Design and Implementation of a Reliable Container-based Service Function Chaining Testbed in Cloud-native System: An Open Source Approach

by

Ziqiang Wang

A thesis submitted to Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

Academia has been moving attention from Virtual Network Function (VNF) to Cloud-native Network Function (CNF) since cloudification has brought the Network Function Virtualization (NFV) to an advanced level. It has already been demonstrated that cloud-native technology brings high flexibility and efficiency to large-scale network service deployment compared to the traditional VNF with Virtual Machines (VMs). However, more work is needed to provide a flexible and reliable Service Function Chaining (SFC) development solution. This thesis proposes a solution to dynamically deploy reliable SFCs consisting of multiple CNFs in a multi-node Kubernetes cluster using Network Service Mesh (NSM). Further, hardware and network usage are presented with the support of an open-source monitoring system, Prometheus, to help the network operator understand the deployed SFC. Additionally, the performance and limitations of this approach are analyzed and presented.

Acknowledgements

The work presented in this thesis would not have been realized without the assistance of many individuals. I would like to take this opportunity to express my greatest gratitude to everyone who helped me during this journey.

Most importantly, I would like to thank my supervisors Dr. Changcheng Huang and Dr.Chung-Horng Lung for their continuous guidance, motivation, understanding, and feedback during the two years of my study. Secondly, I would like to thank my parents for providing the precious opportunity so I can study as an international student at Carleton University. I cannot finish the degree without their encouragement and support.

Many thanks to Abdullah Bittar for discussing various technical challenges concerned with this testbed. Furthermore, I am grateful to Mitacs and Ciena Corporation for providing financial and technical support for this research. A special thank goes to the NSM Slack community for sharing their professional knowledge regarding the development of NSM.

List of Abbreviations

ACL Access Control List

API Application Programming Interface

CENI Ciena Environment for Network Innovation

CIDR Classless Inter-domain Routing

CLI Command Line Interface

CNCF Cloud Native Computing Foundation

CNF Cloud-native Network Function

CNI Container Network Interface

CPU Central Processing Unit

CRI Container Runtime Interface

CSV Comma-Separated Values

DL Deep Learning

DNS Domain Name System

DPDK Data Plane Development Kit

DPI Deep Packet Inspection

DTN Data Transmission Node

GB Gigabyte

GPRS General Packet Radio Service

gRPC Google Remote Procedure Call

HLS HTTP Live Streaming

IDS Intrusion Detection System

IMS IP Multimedia Sub-system

INT In-band Network Telemetry

IPSec Internet Protocol Security

IPv4/IPv6 Internet Protocol version 4/ Internet Protocol version 6

MAC Media Access Control

MANO MANO Management and Orchestrator

MB Megabytes

MEMIF Shared Memory Packet Interface

MiB Mebibyte

ML Machine LearningMPLS

MPLS Multiprotocol Label Switching

NaaS Network as a Service

NAT Network Address Translation

NFV Network Function Virtualization

NOS Network Operating System

NS Network Service

NSC Network Service Client

NSE Network Service Endpoint

NSH Network Service Header

NSM Network Service Mesh

NSMgr NSM Manager

OS Operating System

OVN Open Virtual Network

OvS Open vSwitch

OVSDB Open Virtual Switch Database

QoE Quality of Experience

QoS Quality of Service

RBAC Role-Based Access Control

RFC Request for Comments

RTMP Real-Time Messaging Protocol

RTT Round-Trip Time

SAL Service Agreement Level

SDK Software Development Kit

SDN Software-defined Network

SFC Service Function Chaining

SFF Service Function Forwarder

SPIFFE Secure Production Identity Framework for Everyone

SR-IOV Single Root I/O Virtualization

SSL Secure Sockets Layer

SVID Verifiable Identity Documents

TCP Transmission Control Protocol

TSDB Time-Series Database

UDP User Datagram Protocol

USE Utilization, Saturation, Errors

VAP Virtual Access Point

VFIO Virtual Function I/O

VLAN Virtual Local Area Network

VM Virtual Machines

VNF Virtual Network Function

VPP Vector Packet Processing

V-wire Virtual Wire

VxLAN Virtual eXtensible Local Area Network

WLAN Wireless Local Area Network

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recently, cloud computing has become a popular and attractive topic in the world of telecommunications and networking. The term cloud-native always refers to the concept of creating and running applications to take advantage of the distributed computing resources offered by the cloud delivery model. Cloud native applications are designed and built to exploit the scale, elasticity, resiliency, and flexibility the cloud provides. The Cloud Native Computing Foundation (CNCF) provides the official definition of Cloud-native technology as *"Cloud-native technology empowers organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."* [11]. Even though the mainstream industry is still heavily relying on the Virtual Machine-based (VM-based) Virtual Network Functions (VNFs), the academia has already moved its attention from VNFs to Cloud-native Network Functions (CNFs) since cloudification has brought the Network Function Virtualization (NFV) to a higher level of scalability and flexibility. In short, VNF commonly refers to the software form of network

appliances such as a router, firewall, load balancer, etc. based on the concept of NFV to offer end-to-end functionality provided by a network operator. CNF is an enabler of the VNF that is deployed in a cloud environment. It already has been demonstrated that NFV brings flexibility and efficiency to the field of large scaling networking service deployment.

The technology evolution in the cloud-native world also brings innovation to network applications. Traditionally, deployment of a network service function relies on dedicated networking hardware, while the concept of NFV decouples the hardware and software of network devices by moving applications to VMs in a hypervisor environment. In recent years, with the development of cloudification, network functions can be implemented in the form of containers in a cloud environment as microservices. Microservices are a collection of services that are independently deployed and loosely coupled to provide end-to-end services [12]. Compared with VM-based VNF, container-based virtualization technologies are gaining more and more attraction with the development of cloud computing and 5G networks. According to the International Data Group (IDG) 2020 Cloud Computing Study, 81% of organizations have moved at least one application/workload or part of the computing infrastructure to cloud. In addition, 46% of the current cloud-based applications are purpose-built for the cloud, but 54% of cloud applications have migrated from an on-premises environment [13].

Network applications are mostly developed based on NFV recently. Typically, network applications run on the VMs which are managed by a VM orchestrator and the Software-defined Network (SDN) controller [14]. The orchestrator plays the role of the automatic hardware resource manager. The SDN controller, on the other hand, plays the role of the central brain on the control plane to manage network devices such as switches and routers. However, in a cloud-native environment, microservices

can be deployed in the form of containers running on a cloud [1]. The container uses Linux's Namespace and Cgroup concepts, which oversee process isolation and resource constraints, respectively [15]. The use of container-based VNFs provides benefits to NFV based on the following facts [14]:

1. Container-based Microservices have less overhead than that of VM-based VNFs. VMs run in a hypervisor environment where each VM must have its operating system (OS), along with its related binary code, libraries, and application files which impose high overhead. Unlike VMs, containers are much more lightweight and much faster to deploy. Usually, multiple containers can run on the same physical or virtual host since multiple containers can share the same host OS which has everything needed to run the applications. In comparison, VNFs hosted on VMs may have the challenge to handle increasing traffic created by duplicated common network functions such as packet classification, packet header parsing, etc.

2. Microservices are adaptable and portable. The deployment of a container is independent of OS and hardware platforms. A large application can be run as a cluster of microservices that are managed by one container orchestration system such as Kubernetes [16]. Moreover, the container orchestrator can provide additional features by adopting various plug-ins.

Microservices, furthermore, can be configured to form a Service Function Chaining (SFC) which connects a list of networking applications in a specific order to offer network services [17]. The cloud-native SFC formed by microservices can leverage the agility, flexibility, and scalability features of containers. Most importantly, the cloud-native SFC does not even rely on an SDN controller to manage the application life cycle and does not rely on the underlying physical topology [4]. Leveraging the benefits provided by cloud-native technologies, SFC can automate traffic flow

between cloud-native services and optimize the use of network resources. Without such mechanisms, interconnections of these virtual network functions are complex, time-consuming, and expensive tasks. Hence, cloud-native SFC plays a vital role in the next-generation networks by utilizing different technologies such as 5G, IoT, and edge computing. SFC helps in providing customizable network functions and steering traffic flows between different networks. The demand for SFCs has grown exponentially, aided by the explosion of novel network technologies and infrastructures, such as the success of cloud networks, that have increased the degree of pervasiveness and connectivity between heterogeneous devices.

Despite the two benefits listed above, network measurement is fundamental and essential for any network service, such as SFC deployment. Applying network measurement and a centralized network analyzer can provide reliable and optimal network service performance, especially in the high-speed cloud environment. However, it is challenging for the current management and monitoring system considering the complexity of the cloud-native infrastructure. Many factors need to be taken into consideration, such as power efficiency, cooling, network distribution, CPU and memory utilization, load balancing, fault recovering, Quality of Service (QoS), alert system, etc. The revolution of the cloud-native ecosystem requests the monitoring and management tools to have not only a basic understanding of the hardware resources, but also the ability to proactively detect potential issues to ensure the robustness, reliability, and scalability of SFC deployment.

The theme of the thesis is to establish a reliable and resource-aware cloud-native SFC testbed that combines underlying components to secure the fast service recovery of the SFC traffic steering between different service nodes. Further, the testbed is built with a web front-end interface that enables the network operator to shorten

the SFC deployment time by simply going through the web and filling out the SFC configuration without having to design different software components and connect them from scratch.

1.1 Motivation for the Thesis

The container-based SFC brings agility, flexibility, scalability, and efficiency to the existing NFV architecture. However, compared to the traditional VM-based SFC approach, developing a container-based SFC in a cloud-native environment is still a novel approach in both industry and academia considering the configuration and management challenges brought by the new cloud-native eco-system. On top of that, ensuring SFC's reliability and robustness is another open issue in this field. Not many research works leverage the cloud-native NFV orchestrator (NFVO) with a resource monitoring system for the deployment of a reliable and resource-aware cloud-native SFC. Ensuring the reliability, robustness, and scalability of network service is not only significant but also challenging considering the exponential growth of current end-user devices. With the large demand for services, unreliable and unstable SFCs become the bottleneck in a large-scaled cluster which may suffer from service interruption, loss of data, vulnerability, and other issues [18].

Furthermore, developing an SFC in any platform is a time-consuming task that needs dramatic efforts in downloading software packages and configuring the environment, such as developing SFC in the industrial OpenStack platform. OpenStack is highly challenging to deploy and has high operational complexity [19]. During the configuration process, network operators often face compatible issues, especially when steering traffic based on personalized switching functionalities such as using

Programming Protocol-independent Packet Processors (P4) switch [20]. On top of that, many other issues may concern the network operators such as validating the SFC traffic, prompt service recovery from failure, service security issues, and so on. By conducting the research and experiment, an SFC testbed is proposed and designed that allows the network operator to rapidly deploy reliable and robust container-based SFCs on the Kubernetes platform dynamically by conducting fine-grained network and resources monitoring. The proposed testbed also provides features such as automated service recovery, transparent SFC path, and workload authentication to enhance the robustness of the SFC deployment.

1.2 Research Objectives

To solve the difficulties of creating a robust SFC in a cloud-native environment, this thesis addresses the challenges of (i) designing and developing a cloud-native container-based SFC testbed that composites various open-source modules to provide reliable and robust network services in a Kubernetes environment, (ii) integrating a monitoring system to dynamically validate the SFC traffic path and monitoring hardware and network resources, and (iii) ensuring the reliability and robustness of the developed SFC by recovering an SFC from a failed VNF. The following are the high-level research objectives required for developing the proposed testbed:

- A proof-of-concept that the SFC can be deployed in a cloud-native platform such as Kubernetes among different physical/virtual nodes using open-source technologies.
- Enhancing the usability of the testbed by implementing a web-based interface that allows network operators to easily plan and efficiently deploy the SFC and dynamically monitor SFC traffic and resources.

• Improving the robustness and reliability of the SFC by using a proactive monitoring solution Prometheus [20] in the cloud-native environment with features provided by the Network Service Mesh (NSM) network plug-in [21], such as automatic network service recovery, supporting different types of container interfaces, and SFC workload authentication.

• Evaluating the performance of the container-based SFC deployed in the proposed testbed from the perspective of deployment time, inter-service traffic delay, resource utilization, SFC traffic bandwidth, and inter-service outage time. Comparing the performance with the existing solutions such as VM-based SFC in the OpenStack environment.

1.3 Proposed Solution

This research focuses on developing a robust and reliable cloud-native SFC testbed that steers the traffic by applying networking policy and evaluating the performance based on various criteria. Kubernetes will be used as containerized VNF orchestrator that provides automatic resource management for deploying container-based microservices. The SFC framework will be achieved by using the NSM plug-in which offers a "virtual wire" (V-wire) between network services. The V-wire is essentially the communication tunnel between two containers [22]. The V-wires between services can be quickly recovered from temporary service outages due to the NSM build-in healing features, hence the SFC testbed provides the ability to resist unpredicted change without adapting its initial stable configuration. Network and resource monitoring will be provided by adopting the state-of-art Prometheus monitoring system. The Prometheus system consists of a metrics collector that collects container-level data regarding

hardware and network states and saves those data as ordered metrics into a database [20]. Meanwhile, the network operator can make decisions based on the network states and hardware resource usages to ensure the functionality of the prioritized SFC deployment. On the other hand, the network policy could be enforced to steer traffic between different routing paths, apply alternative routing rules or allocate additional computing resources to achieve better network performance and QoS. In addition, an authentication mechanism provided by the SPIRE project implementing the Secure Production Identity Framework for Everyone (SPIFFE) [23] will enhance the reliability of the deployed SFC by assigning workload tokens to every working service node. Despite the robustness and reliability enhancement, a user web interface will be created to reduce the complexity of deploying the SFC in the proposed testbed.

