Analysis of Unikernels for Load Balancing and Backend Service Deployment

MSci. (Hons) Computer Science

Jonathon Robert Cross

School of Computing and Communications
Lancaster University

March 2021
Abstract

This project explored the application of unikernels as a novel runtime framework and measuring the performance of the unikernel; its claim of tiny image sizes, against another virtualisation/runtime paradigm such as containerisation. This will be measured by creating an identical infrastructure with a Unikernel, using the Unikraft SDK\textsuperscript{1} and with Docker which is a container platform.\textsuperscript{2}

The infrastructure will require HAProxy, a load balancer and proxy application, to be ported to the Unikraft environment - this will act as a load balancer for Nginx instances which will act as simple http servers. This project was aimed at exploring unikernels as a novel virtualisation paradigm and measuring the performance of the unikernel; its claim to inherent security benefits, against another virtualisation paradigm such as Docker.
Acknowledgements

I would like to thank both my supervisors Dr. Charalampos Rotsos and PhD student Alexander Jung for helping me port HAProxy to Unikraft and their patience with my learning.

I would also like to thank my friends and loved ones for their support throughout the pandemic, ensuring this year was a productive one.
Declaration

I certify that the material contained in this dissertation is my own work and does not contain unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation. Regarding the electronically submitted version of this submitted work, I consent to this being stored electronically and copied for assessment purposes, including the School’s use of plagiarism detection systems in order to check the integrity of assessed work.

I agree to my dissertation being placed in the public domain, with my name explicitly included as the author of the work.

Date: 19th March 2021
Signed: Jonathon Robert Cross
# Contents

Abstract i  
Acknowledgements ii  

1 Introduction 1  
1.1 Chapter Overview 1  
1.2 Motivation for Research 1  
1.3 Aims and Objectives 2  
1.4 Report Structure 3  
1.5 Chapter Summary 3  

2 Background Research 4  
2.1 Chapter Overview 4  
2.2 Virtualisation Paradigms 5  
2.2.1 System Virtual Machines 5  
2.2.2 Process Virtual Machines 5  
2.3 Containers 6  
2.3.1 Namespaces 6  
2.3.2 Docker 7  
2.4 Unikernels 9  
2.4.1 Benefits of Unikernels 9  
2.4.2 Current Restrictions of Unikernels 10  
2.4.3 Where Unikraft Fits In 11  
2.5 Standards in Backend Service Deployment 12  
2.5.1 Microservice Architecture 12  
2.5.2 CI/CD 13  
2.5.3 DevOps Services 14  
2.6 Load Balancers 15  
2.6.1 Operation of Load Balancers 15  
2.6.2 Load Balancing Techniques 16  
2.7 HTTP Servers 18  
2.7.1 Nginx 18  
2.7.2 Apache 18  
2.7.3 Internet Information Services (IIS) 18
2.7.4 Means of Testing Services ........................................ 18
2.8 Unikraft Core Functionality ........................................ 20
  2.8.1 POSIX Standard ............................................. 20
  2.8.2 Newlib ...................................................... 20
  2.8.3 Memory Allocation ........................................... 20
  2.8.4 Networking ................................................... 21
  2.8.5 Virtual Filesystem .......................................... 21
  2.8.6 POSIX Threads ............................................... 22
  2.8.7 Stream Compression ......................................... 22
2.9 HTTP Benchmarking ................................................ 23
  2.9.1 wrk ........................................................... 23
2.10 Alternative Unikernel Platforms .................................... 24
  2.10.1 MirageOS .................................................. 24
  2.10.2 OSv ......................................................... 24
  2.10.3 UniK ......................................................... 24
2.11 Why Unikraft? ..................................................... 24
2.12 Chapter Summary .................................................. 25

3 Design ....................................................................... 26
  3.1 Chapter Overview .................................................. 26
  3.2 Unikraft ............................................................ 27
    3.2.1 Makefile.uk ............................................... 27
    3.2.2 Config.uk ................................................... 27
    3.2.3 Kraft .......................................................... 27
    3.2.4 Handling the Virtual Filesystem ......................... 28
  3.3 Docker and Docker Compose ........................................ 29
  3.4 Backend Service Topology ........................................ 30
    3.4.1 Chosen Load Balancing Technique ...................... 30
  3.5 Virtualisation Topology .......................................... 31
    3.5.1 Docker Virtualisation Topology ......................... 31
    3.5.2 Unikraft Virtualisation Topology ....................... 32
  3.6 Chapter Summary .................................................. 33

4 Implementation ....................................................... 34
  4.1 Chapter Overview .................................................. 34
  4.2 Docker Implementations .......................................... 35
    4.2.1 Docker Compose ............................................ 35
    4.2.2 Implementing HAProxy in Docker ....................... 36
    4.2.3 Implementing Nginx in Docker ......................... 36
  4.3 Running Docker Implementation .................................. 37
    4.3.1 Testing Nginx Container ................................. 37
    4.3.2 Testing HAProxy Load Balancer for Nginx Servers .... 38
  4.4 Implementing HAProxy in Unikraft .............................. 40
    4.4.1 Porting libslz .............................................. 40
    4.4.2 Patches ..................................................... 41
    4.4.3 Fixing libuuid ............................................. 41
4.4.4 Configuring HAProxy ........................................... 42
4.4.5 Running HAProxy ............................................ 43
4.5 Implementing Nginx HTTP Server in Unikraft ............... 45
  4.5.1 Configuring Nginx ........................................ 45
  4.5.2 Running Nginx ............................................. 45
4.6 Loadbalancer Contingency Plan ............................. 46
  4.6.1 Configuration of the Nginx Loadbalancer ............... 46
  4.6.2 Implementing Nginx Loadbalancer in Unikraft ........... 47
  4.6.3 Implementing Nginx Loadbalancer in Docker .............. 47
4.7 Chapter Summary .............................................. 48

5 Testing .......................................................... 49
  5.1 Chapter Overview ............................................ 49
  5.2 HTTP Benchmarking with wrk ............................ 49
    5.2.1 wrk Command Template ................................ 49
    5.2.2 Unikernel Testing ..................................... 50
    5.2.3 Containers Testing .................................... 52
    5.2.4 Unikraft Website ...................................... 54
  5.3 Image Sizes .................................................. 55
    5.3.1 Container Sizes ........................................ 55
    5.3.2 Unikernel Sizes ........................................ 55
  5.4 Chapter Summary ............................................ 56

6 Results and Discussion ........................................ 57
  6.1 Chapter Overview ............................................ 57
  6.2 HTTP Benchmark Data ....................................... 57
    6.2.1 Unikraft vs Docker ..................................... 57
    6.2.2 Standalone Instance vs Loadbalancer ................. 58
    6.2.3 Unikraft Website ...................................... 58
  6.3 Image Sizes .................................................. 59
    6.3.1 Slimming Docker Image Sizes ......................... 59
    6.3.2 Slimming Unikraft Image Sizes ....................... 59

7 Project Retrospective ........................................... 60
  7.1 Chapter Overview ............................................ 60
  7.2 What went well ............................................... 60
    7.2.1 POSIX .................................................. 60
    7.2.2 C Programming Language .............................. 61
    7.2.3 Build Systems and Unikraft ......................... 61
    7.2.4 QEMU .................................................. 62
    7.2.5 Virtualisation Paradigms ............................. 62
    7.2.6 DevOps ................................................ 63
    7.2.7 FOSS and OS Development ............................ 63
    7.2.8 Partial Port of HAProxy ............................. 64
  7.3 Unikernel Barriers ......................................... 65
    7.3.1 Porting HAProxy ...................................... 65
7.3.2 Nginx as a Load Balancer Contingency .......... 65
7.3.3 Porting Process ........................................... 66
7.3.4 Adoption .................................................. 66
7.3.5 Connection Limitations in Unikraft ................. 67
7.4 Potential for Unikernels ................................. 68
    7.4.1 Backend Service Deployment ................. 68
    7.4.2 DevOps ................................................. 69
7.5 HAProxy - Missing Features ............................. 69
7.6 Chapter Overview ........................................ 70

8 Conclusion ................................................. 71
    8.1 Project Aims Review .................................. 71
    8.2 Future of Unikernels .................................. 73
    8.3 Future Work .............................................. 74
    8.4 Learning Outcomes and Final Notes .................... 75

9 Appendix ................................................... 76
    9.1 HAProxy Config ........................................ 76
    9.2 Nginx Config ........................................... 78
        9.2.1 HTTP Server nginx.conf ........................... 78
        9.2.2 Loadbalancer nginx.conf ....................... 79
    9.3 HTTP Server Static Content ....................... 80
    9.4 SLZ Unikraft Config .................................. 81
    9.5 HAProxy Unikraft Config .............................. 82
        9.5.1 HAProxy Config.uk .............................. 82
        9.5.2 HAProxy Makefile.uk ........................... 82
        9.5.3 kraft.yaml ........................................ 84
        9.5.4 Makefile ........................................... 85
    9.6 Extra Screenshots .................................... 85
    9.7 Original Proposal .................................... 86
        9.7.1 Abstract .......................................... 86
        9.7.2 Introduction ...................................... 87
        9.7.3 The Project ........................................ 87
        9.7.4 Methodology ....................................... 88
        9.7.5 Programme of Work ............................... 88

References .................................................. 89
Chapter 1

Introduction

1.1 Chapter Overview

This chapter is intended to provide an insight to the reasoning for picking this topic which will include some background on the technology, followed by the aims and objectives of this project.

1.2 Motivation for Research

Enterprise systems have historically used virtual machines, first dating back to the 1960’s with IBM’s CP/CMS (Control Program, Cambridge Monitor System). This enabled users to run several operating systems (OSs) in parallel, with isolation from each instance. This brought multiple benefits to system administrators as different software could be run on a given mainframe; the separation would mean that a bug on one virtual machine would not affect the others.

Fast forwarding to the modern day, with the rise of cloud computing; microservices; fog and edge computing, we have found ourselves increasingly reliant on the provisioning and control of Virtual Machines (VMs) - sometimes in a just-in-time manner such as Amazon Web Services (AWS) Lambda, which are ephemeral services - provisioned with the intent of running a service once and destroying itself upon completion to free computation resources.

This has brought us to improve the traditional VM, which was to simply load up an operating system and its applications into memory which is quite computationally expensive, to instead utilising containers. Containers are a lightweight alternative to a VM and operate by utilising OS-level namespace, sitting on top of the OSs kernel to isolate the service and create an independent environment - a sandbox.

The benefits of containers can be exhibited by looking at the bootup latency
between VMs and Containers, where 512 Docker instances took a total of 987 seconds to load, or 1.89 seconds per container, where 64 VMs were booted in a similar time period. Due to Docker’s expansive popularity - with a sample size of 10,000 companies nearly a quarter fully adopting and a year-on-year growth of 3 to 5 points I decided this would be the implementation for comparison.

Therefore, the scope of this study is to explore Unikernels as a virtualisation paradigm against the current trend towards containerisation (which can be seen through a year-on-year growth of 3 to 5 percent), with unikernels providing not just a lightweight image to be booted, but one that can be optimised and contains alleged inherent security features, such as not sharing the host’s kernel as containers do and instead sitting on the hypervisor, this allows for greater isolation of the program. In addition to this, the overall size and attack surface of a unikernel is exceptionally small, where numerous papers suggest that due to the high-specialisation; reduction of the requirement for backwards-compatibility of systems; the single image and address space model provides benefits to the security of a Unikernel. Examples of the risks this model mitigates is exploitation of large code bases that contain vulnerable backdoors and stopping memory exploits as the address space randomization technique is taken from runtime to compile-time address space randomization.

1.3 Aims and Objectives

The aims of this project can be summarised as follows:

- Generate research for current industry standards in backend service deployment; tools utilised in enterprise environments.
- Study the performance capabilities of the unikernel and container system
- Explore the available technologies to enable a microservice architecture
- Port HAProxy to Unikraft
- Utilise the Unikraft HAProxy port as a load balancer for Unikraft Nginx HTTP server instances
- Create an identical infrastructure with HAProxy as a load balancer with Nginx HTTP servers with Docker
- Observe the performance of a load balancing microservice implemented by unikernels and containers
- Conclude as to the differences between the virtualisation paradigms in a backend-system environment
1.4 Report Structure

- **Chapter 2, Background:** Research into virtualisation paradigms, namely containers and unikernels, discussing the standards in backend service deployment with microservices and DevOps, discussing the components that the projects microservice architecture consisted of, with some background to alternative applications and platforms, research to the core kernel functionality; a review of the tools that were used in the project.

