Master’s Programme in ICT Innovation

Orchestration Mechanism Impact on Virtual Network Function Throughput

Jiaqi Li

Aalto University

MASTER’S THESIS

Aalto University
MASTER’S THESIS 2019

Orchestration Mechanism Impact on Virtual Network Function Throughput

Jiaqi Li

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.
Otaniemi, 30 September 2019

Supervisor: Senior University Lecturer Vesa Hirvisalo
Advisor: M.Sc.(Tech.) Heikki Luukas

Aalto University
School of Science
Master’s Programme in ICT Innovation

Aalto University

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Abstract

Author
Jiaqi Li

Title
Orchestration Mechanism Impact on Virtual Network Function Throughput

School School of Science

Master's programme ICT Innovation

Major Embedded Systems **Code** SCI3024

Supervisor Senior University Lecturer Vesa Hirvisalo

Advisor M.Sc.(Tech.) Heikki Luukas

Level Master's thesis **Date** 30 September 2019 **Pages** 68 **Language** English

Abstract

Virtual Network Function (VNF) has gained importance in the IT industry, especially in the telecommunication industry, because a VNF runs network services in commodity hardware instead of dedicated hardware, thereby increasing the scalability and agility. The container technology is a useful tool for the VNF because it is lightweight, portable and scalable. The container technology shortens the product development cycle by easing the service deployment and maintenance. The telecommunication industry uses service uptime as an important gauge to evaluate if a service is of carrier grade, and keeping services up and running generates most of the maintenance costs. These costs can be reduced by container orchestration such as Kubernetes. Kubernetes handles the automation of deployment, scaling and management for applications with the help of orchestration mechanisms, such as the scheduler and load-balancers. As a result of those mechanisms, the VNFs running in a Kubernetes cluster can reach high availability and flexibility. However, the impact of the mechanisms on VNF throughput has not been studied in detail.

The objective of this thesis is to evaluate the influence of Kubernetes orchestration mechanisms on VNF throughput and Quality of Service (QoS). This objective is achieved by means of measurements run with a packet-forwarding service in a Kubernetes cluster.

Based on the evaluations, it is concluded that the VNF throughput is dependent on 6 parameters: CPU types, CPU isolation, number of Pods, location of Pods, location of load-balancer controllers, and load-balancing techniques.

Keywords virtual network function, container, container orchestration, benchmarking

Acknowledgments

Throughout the writing of this dissertation I have received a great deal of support and assistance. I wish to thank various people for their contribution to this project.

I would first like to thank my supervisor, Senior University Lecturer Vesa Hirvisalo, whose expertise was invaluable in the formulating of the research topic and methodology in particular.

I would like to acknowledge my colleagues from my internship at Oy L M Ericsson Ab for their wonderful collaboration. You supported me greatly and were always willing to help me. I would like to thank Kati Ilvonen for offering me the opportunity to work on my thesis at Ericsson. I would like thank Leena Salmela for arranging resources for my daily work. I would like to thank Rabbe Troberg for the guidance on directions for conducting experiments as well as the supports in writing thesis. I would like to thank Dietmar Fiedler for providing the thesis topic. I would like to thank Heikki Luukas for the advice on my decision making through the thesis. I would like to thank Juha Kamppuri for the daily supports on knowledge about the company, teams and products. I would like to thank Thomas Moll for the advice on experiments. I would like to thank Milán Gyõrki, Keijo Leppälä, Joakim Haldin, Vicent Ferrer Guasch, Hieu Nguyen and Sucheta Bali for the support to my future work.

Abbreviations

API - Application Programming Interface

AWS - Amazon Web Services

BGP - Border Gateway Protocol

CNI - Container Network Interface

COTS - Commercial Off-The-Shelf

DPDK - Data Plane Development Kit

ETSI - European Telecommunications Standards Institut

GB - Gigabyte

GCP - Google Cloud Platform

GGSN - Gateway GPRS Support Node

GPRS - General Packet Radio Service

GTP - GPRS Tunnelling Protocol

HPA - Horizontal Pod Autoscaler

HTTP - Hypertext Transfer Protocol

IP - Internet Protocol

IPVS - IP Virtual Server

IT - Information Technology

LCN - Load-balancer Controller Node

MAAS - Metal As A Service

NFV - Network Function Virtualization

NLCN - non-loadbalancer Controller Node

OS - Operating System

PDU - Protocal Data Unit

PN - N-PDU Number

PT - Protocol Type

QoS - Quality of Service

REST - Representational State Transfer

RTP - Real-time Transport Protocol

SGSN - Serving GPRS Support Node

TCP - Transmission Control Protocol

TEID - Tunnel Endpoint Identifier

TPR - Third Party Resources

TSDB - Time Series Database

UDP - User Datagram Protocol

VM - Virtual Machine

VNF - Virtual Network Function

WAN - Wide Area Network

dh - destination hashing

kpps - kilo packet per second

lc - least connection

nq - never queue

rr - round robin

sed - shortest expected delay

sh - source hashing

vCPU - Virtual Central Processing Unit

Contents

Abstract ii

Acknowledgments iii

Abbreviations iv

Contents vi

1. Introduction 1

1.1 Scope and Objective 1

1.2 Contributions 2

1.3 Thesis Structure 2

2. Background 3

2.1 Network Function Virtualization and Virtual Network Function 4

2.2 Containers 5

2.3 Container Orchestration 7

3. Experiment Tools 8

3.1 Kubernetes 8

3.1.1 Kubernetes Objects 9

3.1.2 Network and Load-Balancing 10

3.1.3 Kubernetes Scheduler 13

3.1.4 Kubernetes Horizontal Pod Autoscaler 15

3.2 Data Collection 16

3.2.1 Prometheus with Kubernetes 17

3.3 Network Protocols 19

3.3.1 GPRS Components 19

3.3.2 GTP Header 20

3.3.3 Path Protocol 21

4. Previous Work **23**

5. Experiment Design **26**

5.1 General Information 26
5.2 Experiment Design 27
5.3 Hardware Structure 28
5.4 Software Structures 29
5.4.1 MAAS 29
5.4.2 Kubernetes 30
5.4.3 Benchmarks 34

6. Results **38**

6.1 Impact of Number of Pods 38
6.2 Impact of CPU Isolation 40
6.3 Impact of CPU Limits 42
6.4 Impact of VNF Pod Location and Load-balancing Techniques . . 43
6.5 Impact of Controller Location 45
6.6 Impact of Local Calculation 47
6.7 Impact of Host CPU Type 48
6.8 Analysis 50

7. Conclusions **54**

A. Appendices **62**

A.1 Node Setup 62
A.2 VNF Deployment 63
A.3 Cluster Setup 64
A.4 Multus Setup 65
A.5 MetalLB Setup 66
A.6 Ingress Setup 66
A.7 Ingress for VNF 67
A.8 Ingress Service 67

1. Introduction

The telecommunication industry has been increasingly adopting the method of network service implementations from hardware-based inplementations to Virtual Network Functions (VNF) since the establishment of Network Function Virtualization (NFV) White Paper in 2012 [1]. The industry can benefit from the convenience offered by NFV, such as reducing R&D costs and development cycles. However, the transformation poses challenges related to crucial aspects of VNF implementation, such as portability, management and orchestration, automation, network stability and performance trade-offs of the VNFs. Many software applications have been built as a response to these challenges. For example, Container and Container Orchestration software are the two most commonly used ones that allow portability, easy deployment and maintenance of the VNFs and hence are widely adopted by companies in their production environments [2][3]. Although these tools are convenient, they introduce many mechanisms with overheads and configurable parameters. The industry has concerns on the impacts of those mechanisms on the performance of network services. Because little research has been conducted on the impact of those mechanisms on VNF throughput, a further study is required to gain a better understand of these tools, thus accelerating the process of NFV.

1.1 Scope and Objective

The thesis is a study for the impact of Container Orchestration mechanisms on VNF throughput and QoS. The main focus is to understand the mechanisms, such as the scheduling and load-balancing, of a Kubernetes cluster. In more detail, the objective of this study is to evaluate the impact of Kubernetes orchestration mechanism configurations, specifically the configurations of Scheduler and Network Load-balancer, on the throughput and QoS of a packet-forwarding service.

1.2 Contributions

The author of this thesis has implemented benchmarks that run on a Kubernetes cluster and a packet-forwarding service following the telecommunication system standard proposed by the European Telecommunications Standards Institute (ETSI). The author has conducted tests, collected data and analyzed the data of the packet forwarding service with several settings for Scheduler, Load-balancers, Pod and HPA.

This study provides setups for a Kubernetes cluster. The setup extends the setup given in Krishnakumar's Thesis [4]. The study provides the implementation of benchmarks for a packet forwarding service running on a Kubernetes Cluster. The benchmarks simulate a telecommunication service that runs on the cluster to obtain VNF throughput and QoS data. The benchmarks follow ETSI standards widely used in the telecommunication industry. The study provides data collected by the benchmarks. The study provides analysis for the data. The cluster is set up with open-sourced projects such as MetalLB, Kubernetes, Ingress and Prometheus. Data is collected using Prometheus and its related projects. Methodology for data collection adopts the same methodology proposed by other studies [5], [6], [7] and [8]. Conclusions for yielding a good VNF throughput is obtained from the analysis.

1.3 Thesis Structure

The thesis describes the study by introduction, background, experiment tools, previous work, experiment design, results and conclusions. Introduction in Chapter 1 states the objective and contribution of this study, as well as the structure of the thesis. Background in Chapter 2 explains concepts and backgrounds related to the study. Experiment tools in Chapter 3 describe the tools and reasoning for using these tools in this study. Previous Work in Chapter 4 prove the feasibility and value of the study together with methodologies. Experiment design in Chapter 5 lists procedures for conducting experiments and obtaining data. Chapter 5 also provides demonstrations for hardware and software interconnections and the procedures of setting up the experiment environment, providing the method for reproducing the study. Results in Chapter 6 provide data collected in the experiments and the analysis for the data. Conclusions in Chapter 7 summarizes results of experiments and provide the logic for designing a scheduler that yields the best VNF throughput. Chapter 7 also gives the problems and gaps found during the experiments and possible new studies in the future to solve the issues.

2. Background

Telecommunication industry used to rely on dedicated hardware-based appliances, such as Message Router, WAN Acceleration and Radio Access Network Nodes, to provide services and meet the high availability requirement of "five nines" [9]. Appliances were manually tuned and connected that they coped well with each other and could work reliably over a long time. However, such appliances are reaching their end-of-life as the Internet traffic grew rapidly on the last two decades, due to their long development cycles and difficulty in scaling [1]. Developing new hardware to handle the increasing traffic could be an option to scale up the service but it would become more difficult and less profitable as Moore's Law is reaching its limit [10]. A more practical solution could be to repeat the same setup multiple times, but this requires many manual operations, which may prolong the product cycle. Besides, such setup is inefficient in power consumption and space utilization. Therefore, the telecommunication industry urges for solutions to scale up and maintain services in a fast and low-cost manner.

Since 2012, when European Telecommunications Standards Institute (ETSI) proposed the White Paper on Network Function Virtualization, the telecommunication industry has been moving towards running their services and applications on Commercial Off-The-Shelf hardware instead of dedicated hardware [11]. Many models have been proposed to achieve high-availability of VNFs [12] with high performance [13]. Many tools have been invented to achieve the goals stated in the White Paper. Docker and Kubernetes are the most popular tools nowadays used in the telecommunication industry for the transformation. Many third-party projects are also published to even extend the usability of those tools.

The tools bring conveniences but they come with performance trade-offs [14][15][16]. In the telecommunication industry, in addition to time-to-market, performance is important because it relates directly to customer satisfaction, which directly affects the income of the industry and hence impacts the survival of those companies. Therefore, how to have a good performance while enjoying the conveniences

is on the research schedule of telecommunication companies.

The art of computer systems performance analysis: techniques for experimental design, measurement, simulation and modeling [17] gives steps for a performance evaluation study. The steps are listed in Table 2.1. This study follows the general guideline from the book. Most of the steps are illustrated in Chapter 5 except for the sixth step, because it affects the selection of tools. In this study we adopt measurement as the evaluation technique. Measurement has the advantage of high accuracy if parameters properly represent the variables found in real world, as well as high practical value. Hence, it requires proper selection for tools.

1. State the goals of the study and define the system boundaries.
2. List system services and possible outcomes.
3. Select performance metrics.
4. List systems and workload parameters.
5. Select factors and their values.
6. Select evaluation techniques.
7. Select the workload.
8. Design the experiments.
9. Analyze and interpret the data
10. Present the results.

Table 2.1. Steps for a Performance Evaluation Study [17]

Conducting experiments is a common approach to obtain empirical configurations that yield to a good performance for those tools [18]. Tools and methods that are used for the experiments should be similar to actual production environments to obtain trust-worthy conclusions. Hence, this study adopts industrial tools, standards and setups to conduct the experiments.

2.1 Network Function Virtualization and Virtual Network Function

In October 2012, the European Telecommunications Standards Institute (ETSI) published a white paper on Network Functions Virtualization (NFV), outlining its benefits, enablers and challenges [1]. The network functions in NFV are referred to as Virtual Network Functions (VNFs). Each VNF works similar to one of those dedicated appliances but it runs on Commercial-Off-The-Shelf (COTS) hardware. Multiple VNFs can be combined as a service, for instance, a Phone Call Service. The VNF usually runs on industry standard high volume servers, switches and volumes that locate in Datacenter and Network Nodes [1].

The NFV White Paper states that NFV could solve some problems of tradi-

tional network structures, for example, high R&D costs and long development cycles. NFV transforms telecommunication developments from hardware-based to software-based, thereby lowering R&D costs and time-to-market.

Hardware-based developments cost more both in time and capitals compared to software-based developments. The book *System Design for Telecommunication Gateways* [19] indicates that it costs from $100,000 to millions of dollars and takes a long time to produce an electronic product from concepts to a mass-manufactured product [19]. In contrast, software can be tested even before it is formally released. It usually updates every few weeks and does not require manufacturing.

Besides, hardware cycle is becoming shorter, which leads to less profitability in hardware-based developments. According to Cisco's recommendation [20], voice equipment needs to be replaced every five to seven years, when traditionally that equipment would have been used for 12 years or more. It means the same design can only be sold and create profit for a short time for the company before it is deprecated.

NFV could also solve the problem of scaling up and down services. Internet traffic has increased dramatically in the last two decades. According to Cisco Visual Networking Index [21], global internet traffic has grown from 100 GB per day in 1992 to 46,600 GB per second in 2017 and predicted to be 150,700 GB per second in 2022. Therefore, network services should be able to scale up to keep up with the traffic increments. Traditional network structures could be scaled up by adding multiple existing setups. However, this approach requires many manual operations and if the network becomes huge it would be too complex to maintain and troubleshoot [22]. VNF, on the other hand, runs on COTS hardware in VMs or Containers as software. Therefore, it can easily be scaled up by just duplicating the software. And it is easier to configure and debug compared to setting up hardware.

Although NFV can solve many problems of the traditional network structures, it brings great challenges. Challenges are portability, management and orchestration, automation, network stability and performance trade-offs. Since the proposal of the NFV White Paper, many software has been established to deal with those challenges. This study focuses on Container and Container Orchestration software.

2.2 Containers

A container is a tool that makes VNF portable and lightweight. VNF needs to be portable to set up easily on any hardware, which was first achieved by

running software on Virtual Machines (VMs). In this case, a VNF is stored as VM images, containing all dependencies for the VNF and is set up by instantiating those images. However, running VNF as VMs could be graceless because of overheads. Every VM is a complete system and it has its complete set of operation system, libraries and applications. Applications may not be the same for VM images, but Operating System (OS) kernel and libraries are mostly the same, which means many computer resources are wasted on those similar components. Containerization is a solution to reduce those overheads.

A container is a user space in which computer programs run directly on the host operating system's kernel but have access to a restricted subset of its resources [23]. As illustrated in figure 2.1, containers run on the same OS kernel and could share libraries, volume and network interfaces with other containers. Therefore, it is more agile and few resources would be wasted on running multiple OSs and system services.

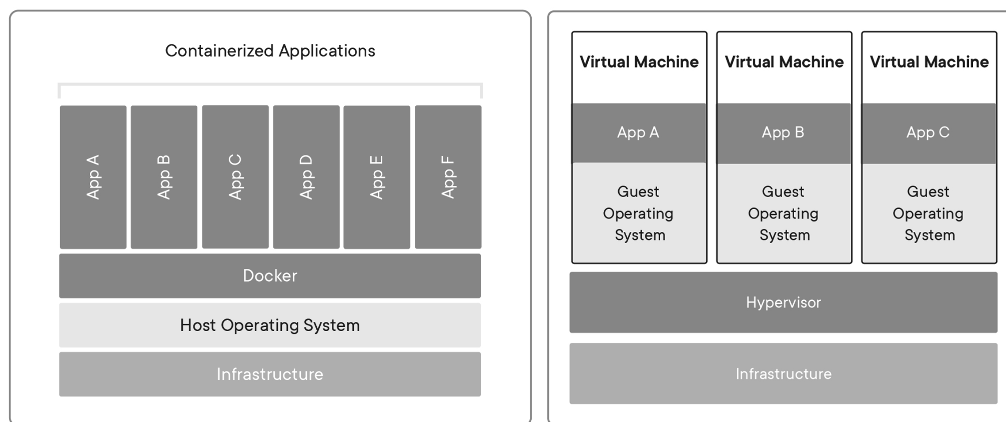

Figure 2.1. differences between container and virtual machine [23]

The container mechanism on Unix-based systems is implemented based on chroot and cgroups to allocate resources to containers and give isolation for containers, limiting the interference among containers. New namespaces will be created for new containers. Resources, such as CPU, network, memory and storage, will be assigned to the container namespaces to handle workloads.

Docker is the most popular container implementation in the last few years. It was released in March 2013, right after the VNF White Paper, and grew rapidly since then. Now Docker has 80B container downloads and more than 100k third-party projects [2]. Docker became popular because of its useful features and its good integration with other projects. Apart from basic container features, Docker ships with useful features such as scaling applications, creating load-balancer and container image management (Registries and Dockerfile). These features allow Docker to set up and scale services quickly in a light-weight manner. It is also integrated with infrastructure tools, such as Amazon Web Services, Google Cloud

Platform, Microsoft Azure and Kubernetes [24], making Docker more useful in production environments.

2.3 Container Orchestration

Although Docker improves the portability and usability of VNF, it operates only on a local machine, which limits the scalability. Besides, each container must be manipulated manually, making it difficult to maintain when the number of containers becomes large. These problems can be solved by an orchestration tool that manages containers on multiple machines. One of the most popular orchestration tools today is Kubernetes (K8s). Kubernetes is an open-source container orchestration system designed by Google and maintained by Cloud Native Computing Foundation [25]. It supports Docker as its most recommended Container Runtime and it is widely adopted by developers to improve the maintenance, management and automation of VNFs.

