DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Distributed Execution of Unikernel Applications on Container Orchestration Platform Kubernetes for IoT Scenarios

-

Author:	Atakan Yenel
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Vladimir Podolskiy
Submission Date:	-

Abstract

Virtualisation is heavily used in cloud computing. Hardware resources in server farms are distributed among clients as virtual machines that are running on a hypervisor. As architecture model of software development moves more on to microservices, containerized applications arisen to allow multiple programs to run on the same virtual machine with isolation. While containerized applications increased the utility of hardware resources , they still require a full fledged operating system underneath to host them. Running a complete operating system to run microservices is problematic in 2 ways. They allow for a greater attack surface , because a vulnerability in the underlying operating system can also compromise the running applications. This causes DevOps engineers to specify a secure host operating system for their applications, which contradicts with the "abstraction from operating system" promise of containers. Second, operating systems waste hardware resources with background jobs that are not required to the application.

Unikernels are specialised programs that package the smallest operating system with the application itself. They don't require a running operating system to be deployed and they can be booted directly from BIOS. They have a very small attack surface as vulnerabilities can only exist within the deployed app. This reduces the code fingerprint of the deployed system greatly. They also don't waste any resources as there are no processes running other than the application.

Because unikernels are very small operating systems , they can be booted with type-1 and type-2 hypervisors. This thesis will create a ecosystem that boots up unikernel applications scheduled by a Kubernetes cluster. The expected result is decrease in both resources used by the cluster and time it takes to scale an application.

It will try to answer questions e.g. How to communicate with hypervisors ? How to deploy external unikernels through internet to hardware ? How to connect unikernels to kubernetes ? How does unikernels affect scheduling of kubernetes applications ?

Keywords: Cloud computing, Unikernel, Kubernetes, Hypervisor

Contents

1 Introduction

Cloud computing moves towards smaller deliverables. First hardware,then infrastructure, virtual machines, platforms, containers, and functions. While customers are facing with smaller deliverables, the background stays mostly the same. Even if the user only deploys a function, cloud provider creates an OS and deploys a containerized application that runs the given user function. There are at least 3 different abstractions between what user wants to run (their function) and what runs it (hardware). That's 2 times more than the desired abstraction level. There should be other ways to decrease the distance between the hardware and the user application without compromising the flexibility of cloud computing.

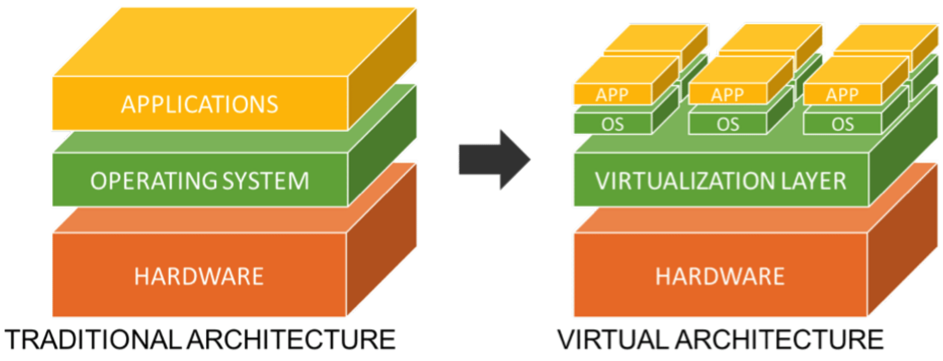

Figure 1.1: Moving from traditional architecture to virtualization based architecture: https://resources.infosecinstitute.com/11-points-consider-virtualizing-security/

1.1 Motivation

Unikernels are special programs that package application code together with a minimal operating system image.**[7396164]** These special programs can be booted like operating systems and more interestingly , they can also be virtualized like operating systems. That means we can do , in theory, any scheduling operation we normally do on virtual machines with unikernels. In the current world of orchestration technologies , this opens many new areas to apply the principles of deployment, scaling and scheduling.

