQ.aUY2Ja81anQqer4_BMEPr-YhfYqa5CXhd57bdDjz0hQ", "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsImhhdCI6MTY3MDU4NzUzNiwiYWVzIjoiZTk0MTY3MDU4NzUzNiwiZXhwIjoxNjc0MTkyMzY2fQ.aUY2Ja81anQqer4_BMEPr-YhfYqa5CXhd57bdDjz0hQ" }</pre> <p>Response headers</p> <pre>access-control-allow-origin: http://localhost:10000 connection: keep-alive content-length: 602 content-type: application/json date: Fri, 09 Dec 2022 12:05:36 GMT vary: Origin</pre>

Figure A.2.: Service Registration Response

After that, the first part of deploying a Unikernel is the Deployment Descriptor, which must be registered with the Root Orchestrator.



POST /api/application/

Parameters Cancel Reset

No parameters

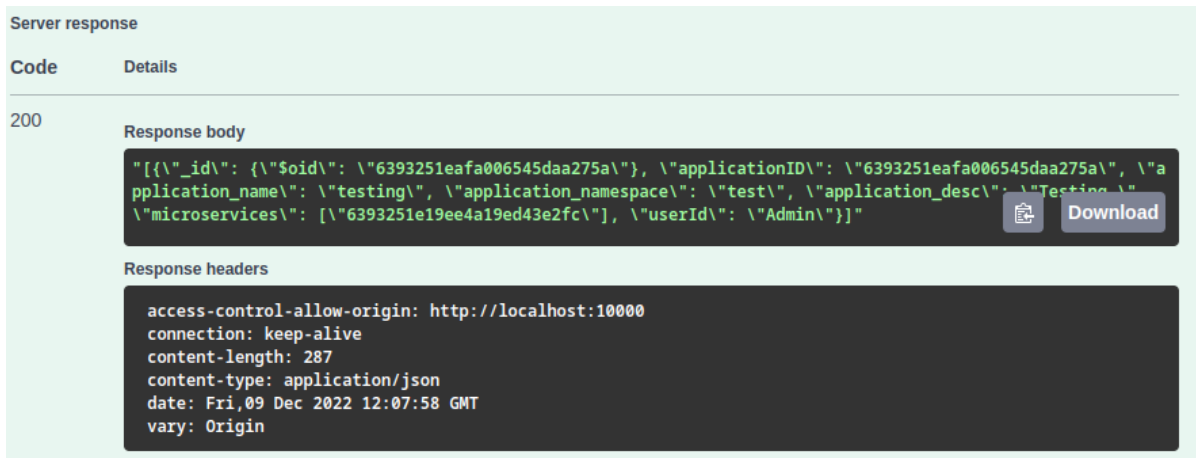
Request body ^{required} application/json

```
{
  "cmd": [],
  "memory": 1000,
  "vcpus": 1,
  "vgpus": 0,
  "vtpus": 0,
  "bandwidth_in": 0,
  "bandwidth_out": 0,
  "storage": 0,
  "code": "http://192.168.1.173:8001/nginx86.tar.gz",
  "arch": [
    "amd64"
  ],
  "state": "",
  "port": "80:80",
  "addresses": {
    "rr_ip": "10.30.30.30"
  },
  "added_files": []
}
```

Execute

Figure A.3.: Service Registration

A. Deployment

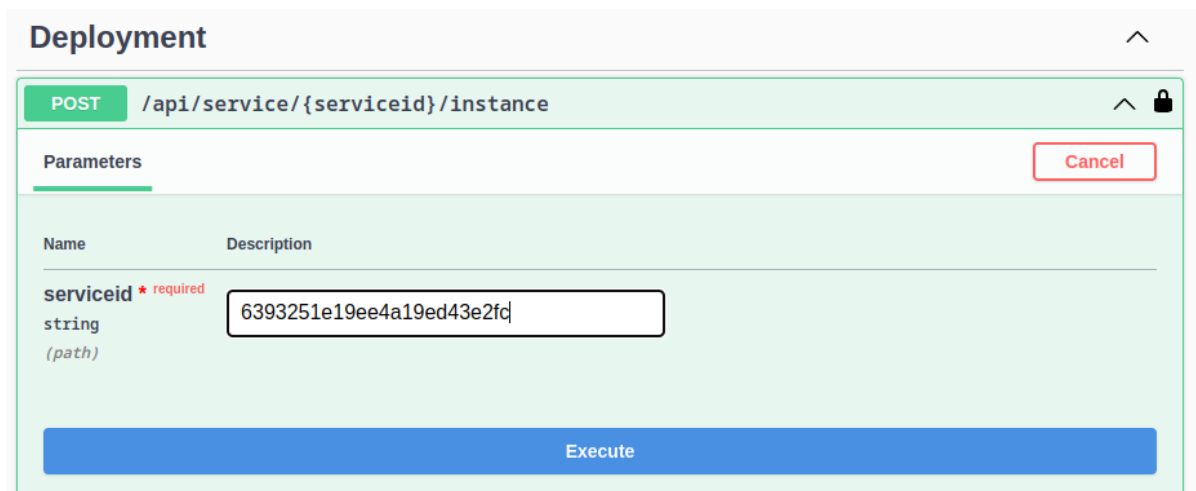


Server response

Code	Details
200	<p>Response body</p> <pre>"[{"_id": {"\$oid": "6393251eafa006545daa275a"}, "applicationID": "6393251eafa006545daa275a", "application_name": "testing", "application_namespace": "test", "application_desc": "Testing \n\n", "microservices": [{"6393251e19ee4a19ed43e2fc"}, {"user": "Admin"}]"</pre> <p>Response headers</p> <pre>access-control-allow-origin: http://localhost:10000 connection: keep-alive content-length: 287 content-type: application/json date: Fri, 09 Dec 2022 12:07:58 GMT vary: Origin</pre>

Figure A.4.: Service Registration Response

The answer to the POST contains the IDs of the microservices. Those IDs can be used to deploy them. In the case of the service used for this example, it is a single microservice running nginx as a Unikernel.



