# EVALUATING FEASIBILITY OF CONTAINER VIRTUALIZATION FOR VIRTUAL NETWORK FUNCTIONS

(SANAL AĞ İŞLEVLERI IÇIN KONTEYNIR SANALLAŞTIRMANIN UYGUNLUĞUNUN İNCELENMESI)

by

**UĞURCAN ERGÜN, B.S.**

**Thesis**

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

**MASTER OF SCIENCE**

in

**COMPUTER ENGINEERING**

in the

**INSTITUTE OF SCIENCE AND ENGINEERING**

of

**GALATASARAY UNIVERSITY**

**JUNE 2018**

Approval of the thesis:

**EVALUATING FEASIBILITY OF CONTAINER VIRTUALIZATION FOR VIRTUAL NETWORK FUNCTIONS**

submitted by **UĞURCAN ERGÜN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Galatasaray University** by,

**Examining Committee Members:**

| | |
|---|---|
| Assist. Prof. Dr. B. ATAY ÖZGÖVDE<br>Supervisor, **Computer Engineering Department, GSU** | ______________ |
| Assoc. Prof. Dr. TOLGA OVATMAN<br>**Computer Engineering Department, ITU** | ______________ |
| Assist. Prof. Dr. TEOMAN NASKALI<br>**Computer Engineering Department, GSU** | ______________ |

**Date:** ______________

# ACKNOWLEDGEMENTS

First, I would like to thank my thesis advisor Atay ÖZGÖVDE for guiding me through the whole process. His knowledge and experience was invaluable. I am grateful that he treated me with utmost care and consideration.

I also want to thank Professor Cem Ersoy from Boğaziçi University for letting me to use their Network Laboratory and also treating me as one of their students.

Finally, I want thank my family and friends for their constant support and encouragements.

# TABLE OF CONTENTS

# LIST OF ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface |
| AUFS | Advanced Multi-layered Unification File System |
| BGP | Border Gateway Protocol |
| CAPEX | Capital Expenditures |
| CDN | Content Delivery Network |
| CNI | Container Network Interface |
| CORD | Central Office Re-architected as a Datacenter |
| CPU | Central Processing Unit |
| CRI | Container Runtime Interface |
| ETSI | European Telecommunications Standards Institute |
| GiB | Gigibyte |
| HLC | Home Location Register |
| HSC | Home Subscriber Server |
| HTTP | Hypertext Transfer Protocol |
| JSON | Javascript Object Notation |
| I/O | Input and Output |
| IP | Internet Protocol |
| KVM | Kernel Virtual Machine |
| LXC | Linux Containers |
| MANO | Management and Orchestration |
| NAT | Network Address Translation |
| NFV | Networking Function Virtualization |
| OASIS | Organization for the Advancement of Structured Information Standards |
| OPEX | Operational Expenditures |
| OS | Operating System |
| REST | Representational State Transfer |
| RNC | Radio Network Controller |
| SDN | Software Defined Networking |
| TOSCA | Topology and Orchestration Specification for Cloud Applications |
| VNF | Virtual Network Function |
| VNFD | Virtual Network Function Definition |
| YAML | Yet Another Markup Language |

**LIST OF FIGURES**

# LIST OF TABLES

## ABSTRACT

In modern computing it has become common practice to run software on virtual environments and commodity hardware rather than specialized devices. But most of the network appliances in use are still runs on propriety hardware. There is a significant amount of work in the network function virtualization domain to actualize these network functions in a virtualized manner. In this work we conducted a series of performance tests to evaluate the compatibility of modern container based virtualization solutions for NFV. To accomplish this, we implemented a generic service on top of the prevalent container orchestration software Kubernetes and assessed its performance with numerous clients. Our conclusion is while container performance varies with different parameters such as resource limits and instance counts, a well adjusted container setup is compatible to use with virtual network functions excluding some use cases.

**Keywords :** Network Function Virtualization, Containers, Cloud Computing, Virtualization, Kubernetes

# ÖZET

Günümüzde yazılımları özelleşmiş donanımlar yerine standart donanımlarda sanallaştırma teknolojileri kullanarak çalıştırmak yaygın bir pratik haline gelmiştir. Ancak bilgisayar ağları altyapısının çoğu halen özelleşmiş ticari çözümler üzerinde çalışmaktadır. Ağ altyapısını sanallaştırmak üzerine litaratürde ağ işlevi sanallaştırma dalı altında pek çok çalışma bulunmaktadır. Biz bu çalışmada ağ işlevi sanallaştırma çözümlerin yakın zamanda yaygınlaşmaya başlayan farklı bir sanallaştırma teknolojisi olan işletim sistemi seviyesi sanallaştırma veya konteynır sanallaştırma ile uyumluluğunu değerlendirebilmek için bir takım testler yaptık. Bunun için önde gelen konteynır yönetim sistemlerinden Kubernetes üzerinde genel bir servis oluşturduk ve çeşitli miktarlarda istemciler kullanarak performanslarını inceledik. Vardığımız sonuç konteynırlarin performansları kaynak limitleri ve konteynır sayısı gibi parametrelere göre değişse de iyi ayarlanmış bir konteynır sisteminin belli durumlar dışında ağ işlevleriyle kullanıma uygun olduğu yönünde oldu.

**Anahtar Kelimeler :** Ağ İşlevi Sanallaştırma, Konteynırlar, Bulut Bilişim, Sanallaştırma, Kubernetes

# 1 INTRODUCTION

In recent years, there has been an explosion of Internet users and Internet based services such as social networking sites, online messaging, video streaming etc. From network operators perspective, this accounts to a huge increase in the network traffic while their earnings stayed stagnant. This is especially true for the telecom operators. They are starting to experience unprecedented levels of mobile data usage and network traffic. Statistics show that worldwide mobile device count and mobile Internet usage has already surpassed desktop computers. This is partially due to the advancements in processor and system on chip technologies that enabled mobile devices to have computing power that rivals their desktop counterparts.

Since mobile Internet usage is only expected to grow further, it is clear that telecom operators need to improve their infrastructures. However, due to the nature of their medium, telecom operators have specific requirements and it would be very costly to increase the number of cell towers and base stations. Operators and industry organizations, well aware of this fact, proposed a different architecture when designing the next generation (5G) mobile networks. 5G mobile networks are being designed with Internet connectivity in mind. They pursue to actualize dynamic, flexible networks that significantly improve mobile bandwidth and latency by integrating various new technologies.

One of the related new technologies in ths perspective is called Network Function Virtualization (NFV). They aim to implement the benefits of standard virtualization for vendor specific network appliances that usually comprises the back bone of telecom networks (Abdelwahab et al., n.d.).

Looking at the design of 5G networks, it can be inferred that telecom operators and industry organizations have an intent to converge telecom and cloud infrastructures. There is a recent trend among internet service companies to use a new and different way to develop and maintain software for cloud infrastructures. This method includes decoupling software into independent services also called microservices and running them inside a more lightweight virtualization solution called containers. This approach is often called cloud native method for application development.

In the effort of converging infrastructures, it would be useful to examine the new developments in the cloud computing domain. While this cloud native approach has clear operational and performance benefits in the cloud environments, telecom environments often have different use cases and requirements and it is not guaranteed that the same approaches works in the different environments.

Cloud infrastructures relies heavily on the virtualization technologies to provide on-demand computing resources. Standard hypervisor based virtualization uses a thick software layer to separate virtual machines from each other. Since this creates a considerable overhead, a new virtualization technology recently gained traction called container virtualization. This technology utilizes advanced kernel technologies to allow creation of virtual machines without the aforementioned seperation layer. This machines or containers can use computing resources far more effectively than their hypervisor counterparts at the cost of weaker isolation.

In this work our goal would be trying to answer the questions "Is network virtual functions compatible with container environments ?" and "Is it feasible to combine these two technologies ?". In a more specific sense our main approach would be evaluating the performance of the container environments. Since one of the most important difference between telecom and cloud environments is the stricter performance requirements. Another factor to consider would be how containers are affected from the co-location. Since the isolation layer is weaker between different containers, it is important that the container sharing the same host does not negatively impact one another even if themselves experiences heavy load.

In our examination of the literature, we found out that previous works are mostly focused on implementation of virtual network functions with traditional virtualization. Few works that implemented VNFs with operating system level virtualization were focused on different areas such as edge computing.

In (Chen et al., n.d.), Chen et al. implemented a NFV system using Openstack and its management and orchestration tool Tacker. They made several alterations to these software for better utilization when used with virtual network functions such as better placement of functions, implementing a custom agent for auto scaling etc. Their work can be considered as a good example for an NFV implementation. A similar implementation with Openstack and Tacker can be seen in (Irshad, 2018-02-12).

In (Haider, Md. Rezzakul, n.d.) Haider used TOSCA orchestration method with a container platform Kubernetes. While TOSCA is commonly used in network function virtualization, the focus of this work is cloud orchestration not NFV. But it shows TOSCA orchestration can be used with containers.

In (Imagane et al., 2018) Imagane et al. designed a system using service chaining, one of the important features of network function virtualization for implementing a multimedia service in an edge cluster. While this work may not be seen directly relevant to NFV, it can be considered as an implementation of a different use case.

In (Cziva, Jouet, White and Pezaros, 2015) Cziva et al. used software defined networking with containers for implementing virtual network functions. They developed a simple network function virtualization platform with using OpenFlow, OpenDayLight and Docker. They also later expanded their work for cloud environments in (Cziva, Jouet and Pezaros, 2015) and edge computing in (Cziva and Pezaros, 2017). Their work can be considered as a solid proof of concept for using containers with network function virtualization.

We examine the features and capabilities of the prominent container orchestration software Kubernetes then we implement an generic function and test its end to end network performance in a variety of different configurations. Our primary means for examining the network performance would be measuring latency. We design and implement experiments around the key factors that may effect network functions as we identified above such as co-location and container performance then observe how latency is impacted by these factors.

## 1.1 Thesis Outline

This thesis follows the structure that is described below.

Chapter 2 presents an introduction on the network function virtualization domain. The reference architecture, most of the works in the literature are based on and one of the implementations of this model is explored.

Chapter 3 presents the history and state of art of the container virtualization. Two prominent technologies Docker and Kubernetes is explained in a detailed manner.

Chapter 4 describes the experimentation phase. The experiment design and used methods is explained alongside with the results of experiments.

Chapter 5 presents the conclusion of this work and discusses the further research probabilities.

# 2 NETWORK FUNCTION VIRTUALIZATION

## 2.1 Introduction

Computers networks are still complex and hard to manage. This is mostly due to proprietary and tightly coupled nature of the most network devices. Since these types network appliances rarely designed with interoperability in mind. They often significantly increase the system complexity. For simplifying network management and adapting to the flexible requirements of the current virtualized environments a new networking method is developed called the software defined networking or SDN (Chayapathi et al., n.d.).

Software defined networking, pursues to reduce the complexity of traditional networks by implementing the lessons learned in virtualization to the networks. The main idea behind software defined networks is the separation of data and control planes which are usually tightly coupled in proprietary vendor appliances. The data plane is moved out of the network devices and they are used as simple forwarding devices that only uses rules that are imposed from a central management software instead of making any decisions about the network logic. The central management softwares that replaces the control planes in the network devices are called SDN controllers or network operating systems. For interoperability all communication between network devices and controllers are handled with a common interface regardless of the device's vendor (Li and Chen, n.d.).