Container Orchestration brings convenience to the containerization of VNF. But as it is indicated in the White Paper, it must have performance trade-offs. How to minimize the impact of overheads and how to utilize the orchestration mechanisms to obtain best performance interests engineers in the telecommunication industry. Tools are needed to run tests and obtain data for the questions. The tools adopted by this study for experiments are Kubernetes, Prometheus, Network Protocols and Network Standards.

3. Experiment Tools

This chapter introduces the tools that are used in this study for benchmarking and obtaining data, together with the reasons for using the tools. Those tools are Kubernetes [25], Prometheus [26] and Network Standards [27].

3.1 Kubernetes

Kubernetes runs in a master-slave manner. A host is indicated as a Node in Kubernetes. Kubernetes Nodes has states, such as Ready, MemoryPressure, OutOfDisk, PIOPressure, DiskPressure, NetworkUnavailable and Unknown, indicating the availability of the Node. Master Node takes charge of serving API, controlling Objects and scheduling, while minion Nodes handle workloads, such as VNF workloads. Master Node takes commands from users and communicates with minion Nodes. Therefore, users do not interact with minion Nodes directly and therefore, operations are similar to local manipulations. This abstraction enables easy auto-deployment, scaling and management. Kubernetes structure is shown in Figure 3.1 [28]. This study uses Kubernetes v1.13.0.

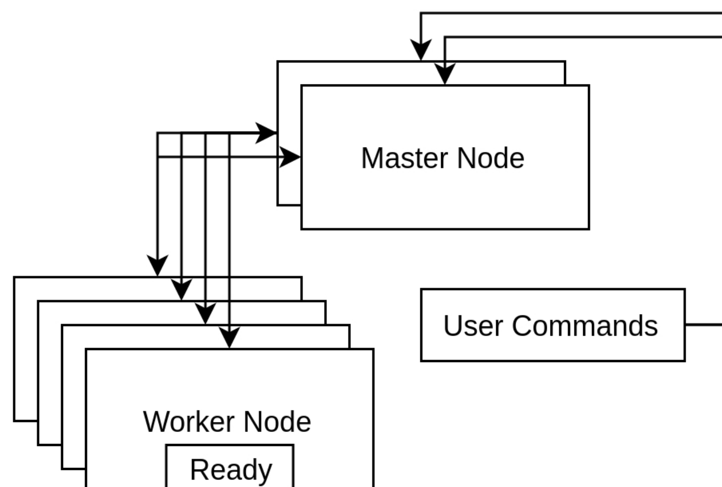

Figure 3.1. Kubernetes Structure

Kubernetes is composed of several components. The Master Node runs kube-apiserver, etcd, kube-scheduler, kube-controller-manager and cloud-controller-manager [29]. Kube-apiserver is the front-end for Kubernetes Control Plane. It exposes the Kubernetes API, allowing users to control Kubernetes by commands or RESTful calls. Etcd is a distributed reliable key-value store for distributed system and it is utilized in Kubernetes to store cluster data, such as Pod names, created date and other Object properties. Kube-scheduler is the default scheduler that keeps watching newly created Pods and assigns them to Nodes. Kube-controller-manager comprises Node Controller, Replication Controller, Endpoints Controller and Service Account & Token Controllers, administering Node states, number of Pods, populations of Endpoints and API access tokens for namespaces. Cloud-controller-manager provides interfaces to interact with cloud providers. But because this study works on bare-metals, the cloud-controller-manager is of no use.

Minion Nodes runs kubelet, kube-proxy and Container Runtime [28]. Kubelet communicates with Master Node and Container Runtime, making sure containers are running in a Pod. Kube-proxy connects Nodes together and abstracts services running in Nodes. Container Runtime is the software that runs containers and in most cases today the software is Docker.

3.1.1 Kubernetes Objects

VNF can be defined as Kubernetes Objects in YAML files. Kubernetes Objects describe the desired VNF states running on a cluster. These Kubernetes Objects include Pod, Service, Volume, Namespaces and Controllers [30].

A Pod is the basic unit of Kubernetes and it is the simplest Kubernetes Object that users can create or deploy [31]. One Pod can run one or multiple containers and attach resources such as disk volumes, memories and networks to it. Containers in a Pod share the same resources. A Pod could be scheduled to a minion Node once it is created. It keeps alive until the program running in it finishes or gets terminated.

A Pod will be deleted if Delete commands are given, the pod is evicted because of lack of resources or the Node it is scheduled on fails. Pods do not self-heal and if they fail, they can restart but they may not restart with the same state as before the failure. This may not be important if the Pod works as standalone but it becomes critical if other Pods rely on the failed Pod. A Pod will have its in-cluster IP address after it is created, which can be accessed by other Pods. However, once the Pod is deleted, the address may not be available, leading to failures of other Pods and thus failure of the VNF.

Controllers, including Deployment, ReplicaSet, StatefulSet and DaemonSet, keep a certain number of Pods alive, and mark Pods with names and labels by which they are accessible to other Pods [32]. This ensures that the VNF is up and accessible within the cluster. However, the Controllers do not meet the demand that the VNF has to be accessed outside the cluster and the requirement that traffic should be routed to Pods. This is solved by Object Service. Object Service can connect Pods to Endpoint Objects that are IP addresses and create rules for Kubernetes network traffic load-balancer to distribute traffic to Pods.

The networking functionality is not completely shipped with Kubernetes but come from many third-party projects.

3.1.2 Network and Load-Balancing

Networking is a major concern to VNF as the nature of the VNF is to deliver network contents between devices. Network load-balancer is hence an important component for the VNF because it distributes traffic among multiple devices in a system, allowing the VNF to have higher capacity and availability, and shorter response time [33]. In [34], Marttila T. introduces container networking and Load-balancing of Docker and briefly Kubernetes. Most of the container networking nowadays remains the same as the one in [34], except that now Kubernetes can use Macvlan with the help of Multus.

Kubernetes Networking

Kubernetes networking concerns four types of communications: container-to-container communication in a Pod, Pod-to-Pod communication in the cluster, Pod-to-Service communication and external-to-Service communication [35].

Container-to-container communication is solved by Pod implementation. Containers in a Pod share the same network namespaces, which means they can access each other by localhost. This also means that containers within a Pod should coordinate port usage to avoid resource conflict.

Pod-to-Pod communication is not shipped with Kubernetes. It is handled by several third-party Container Network Interface (CNI) projects. CNI concerns network connectivity of containers and manages network resources for containers. The projects used in this study are Multus CNI and Flannel.

Multus CNI [36] is a CNI plugin that creates subnets for Pods in Kubernetes Nodes and attaches multiple network interfaces to Pods before linking the network interfaces to subnets. Pods within a Node can communicate with each other over a subnet. But Pods on one node cannot communicate Pods on another Node only by using Multus. In addition, Flannel [37] is used for the communications between

Nodes. Flannel runs a binary agent called flanneld on each Node and provides IPv4 network between Nodes. Multus and Flannel work together, allowing Pods in Kubernetes Nodes to communicate with each other.

While container-to-container and Pod-to-Pod communications have been solved by the networking mechanisms stated above, Pod-to-Service and External-to-Service are solved by load-balancing mechanisms.

Kubernetes Load-Balancing

A VNF in Kubernetes can contain multiple Pods. VNF traffic could target directly to Pods but this is not realistic in practice because it is difficult and unsafe to maintain a list of the addresses of available Pods for every client. Instead, the client would send traffic to a single IP address, which is the IP address of a load-balancer. A load-balancer maintains a list of available VNF backends running on the cluster, accepts traffic outside the cluster and distributes traffic to those backends. According to *Load Balancing in the Cloud* [33], a load-balancer is compulsory for any network service to achieve high availability and flexibility.

High availability means that the network service would stay up and available for a long time. It is one of the most important properties for VNF as stated in Section 2.1. To achieve high-availability, a network service would run at least two backends and if one backend fails the other backends could still serve the traffic. A load-balancer can avoid offline backends and routes traffic only to those online backends. Therefore, the load-balancer ensures that the service will be available from the customers' point of view if at least one of the backends is online.

Flexibility means that a VNF could be scaled up, down, out or in easily. It is also one of the most important properties for a VNF. If the VNF receives more traffic than it can handle, a new host running the same backend can be instantiated and connected to the load-balancer at any time to handle the extra traffic. With the help of a load-balancer, the scaling behaviour does not require changes in the clients, which makes the VNF highly flexible.

Network traffic load-balancing is handled by multiple components in Kubernetes clusters [38]. These components are MetalLB, Kubernetes Service Object, Kube Proxy, IPTables/IP Virtual Server (IPVS) and Ingress.

Kubernetes Service Objects

A Kubernetes Service targets a set of Pods by their labels. Those Pods are the backends of the Service. When a Service is created, it gets an in-cluster virtual IP from Kube Proxy. The Service creates traffic directing rules for IPTables or IPVS from the virtual IP to IPs of Pods. As a result, traffic targeting the virtual IP of the Service will be routed to Pods that the Service selects.

Kube Proxy and IPVS

Kube Proxy watches Kubernetes API servers for the addition and removal of Service or Endpoints. It behaves differently according to the mode it is running on. It can run on three modes: userspace, IPTables and IPVS. This study only focuses on mode IPVS. In the IPVS mode, Kube Proxy calls Netlink interface to sync IPVS rules by a given period. Then the in-cluster traffic routing is handed over to IPVS. IPVS implements transport-layer load balancing inside the Linux kernel [39]. IPVS can capture both TCP and UDP traffic to the Service's in-cluster virtual IP address and Port and redirects the traffic to Service's backends. The selection of backend, referred to as the load-balancing algorithm, is based on the IPVS scheduler setting. Available settings are round-robin(rr), least connection(lc), destination hashing(dh), source hashing(sh), shortest expected delay(sed) and never queue(nq). IPTables is the only approach that Pods can respond to requests.

MetalLB

Service can be of type "LoadBalancer", which allows the Service to obtain an external IP address that is accessible outside the Kubernetes cluster. That external IP is given by an external load balancer. But this load-balancer is not shipped with Kubernetes. In cloud platforms such as Google Cloud Platform and Amazon Web Services, users can request a load-balancer for Kubernetes cluster from the cloud providers to receive external traffic. In a bare-metal cluster, this external load-balancing can be handled by third-party projects such as MetalLB. MetalLB runs one controller in the cluster and one speaker for every Node, which can assign an external IP to a Service. MetalLB can work on two modes: Layer 2 mode and Border Gateway Protocol (BGP) mode. Layer 2 mode is universal. MetalLB in this mode can run on any Ethernet network with no special hardware while in BGP mode an extra router is required. This study runs MetalLB in Layer 2 mode. In this mode, traffic will first go to the Node that runs MetalLB controller and then Kube Proxy will spread the traffic to Service backends.

Ingress

While Service is the load-balancer for Pods, Ingress is a load-balancer for Services in Kubernetes. The Ingress is introduced in Kubernetes v1.2.0 to deal with the limitation in Service that the Service cannot do path-based routing. This limitation means that the user cannot use a single IP address for multiple Services. A company could run multiple Services in the Kubernetes cluster and it is not always possible for each of those Services to have a public IP address. The number of public IPv4 addresses is limited and even business giants in the IT industry such as Google and Amazon could have only a few IP addresses. A practical solution is to use Ingress to connect multiple Services to a single IP address and use different ports or URL sub-links to access those Services.

3.1.3 Kubernetes Scheduler

After a Pod is created, it will be placed in a queue in Pending state and is waiting to be assigned to a Node, before the state turns into Running. The Scheduler is the program that watches the queue and assigns Pods to Nodes. The procedure that assigns a Pod into a minion Node is called scheduling. Kubernetes has a default scheduler that allows users to run Pods with little effort.

The behaviour of the default scheduler is illustrated as Graph 3.2. It first takes a Pod from the pending queue. Then, it filters Nodes that fulfils the specifications of the Pod and scores those Nodes. Then the Pod will be bound to the Node that has the highest score. If no available Node is found and Preemption is enabled for the Node, the scheduler will evict Pods one by one from the Node with the highest score until enough resources is obtained for the new Pod. The eviction is based on the Pod Priority that is set in the Pod specifications. Pods with lower priorities will be evicted to release resources for Pods with higher priorities. After the Pod is assigned to a Node, the scheduler binds plug-ins and other components such as volumes and networks. All the scheduling is then finished.

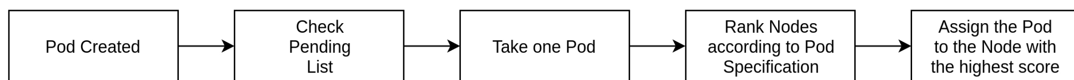

Figure 3.2. Scheduler Behaviour [40]

Therefore, the behavior of the default scheduler is determined by parameters for Pods, Nodes and Scheduler. All available parameters are listed in Table 3.1. This thesis focuses on the parameters that affect VNF throughput and those parameters are marked with * in the table.

Pod	Node	Scheduler
Resource Request and Limit*	Resources Usages*	Scheduler Name
Affinity*	Image Locality	Hard Pod Affinity Symmetric Weight*
Taint Toleration*	Node Label*	Preemption
CPU Pinning and Isolation*	CPU Pinning and Isolation*	Percentage of Nodes to Score
		Bind Timeout Seconds

Table 3.1. Parameters that affect Scheduler Behaviour

The Kubernetes Scheduler places the Pods evenly, in a partial and non-injective approach, on any Node as long as the Node posts Ready state [41]. Instead of using default settings, Pods can be scheduled according to its specification. Specifications could contain Node names, Affinity, Anti-affinity, resource requests and limits. Pods can be scheduled only when all specifications present in the Pod definition are met.

Users can allocate a Pod to a specific Node by putting a Node name in spec.nodeName

[42]. Then the Pod will only run on that Node. Pod specifications regarding resources requests and limits are available on *Managing Compute Resources for Container* [43]. Users can also specify the minimum amount of resources allocated to a Pod by using the spec.containers[].resources.request keyword. Pods will stay in Pending queue until the Node has enough resources. A similar keyword spec.containers[].resources.limits indicates the maximum amount of resources available to a Pod. These resource keywords constrain CPU and memory usages. In Kubernetes, CPU resources are measured in CPU units. One CPU is 1 vCPU which is a standardized unit equivalent to 1 Amazon Web Services (AWS) vCPU, 1 Google Cloud Platform (GCP) Core, 1 Azure vCore, 1 IBM vCPU and 1 Hyperthread on a bare-metal Intel processor with Hyperthreading. vCPU represents a portion of a physical CPU and is usually measured in CPU time. 500m CPU, also read as five hundred millicpu, guarantees a CPU time that is half of the CPU time given by 1 CPU. Memory resources are measured in bytes. For example, 123Mi indicates 123 Mebibytes. Pods that reach the memory limit may be terminated in the runtime.

Affinity and Anti-affinity constrain Pods with labels from Nodes or other Pods [42]. Affinity aims to schedule Pods in a way that satisfies specified expressions while anti-affinity tries to avoid scheduling Pods according to labels. Affinity and Anti-affinity both have Pod and Node constraints. For example, podAffinity tries to schedule the new Pod along with the Pods that have particular labels while podAntiAffinity tries to avoid those Pods working together. Similarly, nodeAffinity would constrain Pods to Nodes with those labels while nodeAntiAffinity tries to avoid allocating Pods to those Nodes.

CPU pinning and Isolation allows exclusive CPUs to be granted to Pods [44]. The CPU control is not enabled by default, which means Pods are sharing CPU with each other and the workload can be migrated between CPUs in runtime. This could worsen the performance because it introduces extra context switching time. Users can enable this feature by setting extra parameters to the kubelet running on Nodes. Those parameters are –cpu-manager-policy, –kube-reserved and –system-reserved as stated in Kubernetes Docs [44]. Pods will have a qosClass indicating if the Pods are running with CPU isolation enabled. Burstable and BestEffort for qosClass indicate that CPU isolation is not enabled for the Pod while Guaranteed indicates that the feature is enabled. To enable the feature, the Pod should be specified with CPU limits and requests set to an integer, together with memory limit and requests.

The Nodes could be VMs or physical servers and therefore have resources limits. The number of available resources of a Node can be checked from the Kubernetes

Metrics API. Pods will not be scheduled if the Nodes do not hold enough usable resources. Then, the Pod Priority and Preemption mechanism from the Kubernetes Scheduler can handle the situation well. Pods could be specified with different priorities beforehand. In the provided situation, Pods with lower priority will get evicted from its current Node to make room for the new Pod.

3.1.4 Kubernetes Horizontal Pod Autoscaler

While Kubernetes Objects enable easy deployment, scaling and maintenance, the network services still require manual scaling. Being able to auto-scale is important for companies to reduce the billing from Cloud Providers or reduce the power consumption of their infrastructures. The workload of the services differs from time to time. Over-reserving resources would waste energy or cause extra billing when the services have few requests. Under-reserving resources reduces costs but it would lead to bad Quality of Service (QoS) when the services encounter more requests than the resources can handle. Manual scaling is not a good option because it requires human interaction, which is slow and error-prone. Kubernetes has a Horizontal Pod Autoscaler (HPA) that addresses the problem [45]. HPA works on Controllers, such as Deployment and ReplicaSet, and changes the number of Pods they control.

HPA can scale up and down Pods according to the observation on cluster metrics. Kubernetes has metrics for Pods and Nodes on an API server. The default metrics include CPU, memory and network usages. HPA takes desired values for metrics and try to match the observed metrics with desired values by scaling up and down Pods. For example, if CPU usage is taken as a metric and 100m is set as the desired value in Pods while 200m is reported by metrics API server, 200m/100m = 2 would be the scaling ratio and therefore, the number of Pods will be doubled. In another case, if the desired value is 100m but 50m is reported, 50m/100m = 0.5 would be the ratio and therefore, half of the Pods will be deleted. However, it takes time for Kubernetes to create Pods and re-distribute workloads. To prevent overshoots in scaling, Kubernetes have flags to control the interval between scalings. Those flags are –horizontal-pod-autoscaler-downscale-stabilization-window, –horizontal-pod-autoscaler-initial-readiness-delay and –horizontal-pod-autoscaler-cpu-initialization-period.

HPA can also use custom metrics. Users can specify their custom metrics such as latency of the services and the number of HTTP requests for scaling by using the Kubernetes Custom Metrics Object. However, both default metrics and custom metrics are not stored anywhere. Besides, defining Custom Metrics Object could be complicated. Tools are needed to manage those metrics and Prometheus is one

of the good options.

3.2 Data Collection

The study uses Prometheus for data collection. Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud. [26] Prometheus joined a hosted project in Cloud Native Computing Foundation in 2016, together with Kubernetes. Since then many projects, including Prometheus-operator, Kube State Metrics, Node Exporter and Prometheus Adapter, have started to integrate Prometheus into Kubernetes to monitor metrics and collect data from Kubernetes Objects.

