Tampere University

Md Towfiqul Alom

Unikernel as Service Units on Distributed Resource-constrained Edge Devices

Faculty of Information Technology and Communication Sciences (ITC)
Master's thesis
November 2020

Abstract

Md Towfiqul Alom: Unikernel as Service Units on Distributed Resource-constrained Edge Devices
Master's thesis
Tampere University
Master's Degree Programme in Information Technology
November 2020

There are almost 31 billion IoT devices in the world. Due to its widespread use the concept virtualization and cloud computing is moving towards the IoT systems support. There are lots of challenges that appears in terms of deploying the computational units in IoT devices due to its resource constrained nature and the absence of high processing power. A new technology called unikernel provides the support to meet these challenges. Unikernel combines the applications stack with a specialised kernel that has the only required libraries of operating systems to run the application stack. The end result is a low memory footprint image that can be deployed in memory constrained IoT devices. In industrial server hundreds of IoT devices run on same network and share their computational units with each other. Managing and orchestrating these services running in IoT devices requires additional overhead application units which makes the whole automation process much complex. Arrowhead framework is an industrial automation tool provides the solution to this challenges. With its service oriented architecture(SOA) approach the framework provides collaborative automation to maintain inter connectivity between computational units in IoT devices in a form of System of Systems(SoS). Arrowhead framework introduces a local cloud concept that deducts the additional application stack overhead to achieve the automation processes like service discovery, registry and orchestration.

Keywords: Operating system, unikernel, ARM 64, orchestration, Arrowhead framework.

Contents

1 Introduction

The term Virtualization has become a great buzzword after its introduction by IBM in 1960s (Kochut and Beaty 2007). The technology lead the foundation to a modern day utility computing and cloud infrastructure. Before virtualization, running an abstract services required a running physical server with other required services. With virtualization it allows the users to create and use multiple environment or resources one physical system. Cloud on the other side use the technology to share the resources as a scalable resource over a network.

The core concept of the cloud infrastructure is to give support to numerous operating systems without physically copying the application to different systems. In that scenario the hypervisor become the core delivery infrastructure. The hypervisor which is also commonly known as virtual machine monitor or VMM is used to create and run virtual machines. The virtual machines contain the guest operating systems like Windows, Linux, MacOS as a standalone system that shares the virtualized hardware resources from a single host physical machine. The hypervisor works on top of the hardware layer to isolate the guest operating system and facilitates them with the underlying hardware resources. The common example of hypervisor is Microsoft Hyper-V hypervisor, VMware ESXi, Citrix XenServer.

In cloud computing infrastructure as a service is a popular resource allocation model where users can run virtual machines in providers server and consume the resources in an on demand basis without buying or deploying a whole physical infrastructure or a underlying hardware system. Usually these virtual machines contains a full-fledged operating system that has all the necessary features to run a variety of applications.

After the introduction of a phenomenal concept called ”scaling” by computer scientist Dr.Douglas Engelbart (Barnes 1997), the computer hardware, CPU chips or embedded micro controller are getting smaller yet generating more computational power. In software industry the trend is quite the opposite. Because of the feature rich nature of the modern day operating systems, it consumes more resources in an IaaS environment. Also, maintaining the up time of the services is a crucial factors, but it is problematic due to the complexity and often creates a mess while downing and redeploying a virtual machines containing an entire operating system.

Using container technology is the most popular solution to this problem. Virtual machines contain an entire operating system along with the application. A physical host machine running two virtual machines need a hypervisor with two standalone operating system running on top of the hypervisor. In containerization, the technology runs a single operating system and the separate container shares the operating

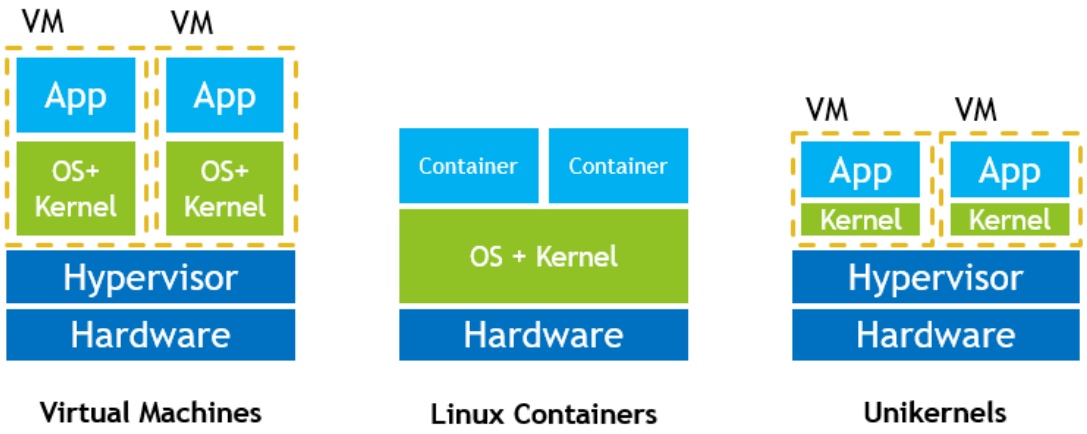


Figure 1.1 *Architecture comparison of virtual machines, Linux containers and unikernels(CETIC 2013).*

system kernel with the other containers. The read only operating system along with the application are packaged to a container and deployed in a second. This makes the container more lightweight than virtual machines. A host machine can deploy more containers than virtual machines as it consumes less resources. One crucial benefit of containerization is that it takes seconds to boot while a virtual machine takes several minutes.

Containers run on a general purpose operating system like Windows or Linux, which performs hundreds of system level operations which the target application running along in the container doesn't need. These operations consumes gigabytes of data storage and computational powers which leads to an additional cost in IaaS cloud scenario. Hence comes the idea of unikernel with a simple concept of providing the application the necessary computational power it needs. Unikernel is a fixed purpose image that is developed using library operating system. Using specially designed compiler system, the application and all the libraries of the operating system the application needs are converted into a bootable image. These images can be run in any hypervisor or hardware without the need of a full featured operating system. The lightweight unikernel image only contains the necessary drivers, I/O and library functions.

MirageOS is a popular unikernel project which has a DNS server unikernel image only 449KB of size (Madhavapeddy, Mortier, et al. 2013). The Ling project runs a website as an unikernel in 25 MB in size (CETIC 2013). Less storage space and power consumption leads to a reduced cost in any IaaS cloud providers like Amazon EC2 or GCP(Google Cloud Platform). Also, unikernel gives a clear advantage on application security as it eliminates the attack surface of any application.

1.1 Problem Statement

The containerization technology itself creates some problems of it own. One of them is the security vulnerabilities. Containers share the kernel of a same operating system on a same host machine. If the security of the host machine is compromised, the containers running on the host machine can be compromised. Also, containerization is not operating system independent. The most popular commercial containerization technology is Docker. Docker was originally designed for Linux containers or for Linux kernels only. Currently Docker uses runC which build on OCI (Open Container Initiative) specifications (Sarkar 2019) which is used to create container based on the host machine operating system.

Docker may rule the x86 or x64 hardware devices but it doesn't provide full containerization support to embedded devices. The Docker images for the containers have to be built on the same hardware architecture as the deployment environment. That means while running a Docker container in Raspberry Pi, the images of the container have to be built in a ARM64 CPU architecture. The embedded system devices including Raspberry Pi uses flash storage which contains a limited storage space. Although Docker has the functionality to provide containerization support for ARM system, low storage space brings difficulties in memory management while deploying a Docker container in a ARM 64 edge device.

1.2 Research Questions

The research questions of the thesis can be summarised as follows,

1. Is unikernel an effective solution for the deployment of large computational stack to resource constrained IOT devices?
2. Which tools to choose based on the solution to deploy virtualized images to resource constrained devices based on performance, execution and maintainability?
3. Can Arrowhead framework provide the necessary support for the management and orchestration of the deployed applications stacks in these devices?

1.3 Objectives

With the aim of exploiting the possible solution for the above mentioned problem statement this thesis will fulfill following goals:

- This thesis aims to analyze a new hypervisor based virtualization technology that supports the need of developing and deploying computational stacks and virtualized images in low memory footprint devices. Over the years some of the state of the art virtualization technologies are developed by open source communities. These technologies have different build structures and compiler platforms. This thesis aims to exploit the development build structure and effectiveness of these solutions to compare them against the Docker based contenarization solution.

- This thesis also aims to develop a comparison model of these state of art virtualization technologies. This structured comparison model approach will lead to a better understanding while choosing the correct solution for developing the container images of the computational units and deploying the images to the resource constrained devices. Each of these new solutions has different characteristics in terms of systems developed with different languages, deploying the images to different hypervisors. The comparison model will be an unified solutions that provides a clear picture on choosing the right solution for the specific application stack.

- Based on the comparison model the thesis's aims to develop virtualized images of a proof of concept solutions using the chosen technologies. The demonstration scenario aims to analyze these technology's capabilities in terms of memory footprint management, deployability, performance, networking etc. Also, the aim includes a comparison diagnostics of advantages and disadvantages of deploying the new images to the limited resourced ARM device.

- Management, proper data communication, effective access control limitation of deployed computational units as virtualized images in different IOT devices is a complicated task. The thesis aims to analyze an internet based tools that provides a proper ecosystem in terms of discovering the required services, maintaining the interconnectivity, faster data communication and work as an central control system to manage the systems in different cloud scenarios.

Thus doing so the thesis will provide a higher order picture of new ways to virtulalise images for embedded devices and a production grade environment for managing these solutions.

1.4 Systematic methodology

The thesis methodology follows an experimental approach. To answer the research question the thesis conducted four experiments. The experiments are as follows:

Experiment I: Unikernel development scope

The experiment aims to analyse the development scope of different unikernel technologies like MirageOS, OPS Nano VM, OSv etc.

Experiment II: Analysis on boot time, footprint monitoring with test cases

The experiment aims to analyse the unikernel images developed in Experiment I in two different test cases and develop a model to choose a preferred unikernel technology for Experiment III.

Experiment III: Deployment of unikernel image of proof of concept solution to ARM device

The experiment aims to analyse the deployment scope of unikernel image of a computational units developed with modern day programming languages in ARM 64 device like Raspberry Pi.

Experiment IV: Orchestration of unikernel proof of concept solution using Arrowhead framework

Experiment IV aims to analyse the scope of orchestrating the computational units deployed in resource constrained ARM 64 devices using Arrowhead framework.

2 Background

2.1 Library Operating System: A new OS concept

The current general purpose operating system is designed to support wide range of operations, solving complex end user need. The operating system Linux supports numerous platforms including low level architecture devices (McKenney and Walpole 2008). The current virtualized server don't need a full fledged operating system to manage the operations as the hypervisors are well equipped to do it.

There are several research are done to solve the problem based on a architecture called library operating system. There are several pioneering projects were developed on the basis of libOS architecture and one of them is Exokernel. The initial idea of these architecture development was to enhance perform ace and reduce required runtime privilege between application stack and hardware layer. Virtual machines are connected to physical hardware using the hypervisor. Hypervisor supports the virtual machines with runtime CPU's, storage, network stacks. LibOS delivers only those drivers to the virtual hardware that is required for hypervisors to connect it with the physical hardware of the host machine.

In Exokernel architecture, application specific hardware abstractions are provided whenever application system needs. These abstraction are provided with untrusted libraries or as library operating system. These ensures precise hardware resource allocation and enhance the performance of the application(Engler, Kaashoek, and O'Toole 1995) .

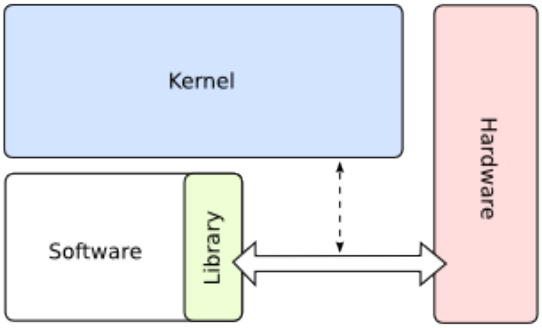


Figure 2.1 *Architecture of exokernel operating system(Engler, Kaashoek, and O'Toole 1995).*

2.2 Virtualization model : Hypervisor

Hypervisor facilitates scheduling infrastructure for the virtual machines by scheduling the resources form the host CPU needed to run the virtual machines on top of the host machine. The great advantage of the technology is that it allows the host hardware to run several operating systems as virtual machines and hypervisors shares the same hardware resources as virtual resources with the virtual machines.

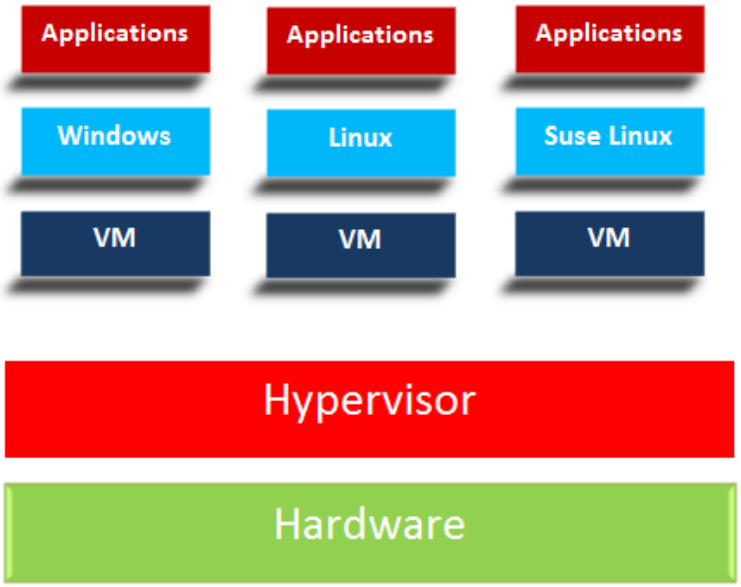


Figure 2.2 *hypervisor architecture(binarymist 2012)*

Currently there are two types of hypervisor is used in operating system virtualization, the Type1 hypervisor and the Type2 hypervisor.

The Type1 hypervisor are also referred to as the native hypervor or as the bare metal hypervisor. These hypervisors operate directly on top of the host operating system to schedule resources for the guest operating systems running as virtual machines. Type1 hypervisors are widely used in cloud server based infrastructures. Kernle based virtual machine(KVM), Hyper-V, VMware ESXi, Xen hypervisors are the common example of Type1 hypervisor.

The Type2 hypervisors are also referred to as the hosted hypervisor. The Type2 hypervisor operates as an application layer on a monolithic general purpose operating system like Linux, windows. With Type2 hypervisor the abstraction of resources for the virtual machines are operated from the host operating system and later the scheduling of the resources are executed for the host machine hardware. Virtual box, VMware workstation player is the common example of Type2 hypervisor.

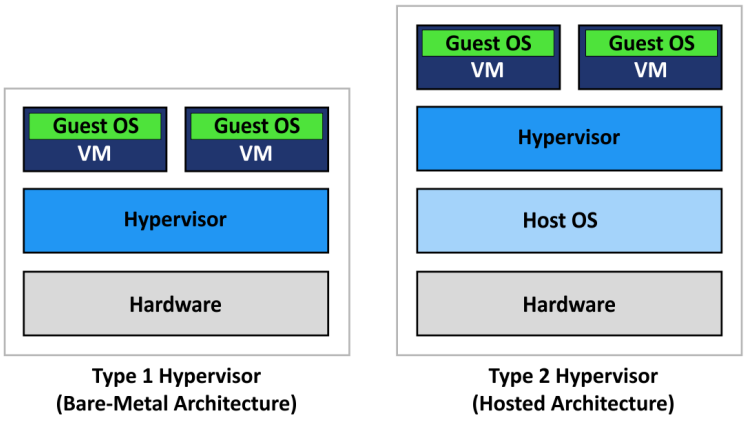


Figure 2.3 *Type1 and Type2 hypervisor architecture(nakivo 2018)*

2.2.1 Kernel based virtual machine

Kernel based virtual machine or KVM is developed by a startup company called Qumranet(Fenn et al. 2008). After the acquisition from Red Hat, KVM is now integrated with the Linux Kernel. KVM enables the Linux kernel to operate as a hypervisor. KVM was originally designed for x86 processor. Now it can function hardware virtualization in numerous platforms including ARM64 processor operating system kernel.

Linux uses KVM to convert itself as a Type1 hypervisor. As KVM runs as part of Linux kernel, it provides all the system resources like memory management, I/O, networking to the virtual machines using the standard Linux scheduler.