Software defined networking is starting to establish itself as the standard way to manage contemporary networks. But there are lots of complementary or auxiliary network functions such as firewalls, CDNs, carrier grade NATs, broadband network gateways etc. which are still provided as proprietary, specialized middle-boxes. Usage of specialized hardware always significantly increases capital expenditures and operational expenditures for network operators. This is especially relevant for telecommunications companies since they have to build and maintain many central offices around the area they provide services. While data centers of cloud companies can use commodity hardware and virtualization they can significantly reduce CAPEX and OPEX. There is little to none commercial of the shelf alternatives exist for mobile network devices such as Home Location Register (HLC), Home Subscriber Server (HSC), RNC, NodeB etc. (Khalid et al., n.d.) Therefore any effort to virtualize these kinds of services would benefit telecommunications companies greatly. Ideally telecommunications companies would want to approximate their central offices to ordinary data centers if there would be a way to run their core services on commodity hardware. This model later would be called Central Office Rearchitectured as a Datacenter or CORD.

Network Function Virtualization is a new research domain to virtualize the aforementioned network functions which is mostly pioneered under the banner of European Telecommunications Standards Institute. European Telecommunications Standards Institute (*ETSI - Welcome to the World of Standards!*, n.d.) is a standards organization that is founded in 1988 at France and mainly works on information and communication technologies. While it is a organization for European Union its membership are not exclusive to companies and organizations from European Union member countries and has members from countries all around the world like Canada, China, Korea, Japan, Taiwan, United States etc.

Network function virtualization is first proposed at 'SDN and OpenFlow World Congress' in Germany 2012 and currently developed by Network Function Virtualization Industry Standards Group of ETSI. ETSI NFV ISG has several workgroups with different interests such as NFV IFV (Infrastructure), NFV MAN (Management and Orchestration), NFV SEC (Security) etc (Kurebayashi et al., n.d.).

While they are often thought together software defined networking and network function virtualization are independent technologies and they contribute different ideas to network research. But if these two technologies are used in a complementary way, computer networks can be utilized more effectively (Gray and Nadeau, n.d.).

## 2.2 ETSI NFV

The reference model (Eur, 2013) proposed by ETSI NFV industry standards group gained traction in both industry and research. While it is not a de-facto standard for network function virtualization, there are many technologies and implementations that are based on this model. As it can be seen in Figure 2.1 ETSI model depicts the architecture as core functional blocks and their interactions with each other. The main contribution of the ETSI NFV model is the block called Management and Orchestration or shortly MANO (Eur, 2014). This block is meant to enable integration of virtual network functions into modern cloud infrastructures.

### 2.2.1 Architecture

#### 2.2.1.1 Network Function Virtualization Infrastructure (NFVI)

Network function virtualization infrastructure block of the ETSI NFV model that provides the necessary environment for virtual network functions. It consists of three parts. The first is the physical hardware that provides the essential resources like processing power, storage and networking etc. They are assumed to be commercial off the self hardware. The second is the virtualization layer that enables virtual network functions to use underlying architecture. As long it supports the use required reference points ETSI model doesn't require any specific virtualization solution. The third one are the pooled virtual resources is then exposed for the use of upper levels.

#### 2.2.1.2 Virtualized Network Function (VNF)

A virtualized network function is a virtualization of the existing network functions that are often implemented as specialized hardware. VNFs are expected to function identically compared to their non virtualized counterparts. It is not necessary for a single VNF to be implemented in a single virtual machine. Implementation details may depend for the nature of the network function.

#### 2.2.1.3 Element Management System (EMS)

Element management system is a functional block that handles the management tasks for one or multiple VNFs. This management tasks include fault management, accounting, configuration (could be vendor specific configuration), security and performance

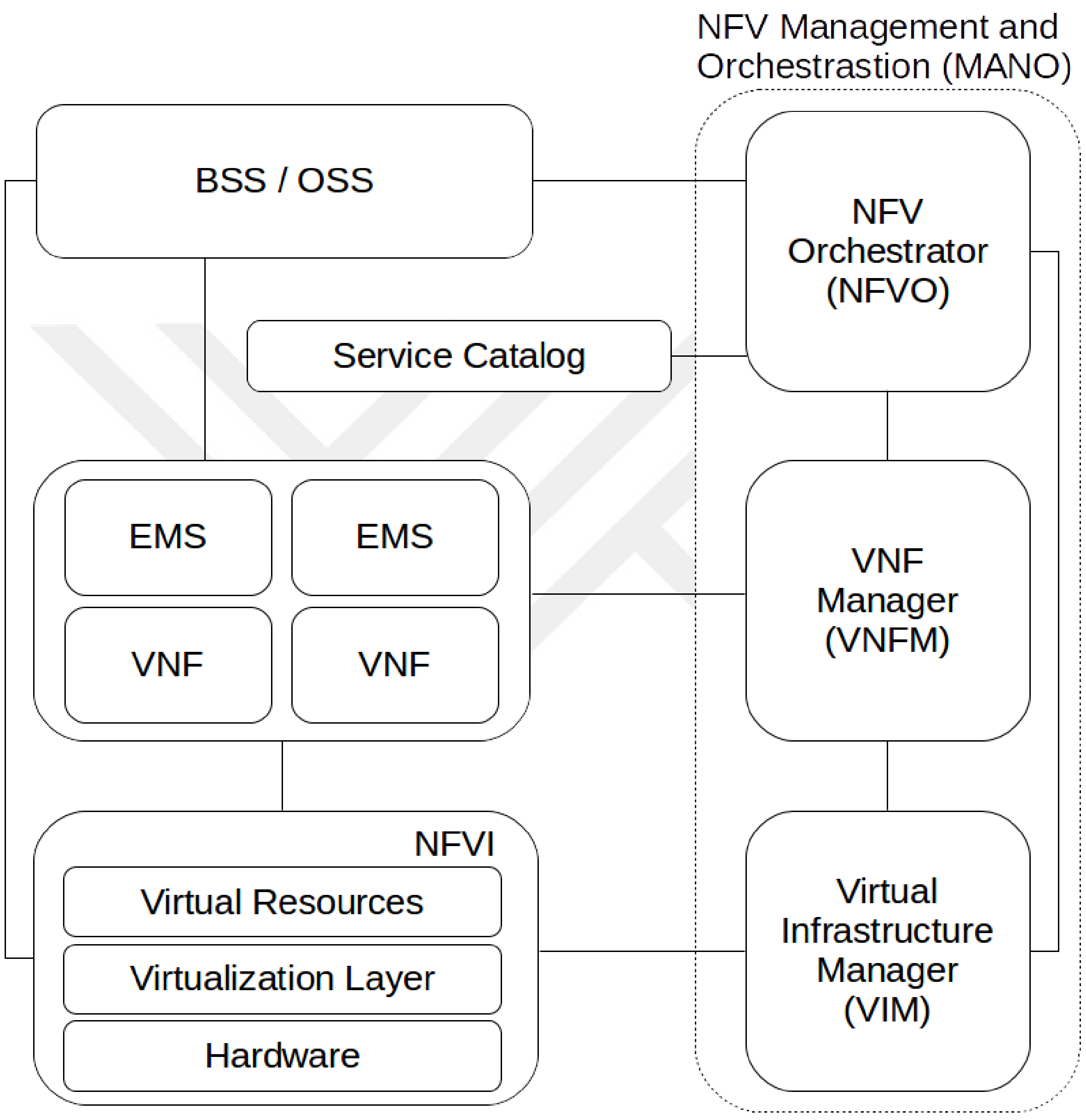

Figure 2.1: ETSI NFV Architecture.

evaluation. The EMS may or may not collaborate with the virtual network function manager.

### 2.2.1.4 Operations Support System / Business Support System (OSS/BSS)

Operations Support System/Business Support System block is reserved for the network operators. They can define any operation that are not explicitly described in the core ETSI architecture. It is expected to share data with other function blocks.

### 2.2.1.5 Virtualized infrastructure manager (VIM)

Virtualized infrastructure manager block is responsible from the management of the network function virtualization infrastructure. Mainly VIM has to orchestrate NFVI resources. For this VIM has to make an inventory of system resource, keep track of what resources available at any time, know how currently allocated resources are utilized, is there any apparent hardware failures etc. A NFV architecture can have multiple VIMs and there can be different VIMs for different resources in the NFVI. Openstack is a usual practical choice as virtualized infrastructure manager for the majority of the ETSI NFV implementations.

### 2.2.1.6 VNF Manager (VNFM)

VNF Manager block is the software that manages the life cycles of the virtual network functions. Life cycle of a VNF includes instantiation (mostly from a template and template generally includes configuration information), modification, termination, monitoring, measuring performance metrics, disaster recovery etc. VNFM generally implements VNF life cycle as a set of general commands that can be applied almost any VNF regardless of type and each VNF in the system is assigned to a VNFM. Likewise in VIM there is no limitation of how many VNFMs can exist together.

#### 2.2.1.7 NFV Orchestrator (NFVO)

NFV Orchestrator is responsible for both orchestrating NFVI resources over multiple VIMs and managing network services' life cycle. A network service is an abstraction for multiple virtual or physical network functions and any service chains or forwarding graphs that interconnects them. A network services' live-cycle is alike to a virtual network function and consists of instantiation (with coordination the VNFM), modification, termination, monitoring, measuring performance metrics, disaster recovery etc. Other duties of an includes keeping a catalog of existing network services and virtual network functions, on-boarding new services, managing VNFMs etc (Gonzalez et al., 2018).

### 2.2.2 Implementation : Openstack Tacker

In NFV MANO implementations Openstack (*OpenStack Open Source Cloud Computing Software*, n.d.) is one of the most commonly used VIMs. To have a complete MANO solution Openstack started a NFVO-NFVM project called Tacker. It is a spin-off of a Neutron project called ServiceVM started in 2014 to provide a unified interface for life-cycles of different appliances. While not successful as a Neutron project it evolved to implementing MANO in Openstack. Since 2015 Tacker project exists as an NFV MANO solution in Openstack (*Tacker - OpenStack*, n.d.).

Tacker has a modular architecture that allows using it use with various different kinds of existing Openstack services like Openstack orchestration service Heat and Openstack monitoring service Ceilometer. VNF definitions (VNFDs) are described as TOSCA templates and saved in a service catalog. Operators can also configure and monitor the VNFs by their requirements by writing custom management or monitoring drivers.

Tacker uses TOSCA YAML templates for its virtual network function definitions. TOSCA or Topology and Orchestration Specification for Cloud Applications is a format for defining applications that are needed to run on a cloud architecture (*OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC | OASIS*, n.d.). The specification is developed by an non-profit standards organization called Organization for the Advancement of Structured Information Standards (OASIS) (*OA-*

*SIS | Advancing open standards for the information society*, n.d.). A TOSCA template defines not only application that user wants to deploy but also its requirements and the infrastructure it runs on. The main method of achieving this feat is by defining relations between these units. An example TOSCA template can be found at APPENDIX A.

