UNIVERSITÀ DEGLI STUDI DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

MASTER DEGREE IN TELECOMMUNICATION ENGINEERING

FINAL THESIS

Performance Evaluation of Containerised Virtual Network Functions on the Edge network and in the Cloud

Graduant:
Joshua Tetteh Ocansey

Supervisor:
Prof. Fabrizio Granelli
Co-Supervisor:
Dr. Riccardo Bassoli

Academic Year: 2016/2017

UNIVERSITY OF TRENTO

Abstract

Department of Information Engineering and Computer Science

Master of Science in Telecommunication Engineering

by Joshua Tetteh OCANSEY

Cloud service providers implement Virtual Network Functions (VNFs) with container technology rather than traditional virtual machines (VMs). Docker is an increasingly popular container technology in recent years. Docker, like other container technologies, increases cloud operations' agility and decreases their cost, while also enhancing application encapsulation and deployment performance. Since VNFs are implemented with virtual machines VMs, significant overheads are added to the operating system running VMs, and even more so when considering Infrastructures with large-scale microservice applications. Our research endeavor will study Docker networking technology options for VNF performance on the Edge network and in the cloud. On the networks, latency and throughput were analyzed and evaluated for the two Linux bridge and Open vSwitch (OVS) techniques. Our results demonstrate that the Linux bridge has 22 percent less latency than the OVS network and 18 percent more throughput than the OVS bridge on average.

We constructed two architecture scenarios with comparable resources, one utilizing virtualization at the network's edge (Host-in-the-lab) and the other utilizing virtualization on a cloud platform. We discover that in the first situation, where virtualization is located at the network's edge, performance is superior. In the preceding case, throughput is boosted by 18%. This is due to the inclusion of the Data Plane Development Kit (DPDK) to the host of the edge network, which expedites packet processing.

Acknowledgements

This document contains the work I completed for my master's thesis at the GranelliLab of the Department of Information Engineering and Computer Science at the University of Trento in Italy.

I'd like to thank Professor Fabrizio Granelli for his professional guidance and direction throughout my project. I would also like to thank assistant supervisor Riccardo Bassoli, whose comments and feedback were greatly appreciated.
Finally, I'd like to express my gratitude to my family for their unwavering support and encouragement, without which none of this would have been possible.

Contents

List of Figures

List of Tables

List of Abbreviations

NFV	**N**etwork **F**unction **V**irtualization
VNF	**V**irtual **N**etwork **F**ucntions
SDN	**S**oftware **D**efined **N**etwork
DPDK	**D**ata **P**lane **D**evelopment **K**it
AWS	**A**mazon **W**eb **S**ervices
OVS	**O**pen **v** **S**witch
NIST	**N**ational **I**nstitute of **St**andard and **T**echnology
SaaS	**S**oftware **as a** **S**ervice **N**etwork
PaaS	**P**latform **as a** **S**ervice **N**etwork
IaaS	**I**nfrastructure **as a** **S**ervice **N**etwork
EC2	**E**lastic **C**ompute **C**loud
vCPE	**v**irtual **C**ustomer **P**remise **E**quipment
VNF	**V**irtual **N**etwork **F**unction
SFC	**S**ervice **F**unction **C**haining
MANO	**MAN**agment and **O**rchestration
NFVI	**N**etwork **F**unction **V**irtualization **I**nfrastrature

I dedicate this work to my wonderful family, including my wife, Selina Budua Quainoo, and my two lovely daughters, Dromie and Djormie Ocansey. . . .

Chapter 1

Introduction

1.1 Background and Motivation

1.1 Background and Motivation
Traditionally, network functions such as firewalls, DPI, and NAT were implemented on vendor-specific middle-boxes. This means that whenever an upgrade or scalability is required, the devices must be completely replaced, increasing the Cloud Service providers' CAPX and OPEX costs. There is a growing trend to virtualize middle-boxes and transform them into Virtual Network functions that can be chained to provide services to tenants in multi-tenant datacentres who require their own network services. [1].(Livi et al., 2016)

When it comes to service virtualization, there are two main approaches: hypervisor-based solutions and container-based solutions. For many years, hypervisor-based virtual machines (VMs) have been the solution of choice for virtualizing services in many data centers. However, (Struye and Spinnewyn, 2017) concluded that VM outperforms the major container-based technologies (ie. Docker, LXC and rkt). This is due to the fact that the kernel of each VM adds significant overhead to the system's resources. Because of this issue, the industry is moving toward VNF containerization. According to a recent survey conducted by DevOps.com and ClusterHQ, Docker is the most popular container technology used by (ClusterHQ and DevOps.com, 2016). Docker is a lightweight Linux container that reuses the host's kernel and libraries, allowing it to load faster and start instantly while using less RAM.

Docker runs virtualized services by packaging applications with a customized view of their runtime environment, which is referred to as a namespace. A namespace is a mechanism that wraps a global resource in an abstraction, giving the container the appearance of having their own isolated instance of a global resource. Docker containers encase software in a full file system that contains everything the containers require to run applications. This adds extra security to the application because it isolates applications from one another while allowing them to communicate with one another (Anderson et al., 2016) via various network options. VNFs (such as firewalls, DPI, NAT, or load balancers) are frequently "stitched" together in a chain to provide a specific service. This is referred to as Service Function Chaining (SFC). During chaining, VNFs can be in the same or different Docker containers.

Understanding VNF networking issues in operating system level virtualization is critical for successful Docker VNF deployment. VNF performance suffers when the proper networking configuration is not used, regardless of whether they are deployed on the edge or in the cloud.

1.2 Contributions

The following are our contributions:

- We present an up-to-date comparison of native Linux bridge and OpenvSwitch bridge on hardware and software for a variety of interesting cloud-relevant benchmarks and workloads.

- We highlight the fundamental performance impact of existing virtualization methods for deployment of Host-to-Host VNFs.

- We elaborate on the practical factors that affect the performance of virtualization (Throughput and Latency).

- We demonstrate that Docker containers are viable at the scale of a full cloud server, with negligible impact on performance.

- We demonstrated that virtualization may be enhanced at the network's edge using performance enhancements such as DPDK.

1.3 Structure of the thesis

The remaining structure of the thesis is as follows: The state-of-the-art technologies are described in depth in Chapter 2. In chapter 3, we describe the methodology employed in the research. Chapter 4 details the system design, architecture, and implementations, while Chapter 5 presents the conclusion

Chapter 2

State of the Art

2.1 Network Function Virtualization(NFV)

Network functions virtualization (NFV) offers innovative way to design , deploy and manage network services within the data center (sdxcentral, 2015). NFV decouples Network functions from the hardware appliances and make them run as software functions. Examples of network functions are NAT, Firewall, vCPE, virtual Evolved Packet Core (vEPC) etc.

The idea of NFV is to consolidate and deploy the networking elements need to support a wholly Virtualized infrastructure which includes virtual servers, storage and virtual networks. It exploits standard virtualization technologies that runs on high volume services, switches and even storage to virtualize network functions(sdxcentral, 2015). NFV is applied within any data plane or control plane which can be run on both wired and wireless infrastructures.

Scalability is the major concern to most operators running on traditional cloud system. This is because traditional cloud takes too long for the system to get upgraded or scaled network functions when the need be. NFV helps to solve this problem by minimizing the dependence on the hardware equipment which ultimately reduces both CAPX and OPEX for the operators.
With NFV some of the hardware network functions are migrated into software, therefore making it easier for the operators to scale when it is needed (Ravidas et al., 2016). In the ideal data center, traditional cloud could run together with NFV enabled cloud infrastructure; this provides environment for the next generation cloud system (Ravidas et al., 2016).

Apart from Scalability, NFV directly address most of the challenges for cloud providers and also brings extra benefits Some of the benefits are as follows:

1. Hardware flexibility: Traditionally, Hardware cloud offers network vendors a limited options for resource capacities; meaning that any modification of these hardware could lead to extra cost to the operator (Rajendra et al., 2017). However with NFV, cloud providers have the choice to operate with any vendor and have the flexibility to

choose the hardware that meets their requirement for network planning and upgrade.

2. Operational Efficiency: With NFV, automation is easier for operational efficiency. It can exploit machine to machine (M2M) tools to seamlessly move network functions from one host to another host. For instance, it is possible to monitor a system that has more demand and automatically allocate the resource it need to carry out its function. Furthermore, efficiency can be achieved when we cloud providers can centralized common operations and monitor them at central location detect faults in real time.

3. Reduce CAPX and OPEX: In NFV operators can deploy the network services on the commercial off-the-shelf (COTS) equipment which is cheaper and can be used for various network functions and run different software on them. These make the ultimately reduces the cost of equipment as well as reduces the operational cost.

4. Security: Virtualization allows Operators to provision and manage the network while allowing their clients to run their own virtual space and firewall securely within the network (Intel, 2017)

2.1.1 NFV Infrastructure

NFV framework was created by Industry Specification Group (ISG) under the auspices of the European Telecommunications Standards Institute's (ETSI) to facilitate and accelerate the progress of the virtual network services. It consist of of three layers namely,Network Function Virtualization Infrastructure (NFVI) , Virtual Network Function layer and Management and Orchestration layer. NFV Infrastructure is separated into three domains namely: Compute domain, the hypervisor and the infrastructure domain [16].

Our VNFs experiment were deployed on NFVI element of the NFV architecture. Figure 2.1 is the NFV reference architecture framework which contain the NFVI with various domains.

The functionalities of the various components of NFVI are further explained below:
The NFV Infrastructure consists hardware resources of the physical infrastructure which includes compute, storage and network elements. NFVI is connected to virtualized infrastructure Manager which manages and control the physical resource component of the NFVI. Under this NFVI layer, network functions are converted into software called virtual Network functions.

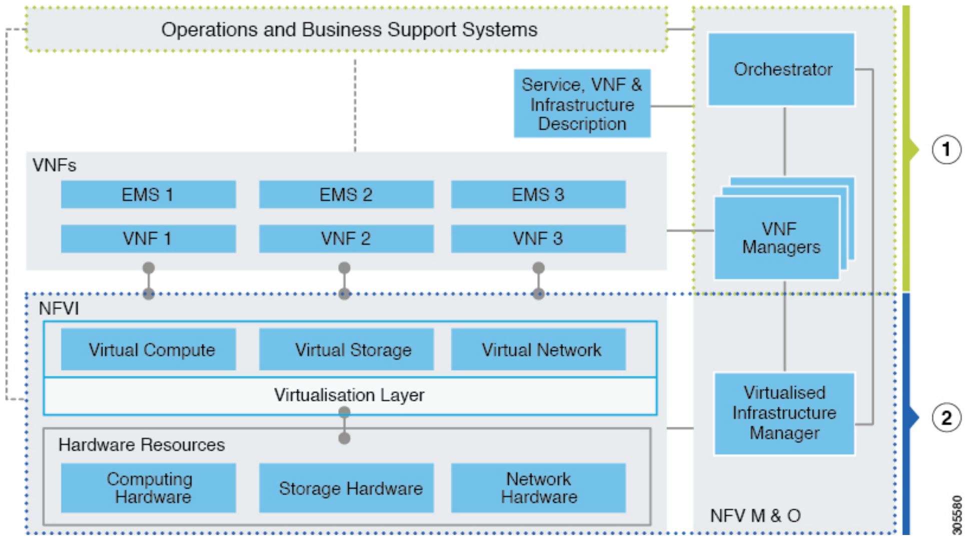


FIGURE 2.1: NFV framework

Virtual Network Function layer is above the NFVI on the NFV framework architecture. VNF is simply a software packages that implements the network functions relying the infrastructure provided by the NFVI. VNF is responsible for handling specific network functions which runs one or more virtual machine (VMs) on top of the physical infrastructure. Virtual network function can function individually or be combined with other network functions to offer specific network service.

NFV Management and Organization (MANO) is the layer responsible for management and orchestration of both physical and virtual resources within the data center. These resources includes networking, storage, networking and virtual machine resources. The main aim of the NFV MANO is allow flexibility and reduces disorderliness that often comes with rapid spin up of the network components. The NFV MANO block is made up of the Virtualized Infrastructure Manager (VIM), the VNF Manager, and the Orchestrator. VIM manages and controls the VNF connections to the NFV Infrastructure. VNF Managers create, update, scale, and terminate VNFs throughout the VNF lifecycle. The Orchestrator is in charge of general network management as well as policy management.

2.1.2 Virtual Customer Premises Equipment (vCPE)

2.3 Virtual Customer Premises Equipment (vCPE)

Virtual customer premises equipment exploits the concept of network function virtualization to help to deploy services on networks and data centers. vCPE adopts models that helps to share common

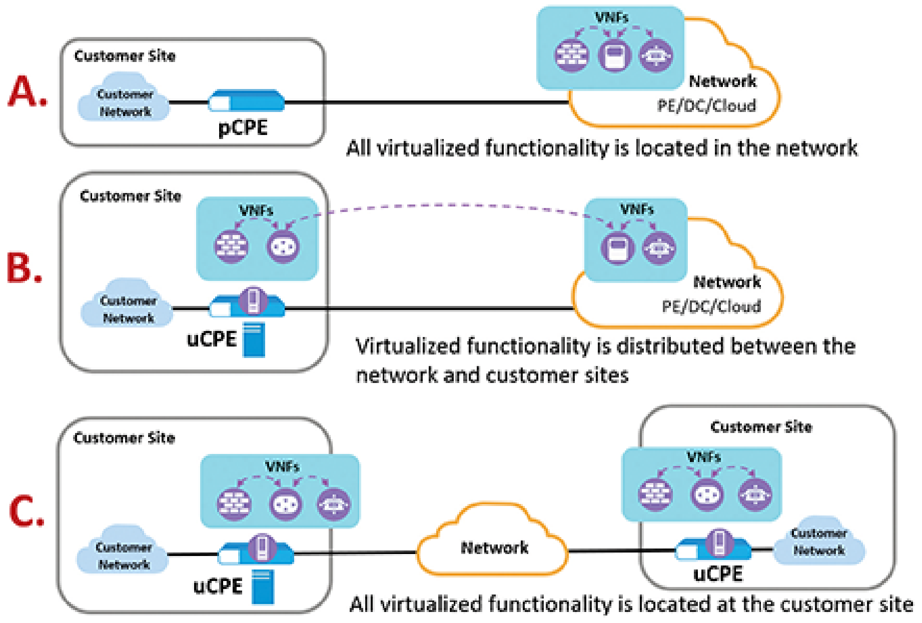


FIGURE 2.2: virtual CPE (isemag, 2017)

pool of resources and dynamaically allocate resources to virtual networks functions[17]. Operators can deploy VNFs on the cloud and offer to the customers. With this service, Enterprise customers can order new services that are provisioned in a single or multiple locations. The emerging vCPE model for enterprise customers enables some of the functions related to traditional CPEs and other specific customer appliances to be virtualized and possibly moved into the cloud (isemag, 2017).