- **Chapter 3, Design:** Documenting the design of the Unikraft and Docker systems and the tools that I will be using. Discussing the network topology and the topology of how Docker and Unikraft will be instantiated.

- **Chapter 4, Implementation:** Discussing how the Unikernel and Docker backend service infrastructure has been architected and implemented.

- **Chapter 5, Testing:** Showcasing the performance of the different infrastructures through tools such as wrk and evaluating image sizes.

- **Chapter 6, Results and Discussion:** This chapter will be analysing the results gained from testing the backend implementations; what we can observe.

- **Chapter 7, Restrospective:** Covering any retrospective comments and notes about the process of porting a Unikernel with the Unikraft tool.

- **Chapter 8, Conclusion:** Generating a final review of whether the aims and objectives of this project were met, outlining the key observations and their implications; thoughts on the future of the Unikernel technology.

1.5 Chapter Summary

This chapter detailed the motivation behind this project - this being the exploration of Unikernels as an efficient and secure novel virtualisation technology; whether it is ready to match the contrasting and popular containerisation systems. It has also detailed the objectives for this study required to be carried out in order to achieve it’s objectives and satisfy the motivations. This foundation will be the backbone for the direction of this project.
Chapter 2

Background Research

2.1 Chapter Overview

This chapter will be covering all of the relevant content and compiling the information gained from the background research into virtualisation paradigms, with a focus on Unikernels with their history and aliases as a niche form of virtualisation leading to the present day. Therefore there will be a large focus on gaining information from whitepapers and industry resources to form a better understanding of how unikernels can be implemented and deliver in a microservices architecture; aid in our final analysis.

This chapter should ultimately also give the reader a comfortable understanding of the current state of the enterprise backend services, the technologies employed in these services and the reason why these services have either risen to popularity, or been cast aside for a different solution.
2.2 Virtualisation Paradigms

There are a variety of virtualisation paradigms which centre around the concept of a virtual machine. These VMs have been defined as “a compute resource that uses software instead of a physical computer to run programs and deploy apps” where one or more virtual (machine) instances known as guests, run on a physical host machine.\(^8\)

The forms that VMs can be created vary into different specifications.

2.2.1 System Virtual Machines

System VMs, also considered to be full VMs, are isolated VMs that provide a complete virtual hardware platform which can support execution of a complete OS.\(^9\) Therefore, an example of a System VM would be VirtualBox, which is a free and open source software (FOSS) virtualisation product for various CPU Instruction Set Architectures such as x86 and supports a variety of guest operating systems.\(^10\) While System VMs provide benefit in their isolation - which can be useful for enterprise systems which run many instances on hardware, requiring that a failure in one users VM cannot affect another's - drawbacks include poor performance such as that seen in the paper “My VM is Lighter (and Safer) than your Container”\(^11\) where the Debian VM in Fig. 4 and Fig. 14 was far inferior in performance against the container and unikernel instances.

2.2.2 Process Virtual Machines

Process VMs in contrast is “designed to run a single program with a single process”\(^9\) which is designed to run its application/process on the host OS and will exit once it has finished or faulted. An example of this being the Java Virtual Machine (JVM) which was designed to run on any system the programmer wanted it to - thus the JVM provides a vital role in taking the application the programmer writes, running its process on a supported host OS and the JVM handles smaller details such as platform-specific instructions. Similarly, for .NET the Common Language Runtime exists as a VM which will execute the instructions written in C\#, for example.\(^12\)

Containers and by extension, unikernels can be considered Process VMs.\(^13\) Containers remove the requirement for the virtualised instance to require its own guest OS, in the way that a normal System Virtual Machine would. Instead, the container includes the programs and libraries required for the software to run; uses the Linux kernel for resource management.
2.3 Containers

As mentioned in the previous section, containers can be considered to be Process VMs, including its software and dependencies and using the Linux Kernel (of which if running on Windows or macOS will require a virtual machine to be run) to conduct resource management for the CPU time, memory, networking etc. However due to this very 'benefit', containers have inherently confined themselves to the problems and restrictions of the Linux Kernel, which in Fig. 1 of the “My VM is Lighter (and Safer) than your Container” paper\textsuperscript{11} displays how the number of syscalls has steadily increased to nearly double over the course of 16 years. This presents a problem in regard to securing a container, with a number of exploits being found due to this unmanaged and insecure growth.

2.3.1 Namespaces

Namespaces are something that is essential in understanding containers. Namespaces evolved from the common concept of scope, which creates a form of encapsulation. The purpose of namespaces is to create isolation from independent namespaces which as mentioned in §2.2.1 ensures that the failure of one VM, does not impact another but also provides benefits of having a process hold its own instance of the global resource.\textsuperscript{14,15}

The namespaces currently defined in the Linux Kernel are:

- **Mount Namespace** - Deals with the systems mounted filesystems allowing different views and namespace isolated mounting.

- **IPC Namespace** - Isolates interprocess communication resources such as POSIX message queues.

- **UTS Namespace** - Derives from “Unix Time-sharing System”, allows containers to have their own hostname and NIS domain name.

- **PID Namespace** - Handles the Process ID numer space, allowing namespaces to have the same PID due to the isolation.

- **Network Namespace** - Ensures a namespace has its own network devices - IP addresses, routing tables, port numbers etc.

- **User Namespace** - Similar to the PID namespace, it isolates user and group IDs and can allow an otherwise unprivileged user to have root permissions within the namespace.
Despite attempts at locking down containers with methods such as namespacing, various exploits of incorrectly configured systems and outright exploits of the Linux Kernel remain an issue, as problems such as privilege escalation which can operate as by default users not namespaced - therefore a process that breaks out of the container will retain the privileges it was given within the container such as root access. An example of the consequences of these actions can be seen in a recent 'hack' involving gaining root access to a surveillance system operated by Verkada which gave access to 150,000 surveillance cameras around the world - including psychiatric wards and schools.

2.3.2 Docker

The container solution that I will be using in this study will be Docker, with the tool Docker-Compose which allows for a multi-container solution to be created. Docker as a popular container platform operates by using a file called Dockerfile to implement commands which will create the image for the container they wish to run and will begin with the FROM instruction.

Docker Compose

Docker Compose is a tool that extends this functionality to define and run a multiple container application or system. In order to define the services used in the application you must create a yaml configuration file called docker-compose.yml which can be seen in §3.3.

Docker Swarm

Docker provides a tool to natively create clusters of Docker Engines which are called a swarm. A swarm is effectively a cluster of nodes that fit into one of three roles:

- **Manager** - Orchestrates cluster management functionality; delegates tasks to workers.
- **Leader** - Elected leader by management nodes to conduct orchestration tasks.
- **Worker** - Receives and executes tasks.

Managers can act as workers (by default) or be set to exclusively perform management tasks. Furthermore, clusters are similar to docker compose in that swarm service stacks can be created and defined via `docker service`. 

Kubernetes

Kubernetes however takes what Docker Swarm is intending to do and has elaborated upon it, with an emphasis of scalable flexibility; namespacing and role-based authorisation; a better integration into a cloud-native environment. Kubernetes is oriented toward teams that want an orchestration system that can not only be scaled to billions of containers a week as Google boasts but also scale within a team with its management capabilities only rivalled by systems such as Amazon Elastic Container Service of which AWS provides solutions such as IAM for user-management and permissions.
2.4 Unikernels

Unikernels have been a traditionally sidelined technology due to the complexity involved in creating what was originally referred to as a “libraryOS” or libOS. However, in recent years with the rise of containers in enterprise systems the next logical step has been suggested to emulate the success of the container with straightforward and ready-to-go implementations of Unikernels.

2.4.1 Benefits of Unikernels

Optimisation and Security

While modern virtualisation systems such as Docker have removed inefficiencies such as the guest-OS previously required for all VMs, there is still an underlying inefficiency and overall vulnerability such as those discussed in §2.3. By stripping all the unnecessary components at compile time such as configuration files, user processes, the language runtime like MirageOS for example had, the product was a specialised kernel with a high level of optimisation.

With an optimised and limited set of functionality to create the specialised kernel which inherently improves the security of the application, comes another related benefit in the form of a limited attack surface. Currently, with VMs or Images that are misconfigured, this can allow for various unrequired services to run which therefore allows the attack surface of the application to expand, which is another way in which Unikernels provide value for inherent security - removing all unnecessary features from the VM to form a microservice-ready image.

Small Image Size

![Figure 2.1: Docker Image Size for an Nginx Instance](image)

Furthermore, Unikernels when compared against Containers and traditional VMs are extremely small, with the sizes of unikernels reported from as little as 3.3MB to ClickOS boasting a 5MB image size and MirageOS describing a binary size of 0.673MB before elimination of dead code, after which it was 0.172MB. In contrast as seen in Fig. 2.1 a standard Nginx image displaying a size of 133MB. At an enterprise level the image size of an application is very important as organisations such as Google are running and closing billions of containers a week worldwide, with larger containers requiring more time to boot and to begin executing functionality it begins to become a question of efficiency in regard to budget and environmental impact as the usage of containers requires a higher energy draw.
2.4.2 Current Restrictions of Unikernels

Novel Technology Status

Currently and historically unikernels have been a difficult application to develop, from barriers such as device drivers having to be written to a specific, new model - which due to the fast pace of technology frequently finds itself obsolete. In addition to this, the simple act of having to create an application that requires lots of low-level development and ample time - therefore budget and expertise - proves itself a difficult challenge; one that does not have a positive cost-benefit for many organisations. While the efforts of MirageOS and Unikraft are significant in lowering the barrier to entry and improving the ability for organisations or individuals; unikernels status as a novel technology still holds challenges for anyone wishing to construct a unikernel.

Lack of Native Orchestration Methods

While solutions such as UniK exist that are geared toward compiling application sources into Unikernels, this is not the same as taking a Unikernel image and orchestrating the building, running and tracking of a kernel built with Unikraft for example. The UniK tool in addition to this, relies upon tools such as Kubernetes to realise the full functionality of its tools.

In addition to this, the footprint of UniK built applications is advertised as 6x the size of another unikernel such as ClickOS, with UniK stating an image size of 30MB and ClickOS as previously mentioned boasting an image size of 5MB; other projects creating image sizes between 3.3MB and 3.5MB or MirageOS with a staggering 0.172MB.

Thus, currently there is no robust native orchestration methods for unikernels that match those seen in §2.3.2, this presents a barrier to enterprise adoption as there is no standard way to quickly and efficiently boot Unikernels while benefitting from the management and task delegation seen in tools such as Docker Swarm.

Potential for Increased Burden on Cloud Orchestration Layers

In addition to the lack of native orchestration methods, another problem reveals itself in the form of a greater strain on cloud orchestration layers, with any changes made to the configuration of a MirageOS unikernel requiring a re-deployment due to the closely-coupled and monolithic nature of the system - which passes the burden to the cloud orchestration system. Thankfully however, with careful thought into the design and implementation of a system the mixture of dependencies can be split into what are effectively microservices - creating a more agile overall system.
2.4.3 Where Unikraft Fits In

Unikraft is an attempt to bridge the gap between the difficulty in developing a highly-specialised kernel application, with the intent to promote a standardised approach supporting multiple platforms - from the bare metal to KVM, Xen, AWS Firecracker and other platforms. Utilising a Docker-like configuration approach to building, running and even porting applications through the Makefile.uk, Config.uk files with the Unikraft build system and allowing for patches to be implemented to codebases to fix any issues such as non-supported functionality or irrelevance due to the unique architecture of the unikernel has created a much more user-friendly environment to encourage the adoption of this novel virtualisation technology.
2.5 Standards in Backend Service Deployment

2.5.1 Microservice Architecture

Microservice Architecture can be considered a distributed application where all of its modules are microservices, of which are defined as a cohesive and independent process interacting via messages. The rise of microservice architectures is attributed to the struggles of increasing complexity in modern software and its constant requirement to be flexible and adaptable as requirements of the system consistently change. The alternative to a microservice architecture can be considered to be a monolithic architecture, which as the name suggests is a system of which is comprised of modules that are closely coupled. The consequences of a monolithic architecture is the inability or obstruction to change or update modules, limits upon the scalability of the product and often causes a single point of failure which is a detriment to a high-availability system.

Fig. 2.2 depicts an example of a Microservice Architecture Topology.

Figure 2.2: Example of a Microservice Architecture Topology
may adopt, showing different microservices that the client can access - which may contact other microservices which can return or mutate data, or that must interact with another, shared, microservice to handle authorisation which will then allow the performance of business logic or CRUD operations on a database.

