UiO : Department of Informatics
University of Oslo

A performance comparison of KVM, Docker and the IncludeOS Unikernel

A comparative study

Tamas Czipri

Master's Thesis Spring 2016

A performance comparison of KVM, Docker and the IncludeOS Unikernel

Tamas Czipri

May 23, 2016

Abstract

Unikernels are trending and becomming more popular in the current paradigm of cloud computing, there is a need to make systems smaller. Hosts of cloud platforms charge users by resources consumed and a small unikernel which is tailored for a specific purpose will save many resources. Big servers running clouds consume much power which can be saved by reducing the resources taken up on them.

A general purpose operating system will be its very nature be large and bloated. Containers are a solution to this, but unikernels are an alternative approach which offers more security as well. This thesis aims to test different systems and compare them to each other through experiments aiming to test their performance in scenarios related to networking.

Three experiments was developed in order to test the different systems which are IncludeOS, a unikernel operating system developed at Oslo University College, a Docker container and an Ubuntu server installation running in a virtual machine in terms of a web-service which can commonly be found in clouds and to test their network performance. The tests collect stats on the systems while giving them load in order to see how they perform. The experiments revealed that currently the container technology that is Docker has an edge over IncludeOS which is still under development while both have an advantage over a general purpose operating system running in a virtual machine.

Contents

I Introduction **1**

1 Introduction **3**
1.1 Problem statement 4
1.2 Thesis outline 5

2 Background **7**
2.1 Unikernels 7
2.1.1 Architecture 7
2.1.2 Security 7
2.1.3 In cloud 8
2.1.4 IncludeOS 8
2.2 Virtualization 9
2.2.1 Hypervisor and virtual machines 10
2.2.2 QEMU/KVM 10
2.3 Containers 11
2.3.1 Docker 11
2.4 General purpose operating systems 12
2.4.1 Security 12
2.4.2 Ubuntu 13
2.5 Related Works 13

II The project **15**

3 Planning the project **17**
3.1 Approach 17
3.1.1 Objectives 18
3.2 Design 18
3.2.1 Metrics 18
3.2.1.1 Underlying infrastructure 18
3.2.1.2 Virtual web-servers 20
3.2.2 Script to run the experiments 21
3.2.3 Underlying Infrastructure 22
3.3 Mimicing IncludeOS 23
3.4 Implementation and Testing 23
3.4.1 Tools for implementation 23
3.4.2 Setting up the environment 24

3.4.2.1 Hypervisors . . . 24
3.4.2.2 Virtual web-servers . . . 25
3.4.3 TCP server program . . . 25
3.4.4 UDP server/client programs . . . 26
3.4.5 Testing the environment . . . 26
3.4.6 Experimental setup . . . 27
3.4.7 Expected results . . . 28
3.5 Measurements, analysis and comparison . . . 28
3.5.1 Data extraction and script . . . 28
3.5.2 Experiments . . . 29
3.5.3 Analysis and comparison . . . 31

III Conclusion **33**

4 Results I - Desgin **35**
4.1 Mimicing IncludeOS . . . 35
4.2 Modification of the IncludeOS service . . . 36
4.3 Scripts . . . 37
4.3.1 QEMU . . . 38
4.3.2 Docker . . . 39
4.4 TCP & UDP programs . . . 40

5 Results II - Implementation and Experiments **43**
5.1 Setting up . . . 43
5.1.1 Hypervisors . . . 43
5.1.2 IncludeOS . . . 44
5.1.3 Docker . . . 44
5.1.4 Ubuntu . . . 45
5.1.5 Numactl . . . 46
5.2 Experiments . . . 46

6 Results III - Data & Analysis **49**
6.1 AMD . . . 49
6.1.1 IncludeOS . . . 49
6.1.1.1 Scenario 1 - TCP 500 req/s . . . 49
6.1.1.2 Scenario 2 - TCP 1000 req/s . . . 51
6.1.1.3 UDP . . . 52
6.1.2 Docker . . . 54
6.1.2.1 Scenario 1 - TCP 500 req/s . . . 54
6.1.2.2 Scenario 2 - TCP 1000 req/s . . . 56
6.1.2.3 UDP . . . 57
6.1.3 Ubuntu VM . . . 59
6.1.3.1 Scenario 1 - TCP 500 req/s . . . 59
6.1.3.2 Scenario 2 - TCP 1000 req/s . . . 60
6.1.3.3 UDP . . . 61
6.2 Intel . . . 62
6.2.1 IncludeOS . . . 62

6.2.1.1 Scenario: TCP 500 req/s 62
6.2.1.2 Scenario: TCP 1000 req/s 64
6.2.1.3 Scenario: UDP 65
6.2.2 Docker 66
6.2.2.1 Scenario: TCP 500 req/s 66
6.2.2.2 Scenario: TCP 1000 req/s 67
6.2.2.3 Scenario: UDP 68
6.2.3 Ubuntu 70
6.2.3.1 Scenario: TCP 500 req/s 70
6.2.3.2 Scenario: TCP 1000 req/s 71
6.2.3.3 Scenario: UDP 72
6.3 Comparison 74

7 Discussion **75**
7.1 Throughout the project 75
7.2 Proposed improvements 76
7.3 Alternative approaches 76

8 Conclusion **77**

Appendices **81**

A Scripts and programs **83**

List of Figures

2.1 Two Unikernels running DNS and a web-server on a bare-metal hypervisor . . . 8
2.2 An overview of IncludeOS' build system . . . 9
2.3 An illustration of a type 1 hypervisor . . . 11
2.4 An illustration of a type 2 hypervisor . . . 12
2.5 An illustration of containers running on a host OS . . . 13

3.1 Illustration of physical CPU utilized by virtual hosts. . . . 19
3.2 Illustration of physical RAM utilized by virtual hosts. . . . 20
3.3 TCP packet . . . 21
3.4 UDP packet . . . 21
3.5 TCP scenario 1 illustration . . . 30
3.6 TCP scenario 2 illustration . . . 31

4.1 High level illustration of collectors functions. . . . 38
4.2 A high level illustration of dockerCollectors functions. . . . 39

6.1 Average CPU utilization of IncludeOS . . . 50
6.2 CPU time of IncludeOS . . . 50
6.3 Showcasing the memory leak in the TCP service . . . 51
6.4 Average CPU utilization of IncludeOS . . . 51
6.5 CPU time of IncludeOS . . . 52
6.6 Response-time for UDP . . . 53
6.7 Average CPU utilization over the experiments . . . 53
6.8 CPU time for UDP . . . 54
6.9 Average CPU usage . . . 55
6.10 CPU time for Docker . . . 55
6.11 CPU usage for Docker . . . 56
6.12 CPU time for Docker . . . 57
6.13 Response-time for UDP . . . 57
6.14 CPU usage for Docker . . . 58
6.15 CPU time for Docker . . . 58
6.16 CPU usage Ubuntu VM . . . 59
6.17 CPU time Ubuntu VM . . . 60
6.18 CPU usage Ubuntu VM . . . 60
6.19 CPU time Ubuntu VM . . . 61
6.20 Response-time for UDP . . . 61
6.21 CPU usage Ubuntu VM . . . 62

6.22 CPU time Ubuntu VM 62
6.23 CPU usage IncludeOS 63
6.24 CPU time IncludeOS 63
6.25 CPU usage IncludeOS 64
6.26 CPU time IncludeOS 65
6.27 Response-time IncludeOS 65
6.28 CPU usage IncludeOS 66
6.29 CPU time IncludeOS 66
6.30 CPU usage Docker 67
6.31 CPU time Docker 67
6.32 CPU usage Docker 68
6.33 CPU time Docker 68
6.34 Response-time Docker 69
6.35 CPU usage Docker 69
6.36 CPU time Docker 70
6.37 CPU usage Ubuntu VM 70
6.38 CPU time Ubuntu VM 71
6.39 CPU usage Ubuntu VM 72
6.40 CPU time Ubuntu VM 72
6.41 Response-time Ubuntu VM 73
6.42 CPU usage Ubuntu VM 73
6.43 CPU time Ubuntu VM 74

List of Tables

5.1 Trident hypervisor 43
5.2 Intel hypervisor 43

Acknowledgements

I would love to thank the following people of the support that they have given me:

- **My parents, Zoltan and Imola Czipri** - For being an awesome mother and father who kept pushing me to continiue my education even though at times it was tempting to quit. Thank you for all your support and love.
- **My sister and her significant other, Nora Czipri and Svein Langeteig** - For letting me be their roommate and for being awesome people in general.
- **Hårek Haugerud** - For being my supervisor. Thank you for the guidance, support and helpful advice you have provided me with during the project work. A truly great person.
- **My teachers** - To all my teachers; Kyrre Begnum, Anis Yazidi, Paal Engelstad, thank you for the knowledge you all have provided us with.
- **To my classmates and my friends** - Thank you all for the time spent together, for the support and motivation you all have given me and for making these 2 years a fun and an overall great learning experience. My friends for all the support provided throughout my time in pursuing higher education.

Part I

Introduction

Chapter 1

Introduction

Virtualization is a technology that had its first inception in the 1970s, but has taken hold over the last decade where many services now are hosted in a cloud environment. The service is hosted on a virtual machine running on a hypervisor that has some operating system running said service. Before virtualization, running a service entailed one physical server having the service running alongside other services or functions. With the addition of virtualization a new abstraction of the hypervisor(server) hardware became available allowing virtual machines to run the desired services making it easier to modify, replace, migrate or repair a service.

A phenomenon that can be observed is that that more power is accumulated in smaller objects, this is especially true in terms of computer hardware. CPU chips are shrinking while outputting more performance, but for software it seems to go the opposite way. A modern operating system is rich in features which directly influence its size, in terms of commercial hosting in a *platform as a service* environment this can be a drawback as the metric for pricing is based on resources used. Furthermore, in the current paradigm where online services are in huge demand, specially in regards to uptime. Uptime is an important metric which directly affects the users, therefore having a system which efficiently can quickly take down and put up virtual machines is an important factor to maintain the services.

Using containers to deploy services is a solution that has gained significant momentum lately as they offer a simplistic way to create and host services for customers, for instance hosting a web server. As opposed to employing a full operating system which is very resource consuming, especially when considering that for instance a virtual machine instance running Ubuntu consumes an abundance of resources as each virtual machine runs a full operating system. This is important as there are many functions and drivers that are not necessarily needed to host a specific service. For reference an instantiation of an Ubuntu server instance requires at least one gigabyte of storage space [5]. Containers on the other hand virtualize the operating system which in turn only requires one instance of it, each container instance shares that kernel which results in being able to run multiple con-

tainers on one single operating system. As a result a container instance is in the megabytes spectrum (in comparison a virtual machine is often multiple gigabyte), inherently affecting performance and resource consumption on the underlying host.

Unikernerls which work as single purpose operating systems in a very stripped down form. Unikernels have not been as considered for deployment as there were support issues for an array of hardware that were used in real-world scenarios [11]. However, modern hypervisors solve this problem as the unikernel can use the device drivers that the hypervisor has while only needing to implement actual drivers for the virtual devices [10]. They are currently on the rise (specially after Docker acquired Unikernel Systems) in this service driven paradigm due to how lightweight they are and the popularity of both offering and using IaaS (infrastructure as a service). When considering Amazon's EC2 cloud (and possibly other IaaS services) where the price is relative to the resources consumed, unikernels can function with minimal resource requirements as they run inside a virtual machine. In an experiment with MirageOS where it was used as a DNS server, it outperformed BIND9 by 45% while being 200kB in size compared to BIND being larger than 400MB [11]. Taking into consideration that this experiment was performed almost three years ago there is reason to believe that improvements have been made since then, possibly on both sides. In addtion to small images an important feature present on unikernels is the security it offers [4].

In addition to prices for the user of an IaaS service, power consumption should also be considered as one of the main consumers of electricity is computers, with giant data centers consuming the most. The CEO of Digital Power Group estimated that the ICT ecosystem consumed about 10% of the worlds electricity in 2013 [14], which translated to the total electricity generated the same year by Germany and Japan combined [14]. This data indicates that employing an approach where resources that are not necessarily needed would decrease the electrical usage for both servers and the cooling of them [8].

1.1 Problem statement

As IncludeOS is still undergoing development as of writing this thesis, there will be need to implement an environment that guarantees fairness between the tests so that the same tests can be performed on all test subjects, IncludeOS being the limiting factor as it is not yet fully developed. As such the main focus will be to create a fair environment.

On both an Intel and an AMD based Ubuntu server compare the performance and resource utilization of IncludeOS, Ubuntu VM and Docker

- when providing a TCP based web-service.

- in terms of network throughput performance with UDP.

1.2 Thesis outline

This thesis is composed of 5 chapters:

Chapter 1 - Introduction - Defines the motivation for this thesis in addition to the problem statement.

Chapter 2 - Background - Defines the technologies, what they are and how they work which will be used in this thesis.

Chapter 3 - Approach - Defines what experiments will take place and how they are designed in addition to the purpose of the experiments.

Chapter 4 - Results I - Design - Displays the results of what was done in terms of design from the approach.

Chapter 5 - Results II - Implementation & Experiments - Displays how the project was implemented and how the experiments was conducted.

Chapter 6 - Data & Analysis - Shows the data that was gathered during the experiments and presents them through an analytical approach

Chapter 7 - Discussion - Discusses the progression of the project in terms of what was done, what could have been, constraints and such.

Chapter 8 - Conclusion - Through data gathered and analyzed, in accordance to the problem statement gives a conclusion to the problems defined.

Chapter 2

Background

2.1 Unikernels

Unikernels, also referred to as *libraryOS/libOS* or *modular operating system* have been in a major spotlight lately as support for the increase, earlier they have not been as considered for deployment as there were support issues for an array of hardware that were used in real-world scenarios [11]. A unikernel is a specialized operating systems written in high level code, like C++(IncludeOS) or OCAML(MirageOS) that are compiled from the application code. The drivers and operating system functions that are included in the compilation are only the ones needed to run the application, making them extremely lightweight and very fast as there are no context switches. This gives unikernels a significant edge in terms of size, scalability, migration and deployment. In addition, as they are only designed to run the one service which often is hard coded into the image that runs the operating system, which results in a small code base, in addition to limited possibilities as to what one can do if one were to be able to get in as it runs in read-only mode making unikernels very hard to compromise for an attacker. Furthermore this also accomplishes isolation at an application level due to the ability to simply creating and booting another virtual machine instance for each application.

2.1.1 Architecture

The way which unikernel operating systems are built is that they run in the systems memory space which leave a very small resource footprint on the system. They are in their very essence a small code base running directly on a hypervisor in a virtual machine instance as illustrated by figure 2.1. The unikernel itself are libraries and the operating system objects that are linked with a linker which creates the image file that will be used by the spawning virtual machine.

2.1.2 Security

In terms of security the unikernels stand very strong due to the miniscule attack surface they have as a result of being stripped of functionality that

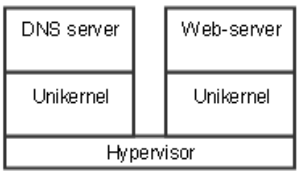

Figure 2.1: Two Unikernels running DNS and a web-server on a bare-metal hypervisor

is not needed or used [10]. Furthermore, when an unikernel is launched in a virtual machine instance it serves one purpose and one purpose alone; meaning that once the desired service is compiled and ran there is nothing more to it. Should a scenario arise where an attacker manages to compromise the virtual machine, the attacker's scope of disruption would be immensely limited.

2.1.3 In cloud

With cloud or IaaS being the current paradigm in hosting online services, unikernels are absolutely relevant as they are designed for cloud environments. Considering hosting services like Amazon's Elastic Compute Cloud(EC2), Google Compute Engine, Windows Azure, that is just to name a few [16] where a customer pays for the allocated resources that are consumed. That means having a small resource footprint; mainly in the kilobytes spectrum is a colossal advantage over an operating system spanning multiple gigabyte where only a fraction of what is installed is needed on a case to case basis(as they are general purpose). Specially when considering that unikernels are specialized for the task which they are to perform.

2.1.4 IncludeOS

IncludeOS is a unikernel operating system currently being developed at Oslo & Akershus university college and is written in C++ for the purpose of operating in a cloud environment. The operating system is compiled so that when writing a service that it will host, the user starts with; `#<include> os`. This will then wrap the operating system around the code for the service and run it in a virtual machine, IncludeOS does not have a *main* function, there is however a *Service* class in which the user will implement the desired service. `Service::start` will be called when the operating system finishes initialization [3]. When it builds the build system will link the service with only the required operating system objects which creates a binary file after which a boot-loader is attached and then it gets recompiled into an image which will run [1] as explained on their wiki page and illustrated[1] by figure 2.2.

As this operating system is currently still under development, as of today

[1]The illustration is made based on the figure found on https://github.com/hioa-cs/IncludeOS/wiki

14.03.16 it is in version 0.7.0 - which dictates that new functionality is being developed and added over time and as such also means that it is not yet fully complete.

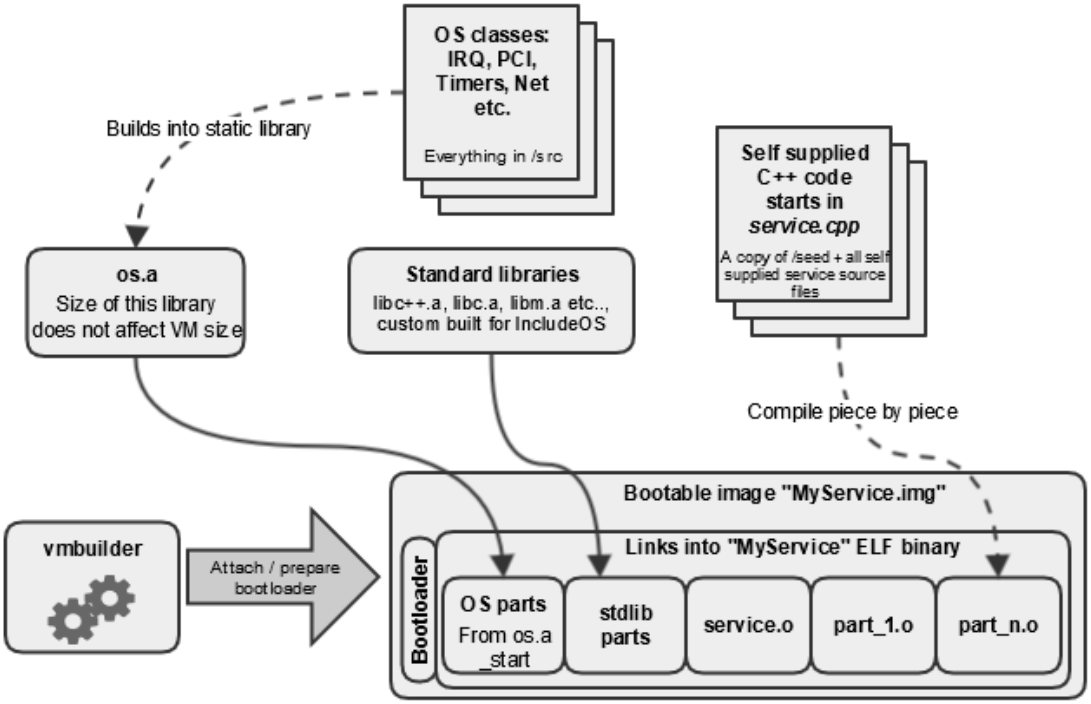

Figure 2.2: An overview of IncludeOS' build system

2.2 Virtualization

Virtualization provides the ability to split up resources on a physical machine and reallocate them in virtual machines which can be used for a plethora of tasks as they are each a functional machine. Furthermore, virtualization simplifies logistical tasks with little to no downtime [9]. In terms of infrastructure virtualization can reduce costs as one powerful hypervisor is able to run multiple instances of machines, centralize and simplify large scale management and also help the environment [9].

There are three techniques for virtualization;

- **Paravirtualization**
  An API is provided by the hypervisor for the guest operating system to use when operating system calls are made. This method does require a modified version of the guest operating system to be able to make the API calls.
- **Full virtualization**
  When operating system calls are made from the guest operating

system these calls will get trapped and emulated in the case where the calls are not virtualizable, the technique for catching the un-virtualizable instructions is called binary traslation which incurs a large performance overhead.

- **Hardware-assisted virtualization**
  Arrived with Intel VT and AMD V CPUs, this type allows the guest operating system to function as it normally would while the hypervisor virtualizes its hardware which then the virtual machine can access and use in an isolated fashion.

2.2.1 Hypervisor and virtual machines

A hypervisor or virtual machine monitor allows running one or more virtual machine on a single host, while each operating system governing the virtual machines sees the hardware on the hypervisor as solely used by them. Virtualization takes place at the hardware level. Hypervisors are either hosted or bare metal:

- **Type 1 - Bare metal/native**
  In a type 1 hyperisor the hypervisor itself runs directly on top of the hosts hardware with no operating system it. The hypervisor then allows the running and management of virtual machines which it hosts. A type 1 hypervisor is managed by console as there is no native operating system present on the host system. Figure 2.3 illustrates how a type 1 hypervisor works.

- **Type 2 - Hosted**
  A type 2 hypervisor runs within an operating system with the hypervisor being a layer in the operating system which runs directly on the hardware. This is achieved by the hypervisor managing and timing calls for hardware and IO (CPU, RAM, disks and network). Figure 2.4 illustrates how a type 2 hypervisor works.