According to projection by (Research, 2018), vCPE will be the single largest drivers of NFV deployment which constituent for more 20% of the NFV related service by the end 2018. This is due to operators leveraging vCPE to provide customers with rapid and dynamic services, where they can easily choose which services they want to be on at any given time. Aside from the benefits to customers mentioned by [20], vCPE provides providers with a scalable and automated solution for rapidly deploying existing and new revenue-generating services.

There are three deployment options as described by (isemag, 2017) shown in figure below.

Architecture Type A describes physical CPE (pCPE) which is not virtualized at customer sites while Network functions are virtualized at location of the cloud.

Architecture Type B describes a universal CPE (uCPE) with a virtualization at customer sites while virtualization is distributed between the cloud and customer sites.

Architecture Type C describes universal CPE (uCPE) at customer sites while all virtualization sits at the customer sites.

Our proposed research Architectures are based on the vCPE use cases architecture except that our proposed architecture the Internet in between the customer and cloud.

2.1.3 Service Function Chaining (SFC)

Service function chaining (SFC) is a mechanism that exploits software-defined networking (SDN) capabilities to create a service chain of connected network services. End-to-end application traffic flows are required to traverse several network functions such as firewall, load balancers, DPI along a predetermined route (Foundation, 2015). For instance, in our research architecture, a user from remote accessing a web application will go through a firewall for security, DPI for packet selection before reaching the web application.

Service Function Chaining can be of great benefit for the operator by provisioning the network applications that have different behaviors and characteristics. For instance a video traffic demand more than a simple web application therefore automating the service function chaining dynamically schedule or allocate resources need for the high demand applications for specific time or occasion (Foundation, 2015).

Service Function chaining can used to control network traffic congestion issues by exploiting SDN in optimizing the network resources and also improving applications. For Instance in Docker container instances, one can leverage on SDN analytical tool like kubernetes to help negotiate network congestion in seamlessly manner.

Network service chaining potentials mean that large number of virtual network functions can be “stitched” together in an NFV environment to torn down the service chain provisioning. Figure 2.3 demonstrates a simple SFC on the NFV platform that allows different traffic classifiers to traverse different chains before reaching the final application

2.2 Containers for Virtualization

Containers are operating-system-level virtualization solution that helps to run software reliably from one computing environment to another not having to bother about dependencies on the infrastructure platform. For instance a cloud developers can move test software on staging environment can be transferred seamlessly onto production on the cloud without a hitches. Software on container is not affected by the operating systems, where it is affected by the platform runs

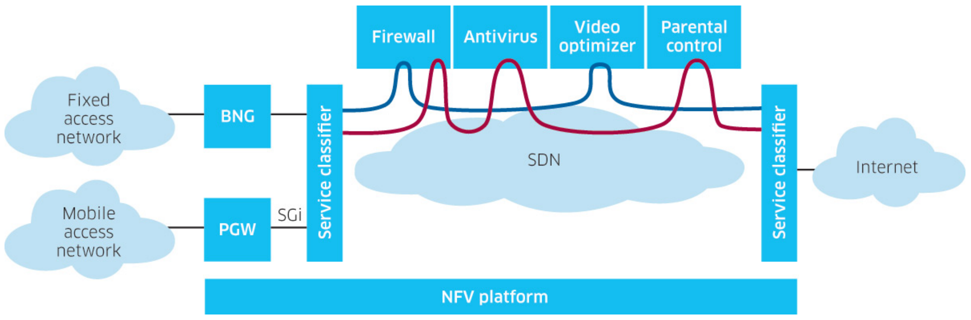


FIGURE 2.3: Service function Chaining (Foundation, 2015)

on, neither is it modified by the network topology nor the security policies.

A container comprises a complete runtime environment, which includes an application, together with all of its dependencies, libraries and other components, and configuration files, which are wrapped into a single package. Containerization abstracts differences in Operating system distributions and supporting infrastructure from the application platform and its dependencies (Rubens, 2017). The container technology is founded on two kernel features: namespaces and cgroups (Anderson et al., 2016) .

Multiple containerized apps can run on servers with an operating system, with each containerized application sharing the operating system kernel with other containerized applications. Containers are hence considered to as lightweight and utilize little server resources.

2.2.1 Container technology vs Hypervisor

A Hypervisor, sometimes known as a Virtual Machine Manager, is a program that enables different operating systems to share the same hardware resources (Bias, 2016a). Hypersivors has gained in popularity and is utilized in numerous data centers that employ VM technologies such as ESX, Xen, HyperV, etc. Virtual machine (VM) architecture and characteristics are distinct from container technologies. Due to their inherent characteristics, they require more resources for optimal performance than container technology. The variation in architecture is depicted in figure 2.5, while the tablet's network stack differs from that of other devices in figure 2.1. Since stated by (Anderson et al., 2016), the most common container technology is docker, as 94

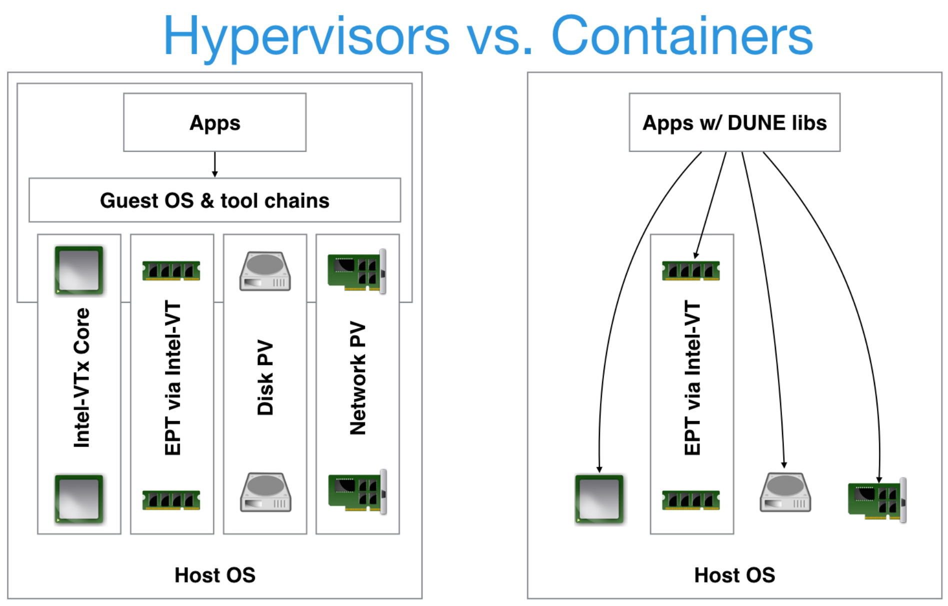
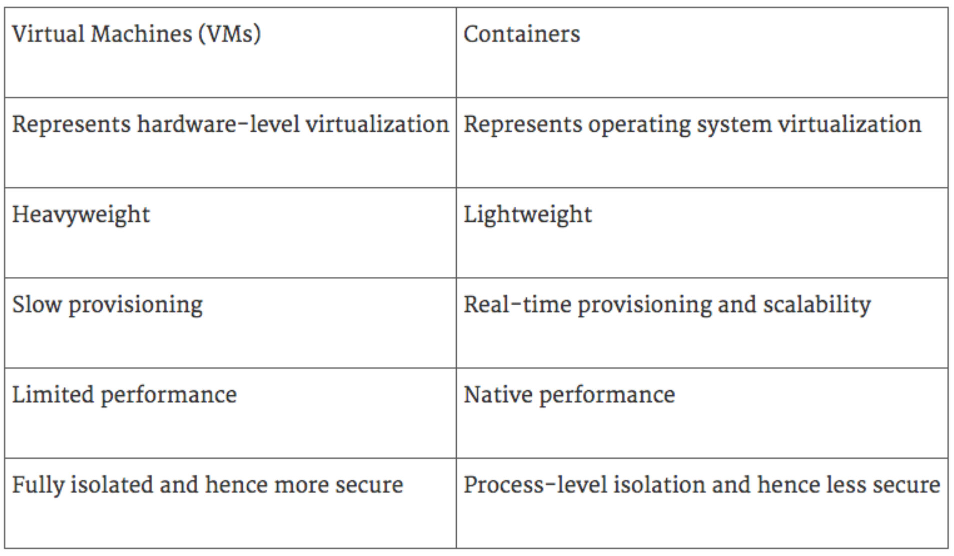


FIGURE 2.4: Container and Hypervisor Architecture (Bias, 2016b)



Virtual Machines (VMs)	Containers
Represents hardware-level virtualization	Represents operating system virtualization
Heavyweight	Lightweight
Slow provisioning	Real-time provisioning and scalability
Limited performance	Native performance
Fully isolated and hence more secure	Process-level isolation and hence less secure

TABLE 2.1: Difference between Container and Hypervisor (Reilly, 2017)

2.2.2 rkt container

rkt container is application container built for cloud-native environment. It provides a pluggable execution environment, making integration with other systems easier and more appropriate. The main execution unit of rkt is the pod, which aggregates one or more applications executing in a common context (OS, 2017) and enables users to install different configurations at both the pod-level and the per-application level. According to the rkt design, each pod executes in Linux or Unix computers in an isolated environment (OS, 2017). Rkt utilizes an open and standardized container, which fosters application innovation.

2.2.3 LXC container

LXC containers are meant to be utilized as full-fledged machines, as opposed to running a single application (Anderson et al., 2016). The LXC API simplifies the creation and management of systems and apps, but there is no straightforward way to obtain a container with a certain application already installed. LXC containers are regarded as falling between between chroot and a fully-fledged virtual machine (linuxcontainer.org, 2017)

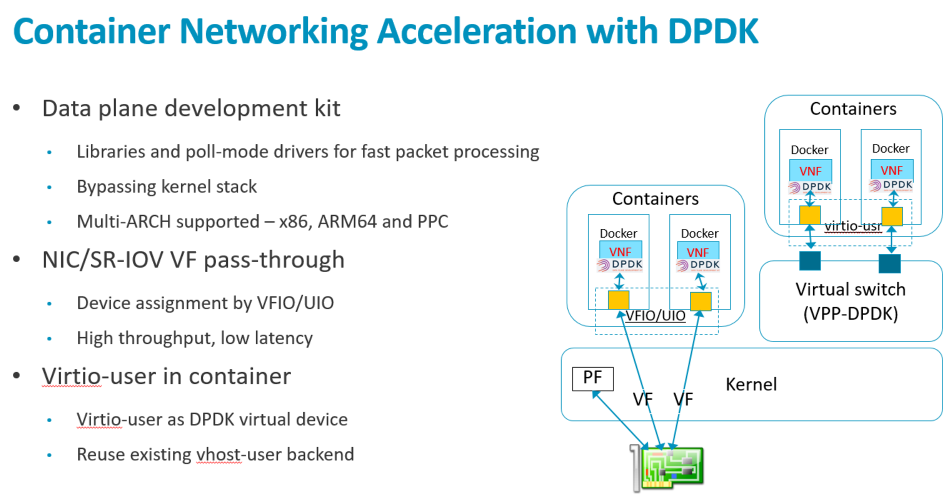
2.2.4 Docker container

According to a survey conducted by ClusterHQ and DevOps.com on (ClusterHQ and DevOps.com, 2016), Docker has been the most used container platform since its 2013 debut. Docker containers encase an application in a complete file system. They ensure that apps always run independent of the deployment environment. On a platform, containers share the kernel of their host. As a result, they are instantiated rapidly and require fewer resources. Similar to the two containers discussed previously, docker isolates applications executing on another container instance, hence enhancing system security. Docker offers advantages over containers explained on (Openstack, 2017)

Docker may collect the standard outputs and inputs of each container's running process for logging or direct interaction via a process-level API.

2.Image portability: The state of any Docker container may be saved as an image and shared via a centralized image registry, making them ideal for system integration. You could create images and store them in the Docker registry for later use.

Docker can automate the building of a container from the source code of an application. This facilitates the deployment of payloads to an OpenStack cluster as part of the development workflow (Openstack, 2017).



Container Networking Acceleration with DPDK

- Data plane development kit
  - Libraries and poll-mode drivers for fast packet processing
  - Bypassing kernel stack
  - Multi-ARCH supported – x86, ARM64 and PPC
- NIC/SR-IOV VF pass-through
  - Device assignment by VFIO/UIO
  - High throughput, low latency
- Virtio-user in container
  - Virtio-user as DPDK virtual device
  - Reuse existing vhost-user backend

FIGURE 2.5: Docker with DPDK

2.2.5 Docker performance

One of the primary benefits of Docker over VM is its ability to start instantaneously and utilize fewer process resources, making it far more effective in terms of performance. Therefore, networking performance is crucial when selecting a virtualization solution for the data center. Nevertheless, an IBM study (Felter et al., 2015) comparing Docker with KVM revealed that: Although containers have nearly no overhead, Docker has a number of performance issues. The performance of Docker volumes is noticeably superior than that of AUFS-stored files. Additionally, Docker's NAT incurs overhead for workloads with high packet rates. These characteristics reflect a trade-off between managerial simplicity and performance and must be evaluated on a case-by-case basis (Felter et al., 2015) This analysis reveals that Docker may not be significantly superior to other VM solutions, but it is significantly more efficient.

To improve the performance of the Docker instance, we are considering integrating DPDK into the configuration. Data Plane Development Kit is a product of the Linux Foundation. DPDK is a collection of libraries and drivers for bypassing the kernel network stack while processing packets. When DPDK is added to Docker, it increases packet processing performance and throughput, allowing Docker applications to spend more time on the data plane. Intel estimates that DPDK can improve packet processing by a factor of ten. As a result, Operator is able to reduce development costs and manufacture goods that reach the market more rapidly.

Figure 2.5 shows docker instances with DPDK.