Deployment

POST /api/service/{serviceid}/instance

Parameters Cancel

Name	Description
serviceid * required string (path)	<input type="text" value="6393251e19ee4a19ed43e2fc"/>

Execute

Figure A.5.: Microservice Deployment



Figure A.6.: Microservice Deployment Response

After the Root sends back the response, it tries to schedule the microservice to a cluster, and in the end, the cluster will assign a node with the microservice. The status of the service can be checked in the following way.

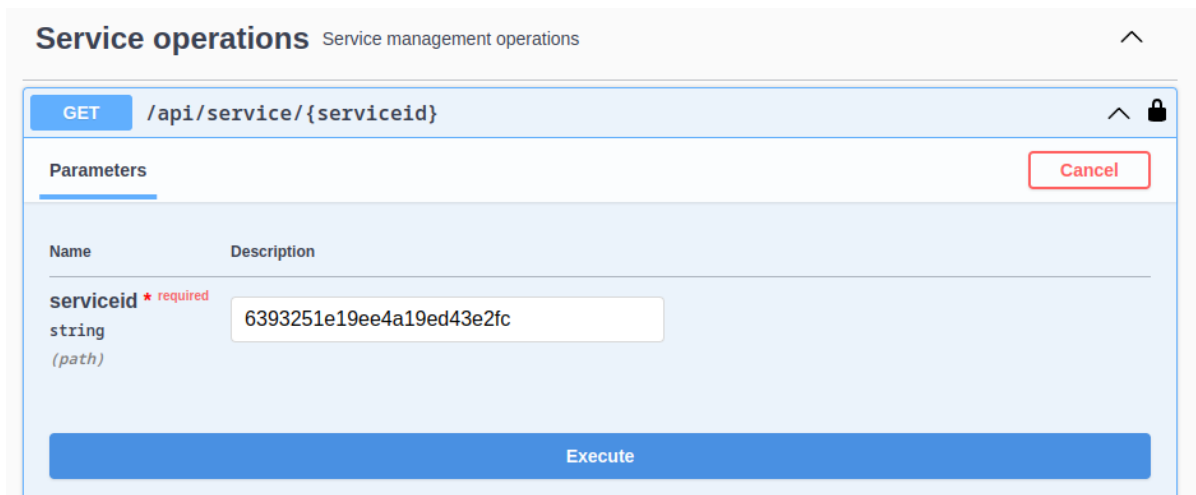


Figure A.7.: Check on Service

A. Deployment

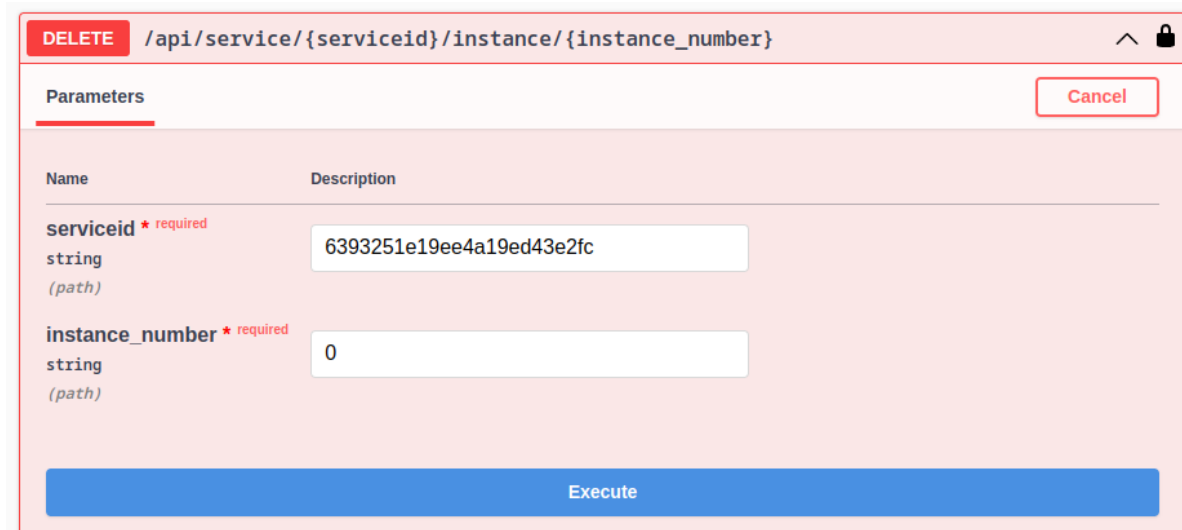


Server response

Code	Details
200	<p>Response body</p> <pre>"{"_id": {"\$oid": "6393251e19ee4a19ed43e2fc"}, "job_name": "testing.test.testaggr.test", "RR_ip": "10.30.30.30", "added_files": [], "addresses": {"rr_ip": "10.30.30.30"}, "app_name": "testing", "app_ns": "test", "applicationID": "6393251eafa006545daa275a", "arch": ["amd64"], "bandwidth_in": 0, "bandwidth_out": 0, "cmd": [], "code": "http://192.168.1.173:8001/nginx86.tar.gz", "image": "http://192.168.1.173:8001/nginx86.tar.gz", "instance_list": [{"instance_number": 0, "cluster_id": "6393243719ee4a19ed43e295", "cluster_location": "", "cpu": "0.00000", "disk": "0", "memory": "94392320.000000", "publicip": "192.168.1.176", "status": "RUNNING", "status_detail": null}], "memory": 1000, "microserviceID": "6393251e19ee4a19ed43e2fc", "microservice_name": "testaggr", "microservice_namespace": "test", "next_instance_progressive_number": 1, "port": "80:80", "service_name": "testaggr", "service_ns": "test", "state": "", "storage": 0, "vcpus": 1, "vgpus": 0, "virtualization": "unikernel", "vtpus": 0, "status": "NOT_FOUND", "status_detail": null}"</pre> <p>Response headers</p> <pre>access-control-allow-origin: * connection: keep-alive content-length: 1193 content-type: application/json date: Fri, 09 Dec 2022 12:11:16 GMT</pre>

Figure A.8.: Check on Service Response

The response contains all the information about the microservice and, in this case, shows that one instance of the service is running. This can be taken from the *instance_list* array. There it specifies the microservice with instance number 0 and shows that the deployment was successful with the status of the instance being RUNNING. With the ID, the microservice can also be undeployed in the following way.