Prometheus architecture is shown in graph 3.3. Systems can run Prometheus Client to export purely numeric time series by RESTful API at /metrics paths on the Client address. Prometheus Server scrapes those time series from systems by a given interval via the API and stores the data in its Time Series Database (TSDB). The Server also has RESTful API that allows Alertmanager, Data Visualisation Tools or other API clients to read data of those monitored systems. Time series

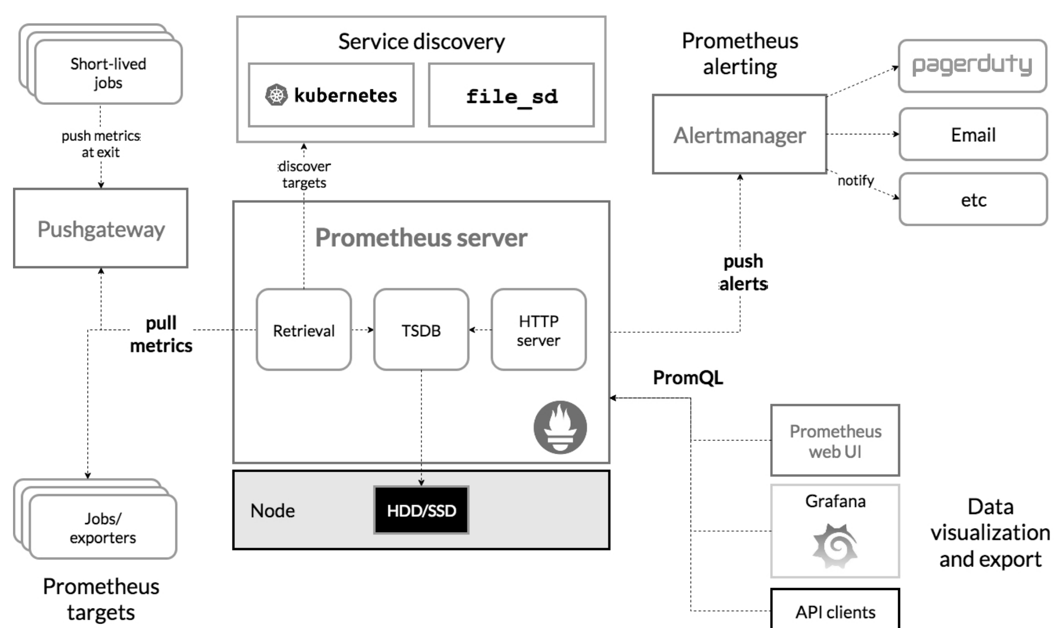

Figure 3.3. Prometheus Architecture [26]

are time-stamped data. They could be server metrics, application performance monitoring, network, sensor and other analytics data. TSDB is the database optimized for handling time series data. [46] It provides fast writing and reading for time series data, and at the same time supports specialized features such as precision reduction, interpolation and multimetric aggregate computing.

TSDB has gained importance over the last decade because of the IT industry

evolution from monolithic applications to serverless applications. Monolithic applications combine all user interface and data access code into a single program running in a platform. Serverless applications, known as microservices, separate the program into components. The modularization enables easier development, testing and more importantly, resistance to architecture erosion. However, this transition brings challenges for monitoring systems. As the program becomes compartmentalized, there will be more data sources and more data points. Every data source may have a few metrics collected every few seconds or even nanoseconds for months, leading to a large amount of data. Those time series may be queried with different intervals or precisions, and sometimes queried for the summaries over a long time span. Traditional databases perform poorly on those queries according to the research done by Influxdata [47]. Hence, the new database TSDB is built to meet the demands. TSDB could outperform traditional databases 14 times in throughput [47] and therefore, it is becoming popular nowadays.

3.2.1 Prometheus with Kubernetes

Prometheus collects and stores data for Kubernetes. When working with Kubernetes, instead of simply scraping data directly from Prometheus Clients, Prometheus runs multiple plugins to scrape data from Pods and Kubernetes Metrics API Server and exposes metrics as Custom Metrics Objects at Kubernetes Metrics API Server in order to use the metrics at HPA. Those plugins are Prometheus Operator, Prometheus Node Exporter, Prometheus Adapter and Kube State Metrics. The interconnection of the plugins is illustrated in graph 3.4

Prometheus needs target IP addresses to scrape data. Users can check the IP addresses of Pods and manually put them in the monitoring list. This is time-consuming and also problematic because Pods can fail and IP addresses will be lost. Prometheus Operator manages Prometheus Servers and their configurations. It allows Prometheus to discover Pods by their labels by defining two Third Party Resources (TPR): ServiceMonitor and Prometheus. Provided that Pods are running Prometheus Clients and a Service has been created to connect those Pods to an Endpoint, a ServiceMonitor defines the Service that Prometheus Server should monitor and scrape data from and a Prometheus Object ensures that the Prometheus Server can run with desired settings to monitor the Service described in the ServiceMonitor.

Prometheus Node Exporter exports Node level metrics to Prometheus Server, including CPU, memory, disk and network utilization. The Exporter mounts cluster metric files in /proc before exposing the data in those files by RESTful API. Not all Node level metrics can be collected by the Node Exporter. For example,

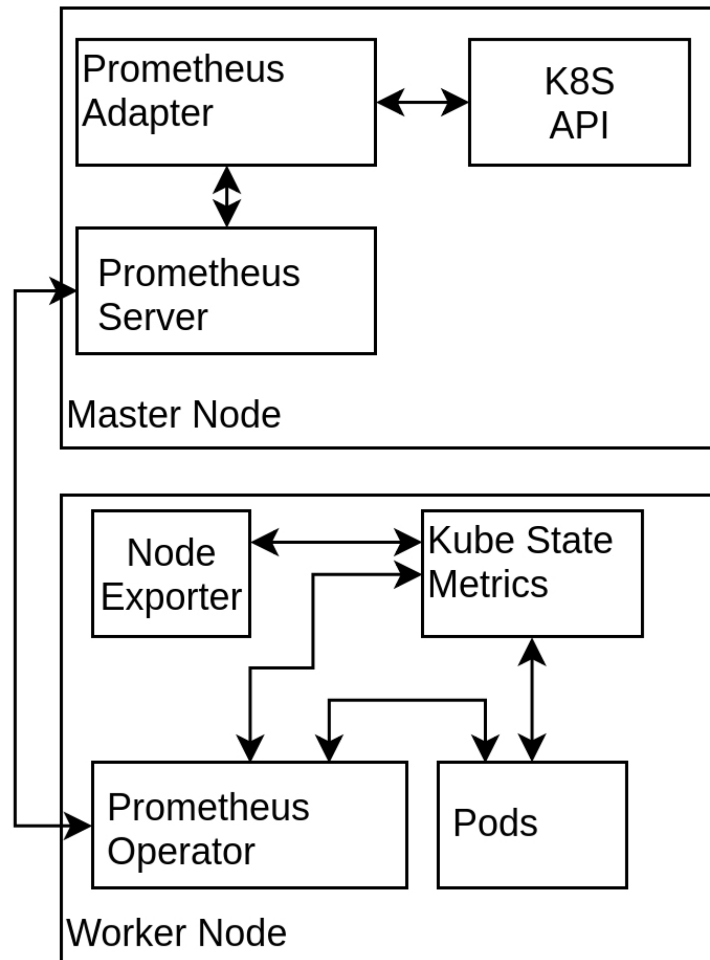

Figure 3.4. Interconnection of Prometheus Plugins

the metrics from Kubernetes components such as kubelet and kube-scheduler is not stored in cluster and therefore the Exporter cannot collect them. Prometheus Operator provides options for running those components in Pods in order to obtain monitoring data from them.

Prometheus Adapter runs a Metrics API Server to replace the original Kubernetes Metrics Server and exports collected metrics to the Metrics API Server. It allows HPA to use all the metrics in Prometheus for auto-scaling while reducing the queries on the Kubernetes API server.

Kube State Metrics listens to the original Kubernetes API Server and generates metrics for various objects running on the cluster before exposing it on HTTP endpoint /metrics. The metrics will then be scraped by Prometheus Operator.

Data collected by Prometheus can be queried by PromQL via HTTP API or via Prometheus's expression browser. PromQL is a functional query language that enables users select and aggregate time series in real time [48]. Metrics can be queried by given names and labels together with a time span. The PromQL has four data types for evaluating expressions: Instant Vector, Range Vector, Scalar and String. PromQL has some built-in functions, such as sum() and rate(). These

functions can only accept specific types of data. For example, rate() accepts only Range Vector data.

PromQL queries metrics by names and filters data with labels matching specific regular expressions. Generally, PromQL syntax for querying data is <instant_query> '[' <range> ':' [<resolution>] ']' [offset <duration>] [48]. For example, when querying for container_cpu_usage_seconds_total, Prometheus will return all the time series with the name. If the CPU usages of the Pods in the default namespace is wanted, filtering can be done by adding label names and conditions in curly braces behind the metric name, which would be container_cpu_usage_seconds_totalnamespaces="default" in this case. Metrics can be filtered by multiple labels. Conditions can be regular expressions in RE2 syntax [49]. PromQL allows four types of operators to match label names and conditions. Those operators are "==", "!=", "= " and "! ", meaning exactly equal, exactly not equal, matching regular expressions and not matching expressions. Metrics can be queried with a specific range. If the metric from the last 5 minutes is wanted, [5m] can be placed after the curly braces. Time duration is specified as a number and a unit. Units could be s for second, m for minute, h for hour, d for day, w for week and y for year. Metrics can be queried with a given offset following the range, indicating the earliest data from now. If the data from 2 days ago till now is wanted in the example above, the complete query would be container_cpu_usage_seconds_totalnamespaces="default"[5m] offset 2d.

PromQL has functions for easy concatenations of the metrics. Two of those functions are concerned in this study. Function sum() sums up all the queried metrics and function rate() returns the changing rate of the queried metrics.

3.3 Network Protocols

This study follows ETSI TS 129 060 V12.6.0 [50], the most popular standard adopted by Digital Cellular Telecommunications System, to implement the VNF testing benchmark. The ETSI TS 129 060 V12.6.0 is a standard for GPRS Tunnelling Protocol (GTP) proposed in October, 2014. The protocol is a group of IP-based communication protocols used within General Packet Radio Service (GPRS) Core Network [51].

3.3.1 GPRS Components

GPRS core network is the central part of GPRS and connects 2G, 3G or other mobile networks with external networks such as the Internet [50]. It provides

functions for mobility management, session management, transport for Internet Protocol packet services, billing and lawful interception. The core network is composed of Gateway GPRS Support Node (GGSN) and Serving GPRS Support Node (SGSN). GGSN acts as a router that interconnects SGSNs or redirects mobile traffic to external networks. SGSN delivers data packets among mobile devices and mobile stations within its geographical service area. The study focuses on the situation where mobile devices are communicating with each other through a GGSN among multiple SGSNs.

GTP allows users to move from place to place without losing network connections by combining user session information and media data. GTP contains three protocols: GTP-U, GTP-C and GTP' (GTP prime). GTP-C is used for signalling between GGSN and SGSN to activate users' sessions, deactivate the sessions and adjust QoS parameters. GTP-U carries user data within the GPRS core network. GTP' is responsible for billing functionalities in the GPRS core network. The study uses GTP-U for data transfer and throughput measurements. GTP-C is used for signaling between the traffic generator and the receiver.

3.3.2 GTP Header

A GTP header has a minimum length of 8 octets and contains information about the user session, for instance, information about the GTP protocol version, protocol type, extension header, message type, length of the payload and tunnel endpoint identifier. All GTP protocols share the same Header. Throughput would be greatly affected by the Header structure because that information determines the interpretations of packets and different interpretations require a different amount of calculations. Hence, it's necessary to follow the GTP Header standard to obtain realistic throughput data. GTP Header structure is shown in graph 3.2.

The Version field is a 3-bit field indicating the GTP protocol version. GTP-U only have one version which is GTPv1-U and therefore, the Version would be set to 1.

Protocol Type (PT) field is a one-bit field following Version. It distinguishes GTP' from other protocols. GTP' has PT value 0 while the other GTP protocols have PT value 1. The header would be interpreted differently according to PT. As the study uses GTP-U, PT would be set to 1.

Extension Header flag (E) defines the presence of a meaningful Next Extension Header field. In this study, communications happen within the GPRS core network and thus no extra header is concerned. This field will be set to 0.

Sequence Numbers flag indicates if the sequence numbers in the 9th and 10th octets are available. This study does not send sequential packets and thus the flag would be set to 0.

Octets	Bits 8	7	6	5	4	3	2	1
1	Version			PT	(*)	E	S	PN
2	Message Type							
3	Length (1st Octet)							
4	Length (2nd Octet)							
5	Tunnel Endpoint Identifier (1st Octet)							
6	Tunnel Endpoint Identifier (2nd Octet)							
7	Tunnel Endpoint Identifier (3rd Octet)							
8	Tunnel Endpoint Identifier (4th Octet)							
9	Sequence Number (1st Octet)							
10	Sequence Number (2nd Octet)							
11	N-PDU Number							
12	Next Extension Header Type							

Table 3.2. GTP Header Structure [50]

N-PDU Number flag (PN) indicates if the N-PDU number is present or not. N-PDU stands for Network Protocol Data Unit. The N-PDU number represents the sequence number for N-PDU. This would be needed if the mobile devices need to communicate with an external network. In this study, no external network is concerned and this flag would be set to 0.

Message Type indicates the type of the GTP message with a single octet. The full list of message types is available in the standard [50]. Redirection Response and Redirection Request are the message types in the study and therefore, 6 and 7 will be the value in this field for those messages separately.

The Length field is split into two octets and indicates the payload length. The maximum payload length is 65535 octets.

Tunnel Endpoint Identifier (TEID) is a number of 32 bits identifying the target tunnel endpoint for the message.

The length of the header depends on the values of those fields. With any one of the E, S and PN set to one, the length would be 12 octets. In the study, all of the three flags are 0. Thus, the header length is 8 octets.

3.3.3 Path Protocol

UDP/IP is the only path protocol defined to transfer GTP messages in GTP version 1, according to the standard [50]. UDP stands for User Datagram Protocol. It is one of the core members of the Internet Protocol Suite. UDP/IP means to send messages, referred to as datagrams, on an Internet Protocol (IP) network. A

datagram contains headers and payloads.

Communications using UDP do not require handshakes to set up communication channels. UDP itself has no error checking and correction. Therefore, communications are not ordered and not guaranteed for delivery. The unreliability of UDP reduces overheads in Protocol Stack, which makes processes fast and hence makes it suitable for time-sensitive applications, such as Voice over IP and online games.

UDP provides verification in its header. UDP Header is shown in graph 3.3.

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source Port																Destination Port															
4	32	Length																Checksum															

Table 3.3. UDP Header Structure [50]

All four fields in the header are of 2-octet in length. Source port indicates the port of the sender. Destination port serves a similar functionality but it indicates the port of the receiver. Length field implies the message length. The length includes lengths for header and the payload. Checksum can be used for error-checking.

In the standard, UDP Destination Port for GTP-U request messages is 2152. The port for GTP-C is 2123.

4. Previous Work

Since the establishment of Container and Kubernetes, some researches have been conducted to test the performance of those tools. This study learns from previous researches and fills certain gaps in those researches. Therefore, this study is built on those studies and extends to solve some unknowns.

In *Operating Systems: Internals and Design Principles* [5], Stallings W. classifies multiprocessor systems as three types. Cluster, also referred to as loosely coupled or distributed multiprocessor, is one of the multiprocessor systems. For such systems, each processor has its own main memory and I/O channels. This structure has benefits such as absolute scalability, incremental scalability, high availability and superior price per performance. Stallings lists several clustering methods. One of the methods is Separate Servers, which is the clustering method of Kubernetes. This method has high availability with the drawback of high network and server overhead. The benefit of high-availability fits the requirement of carrier grade in the telecommunication industry. But the impact of the overheads on telecommunication VNF throughput is still unknown in quantity.

Load Balancing in LTE Core Network with OpenStack Clouds [52] evaluates network performance of a load balancing system in an OpenStack Cloud Cluster. This study demonstrates that network performance would be affected by virtual machine locations and the number of VNF backends. The study draws a conclusion that the VNF would have the highest throughput if the VNF runs in one single OpenStack instance on the same physical virtualization host. However, the study runs an application that handles HTTP RESTful API in TCP instead of UDP. Therefore, it's unclear that if the same conclusion can apply to a telecommunication VNF that runs in containers in UDP. The study runs VMs each with 2 vCPUs. The figures for a bigger number of vCPUs are still to be measured. It would also be interesting to see how the number of VNF backends influences VNF throughput in the container case.

The study *Characterising Resource Management Performance in Kubernetes*

[6] conducted experiments to estimate starting time, termination time and the execution time for different applications. For benchmarking starting time, the authors run test cases on three variables and measured total deployment times. The starting time is affected by the number of machines, the number of containers in a Pod and the total number of containers. The termination time is also relevant to the same variables mentioned above. The results for the execution time of CPU intensive applications were also measured and those results are relevant to the same parameters. The study includes an estimation of the throughput for network-intensive applications as well. The study shows that the performance of both CPU and network intensive applications in Kubernetes is affected by the number of Pods, the number of Containers in a Pod and the number of Nodes. The study draws the conclusion that network-intensive applications would reach higher throughput if a Pod includes only one container. However, the hypothesis tests for the network is not throughout enough. The network tests ran on TCP protocol and the tests were either run on the same host or on the same Kubernetes cluster. It means the performance for traffic in UDP protocols and the performance for dealing with traffic from outside the Kubernetes cluster is missing.

NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions [7] proposes a framework for characterizing the performance of a VNF running on OpenStack VMs. In this framework, the VITAL acts as an orchestrator that can configure VNF and workload generation, and monitor metrics of the VNF. The configuration for VNF could be the number of CPU cores, memory and network interface limit. The configuration for workload generation could be call setup rate and the traffic intensity. Then metrics of the VNF during the test will be collected by VITAL. Therefore, users can use this framework to estimate VNF capacity for a given resource configuration, compute virtualization overheads, determine optimal resource configuration for a given workload, evaluate different virtualization and hardware options and fine-tune VNF implementation and performance [7]. The methodologies and benchmarking models proposed in the study are reasonable and could fit well in benchmarking telecommunication VNFs. However, the demonstrations in the study could be improved and extended. The study measures VNF capacity by the maximum number of input calls and successful calls for a given set of CPU settings without indicating the packet rate of each call. Therefore, the relationship between CPU number and maximum packet processing rate remains to be filled. Besides, the study runs VNF on VMs and adopts Clearwater architecture, which may have a different result than a VNF that runs on containers and other microservice architectures. Therefore, new tests that adopt the same methodologies can be conducted in a Kubernetes cluster

with other VNF architectures to improve the study. The tests run only one type of VNF for benchmarking, which means it remains unknown how the number of VNFs affects throughput and how different VNF types influence each other.

Multi-VNF Performance Characterization for Virtualized Network Functions [8] conducts experiments to test the influences on VNF throughput with regards to the number of VNFs running on the same host and with regards to network traffic routing software. The study connects an IXIA traffic generator to a host. The traffic generator sends traffic to the host and the traffic will then be directed to VNFs running on the host as VMs, using I/O technologies. The throughput is measured with different numbers of VNFs running on the host ranging from 1 to 10. The throughput is also measured with different I/O technologies such as OVS, SR-IOV and FD.io VPP. This study shows that the VNF throughput would be affected by the number of VNFs sharing the same resources and affected by traffic routing technologies. However, the study focuses on vertical scaling but does not concern horizontal scaling. Therefore, it remains unknown how horizontal scaling affects the throughput in relation to the number of VNFs. Also, it remains unknown for the impact of multi-host load-balancing mechanisms on VNF throughput.