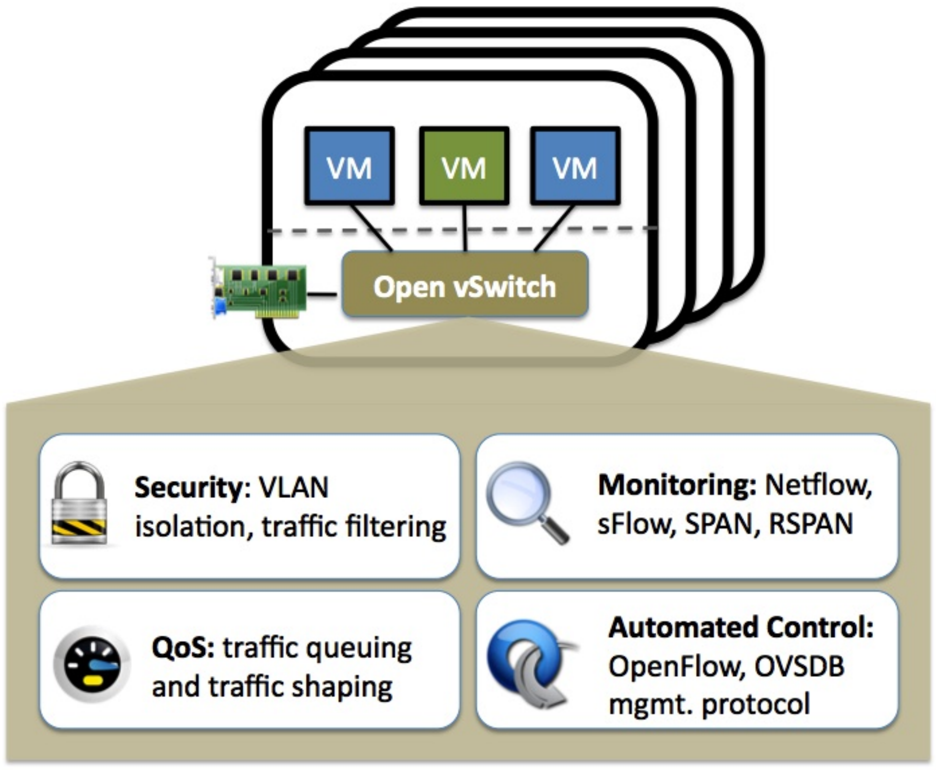

VM
VM
VM
Open vSwitch
Security: VLAN isolation, traffic filtering
Monitoring: Netflow, sFlow, SPAN, RSPAN
QoS: traffic queuing and traffic shaping
Automated Control: OpenFlow, OVSDB mgmt. protocol

FIGURE 2.6: OVS and its benefits

2.3 Open Virtual Switch OVS

OVS is an implementation of a virtual distributed multilayer switch that provides standard management interfaces. It is intended to provide network automation through programmatic extension, as well as standard management interfaces and protocols (such as NetFlow, sFlow, IPFIX, RSPAN, CLI, LACP, and 802.1ag) (Struye and Spinnewyn, 2017). It was initially designed to run on virtual machines, but later modifications enabled its implementation on containers, especially Docker. There are two components: the control plane and the data plane. The control plane is the switch's central intelligence and is responsible for discovery, routing, path computation, communication with other switches, etc. The control plane generates a forwarding/flow table that the data plane use to process incoming packets at line speed. (Struye and Spinnewyn, 2017). It has been incorporated into numerous virtual management systems, including as OpenStack, openQRM, and OpenNebula. We were interested in the OVS data plane for our research, therefore we deployed OVS using the Openstack software.

The figure depicts OVS with VM and its virtualization benefits.

2.4 Research Challenges

The study issues in obtaining high performance (low latency and high throughput) of Service Function Chaining using host-to-host architecture via the Internet, with one end on a dedicated cloud infrastructure such as Amazon AWS EC and the other end on any Internet-connected network facility. For the purpose of our experiment, we linked the AWS EC and the Host-in-lab server placed on the University Network. Our Architecture is similar to that of the vCPE use cases, with the exception that with vCPE, both connections belong to the same operator-provided network. Our objective is to comprehend the tradeoff between doing Service function chaining at the network's edge versus having it run on a dedicated cloud architecture.

Understanding the behavior of traffic when docker networks are implemented on host-to-host over the Internet is a further problem. Some state-of-the-art research has compared the behavior of other docker networking solutions, such as Linux bridge, OVS bridge, and macvlan networks, but not our chosen design. To correctly optimize VNFs or SFC to reduce latency while improving throughput for clients on different networks, we must utilize client-side resources. This study was the first step toward optimizing VNFs deployed over multiple Internet-connected hosts.

Chapter 3

Methodological Approach

We constructed two testbeds on which we conducted multiple experiments to determine the performance of docker container-based VNF chaining. Initially, we rented a cloud platform from Amazon AMS EC in the Frankfurt data center with the following specifications: 2 Core processor, 12GB Memory, and 1TB disk speed with multithreading capabilities. The instance of AWS infrastructure shipped with Ubuntu Server 16.04 and kernel version 4.13.

The second is a local testbed (with Dell server Precision T7500) placed in an Italian university's laboratory. This testbed is outfitted with two 2.6GHz Intel Xeon quad-core CPUs and 8GB RAM without Hyper-Threading. This test server will be referred to as "Host-in-the-Lab" throughout this text.

First, tiny and big packets are sent from the testbed in Italy to the AWS cloud in Frankfurt, which is equipped with DPI and firewall VNFs. The second scenario involved moving the VNFs to the local testbed and sending identical packets to an AWS cloud server hosting a web service.

We created and measured traffic using iperf3 and performed the experiment ten times. We chose iperf3 over other versions of iperf since version 1 and 2 were no longer being actively developed (might produce inaccurate results). Additionally, version 3 includes new parameters, including affinity, which allows the process to be pinned to a certain processor, and zero-copy, which decreases the number of system calls. Additionally, you could configure iperf3 to utilize the entire available bandwidth on the link.

Experiments were undertaken with single and, in certain instances, multiple traffic flows. We discovered that a single flow does not utilize all CPU cores. For instance, while measuring for efficiency, we utilize numerous concurrent flows to saturate the CPU core. We observed and assessed throughput and latency for the two most common networking solutions for docker containers for our two recommended design scenarios.

The basic blocks of topology are network namespaces. A network namespace is a networking stack instance that has its own interfaces and routing tables. We are aware that docker engine incurs minimal costs, but we think that their impact will be negligible. Additionally, we run experiments with zero-copy, which means we presume the testbeds are not used for any other processes. Our study approaches for joining namespaces together to form namespace chains. We implemented the two most prevalent Docker networking technologies, Linux Bridge and OpenvSwitch OVS bridge. The Linux Bridge includes a Linux kernel, but we loaded Openstack software via OVS. We also deployed Weave network to connect the two testbeds' host-to-host container namespaces

3.1 Cloud Computing Plaforms

According to information found on (Harvey, 2017), the cloud computing business is projected to reach over 200 billion dollars by 2020, which is around 70 percent of the 2017 projection. It is because operators want to reduce data center costs while delivering innovative solutions to customers.

Hence, Garners (Gartner, 2017) Computing is described by WEBSITE:33 as "a form of computing in which scalable and elastic IT-enabled capabilities are supplied as a service utilizing Internet technologies." Cloud computing uses several essential technologies to improve the productivity of software manufacturing. Docker container, an operating system virtualization that facilitates application development and enables consistent deployment across computing environments, is one of the essential technologies. Docker technology has been popular among data center operators since 2013, and it is anticipated (Livi et al., 2016) (ClusterHQ and DevOps.com, 2016)

The U.S. Department of Commerce's National Institute of Standards and Technology (NIST) categorized cloud computing into three cloud service models: software as a service (SaaS), platform as a service (PaaS), and infrastructure as a service (IaaS) (IaaS).

• SaaS: In this model, a third-party software vendor hosts the applications for the customers on the cloud infrastructure and makes them accessible via the Internet. Users can access the service over the web and are often charged on a monthly or annual basis. Typically, they are billed on a pay-as-you-go basis. AWS Elastic Beanstalk, Windows Azure, Heroku, Google App Engine, and Apache Stratos are examples of SaaS.

IaaS is a service paradigm that provides consumers with access to compute, storage, networks, and other infrastructure resources. This

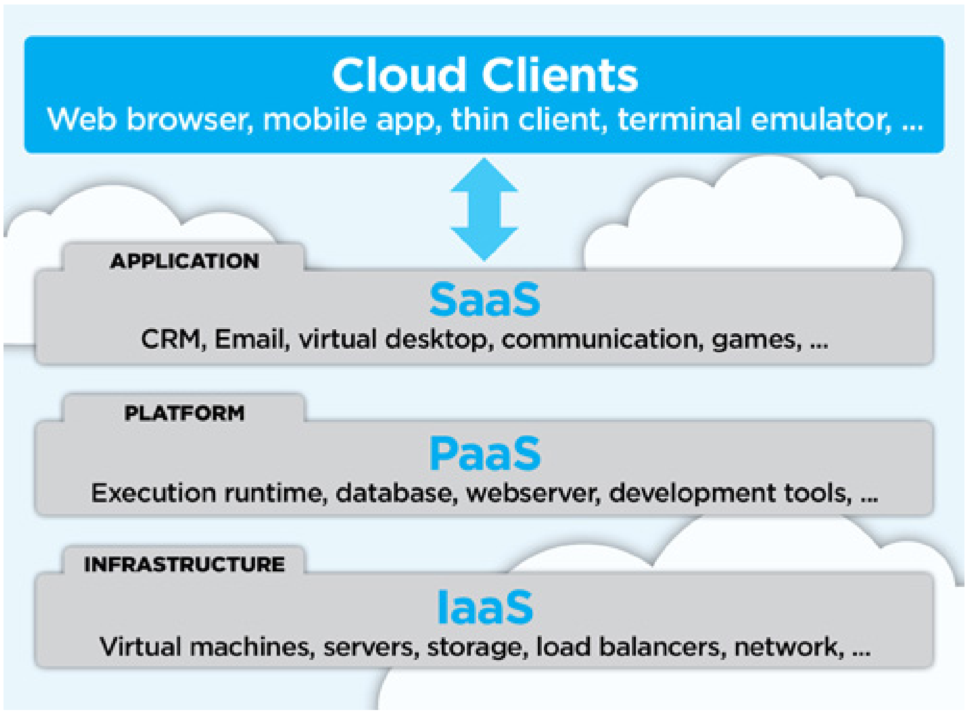

Cloud Clients
Web browser, mobile app, thin client, terminal emulator, ...
APPLICATION
SaaS
CRM, Email, virtual desktop, communication, games, ...
PLATFORM
PaaS
Execution runtime, database, webserver, development tools, ...
INFRASTRUCTURE
IaaS
Virtual machines, servers, storage, load balancers, network, ...

FIGURE 3.1: Cloud Computing Models

is comparable to having a server, storage equipment, and networking devices integrated for you, with the exception that they can be accessed and managed via the cloud. According to Garner's research (Harvey, 2017),Amazon AWS EC2 is the dominant IaaS provider with over 70

• PaaS: This service model provides a platform and environment for developers to create Internet-based applications and services. (Harvey, 2017) (Gartner, 2017).PaaS is a platform that enables clients to develop, run, and manage applications without the difficulty of creating and maintaining the infrastructure traditionally associated with developing and launching applications. Amazon Web Services, Microsoft Azure, IBM Bluemix, Google App Engine, Salesforce App Cloud, Red Hat OpenShift, Cloud Foundry, and Heroku are examples of PaaS providers.

The figure 3.1 is the summary of the Cloud computing service models.

3.1.1 Amazon Elastic Compute Cloud (EC2)

Amazon Web Service's Amazon Elastic Compute Cloud (EC2) is Amazon's IaaS cloud-computing platform (AWS). It enables clients to rent virtual computing resources for application execution. EC2 improves the scalability of application deployment by providing a web service through which a client user can boot an Amazon Machine (AMI) to launch an instance. An Instance may contain any pre-installed operating systems and apps. Docker is an instance of pre-installed software on EC2 instances. AWS is connected with Docker so that a user can retrieve any image from Docker's public registry. In Section 3.1.2, the docker workflow on AWS cloud is described. The instance can be launched, terminated, and upgraded at any moment.

Customers have discretion over the geographic placement of their instances, allowing for latency optimization and high degrees of redundancy. EC2 instances can be picked from Amazon's seven global data centers. Amazon's Elastic Compute Cloud (EC2) provides an application programming interface (API) that enables developers to select where an application runs physically (Cnet, 2008). As a testbed server, we selected an EC2 instance from the AWS data center in Frankfurt for our investigation.

3.1.2 Docker Workflow with AWS cloud

Amazon Web Services (AWS) offers a platform that has been connected with the docker public registry, making application development and deployment easier for their clients or users. A user can pull and push images to and from the Docker public registry to serve as a starting point for installing its own code (Merkel, 2014). A user may also generate or change a basic image and save it to a private registry or push it to a public register for other users to access. The scalability and elasticity of Amazon EC2 facilitates the deployment and administration of docker images for developers and operators.

The figure 3.2 describe step by step docker work flow on AWS cloud.

3.2 Docker Intra Host Network Device Virtualization

There are multiple ways to connect docker containers or namespaces to a physical interface or another container namespace. Linux bridge (which comes with the Linux kernel), OVS bridge, and Macvlan network are the most prevalent. According to (ClusterHQ and DevOps.com, 2016), we chose the Linux bridge and OVS for our investigation since they perform better.

3.2.1 Linux Bridge

Part of the Linux kernel, Linux Bridge is also known as Linux virtual Ethernet (veth) and is the simplest docker network option. "docker0" is the default docker bridge that is generated when Linux boots and

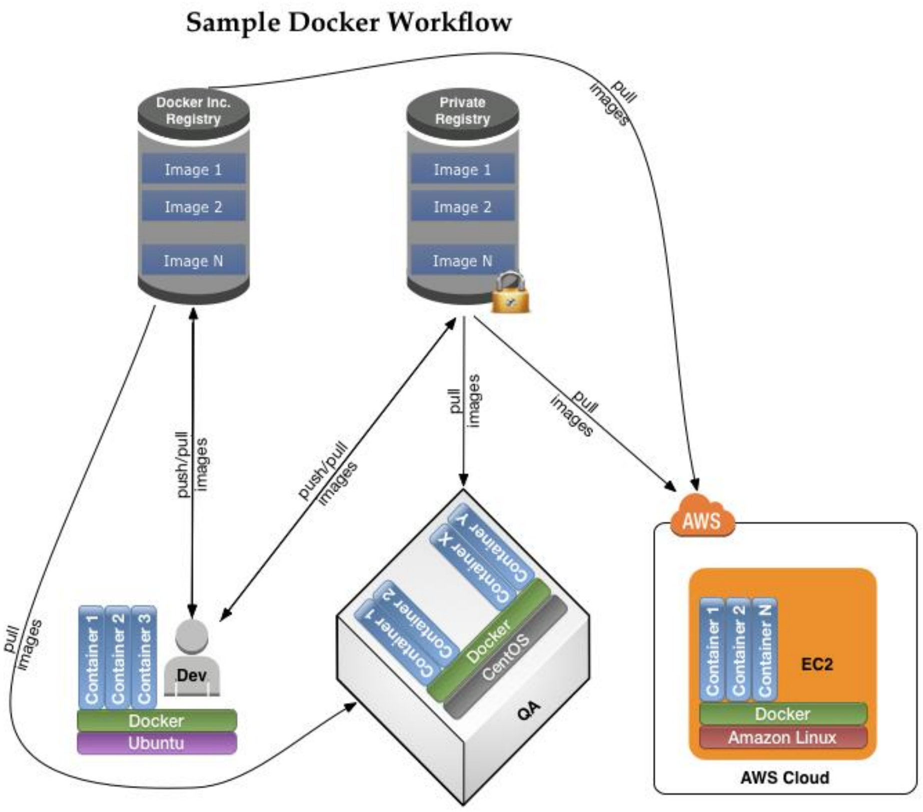


FIGURE 3.2: Cloud Computing Models

generates a virtual subnet on the docker host when a container is mounted. It moves packets between containers and their host. Additionally, it can join two or more containers on the same subnet or vlan. In the case of a container-to-container connection, it creates two veths, one of which is connected to the host and the other to the interfaces of other containers.