2.2.2 QEMU/KVM

QEMU is a software solution in terms of virtualization which gives a host the ability to emulate and virtualize hardware (not the same as hardware-assisted virtualization). QEMU has different modes in which it can work, together with the way QEMU emulates a CPU, which is through dynamic binary-translation it allows QEMU to host a variety of guest operating system that does not need to be modified. In addition, QEMU is able to emulate a plethora of hardware, such as network cards, cd-roms and hard drives to mention some and comes with built-in features for SMP, SMB and VNC.

KVM is a version of QEMU which means *Kernel Virtual Machine*, in relation to QEMU when enabled it allows guests to run on the hypervisor as processes. When KVM is enabled the guest machines effectively run on

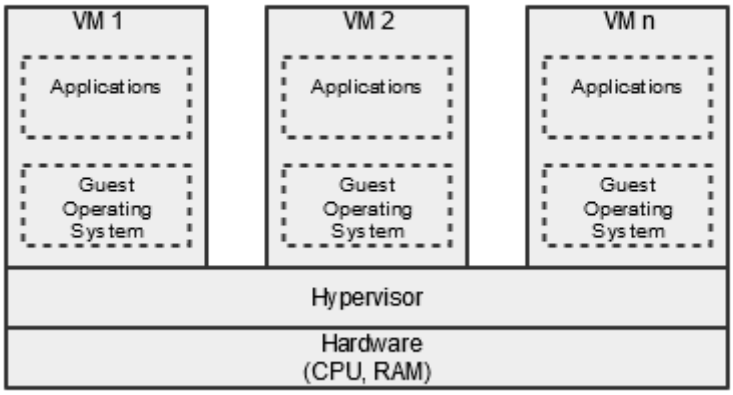

Figure 2.3: An illustration of a type 1 hypervisor

bare-metal which provides a performance gain, as such the guests run at near-native speeds, should that not work it will emulate software-only as a fallback.

2.3 Containers

Container-based virtualization is an approach where the host kernel is shared between then software-containers and is used to run them. The main difference is how they do the virtualization [12], hypversiors virtualize at hardware level where as containers at operating system level. They act like sand-boxed operating systems, a host operating system runs in the bottom layer from which the containers share the kernel as shown on figure 2.5. The shared parts operate in read-only mode. The container engine runs on top of the host operating system where it manages the containers themselves. A big advantage with containers is that they a capable of running platform independently(assuming that the platform can run it) due to the fact that they share the host kernel which makes tasks like migration from one server to another.

2.3.1 Docker

Written in a language developed by Google with a syntax based on C [13], Docker is an operating system virtualization tool based on LXC that acts as a wrapper around applications with a complete filesystem. The filesystem has all the required dependencies [7] for the application to run platform independently, if Docker can run on the platform, then so can the application packaged in Docker. The way Docker achieves this is through a unified API that manages kernel-level technologies [15], the filesystem that Docker uses is called *Advanced Multi-Layered Unification Filesystem* (AuFS). An advantage that AuFS comes with the ability to save storage and memory consumption as there is only one base image required which then can be used

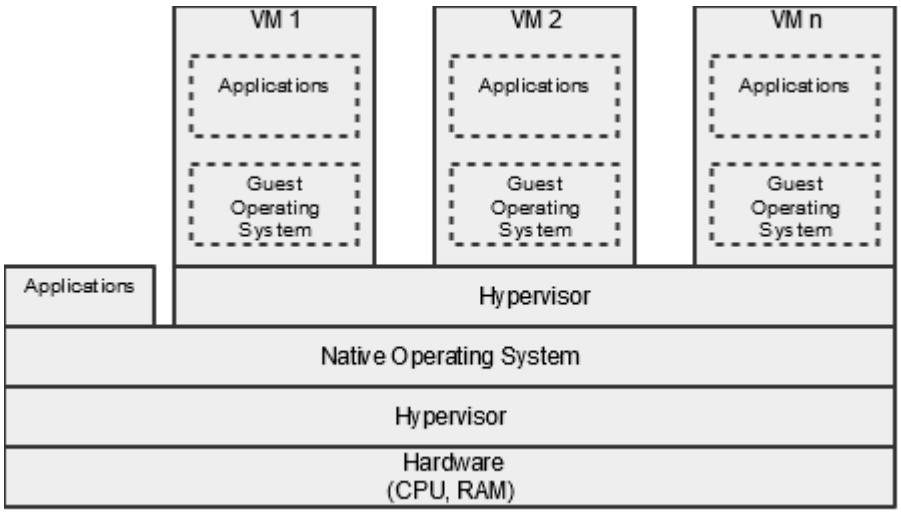

Figure 2.4: An illustration of a type 2 hypervisor

as a basis to build multiple Docker containers.

The security within Docker resembles that of LXC containers, whenever a Docker container is started there will be created a set of namespaces and control groups which is relevant for that container. The namespaces hide anything that runs outside of the specific container for the process running inside it. The control groups allow for controlling resource allocation and limitations for the container, the way control groups work is they make sure that the container has all the CPU power, RAM allocation and disk I/O it needs, however they implement an additional feature; they prevent the possiblity of a container exhausting either of those resources on the host system and acts as a defense for denial-of-service attacks.

2.4 General purpose operating systems

A general purpose operating is a regular operating system like Windows or Linux, these operating systems are full of features to serve a general purpose, i.e jack of all trades. As such an installation of either consumes a substantial amount of space in return for features one does not necessarily need. In addition, these services are running alongside the kernel leaving a larger resource footprint in terms of CPU and RAM utilization.

2.4.1 Security

Most general purpose operating systems come with built-in features for security. The security implementations in an operating system is based on a concept called *CIA*; Confidentiality, integrity and availability and means that unauthorized entities should not be able to do anything an administrator of a system does not want such entities to do. As a result this

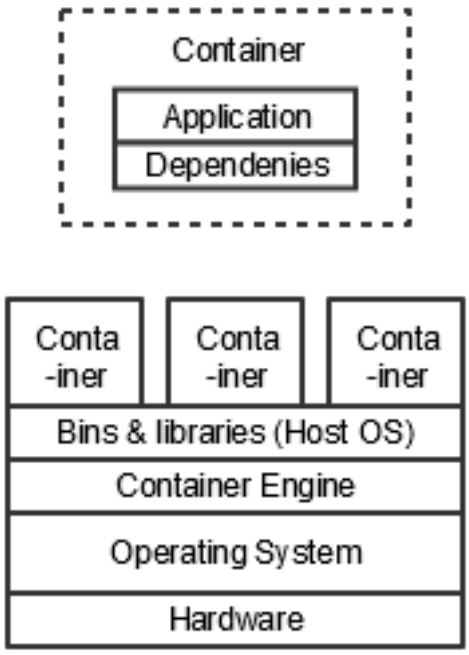

Figure 2.5: An illustration of containers running on a host OS

is implemented in the operating system itself, adding to size and resource consumption.

2.4.2 Ubuntu

The general purpose operating system which will be used in the experiments is the Ubuntu distribution of Linux, version 14.04 (Trusty Tahr) server edition. The justification for selecting this particular operating system over others is that it is widely used as server host operating system and is free.

2.5 Related Works

1. **An Updated Performance Comparison of Virtual Machines and Linux Containers** by Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio IBM Research, Austin, TX
   Here the authors test the performance of virtual machines against Linux containers, this paper was published by the research division of IBM.

Part II

The project

Chapter 3

Planning the project

3.1 Approach

This chapter will focus on the methodology and the required actions in order to ensure that there is an answer to the problem statement. In order to develop a testing framework which can be implemented to mimic the IncludeOS web-service and ensure a fair test between the different software solutions, it is important to look at the underlying technology of the system in order to determine limiting factor. The approach will consist of multiple sub-sections which discuss the key features of the development of such a framework. Furthermore this chapter will describe what technologies are selected, how the technologies are combined with design and utilized throughout the experiments.

The problem statement which was defined in section 1.1:
On both an Intel and an AMD based Ubuntu server compare the performance and resource utilization of IncludeOS, Ubuntu VM and Docker

- when providing a TCP based web-service.
- in terms of network throughput performance with UDP.

As such the following outline is an overview of what this chapter will look at in detail:

- Look at the objectives which need to be done to achieve an answer in relation to the problem statement.
- Looking at how IncludeOS implements its web-server service.
- The design of the models which IncludeOS will be tested against.
- Implementation and experiments.
- Analysis and comparison.

3.1.1 Objectives

As described in the problem statement and the introduction chapter, this thesis will attempt to compare how the IncludeOS operating system performs in relation to other, more traditional software solutions for web hosting in terms of performance. The experiment framework will be based on what IncludeOS implements, which is a socket-server written in C++, the framework itself will be built in manner which will achieve simplicity in terms of performing the experiments. As such, this thesis will focus on the quantitative data which is gathered through the experiments. The experiments require multiple software solutions, in addition an infrastructure on which they can run. One of the goals of this thesis is to acquire, look at and analyze data for IncludeOS to determine its efficiency as a web-server in terms of hardware and resource consumption to determine how it would perform in for instance a cloud environment. Furthermore, how consuming, or the lack thereof as it is directly related to green computing along with costs and efficiency as mentioned in chapter 1.

To find an answer to the questions proposed by the problem statement the work will be split into three phases which are the following:

1. Design.
2. Implementation and testing.
3. Analysis and comparison of the results.

3.2 Design

The design phase is the first part of what must be accomplished in this project and will be divided into the tasks outlined below:

1. Define which metrics will be monitored in order to acquire the data.
2. Actions needed to be performed by an automated script for data collection.
3. The underlying infrastructure.
4. Mimicing IncludeOS' web-server in Ubuntu and Docker.

3.2.1 Metrics

3.2.1.1 Underlying infrastructure

The first objective is to consider what hardware in the hosting hypervisor will be under load when a virtual machine functioning as a web-server is hosted on it. Further, the time which all three of the web-server use in order

to answer a requesting client. When hosting a virtual machine there is a lot of abstraction added to hardware so that virtual machines can utilize them without having kernel mode access.

As such, there is additional load on the CPU which gets abstracted into virtual cores that enables the virtual hosts to utilize them as if they had their own dedicated CPU. Therefore, utilization of CPU is highly relevant in this case. In addition to the CPU, RAM also has an additional abstraction layer for the virtual machines to be able to utilize them illustrated in figure 3.1 and 3.2. This abstraction, like the CPU allows the virtual machine to believe it has its own memory space and this is done through additional layers of paging, called shadow-pages. RAM utilization of the underlying hypervisor therefore is a good metric to help determine the overall performance cost of the hosted system. Further, deploying the implemented experiments from an outside host would in addition require NAT rules of some kind to reroute traffic to the virtual machine being tested which would add additional consumption of the mentioned hardware resources.

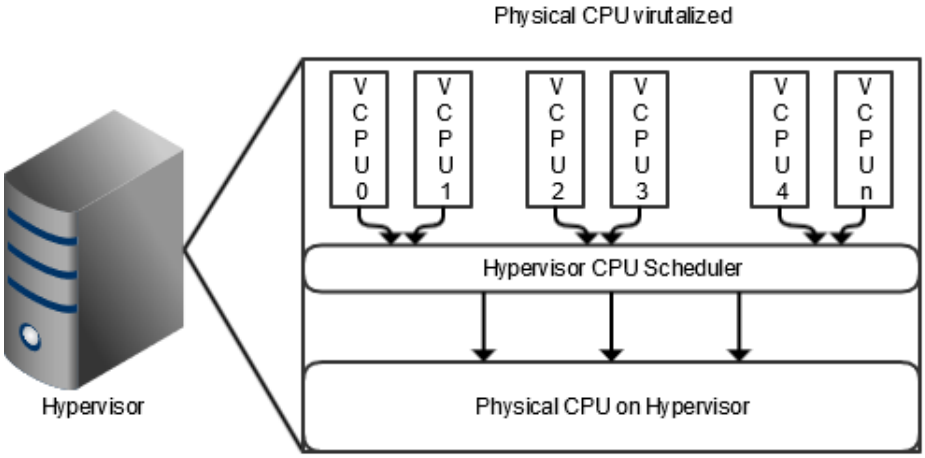

Figure 3.1: Illustration of physical CPU utilized by virtual hosts.

It should however be mentioned that during the experiments these additional overheads will not have any drastic impact on the system as a whole and it can be assumed with certainty that this overhead will not interfere with the experiments intended to run on the hypervisors as they are both very powerful. As such, the additional usage of the hardware on the hypervisor should not affect virtual machine performance as there will not be an instance of the load necessary to saturate the power which the hardware provides. However, the values regarding real-time usage

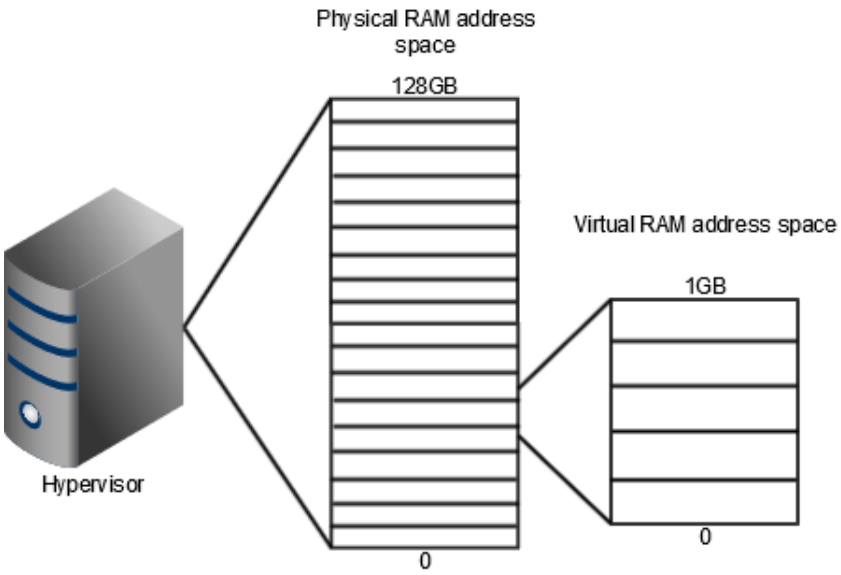

Figure 3.2: Illustration of physical RAM utilized by virtual hosts.

during the experiments will be monitored to give a vizualization of the actual resource footprint each of the operating systems set.

3.2.1.2 Virtual web-servers

There are two important metrics that most users are concerned with when it comes to web-servers, those are how fast it takes something to happen when a request is made and the other being that there are no random interruptions when using a web-service. Therefore, on a web-server in a virtual machine running on a hypervisor there are mainly three metrics in this thesis that are relevant; that is the response time of the web-server. In each of the three web-servers the goal is to see which of them performs better in this regard. The second metric is stability; how IncludeOS, Ubuntu and Docker performs under heavier loads for extended periods of time while still maintaining the ability to serve a request within a reasonable time frame. The third metric which is for UDP, the speed at which the packets are received and returned to the sending client.

In addition to exploring the performance of the web-servers in TCP, UDP performance will also be looked at to give an idea on pure performance in terms of throughput without the overhead that the TCP protocol brings. This overhead is caused by the architecture of the TCP frame and how the protocol works, TCP is sequenced and acknowledged meaning that there is a particular order to how packets should be transmitted and received. Then each of these packets are acknowledged by the receiving system in order for the transmitting server to know when

it can send the next packet. In short, this process of maintaining order and assurance incurs a cost in terms of time/overhead. The UDP protocol has no such mechanisms and as a result should be faster. Figures 3.3 and 3.4 illustrates the differences in the two packets, and shows how much more information a TCP packet contains which has to be processed upon arrival.

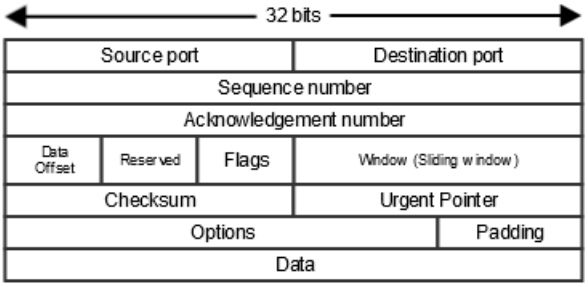

Figure 3.3: TCP packet

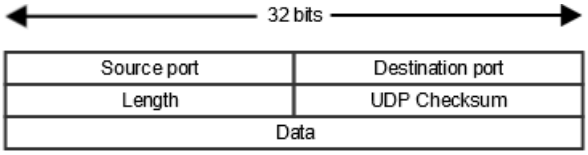

Figure 3.4: UDP packet

3.2.2 Script to run the experiments

To achieve the most efficient way of executing the experiments, the experiments themselves need to be automated. As such, the script will need to:

- Initialize the virtual machine.
- Make sure the virtual machine is running and responsive.
- Run the experiments.
- Log data.

This will outline the basics which needs to be in a script such as this, and section 3.4 will further explain in depth how it works. The list above this paragraph already explains the sequence of which the actions performed by the script needs to happen.

First, a virtual machine needs to be initialized. This can be either IncludeOS, Docker or Ubuntu, an argument will be taken in the form of `./script.sh includeos` which would launch IncludeOS in QEMU. In

addition, a second argument will be supplied which can be either `tcp` or `udp` to determine what benchmark should be executed. The second step after bringing up the virtual machine is to ensure that it is connected to the virtual network and can communicate with the hypervisor. This will be achieved by transmitting ICMP requests to the virtual machine until it responds to the request. Taking into account a scenario where the virtual machine should not be able to connect or something goes awry during boot there needs to be a constraint that acts as a safeguard so that the program does not get stuck trying to ping infinitely, which will be based on time.

The third step will be to execute the actual experiments which will, based on an argument decide the correct test to run. For the Ubuntu virtual machine, it will also need to `ssh` into to the host in order to start the socket server program. In the case of testing TCP, it would run for example `httperf` a defined amount of iterations and then collect the results from the benchmark along with CPU and RAM data from the hypervisor in a human-readable text-file which then can be used to generate visualized data.

3.2.3 Underlying Infrastructure

The physical hardware where the virtual machines and the experiments take place, this is two hypervisors with different CPU platforms to see if that has any bearing on the performance results of IncludeOS. The two hypervisors are running AMD and Intel platforms. The experiments will be conducted on both of the hypervisors, the deployment of the virtual machines will be executed using the same images to ensure that they run the exact same instance of the virtual machines.

Both the AMD and the Intel hypervisors are research servers at Oslo & Akershus University College and are mainly employed for the purpose of research projects, the Intel hypervisor is newer than the AMD meaning that the hardware on-board it has more power which will be looked at more in-depth in section 3.4, however it should again be noted that the experiments will not be able to saturate the servers of resources which means that the main difference shown, if any, is due to different platforms.

In order to get the least amount of noise, the experiments will be executed on the same hypervisor that hosts the virtual machines and not from a third outside system. This decision is due to doing it that way would not add any meaningful data for this thesis as it would only incur additional overhead for the responses, as most of the time spent receiving a resposone would be consumed by waiting for it to travel the ethernet link while only a fraction of the time would be representative for the actual guest systems on the hypervisor.

3.3 Mimicing IncludeOS

The first step is to find out how IncludeOS implements its web-server service. As mentioned earlier, IncludeOS comes with a file called `services.cpp`, this is a C++ file which is the service that will be compiled into the operating system at run time. The web-server being one of the first services ever written for IncludeOS and then later restructured and updated implements a C++ socket server which listens to requests at port 80 and serves the default HTML page that comes with it.

For the experiments regarding network throughput, the UDP experiments. In the samme manner as TCP, IncludeOS has a UDP service which makes the virtual machine act as a UDP server. This functionality also exists as a standalone C program which will be deployed and run on Ubuntu and Docker when the experiments are run.

As IncludeOS' transfer currently is limited to 1 ethernet frame (about 1.5KB) it will be rewritten to produce 1000 copies of the character 'A'. 1 character is equivalent to 1 byte therefore 1000 copies of 'A' equates to 1KB worth of data along with the HTTP header which accounts for the last 512 bytes. To be able to make a fair comparison, such a web-server will have to be implemented to run on Ubuntu and Docker as well. The implementation will be done in C++ and serve exactly the same as IncludeOS, including the same HTTP header used in it.

3.4 Implementation and Testing

The implementation and experimentation phase is the second and the practical part of the tasks which need to be completed as defined by the design in section 3.2. This phase consist of a few main objectives which are outlined, but not limited to:

1. Selection of tools to implement the design.
2. Setting up the environment on the infrastructure.
3. Ensuring that the setup functions on both hypervisors.
4. Experimental setup.

3.4.1 Tools for implementation

In order to achieve the environment which is explained in the design and implementation sections and the goal of this thesis which is answering the problem statement, a multitude of tools has to be used and combined. It is important to review the options of different tools to select the appropriate software and technologies based on their functionality and limitations.

First, what needs to be looked at is exactly what needs to be accomplished. Looking at section 3.2 and chapter 2 which showcases

the possible tools and technologies in detail, what will be needed is a virtualization technology to host the virtual machines in order to run the different operating systems which will be compared. A program language that will allow mimicing IncludeOS' TCP and UDP services as closely as possible, a program to be able which will allow the tracking of the real-time performance and resource utilization of the virtual machines and container in terms of CPU and RAM. A scripting language which will serve two purposes, both of which will enable automation. The first script will be used to set up and run the virtual machines, after which it will execute a benchmark to test the ability of the operating systems TCP in terms of average reply rate under load and then save the data extracted from the experiment to a file. The second script will be tasked with gathering data from the experiments, i.e extract the relevant data from the files the first script creates and collect them in order to easily use them. Lastly, a tool for benchmarking the web-servers ability to serve data.