# 3 CONTAINER VIRTUALIZATION

## 3.1 Introduction

National Institute of Standards and Technology defines cloud computing as a model for enabling easy access to a pool of computing resources that requires minimal intervention from the service provider. With cloud services, any person or organization can buy computing resources from providers without the need of investing in computing infrastructure themselves.

One of the essential technologies for implementing cloud services is virtualization. In modern cloud environments the usually preferred method for virtualization is called hypervisor based virtualization. Xen and KVM can be called the two most common free and open source hypervisors. While Xen is used in biggest public cloud platform Amazon Web Services, KVM is used in Google Cloud Platform and Openstack.

The main goal for the hypervisors is to isolate virtual machines in physical host to a degree that they are no different than separate computers in a network. Virtual machines run a separate operating system called guest operating system and hypervisor mediates the system calls to the host operating system. This technique is very efficient at isolating virtual machines but it also introduces an overhead since there is an extra layer of abstraction. Previous studies show that while CPU and memory overhead is minimal, biggest overheads occur in I/O operations (Hwang et al., n.d.). There is also some operational overheads that comes from running an separate full-fledged guest OS. It also needs to be maintained and updated.

Recently there is another method of virtualization gaining traction in both research and the industry. It is called operating system level virtualization or more commonly container virtualization.With container virtualization virtual machines or containers have their own process and resource space but they use host machines kernel and resources directly. Without the need of complete isolation and an unaware guest OS container virtualization can achieve faster start up times, less resource consumption and smaller images (Joy, 2015). But since all containers use the same kernel, isolation between different containers is weaker.

The earliest attempt for operating system level virtualization can be traced to the UNIX chmod command. It is introduced in UNIX version 7 at year of 1979. By that time it could only provide file system isolation. But FreeBSD jails is the first mechanism that can truly be called a container in a modern sense. It was developed by Poul-Henning Kamp at 1998 and introduced in FreeBSD version 4 at 2000 (Kamp and Watson, n.d.). Jailed processed can't see other processes, can access only a specific part of their file systems and have their own IP addresses. At Linux side, OpenVZ project developed a specialized kernel for running containers at 2006 (Kolyshkin, 2006). Contrary to jails, OpenVZ is more akin to a virtual machines. OpenVZ containers have their own user structure, subject to configurable resource limits and can be migrated to separate physical machines without the need of shutting down (Bernstein, n.d.).

Despite both the concept and the technologies that make it possible were around for some time. Containers are only recently started to see wide spread adoption. This can be attributed to a few reasons. While Linux became one of the major platforms in the server domain there wasn't any container technologies available that can be run on an unmodified Linux kernel. Furthermore Linux kernel lacked some of the essential capabilities to implement a container system such as resource management for processes. Before cgroups was introduced in 2008 Linux kernel didn't have the capacity for controlling and limiting the resources of a process or group of processes. Using cgroups and earlier introduced kernel namespaces in the same year first native Linux container project LXC (*Linux Containers*, n.d.) was announced. While it didn't see much adopted by itself. It formed the basis for the most of the modern container technologies (Singh and Singh, 2016).

## 3.2 Docker

Docker is a container virtualization technology (*Docker*, n.d.) that is developed in 2013 by a software company called dotCloud. Among the modern container technologies Docker is the most widely used one and it would also be fair to say Docker project's success is partially responsible for the recent adoption of container virtualization technology. This might be attributed to Docker's different approach on how to prepare and use containers.

Virtual machines are used to be considered as virtual equivalents of physical dedicated servers and for some time containers are also thought as faster and more lightweight virtual machines (Adufu et al., 2015). But the main contribution of Docker is focusing into an application instead of a server. Docker provides tools for packaging an application and all of its dependencies together to run the application inside of a container. This model of packaging and deployment of applications using containers also found out to be a effective solution for an important problem at software development and maintenance. Because differences between development and production environments such as different operating systems, different versions of libraries and applications etc. is a common source for software bugs. Since they are being used and maintained by different people and there are no way any single person would be aware of all the differences. Packaging the application and dependencies with Docker and running it in containers prevents this problem. Because it guarantees the environment that runs in developer's machine and the production machine is the same (Boettiger, 2015).

Furthermore Docker proposes that each component of an application should be run in a different container thus encouraging developers to use a loosely coupled software architecture that is also called micro service architecture. According to micro service architectures, managing and updating individual components of applications are easier compared to managing a monolithic application which all of its components runs in same environment and problems in one component can affect the whole application.

### 3.2.1 Docker Images

As mentioned before packaging the applications and dependencies together is one of the core ideas behind Docker. This is achieved by using the Docker images. Contrary to virtual machines where user has to install a new operating system on a clean virtual disk or use pre-installed disk image. Docker provides the tools to easily create new images from scratch or from existing images. Docker also provides a central registry called Docker Hub where official and community created images can be found. New images are created by using a template format called Dockerfiles. To create an image user has to choose a base image and then has to describe what changes has to be made on this base image with provided commands. An example Dockerfile can be found at the APPENDIX B. Then using the Docker build command creates the intended image.

Docker images are declarative constructs. An existing Docker image can not be edited or changed. This way it is guaranteed to have the same environment every time an image is used. To make changes to an image a new image has to be created. Docker has a version control system for images and a new version of the image doesn't have to overwrite the old images. This way in case of errors and failures old versions can still be used. Since storing whole copies of all images would take too much disk space. Docker uses a different storage technologies for storing images.

Docker images are stored using a storage model called union file systems to minimize storing duplicate data. Using this model images are consisted of several ordered layers. Each layer has its own file system and if a file is present in multiple layers only the file on the top most layer is visible. When creating a new image Docker takes all the layers from the base image and adds new layer or layers according to the Dockerfile. If we make create a new version of a image only the changes are also written to a new layer. Normally layers are read only file systems but since container that can not write data to their disks would be unusable a writable layer is added when a container is created. This layer would be deleted when the container is removed until it is saved as a version of the image. While Docker can use different union file systems, by default AUFS is used.

### 3.2.2 Docker Architecture

Docker uses a rather simple server-client architecture for the creation and management of the containers. The main components can be summarized as the Docker engine, the component that manages images and containers, Docker client, the component that users and administrators can send their commands to Docker engine with and the Docker registry the component images that constitutes the core of the containers resides. Docker provides an central registry with official and community images free for use. This central registry is called the Docker Hub (*Docker Hub*, n.d.) .The communication between these components are achieved via REST APIs.

It is clear to see that the Docker engine is at the core of this architecture. It is easier to understand how the Docker Engine currently works if it is broken down to three sub components. The container runtime is the component that runs the containers themselves. The container engine provides the extra services that a container needs such as networking, storage, security etc. And The Docker services is the set of components that provides services that are crucial for Docker's internal operation such as the Docker API, image management, authentication etc.

Docker's commitment to the above mentioned micro service architecture can also be seen in the evolution of the Docker engine as it matures over time. Initially it was just a monolithic daemon. While it is convenient to have a centralized system in the short run. In time the shortcomings of this model started to appear and components started to become independent from the daemon. At first a plug-in model developed to manage the aforementioned extra services. Then the container runtime was modified to be able to run in a stand alone way. While the first container runtime for Docker was the LXC. It was already replaced by a runtime called libcontainer which is developed by Docker themselves. The new runtime that enabled libcontainer to be used without the docker daemon is called runC and it is managed by a library called containerd. With this change docker daemon became a component that is only connecting the API, service plugins and the containerd.

Current state of Docker architecture can be seen at the Figure 3.1

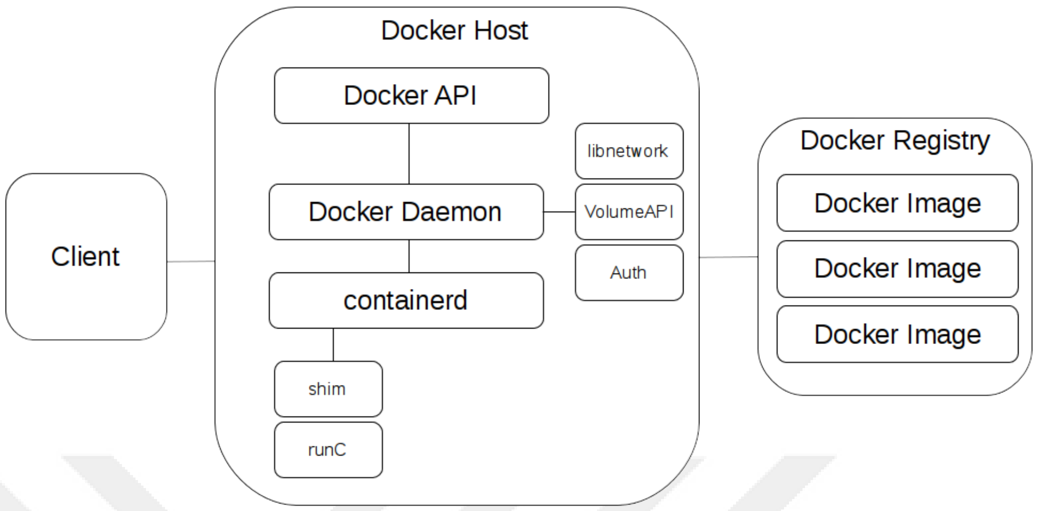

Figure 3.1: Docker Architecture.

## 3.3 Kubernetes

The one of most important topics about operating cloud environments is the question of scale. In a practical case it would be expected to run numerous amounts of containers in many hosts. Since cloud systems has to run with minimum human intervention, the obvious problem would be coordination of the containers in a automated manner. The problem domain can be defined as orchestration. For container orchestration the main issues would be how containers is distributed among hosts, how they communicate, what happens when the containers fail, what happens in case of on outage, how applications scale etc.

Kubernetes is a container orchestration software that is developed by Google in 2014 (*Kubernetes*, n.d.). Its design is inspired by Google's internal container orchestration systems Borg and Omega (Burns et al., 2016). Kubernetes is a free and open source software and its ownership is later transfered to a more neutral organization Cloud Native Computing Foundation (*Cloud Native Computing Foundation*, n.d.) that is run by Linux Foundation (*Linux Foundation*, n.d.). Kubernetes is written in Go programming language. It can be installed into physical servers, local virtual machines or public cloud services. Although Docker now has its own integrated container orchestration

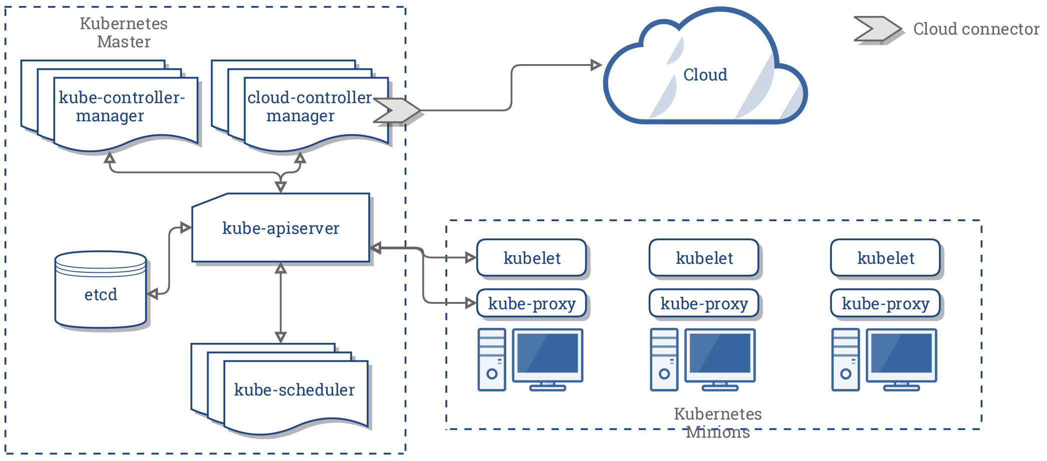

Figure 3.2: Kubernetes Architecture (*Kubernetes Architecture*, n.d.).

solution called Docker Swarm (*Swarm mode overview*, 2017), Kubernetes stays as the most widely used container orchestration software.