2.5.2 CI/CD

Figure 2.3: Example of a full Continuous Integration/Continuous Deployment Pipeline

Continuous Integration and Continuous Deployment has become a big component of modern service deployment, with an example shown in Fig. 2.3 showcasing how a common CI/CD pipeline can be created, for example going from a staging branch to a master branch - ensuring checks are made to allow for merging the codebases; executing the automated tests to reduce the likelihood of a bug being introduced into the system.

Continuous Integration

Continuous integration (CI) involves automating the integration of code changes in a software project, for example when a pull-request is detected to merge into the master, or main, branch a condition can be made that all the tests must pass before it can be merged. There are many tools that have been created to achieve this goal such as Travis CI and Jenkins.

Continuous Deployment

Continuous deployment (CD) on the other hand involves deploying the codebase to the platform it is hosted on such as Heroku, AWS, Google Cloud Platform or an on-premise server. This stage of the CI/CD pipeline can also involve ensuring that the build is functional before pushing the build to its destination. This functionality is also incorporated by tools such as Travis CI and Jenkins.
2.5.3 DevOps Services

DevOps services/solutions such as Azure DevOps embrace the concept of centralising project management; repositories; pipelines and other features to create a synergetic ecosystem for a software project. DevOps as a practice is “about fast, flexible development and provisioning businesses processes”\textsuperscript{31} and has become an integral part of modern software development dedicated toward quick delivery of solutions in Agile “vertical slices” which combines all the architectural layers such as creating a UI, Backend API and Database in one user story as opposed to developing each independently.\textsuperscript{32}

Examples of services which enable this behaviour include Azure DevOps from Microsoft; Atlassian which has many tools such as JIRA for Agile management, integration with handling repositories and CI/CD with Bamboo; GitLab which is branded as a DevOps platform with a massive number of features.
2.6 Load Balancers

Load balancers have become an essential part of a backend service architecture, with a drive to create high-availability and responsive systems, which with the rise of cloud computing and the ever increasing requirement to create distributed systems such as the benefits have been discussed by papers gauging the applicability of load balancing techniques and health checking for a cloud-based architecture in context of a hospital data management system. Further literature showcases how HAProxy in particular can be used to achieve an operational availability of 99.905% when creating a web services backend consisting of multiple load balancers and servers.

2.6.1 Operation of Load Balancers

![Figure 2.4: Operation of a Load Balancer](image)

Figure 2.4: Operation of a Load Balancer
Load Balancers operate through routing traffic utilising different techniques such as those in §2.6.2 with the ultimate goal being the even distribution of requests across a backend service’s infrastructure (as seen in Fig. 2.4); in the event of a system failure the traffic can be routed to another server and administrators can be notified of a fault which satisfies the goal of high-availability.

The drawback of this can mean that a load balancer can act as a single point of failure - in the event of a hardware or software fault this could cripple elements of a system whether they utilise load balancers for a region or for the entire infrastructure. Fortunately as mentioned in §2.6 loadbalancing with HAProxy which is the library this study focuses on, resulted in an operation availability of 99.905%.

Traefik
An element worth discussing is a competitor to HAProxy, called Traefik. The selling point of Traefik as a load balancer is its native support for cloud-native microservice architectures such as those discussed in §2.5.1 and an integration with Docker Swarm and Kubernetes.

2.6.2 Load Balancing Techniques

Roundrobin
Roundrobin load balancing functions through cycling over each server when delegating requests, for example, if 3 requests were received then one would be given each to Server A, Server B and Server C. This is effective especially when each server has the same or similar capabilities in handling requests.

Random
Random load balancing operates through randomly assigning clients to servers; with enough requests the load will be evenly distributed across the nodes in a network.

Weighted Load Balancing
Weighted load balancing allocates requests to servers by judging how each server can handle them. I.e. if Server A can handle half the requests per second Server B can, then upon 15 requests 5 would be allocated to Server A; 10 would be allocated to Server B.
DNS Delegation

While not strictly a load-balancing technique, DNS delegation is an important component of modern task delegation and is used for internet-based services; is particularly useful in cases where a service provider has geographically sprawling infrastructure to serve different regions for example. This operates by separating a namespace into different DNS zones where in a given zone there will be servers to handle the requests.
2.7 HTTP Servers

The concept of a HTTP server, also known as a web server, is to handle HTTP requests and serve HTTP content to the client. To date there are a number of highly-popular web server applications that can be used with the 3 most popular contenders\textsuperscript{35} listed below:

2.7.1 Nginx

Nginx is a popular open-source HTTP Server and is what will be used during this study, primarily due to its effectiveness but also as it has already been ported to Unikraft. Nginx provides functionality in serving HTTP content, with the ability to configure serving different content based on the DNS input (via DNS Resolving); TLS/SSL. Furthermore, Nginx also includes load-balancing capabilities and has been used to create microservice architectures for companies such as BuzzFeed.\textsuperscript{36}

2.7.2 Apache

Apache is another open-source HTTP server that boasts being the number one HTTP server on the internet,\textsuperscript{37} although this is marginal and depends upon which metric is picked.\textsuperscript{35} Apache also includes features such as load balancing; TLS/SSL.

2.7.3 Internet Information Services (IIS)

Finally, the third contender is IIS, which is a Microsoft-built web server intended to run on Windows with the .NET platform. In addition to being a web server with direct integration with tools such as ASP.NET\textsuperscript{38} features of IIS include various authentication mechanisms; TLS/SSL.

2.7.4 Means of Testing Services

This project will be using various methods to get data from the HTTP server instances such as curl, wget and Postman.

Curl

Curl defines itself as a tool to transfer data to or from a server, using a multitude of supported protocols including HTTP.

Wget

GNU Wget is defined as a free utility for non-interactive download of files from the web, supporting HTTP/HTTPS and FTP protocols.
Postman

Postman is an application primarily for testing APIs, which allows the specification of protocols and data that is being passed through, which is useful when testing on a system such as windows which does not have an easy application such as curl.
2.8 Unikraft Core Functionality

This section details the core functionality that is going to be required in the Unikraft Nginx and HAProxy applications being constructed, covering essentials for emulating a filesystem that will hold the configuration files and logfiles to understanding the networking modules and memory allocation.

2.8.1 POSIX Standard

“POSIX is a family of IEEE standards that supports portable programming.”

POSIX (Portable Operating System Interface) is the definition for a standard relating to interfacing with Operating Systems, the goal of a common interface is to abstract the finer details of the Operating System and any technicalities such as Instruction Set Architectures (ISA); instead focusing on providing a common interface including functions such as the one below which converts a string to an integer:

\[
\text{int atoi(const char *str)}
\]

Due to the power of POSIX for the programmer, its status as a standard and to remove an arbitrary barrier to porting applications to Unikraft, the project has implemented many libraries that implement the functionality of POSIX. These include: Newlib, LwIP; pthread-embedded and libvfscore which derives from another FOSS OS project.

This section will be detailing some of these libraries; how they fit into the Unikraft HAProxy and Nginx unikernels being created.

2.8.2 Newlib

Newlib is a FOSS C library intended for use in embedded systems, which is to say that it is a lightweight implementation of the C standard library. Unikraft offers this library for users that wish to run any programs involving C; offers libcxx as an implementation of the C++ standard library.

A primary selling point of Newlib is its boasted ease of portability requiring “altering a number of files and adding some directories”, with this ease of porting to a new system and its lightweight features such as providing an alternative to the sub-optimal printf() function via iprintf() which removes the bulky floating point capabilities of printf().

2.8.3 Memory Allocation

nolibc

To satisfy the POSIX standard, Unikraft implements POSIX memory-related functionality with a library called nolibc which implements features such as malloc, by calling the ukalloc implementation.
ukalloc

Features such as malloc (memory-allocate) are popular functions in C programming, Unikraft offers internal abstractions such as ukalloc which creates a native implementation of malloc, created to avoid the restriction of sufficient locking support. In addition to this, a rudimentary workaround for mmap and munmap functionality has been implemented via the ukmmap library.

2.8.4 Networking

Networking is a critical component of this project, requiring messages to be passed between not just client and server, but for the load balancer to communicate with the client and the servers, performing health checks, delegating tasks and serving the content.

LwIP

Thankfully, Unikraft has implemented LwIP which is a lightweight TCP/IP stack, also designed for embedded systems. HAProxy, for example invokes a function inside of proto_tcp.c:

```c
if (connect(fd,(const struct sockaddr *) addr,
    get_addr_len(addr)) == -1) {
```

LwIP implements the POSIX function `connect` which is defined as:

```c
int connect(int sockfd, const struct sockaddr *addr,
    socklen_t addrlen);
```

Ultimately proving this crucial functionality and POSIX-compliance is readily and effectively implemented by LwIP and thankfully included as a library module by Unikraft.

2.8.5 Virtual Filesystem

The virtual filesystem is another key component to the successful port of HAProxy and the successful usage of Nginx. This will be used to store the required configuration files for the two applications; any log files that the two may generate; regarding Nginx - host the content that will be served to the user(s).

vfscore

Vfscore is an internal library handles the POSIX standard for VFS implementation, with files such as vfs_syscalls.c that implement all VFS syscalls; other functionality includes handling mounting; handling file descriptors.
Persistent Storage with 9PFS

Unikraft presents some options for storage, ramfs and 9pfs, as data is required to be persistent for this project to emulate the normal conditions of a virtualised image. The 9pfs library inside of Unikraft implements further specifics regarding the file system through calls from vfscore.

2.8.6 POSIX Threads

POSIX threads are a means to enabling the functionality of threads, or threading, on POSIX-compliant systems. A thread shares the resources of a given program, as a form of lightweight process for executing concurrent applications it acts as a program counter, stack and a set of registers. The power of threads are revealed when programming in uniprocessor or multiprocessor environments by handling operations that can otherwise block the CPU, or by delegating the processing to other processors on the machine.43

pthread-embedded

Unikraft enables this functionality through its port of the pthread-embedded library. While HAProxy seems to use this functionality sparingly, it is a required module none the less; benefits from being implemented in Unikraft, where it was taken from another FOSS OS project called RWTH-OS.44

2.8.7 Stream Compression

Stream compression are a key component to the transfer of data between client and server, requiring one of a number of applications that specialise in streaming algorithms, such as gzip, zlib and libslz.

zlib

zlib is a FOSS compression library that handles compression, decompression and is an option for HAProxy to use; is otherwise used in a multitude of other very large software projects.45

libslz (SLZ)

SLZ is defined as a fast and memory-less stream compressor with no functionality regarding decompression. This was designed by Willy Tarreau who also created HAProxy, with the intent for SLZ to be a highly performant stream compression application for the load balancer, with SLZ boasting being up to 3 times faster with a 29% larger payload when compared to zlib.46
2.9 HTTP Benchmarking

2.9.1 wrk

What is wrk?

wrk defines itself as a popular HTTP benchmarking tool which is capable of generating significant load when run on a single, multi-core CPU; implements multithreading and scalable event notification systems such as epoll. For the purposes of this project, it will be used to generate data relating to latency, connection errors and ensuring that the backend infrastructures created can be compared to each other empirically.

How is wrk used?

![wrk being invoked from the CLI against google.co.uk](image)

wrk can be used via the CLI through invoking the command `wrk` followed by parameters specifying connections, duration of the test and the url, such as `https://google.co.uk` as seen in Fig. 2.5. Additionally, wrk supports the execution of LuaJIT (Lua Just-In-Time) scripts with a simple API to allow this behaviour, with functions such as:

```lua
function wrk.connect(addr)
function request()
function response(status, headers, body)
```
2.10 Alternative Unikernel Platforms

Although the primary focus of this paper is to utilise and discuss Unikraft as a unikernel platform; it is worthwhile to understand some of the alternative platforms and how Unikraft has leveraged components of these, or them from each other in order to create a robust unikernel.

2.10.1 MirageOS

MirageOS is a popular library OS referenced in much of the literature in this chapter, such as in §2.4.1 which compiles each application with the OS libraries and utilises the OCaml language, as opposed to C.

2.10.2 OSv

OSv is another unikernel platform designed with cloud-based infrastructure in mind, with claimed support for DevOps-style deployments; supporting a variety of runtimes from JVM to Erlang. As such, OSv provides support for locally running the unikernel in addition to pushing the unikernel to platform computing services such as AWS EC2 and Google GCE.