For that purpose the following tools has been chosen:

- KVM/QEMU - As a virtualization platform.
- C/C++ - As the language for implementing IncludeOS' web-server on other machines and to test UDP performance.
- Bash - As the scripting language for automating experiments and extracting data.
- Pidstat - As monitoring tool for CPU and RAM on the hypervisors.
- Httperf/Autobench - As a tool for benchmarking TCP performance.

3.4.2 Setting up the environment

3.4.2.1 Hypervisors

This part will explain the set up and configuration the environment to run the experiments. As there are three systems that are to be tested against each other it is important to ensure that each of them can be tested separately in a fashion which guarantees that neither system interferes with the other. In order to achieve this isolation, how the systems work and run is important to understand. The IncludeOS and Ubuntu virtual hosts will be hosted through QEMU/KVM and will not interfere with each other as they are run due to the process no longer existing upon closing the virtual machine. Docker uses a daemon to run and manage the containers which inherently also consumes some resources and will have to be stopped manually after finishing the experiments with Docker.

As mentioned the experiments will take place on two hypervisors, where one system runs on the Intel platform and the other on AMD. In order to ensure that the systems being tested are equivalent on both hypervisors, they will first be set up on one of them and each system will be

configured according to specifications in terms of hardware provisioning, services and programs. When the systems are tested and proven to work as expected they will then be copied and moved over to the other hypervisor, and will be rerun there as both hypervisors have the close to the same services and programs installed, services unnecessary for the experiments that are running will be stopped prior to running the experiments. Both hypervisors are running equivalent operating systems, while being close to each other in terms of performance.

3.4.2.2 Virtual web-servers

Once the hypervisors are ready to run the virtual machines, which is when the environment on them is set up for that, the point where nothing that does not need to run is stopped. When completed, the virtual machines can be set up and installed in order to ensure that they run as they are supposed to, it is important to see that all systems function as expected in order to perform consistent experiments.

The first step for IncludeOS is having to clone it from *Git* and then installing the operating system on the hypervisor. When installed, a service file will be modified according to the specifications of this thesis to act like the web-server explained in section 3.2 and illustrated more in-depth in sections 4 and 5. The service then needs to be compiled to an image which can be booted by using QEMU.

The Ubuntu virtual host requires fetching a bootable image for the 14.04 LTS server version. The hardware available to the virtual machine will be restricted through QEMU in order for Ubuntu to have the same hardware specifications as the IncludeOS virtual machine. When the host is running the scripts that emulate the servers will need to be copied over to it, which will be compiled and ran on the host.

The installation and setup of Docker are done by the Linux command line through APT, when Docker is installed on the host system there are some approaches that can be taken in terms of setting up the OS images. For many operating systems Docker has a repository to chose from, also here it will run 14.04 LTS server edition. When building the images from which Docker launches its containers a *Dockerfile* will be utilized to define what the container will contain in terms of what comes installed on it, and what local files (being the server applications here) are copied over to it and compiled when it launches.

3.4.3 TCP server program

Two of the three virtual hosts, Docker and Ubuntu will be deployed using an identical solution in terms of handling TCP. In order to ensure that there is a fair comparison between IncludeOS, Docker and Ubuntu at a virtualization and operating system level as mentioned in section 3.2, a program that based on and mimics the way IncludeOS is set up to handle

TCP. To achieve this, a C++ program will be developed which will work as a TCP socket-server that replies to requests received while running.

3.4.4 UDP server/client programs

In addition to testing the difference in how the operating systems and virtualizations perform when TCP is tested in terms of reply rate and underlying hardware utilization, UDP performance will be tested as well. As UDP is a faster protocol than TCP in the sense that it has no overhead for initializing a connection, sorting packets, and waiting for acknowledgements of received packets and in case of loss retransmitting them it will be a much better test for determining pure throughput of the network which will help determine the performance of the network.

There will be a program to act as a UDP server on Docker and Ubuntu, IncludeOS has a UDP service already written which will be utilized which are close in terms of equivalence. In addition, there will also be C programs that will be used which will test the UDP performance of the hosts, the program will send a specific amount of UDP packets to the server which iterate through the packets and send a new packet containing 1000 instances of the character 'A' which is about 1KB back to the client which will stop when the last packet is received, this will be timed.

3.4.5 Testing the environment

After setting up the environment in which the experiments will take place it is important that everything works as expected. The communication between the hypervisor and the guests, the programs running on the guests and the benchmarking tools against them.

A simplistic way to test that everything works as expected is by running a few initial experiments so that any issues that might occur can be preemptively caught and fixed. As such for each of the three guest operating systems a simple run, alive and benchmark is executed. First run, booting up the virtual machine on the hypervisor and monitoring that it does run, in addition, that the process exists. To ensure that the virtual machine is able to communicate is to check if it responds to ping, if it does then the virtual bridge works as expected.

The tests that will be run to ensure functionality can be split into:

- *Test 1 - The deploying of the virtual instances*
  Testing to see that everything deploys and runs as expected by booting each individual guest host and confirming that it boots and runs.

- *Test 2 - Alive status of the virtual instances*
  Testing to see that everything communicates as expected by issuing ICMP requests to the guest hosts on the corresponding virtual bridge and confirming a response from the system. Then issuing a cURL

to confirm the web-server functionality and that the virtual bridges work.

- *Test 3 - Experiment test on the virtual instances*
  Setting up the TCP and UDP server applications on Docker and Ubuntu and running a short benchmark on them to ensure that it works as expected. For IncludeOS the equivalent is to run the OS and perform the same benchmark towards its service that is running.

3.4.6 Experimental setup

The experiments in this project regarding TCP will be split into two scenarios:

- Continuous load over time.
- Increasing load over time.

This will attempt to measure how all systems respond to different traffic patterns, a pattern that creates an equal amount of load over time will establish the stability of the system as the load in this case will end up being lower on average than the peak of the increasing pattern.

For the continuous load pattern *httperf* will be used as it can generate a steady flow of traffic over time to the virtual client, while it will be measured how the hardware is utilized during the time the load is active on the virtual clients.

For the increasing load pattern *autobench* will be used in place of just *httperf* alone as it can generate traffic that increases over time which should put a different amount of resource requirements on the virtual clients until the point where they can no longer reliably and in a stable manner respond to the the requests being sent.

In regards to UDP, IncludeOS has a UDP service which will be utilized after which there are an as-close-as-possible replica coded in C that will be running on Docker and Ubuntu. The tests towards the UDP servers will be conducted with programs that will send a specified amount of packets, which at the server for each packet received will send a 1KB packet back to the client together with the time it took before the last packet arrived back at the client as it is mainly speed that UDP offers in comparison to TCP. For the program that is sending packets to the UDP server there needs to be specified the amount of packets that will be sent. The amount of packets that will be sent and received is 100000 or in other words, 100MB in total from the server to the client. Listing 3.1 shows how the the program will be executed and what the response looks like.

Listing 3.1: Input UDP send

```
1 $ time ./sendUDP.sh HOSTNAME AMOUNT_OF_PACKETS
```

Listing 3.2: Output UDP send

```
1 $ Received LAST, AMOUNT_OF_PACKETS packets received
2
3 real   XmY.Zs
4 user   Xm,Y.Zs
5 sys    Xm,Y.Zs
```

3.4.7 Expected results

When comparing the results after each experiment on both the AMD and the Intel platforms, considering previous work done on the research of container efficiency [6] the expectation is that Docker most likely will come out ahead in terms of raw performance as its speeds are near native. Further, virtualized Ubuntu is expected to perform the least efficient in comparison as it is large in terms of what is on it and what runs in order to maintain it. Docker also runs Ubuntu, however it is not installed and running as a standalone operating system, merely a virtualized kernel. IncludeOS is expected to fall between Docker and virtualized Ubuntu as it is a virtualized operating system, however stripped of all unnecessary functionality, e.g. the only driver installed on the operating system is virtio.

For the responses in terms TCP and UDP the same performance is expected, both Docker and Ubuntu have an advantage over IncludeOS. That is how long they have existed, the developers have had more time to perfect how the networking within the kernel works. However, research does suggest [2] that IncludeOS will be the middleground here as well at least in terms of UDP.

3.5 Measurements, analysis and comparison

The last stage of the project, the data that has been gathered throughout the project will be looked at in depth, analyzed and compared between the virtual hosts. The data that has been gathered in realtime during the experiments will be extracted and plotted into charts/graphs for an illustrated comparison and analysis. The tasks that will be done in this phase of the project are described below:

- Building the script for data extraction.
- Executing the experiments.
- Analyzing the the extracted data.
- Compare the data from the three systems and two platforms.

3.5.1 Data extraction and script

The output recorded by the script which runs the experiments that are saved in text files will need to extracted and arranged in a logical fashion as

the measurement script saves the raw output of the programs that track the metrics. This will make the data easier to read and input into programs that will be used to plot the data as opposed to doing it manually. It is important to have a good overview of the gathered data in order to be able to properly analyze it, specially when there are multiple data-sets. Achieving a solid basis which analysis and conclusion is built upon relies on having logically structured and easily read data.

The intention is that the script will take a file as a parameter which is created by pidstat while it is monitoring the CPU and RAM of the virtual machines during the experiments and in essence just rearrange the data in them as they are not logically structured.

Listing 3.3: Proposed input

```
1 $ ./extract.sh includeos22_cpu_ram.txt
```

Listing 3.4: Proposed output

```
1 CPU      RAM     USED/MAX    USER     SYSTEM
2 17%      4%      51M/1024    X sec     Y sec
3 24%      9%      92M/1024    X sec     Y sec
4 33%      16%     153M/1024   X sec     Y sec
```

The timing in pidstat can be set manually and will be set to a low interval to get a precise measurement of the utilization. Considering the proposed output 3.4 it is easy to see the amount of usage in terms of both CPU and RAM, furthermore how much time the CPU spent execution instructions in user mode on the hypervisor for the virtual machine, how much time was spent on executing instructions in kernel mode and how much time was spent executing instructions inside the virtual machine.

Httperf also generates data about the virtual machines in terms of length, connections, requests sent, replies recevied, reply rate and errors in addition to more statistics. Httperf does however generate reports for each run that is recorded which is important to look at to get a good overview of the performance of the actual web-servers in terms of TCP performance.

When the UDP experiments are ran they will be monitored with the systems `time` command which when run before a command or script will record statistics of the program when it finishes executing. The most important feature of the time command is that it returns the time it took for the program to finish which will identify how long the UDP experiments took until the last packet of the response is received in order to show how efficient the web-server are in terms of pure network throughput.

3.5.2 Experiments

The experiments in this project will be done in a sequential manner, the first experiments will be executed on IncludeOS to confirm that they work

as expected since as mentioned earlier it is the limiting factor in this thesis being under development still. Docker and the Ubuntu VM instance will then be tested, both will be running 14.04 as stated. All the virtual machines are able to run on platforms where QEMU and Docker is supported in the event of some research being done that would redo these experiments.

As described the experiments entail that the virtual machines are tested in terms of TCP performance, UDP performance and resource footprint on the underlying host. For TCP there will be two scenarios produced where there are different types of load applied to the systems. In the first scenario of TCP, httperf will generate a steady load of requests to the virtual hosts' web-servers over time illustrated in figure 3.5 to establish how each of the systems perform under a steady stream of requests. The second scenario will test the systems while the load, or amount of requests will increase with each iteration which will demonstrate their performance at different levels of load illustrated by 3.6 until the virtual machines encounter a stop in the response, at which point the experimental run will end. This is done by using autobench to generate load, which is httperf with a wrapper to add some additional options to manipulate the load pattern. To ensure consistency between the two scenarios it will be calculated how long the increasing load pattern will run on the systems and the same amount of time constraint will be applied to the continuous load. The experiments will be repeated multiple times over the systems on both hypervisors in order to achieve as precise and complete results as possible.

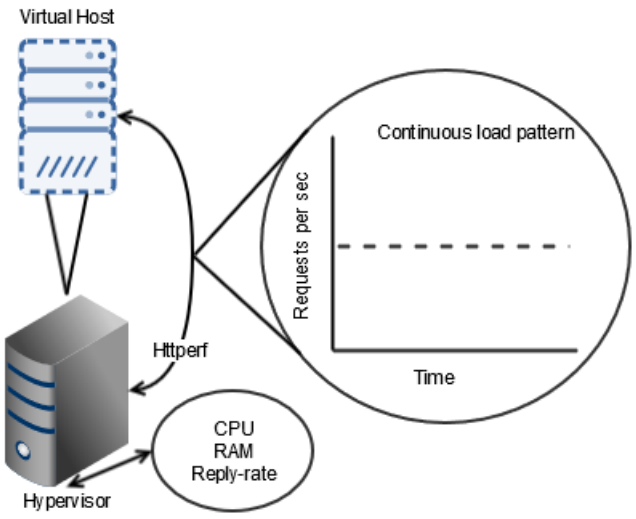

Figure 3.5: TCP scenario 1 illustration

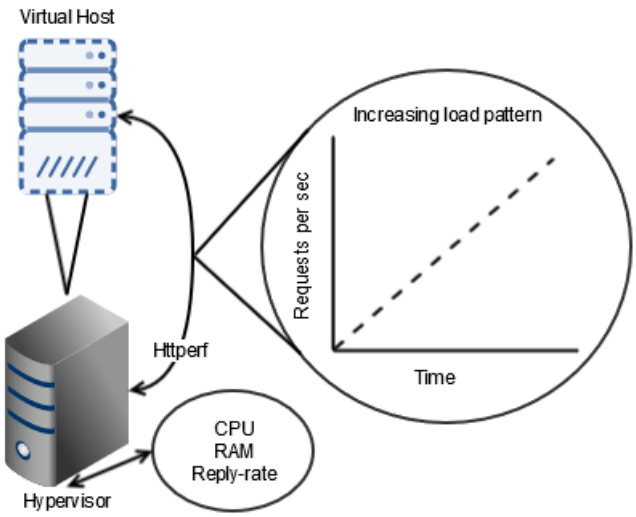

Figure 3.6: TCP scenario 2 illustration

A short time before the experiments are initiated pidstat will start to record the processes utilization of CPU and RAM in order to see the development in usage over the time the experiments use to finish. After the experiments finish running and the data is properly extracted and aggregated, the analysis of the data will be the next step.

3.5.3 Analysis and comparison

The last objective in the project consists of looking at the data in the data-sets which was extracted during the experiments in-depth. The data will be analyzed in the attempt to find an answer regarding the problem statement which is comparing the efficiency of the different systems with focus on IncludeOS and how it performed in comparison to Docker and Ubuntu. The factors which will be looked in order to determine an answer are:

- Average response-rate of the systems.
- How far the systems can go before they start to saturate.
- The time it takes to receive and return 100000 UDP packets, where the returning packets are 1KB each.
- Utilization of CPU and RAM of the systems during the experiments.

Average response-rate is the first factor to be considered, this will determine the overall speed of the systems and how the kernel in the different systems handle the types of load in terms of performance of the web-servers on them. For UDP the factor which will be considered is the amount of packets that are transmitted to the virtual hosts and sent back in regards to the time it takes for this action to be completed.

The second factor is the systems ability to sustain the loads over time, in addition to how far the system can be pushed with increasing loads over time before the system no longer is stable in regards to response-rates and/or replies to requests made within a reasonable time-frame. The longer the test goes on successfully the better the result for the serving virtual machine as the traffic will increase linearly with each iteration of the benchmark.

The final factor is the CPU and RAM utilization of the systems during the the different experiments, the utilization will show what resource footprint they put on the underlying host while they are executing and replying to the requests made. This factor is not to be considered alone, rather in relation to the two above-mentioned factors as this is measured while the data concerning the two factors are tested.

Part III

Conclusion

Chapter 4

Results I - Desgin

This chapter contains and displays the results of the tasks which were defined in the approach chapter, how they were implemented and if something was changed, anything from design to implementation will be explained further in-depth after which an analysis will follow.

4.1 Mimicing IncludeOS

IncludeOS implements its services with C++ code which is located in the `/seed` directory, there is a file, *service.cpp* which is the actual service which the virtual machine will execute when running. As there is a TCP-server implemented, which by default serves an IncludeOS HTTP web-page to demonstrate TCP functionality in the OS through the use of C++ sockets. This service has been modified in order to increase the size of the page being served, as mentioned IncludeOS can currently handle sending one ethernet frame in terms of data, that is 1.5KB which is the size of a packet. In this implementation, the modification to the service entails replacing the current HTML which exist in the *service.cpp* file to instead output 1000 instances of the character 'A' in order to have a web-page being served at a kilobyte with the HTTP header included.

The decision to employ a character string of 1000 characters to be served was taken as at the time there was no filesystem implemented within IncludeOS making it unable to serve anything other than text. Furthermore, in order to serve other types of web-programming languages than HTML (e.g. PHP) there needs to be pre-requisites installed which at the current time is not supported by IncludeOS as is. Further, as the service needs to be reproduced on other systems in order to achieve a fair comparison a simple solution is required. Therefore, a character string allowed the author to define the size of the web-page which was served. In addition, using a character string allowed for an easier job in consideration to implementing a mimic of the service.

In terms of what was modified inside the service before compilation in addition to manipulating the output, it was stripped of everything

that was deemed unnecessary for it to provide the function of a web-server. Every command which directs output to the screen, meaning `printf` commands as they are costly in terms of resource usage when executed for every request throughout the benchmarks as they require kernel instructions to perform I/O meaning that the virtual machine would need to perform. Kernel calls within a guest operating system gets handled by the virtual machine manager which analyses the trap instruction and simulates behaviour accordingly. In addition, all HTML and CSS was removed as they were not used.

When considering what was done to mimic the service, a program in C++ was written in order to achieve the creation of a TCP socket server as close as possible to do what IncludeOS' service does. The program opens and starts listening to a *socket* on port 5000 in a loop, until a request is received at which point the program will serve the page containing the characters string. The HTTP header was copied from IncludeOS to ensure that they serve equal content.

For UDP, IncludeOS has a service that was developed in the start of IncludeOS' life-cycle. The service works in the same sense as the TCP server, except it does not explicitly serve own data. When the UDP service hosted on IncludeOS receives packets, it iterates through them in a loop and then starts sending the packets back to the client with the contents untouched. The service was modified in order to implement a way to be able to tell when the last packet of a packet-stream was received, this is done by the sender/client program sending the last packet by injecting the string "LAST" into the packet which the server checks for and if received it returns it to the client.

4.2 Modification of the IncludeOS service

The service file of IncludeOS was modified as mentioned in terms of output, listing 4.1 shows the modified service. As mentioned, all `printf` commands have been removed as they are costly to run in the frequency that the benchmarks issue requests at. In addition, the example below also displays the original output which was removed.

Listing 4.1: Modified IncludeOS TCP service excrept

```
1  //ADDED
2  std::string outputString = "";
3  outputString.append(1000u, 'A');
4  stream << outputString;
5  std::string html = stream.str();
6
7  //REMOVED
8  // ALL printf //
9    /* HTML Fonts */
10 std::string ubuntu_medium = "font-family: \'Ubuntu\', sans-serif;
       font-weight: 500; ";
```

```
11 std::string ubuntu_normal = "font-family: \'Ubuntu\', sans-serif;
       font-weight: 400; ";
12 std::string ubuntu_light = "font-family: \'Ubuntu\', sans-serif;
       font-weight: 300; ";
13   /* HTML */
14 stream << "<html><head>"
15 << "<link href='https://fonts.googleapis.com/css?family=Ubuntu
       :500,300' rel='stylesheet' type='text/css'>"
16 << "</head><body>"
17 << "<h1 style= \"color: " << "#" << std::hex << (color >> 8) << "\">
       "
18 <<  "<span style=\""+ubuntu_medium+"\">Include</span><span style=\""
       +ubuntu_light+"\">OS</span> </h1>"
19 <<  "<h2>Now speaks TCP!</h2>"
20 // .... generate more dynamic content
21 << "<p> ...and can improvise http. With limitations of course, but
       it's been easier than expected so far </p>"
22 << "<footer><hr /> &copy; 2015, Oslo and Akershus University College
        of Applied Sciences </footer>"
23 << "</body></html>\n";
```

As can be seen, the change in the service file is not significant, however important for optimum performance during the experiments. The UDP service was also modified in accordance to the experiments, which is noted in chapter 3.1, mainly the *socket read* function within the service was changed, the modified function is displayed in listing 4.2.

Listing 4.2: Modified IncludeOS UDP service excerpt

```
1  sock.onRead([] (UDP::Socket& conn, UDP::addr_t addr, UDP::port_t
       port, const char* data, int len) -> int
2      {
3          std::string payload = "";
4          payload.append(1000u, 'A');
5          const char* loadpay = payload.c_str();
6          std::string str(data);
7          if( str.find("LAST") != std::string::npos )
8          {
9              conn.sendto(addr, port, data, len);
10             return 0;
11         }
12         conn.sendto(addr, port, loadpay, strlen(loadpay));
13         return 0;
14     });
```

4.3 Scripts

As mentioned in section 3.2, scripts which allows for automation has been developed. There are two scripts in total that was created:

- Benchmark TCP and collect stats.

- Benchmark UDP and collect stats.

Both scripts essentially execute the same benchmark and collect statistics on the systems while the benchmarks are active. The scripts take a snapshot of the current ticks of the CPU before executing the benchmarks, then start the pidstat/docker stats process which collects percentage usage of the CPU and RAM on the running guest, for Docker the script also watches the daemon that manages Docker.