### 3.3.1 Kubernetes Architecture

Kubernetes implements a typical master slave architecture. These are named as master nodes and worker nodes. Nodes are only separated among themselves by which Kubernetes components they run and each role has to run a required set of components. For master nodes the required components are API server, etcd, scheduler and controller manager, for worker nodes the required components are kubelet and proxy. A representation of Kubernetes architecture can be seen at Figure 3.2

API server is the component that serves the Kubernetes API. The entire control plane of Kubernetes is built on the API and components doesn't have any other way to communicate with each other. API server is designed to be stateless and all the necessary data is stored on etcd. Etcd is a distributed key value store that is developed by CoreOS (*CoreOS*, n.d.). It serves as a central database for the Kubernetes cluster. Since etcd stores the whole cluster state, consistency and fault tolerance is critical. It uses a consensus algorithm called raft (Ongaro and Ousterhout, 2014) to keep track of all the changes in etcd nodes. Scheduler is responsible for assigning containers to appropriate worker nodes considering the resource requirements, hardware conditions and several other factors. Controller manager is a collection of software that watches the current cluster state and checks if it matches with the records in the API such as node availability, current endpoints etc. Kubelet is the main Kubernetes daemon in the worker nodes. Kubelet's main responsibilities are to manage containers and check both node's and container's current status and report it to the API. Proxy is responsible from the networking in the worker nodes. It listens the API and configures the node network accordingly so that containers can communicate without any problems.

Kubernetes used Docker as the main container runtime. To support different container runtimes kubelet has to be modified significantly. Since it would be quite difficult for each runtime vendor to make this modifications a common API is developed to easily integrate other runtimes into Kubernetes. This API is called the Container Runtime Interface (CRI). While this is a relatively new technology, introduced with Kubernetes version 1.5 there is some implementations using CRI such as CRI-O (*cri-o*, n.d.) which tries make Kubernetes compatible with Open Container Initiative runtimes and Rktlet a CRI implementation of the CoreOS runtime rkt (*rkt is a pod-native container engine for Linux. It is composable, secure, and built on standards*, 2017).

### 3.3.2 Workloads

For creating or accessing Kubernetes resources a developer first has to select an API client to work with. It can be the web interface, one of the programming language clients or the kubectl command line interface. Then they would need a definition of the resource or resources they want to create as an Kubernetes API object. These object definitions or specs can be in JSON or YAML format and it would be fair to liken them to orchestration templates. An example Kubernetes object spec can be found in APPENDIX C. One of the crucial features of Kubernetes specs is the importance of meta data about the resources. Once resources are described via labels and annotations, this meta data is used both throughout Kubernetes and other third party software using Kubernetes. This emphasis on meta data is one of the strong suits of Kubernetes specs.

The main unit of operation in Kubernetes is called a Pod. It can be seen as an abstraction of a single application or a service. Pods are made from container or containers, IP address and if there is any storage resources. Multiple containers within a Pod shares the Pod IP address and can communicate via localhost. The namespaces and the IP address belongs to infrastructure container called, pause container and all other containers created runs as it's children. so that failures among any of the pods containers don't affect each other and any of the containers can be safely restarted. (Sayfan, n.d.) It is recommended to only put applications that needs to be on the same namespace into a Pod together. This includes the complimentary services such as logging, metric collection etc. Bundling application containers and complimentary containers into a single pod also called the side car pattern. Pods are considered temporary and disposable. They can't heal themselves in case of failures and generally in need of other structures for management like ReplicaSets and Deployments. These and a few other constructs are called controllers.

ReplicaSets are a controller construct that can run any number copies of a Pod at the same time. A ReplicaSet monitors the Pods and ensure that number of Pods designated in ReplicaSet definition are running at any given time. It spawns new Pods if some of current ones fail etc. ReplicaSets can be scaling for managing different numbers of Pods.

Developments are higher level controllers that also uses ReplicaSets for their operations. Main feature of Deployments is to make updates to Pods easy and safe. They can progressively update the Pods they monitor. Meaning it spawns new Pods with updated image and terminate the Pods with old images as the new ones becomes ready. In case of errors updates can be rollback to previous versions. Since Deployments can do anything that ReplicaSets do and more. They are usually recommended over ReplicaSets for general use.

There are also other controllers like DaemonSets which ensure copies of a Pod runs in every Node and Jobs which are temporary Pods terminated after they completed their designated operation.

Since it's expected from Pods to be disposable. It would not be reliable to try use Pod IPs for using any service that a Pod provides. For example a front end shouldn't have to track the ever changing IP addresses of Pods that hosts it's back end. Services are abstractions that provides reliable access to Pods. The default behavior of services is to assign a virtual IP (called ClusterIP) inside a cluster then forward traffic to this virtual

IP to the Pods defined. Inside a node kube-proxy is responsible from getting service definition from API and preparing forwarding rules generally using IP tables. Beside using ClusterIP's services can also define a port in each node for access called NodePort and Services can also use load balancers if the infrastructure below supports it. (Marmol et al., 2015) IP addresses of Services are only accessible from inside the cluster. For inbound connections to reach any Service therefore any Pod there is a resource called Ingress. Ingresses are basically a set of rules that forwards any inbound connection to any Service according to its definition. This definitions can be IP addresses, domains and subdomains or paths to a domain. Besides creating an Ingress resource users also have to create an Ingress controller. Ingress controller is actually a standard Pod that is registered into controller list via the API. There are several alternatives for an Ingress controller, Kubernetes project itself also provides an nginx based controller in their Github repository.

# 4 EXPERIMENTATION

In this chapter we design and implement a series of experiments to evaluate if Kubernetes is a feasible platform to use virtual network functions. We previously identified container perfomance and effects of co-location as important factors that may impact operation of network functions in container environments. The focus of the experiments would be those two factors by using end to end network latency as the primary metric for evaluating the results.

The experiments is grouped into two sets. First experiment examines the effects of co-location by generating the same amount of load with a different total amount of containers running in system for each step. The main goal of this experiment is to find out how the network latency is affected by the total number of containers in system. The second experiment examines the how different configurations of containers perform under the same amount of load. The main goal of this experiment is to examine how container performance is affected by different parameters such as resource limits and instance counts.

## 4.1 Experiment Design

We designed a set of experiments for measuring end to end network performance on Kubernetes using several different setups.

Contrary to virtual machines, containers have full access to host machines computing resources and in a typical cloud environment we can assume that containers always run in constrained mode. Therefore, to mimic real world operation, in all our experiments the container pods are constrained in CPU power, ranging from a tenth to a quarter of a CPU.

As a representative network function, nginx is used in the experiments. Although commonly known as a web server, it is often used in network functions such as load balancer and reverse proxy. Nginx is deployed to the cluster via Kubernetes deployments using

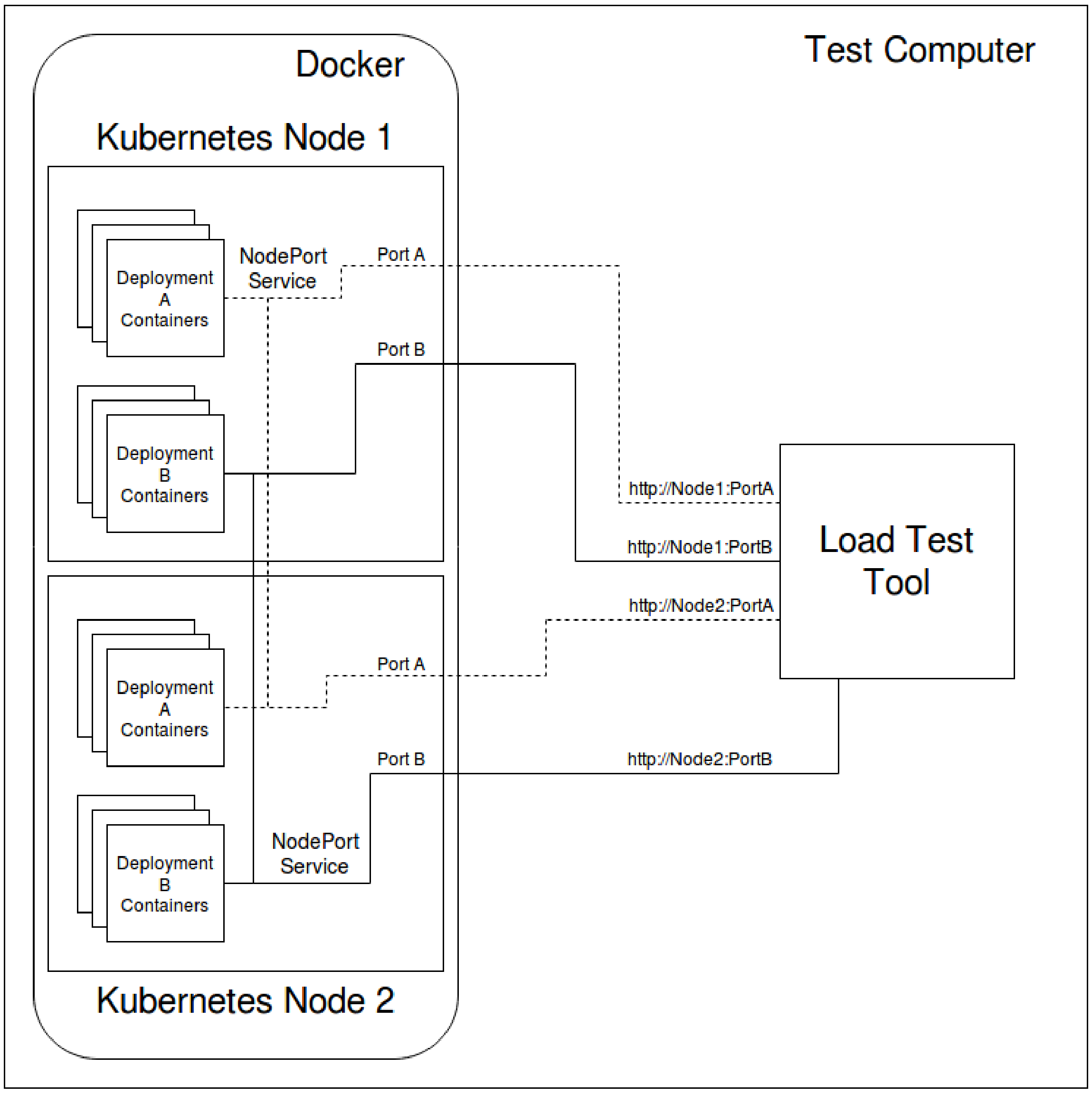

Figure 4.1: A representation of the experiment setup.

a custom Docker image.To use in the tests, we also prepared a Docker image with a modified web server to be able send usage metrics to Prometheus. This Docker image contains a nginx version 1.12.2 compiled with nginx-vts module. Kubernetes deployments are ideal deployment tools for our case since they can create and maintain a specified number of instances for an application and they also can easily be scaled horizontally. All containers inside a deployment is exposed through a single endpoint using Kubernetes NodePort services. An example representation of the experiment setup can be seen in Figure 4.1

For the first experiment, we create three separate deployments and for a varying number of instances in each deployment we run a series of tests generating a pre-determined amount of network load. For load generation a HTTP load testing tool bombardier is used. It is a command line tool written in Go programming language and it can save results in JSON format for easier further processing.