2.10.3 UniK

UniK is a tool that attempts to better automate and simplify the process of creating a unikernel through effectively creating a docker container with a unikernel architecture - in addition to handling the orchestration with a large emphasis on the use of Kubernetes - UniK also derives some of its modules from the MirageOS and OSv projects.

2.11 Why Unikraft?

Unikraft is a performant and effective tool for the purposes of this project; already contained the libraries, sub-systems and some of the applications that were required to fully realise the goal of porting HAProxy to a unikernel format. While MirageOS is also quite performant and has an extensive list of library functionality, an application such as Nginx was not readily available and the overhead of porting all the functionality of HAProxy to an OCaml environment was too much.
2.12 Chapter Summary

This chapter has provided an in-depth dive into the technologies surrounding virtualisation paradigms, the differences between them and the substance that they are formed from. In addition to this an analysis has been provided as to the positives and negatives of each technology.

Furthermore, technologies relating to what this study is focused around have also been discussed; load balancers, HTTP servers and a look into the modern approach to deploying these applications in an enterprise environment through microservices and DevOps environments has also been explored.

Finally, there has also been a discussion as to the means of benchmarking the backend service that this study has created and how the technologies chosen present the most optimal approach against other tools and applications.
Chapter 3

Design

3.1 Chapter Overview

This chapter is discussing the design of the Unikraft system, the tools and methods I will be using to complete the porting, configuring, building and running of HAProxy and Nginx. This information is sourced from the Unikraft documentation, the Plan 9 documentation and from the Docker Compose documentation.

In addition to this, there will be a discussion on the design of the network topology that will be created to satisfy the Backend Service Deployment aspect of this study; furthermore will include the topology of how each virtualisation paradigm requires each application to be instantiated and linked.
3.2 Unikraft

3.2.1 Makefile.uk

This file is integral to the Unikraft build system, it usually specifies where to grab the source files, what to include, any flags and targets to be set and which sources it will be building, allowing for maximum control over what you want to include in the final product.

![Diagram of Makefile.uk flow](image)

Figure 3.1: Showing the makefile flow

3.2.2 Config.uk

The Config.uk file is similar to the KConfig system and provides the Unikraft configuration menu with information and flags for the application. For example, if you were to include the lwip library, you would have the option to enable IPv6 functionality. In summary, Config.uk integrates the library build configuration options with the Unikraft build system.

3.2.3 Kraft

Kraft is a tool used to define, configure, build and run Unikraft kernels. In other words, this is intended to be a command-line tool for efficiently getting official and unofficial applications and libraries, configuring them; building them and running them. This provides a much more user-friendly and abstract method to creating a web server for example.
3.2.4 Handling the Virtual Filesystem

The virtual filesystem will be handled by Unikraft via vfscore (Virtual Filesystem Core) and the filesystem itself will be implemented by 9pfs (Plan 9 Filesystem Protocol) which is a network protocol created for Plan 9, an OS kernel and set of systems developed by Bell Labs in the 1980s.

For our use cases this allows for HAProxy to find its configuration in a path such as:

/etc/haproxy/config.cfg

Additionally, for Nginx as our HTTP Server, this means that the configuration files can be found; the files we wish to serve such as index.html.

![Figure 3.2: Configuring libvfs inside the Unikraft Configuration System](image)

In order to enable the virtual filesystem for the Unikernel, we must configure it, for example as seen in Fig. 3.2. This enables functionality with 9pfs and statically sets the default root device to fs0.
3.3 Docker and Docker Compose

As mentioned in §2.3.2, for the purposes of this project I will be using Docker and Docker Compose.

Below is an example of the docker-compose.yml configuration provided by the docker compose documentation:\footnote{51}

```
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
  volumes:
    - .:/code
    - logvolume01:/var/log
  links:
    - redis
  redis:
    image: redis
    volumes:
      logvolume01: {}
```
3.4 Backend Service Topology

The backend service for this project will be quite straightforward, as Fig. 3.3 above shows. This involves having a HAProxy instance sit in front of the Nginx instances, which for the purposes of this project will be 3 instances. The client will request the index page from our service, which HAProxy will handle and delegate to one of the instances.

3.4.1 Chosen Load Balancing Technique

The HAProxy instance that will be created will use the Roundrobin technique, due to the factors discussed in §2.6.2 - Roundrobin is beneficial for simplicity's sake it will be assumed that each Nginx instance will have the same capabilities; the requests will not be coming from separate geographical zones therefore Roundrobin will be an acceptable load balancing technique for this study.
3.5 Virtualisation Topology

3.5.1 Docker Virtualisation Topology

Fig. 3.4 shows how the HAPerxy and the Nginx server will be created with Docker Compose\textsuperscript{51} which is a tool used to define and run multi-container Docker applications. This involves creating a network bridge between host and the docker compose system, which will handle each container. The original design in Fig. 3.5 had more Nginx instances to emulate a real loadbalancing situation, however due to Unikraft networking bugs and constraints this had to be reduced to one.
3.5.2 Unikraft Virtualisation Topology

Fig. 3.6 depicts how HAProxy and the Nginx servers will be created. Unlike Docker, which has the Docker Compose tool, Unikraft does not yet have such a tool, meaning that each instance will have to be manually built and run. For each of these instances created, a network bridge will be provided between the host and the Unikraft qemu system. As mentioned previously, due to Unikraft constraints the original design of this topology had more Nginx instances as displayed in Fig. 3.7.
3.6 Chapter Summary

This chapter covered the tools at Unikrafts disposal such as Makefile.uk and how the virtual file system will be handled; it has also covered some of the tools that Docker enables us to quickly setup a multiple container system.

Lastly, there has been a discussion on how the Unikraft and Docker virtualisation paradigms require us to instantiate our applications; how the backend service topology has been designed and will be implemented for testing.
Chapter 4

Implementation

4.1 Chapter Overview

The Implementation chapter will be covering how HAProxy and Nginx will be configured, how each will be implemented in Unikraft; in Docker with an explanation and demonstration on running each.
4.2 Docker Implementations

4.2.1 Docker Compose
As previously mentioned in the design chapter, under §3.3, Docker Compose is a tool to create multi-container applications. Here we will be building, running and configuring 3 Nginx instances with a HAProxy load balancer container. This will be achieved with the Docker Compose yaml configuration and the Dockerfiles for HAProxy and Nginx.

```
docker-compose.yml
version: "3.9"

networks:
  backend:

services:
  weba:
    build: ./nginx
    expose:
      - "80"
    networks:
      - backend

  webb:
    build: ./nginx
    expose:
      - "80"
    networks:
      - backend

  webc:
    build: ./nginx
    expose:
      - "80"
    networks:
      - backend

  haproxy:
    build: ./haproxy
    networks:
      - backend
    ports:
      - "80:80"
```
This defines a network known as the backend, this will enable the containers to communicate with each other within the system - it also declares which services that we will be using such as with build specifying the build context, expose showing port 80 within the backend network (not to the host running docker) and connecting to the defined network called backend. Lastly, with the service called haproxy, the port configuration is used - which is exposing itself to the host machine.

The astute will realise that build is specifying a local directory such as ./nginx or ./haproxy - this is where the Dockerfiles for each service reside.

### 4.2.2 Implementing HAProxy in Docker

The setup for HAProxy was straightforward, requiring a Dockerfile and the haproxy.cfg file defined as seen in section 9.1 which would be parsed for configuration by HAProxy.

**HAProxy Dockerfile**

```bash
FROM haproxy:2.3.2
COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
```

### 4.2.3 Implementing Nginx in Docker

Setup for Nginx was slightly less straightforward, requiring the Dockerfile; the nginx.conf and site.conf files as seen in section 9.2 in addition to a directory holding the website content as seen in section 9.3.

**Nginx Dockerfile**

```bash
FROM nginx:1.15.6
COPY nginx.conf /etc/nginx/nginx.conf
COPY site.conf /etc/nginx/conf.d/site.conf
COPY static-html-directory /usr/share/nginx/html
```
4.3 Running Docker Implementation

Running the docker implementation is very straight forward, requiring the use of the docker and docker-compose tool.

4.3.1 Testing Nginx Container

In order to test whether our configuration works, it would require a quick test of the Nginx config which can be done with docker run. First we must build the docker image with docker build:

```
docker build --rm -f ./Dockerfile -t nginx-debug .
```

This line can be explained with the docker cli documentation:

- `--rm` Deletes the intermediate containers after a successful build.
- `-f file` Provides a path to the Dockerfile we will be using.
- `-t name` Gives the image a name.
- `.` Specifies that the PATH is in the current directory, allowing Docker to find all the files in question.

Secondly we run the application:

```
docker run --rm -it -d -p 8080:80 --name nginx-debug
```

Again, we can summarise the instruction as so with the docker cli documentation:

- `--rm` Removes the container upon exit
- `-it` Allocates a pseudo-TTY, creating an interactive bash shell in the container.
- `-d` Runs the container in the background and prints the ID of the container. (Rendering this optional)
- `-p ports` This command publishes the containers ports to the host machine.
- `--name name` Assigns a name or alias to the container which if left empty will be automatically generated.
- `nginx-debug` This is a reference to the name of the image created in build.
Running curl against localhost port 8080, where the container has been assigned gives the output as seen in Fig. 4.1, resulting in successfully getting the index.html page.

### 4.3.2 Testing HAProxy Load Balancer for Nginx Servers

To test HAProxy as the load balancer, I decided to push forward with docker compose now that the validity of Nginx was assured and invoked the up command:

```
docker-compose up
```

As seen in Fig. 4.2 docker successfully built the containers; created each service.
By running curl again against the localhost address, it produced the output as seen in Fig. 4.3, successfully returning the index document and proving that the docker virtualisation implementation is complete.
4.4 Implementing HAProxy in Unikraft

With a primary focus of this project being to port HAProxy to Unikraft in order to utilise the powerful application as a load balancer, this requires a lot of leg work both regarding configuring HAProxy to function with the Unikraft build system, and creating patches to the HAProxy source to become compatible with the current state of Unikraft; for example removing functions that handle cases for proprietary software which is out of scope for this project and Unikraft.

4.4.1 Porting libslz

As an introduction to the process of creating a port for Unikraft; generating a valuable product I set out to port libslz as libslz (SLZ) is a quick and powerful stream compressor HAProxy uses instead of other solutions such as zlib, as mentioned in §2.8.7. This involved creating a repository for the library, which would include a Makefile.uk file which would implement the process seen in Fig. 3.1 and the Config.uk file. Both of which can be seen in sections “SLZ Config.uk” and “SLZ Makefile.uk”.

4.4.2 Patches

In order to allow the HAProxy port to function correctly with the restrictions currently available, such as

4.4.3 Fixing libuuid

Unfortunately during development the libuuid library broke when trying to run make, with the error:

```
fatal error: uuidP.h: No such file or directory
```

This had to be resolved by changing the Makefile.uk for libuuid, fixing the path to the includes folder for example

From:
```
-I$(LIBUUID_BUILD)/include/public
```

To:
```
-I$(LIBUUID_ORIGIN)/include/public
```

Which finally resolved the issue.
4.4.4 Configuring HAProxy

The version of Unikraft used is 0.5.0, with the alias Tethys; the version of Unikraft being used is version 2.3.2 which is the same version that Docker will be using.

In addition to this, the architecture being used will be x86_64; the platform being used is KVM also known as “Kernel-based Virtual Machine”.

For configuring, I relied upon the kraft tool to automatically generate the .config file which Unikraft utilises, allowing me to use the KConfig style menu like seen in Fig. 3.2.

lwIP

lwIP is a lightweight TCP/IP stack primarily used in embedded systems, in order for HAProxy to build without error the IPv6 option had to be turned on as otherwise it would face issues with netinet includes.

HAProxy Dependencies

As seen in section 4.5.4, covering the kraft.yaml file we can see the dependencies for HAProxy that we will be using for this unikernel.

These are as follows:

- UK9P - Unikraft implementation of the 9 Plan system
- 9PFS - 9 Plan Filesystem
- Devfs & Devfs_automount - Handles the device filesystem i.e. forwarding input output requests to the appropriate drivers
- vfscore_automount_rootfs - Virtual File System Component
- vfscore_rootfs_9pfs - Virtual File System Component
- libparam - Handles parameters passed into the unikernel upon instantiation
4.4.5 Running HAProxy

To run HAProxy I will be using qemu-guest which is a Unikraft abstraction of the qemu-system tool.