DELETE /api/service/{serviceid}/instance/{instance_number}

Parameters Cancel

Name	Description
serviceid * required string (path)	<input type="text" value="6393251e19ee4a19ed43e2fc"/>
instance_number * required string (path)	<input type="text" value="0"/>

Execute

Figure A.9.: Undeploy

A. Deployment

The screenshot shows a REST client interface. At the top, the 'Request URL' is `http://localhost:10000/api/service/6393251e19ee4a19ed43e2fc/instance/0`. Below it, the 'Server response' is shown with a status code of 200. The 'Response body' is a JSON object: `{ "message": "ok" }`. The 'Response headers' include: `access-control-allow-origin: http://localhost:10000`, `connection: keep-alive`, `content-length: 22`, `content-type: application/json`, `date: Fri,09 Dec 2022 12:12:51 GMT`, and `vary: Origin`.

Figure A.10.: Undeploy Response

This results in the microservice instance being terminated, which can be confirmed by rechecking the ID. The response does not contain any service instances anymore.

The screenshot shows a REST client interface. The 'Server response' is shown with a status code of 200. The 'Response body' is a large JSON object containing detailed information about the microservice instance, including its ID, job name, IP address, application ID, architecture, bandwidth, command, code, image, instance list, memory, microservice ID, name, namespace, state, storage, vCPU, vGPU, uniker, and vtpus. The 'Response headers' include: `access-control-allow-origin: *`, `connection: keep-alive`, `content-length: 939`, `content-type: application/json`, and `date: Fri,09 Dec 2022 12:13:55 GMT`.

Figure A.11.: Undeploy Response

Further, without the web interface using the API directly, the same can be achieved. Here, for example, with Python, going through the same steps.

Listing A.1: Python Service Deployment

```
1 import requests
2 import json
3 from time import sleep
4
5 rooturl = "http://<Root Orchestrator Address>"
6 r = requests.post(rooturl+"/api/auth/login",json={"username": "Admin", "
    password": "Admin"})
7 token = r.json()["token"]
8 service = open("<Deployment Descriptor>","r")
9 service_json = service.read()
10 service.close()
11 service_json = json.loads(service_json)
12 Auth_token = {"Authorization" : "Bearer "+token}
13
14 #Register Deployment Descriptor
15 r = requests.post(rooturl+"/api/application",json=service_json,headers=
    Auth_token)
16 microservices = json.loads(r.json())[0]["microservices"]
17
18 #Deployment of microservice
19 r = requests.post(rooturl+"/api/service/"+microservices[0]+"/instance",headers=
    Auth_token)
20
21 sleep(5)
22
23 #Information about the microservice
24 r = requests.get(rooturl+"/api/service/"+microservices[0],headers=Auth_token)
25 j = json.loads(r.json())
26 instance_number = j["instance_list"][0]["instance_number"]
27 print(j)
28
29 sleep(5)
30
31 #Undeployment of microservice
32 r = requests.delete(rooturl+"/api/service/"+microservices[0]+"/instance/"+str(
    instance_number),headers=Auth_token)
33 r = requests.get(rooturl+"/api/service/"+microservices[0],headers=Auth_token)
34 j = json.loads(r.json())
35 print(j)
```


The following depicted the Deployment Descriptors for the Aggregations Service in both Container and Unikernel form

Listing A.2: Deployment Descriptor for the Aggregaton Service (Unikernel)

```
1 {
2   "sla_version": "v2.0",
3   "customerID": "Admin",
4   "applications": [
5     {
6       "applicationID": "",
7       "application_name": "testing",
8       "application_namespace": "test",
9       "application_desc": "Testing ",
10      "microservices": [
11        {
12          "microserviceID": "",
13          "microservice_name": "testaggr",
14          "microservice_namespace": "test",
15          "virtualization": "unikernel",
16          "cmd": [
17            "--video-source",
18            "rtsp://192.168.1.173:8554/stream",
19            "--detector-address",
20            "192.168.1.173",
21            "--tracker-address",
22            "192.168.1.173",
23            "--metric-address",
24            "http://192.168.1.173:8000/aggregation",
25            "--time-address",
26            "http://192.168.1.173:8000/time",
27            "--frame-rate",
28            "500"
29          ],
30          "memory": 300,
31          "vcpus": 1,
32          "vgpus": 0,
33          "vtpus": 0,
34          "bandwidth_in": 0,
35          "bandwidth_out": 0,
36          "storage": 0,
37          "code": "http://192.168.1.173:8001/aggregationarm.tar.gz",
38          "arch": [
39            "arm"
```

```

40         ],
41         "state": "",
42         "port": "",
43         "addresses": {
44             "rr_ip": "10.30.30.30"
45         },
46         "added_files": []
47     }
48 ]
49 }
50 ]
51 }

```

Listing A.3: Deployment Descriptor for the Aggregaton Service (Container)

```

1 {
2     "sla_version": "v2.0",
3     "customerID": "Admin",
4     "applications": [
5     {
6         "applicationID": "",
7         "application_name": "testing",
8         "application_namespace": "test",
9         "application_desc": "Testing ",
10        "microservices": [
11        {
12            "microserviceID": "",
13            "microservice_name": "testaggr",
14            "microservice_namespace": "test",
15            "virtualization": "container",
16            "cmd": [
17                "./aggr",
18                "--video-source",
19                "rtsp://192.168.1.173:8554/stream",
20                "--detector-address",
21                "192.168.1.173",
22                "--tracker-address",
23                "192.168.1.173",
24                "--metric-address",
25                "http://192.168.1.173:8000/aggregation",
26                "--time-address",
27                "http://192.168.1.173:8000/time",
28                "--frame-rate",