Accelerated DPDK in Containers for Networking Nodes [4] measures network latencies with DPDK, SR-IOV, Open vSwitch and Open Virtual Network in a Kubernetes cluster. The study sets up a Kubernetes cluster with MAAS and kubeadm. VNF runs in a Pod in Kubernetes. In the experiments of the study, traffic is routed to the Pod and is handled with different techniques, such as OVS, DPDK, SR-IOV, Multus and OVN, and the corresponding latencies are measured. The study provides a practical hardware setup for benchmarking a Kubernetes cluster and potential tools used in Kubernetes networking. However, the study does not measure the latencies for different number of Pods and different locations for the Pods. Therefore, the impact of the number and location of Pods on latency could be measured to extend the research.

To summarize, the previous researches provide practical directions and setups for benchmarking VNFs, meanwhile providing reasoning to justify the methodologies adopted in this study. This study is developed from those studies and fills the gap in the domain of NFV.

5. Experiment Design

This chapter introduces the experiment procedures for benchmarking and obtaining data. The studies mentioned in Chapter 4 provide guidelines for benchmarking VNF throughput. Basic structures of those tests are similar. This study follows the setup structures and benchmarking methodologies of the previous studies while extends to solve the gaps in the research area of benchmarking NFV.

5.1 General Information

In the experiments, a traffic generator sends traffic to the VNF running on physical hosts. A metrics collector is constantly running during tests and collects data from those hosts. The metrics are CPU usage, memory usage and network usage. Network usage is highly relevant to the VNF throughput and hence is taken as the gauge of VNF throughput. Test cases differ from each other by parameters of the VNF. VNF throughput could be affected by several parameters that could be the number of VNFs sharing the same resources and traffic routing technologies. Tests are conducted by changing those parameters.

The workload is a packet forwarding benchmark running in Pods. The benchmark receives packets from a sender and forwards packets to a receiver. The size of the packets is 256 bytes, which is given by the standard [50]. The metrics are packet rate, packet loss and latency that together define Quality of Service (QoS). The maximum VNF throughput is defined as the maximum throughput with the packet loss of less than 1%. It is also notated as maximum feasible throughput in this thesis.

There are gaps in the previous studies that could be filled in. Those gaps are: 1) the performance of a VNF that handles traffic in UDP protocol; 2) the performance of a VNF dealing with external traffic; 3) the performance of a VNF that runs in containers; 4) the performance of a VNF that runs on multiple physical hosts; 5) the relationship between the resources allocated to a VNF and VNF throughput;

6) the influence from other VNF types on the throughput.

To fill those gaps, some parameters need to be changed to measure the impacts. For the first gap, traffic packets need to adopt UDP. For the second gap, traffic needs to be sent by an external traffic generator and forwarded to an external receiver. For the third gap, the VNF should be run in containers. For the fourth gap, the containers for the VNF need to be allocated in different physical machines in the experiments. For the fifth gap, the amount of resources given to the VNF should be changed during the experiment. Also, because the VNF comprises of one or more containers, resources given to the VNF should be changed in the container level. The resources are mainly CPU resources and therefore, they are changed on VNF level and container level. For the sixth gap, a different type of containers should be run on the same host where the VNF runs.

Therefore, this study is based on the structure of previous studies and aims at filling the gaps.

5.2 Experiment Design

In order to fill the gaps, measurements are conducted in the following steps:

1) The Kubernetes cluster is configured so that it can receive traffic from outside the cluster.
2) VNFs are running on the Kubernetes cluster as Pods.
3) Traffic generator sends packets to the Kubernetes cluster using UDP protocol and gradually increases the packet rate.
4) Traffic can be routed to Pods by load-balancing mechanisms.
5) Data is collected by Prometheus.
6) MetalLB controller runs on *Host 4* while Ingress controller runs on *Host 2*.
7) CPU isolation is enabled and the CPU limits and requests of a Pod are set to a constant. The number of Pods is changed to measure the influence of number of Pods on VNF throughput.
8) CPU isolation is disabled and the step 7) is repeated.
9) The number of Pods is set to a constant. The CPU resource (CPU limits and requests) is changed to measure the impact of CPU resource on VNF throughput. In this case, CPU isolation is disabled because of the single-thread benchmark and the requirement that Kubernetes CPU manager can enable CPU isolation only when CPU limit is set to an integer.

10) The CPU resource and number of Pods are set to a constant. Pods are allocated on different hosts to measure the influence of the load-balancing mechanism on VNF throughput. Traffic is sent to the IP address of Service. All Pods are first allocated to Load-balancer Controller Node (LCN). Then, half of the Pods are allocated to MCN and the other half on non-loadbalancer Controller Node (NLCN). Last, all Pods are allocated to NLCN.

11) Traffic is sent to the IP address of Ingress and step 10) is repeated.

12) Location of MetalLB and Ingress controllers are run in the same Host and step 10) and 11) are repeated to test the impact of load-balancing mechanisms on VNF throughput.

13) The CPU resource and number of Pods are set to a constant. Pods are allocated on a Host. Another type of Pods, which runs intensive local calculations, is run on the same host to measure the influence from other Pods on the VNF throughput. The new type of Pods is first run with CPU limits and then without CPU limits.

14) Kubernetes cluster composition can be changed to test the impact of CPU types on VNF throughput.

The experiment setup is composed of hardware and software structures. The hardware structure illustrate the hardware specification of hosts and interconnection of hosts. The software structures illustrate the software setup and the benchmarks for tests.

5.3 Hardware Structure

The hardware adopted in this study is composed of five physical machines. Each machine is numbered for identification. The numbers of machines are 0, 2, 3, 4, 5 and 6. Specifications of the machines are listed in Table 5.1. All CPUs have hyper-threading enabled and therefore, the number of CPU logical cores equals to twice the number of physical cores.

The machines are connected by two switches. Each switch handles one network. The network topology is shown in Graph 5.1.

All machines are connected to *switch 1* in subnet 10.10.10.0/24. *Host 2*, *4*, *5* and *6* are also connected to switch 2 in subnet 10.10.9.0/24. Switch 1 has a capacity of 100 Mbps while *switch 2* has 1000 Mbps.

Host 3 is also connected to the Internet, which is not shown in Graph 5.1

		Host 0	*Host 2*	*Host 3*	*Host 4*	*Host 5*	*Host 6*
CPU logical cores No.		8	64	8	8	8	56
Max. CPU frequency/GHz		2.5	3.2	2.5	3.9	3.9	2.8
CPU Micro-architecture		Haswell	Skylake	Haswell	Haswell	Haswell	Broadwell
Memory/GB		4	32	4	32	32	8
Storage/GB		150	120	110	1000	120	230
Network Interface Card 1	Name	enp0s25	enp94s0	ens4	enp5s0	enp5s0	ens15f0
	Capacity/ Mbps	100	100	100	100	100	100
	IP address	10.10.10.30	10.10.10.33	10.10.10.1	10.10.10.31	10.10.10.32	10.10.10.29
Network Interface Card 2	Name	-	traffic0	enp0s25	traffic0	traffic0	ens15f1
	Capacity/ Mbps	-	1000	1000	1000	1000	1000
	IP address	-	10.10.9.12	Internet	10.10.9.10	10.10.9.11	10.10.9.23

Table 5.1. Hardware Specifications

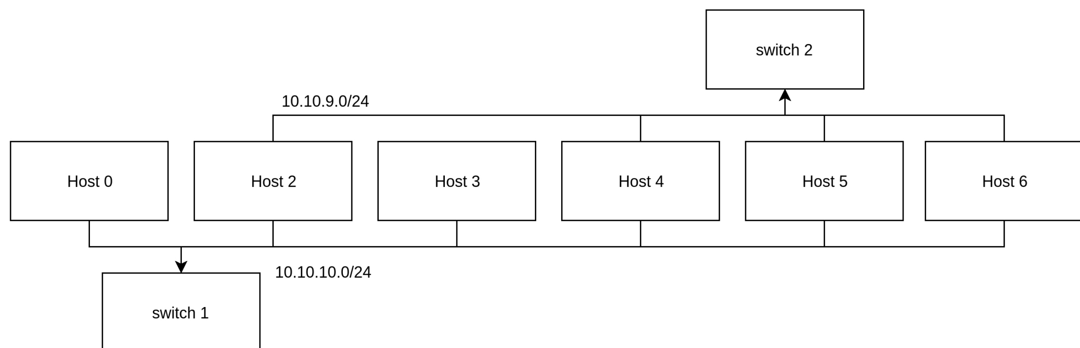

Figure 5.1. Network Topology

Network 10.10.9.0/24 is used as a traffic network. Network 10.10.10.0/24 is used as a control network.

5.4 Software Structures

5.4.1 MAAS

The MAAS setup in this study follows the setup proposed by Krishnakumar [4]. MAAS enables easy setup and management for private clusters. The MAAS structure is shown in Graph 5.2. *Host 3* runs as the MAAS controller. It provides a subnet (10.10.10.0/24) and a corresponding DNS server, while other machines run as MAAS nodes and are managed by the controller through the subnet. Each machine has an IP address in the MAAS subnet. The IP addresses are shown in Graph 5.2. *Host 3* has direct access to the Internet and all other nodes can access the Internet through *Host 3*.

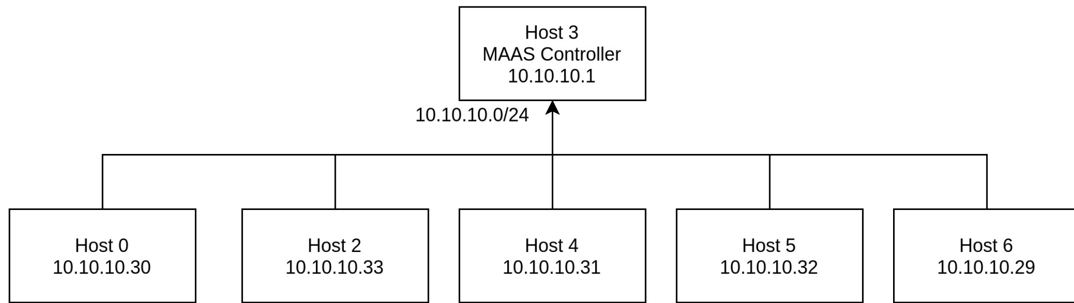

Figure 5.2. MAAS Structure

The MAAS nodes requires an IPTable forwarding rule to get access to the Internet. Host 3 is the machine that bridges two networks. NIC ens4 connects to the MAAS subnet and NIC enp0s25 connects to the Internet. Therefore, on Host 3, the following commands can be run to forward traffic between MAAS nodes and the Internet.

```
$sudo iptables --table nat --append POSTROUTING \
          --out-interface enp0s25 -j MASQUERADE
$sudo iptables --append FORWARD --in-interface ens4 -j ACCEPT
```

5.4.2 Kubernetes

The composition of the Kubernetes cluster changes during this study. *Host 0* is the Kubernetes Master. There are always two Kubernetes Nodes in the Kubernetes cluster. One of the Nodes is always *Host 4* while the other one is either *Host 2* or *Host 5*. Kubernetes shares the same control network as MAAS in the subnet 10.10.10.0/24. Kubernetes cluster structure in this study in shown in Graph 5.3.

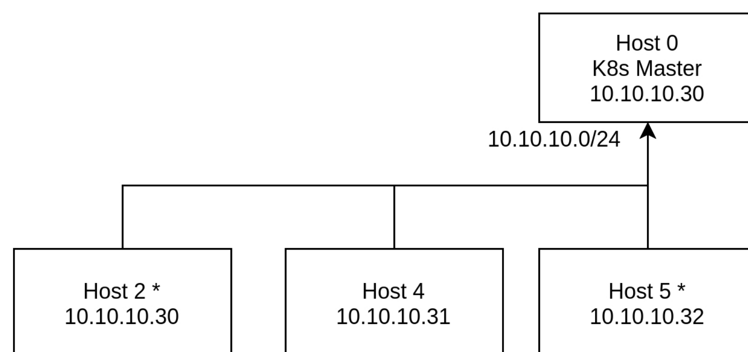

Figure 5.3. Kubernetes Cluster Structure

Docker, kubelet and kubeadm are mandatory for all machines in a Kubernetes cluster. The installation script is available in Appendix A.3.

After the installations for Docker, kubelet and kubeadm, the Kubernetes cluster can be set up by using kubeadm. In *Host 0*, the following command can be run to initiate Kubernetes Master:

```
$kubeadm init --config ./kubeadm.config
```

The Kubernetes cluster needs to run in IPVS mode in order to have a higher performance [53] and the IPVS should use the least connection algorithm for load-balancing. These can be stated in the kubeadm.config file. Contents for the kubeadm.config is appended on Appendix A.3.

After the Kubernetes Master has been initiated, a command that can be used to add Nodes to the Kubernetes cluster would be available in the terminal. After adding *Host 2/5* and *Host 4* to the cluster, 'kubectl get node' can be run to check the availability of the Nodes. But at this moment, all the nodes should be in NotReady state because at least one CNI plug-in is needed for Pods to communicate with Kubernetes API Server. Multus and Flannel are the CNI plug-ins adopted in this study. Flannel requires Pods to run in subnet 10.244.0.0/16. Therefore, when initializing the cluster, the podSubnet in ClusterConfiguration should be set to the range. Kubernetes subnet is illustrated in Graph 5.4.

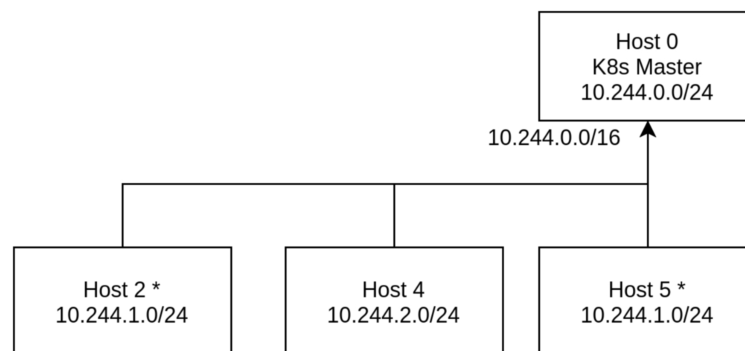

Figure 5.4. Kubernetes Subnet Topology

This subnet runs through switch 1 and thus, it shares the same network as the MAAS and the Kubernetes control network.

The initialization of Multus CNI and Flannel can be done by running the following commands on *Host 0*:

```
$git clone https://github.com/intel/multus-cni.git
        && cd multus-cni
$cat ./images/{multus-daemonset.yml,flannel-daemonset.yml}
        | kubectl apply -f -
```

The Multus can create multiple network interfaces for a Pod. This allows Pods to run in two networks. As ETSI TS 129 060 V12.6.0 [50] indicates, the GTP has

three protocols: GTP-U, GTP-C and GTP', where GTP-U is used in a traffic network while GTP-C and GTP' are used in a control network. This division is critical for a VNF. Traffic network requires a larger bandwidth while control network requires smaller. Control network should be more robust while traffic network is allowed to have larger network fluctuations and packet loss. The division of two networks between traffic and control signals enables better stability for the VNF.

Pods in this study have two network interfaces; one is used for Kubernetes control network and the other one is for running traffic. This can be configured by the Multus ConfigMap shown in Appendix A.4.

The traffic network interface adopts macvlan as its network driver and *traffic0* as its network device on the host. The IP addresses of the traffic network interfaces ranges from 10.10.9.50 to 10.10.9.149. The traffic travels directly from network interface *traffic0* to Pods.

MetalLB can assign an external IP address for Kubernetes Objects. It can be installed by running the following command on *Host 0*:

```
$kubectl apply -f ./metallb.yaml
```

The metallb.yaml implements the default configuration [54] but with nodeName of the controller specified as nodeName: Host2 or nodeName: Host4. This could measure the impact of controller location on VNF throughput.

The external IP address for Kubernetes Objects can be configured by MetalLB ConfigMap as shown in Appendix A.5. This ConfigMap indicates that MetalLB runs in layer 2 mode and the external IP address assigned by MetalLB would range between 10.10.9.150 and 10.10.9.253.

Ingress can collect Services into one IP address and thus acts as a load-balancer for Services. The ingress can be installed by running the following command:

```
$kubectl apply -f ./mandatory.yaml
```

The mandatory.yaml implements the default ingress controller setup file [54] with changes in the port and protocols, as shown in Appendix A.6. For Ingress to work with UDP traffic, tcp-services and udp-services need to be configured. Also, according to the methodology adopted in this study, the location of the Ingress controller would change between *Host 2* and *Host 4*, stating by nodeName: Host2 or nodeName: Host4.

Vnf-benchmark is the Service name for the VNF benchmark used in this study and 2152 is the port used by GTP-U. In order to expose Ingress with an external IP address, a Service object for Ingress is needed and it is shown in Appendix A.7.

This Service collects UDP traffic from Ingress controller at port 2152 and can be accessed by an external IP address with the type set to LoadBalancer. Ingress rules could be configured by Kubernetes Ingress Object with following command run in *Host 0*:

```
$kubectl apply -f ./ingress.yaml
```

Contents of the ingress.yaml is shown in Appendix A.8.

This Ingress rule selects vnf-benchmark-svc at port 2152 as its backend for the IP address and port described in the previous step.

Data is collected by Prometheus. This study uses kube-prometheus for simple setup of Prometheus to monitor the Kubernetes cluster and to collect metrics. It can be set up by running the following commands in *Host 0*:

```
$git clone https://github.com/coreos/kube-prometheus.git
$cd kube-prometheus
$kubectl create -f manifests/
$until kubectl get customresourcedefinitions \
servicemonitors.monitoring.coreos.com ;
do date; sleep 1; echo ""; done
$until kubectl get servicemonitors --all-namespaces ; do date; \
sleep 1; echo ""; done
$kubectl apply -f manifests/
```

This installs all components and services for monitoring and can be accessed on http://localhost:9090 on *Host 0* with the following command run:

```
$kubectl --namespace monitoring port-forward
svc/prometheus-k8s 9090
```

All the procedures for setting up the Kubernetes cluster are done.

5.4.3 Benchmarks

Benchmarks are composed of a traffic generator, a packet forwarding VNF and a receiver. Traffic is transmitted in traffic network 10.10.9.0/24. The traffic generator sends traffic to the packet forwarding VNF before the traffic is redirected to the receiver. Interconnections of the benchmarks are illustrated in Graph 5.5. Traffic flow is illustrated in Graph 5.6.