After running make all which built the HAProxy Unikernel and inserted all of its dependencies, it was ready to run with the command:

```
qemu-guest -k ./build/haproxy_kvm-x86_64 -m 128 -b kraft0 -e $(pwd)/fs0 -a "vfs.rootdev=fs0 netdev.ipv4_addr=172.50.0.5 netdev.ipv4_gw_addr=172.50.0.1 netdev.ipv4_subnet_mask=255.255.255.0 -- -f /etc/haproxy/haproxy.cfg -d -V"
```

Where the arguments for the `qemu-guest` command can be summarised as:

- **“-k”** Specifies the path to the built kernel.
- **“-m”** Supplies the guest with a specific amount of memory in megabytes.
- **“-b”** Attaches a virtio-NIC to the existing Linux bridge, setting up the gateway for the Unikernel.
- **“-e”** Forwards the path as a filesystem device, which is to say it treats the specified directory as the filesystem.
- **“-a”** Provides the arguments that we wish to give to the kernel.
- **“vfs.rootdev=fs0”** Telling the VFS the root directory name
- **“netdev.ipv4”** Setting the IPv4 address, gateway and subnet mask for the unikernel.
- **“-f config path”** Tells HAProxy where to find the path to a config file, or directory containing config files (Within fs0)
- **“-d -V”** Setting HAProxy in debug and verbose mode
Fig. 4.4 displays HAProxy running after assigning the appropriate network configuration with lwip, calling HAProxy’s main function with the parameters passed in through QEMU; asserting checks such as the polling mechanism; parsing the configuration file and trying to bind the tcp address to port 80. Unfortunately when running `curl -Lvk 172.50.0.5` the connection to the IP address on port 80 failed, with no route to host found - showing that the bind to port 80 failed.

Fig. 9.2 (found in the appendix) shows that core 4 is at 100% load, which may suggest that HAProxy is stuck in a loop trying to bind to port 80.
4.5 Implementing Nginx HTTP Server in Unikraft

4.5.1 Configuring Nginx

Nginx has already been ported in Unikraft, meaning that the setup process is quite streamlined. Using the kraft tool to initialise the Nginx template and configure it, I enabled the option for Unikraft to provide the main class; set the virtual filesystem root to fs0.

4.5.2 Running Nginx

Instead of utilising the kraft run tool, I decided to use qemu-guest, as I did for HAProxy. The command for this was executed as such:

```
qemu-guest -k ./build/nginx_kvm-x86_64 -m 12 -b nginx -e $(pwd)/fs0/ -a "netdev.ipv4_addr=172.88.0.4 netdev.ipv4_subnet_mask=255.255.255.0 --"
```

The arguments that we pass in specify the IPv4 address for the guest and the subnet mask. Once the kernel is running we can now make a request to Nginx via:

```
curl 172.88.0.4
```

which returns the output that can be seen in Fig. 4.5 below.

Figure 4.5: Unikraft Nginx returning example index.html file
4.6 Loadbalancer Contingency Plan

Due to the unfortunate circumstance of the load balancers failure to bind to port 80, a contingency plan had to be made in the interest of time in order for testing the two virtualisation paradigms to be conducted. The result of this was a creation of a load balancer using Nginx.

4.6.1 Configuration of the Nginx Loadbalancer

The configuration for the Nginx loadbalancer can be seen below, with the only differences between the two implementations is the server IPv4 address, it is specified in the Unikraft config yet Docker utilises the specifier "weba:80" due to the configuration inside Docker-Compose.

```bash
worker_processes 1;
daemon off;
master_process off;
error_log /dev/null;

events {
    worker_connections 32;
}

http {
    upstream backend {
        #server 172.88.0.2:80;
        #server 172.88.0.3:80;
        server 172.50.0.2:80;
    }

    server {
        access_log /dev/null;

        listen 80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```
4.6.2 Implementing Nginx Loadbalancer in Unikraft

The Unikraft implementation was partially completed in terms of configuration and setup for running, requiring a copy of the directory nginx was created in and cloning it to begin the setup for the loadbalancer. Next was inserting the nginx.conf file into the fs0 root; running the load balancer with a shell script called runNginx.sh, with the parameter for an IPv4 address that would be assigned to the loadbalancer. Used as so: 

```
./runNginx.sh 172.50.0.3
```

Where the contents of this script were:

```
qemu-guest -k ./build/nginx-loadbalancer_kvm-x86_64 -m
1024 -b kraft0 -e $(pwd)/fs0/ -a "vfs.rootdev=fs0
netdev.ipv4_addr=$1 netdev.ipv4_subnet_mask
=255.255.255.0 netdev.ipv4_gw_addr=172.50.0.1 --"
```

The expected output however was not expected when using 3 Nginx instances, with Nginx trying to communicate with the connected nodes only to find that they shared the same MAC address, causing mayhem. Reducing the number of nodes 1 did not fix things either, with the response to ARP 42 Who has 172.50.0.2? Tell 172.50.0.3 as observed via tshark on the kraft0 interface, getting stuck in the gateway.

With this issue relayed to the NEC team, this issue was quickly fixed through changes to the qemu-system abstraction, qemu-guest - finally allowing for requests to 172.50.0.3 routing to the Nginx HTTP server node. (Many thanks here to Alex Jung and NEC Labs for resolving this issue.)

4.6.3 Implementing Nginx Loadbalancer in Docker

Aside from the nginx.conf file, nothing significant changed with the replacement of the HAProxy loadbalancer, with the Nginx implementation other than:

```
nginx-lb:
build: ./nginx-loadbalancer
```
4.7 Chapter Summary

In summary, this chapter has covered the work that has gone into creating a Unikraft port for SLZ and HAProxy, provided the configuration for HAProxy and Nginx and discussed some overcome challenges such as the patches implemented in the HAProxy port; fixing the libuuid projects Makefile.uk. In addition to this, the tools used use as kraft, qemu-guest as a wrapper for qemu-system and the finer configuration details have been discussed. These factors resulting in an observable success with HAProxy and Nginx running in Unikraft.

Furthermore, this chapter has discussed the steps taken to setting up the container system with docker and docker-compose, verified that Nginx is working; that HAProxy delegates requests to the Nginx servers as intended.
Chapter 5

Testing

5.1 Chapter Overview

This testing chapter is covering the work that was required in setting up suitable testing and benchmarking for the infrastructure created in this project. In addition to this, there is a slight review on the image sizes of the Unikernel application against the Docker equivalent.

5.2 HTTP Benchmarking with wrk

As discussed in §2.9, in order to stress-test and benchmark the solutions created in this study the tool “wrk” will be used, this tool allows the users to specify the threads used, number of connections and the time that the test will be run against a given HTTP endpoint.

5.2.1 wrk Command Template

The tests that will be conducted against each implementation will be following the format as described below:

```
wrk -t10 -c32 -d60s {URL}
```

The above command invokes wrk and supplies the number of threads with `-t10` as 10, the number of connections with `-c32` as 32 and the duration with `-d60s` as a 60 seconds long test. The URL will be dependant on the address supplied to each implementation, such as `127.0.0.1`. 
5.2.2 Unikernel Testing

To purely measure the performance of the virtualisation paradigms `qemu-guest` will be supplied with arguments that run the qemu system and the images/binaries being tested on different CPU cores. This is achieved through the “-c” and “-p” flags, of which will be supplied with the values of 2 and 3 respectively.

**Single Nginx Instance**

<table>
<thead>
<tr>
<th>Latency (ms)</th>
<th>Average</th>
<th>Stdev</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,500</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No. Requests handled per second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average</td>
</tr>
<tr>
<td>100</td>
</tr>
</tbody>
</table>

**Thread Statistics**

- **Latency (ms):** 523.04/241.53/1940
- **Requests Handled Per Second:** 8.61/7.3/90
- **Socket Error Count - Timeout:** 13

**Raw Stats (Avg/Stdev/Max):**

- Latency (ms): 523.04/241.53/1940
- Requests Handled Per Second: 8.61/7.3/90
- Socket Error Count - Timeout: 13
Nginx Loadbalancer

Raw Stats (Avg/Stdev/Max):

- Latency (ms): 317.05/211.53/1950
- Requests Handled Per Second: 8.74/6.53/50
- Socket Error Count - Read: 4948 / Timeout: 3
5.2.3 Containers Testing

Testing the container system was a straightforward venture, requiring the command `docker run --rm -it -d --cpus="1" --cpuset-cpus="3" -p 8080:80 --name nginx-single-inst nginx-single` which is designed to limit the Docker image to one CPU core and `docker-compose up` to be run and using `http://127.0.0.1` as the URL value.

Single Nginx Instance

![Graph showing latency and requests per second](image)

Thread Statistics

No Socket Errors.

Raw Stats (Avg/Stdev/Max):

- Latency (ms): 1.71/1.27/71.78
- Requests Handled Per Second: 1850/270.04/2470
Nginx Loadbalancer

No Socket Errors.

Raw Stats (Avg/Stdev/Max):

- Latency (ms): 11.86/8.68/57.31
- Requests Handled Per Second: 276.3/182.3/940
5.2.4 Unikraft Website

While not in a controlled environment, it was decided that a good comparison for the local Unikernel performance would be to test against the Unikraft website, which is run on an Nginx Unikraft binary.

No Socket Errors.

Raw Stats (Avg/Stdev/Max):

- Latency (ms): 82.23/9.85/163.37
- Requests Handled Per Second: 36.5/7.97/60
5.3 Image Sizes

An important benefit and boast of unikernels is the relatively tiny image sizes, as discussed in §2.4.1. This section is dedicated to observing the size differences between the containers and unikernels creating during this project.

5.3.1 Container Sizes

![Image Size data for Docker infrastructure](image)

Figure 5.1: Image Size data for Docker infrastructure

Note: Image edited to crop empty space and irrelevant values, the full screenshot can be found in the appendix.

In order to get the image sizes from the docker infrastructure the command `docker system df` with the attribute `-v` had to be used, which gave a set of data relating to the currently running images.

**Nginx**

There are 3 Nginx containers as seen in Fig. 5.1, of which Docker has utilised to create a shared size of 109.1MB, with each instance having a unique size of 555 bytes for runtime.

**HAProxy**

The HAProxy container has an image size of 94.22MB.

5.3.2 Unikernel Sizes

To get the sizes of the unikernels, the command `ls -sh` had to be used against the created binary in the build directory. This produced an output describing the size of the file in mebibytes (MiB)

**Nginx**

![Size of the Nginx unikernel](image)

Figure 5.2: Size of the Nginx unikernel

The size of the Nginx binary is 1.7MiB, which converts to 1.78MB.
The size of the HAPProxy Unikraft binary is 2.9MiB, which converts to 3.04MB.

5.4 Chapter Summary

This chapter has covered the usage of wrk in producing HTTP benchmarking data and producing results regarding: latency, requests handled per second, and the number of socket errors, for each type. Additionally, the chapter has covered the image sizes for Nginx and HAPProxy for the Docker containers and the Unikraft binaries.
Chapter 6

Results and Discussion

6.1 Chapter Overview

This chapter will involve discussion from the results of testing in the previous chapter and offer an insight as to what has been gathered; why it could have been. The two major points covered are the HTTP benchmark statistics and the image/binary size numbers.

6.2 HTTP Benchmark Data

6.2.1 Unikraft vs Docker

![Figure 6.1: Showing CPU usage during Docker image testing](image)

Firstly, the data gained from testing gave an excellent insight to the differences in capability between Unikraft and Docker. While the single Unikraft binary was able to handle at maximum, 90 requests a second the single Docker image was able to handle 2470, a 27.4x increase in request response capability. Furthermore, the maximum latency from the Docker image was 71.78ms whereas Unikraft achieved a maximum latency of 1940ms, which shows that the comparable max latency was 27x larger in Unikraft, than with Docker. Similarly, the average latency for the Unikraft binary was 523.04ms, where the Docker image achieved a latency average of 1.71ms.
This experiment has depicted a very damning image of Unikraft, however it is worth noting that while attempts were made to restrict the Docker image to one CPU core, the same conditions Unikraft was operating - the Docker image actually utilised all 4 cores, as depicted in Fig. 6.1 creating an unfair advantage to the Docker image. This was an issue that could not be resolved despite close scrutinisation against the Docker documentation.

6.2.2 Standalone Instance vs Loadbalancer

The testing results also provided insight to the overhead added when using the loadbalancer infrastructure, with the average latency of the Docker loadbalancer increasing to 11.86ms, although a maximum of 57.31ms, lower than the standalone image; severely lessened throughput regarding requests handle per second, with the average going from 1850 to 276.3 and the maximum going from 2470 to 940 once the load balancer was being used.

An observation from this would be that in the event a load balancer is not required to evenly distributed high-loads across a variety of nodes it would be better to simply directly utilise the the standalone instance.

To contradict this however, the unikraft infrastructure had better overall statistics with much lower average latency and a similar number of requests handled per second, aside from the maximum requests made per second. An answer for this however, can be revealed from the number of socket errors thrown, which was significantly high, with almost 5000 read socket errors.

6.2.3 Unikraft Website

While not an official part of the experiment, some testing was conducted against the Unikraft website, which is hosted with an Nginx Unikraft, unikernel. The results aside from latency, which for the locally tested unikernel were on average 6.4x longer than the Unikraft website. The requests handled per second however were comparable, with the website handling over 4x the requests per second on average; 30 less than the local unikernel at maximum.
6.3 Image Sizes

The image sizes between the Docker and Unikraft implementations differed drastically, with the Unikraft implementation showing a max image size no larger than 3.04MB, where the smallest Docker image size was 94.22MB, showing that the HAProxy container was 31x larger than the HAProxy unikernel image. Similarly, the Nginx Docker implementation was 61.3x larger than the Unikraft implementation. The effects of this are particularly large when boot times have been shown to grow linearly with the increase of VM image size.\textsuperscript{11}

<table>
<thead>
<tr>
<th>Virtualisation Method</th>
<th>Image Size in MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Docker</td>
<td>94.22MB</td>
</tr>
<tr>
<td>Unikraft</td>
<td>3.04MB</td>
</tr>
</tbody>
</table>

6.3.1 Slimming Docker Image Sizes

There are considerations that were not taken during this project, such as utilising alpine version of the Docker images.\textsuperscript{58} The Docker Alpine image is based off a tiny security-oriented Linux distro with a base size of 5MB.\textsuperscript{59} With this in mind, many Docker Images support an alpine version to create a lightweight image for users constrained by requirements of small image sizes.

6.3.2 Slimming Unikraft Image Sizes

The Unikraft implementations could have also undergone improvements regarding image size through careful optimisation, with Nginx already being set up there were considerations made in separating modules which could be toggled on or off, with better analysis and elimination of any unrequired modules this would have likely shrunk the size of the Nginx image.

Further, the HAProxy port did not have any modules other than the inclusion of the main file that were able to be toggled on or off. An example of a module that could have been disabled is the \texttt{proto udp.c} file, which would have been irrelevant for the use cases of this project which relied on TCP/IP functionality.
Chapter 7

Project Retrospective

7.1 Chapter Overview

This chapter intends to discuss what has been learned over the course of this project and to provide a retrospective analysis of the positive outcomes of the projects, the barriers during development and to consumer adoption; understanding some of the shortcomings in regards to features the proof of concept port could have delivered upon.

7.2 What went well

This project has generated lots of knowledgeable material about low-level systems such as unikernels and the modules they comprise of: POSIX standards and their applicability to modern OS’s; build systems through Makefiles and the Unikraft build system. The project has also given good insight to virtualisation systems and their applications in enterprise environments such as cloud-native architectures; highlights some key features that unikernels will need to adopt to reach mainstream status.

7.2.1 POSIX

Learning about POSIX in §2.8.1 revealed lots about the requirements of low-level system through programming from a high-level language such as C and the effectiveness of projects such as newlib and lwIP which provide lightweight implementations for some of POSIX’s intended functionality. In addition to this, it has further highlighted concepts such as file descriptors and their applicability outside of file systems and the existence of a common interface for functionality such as threading.
7.2.2 C Programming Language

Through scrolling through the codebase for HAProxy and lwIP chasing bugs and developing a greater understanding of the network code and implementation there has been extensive exposure to the common practices in the C language, frequently dealing with macro arguments acting as functions, grasping various bitwise operations being utilised in conjunction with flags such as:

\[(\text{flags} \& \sim (\text{MSG\_DONTWAIT} | \text{MSG\_MORE})) == 0\]

And a general reinforcement of my personal understanding regarding C fundamentals such as pointers when passing a pointer datatype into a function to receive the target IP address to connect to.

7.2.3 Build Systems and Unikraft

A crucial part of the process in porting the libraries SLZ and HAProxy was to utilise the Unikraft build system with its Makefile.uk and Config.uk files; the custom Makefile. Beyond this further delving into Makefiles was required when building HAProxy with its own Makefile on Ubuntu and invoking the Makefile for wrk. Facing problems such as libuuid not building correctly due to missing header files; being able to solve them as seen in §4.4.3 was a satisfying result of being acclimatised with the Unikraft workflow.

Through facing other problems such as libslz not being pathed correctly and investigating the libuuid problem with tools such as `make -n` and piping the output to a file; observing how all of the object files are being linked by the linker to create the binary image which is the Unikernel as demonstrated in Fig. 7.1.

![Figure 7.1: Diagram of the Unikraft Build Flow with HAProxy](image_url)
7.2.4 QEMU

Another important tool that has been utilised through the abstracting application qemu-guest is QEMU, which is a popular processor emulator that runs KVM and Xen virtual machines, both of which Unikraft actively uses and supports. This has allowed for the Unikraft binaries to be quickly and efficiently virtualised as guests on the Ubuntu host.

7.2.5 Virtualisation Paradigms

This paper’s primary focus was exploring different virtualisation paradigms and has come to display a familiarity for the different methods of virtualisation available and has developed an advanced understanding of the current enterprise standards employed by companies such as Google who are spinning up billions of containers each week.

Containers

Containers have proven a currently effective method for executing scalable infrastructure on on-prem, cloud native and hybrid infrastructures with a big supporting base for images; extensive documentation. While evidence in literature and this project have shown that containers have comparatively large image sizes against even inefficient unikernel builds, the removal of the guest OS and sitting on the hypervisor instead has been a significant improvement, and as seen in Fig. 5.1, allows for common data to be shared; how improvements have been made through the Docker platform to accommodate limiting image sizes further through the use of lightweight Linux distros such as Alpine.

Unikernel Platforms

Unikernel platforms has also been a point of discussion, not just Unikraft which this project has extensively used but also the literature heavily referencing the MirageOS project; UniK, with each presenting their own benefits and workflows to the concept of a unikernel. Where UniK tries to imitate Docker through taking applications and automatically generating a Unikernel binary, MirageOS tries to create a more precise and bespoke system with OCaml, similar to the Unikraft workflow. Additionally, this paper has shown how this novel technology rips components from each other and other OS projects to serve their own means which is discussed further in §7.2.7.

Orchestration Systems

Furthermore, this paper has taken an extended look into how containers are used in various infrastructures through orchestration tooling such as Kubernetes, which fits into modern workflows, both of which are discussed further in §7.2.6 and §7.2.7, highlighting that orchestration systems have become a must in controlling the infrastructure in a microservices architecture.
7.2.6 DevOps

The process of DevOps, in extension of service deployment is something that was not something actioned in this paper but simply discussed; while Docker images can be seamlessly integrated into workflows with platforms like Azure DevOps, who provide documentation of deploying apps with containers, building and pushing images; how to handle Kubernetes clusters - unikernels do not receive the same level of integration and support, similar to the lack of native orchestration systems.

Thankfully, however, this issue mostly persists in the issue of continuous deployment; presents a limited barrier as tools such as Unikraft push to make the pulling, configuration, building and running process a trivial problem.

7.2.7 FOSS and OS Development

Over the course of this paper there has been an extensive look into free, open source software in the form of Unikraft and the libraries it has implemented, HAProxy and SLZ to MirageOS and the MiniOS project which Unikraft has cannibalised components of such as vfscore. In addition to this, it has shown the work that goes into an OS to provide functionality on the bare metal and standardised abstractions for developers to interface with and generate cross-platform functionality as discussed in §7.2.1.

Moreover, generating a deeper understanding of networking with sockets, bridges and gateways to interface from host to guest and for guests to communicate with eachother through using their own gateway via tools such as tshark, a network analyser and observing how IPv4 addresses are polled and returned has been a crucial part in networking with Unikraft.
7.2.8 Partial Port of HAProxy

While unfortunately the full port of HAProxy was unable to be completed during this project due to constraints discussed in §7.3.1 there was significant progress made, getting the application running, parsing the configuration file and with rudimentary networking, aside from the binding to port bug. This provided an excellent insight to not only working with the Unikraft build system and fixing issues as mentioned in §7.2.3 but also to how HAProxy was constructed, its interfacing with the users and groups namespace; having to implement patches to accommodate Unikraft specific behaviour such as removing netinet header files and defining \texttt{AF\_MAX} manually.

Additionally, this paper covering the port has provided lots of knowledge regarding how to configure HAProxy; the various options available such as load balancing methods and DNS resolving which can be used to resolve to regional load balancing servers.

Development of the Contingency

Due to the evident barrier of HAProxy not being functional in time for testing and an Nginx unikernel having to also be configured as a load balancer to achieve the papers primary aim of comparing the virtualisation paradigms, this was also a positive experience born of a negative - covering how to also configure Nginx to balance load across nodes in a network. Despite this also being barriered due to technical constraints which is covered in §7.3 it has at least provided an insight to the alternative of having a pure Nginx infrastructure.
7.3 Unikernel Barriers

This section will be detailing multiple barriers this paper and project has found regarding Unikernels, from the barriers met during porting HAProxy; setting up contingency plans with existing libraries; the overall porting process to barriers stopping the adoption of unikernels in their current state.

7.3.1 Porting HAProxy

A major barrier in this project was the porting of HAProxy, while there were successes in getting SLZ ported and ready, getting the dependencies of HAProxy correctly satisfied and selectively pruning source files such as wurfl.c (which were irrelevant for this study) resulting in the successful building of the HAProxy unikernel binary, the project was hindered with many issues such as those involving proprietary code, code that referenced user groups for namespaces which is irrelevant regarding a unikernel or code that had to be simply dropped as no solution could be found, such as an issue with allocating proxy counters and finally the issue regarding HAProxy’s failure to connect to the Nginx nodes.

While this has presented an excellent opportunity for personal growth and familiarity with low-level systems it did however stop the project in its tracks and result in a failure to deliver an aim of this project. Moreover, it confirms that while the process of creating a unikernel has been made much more user-friendly, it is not quite yet at the stage to compete against grab-and-go enterprise ready systems such as Docker.

7.3.2 Nginx as a Load Balancer Contingency

Due to the issues mentioned in §7.3.1, a contingency plan was executed in order to get some form of load balancer running on the Unikraft platform; this was to be executed with Nginx. Unfortunately there were further issues and complications with this as issues with networking and qemu-guest persisted causing the loadbalancer to timeout after repeatedly entering a loop of TCP Retransmissions. Thankfully, a fix was pushed by the NEC team however which fixed this issue by ensuring that each Unikraft instance would be assigned a unique MAC address.

Symptoms of the MAC Address Bug

Part of the issue was found by the Nginx loadbalancer returning 502 Bad Gateway; upon closer inspection with the tshark tool where curling the IP address of the load balancer would result in the load balancer polling for a chosen endpoint, and a request was sent back to the Nginx load balancer which did not arrive, getting stuck in the gateway - or sending requests to the Nginx node and getting caught in an endless cycle of TCP Retransmissions until it timed out.
7.3.3 Porting Process

The process of porting unikernels has been greatly improved from the requirements of low-level systems knowledge; the time; the budget to sustain a project to take an application and ensure it could enact its functionality on bare metal, or at least on a type 1 hypervisor, such as Xen. Despite this, porting a unikernel requires still requires some prerequisite knowledge in the best case: of getting project sources, interfacing with Makefiles and tools such as QEMU in order to get, configure, build and run the binaries. However depending on the scope of a library port, further knowledge may be required, as has been in this study to delve into the project code and resolve any problems in the form of patches.

7.3.4 Adoption

Lack of Awareness

It would appear that unikernels, while not a new technology, have limited awareness outside of researchers and enthusiasts of the technology. This presents a natural barrier as the pool of open-source contributors is limited through a lack of exposure to the technology in its current state; still holding the stigma of being a time consuming and expensive tool to utilise which renders it niche. This serves to only hinder the growth of unikernels as a paradigm and limits the amount of early adopters with the new direction of application unikernels are trying to fit into.

Lack of Orchestration

Aside from the lack of awareness of unikernels as a novel technology, there is a distinct lack of native tools for orchestrating unikernels. While tools such as UniK attempt to bridge this gap, it does not provide the full benefits benefits that unikernels provide such as a small image size; instead creating a container-based unikernel-architecture image. This presents a barrier as applications are being virtualised constantly and infrastructure that is defined using orchestration systems with Kubernetes for example is able to handle this, where unikernels would currently require an in-house system to be created, or lots of manual work.

Missing Functionality

Further issues that present barriers to the adoption of unikernels is the lack of adequate and robust support for libraries, with a limited set of libraries being able to run certain applications, more complex applications that require dependencies that unikernels cannot currently provide are certainly current issues. In addition to this, platforms such as MirageOS require that applications be built with OCaml, creating further barriers to the uptake of the unikernel project.
7.3.5 Connection Limitations in Unikraft

Currently, due to restrictions in the implementation of the networking with Unikraft, there can only be 32 connections made to a node. This is a massive restriction where in a normal environment handling 10,000 concurrent connections with a single node have become commonplace.
7.4 Potential for Unikernels

Despite this paper finding the porting of a unikernel application as a challenging feat; there has been lots of positive insights to the novel technology, which show great promise for the technology once it begins to grow out of its teething phase.

7.4.1 Backend Service Deployment

Unikernels have demonstrated their potential in enterprise backend service deployment for reasons that can be summed up as: Efficient, Ideal for a Microservice Architecture; ideal for an orchestration system.

Efficiency

The efficiency of unikernels is displayed in §6.3 which shows how tiny the binary size even an unoptimised unikernel can achieve; when compared against a standard docker container the difference is staggering. Video streaming alone is estimated to account for 79% of global mobile data traffic by 2022, with an expected data transfer size growth of 55% from 2017 figures. It is evident that services such as Youtube, Netflix and Spotify that specialise in media streaming would stand to benefit greatly from harnessing the efficiencies of technologies such as unikraft to serve this colossal amount of data and content, through means such as elastic content delivery networks powered by unikernels.

Ideal Microservice Platforms

Due to the efficiency presented with the unikernel virtualisation paradigm, this also renders it an appropriate technology for implementing a microservice architecture, with microservices designed to be as simple as possible, limiting the codebase for a given service, being independently deployable and scalable and ultimately during development microservices are usually created, tested, re-implemented; redeployed rapidly - a use case unikernels are already poised to fit. Moreover, a microservices architecture is design to be fault tolerant - with the optimisation and security benefits of a unikernel this does nothing but show the potential this novel technology has for software engineers.

Orchestration

Lastly, while no native solution currently exists - there is potential for unikernel projects to take precedent from container orchestration systems such as Docker Swarm and Kubernetes and develop a robust application for the seamless automation of deployment, scaling and management of unikernels.
7.4.2 DevOps

Similar to orchestration, unikernels have the potential to be actively supported in DevOps environments and CI/CD pipelines, as previously mentioned, with tools such as Unikraft making the unikernel creation process trivial, it would be straight forward for organisations to begin supporting and providing documentation on how to set up the configuration for a CI/CD pipeline using unikernel binaries.

7.5 HAProxy - Missing Features

The HAProxy port was unable to implement all of the intended functionality the project had created, with some notable systems excluded such as OpenSSL, primarily due to time-constraints and the barriers the project encountered, for an effective product to be created, extending beyond the proof-of-concept the port is in its current state, TLS/SSL functionality would be an absolute must. Additionally, modules for systems such as 51d (51degrees) and wurfl which deal with device detection/discovery were removed from the final build which while not essential, are features that users of HAProxy would expect to be present, by offloading the work of device detection from application servers to the load balancer for instance.

Lastly, a big feature missing from the HAProxy port is the ability to use epoll(), which is just being implemented into Unikraft and would potentially require some tweaking of the systems to ensure its reliability, constraining any developers further as to what features of the robust load balancer they can use.
7.6 Chapter Overview

This chapter has generated a detailed project retrospective for the work involved in this paper. The successful actions in the research and development with a particular interest in the compared paradigms and their outcomes in addition to the highlighted workflows that have been discovered and required to reach this point. Moreover, there has been a discussion on the shortcomings of this project, the unikernel virtualisation paradigm and the consequences involved.

However, the potential for this novel technology have also been outlined with current and upcoming niches highlighted which would be ideal for a unikernel to be deployed.
Chapter 8

Conclusion

8.1 Project Aims Review

Generate research for current industry standards in backend service deployment; tools utilised in enterprise environments and technologies to enable a microservice architecture.

Over the course of this project, from Chapter 2 providing research on concepts such as Microservices, DevOps and the tools that enable them in addition to expanding on the applications and technologies that containers and unikernels consist of to the Chapter 6 discussions of tools such as Docker being put to test and analysed; Chapter 7 reviewing how the technologies held up to scrutiny. To this end the paper has been successful, highlighting key points in modern service deployment and infrastructure.

Study the performance capabilities of the unikernel and container system. Despite the barriers mentioned in other sections, there has been valuable test data generated from this project that shows how capable the two virtualisation paradigms currently are regarding handing requests and availability of the service meant to provide. Additionally, there has been considerable insight as to the quirks and restrictions of Unikraft and by extent, unikernels in particular in its current state. The consequence of this is the ability to create an informed decision on the applicability of the novel technology in its infancy for an enterprise environment.

Create a unikernel port of HAProxy to the Unikraft platform; to integrate this into a microservice architecture.

Unfortunately, this goal could not be fully realised, with the project being stuck on some bugs for a manner of weeks; finally requiring an external fix in order to get rudimentary networking working at a point in time where fixing the remainder of the bugs was simply not possible. Despite this, there was lots of research generated showing the barriers involved in porting a large FOSS project such as HAProxy to a unikernel; additionally the contingencies actioned by
creating the Nginx load balancer as a working substitute showed the similarities in configuring a simple load balancer; the shared issues they might face through the Unikraft build process with its currently restricted networking capabilities. Furthermore, the learning curve for integrating the HAProxy application was initially small regarding Makefiles, configuration and Unikraft's own build system requirements - only requiring a much more involved approach once application specific functionality began breaking down due to internal problems.

**Create an identical microservice architecture using the Docker platform.**

Contrasting the Unikraft experience, the creation of the Docker architecture was expectedly a pleasant experience with the years of fine-tuning and large contributor pool with large industry backing. Aside from the short learning curve involved in setting up Dockerfiles, the Docker-Compose yaml and handling internal networking this section of the project was unproblematic aside from an issue relating to the unpermitted usage of CPU cores.

**Conclude the differences between the virtualisation paradigms in a backend system environment.**

This paper was able to succinctly find the differences between each virtualisation paradigm from an abstract research level, understanding the benefits of unikernels and the steps containers have taken to greatly improve and distance itself from traditional VMs. Moreover, the test data gave empirical evidence as to the current effective of each tool; how that affects a backend system environment in production.
8.2 Future of Unikernels

Over the course of this project, unikernels have frequently shown their potential in taking an enterprise environment by storm; the emphasis on inherent security, tiny image sizes and its efficiencies in mass-orchestration, despite a lack of native orchestration system. However there are many barriers unikernel projects such as Unikraft need to overcome to make the transition from VMs and Containers seamless for users, namely in getting applications ported to the platform and having a robust implementation of system libraries including networking capabilities which are currently heavily restricted, rendering performance completely unsuitable for an enterprise environment in its current state.

In addition to creating the tools and infrastructure that incite usage of the platform, further work needs to be conducted from the unikernel community as a whole to raise awareness of the paradigm. Unikernels still carry the stigma of being a colossal undertaking from projects involving creating library OS’s from scratch.

Finally, another barrier to enterprise adoption is the lack of native orchestration tooling that traditional VMs and containers have readily available, enticing developers to utilise those solutions to create a robust, scalable, efficiently managed and fault tolerant system over unikernels. This is an issue that the unikernel community should be addressing; while projects such as UniK attempted to bridge this gap it cannot be classed as a native tool and still carries the baggage of a container system.
8.3 Future Work

As a whole this study has been a success in generating research and knowledge around virtualisation paradigms, backend service deployment, low-level systems, tooling such as tshark, wrk and even tmux to enable the fixing and understanding of application features, diagnosing problems, HTTP benchmarking and optimising terminal-based workflows. However, due to the unfortunate state HAProxy has been left in it is evident that the immediate next course of action would be officially diagnosing the problem stemming from a failure to bind the supplied IPv4 address to port 80; complete a minimum viable product for the loadbalancer.

Secondly, there are many missing features from HAProxy that would be integral in a normal usecase such as TLS/SSL functionality, powered by the OpenSSL library which is already available in Unikraft. These features would be the next point of action in continuing and finalising the HAProxy port. Furthermore, the entire workflow for creating a HAProxy Unikraft binary is currently sub-optimal, with changes to the Config.uk allowing the user to fine-tune the application beyond whether it will use the original main function, similar to how Nginx operates, this would enable a much more efficient and rounded implementation. Small examples of this would be the ability to toggle debug messages on or off, giving the user the ability to understand what HAProxy is doing without having to delve into the implementation and work it out, enabling OpenSSL support, enabling the 51degrees library (Should it be implemented at all) and enabling or disabling functionality such as TCP, UDP and so on.
8.4 Learning Outcomes and Final Notes

Regarding my personal growth across this study, it has been an often challenging and frustrating project - which in turn has made it extremely interesting and provided a wealth of information across the topics covered, with a personal highlight being the ability to delve into the lwIP library and chasing connection bugs which in turn allowed me to become acquainted with how the library functioned and the quirks of C development architecture, sometimes requiring the grep tool to find where functionality was being called or set inside the codebase. Additionally, the requirement to do background research has reinforced my knowledge of DevOps workflows and the robust scalable orchestration methods available to developers.

Additionally, this project has taught that persistence is key when solving complex challenges; that there is high benefit to a regimented working and learning schedule with an emphasis time management. Over the first term time spent on the project was touch-and-go causing delays where in the second term the adoption of a tight schedule, using tools such as JIRA benefitted me greatly in working to the deadline, despite issues with the full functionality of the product.

To conclude, this project has been a big personal undertaking, requiring lots of learning and in turn providing me with an enhanced skill set for problem solving ability, toolsets that can be used and low-level systems knowledge bound to benefit future projects when facing bugs of a complex nature and above all enterprise relevant skills appropriate for future work.
Chapter 9

Appendix

9.1 HAPerxy Config

Below is the haproxy.config configuration that will be used for both Unikraft and Docker implementations aside from bind and server weba IPv4 addresses, where in the Docker implementation they are set to * and weba respectively.

global
    log /dev/stdout local0
    maxconn 32

defaults
    mode http
    log global
    option dontlognull
    retries 3
    timeout check 50s
    timeout connect 50s
    timeout client 50s
    timeout server 50s

frontend www
    bind 172.50.0.5:80
    maxconn 32
    mode http
    timeout http-keep-alive 500
    default_backend nginx-servers

backend nginx-servers
    mode http
    balance roundrobin
    option forwardfor
option http-server-close
option httpchk GET /
server weba 172.50.0.2:80 fall 3 rise 2 maxconn 32

http-check expect status 200
9.2 Nginx Config

9.2.1 HTTP Server nginx.conf

worker_processes 1;
daemon off;
master_process off;

error_log logs/error.log debug;

events {
    worker_connections 32;
}

http {
    include mime.types;
    default_type application/octet-stream;

    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;

        location / {
            root /nginx/html;
            index index.html index.htm;
        }
    }
}
9.2.2 Loadbalancer nginx.conf