4.3.1 QEMU

The first script, referred to as `collector.sh` takes three parameters as input. These parameters are duration, rate and an extension to the name of the file it outputs when it completes a run. The duration defines how long the benchmark will run in seconds by taking the inputted rate and multiplying it with the inputted duration which is the maximum connections value, e.g. 60 500 1 would execute a httperf benchmark which runs for 60 seconds (60x500=30000, which at 500 connections per sec takes 60 seconds). Figure 4.1 shows an overview how the script functions.

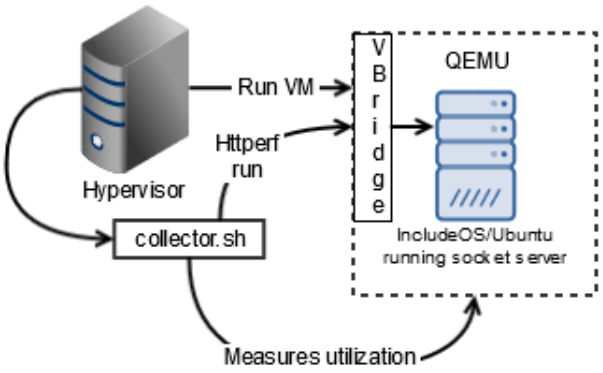

Figure 4.1: High level illustration of collectors functions.

While running the script calls *pidstat* a second before executing the benchmark to follow the resource footprint of QEMU while generating a report every second and putting it into a file and exiting a couple of seconds after the benchmark finishes. The data from the runs of httperf gets stored in a separate text-file in order to display data from the TCP performance itself. In addition, ticks of the CPU is recorded, an initial value is extracted a moment before the benchmarks executes and a new value as soon as it finishes running, subtracting the difference from end to start giving the ticks throughout the experiment. When the script finishes its run, it generates three text-files for that run, one which holds the httperf statistics after the run, the second text-file contains the data of the utilization of CPU and RAM throughout the run, while the last one contains the CPU ticks for user, kernel and guest during the run. Listing 4.3 shows pseudo-code for the script, the actual code can be viewed in the appendix chapter.

Listing 4.3: Collector.sh pseudo-code

```
#!/bin/bash

PID=QEMU_PID
start pidstat on QEMU and keep writing output to file
record user,kernel ticks from /proc
start httperf towards the QEMU guest OS >> to file
stop pidstat
record user,kernel ticks from /proc
totalTicks=endTicks-startTicks
write tick data to text-files
```

4.3.2 Docker

Docker, as mentioned is different from most applications in terms of measuring utilization. To the authors best knowledge, this could only be achieved by employing a tool which comes with docker, `docker stats`. When this command is executed a live stream of the containers usage of CPU and RAM is displayed, meaning that measuring with *pidstat* would not give accurate results, in addition, according to the authors research neither built-in commands in Linux (`top` and `ps`) would give the correct usage as they utilize. As such, a modified version of the script was implemented for Docker.

The script `dockerCollector.sh` executes the same benchmark towards the Docker containers as `collector.sh` does towards QEMU and is executed in the same fashion; `./dockerCollector.sh 60 500 1`. One of the main differences lie in measuring the resource footprint of the Docker-daemon as well which manages the Docker containers as it is a process that is required to be able to host containers it is natural to add its usage together with the containers. Figure 4.2 gives an overview of how this script functions and listing 4.4 shows pseudo-code for the script, the actual code can be viewed in the appendix chapter.

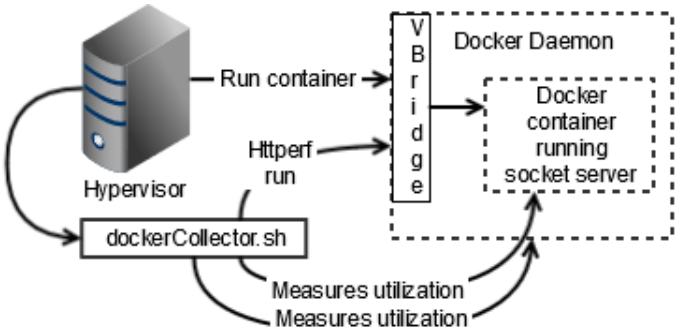

Figure 4.2: A high level illustration of dockerCollectors functions.

Listing 4.4: dockerCollector.sh pseudo-code

```
1  #!/bin/bash
2
3  PID=DOCKER_DAEMON
4  start pidstat on DOCKER_DAEMON
5  start docker stats and keep writing output to file
6  record user,kernel ticks from /proc
7  start httperf towards the Docker container >> output to file
8  stop pidstat
9  stop docker stats
10 record user,kernel ticks from /proc
11 totalTicks=endTicks-startTicks
12 write tick data to text-files
```

Initially, as mentioned the script functions in much of the same manner as the one that gathers statistics for QEMU. It starts by getting the ID for the running container which is used to extract a snapshot of the ticks for the CPU on the server. Moving on, the script calls the `docker stats` command and runs it as a background process while its PID is recorded in order to kill the process after the benchmark is done executing. When the benchmark finishes running another snapshot is taken of the ticks on the server in order to acquire information about the ticks spent by both. All the information is stored in text-files that contain CPU, RAM, ticks and httperf data.

For UDP the same scripts were used as they already act as a skeleton for benchmarks, the difference is that the httperf command was replaced by `{ time ./udpSend IP PORT PCKT_AMOUNT MICRO_SLEEP; } 2> outputFile` to record the time it takes for the UDP send program to send 100000 packets to the guest and receive 100000 packets of 1KB back from the server running on the guest.

4.4 TCP & UDP programs

The implementation of the server programs for TCP and UDP was not necessary as there existed programs from earlier experiments conducted on the hypervisors which employed these programs to run them. However, the programs had to be modified in order to fit the purpose of the experiments for this project. For both of the server programs, in the same manner as IncludeOS' services (TCP and UDP) the modification entailed the configuration of the output when serving a request on the socket.

As with IncludeOS, these programs were also configured to return a string of characters of 1000 instances which corresponds to about 1.3KB with the HTTP header included for the TCP service, in the case of UDP which does not transmit a HTTP header it is roughly equivalent to 1KB in order to stay under the size of an ethernet frame, which as mentioned IncludeOS needed during the initial tests in order to be able to send all the data for a single request. In addition, the UDP server also iterates through

the packets content, if the string "LAST" is in one packet it will return it to the client. Listings 4.5 and 4.6 illustrates the part of the programs where modification was made. The setup for initializing the socket server has been omitted as the part for handling the sending is what is important and the initialization can be recreated in multiple ways.

Listing 4.5: Modified TCP server C-program excerpt

```
1 sendBuff=1000 instances of A;
2 http=INCLUDEOS_HTTP_HEADER;
3 http += sendBuff;
4         size_t http_size = http.size();
5         write(connfd, http.data(), http_size);
```

Listing 4.6: Modified UDP server C-program excerpt

```
1  for (;;)
2  {
3      len = sizeof(cliaddr);
4      n = recvfrom(sockfd,mesg,10,0,(struct sockaddr *)&cliaddr,&len);
5      if( strcmp(mesg, "LAST") == 0 )
6      {
7          sendto(sockfd,mesg,n,0,(struct sockaddr *)\
8          &cliaddr,sizeof(cliaddr));
9      }
10     else
11     {
12         sendto(sockfd,sendArray,sizeof(sendArray),0,(struct sockaddr
               *)\
13         &cliaddr,sizeof(cliaddr));
14     }
15 }
```

In addition to the server programs, there also exists a UDP send program which was used, the program starts a loop which `forks` the process. The main process sends the packets to the specified server where the child-process listens to replies for each packet and when the last packet is received, the program terminates. This is done by checking the contents of each received packet and comparing the content towards a pre-specified string, in this case the string "LAST", however it could be anything from a letter or word to a specific number assuming it is modified in the server as well. Listing 4.7 shows an excerpt of the fundamental function of the program.

Listing 4.7: Send and receive loop of udpSend

```
1 if(!fork())
2 {
3   for(i=0;i<max;i++)
4   {
```

```
5     sprintf(buf,"%d",i);
6     usleep(microSeconds);
7     n = sendto(sockfd, buf, strlen(buf)+1, 0,\
8     (struct sockaddr *)&serveraddr, serverlen);
9   }
10    sprintf(buf,"LAST");
11    usleep(microSeconds);
12    n = sendto(sockfd, buf, strlen(buf)+1, 0,\
13    (struct sockaddr *)&serveraddr, serverlen);
14 }
15 else
16 {
17   while(recvfrom(sockfd, reply, 150, 0, (struct sockaddr *)\
18   &serveraddr, &serverlen))
19   {
20     if(strcmp(reply,"LAST") == 0)
21     {
22       printf("Received LAST, %d packets received\n",packets);
23       return 0;
24     }
25     packets++;
26   }
27 }
```

Chapter 5

Results II - Implementation and Experiments

This chapter will explain the implementation of the the work that done through this project in terms of the set-up of the infrastructure and the experiments in order for other researchers to replicate what has been done in the event of wanting to build upon it or testing it themselves.

5.1 Setting up

This section explains how the hypervisors were set up and how the different technologies were installed and configured.

5.1.1 Hypervisors

The hypervisors that are used during this project will both run a 64-bit version of Ubuntu 14.04. The main difference between them is that one of them is an AMD platform while the other one is Intel. The hardware specifications for the hypervisors is shown in table 5.1 and 5.2 below.

Table 5.1: Trident hypervisor

Hardware specificatons		
Hardware	Name/Type	Spec
CPU	AMD Opteron 6234	48 cores @ 2.4GHz
RAM	DDR3 DIMM	135.2GB @ 1.6GHz

Table 5.2: Intel hypervisor

Hardware specificatons		
Hardware	Name/Type	Spec
CPU	Intel(R) Xeon(R) E5-2699 v3	36 Hyperthreaded cores @ 2.3GHz
RAM	DDR3 DIMM	135.1GB @ 2.1GHz

Before continuing with the operating system virtualizations there are some prerequisites[1].

5.1.2 IncludeOS

The first step is downloading IncludeOS through git which can be cloned from their repository[2]. Then `cd` into the cloned directory and run `./etc/install_from_bundle.sh`. The background chapter mentioned that at the time of writing the version of IncludeOS was 0.7.0, however due to constraints in that version (further explained in chapter 7) the version which the experiments were conducted on is the 0.8.0 release candidate.

Once the script finishes installing there should be a directory named `/home/user/includeos_install/` where the specifications for the QEMU instance is located. In these experiments the RAM is set to 1024MB (default is 128MB) to handle the amount of requests.

The installation can be tested by running `./test.sh`, IncludeOS can then be booted after compiling the `service.cpp` file in the `seed/` directory and running `./run.sh IncludeOS_service.img`.

5.1.3 Docker

To install and set up a Docker environment first the host on which it is installed requires some actions to be taken to ensure that Docker is installed correctly which can be found on Dockers installation page[3].

The next step is building an image from which Docker can launch its containers, for this a `Dockerfile` is created. This file is an user specified commands for what Docker will do once the container is booted. Listing 5.1 illustrates the contents of the Dockerfile used in this thesis, what this Dockerfile does is:

- Copy the file `server.cpp` illustrated in listing from the same directory as the Dockerfile unto to the containers root directory.
- Install the software-properties-common package.
- Update the APT-list.
- Install the build-essential and gcc packages.
- Compile the C and C++ files `server.cpp` and `udpserver.c`.
- Run the compiled program.

[1]The hypervisor has: A) *git*, B) *pidstat*, *httperf+autobench*, D) *numactl* and E) *QEMU* installed.

[2]https://github.com/hioa-cs/IncludeOS.git

[3]https://docs.docker.com/engine/installation/linux/ubuntulinux/

Listing 5.1: Dockerfile contents

```
1 FROM ubuntu:14.04
2 ADD server.cpp /server.cpp
3 ADD udpserver.c /udpserver.c
4 RUN apt-get -y install software-properties-common
5 RUN apt-get update
6 RUN apt-get install -y build-essential gcc
7 RUN apt-get install -y g++
8 RUN g++ server.cpp -o tcpServer
9 RUN gcc udpserver.c -o srv
```

Building a docker image from this Dockerfile is done by running `docker build -f /path/to/Dockerfile` which starts the process of building the image, a name can be specified as well. After the image is built, its name or ID can be utilized to run a container from the image: `docker run -i -t ID/IMAGE_NAME` which results in shell on a container that acts like an Ubuntu 14.04 VM with the commands specified in the Dockerfile already executed. By default Docker creates a virtual bridge interface on the address `172.17.0.1` and the first container takes `172.17.0.2`.

For measuring the hardware utilization of a Docker container under load, Docker has has a built-in system which live streams a containers runtime metrics and can be used by executing the command `docker stats CONTAINER`. The output of the command lists the utilization of among many metrics, CPU and RAM, to acquire the data it uses the */sys/fs/cgroup* system as opposed to *pidstat* which uses */proc*.

5.1.4 Ubuntu

The virtual machine that runs Ubuntu is set up through a server image that is run by QEMU. The network is set to connect to a virtual network bridge that the guest system and the hypervisor communicate through.

The specifications of the virtual machine is defined in the run command of QEMU, the guest is set to utilize 1 CPU core and have 1024MB of RAM with KVM enabled. This is done by adding the flags; `cores=1, -m 1024` and `-enable-kvm`. Listing 5.2 shows the full command to start the virtual machine. The version of Ubuntu here is 14.04[4] as that was the latest long term support version at the start of this thesis. As with Docker, the C/C++ socket server application `server.cpp` and `udpserver.c` is used to emulate IncludeOS' web-server by compiling and then running the program.

Listing 5.2: QEMU run commnad

```
1 qemu-system-x86_64 -enable-kvm -name server3 -m 1024 -realtime \
2 mlock=off -smp 1,sockets=1,cores=1,threads=1 \
3 -hda /root/ubuntu14.04.amd64.3G3.img \
```

[4]http://releases.ubuntu.com/14.04/ubuntu-14.04.4-server-amd64.iso

```
-netdev tap,script=/root/qemu-ifup,id=hostnet0 \
-device rtl8139,netdev=hostnet0,id=net0 -vga none -curses&
```

5.1.5 Numactl

Numactl is an application that allows its users to bind processes to specific CPUs and RAM chips. On systems with multiple CPU sockets there is bound to be some discrepancy in terms of performance when it comes to CPU and RAM as one bus for a specific RAM chip can be congested. NUMA allows a CPU to access its local memory faster than memory that is not local, it displays the system in what is called NUMA nodes which is a representation as to which CPU has local memory on which node. Listing 5.3 shows an example of how this looks, this list has been shortened.

Listing 5.3: Numactl displaying the nodes (excerpt)

```
available: 8 nodes (0-7)
node 0 cpus: 0 4 8 12 16 20
node 0 size: 16039 MB
node 0 free: 15413 MB
node 1 cpus: 24 28 32 36 40 44
node 1 size: 16126 MB
node 1 free: 15450 MB
node 2 cpus: 2 6 10 14 18 22
node 2 size: 16126 MB
node 2 free: 15245 MB
node n cpus: 26 30 34 38 42 46
node n size: 16126 MB
node n free: 15326 MB
```

During the experiments on both trident and intel NUMA has been employed to bind the virtual machines as processes move between the different CPUs on a system depending on the scheduler. The scheduler can not guarantee that the process ends up on a CPU in the same node as the memory it uses meaning that there will be some small inconsistencies in terms of overall performance between different runs of the experiments. Docker has this built in on NUMA enabled systems and can be utilized by specifying two options in its `run`, `--cpuset-cpus=""` and `--cpuset-mems=""`.

5.2 Experiments

This sections will explain the experiments in-depth and the programs which are used to run them. Each experiment was executed 30 times on each guest host in order to acquire a data-set which has adequate amount of entries in order to analyze.

Following the set-up for installing the different guest operating systems on the servers the next step was to run the experiments towards the

virtual hosts and container. There were two metrics chosen for the persistent connection experiments, the first was to run httperf for 60 seconds towards the serving guest with a rate of issuing requests at 500 requests each second for the index page of the web-service, which in this case is the 1000 instances of 'A' getting returned to the client. Running the script for this experiment is done by `./collector.sh 60 500` where the output file-extension has been omitted. The result is a httperf command being run by the script which is equivalent to running httperf with: `httperf --server GUEST_HOST_IP --rate 500 --num-conn 30000`, the timeout value for httperf is set to 2 seconds meaning that if a request does not get a response for 2 seconds after being issued, it will be regarded as an error. The rate of 500 was selected as it indicates that the web-service is being used by a medium amount of users as the author did not consider to test it with any lower rates in order to be able to extract data under load.

The second experiment for TCP was repeating the first one, with the exception of doubling the amount of requests per second towards the serving virtual hosts and container. In the same manner as the first experiment this also entails 60 second runs of httperf with the rate set to 1000 in order to to test the guests performance under what can be considered heavy load as there is only one server which only uses one CPU core meaning that there is no load-balancer to even out heavy load across multiple guests. Both of the scenarios for httperf is conducted with the option `num-call` set to the default value of 1, this options dictates the amount of requests for each connection made to the serving guest.

The UDP experiments were conducted in a similar fashion to TCP, as shown earlier the *sendUDP* program was utilized to send the packets to the listening *udpServer* program on the Ubuntu VM and Docker, and IncludeOS used its own modified service which is shown in chapter 4. The experiments executed towards the guest hosts was as previously explained, done by sending a packet-stream of 100000 packets towards the guest host running a UDP web-service. Each packet sent to the server contained the number which represented the current iteration of the sending progression from the client. The UDP server on the guest hosts receive the packets after which it returns a 1KB packet to the client. In order to ensure that all 100000 packets arrive back at the sending client there is a *microsleep* function added in order to have some delay between the packets that are sent, for IncludeOS and Docker it was adequate to set this timer equal to 2 microseconds, whereas on the Ubuntu VM this timer had to be increased to 45 microseconds otherwise there would be some packets lost in transmission. Inherently this timer does have an effect on the overall time it takes for the guests to finish a run of the experiment, however this is needed in order for consistent tests.

Each experiment conducted on the systems will be split the scenarios when the data from the finished experiments is presented, which is the next step.

Chapter 6

Results III - Data & Analysis

This chapter is split into two parts, data presentation and analysis. The presentation will look at the general data which was acquired through the experiments described in chapters 3.1 and 4. The analytical part will look at the data through statistical methods, in the case of this project. Each experiment was conducted 30 times on each guest operating system of which the results have been recorded and plotted into graphs.

The first scenario, as described was executing httperf towards the guest operating system with a demanded request-rate of 500 requests per second over 1 minute, the second scenario was executing httperf towards the guest operating systems with a demanded request-rate of 1000 requests per second over 1 minute. Finally, the was scenario was to send a stream of 100000 UDP packets towards the guest operating system which was replied to with 100000 packets at 1KB each. The CPU time for all scenarios were recorded by taking a snapshot of the current CPU ticks on the system before the experiment and a new snapshot after the experiment, then subracting the start value from the end value to obtain the ticks used.

6.1 AMD

6.1.1 IncludeOS

6.1.1.1 Scenario 1 - TCP 500 req/s

For this scenario, in terms of reply rate IncludeOS averages at 0.22 ms. There were two cases where the reply time deviated from 0.2 ms where the reply times were 0.7 ms, no evidence was found to indicate why there were a spike of which is almost 250% higher than the average was. Over the course of the experiments for each run 300003 packets was transferred between the server and the client.

The total average CPU utilization of IncludeOS during the experiments were at 32.7% CPU usage, the figure 6.1 shows the average CPU per run of the experiments. Also here there was a case where the usage deviated from

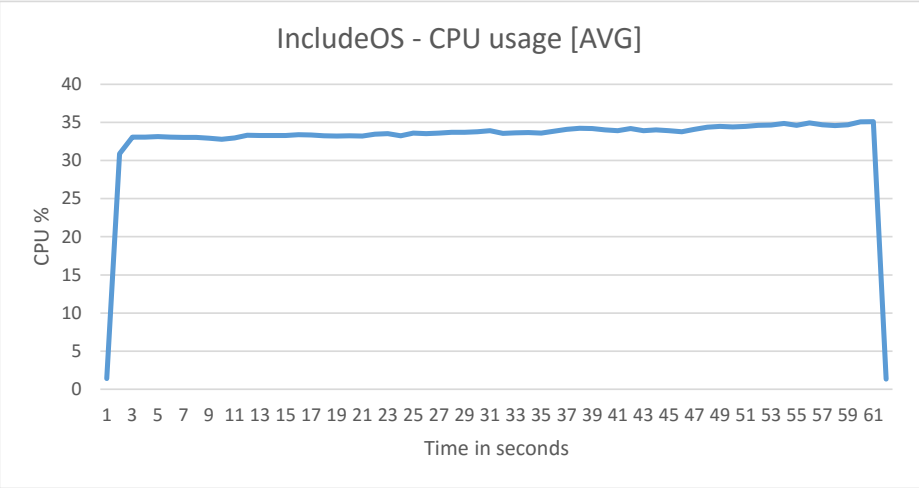

Figure 6.1: Average CPU utilization of IncludeOS

the average at which point the CPU climbed to 75% usage from the start of the experiment, this spike is for the same run as where the response time also experienced a major deviation.

In terms of CPU time, figure 6.2 shows the average time IncludeOS spent inside the CPU (arbitrary time unit measured in ticks/jiffies) for the experiments, the total average CPU time for IncludeOS was 2026.7 ticks. The deviation from the earlier graphs is evident here as well where for the run it happened, IncludeOS used about 50% more CPU compared to the the other runs.