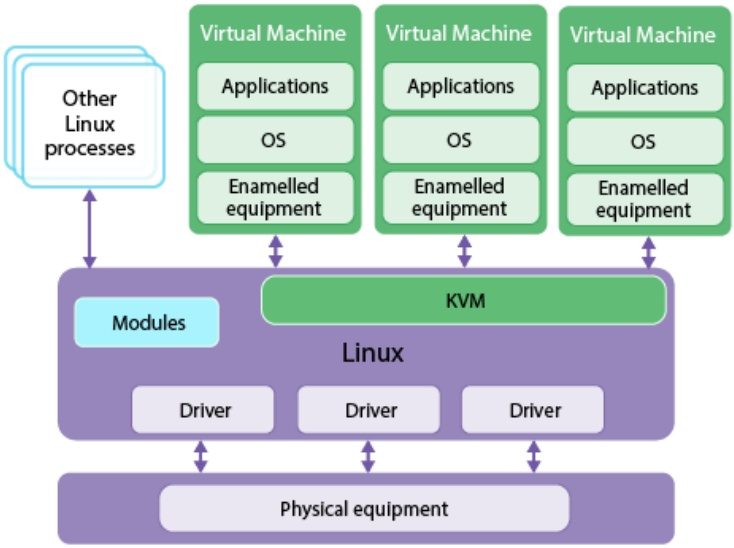


Figure 2.4 *KVM architecture(tuchacloud 2020).*

2.2.2 VMWare ESXi

VMware ESXi is Type1 hypervisor that runs directly on top of a physical server. VMware ESXi comes with a loaded smaller footprint specialized kernel. Previously VMware used ESX VM kernel with the virtualized components. Now it's using ESXi which doesn't have a Linux kernel at all. Instead of a Linux based service console it now uses a smaller footprint command line interface (Fayyad, LucPerneel, and Timmerman 2013).

VMware architecture is consists with a POSIX operating system named vmkernel and the processes that runs along with it. Crucial operations like scheduling of resources, I/O, device driver supports for virtual machines are provided by vmkernel in VMware ESXi architecture model(Fayyad, LucPerneel, and Timmerman 2013).

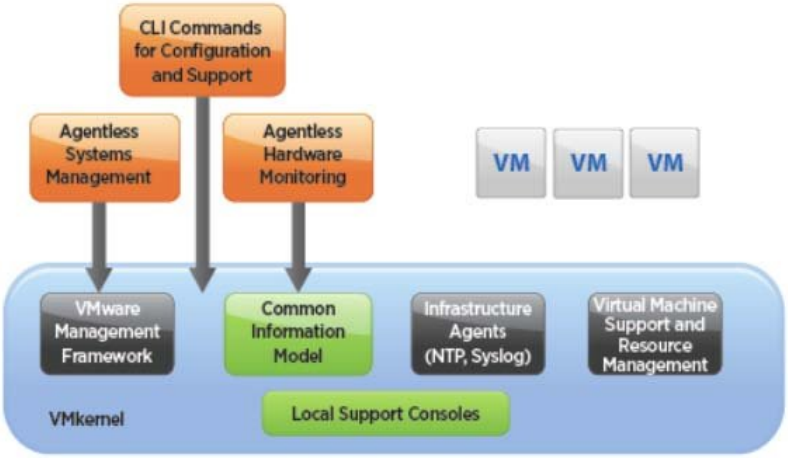


Figure 2.5 *VMware ESXi architecture(Fayyad, LucPerneel, and Timmerman 2013)*

2.2.3 Xen hypervisor

Xen hypervisor is another bare metal Type1 hypervisor developed by XenSource which is later acquired by citrix in 2007. This open source hypervisor can be installed directly on the host machine hardware without the runtime functionalities of the host operating machine.

Virtual machine's guest operating machines requests required resource abstraction from the hardware component to the Xen. Xen uses Xen virtual device drivers to provision the resources for the virtual machines from the hardware component. In terms of security, smaller footprint, interface limitaion Xen becomes a best choice hypervisor. Citrix Xen hypervisor supprts most of the operating systems and can be deployed to x86, x86-64 or ARM processors

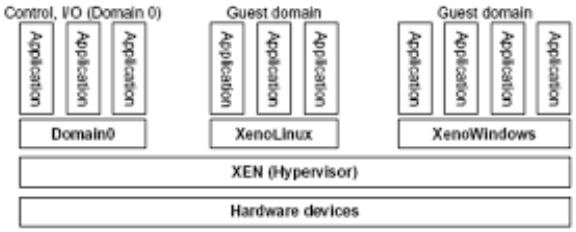


Figure 2.6 *Xen hypervisor architecture (Robin 2019)*

2.3 Containerization technology

Container are the similar concept of virtual machine, the only different is that they serve two different purposes and the container technology follows more lightweight approach than the virtual machine. The container technology creates a platform as a service(Paas) software environment to provide great inter connectivity while utilizing the host operating machine resources(Pahl 2015). The containerization technology offers three different advantages against the VM's,

- A lightweight portable run time virtual machines on top of host operating system.
- Platform independent portable software stack with the capabilities updates and redeploy.
- Network connectivity between different containers deployed in a single host machine.

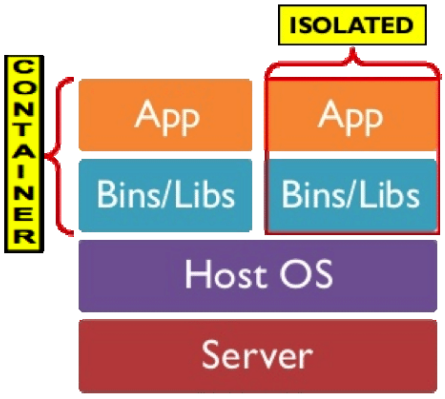


Figure 2.7 *A container image architecture(Pahl 2015).*

The main difference the containerization technology brings is in the boot process. In the virtual machine deployed application stacks requires traditional boot process

resulting in the usage of numerous hypervisor of the host machine. On the other hand containers use daemons such systemd to boot the application container images on the host machine(Pahl 2015).

2.3.1 Docker Containerization

The modern application stack deployed on the virtualized environment requires continuous re deployment and proper utilization of host machine hardware. The traditional monolithic approach of virtualization creates numerous over heads and creates a complex re deployment process(Jaramillo, Nguyen, and Smart 2016). Hence comes the Docker based containerization technology. Docker technology is an open source containerization technology that provides easy infrastructure faster deployment scopes for small to larger application units. In virtual machine concepts the hypervisor works on top of host operating system to control the virtualized system. On the other hand the in Docker the isolated containers are hosted on top of the host machine and the specialised Docker daemon co-ordinates the container with the host machine(Jaramillo, Nguyen, and Smart 2016).

The Docker container uses linux container(LXC) which can run multiple linux kernel on a single host machine. The whole Dockerization process uses client architecture model where the the Docker client uses the Docker daemon to create, run and and deploy Docker containers. The Docker client running a host machine can communicate with Docker daemon running on a host machine and also to a remote daemon through RESTful API. The Docker client receives the instruction in a form of Dockerfile and re directs these instructions to Docker daemon to create the Docker images(read- only templates) to create the Docker containers. The Docker images holds all the application codes and dependencies that needed to run the application in a Docker container(Preeth E N et al. 2015).

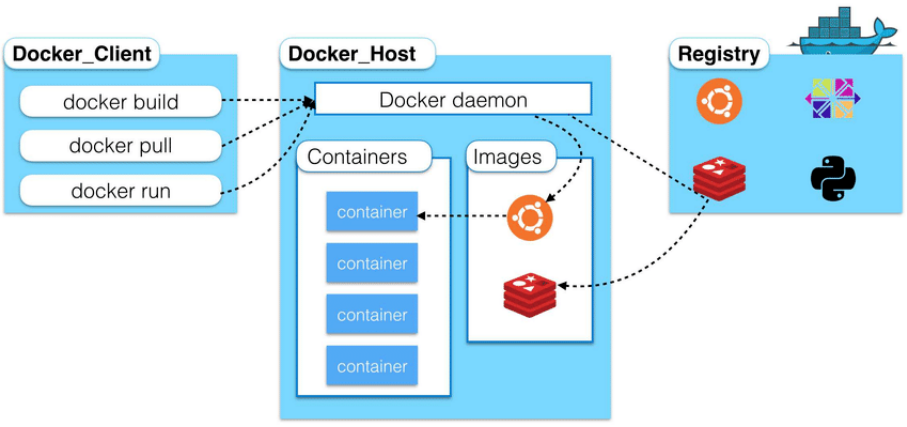


Figure 2.8 *Docker containerization architecture(Fawaz et al. 2016).*

2.3.2 Other technologies

There are several other containerization technologies that have emerged for the sack of serving a proper ecosystem Docker containers running in same environments. One of them is Docker swarm. Docker swarm is orchestration tool that binds the containers in a single cluster. The containers deployed in multiple host machines can join together to form a swarm cluster. The containers are referred to as worker node and the one container lead as manager node(Marathe, Gandhi, and J. M. Shah 2019).

Another orchestration technology for container is the kubernetes developed by Google. With kubernetes not only the Docker containers also the other containers can be managed and orchestrated. There are several manual processes involved in terms of managing, redeploying containerized application, kubernetes automates these processes to make sure the containers are always available and serving the purposes(J. Shah and Dubaria 2019).

2.4 Unikernel : A new containerization technology

2.4.1 Why we need a new technology for containerization?

In terms of running computational units in resource constrained devices has some challenges. Although solutions of some of these challenges are solved by the traditional containerization technology, there are some of the below challenges that needs to be met in terms deploying containers on resource constrained IOT devices:

- Low memory footprint
- Faster boot process
- Security

The Docker based containerization technology doesn't provide memory optimised images and ship a full fledged linux kernel inside which leads to a bigger container in size. The boot process of Docker container is much easier in terms of other containerization technology, but when booting a Docker container in ARM architecture needs additional configuration to make the boot process faster(CETIC 2013).

The crucial aspect of any production devices that are running computational units needs to offer less attack surfaces so that the container can be deployed in secure environment. For that a new approach is needed to remove all the unnecessary libraries from the container OS and reduce the attack surface(CETIC 2013).

For that considering the aspects of deploying computational units in resource constrained memory devices the thesis aims to introduce a new technology called unikernel. Unikernel coverts the computational units into single purpose images using library operating system(CETIC 2013).

Unikernel technologies provides the solutions for low memory footprint container images which has the faster boot process. The attack surface on unikernel images is significantly reduces as it ships the required libraries to the container and adds a shinked kernel to the images(CETIC 2013).

2.4.2 What is unikernel?

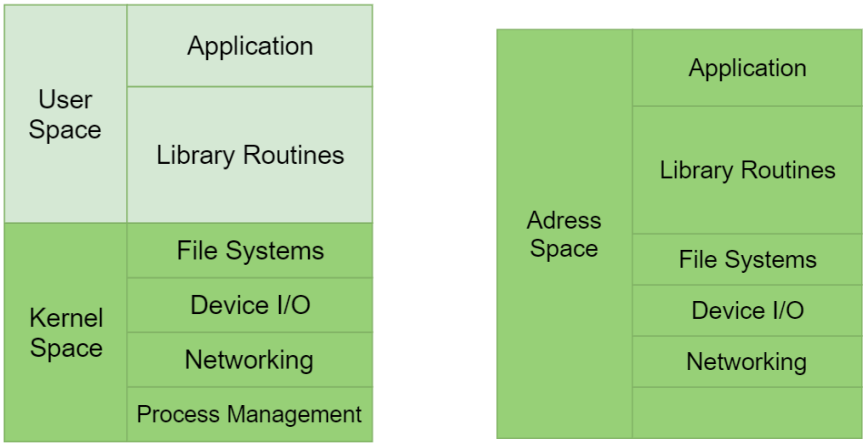
Unikernel are single purpose images developed with a series of libraries as library operating system. Unikernel images are constructed with the minimal set of operating system libraries that are needed to run the application. These libraries and the application code is then compiled into a fixed purpose unikernel images that can be booted up using a hypervisor(Mavridis and Karatza 2019).

2.4.3 Unikernel architecture

Any software application stack uses two address spaces, the user space and the kernel space. The kernel space contains the operating system and the operating system libraries that is shared with the application stack operations(CETIC 2013). The kernel space has all the core system functions like the file systems. memory management, disk. The user space holds the application code. In simpler terms the user space has the application code and the kernel space has all the code that are need to run the user space application code.

The application stack architecture is quite the opposite. In unikernel architecture there are no user space and kernel space instead there is only one address space. This address space holds the application codes along with the only necessary kernel space functions the application need to run(CETIC 2013). All these are compressed to singular image that can boot up itself without the need of a intervening full fledged operating system.

Figure 2.9(a) represents that in traditional operating systems needs the both user space and kernel space to operate. In unikernel architecture the address space holds the higher order application runtime and the lower order host operating system support. To achieve such computational units unikernel uses specialized compilation methods and that cross compiler adds the library operating system(libOS) function that is needed to run the application along with the application code and configuration. The libOS functions come in compilable form and the result leads to a unikernel image which can be run using hypervisors(CETIC 2013).



(a) Address space of a traditional application stack.

(b) Address space of a Unikernel application stack.

Figure 2.9 *Architectural comparison of traditional application stack and Unikernel application stack(Pavlicek 2017).*

2.4.4 Advantages of unikernel

Smaller footprint: The whole concept of unikernel technology is to get rid of the complexity of handling a full operating system and choose the minimalist approach of a compressed library operating system. The unikernel image contains only the functions that application stack needs to perform its operations. The end product unikernel images are smaller in size, usually couple of megabytes(Madhavapeddy and Scott 2014).

Faster boot process: As the unikernels are very small in size, they take seconds to boot. Also unikernel images are free of traditional operating system privilege processing of thousands of driver operations, it makes the boot process really faster comparing to the other virtual machine containing a full fledged general purpose operating system(Madhavapeddy and Scott 2014).

Technology for the cloud: Less resource utilization leads to less costing is Iaas cloud infrastructure. Running a full general purpose operating system in a VM instance just to run application can be costly while comparing a smaller and faster unikernel stack running the application in cloud(Madhavapeddy and Scott 2014).

Security perspective: In a unikernel image there is no command line interface, there is no unnecessary device drivers, there are no files that has authorization information, there are no access control system, databases are not connected to host machine. These limits the scope of attack platform to any production deployed unikernel(Madhavapeddy and Scott 2014).

2.5 State of the art unikernel technologies

2.5.1 Mirage OS

Mirage OS is one of the first unikernel state of the art technology, a library operating system for developing unikernel. The compilation code for Mirage OS can be developed in a general purpose operating system like Linux or MacOS. The compiled unikernel images can be run using XEN or KVM hypervisor (Madhavapeddy and Scott 2014). MirageOS can compile application stack written in Ocaml programming language. Ocaml has the libraries for networking and event driven concurrency threading. The compiled unikernel images can also be deployed with lightweight hypervisors like FreeBSD's BHyve, OpenBSD's VMM. Mirage OS is actively supported by open source community and the latest version of Mirage OS version 3.9.0 was released on October 27, 2020(CETIC 2013).

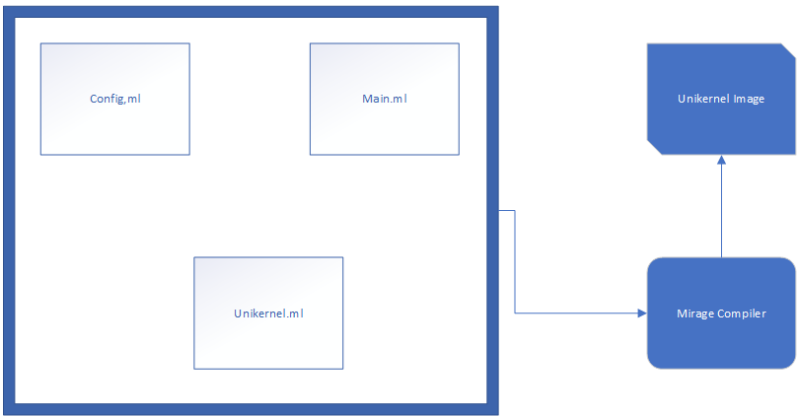


Figure 2.10 *Traditional OS vs MirageOS(Sapper 2015).*

2.5.2 Rumprun

The Rumprun unikernel is developed with driver component offered served by the rumpkernels. The rump kernels provides the necessary supports to run the applications as unikernels in hypervisor(Kantee 2015). Because of rump kernels minimalist nature it is easier to port several application to unikernel without the requirement of large scale modification. The unikernel technology supports application developed with programming languages like C, C++, Erlang, Go, Java, Javascript (node.js), Python, Ruby, Rust etc.

Rumprun unikernel can be deployed in both x86 and x64 hardware and can be run in XEN and KVM(CETIC 2013). Rumprun project is currently maintained under the NetBSD foundation project in GitHub.

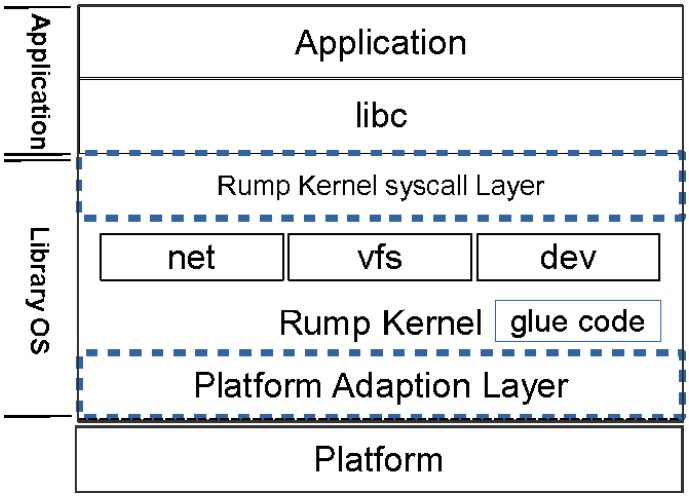


Figure 2.11 *Rumprun unikernel using rumpkernel(Elphinstone et al. 2017).*

Along with raw hardware support, rumprun provides hw platform which enables deploying the unikernels in embedded devices and cloud infrastructures. Its supports virtio drivers and provides deployment environment for ARM64 devices(Kantee and Andkjar 2016).