1.4 Contributions of the Thesis

The major contributions of the thesis are listed below:

• *Prototype*: to the best of my knowledge, this thesis is the first research work that develops a reliable and robust cloud-native container-based SFC testbed with a state-of-the-art open-source SFC framework and monitoring systems. The testbed provides insight into the SFC performance along with valuable environment parameters and ensures the robustness and reliability of the deployed SFCs across different physical compute nodes. By adopting this testbed, developers and network operators can conduct suitable tasks by rapidly deploying customized SFC instead of building from scratch.

• *Performance Insight*: the performance of the container-based SFCs deployed using the proposed testbed is analyzed in terms of hardware resource utilization,

network bandwidth consumption, network service start-up time, and network latency under different use-case scenarios. The performance of the proposed testbed is analyzed, and compared with other existing testbed solutions such as OpenStack with ONOS. The result not only shows that the container-based SFC deployment is much faster and has less overhead than the VM-based SFC, but also demonstrates some disadvantages of using the NSM-based SFC such as high latency and low bandwidth between container-based services.

- *Reliability Insight*: the proposed testbed also demonstrates that the deployed SFC will remain reliable and robust when facing unexpected service outages by automatically recovering from the missing service in a short period. The adopted Prometheus monitoring system provides fine-grained container-level metrics collection that helps the network operator understand the performance of the deployed SFC in detail. Meanwhile, the workload authentication mechanism also ensures the hardware resources will not be occupied by unknown network services.

1.5 List of Publications

The publications based on the thesis are listed below:

Z. Wang, A. Bittar, C. Huang, C.-H. Lung and G. Shami, "A Web-based Orchestrator for Dynamic Service Function Development with Kubernetes," in 2022 IEEE 8th International Conference on Network Softwarization (NetSoft), Milan, Italy, 2022.

A. Bittar, Z. Wang, A. Aghasharif, C. Huang, G. Shami, M. Lyonnais, and R. Wilson, "Service function chaining design & implementation using network service mesh in kubernetes," in Supercomputing Frontiers, D. K. Panda and M. Sullivan, Eds. Cham: Springer International Publishing, 2022, pp. 121–140.

1.6 Thesis Outline

After the introduction section, the rest of the thesis can be organized as follows:

- *Chapter 2*: provides the background information related to the technologies mentioned in this thesis which includes SDN, NFV, SFC, Kubernetes, NSM, SPIRE project, container runtime/engine, and Prometheus monitoring system. Moreover, it also includes the previous related works directly related to the field of SFC testbed.
- *Chapter 3*: describes the high-level design of the proposed cloud-native testbed for container-based SFC. It explains how different elements correlate together to support dynamic traffic steering, SFC path validation, SFC recovery, and network and resource analytics.
- *Chapter 4*: denotes the implementation detail of the testbed including the test environment, customized service function docker images, and the SFC reliability.
- *Chapter 5*: provides the performance metrics of various SFC use-case implementations using the proposed testbed. Conduct the different measurements to prove the benefits of using this testbed from both qualitative and quantitative perspectives. Furthermore, the limitations of the proposed framework are discussed.
- *Chapter 6*: conclude the thesis with possible future development.

Chapter 2

Background and Literature Review

NFV, with the development of SDN during the last decades, becomes the trending abstraction to overcome various challenges in many fields [24]. NFV describes the idea of virtualization in terms of network services at both the virtualization layer and the application layer of SDN. The main idea of NFV is to replace proprietary network devices with industry-standard high-capacity servers, storage, and switches in various locations such as data centers and content distribution networks. The NFV architecture inherently promises advantages such as software and hardware decoupling, flexible network function deployment, and dynamic functioning. CNF is a software implementation of NFV deployed in a cloud-native method. Despite all the benefits gained from integrating containers into the NFV environment, there will be management and orchestration challenges that may affect the utilization of container-based VNFs due to introducing an entirely new infrastructure ecosystem.

This chapter goes over the concept of software-defined networking (SDN) which is the base of the thesis. All the abstractions in this thesis are based on the development of SDN. Then the thesis introduces the concept of Network Function Virtualization

(NFV) which describes the idea of virtualization in terms of network services at both the virtualization layer and application layer of SDN. After that, container technology is introduced since the service functions mentioned in this thesis are deployed in the form of container-based microservices running on the cloud platform. Furthermore, the thesis focuses on describing an overview of the most popular open-source platform for managing containerized workloads and services. It also includes the section that introduces the SFC framework provided by NSM, an open-source plug-in that provides a separate data plane and control plane. Then, the chapter concentrates on the Prometheus system, a state-of-art framework for resource monitoring and network data measurement. Finally, the chapter demonstrates the research gaps by conducting the literature review which includes some of the state-of-art related works presented by other authors in the field of cloud-native SFC testbed.

2.1 Software-defined Networking

The project delivered with this thesis is relying on Software-defined Networking. Everything mentioned in this thesis is built on top of this concept. It is important to understand the SDN.

2.1.1 The Concept of SDN

The Internet has spawned a digital civilization in which nearly everything is connected and available from any location. Traditional IP networks, however, are complicated to manage, despite their broad acceptance. Configuring the network to follow specified policies and reconfiguring it to respond to faults and changes are both demanding tasks. To make problems worse, today's networks are vertically integrated, with the

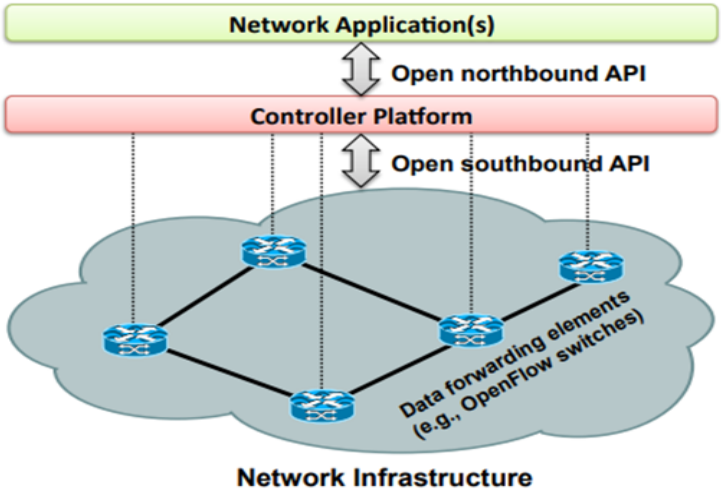


Figure 2.1: Simple version of SDN architecture [1]

control and data planes grouped which make the network tight up and lose flexibility.

By breaking vertical integration, detaching the network's control plane from the underlying physical routers and switches, promoting a logic centralized network control panel, and offering the ability to program the network, Software-Defined Networking (SDN) is a novel approach that promises to improve this situation. SDN separates the network abstraction into network policies, implementation in the hardware layer, and the process of routing/forwarding traffic [1]. On the other hand, SDN also simplifies the development of new network abstraction by breaking network control issues into tractable parts. A simple version of the SDN architecture is shown in Fig. 2.1 above [1].

As shown in Fig. 2.1, a well-defined open southbound application programming interface (API) has been involved between the switches and the SDN controller allowing the separation of the control plane and the data plane. OpenFlow is the most remarkable example of such an API [1]. Typically, an OpenFlow-based switch will have one or multiple flow tables where match fields can be found. Decisions are made

by matching rules in the flow table. As an example of an OpenFlow-based switch, Open-vSwitch can, as the controller instructs, act as the router, switch, firewall, or fulfill other tasks according to the controller application's requirements (e.g., load balancer, traffic shaper, and traffic forwarder) [16].

In general, the whole process of packet processing in a software-defined network can be separated into three parts: (i) the creation of network policy, (ii) its execution in the use of network hardware, and (iii) traffic transmission. This separation satisfies the flexibility required, breaks down the challenge of network control into tractable parts and supports the creation and establishment of new networking abstractions, simplifies network administration, and promotes network development and innovation. In the past decade, SDN has drawn considerable interest in the industry. Most commercial switch makers now supply their equipment using the OpenFlow API that promotes the open standardization of SDN and OpenFlow [1].

2.1.2 SDN Architecture

The overall SDN architecture is quite straightforward. The Data plane, control plane, and management plane are the three segments that formed the plane-view SDN architecture as the Fig. 2.2 (a) shown. The data plane, which consists of switches, routers, and bridges, is where the business traffic flows. The control plane, on the other hand, declares the protocol used by the data plane. The management plane corresponds to the software tool monitoring and configuring the network. A typical user scenario would be the network policy is defined in the management plane; the control plane then enforces the implementation in the data plane.

Traditional IP networks have tight links between the control and data planes resulting in an extremely centralized network environment where software is only

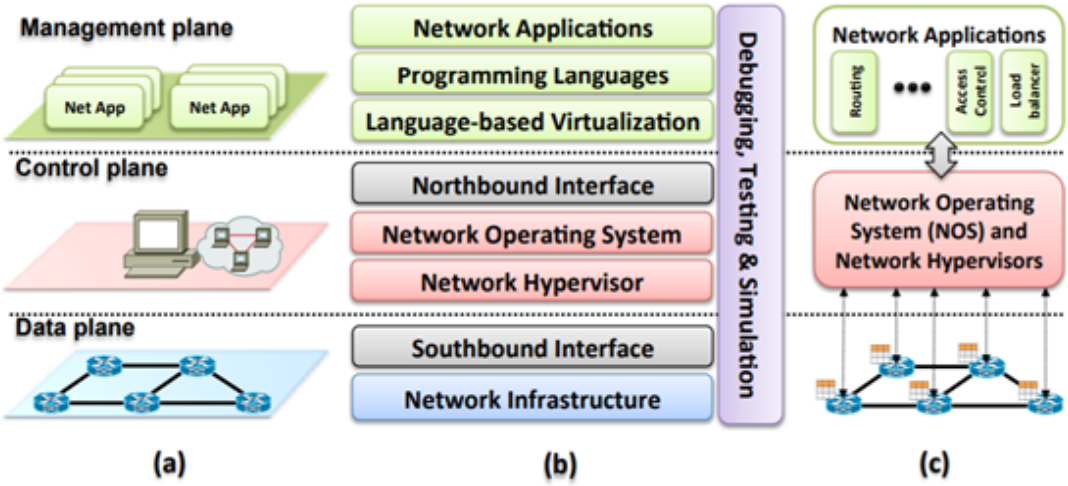


Figure 2.2: The plane, layer, and system view of SDN [1]

supported by the same hardware provider. Frankly, it was the best approach to ensure stability and network performance in the early days. As provided in Fig. 2.2 (b), SDN, in layer perspectives, can be separated into 8 layers. There are special functions for each tier. Some of the layers are always available in an SDN deployment, while others are optional such as hypervisor virtualization or language-based virtualization [1]. The details of each layer are explained as follows:

1. Network infrastructure

From the infrastructure perspective, the SDN can be managed as a composition of different layers. The bottom layer of the SDN is the network infrastructure layer which consists of network devices supporting the OpenFlow protocol. The up-to-date version of OpenFlow had specified match fields such as IPv4/v6, Multiprotocol Label Switching (MPLS), Network Service Header (NSH), Transmission Control Protocol/ User Datagram Protocol (TCP/UPD), Ethernet, etc. [1].

2. Southbound API

The layer above the bottom layer is the southbound API which establishes the

connection channel between the control plane and data plane. On the other hand, the next layer, Network Operating System (NOS) can collect flow-level information through southbound API. One instance of the southbound APIs is the Open vSwitch Database (OVSDB) which provides abundant features including creating, configuring, and managing Open vSwitch (OvS).

3. Network hypervisor

The network hypervisor layer is set on top of the southbound API layer. This layer provides virtualization for network topology and address space. The hypervisor allows hardware resource sharing including CPU cycle, memory usage, and storage. It also provides on-demand resource allocation which the resource can be dynamically allocated based on application demand. It ensures SDN provides elastic services with modularity such as providing L2/ L3 service alone or service with L4 and above [16].

4. Network operating system

The next layer is the NOS layer which provides standardized APIs and a user-friendly operating system for controlling and managing low-level devices at the network infrastructure layer. Generally, NOS could provide network topology information, device discovery, and distribution of network configurations as services that aim to improve productivity, and application life-cycle management. ONOS [25], Ryu [26], and ODL [27] are the most popular SDN controllers in the market.

5. Northbound interface

Similar to the southbound interface, on the top of NOS, the northbound interface is the application-level interface. However, unlike the southbound interface that connects hardware, the northbound interface is a software ecosystem that would allow network applications not to depend on a specific implementation of the forefront driver.

6. Language-based virtualization

Layer Six is the Language-based virtualization. By taking advantage of the network hypervisor layer, it provides modularity and permits various abstract levels while yet ensuring the necessary characteristics.

7. Programming languages

On top of the Language-based virtualization layer is the programming languages layer. With the development of high-level language, the network administrator can easily develop a higher-level network topology, create a problem-oriented environment, and improve software reusability.

8. Network application

The top layer of the SDN structure is the network application layer. It is the controller and user interface of the network to ensure fine communication between tasks and avoid functionality overrides. There is a wide range of network applications because SDNs may be implemented in any traditional network environment from residential and business networks to data centers. Network applications could handle various conventional network functions for example forwarding, routing, load-balancing, packet inspection, implementation of security policies, computing resource allocation, as well as exploring new methods to accomplish energy reduction. Most SDN network applications can be categorized into one of the five types according to their characteristics [1]. The five types are:

(i) traffic engineering: the main goal of this type of network application is traffic optimization.