Our methodology for running the tests for the first experiment is the following.

1. First we initialize three deployments with minimum amount of instances
2. For each step we scale the deployments to a pre-determined amount of instances for each deployment.
3. For each deployment we run the load test software to send various amount of requests from various amount of concurrent connections

Number of instances used in the tests and parameters used for the load testing tools can be found below. Probability distribution of the request traffic in the experiments is an uniform distribution.

Table 4.1: Web server load testing parameters

| Connections | Total Requests |
|---|---|
| 20 | 10000 |
| 40 | 30000 |
| 60 | 50000 |
| 80 | 80000 |
| 100 | 100000 |

Table 4.2: Number of instances for each deployment

| Application 1 | Application 2 | Application 3 |
|---|---|---|
| 1 | 1 | 1 |
| 10 | 1 | 1 |
| 25 | 1 | 1 |
| 50 | 1 | 1 |
| 75 | 1 | 1 |
| 100 | 1 | 1 |
| 1 | 1 | 1 |
| 5 | 5 | 1 |
| 12 | 12 | 1 |
| 25 | 25 | 1 |
| 37 | 37 | 1 |
| 50 | 50 | 1 |

The main motivation for selecting such an instance distribution can be explained with a number of reasons. First, it is desirable to learn how co-locating containers are affected if other containers sharing the same host starts to experience heavy load. Second, we would like to find out the affect of multiple different services experiencing heavy load on each other and on other containers within the host. For these reasons, we included a control application in all the schemes which always has the same amount of instances. By utilizing another application we observed how other applications are affected from the system load and total number of containers in a node.

The difference between microservices and monolithic applications is considered essential in both Docker and Kubernetes. For figuring out how this impacts the performance of applications, we designed a second set of experiments. In this second setup we compared two deployments with equal amount of resources. One with a single container and other with a number of instances. While this does not realisticly represent the real world microservice behavior, it is useful for measuring the overhead coming from using multiple containers.

There are certain changes at the methodology for this experiment. Since, instance counts for this experiment is static, it allows for more variety in testing parameters. Concurrent connection count varies from 100 connections to 2000 connections with increments of 50. Since connection count is proved to be the more important variable from the first experiment we use static duration of 2 minutes instead of total request counts for this experiment. This allows for accurate and predictable testing durations. For this experiment, we use another go based open source load testing tool called vegeta for getting percentile metrics.Testing parameters for the second experiment can be found in Table 4.3. CPU limits are expressed as the Kubernetes resource unit

called milicores or milicpu that is equal to a thousandth of a CPU core. CPU limits are enforced by not letting a container use more CPU time than its CPU limit as milicores * 100 in 100 miliseconds.

Table 4.3: Second experiment setups as instance count * CPU limits

| Setup | Configuration 1 | Configuration 2 |
|---|---|---|
| Setup A | 5 * 250m | 10 * 250m |
| Setup B | 10 * 250m | 20 * 250m |
| Setup C | 1 * 1250m | 1 * 2500m |
| Setup D | 1 * 2500m | 1 * 5000m |
| Setup E | (a) 10 * 125m | 20 * 125m |
| | (b) 5 * 250m | 10 * 250m |
| | (c) 3 * 500m | 5 * 500m |
| Setup F | (a) 20 * 125m | 40 * 125m |
| | (b) 10 * 250m | 20 * 250m |
| | (c) 5 * 500m | 10 * 500m |

## 4.2 Setup

This section explains the test environments that all the experimentations runs on. While this setup has many components, all the software components are selected from available free and open source tools.

The hardware used in this setup is a desktop computer. Its specifications are in the following

— 8 core Intel i7-920 processor

— Asus P6T Motherboard

— 20 GiB 1066 Mhz DDR3 memory

— Ubuntu 16.04 64 bit operating system

One of most essential parts of this setup would be a cloud platform that can run and manage containers. We chose the Kubernetes container orchestration software as a cloud platform after considering its features, widespread adoption and maturity. As of yet, there is no standard way to build a Kubernetes cluster. Considering manually installing Kubernetes to each node would be unrealistic at scale, there are several installation tools that can be selected to build a Kubernetes cluster.

While minikube is one of the easiest to installation tool available, it is actually created for development and testing purposes. Minikube's method of creating a single node cluster inside of a virtual machine makes it unsuitable for mimicking real world clusters. Another tool for building clusters is kubeadm which is developed by the Kubernetes team. Although kubeadm currently is at beta stage, it allows easy installation among multiple nodes regardless of type. Kubeadm also leaves the network setup to the administrator's choice via the CNI, thus allowing to experiment with different network configurations. We selected kubeadm as the installation tool for our work. We installed kubeadm inside of Docker containers instead of virtual machines. This is a pattern often called dind or Docker in Docker which consists of installing another Docker daemon inside of a container. Then we built a three machine test cluster using Kubernetes with one master and two nodes.

For monitoring the cluster state and container performance, an external tool is needed. As a monitoring tool, we chose Prometheus which is a modern pull based monitoring software that is developed by the same non-profit organization that developed Kubernetes itself. Prometheus can also receive metrics from other applications via extensions called exporters. Prometheus can be easily installed using the prometheus-operator. Operators are special types of resources in Kubernetes. Each operator has a set of custom defined workloads for managing that application. Operator pattern is preferred for automating applications that requires specific operational know-how to manage. These kinds of applications are often stateful in operation such as databases, monitoring tools etc.

## 4.3 Results

We executed the test set described above on Kubernetes cluster that is defined in the setup section. The experiment setup was run by automated scripts with 10 repetitions. The results were then plotted using the arithmetic mean of the data collected from 10 experiments.

In the Figure 4.2 there are three applications consist of identical containers created by different deployments to measure the effect of co-location. While applications 1 and 2 change their instance counts on different tests, application 3 stands as a control application to see if unrelated containers get affected from the overall system load.

It can be seen for all cases containers are very effective for utilizing system resources. Even a dated desktop computer can run high number of container instances. Running

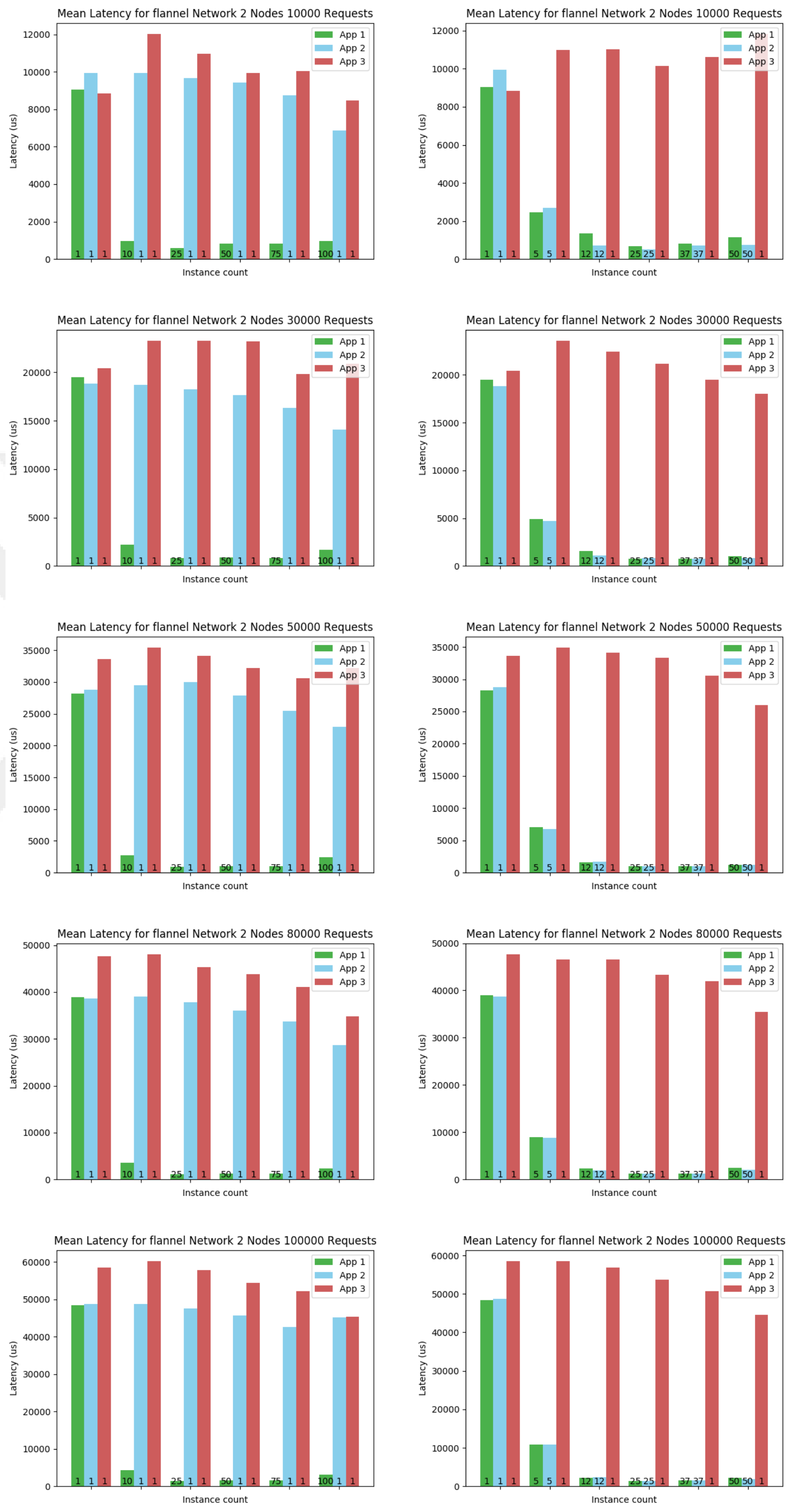

Figure 4.2: Test results for the first experiment.

more than 100 instances affected the latency minimally. High latency at the initial step is expected because only one container instance with constrained resources is trying to respond to all of the requests. However, when the instance count is increased, latency improves linearly and after a certain point saturates. There is slight increase at the final step because of the load increase of the host.

While the latency of each application increases considerably when both applications experience load this can only be observed on the step two and not other steps. We can infer that this is only caused by the lower instance counts rather than any problems about the co-location. Latency of the control application does not seem to get affected from the other applications' load, this fulfills one of important requirements for network function virtualization.