2.5.3 Include OS

IncludeOS is one of the well maintained unikernel projects initialized by y Alfred Bratterud from Oslo Metropolitan University and currently maintained by independent open source developer community and Alfred Bratterud. Initial idea behind IncludeOS was to develop an effective solo purposed OS to develop virtualized environment. The smaller footprint operating system holds the necessary operating system functions including the boot-loader and the image file. This specialized OS can be deployed to any hypervisor(Kot 2019).

Just like MirageOS support the Ocaml language, the IncludeOS unikernel is developed on C++. IncludeOS is extremely efficient with utilizing very little resources and during idle hours it lmits the CPU usage to zero. IncludeOS uses a GCC based toolchain so adding a simple ”include <os>” adds the whole tiny operating system(Bratterud et al. 2015). Figure 2.12 explains the build structure of Include OS unikernel.

Include OS received project funds from Horizon 2020 for porting unikernel to ARM architecture devices on the summer of 2018. The current stable version is version 0.15.0 released on May 9, 2020. Since the release the repository is actively maintained and new features and improvements are added continuously.

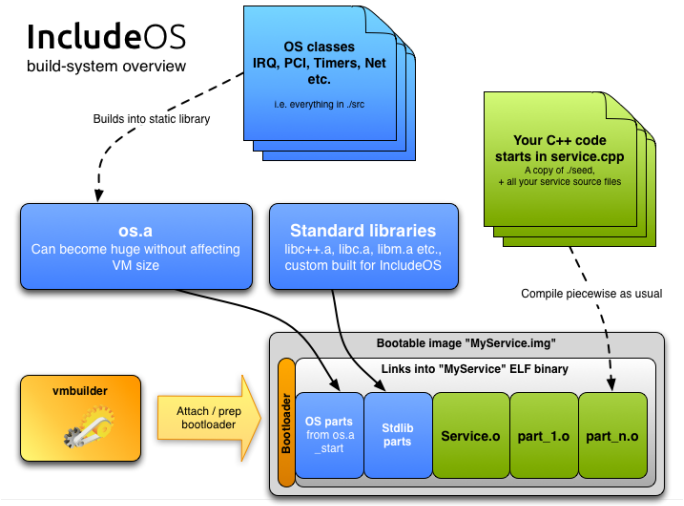


Figure 2.12 *IncludeOS build system(Kot 2019).*

2.5.4 OPS-NanoVM

OPS unikernel compiler provided by Nano VM is one of the industrial grade unikernel technology available in the market. Nano VM focused on creating a unikernel compiler that will provide ease of access to the non developers in terms of developing and deploying unikernels in different platforms including cloud platforms.

Most unikernel technologies offers unikernels in higher level languages, but OPS-NanoVm can compile application stack written any programming languages that has valid ELF binary. The tool has compilation packages for GOlang, Python 2.7, Python 3.6, Scheme, Forth, Java, PHP, NodeJS, Ruby, Lua, Perl etc. OPS-NanoVM provide supports for various operating systems like MacOS,Ubuntu,Debian, Fedora, Centos etc. OPS can run and deploy unikernel on bare metal, kubernetes, Firecracker, Hyper-V,VSphere, Vultr, Digital Ocean, GCloud, Microsoft Azure and Amazon AWS(Eyberg 2020).

OPS latest release in version 0.1.13 released on october 2020. OPS is currently under heavy development and the updates are released in every two months with newer features and improvements(Eyberg 2020).

2.5.5 OSv

OSv is an open source unikernel modular and guest operating system developed for running a single linux application as virtual machine on top of hypervisors. OSv has language runtime support for JVM, Python, Clojure, NodeJS, Ruby, Erlang etc and can run on hypervisors like QEMU, KVM, VMWare ESXi, VirtualBox, Firecracker etc. It provides runtime for cloud operators like AWS, GCE and OpenStack

etc(Kozaczuk 2020).

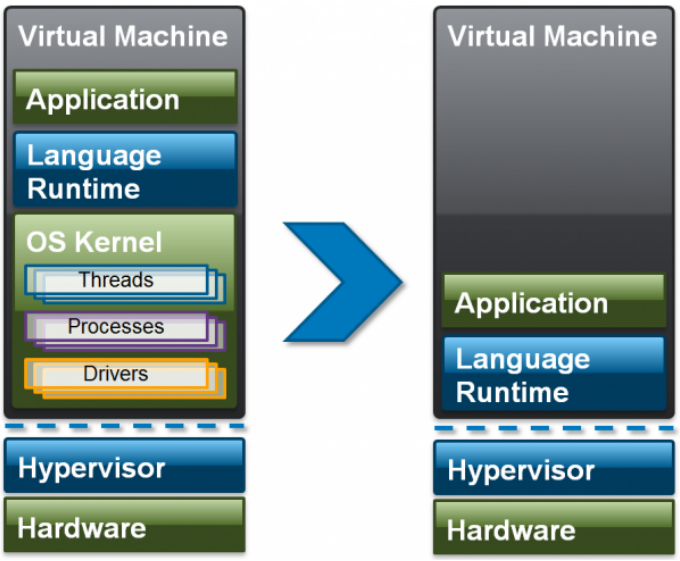


Figure 2.13 *OSv unikernel implementation(Penninkhof 2015).*

OSv is constructed with LibOS(library operating system) design system, each virtual machine running the application stack as unikernel has its own libOS. The core of OSv is written in c++ 11 and for file system it follows the UNIX VFS(virtual file system design)(Kivity et al. 2014).

Previously OSv was supported by cloudius systems. Now it is maintained by the open source community and regularly maintained. The current version 0.55.0 was released om May 12, 2020

2.6 Different unikernel echo system

Over the years along with unikernel technologies some of the unikernel echo system technologies also introduced. Most of these echo system technologies aim is to make the whole development process of the unikernel images more faster and easier. Also managing and administering solutions deployed in different environment requires a core management tools. Some these solutions evolves around providing a perfect solution to manage unikernel images deployed in different environments and clouds. This section is going to introduce some these echo system technologies.

2.6.1 Jitsu

Jitsu is an orchestration server that receives the DNS requests and automatically starts the boot process of the unikernel on an on demand basis(Skjegstad 2015). First a DNS query is sent to Jitsu and it starts the orhestration process by checking unikernel's availability in requested domain. If found the IP is then forwarded to

the requested client. Jitsu is so far tested on MirageOS and Rumprun unikernel technologies (Madhavapeddy, Leonard, et al. 2015).

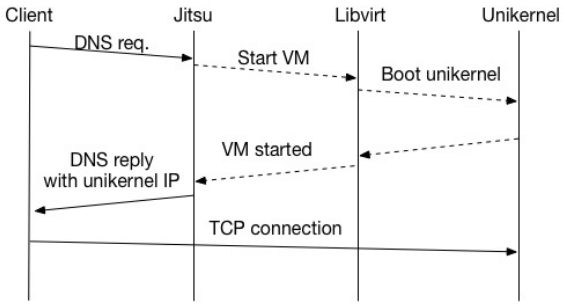


Figure 2.14 *Jitsu unikernel implementation(Madhavapeddy, Leonard, et al. 2015).*

2.6.2 Solo5

Solo5 is a sandboxing environment. It's a middleware purposed to attach unikernel application with the host machine. Initially solo5 was developed to facilitate MirageOS to run on KVM hypervisor by Danial Williams from IBM research. It has a public API to enable developers porting the running unikernels. It provides debugging of unikernel images. Solo5 runs on 64bit Linux FreeBSD and OpenBSD system(Williams, Lucina, and Koller 2020).

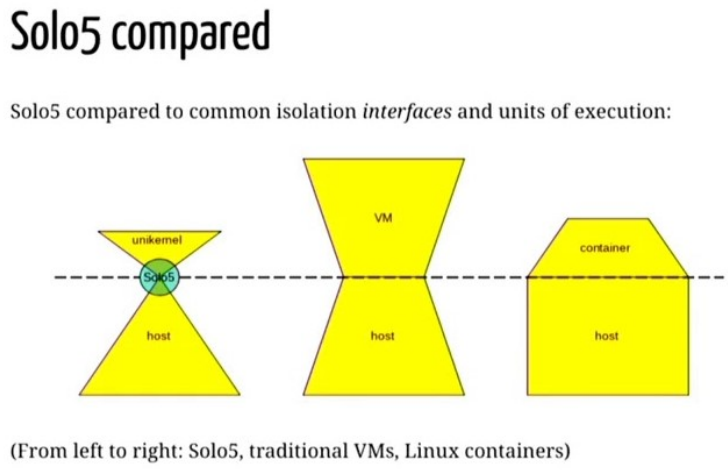


Figure 2.15 *Solo5 comparison(Lucina, Koller, and Williams 2019).*

2.6.3 Unik

Unik is an unikernel compilation tool that compiles the computational units to unikernels using various state of the art unikernel technologies. Unik converts the

unikernel images to bootable disk instead of creating a binary. Unik eases the development approach of unikernel and provides a Docker like interface to develop unikernels. Unik allows users to build unikernels using OSv, Rumprun, IncludeOS and MirageOS. Unik build bootable images can be deployed on VB, AWS cloud, GCloud, VMWare Vcenter, OpenStack, Photon, KVM and Qemu etc (Levine 2020). Unik has three major systems, an API server, a full fledged compiler and the hypervi-

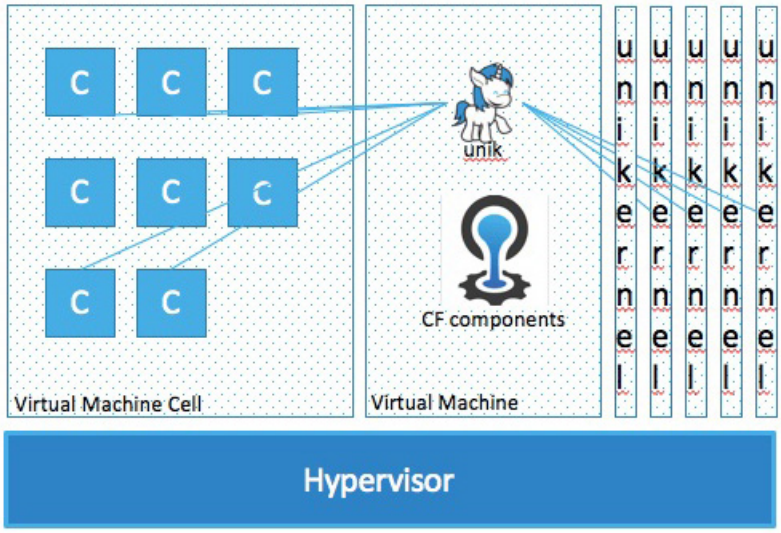


Figure 2.16 *Unik compilation example(STRUKHOFF 2017).*

sors or providers. The unik daemon architecture is explianed in below figure(Kohavi, Levine, and Weiss 2020).

The API server of unik receives the request for compiling an computaional unit from an HTTP client, then it decides which provider to choose in terms of compiling the computational unit. The compiler than compiles the computational unit using the appropriate provider. The end result is a bootable unikernel image of that compiler(AppFleet 2020).

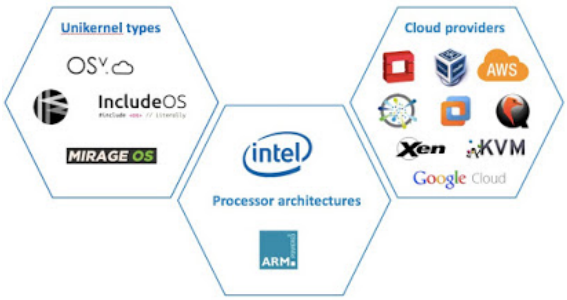


Figure 2.17 *Unik Daemon design(Levine 2020).*

2.7 Arrowhead Framework

2.7.1 Automation vs Orchestration

Our virtual unikernel images that are running the computational units are aimed to run in different distributed environments and embedded devices. Maintaining inter connectivity and inter portability between these computational units is a crucial task that often leads to complex overheads and additional workloads.

Both the concept of automation and orchestration is introduced to tackle these challenges. Automation refers to completing the functionality of computational units with the manually performed actions. Automation converts the time consuming, costly manual interventions and processes into more reliable, effective dynamic processes(KEEN 2019).

On the other hand orchestrations automates tasks of inter connectivity and integration between automation tasks running in a large scale virtual environment. Using the orchestration we can automate the management, scheduling and coordinating the computational units running in virtualized distributed systems and it enables the whole system to support the processes for a much larger workflow(KEEN 2019).

2.7.2 Need of orchestration framework: Unikernel perspective

Any virtual environment requires portability to maintain an inter connectivity between other virtual environment so that the computational units that are deployed in that virtual environment can be used as much as possible.

The unikernel containers having application units can provide higher order services while running in hundreds of embedded devices in production scenario. In that scenario the critical operations depends on the availability of the application units. Traditionally any computational unit requires updates, modifications, improvements in order meet the end purposes demands. A down time in any of the unikernel container node will lead to the disruption of the whole work process of a production environment. In that case the availability of the unikernel units functionality becomes an crucial aspect even during down time.

Lets consider a scenario like the below figure(Fig. 2.18) where a unikernel container node provides two different services. The node can handle hundreds of API calls in seconds and provide the necessary functionality for the whole process to run. Now the resources of these containers are not infinite. As the request increases the resources gradually decreases. For that the containers need to be updated to another environment so that it can meet the end users limit. Now during the upgrade the

system's critical services that are running in that node will be unavailable which is unacceptable in modern days production scenarios. This is where the orchestration comes into play. We can distribute the critical processes in several nodes where the services the parent node offers will be available. When a service provider node is down the orchestrator looks for other providers. When a new provider node is found, it offers the necessary inter-connectivity rules to provide the requested services. The orchestrator works as a mediator to introduce the requested service from the available provider node.

The whole orchestration process removes the management complexity from the whole system by providing services like service discovery, service registry, authorization and orchestration etc.

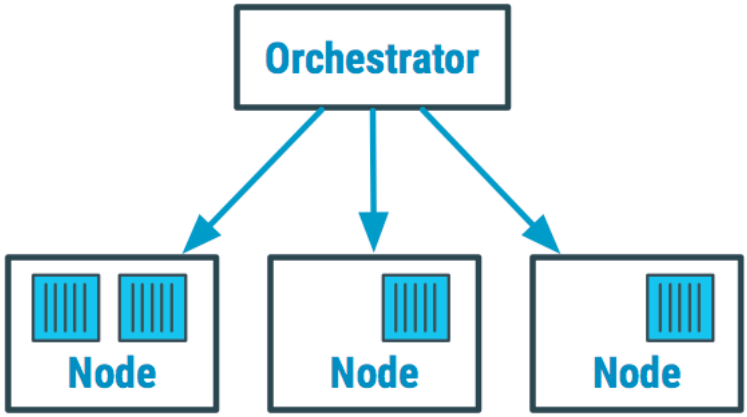


Figure 2.18 *Example orchestration process(Padmanabhan and Bashir 2018).*

2.7.3 Arrowhead Framework: The Solution

Arrowhead framework is a service oriented architecture (SOA) based orchestration framework developed on a local cloud concept to create a full scalable automation system. One of the core purpose of arrowhead framework is to reduce the challenges of inter-connectivity between embedded devices(C. Paniagua, Eliasson, and J. Delsing 2020). The main objective of arrowhead framework is to provide,

- Necessary functionality for access control limitation(ACL) with advanced certification based authentication system.
- Service level orchestration of systems with dynamic processes without the need of complex manual configurations and higher order additional computation unit to manage the orchestration process.
- Continuous resource allocation and communication between systems.

- Advanced functionality to perform dynamic orchestrations in inter cloud scenario.

The whole concept of arrowhead framework evolved around with four different entities shown in Figure 2.20 ,

- Using **Service** the framework inter change information between networked systems, consumes the information's or data from provider systems(C. Paniagua, Eliasson, and J. Delsing 2020).
- **System** is the micro units of the whole computational unit that either provides or consumes the requested services(C. Paniagua, Eliasson, and J. Delsing 2020).

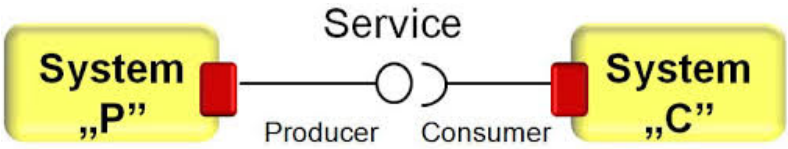


Figure 2.19 *SOA architecture of Arrowhead framework(Hegedus, P. Varga, and Frankó 2018).*

- The **device** provides the infrastructural and hosting environments for the systems to maintain communications and automation functionalities between themselves(C. Paniagua, Eliasson, and J. Delsing 2020).
- The **local cloud** is the network of system of systems that provide the functionality of full fledged IOT automation using its mandatory code systems and services(C. Paniagua, Eliasson, and J. Delsing 2020).