```nginx
worker_processes 1;
daemon off;
master_process off;

error_log /dev/stdout debug;
#logs/error.log debug;

events {
    worker_connections 32;
}

http {
    upstream backend {
        server 172.50.0.2:80;
    }

    server {
        access_log /dev/stdout;
        listen 80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```
9.3 HTTP Server Static Content

The HTTP Server being created with Nginx will contain one document called index.html with the contents seen below:

```html
<!DOCTYPE html>
<html>
<head>
    <title>Nginx Server Reply</title>
</head>

<body>
    <h1>Hello!</h1>
</body>
</html>
```
9.4 SLZ Unikraft Config

SLZ Config.uk

```
menuconfig LIBSLZ
  bool "slz - stateless ZIP library"
  default y
```

SLZ Makefile.uk

```
# Library registration
$(eval $(call addlib_s,libslz,$(CONFIG_LIBSLZ)))

# Original sources
LIBSLZ_VERSION=1.2.0
LIBSLZ_URL=https://delta.lancs.ac.uk/mirrors/libslz/-/
       archive/v1.2.0/libslz-v$(LIBSLZ_VERSION).tar.gz
$(eval $(call fetch,libslz,$(LIBSLZ_URL)))

# Helpers
LIBSLZ_SUBDIR=libslz-v$(LIBSLZ_VERSION)
LIBSLZ_SRC=$(LIBSLZ_ORIGIN)/$(LIBSLZ_SUBDIR)/src

# Library includes
CINCLUDES-$(CONFIG_LIBSLZ) += -I$(LIBSLZ_SRC)

# Global flags
OPTIONS_CFLAGS += $(if $(SLZ_INC),-I$(SLZ_INC))
OPTIONS_LDFLAGS += $(if $(SLZ_LIB),-L$(SLZ_LIB)) -lslz

LIBSLZ_FLAGS += -fomit-frame-pointer \ 
   -fPIC -DPRECOMPUTE_TABLES
  #\ -Wno-implicit-fallthrough
  #\ -Wno-

LIBSLZ_CFLAGS-y += $(LIBSLZ_FLAGS)
LIBSLZ_CXXFLAGS-y += $(LIBSLZ_FLAGS)

# libslz code
LIBSLZ_SRCS-y += $(LIBSLZ_SRC)/slz.c
```
9.5  HAProxy Unikraft Config

9.5.1  HAProxy Config.uk

menuconfig LIBHAPROXY
bool "HAProxy - Reliable, High Performance TCP/HTTP
    Load Balancer"
default y

if LIBHAPROXY
config LIBHAPROXY_MAIN_FUNCTION
bool "Provide main function"
default n
endif

It is worth noting that we will be using the provided main function from
HAProxy and this will be set to “y” once pulled and configured.