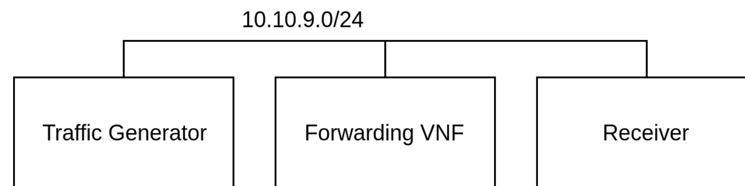

Figure 5.5. Interconnection for Benchmarks

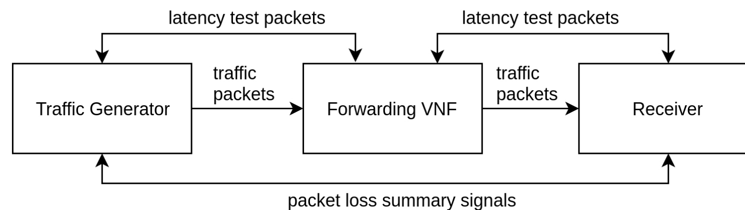

Figure 5.6. Traffic Flow among Benchmarks

The traffic generator sends traffic packets to the forwarding VNF running on the Kubernetes cluster. The VNF forwards packets to the receiver. The traffic generator also sends a test latency packet in every second to the forwarding VNF. The latency packet will then be sent to the receiver. The Receiver will respond the packet back to the traffic generator to obtain Round Trip Time (RTT) of the VNF. The traffic generator encodes an identification number in the header of traffic packets. The number changes in every two seconds. Meanwhile, the receiver calculates the number of packets received within the two seconds and sends the information back to the traffic generator. Hence, the traffic generator can calculate packet loss rates in every two seconds when the number of the packets received by the receiver is divided by the number of packets it sends with the specific identification number.

Kubernetes Nodes are allocated to different hosts during the study to test the impact of CPU clock frequency and architecture of the hosts on VNF throughput. The Hosts marked with * in Graph 5.7 indicates that a host is a Kubernetes host running VNF in some experiments and it changes to traffic generator in some other cases. The Kubernetes cluster always has two hosts. *Host 4* is always a Kubernetes Host while *Host 2* and *Host 5* swaps their roles.

The flowchart for the traffic generator is illustrated in Graph 5.8.

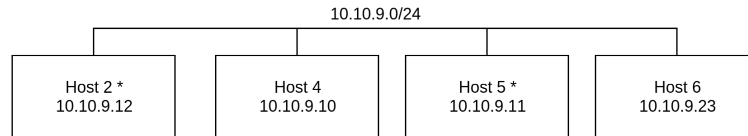

Figure 5.7. Traffic Network Topology of the Kubernetes Cluster

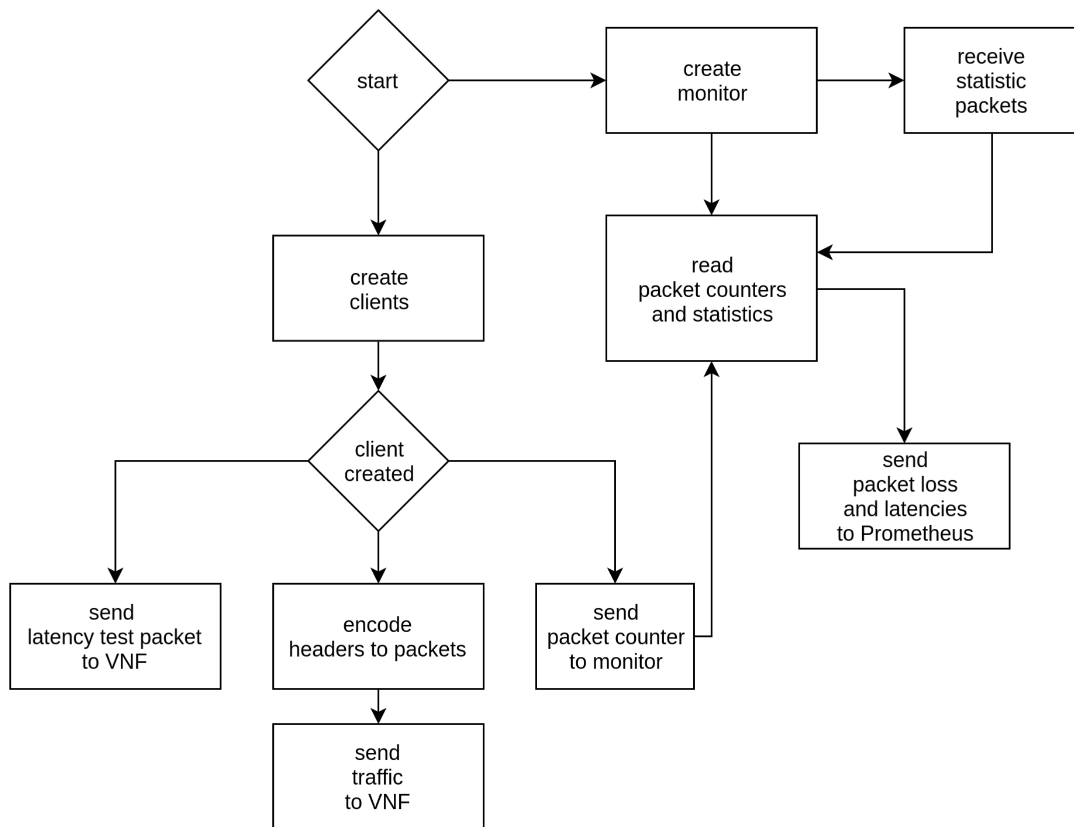

Figure 5.8. Flowchart for Traffic Generator

The traffic generator starts with creating two processes. One process handles the monitoring of statistical counters while the other one creates traffic clients to send traffic packets and test packets. For every traffic client, it sends a latency test packet to VNF once per second. The test packet has a timestamp of the moment when it is sent. The traffic clients keep sending traffic packets in UDP protocol to the VNF by a varying interval. The interval decreases by 5% in every 10 seconds to increase the packet rate of the client. The traffic packet is encoded with a GTP header described in Section 3.3.2 with a TEID that changes in every two seconds. All clients have the same TEID in the GTP header over the two-second interval. At the end of the interval, an ending signal is sent to the receiver at port 2123 with the TEID. Each client stores a counter for the traffic that it has sent before the counter gets sent to the monitoring process. The monitoring process receives counters from clients and calculates the total packet rate for all the clients. Then the packet rate will be sent to Prometheus and stored in the TSDB for further usage. The monitoring process listens to port 2123 and waits for statistical packets

sent from the VNF and the receiver. The statistical packets contain information about RTT from the traffic generator to the VNF and the receiver, together with the number of packets with a certain TEID received by the receiver. The number of packets received by the receiver is compared to the number of packets sent with the same TEID, thereby obtaining the packet loss.

The flowchart for the VNF forwarding benchmark is illustrated in Graph 5.9.

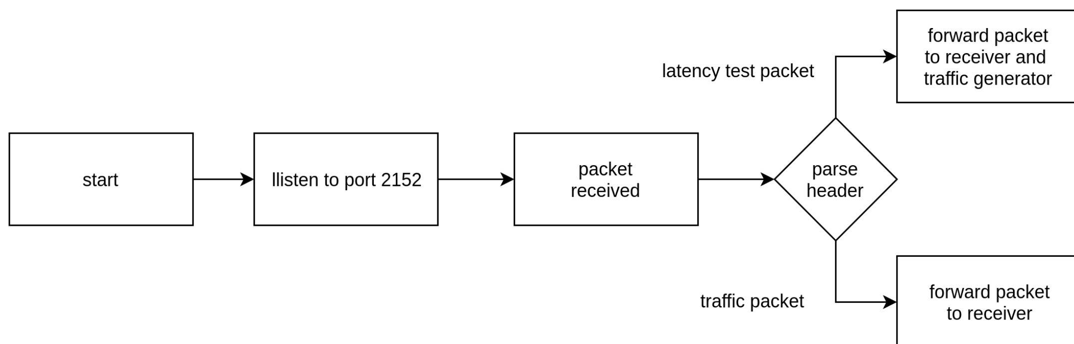

Figure 5.9. Flowchart for the VNF

The forwarding VNF listens to port 2152, which is defined in ETSI [50]. When it receives a packet, it parses the header of the packet to check the type of the packet. If the packet is a latency test packet, the VNF sends the packet back to the traffic generator as well as to the receiver. If the packet is a traffic packet, the VNF sends the packet to the receiver only.

The flowchart for the receiver is shown in Graph 5.10.

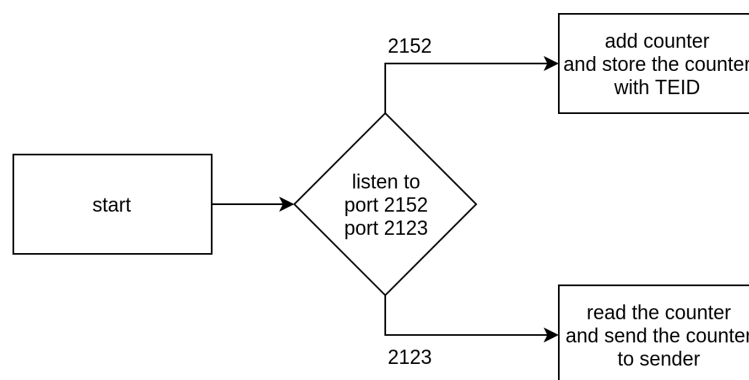

Figure 5.10. Flowchart for the Receiver

The receiver listens to port 2152 and 2123. When a packet arrives from port 2152, the packet header is parsed to obtain a TEID, which is used for counting the number of packets sent by the VNF. When a packet arrives from port 2123, the packet is parsed to obtain a TEID which is used to read the counter in the previous step. The counter will then be sent to the sender to calculate packet loss. For the receiver to have the best performance, port reusing is enabled [55] and more than three threads are running and fetching data from port 2152.

The VNF Service Object and Deployment is shown in Appendix A.2. After the

VNF is deployed, Service has an IP address of 10.10.9.150 and Ingress has an IP address of 10.10.9.151.

6. Results

This chapter contains results for 7 experiments. Each experiment has multiple test cases. Each test case is labelled with Section number-test case number. For example, the first test case in the first experiment is labelled as 6.1-1. If the experiment has more than one subset of experiments, the test cases is labelled as Section number-subset number-test case number. For example, the first test case in the first experiment subset in the fourth experiment is labelled as 6.4-1-1.

6.1 Impact of Number of Pods

This experiment measures the impact of the number of Pods on VNF throughput. The number of Pods varies from 1 to 5. The number of Pods is increased by 1 between adjacent test cases. The experiment results are shown in Figure 6.1. Each column in the graph indicates one test case. Each test case has three rows for three metrics.

The x-axis in all graphs indicates the time that passes in the test in seconds. Time ranges from 0 to 1000 seconds for all tests.

The first row shows the throughput of the VNF. It is notated in packets per second (pps). In this row, blue curves indicate the number of packets sent by the Traffic generator within one second, ranging from 10,000 to 400,000 pps. Red curves indicate the number of packets received by the Receiver within one second.

The second row shows the packet loss of the VNF in the y-axis. The packet loss range is shown up to 10% to illustrate the effect on the level where the packet loss starts to increase rapidly while still having enough granularity on the lower end of the range where the QoS targets for typical telecommunication VNFs can be met. If a point is missing in the row, it is either not recorded due to network connection problems or the measured loss is higher than 10%.

The third row shows the latency of the VNF in the y-axis in milliseconds. The latency range is determined by the maximum value in the corresponding exper-

iment and is the ceiling integer of the maximum value. If the latency is higher than 5 ms, it will be missing from the graph.

The horizontal dashed line in the figure indicates the best throughput from all the test cases in this experiment. The vertical lines indicate the moment when the test case has the best throughput. The maximum throughput of each test case in the experiment is indicated by a dot which is in the intersection of the vertical dashed line and the blue curve. The red dot indicates the best throughput case in the experiment.

MetalLB controller is running on *Host 4*. Each Pod has a CPU limit of 1 vCPU and the Pods are running on *Host 4*. The packet forwarding benchmark is a single thread programme, which means it cannot benefit from having more than 1 vCPU. Therefore, CPU limits would have to be 1 vCPU. The Pod will be run with CPU isolation turned on, meaning that the benchmark has an exclusive CPU. The first column is the result for running the VNF in one Pod. The last column is the result for running the VNF in five Pods.

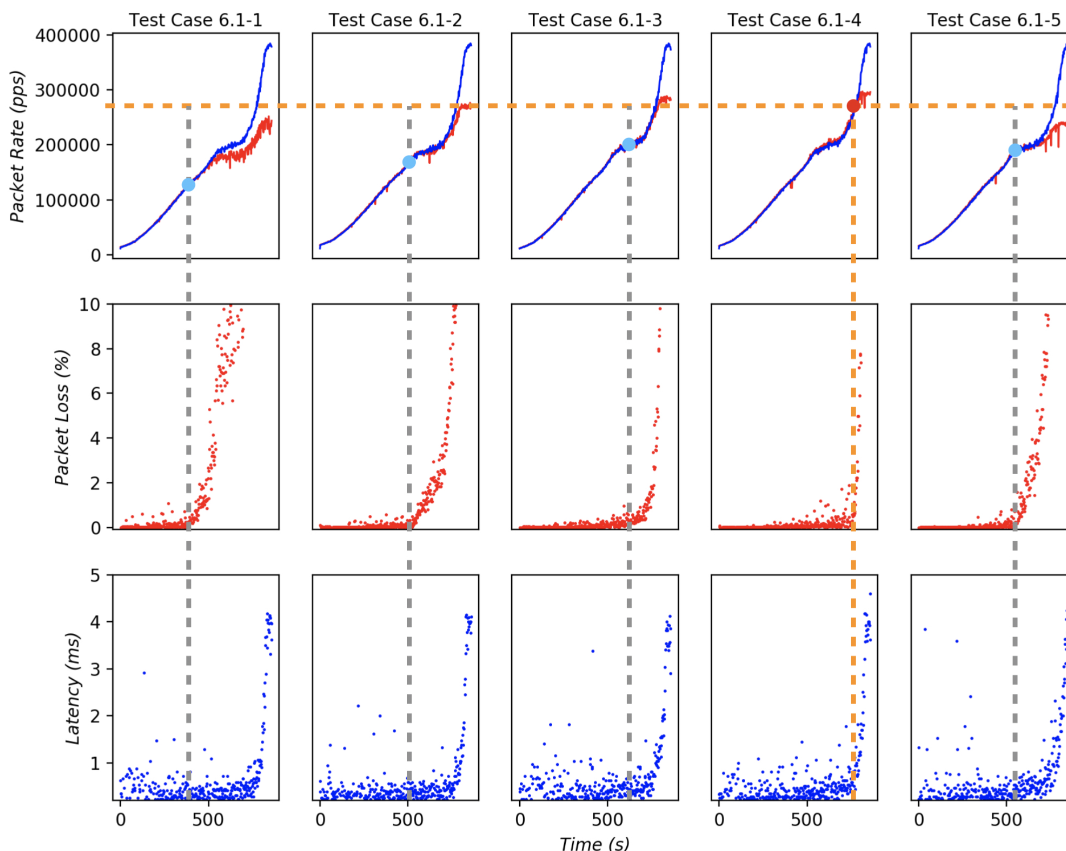

Figure 6.1. Results for the Impact of Number of Pods

The VNF gets the best throughput when it is run in four Pods with CPU limited to 1. When the VNF is run in one Pod, packet loss begins to rise earlier and the variance in latency increases. When the VNF is run in more than four Pods, the maximum throughput drops when the number of Pods increases.

The figure in the first row in the first column shows that 140 kilo packet per second (kpps) is the maximum packet rate with feasible packet loss. Hence, a conclusion can be drawn from the figure that one vCPU is capable of handling at maximum 140 kpps. When the number of Pods increases, the VNF gets more CPU time for processing the packet and the VNF is able to process more packets. The throughput increases almost twofold to 270 kpps when three additional Pods are added. However, when more Pods are added, the throughput does not get improved but instead, it becomes worse. The figures in the second row show that increasing the number of Pods decreases the variance in packet loss. However, when the VNF is run on 5 Pods, variance increases. When the figures in the second and third row are examined together, it shows that when packet loss increases, latency would also increase.

To summarize, the maximum feasible packet rate for 1 vCPU is 140 kpps and the maximum feasible packet rate that the VNF can reach is 270 kpps when the VNF is run in four Pods. Increasing the number of Pods will deteriorate the throughput.

6.2 Impact of CPU Isolation

In the previous experiment, each Pod has an exclusive CPU that it can run programs without affecting other Pods or getting affected by other Pods. In this experiment, Pods are running without CPU isolation, which means they are sharing CPUs and the programs running on those Pods could run on any of the shared CPUs in runtime.

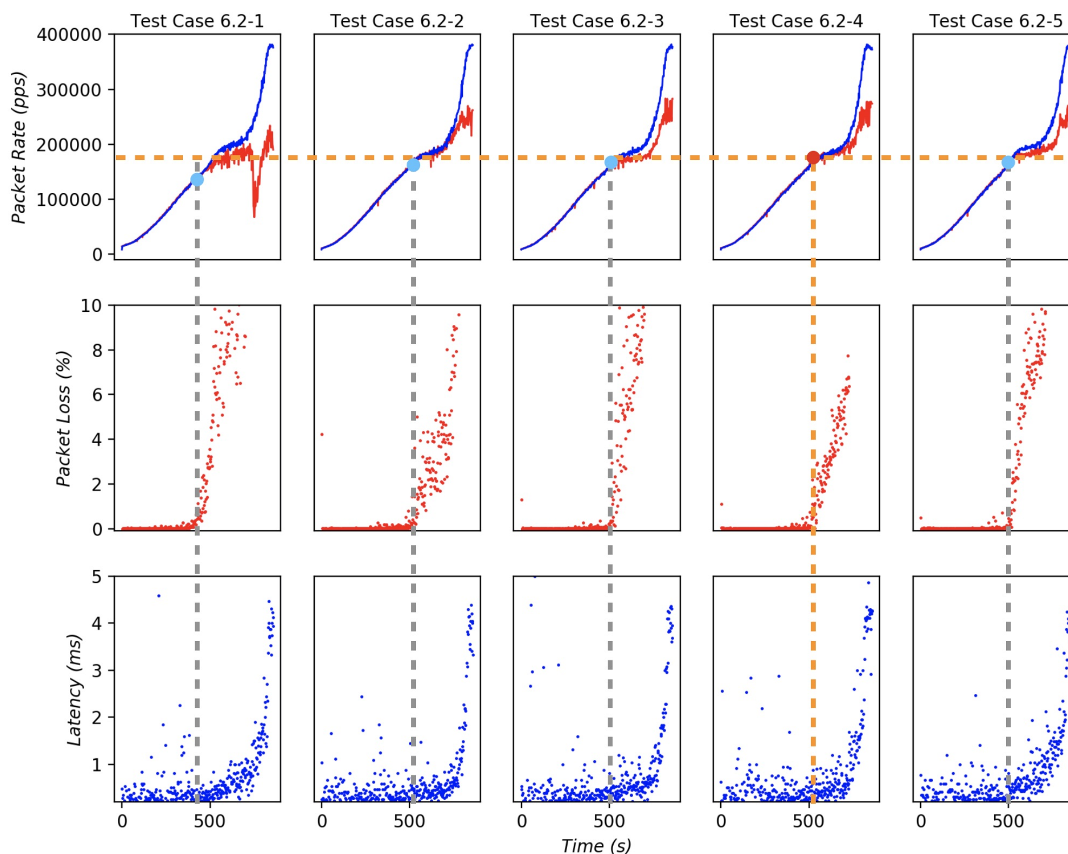

Figure 6.2. Results for the Impact of CPU Isolation

The results of this experiment are shown in Figure 6.2. The graph structure is the same as the previous experiment. The experiment method is slightly different from 6.1 that Pods are run with qosClass set to Burstable instead of Guaranteed in the previous case. qosClass is a keyword in Pod specification that sets CPU isolation for the Pod. It can be set to either Burstable (CPU isolation off), or Guaranteed (CPU isolation on).