```

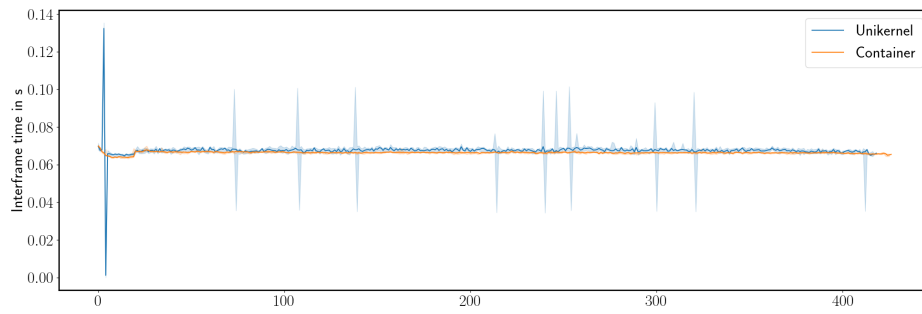
```
29         "500"
30     ],
31     "memory": 100,
32     "vcpus": 1,
33     "vgpus": 0,
34     "vtpus": 0,
35     "bandwidth_in": 0,
36     "bandwidth_out": 0,
37     "storage": 0,
38     "code": "docker.io/psabanic/caggregation:arm",
39     "images": [
40         ""
41     ],
42     "arch": [
43         "arm"
44     ],
45     "state": "",
46     "port": "",
47     "addresses": {
48         "rr_ip": "10.30.30.30"
49     },
50     "added_files": []
51 }
52 ]
53 }
54 ]
55 }
```

B. Additional Measurements

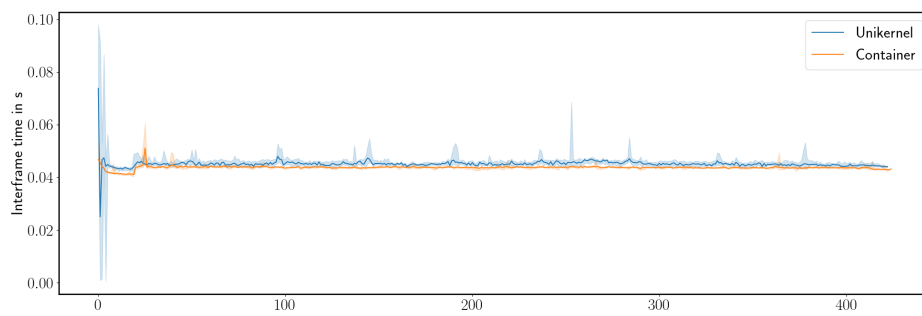
Figure B.2 and Figure B.1 show the time between separate frames from the receiver side. This is the time from the current frame to the next frame from the Aggregations service. The first one uses an RTSP video stream and the second uses a local file as a video source. Further, in Figure B.3 and Figure B.4, the CPU usage over time for the entire process of the aggregation is shown.

In Figure B.5, the latency of HTTP GET requests for the networking tests can be seen, with one connection measurement removed in Figure B.5b to show a more detailed version of the changes.