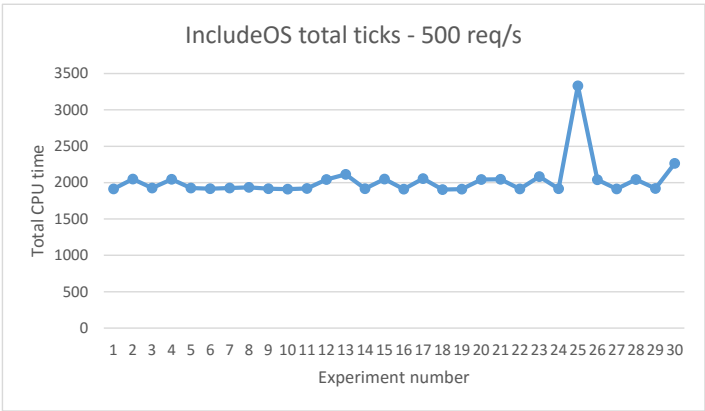

Figure 6.2: CPU time of IncludeOS

In terms of RAM utilization, it should be mentioned that in the TCP service of IncludeOS there exists a memory leak as evidenced by figure 6.3. This memory leak is constant throughout all the experiments involving TCP for both platforms which invalidates the data on RAM utilization in comparison to the other technologies. The leak is further evidenced by *pidstat* displaying a steady increase in resident set size and virtual memory

over time which should be constant. However, it will be mentioned as in comparison to the 1000 req/s experiments it is still relevant. The average usage of memory during the runs is equivalent for all the experiments, and the average consumption for all the runs is 0.086% of total memory which equates to 117.3 MB with the lowest and highest being 27.04 MB and 202.8 MB respectively.

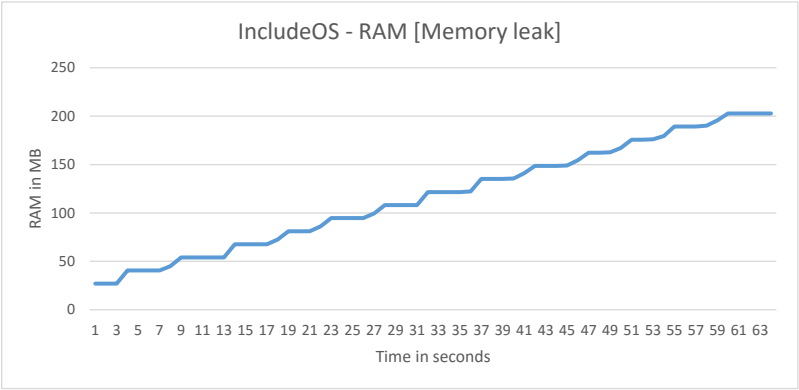

Figure 6.3: Showcasing the memory leak in the TCP service

6.1.1.2 Scenario 2 - TCP 1000 req/s

For this scenario, the average reply rate were close to that of the 500 req/s experiments. It was observed at a total average of 0.23 ms with the highest point being 0.4 ms and the lowest at 0.2 ms. Here almost double the amount of packets were transferred between the server and the client at 600002 packets for each run.

The total average CPU utilization during the experiments at 1000 requests per second can be seen in figure 6.4

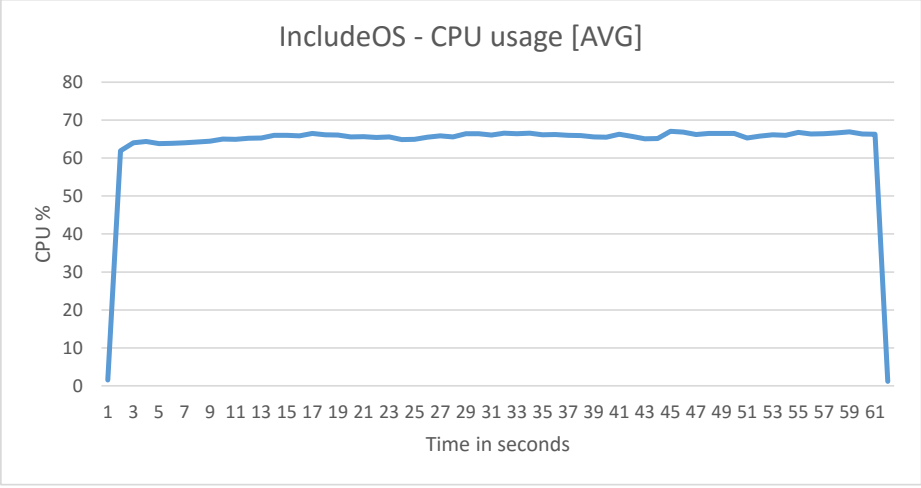

Figure 6.4: Average CPU utilization of IncludeOS

As can be seen the amount of CPU utilization is almost doubled, the total average being at 65.7%. Also throughout the runs of this experiment there are cases where CPU activity has a spike in usage where the CPU climbs to 100% multiple times during one of the runs and once for another run.

The total average CPU time during the experiments for this scenario is 3950.6 which is a little less than the double of the 500 req/s runs. Figure 6.5 shows the average CPU time for IncludeOS in this scenario.

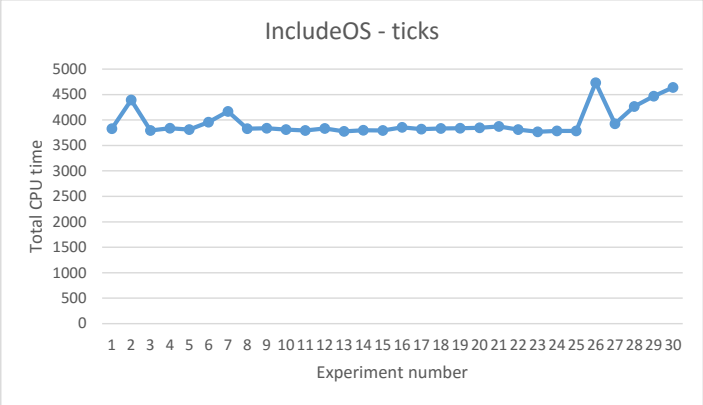

Figure 6.5: CPU time of IncludeOS

The spikes and the increasage towards the end of the experiments is consistent with the amount of CPU usage as there is 4 runs where the average CPU usage for those runs exceed 70% usage. For those runs the amount of CPU time spent is also about 700 larger than the average. In terms of memory IncludeOS averages out at 0.15% of total for almost all of the runs, this equates to 204.2 MB of total. There were some cases where memory usage were higher, although no more than 0.004% at the highest.

6.1.1.3 UDP

For the case of UDP, the average time for the runs came out at 6.61 seconds when receiving 100000 packets and sending 100000 packets back (100MB total of return packets). The figure 6.6 shows the time it took for IncludeOS to successfully respond to the packets over the course of the experiments. As observed the variation is not significant and at the most 150 ms.

In terms of CPU usage during the experiments IncludeOS averages at 94.4% usage over 30 runs. The average start to end usage is shown in figure 6.7, it can be seen that during the experiments the CPU was close to 100% at the time the sending and receiving started.

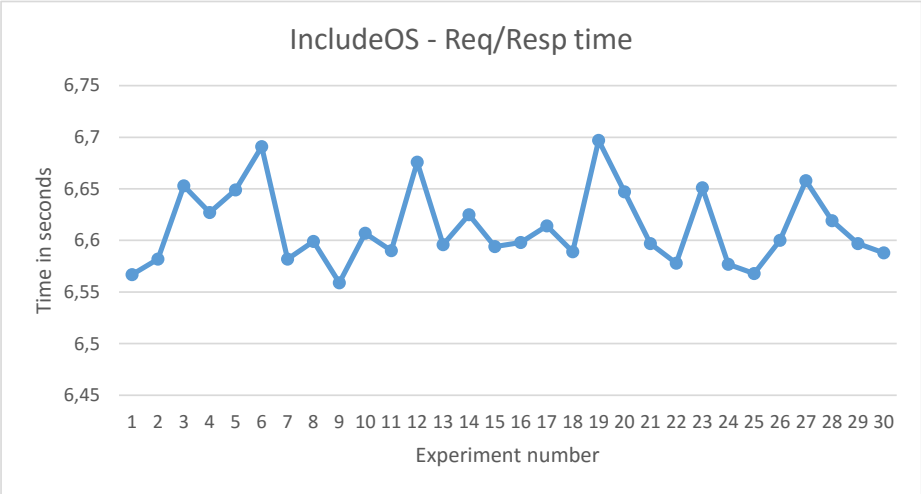

Figure 6.6: Response-time for UDP

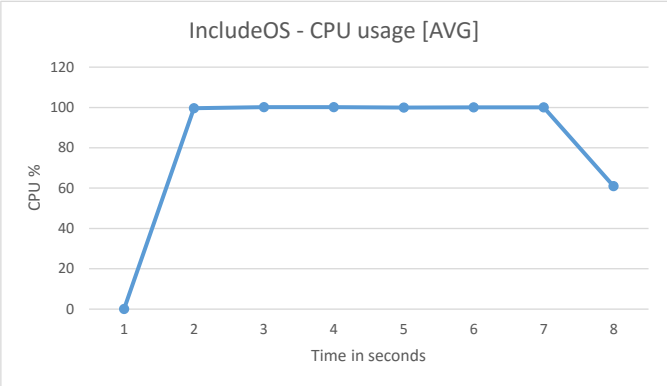

Figure 6.7: Average CPU utilization over the experiments

The UDP-service for IncludeOS does not contain the memory leak which is found in the TCP-service. For the case of the experiments, in all instances the memory footprint on the server were steady at 0.02%, or 27 MB.

In terms of CPU time, the total average over all the experiments is 767.4. Figure 6.8 shows the total CPU time over all the experiments. As can be seen, the variance is not significant.

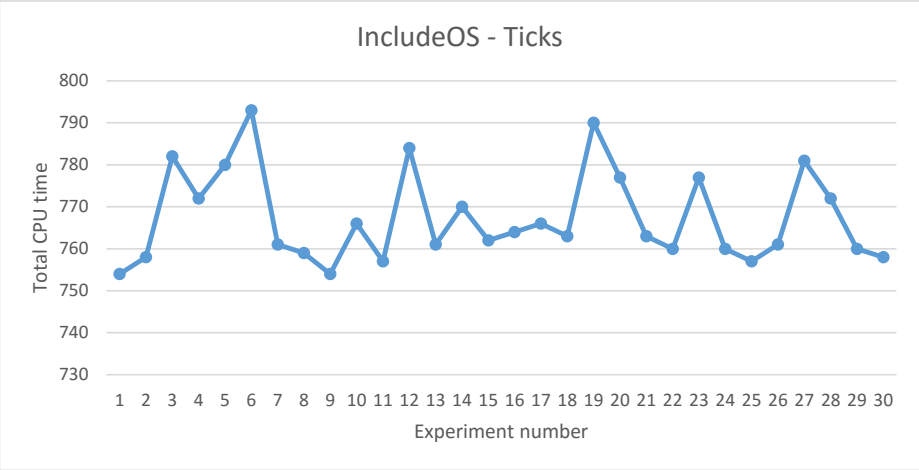

Figure 6.8: CPU time for UDP

6.1.2 Docker

6.1.2.1 Scenario 1 - TCP 500 req/s

In terms of response-time, for all the runs Docker is stable at 0.2 ms, there were no deviations in either of the runs. Over all the runs, the packets which were transferred between server and host are an average of 329942 packets, the variance between the lowest and the highest amount is less than 80 packets.

In terms of CPU usage, Docker averages out at 4.9% CPU usage over the experiments with the highest for one run at 5.5% and lowest at 4.6%. Figure 6.9 shows the total average CPU usage over all the runs. In terms of RAM usage, Docker used an average over the course of the runs of 0.087% which is 117.4 MB. During the experiments this usage was very stable and only deviated by 1 MB towards the lowest point and 2.3 MB towards the highest. It should also be mentioned that when calculating the memory footprint of Docker, both the containers and the daemons usage is added together.

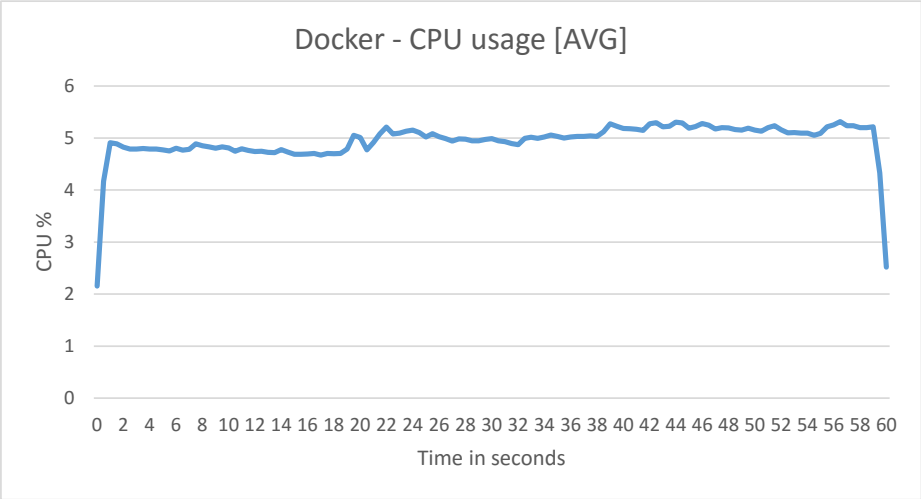

Figure 6.9: Average CPU usage

In terms of CPU time, Docker spent an average of 430 CPU ticks under the experiments, figure 6.10 shows the total ticks spent for each run of the experiment.

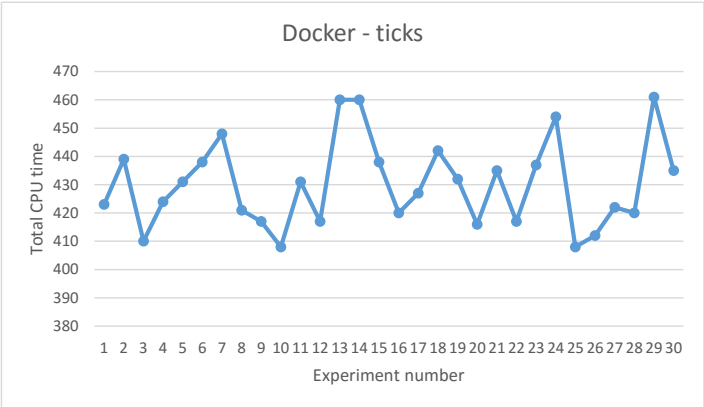

Figure 6.10: CPU time for Docker

As seen, the variance is not significant, and in contrast to the other technologies Docker spent a lot less CPU time being nearly equivalent with bare-metal.

6.1.2.2 Scenario 2 - TCP 1000 req/s

In terms of response-time, the average came out a bit lower than the 500 connection experiments at an average of 0.16 ms response-time. There was however some variance between the runs ranging from 0.1 ms to 0.2 ms below and above the average respectively. The amount of packets transferred between host and client 659534 which is as expected almost the double of that transferred on the 500 req/s experiments.

The CPU usage during the runs for Docker were 10.25% where there were one run at which it used 21.5% and otherwise were stably around the 10% mark. Figure 6.11 shows the average CPU usage over each run.

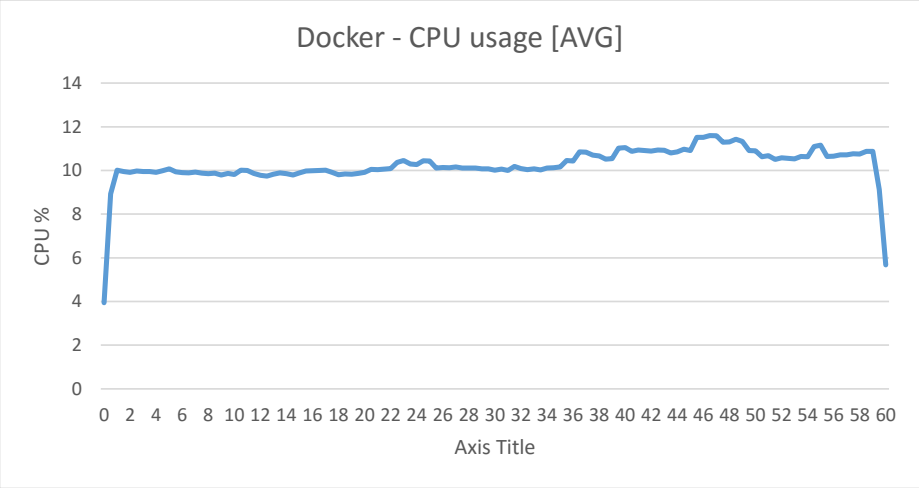

Figure 6.11: CPU usage for Docker

The RAM utilization of Docker over the course of the experiments averaged at 0.15% which equates to an average of 204.2 MB. The variance between the runs is not significant and is in the range of 0.002% at the most.

Over the course of the 1000 req/s experiments Docker used an average of 666 ticks in the CPU, with consideration to the 500 req/s experiments it is about a 50% increase. The total CPU time used per run of the experiments is shown in figure 6.12.

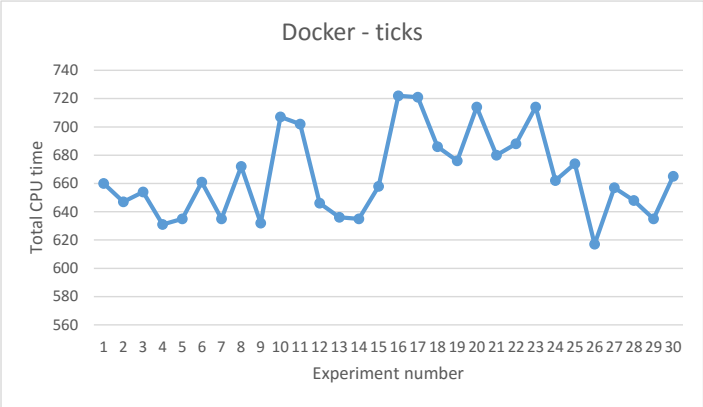

Figure 6.12: CPU time for Docker

6.1.2.3 UDP

For UDP the average total time between receiving packets and responding to them Docker used 7.2 seconds. Figure 6.13 shows the time it took for each experiment to finish.

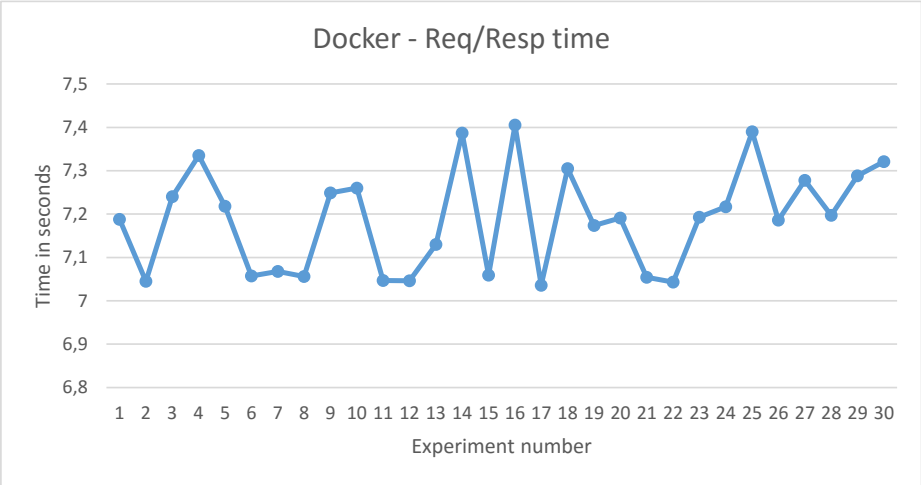

Figure 6.13: Response-time for UDP

As observed, the variance in times for the experiment to complete is close to 0.4 seconds over the course of the runs. The RAM utilization for Docker during the experiments contains no variance, and is stable at 0.44% usage of the total RAM which is equal to 59.6 MB.

The average CPU usage during the experiments was 29.5%, there were some variance during the experiments, there was observed a 2% drop in usage for some of the runs and an increase at about 2.5% in other runs. However most of the runs stayed at around 27%. Figure 6.14 shows the average usage of the CPU over the course of the experiments.

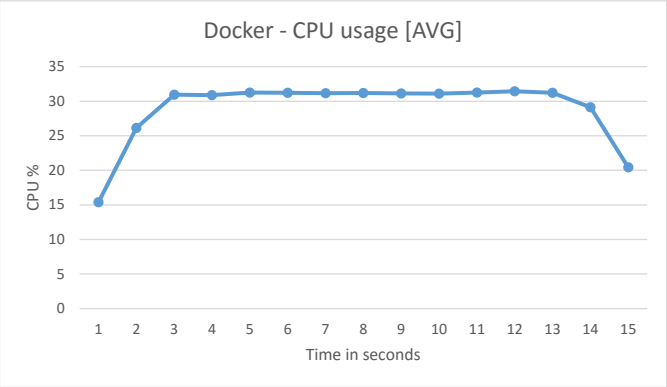

Figure 6.14: CPU usage for Docker

In terms of CPU time during the experiments Docker used an average of 365 total ticks on the CPU. Figure 6.15 shows the total ticks used for each run of the experiment.