When the VNF is run in one Pod, the maximum feasible throughput is similar to 6.1-1 at around 140 kpps. However, the throughput drops to 100 kpps when the traffic is 250 kpps and eventually increases to 200 kpps level without the CPU isolation while CPU isolation in 6.1-1 rises the throughput to 220 kpps. The sudden drop happens when the QoS of the application is not acceptable and therefore, the corresponding throughput is not useful in practice. But it indicates the instability of the VNF when CPU isolation is not enabled.

When the VNF is run in two to four Pods, the maximum feasible throughput is lower than the one in 6.1-2, 6.1-3 and 6.1-4. When the VNF is run in five Pods, the maximum throughput in the non-isolated test case is 180 kpps, which is the same as in the isolated case. However, CPU isolation keeps the packet loss low while the non-isolated cases have higher packet loss on average.

To summarize, CPU isolation brings better and more stable throughput for the

VNF.

6.3 Impact of CPU Limits

The experiment measures the impact of CPU limits on the throughput. CPU isolation is not enabled in this experiment. Each Pod has a maximum CPU limit of 1 vCPU because of the single-thread VNF benchmark. The VNF is composed of five Pods and the Pods are running on *Host 4*. The CPU limit for the Pod will change from 200m to 1000m and has the same amount of total CPU time ranging from 1 to 5 as the experiment in Section 6.2 and therefore, this experiment can be compared with the previous section.

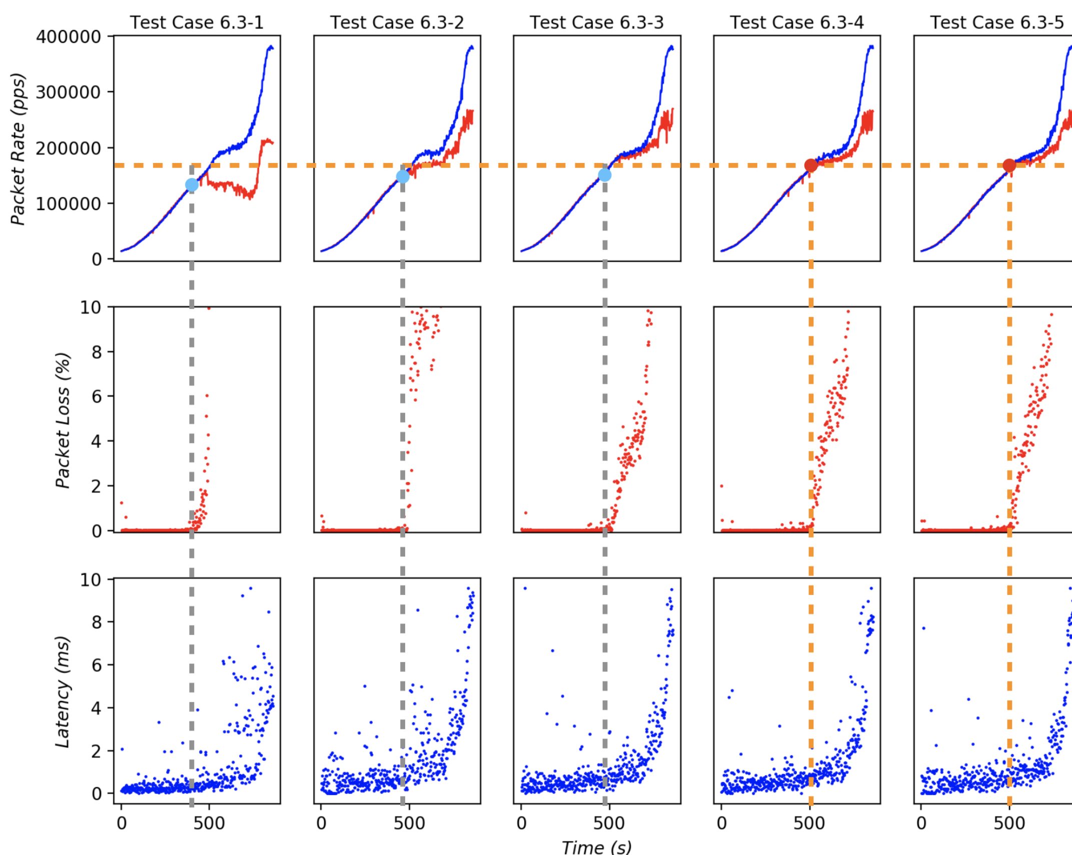

Figure 6.3. Results for the Impact of CPU Limits

The experiment results are shown in Figure 6.3. The first column runs the Pod with CPU limit of 200m. The last column runs the Pod with CPU limit of 1000m. The CPU limit increases by 200m between adjacent test cases. The graph structure is the same as in the previous experiments.

When the CPU limit is set to 200m for a Pod, the VNF has 1 vCPU in total and it has a maximum feasible throughput of 140 kpps, which is the same as in the case 6.2-1 (both with or without CPU isolation enabled). Latency is similar to the first test case in the first experiment.

When the CPU limit is set to 400m and 600m for a Pod, the VNF has 2 vCPU and 3vCPU in total respectively. The maximum throughput in two test cases is similar. The maximum VNF throughput increases to 150 kpps. It is 10 kpps more than in 6.3-1 but 10 kpps less than in 6.2-2 and in 6.2-3.

The results for the cases with CPU limits of 600m and 800m are similar to each other. They have maximum feasible throughput around 170 kpps, which is slightly lower than in 6.2-4 and 6.2-5.

Latencies of these cases do not differ much from each other. When the CPU limit increases, the variance of latency increases slightly but the latency still stays within 2.5 ms.

To summarize, this experiment has a similar throughput compared to 6.2. The best throughput in this experiment is reached in the case with CPU limit of 600m.

6.4 Impact of VNF Pod Location and Load-balancing Techniques

This experiment measures the impact of Pod location on VNF throughput. In this experiment, the same tests as in 6.1 are conducted with all the Pods running on *Host 5* instead of running on *Host 4*. Both the Ingress controller and MetalLB controller are run on *Host 4*.

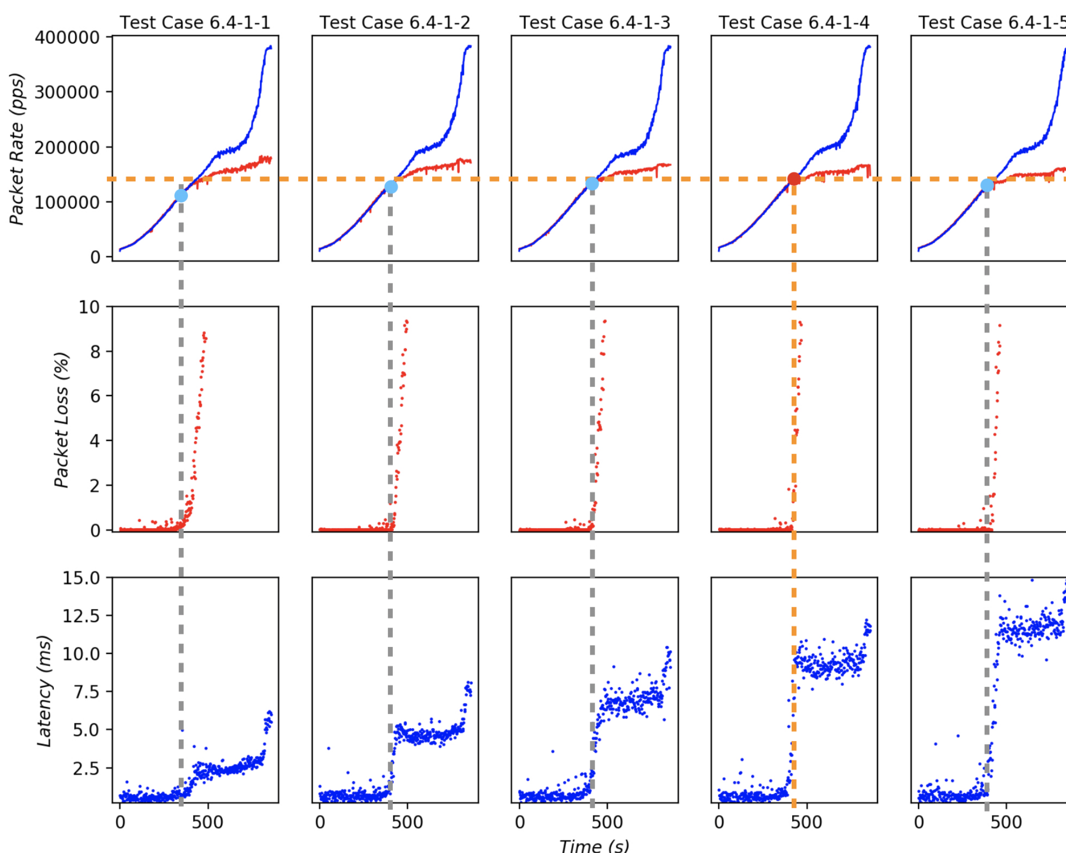

Figure 6.4. Results for the Impact of number of Pods with VNF Pods running on *Host 5*

The results of this experiment are shown in Figure 6.4. The maximum VNF throughput is 130 kpps. Increasing the number of Pods does not increase the maximum VNF throughput but it increases the latency of the VNF when the packet loss starts to increase.

The results show that the VNF throughput depends on the physical host of the Pods. Therefore, an experiment for running Pods in different Hosts should be conducted to measure the impact of Pod location. Kubernetes uses two load-balancers: Ingress and Service. Therefore, the performance of the different load-balancing techniques should also be compared.

The results for different combinations of VNF Pod locations and load-balancing techniques are shown in Figure 6.5. The VNF is run in 4 Pods with each having 1 exclusive vCPU. The graph structure is illustrated in Table 6.1.

Load-Balancer	Ingress	Service
All Pods run on *Host 4*	Test 1	Test 2
2 Pods run on *Host 4* 2 Pods run on *Host 5*	Test 3	Test 4
All Pods run on *Host 5*	Test 5	Test 6

Table 6.1. Figure Structure for the Results of Pod Locations

The first column in Figure 6.5 indicates that the traffic generator sends traffic to 10.10.9.151, which is the address for Ingress service. The second column in Figure 6.5 indicates that the traffic generator sends traffic to 10.10.9.150, the address for Kubernetes Service. Each column contains three test cases for different locations for VNF Pods. Hence, there are 6 test cases in this experiment: 1) All Pods are run on *Host 4* with traffic sent to Ingress (Test 1); 2) All Pods are run on *Host 4* with traffic sent to Service (Test 2); 3) Half of the Pods are run on *Host 4* while the other half on *Host 5* with traffic sent to Ingress (Test 3); 4) Half of the Pods are run on *Host 4* while the other half on *Host 5* with traffic sent to Service (Test 4); 5) All Pods are run on *Host 5* with traffic sent to Ingress (Test 5); 6) All Pods are run on *Host 5* with traffic sent to Service (Test 6). Each test case contains three rows for three metrics that are the same as in the previous tests.

The best VNF throughput is reached in 6.4-2-2 with a maximum feasible throughput of 270 kpps. The second-best throughput is obtained by 6.4-2-1 with a maximum feasible throughput of 190 kpps. 6.4-2-5 and 6.4-2-6 have similar maximum VNF throughput and they rank third in this experiment. 6.4-2-3 and 6.4-2-4 have the worst maximum VNF throughput at around 70 kpps. In all cases, packet loss and latency start to deteriorate at the same time.

This experiment draws a conclusion that to obtain the best throughput, Pods

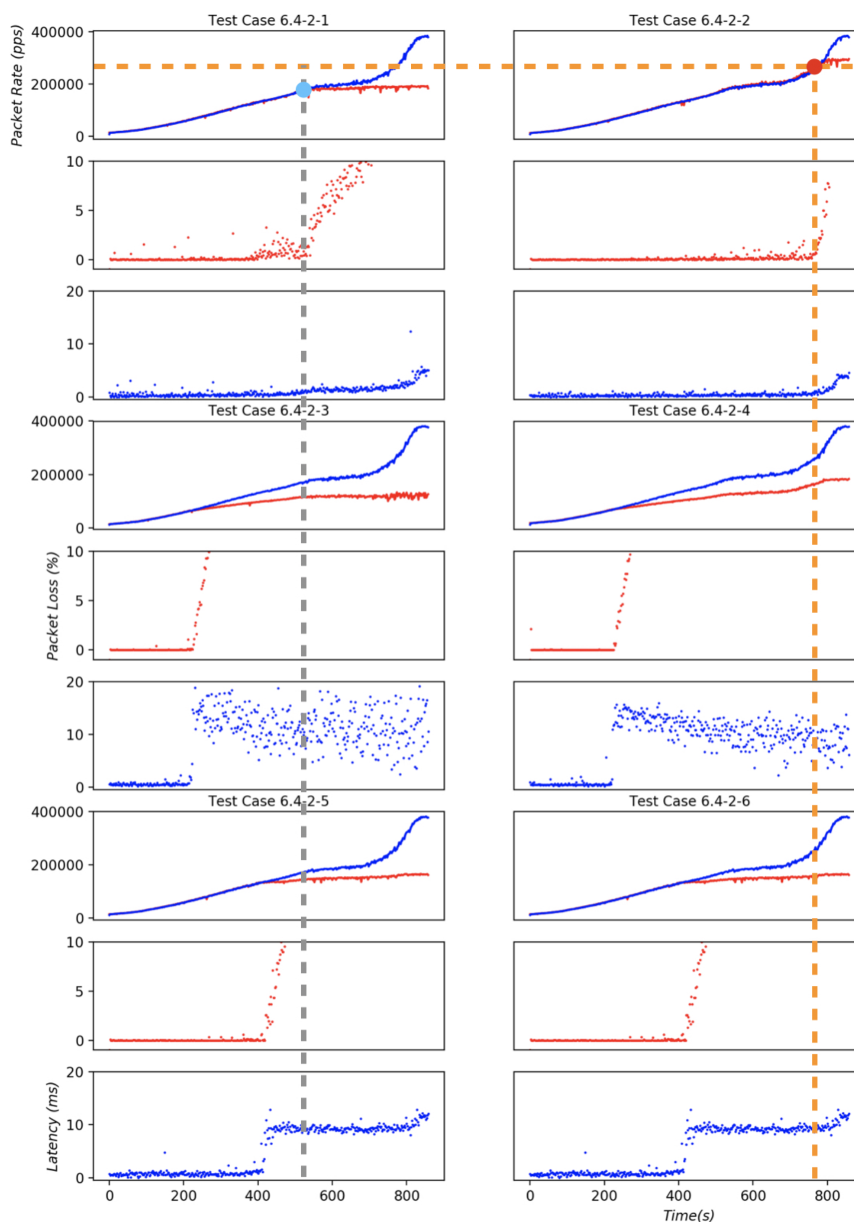

Figure 6.5. Results for the Impact of Pod Locations and Load-balancers

should run on the Node where the load-balancer controller runs on. If it is not applicable in practice, all VNF Pods should at least be run on the same Node. The least feasible option would be to run VNF Pods on different Nodes.

6.5 Impact of Controller Location

In experiment 6.4, both Ingress controller and MetalLB controller are run on *Host 4*. In this experiment, MetalLB controller stays on *Host 4* while Ingress controller is run on *Host 5*. The VNF is run on four Pods with each having 1 exclusive vCPU.

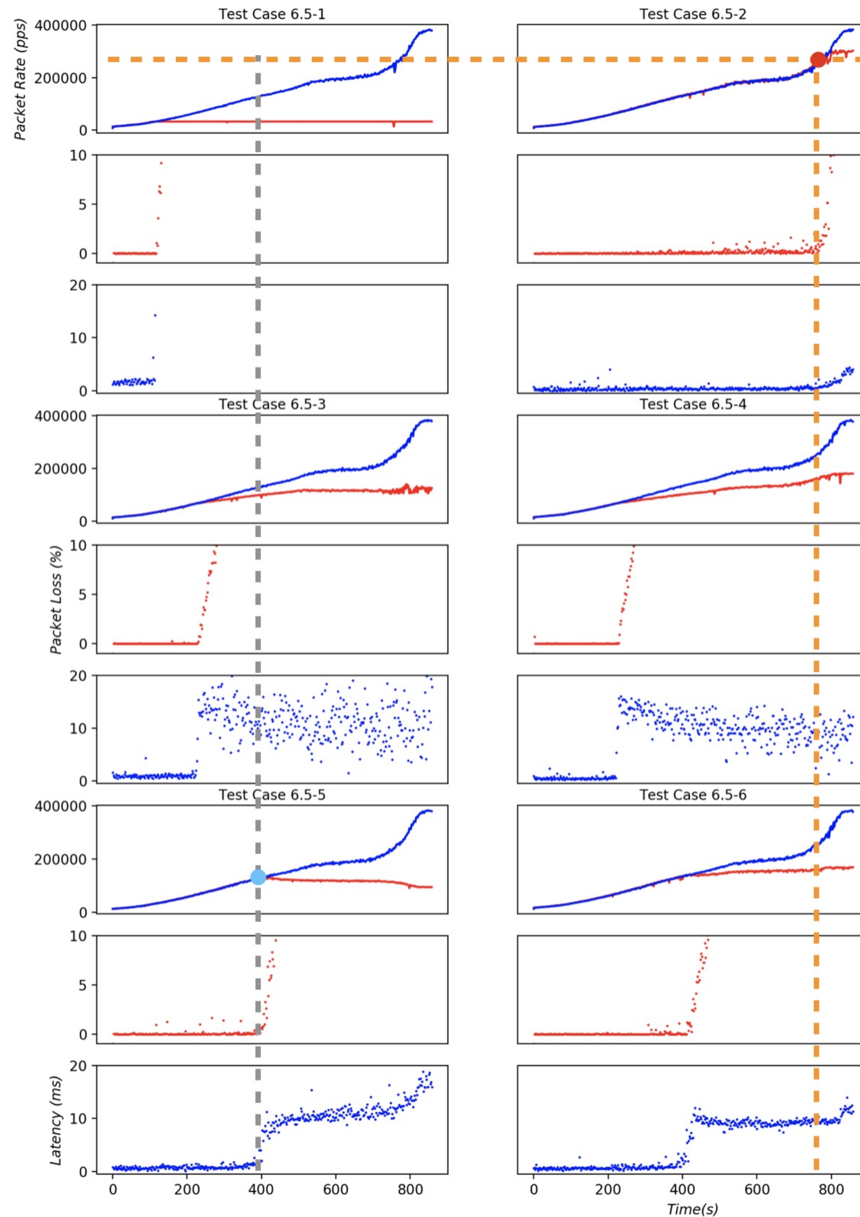

Figure 6.6. Results for the Impact of Pod Locations

The results are shown in Figure 6.6. The results for using Service for load-balancing are the same as in 6.4. The results for using Ingress for load-balancing change because of the change of the location of the Ingress controller.