Single instances were affected greatly from the changes in number of concurrent connections and total request count. Mean latency increased linearly with number of connections. From our first experiment with 20 concurrent connections and 10000 requests to our last experiment with 100 connections and 100000 connections. There has been a nearly total 500 percent increase with latency. However for higher instance counts increase was around 150 percent.

The second experiment was also run by automated scripts. The results were then plotted using the arithmetic mean of the data collected from the experiments. Besides from mean and maximum latency this results also include a new type of a metric called percentile metrics. Percentile metrics are calculated by sorting the results from minimum to maximum and then disregarding highest 100-n percent of the results. The maximum remaining result is called the nth percentile result. Percentile results are used for presenting a more accurate state about the application performance in comparison to the arithmetic means. This is mainly because of a few very slow requests can affect the average latency and percentile metrics allow for disregarding outlier results and presents a more accurate representation of the applications general performance. The percentile values are generally used in practice are 90th, 95th and 99th. They are often accompanied by the median 50th result. We use 95th and 99th percentile values for this experiment.

For setup A and B instance counts of the deployments are doubled in each configura-

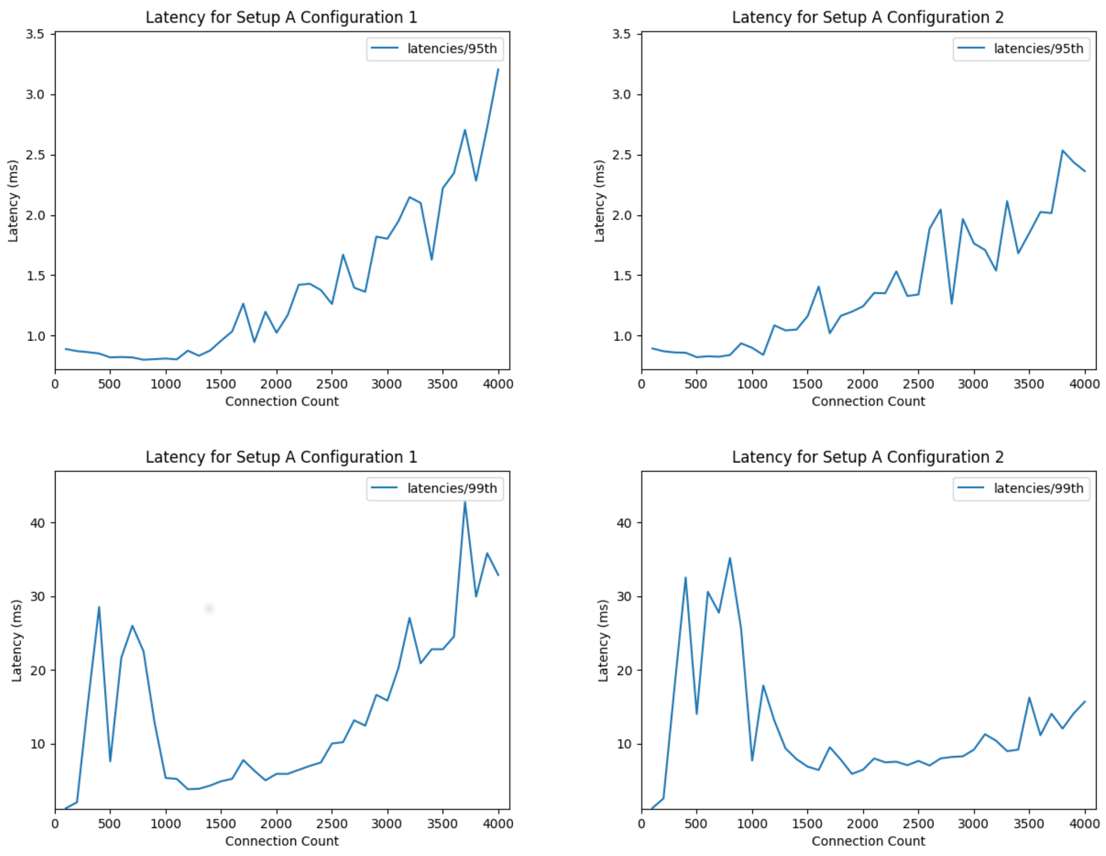

Figure 4.3: Setup A results for the second experiment.

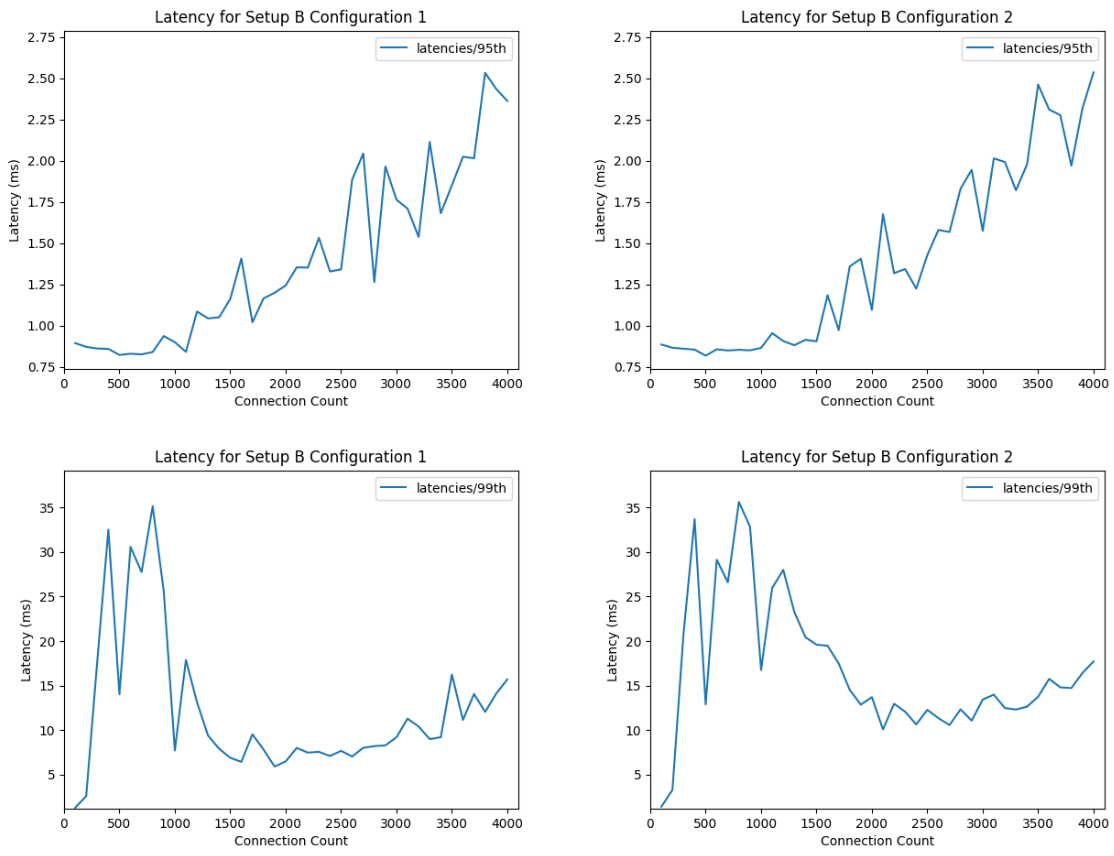

Figure 4.4: Setup B results for the second experiment.

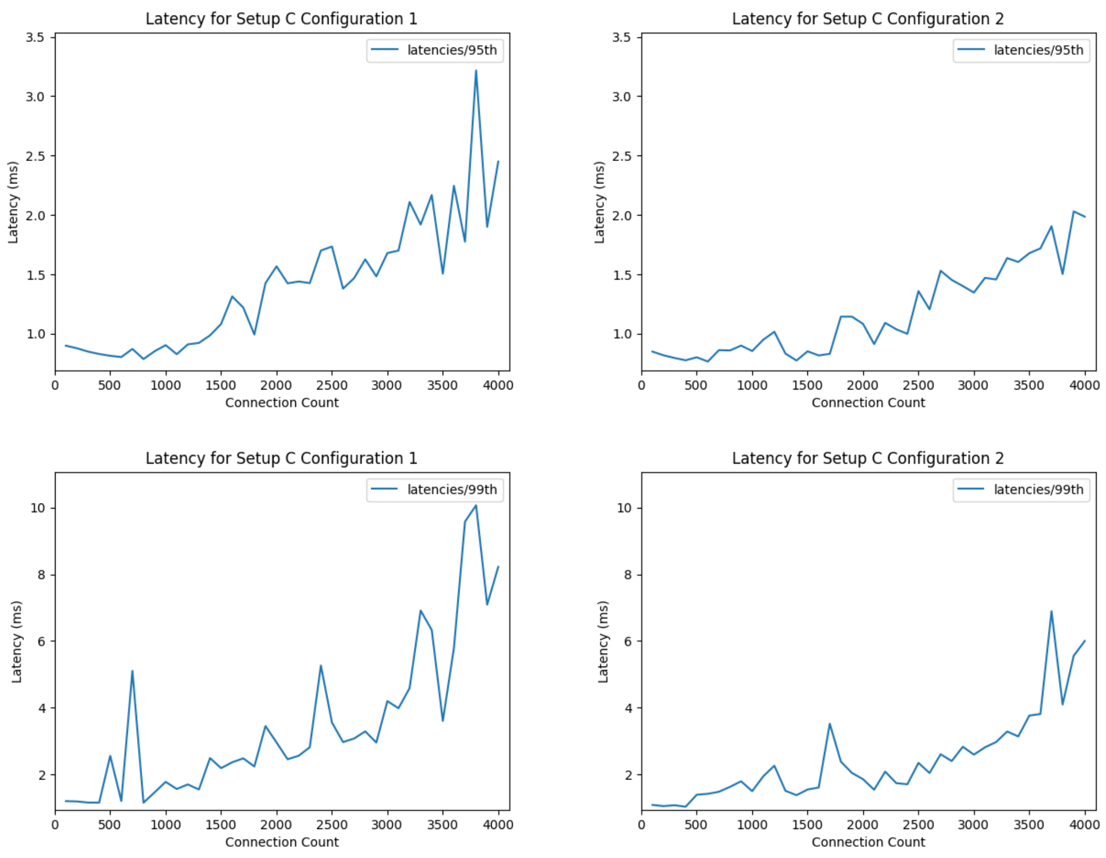

Figure 4.5: Setup C results for the second experiment.

tion. For setup A there is a clear decrease in latency from configuration 1 to 2. But when instance count increased again from configuration 1 to 2 in setup B, performance subtlety worsened. Even thought the increase in latency was barely noticeable compared those in setup A.

This shows that after a certain point the extra instances started to add more overhead and caused latency to increase. it became clear that increasing the instance count wouldn't always improve the application performance. An optimal amount of instances has to be found for each application to achieve best performance.