(ii) mobility & wireless: SDN approach is an opportunity to simplify the deployment and management of many types of wireless networks, such as WLAN and mobile networking.

(iii) measurement and monitoring: there are two classes of application in this

category. The first one provides more functionalities and features for the existing network such as increasing visibility at the data plane. There are plenty of applications in the market that serve this purpose like Prometheus [20], Skydive [28], and Jaeger [29]. On the other hand, the second class tries to reduce the overhead bought by network measurement data collection itself. Typically, this involves developing a new algorithm for data sampling, updating period, and data analysis such as In-band Telemetry (INT).

(iv) security and dependability: security has been dramatically improved as security services such as access control, package inspection, firewall, route tracing, and intrusion detection can be easily implemented in the SDN network.

(v) data center networking: the highly scalable and efficient data center has been designed in such a way that it can provide services, data transactions, and storage while maintaining QoS, failure tolerance, central management, and stability.

2.2 Network Function Virtualization

Network function virtualization (NFV), with the development of SDN during the last decades, becomes the trending abstraction to overcome various challenges in many fields [24]. The use of NFV is part of this thesis, it is critical to understand its characteristics.

2.2.1 The Concept of NFV

Network function virtualization (NFV) is an abstraction that decouples the networking software from the underlay dedicated hardware in conjunction with the use of SDN. Fig. 2.3 shows the illustration of the NFV framework at a high-level [2].

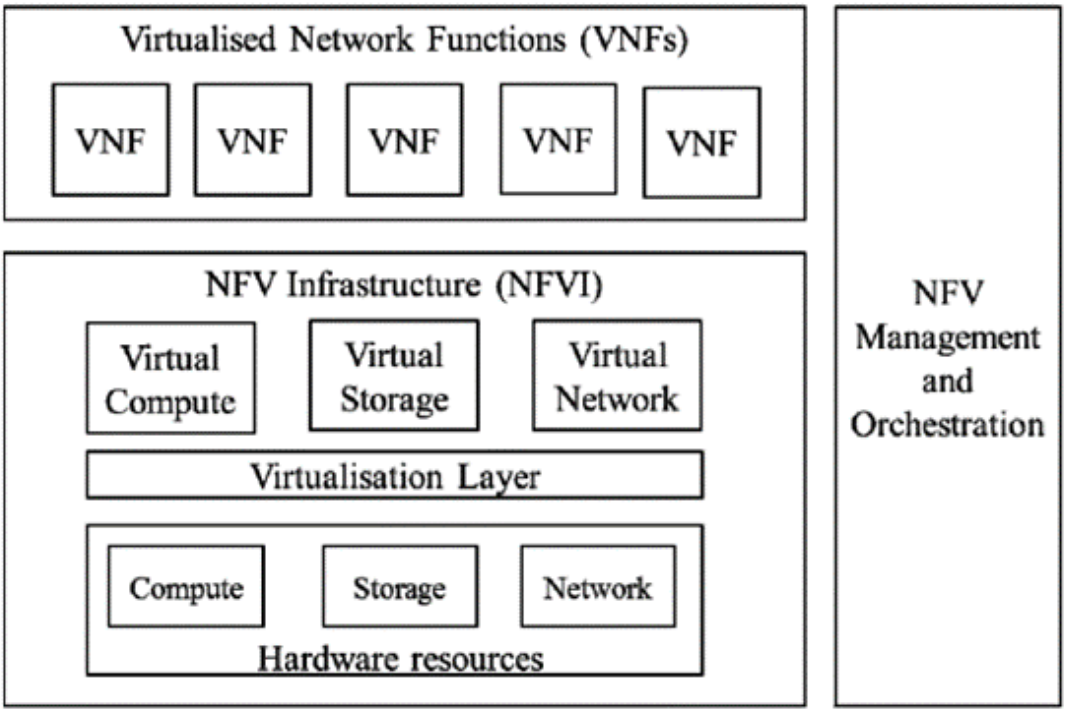


Figure 2.3: Overview of high-level NFV infrastructure [2]

NFV allows network applications to run on commercial off-the-shelf equipment based on virtualization technology brought by SDN. It can run on standard IT platforms like a high-performance switch, general-purpose computer, VM, or container. In addition to the decoupling between software and hardware, NFV offers dynamic resource allocation which addresses the problem of limited computing and storage resources. On top of that, it could efficiently run multiple applications on the shared physical infrastructure to reduce the operational cost, while supporting a centralized control system to assign network applications on-demand and manage the entire software life cycle. Despite the remarkable features NFV provided, the focus is always on the development of various virtual network applications to overcome challenges and accomplish certain functionalities.

On the other hand, even though the NFV is a promising model, some challenges and problems need to be overcome for performing NFV itself. One of the challenges is the performance of virtualized network functions when compared to network functions running on hardware middle-boxes. Typically, a physical platform such as a server may run more than one virtual network function. It is crucial to organize the hardware resources such as CPU cycle, Memory layout, and input/output (I/O) bus to have optimal performance including high-speed package throughput, minimal latency, etc. Furthermore, the multi-tenancy environment also required isolation among network applications not only from the security perspective but also from the operational perspective since network application always carries different binaries, software package, and code dependencies. There are some dedicated chips that have been developed to enhance resource usage for multi-tenancy networks. Other important related works have been conducted to satisfy the fast packet processing requirement such as Vector Packet Processing (VPP) and Data Plane Development Kit (DPDK). The VPP platform is the open-source version of Cisco's VPP technology that uses vector processing to support a high-performance packet processing stack [30]. The VPP could also be integrated with the best-of-breed open-source driver technology DPDK to bypass the Linux Kernel which usually can only handle one packet at a time. Hence the fast I/O such as Network Interface Card (NIC) is virtually connected with user-space applications to enhance the network performance.

2.2.2 NFV Implementation: SFC Perspective

A typical use-case of software-defined NFV is Service Function Chaining (SFC). SFC utilizes the features of NFV and SDN by allowing users to connect multiple virtual network functions in specific orders to process network traffic. Combined which other

features of SDN, SFC also offers real-time and dynamic provisioning along with flexible traffic steering. In conventional networks, a service function chain contains several hardware-specialized network appliances giving services such as load balancers, firewall, Deep Packet Inspection (DPI), Intrusion Detection System (IDS), etc. Consequently, any change that happens to SFC involves adding or deleting hardware devices. This will not only disturb every other component on the chain, but also requires additional effort such as new installation and configuration which is time-consuming and complex. On the other hand, software-defined SFC not only improves flexibility, but also could benefit from network upgrades.

There are various class tiers of NFV since each of them has a specific focus to satisfy different requirements [24]. In general, NFV applications can be dedicated to the following purposes:

• Network switching elements, i.e., virtual switch and router which processes, transmits, and transforms network traffic.

• Mobile network functions, i.e., General Packet Radio Service (GPRS) which is a packet-oriented mobile communication system provided for 2G and 3G cellular communication networks.

• Virtualized home environments, i.e., a future home network environment in which available hardware resources are efficiently shared and balanced among end-users, such as CPU cycles, memory, and network bandwidth [31].

• Tunneling gateway functions, i.e., security applications like Internet Protocol Security (IPSec)/ Secure Sockets Layer (SSL) virtual private network gateways which provide virtual access entry for secure traffic flow [24].

• Traffic analysis elements, i.e., Deep Packet Inspection (DPI) analyzes different traffic characteristics at the path level which typically involves pattern matching, deep

protocol dissection, semantic and conversational awareness, behavior analysis, and flow registration.

• Service Assurance, i.e., QoS measures the overall performance considering packet loss, bit rate, throughput, transmission delay, availability, jitter, and so on.

• Next-Generation Networks, i.e., IP Multimedia Subsystem (IMS) provide a regulated architectural framework for multimedia service on the mobile network.

• Application-level optimization, i.e., a load balancer splits network traffic across multiple servers which contain identical functions to achieve optimal performance.

• Network security, i.e., Role-Based Access Control (RBAC) allows users have access to services based on their assigned privileges.

2.2.3 Cloud-native Network Function

Cloud computing is a solution for VNF isolation and performance improvement by moving VM-based VNFs from general-purpose computers to high-performance cloud computing centers that can run the application in terms of decoupled container-based microservices. Container is a lightweight and OS-independent technology that self-contains all packages needed for an application to run which is a suitable form for developing microservices. A cloud-native VNFs can be quickly deployed as a container running on a container engine such as Docker [32] or Containerd[33], then container orchestrators like Kubernetes and Docker Swarm [34] can be introduced for scaling purposes. By merging this cloud-native approach, container image can be globally distributed, hence developing a new VNF could be agile. Even though a container is not a complete replacement for VM but using it to deploy VNFs can be beneficial. Some of the advantages of using containers instead of VMs are listed below [3]. Fig. 2.4 summarizes and highlights the features of the container compared with VM.

Virtual machines	Containers
Each VM runs its own OS	All containers share the same Kernel of the host
Boot up time is in minutes	Containers instantiate in seconds
VMs snapshots are used sparingly	Images are built incrementally on top of another like layers.
Not version controller	Images can be differed and can be version controlled.

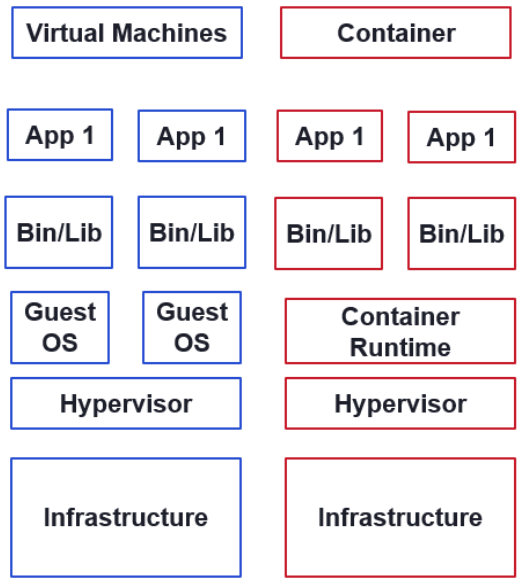


Figure 2.4: The brief feature summary of container vs VM [3]

• Container reduces operational overhead since it is lightweight and resource-efficient.

• Containers are isolated and OS independent. SFC can be deployed as a set of containers running inside one VM.

• Containers can be easily and quickly deployed, managed, and scaled.

2.3 Kubernetes

Kubernetes is an open-source CNCF project, which is meant to be a container orchestration engine for the automation of containerized application deployment, scalability, and administration [4]. It is not a traditional all-inclusive Platform as a Service (PaaS) system since it is not monolithic, but it provides common PaaS features such as deployment, scaling, and letting the user integrate their applications. Kubernetes, also known as K8s, becomes the world's most popular container orchestrator nowadays

and has an active community and various plug-ins that fit the needs of different users.

Typically, in a productive environment, an application runs inside the container is isolated and transparent to the user. It is unlikely for the user to find out whether the container is working properly in real-time, and a manual container reboot is usually involved in this debugging process. Kubernetes takes this job and provides other services along with it. In general, it takes care of failover and scaling for applications, provides deployment patterns, load balancing and secure communications between services, storage management, and provides various customized features with an abundant amount of plug-ins.

Kubernetes nowadays is a huge cloud-native eco-system that have various concepts, many plug-ins, and add-ons. Most importantly, due to the active usage and distribution, Kubernetes is currently undergoing constant evolution. This thesis will not be able to cover everything but only introduce what has been used in this thesis.

2.3.1 Control Plane in K8s

Kubernetes are deployed in form of clusters. Usually, a Kubernetes cluster needs to have at least one working machine called the master node in which all the control plane components run and one or more worker node that runs the containerized application. The control plane and its replicas usually run across multiple computers to provide fault tolerance and high availability. The high-level architecture of the Kubernetes cluster is shown in Fig. 2.5 [4]. Kubernetes control plane consists of

- *API server*: it receives requests and acts accordingly. It is the front end of Kubernetes' control plane.
- *Controller-manager*: It runs the controller process. There are different types of controllers in K8s' environment such as node controller for node management and

service account controller for service account management.

• *Scheduler*: it decides when and where a new *Pod* run based on multiple criteria including required resources, policy constraints, affinity, load-balancing, and so on.