Kubernetes is an open source system for automating orchestration of containerized applications. **[Hightower:2017:KUR:3175917]** Kubernetes by default uses Docker as it's runtime. Both it's own services and user defined services are deployed as docker containers. Instead of communicating with virtual machines running on the Kubernetes cluster, user only interacts with the Kubernetes API and it schedules deployments according to rules.

The docker runtime requires a host operating system to run the docker daemon, thus this requires every node to have a full fledged operating system to run even simple applications. Those operating systems also run background jobs that come with the OS but not required by the running applications. E.g. a usb driver or a sound driver is not vital for a web application but operating system still runs them by default. Unikernel programs package only functions required by the running application in a different configuration. That decreases the scaling time[**Podolskiy:2017:QCA:3069383.3069390**] of the applications because first no operating system is required to boot up, and second the applications are much more smaller , so they can be downloaded faster.

The proposed effects of a kubernetes cluster running together with a unikernel system can be summarised as follows:

1.1.1 Security

These OS's are required for the multi-tenant cloud architecture for isolation. Otherwise, vulnerability in one of the containers can compromise all other containers running on the same VM. On the other hand, a vulnerability found in the operating system can also compromise the running applications. Unikernels packages only include required parts of the operating system so the attack surface is greatly reduced.

1.1.2 Resource Utilization

The user never interacts with the underlying operating system that their application is running on. It is only used by cloud provider infrastructure. Nevertheless, the user still pays for the resources used by the operating system, even though their application doesn't.

1.1.3 Scaling Time

Unikernels have a very small footprint and can run on hardwares only with a hypervisor.

1.1.4 IoT Scenarios

Unikernels also have role in the IoT world. Their size allows them to be deployed to places with sparse internet connection or low-end devices. Their security story makes them hard targets in the IoT ecosystem, where security practices are still a challenge for developers. [**iot-sec**] On the other end, the single-address space feature of unikernels is a downside for their usage in IoT applications, because it makes it hard for unikernels to read sensor data from the device itself. Nevertheless, for simple, proof of concept applications, kubernetes can be used to schedule them to IoT devices. Kubernetes comes in handy for such scenarios, because users can tag deployments they make and K8s will deploy them to their respective devices. Multiple unikernel applications can be deployed to the same device with a single deployment through the Kubernetes API or resources provided by the IoT devices can be accessible from the Kubernetes , and unikernels can be deployed to those. Given that multiple unikernels can run on the same device, a microservice based architecture can also be used for IoT programs where a single unikernel is tasked with only one job and it communicates with other unikernels through a shared medium locally on the device or through Kubernetes services. This brings the advantages of microservice based development to the IoT domain.

1.2 Research Problem

1.2.1 Deployment

Unikernels can be booted only in systems that either have a type-1 or a type-2 hypervisor. While they can be booted directly from BIOS on hardware, this does not give the flexibility required by cloud computing standarts so a hypervisor is required to boot up and remove programs to achieve the desired state by the system.

Type-2 hypervisors run on a host operating system and they are easier to work with. If they provide an API, the host operating system can be used to develop programs to communicate with them. Given an example , a linux machine with virtualbox installed can boot up unikernel programs with terminal commands and no GUI is required. Then ,this terminal commands can be automated with a program that communicates with the internet.

For type-1 hypervisors, the task at hand is harder. There is no operating system involved and one has to work mostly with the API provided by the hypervisor itself. This calls for a program that either can be deployed to the hypervisor as a unikernel itself or for a virtual machine that can talk with the underlying hypervisor.