One Linux bridge could only support 1024 ports, limiting the scalability of Docker as we can only build 1024 containers per bridge. Figure 3.3 depicts the Linux bridge connection between two namespaces for containers.

3.2.2 OVS Bridge

OVS is a licensed multi-layer switch developed by Apache. It supports vlans and enables network automation programmatic extension. It is scalable since it may be set with as many ports as possible, allowing it to connect to many docker container namespaces. OVS is not included with the Linux kernel by default. It may be installed on Linux via third-party applications such as Openstacknamespaces.

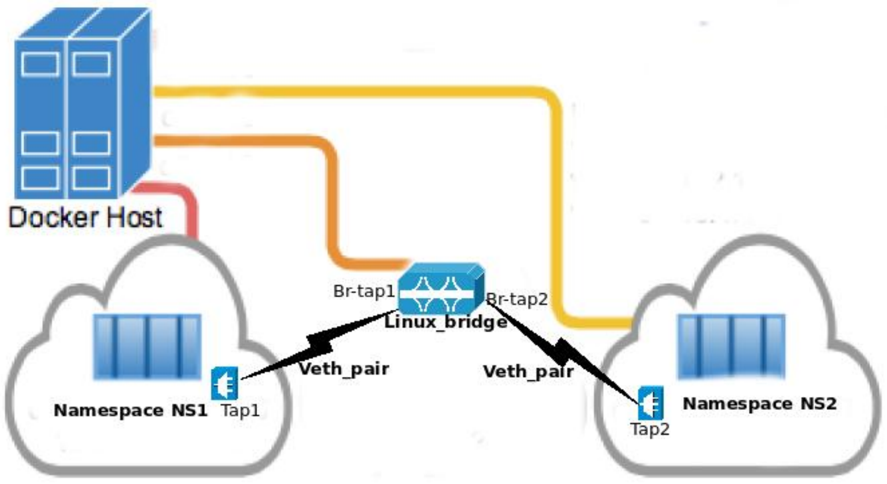


FIGURE 3.3: Linux Bridge Docker connection

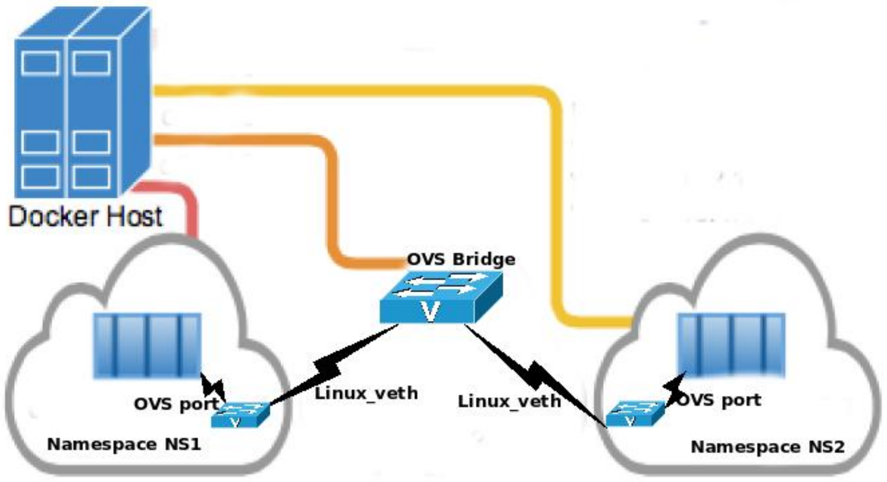


FIGURE 3.4: OVS bridge docker connection

It's connection similar to the Linux bridge as show in the figure 3.3.

3.2.3 Macvlan Network

Unlike the Linux bridge and OVS bridge approaches, Macvlan network operates differently. It connects the container interface to the physical interface of its parent. Using the macvlan driver, it links the container to the physical layer 2 network. Due to the nature of its connection, connecting two or more containers together requires a router (typically the physical interface of the host).

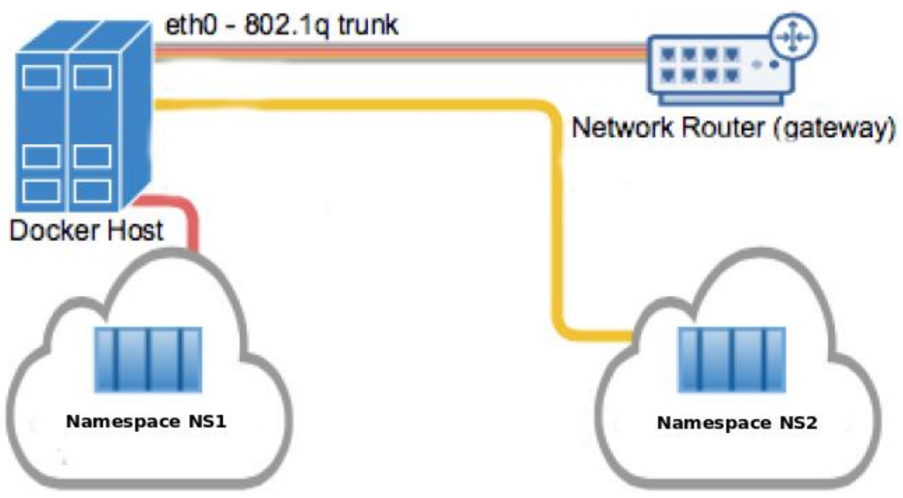


FIGURE 3.5: Macvlan docker connection

The Figure 3.5 shows the macvlan connection to docker containers

3.3 Docker Host-to-Host Network Device Virtualization

Various ways exist for connecting containers from host to host. In our investigation, a weave network was used to connect the two testbeds. However, additional connecting techniques exist, such as overlay network.

3.3.1 Weave Network

Weave network installs a vRouter on each host to begin a connection between containers running on separate hosts (Stacktutor, 2015). The vRouters utilize both TCP and UDP port 6783 to establish host-to-host connections. TCP establishes the connection topology, but UDP forwards traffic between containers using a proprietary tunneling protocol. Create a bridge between a container and a vRouter using Weave. The vRouter is not utilized for intra-host connections, but rather takes traffic from the bridge interface and forwards it to the peer vRouter on the other host via UDP.

Figure 3.6 demonstrate the architecture of weave connection.

3.3.2 Overlay Network

We use deploy OVS or Etcd for the host-to-host docker connection in this technique. It can be accomplished in two ways:

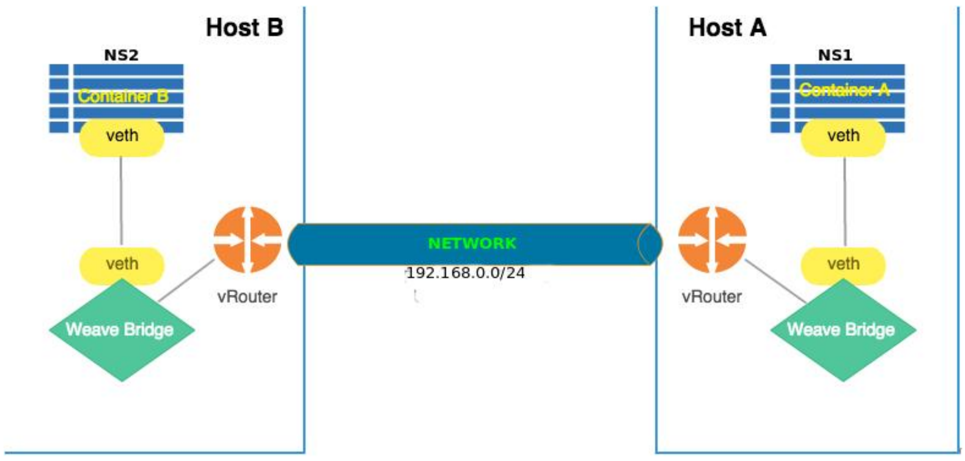


FIGURE 3.6: Weave Network on Host-to-Host docker connection

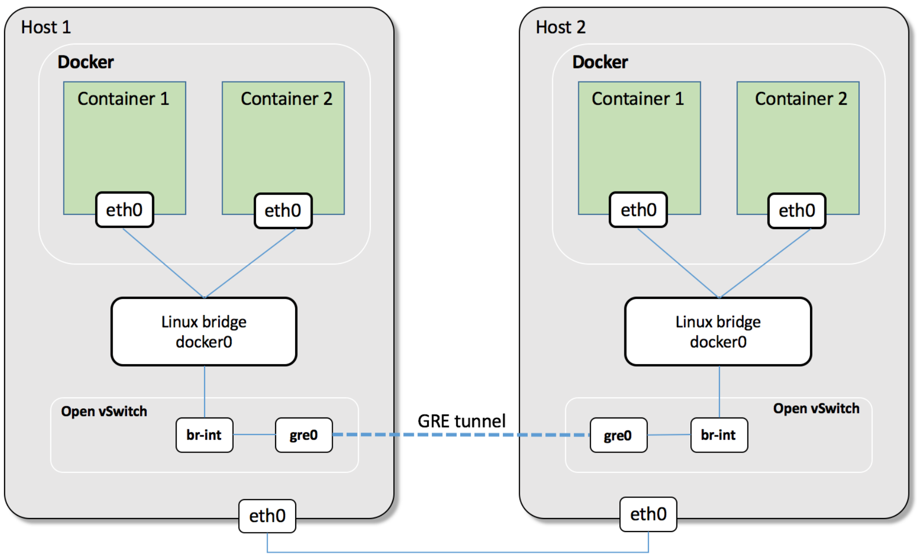


FIGURE 3.7: Overlay-OVS Network on Host-to-Host docker connection

•Connect docker0 by default to the ovs bridge
•Connect the container directly to the ovs bridge through the veth pair.

This method is not too reliable according observation in (k8, 2016). Figure 3.7 shows the architecture of the host-to-host connection of overlay.

Chapter 4

Architecture and Design

This chapter provides an overview of the architecture and design of the system, as well as its components. Our proposed architecture design is influenced by the virtualization use case for virtual Customer Premises Equipment (vCPE). The only distinction is that in our suggested system, communication between the consumer end and the cloud infrastructure is propagated across the Internet. We also describe how each component of our architectural design operates.

4.1 Evolved vCPE

Virtual CPE is one of the original Network Function Virtualization Concept use cases. vCPE substitutes the functionality of hardware appliances deployed at the customer's location for connectivity, while adding additional capabilities that increase Quality of Experience (QoE). Customers are able to remotely install, configure, and administer a number of features and functions that were previously delivered by a dedicated appliance. This allows businesses to outsource their mission-critical Information Systems infrastructure to cloud operators and migrate their data, apps, and storage to the cloud.

4.1.1 vCPE Implementation Options

Regardless of the deployment topologies represented in Figure 2.2, there are two primary modes of operation in the installation of vCPE.

- Option 1 begins virtualization initialization at the customer's location. Using pCPE that enables tunneling and security makes this possible. This option was derived from the standard CPE configuration. On the cloud network, network virtualization functionalities are implemented. Virtualization can be installed at the client site using a standalone white box, or it can be connected with the pCPE (isemag, 2017)

- Option 2 begins with a white box (typically a commercial, off-the-shelf COTS device) that hosts VNFs. In addition to VNFs, the

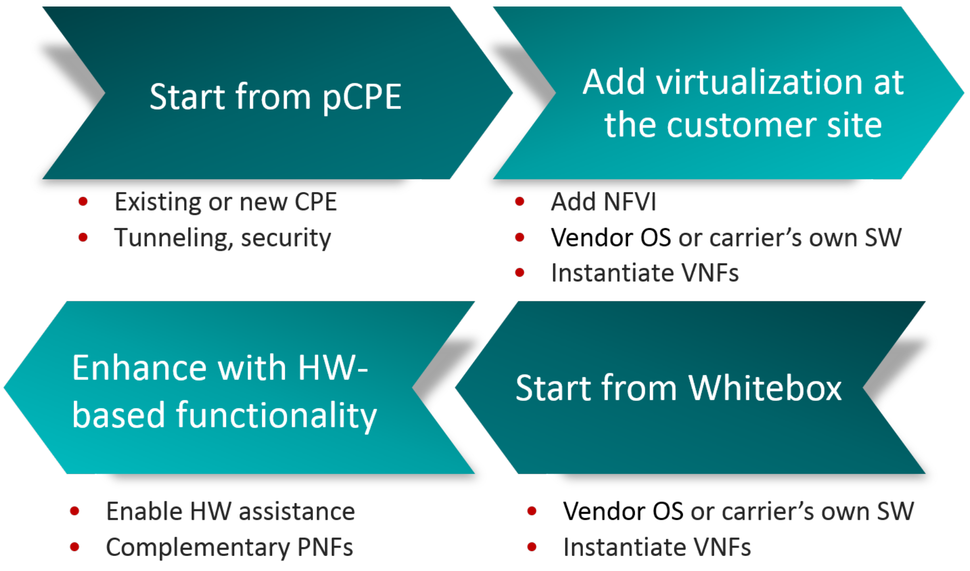


FIGURE 4.1: The two vCPE implementation options (isemag, 2017)

Cloud device can be configured to manage hardware-based functions such as performance acceleration and monitoring (firewall, Deep packet inspection, vRouting and vSecurity). This method necessitates a universal CPE at the client sites, with virtualization dispersed between the Cloud and the customer sites (isemag, 2017) .

The implementation possibilities are dependent on the client's requirements. Both choices are viable alternatives that can be examined. Certain applications, such as end-to-end encryption, WAN optimization, and testing tools, must be located on the customer's premises by definition. Figure 4.1 illustrates the vCPE deployment possibilities.

4.2 Proposed Architecture Design

Our Research architecture is modification of the vCPE except the link between the customers edge network and the cloud network is over the Internet. In the design, pCPE is the dedicated server located at the customer site and cloud site is located and deployed on Amazon Cloud platform.

4.2.1 Architecture Design

Figure 4.2 is the proposed architecture design for our project. Below are detail of the main components of the design.