• *etcd*: it is a consistent database for K8s' cluster data storage

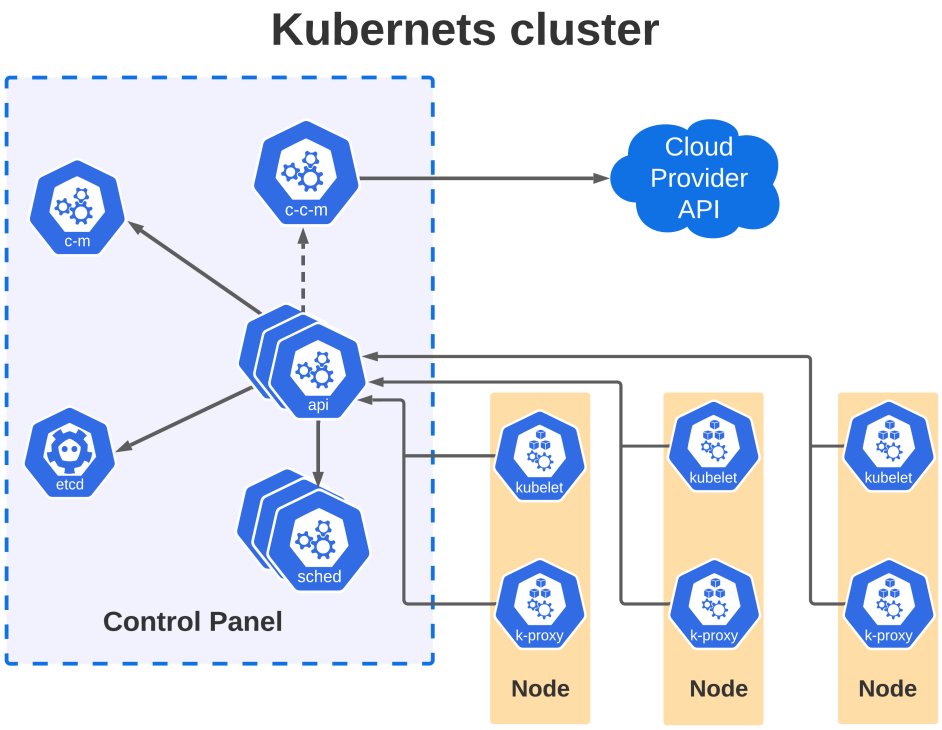


Figure 2.5: High-level architecture of Kubernetes cluster [4]

2.3.2 K8s Node Components

As Fig. 2.5 shows, each worker node has several Kubernetes services running to maintain the environment:

• *Kubelet*: It is a process that runs on each node of a Kubernetes cluster and creates, destroys, or updates *Pods* and their containers for the given node when instructed to do so.

• *Kube-proxy*: It manages the network rules for the nodes. It controls the traffic between *Pods* and clusters by using the OS packet filtering layer by default. It is also controlling the ingress and egress traffic of the cluster by using Kubernetes *Service*.

• *Container runtime*: Another hidden node component is the container runtime which is the container engine the Kubernetes uses to run the container image.

Kubernetes uses Containerd as container runtime by default after version v1.24 [32]. Before the v1.24, the default container runtime was Docker, and the Container Runtime Interface (CRI), Dockershim was used to integrate Docker with Kubernetes. Recently, since the Dockershim is deprecated due to its heavy weight in the updated Kubernetes environment, if the developer still wants to use Docker as the container runtime, the cir-dockerd must be used as the CRI instead. The CRI is a protocol defined by K8s, in which kubelet uses CRI to communicate with container runtime based on Remote Procedure Call (gRPC) which is not naturally supported by Docker [35]. Hence, Dockershim establishes the bridge between the container runtime and the kubelet using API encapsulation in the old version of K8s. Nowadays, with the development of K8s and Docker, not only the Dockershim replaced by the cri-dockerd, but also another lightweight container engine is used by default in the Kubernetes cluster called Containerd which supports gRPC naturally and has integrated CRI [33]. Containerd also eliminates the encapsulation process which makes deploying containers easier and faster.

2.3.3 K8s Data Plane

The Kubernetes data plane consists of *Pods*. Each *Pod* is the smallest component in the Kubernetes cluster where the application workload is located. A *Pod* can encapsulate an application composed of multiple containers that are typically functionally related.

These co-located containers may share contents and resources, Namespaces, and Cgroups. A typical example of a co-located container is *init-container* which runs and completes before the service container started. Another example is the *sidecar* container which has dedicated complementary functions. A *Pod* running on the node has a defined lifecycle which means the *Pod* will not be self-healing after the node restarts. In Kubernetes, instead of managing *Pod* itself as a workload, there are better ways to manage it since *Pod* is designed as relatively ephemeral, disposable entities [36]. Instead, to expand the life-cycle of the *Pod*, it can be managed as follows:

- *Deployment*: *Pod* can be running as deployment which provides extra features such as duplicate *Pod* replicates, load balancing among the same *Pods*, and self-healing. This is a good fit for managing a stateless application workload.
- *StatefulSet*: *StatefulSet* is used whenever the state of the application needs to be tracked. A good example of that will be a database where data are persistently recorded among different replicates.
- *DaemonSet*: *DaemonSet* provides facilities for each node which means that whenever a new node has been discovered, a new *DaemonSet Pod* will be initiated in that node automatically.

2.3.4 K8s Service

Another essential object in Kubernetes networking model is the *Service* which maps a single IP address to a set of *Pods* no matter if the *Pods*' IP address has been changed [37]. By using the *Service*, the mapped *Pods* can be discovered by the rest of the *Pods* running in the cluster as the backend of the *Service*. In general, a backend *Pod* provides functionality to other *Pods* in the cluster but usually needs to combine with a *Service* to expose itself. Most importantly, the traffic from the backend *Pods* can

be redirected outside of the Kubernetes cluster using *Service*. A *Service* will have its internal cluster IP assigned by the cluster Domain Name System (DNS) server. As mentioned before in Section 2.3.3, the state of a *Pod* is not permanent, and the assigned IP of a *Pod* will be dynamic since *Pod* can be created and deleted based on the needs of the cluster. *Service* is the solid solution that keeps track of which backend *Pod* it connects to and exposes the backend *Pod* to the cluster in many different formats. Instead of using an IP address, the *Service* connects to a backend *Pod* by leveraging a field named *Selector* which selects the backend *Pod* based on the *Pod* Labels. The most common type of *Service* is *ClusterIP* which exposes the service to the rest of the components in the cluster by internal IP. It can be used to connect applications running inside the cluster. On the other hand, *NodePort* maps the application port in a container to a port in the host. It exposes the *Service* to the outside of the cluster. Other *Service* types such as *LoadBalancer* are not used in this thesis because it needs a load balancer from cloud providers such as GCE, AWS, Azure, Red Hat, etc.

After the *Service* is created, there are two primary methods to exposing a *Service* to *Pods*. The first one is using environment variables in each *Pod*. The *Pod* will be injecting some environment variables when it gets created and environment variables could be constantly updated when *Service*'s fields are changed. *Pods* are using these environment variables to trace Services. Another method to discover *Service* is through DNS. A cluster-aware DNS server, such as CoreDNS, watches *Services* creation from the Kubernetes API server and creates a set of DNS records for each *Service*. A *Pod* that requests a particular *Service* using a DNS record will receive the assigned cluster IP address for the *Service*.

2.3.5 K8s Secret

The *Secret* is another key object of the Kubernetes API that has been used in this thesis. It secures sensitive information, communication channels, and application authentication. A *Secret* is a Kubernetes object that contains sensitive data or the path to the sensitive file [38]. To use a *Secret* in the Kubernetes cluster, a self-signed certificate needs to exist before the secret object is created, so that any *Pod* that has access to the certificate can reference that *Secret*. Kubernetes itself will create some *Secrets* to use for its control plane application and default user authentication.

2.3.6 K8S ConfigMap

Compared to the *Secret*, the *ConfigMap* is used to store non-confidential data in key-value pairs [39]. *Pod/Pods* then can use this key-pair value to configure application containers. After the *ConfigMap* has been created, *Pods* can use the data inside the *ConfigMap* as environment variables, command-line arguments, and configuration files. The *ConfigMap* and *Secret* can be attached to *Pod* in terms of *Volume*. It essentially decouples the container images from the environment-specific configuration. The *Pod* refers to the *ConfigMap* must be in the same namespace as *ConfigMap*. Like *Secret*, Kubernetes also creates some *ConfigMap* for control panel operation.

2.3.7 K8s Volume

Similar to the volume concept in Docker, the K8s *Volume* was brought up to solve problems such as ephemeral on-disk files and file sharing between containers [40]. In the Kubernetes environment, container restart losses the on-disk files loaded with it hence bringing the problem of limited file management compared with the lifespan of

the *Pod*. In addition, by using this object, one single *Volume* can be simultaneously loaded by several containers and one container can mount many *Volumes* at once. *Volume* essentially is a directory where some data is stored, and the data is usually present in the form of *ConfigMap* or *Secret*. By adopting the *Volumes* concept, the data in the *Volume* can have a lifetime of a *Pod* no matter whether the *Pod* is restarted or not, and the data is also reusable and scalable. Kubernetes support many types of *Volume* for different user requirements.

2.3.8 Network Models in K8s

In a nut shell, Kubernetes provides basic network services such as highly-coupled container-to-container communications in same Pod via *localhost* and port number. However, the *Pod* to *Pod* communication has to rely on implementing the Kubernetes networking model. Moreover, Kubernetes depends on third-party tools to provide network functionality that supports SFC deployment such as DNS services and traffic steering. Kubernetes only provides the networking model placeholder and does not focus on providing networking services such as separated layer 2/layer 3 (L2/L3) data plane communication between *Pods*. Due to its flat network structure, it also lacks support for separating the SFC data plane from the Kubernetes control plane.

Third-party tools develop networking extensions to support the networking module requirements by inserting the Container Network Interface (CNI) to containers running inside *Pod*, while each may have a different focus. For instance, Calico supports multiple types of data planes such as standard Linux networking data plane, Linux eBPF data plane; Cilium provides IPv6 container network and enforces network policies on L3-L7. Some of the most common network plug-ins are Calico, Cilium, and Flannel [41]. Meanwhile, two Kubernetes network plug-in that supports SFC deployment are

Contiv-VPP [42] and OVN4NFV [43]. The Contiv-VPP network model accommodates SFC deployment by supporting Segment Routing IPv6 (SRv6) using contiv-vswitch. In comparison, OVN4NFV achieves SFC deployment by adopting the Open Virtual Network (OVN)-based CNI controller. However, those two SFC solutions are not suitable for the SFC development presented in this thesis since Contiv-VPP does not focus on the development of SFC itself but rather achieves the SFC by taking the advantage of SRv6. On the other hand, the OVN4NFV solution is archived by the owner and not maintained for the later Kubernetes version. Hence, the NSM approach is brought up for developing the container-based SFC in the proposed testbed for a dedicated Kubernetes environment. The next section describes the details of the NSM.

2.4 Network Service Mesh

NSM is also an open-source CNCF project which exploits the idea of networking the microservices from traditional service mesh solution such as Istio and Linkerd. Instead of establishing connections at layer 7, it focuses on layers 1-3 communications. Its initial design is to solve the problem of inter-communication between workloads running in multiple K8s clusters in a multi-cloud environment. NSM provides connectivity between workloads independent of where they are running by using service mesh at L2/L3. Hence, traditional service mesh itself can be considered as a single workload that can be connected to a bigger service mesh network by NSM. This feature also can be used to deploy SFC in a cloud environment.

As mentioned above in Section 2.3, Kubernetes by itself does not provide advanced L2/L3 network features and it lacks the support for separating the data plane and

control plane. For these reasons, the NSM can be integrated with K8s to provide the control plane for the SFC deployment. In addition, a separated SFC data plane can be established by the NSM traffic forwarder. NSM utilizes Kubernetes' networking model to perform specific networking functions regarding the requirements of SFC. NSM is a novel approach to solving complicated L2/L3 use cases in Kubernetes by providing V-wire connections between *Pods*. The NSM approach to SFC is based on three basic concepts [44]:

(i) *Network Service* (NS): It defines the traffic rules for L2/L3 service by selecting *Pods* with Network Service Endpoint (NSE) label and declaring the order of the traffic flow. It also defines the metadata of the SFC such as name, namespace, and traffic payload type.

(ii) *Network Service Client* (NSC): It is the client *Pod* that requests an SFC by using the metadata name declared in the NS.

(iii) *NSE*: It is a *Pod* in the Kubernetes cluster that provides the network services. NSE has to be registered with the NSMgr and *NSM Registry* with an NSM_LABELS before consuming it by any NS. Besides, in the NSM architecture, operations such as registering an NSE with a label, or requesting NS using metadata name must be done through setting container environment variables. Furthermore, NSM extends beyond kernel interface to support complex use-cases and provides other interfaces such as Shared Memory Packet Interface (MEMIF) or Virtual Function I/O (VFIO) interfaces. A MEMIF interface provides high-performance packet transmitting and receiving between the user application and VPP [30].

By implementing a separate network interface, NSM allows SFC creation independent of the infrastructure they are running on. That is a benefit from the structure of the NSM control plane components which are running as *Pods* in the Kubernetes

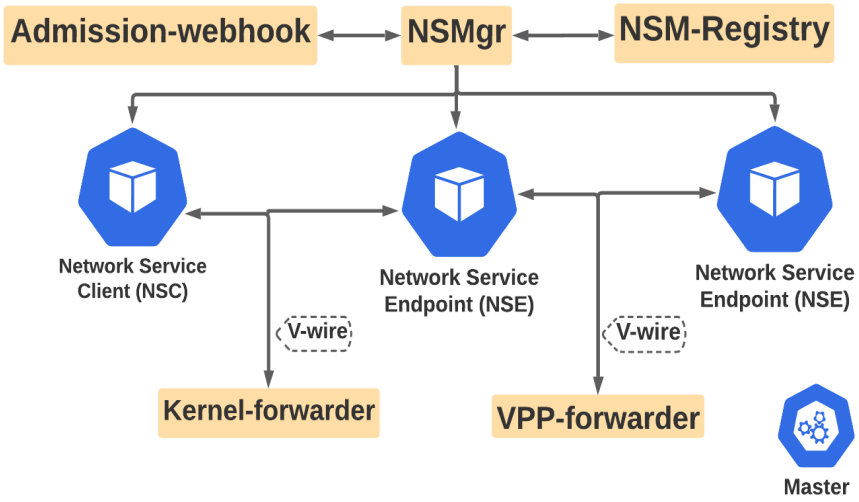


Figure 2.6: The Key components of the NSM framework [5]

cluster. The followings are the NSM control plane elements:

(i) *Network Service Manager* (NSMgr): It handles the communications in the control plane and manages the collaboration of elements involved in SFC creation. In general, the NSMgr *Pod* consists of three essential duties. Firstly, it is responsible for receiving all requests that involve cross-connections construction. The second duty is to take actions based on the request received, such as communicating the network service registry to register an NSE and sending requests to the *Admission-Webhook* to inject an initiation container for the Client *Pod*. Finally, it periodically checks the rest of the control plane components to ensure the deployed SFC is working correctly. It organizes the control plane components to achieve automatic SFC recovery when detecting any faulty component.