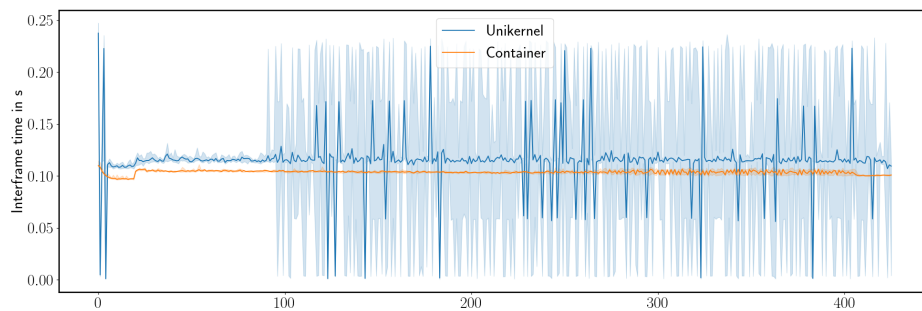
B. Additional Measurements



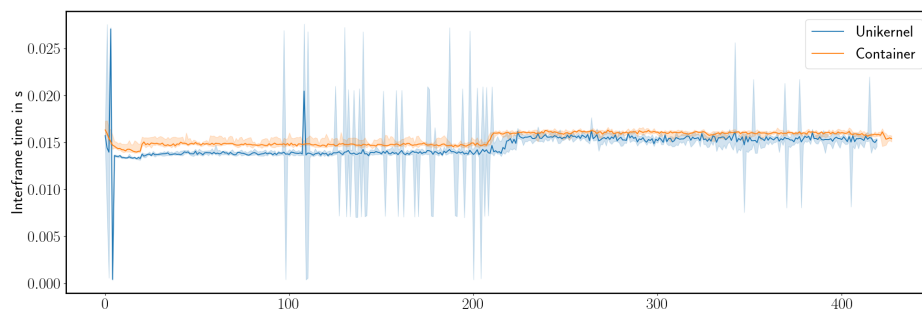
(a) UDOO x86



(b) Pi4



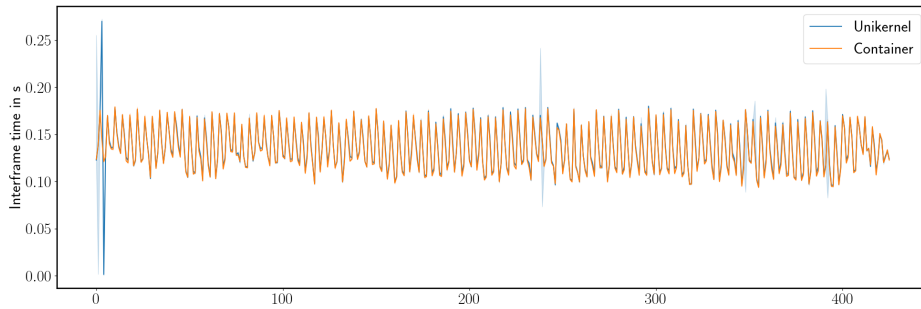
(c) Pi3



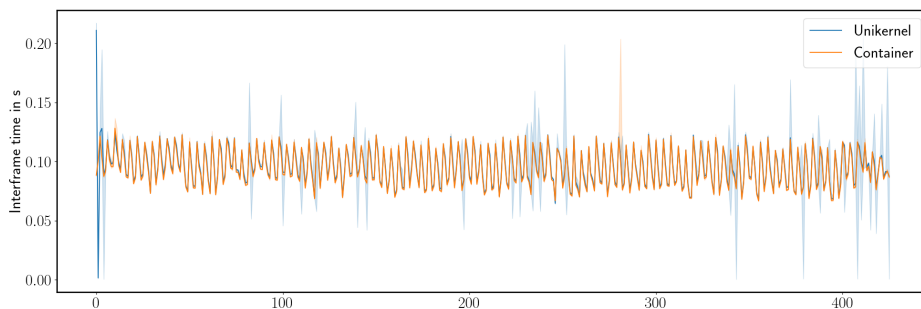
(d) Fujitsu Esprimo G558

Figure B.1.: Inter-Frame Time from Receiver Side (RTSP)

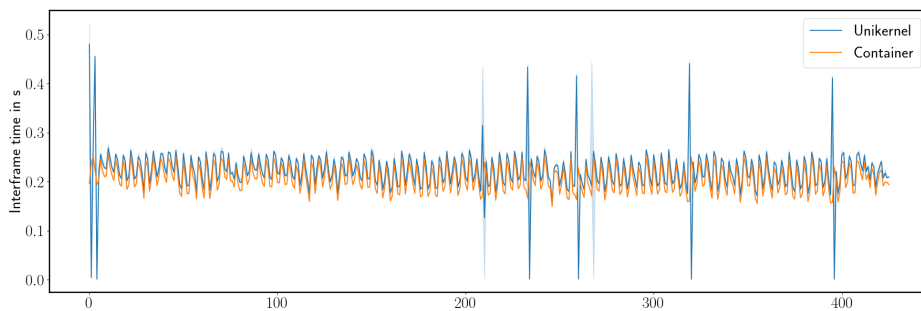
B. Additional Measurements



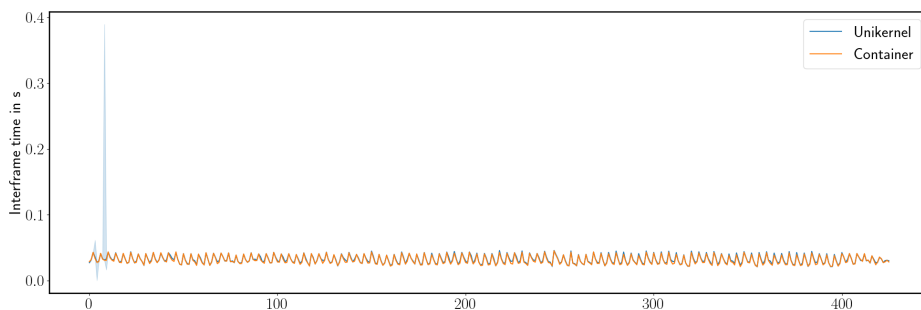
(a) UDOO x86



(b) Pi4



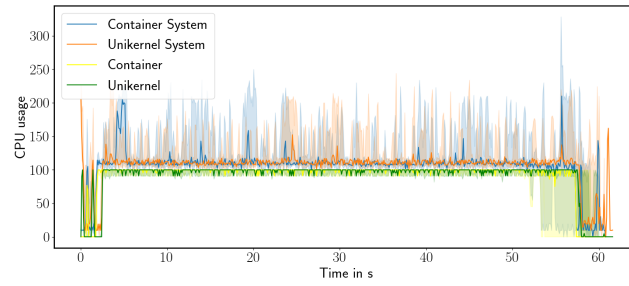
(c) Pi3



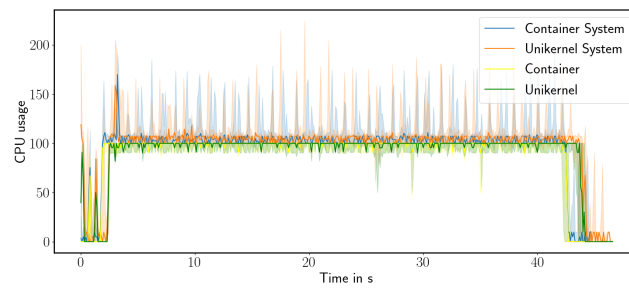
(d) Fujitsu Esprimo G558

Figure B.2.: Inter-Frame Time from Receiver Side (Local File)