The Arrowhead local cloud uses mandatory core systems to perform operational activities like authentication and authorizing the systems, discovery of services, registering the services to the cloud and perform orchestration of services (C. Paniagua, Eliasson, and J. Delsing 2020).

2.7.4 Mandatory core systems and services

The are three mandatory core systems of arrowhead framework local cloud and some additional supporting core systems. The mandatory core systems are,

1. Service registry.
2. Authorization system.

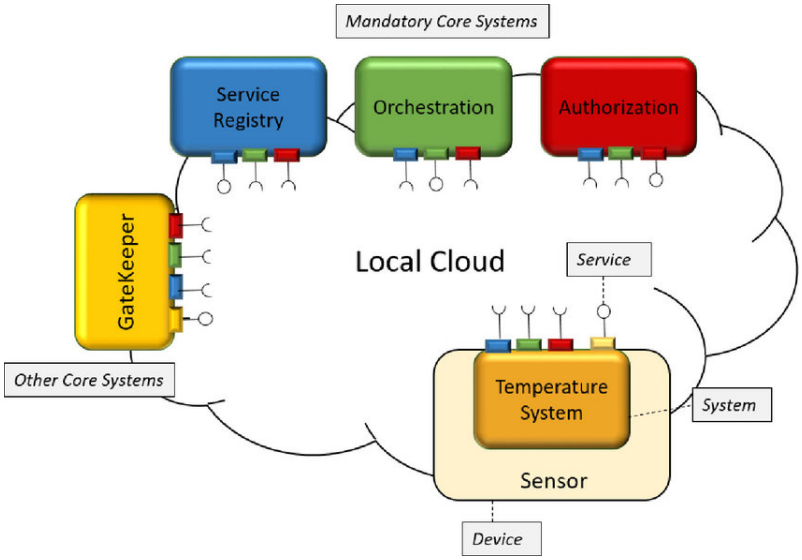


Figure 2.20 *Example Arrowhead local cloud(Cristina Paniagua and Jerker Delsing 2020).*

3. Orchestration system.

The **service registry** core systems job is to supply the information whether the requested service is online or not. The service registry store the connectivity configurations of the systems and their services. And upon request the systems announce their availability and services using service registry. This core systems also registers all the systems including the other core systems to the local cloud network(Hegedűs et al. 2016).

The **authorization system** job is to perform advance authorization and authentications inside the local cloud. The consumer system need to have correct authorization to consume the requested services. The authorization core system evaluate the authorization certificates and grant the approval to the consumer to consume the service. The crucial task of certificate handling also is done inside the authorization core system(Hegedűs et al. 2016).

The **orchestration system** is the most crucial core system in arrowhead local cloud system. Upon receiving the service requests from the consumer, it fetches a list of service provider systems from the service registry. Then it filters out the service providers based on the availability and authorization. Then it does the final match making and provides the suitable system provide to the consumer. The orchestration core system has the capabilities of expanding its service matchmaking process in interconnected different cloud.

Besides the core system, there are some supporting core system as depicted in the Fig 2.21.

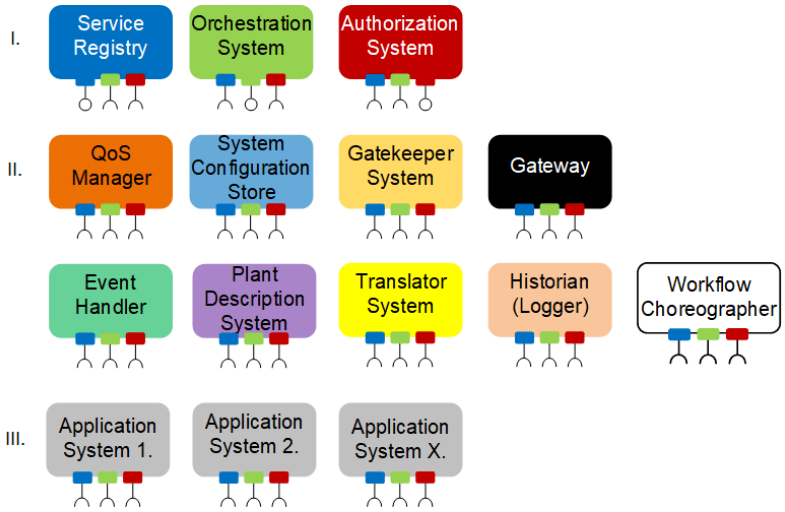


Figure 2.21 *Core systems of Arrowhead framework(Kozma, Soos, and Pal Varga 2019).*

2.7.5 Application services and systems

Application systems and the services are the user side application that connects with the core systems of the arrowhead local clouds. Using the REST API interface of arrowhead core systems the arrowhead compliant applications systems submit a service registry form. The service registry form includes the configuration of connectivity and authorization. Service registry evaluates the form and of authorized by the authorization core systems the provider system get registered. When the service registry receives a service request form from the orchestrator, it provides these configurations to the orchestrator so that the requester consumer system can consume these services.

The figure 2.22 shows a quick example on arrowhead workflow.

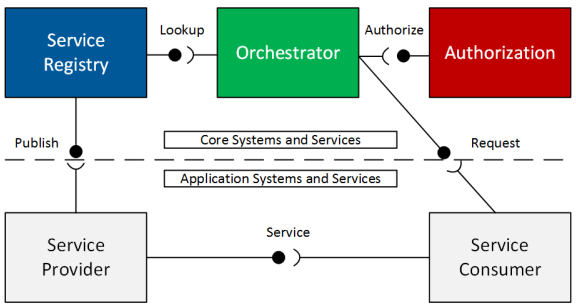


Figure 2.22 *Example workflow of arrowhead framework(J. Delsing 2017).*

3 Proof of Concept

As described on the objectives section, this chapter is going to define how the thesis fulfilled those objectives. The key approaches to the proof of concept is divided into two parts. On the first part the solution evaluates the new state of the art technology unikernel. Then on the second part it evaluates the advantages of orchestrating and managing the unikernel images deployed in ARM devices. Along the whole proof of concept has four experimentation model. These experimentation's are developed to evaluate the both advantages and disadvantages of the technology stack.

3.1 Framework of Possible Approaches

On the first stage of the proof of concept, to compare the build process of unikernel technologies, a special environment is setup where the unikernel application is run and using the different application stack. The environment evaluates the build process of each of the unikernel technologies. Then the developed unikernel images is added in a test environment where the unikernel images are evaluated using two different metrics like boot time, memory footprint etc. Then the later phase the implementation forwards to deploying the unikernel image to an ARM device like Raspberry Pi.

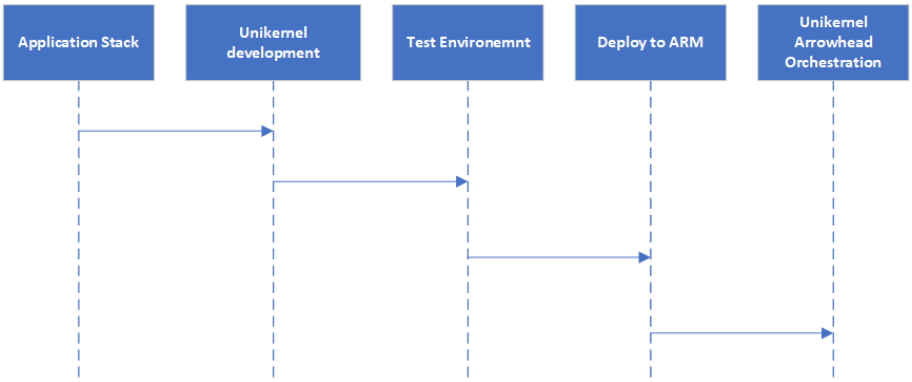


Figure 3.1 *Process framework of proof of concept solution.*

On the second stage of the proof of concept solution a different experimentation environment is setup where arrowhead framework is setup in a remote server and it performs orchestration in an inter cloud scenario of service that are deployed in ARM 64 device. The experimentation evaluates the whole orchestration process of arrowhead framework and its advantages, also disadvantages.

3.2 Environment setup: unikernel Implementation

As most of the unikernel image development requires linux operating system and uses its libraries. To create a whole virtual system where the unikernel images is developed and deployed in hypervisors, a remote SSH server virtual machine is created using VMWare Esxi with Ubuntu 18.04.4 LTS ISO. The virtual machine offers a GNU/Linux operating system kernel. The virtula machine is mounted on forwarded port on a remote server. The linux kernel is accessed using MobaXterm Xserver and SSH client. The CPU of the virtul machine is Intel(R) Xeon(R) CPU which has 24 cores. To give more processing power to the virtual machine 12GB RAM and 32GB disk space added.

The environment setup required an additional SFTP(FTP over SSH) server. The reason for choosing SFTP over FTPS(FTP with SSH) as it provides a secure file transfer system in inexpensive way (Liu Xia et al. 2010). Also another reason for choosing SFTP server as the implementations and virtual machine required user authentication, unlimited file transfer and controlled port usage.

Table 3.1 *Environment Configurations.*

#	Name	#	Type	#	Version
1.	Operating System	1.	GNU/Linux	1.	18.04.4 LTS
2.	Image	2.	Linux Kernel	2.	4.15.0-122-generic
3.	Ram	3.	12 GB		
4.	CPU	4.	Intel(R) Xeon(R) CPU	4.	E5-2620 v4
5.	Disk	5.	32GB		

For SFTP server setup there were couple of options to choose from. MobaXterm Xserver provides a SFTP server which pops up in SSH client and using that the remote SSH server file system can be accessed. FileZilla another SSH client also offers interactive graphical user interface, which can be used to access the SSH server and upload the application files. For the experiment setup, SFTP server need to have transfer queue scheduling and automated file transfer used. Considering that, another SSH client is used called smartSSH. Using smartSSH an automated file transfer is imlemented. For automated file transfer an MFT(Managed file transfer) server is developed using the NPM(node package manager) version of smartSSH. The client is added to the MFT server and whenever the file is changed it's queued to the SFTP server and later SFTP server upload the files to the remote SSH server.

As the virtual machine only provide command line linux kernel, for code debugging the implantation used GNU nano v5.3(Allegretta 2001). Unikernel implementation required handling of application stack developed in different programming languages. For that the nanorc is configured to support auto indentation and no

wrap feature.

The whole environment setup process is depicted in below figure,

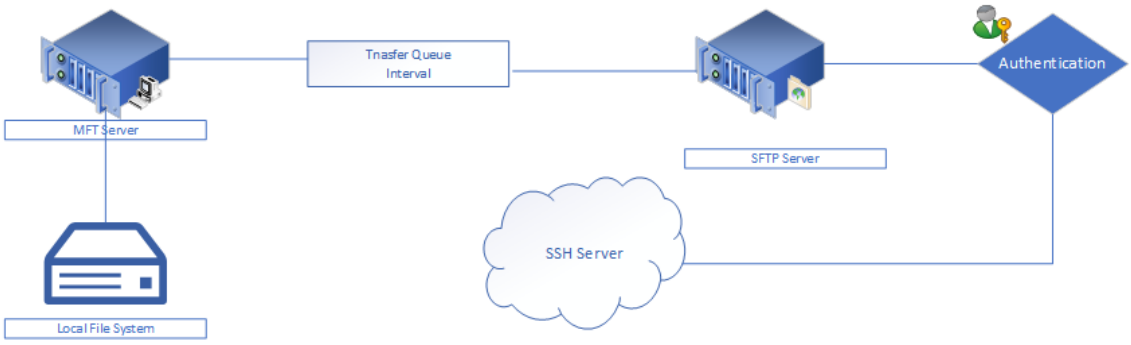


Figure 3.2 *Environment setup proof of concept solution.*

3.3 Experiment I: Unikernel development scope

3.3.1 Unikernel technology solutions choices

There are quite a lot of unikernel compiler technology available and some of the them are still in development stage. Not all the unikernel technology support all the programming languages. Unfortunately still most of them are programming language. There are some unikernel technologies that support multiple core languages. Based on that we have categorised the unikernel technologies into two categories,

1. Global unikernel compiler
2. Program specific unikernel compiler

To categorize the unikernel solutions we have followed the below comparison model. The comparison model is developed based on our analysis on chapter 2.4.5 ”State of the Art unikernel technologies”. There are some key factors that needed to extracted form the comparison model to choose the unikernel technologies that the experimentation will explore.

- The unikernel technology compiler can be run on GNU Linux specialised kernel. The reason for this the experimentation aims to follow a single unique approach to test the performance, footprint and security aspect of the unikernel images in a unified way.
- The unikernel technologies can deploy the unikernel images of application stack in different hypervisors like XEN, KVM, VBox, GCloud, AWS etc. The experimentation aims to choose one of these hypervisor, so that one unified

approach is followed. From different hypervisors the experimentaion choose the KVM. KVM(Kernel Based Virtual Machine) can run multiple virtual machine in single host and also it is compatible with the environment setup of the experimentation.

- The unikernel technology has the compiler support to compile HTTP server of the computational unit.

Project	Language	Middleware	ARM Support	Deployment
Mirage OS	OCaml	MiniOs. Solo5	Yes	XEN, KVM
Rumprun	C, C++, Erlang, Go, Java, Javascript (node.js), Python, Ruby and Rust	Rump Kernel	Yes	XEN, KVM
ClickOS	C	MiniOS	No	XEN
OSv	C, C++, Java, Node.js	Java	Yes	XEN, KVM, AWS
OPS	Golang, Node, Python	Golang	No	VBox, AWS, GCloud
IncludeOS	C++	MiniOS, Solo5	No	XEN
Unik	GO, Java, Node, Python	Golang	Yes	VBox

Figure 3.3 *Comparison model of unikernel technologies.*

According to the comparison model, the categorization goes as follows,

Table 3.2 *Categorization of Unikernel technologies*

#	Program Specific Unikernel compiler	#	Global unikernel compiler
1.	MirageOS	1.	OSv
2.	ClickOS	2.	OPS Nano VM
3.	IncludeOS	3.	Rumprun
4.		4.	Unik

Mirage OS, OSv, OPS Nano VM, Rumprun has the support for deploying unikernel images in KVM hypervisor. The middleware used in compiling the unikernel images of these unikernel technologies also support installation in limited GNU Linux kernel.

For developing the unikernel images the experimentation followed two approaches, using the traditional build processes of the unikernel technologies and the second one developing the images using the Unik compiler. Unik build the unikernel images using the compiler of specific unikernel technologies. For example, to compile

a server application developed with NodeJS it uses rumprun to create the unikernel image, for Java application it uses the OSv compiler to compile the application stack to an unikernel image.

3.3.2 Hierarchy of unikernel build process

Each of the unikernel technologies have different build processes and uses different strategies to deliver the end product. In the build process the unikernel technologies almost all of them have four units. These are combined to compile the unikernel images.

1. Application Code
2. Config Files
3. Runtime Compiler
4. Unikernel image

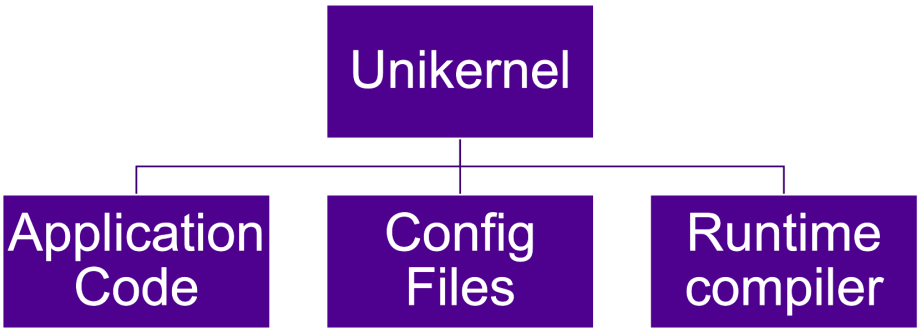


Figure 3.4 *Build process analysis of unikernel technologies.*

The **application code** holds all the codes that are needed to show end result. The application code file includes the modules and libraries needed to run the application.

The **config files** holds a set of instructions which the compiler interprets and based on those instructions it compiles the unikernel image. There can be multiple config files.

Runtime Compiler compiles the application code based on config file to an unikernel image.

The **unikernel image** is the end product after the compilation process.

3.3.3 Program Specific Unikernel development

For program specific unikernel development the experimentation tested Mirage OS. MirageOS can compile application code developed with Ocaml programming language.

3.3.1 MirageOS Unikernel development

MirageOS has Ocaml runtime libraries that forms linkage between the runtime compiler to compile the application code. MirageOS can be installed in any linux operating systems and some lates MacOS operating system distribution. OPAM tool manages the runtime libraries of MirageOS unikernel compiler.

The installation process of MirageOS is as follows,

- First step is to install OPAM version 2.0. OPAM can be installed using Linux package manger.
- Our environment setup has lower version of ubuntu, it cannot install Mirage using the OPAM package manager. For that, this part of installation process required creating a custom personal package achive (PPA).
- After creating the PPA repository, the installation process some additional libraries like ocaml native compiler, caml4 extra, ocaml etc.
- Now the new installed OPAM can install the Mirage

Configuration setup

The experimentation used an example Mirage http server by the mirage repository. The server is reconfigured to return the status code as response when requesting on the root IP of the server. The config.ml start with command for opening the command, then a main function that calls the foreign packages. The function also registers a job that will run the web server.