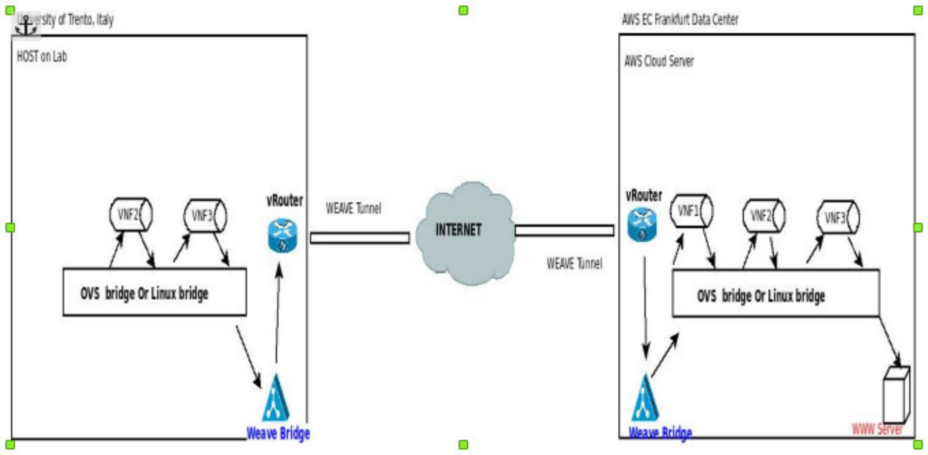
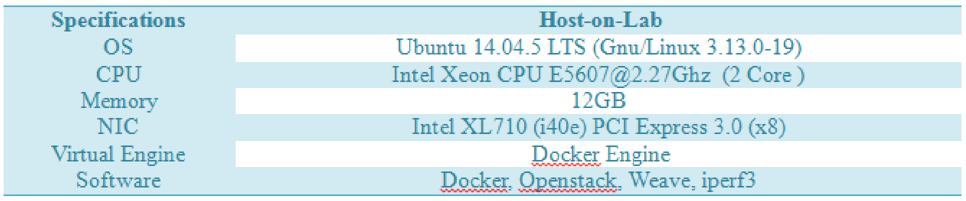


FIGURE 4.2: Proposed Architecture Design



Specifications	Host-on-Lab
OS	Ubuntu 14.04.5 LTS (Gnu/Linux 3.13.0-19)
CPU	Intel Xeon CPU E5607@2.27Ghz (2 Core)
Memory	12GB
NIC	Intel XL710 (i40e) PCI Express 3.0 (x8)
Virtual Engine	Docker Engine
Software	Docker, Openstack, Weave, iperf3

TABLE 4.1: Specification detail of Host-in-Lab

- Host on Lab:

This is virtualization host machine is located in a laboratory of University of Trento, Italy. Table 4.1 detail the specification of the Host server.

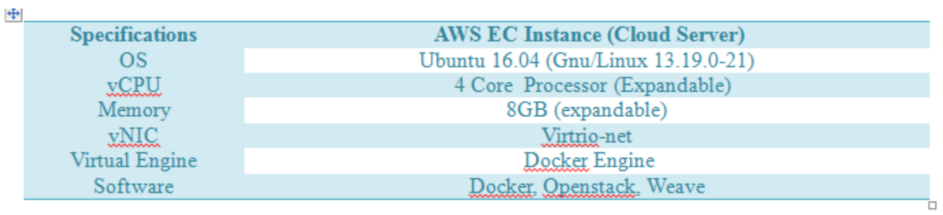
- Weave Bridge:

Weave bridge is created during the installation of weave software. The containers are connected to the bridge via veth pairs. Weave bridge is connected to the vRouter interface and captures traffic and forward it over UDP port to the peer vRouter at the other hosts. (Host-on-lab and AWS cloud server). vRouter is not used for intra-host communication (stacktutor, 2015).

- vRouter

vRouters communicate using both TCP and UDP on port 6783. TCP to communicate topology and UDP, using a tunneling, to forward data between the containers of different the two hosts (Host-on-Lab and the AWS cloud server). It communicate as well with Weave bridge.

- AWS Cloud Server



Specifications	AWS EC Instance (Cloud Server)
OS	Ubuntu 16.04 (Gnu/Linux 13.19.0-21)
vCPU	4 Core Processor (Expandable)
Memory	8GB (expandable)
vNIC	Virtrio-net
Virtual Engine	Docker Engine
Software	Docker, Openstack, Weave

TABLE 4.2: Specification detail of Cloud Host

AWS Cloud server is an "instance" of AWS Elastic cloud service. Table 4.2 provides specification detail of the AWS cloud server. We have also install and configured web server as our end service.

• VNFs
VNFs are virtual network functions. Each container instance is configured with specific VNF. They are chained together in way of static routes; each container is configured to forward traffic to another container. For our project, we used shorewall (iptable firewall) and Deep Packet Inspection (DPI) for our experiment.

4.3 Experimental Setup

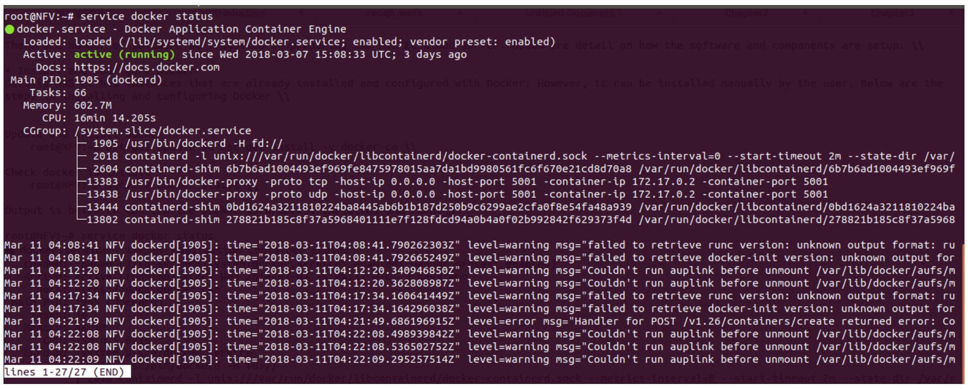
In every experiment configuration scenario, interfaces are added to containers using network namespaces. All of our experimental network virtualization methods (OVS, Linux bridge, and Weave network) use a virtual Ethernet device pair that is assigned to both the switch and the container. The interfaces of container namespaces resemble the physical interface of the host.

This experiment was conducted in accordance with the RFC 2544 (Jha, 2016) standards for evaluating transmission networks, network deployment, and network utilization. Iperf3 was used to create UDP or TCP packets for end-to-end transmission of traffic via VNF chains to a web server.

4.3.1 AWS EC Setup

There are number of software we installed in preparing the AWS EC Cloud server. Below are detail on how the software and components are setup.

• Install and configure Docker Usually AWS EC has instances that are already installed and configured with Docker. However, it can be installed manually by the user. Below are the steps in installing and configuring Docker



```
root@NFV:~# service docker status
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2018-03-07 15:08:33 UTC; 3 days ago
     Docs: https://docs.docker.com
 Main PID: 1905 (dockerd)
    Tasks: 66
   Memory: 602.7M
      CPU: 16min 14.205s
   CGroup: /system.slice/docker.service
           ├─ 1905 /usr/bin/dockerd -H fd://
           ├─ 2018 containerd -l unix:///var/run/docker/libcontainerd/docker-containerd.sock --metrics-interval=0 --start-timeout 2m --state-dir /var/
           ├─ 2604 containerd-shim 6b7b6ad1004493ef969fe8475978015aa7da1bd9980561fc6f670e21cd8d70a8 /var/run/docker/libcontainerd/6b7b6ad1004493ef969f
           ├─13383 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 5001 -container-ip 172.17.0.2 -container-port 5001
           ├─13438 /usr/bin/docker-proxy -proto udp -host-ip 0.0.0.0 -host-port 5001 -container-ip 172.17.0.2 -container-port 5001
           ├─13444 containerd-shim 0bd1624a3211810224ba8445ab6b1b187d250b9c6299ae2cfa0f8e54fa48a939 /var/run/docker/libcontainerd/0bd1624a3211810224ba
           └─13802 containerd-shim 278821b185c8f37a5968401111e7f128fdcd94a0b4a0f02b992842f629373f4d /var/run/docker/libcontainerd/278821b185c8f37a5968

Mar 11 04:08:41 NFV dockerd[1905]: time="2018-03-11T04:08:41.790262303Z" level=warning msg="failed to retrieve runc version: unknown output format: ru
Mar 11 04:08:41 NFV dockerd[1905]: time="2018-03-11T04:08:41.792665249Z" level=warning msg="failed to retrieve docker-init version: unknown output for
Mar 11 04:12:20 NFV dockerd[1905]: time="2018-03-11T04:12:20.340946850Z" level=warning msg="Couldn't run auplink before unmount /var/lib/docker/aufs/m
Mar 11 04:12:20 NFV dockerd[1905]: time="2018-03-11T04:12:20.362808987Z" level=warning msg="Couldn't run auplink before unmount /var/lib/docker/aufs/m
Mar 11 04:17:34 NFV dockerd[1905]: time="2018-03-11T04:17:34.160641449Z" level=warning msg="failed to retrieve runc version: unknown output format: ru
Mar 11 04:17:34 NFV dockerd[1905]: time="2018-03-11T04:17:34.164296038Z" level=warning msg="failed to retrieve docker-init version: unknown output for
Mar 11 04:21:49 NFV dockerd[1905]: time="2018-03-11T04:21:49.686196915Z" level=error msg="Handler for POST /v1.26/containers/create returned error: Co
Mar 11 04:22:08 NFV dockerd[1905]: time="2018-03-11T04:22:08.498939842Z" level=warning msg="Couldn't run auplink before unmount /var/lib/docker/aufs/m
Mar 11 04:22:08 NFV dockerd[1905]: time="2018-03-11T04:22:08.536502752Z" level=warning msg="Couldn't run auplink before unmount /var/lib/docker/aufs/m
Mar 11 04:22:09 NFV dockerd[1905]: time="2018-03-11T04:22:09.295257514Z" level=warning msg="Couldn't run auplink before unmount /var/lib/docker/aufs/m
lines 1-27/27 (END)
```

FIGURE 4.3: Proposed Architecture Design

Update and Install Docker
apt-get update
apt-get install -y docker-ce

Check docker is install
service docker status

Output is displayed on Figure 4.5 to confirm the docker is properly installed and running

- Create VNFs namespace containers with Linux Bridge

1. Create Linux Bridge
brctl addbr br0
ip link set br0 up

2. Add Interface to bridge
brctl addif br0 em0

3. Configure ip address and default route ip addr add 10.10.0.0/24 dev br0
ip route add default via 10.0.0.1 dev br0

4. Create Web Server Container
docker run -itd –name=web web-new

5. Create DPI VNF with container docker
Pull DPI (ndpi) form the Docker registry
docker pull lucaderi/ntopng-docker

6. Create DPI VNF container
docker run -itd –name=dpi lucaderi/ntopng-docker

7. Create Firewall VNF with container docker
pull Firewall (shorewall) from the docker external registry

```
docker pull cjntaylor/shorewall:latest
```

8. Create Firewall VNF container

```
docker pull cjntaylor/shorewall:latest
```

9. Create a veth interface pair

```
ip link add web-int type veth peer name web-ext0
brctl addif br0 web-ext0
```

10. Repeat the point 8 to connect the rest of the VNF chains

• Create VNFs namespace containers with OVS bridge

1. Install OVS

```
apt-get install openvswitch-switch
```

2.Create an OVS bridge

```
ovs-vsctl add-br ovs-br0
ip link set br0 up
```

3. Set ip address and default route

```
ip addr add 10.0.0.0/24 dev ovs-br0
ip route add default via 10.0.0.1 dev br0
```

4. Assume we created VNFs using the same commands from the points 4 to 7.

5. Connect the VNF containers to OVS bridge

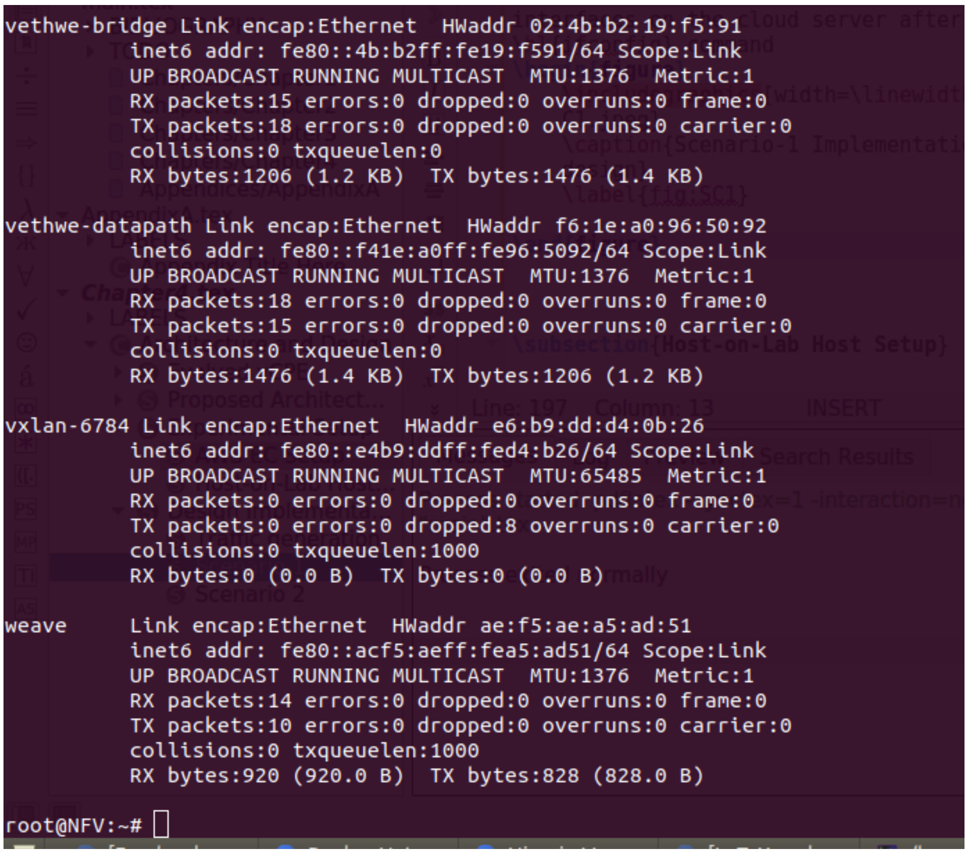
```
ovs-docker add-port ovs-br0 eth1 web –ipaddress=10.0.0.2/24
ovs-docker add-port ovs-br0 eth1 dpi –ipaddress=10.0.0.3/24
ovs-docker add-port ovs-br0 eth1 firewall –ipaddress=10.0.0.4/24
```

• Implement Weave on Host-on-Lab and AWS Cloud server to connect

1. Install Weave on Hosts
Download Weave package from weave repository and save in /usr/local/bin/weave
chmod a+x /usr/local/bin/weave (Stacktutor, 2015)

2. Start weave Network
weave launch

Figure 4.4 shows the detail of the Weave interfaces on the cloud server after ifconfig command



```
vethwe-bridge Link encap:Ethernet  HWaddr 02:4b:b2:19:f5:91
          inet6 addr: fe80::4b:b2ff:fe19:f591/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1376  Metric:1
          RX packets:15 errors:0 dropped:0 overruns:0 frame:0
          TX packets:18 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1206 (1.2 KB)  TX bytes:1476 (1.4 KB)

vethwe-datapath Link encap:Ethernet  HWaddr f6:1e:a0:96:50:92
          inet6 addr: fe80::f41e:a0ff:fe96:5092/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1376  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1476 (1.4 KB)  TX bytes:1206 (1.2 KB)

vxlan-6784 Link encap:Ethernet  HWaddr e6:b9:dd:d4:0b:26
          inet6 addr: fe80::e4b9:ddff:fed4:b26/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:65485  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:8 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

weave     Link encap:Ethernet  HWaddr ae:f5:ae:a5:ad:51
          inet6 addr: fe80::acf5:aeff:fea5:ad51/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1376  Metric:1
          RX packets:14 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:920 (920.0 B)  TX bytes:828 (828.0 B)

root@NFV:~#
```

FIGURE 4.4: Weave Interfaces

4.3.2 Host-in-the-Lab Setup

The same installation procedure were followed to install Weave, OVS and Docker.