9.5.2  HAProxy Makefile.uk

# Library registration
$(eval $(call addlib_s,libhaproxy,$(CONFIG_LIBHAPROXY) ))

# Original sources
LIBHAPROXY_BRANCH=2.3
LIBHAPROXY_VERSION=2.3.2
LIBHAPROXY_URL=http://www.haproxy.org/download/$(LIBHAPROXY_BRANCH)/src/haproxy-$(LIBHAPROXY_VERSION ).tar.gz

LIBHAPROXY_ZIPNAME=haproxy-2.3.2
LIBHAPROXY_PATCHDIR=$(LIBHAPROXY_BASE)/patches

$(eval $(call fetch,libhaproxy,$(LIBHAPROXY_URL))))
$(eval $(call patch,libhaproxy,$(LIBHAPROXY_PATCHDIR),
    $(LIBHAPROXY_ZIPNAME))))

# Helpers
LIBHAPROXY_PRIMARY_FOLDER=$(LIBHAPROXY_ORIGIN)/$(LIBHAPROXY_ZIPNAME)
LIBHAPROXY_SRC=$(LIBHAPROXY_PRIMARY_FOLDER)/src

# Sources
LIBHAPROXY_SRCS-$(LIBHAPROXY_MAIN_FUNCTION) += $(LIBHAPROXY_BASE)/main.c

# Flags
# Taking precedent from HAProxy’s Makefile (Ln 188)
LIBHAPROXY_CFLAGS -y += -Wno-unused-parameter \ 
  -Wno-address-of-packed-member \ 
  -Wno-sign-compare \ 
  -Wno-missing-field-initializers \ 
  -Wno-unused-label \ 
  -Wno-unused-value \ 
  -Wno-clobbered \ 
  -Wno-array-bounds \ 
  -Wno-cast-function-type

# Library includes
CINCLUDES-$(CONFIG_LIBHAPROXY) += -I$(LIBHAPROXY_PRIMARY_FOLDER)/include
LIBHAPROXY_CINCLUDES -y += -DCONFIG_HAPROXY_DATE ="2020/11/05"
LIBHAPROXY_CINCLUDES -y += -DCONFIG_HAPROXY_VERSION ="$(LIBHAPROXY_VERSION)"
LIBHAPROXY_CINCLUDES -y += -DUSE_OBSOLETE_LINKER
LIBHAPROXY_CINCLUDES -y += -DUSE_SLZ
LIBHAPROXY_CINCLUDES -y += -DUSE_POLL

# Global flags
LIBHAPROXY_HAPROXY_FLAGS -y += -Dhaproxy=haproxy_main

# HAProxy code
LIBHAPROXY_SRCS -y += $(LIBHAPROXY_SRC)/haproxy.c
  
For brevity, I have not included all the source files here however the full Makefile.uk including all of the outlined sources can be found on Github.62
9.5.3 kraft.yaml

specification: '0.4'

unikraft:
version: '0.4'
kconfig:
# Enable 9pfs filesystem support
- CONFIG_LIBUK9P=y
- CONFIG_LIB9PFS=y
- CONFIG_LIBDEVFS=y
- CONFIG_LIBDEVFS_AUTOMOUNT=y
- CONFIG_LIBVFSCORE_AUTOMOUNT_ROOTFS=y
- CONFIG_LIBVFSCORE_ROOTFS_9PFS=y
- CONFIG_LIBUKLIBPARAM=y

architectures:
x86_64: true

platforms:
kvm: true

libraries:
pthread-embedded: '0.4'
lwip: '0.4'
libslz: staging
libuuid: '0.4'
newlib: '0.4'
haproxy: staging

run:
volumes:
fs0:
driver: 9pfs
source: -**-**
networks:
et0:
pre_up: kraft-net --with-dnsmasq up
post_down: kraft-net --with-dnsmasq down
9.5.4 Makefile

UK_PARENT_DIR ?= $(PWD)/../..
UK_ROOT ?= $(UK_PARENT_DIR)/unikraft
UK_LIBS ?= $(UK_PARENT_DIR)/libs

LIBS := $(UK_LIBS)/haproxy:$(UK_LIBS)/pthreads-embedded :$(UK_LIBS)/lwip:$(UK_LIBS)/libslz:$(UK_LIBS)/
        libuuid:$(UK_LIBS)/newlib

all:
  @$(MAKE) -C $(UK_ROOT) A=$(PWD) L=$(LIBS)

$(MAKECMDGOALS):
  @$(MAKE) -C $(UK_ROOT) A=$(PWD) L=$(LIBS) $(MAKECMDGOALS)

9.6 Extra Screenshots

Figure 9.1: Unedited, full Image Size data for Docker infrastructure

Figure 9.2: HTOP Showing Processor Load
9.7 Original Proposal

Analysis of Unikernels for Load Balancing and Backend Service Deployment
Jonathon Cross, Lancaster University

9.7.1 Abstract

The primary aim of this project is to uncover the differences, if any, in using an identical infrastructure powered via Unikraft (NEC Laboratories Europe GmbH, n.d.), a Unikernel SDK. This project will explore Unikernels as a novel virtualisation paradigm and measure the performance of the Unikernel while investigating the security benefits a Unikernel claims to inherently boast. For this I will be utilising a container via Docker (Docker, Inc., n.d.), and a Virtual Machine running Ubuntu as comparisons to illustrate the capabilities and performance of the Unikernel. The reasoning behind my aim is due to the demand for mass infrastructure deployment, containers for example have been utilised widely and the Docker container platform alone has shown through a sample size of 10,000 companies, to have been adopted by a quarter of companies as of 2018 (Datadog, 2018). Therefore, due to this I have chosen Docker as the containerisation platform I will be using. Furthermore due to the year-on-year increasing prevalence in data breaches and ransomware attacks, as displayed by 4.1 Billion records being exposed in the first half of 2019 (Forbes, 2019) and examples such as WannaCry penetrating even vital health service architecture (Acronis, n.d.) I would like to explore Unikernels as a viable alternative to containers, boasting a smaller attack surface with similar if not better performance with service deployments. To achieve this aim, I will need to port the HAProxy library, and all its dependencies that do not already exist in the Unikraft ecosystem. HAProxy is significant due to its extremely popular and performant use as a load balancer (LB) which will be compared against an equivalent with Nginx (Nginx, Inc., n.d.).
9.7.2 Introduction

The increase in cloud computing, microservices, fog and edge computing has created a large demand for the provisioning and control of Virtual Machines (VMs) as applications are deployed and destroyed (just-in-time delivery-esque), an example of this would be an Amazon Web Services (AWS), Lambda which are ephemeral services, provisioned with the intent of running a service only once and run the program with any inputs it may have been given when invoked, where it then destroys itself once completed to optimise resources which can now be allocated elsewhere. A popular solution employed to facilitate this activity are containers (Google Cloud, n.d.) such as Docker, which is a lightweight alternative to a VM, containers operate by utilising OS-level namespaces in order to isolate the service which creates an independent of the environments that they will be run in. Additionally, in contrast to VMs, containers replace the need for a hypervisor that a VM utilises to then create operating system instances, to then use applications and their libraries – by replacing this layer and allowing the container to generate lightweight instances which benefits use-cases such as running a service once, for seconds or minutes to complete a job. These performance increases, with bootup latency for example looking at Qi Zhang (2018), A Comparative Study of Containers and Virtual Machines in Big Data Environment at Figure 3 we observe that at scale, containers were vastly superior at bootup times, whereby booting 512 Docker containers took 1.89 seconds per container for a total of 987 seconds which is a contrast to being able to boot 64 VMs in a similar time period. Therefore, this project involves providing an analysis for a lightweight solution in the form of a Unikernel in an attempt to display if this technology is a secure; optimisable alternative to a container via the boasts of Unikernels as seen in Pavliceck, Russell (2017), Unikernels: Beyond containers to the next generation of cloud under Chapter 1, “unikernels are small—very small; many come in at less than a megabyte in size”, “The attack surface of a unikernel machine is quite small, lacking the utilities that are often exploited by malicious hackers”. The major selling point is as Russell Pavliceck describes, unikernels are predisposed to being secure as opposed to popular solutions that currently exist which increases the overall security of a system with no added workload which may be put off by a mismanaged, over-worked engineering team. This presents great opportunity for cloud providers or those providing Internet of Things solutions to quickly implement and benefit from the security features of a Unikernel with no performance sacrifices compared to a container, while potentially even providing better performance than a container.

9.7.3 The Project

Aims and Objectives

The aim of this project is to analyse the performance of Unikernels in a backend stack including a load balancer and to measure this against containers using Docker of an identical setup. Therefore, there are a number of objectives that
must be completed in order to fulfil this aim.

- Port HAProxy which is a “High availability, load balancing and proxying solution for TCP and HTTP-based applications” (HAProxy, n.d.) to the Unikraft platform and the dependencies of HAProxy such as libslz (libslz, n.d.) which are not already present in Unikraft.

- Create a backend stack using the Unikraft platform using HAProxy as a load balancer, Nginx as a HTTP server

- Create an identical stack to the Unikraft backend, in Docker

- Create an identical stack to the Unikraft backend, in the Ubuntu Virtual Machine

- Perform stress tests and capture data

### 9.7.4 Methodology

Due to the very defined nature of this project with little need for change in specification I have chosen the Waterfall Methodology - the non-adaptive nature of the waterfall method will not be a barrier for this project and allows for an emphasis on structure and milestones for the duration of the project. The milestones for this project are discussed under “Programme of Work”.

### 9.7.5 Programme of Work

The project will be conducted between October 2020 and the 19th of March 2021 and as per the Waterfall Methodology will be structured as such:

- Analysis of HAProxy, its dependencies; how to port this to Unikraft. This will also involve researching how to set up Containers and VMs with the intended technology stack. This will take no longer than a week.

- Design of the system architecture and technology stack for HAProxy inside Unikraft, how this will interact with Nginx between load balancing and serving content and also running this identical setup on the container and VM instances. This will take another week to complete.

- Implementation & Testing, this will be the actual leg work of porting HAProxy to Unikraft, setting up the Unikernel system to handle load balancing and interacting with Nginx and the identical set-ups on the container and VM instances. This should take roughly 12-14 weeks in order to complete the porting and testing of the applications; ensure that they are ready to begin the performance analysis.

- Performance Testing & Analysis, this is the stage where testing will be carried out across the virtualisation setups to gain insights to the performance and differences, if any, of these solutions. This should take roughly 2 weeks.
References


[16] Adrian Mouat. 5 security concerns when using Docker. 2016. URL: https://www.oreilly.com/content/five-security-concerns-when-using-docker/.


[40] Red Hat Software. Newlib Homepage. URL: https://www.sourceware.org/newlib/.


[51] Docker Inc. Docker Compose Documentation. URL: https://docs.docker.com/compose/.


[56] AVR Libc. lwIP site. URL: https://www.nongnu.org/lwip/2_1_x/index.html.