For setup C and D single container instances that have equal computing power of their counterparts in setup A and B. Similar to previous setups, the resource limits are doubled for each configuration. It is expected for instances in setup C's latency to decrease. It was assigned more resources while load stayed the same. But decrease in latency was significant from configuration 1 to 2, It was minimal for most connections

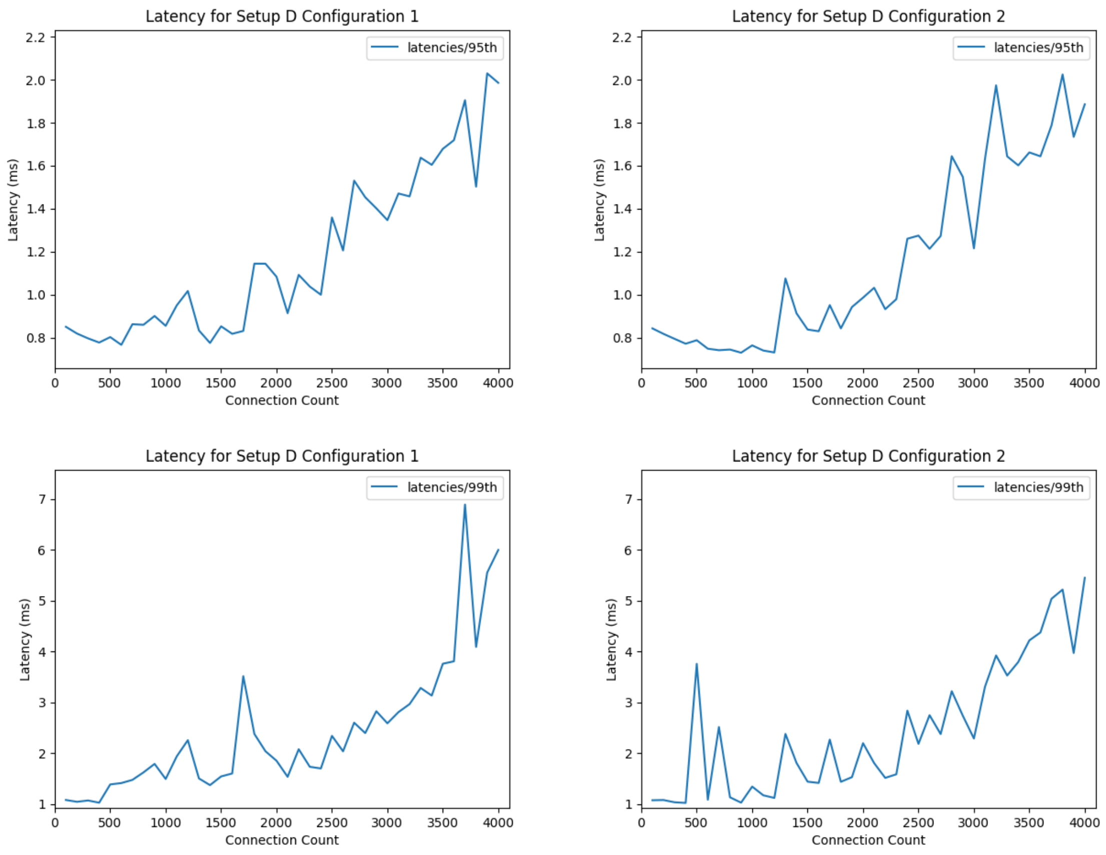

Figure 4.6: Setup D results for the second experiment.

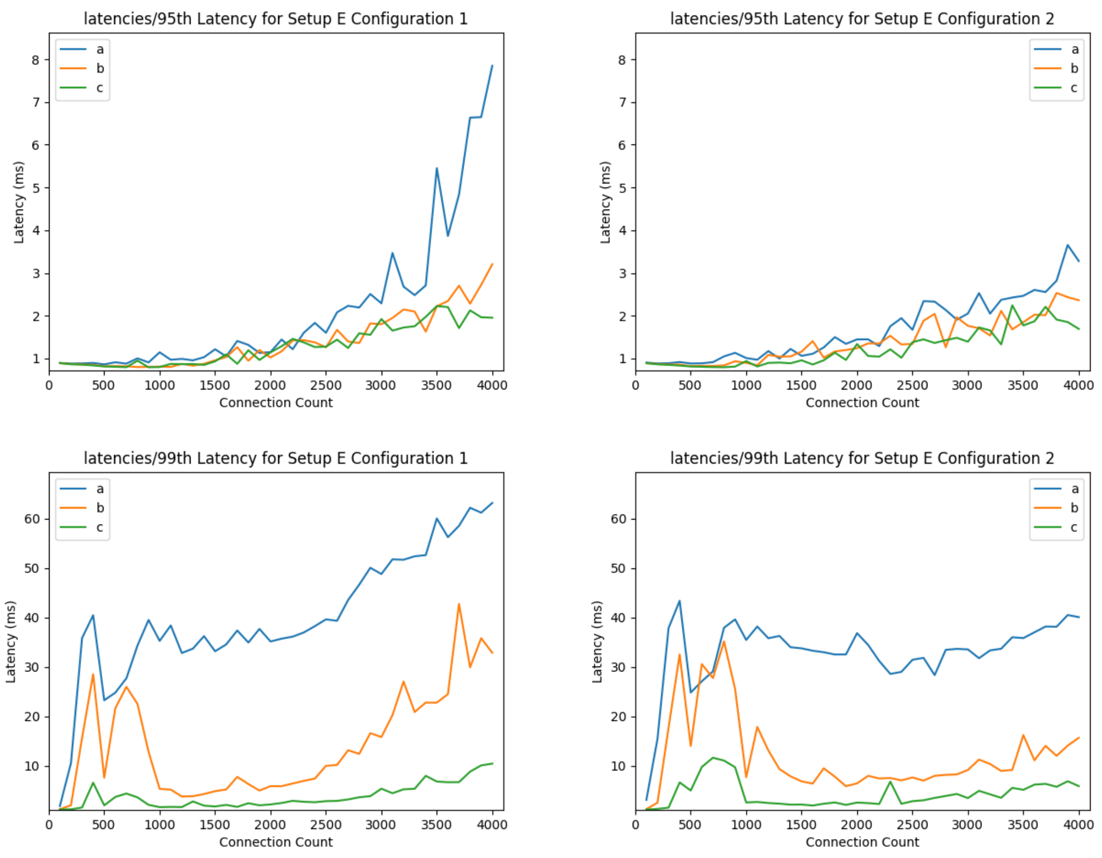

Figure 4.7: Setup E results for the second experiment.

from deployment 1 to 2 in setup D and only become noticeable in the slowest 1 percent of the connections.

Without use of 95th percentile values, there seems to be a significant difference in latency between setup A and B versus setup C and D that suggests there is an huge overhead for using multiple containers. But percentile data clearly shows for 95 percent of the connections performed very similarly in these cases. But between 95th and 99th percentile deployments from setup A and B's performance decays more rapidly compared to those in setup C and D whose performance only starts to decay in slowest 1 percent of the connections.

From setup E and F approach of experiments changes slightly. While comparing a single container with multiple containers can be useful in order to understand the amount of overhead. In actual use such a setup would be impractical. To mitigate that problem, a different series of experiment setups that includes a more evenly matched

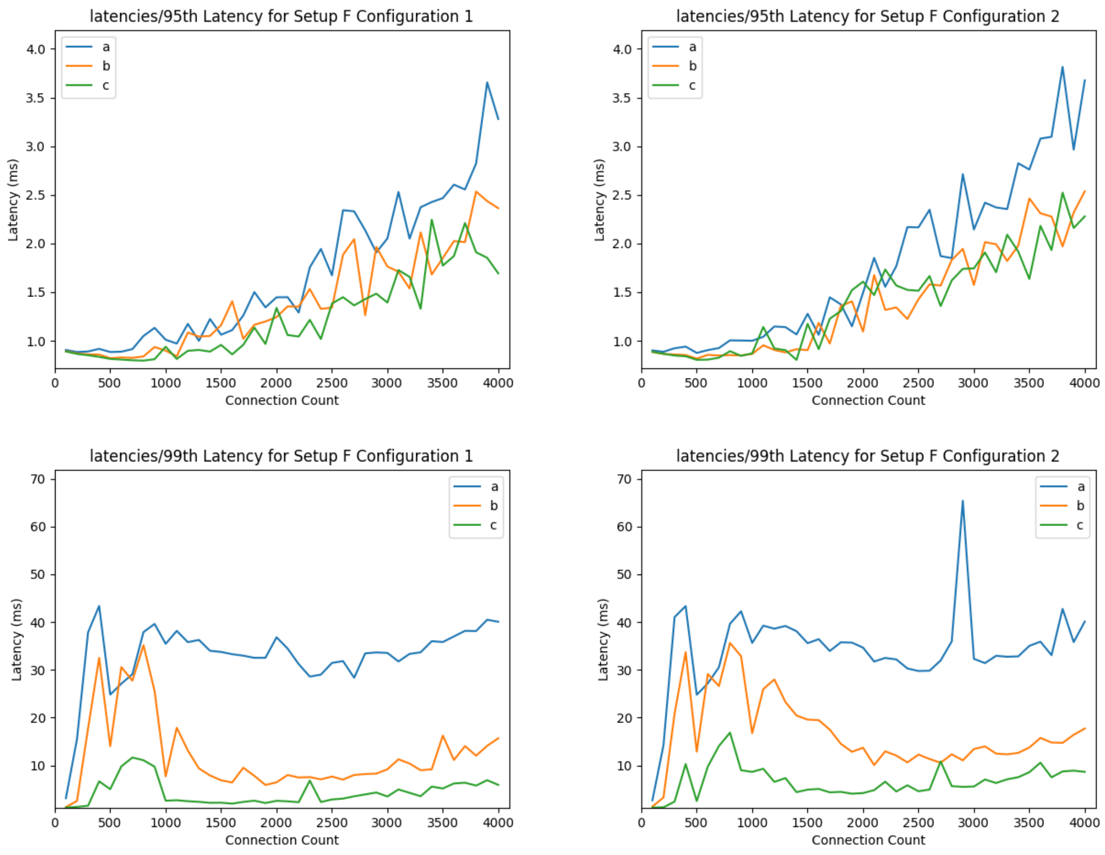

Figure 4.8: Setup F results for the second experiment.

containers are introduced. Each setup contains one deployment with more resources for each container and one container with more instances. The rate for extra resources and instances is always 2.

While latency decreased for all cases from configuration 1a, 1b, 1c to 2a, 2b and 2c, in setup E latency increased or stayed similar from configuration 1a, 1b, 1c to 2a, 2b, 2c in setup F. This results from setup also supports the observations from the previous experiments that after a saturation point adding more instances increases the overhead and therefore latency. This saturation point amounts to 2500m CPU resources for these test cases.

The other important result from setup E and F is the significance of the balance between resource limits and instance counts. All the deployments in setup E and F has an equal amount of total resources but there is stark difference between their performance. With a correct configuration it is possible for multiple containers to achieve similar performances to their single container counterparts. While configuration 1b of setup E is seem to perform better than configuration 1 of setup C, it should be noted that the likely cause is the extra 250m resources the former has. But for configuration 2b of setup E and configuration 2 of setup C. Albeit latter performs better, the overall performances is very similar.

# 5 CONCLUSION

Virtualization is one of the essential technologies behind modern computing infrastructure. In this work, we tried to examine the feasibility of combining two active subdomains of virtualization field : Network function virtualization and container based virtualization.