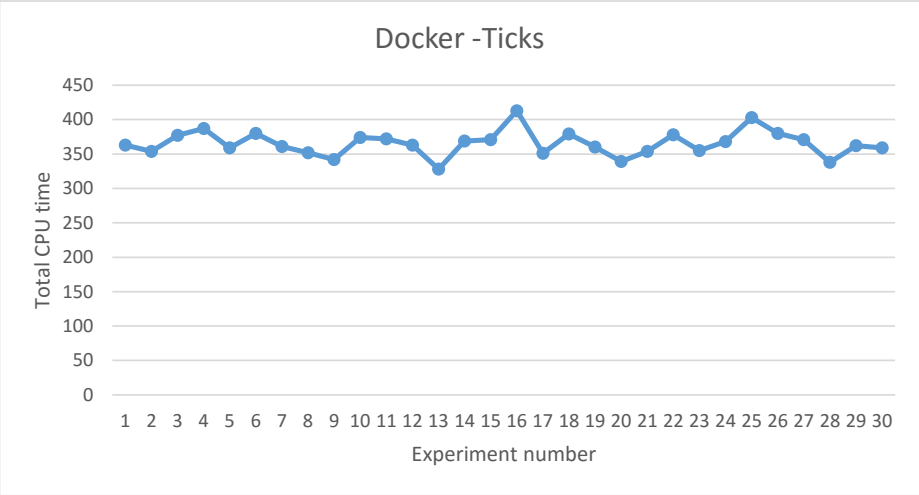

Figure 6.15: CPU time for Docker

As observed, the time spent in the CPU for Docker was mostly around 340 and 350 ticks with some variance going up to 360 and even as high as 400 for two of the runs.

6.1.3 Ubuntu VM

6.1.3.1 Scenario 1 - TCP 500 req/s

In terms of response-time, the Ubuntu VM requried 0.3 ms to answer the requests on each of the runs. The average amount of packets transferred between the host and the server for each run was 300012 packets.

The average CPU usage over the course of the runs were 40.1% with very little variance, the lowest run showed an average usage of 39.9% where as the highest being 40.7%. The average CPU usage for each experiment can be seen in figure 6.16. The RAM usage of the Ubuntu VM over the course of the experiments stayed at 0.18% or 243.5 MB where there were one run at which it used 0.19% or 257 MB. This averaged the memory usage out at 0.183% which is 244 MB.

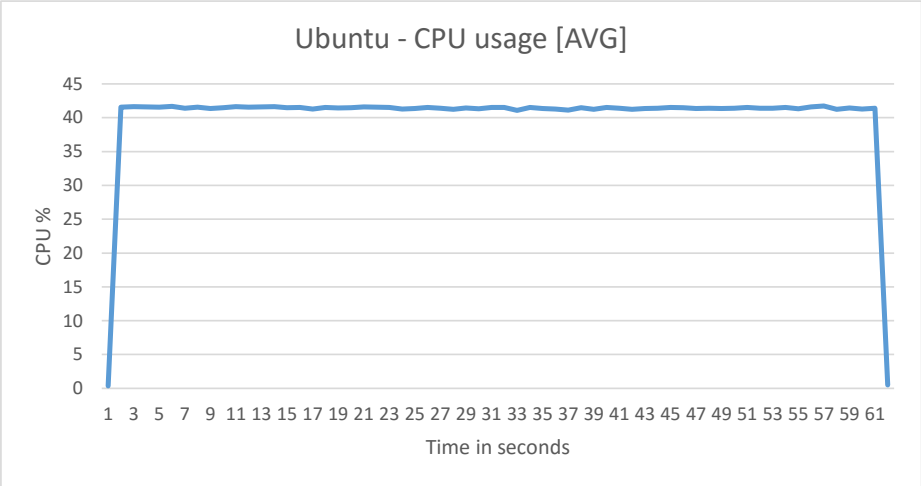

Figure 6.16: CPU usage Ubuntu VM

In terms of CPU time spent by the Ubuntu VM during the experiments, it was a total average of 2487 ticks. The total time spent for each run can be seen in figure 6.17, it can be seen that the variance here was about 30 ticks towards each way of the spectrum.

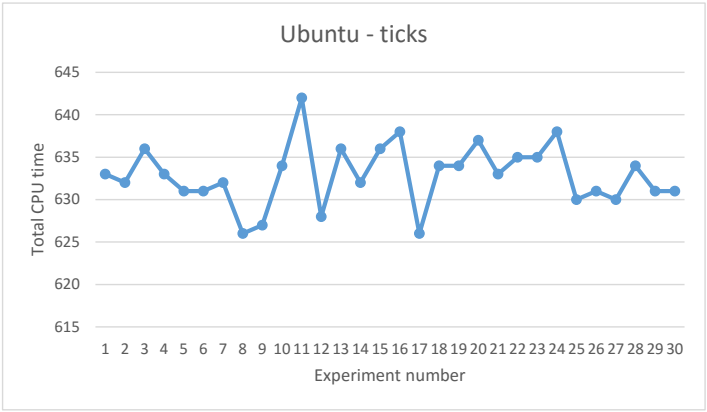

Figure 6.17: CPU time Ubuntu VM

6.1.3.2 Scenario 2 - TCP 1000 req/s

For the experiments at 1000 requests per second the average response-time for the Ubuntu VM was also 0.3 ms as with the previous scenario. The amount of packets transferred between the server and the client averaged out at 600024 packets per run.

The average CPU usage during the experiments was 71.7% with no significant deviation from the average, the highest difference in usage was 2% higher usage in a few of the runs. Figure 6.18 shows the average CPU usage through the experiments. In terms of RAM usage 50% of the runs utilized 0.18% memory and the other 50% utilized 0.19%, or 243.3 MB and 257 MB respectively. A total average of the RAM usage was recorded at 0.185% which is equivalent to 250 MB.

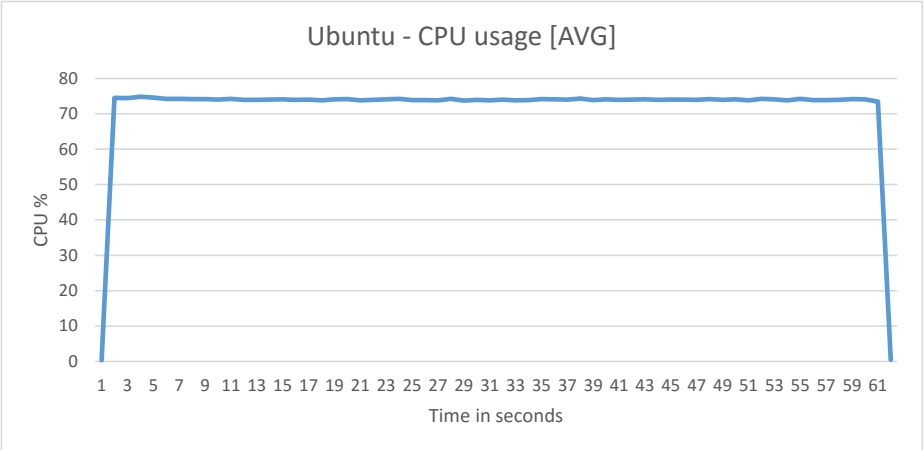

Figure 6.18: CPU usage Ubuntu VM

The CPU time spent by the Ubuntu VM during the experiments was an total average of 4248 ticks on the CPU during the runs of the experiments. Figure 6.19 shows the total CPU time spent for each run of the experiment.

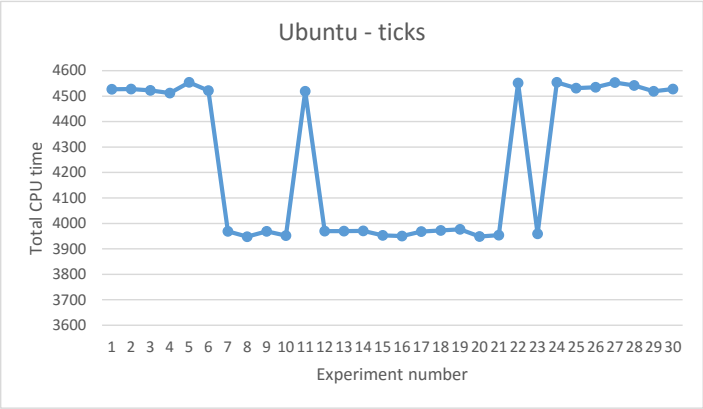

Figure 6.19: CPU time Ubuntu VM

It can be observed that the amount of ticks had a higher level of variance than earlier experiments. When considering the CPU usage it can be seen that this is to be expected, also here half of the runs were in 4500 ticks range where as the other half in the 3970.

6.1.3.3 UDP

The average time used for the Ubuntu VM to receive and send an answer back to the client was 17.7 seconds. The time it took for each run of the experiment can be seen in figure 6.20. As observed, the variance here is at about 100 ms betwenn the runs.

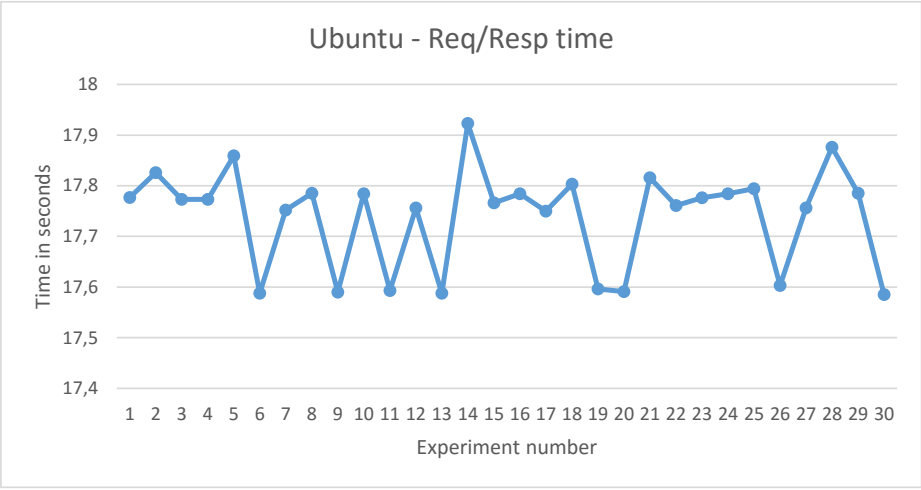

Figure 6.20: Response-time for UDP

The average CPU usage throughout the experiments was 93.3% of the CPU where in some instances it uses 92.7% and 94.3%. Figure 6.21 shows the average CPU usage for all the runs. In terms of RAM usage during the

experiments, the Ubuntu VM used 0.14% during each of the runs which is 189 MB.

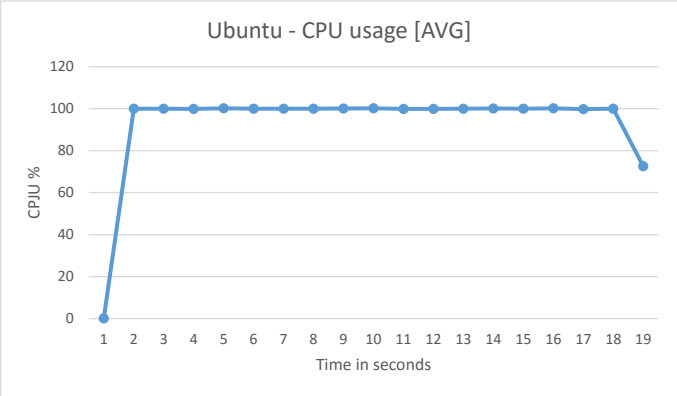

Figure 6.21: CPU usage Ubuntu VM

The CPU time spent by the Ubuntu VM during the experiments was 1900 ticks. with a variance of about 30 ticks in both lower and higher end of the spectrum. Figure 6.22 shows the CPU time spent during each run of the experiment.

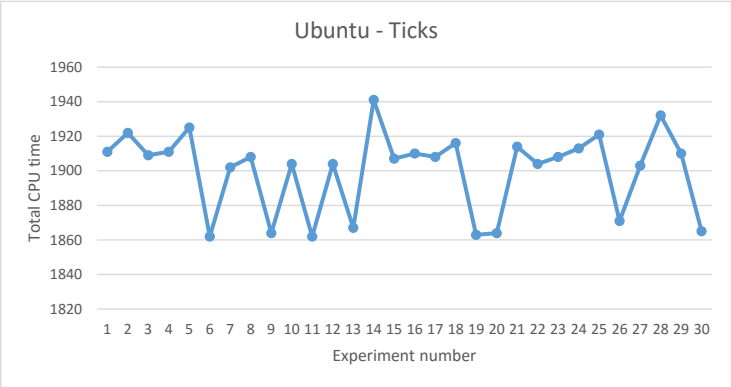

Figure 6.22: CPU time Ubuntu VM

6.2 Intel

6.2.1 IncludeOS

6.2.1.1 Scenario: TCP 500 req/s

In terms of response-time, IncludeOS over the course of each experiment used 0.2 ms to respond to each request. The amount of packets transferred between the server and the client was an average of 300002 packets.

For CPU usage IncludeOS used an average of 30.8% CPU for the experiment runs. Figure 6.23 shows the average CPU usage for all the experiments. For each experiment the CPU stayed mostly around 30% usage where it varied at most 1%. The average RAM utilization on the server where 0.91% which translates to 124 MB on average. However, as mentioned earlier, the TCP service of IncludeOS has a memory leak and the memory consumption keeps steadily increasing from an average of 0.022% to 0.15% or 29.7 MB to 204.5 MB

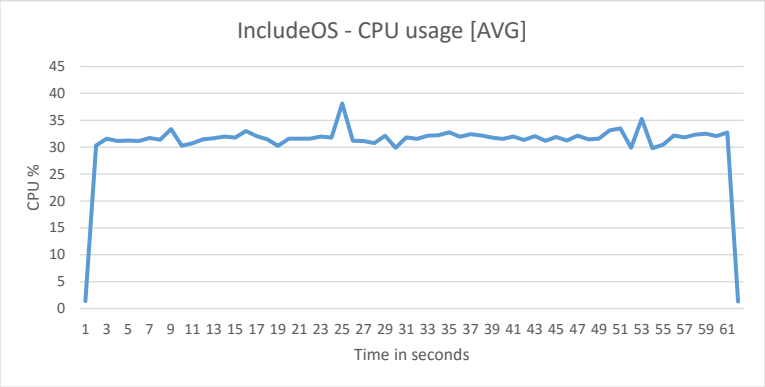

Figure 6.23: CPU usage IncludeOS

The CPU time spent by IncludeOS was 1910 ticks for all experiments, figure 6.24 shows the total amount of ticks spent for each experiment.

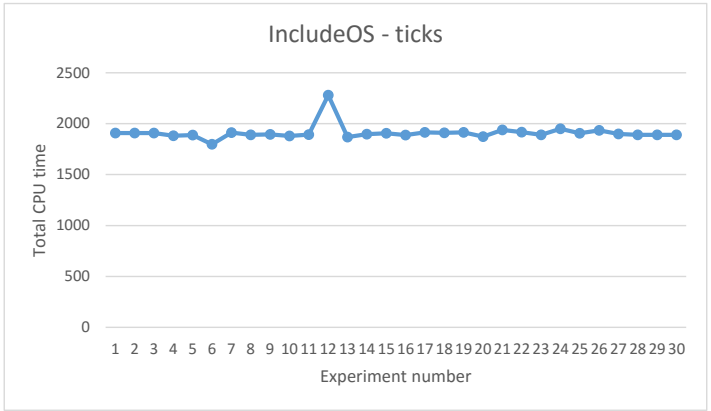

Figure 6.24: CPU time IncludeOS

Here it can be observed that the variance between the least and most amount of ticks is close to a 100 ticks of CPU time and is consistent with the usage of CPU.

6.2.1.2 Scenario: TCP 1000 req/s

The average response-time for the experiments were 0.23 ms, where there are two instances where the response-time was 400% and a little higher, at 0.6 ms and 0.7 ms. The average amount of packets sent between the server and the client during the experiments was 600002.

In terms of CPU usage was 60% over the course of the experiments. there was however some variance where two of the experiments used as much as 75% average CPU during the run. The RAM usage for the experiments was an average of 0.15% or 212 MB, staying consistent for each run and leaking. Starting from an average of 0.023% up to 0.28% or 31.5 MB to 380MB when the experiments completed. Figure 6.25 shows the average usage of the CPU throughout the experiments.

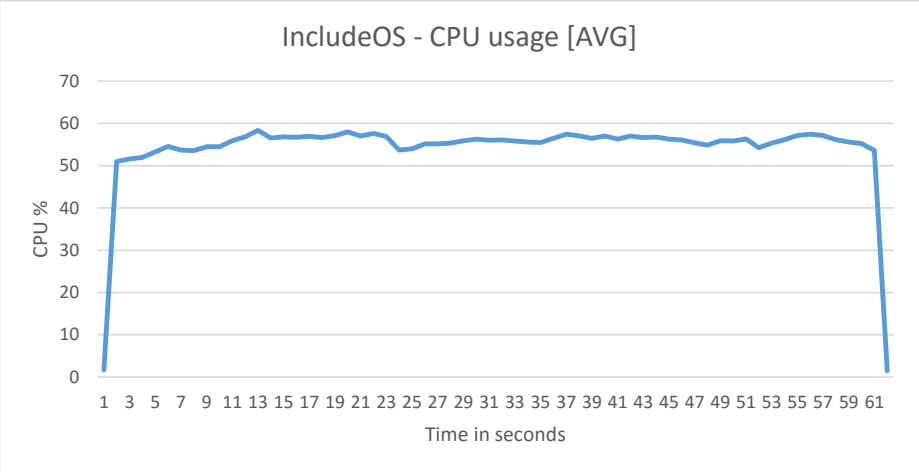

Figure 6.25: CPU usage IncludeOS

The average total amount of CPU time used by IncludeOS over the experiments was 3346 ticks. Figure 6.26 shows the CPU time spent during each run of the experiment, as seen there is some variance consisten with the CPU usage.

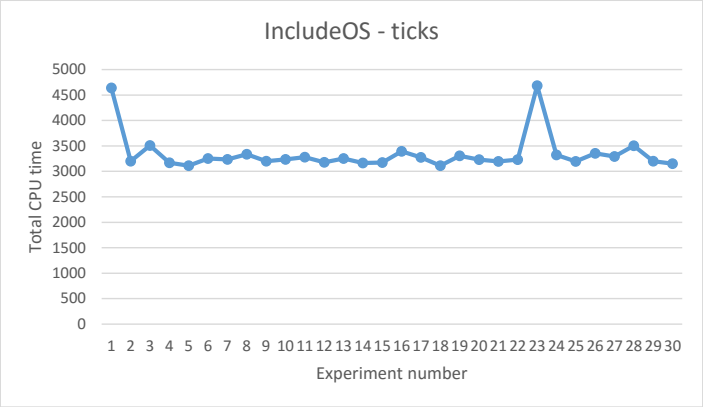

Figure 6.26: CPU time IncludeOS

6.2.1.3 Scenario: UDP

The average time used to receive and send the packets over the experiments was recorded at 6.4 seconds, the time for each experiment is shown by figure 6.27.

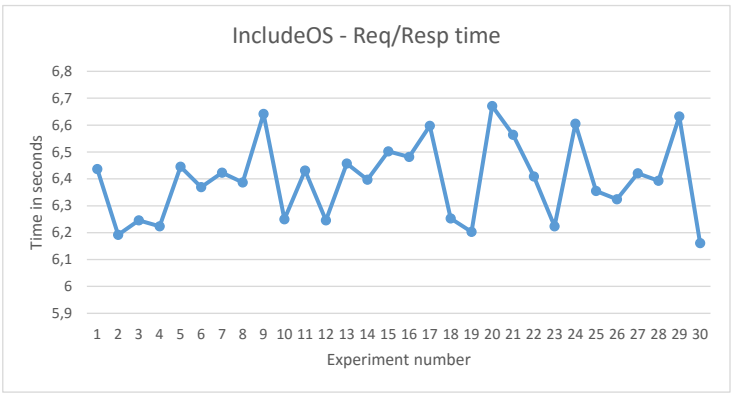

Figure 6.27: Response-time IncludeOS

The average usage of the CPU over the experiments was recorded at 68.5% where there are cases when the CPU usage is as high as 75%, the lowest recorded CPU usage was 63.75%. Figure 6.28 shows the average CPU usage for each run of the experiment. The average RAM usage over the experiments was recorded at 0.023% which is 31 MB.

The average total CPU time spent by IncludeOS over the experiments was recorded at 610 CPU ticks. The variance here is around 30 ticks between the experiments, figure 6.29 shows the total CPU time used for each of the runs of the experiment.

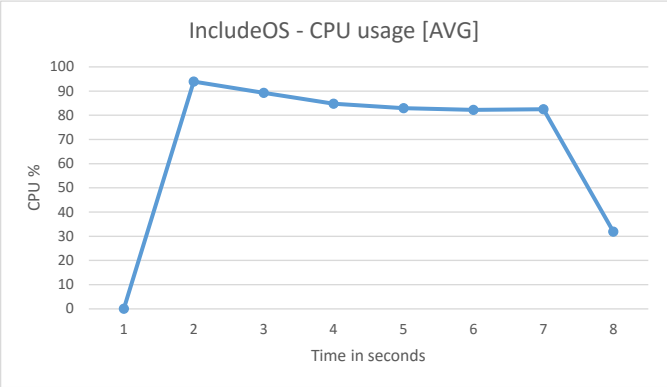

Figure 6.28: CPU usage IncludeOS

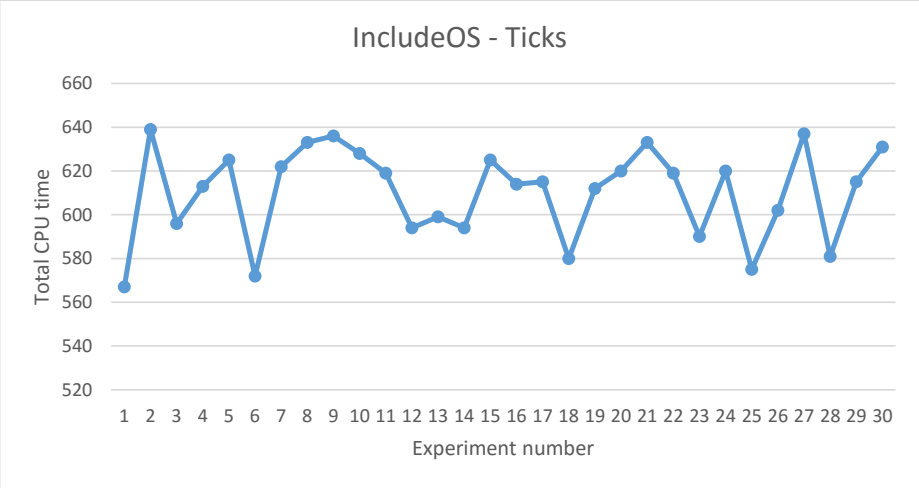

Figure 6.29: CPU time IncludeOS

6.2.2 Docker

6.2.2.1 Scenario: TCP 500 req/s

The average response-time of Docker for every run of the experiment was recorded at 0.1 ms. The average amount of packets that were transferred between the host and the client was 301613 packets.