Creating the unikernel

The application codes are written in unikernel.ml file. Using the mirage configure command it creates a main.ml file that has the entry point of running the unikernel and the list of Ocaml packages library needed to run the unikernel.

By default, the server runs on 10.0.0.2 unless the network ipv4 is configured. The implementation is tested by pinging to 10.0.0.2 in port 8080.

The unikernel image is later deployed to an ARM architecture. The experimentation tested the Mirage OS unikernel deployment in Raspberry Pi 4 Model B and Raspberry Pi 3 Model B+. The unikernel can be successfully booted up in raspberry pi 4. In raspberry pi 3 the boot process fails with compiler compatibility issue.

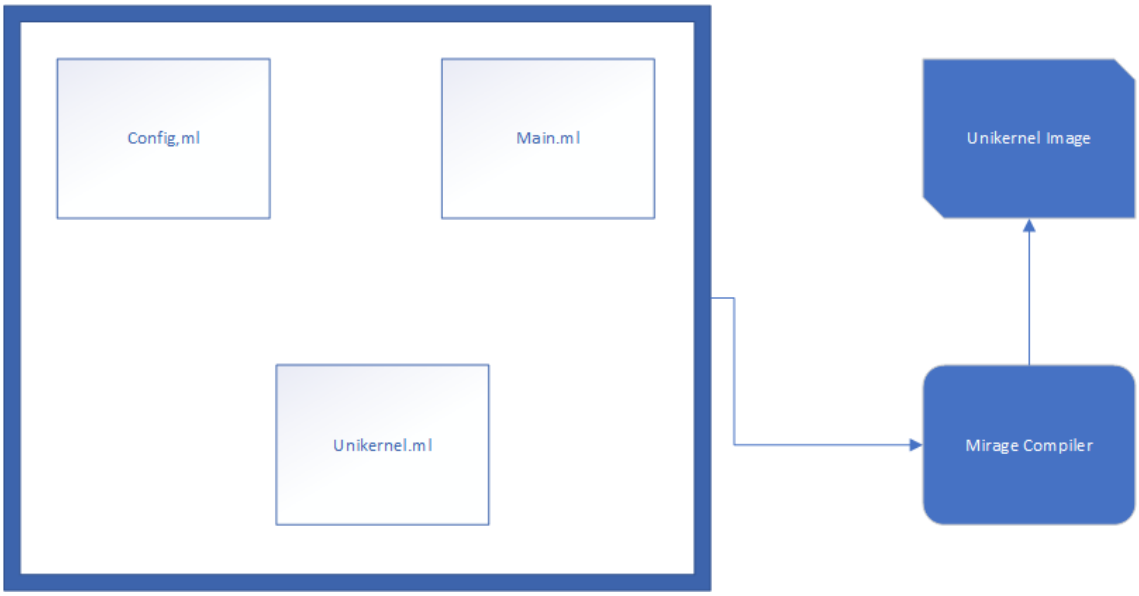


Figure 3.5 *Unikernel development with Mirage*

3.3.4 Global unikernel compiler based development

For global unikernel development the experimentation goes with the development of OPS Nano VM, OSv unikernel development. Each of these unikernel technology can compile application stack developed in different languages.

3.4.1 OPS Nano VM Unikernel development

OPS Nano Vm offers a very simplistic way to compile application stack to unikernel. One of the greatest barrier in global unikernel compiler based development is the absence of required compiler package. Nano VM is constantly adding new compiler packages for modern database and web server technologies. The core concept of OPS unikernel development is to ease the complex development process of unikernel technologies to a more user oriented process with its pre-built compiler packages.

Installation of OPS

OPS can be installed in MacOS, Debian, Fedora, Ubuntu and centOS operating systems. The installation process is as follows,

- OPS Nano VM uses Qumu emulator and KVM hypervisor to run the virtual machine.
- Qemu and KVM can be installed from linux package manger.
- OPS provide installation script in linux shell executable file format(sh).
- Using the URL syntax and curl the sh file can be executed.

- The script file takes care of the rest and install OPS.

Configuration setup

OPS Nano VM uses a config.json file to all the code execution command. The config file imcludes directory structure, environment variable, server configuration etc. Below figure shows a sample example config.json file(Eyberg 2020).

```
{
    "Files": ["ex.js"],
    "Dirs": ["src"],
    "Args": ["ex.js "],
    "Env": {
        "NODE_DEBUG": "*",
        "NODE_DEBUG_NATIVE": "*"
    },
    "MapsDirs": {
        "src": "/myapp/code"
    },
    "Boot": "./staging/boot2.img",
    "Kernel": "./staging/stage4.img",
    "Mkfs": "./staging/mkfs",
    "DiskImage": "disk-image",
    "NameServer": "10.8.0.1",
    "RunConfig": {
        "Verbose": true,
        "Bridged": true,
        "Ports": [8008],
        "Memory": "2G"
    }
}
```

Figure 3.6 *OPS Nano VM sample config file*

Creating the Unikernel

The experimentation tested two web server developed with two different programming languages. The first server is developed with Python and Flask and the second server is developed with NodeJS. Both the server returns a JSON response of the server status while sending a REST call on the root. The configuration file of the python application is,

```
{ "Env": { "FLASK_APP": "rest-server.py" },
  "MapDirs": {"/home/user/.local/*": "/.local" },
  "Args": ["/.local/bin/flask", "run", "--port=5000",
  "--host=0.0.0.0"],
  "Files": ["rest-server.py"],
  "Boot": "./staging/ops-python.img",
    "Kernel": "./staging/op-python.img",
    "Mkfs": "./staging/mkfs",
    "DiskImage": "disk-image",
}
```

The server application code is added in the root path and on the env the instruction is given on which file to compile. The MapDir here is mapping the file

directory from the local directory to a directory inside the virtual machine. The Args is getting the instruction to get the flask from the virtual machine local bin directory and compile the application. There are two packages available to compile the python appplication Python 2.7.15rc1 and Python 3.6.7. As the proof of concept solution is developed with Python 2.7, the experimentation used the Python 2.7.15rc1 package to compile the unikernel image. A complete list of available OPS Nano VM package is added in table 3.3.

The configuration file of the NodeJS application is as follows,

```
{
    "Files": ["app.js"],
    "Args": ["app.js "],
    "Env": {
        "NODE_DEBUG": "*",
        "NODE_DEBUG_NATIVE": "*"
    },
    "MapsDirs": {
        "src": "app.js"
    },
    "Boot": "./staging/ops-node.img",
    "Kernel": "./staging/op-node.img",
    "Mkfs": "./staging/mkfs",
    "DiskImage": "disk-image",
    "NameServer": "127.0.0.1",
    "RunConfig": {
        "Verbose": true,
        "Bridged": true,
        "Ports": [3000],
        "Memory": "2G"
    }
}
```

The NodeJS application server code is added in a root app.js file. There are couple of compiler packages are available for compiling a Node server application based on different node version. The experimentation compiled the unikernel image using the node v14.2.0 package. The application stack that the experimentation tested doesn't include a complex node modules as a clear instruction is missing in the documentation of OPS Nano Vm.

The Figure 3.7 depicts the overall build process of unikernel development with OPS Nano VM.

Table 3.3 *OPS Nano VM package List*

#	Name	#	Version	#	Language	#	Run Time
1.	lua 5.2.4	1.	5.2.4	1.	lua	1.	lua
2.	php 7.2.13	2.	7.2.13	2.	php	2.	php
3.	java 1.8.0 191	3.	1.8.0 191	3.	java 8	3.	JRE 8
4.	node v13.6.0	4.	13.6.0	4.	javascript	4.	node v8
5.	mongodb 4.0.6	5.	4.0.6	5.	c++	5.	c++
6.	mosquitto 1.5.7	6.	1.5.7	6.	c	6.	mosquitto r
7.	memcached 1.5.15	7.	1.5.15	7.	c	7.	memcached
8.	hiawatha 10.9	8.	10.9	8.	c	8.	hiawatha
9.	kache 1.0.1	9.	1.0.1	9.	go	9.	kache
10.	nginx 1.15.6	10.	1.15.6	10.	nginx	10.	nginx
11.	python 2.7.15rc1	11.	2.7.15rc1	11.	python	11.	python
12.	sonic 1.1.0	12.	1.1.0	12.	rust	12.	sonic
13.	lwan 0.0.1	13.	0.0.1	13.	c	13.	lwan
14.	php 7.3.5	14.	7.3.5	14.	php	14.	php
15.	scala 2.13.0	15.	2.13.0	15.	scala	15.	scala
16.	neo4j 3.5.7	16.	3.5.7	16.	java	16.	jvm
17.	tomcat 9.0.19	17.	9.0.19	17.	java	17.	jvm
18.	mysql 5.7.29	18.	5.7.29	18.	c++	18.	mysql
19.	node v14.2.0	19.	14.2.0	19.	javascript	19.	node
20.	clojure 1.10.0	20.	1.10.0	20.	clojure	20.	clojure
21.	elixir 1.8.1	21.	1.8.1	21.	elixir	21.	elixir
22.	java 11.0.8	22.	11.0.8	22.	java 11	22.	openjdk 11
23.	node v12.13.0	23.	12.13.0	23.	javascript	23.	node v8
24.	influxdb 1.7.0	24.	1.7.0	24.	go	24.	influxdb
25.	coredns 1.5.0	25.	1.5.0	25.	go	25.	coredns
26.	gforth 0.7.3	26.	0.7.3	26.	gforth	26.	gforth
27.	bind 9.16.4	27.	9.16.4	27.	c	27.	bind9
28.	loki 0.1.0	28.	0.1.0	28.	go	28.	loki
29.	R 3.4.4	29.	3.4.4	29.	R	29.	R
30.	tarantool 1.10.2	30.	1.10.2	30.	tarantool	30.	tarantool
31.	perl 5.22.1	31.	5.22.1	31.	perl	31.	perl
32.	keydb 5.0.2	32.	5.0.2	32.	c	32.	keydb
33.	openresty 1.15.8	33.	1.15.8	33.	c	33.	openresty
34.	openldap 2.4.50	34.	2.4.50	34.	c	34.	openldap
35.	bind 9.13.4	35.	9.13.4	35.	c	35.	bind9
36.	caddy 1.0.0	36.	1.0.0	36.	go	36.	caddy
37.	ruby 2.7.0	37.	2.7.0	37.	ruby	37.	ruby
38.	vector 0.9.0	38.	0.9.0	38.	rust	38.	rust
39.	java 9.0.4	39.	9.0.4	39.	java 9	39.	JRE 9
40.	ruby 2.5.1	40.	2.5.1	40.	ruby	40.	ruby
41.	gnatsd 1.4.1	41.	1.4.1	41.	go	41.	gnatsd
42.	cockroachdb 19.2.6	42.	19.2.6	42.	go	42.	go
43.	spark 3.0.0	43.	3.0.0	43.	java	43.	jvm
44.	redis 5.0.5	44.	5.0.5	44.	c	44.	redis
45.	node v11.5.0	45.	11.5.0	45.	javascript	45.	node v8
46.	python 3.6.7	46.	3.6.7	46.	python3	46.	python3
47.	ruby 2.3.1	47.	2.3.1	47.	ruby	47.	ruby
48.	haproxy 1.8.8	48.	1.8.8	48.		48.	haproxy
49.	grafana 7.1.5	49.	7.1.5	49.	go	49.	go
50.	wasmer 0.1.4	50.	0.1.4	50.	rust	50.	wasmer

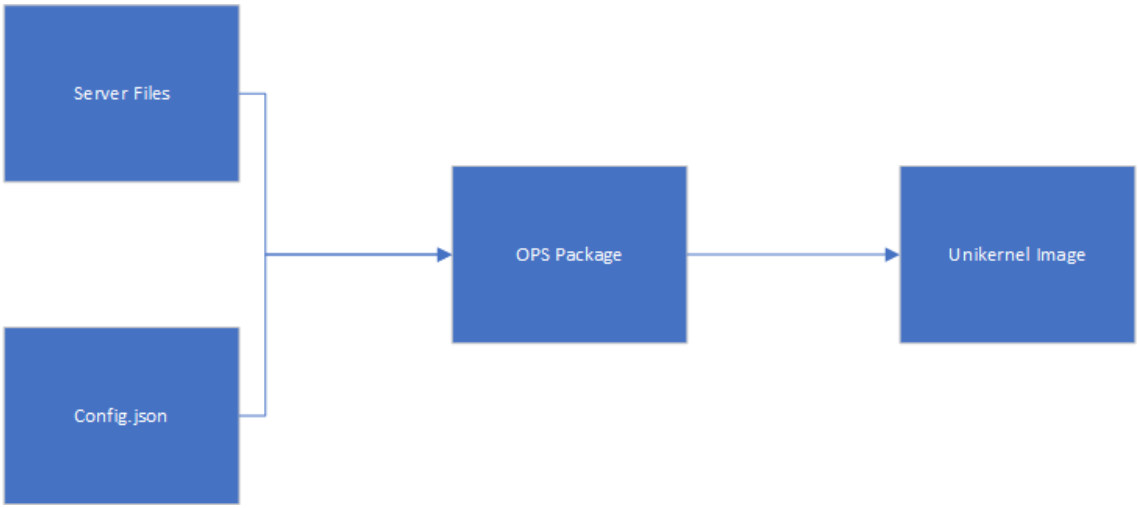


Figure 3.7 *OPS Nano VM unikernel build process*

3.4.2 OSv unikernel development

OSv unikernel development approach also follows a similar approach like OPS Nano VM. Instead of creating an unikernel using a heavy configuration based approach it introduced a command line tool called capstan. Capstan is developed on Golang and can compile unikernel in two different way,

- Along with application code a capstanfile can be added just like Dockerfile which holds the information and set of instruction that can be used to compile the application into unikernel.
- Another approach is the package mode. In the package mode capstan compiles the application using pre-compiled package images, just like approach of creating container using Docker compose.

Capstan added some level of advantages in preparing the inikernel images. Like using pre built packages capstan can create and run unikernel very quickly, using capstanfile and introducing some sets of compilation instructions developer can create their own base image.

Previously capstan and its packages were maintained by Mikelagelo projects and the packages can be pulled from the Mikelagelo Amazon S3 bucket. Now it is maintained by open source contributors and the latest packages can be pulled from the OSv github repsitory.

Installation of Capstan

Capstan requires qemu hypervisor to run the unikernel, so the first step to install capstan is to install the Qemu. As our environment setup is running on a low resource GNU/Linux kernel based os, the experimentation followed to install

Capstan from the source. Another approach is to downloading the capstan and it in the home bin directory. After assigning the bin path capstan command can be used. The source installation process is as follows,

- To install capstan from the source, Golang needed to be configured first. To install and export the golang path the experimentatin used a bash script that pulled the golang file and exported the path.
- After cloning the capstan repository the capstan installation is done with the go install command.

Configuration setup

As the experimentation already tested the development of NodeJS, Python and Ocaml unikernel implementation, for OSv a Java web server application is developed. The server application returns the status code of the response when sending a GET request in the root path.

The experimentation used pre build OSv images to compile the application. a run.yaml file needs to be added that holds the compilation instructions. A basic run.yaml instruction is as follows,

```
runtime: python
config_set:
    rest-server:
        main: /server.py
        env:
            PORT: 4000
```

runtime defines the base runtime image that will be used to compile the application. **configset** is the entry point of the unikernel image

Creating the unikernel

The application code is added in the root directory. On the root directory the capstan init command generated one addition file in the meta folder called package.yaml. The meta folder also holds the run.yaml file. The init command generates a the package.yaml file which looks as follows,

```
name: java-app
title: Java Server Application
author: Alom-Md-towfiqul
version: "1.0"
require:
```

– osv.openjdk10-java-base
– osv.run-java

Table 3.4 *Some OSv package List*

#	Name	#	Creation Date	#	Description	#	Version
1.	osv.bootstrap	1.	2020-05-12 19:30	1.	OSv Bootstrap	1.	0.55.0
2.	osv.httpserver-api	2.	2020-05-12 19:34	2.	OSv http server	2.	0.55.0
3.	osv.python2-from-host	3.	2019-10-20 21:48	3.	Python 2.7 8	3.	2.7.16
4.	osv.python3x	4.	2019-03-12 03:11	4.	python3x	4.	3.6.6
5.	osv.openjdk10-java-base	5.	2018-05-30 04:03	5.	Java	5.	10.0.1
6.	osv.run-java	6.	2020-05-12 19:33	6.	Java wrapper	6.	0.55.0
7.	osv.node-js	7.	2018-05-30 04:03	7.	NodeJS	7.	8.11.2

There are many available packages for composing the application stack to OSv unikernel. The experiementation used osv.openjdk10-java-base package to compile the Java application. The run command fetch the optional arguments from run.yaml like environment port and fetch the compose configuration from package.yaml to boot the unikernel.

Capstan adds the kernel for unikernel image from the OSv loader which resides in the capstan repository folder. For the experimentation, the latest OSv kernel needed to be added in the osv-loader directory of capstan. The new OSv kernel is available on the latest releases in the OSv github repository.