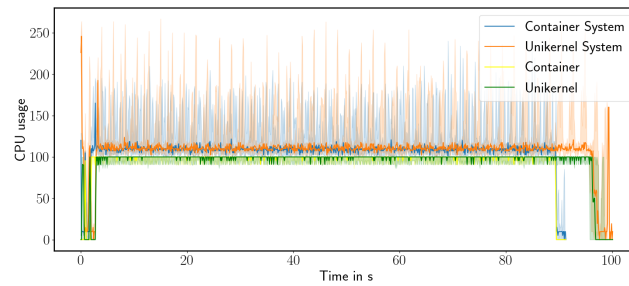
B. Additional Measurements



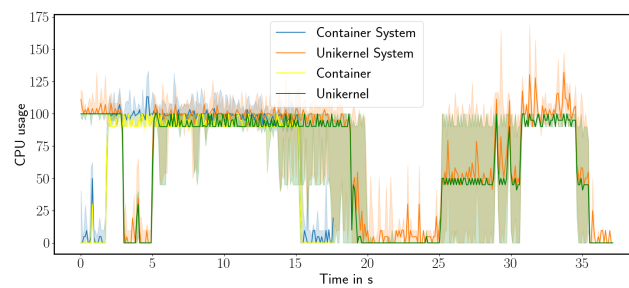
(a) UDOO x86



(b) Pi4



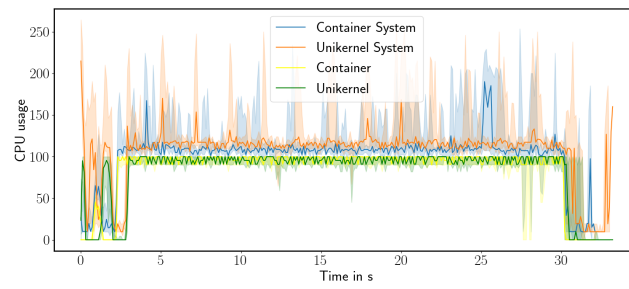
(c) Pi3



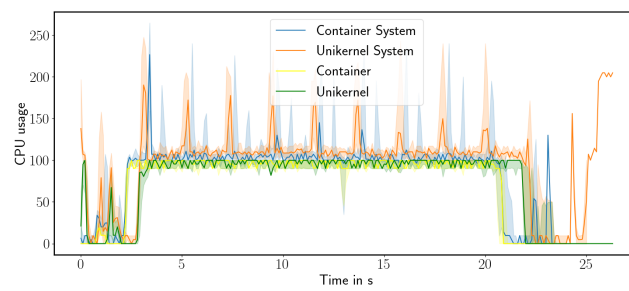
(d) Fujitsu Esprimo G558

Figure B.3.: CPU Usage over Processing of one File

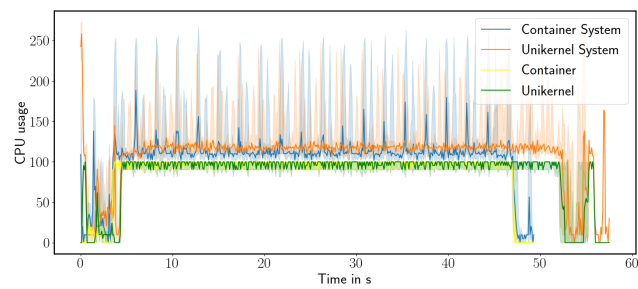
B. Additional Measurements



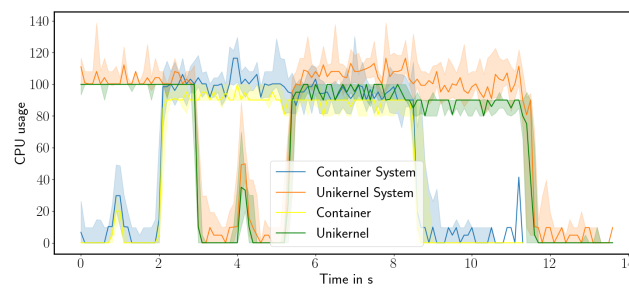
(a) UDOO x86



(b) Pi4



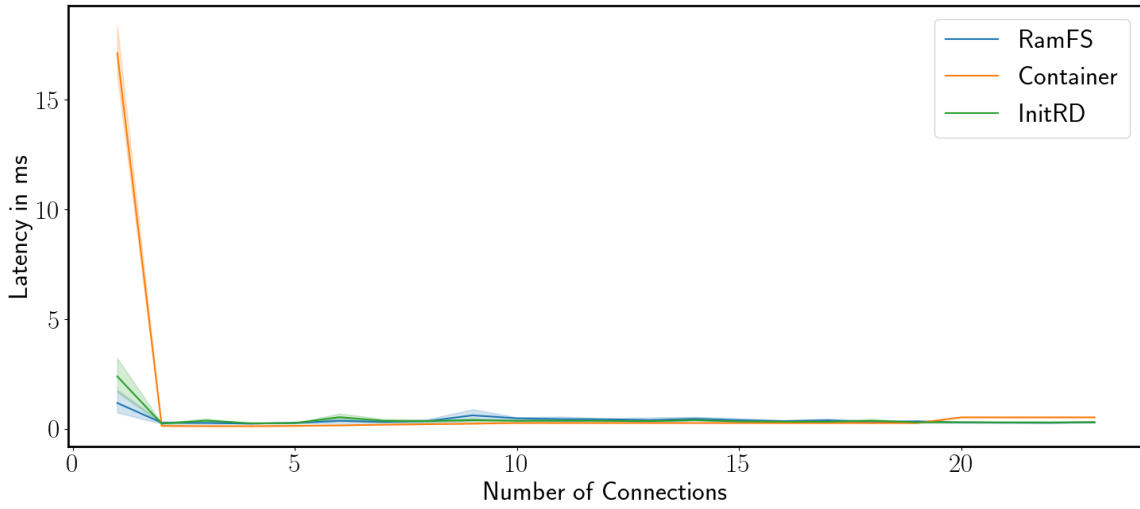
(c) Pi3



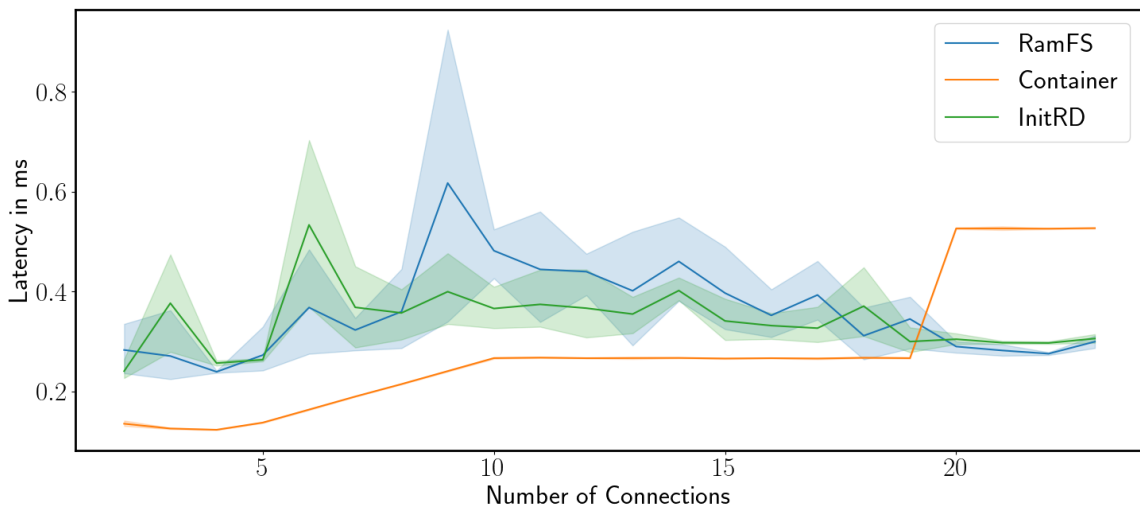
(d) Fujitsu Esprimo G558

Figure B.4.: CPU Usage over Processing of one Stream

B. Additional Measurements



(a) 2 to 24 Parallel Connections



(b) 1 to 24 Parallel Connections

Figure B.5.: Latency in Nginx Communication

List of Figures

2.1. Difference between Virtualization Design	4
2.2. Cgroups and Namespaces	5
2.3. Virtual Machine Networking	7
2.4. Virtual Machines and Unikernels	8
2.5. Address Space	9
2.6. Unikraft Architecture	11
2.7. Root Scheduling and Forwarding to Cluster	13
2.8. Cluster Scheduling and Deployment	15
2.9. Node Deployment	16
2.10. Oakestra Architecture	17
3.1. Unikraft Workspace Directory	21
3.2. Video Aggregation in the Video Analytics Pipeline	23
3.3. Time Synchronization with Host and Unikernel	25
3.4. Cumulative Distribution of the Time Difference	26
4.1. Oakestra Components	30
4.2. Unikernel Tar structure	33
4.3. Namespace Networking	38
4.4. Cluster Node Selection	40
4.5. Root Cluster Selection	43
5.1. Unikernel Time Measurements	49
5.2. Execution without Virtualization	50
5.3. CPU Usage over Execution Time	50
5.4. Unikernel and Container Metrics	52
5.5. Nginx HTTP Request/s	52
5.6. Python Application Execution Time	53
5.7. Startup Time	55
A.1. Service Registration	58
A.2. Service Registration Response	59
A.3. Service Registration	59
A.4. Service Registration Response	60
A.5. Microservice Deployment	60
A.6. Microservice Deployment Response	61
A.7. Check on Service	61

List of Figures

A.8. Check on Service Response	62
A.9. Undeploy	62
A.10. Undeploy Response	63
A.11. Undeploy Response	63
B.1. Inter-Frame Time from Receiver Side (RTSP)	69