In terms of CPU usage, Dockers consumption over the course of the experiments was recorded at 6.2% with a variance of up to 0.6%. Figure 6.30 shows the average CPU usage for all the experiments. The average amount of RAM used for the experiments was recorded at 0.050% or 60 MB, there were two cases where the RAM usage went as low as 0.04% or 55 MB and as high as 0.053% which is 68 MB.

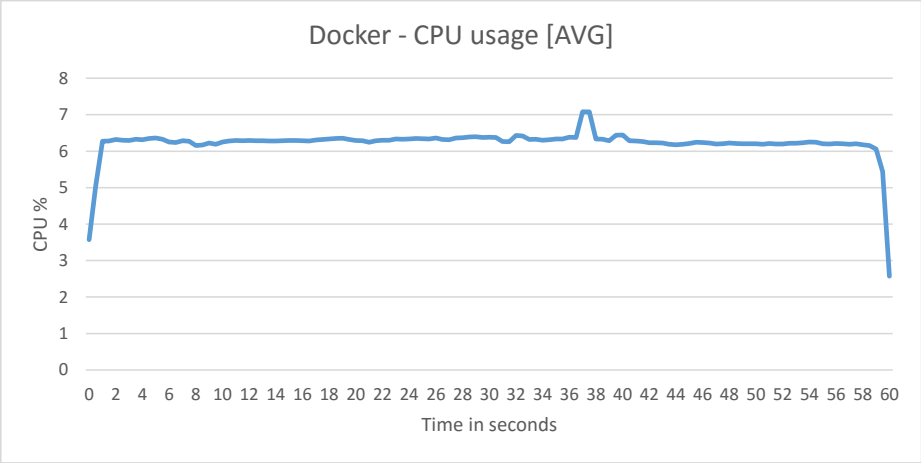

Figure 6.30: CPU usage Docker

The amount of CPU time spent by Docker over the experiments is an average of 382 ticks. There is some variance here as indicated by the CPU usage as well, where it was 100 more than the recorded average, there was also a run where it was almsot 100 less that the average as well. Figure 6.31 shows the total amount of ticks for each experiment.

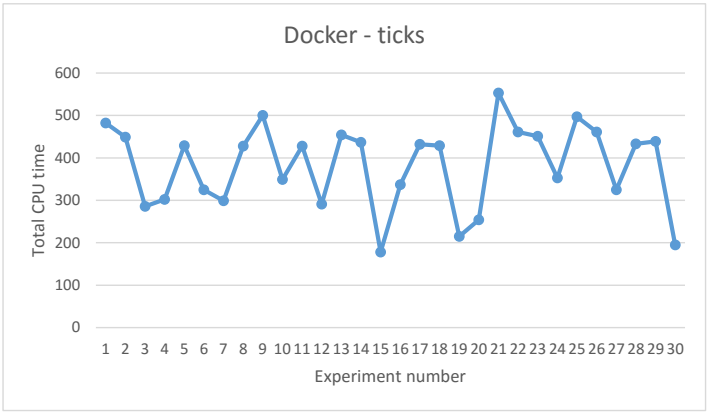

Figure 6.31: CPU time Docker

6.2.2.2 Scenario: TCP 1000 req/s

The average response-time for Docker was recorded at 0.1 ms with no variance. The amount of packets transferred between the server and the client was an average of 606976 packets.

In terms of CPU usage, the average for Docker was recorded at 11.8%, the variance is at most 1.3%. Figure 6.32 shows the average CPU usage for the experiments conducted. The RAM usage average was recorded at 0.044% which translates to 59 MB. There were some variance in the

experiments where the usage was between 0.04% and 0.05% or between 54 MB and 68 MB at the lowest and highest respectively.

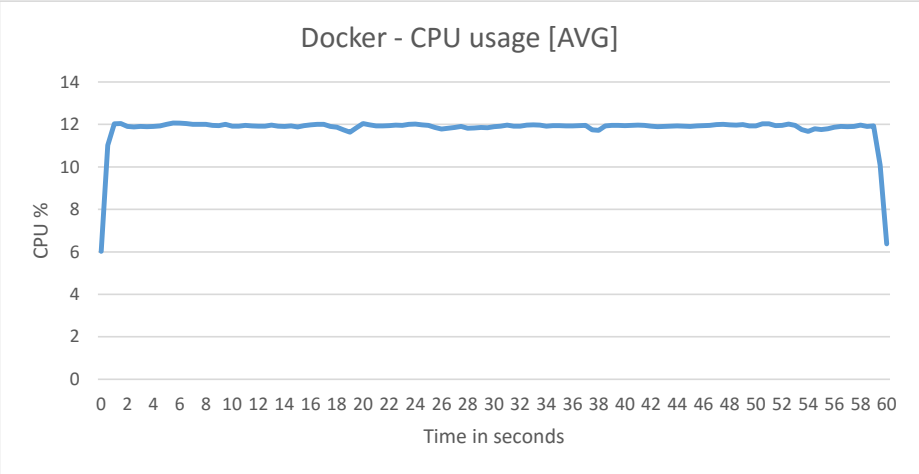

Figure 6.32: CPU usage Docker

The average total amount of CPU time spent by Docker through the experiments was recorded at 647 CPU ticks. The variance between the runs is about 100 ticks, the total amount of ticks for each run is showed in figure 6.33.

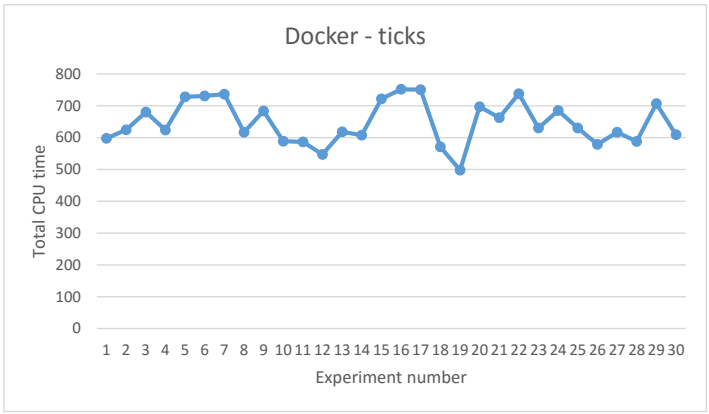

Figure 6.33: CPU time Docker

6.2.2.3 Scenario: UDP

The average time it took for Docker to send and receive the packets was 6.7 seconds. there is a variance of about 1 ms between the runs. Figure 6.34 shows the time spent on each run of the experiment by Docker.

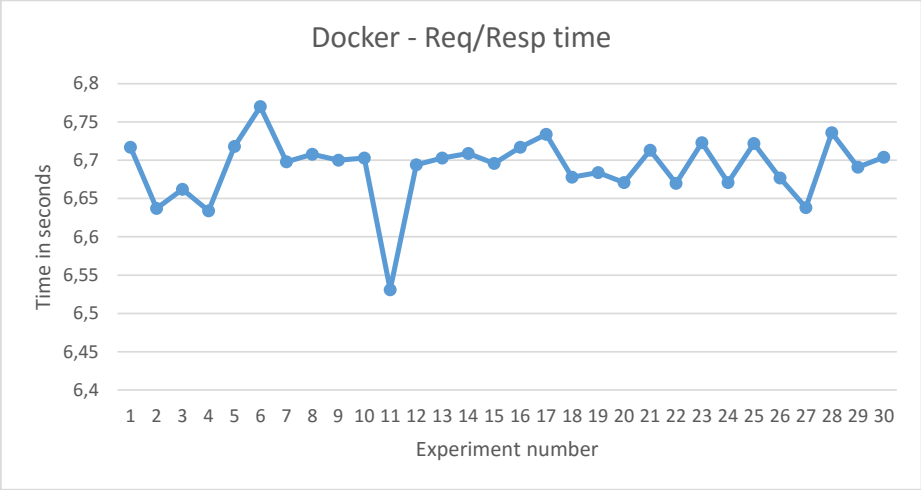

Figure 6.34: Response-time Docker

The amount of average CPU usage by Docker during the experiments was recorded at 21.4% with a variance of about 1.5% towards both ends of the spectrum. Figure 6.35 shows the CPU usage for each run of the experiment. The amount of average RAM utilization of Docker was recorded at 0.035% which is equal to 48 MB, this stayed constant throughout all the runs.

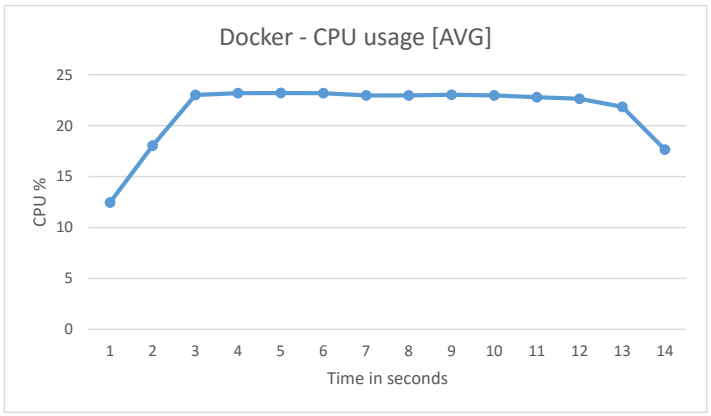

Figure 6.35: CPU usage Docker

In terms of CPU time, Docker spent an average of 212 CPU ticks through the experiments, with a variance of about 20 ticks towards each end with the exception of one run where the difference from the average was 43 ticks. Figure 6.36 shows the amount of CPU time Docker used for each run of the experiment.

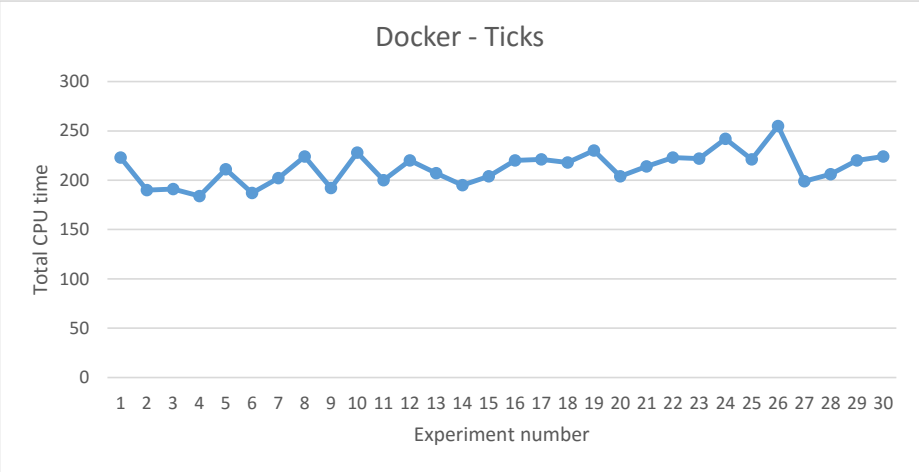

Figure 6.36: CPU time Docker

6.2.3 Ubuntu

6.2.3.1 Scenario: TCP 500 req/s

The average response-time of the Ubuntu VM through the experiments was at a stable 0.3 ms. The amount of packets transferred between the host and the client was recorded at an average of 300024 packets.

The average CPU usage of the Ubuntu VM through the experiments were recorded at 37,8% where it varies by around 1% between the runs. Figure 6.37 shows the average amount of CPU used for the experiments. The average RAM usage for the Ubuntu VM was recorded at 0.33% for all the runs, this translates to 446 MB.

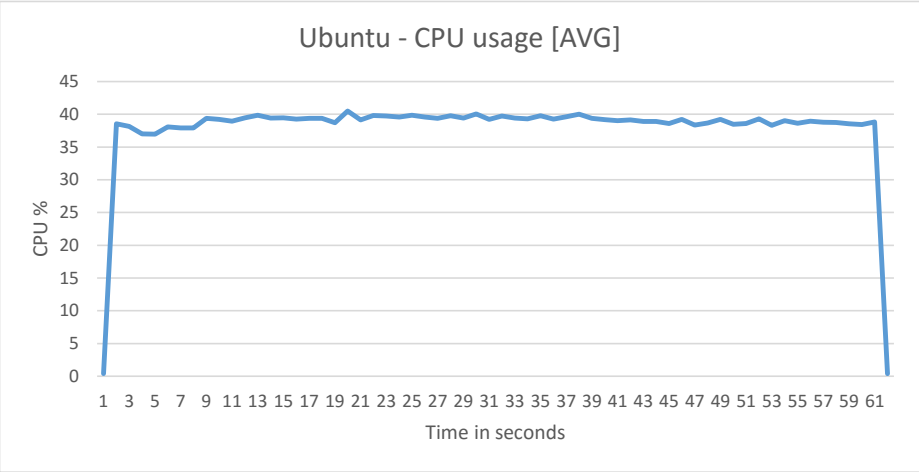

Figure 6.37: CPU usage Ubuntu VM

The amount of CPU time spent by the Ubuntu VM was recorded at a total average of 2344 ticks for the experiments. The variance here is small,

which is about 50 ticks at the highest. Figure 6.38 shows the CPU time spent for each run of the experiment.

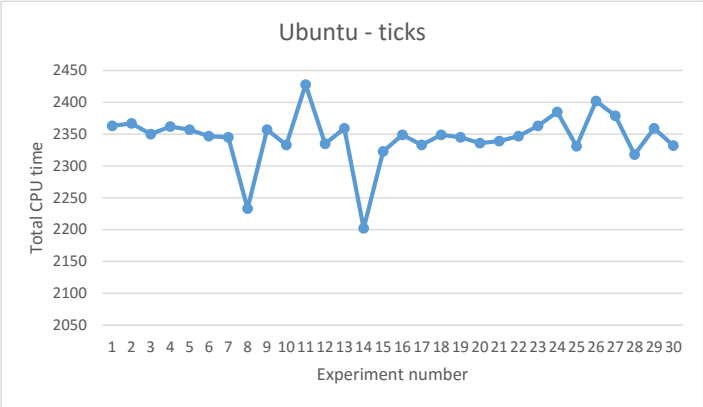

Figure 6.38: CPU time Ubuntu VM

6.2.3.2 Scenario: TCP 1000 req/s

The average response-time for the Ubuntu-VM was recorded at 0.2 ms for the experiments with no variance. The amount of packets transferred between the host and the client for all the experiments was recorded at an average of 600025 packets.

In terms of CPU usage, the Ubuntu VM used an average of 64% CPU during the experiments, the variance between the runs was recorded at about 1.5% both higher and lower than the average. Figure 6.39 shows the average CPU usage for the experiments. The amount of RAM used by the Ubuntu VM was recorded at an average of 0.22% which is 297 MB, the RAM usage for half of the runs is 0.19% and the other half 0.25% or 256.6 MB and 337.75 MB.

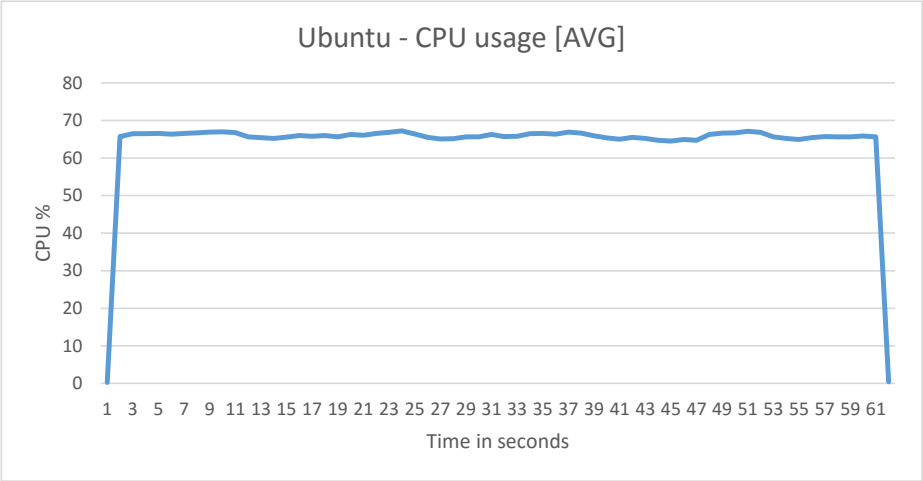

Figure 6.39: CPU usage Ubuntu VM

The amount of CPU time spent by the Ubuntu VM was recorded at an average of 3958 ticks for the experiments, the ticks have a variance at around 50 ticks. Figure 6.40 shows the amount of CPU time for the experiments.

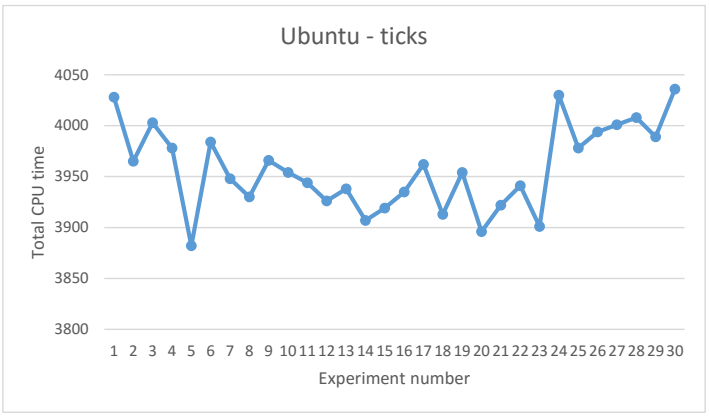

Figure 6.40: CPU time Ubuntu VM

6.2.3.3 Scenario: UDP

The time it took for the Ubuntu VM to repond to all the packets sent was an average of 10.6 seconds. Figure 6.41 shows the time for each of the experiments ran.

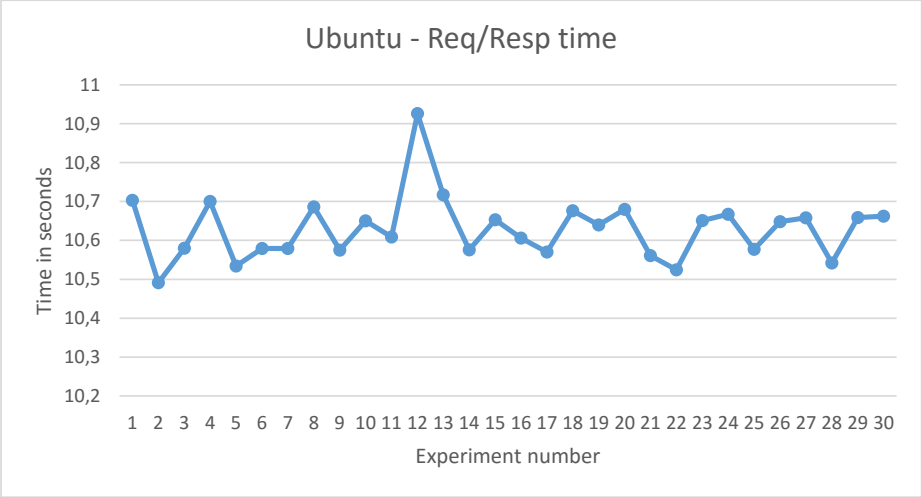

Figure 6.41: Response-time Ubuntu VM

The CPU usage of the Ubuntu VM was recorded with an average of 67.6% for the experiments. There is a variance of the CPU usage was recorded at 2% towards the lower end and 5% towards the higher end, figure 6.42. The RAM usage stayed constant at 0.15% which is equal to 202 MB, this was constant for all the experiments.

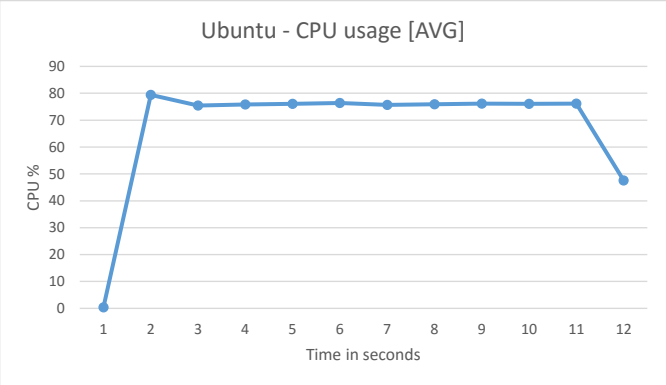

Figure 6.42: CPU usage Ubuntu VM

The average CPU time recorded for the Ubuntu VM was recorded at 857 ticks for the experiments, figure 6.43 shows the CPU time spent by the Ubuntu VM for the experiments.