However on the Host-in-the-lab, we compiled Data Plane Development kit (DPDK) for docker engine.

• Implementation DPDK with Docker
1. Compile DPDK
make install RTE_SDK=‘pwd‘ T=x86_64-native-linuxapp-gcc

2. Develop a dockerfile and save the file
cat «EOT » Dockerfile
FROM ubuntu:latest
WORKDIR /usr/src/dpdk
COPY . /usr/src/dpdk
ENV PATH
"$PATH:/usr/src/dpdk/x86_64-native-linuxapp-gcc/app/"
EOT

3. Build a docker image
docker build -t dpdk-app-testpmd

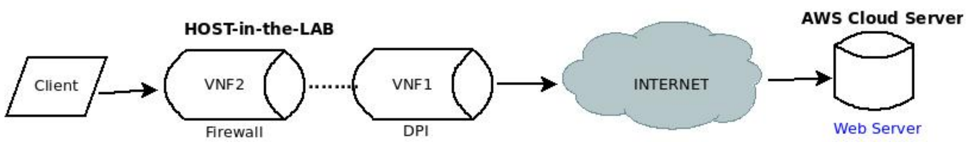


FIGURE 4.5: Scenario-1 Implementation of design

4. Start a testpmd on the host

```
(testpmd) -l 0-1 -n 4 –socket-mem 1024,1024
–vdev 'eth_vhost0,iface=/tmp/sock0' –file-prefix=host –no-pci – -i
```

5. Start a container instance with a virtio-user port

```
docker run -i -t -v /tmp/sock0:/var/run/usvhost
-v /dev/hugepages:/dev/hugepages
dpdk-app-testpmd testpmd -l 6-7 -n 4 -m 1024 –no-pci
–vdev=virtio_user0,path=/var/run/usvhost
–file-prefix=container
– -i –txqflags=0xf00 –disable-hw-vlan
```

4.4 Design Implementation

In this section, we discussed our project's two architectural options. We examined the experiment's diverse traffic production and measurement technologies.

4.4.1 Scenario 1

The first scenario, depicted in Figure 4.5, places virtualization closer to the customer (Host-on-Lab). Chains of Virtual Network Functions are configured. Traffic is filtered by the firewall in VNF1 and then deep inspected by VNF2 before it exits the Host-on-Lab Network and connects to the web server in the AWS cloud via the Internet. The iperf3 generates traffic that is routed through the VNFs in chains where actions are performed in the form of Service function chaining (SFC) before reaching the web server. In this scenario, virtualization begins at the customer's site and is completed on a dedicated physical hardware server.

4.4.2 Scenario 2

Figure 4.6 portrays Scenario 2; however, in this scenario, virtualization is located in the AWS cloud network. Traffic is generated from the Host-on-lab network to the Cloud server via the Internet. Traffic is then routed through the virtual function chains before reaching the web server.

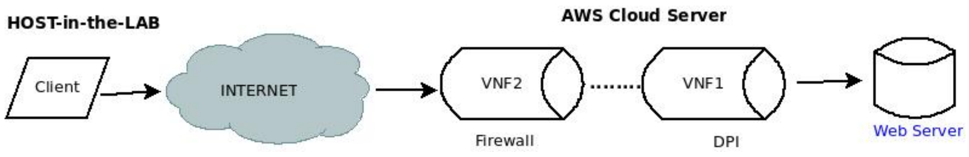


FIGURE 4.6: Scenario-2 Implementation of design

Performance assessments are carried out, and the two scenarios are compared.

Chapter 5

Performance Evaluation and Analysis

Our experimental analysis focuses on the overheads associated with the evaluated networking methods. In addition, we evaluate workload indicators like as throughput and latency to determine the entire overhead of virtualization. In addition, we analyzed two situations in which network operations were executed on Cloud infrastructure and VNFs were transferred to a server closer to the network's edge (Host-in-the-Lab server). In both cases, latency and throughput measurements were taken to determine the cost-benefit analysis of putting VNFs in the cloud vs at the network's edge.

Using performance cpufreq governor, our testbeds' power management was disabled. Cpufreq is a Linux kernel-integrated program for CPU frequency scaling. Operating systems can adjust CPU frequency up or down to preserve energy. Because Docker containers were not bound by cgroups, they might consume all available resources. For the measurement of the metrics, zero-copy was enabled, allowing the CPU to conserve energy and memory for the experiment alone. All docker images use Ubuntu 16:04 64-bit with Linux kernel 4.13.0-32-generic. The specifications of our testbed's servers are provided in detail in tables 4.1 and 4.2.

This project evaluation analysis is a two-pronged experiment; first, we assessed the common Docker network options, and second, we evaluated network function virtualization in two separate design scenarios. Virtualization of standard server hardware on the Host-in-lab server and network function virtualization on the AWS cloud server.

5.1 Docker Network methods

The objective of the evaluation results and analysis is to comprehend the traffic pattern performance features of the two docker network approaches under consideration. The implementation of the methods has a substantial effect on the cloud server's VNF performance.

This experiment was conducted using the AWS cloud metal instance described in Table 4.1.

In each implementation, both throughput [Gbps] and latency/jitter [ms] were assessed. We ran two throughput trials, one per parallel stream and the other per packet size. Additionally, we ran two experiments on latency: latency per packet size and latency per container count. Each test stream is repeated five times, and the average value is calculated.

5.1.1 Evaluated Results

• Throughput Metrics :
In this experiment, as represented in Figure 5.1, several streams of parallel streams of 10-megabyte packets were transmitted in an attempt to overwhelm the network. In this investigation, we compared the gains made by the two networking technical solutions utilizing Throughput. Figure 5.1 depicts the outcomes of the experiment. A client running on the external host (Host-in-the-lab) forwarded TCP packets to a cloud server executing the virtualization.

We discovered that both Linux bridge and OVS gradually rise as the number of parallel streams grows. However, between eight and ten parallel streams, the graph becomes nearly stable. This is because the parallel stream between 1 and 8 did not reach the connection pipe's maximum bandwidth. The link was capable of exceeding the maximum of 10 parallel streams. TCP-window-scale may be a factor that impacts streams. If TCP-window-scale is enabled, some packets may be dropped if the advertised receive window is smaller than what the sender is transmitting.

In Figure 5.2 (throughput per packet size), we also discovered that, on average, the Linux bridge had a better throughput increase than the OVS-bridge. The average gain of the Linux bridge approach is approximately 20% greater than that of OVS. This is due to the fact that Linux bridge includes the Linux kernel, whereas OVS software installed on the host incurs some overhead when processing packets.

• **Latency Metrics :**

In the Latency test, UDP packets were delivered from the Host-in-lab to the AWS cloud-based Virtualization server. Figure 5.3 depicts the relationship between delay and packet size (10kb,20kb,30kb,40kb,50kb and 60kb). We examined the cost of sending a packet to a Docker container running on a Linux bridge vs an OVS network. Figure 5.3 demonstrates that the Linux bridge has comparatively lower latency

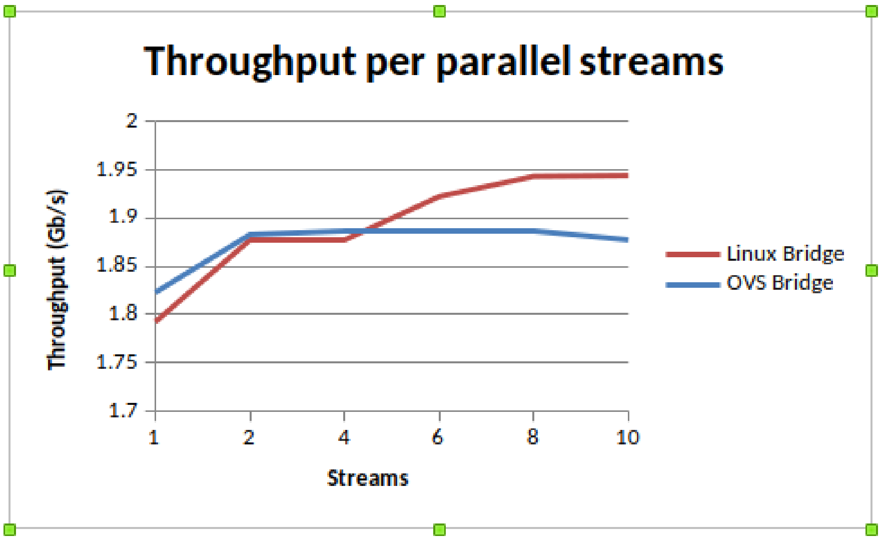


FIGURE 5.1: Measure of throughput per Parallel stream]

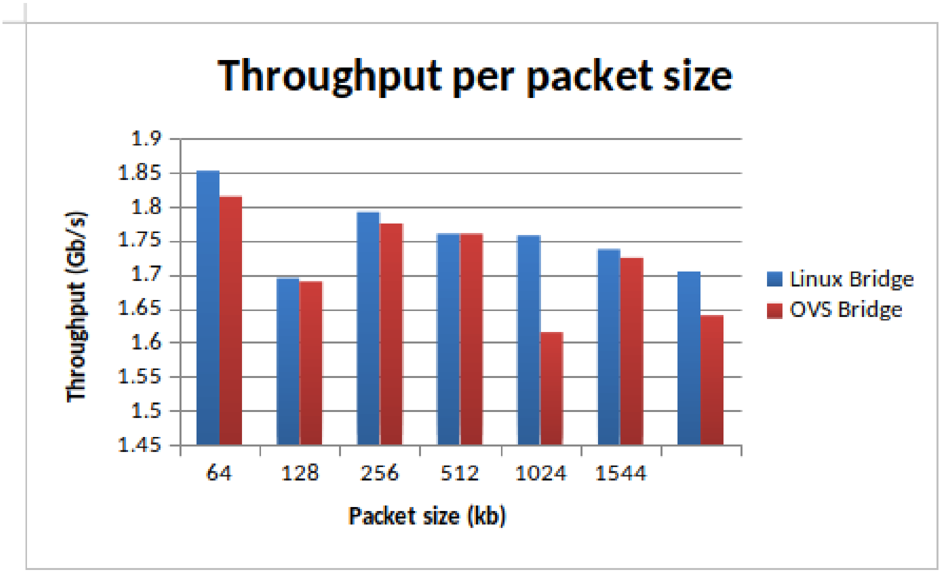


FIGURE 5.2: Measure of throughput per packet size

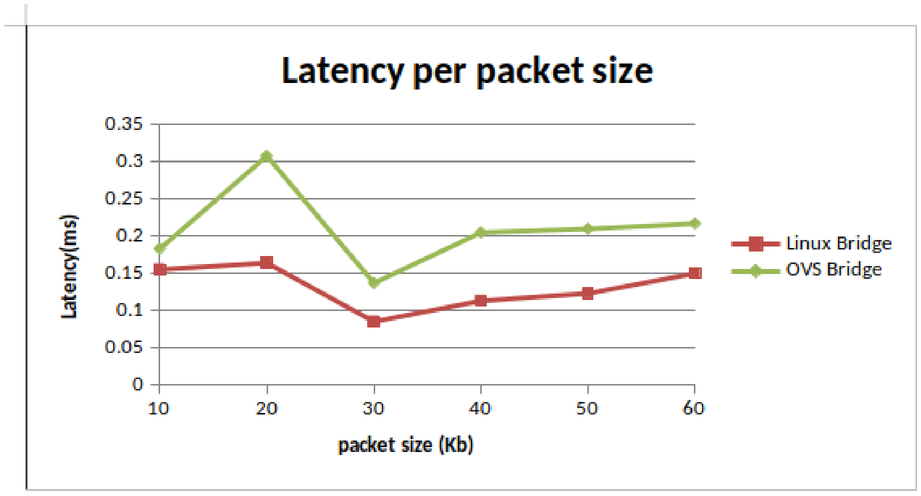


FIGURE 5.3: Measure of throughput per packet size

than the OVS bridge. OVS has a (25 percent to 65 percent) increase in latency over the Linux bridge. This conclusion is consistent with the findings of other researchers in (Livi et al., 2016) (Jha, 2016).

Large UDP packets were delivered to the two networking technologies for a second latency test to ascertain the number of docker containers supported by each technology. Figure 5.4 illustrates the progressive increase in latency for Linux and OVS until the number of containers exceeds 100. Observations indicate that there is an 82 percent increase in latency from 100 to 150 for the Linux bridge. The rapid growth of Linux bridges validates the theoretical assumption (Varis, 2012). that a single Linux bridge can support up to 124 ports, which restricts the scalability of deployment. In contrast, the latency for OVS gradually increases by 4% on average. This explains why data centers favor OVS bridges over Linux bridges. Therefore, there is a trade-off between the number of containers to deploy in OVS and Linux bridge's capacity to grow to high throughput and low latency for a small number of containers. In addition to scalability, OVS also offers programmatic extension and supports huge VLANs.

- **Jitters metrics** :

In this test, the delay variance of packets delivered from the external host (Host-in-the-lab) to a receiver on the virtualization host using each network mode was measured. We measured 20 samples of UDP packets with varying bandwidth or link capacity and documented jitters for each bandwidth value. Tables 5.1 and 5.2 detail the jitters, percentage loss, and Out-of-order datagrams that were lost during the OVS and datagram processes, respectively.