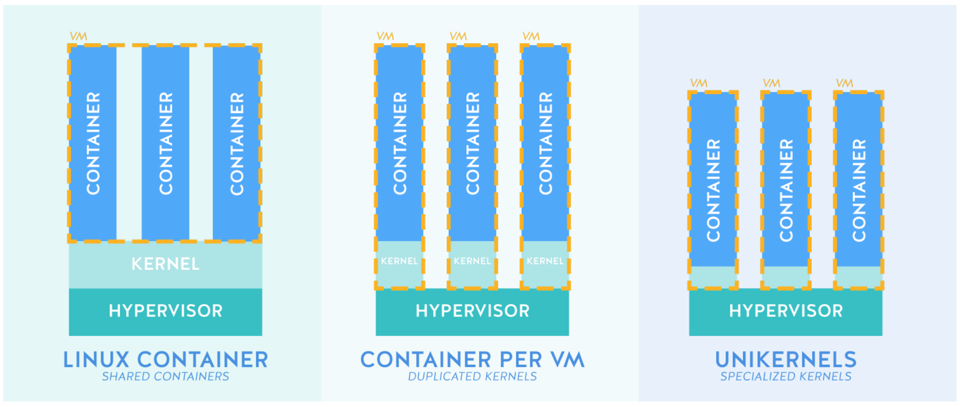

Figure 1.2: Different virtualization techniques: https://nordicapis.com/introduction-to-unikernels/

1.2.2 Virtual Kubelet

Virtual Kubelet[**virtual**] is a Microsoft project that aims to provide a programmable kubelet API interface for developers. It allows to deploy Kubernetes resource to different runtimes, transparent to the Kubernetes API. This project is currently a great candidate to deploy the unikernel solution proposed by this thesis. Virtual kubelet communicates with a custom pods provider to deploy resources. If the implementation of this thesis follows this approach, Virtual Kubelet will be forked and a new provider will be written for certain hypervisors to deploy unikernels. This also gives the opportunity to do a pull request to the project directly at the end of this thesis.

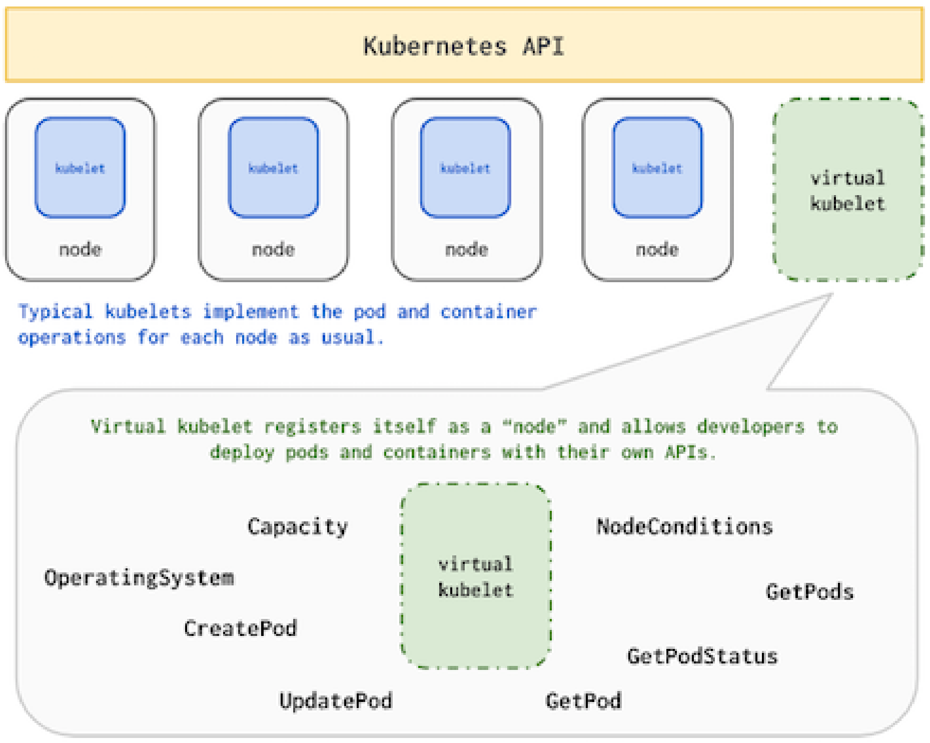
Kubernetes API
kubelet
node
kubelet
node
kubelet
node
kubelet
node
virtual kubelet
Typical kubelets implement the pod and container operations for each node as usual.
Virtual kubelet registers itself as a "node" and allows developers to deploy pods and containers with their own APIs.
Capacity
OperatingSystem
CreatePod
UpdatePod
virtual kubelet
NodeConditions
GetPods
GetPodStatus
GetPod

Figure 1.3: Working principle of Virtual Kubelet [**virtual**]

Virtual Kubelet also opens possibilities to deploy unikernels to IoT devices through the Kubernetes API. Different virtual kubelet instances with different providers can work in parallel to deploy to target environments. This, again requires a receiver on the IoT device part.