At first, we examined current systems for network function virtualization and tried to determine what technologies are required to implement network functions, which architectures are preferred when building systems to run network functions etc. Then we examined current container technologies and tried to understand their capabilities, if there are alternatives or counterparts to technologies that are required by virtual network functions.

We built a container testbed with Kubernetes for testing how containers perform under load and can they adhere to often strict requirements for network functions. Our preliminary results show that while the vast majority of the requests performed within acceptable limits, the slowest one percent of the requests latency varied significantly. This may not be acceptable for the network functions with strict latency constraints. Another important factor to examine was the how containers are affected from co-location. Results of our experiments showed that even at a high number of total containers in hosts, co-location had an observable but minimal impact on the performance.

Our conclusion was as is standard container clusters can be used for implementing virtual network functions. But with using containers it should noted that these systems were made to utilize a specific cloud native approach in designing the software. Since this requires a system redesign for the most of the network services it can not be recommended for each case to use container based systems.

For future work, we can use of multiple different computing units instead of a single computer for our experiments. This would provide far more realistic results that can be better applied to cloud environment. Tracking the hosts' resource usage and using it as a complementary metric for how latency is affected would improve the reliability of our results. This work's scope can be extended by including other technologies such as specialized high performance networking to see if they improve network performance on container environments

As a conclusion, fundamental technologies that can be used for implementing virtual network functions in container environments are present. Considering there are still no out of box solutions and the existing solutions are still immature and hard to install and use. We believe there are promising developments achieve network function virtualization in containers. In the future more practical solutions would be developed.

# REFERENCES

Abdelwahab, S., Hamdaoui, B., Guizani, M. and Znati, T. (n.d.). Network function virtualization in 5g, **54**(4) : 84–91.

Adufu, T., Choi, J. and Kim, Y. (2015). Is container-based technology a winner for high performance scientific applications ?, *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pp. 507–510.

Bernstein, D. (n.d.). Containers and cloud : From LXC to docker to kubernetes, **1**(3) : 81–84.

Boettiger, C. (2015). An Introduction to Docker for Reproducible Research, *SIGOPS Oper. Syst. Rev.* **49**(1) : 71–79.
**URL:** *http ://doi.acm.org/10.1145/2723872.2723882*

Burns, B., Grant, B., Oppenheimer, D., Brewer, E. and Wilkes, J. (2016). Borg, Omega, and Kubernetes, *Commun. ACM* **59**(5) : 50–57.
**URL:** *http ://doi.acm.org/10.1145/2890784*

Chayapathi, R., Hassan, S. F. and Shah, P. (n.d.). *Network Functions Virtualization (NFV) with a Touch of SDN*, Addison-Wesley Professional.
**URL:** *http://proquestcombo.safaribooksonline.com/book/networking/9780134464312*

Chen, J., Chen, Y., Tsai, S. C. and Lin, Y. B. (n.d.). Implementing NFV system with OpenStack, *2017 IEEE Conference on Dependable and Secure Computing*, pp. 188–194.

*Cloud Native Computing Foundation* (n.d.).
**URL:** *https://www.cncf.io/*

*CoreOS* (n.d.).
**URL:** *https://coreos.com/*

*cri-o* (n.d.).
**URL:** *http://cri-o.io/*

Cziva, R., Jouet, S. and Pezaros, D. P. (2015). GNFC : Towards network function cloudification, *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pp. 142–148.

Cziva, R., Jouet, S., White, K. J. S. and Pezaros, D. P. (2015). Container-based network function virtualization for software-defined networks, *2015 IEEE Symposium on Computers and Communication (ISCC)*, pp. 415–420.

Cziva, R. and Pezaros, D. P. (2017). Container Network Functions : Bringing NFV to the Network Edge, *IEEE Communications Magazine* **55**(6) : 24–31.

*Docker* (n.d.).
**URL:** *https://www.docker.com/*

*Docker Hub* (n.d.).
**URL:** *https://hub.docker.com/explore/*

*ETSI - Welcome to the World of Standards!* (n.d.).
**URL:** *http://www.etsi.org/*

Eur (2013). *ETSI Standard GS NFV 002.*
**URL:** *https://portal.etsi.org/webapp/workprogram/Report_WorkItem.asp?WKI_ID=43827*

Eur (2014). *ETSI Standard GS NFV-MAN 001.*
**URL:** *https://portal.etsi.org/webapp/workprogram/Report_WorkItem.asp?WKI_ID=41954*

Gonzalez, A. J., Nencioni, G., Kamisiński, A., Helvik, B. E. and Heegaard, P. E. (2018). Dependability of the NFV Orchestrator : State of the Art and Research Challenges, *IEEE Communications Surveys Tutorials* pp. 1–1.

Gray, K. and Nadeau, T. D. (n.d.). *Network Function Virtualization*, Morgan Kaufmann.
**URL:** *http://proquestcombo.safaribooksonline.com/book/networking/9780128023433*

Haider, Md. Rezzakul (n.d.). *Deployment of TOSCA cloud services archives using kubernetes*, Master's thesis.

Hwang, J., Zeng, S., Wu, F. y. and Wood, T. (n.d.). A component-based performance comparison of four hypervisors, *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pp. 269–276.

Imagane, K., Kanai, K., Katto, J., Tsuda, T. and Nakazato, H. (2018). Performance evaluations of multimedia service function chaining in edge clouds, *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pp. 1–4.

Irshad, T. (2018-02-12). *Design and implementation of a testbed for network slicing*, G2 pro gradu, diplomityö.
**URL:** *http ://urn.fi/URN :NBN :fi :aalto-201802231619*

Joy, A. M. (2015). Performance comparison between Linux containers and virtual machines, *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342–346.

Kamp, P.-H. and Watson, R. N. M. (n.d.). Jails : Conning the omnipotent root., p. 15.

Khalid, J., Coatsworth, M., Gember-Jacobson, A. and Akella, A. (n.d.). A standardized southbound API for VNF management, *Proceedings of the 2016 Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMIddlebox '16, ACM, pp. 38–43.
**URL:** *http://doi.acm.org/2940147.2940156*

Kolyshkin, K. (2006). Virtualization in linux, *White paper, OpenVZ* **3** : 39.
**URL:** *http://mirror.ihc.ru/download.openvz.org/doc/openvz-intro.pdf*

*Kubernetes* (n.d.).
**URL:** *https://kubernetes.io/*

*Kubernetes Architecture* (n.d.).
**URL:** *https://kubernetes.io/docs/concepts/architecture/cloud-controller/*

Kurebayashi, R., Khan, A. and Obana, K. (n.d.). Activities toward nfv standardization, *Technical report*, NTT Docomo.

Li, Y. and Chen, M. (n.d.). Software-defined network function virtualization : A survey, **3** : 2542–2553.

*Linux Containers* (n.d.).
**URL:** *https://linuxcontainers.org/*

*Linux Foundation* (n.d.).
**URL:** *https://www.linuxfoundation.org/*

Marmol, V., Jnagal, R. and Hockin, T. (2015). Networking in containers and container clusters, *Proceedings of netdev 0.1* .

*OASIS | Advancing open standards for the information society* (n.d.).
**URL:** *https://www.oasis-open.org/*

*OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC | OASIS* (n.d.).
**URL:** *https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca*

Ongaro, D. and Ousterhout, J. K. (2014). In search of an understandable consensus algorithm., *USENIX Annual Technical Conference*, pp. 305–319.

*OpenStack Open Source Cloud Computing Software* (n.d.).
**URL:** *https://www.openstack.org/*

*rkt is a pod-native container engine for Linux. It is composable, secure, and built on standards* (2017). original-date : 2014-11-11T23 :13 :18Z.
**URL:** *https://github.com/rkt/rkt*

Sayfan, G. (n.d.). *Mastering Kubernetes*, 2 edn, Packt Publishing.
**URL:** *http://proquestcombo.safaribooksonline.com/book/operating-systems-and-server-administration/9781788999786*

Singh, S. and Singh, N. (2016). Containers Docker : Emerging roles future of Cloud technology, *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, pp. 804–807.

*Swarm mode overview* (2017).
**URL:** *https://docs.docker.com/engine/swarm/*

*Tacker - OpenStack* (n.d.).
**URL:** *https://wiki.openstack.org/wiki/Tacker*

## APPENDIX A EXAMPLE TOSCA TEMPLATE

```
tosca definitions version:  tosca simple profile for nfv 1 0 0

description: OpenWRT with services

metadata:
  templatename: OpenWRT

topology template:
  node templates:

    VDU1:
      type: tosca.nodes.nfv.VDU.Tacker
      capabilities:
        nfv compute:
          properties:
            num cpus: 1
            mem size: 512 MB
            disk size: 1 GB
      properties:
        image: OpenWRT
        config:
          param0: key1
          param1: key2
        mgmt driver: openwrt
        monitoring policy:
          name: ping
          parameters:
            count: 3
          interval: 10
        actions:
          failure: respawn
CP1:
  type: tosca.nodes.nfv.CP.Tacker
  properties:
    management: true
    order: 0
    anti spoofing protection: false
  requirements:
      virtualLink:
        node: VL1
      virtualBinding:
        node: VDU1

VL1:
  type: tosca.nodes.nfv.VL
  properties:
    network name: net mgmt
    vendor: Tacker
```

## APPENDIX B EXAMPLE DOCKERFILE

```
FROM ubuntu:xenial

COPY . /opt/nginx
WORKDIR /opt/nginx

RUN apt get update    apt get install  y
        libexpat1
        libgeoip1
        libgd3
        libluajit 5.1 2
        perl
        libxml2
        libxslt1.1
        libhiredis0.13
        libssl1.0.0

RUN dpkg  i
        nginx common1.12.2 0+xenial0 all.deb
        libnginx mod
        nginx full 1.12.2 0+xenial0 amd64.deb

WORKDIR /
RUN rm  rf /opt/nginx

EXPOSE 80
CMD ["nginx", " g", "daemon off;"]
```

## APPENDIX C EXAMPLE KUBERNETES OBJECT SPEC

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: garasu
  labels:
    app: garasu
spec:
  replicas: 1
  selector:
    matchLabels:
      app: garasu
  template:
    metadata:
      labels:
        app: garasu
        video: dash
    spec:
      containers:
        name: nginx vts
        image: ugurcan377/nginx vts
        ports:
          containerPort: 80
        volumeMounts:
          mountPath: /var/www/html/dash videos/garasu
          name: dash videos
          mountPath: /var/www/html/index.html
          name: dash index
        resources:
```

```
    requests:
      cpu: "10m"
    limits:
      cpu: "250m"
  name: vts exporter
  image: sophos/nginx vts exporter
  ports:
    containerPort: 9913
volumes:
  name: dash videos
  hostPath:
    path: /opt/dash videos/garasu
    type: Directory
  name: dash index
  hostPath:
    path: /opt/dash videos/garasu/garasu.html
    type: File
```

## BIOGRAPHICAL SKETCH

Uğurcan Ergün was born on June 19, 1991 in Istanbul. He graduated from Istanbul Köy Hizmetleri Anatolian High School at 2009. He studied Computer Engineering in Kocaeli University and got his Bachelor's Degree at 2013. In September 2013 he became a software developer for the Istanbul based cloud service provider Skyatlas Inc. Since October 2015 he is a graduate student at Galatasaray University.