(ii) *Network Service Forwarder*: It is the SFC traffic adapter that carries packets from one end of the V-wire to another end. When the communication tunnel between *Pods* is provided, it receives traffic from the ingress interfaces and forwards the traffic to

the egress interfaces. It creates a cross-connection between involved *Pods* by providing various implementations of interfaces such as VPP, Kernel, VFIO, etc.

(iii) *NSM Registry*: It registers NSCs and NSEs which will be selected when creating SFC.

(iv) *Admission-Webhook*: It injects the *init-container* for NSE and NSC when *Pods* start. The *init-container* is responsible for creating the NSM interface.

(v) *SPIRE Agent/Server*: SPIFFE is an open-source standard for securely identifying software systems in a dynamic and heterogeneous environment. In our use case, the SPIFFE is implemented as SPIRE Agent/Server that registers workload identities and provides the network policies to manage [10].

Fig. 2.6 demonstrates the key control plane elements of the NSM system with the communication links [5].

2.5 Prometheus

This section provides the fundamentals of the adopted monitoring solution which can proactively monitor the hardware and network resources while adding minimum overhead to the system, it has also been used to provide SFC performance analysis in the proposed testbed.

The scale of the cloud-native ecosystem such as data centers and cloud platforms has grown dramatically in the last decade. The revolution of the cloud-native ecosystem requests the monitoring and management tools to not only have a basic understanding of the hardware resources, but also can proactively detect potential issues and warn the administrators. Prometheus, an open-source CNCF project plays a vital role in providing proactive and high-availability data monitoring solutions [45]. It implements

a highly dimensional data model with a time-series database (TSDB). TSDB is a type of database optimized for time-stamped metrics. Furthermore, it has a powerful, user-friendly query interface for users to access and displays the data. Generally, the Prometheus system was designed for the cloud-native environment giving the following characteristics: (i) energetic monitoring; (ii) high availability and scalability; (iii) proactive alerting; (iv) trend analysis [45].

In the cloud-native SFC deployment testbed demonstrated in this thesis, Prometheus was used to monitor the host-level and container-level hardware and network resources and validate the SFC traffic flows. The Prometheus community has developed multiple Prometheus project branches using different mechanisms. The two main branches are Prometheus Operator and Prometheus-community Helm chart. The Helm is a Kubernetes development tool for automating, packaging, and managing the Kubernetes objects in form of a Helm chart [46]. Helm automates maintenance of YAML manifests for Kubernetes objects by packaging information into charts and advertises them to a Kubernetes cluster. It is worth noticing that YAML config files are human-readable data-serialization standard configuration files in the industry. Fortunately, the latest version of the Prometheus Helm chart has integrated the Node Exporter (NE) and a third-party exporter, cAdvisor for monitoring node-level and container-level metrics. In addition, various types of exporters are available for conducting dedicated data collection such as host hardware resources, IoT devices, etc. Similar to the NSM framework, the Prometheus core services need to run as *Pods* inside the K8s cluster because the K8s cluster is highly isolated. Hence, Prometheus can skip the extra authentication process to collect the data from the SFC *Pods* located inside the K8s cluster. By running inside the K8s, it can also extract both the data metrics from the K8s cluster and the host environment.

The Prometheus control and management plane have the following critical elements running inside the K8s namespace after employing the Prometheus Helm chart [6]:

(i) *Prometheus Server*: this is essentially the TSDB that keeps all the collected metrics and runs the PromQL API server which is responsible for the data query. Prometheus Server *Pod* also runs the integrated Express Browser UI that can be used to visualize data.

(ii) *Alert Manager*: Prometheus supports an alarming system that is based on the PromQL rules. A customized alert can be sent out in the form of email, Slack, or SMS text messages.

(iii) *NE*: different types of metrics collectors are used to collect data from the node environment by applying both black box and white box surveillance. NE employs HTTP protocol to pull the metrics from the integrated endpoint. One of the advantages of using such decoupled and extensive architecture is that the user has the flexibility to choose which exporter is needed, and the developers can use the client library to develop a customized NE to collect data they are interested in.

(iv) *Kube-state-metrics*: integrated with kubelet and cAdvisor to get the cluster metrics directly from Kubernetes API which includes container-level metrics.

(v) *Grafana*: A open-source web-based visualization tool to display the data quired from the Prometheus database. Many pre-built layouts are available for users to choose from.

The general architecture of the Prometheus system is shown in Fig. 2.7. By adopting the cAdvisor, the Prometheus would collect some information that benefits us the most such as (i) the latency for *Pods* initialization, (ii) the network device name and speed, (iii) the network throughput during the sample period, (iv) the node CPU temperature, (v) the container CPU and Memory utilization, (vi) the HTTP

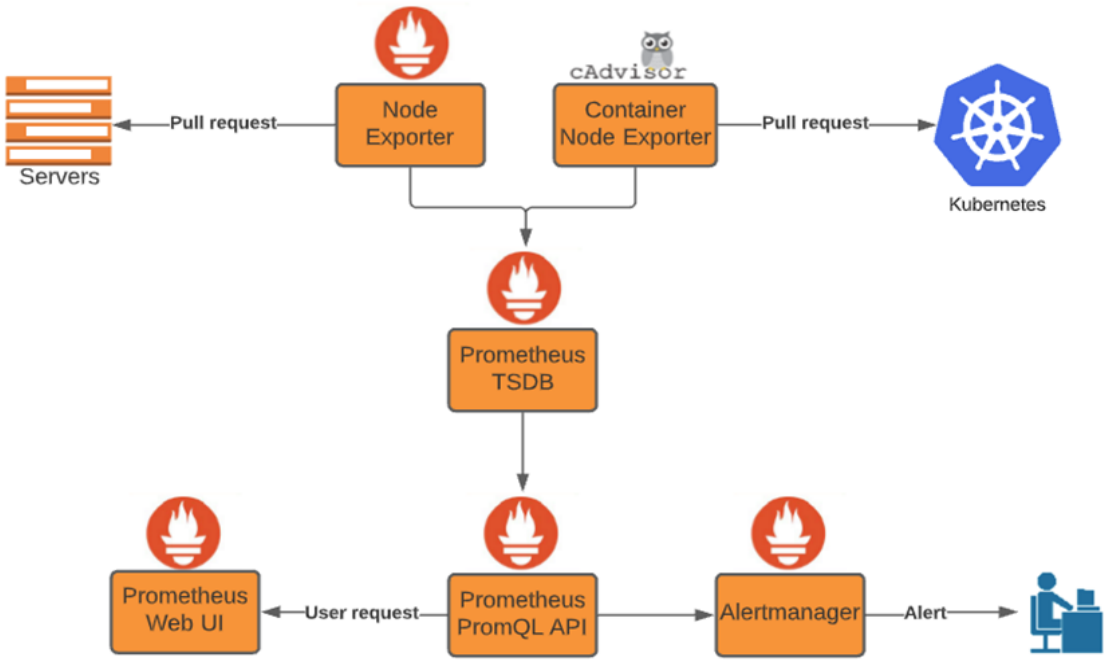


Figure 2.7: The key components of the Prometheus monitoring framework [6]

request delay,(vii) the HTTP request error rate, (viii) the total number of HTTP request and its increasing rate, etc. In addition, by leveraging the powerful PromQL built-in functions and aggregate operations, it would be possible to calculate the data pattern and predict the trends.

2.6 Literature Review

Bhamare et al. [9] indicated different open issues and challenges of SFC such as dynamic service mapping, dynamic traffic steering, reliability, and service availability. The survey also pointed out several implementations of the SFC architecture with the OpenFlow switch and NSH protocol. Several resource management systems for implementing SFC were mentioned by Bhamare et al., such as OpenStack, and Metro Ethernet Forum (MEF).

Morabito et al. [47] implemented the container-based SFC and evaluated its performance in IoT scenarios. The authors demonstrated that the SDN with container-based virtualization ensures several benefits in terms of scalability, flexibility, easy programmability, and versatility. The performance of the container-based SFC were analyzed in terms of overhead, network bandwidth, and energy consumption in the IoT Capillary *Gateway* testbed. The Docker was used as the container platform deployed on top of the RasberryPi3. By leveraging the *Docker Network Driver Plugin*, the container network was provided by a virtual OvS that is controlled by OpenFlow controller *POX* [48]. The performance results show that there was a significant resource-saving in terms of computation cost, and system overhead by using the containers. The CPU utilization of the testbed is a roughly linear relationship against the number of containers. The RAM usage highlights the lightweight features of the container-based application. Meanwhile, Livi et al [49] proposed a similar container-based SFC testbed using OvS and Docker. The authors focused on evaluating the performance of the UDP/TCP type of SFC data path that composes twenty OvS switches and comparing it with the data path composed of virtual ethernet. The authors concluded the container is an appealing candidate for implementing NFV-based service chains in public and private data centers based on the performance loss model. However, they noted that the networking performance loss was increased with the increment of chain length. Hence the performance and the reliability of the service were not guaranteed when the number of middle-boxes increases. This is one of the vulnerabilities container-based SFC was facing due to the lack of stable and high-performance CNI. In both testbeds mentioned above, the container orchestrator Kubernetes that provides a high-available cluster was not used to enhance the reliability of the deployed network services. The SFC recovery feature was not supported, and the real-time network traffic metric was

not collected comprehensively. Hence, network operators could not be aware of the SFC status which leads to an unreliable situation.

Nowadays, container-based network services with cloud-native architecture such as Kubernetes become the hot topic for 5G stakeholders. In [14], the authors offered a Ketama-based traffic steering to maximize the QoS satisfaction rate by load-balancing the traffic over the SFC path using the Contiv-VPP network plug-in. The Contiv-VPP also supports the SFC deployment in Kubernetes using DPDK and VPP modules that allow the application running in user space to leverage the high processing speed of the NIC. Network states and data collection were involved in the paper, but the authors used different SFC and data collection approaches. The iperf tool was used to generate traffic and collect network stats which is not the ideal case since iperf cannot detect the communication in the application layer and is not able to predict the traffic trend based on historical data. In addition, the iperf testing itself consumes a huge amount of bandwidth which is not feasible in a production environment if the network operator wants to get the real-time bandwidth. The authors were aware that the traffic steering methodology still needs to be improved by using a fine-grained monitoring solution to provide better QoS. The monitoring system, Prometheus, is much more comprehensive considering its proactive features and hardware resource monitoring feature for recent network development. Moreover, the reliability of the Contiv-VPP-based SFC was not studied comprehensively.

Meanwhile, the author in [7] focused on integrating Kubernetes with OpenStack to deploy a complex system that leverages both the advantages of VM and container to build an SFC crossing multiple virtualization domains, named *Phishahang*. *Phishahang* is built on top of the SONATA MANO (Management & Orchestration) [50] framework that allows network services to run their orchestration code. The high-level

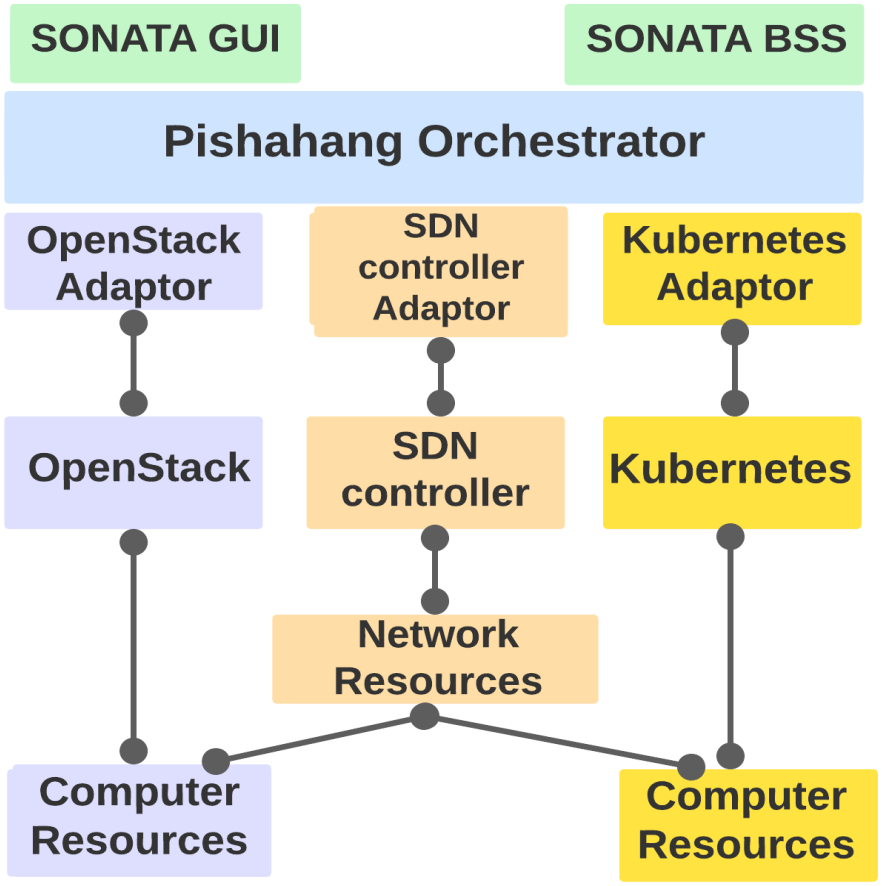


Figure 2.8: The high-level architecture of the Phishahang testbed [7]

architecture is demonstrated in Fig. 2.8. The authors claimed that the container orchestrators like K8s were incapable of providing service chaining due to the dynamic IP address allocation in *Pod* life-cycle management. Hence, they extended the *Pishahang* orchestrator to support this missing feature by adding multiple adapters to collaborate with different orchestrators and handle the communication between the SDN controller and the *Pishahang*. However, the approach requires a fixed IP address for *Pod* deployment and a MAC address for VMs deployed in OpenStack. The authors described a possible demo scenario which is a service chain consisting of two VNFs from different virtualization domains by using the proposed *Phishahang*

framework, but it required the SFC framework from the OpenStack Neutron Networking [51] module. This approach implemented a combination of NFV tools to support SFC across heterogeneous domains, however, the reliability and robustness of the multi-domain SFC were not studied. Similar to the previous approach, the author in [52] introduced an extended Tacker architecture to support and manage the container-based VNFs in the Kubernetes environment. The Tacker is an NFV manager and orchestrator in the OpenStack environment [53]. The authors adopted Kubernetes as the container orchestrator since they demonstrated that the container is a promising virtualization technology for deploying and running distributed applications due to its lightweight and agility features. However, the proposed testbed only considered Kubernetes as the container orchestrator and the SFC architecture relied on the OpenStack networking-sfc [54]. The authors did not focus on the resource-aware features provided by adopting proactive monitoring or failure recovery features provided by the Kubernetes application life-cycle management.