As the out-of-order datagram loss increases from 120MB to 160MB,

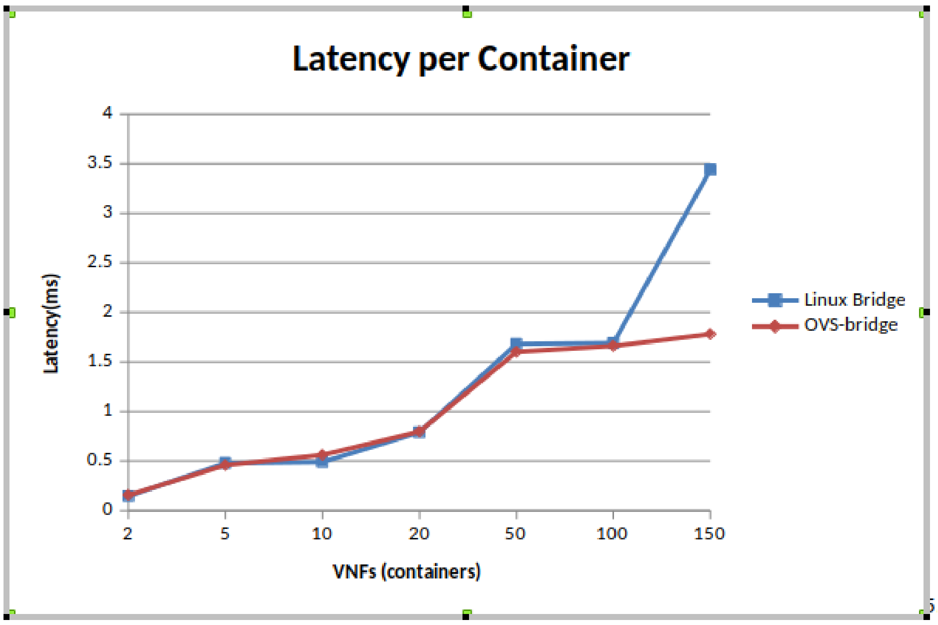
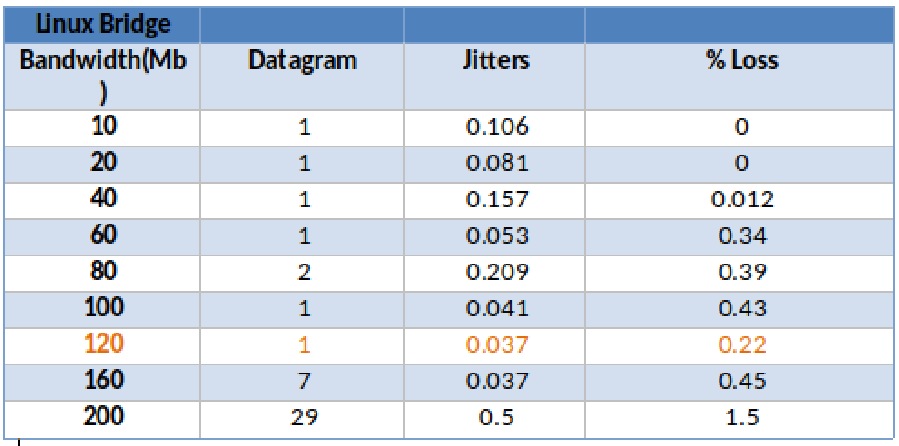


FIGURE 5.4: Measure of Latency per number of containers

OVS Bridge			
Bandwidth(Mb)	Datagram	Jitters	% Loss
10	1	0.126	0
20	1	0.145	0
40	1	0.062	0
60	4	0.092	0.08
80	3	0.112	0.16
100	2	0.02	0.31
120	1	0.057	1.2
160	21	0.03	0.79
200	27	0.045	1.3

TABLE 5.1: Table delay variation of OVS



Linux Bridge			
Bandwidth(Mb)	Datagram	Jitters	% Loss
10	1	0.106	0
20	1	0.081	0
40	1	0.157	0.012
60	1	0.053	0.34
80	2	0.209	0.39
100	1	0.041	0.43
120	1	0.037	0.22
160	7	0.037	0.45
200	29	0.5	1.5

TABLE 5.2: Table delay variation of Linux Bridge

the percentage of datagrams lost increases for both the Linux bridge and the OVs bridge. Figure 5.5 demonstrates that data loss for OVS is greater than the average of 55% for this Linux bridge. This is because of the additional application overhead OVS software has.

5.2 Architecture Scenarios

By altering performance requirements and offered loads, we examine the performance of two distinct scenarios. Figures 4.5 and 4.6 depict the two architecture scenarios under consideration.

5.2.1 Evaluated Result

- **Throughput Measurement:**

Figure 5.6 depicts the outcome of a test in which TCP packets with varying frame sizes were transferred from Host-in-the-lab to the web-hosted web server via the VNF chain for both situations (Virtualization at the Host-in-the-lab or the Cloud server). Figure 5.6 demonstrates that there is no substantial difference in throughput between the two circumstances.

Figure 5.7 demonstrates that when **DPDK** was deployed for Docker containers on the Host-in-the-lab, scenario 1 with the Host-in-the-lab had a much greater throughput than scenario 2 with the cloud server. Figure 5.7 demonstrates that virtualization on Host-in-lab servers has an average 18% throughput than cloud servers. This is because the introduction of DPDK enables faster packet processing by bypassing the kernel's network stack.

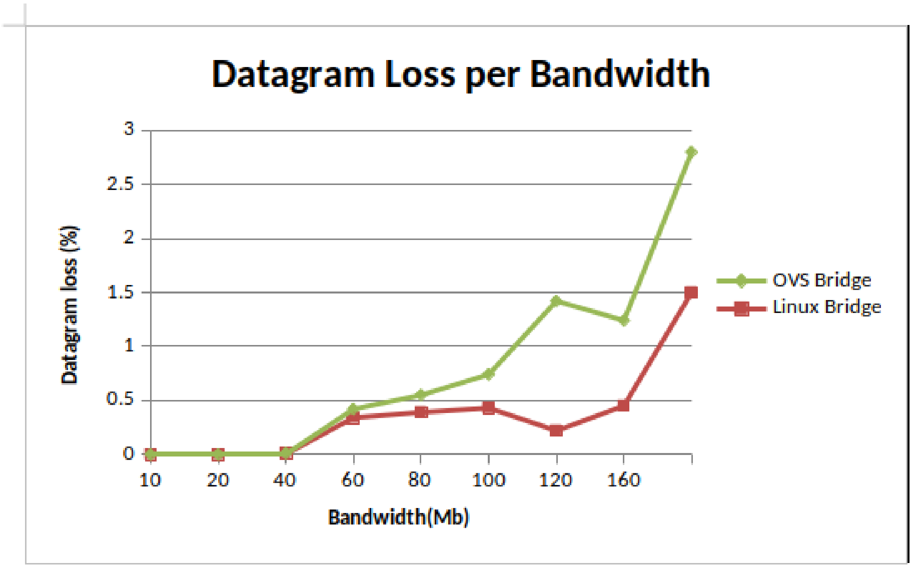


FIGURE 5.5: Measure of percentage data loss vrs Bandwidth

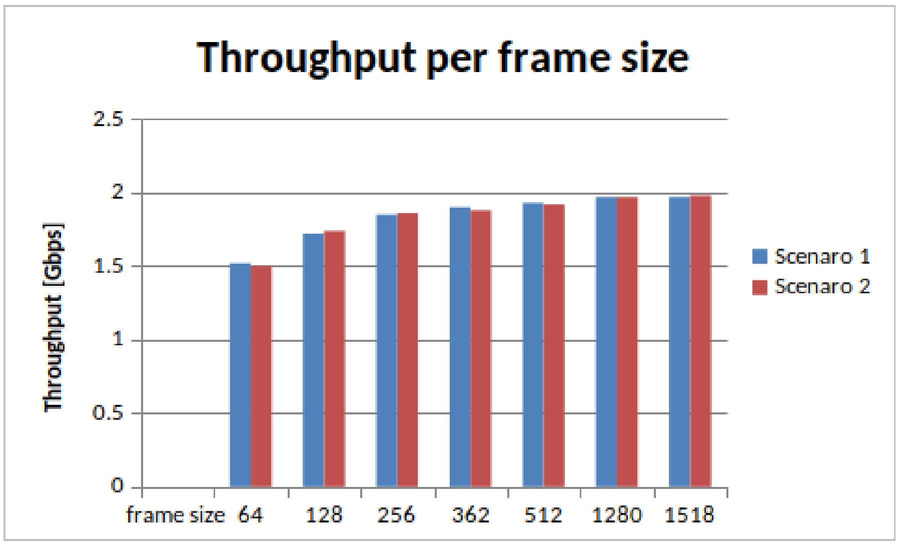


FIGURE 5.6: Measure of throughput per frame size

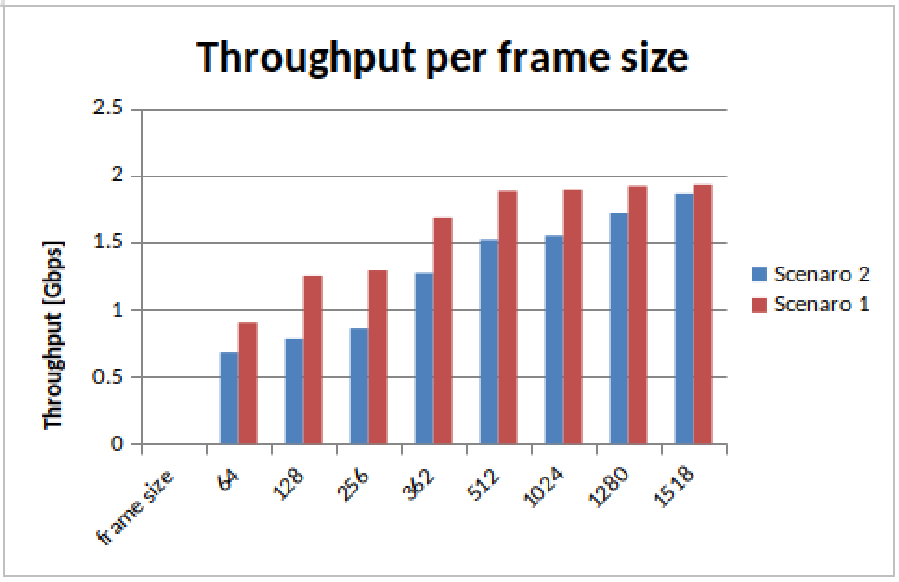


FIGURE 5.7: Measure of throughput per frame size

- **Latency Measurement:**

Again, we measure latency by sending different frame sizes to the web server via VNF chains for 60 seconds. Figure 5.8 shows that after introducing DPDK to the host-in-the-lab containers, scenario 1 has lower latency than scenario 2.

Chapter 6

Conclusion

6.1 Conclusion

Docker Container has been the most popular container technology for virtualization since its introduction in 2013. Docker, like other container technologies, is lightweight, which means it can run applications instantly and uses fewer resources than a virtual machine. As a result, Docker is critical in virtualizing large-scale micro-service-based applications, and selecting the best performing networking technology could result in cheaper and faster virtual function deployment.

In this study, we analyzed the networking performance of Docker utilizing the two most effective networking solutions, Linux bridge and OVS bridge. We evaluate the performance of the two solutions by running one or more containerized VNFs on the host. We noticed that the Linux bridge provides superior performance in terms of latency and throughput. For instance, the average advantage of Linux bridge over OVS bridge is approximately 18%. Again, OVS provides inferior performance when measuring latency; it is at least 20 percent slower than the Linux bridge.

However, when we examined delay per number of containers for each network bridge, we discovered a dramatic increase in latency when the number of containers on the Linux bridge network exceeds 100. It illustrates that the number of containers any Linux bridge can support is restricted. Therefore, there is a trade-off between the superior performance for a limited number of containers and the scalability of OVS. This explains why large-scale microservice-based applications operate more efficiently on OVS switches than Linux bridge.

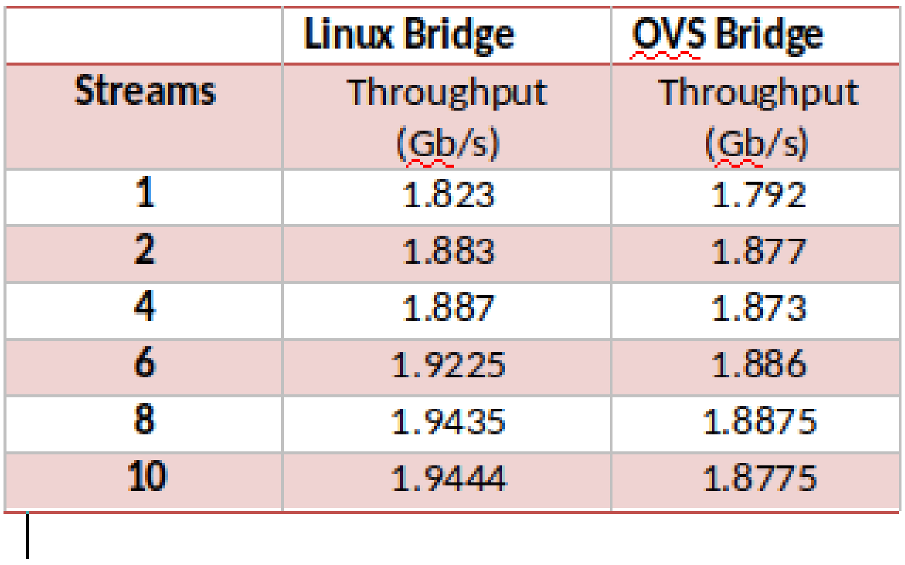
In the second phase of the project, we devised two scenarios; one scenario implements virtualization at the network's edge (Host-in-the-lab), while the other scenario implements virtualization on the AWS cloud. Performance is superior in the first situation where virtualization is located at the network's edge. The scenario 1 throughput was 18% greater due to the addition of Data Plane Development

Kit (DPDK) to the Host-in-the-lab, which expedites packet processing.

These results provide assurance to network experts who wish to install high-performance solutions at the service provider edge utilizing technologies such as low-end COTS servers and DPDK. However, there are trade-offs; virtualization at the network's edge may be more expensive than renting a cloud platform. In addition, the cloud platform's auto-scaling of resources allows memory to be updated on the fly.