The Figure 3.8 depicts the overall build process of unikernel development with OPS OSv.

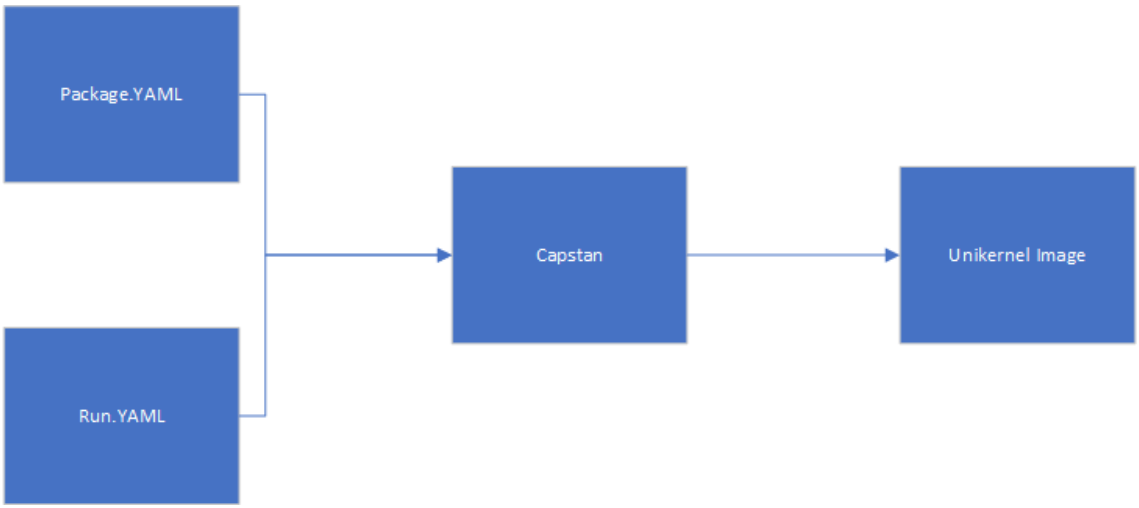


Figure 3.8 *OSv unikernel build process*

3.4 Experiment II: Analysis on boot time, footprint monitoring with test cases

The claim of unikernel technology is that it can boot fast, provide small memory footprint and can perform better in HTTP requests. The experimentation II aims to analyse the developed unikernel images in two different matrix,

- Boot time
- Memory footprint

3.4.1 Test setup

During the experimentation process all the unikernel technologies were developed on a same environment. The technologies used the same cpu, os and RAM configuration. All the unikernel technologies were booted up using the same Qemu KVM hypervisor. The unified system led the experimentation to get a clear picture on the unikernel images status in different test scenarios. One particular factor that can affect the result is that, the RAM consumption during the boot process was not make finite. The unikernel images consumed the required amount of RAM during the boot process. During the boot process the RAM consumption is constantly monitored.

The test scenarios results are recorded for MiragoOS, OSv and OPS Nano VM unikernel during the completion of Experimentation I. The test scenario is run ten times to get a more accurate results. The average result is counted.

3.4.2 Boot time test case

Unikernel technologies does numerous task while booting the unikernel images like system background process, perform kernel operations, create shrink VM kernel, compile the application code by pulling the packages, create the isolated VM image, perform tasks to boo the image etc. All these processes are performed simultaneously. Different unikernel technologies might do some additional tasks during the boot process.

In this test scenario the boot process is measured from the starting of the initial command till the unikernel receives the first http request. The time difference is measured specifically on starting the boot process of connecting to the hypervisor till the unikernel vm is actually running.

To measure the time difference a benchmark shell script is added that contains the initial start time and continuously start sending GET requests to the unikernel image. After getting the first response it stores the end time.

Test result

Table 3.5 *Boot Time Test Result*

#	Unikernel	#	Boot Time in Seconds
1.	Mirage OS,	1.	4.506
2.	OPS Python App,	2.	2.036
3.	OPS Node App,	3.	3.088
4.	OSv,	4.	2.137

3.4.3 Memory footprint test case

Low memory footprint in unikernel images is the main focus point on the idea of unikernel. Memory footprint test case analyses the unikernel images created by each of the unikernel technologies. Almost each of these technologies follows the same pattern of image creating and adding the image to the virtual machine. After the completion of the image creation the unikernel image is generally stored in a image directory along with other VM images. To complete the boot process the unikernel technology then mounts the unikernel image to the virtual machine.

For this test scenario, the size of the image that is stored before attaching it to the VM is measured. The benchmark technology could not measure the deployed image size.

For testing the image size an open source benchmark shell script is used. The script is developed by Vivek Gite (Gite 2020). The shell script displays the stat command information of file size in byte.

For updated images, the previous image is replace with the new image and later the benchmark script analyse the file size of the application.

As the experimentation used different programming language application stack for different unikernel technologies, the libraries and modules that attached with the image affected the file size of the unikernel image. For example, for OPS Nano VM pyhton application the full flask framework needed to be shipped to the image in order to use the flask functions to run the application. On the OSv image JVM is added with the image that consumed some additional space on the virtual machine.

Test Result

The test result is depicted on table 3.6.

Table 3.6 *Memory Footprint Test Result*

#	Unikernel	#	File Size
1.	Mirage OS	1.	3.3 MB
2.	OPS Python App	2.	73 MB
3.	OPS Node App	3.	61 MB
4.	OSv	4.	16 MB

3.5 Experiment III: Deployment of unikernel image of proof of concept solution to ARM device

In the Experiment III the thesis aims to analyse the scope of deploying the unikernel image to the resource constrained ARM-64 devices. The analysis begins with choosing the preferred unikernel technology. The unikernel technology is chosen from the test case analysis from the experimentation II. A real world application is developed to deploy as unikernel in resource constrained ARM devices. The unikernel of the real world application later deployed in the ARM device (Raspberry Pi).

3.5.1 Preferred unikernel technology

For the analysis of the experimentation done in section 3.4, it clearly shows that OPS Nano VM provides better flexibility and technological advantages in terms of developing unikernel. Mirage OS provides smaller footprint in terms of memory footprint.

The experimentation target is to deploy the unikernel in Raspberry Pi 4. Raspberry Pi 4 runs of ARM processor. Mirage OS and OSv provides support for deploying unikernel in ARM processor. The OPS Nano VM does't have ARM support yet. So the prefferd technologu needed to be choosen between Mirage OS and OSv.

The unix kernel in Raspberry Pi can not run virtual machines in KVM alone, it needs Qemu emulator support. Both Mirage OS and OSv can run unikernel in KVM hypervisor. Mirage OS has the limitation of supporting application stack developed with Ocaml programming language. On the other hand OSv can deploy unikernel of application stack developed in different programming language.

As the proof of concept real world application is not developed with Ocaml, the experimentation decided to go with OSv.

3.5.2 Proof concept solution

For the experimentation a real world application is developed using Python. The application is a full fledged web API that provides API's for task management. To create API Python's Flask framework is used. Flask is a WSGI(web service gateway interface) web application framework that provides ease of access web application development with its rich collection of tools and libraries(Hunt-Walker 2018).

Python version 2.7 is used to develop the proof of concept solution. There are couple of flask libraries is used for various purposes like Jsonify, requests etc.

The task management web application receives REST API calls and serves the data as JSON formatted in response.

GET /todo/api/v1.0/tasks The endpoint returns a list of task created with the title and descriptions in JSON format.

POST /todo/api/v1.0/tasks The endpoint receive task information in request body and saves the task in storage. It returns success status 201 if save is successful and for error it returns 401 as alert. The POST request also includes a Boolean Flag that represents the status of the task.

GET /todo/api/v1.0/tasks/task-id The endpoint returns the details information of an individual task.

DELETE /todo/api/v1.0/tasks/task-id The endpoint deletes the details information of an individual task.

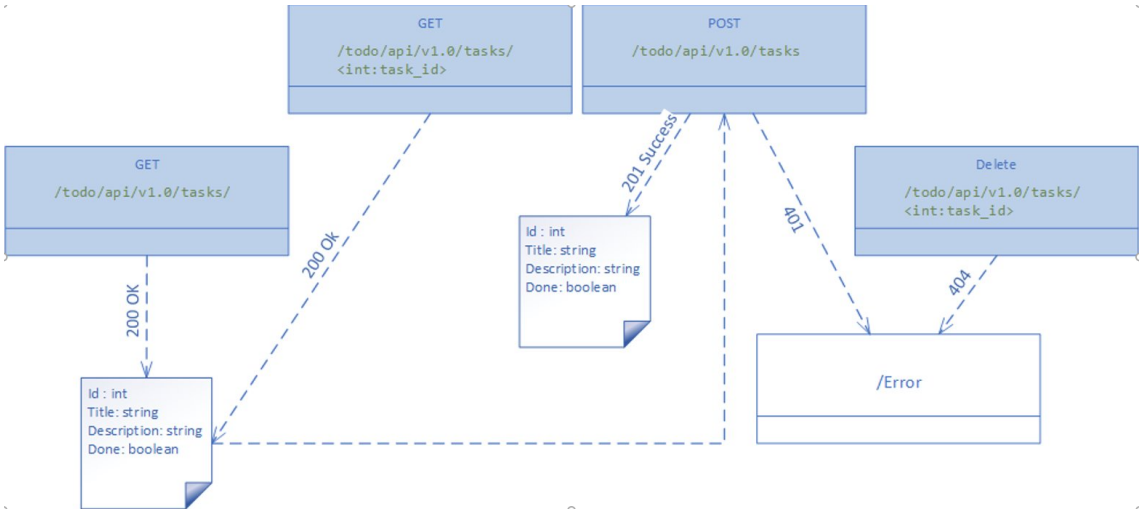


Figure 3.9 *Proof Of Concept solution*

For the application no database or persistent storage is used. A data file with formatted JSON data is used to store and retrieve the data. A sample data format is depicted below.

```
tasks = [
    {'id': 1,
     'title': 'Task1',
     'description': 'Task1 Description',
     'Done': True,
     },
    {'id': 2,
     'title': 'Task2',
     'description': 'Task2 Description',
     'Done': False,
     }
```

]

3.5.3 Unikernel of Proof concept solution

Capstan can not provision Flask framework form the local environment, also the python package uses python old versions. The Flask and python both added to unikernel image and an init local python script added that provision the Flask and its packages from the local repository.

The OSv test setup is done during the Experiment I. The init command creates the package.yaml file which looks like below.

```
capstan package init
    --name "python-task-manager"
    --title "Task Manager" \
    --author "Alom Md Towfiqul"
    --version "0.0.55"
    --require "python-2.7"
```

in run.yaml file config set some additional instruction set added for the provisioning of flask and python environment.

3.5.4 Deployment of unikernel to ARM 64

Raspberry Pi is prepared with Ubuntu Desktop 20.04 version and booted up with micro SSD. For OS booting the experiment used Balena etcher to flash the ubuntu OS image to the SD card. The unikernel image requires KVM and Qemu emulator.

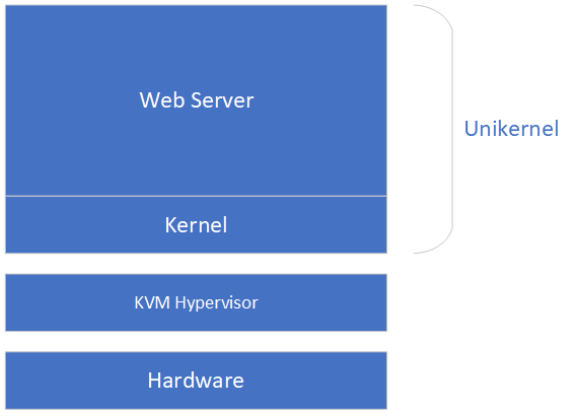


Figure 3.10 *Proof Of Concept unikernel image running in Raspberry Pi*

The quemu emulator provides the necessary environment for KVM hypervisor usage. A boot script is added that boots the unikernel image and attach a static IP to the runtime server.

The unikernel receives a DHCP IP address from the OSv but port forwarding needed to be done in order to publicly access the web server.

3.6 Experiment IV: Orchestration of unikernel proof of concept solution using Arrowhead

Arrowhead orchestration framework can provide solution to many networked IOT device related issues like controlled service discovery and provisioning, device level authorised communication, reduction of protocol layer etc. This experimentation analyze the scope of orchestrating the deployed unikernel service running in IOT device using arrowhead and the advantages the framework provides.

3.6.1 Environment Setup

For the experimentation two arrowhead local cloud needed to setup. The service provider is the IOT device that offers the service using the arrowhead local cloud. The consumer application is the computational unit that request the service from the arrowhead local cloud.

The experimentation setup two arrowhead cloud in two remote SSH server. These SSH server are created using the VMWare Esxi Ubuntu 18.04.4 LTS ISO. The servers are named as local arrowhead cloud 1 and local arrowhead cloud 2. Both the server has 20GB disk space and 32 GB of processing power. The SFTP file server sertup for unikernel experimentation also used in this experimentation to update the required files.

Execution Plan

The experiments proceeds with the following,

- Two Arrowhead cloud in two different virtual machines running in Docker.
- Both the arrowhead server has their own sets of core system and database support.
- The provider service registered in local cloud 1.
- The consumer service registered in local cloud 2.
- The consumer service consumes the service registered in local cloud 1 using the local cloud 2.

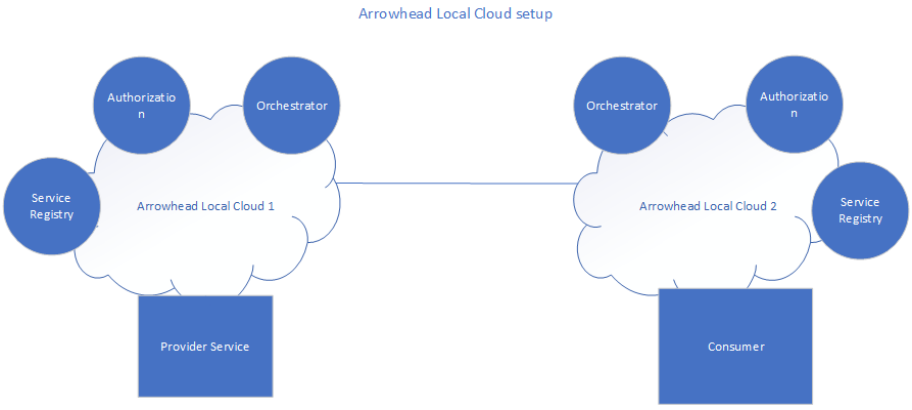


Figure 3.11 *Arrowhead orchestration environment setup*

3.6.2 Arrowhead Cloud Setup

The Arrowhead framework has six core systems in its current local cloud setup service registry, authorization, orchestration, event handler, gatekeeper, gateway etc. Also, for database it uses MySQL and MySQL server.

The current arrowhead release in Github is 4.2.0. The latest version comes with its own sets of Docker images. The experimentation setup used the version 4.1.3. In order to meet the experimentation purposes a new set of Docker images are developed. Each of these Docker images have there own Dockerfile (ex: sr.Dockerfile). Docker images are build in the host virtual machine using Docker Compose. These built images later pushed to the Docker hub repository. From the Docker hub repository, the VM fetch the images and build the servers.

The Docker container running in different port with host IP doesn't get exposed to outside world. For that firewall rules is configured and using the Docker compose port forwarding, each of these container received a forwarding port. The container are exposed to outside world on the forwarding port.

Table 3.7 *Port Setup of Local Cloud 1*

#	Docker Container	#	Port	#	Forwarded Port
1.	Service Registry	1.	8443	1.	38443
2.	Authorization	2.	8445	2.	38445
3.	Event Handler	3.	8455	3.	38455
4.	Gatekeeper	4.	8449	4.	38449
5.	Gateway	5.	8453	5.	38453
6.	Orchestrator	6.	8441	6.	38441
7.	MySQL	7.	3306	7.	33306

Table 3.8 *Port Setup of Local Cloud 2*

#	Docker Container	#	Port	#	Forwarded Port
1.	Service Registry	1.	8443	1.	48443
2.	Authorization	2.	8445	2.	48445
3.	Event Handler	3.	8455	3.	48455
4.	Gatekeeper	4.	8449	4.	48449
5.	Gateway	5.	8453	5.	48453
6.	Orchestrator	6.	8441	6.	48441
7.	MySQL	7.	3306	7.	43306

The experimentation Docker hub repository also containe the build images of the previous versions of Arrowhead like 4.1.2. But due to lack of stability in code base level and the version 4.1.2 used an older version of MySQL server the experimentation continuous with the containers of version 4.1.3.