Most importantly, the authors in [41] proposed a novel traffic steering algorithm to route traffic in cloud-native SFC using a dynamic weighted round-robin. The authors proposed the SDNless container-based SFC solution by using the NSM approach. They addressed the traffic steering problem between service replicas to provide high-quality end-to-end network services. They focused on developing and testing the traffic steering algorithm rather than building a reliable cloud-native SFC testbed. The paper also did not emphasize collecting network states and hardware resource metrics. Similar to the solution proposed by the authors, this thesis adopts the NSM framework as the SNDless cloud-native SFC solution due to the efficient traffic steering features. Furthermore, the reliability and robustness of the deployed SFC are enhanced by adding state-of-art modules to provide features such as real-time SFC

traffic monitoring, automated failure recovery, workload authentication, computation, and network resource monitoring.

In terms of reliability and fault recovery, the authors in [8] proposed a novel fault management system that can monitor the current SFC deployment and dynamically recover the fault container-based VNF by creating a backup VNF with the same functionality to replace it. This approach utilized Tacker, networking-sfc, and Kuryr-Kubernetes [55] to launch SFC between VMs and containers in both OpenStack and Kubernetes environments. The monitoring system constantly sent ICMP ping echo messages to check the health of the deployed VNFs. However, this approach did not monitor hardware resources such as CPU usage and memory usage or the network status such as latency and connectivity which may potentially be the root reason cause of the unreliable network service.

In terms of environmental monitoring, the authors of [8] claimed the Kubernetes can be deployed in an edge computing environment to serve latency/QoS-critical applications benefiting from the flat network structure in K8s. The authors presented a comprehensive testbed enabling the K8s scheduler to interact with the SDN controller for deploying services, taking into account network constraints. The testbed implemented an ONOS SDN controller with P4 switches which support in-band and post-card telemetry (INT) to collect network-related data. Further, the SLA broker constantly monitors the collected dataset and reacts based on pre-defined policies. Fig. 2.9 demonstrates the workflow of this testbed. Similar to their approach, the proposed testbed in this thesis adopted the idea of collecting network metrics while using a different mechanism. In addition, hardware resource usage is collected in our proposed testbed to give more information for the network operator to diagnose. Hence, the deployed latency/QoS-critical services not only benefit from the flat network structure

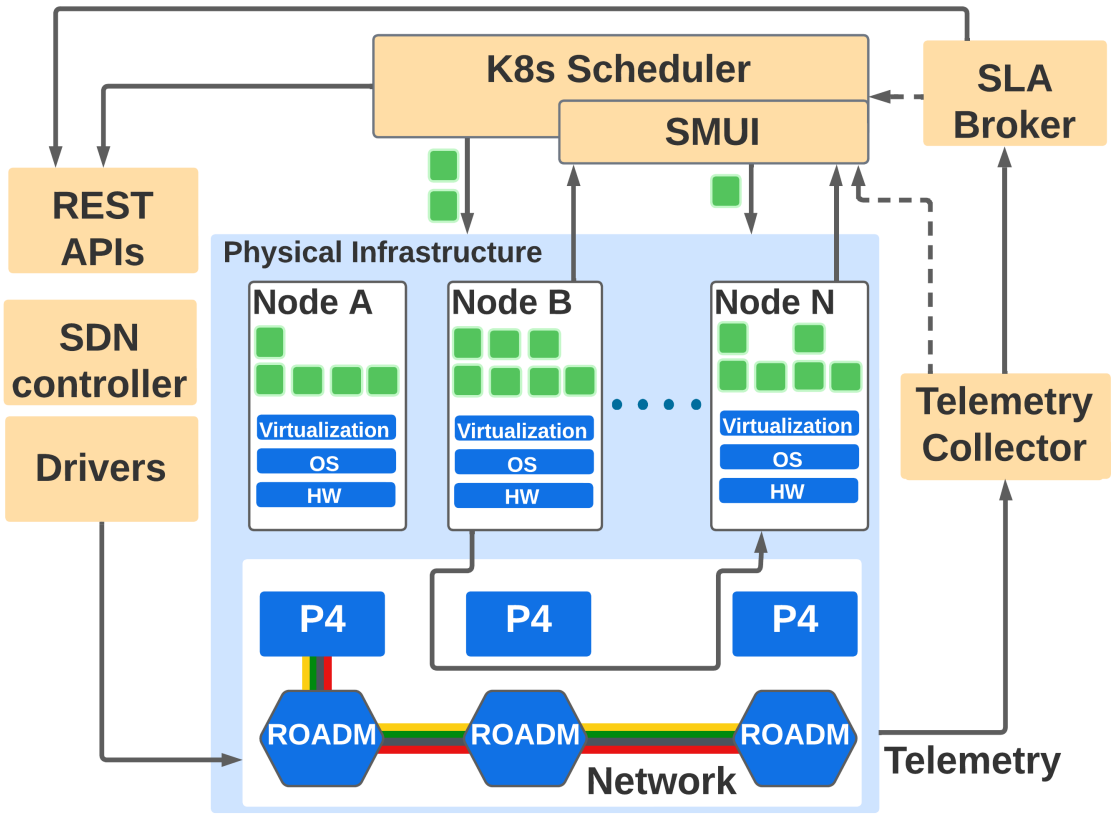


Figure 2.9: The workflow of INT-based NFV testbed [8]

but also benefit from the resource usage and utilization.

In [56], the authors showed that the container-based microservices architecture brings challenges to service performance and resource management due to the additional granularity compared to VM-based microservices. The performance and the reliability of the container-based microservices were constrained by QoS metrics (i.e., latency, serving throughput, request rate) which were collected by NFV-Inspector [57]. They used the machine learning approach to estimate the relationship between the QoS and microservice resource configuration, then the Machine Learning (ML) model was used to predict the required resources to set up a productive environment and the capability of a productive environment. However, this method did not specifically apply to the

usage of SFC deployment since the performance of SFC depends on dependent microservices that collectively provide an end-to-end service. Analyzing the QoS metrics of a single microservice may not reflect the performance of the entire SFC. The issue was addressed by the authors in [58]. The cloud-native SFC composed of inter-dependent microservices were usually assigned across multiple services and network domains. Due to the mobility events caused by moving end-users, the reliability of the SFC is often in danger while reallocating resources to immigrate dedicated VNF close to end-users. Hence, monitoring resources across multiple domains are essential to ensure the success of SFC deployment. The theme of the thesis is to establish a reliable and robust cloud-native SFC testbed using the state-of-art monitoring system Prometheus in the Kubernetes environment to effectively support white-box and black-box monitoring. Through the white box, it is possible to understand the actual operating status of its internal, and through the observation of monitoring indicators, it is possible to predict possible problems, to optimize the potential uncertain factors. On the other hand, black-box monitoring, such as HTTP probes, TCP probes, etc., can quickly notify relevant personnel for processing when a system or service fails.

To this end, here comes the related work regarding resource management and QoS monitoring. The authors of [59] categorized the application running inside *Pod* into CPU-bound jobs or resource-bound jobs from the resources overbooking perspective, and they suggested some common hardware parameters to monitor such as CPU and memory usage and read/write disk sectors. Similar to that, authors from other papers [60] [61] also suggested valuable parameters to collect such as network send/receive bytes per second, disk/CPU io idol time, page fault, and so on to give the network operator a better understanding of the system network. Meanwhile, in terms of QoS, the authors in [62] focused on monitoring other Service Agreement Level (SAL)

parameters such as the number of denied requests, total thread count, number of jobs in the queue, and total HTTP requests. This type of information is worth collecting since it reflects the reliability and robustness of the deployed SFC.

2.6.1 Literature Review Summary

Most of the previous research works are mainly focused on the development of VM-based SFC by leveraging the OpenStack Neutron Networking framework. Only a few research jobs target container-based SFC development in a bare Kubernetes environment. Meanwhile, previous research works mainly focus on innovating new SFC approaches and new algorithms for traffic engineering but not on enhancing the SFC reliability and performance analysis. The SFC fault detection and recovery mechanism is lack of investigation in the Kubernetes environment and the benefits of leveraging fine-grained resources and monitoring system for SFC development is not well studied.

Unlike the existing SFC solution mentioned above, our approach focuses on establishing a reliable testbed to support the dynamic creation of customized container-based SFC across multiple nodes in the Kubernetes environment. The proposed testbed integrates a resource and network state monitoring solution supported by the Prometheus system to deliver fine-grained network and hardware resource monitoring. The NSM framework also enhances the reliability of the deployed SFC by promptly recovering the SFC connections from the fault network service. In addition, our approach increases the usability of the proposed testbed by utilizing a web interface that automates the SFC developing process while allowing users to configure the container-based microservices and routing the traffic among them. To the best knowledge of the authors, this work is the first to address the difficulties of developing a reliable SFC

in a cloud-native environment practically and concretely.

Chapter 3

System Design

In this chapter, the devised model of the testbed is presented in detail. Section 3.1 introduces the system architecture. It explains the collaboration between each pair of elements to achieve the creation of an SFC starting from a user request. Hence, the entire process of the container-based SFC creation behind the scenes is delivered in Section 3.2. Finally, Section 3.3 discusses the SFC performance and reliability from the perspective of data collection.

3.1 System Architecture

This section describes the proposed container-based SFC architecture, as shown in Fig. 3.1, with additional extensions to overcome the SFC development issues in the cloud-native environment. The system has integrated the Kubernetes cluster with the Prometheus monitoring solution to have a better understanding of the resource and network states. The Kubernetes plays the role of NFVO that manages the life cycle of the container-based network services that include dynamic resource allocation, optimal application scheduling, load-balancing traffic between application replicas, automatic

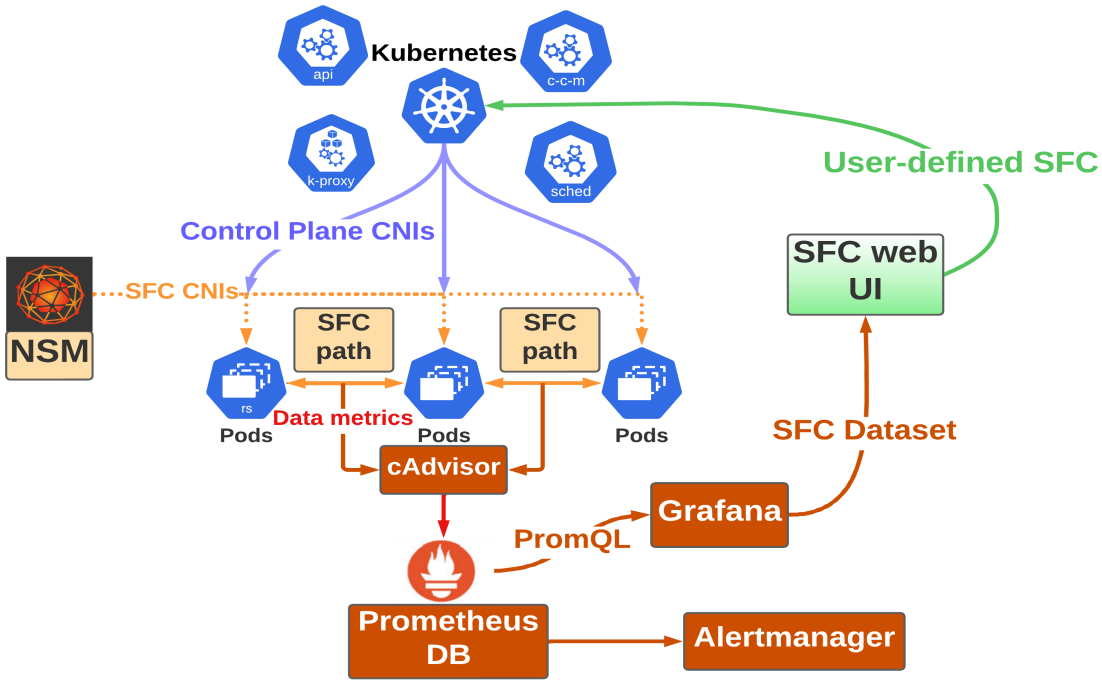


Figure 3.1: The overall system architecture of the proposed testbed

Pod failure recovery, etc. It also provides the centralized control plane for our testbed by offering the kubectl API through the command-line interface (CLI) or Kubernetes web UI. The adopted NSM network plug-in provides the SFC data plane for steering the traffic through the pre-defined SFC path based on network policies. When the traffic is on the fly, the network metrics such as data throughput on egress/ingress ports are collected by Prometheus to validate the SFC path. Prometheus also collects other valuable environment variables such as inter-*Pod* latency, system resource utilization, and services request rate for the network operator to apply further investigation. Meanwhile, all NSM-based SFC workload is authenticated by the SPIFFE protocol implemented in every registered namespace. The individual modules used in this model are described in detail as follows.