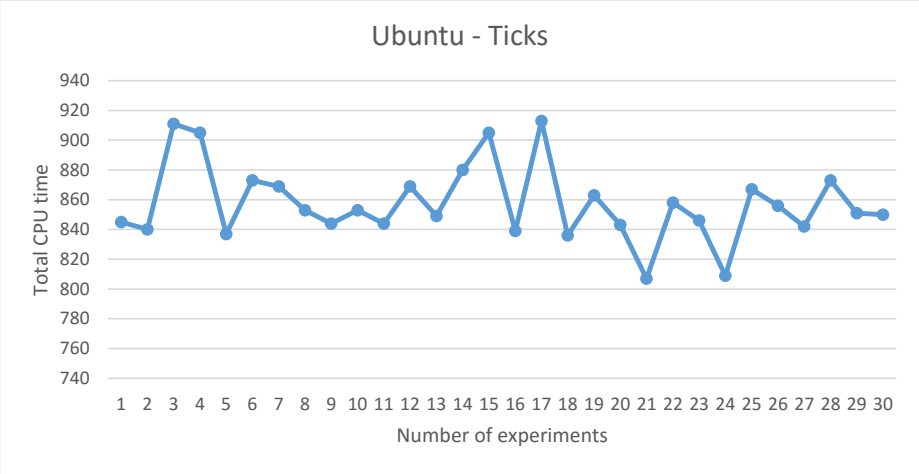

Figure 6.43: CPU time Ubuntu VM

6.3 Comparison

Here the different systems will be compared with regards to the criterion stated in the problem statement.

Looking at the data gathered the IncludeOS leaves a smaller resource footprint on the server than the Ubuntu VM by being 30% slower in terms of response time on the AMD server and by 50% in the first scenario and a fraction slower in the second scenario. In the third scenario, IncludeOS performs significantly better on the AMD server by 60% faster in terms of response. On the Intel server IncludeOS outperforms the Ubuntu VM in response-time for the third scenario by almost 36%. For Memory IncludeOS outperforms the Ubuntu VM by using 170 MB less memory performing the same task while on AMD 162 MB less for the same.

In terms of CPU cycles the Ubuntu VM and IncludeOS are almost on par with only a couple of 100 ticks being the difference between them. Docker however is superior to both systems by performing the same tasks in terms of CPU cycles with a difference in the thousands. The only metric where IncludeOS came out stronger than Docker is RAM performance for the third scenario.

When considering CPU usage in all cases for both servers Docker uses about almost 6 times less CPU in the first scenario and 6 times less CPU for the second scenario on AMD and. On Intel Docker consumes 30% of what IncludeOS uses for the first scenario and the 30% as well for the second scenario.

Chapter 7

Discussion

This chapter will look the evolution of the project, reflect on the approach, results and analysis and the project itself.

7.1 Throughout the project

The project was initially picked up as IncludeOS sounded like an interesting operating system designed for the cloud environment. Early throughout the project it was very hard to test the operating system for TCP as the drivers were not completely functional. It would not be able to handle more than 10 requests per second, and sometimes not even that. As there was no real point in testing the other systems before IncludeOS since it had to be ensured that the tests could be executed on it as well. The experiments were originally tested on multiple servers in order to locate what was wrong as IncludeOS did not function as mentioned. It was in the middle of April when the TCP service started working properly in terms of being able to handle load, not regarding the memory leak. It is then the experiments first started. There were another test which also were planned to be executed, namely the increasing load for TCP but was taken out as a result of short time. There were also no need to develop a data-extraction script either as the results were taken out of the files through UNIX string manipulation.

The results show that Docker performed the experiments which were defined in the approach being the testing of TCP and UDP better than IncludeOS on most points, and IncludeOS outperformed the Ubuntu VM for the same experiments. In retrospect there could have been additional or different tests and experiments developed in order to test more parts of the operating system. It is worth noting that security was not tested and is a major part of what unikernels are, e.g an IncludeOS instance designed to run a web-service will only run that web-service, an attacker could not change that.

In addition, without a filesystem (it is however in development, for maximum security one can run a version without it or not use it) makes for a host which even when compromised is very limited in terms of an

attacker as it cannot perform any other tasks than the service on it. Also, the amount of storage required to run IncludeOS is minuscule when compared to the other systems. Docker can save storage by being able to run multiple instances from the same image, however a base image is still required which already exceeds the size of IncludeOS and unikernels in general. General purpose operating systems will always consume a large amount of storage in comparison as they are built for multiple purposes.

The approach itself could possibly have been changed to include different types of tests which the author did not consider, however looking back it should have been done as the time it took for the proper testing with the current approach to begin constrained the quality of the end of this report which is the next point, analysis.

The analysis of the results in this project is not the authors best work, but was rather rushed as the remaining time set for this project was not adequate for an analysis that would be more in-depth and a discussion which went more into depth in regards to the entirety of the project.

7.2 Proposed improvements

The experiments were quite similar in nature, specially the two considering the first and the second scenario. What could have been done instead is implementing tests which instead would have tested the different parts of IncludeOS. E.g one experiment which was CPU intensive, another experiment which was RAM intensive, an experiment which would have solely tested NET I/O and possibly other features which is not mentioned here.

7.3 Alternative approaches

The author of this paper did not actually consider any alternative approaches due to attempting to make the current one work, however it should have been done due to the time it took for this approach to be set into action which was defined by both IncludeOS and the authors lack of proficiency in terms of C-programming.

Chapter 8

Conclusion

This chapter will attempt to conclude based on the results gathered through the experiments which was analyzed.

The problem statement as defined is:

On both an Intel and an AMD based Ubuntu server compare the performance and resource utilization of IncludeOS, Ubuntu VM and Docker

- when providing a TCP based web-service.
- in terms of network throughput performance with UDP.

The goal of this project was to find an answer to how the different systems perform in comparison to each other with a stronger focus on IncludeOS.

The experiments were designed to test how IncludeOS performed in comparison to container and VM. Being under development still during the project added a limitation as to what could be tested. There were developed three different experiments which were to test different systems performance while the web-service they where hosting were put under load, the first two experiments were the same in nature with different loads. The third experiment tested network throughput and performance during the experiment, in addition an experiment was created in place of the first experiment, it was however taken out as time did not allow for the experiment to gather adequate results.

Based on the results gathered currently in terms of raw performance on the server, Docker performs better than IncludeOS, while IncludeOS performs better than an Ubuntu VM. Docker is able to run at near bare-metal speeds in terms of using the CPU, while KVM should enable the same. However even with that Docker ended up consuming less CPU, less RAM and responded in an almost equal manner to IncludeOS. IncludeOS was less resource demaning than the Ubuntu VM. However IncludeOS currently was the most unstable while conducting the experiments (but not yet completely developed).

Bibliography

[1] Alfred Bratterud. *Home - IncludeOS Wiki*. Nov. 2015. URL: https://github.com/hioa-cs/IncludeOS/wiki (visited on 03/14/2016).

[2] Alfred Bratterud et al. "IncludeOS: A minimal, resource efficient unikernel for cloud services." In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2015, pp. 250–257.

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

[4] Sarah Conway. *Why Unikernels Can Improve Internet Security*. Apr. 2015. URL: https://blog.xenproject.org/2015/04/23/why-unikernels-can-improve-internet-security/ (visited on 02/03/2016).

[5] C. Dekker. *Installation/SystemRequirements*. Oct. 2014. URL: https://help.ubuntu.com/community/Installation/SystemRequirements (visited on 02/03/2016).

[6] Wes Felter et al. "An Updated Performance Comparison of Virtual Machines and Linux Containers." In: *technology* 25 (July 2014), p. 12. URL: http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195dd819c85257d2300681e7b/%5C$file/rc25482.pdf.

[7] Solomon Hykes et al. *What is Docker?* URL: https://www.docker.com/what-docker (visited on 03/14/2016).

[8] J. Fredrik Karlsson and Bahram Moshfegh. "Investigation of indoor climate and power usage in a data center." In: *Energy and Buildings* 37.10 (2005), pp. 1075–1083. ISSN: 0378-7788. DOI: http://dx.doi.org/10.1016/j.enbuild.2004.12.014. URL: http://www.sciencedirect.com/science/article/pii/S0378778805000150.

[9] Evangelos Kotsovinos. "Virtualization: Blessing or Curse?" In: *Commun. ACM* 54.1 (Jan. 2011), pp. 61–65. ISSN: 0001-0782. DOI: 10.1145/1866739.1866754. URL: http://doi.acm.org/10.1145/1866739.1866754.

[10] Anil Madhavapeddy and David J. Scott. "Unikernels: Rise of the Virtual Library Operating System." In: *Queue* 11.11 (Dec. 2013), 30:30–30:44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. URL: http://doi.acm.org/10.1145/2557963.2566628.

[11] Anil Madhavapeddy et al. “Unikernels: Library Operating Systems for the Cloud.” In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: http://doi.acm.org/10.1145/2499368.2451167 (visited on 02/16/2016).

[12] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment.” In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[13] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment.” In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[14] Mark P Mills. “The cloud begins with coal.” In: *Digital Power Group. Online at: http://www. tech-pundit. com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf* (2013).

[15] Mathijs Jeroen Scheepers. “Virtualization and containerization of application infrastructure: A comparison.” In: *21st Twente Student Conference on IT*. 2014, pp. 1–7.

[16] Dan Sullivan. *IaaS Providers List: Comparison And Guide*. Feb. 2014. URL: http://www.tomsitpro.com/articles/iaas-providers,1-1560.html (visited on 04/03/2016).

Appendices

Appendix A

Scripts and programs

Retrieved from an internal server server at Oslo University College[1]

Listing A.1: udpclientSendFastFork.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/types.h>
6  #include <sys/socket.h>
7  #include <netinet/in.h>
8  #include <netdb.h>
9  #include <unistd.h>
10
11 #define BUFSIZE 1024
12 #define SENDSIZE 20
13
14 /*
15  *  * error - wrapper for perror
16  *  */
17 void error(char *msg)
18 {
19    perror(msg);
20    exit(0);
21 }
22
23
24 int main(int argc, char **argv)
25 {
26
27    int sockfd, portno, n,i,max,microSeconds,packets;
28    int serverlen;
29    struct sockaddr_in serveraddr;
30    struct hostent *server;
31    char *hostname;
32    char buf[SENDSIZE];
33    char reply[BUFSIZE];
34
```

[1]Made by Hårek Haugerud

```
35    /* check command line arguments */
36    if (argc != 5)
37        {
38
39         fprintf(stderr,"usage: %s <hostname> <port> <max> <
              microSeconds>\n", argv[0]);
40         exit(0);
41      }
42
43    hostname = argv[1];
44    portno = atoi(argv[2]);
45    max = atoi(argv[3]);
46    microSeconds = atoi(argv[4]);
47    packets = 0;
48
49    /* socket: create the socket */
50    sockfd = socket(AF_INET, SOCK_DGRAM, 0);
51    if (sockfd < 0)
52      error("ERROR opening socket");
53
54    /* gethostbyname: get the server's DNS entry */
55    server = gethostbyname(hostname);
56    if (server == NULL)
57      {
58         fprintf(stderr,"ERROR, no such host as %s\n", hostname);
59         exit(0);
60      }
61
62    /* build the server's Internet address */
63    bzero((char *) &serveraddr, sizeof(serveraddr));
64    serveraddr.sin_family = AF_INET;
65    bcopy((char *)server->h_addr,
66          (char *)&serveraddr.sin_addr.s_addr, server->h_length);
67    serveraddr.sin_port = htons(portno);
68
69    /* send the message to the server */
70    serverlen = sizeof(serveraddr);
71
72    if(!fork())
73      {
74         for(i=0;i<max;i++)
75           {
76              sprintf(buf,"%d",i);
77              usleep(microSeconds);
78              n = sendto(sockfd, buf, strlen(buf)+1, 0, (struct
                    sockaddr *)&serveraddr, serverlen);
79           }
80         sprintf(buf,"LAST");
81         usleep(microSeconds);
82         n = sendto(sockfd, buf, strlen(buf)+1, 0, (struct sockaddr *)
                &serveraddr, serverlen);
83      }
84    else
85        {
```

```
86          while(recvfrom(sockfd, reply, BUFSIZE, 0, (struct sockaddr *)
                &serveraddr, &serverlen))
87            {
88              //printf("Received '%s'\n",reply);
89               if(strcmp(reply,"LAST") == 0)
90                 {
91                    printf("Received LAST, %d packets received\n",
                          packets);
92                    return 0;
93                 }
94               packets++;
95            }
96       }
97     return 0;
98  }
```

Retrieved from an internal server at Oslo University College[2]

Listing A.2: udpserver.c

```
1  #include <sys/socket.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <strings.h>
5  #define PORT 8081
6
7  int main(int argc, char**argv)
8  {
9     int sockfd,n;
10    struct sockaddr_in servaddr,cliaddr;
11    socklen_t len;
12    char mesg[1400];
13
14    sockfd=socket(AF_INET,SOCK_DGRAM,0);
15
16    bzero(&servaddr,sizeof(servaddr));
17    servaddr.sin_family = AF_INET;
18    servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
19    servaddr.sin_port=htons(PORT);
20    bind(sockfd,(struct sockaddr *)&servaddr,sizeof(servaddr));
21
22    for (;;)
23      {
24         len = sizeof(cliaddr);
25         n = recvfrom(sockfd,mesg,1400,0,(struct sockaddr *)&cliaddr,&
               len);
26         sendto(sockfd,mesg,n,0,(struct sockaddr *)&cliaddr,sizeof(
               cliaddr));
27         /**
28            mesg[n] = 0;
29            printf("%s",mesg);
30         */
```

[2]Made by Hårek Haugerud

```
31      }
32  }
```

Retrieved and modified from thegeekstuff[3]

Listing A.3: server.cpp

```
1  #include <sstream>
2  #include <stdlib.h>
3  #include <sys/socket.h>
4  #include <netinet/in.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <sys/types.h>
8  #include <string>
9  #include <unistd.h>
10 #include <stdio.h>
11
12 int getNumberOfA(std::string request) {
13         std::string function;
14         for(int i = 5;;i++) {
15                 if (isspace(request.at(i))) {
16                         break;
17                 }
18         function += request.at(i);
19         }
20         return atoi(function.c_str());
21 }
22
23 int main(int argc, char *argv[])
24 {
25    int serverPort = 5000;
26    int listenfd = 0, connfd = 0;
27    struct sockaddr_in serv_addr;
28
29    //char sendBuff[1000000];
30    std::string sendBuff;
31    char receiveBuff[1025];
32    std::string http;
33
34    listenfd = socket(AF_INET, SOCK_STREAM, 0);
35
36    serv_addr.sin_family = AF_INET;
37    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
38    serv_addr.sin_port = htons(serverPort);
39
40    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
41
42    listen(listenfd, 10);
43
44    sendBuff.append(1000u, 'A');
```

[3]http://www.thegeekstuff.com/2011/12/c-socket-programming/ Modified to accordance by Hårek Haugerud

```
45
46    while(1)
47        {
48        int i,nr;
49        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
50        recv(connfd,receiveBuff,sizeof(receiveBuff),0);
51        //nr = getNumberOfA(receiveBuff);
52
53        //for(i=0; i<nr; i++)
54        //{
55        //sendBuff.append(1000u, 'A');// = 'A';
56        //}
57        //sendBuff[nr] = '\0'; /* Sets end of buffer */
58        //sendBuff.append(1u, '\0');
59        size_t buff = sendBuff.size();
60        std::ostringstream ostr;
61        ostr << buff;
62        std::string sbuff = ostr.str();
63
64        http = "HTTP/1.1 200 OK \n "     \
65        "Date: Mon, 01 Jan 1970 00:00:01 GMT \n"  \
66        "Server: IncludeOS prototype 4.0 \n"    \
67        "Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT \n" \
68        "Content-Type: text/html; charset=UTF-8 \n" \
69        "Content-Length: 1000 \n" \
70        "Accept-Ranges: bytes\n"       \
71        "Connection: close\n\n";
72
73        http += sendBuff;
74        size_t http_size = http.size();
75        write(connfd, http.data(), http_size);
76        //printf("Length sent: %zu\n",buff);
77 //     sendBuff = "";
78        close(connfd);
79 //     usleep(1000000);
80     }
81
82 }
```

Listing A.4: dockerCollector.sh

```
1  #!/bin/bash
2
3  if [ "$#" != 3 ]
4  then
5          echo -e "Usage: $0 DURATION RATE FILE-EXTENSION"
6          exit 1
7  else
8          DURATION=$1
9          RATE=$2
10         MAX_CONN=$(($RATE*$DURATION))
11         EXTEN=$3
12 fi
13
```

```
14 DOCKER="172.17.0.2"
15
16 #CONT_NAME=$(docker ps -a | sed -n 3p | awk '{print $12}')
17 CID=$(docker inspect --format="{{.Id}}" romantic_lamarr)
18 DAEMON_PID=$(cat /var/run/docker.pid)
19
20 uCPU_start=$(head -n 1 /sys/fs/cgroup/cpuacct/docker/$CID/cpuacct.
       stat | awk '{print $2}')
21 kCPU_start=$(tail -n 1 /sys/fs/cgroup/cpuacct/docker/$CID/cpuacct.
       stat | awk '{print $2}')
22 DAEMON_uCPU_start=$(cat /proc/$DAEMON_PID/stat | awk '{print $14}')
23 DAEMON_kCPU_start=$(cat /proc/$DAEMON_PID/stat | awk '{print $15}')
24
25 $(date | awk '{print $4}' > docker_cpu_ram$EXTEN.txt)
26 echo -e "CONTAINER\t  CPU%\t\tMEM USAGE / LIMIT MEM%\t\tNET I/O\t\t\
       tBLOCK I/O" >> docker_cpu_ram$EXTEN.txt
27 ( docker stats romantic_lamarr >> tmpDockerCPURAM.txt & )
28 ( pidstat -h -r -u -v -p $DAEMON_PID 1 64 >> daemon_cpu_ram$EXTEN.
       txt & )
29 STAT_PID=$(ps aux | grep 'docker stats' | grep -v grep | awk '{print
        $2}')
30
31 sleep 1
32
33 ( httperf --server $DOCKER --rate $RATE --port 5000 --num-conn
       $MAX_CONN --timeout 2 --verbose >> docker_perf$EXTEN.txt )
34
35 uCPU_end=$(head -n 1 /sys/fs/cgroup/cpuacct/docker/$CID/cpuacct.stat
        | awk '{print $2}')
36 kCPU_end=$(tail -n 1 /sys/fs/cgroup/cpuacct/docker/$CID/cpuacct.stat
        | awk '{print $2}')
37
38 DAEMON_uCPU_end=$(cat /proc/$DAEMON_PID/stat | awk '{print $14}')
39 DAEMON_kCPU_end=$(cat /proc/$DAEMON_PID/stat | awk '{print $15}')
40
41 sleep 1
42 kill -9 $STAT_PID
43 endTIME=$(date | awk '{print $4}')
44
45 uCPU_TOTAL=$(($uCPU_end-$uCPU_start))
46 kCPU_TOTAL=$(($kCPU_end-$kCPU_start))
47
48 DAEMON_uCPU_TOTAL=$(($DAEMON_uCPU_end-$DAEMON_uCPU_start))
49 DAEMON_kCPU_TOTAL=$(($DAEMON_kCPU_end-$DAEMON_kCPU_start))
50
51 echo -e "USER TIME\t\tKERNEL TIME" > daemon_ticks$EXTEN.txt
52 rm tmpDockerCPURAM.txt
```

Listing A.5: collector.sh

```
1 #!/bin/bash
2
3 if [ "$#" != 3 ]
4 then
```

```
5         echo -e "Usage: $0 DURATION RATE FILE-EXTENSION"
6         exit 1
7 else
8         DURATION=$1
9         RATE=$2
10        MAX_CONN=$(($RATE*$DURATION))
11        EXTEN=$3
12 fi
13
14 INCLUDEOS="10.0.0.42"
15 DOCKER="172.17.0.2"
16 UBUNTU="192.168.122.3"
17
18 PID=$(pgrep qemu)
19
20 uCPU_start=$(cat /proc/$PID/stat | awk '{print $14}')
21 kCPU_start=$(cat /proc/$PID/stat | awk '{print $15}')
22 gCPU_start=$(cat /proc/$PID/stat | awk '{print $43}')
23
24 ( pidstat -h -r -u -v -p $PID 1 64 >> include_cpu_ram$EXTEN.txt & )
25
26 sleep 1
27
28 ( httperf --server $UBUNTU --port 5000 --rate $RATE --num-conn
      $MAX_CONN --timeout 2 --verbose >> include_perf$EXTEN.txt )
29
30 uCPU_end=$(cat /proc/$PID/stat | awk '{print $14}')
31 kCPU_end=$(cat /proc/$PID/stat | awk '{print $15}')
32 gCPU_end=$(cat /proc/$PID/stat | awk '{print $43}')
33
34 uCPU_TOTAL=$(($uCPU_end-$uCPU_start))
35 kCPU_TOTAL=$(($kCPU_end-$kCPU_start))
36 gCPU_TOTAL=$(($gCPU_end-$gCPU_start))
37
38 echo -e "USER TIME\t\tKERNEL TIME\t\tGUEST TIME" >
      include_ticks$EXTEN.txt
39 echo -e "$uCPU_TOTAL\t\t\t$kCPU_TOTAL\t\t\t$gCPU_TOTAL" >>
      include_ticks$EXTEN.txt
```