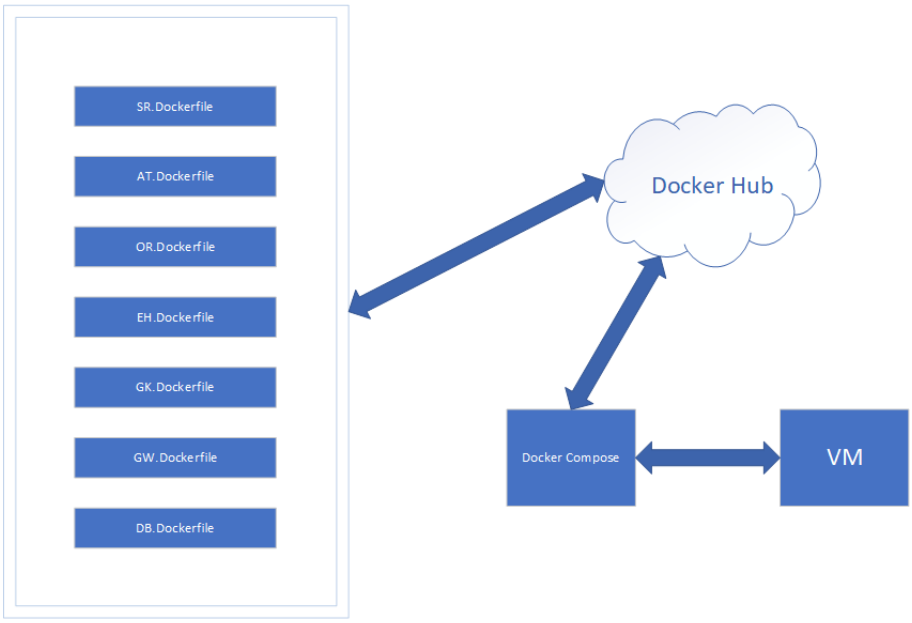


Figure 3.12 *Arrowhead local cloud setup setup*

The Figure 4.1.3 depicts the build process of Arrowhead clouds. Arrowhead local clouds server can be run in two server mode. insecure http mode and secure https mode. Each of the core system comes with application property file. These files contains the server configurations of each of these core system servers. In the server configuration file's SSL server configs the HTTPS mode need to be enabled and a certificate need to be attached to each of the core system.

Arrowhead use self signed SSL certificate. The trust chain is used to verify the authentication of core system before registering it to the local cloud. The service that wants to register itself to a local cloud, it needs to have a SSL certificate

corresponding to the trust chain of that local cloud. The experimentation followed the three level formats of Trust chain as per the below configurations. Arrowhead provides a certificate script to create new certificates for a new client.

```
CA certificate: arrowhead.eu
Local Cloud certificate: tow1.tuni.arrowhead.eu
Services certificate: serviceregistry.tow1.tuni.arrowhead.eu
```

Arrowhead provides a MySQL schema SQL file that needs to be provisioned to the database. The experiment set up a Docker volume persistent storage that holds the schema and all the recurring data. Whenever the container is updated it updates the data from that persistent Docker volume.

After successful setup of the local clouds, the core systems can accessible to the ”https://ServerIP:ForwardedPort” path. The root path of the core systems provide a Swagger UI wich has all the endpoints of that core system. The swagger UI can be used to perform operations of that core system.

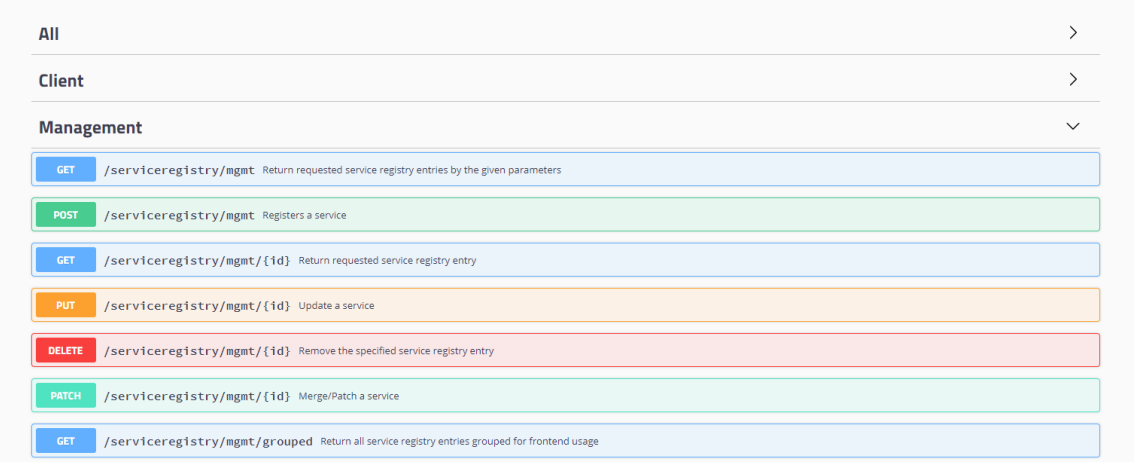


Figure 3.13 *Arrowhead core system swagger UI*

3.6.3 Inter Cloud Orchestration

A network of services running is multiple IOT devices can be managed by multiple arrowhead cloud. Each arrowhead cloud can have different sets of services. These cloud can maintain device to device communication using arrowhead intra cloud dynamic orchestration. In intra cloud orchestration scenario the arrowhead cloud runs in the same cloud network. Intra cloud orchestration process provides necessary interface to forward connectivity configurations of services running in different cloud. This way a seamless data communication is maintained.

Arrowhead orchestration core component also provides orchestration support in inter cloud scenario where arrowhead cloud resides in different network. In inter cloud, data communication is maintained via relay. The relay connects the neighbouring clouds together and maintain cloud connectivity to provide dynamic orchestration support. The relay communicate thorough the gatekeeper core system of each local cloud.

This experimentation analyses the inter cloud orchestration. Arrowhead version 4.1.3 uses ActiveMQ relay for inter cloud orchestration. The experimentation sets up an ActiveMQ relay in virtual machine of Local Cloud 1. The ActiveMQ relay runs in port 61616.

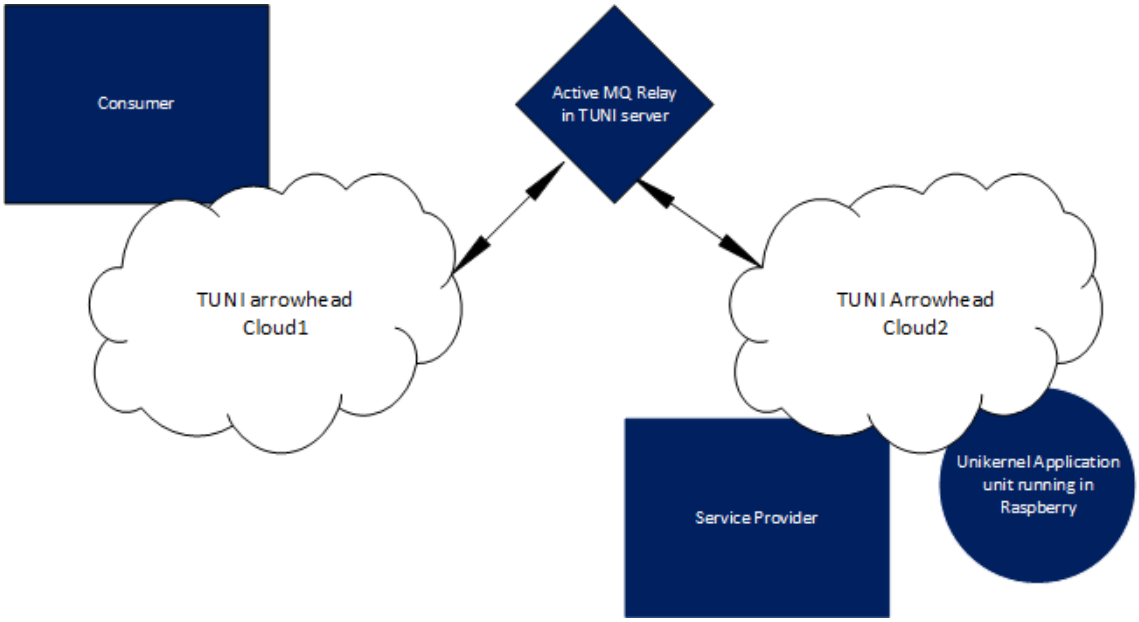


Figure 3.14 *Arrowhead inter cloud orchestration setup*

The process of dynamic inter cloud orchestration is as follows,

Provider System Registry to Local Cloud 1

In the test scenario the provider system is the Raspberry PI 4 and the service is the unikernel running in the IOT device. The orchestration process start with registering the service to the local cloud. For registering the service a service registry form needs to be submitted as JSON and in response the system returns the registry information.

The experimentation named the provider system as ”unik1” and the service as ”osv1”.

POST -> https://SERVER-IP:38443/serviceregistry/mgmt

Service Registry Form

```
{
 "interfaces": [
    "HTTPS-SECURE-JSON"
  ],
  "providerSystem": {
    "address": "0.0.0.0",
    "authenticationInfo":
        "MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AM...",
    "port": 80,
    "systemName": "osv1"
  },
  "serviceDefinition": "unik1",
  "serviceUri": "88.343.56.01",
  "version": 1
}
```


In the service registry form an Interface need to be added. There are three interface in arrowhead "HTTP-INSECURE-JSON", "HTTP-SECURE-JSON","HTTPS-SECURE-JSON". Each of these interfaces defines how the core system will handle this service registry. In the authenticationInfo object the public key of the system. Authorization core system cross check the public key with the cloud public.

JSON Returned by service registry


```
{
  "id": 62,
  "serviceDefinition": {
    "id": 36,
    "serviceDefinition": "unik1",
    "createdAt": "2020-10-23 11:06:33",
    "updatedAt": "2020-10-23 11:06:33"
  },
  "provider": {
    "id": 33,
    "systemName": "osv1",
    "address": "0.0.0.0",
    "port": 80,
    "authenticationInfo": "MII...",
    "createdAt": "2020-10-23 11:06:33",
    "updatedAt": "2020-10-23 11:06:33"
  },
  "serviceUri": "88.343.56.01",
```

```
  "secure": "SECURE",
  "version": 1,
  "interfaces": [
    {
      "id": 3,
      "interfaceName": "HTTPS-SECURE-JSON",
      "createdAt": "2020-09-27 06:45:42",
      "updatedAt": "2019-09-27 06:45:42"
    }
  ],
  "createdAt": "2020-10-23 11:06:33",
  "updatedAt": "2020-10-23 11:06:33"
}
```


Consumer System Registry to Local Cloud 2

The experimentation setup created a consumer server with NodeJS. The consumer client sends the orchestration request to local cloud 2 to consume the unikernel service running in local cloud 1. The consumer service is registered as "client1" in local cloud 2.

Set up relay

The relay needs to be added as Gatekeeper relay in both the cloud. The experimentation setup has the relay in insecure mode, so addition configuration regarding SSL certification were not necessary. The relay added using the following endpoint,
POST -> https://SERVER-IP:PORT/gatekeeper/mgmt/relays

Set up neighbour cloud

Each of the local cloud running in different network needs to recognize themselves as neighbouring cloud. Adding the clouds as neighbour cloud needs to be done vice versa in the gatekeeper of the each cloud using the following endpoint,
POST -> https://SERVER-IP:PORT/gatekeeper/mgmt/clouds

The form also includes the relay connectivity information.

Set up inter cloud rule A inter cloud have to be added on the provider cloud. The rules defines that the consumer of requester cloud has the authorization to connect with the provider service.

The form of the inter cloud rule look a follows,
POST -> https://SERVER-IP:PORT/authorization/mgmt/intercloud


```
{
  "cloudId": 1,
  "providerIdList": [
    33
```

```
  ],
  "interfaceIdList": [
    3
  ],
  "serviceDefinitionIdList": [
    36
  ]
}
```


Dynamic Orchestration

Arrowhead can perform two types of orchestration, store based orchestration and dynamic orchestration. In the store based orchestration some orchestration store entries are added with the provider service configurations with priority flag. In store base orchestration, the orchestrator returns the service that has highest priority.

In the experimentation a dynamic orchestration is tested. The process is as follows,

- The Consumer system submits a orchestration form to POST -> https://SERVER-IP:PORT/orchestration.
- Upon receiving the orchestration process the orchestration core system start a service called GSD(Global Service Discovery). The GSD looks for service in neighbouring clouds. After geting a potential service list it returns the response to orchestration.
- Orchestration then choose the provider system from the neighbouring cloud and initiate ICN(inter cloud nagotiation) process with its Gatekeeper core system.
- Gatekeeper via relay exchange the token to check the authorization. When the ICN process ends, the orchestrator provides the response to the consumer.

The orchestration response looks as below,


```
{
    "response":[
        {
         "provider":{
            "id":33,
            "systemName":"osv1",
            "address":"0.0.0.0",
            "port":80,
```

```
            "authenticationInfo":"MIIBIjAN...",
            "createdAt":"2020-03-23 11:06:34",
            "updatedAt":"2020-03-23 11:06:34"
        },
        "service":{
            "id":36,
            "serviceDefinition":"unik1",
            "createdAt":"2020-10-23 11:06:34",
            "updatedAt":"2020-10-23 11:06:34"
        },
        "serviceUri":"88.343.56.01",
        "secure":"SECURE",
        "metadata":null,
        "interfaces":[
            {
                "id":3,
                "interfaceName":"HTTPS-SECURE-JSON",
                "createdAt":"2020-12-27 06:45:42",
                "updatedAt":"2020-12-27 06:45:42"

            }
        ],
        "version":1,
        "authorizationTokens":null,
        "warnings":null
        }
    ]
}
```

4 Evaluation

In this chapter the thesis evaluates the promises of unikernel in terms of becoming a new generation containerization technology for resource constrained IOT devices. The chapter also evaluates the arrowhead tools orchestration process, its advantages and disadvantages.

4.1 Evaluation of unikernel technology

Although the unikernel technologies are new but the concept of unikernel is there for some time now. The architecture and build process of unikernel technologies are evolving continuously. Containerization technologies like Docker can provide better performances when application stacks are deployed as micro services. But when it comes to containerize and deploy a monolithic application with large computational stack scenario there are couple of disadvantages like boot time, container memory footprint etc.

Containerization technologies are widely embraced in x86 and x64 hardware architecture. The idea of virtualization support in ARM 64 architecture devices are now a widespread popular option. The great debacle on achieving the virtualization in ARM devices was how to share the host machine hardware resources to the container images to run the application stack. Deploying isolated containers in low memory footprint ARM devices is accelerated by the development of hypervisors for ARM architectures like KVM and Xen. Docker itself has now developed a separate application stack for ARM architecture devices for container virtualization. The sole difference between the hypervisor based virtualization and the Docker container based virtualization is that the hypervisors virtualize the hardware resources of the host machine and containerization technology virtualize the host operating system and deploys the containers in shared OS.

Raho et al. 2015 conducted a research on developing a comparison model between KVM, Xen and Docker in ARM 64 architecture. The study concluded that Docker provides greater advantages in developing and deploying applications stacks as container in virtual environment in ARM 64 architecture. But in some scenarios hypervisors provide low overheads compared to Docker. Also, Docker provides a low isolation in virtualization environments which raises security threats.

The thesis analyses unikernel a hypervisor based virtualization technology. The unikernel technology promises to provide isolated computational stack images with low memory footprint, faster deployment and less security threats. The experimentation on unikernel technologies found results that supports that claims and also

some disadvantages along the ways.

The experimentation's were conducted in three unikernel technologies, MirageOS, OPS Nano VM and OSv. The test case analysis on boot time shows that the hypervisor based unikernel technologies provide faster boot time while deploying a large computational stack as unikernel images in hypervisors like KVM. Both OPS Nano VM and OSv can boot a full fledged web server in 2 seconds. In a shared network scenario this gives a great edge where hundreds of embedded devices are interconnected, quickly resurfacing a dependent inter linked computational stack is crucial. A second delay in resurfacing process can bring unexpected downtime that can hinder the whole production process. The unikernel solution's shrinked kernel makes the boot process faster. The experimentation used computational units that serve multiple purposes. A static simple computational unit that serves a single purpose which is ideal in embedded device virtualization scenario the boot time will be much faster during unikernel image deployment as less library modules will be used.

In terms of memory footprint the unikernel solution provides clear advantage over any vitualization technology. The experimentation for analysing the memory footprint is done with real world computational units. On the test case scenarios the application stacks were developed with multiple modules and libraries. All unikernel technologies that were tested provide smaller memory footprints with shrinked kernel and utmost necessary hardware resources. The MirageOS application stack had a web server with Ocaml modules needed for compiling the application and it provided an unikernel image of just 3.3MB. The memory footprint is quite low compared to linux container of similar application that is 50 times higher. The OPS Nano VM Python application stack had a full fledged Flask framework and Python modules. Also the tested NodeJS application for OPS Nano Vm had large node modules needed to support the application functionality. Both the unikernel images provided by the OPS Nano VM are under 100MB. Considering the library modules provisioned to the images the memory footprint is quite low. The application stack used for OSv unikernel had some JVM libraries yet the the image size is 16 MB. To achieve such low memory footprint the unikernel technologies adds a customised kernel that provision the required hardware resources to the unikernel image. The computational units that were converted to the unikernel image runs as a part of that custom kernel, thus providing a smaller footprint to unikernel image.

ARM architecture embedded devices have small memory footprints and the processing power of devices like raspberry Pi is limited. Although more powerful ARM 64 devices are being introduced but they are not on the scale of a full fledged x86 machine processing power. Due to the low memory footprint of unikernel images, it provides ideal support for deploying computational units in ARM 64 devices. Low

memory footprint unikernel images containing a custom kernel also requires less processing power compared to other virtualization technologies. In terms of deploying a real world computational unit in ARM 64 devices this gives a great advantage.