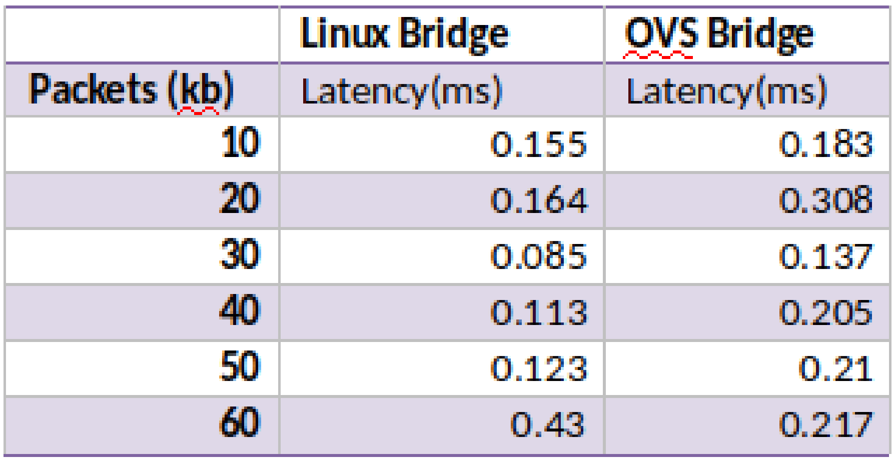
Appendix A

Experiment results:Network Technologies



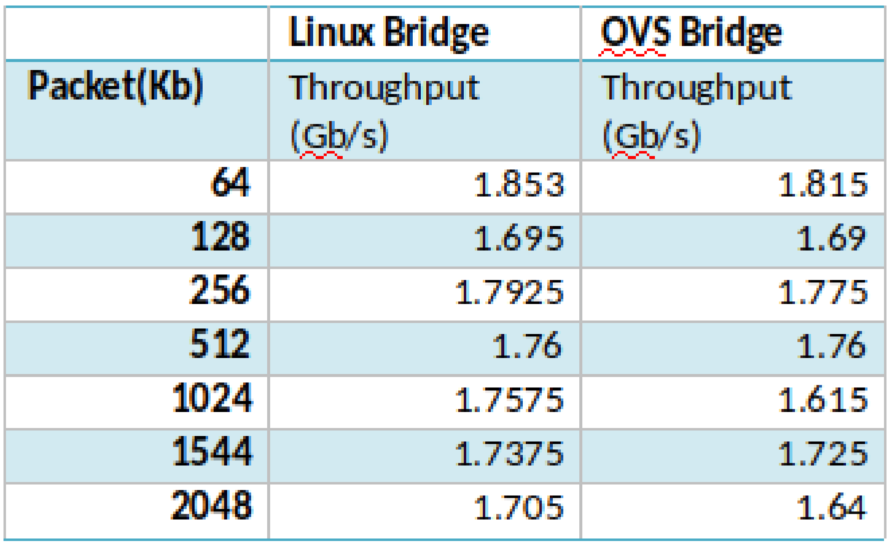
	Linux Bridge	OVS Bridge
Streams	Throughput (Gb/s)	Throughput (Gb/s)
1	1.823	1.792
2	1.883	1.877
4	1.887	1.873
6	1.9225	1.886
8	1.9435	1.8875
10	1.9444	1.8775

TABLE A.1: Throughput measure for different parallel streams



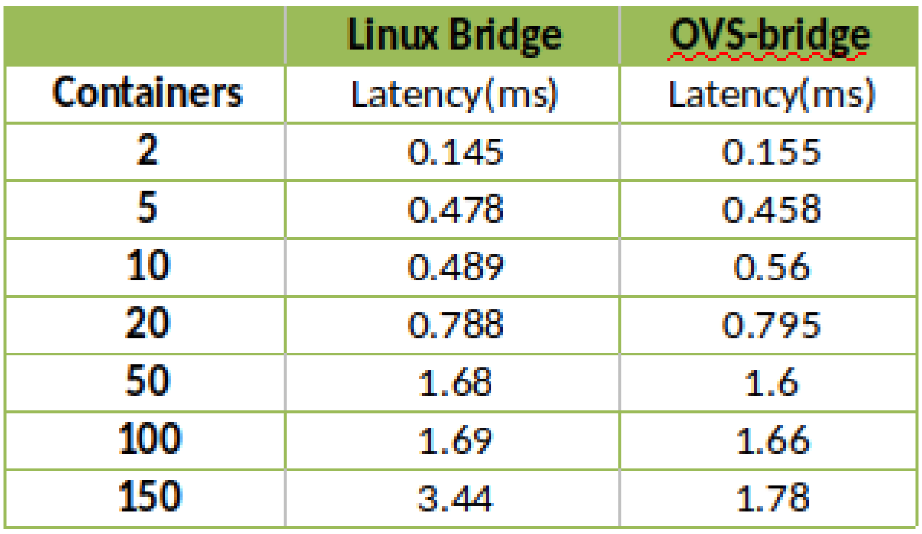
	Linux Bridge	OVS Bridge
Packets (kb)	Latency(ms)	Latency(ms)
10	0.155	0.183
20	0.164	0.308
30	0.085	0.137
40	0.113	0.205
50	0.123	0.21
60	0.43	0.217

TABLE A.2: Latency measure for different packet sizes



	Linux Bridge	OVS Bridge
Packet(Kb)	Throughput (Gb/s)	Throughput (Gb/s)
64	1.853	1.815
128	1.695	1.69
256	1.7925	1.775
512	1.76	1.76
1024	1.7575	1.615
1544	1.7375	1.725
2048	1.705	1.64

TABLE A.3: Throughput measure for different packet size

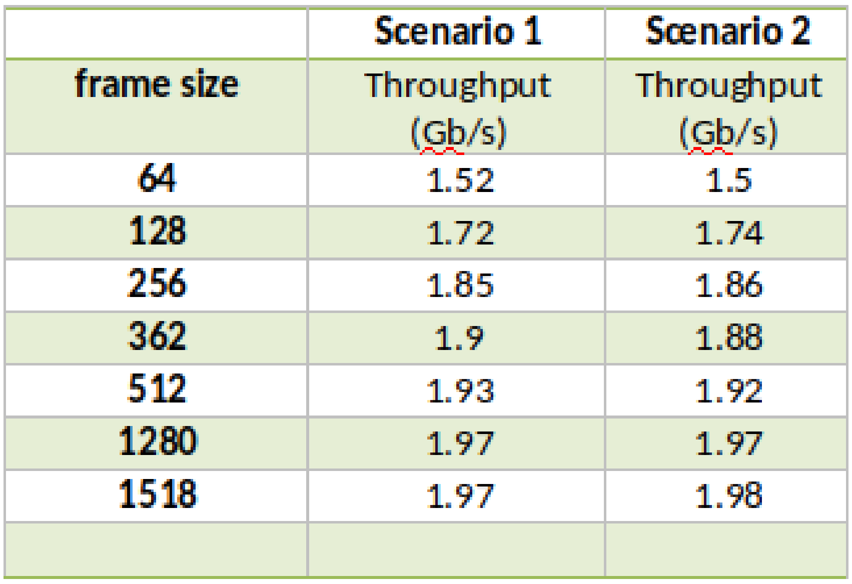


	Linux Bridge	OVS-bridge
Containers	Latency(ms)	Latency(ms)
2	0.145	0.155
5	0.478	0.458
10	0.489	0.56
20	0.788	0.795
50	1.68	1.6
100	1.69	1.66
150	3.44	1.78

TABLE A.4: Latency measure for number of containers

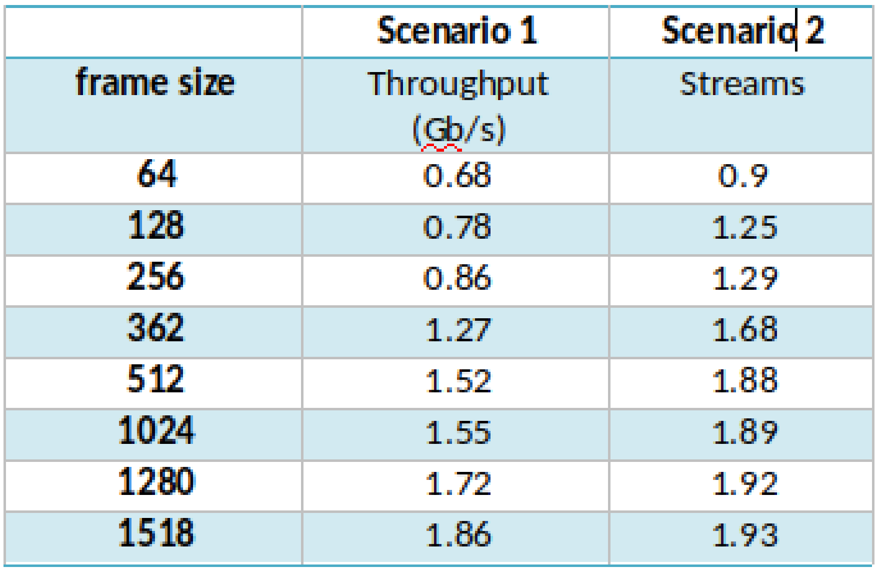
Appendix B

Experiment results: Architecture Scenarios



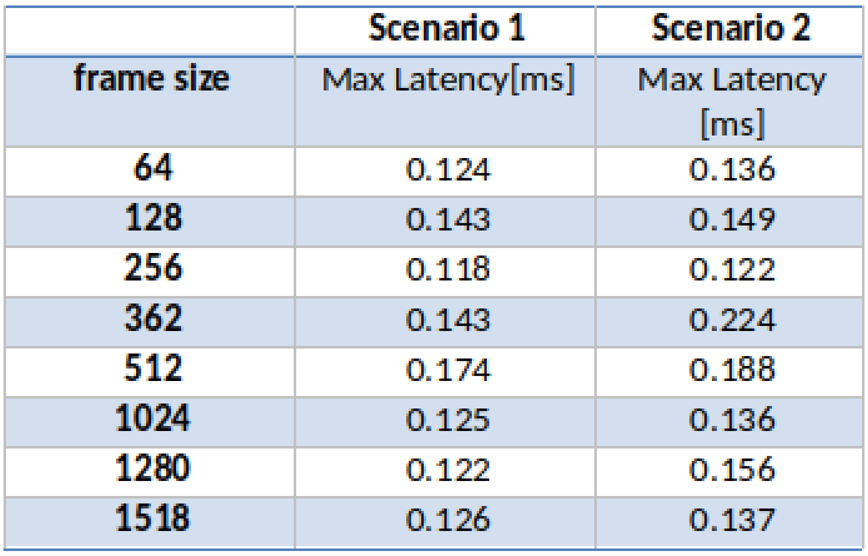
	Scenario 1	Scenario 2
frame size	Throughput (Gb/s)	Throughput (Gb/s)
64	1.52	1.5
128	1.72	1.74
256	1.85	1.86
362	1.9	1.88
512	1.93	1.92
1280	1.97	1.97
1518	1.97	1.98

TABLE B.1: Throughput measure for different frame size



	Scenario 1	Scenario 2
frame size	Throughput (Gb/s)	Streams
64	0.68	0.9
128	0.78	1.25
256	0.86	1.29
362	1.27	1.68
512	1.52	1.88
1024	1.55	1.89
1280	1.72	1.92
1518	1.86	1.93

TABLE B.2: Throughput measure for different frame sizes



	Scenario 1	Scenario 2
frame size	Max Latency[ms]	Max Latency [ms]
64	0.124	0.136
128	0.143	0.149
256	0.118	0.122
362	0.143	0.224
512	0.174	0.188
1024	0.125	0.136
1280	0.122	0.156
1518	0.126	0.137

TABLE B.3: Latency measure for different frame

Bibliography

Anderson, Jason et al. (2016). "Performance considerations of network functions virtualization using containers". In: *Computing, Networking and Communications (ICNC), 2016 International Conference on*. IEEE, pp. 1–7.

Bias, Randy (2016a). *Will Containers replace Hypervisors ?* http://cloudscaling.com/blog/cloud-computing/will-containers-replace-hypervisors-almost-certainly.

— (2016b). *Will Containers replace Hypervisors ?* http://cloudscaling.com/blog/cloud-computing/will-containers-replace-hypervisors-almost-certainly.

ClusterHQ and DevOps.com (2016). *Container Market Adoption Survey 2016*. https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf.

Cnet (2008). *AWS resiliency to EC 2*. https://www.cnet.com/news/amazon-web-services-adds-resiliency-to-ec2-compute-service.

Felter, Wes et al. (2015). "An updated performance comparison of virtual machines and linux containers". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE, pp. 171–172.

Foundation, Open Networking (2015). *Service Function Chaining Architecture*. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/L4-L7_Service_Function_Chaining_Solution_Architecture.pdf.

Gartner (2017). *Public Cloud Market*. https://www.gartner.com/newsroom/id/3616417.

Harvey, Cynthia (2017). *Cloud Computing*. https://www.datamation.com/cloud-computing/what-is-cloud-computing.html.

Intel (2017). *NFV Benefits 'I&' The Trends Driving Them*. https://www.networkcomputing.com/networking/5-nfv-benefits-trends-driving-them/1187036662.

isemag (2017). *the-continuing-evolution-of-the-vcpe*. http://www.isemag.com/2017/01/the-continuing-evolution-of-the-vcpe.

Jha, Kriti (2016). *Docker Network Function Chaining*. http://www.ijaert.org/wp-content/uploads/2017/06/131415.pdf.

k8 (2016). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. http://docker-k8s-lab.readthedocs.io/en/latest/docker/docker-ovs.html.

linuxcontainer.org (2017). *LXC container*. https://linuxcontainers.org/lxc/introduction.

Livi, Sergio et al. (2016). "Container-Based Service Chaining: A Performance Perspective". In: *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*. IEEE, pp. 176–181.

Merkel, Dirk (2014). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. https://nnc3.com/mags/LJ_1994-2014/LJ/239/11600.html.

Openstack (2017). *Advantage of Docker*. https://wiki.openstack.org/wiki/DockerWhat_unique_advantages_Docker_bring_over_other_containers_technologies.3F.

OS, Core (2017). *rkt container*. https://coreos.com/rkt/.

Rajendra et al. (2017). *The Journey to Network Functions Virtualization (NFV) Era*.

Ravidas, Sowmya et al. (2016). "Incorporating Trust in Network Function Virtualization". In.

Reilly, Dennis O (2017). *The Drawbacks of Running Containers on Bare Metal Servers*. https://www.morpheusdata.com/blog/2017-04-28-the-drawbacks-of-running-containers-on-bare-metal-servers.

Research, Doyle (2018). *Quantitative Research*. http://doyleresearch.com/.

Rubens, Paul (2017). *Why we need Containers*. https://www.cio.com/article/2924995/software/what-are-containers-and-why-do-you-need-them.html.

sdxcentral (2015). *whats is network-functions-virtualization-nfv*. https://www.sdxcentral.com/nfv/definitions/whats-network-functions-virtualization-nfv/.

Stacktutor (2015). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*. https://stacktutor.org/2015/04/13/docker-networking-weave.

stacktutor (2015). *Docker-networking weave*. https://stacktutor.org/2015/04/13/docker-networking-weave/.

Struye, Jakob and Spinnewyn (2017). *Assessing the Value of Containers for NFVs A Detailed Network Performance Study*. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8256024.

Varis, Nuutti (2012). *Docker Network Function Chaining*. https://wiki.aalto.fi/download/attachments/70789083/linux_bridging_final.pdf.