1.2.3 Kubernetes service discovery

Kubernetes has a built-in service discovery. Deployed containers use hostnames instead of IPs to communicate between each other. Kubernetes runs a DNS server and maps those hostnames to IP addreses of individuals containers or mostly services. Unikernel deployments should also have the same functionality to work flawlessly with the rest of the kubernetes cluster.

1.2.4 Communication between kubernetes and hypervisor

Kubernetes has a single endpoint for all the communication between the user and the cluster. This single endpoint should be aware of unikernel deployments so that it can schedule them accordingly. Kubernetes provides different ways to achieve this extensibility. First, it allows developers to create custom resource definitions such that they can be used together with other kubernetes resources. Second, kubernetes allow for deployment for different scheduler that can operate on specificly-tagged deployments.

1.3 Proposed Solution

Before starting to implement unikernels to Kubernetes, a couple of proof of concept unikernel programs will be created. To create those programs, open-sourced unikernel solutions will be used. There are multiple projects with different approaches and different runtime environments. Next chapter explains more about those solutions. Selecting the most suitable one for the purpose of this thesis plays a crucial role because, it is out of scope for this thesis to develop a new unikernel solution just for Kubernetes. Once those programs are created, unikernel deployments will be added to kubernetes arsenal with the following steps:

1) A node with a type-2 hypervisor will be created. A custom program will be written that takes orders from a server and boot up unikernels according to parameters coming from the order. The custom program will run on the host operating system and there will be no kubernetes involvement.

2) A kubernetes cluster will be created and that custom program will be modified to communicate with kubernetes. Everytime a unikernel deployment is made, kubernetes will notify this program, and program will do the deployment.

3) Kubernetes DNS system will be integrated to the node with the type-2 hypervisor so they can communicate with other applications.

4) A node with a host operating system with type-1 hypervisor will be created. E.g. Xen hypervisor can also run on a host operating system. This node will be modified to communicate with the kubernetes cluster through the host operating system.

5) A node without a host operating system with type-1 hypervisor will be created. This node should also run the custom program that communicates with the kubernetes cluster on the hypervisor and should also do the networking of unikernels between other kubernetes resources.This will be the final outcome of the thesis.

A scope of this project is not to come up with a new unikernel solution and it will use the ones already developed by the open-source community.It also won't compare application performance between docker containers and unikernels. It will although compare the deployment time between those.