The SFC web UI (the green section in Fig. 3.1) is a front-end web interface

that runs on a Python-based web server. The web UI increases the usability of the testbed by allowing the user to create an SFC using some pre-configured SFC from the drop-down button on the top section of the website or to create a new SFC structure by selecting pre-configured microservices in a specific order. After the user finishes configuring the SFC, a request is sent to the backend of the testbed which is essentially the Kubernetes cluster. During this process, Kubernetes CLI, kubectl is used to deploy manifest YAML files in the Kubernetes cluster. Generally, the request sent from the web UI is received by the Kubernetes cluster to create all the *Pods* and network policies that are needed for the SFC if the destination cluster has all the required elements. Despite the NSEs, NSC is usually required since the NSM control plane needs to start the chain by creating the client *Pod* as the SFC gateway. The NSC will be injected with an NSM-based CNI and environment variables to start the chaining process. Meanwhile, the CLI displays the *Pod* creation process so that the user can have a clear understanding of the states of the cluster.

The Kubernetes cluster (the deep blue elements in Fig. 3.1) is the NFVO in our framework which makes the testbed lighter and simpler than the OpenStack cluster. SFCs can be composed of microservices rather than services running on VMs. One of the important advantages of Kubernetes is implementing decentralized architecture based on a declarative model that focuses on defining the container's ultimate state. Users just need to describe the structure and states of the application to be deployed when using the declarative model. Benefiting from the declarative model, the deployed container-based SFC composed of containers can recover to the initially declared states from unpredicted environment changes. As mentioned, Kubernetes provide networking model placeholders that allow third-party network plug-ins to provide network services such as layer 2/3 communication between *Pods*, layer 7 communication between

applications running as *Pod*, etc.

In the theme of this thesis, the proposed testbed implements the WeaveNet (the light-blue lines) as the network model to provide non-SFC communication connections between container-based microservices and the Kubernetes control plane. The WeaveNet is a third-party open-source network model that provides CNIs when *Pods* are created and connects all the *Pods* through regular DNS services [62]. Meanwhile, the testbed adopts the NSM network plug-in (the light-yellow part) to offer the separated high-performance data plane dedicated to SFC traffic. On top of the automatic *Pod* recovery feature provided by the Kubernetes application life-cycle management, the NSM framework provides the SFC connection recovery by resuming the V-wires between the restarted *Pods*.

In addition, the Prometheus system (dark-orange components) can collect container-level information through the cAdvisor and collect host/network-related info from the NE. The monitoring process has been done in control plane which is using the WeaveNet interfaces. Then the collected data saved as metrics in the TSDB can be accessed by the PromQL which is a query language used by Prometheus API to inquire about data from TSDB. Next, an integrated visualization tool, Grafana, is used to display the queried data to give the user an intuitive conviction about the system. Prometheus system also consists of a component called alarm manager which can send an alarm in form of an email or text message to whoever may concern if some pre-configured values such as latency, CPU/memory overhead excess threshold.

3.2 NSM-based SFC

In this section, the SFC creation process is discussed in detail. Fig. 3.2 illustrates this entire SFC creation process with the process step number.

• Step 1 shows an NSE (eg. VNF) send request to NSM manager to start a registration process.

•Step 2 demonstrates the communication between NSM manager and *NSM Registry* after NSM manager receives the registration request from an NSE.

•Step 3 shows a client requesting a new SFC.

•Step 4 shows that when the NSM manager receives an NS request from a client, it will be looking for the existence of a registered NSE.

• Step 5 shows the NSM manager establish the chained V-wire connections between the NSC and NSEs in the NSM forwarding plane.

• Step 5 shows the SFC is built and ready for the traffic.

The NSM Manager, also known as NSMgr, will examine the annotation key-value pair in the client's *Pod* to find out which SFC the client is requesting. Then the NSMgr checks if the required NSEs and connection interface mechanism are available in the cluster. If there is a match, the NSM manager will respond to the request by creating connections between the appropriate requested NSEs. It is worth mentioning that if the required interface mechanism is partially available and all NSEs are registered, some of the requested V-wires can be established. However, in this special scenario, without all the requested V-wires, the partially connected SFC will eventually break since the NSM manager will frequently check the health of the SFC *Pods*. It is worth noting that the container interfaces at the ends of a single V-wire can have different types.

As mentioned before in Section 2.4, any *Pod* that wants to participate in the

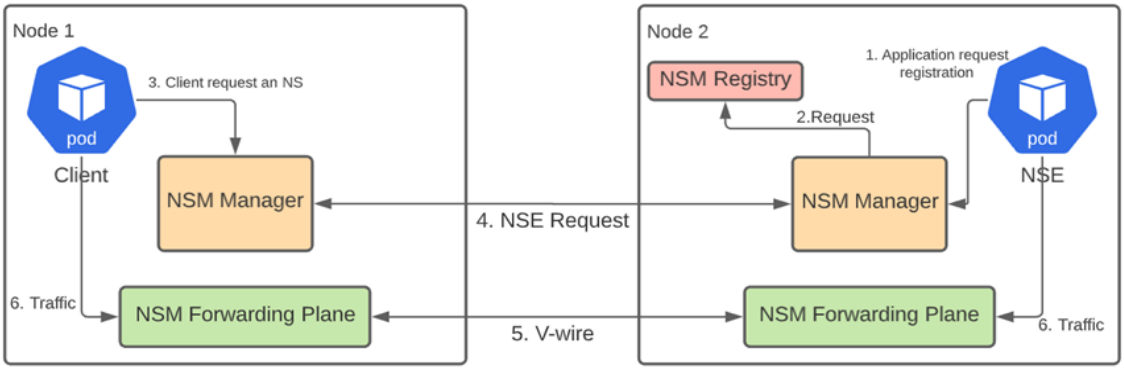


Figure 3.2: The workflow of NSM components when creating a new SFC

SFC must be registered with *NSM Registry* and must support the request interface mechanism. In general, if the user wants to use the Kernel interface in the NSE, it must use the NSM-provided NSE as an SFC-proxy container to inject the Kernel interface for the network service container. Fig. 3.3 demonstrates the functionality and capability of the SFC-proxy container. The SFC-proxy container can be used to accommodate the SFC-unaware container with Service Function Forwarder (SFF) for backward compatibility, but it must run with the network service container in the same *Pod*. The SFC-proxy container is developed and maintained by NSM community developers and currently only supports the Kernel interface. The alternative approach is to build an application container from scratch with the dedicated Software Development Kit (SDK) socket that either supports Kernel, MEMIF, or other types of interfaces. In this case, no additional SFC-proxy container is needed. In conclusion, one NSE can be requested by different NSs at the same time, also any client can request many NSs simultaneously. It means that a single instance of both NSC and NSE can have multiple NSM interfaces at any given time.

The network service function *Pod* in the Kubernetes domain can be modeled as $NSE_i\ (e_k, v_k)$. It is a set of containers running across different K8s domains. The

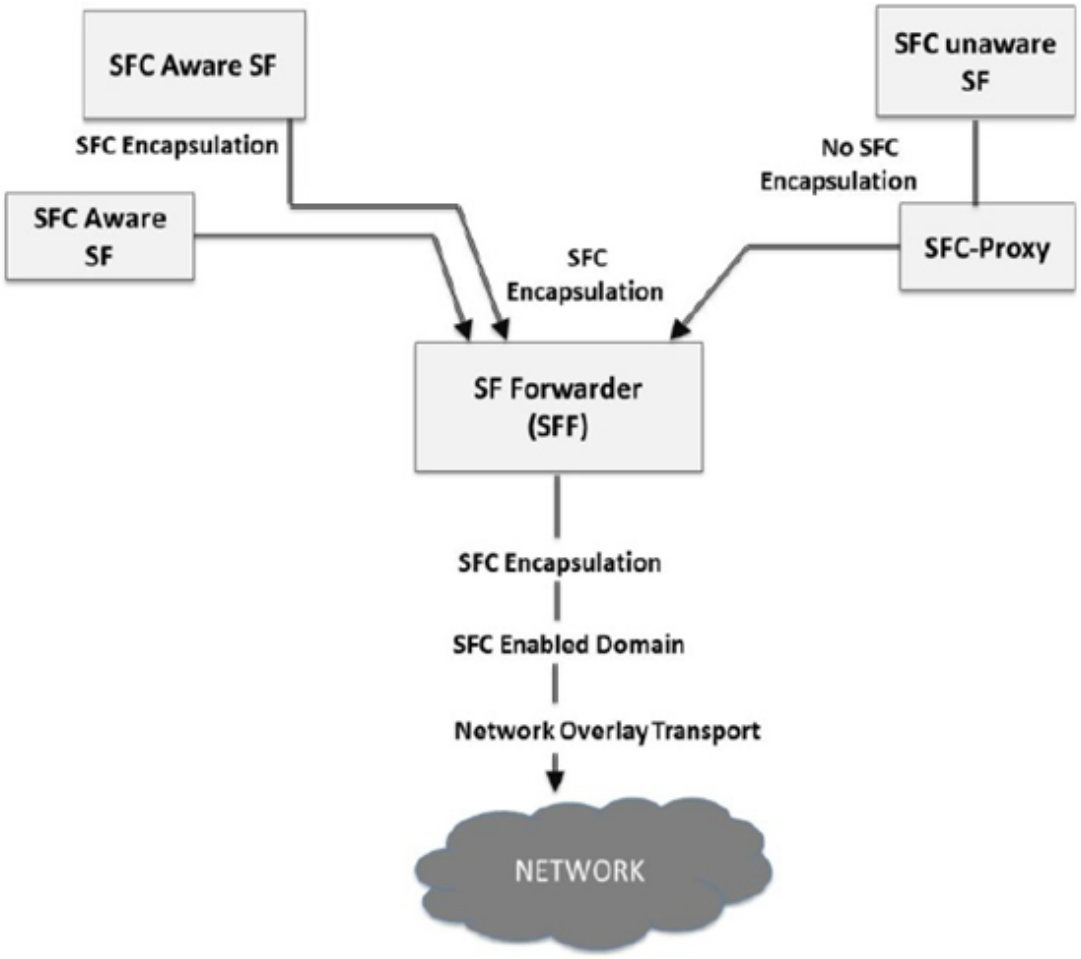


Figure 3.3: The SFC instance with SFC aware SF and SFC unaware SF [9]

first variable e represents the list of interfaces the service function uses. The current provided CNIs are Kernel, MEMIF, VFIO, VLAN, VxLAN, and Single Root I/O Virtualization (SR-IOV). On the other hand, v represents the list of V-wires established by the NSE that may be consumed by multiple NSs concurrently. The annotation $i \in (1, \ldots, n)$, and $k \in (1, \ldots, n)$ represent the indices of the network service and the V-wire. The variables e and v are arrays that may contain a list of elements since the NSE can be consumed by many NSs. For example, a firewall NSE can be represented by $NSE_1([MEMIF_1, MEMIF_2], [v_1, v_2])$. It means that this firewall *Pod*

has two MEMIF interfaces and connects with V-wire 1 and V-wire 2. It is the first NSE in the NS chain indicated by the NSE notation. If an NSE is represented as $NSE_1([MEMIF_1, MEMIF_2, MEMIF_3, MEMIF_4], [v_1, v_2, v_3, v_4])$, it may be used by two SFCs simultaneously.

The network service client, NSC, could be represented as $NSC_i(e_k, v_k)$. Like NSE, the first variable e represents the list of interfaces the client requests. And the second variable v presents the list of established V-wires. The variables e and v are arrays that may contain a list of elements since a single NSC can request multiple NSs. An instance of a client *Pod* could be $NSC_1([Kernel_1, Kernel_2], [v_1, v_2])$ which means that this NSC is attached with two SFCs simultaneously with Kernel interfaces.

The model of the network service, $NS_i(p, NSE_n)$, defines the payload type of an SFC and the traffic steering rules. The first variable p defines the payload type. Two payload types are currently supported: IP and Ethernet. One NS can only carry one type of payload. Hence the variable p is either IP or Ethernet. On the other hand, the second parameter, NSE_n, is an array containing a list of NSEs consumed by this single NS, hence $(NSE_1, NSE_2, NSE_3, \ldots, NSE_n) \in NSE_n$. It is worth mentioning that the NSC is not part of any NS since the NSM system has no control over the client, which is also the purpose of this proposed architecture.