B.2. Inter-Frame Time from Receiver Side (Local File)	70
B.3. CPU Usage over Processing of one File	71
B.4. CPU Usage over Processing of one Stream	72
B.5. Latency in Nginx Communication	73

List of Tables

- 5.1. Test System Overview 47
- 5.2. Unikernel and Container Size Comparison 54

Bibliography

- [1] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiawicz. “The Cloud is Not Enough: Saving IoT from the Cloud”. In: *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. Santa Clara, CA: USENIX Association, July 2015. URL: <https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/zhang>.
- [2] Y. Li and S. Wang. “An Energy-Aware Edge Server Placement Algorithm in Mobile Edge Computing”. In: *2018 IEEE International Conference on Edge Computing (EDGE)*. 2018, pp. 66–73. DOI: 10.1109/EDGE.2018.00016.
- [3] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ș. Teodorescu, C. Răducanu, et al. “Unikraft: fast, specialized unikernels the easy way”. In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 376–394.
- [4] S. Chen and M. Zhou. “Evolving container to unikernel for edge computing and applications in process industry”. In: *Processes* 9.2 (2021), p. 351.
- [5] T. Goethals, M. Sebrechts, M. Al-Naday, B. Volckaert, and F. De Turck. “A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing”. In: *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*. 2022, pp. 60–68. DOI: 10.1109/EDGE55608.2022.00020.
- [6] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. “Unikernels: Library Operating Systems for the Cloud”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. URL: <https://doi.org/10.1145/2451116.2451167>.
- [7] *QEMU User Documentation*. 2022. URL: <https://www.qemu.org/docs/master/system/qemu-manpage.html>.
- [8] *Docker*. 2022. URL: <https://www.docker.com/>.
- [9] *LXC*. 2022. URL: <https://linuxcontainers.org/>.
- [10] *Linux Cgroups*. 2022. URL: <https://man7.org/linux/man-pages/man7/cgroups.7.html>.
- [11] *Linux Namespaces*. 2022. URL: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [12] *Open Container Initiative*. 2022. URL: <https://opencontainers.org/>.