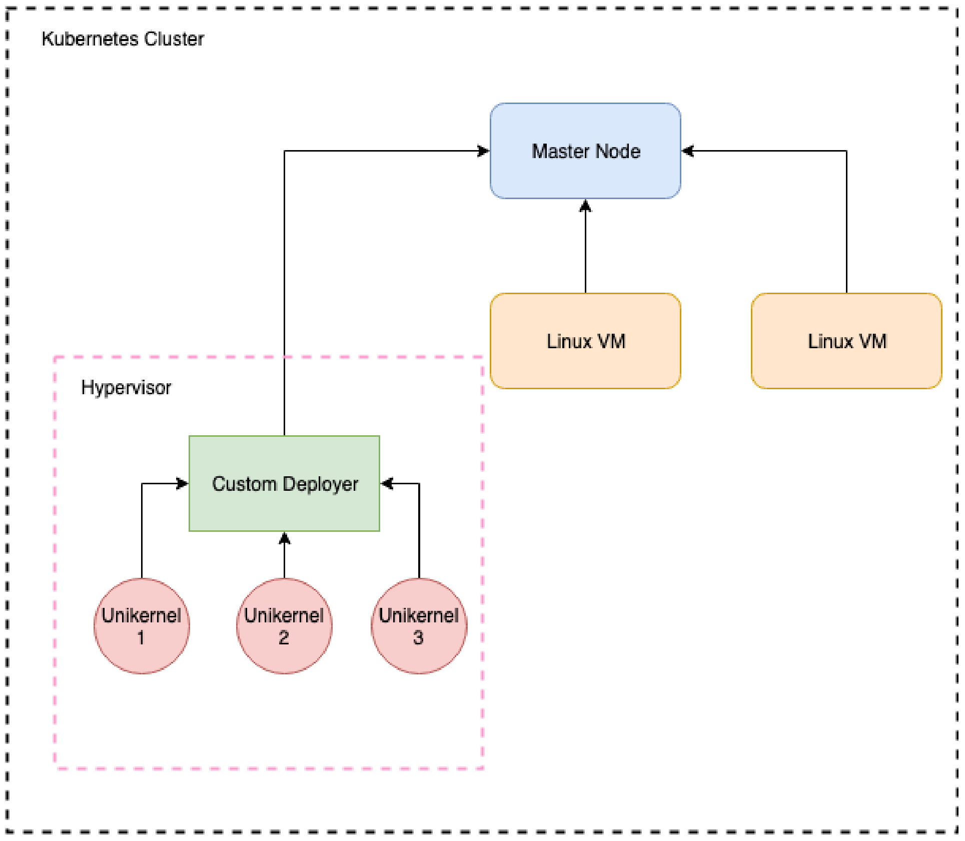

Figure 1.4: A kubernetes cluster with a only hypervisor enabled node for unikernel deployment

2 Literature Review

The search on "unikernel"s in Google Scholar brings up 843 articles and the search "unikernel AND kubernetes" only brings 137, while "kubernetes" itself brings 5,600. Some articles only mention unikernels as a potential way to deploy their application and does not work on the topic itself. Many articles compare unikernels with other virtualisation methods and benchmark them. These articles might be useful when measuring deployment times of unikernels in Kubernetes.

The practical part of unikernel development is mostly running through open source projects. Nevertheless, there is currently no production ready unikernel project that can be used for any arbitrary need. There are multiple groups developing unikernel solutions. A small group of these solutions include:

- MirageOS `https://github.com/mirage/mirage` [**madhavapeddy2014unikernels**] MirageOS is a unikernel solution for the OCaml language. It provides libraries for e.g. networking, storage that become operating system drivers when the application developed for production. It produces artifacts that run either under the XEN or KVM hypervisors. It also runs on the ARM64 CPUs , which makes it possible to deploy MirageOS unikernels to Raspberry Pis as IoT targets.

- Unik `https://github.com/solo-io/unik`[**levine2016unik**] Unik is a promising project for this thesis. The project brands itself as "Compilation and Deployment Platform" and has support for many providers. They also have a documentation to add new providers. They also support different languages to write unikernels, which might come in handy when developing logic for end devices.

- IncludeOS `https://github.com/includeos/IncludeOS` [**7396164**] IncludeOS includes the operating system code to the application as a library when compiled.It can compile C and C++ applications. They target IoT devices as well for improved security. IncludeOS can be used to develop function as a Service (FAAS). Kubernetes can be used to realize that architecture.

- Ops `https://github.com/nanovms/ops` Ops is an interface for creating unikernels in the nanovms infrastructure. [**nanovms**] It is a wrapper around QEMU [**qemu**]. Ops uses configuration files to build unikernel images at compile time. It also supports multiple languages.

Some of these projects also include Kubernetes as their deployment targets but they do it by including a host OS, thus does not use the full potential of unikernels.

There are companies working on development of unikernels. The most prominent one is Docker, which is the defacto standart for containerised applications. [**francia_2016**] Unikernels was also subject to CNCF conferences in the past.

3 Background

4 Implementation

4.1 Type-2 Hypervisor

4.2 Type-1 Hypervisor

4.3 IoT Devices

5 Evaluation

5.1 Usability

5.2 Code Footprint & Security

5.3 Resource Utilization

5.4 Scaling Time

5.5 Use Cases in Internet of Things

6 Conclusion

List of Figures

List of Tables