2016 a massive DDoS attack was launched on DNS provider Dyn. The attacks were conducted mostly using IoT devices like security cameras. These devices are less secure and usually contains vulnerable hardware. The malware used in the attack is called Mirai and hijacked the IoT devices, added a bot and launched an infrastructural DDos attack on Dyn. The concept of highly secured IoT devices came into light that day. The unikernel technology provides lightweight images which don't contain insecure remote access services like telnet or ssh. It reduces the attack surfaces by getting rid of a shell that is needed to run any bot for DDos attack.

Sometime the security attack is conducted by inputting a malicious file on the device file system. Unikernel images don't contain a file system, so it is not possible to share or plant a malicious file inside the file system of virtual images and execute anything that was not intended to be executed inside unikernel image.

Due to the very less attack surface unikernel can provide more securities in ARM 64 devices which makes it an ideal solutions for deploying computational units in low memory footprint embedded IoT devices. DDoS attacks like 2016 Dyn attack can also be prevented using technologies where bots like Mirai wont be able plant a botnet on the embedded device and execute infrastructural attack.

Although the unikernel technologies can provide a high security aspect on ARM 64 embedded devices, there is one security aspect the thesis analysis found which can make the unikernel images vulnerable to the outside world. Modern day operating systems like Windows and Linux have device drivers that work as a shield for the kernel to prevent unnecessary remote access to the kernel. Unikernel gets rid of these device drivers the OS provides and makes the kernel vulnerable. A privileged access control limit (ACL) can be added to the unikernel application units with proper authentication methods but that may not be enough in some scenarios.

There is also another drawback the analysis found regarding debugging of application units. There a greater needs of debugging and logging the applications status in a production server. A large computational unit running in production scenarios requires debugging and logging of error or potential errors to avoid downtime. In terms of unikernel images it is difficult to debug the application units. Although debugging units can be added separately along with application units but it's not ideal in some cases.

Docker containerization technology provides easier development approach with its run time commands and Dockerfile based compilation structure. On the other hand developing unikernel is still very difficult due its lack of proper documenta-

tions and continuously changing build processes of different unikernel technologies. The introduction of compilation tools called unik it's now getting easier to create unikernel, yet its difficult to compile unikernel due to its requirement of additional configuration based on the development environment of different application units.

4.2 Evaluation of Arrowhead framework

The thesis analyses the orchestration scope of deployed unikernel images in ARM 64 devices using Arrowhead. The sole purpose of Arrowhead framework is to provide inter-connectivity and orchestration support between networked systems on system of system environment. The thesis analyses the local cloud concept of Arrowhead framework and inter cloud orchestration of unikernel services running in ARM devices.

The system of system local cloud concept of arrowhead framework allows the systems to discover services on the same local cloud or using the gatekeeper and gateway core systems to discover services in other clouds. This gives a great advantage on maintaining a centralized approach. The individual local cloud has the core system of its own. If the requester system belongs to a particular local cloud, that local clouds core systems directs the required services details to the requester system with proper authentication. This removes an overhead of creating an additional authentication system for a system to consume other required system services.

Arrowhead framework provides a great advantage on scalability. The local structure of Arrowhead forms a SoS administration that performs service discovery and orchestrating the services with connected local clouds with its GSD(global service discovery) and ICN(Inter cloud negotiations).

Before redirecting services for the neighbouring cloud the ICN process of Arrowhead framework establish trust between two local clouds using (self-signed) certificates. Each systems including the core systems have to have a certificate from the same hierarchy in order to be entrusted. There is a major drawback in terms of Arrowhead authentication based on self signed SSL certificates. The SSL certificates are slow, they introduce the processing power due to encryption and decryption of SSL tokens. Also, the Arrowhead approach of certification is a centralized approach. In a production scenario the centralized approach of certificates can create additional overhead for certification management. The certificate can expire, can corrupt, this will lead to downtime of services. One possible approach would be to decentralize the certificate hierarchy and dividing the SoS on clusters of certificates. The systems in individual local clouds will be entrusted to one certificate cluster. That way a bulky certification hierarchy can be reduced for proper maintainability.

The current orchestration core system(tested version 4.1.3) of Arrowhead framework returns a service based based priority and based on authorization rules on

local cloud. The authorization rules hold the information on which system services can be consumed by a particular consumer system. To set an authorization rule a manual configuration is needed on the database level or using the core system API. This manual intervention makes the whole automation process go against the core concept of automation. The Arrowhead framework promises to make interconnected IOT devices data communication and service management in automated manner, yet the framework itself is not fully automated.

Despite some of the drawbacks of Arrowhead framework found, it provides some great advantages in establishing a proper orchestration environment for services deployed in hundreds of devices. Some of the crucial jobs required to maintain the inter connectivity between IoT devices are fully managed by the framework which makes the automation process free of overheads and additional automation processes outside the scope of the application stack.

5 Conclusions

The concept of library operating system(LibOS) introduced in projects like Exokernel has existed for a long time. The unikernel technology has taken the libOS concept onto a new level. Unikernel technologies like MirageOS, OSv continuously add different new hypervisor, hardware architecture support in their roster. This makes the unikernel technology a good solution for developing virtualization environments for low memory footprint ARM 64 devices. The unikernel technology provides good answers to some of the challenges regarding deploying virtualized computational units in resource constrained IoT devices.

Using service oriented architecture the Arrowhead framework provides inter connectivity between system of systems in an effective way when it comes to cloud to cloud orchestration. The full automated process of orchestrating service in IoT network without using numerous protocols is time demands. The Arrowhead framework answers the challenges regarding achieving fully automated industrial automation.

The thesis raised three questions, to find effective solutions for deploying computational units in resource constrained ARM devices, facilitating aspects of the solution and an effective way to manage and orchestrate computational units. Chapter 3 of the thesis experimented Unikernel and Arrowhead framework to answer these questions. The experimental approach of the thesis led to a better understanding in which way unikernel is a better solution for deploying computational units in resource constrained ARM devices and how the Arrowhead framework is an effective tool for automating the orchestration and management of unikernels deployed in different ARM devices.

The chapter 4 evaluation of unikernel technologies conclude that unikernel technologies like MirageOS, OSv, OPS Nano VM provides faster boot process, low memory footprint and less attack surface. The current implementations of these unikernel technologies is focus on providing support for different CPU architectures like ARM 64. Whether unikernel is mature enough to be deployed in production server with real world applications is still debatable. But it provides some good advantages in ARM architecture devices which some of the containerization technologies like Docker are still lacking.

In 2016 Docker acquired Unikernel systems a UK based startup. The idea of the acquisition is to add unikernel supports to Docker container build process. This gives Docker platform an edge on managing its bulky container stack footprint. So it is safe to say that the growth of unikernel technologies is progressing and in future the developer community can benefit if the Docker like deployment process of unikernel is introduced.

The latest version of Arrowhead framework 4.2.0 introduced a core system called ”on-boarding controller” which answers some concerns regarding authentication of Arrowhead framework the thesis raised. The Arrowhead framework is constantly evolving and new features are introduced regularly to make the orchestration process more effective and more agile. The collaborative automation approach of Arrowhead framework can be an ideal solution for cloud to cloud inter connectivity in shared networked industrial IoT devices.

The future study on the architecture of unikernel technologies can be done to identify scope of deploying unikernels in production scenarios and with real world application stacks. Also, a future study can be conducted on the quality assurances of Arrowhead framework in terms of production deployment.

References

Allegretta, Chris (2001). "GNU nano a small and friendly text editor". In: URL: https://ftp.gnu.org/old-gnu/Manuals/nano-0.9.99pre3/html_mono/nano.html.

AppFleet (2020). "Demystifying Open-Source Orchestration of Unikernels With Unik". In: URL: https://medium.com/appfleet/demystifying-open-source-orchestration-of-unikernels-with-unik-6bedf909fc01.

Barnes, S. B. (1997). "Douglas Carl Engelbart: developing the underlying concepts for contemporary computing". In: *IEEE Annals of the History of Computing* 19.3, pp. 16–26.

binarymist (2012). "Bare-metal Hypervisor Setup Evaluation". In: URL: https://blog.binarymist.net/2012/01/23/bare-metal-hypervisor-setup-evaluation/.

Bratterud, A. et al. (2015). "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

CETIC (2013). "Unikernel and Immutable Infrastructures". In: URL: https://github.com/cetic/unikernels.

Delsing, J. (2017). "Local Cloud Internet of Things Automation: Technology and Business Model Features of Distributed Internet of Things Automation Solutions". In: *IEEE Industrial Electronics Magazine* 11, pp. 8–21.

Elphinstone, K. et al. (2017). "A Performance Evaluation of Rump Kernels as a Multi-server OS Building Block on seL4". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*.

Engler, D. R., M. F. Kaashoek, and J. O'Toole (Dec. 1995). "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *SIGOPS Oper. Syst. Rev.* 29.5, pp. 251–266. ISSN: 0163-5980. DOI: 10.1145/224057.224076. URL: https://doi.org/10.1145/224057.224076.

Eyberg, Ian (2020). "OPS NanoVM documentation". In: URL: https://nanovms.gitbook.io/ops/.

Fawaz, Paraiso et al. (July 2016). "Model-Driven Management of Docker Containers". In: DOI: 10.1109/CLOUD.2016.0100.

Fayyad, Hasan, LucPerneel, and Martin Timmerman (Dec. 2013). "Benchmarking the Performance of Microsoft Hyper-V server, VMware ESXi and Xen Hypervisors". In: *Journal of Emerging Trends in Computing and Information Sciences* Vol. 4, No. 12 , December 2013, pp: 922-933, ISSN 2079-8407.

Fenn, Michael et al. (2008). “An Evaluation of KVM for Use in Cloud Computing”. In:

Gite, Vivek (2020). “How to check the file size in Linux/Unix bash shell scripting”. In: URL: https://www.cyberciti.biz/faq/howto-bash-check-file-size-in-linux-unix-scripting/.

Hegedus, C., P. Varga, and A. Frankó (2018). “Secure and trusted inter-cloud communications in the arrowhead framework”. In: *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pp. 755–760. DOI: 10.1109/ICPHYS.2018.8390802.

Hegedűs, C. et al. (2016). “Enhancements of the Arrowhead Framework to refine inter-cloud service interactions”. In: *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, pp. 5259–5264. DOI: 10.1109/IECON.2016.7793757.

Hunt-Walker, Nicholas (2018). “Flask Python web app framework”. In: URL: https://opensource.com/article/18/4/flask.

Jaramillo, D., D. V. Nguyen, and R. Smart (2016). “Leveraging microservices architecture by using Docker technology”. In: *SoutheastCon 2016*, pp. 1–5. DOI: 10.1109/SECON.2016.7506647.

Kantee, Antti (2015). “On Rump Kernels And The Rumprun Unikernel”. In:

Kantee, Antti and Andrew Andkjar (2016). “Rumprun”. In: URL: https://github.com/rumpkernel/rumprun/blob/master/README.md.

KEEN, ED (2019). “Automation vs. Orchestration: What’s the Difference?” In: URL: https://www.burwood.com/blog-archive/automation-vs-orchestration-whats-the-difference.

Kivity, Avi et al. (June 2014). “OSv—Optimizing the Operating System for Virtual Machines”. In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, pp. 61–72. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

Kochut, A. and K. Beaty (2007). “On Strategies for Dynamic Resource Management in Virtualized Server Environments”. In: *2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 193–200.

Kohavi, Yuval, Idit Levine, and Scott Weiss (2020). “unik: A platform for automating unikernel MicroVM compilation and deployment”. In: URL: https://github.com/solo-io/unik.

Kot, Martin (2019). “Evaluation of Bare Metal CPU and Memory Performance of the Unikernel IncludeOS and Ubuntu”. In:

Kozaczuk, Waldemar (2020). “OSv, a new operating system for the cloud”. In: URL: https://github.com/cloudius-systems/osv.

Kozma, Daniel, Gabor Soos, and Pal Varga (July 2019). "Supporting Digital Production, Product Lifecycle and Supply Chain Management in Industry 4.0 by the Arrowhead Framework - a Survey". In: DOI: 10.1109/INDIN41052.2019.8972216.

Levine, Idit (2020). "unik Cloud Foundry". In: URL: https://docs.google.com/document/d/1Q9GakKpm6DMniJpWB-fqhE13SSPaj4-3sOWZ-I5nVyA/edit#.

Liu Xia et al. (2010). "Design of secure FTP system". In: *2010 International Conference on Communications, Circuits and Systems (ICCCAS)*, pp. 270–273. DOI: 10.1109/ICCCAS.2010.5582002.

Lucina, Martin, Ricardo Koller, and David Williams (2019). "Solo5: A sandboxed, re-targetable execution environment for unikernels". In:

Madhavapeddy, Anil, Thomas Leonard, et al. (2015). "Just-In-Time Summoning of Unikernels". In: URL: https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-madhavapeddy.pdf.

Madhavapeddy, Anil, Richard Mortier, et al. (Mar. 2013). "Unikernels: Library Operating Systems for the Cloud". In: *SIGPLAN Not.* 48.4, pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: https://doi.org/10.1145/2499368.2451167.

Madhavapeddy, Anil and David J. Scott (2014). "Unikernels: Rise of the Virtual Library Operating System". In:

Marathe, N., A. Gandhi, and J. M. Shah (2019). "Docker Swarm and Kubernetes in Cloud Computing Environment". In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, pp. 179–184. DOI: 10.1109/ICOEI.2019.8862654.

Mavridis, I. and H. Karatza (2019). "Lightweight Virtualization Approaches for Software-Defined Systems and Cloud Computing: An Evaluation of Unikernels and Containers". In: *2019 Sixth International Conference on Software Defined Systems (SDS)*, pp. 171–178. DOI: 10.1109/SDS.2019.8768586.

McKenney, Paul E. and Jonathan Walpole (July 2008). "Introducing Technology into the Linux Kernel: A Case Study". In: *SIGOPS Oper. Syst. Rev.* 42.5, pp. 4–17. ISSN: 0163-5980. DOI: 10.1145/1400097.1400099. URL: https://doi.org/10.1145/1400097.1400099.

nakivo (2018). "Type 1 and type 2 hypervisor". In: URL: https://www.nakivo.com/blog/hyper-v-virtualbox-one-choose-infrastructure/type-1-and-type-2-hypervisor/.

Padmanabhan, Arvind and Faizan Bashir (2018). "Devopedia: Container Orchestration". In: URL: https://devopedia.org/container-orchestration.

Pahl, C. (2015). "Containerization and the PaaS Cloud". In: *IEEE Cloud Computing* 2.3, pp. 24–31. DOI: 10.1109/MCC.2015.51.

Paniagua, C., J. Eliasson, and J. Delsing (2020). "Efficient Device-to-Device Service Invocation Using Arrowhead Orchestration". In: *IEEE Internet of Things Journal* 7.1, pp. 429–439. DOI: 10.1109/JIOT.2019.2952697.

Paniagua, Cristina and Jerker Delsing (May 2020). "Industrial Frameworks for Internet of Things: A Survey". In: *IEEE Systems Journal* PP, pp. 1–11. DOI: 10.1109/JSYST.2020.2993323.

Pavlicek, R. (2017). "Unikernels: Beyond Containers to the Next Generation of Cloud. O'Reilly Media". In:

Penninkhof, Jan (May 2015). "Minimalist Cassandra VM using an OSv Unikernel". PhD thesis.

Preeth E N et al. (2015). "Evaluation of Docker containers based on hardware utilization". In: *2015 International Conference on Control Communication Computing India (ICCC)*, pp. 697–700. DOI: 10.1109/ICCC.2015.7432984.

Raho, M. et al. (2015). "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing". In: *2015 IEEE 3rd Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pp. 1–8. DOI: 10.1109/AIEEE.2015.7367280.

Robin (2019). "Detail Of XEN Architecture". In: URL: https://it.4ditsolution.com/detail-of-xen-architecture/.

Sapper, Kevin (2015). "Unikernels No OS ? No problem !" In:

Sarkar, Avijit (2019). "Docker and OCI Runtimes". In: URL: https://medium.com/@avijitsarkar123/docker-and-oci-runtimes-a9c23a5646d6.

Shah, J. and D. Dubaria (2019). "Building Modern Clouds: Using Docker, Kubernetes Google Cloud Platform". In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0184–0189. DOI: 10.1109/CCWC.2019.8666479.

Skjegstad, Magnus (2015). "A unikernel experiment: A VM for every URL". In: URL: http://www.skjegstad.com/blog/2015/03/25/mirageos-vm-per-url-experiment/.

STRUKHOFF, ROGER (2017). "A Unikernel Compiler (UniK) Added to the Cloud Foundry Extensions Incubation". In: URL: https://www.altoros.com/blog/the-unik-compiler-added-to-the-cloud-foundry-extensions-incubation/.

tuchacloud (2020). "HYPERVISOR KVM". In: URL: https://tuchacloud.com/hypervisor-kvm/.

Williams, David, Martin Lucina, and Ricardo Koller (2020). "About Solo5". In: URL: https://github.com/Solo5/solo5.