- [13] *Podman*. 2022. URL: <https://podman.io/>.
- [14] *OCI Runtime Specification*. 2022. URL: <https://github.com/opencontainers/runtime-spec/blob/main/spec.md>.
- [15] *OCI Image Specification*. 2022. URL: <https://github.com/opencontainers/image-spec/blob/main/spec.md>.
- [16] *OCI Distribution Specification*. 2022. URL: <https://github.com/opencontainers/distribution-spec/blob/main/spec.md>.
- [17] *Docker Hub*. 2022. URL: <https://hub.docker.com/>.
- [18] *Hypervisor RedHat*. 2022. URL: <https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor>.
- [19] *KVM*. 2022. URL: http://www.linux-kvm.org/page/Main_Page.
- [20] *Intel VT*. 2022. URL: <https://www.intel.co.uk/content/www/uk/en/virtualization/virtualization-technology/intel-virtualization-technology.html>.
- [21] *Firecracker*. 2022. URL: <https://firecracker-microvm.github.io/>.
- [22] *Solo5*. 2022. URL: <https://github.com/Solo5/solo5>.
- [23] *Virtio specification 1.2*. 2022. URL: <https://docs.oasis-open.org/virtio/virtio/v1.2/virtio-v1.2.pdf>.
- [24] *QEMU microvm*. 2022. URL: <https://qemu.readthedocs.io/en/latest/system/i386/microvm.html>.
- [25] *QEMU TCG*. 2022. URL: <https://www.qemu.org/docs/master/devel/index-tcg.html>.
- [26] R. Morabito, V. Cozzolino, A. Y. Ding, N. Bejar, and J. Ott. “Consolidate IoT Edge Computing with Lightweight Virtualization”. In: *IEEE Network* 32.1 (2018), pp. 102–111. DOI: 10.1109/MNET.2018.1700175.
- [27] R.-S. Schmoll, T. Fischer, H. Salah, and F. H. P. Fitzek. “Comparing and Evaluating Application-specific Boot Times of Virtualized Instances”. In: *2019 IEEE 2nd 5G World Forum (5GWF)*. 2019, pp. 602–606. DOI: 10.1109/5GWF.2019.8911615.
- [28] J. Talbot, P. Pikula, C. Sweetmore, S. Rowe, H. Hindy, C. Tachtatzis, R. Atkinson, and X. Bellekens. “A Security Perspective on Unikernels”. In: *2020 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. 2020, pp. 1–7. DOI: 10.1109/CyberSecurity49315.2020.9138883.
- [29] *Mirage OS*. 2022. URL: <https://mirage.io/>.
- [30] *Ling Unikernel*. 2022. URL: <https://github.com/cloudozer/ling>.
- [31] *HaLVM Unikernel*. 2022. URL: <https://github.com/GaloisInc/HaLVM>.
- [32] *includeOS Unikernel*. 2022. URL: <https://github.com/includeos/IncludeOS>.
- [33] *runtime.js Unikernel*. 2022. URL: <http://runtimejs.org/>.
- [34] *OSv Unikernel*. 2022. URL: <http://osv.io/>.

- [35] Unikraft. 2022. URL: <https://unikraft.org/>.
- [36] Unikraft Design Principles. 2022. URL: <https://unikraft.org/docs/concepts/design-principles/>.
- [37] Unikraft Architecture. 2022. URL: <https://unikraft.org/docs/concepts/architecture/>.
- [38] Mimalloc. 2022. URL: <https://microsoft.github.io/mimalloc/>.
- [39] 9P QEMU Documentation. 2022. URL: <https://wiki.qemu.org/Documentation/9p>.
- [40] lwip. 2022. URL: <https://savannah.nongnu.org/projects/lwip/>.
- [41] Xen. 2022. URL: <https://xenproject.org/>.
- [42] Kubernetes. 2022. URL: <https://kubernetes.io/>.
- [43] Kubernetes Documentation. 2022. URL: <https://kubernetes.io/docs/home/>.
- [44] Kubernetes Service Documentation. 2022. URL: <https://kubernetes.io/docs/concepts/services-networking/service/>.
- [45] Kubernetes Networking Documentaions. 2022. URL: <https://kubernetes.io/docs/concepts/cluster-administration/networking/>.
- [46] K3s. 2022. URL: <https://www.k3s.io/>.
- [47] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. “Oakestra white paper: An Orchestrator for Edge Computing”. In: *arXiv preprint arXiv:2207.01577* (2022).
- [48] G. Bartolomeo. *Enabling Microservice Interactions within Heterogeneous Edge Infrastructures*. 2021.
- [49] V. Cozzolino, A. Y. Ding, and J. Ott. “Fades: Fine-grained edge offloading with unikernels”. In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. 2017, pp. 36–41.
- [50] C. Mistry, B. Stelea, V. Kumar, and T. Pasquier. “Demonstrating the practicality of unikernels to build a serverless platform at the edge”. In: *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2020, pp. 25–32.
- [51] *AMD64 Architecture Programme’s Manual Volume2: System Programming*. 2022. URL: <https://www.amd.com/system/files/TechDocs/24593.pdf>.
- [52] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. 2022. URL: <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf>.
- [53] Unikraft Kraft. 2022. URL: <https://github.com/unikraft/kraft>.
- [54] newlib. 2022. URL: <https://sourceware.org/newlib/>.
- [55] musl. 2022. URL: <https://musl.libc.org/>.
- [56] Unikraft Python. 2022. URL: <https://github.com/unikraft/lib-python3>.
- [57] Unikraft Ruby. 2022. URL: <https://github.com/unikraft/app-ruby>.

- [58] S. Bäurle and N. Mohan. “ComB: a flexible, application-oriented benchmark for edge computing”. In: *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*. 2022, pp. 19–24.
- [59] *FFmpeg*. 2022. URL: <https://ffmpeg.org/>.
- [60] *Python asyncio*. 2022. URL: <https://docs.python.org/3/library/asyncio.html>.
- [61] *Unikraft Release 0.10.0*. 2022. URL: <https://github.com/unikraft/unikraft/releases/tag/RELEASE-0.10.0>.
- [62] *Libcurl*. 2022. URL: <https://curl.se/libcurl/>.
- [63] *Redhat Blog: Vhost*. 2022. URL: <https://www.redhat.com/en/blog/hands-vhost-net-do-or-do-not-there-no-try>.
- [64] *QEMU QMP Reference Manual*. 2022. URL: <https://qemu-project.gitlab.io/qemu/interop/qemu-qmp-ref.html#qapidoc-2203>.
- [65] *NGINX*. 2022. URL: <https://www.nginx.com/>.
- [66] *wrk*. 2022. URL: <https://github.com/wg/wrk>.