The best throughput is reached in 6.5-2. 6.5-6 ranks second with around 10 kpps more than the VNF throughput of 6.5-5. 6.5-3 and 6.5-4 remain similar results as in 6.4. 6.5-1 becomes the worst case with a maximum feasible throughput of 30 kpps.

The reason for the differences between 6.4 and 6.5 is that Ingress gets external IP addresses from MetalLB. The traffic will be sent to the Node that runs MetalLB controller before being directed to the Node running Ingress controller. Then the ingress controller will send the packets to Pods.

Most of the cases using Ingress for load-balancing have worse throughput than

the cases in 6.4. Therefore, if the VNF uses Ingress for load-balancing, the Ingress controller should be located in the same Node where MetalLB controller is running to achieve a better throughput.

6.6 Impact of Local Calculation

This experiment measures the impact of local calculations on VNF throughput. The VNF is run on 4 Pods each having CPU limit of 600m. The traffic generator sends traffic at a rate that equals to the maximum feasible throughput in 6.3-3. In this experiment, another Pod type is run on Host 4, which is the same Node where the VNF Pods are running on. The new Pod runs a CPU intensive program. The graph structure is the same as a single column described in 6.1.

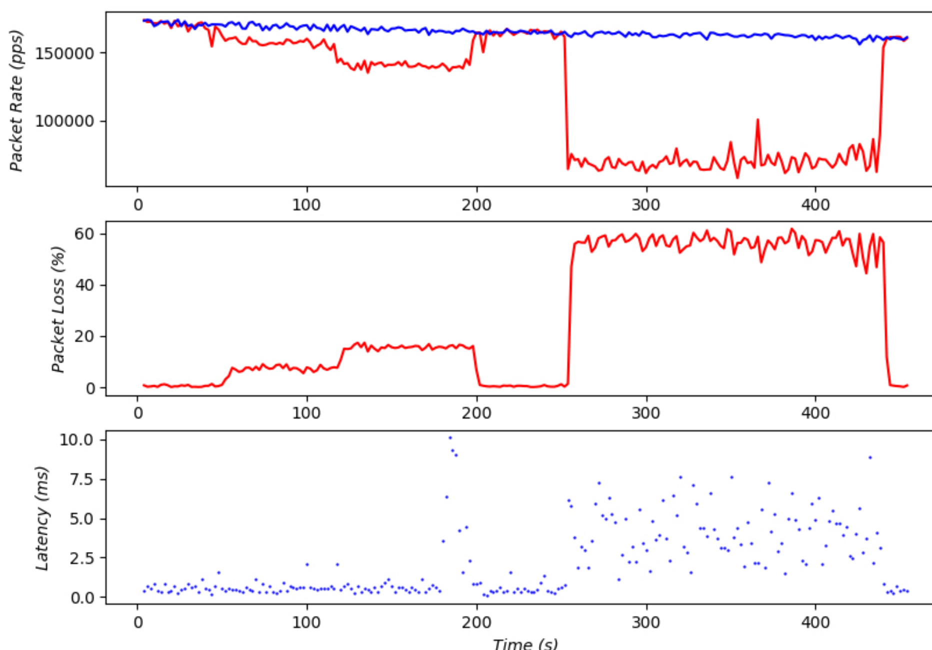

Figure 6.7. Results for the Impact of Local Calculations

The measurements are shown in Figure 6.7. First, each local calculation Pod has a CPU limit of 250m. At 62 seconds from the start, four calculation Pods are deployed to the Node. At 102 seconds, another four calculation Pods are added to the Node. At 142 seconds, another four Pods are added but because the Node runs out of CPU resources, only one additional Pod is running on the Node. At 182 seconds, all local calculation Pods are removed from the Host.

Then, each local calculation Pod is run without specifying CPU limits. At 242 seconds, four calculation Pods are added. At 302 seconds, another four calculation Pods are added. At 342 seconds, another four calculation Pods are added. At 402

seconds, all calculation Pods are removed from the Host.

When the calculation Pods are specified with 250m CPU limit, every time when four additional local calculation Pods are added, the packet loss will increase approximately by 10%-units but it will not increase to more than 18% before the Node runs out of CPU resources. When the calculation Pods are not specified with CPU limits, the packet loss increases significantly to almost 60% when the Node runs four calculation Pods. The packet loss does not increase further when more calculation Pods are added.

To summarize, VNF throughput can be affected by local calculations running on the same Node. In order to minimize the impact from other Pods that run CPU intensive programs, those calculation Pods should have specified CPU limits.

6.7 Impact of Host CPU Type

In the previous experiments, two Kubernetes Nodes have the same type of CPU. In practice, Kubernetes Nodes can have different CPU types. This cloud setup for hosts having different hardware is referred to as non-heterogeneous architecture. This experiment measures the VNF throughput for such architecture.

This experiment repeats the same procedures as in 6.4 but instead of conducting the experiment on *Host 4* and *Host 5*, *Host 4* and *Host 2* are used. The hardware change is illustrated in section 3.1 and marked with asterisks in Graph 5.3.

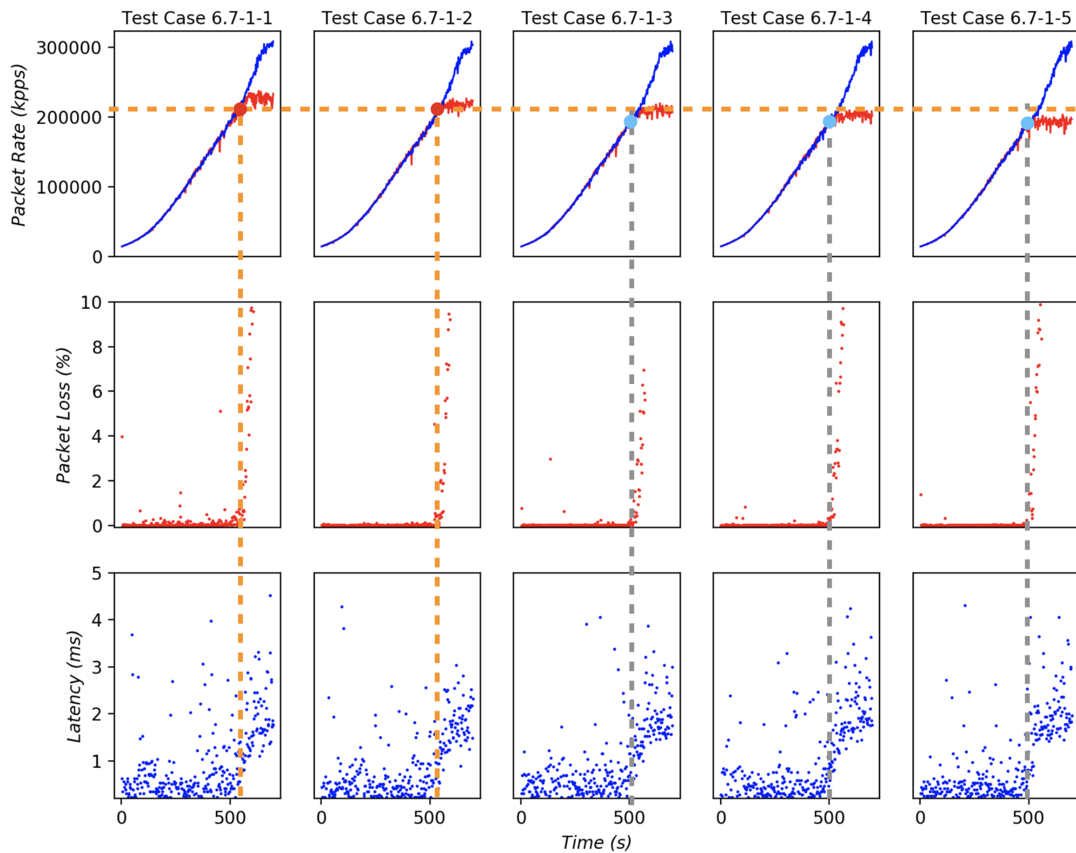

Figure 6.8. Results for the Impact of the number of Pods with Another CPU Type

Results for the effect of the number of Pods is shown in Figure 6.8. The results can be compared with Figure 6.4. The maximum feasible throughput increases from 130 kpps to 210 kpps. Increasing the number of Pods does not affect maximum VNF throughput, which is the same as in 6.4.

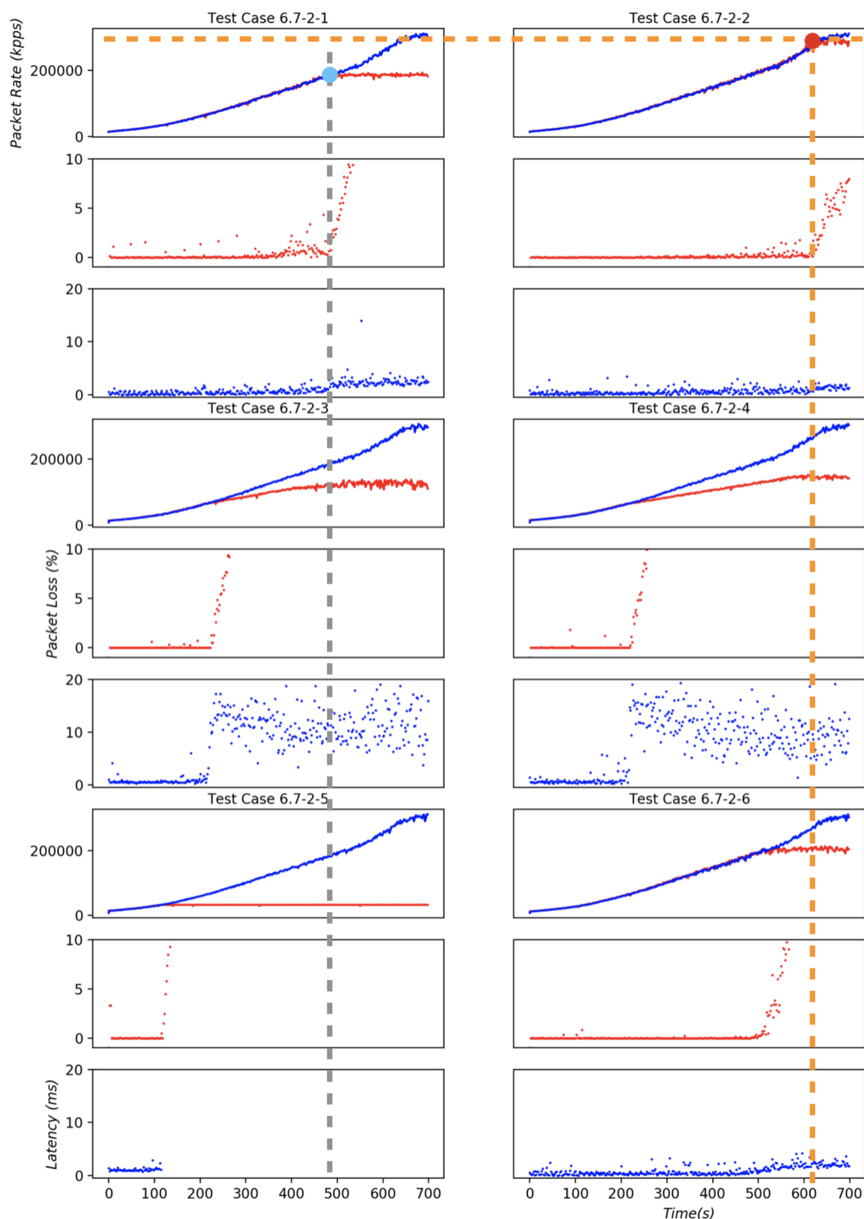

Figure 6.9. Results for the Impact of Pod Locations and Load-balancers with Another CPU Type

The results for Pod locations are shown in Figure 6.9. When comparing the results to the ones in 6.4, The throughput in 6.7-2-1, 6.7-2-2, 6.7-2-3 and 6.7-2-4 remain the same level while 6.7-2-6 in this experiment has 60 kpps higher maximum throughput than the 6.7-2-6 in 6.4. The throughput in 6.7-2-5 decreases dramatically from 124 kpps in 6.4 to 33 kpps in this experiment.

To summarize, CPU types have an impact on the throughput and therefore, results for one CPU type may not be applied to another CPU type.

6.8 Analysis

The maximum VNF throughput is affected by multiple factors, such as the total CPU resource given to the VNF, the number of Pods, Pod locations, controller

locations, load-balancers and CPU types of the hosts. The results in this study give data for designing a scheduler that maximizes throughput for a packet forwarding service, which is a typical use case in the telecommunication industry.

The first experiment shows that one exclusive vCPU is capable to process 140 kpps. Increasing the CPU resources given to the VNF can increase the maximum throughput up to 270 kpps. The VNF throughput almost doubles when the number of vCPUs is increased from 1 to 2. The throughput is slightly improved when the number of vCPU changes from 2 to 4. The throughput decreases when the number increases from 4 to 5. The results indicate that when the VNF is run in 1 vCPU the VNF does not have enough CPU time to process packets. Increasing the CPU resources grants more processing power to the VNF and therefore it yields better throughput. When the VNF is run in more than four Pods, the overheads from load-balancing become significant and that decreases the throughput. The reason for this is the load-balancer. The load-balancer could proceed no more than 300 kpps and it performs worse when the total packet rate from the traffic generator exceeds that level. Four exclusive vCPUs guarantees the maximum throughput in this case. The result shows similar pattern as *Characterising Resource Management Performance in Kubernetes* [6]. The VNF has an optimal number of backends for the best throughput. It means that the performance of the VNF for external traffic in UDP has the same pattern as with internal traffic in TCP. But in this study, the optimal number of VNF backends is smaller. This experiment also gives answer to the gap in *NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions* [7] about the relationship between CPU number and maximum packet rate.

The second experiment shows that the VNF has a similar maximum throughput for the corresponding cases between enabling CPU isolation and disabling CPU isolation. The CPU isolation brings a lower packet loss before the traffic reaches the maximum throughput. This is because the VNF gets exclusive CPUs and hence there is no context switch time between the VNF program and other programs.

The third experiment shows that when the VNF obtains 1 shared vCPU, it has a maximum feasible throughput of 140 kpps. Increasing the CPU limit can improve the maximum feasible throughput to no more than 170 kpps. The third experiment can be compared to the second experiment because the total CPU resource given to the VNF is the same in each column of the two experiments and in both of the experiments, CPU isolation is disabled. Even though the corresponding test cases have the same total CPU resource, they have a different throughput. The maximum throughput in the third experiment is in general smaller than in the second experiment. Therefore, for a given number of total CPU resources, a

composition of fewer Pods and larger CPU limits results in a higher throughput. This experiment, together with the second experiment, answers the gap for *Load Balancing in LTE Core Network with OpenStack Clouds* [52]. The performance of a VNF in container with bigger number of vCPUs is measured in the experiments. The VNF would have a worse performance when it has more backends with the same amount of vCPUs.

The first part of the fourth experiment shows that when the VNF Pods and the MetalLB controller are located in different Nodes, the maximum feasible throughput is 130 kpps. Because *Host 4* and *Host 5* have the same hardware, the difference in throughput is caused by the Pod location. Traffic will always be received by the controller Node and handed over to kube-proxy where the traffic will be directed to Kubernetes internal networks. When the Pod is located in the MetalLB controller Node, traffic is sent to the loopback network interface (localhost) that does not require a physical NIC and hence the traffic would not be limited by the NIC. However, when the Pod is located in a Node other than the MetalLB controller Node, the traffic is sent to a NIC before being received by another NIC. Reading packets from a NIC is slower than reading from localhost and therefore the throughput would be lower in the former situation.

The second part of the fourth experiment shows that the two load-balancing techniques, Service and Ingress, lead to different throughput. The Service case has the highest throughput of 270 kpps when all the Pods are run on the MetalLB controller Node. Service and Ingress have the same throughput of 130 kpps when all the Pods are run on the non-MetalLB controller Node. The case that all Pods are run on different Nodes produces the smallest throughput, at around 70 kpps. When the VNF Pods are run on the non-MetalLB controller Node, the throughput bottleneck is the NIC as in the first part of this experiment. The fourth experiment fills the gap in *Multi-VNF Performance Characterization for Virtualized Network Functions* [8] about how the vertical scaling affects the throughput. Conclusion is that vertical scaling downgrades the performance of the VNF, which aligns with the conclusion in *Load Balancing in LTE Core Network with OpenStack Clouds* [52] that the VNF gets the best throughput when all the VNF backends are run on the same physical virtualization host.

The fifth experiment shows that when the Ingress controller and the MetalLB controller are located in different Nodes, the Service case when all the Pods are run on the MetalLB controller Node still has the best throughput of 270 kpps. The Service and Ingress cases still have similar throughput of around 130 kpps when all the Pods are run on the non-MetalLB controller Node. However, the Ingress cases where all the Pods are run on the MetalLB controller Node produce the

worst throughput at around 30 kpps. The reason is that Ingress gets the external IP address as well as traffic from the MetalLB controller. Traffic targeting Ingress will be sent to the MetalLB controller before being directed to the Ingress controller. Then the traffic will be distributed to Pods by the Ingress controller. Hence, traffic needs to travel to *Host 2* and then to *Host 4*, introducing two extra overheads compared to Service. Therefore, if Ingress is to be used, the Ingress controller should be located in the same Node with the MetalLB controller for a better throughput.

Experiment 6.4 and 6.5 have implications for the importance of networking to a VNF, as indicated in 3.1.2. Networking is about the establishment or use of a computer network [56]. In the two experiments, VNFs have the same CPU resources but the throughput shows significant differences because of networking.

The sixth experiment shows that Kubernetes Pods can affect each other. If all Pods are run with CPU limits specified, the packet loss increases to at most 18%. If the VNF Pods are specified with CPU limit while the local calculation Pods are not, the VNF packet loss increases to 60%. Hence, in order to minimize the impact between Pods, CPU isolation should be enabled and CPU limits should be specified for all the Pods. This experiment fills the gap in *NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions* [7] about how other VNFs influence each other.

The seventh experiment shows that CPU model of the hosts affects the VNF throughput. *Host 2* in this experiment has the same role as *Host 5* in the fourth experiment but *Host 2* produces a better throughput. Although the CPU of *Host 5* has a higher clock frequency than the *Host 2* CPU, it is a Haswell architecture CPU while the *Host 2* CPU type is Skylake, which is two generations newer architecture than Haswell. Newer CPU architectures tend to perform better than the old ones and therefore, VNF Pods running on those newer CPUs have a better throughput. However, when all the controllers and VNF Pods are located on *Host 2*, the VNF performs worse than in experiment 6.1. In this case, the CPU clock frequency becomes the bottleneck for the VNF throughput.

In this study, HPA is only useful when the VNF Pods are run on *Host 4*, the controllers are also run on *Host 4* and the traffic rate changes from lower than 200k pps to over 200 kpps. In this case, the HPA can scale up the VNF to increase maximum throughput. When the packet rate drops below 200 kpps, the HPA can also scale down the Pods to reduce resource consumption.

7. Conclusions

Kubernetes is a popular tool for managing and deploying containerized VNF. It introduces container orchestration mechanisms that could potentially affect VNF throughput. The impact of those mechanisms has remained unknown and it is investigated by conducting experiments in this study.

This study builds a Kubernetes cluster to run benchmarks that implement a packet forwarding service typical for telecommunication industry by following ETSI standards. The benchmarks are used to conduct several experiments to measure VNF throughput in different cases. The experiments are designed by following methodologies from previous studies. The VNF throughput is measured as maximum packet rate in the receiver. The VNF QoS is measured as packet loss and latency. Experiment cases differ from each other by CPU limits and the number of Pods, CPU isolation for Pods, Pod locations, controller locations and CPU types.

The VNF performs better when the VNF Pods are allocated in the same Node and the Node should be the same Node where the load-balancer controllers are run on. For the same amount of CPU resources, the VNF has a better throughput when higher CPU limits are used for the Pods and the number of Pods is small. Regarding load-balancers, Service performs better than Ingress. All Pods should be run with CPU limits specified in order to reduce the CPU contention among Pods. The VNF throughput results for different CPU architectures are not directly comparable but generally newer CPU architecture and higher CPU frequency tend to yield higher throughput.

This study draws conclusions for VNF throughput regarding CPU limits and the number for Pods, CPU isolation for Pods, Pod locations, controller locations and CPU types. There are still some gaps in this study to be filled in the future.

In telecommunication applications, not only packet loss and latency but also jitter is a factor that influences the QoS. The QoS with regard to jitter can be evaluated for a large number of speech streams using the E-model [57] or for a

few speech streams using POLQA [58] because of the computation-heavy nature of POLQA. Generally, jitter means variance in packet delay and it may lead to out-of-order packets that need to be reordered. When jitter increases, speech quality, or QoS, will decrease. In practice, when the data handled by the VNF is Real-time Transport Protocol (RTP) speech data, it has a timestamp that reflects the sampling instant of the first octet in the RTP data packet and usually a packetization time of 20ms that together can be used to calculate jitter [59]. This study uses simplified data streams to maximize the throughput in the traffic generator and hence, this study does not adopt packetization time and does not calculate jitter. Therefore, a study that uses RTP speech data and the E-model can be conducted in the future to evaluate jitter and the corresponding QoS.

This study uses the default CPU manager for CPU isolation. This manager has a basic CPU isolation feature which gives a separate cgroup. Other advanced features, such as CPU pool and CPU pinning, are not provided by the default CPU manager. Therefore, a study for obtaining the best VNF throughput using an advanced CPU manager [60] could be conducted in the future.

This study uses MetalLB to load-balance external traffic and the MetalLB controller is configured to run at Layer 2 mode. The IPVS is configured with default settings. A study for measuring the VNF throughput with another external network traffic load-balancing software and load-balancing settings could be conducted in the future.

The results in this study show that the load-balancers performs undesirably when Pods are distributed to different Nodes. This limits the scalability of any larger-scale VNFs. Future works on improving the load-balancer performance and scalability could be conducted.

This study does not adopt data plane libraries, for example, Data Plane Development Kit (DPDK) [61]. DPDK could provide higher throughput but it requires additional settings for software-defined networks. A study for obtaining the best VNF throughput using data plane libraries could be conducted in the future.

This study observes that different CPU architectures do not present comparable results. Two CPU types are measured in this study but in order to obtain more thorough data for the impact of CPU types, throughput should be measured by using a wider selection of CPU architectures and clock frequencies.

References

[1] Chiosi Margaret et al. *Network Function Virtualization White Paper*. Oct. 24, 2012. URL: https://portal.etsi.org/NFV/NFV_White_Paper.pdf (visited on 06/04/2019).

[2] *About Docker - Management & History*. Docker. URL: https://www.docker.com/company (visited on 03/18/2019).

[3] *Kubernetes commands 9.06% market share in Virtualization Management Software*. URL: https://idatalabs.com/tech/products/kubernetes (visited on 03/26/2019).

[4] Rohan Krishnakumar. "Accelerated DPDK in containers for networking nodes". In: Aalto University (Mar. 2019). M.Sc. Thesis. URL: http://www.urn.fi/URN:NBN:fi:aalto-201903172255.

[5] William Stallings. *Operating systems : internals and design principles*. Eighth edition. Always learning. Pearson, 2015. ISBN: 978-1-292-06194-8.

[6] Víctor Medel et al. "Characterising resource management performance in Kubernetes". In: *Computers & Electrical Engineering* 68 (May 2018), pp. 286–297. ISSN: 00457906. DOI: 10.1016/j.compeleceng.2018.03.041. URL: https://linkinghub.elsevier.com/retrieve/pii/S0045790617315240 (visited on 03/14/2019).

[7] L. Cao et al. "NFV-VITAL: A framework for characterizing the performance of virtual network functions". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN). Nov. 2015, pp. 93–99. DOI: 10.1109/NFV-SDN.2015.7387412.

[8] N. Pitaev et al. "Multi-VNF performance characterization for virtualized network functions". In: *2017 IEEE Conference on Network Softwarization*

(NetSoft). 2017 IEEE Conference on Network Softwarization (NetSoft). July 2017, pp. 1–5. DOI: 10.1109/NETSOFT.2017.8004221.

[9] *Migration from Physical to Virtual Network Functions: Best Practices and Lessons Learned*. Future Networks. Oct. 12, 2018. URL: https://www.gsma.com/futurenetworks/5g/migration-from-physical-to-virtual-network-functions-best-practices-and-lessons-learned/ (visited on 06/06/2019).

[10] Laszlo B Kish. "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3 (Dec. 2002), pp. 144–149. ISSN: 03759601. DOI: 10.1016/S0375-9601(02)01365-8. URL: http://linkinghub.elsevier.com/retrieve/pii/S0375960102013658 (visited on 03/14/2019).

[11] H. Woesner and D. Verbeiren. "SDN and NFV in telecommunication network migration". In: *2015 Fourth European Workshop on Software Defined Networks*. Sept. 2015, pp. 125–126. DOI: 10.1109/EWSDN.2015.80.

[12] Z. Kharkhalis and I. Demydov. "The synthesis methodology of scalable telecommunication service platforms". In: *2017 2nd International Conference on Advanced Information and Communication Technologies (AICT)*. 2017 2nd International Conference on Advanced Information and Communication Technologies (AICT). July 2017, pp. 299–303. DOI: 10.1109/AIACT.2017.8020124.

[13] I. Ouveysi {and} A. Wirth et al. *Large Scale Linear Programs and Heuristics for the Design of Survivable Telecommunication Networks | Kopernio*. URL: https://kopernio.com/viewer?doi=10.1023/B:ANOR.0000004774.11601.b6&token=WzclODE0OCwiMTAuMTAyMy9COkFOT1IuMDAwMDAwNDc3NC4xMTYwMS5iNiJd.5gTHngHRWIT8pFnzQtz6XyFY4RM (visited on 06/10/2019).

[14] Emiliano Casalicchio and Vanessa Perciballi. "Measuring Docker Performance: What a Mess!!!" en. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion - ICPE '17 Companion*. L'Aquila, Italy: ACM Press, 2017, pp. 11–16. ISBN: 978-1-4503-4899-7. DOI: 10.1145/3053600.3053605. URL: http://dl.acm.org/citation.cfm?doid=3053600.3053605 (visited on 06/07/2019).

[15] Zhanibek Kozhirbayev and Richard O. Sinnott. "A performance comparison of container-based technologies for the Cloud". en. In: *Future Generation Computer Systems* 68 (Mar. 2017), pp. 175–182. ISSN: 0167739X. DOI: 10.1016/j.future.2016.08.025. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167739X16303041 (visited on 06/10/2019).

[16] Z. Li et al. "Performance Overhead Comparison between Hypervisor and Container Based Virtualization". In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. Mar. 2017, pp. 955–962. DOI: 10.1109/AINA.2017.79.

[17] Raj Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation and modeling*. Wiley, 1991. ISBN: 978-0-471-50336-1.

[18] Jay Aikat. *The effects of traffic structure on application and network performance*. Springer, 2013. ISBN: 978-1-4614-1848-1.

[19] Safari Books Online. *3.5 Hardware Solutions for Data, Control and Management Planes Processing - System Design for Telecommunication Gateways*. URL: https://learning.oreilly.com/library/view/system-design-for/9780470743003/9780470743003_toc.xhtml (visited on 06/10/2019).

[20] Caio Misticone and Mark Fabbi. "Know When It's Time to Replace Enterprise Network Equipment". In: Gartner (Aug. 15, 2012).

[21] *Cisco Visual Networking Index: Forecast and Trends, 2017–2022 White Paper*. Cisco. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html (visited on 03/15/2019).

[22] H. Nishimura et al. "Applying flexibility in scale-out-based web cloud to future telecommunication session control systems". In: *2012 16th International Conference on Intelligence in Next Generation Networks*. 2012 16th International Conference on Intelligence in Next Generation Networks. Oct. 2012, pp. 1–7. DOI: 10.1109/ICIN.2012.6376026.

[23] *What is a Container?* Docker. URL: https://www.docker.com/resources/what-container (visited on 03/18/2019).

[24] *Docker Documentation*. Docker Documentation. June 8, 2019. URL: https://docs.docker.com/ (visited on 06/10/2019).

[25] *Production-Grade Container Orchestration*. URL: https://kubernetes.io/ (visited on 06/10/2019).

[26] *Overview | Prometheus*. URL: https://prometheus.io/docs/introduction/overview/ (visited on 03/19/2019).

[27] *ETSI - Welcome to the World of Standards!* URL: https://www.etsi.org/ (visited on 09/12/2019).

[28] *Nodes*. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (visited on 06/05/2019).

[29] *Kubernetes Components*. URL: https://kubernetes.io/docs/concepts/overview/components/ (visited on 06/05/2019).

[30] *Understanding Kubernetes Objects*. URL: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/ (visited on 06/05/2019).

[31] *Pod Overview*. URL: https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/ (visited on 06/05/2019).

[32] *Controllers*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/ (visited on 06/05/2019).

[33] Derek DeJonghe. *Load Balancing in the Cloud*. 1st edition. O'Reilly Media, Inc, 2018.

[34] Tero Marttila. *Design and Implementation of the clusterf Load Balancer for Docker Clusters*.

[35] *Cluster Networking*. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (visited on 04/23/2019).

[36] *Contribute to intel/multus-cni development by creating an account on GitHub*. original-date: 2016-12-13T14:40:51Z. Apr. 19, 2019. URL: https://github.com/intel/multus-cni (visited on 04/23/2019).

[37] *flannel is a network fabric for containers, designed for Kubernetes: coreos/flannel*. original-date: 2014-07-10T17:45:29Z. Apr. 23, 2019. URL: https://github.com/coreos/flannel (visited on 04/23/2019).

[38] *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 06/05/2019).

[39] *IPVS Software - Advanced Layer-4 Switching*. URL: http://www.linuxvirtualserver.org/software/ipvs.html (visited on 04/24/2019).

[40] *Production-Grade Container Scheduling and Management: kubernetes/kubernetes*. original-date: 2014-06-06T22:56:04Z. June 5, 2019. URL: https://github.com/kubernetes/kubernetes (visited on 06/05/2019).

[41] *Advanced Scheduling in Kubernetes*. en. URL: https://kubernetes.io/blog/2017/03/advanced-scheduling-in-kubernetes/ (visited on 06/13/2019).

[42] *Assigning Pods to Nodes*. URL: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/ (visited on 06/05/2019).

[43] *Managing Compute Resources for Containers*. URL: https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/ (visited on 06/05/2019).

[44] *Control CPU Management Policies on the Node*. URL: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/ (visited on 05/16/2019).

[45] *Horizontal Pod Autoscaler*. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visited on 06/05/2019).

[46] *Time series database*. In: *Wikipedia*. Page Version ID: 885536705. Feb. 28, 2019. URL: https://en.wikipedia.org/w/index.php?title=Time_series_database&oldid=885536705 (visited on 03/20/2019).

[47] *Time Series Database (TSDB) Explained*. InfluxData. URL: https://www.influxdata.com/time-series-database/ (visited on 03/21/2019).

[48] *Querying basics | Prometheus*. URL: https://prometheus.io/docs/prometheus/latest/querying/basics/ (visited on 03/25/2019).

[49] *RE2 is a fast, safe, thread-friendly alternative to backtracking regular expression engines like those used in PCRE, Perl, and Python. It is a C++ library.: google/re2*. original-date: 2014-08-18T21:21:26Z. Mar. 25, 2019. URL: https://github.com/google/re2 (visited on 03/25/2019).

[50] ETSI, 3GPP, and GSM. *ETSI TS 129 060 V12.6.0*. June 4, 2019. URL: https://www.etsi.org/deliver/etsi_ts/129000_129099/129060/12.06.00_60/ts_129060v120600p.pdf.

[51] R. K. Ghosh. *Wireless Networking and Mobile Data Management*. Springer Singapore, 2017. ISBN: 978-981-10-3941-6.

[52] Kimmo Ahokas. “Load balancing in LTE core network with OpenStack clouds: Design and implementation”. In: (2015). URL: http://urn.fi/URN:NBN:fi:aalto-201512165613 (visited on 08/19/2019).

[53] *IPVS-Based In-Cluster Load Balancing Deep Dive*. URL: https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/ (visited on 03/06/2019).

[54] *metallb*. June 4, 2019. URL: https://raw.githubusercontent.com/google/metallb/v0.7.3/manifests/metallb.yaml (visited on 05/20/2019).

[55] *The SO_REUSEPORT socket option [LWN.net]*. URL: https://lwn.net/Articles/542629/ (visited on 05/16/2019).

[56] *Definition of NETWORKING*. URL: https://www.merriam-webster.com/dictionary/networking (visited on 09/12/2019).

[57] *G.107 : The E-model: a computational model for use in transmission planning*. URL: https://www.itu.int/rec/T-REC-G.107-201506-I/en (visited on 06/25/2019).

[58] *POLQA - The Next-Generation Mobile Voice Quality Testing Standard*. URL: http://www.polqa.info/ (visited on 06/25/2019).

[59] *RFC3550*. URL: https://www.ietf.org/rfc/rfc3550.txt (visited on 06/25/2019).

[60] *Kubernetes Core Manager for NFV workloads. Contribute to intel/CPU-Manager-for-Kubernetes development by creating an account on GitHub*. original-date: 2016-12-20T17:59:39Z. Apr. 26, 2019. URL: https://github.com/intel/CPU-Manager-for-Kubernetes (visited on 05/08/2019).

[61] *Home*. DPDK. URL: https://www.dpdk.org/ (visited on 06/20/2019).

A. Appendices

A.1 Node Setup

```
$apt-get update
$apt-get install -y \
     apt-transport-https \
     ca-certificates \
     curl \
     gnupg2 \
     software-properties-common
$curl -fsSL "https://download.docker.com/linux/\
    $(. /etc/os-release; echo "$ID")/gpg"\
    | apt-key add -
$apt-key fingerprint 0EBFCD88
$add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/\
   $(. /etc/os-release; echo "$ID") \
   $(lsb_release -cs) \
   stable"
$apt-get update
$apt-get install -y docker-ce apt-transport-https curl
$curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg\
    | apt-key add -
$cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
$apt-get update
$apt-get install -y kubelet kubeadm kubectl
```

```
$apt–mark hold kubelet kubeadm kubectl
$systemctl start kubelet
$systemctl enable kubelet
```

A.2 VNF Deployment

```
apiVersion: v1
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vnf–benchmark
  labels:
    app: vnf
spec:
  replicas: 2
  selector:
    matchLabels:
      app: vnf
  template:
    metadata:
      labels:
        app: vnf
      annotations:
        k8s.v1.cni.cncf.io/networks: macvlan–conf
    spec:
      containers:
      – name: vnf
        image: python:3
        command:
        – /pyrun/server
        args:
        – ""
        ports:
        – containerPort: 2152
          name: udps
          protocol: UDP
        volumeMounts:
```

```
          - mountPath: /pyrun
            name: pyexec
          resources:
            limits:
              memory: "1024Mi"
              cpu: "1"
            requests:
              memory: "1024Mi"
              cpu: "1"
        nodeName: labmpd2
        volumes:
        - name: pyexec
          hostPath:
            path: /home/ubuntu/share
            type: Directory
---
kind: Service
apiVersion: v1
metadata:
  name: vnf-benchmark-svc
  labels:
    app: vnf
spec:
  selector:
    app: vnf
  ports:
  - protocol: UDP
    port: 2152
    targetPort: 2152
    name: udp
  type: LoadBalancer
```

A.3 Cluster Setup

```
apiVersion: kubeadm.k8s.io/v1beta1
kind: MasterConfiguration
kubernetesVersion: v1.13.0
```

```
bootstrapTokens:
- ttl: "0s"
---
apiVersion: kubeadm.k8s.io/v1beta1
kind: ClusterConfiguration
kubernetesVersion: v1.13.0
networking:
  podSubnet: "10.244.0.0/16"
---
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
mode: "ipvs"
ipvs:
  scheduler: "lc"
```

A.4 Multus Setup

```
cat <<EOF | kubectl create -f -
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: macvlan-conf
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "traffic0",
      "mode": "bridge",
      "ipam": {
        "type": "host-local",
        "subnet": "10.10.9.0/24",
        "rangeStart": "10.10.9.50",
        "rangeEnd": "10.10.9.149",
        "routes": [
          { "dst": "0.0.0.0/0" }
        ]
      }
```

```
    }'
EOF
```

A.5 MetalLB Setup

```
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: my-ip-space
      protocol: layer2
      addresses:
      - 10.10.9.150-10.10.9.253
```

A.6 Ingress Setup

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: tcp-services
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
data:
  9011: "default/vnf-benchmark-svc:2152"
---
kind: ConfigMap
apiVersion: v1
metadata:
  name: udp-services
  namespace: ingress-nginx
  labels:
```

```
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
data:
  9011: "default/vnf-benchmark-svc:2152"
```

A.7 Ingress for VNF

```
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: UDP
    - name: https
      port: 443
      targetPort: 443
      protocol: UDP
    - name: ws-udp-2152
      port: 2152
      targetPort: 2152
      protocol: UDP
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
```

A.8 Ingress Service

```
apiVersion: extensions/v1beta1
```

```
kind: Ingress
metadata:
  name: vnf-ing
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - http:
      paths:
      - path: /*
        backend:
          serviceName: vnf-benchmark-svc
          servicePort: 2152
```