3.3 A Reliable SFC

The services offered by an ISP as a set of functions should be available all the time for end-users. The service connectivity should always be dynamically checked and there is a need to develop mechanisms for fault detection, fault recovery, and fault isolation. On the other hand, most of the NFV testbeds for deploying SFC, by

themselves, do not have any insight into the system environment, especially in terms of resource usage trends and network traffic flows. Developers are not able to know the system overhead for the SFC deployment and lack of information about the actual traffic steering. Meanwhile, the robustness and reliability of the SFC may also be at risk considering the security level of the container. More importantly, any further improvement requires feedback to provide the capacity and ability of the system from the center and historical view. In the proposed testbed, the Prometheus monitoring solution, NSM, and SPIRE projects have been employed to ensure the performance and reliability of the deployed container-based SFC in a cloud-native environment. Hence the collected features can be cataloged into performance and reliability.

3.3.1 Performance of SFC

The performance of the SFC can be dependent on many factors that are difficult to measure such as the complexity of the function itself, the inter-relationship between functions, and so on. However, despite those infeasible parameters, other important parameters that reflect the overall system performance can be measured and collected. Some methodologies can be applied to help identify the overall system performance, such as Four Golden Signals which Google has been using to evaluate the QoS for end-users and service interruption [63]. It involves measuring: (i) the latency of requests: the key concept is to differentiate between the working request with large latency and large latency due to errors; (ii) the volume of the traffic in the system: monitoring the volume of the current activities; (iii) the error rate: the total number of the error request and the trend of the error rate; (iv) saturation: measures the saturation of the current service: the main emphasis is on the constrained resources that most affect the state of the service.

The Four Golden Signals along with other similar methods such as the USE (Utilization, Saturation, Errors) method can help the operator quickly identify the states and the issues of the system. By actively adopting this principle, network operator can effectively indicate the SFC performance and bottleneck of the deployed SFC in a cloud-native environment. For instance, network operator can measure the start latency of new *Pod* deployment to determine if the K8s scheduler is busy or out of service, if the K8s API has communication delays, or if the resource of the cluster is occupied by other jobs. Further, if the start latency increases for deploying identical *Pods*, it indicates that the computer node's CPU and Memory utilization is approaching the limitation. In this case, the scheduler may consider load-balancing the coming required SFC to another available compute node. By measuring those environment variables, the network operator would be able to debug whether the delay of deployment is due to the hardware resource limitation or because of the distortional communication between the K8s API and the K8s scheduler.

Moreover, some of the hardware parameters are also useful when analyzing the SFC performance such as the CPU temperature, and other I/O devices on the host. Another aspect of the performance evaluation for SFC deployment is the network performance. Unlike the hardware resources associated with the host, the network states of the container mainly depend on the overlay network devices used by SFC traffic. In other words, it usually involves evaluating the container interfaces. Some of the basics are the total network device numbers, the name of interfaces, and the traffic received or transmitted by every interface located at individual *Pods*. Meanwhile, additional environment variables would be counted to cross-validate the network constraints by measuring QoS-related metrics such as the services latency, network throughput, HTTP request error rate, and the trend of request delays. By

combining the real-time and continuous hardware and network metrics collection with a proactive monitoring system, the network operator would not only have a basic idea of the testbed constraints, but also have an intuitive understanding of the deployed container-based SFC.

3.3.2 Reliability of SFC

Reliability and robustness have always been some of the most critical concerns of any network developer. It usually involves a highly-available cluster, preventing the services from compromission, assigning accessibilities based on roles, encrypting traffic with different transport protocols, etc. The reliability of a cloud-native container-based SFC deployment can benefit from those aforementioned features. However, from the reliability perspective, the thesis focuses on:

1. Validating the traffic flows and ensuring that the traffic has been steered through the SFC path in the pre-defined order. Benefiting from the cAdvisor, network operators can monitor the traffic passthrough V-wires. The collected data can be used to validate the performance of the SFC. Even though the Prometheus system cannot analyze the content of the traffic, by comparing the ingress and egress traffic in every V-wires, it would be able to show the consistent pattern of the SFC traffic coming and leaving from one end to another end of a V-wire.

2. Ensuring the functionality provided by SFC is unchanged against unpredicted service interruptions including accidental service fault, resource outage on a computer node, etc. The fault detection and recovery mechanism provided by the testbed enhance the robustness and reliability of the container-based SFC in the Kubernetes environment. The Kubernetes by default constantly checks the status of the *Pods* developed as *Development* or *DaemonSet* using healthiness and liveness probes and

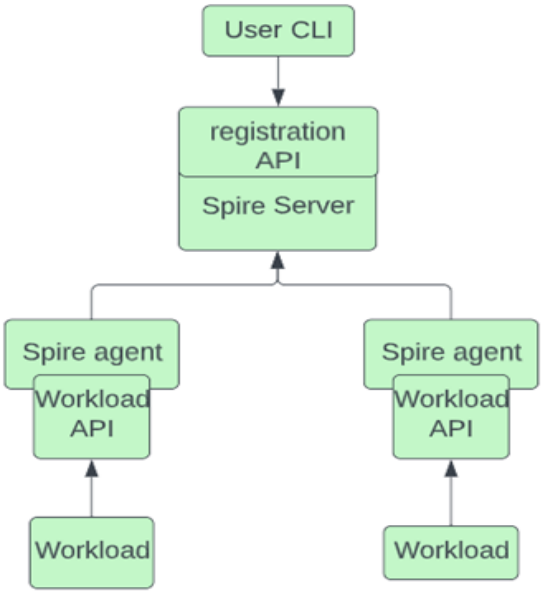


Figure 3.4: The structure and workload of SPIRE components [10]

recursively re-deployed any unhealthy *Pod* to an optimal location. However, the integrity of the SFC path is compromised if any of the *Pod* consumed by SFC is re-deployed because the connection between the re-deployed function and its neighbor functions is not resumed after the *Pod* restart.

Our testbed provides the SFC-wise connection healing function after *Pod* restart to enhance the reliability of the SFC. By leveraging the NSM framework, the restart *Pod* can be injected with new V-wires and its corresponding CNIs to resume the previous SFC connection without interrupting the rest of the deployment. The rest of *Pod* in that SFC does not need to be re-deployed to accommodate the changes. Most importantly, this healing feature also applies to the restart of the NSM control plane components.

3. The additional security has been guaranteed by employing the Zero Trust Architecture within the NSM framework. Fig. 3.4 demonstrates the SPIFFE architecture used in this thesis. It provides extra reliability to the network by dynamically

authenticating the SFC-joint *Pods*. This architecture allows the user to get access to the cluster but only to the bare minimum to finish the job. Applications can be contained within a highly purposed namespace that only involves the necessary component. Hence, the damages can be contained if the workspace is compromised. As mentioned before in Section 2.4, this architecture is implemented with the support of the SPIRE agent/server that running on the NSM control plane. Hence, the workload in NSM-based SFC is authenticated by the same Verifiable Identity Documents (SVIDs) that were issued and recorded by the SPIRE agent and server. Network services cannot participate in the SFC if it does not have the same SPIFFE token. A SPIRE server acts as a signing authority to issue identities to a set of workloads via agents [10]. It also maintains a registry of workload identities and the conditions that must be verified for those identities to be issued. SPIRE agents need to be installed on every node that workload runs, and it exposes the SPIFFE Workload API locally to workloads.

Chapter 4

System Implementation

This chapter explains the detailed implementation process along with different implementation choices. The rest of this chapter introduces the related management tools, experiment environment, including hardware specifications and software versions, details about NSM and Prometheus implementation in the Kubernetes environment, custom-built docker container images used for video streaming SFC, and the process of chaining services in detail.

4.1 Kustomization

It is worth noticing that the creation of Kubernetes objects and network policy are implemented by applying the YAML configuration file which is standard in industry. To set up the SFC testbed and chain the services together, *Pods* and corresponding *Services* need to be created in the Kubernetes cluster using YAML files. As mentioned before in Section 2.3.3, *Pods* can be managed in many formats including *DaemonSet* and *Deployment*. By using *Deployment* and *DaemonSet*, the network operator can describe the desired state for a *Pod* and leave the management of *Pods* to the

Kubernetes controller that can allocate a suitable node for the *Pods* to run based on the required resources, the schedule of the node, and other criteria. Hence, all *Pods* used in this thesis were deployed as *Deployment* or *DaemonSet*. It is worth mentioning that the management of the *Deployment* and *DaemonSet* was through a Kubernetes built-in management tool called *Kustomization*.

The *Kustomization* is a configuration management solution that leverages layering to preserve the base settings of the application by overlaying declarative YAML artifacts called patches layer which selectively override default settings without changing the original files. *Kustomization* makes the base layer YAML file reusable across all the environments while allowing adding different twists into the deployment by declaring different patch files. Fig. 4.1 is an instance of the *Kustomization* YAML file used in this thesis to deploy all the components used by the video streaming SFC. From the YAML file in Fig. 4.1, line 2 and line 3 indicate that this YAML file uses *Kustomization* API to manage the Kubernetes deployment in both the base layer and patch layer. From Line 7 to line 9, the *Kustomization* indicates some resource files. The resources files are nothing but regular Kubernetes YAML files that does not need the multi-layer structure. Meanwhile, the base layers located from line 11 to line 14 indicate the paths to some deployable Kubernetes objects. As the name implies, the base layer is used as the plain configuration for *Pod*, *Service*, and other Kubernetes objects which can be altered by patch files that are written between line 16 to line 18. It is worth noting that each base layer has to contain another local *Kustomization* file so that the *Kustomization* API can manage the layer structure through the nested *Kustomization* file. For example, line 12 uses the local system directory called "$\../../../apps/nsc - kernel$" as a base layer. Inside this base layer, there are two files: kustomization.yaml and nsc.yaml. The kustomization.yaml is

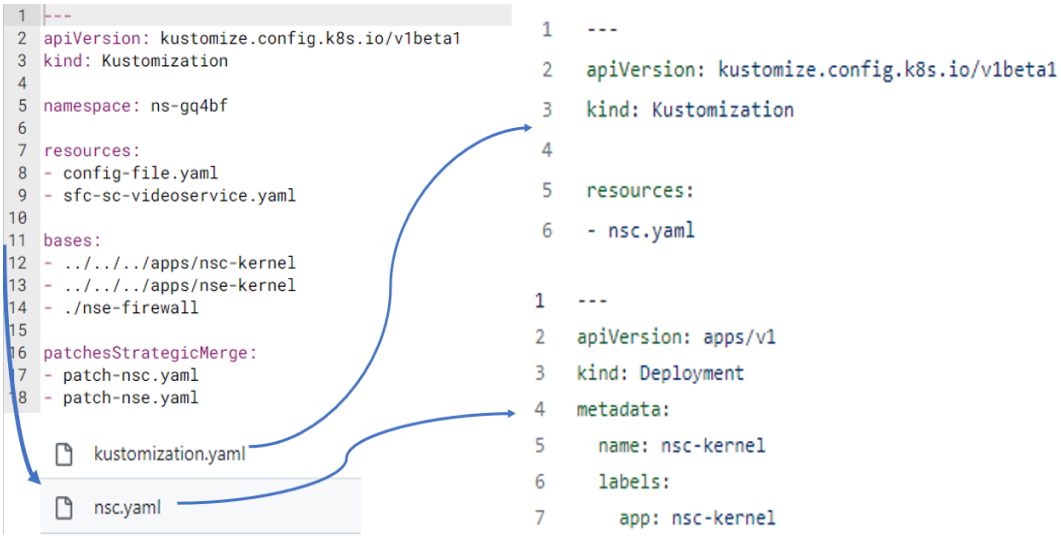


Figure 4.1: The nested Kustomization structure for deploying an instance of SFC

simply used to deploy the resource nsc.yaml file as demonstrated in Fig. 4.1. The nsc.yaml files then contains all the detailed configuration regarding the NSC object. That's how the *Kustomization* management tool achieves the nested layer structures by using resources, bases, and patches.

4.2 Environment

The experimental environment consists of multiple components, each is elaborated as follows.

Testbed. The cluster is composed of 1) one master node which is running on a server that has a 40-core CPU (Intel Xeon E5-2650 @ 2.30) with 1GbE NIC (Intel I350); 2) one worker node which resides on a server that has a 40-core CPU (Intel Xeon Silver 4114 @ 2.20GHz) with one 1GbE NIC (BRCM 5720). Both servers are running Ubuntu 16.04.5 with Linux Kernel version 4.15.0-156-generic. Two servers are sufficient for this thesis to demonstrate the SFC deployment across a multi-node

Kubernetes cluster.

Kubernetes. The latest version of Kubernetes was used. The client version for Kubernetes was 1.24.1 and the server version for Kubernetes was 1.24.1 at the time K8s was deployed on the testbed. The Kubeadm tool was used to create multi-node clusters in Kubernetes over two physical servers. The Kubernetes cluster can be deployed using many installation tools such as Kind, MicroK8s, MiniKube, and Kubeadm during the testing process. Eventually, Kubeadm was chosen due to the flexibility gained from the bare minimal default network configurations. In addition, the corresponding version of kubectl was installed. The kubectl is the user CLI that allows users to communicate with the Kubernetes API. The detail of deploying and customizing Kubernetes is out of the scope of this thesis.

Container Runtime. The container runtime runs on each node in the cluster so that containers can run inside *Pods*. Kubernetes releases before v1.24.0 included direct integration with Docker Engine, using a component named Dockershim which has been removed from the new Kubernetes update v1.24.1 and replaced by cri-dockerd for the reasons mentioned in section 2.3.2. It is worth mentioning that even though the Docker Engine still can be used as the container runtime with cri-dockerd in the Kubernetes environment, the container-level metrics cannot be collected through the cAdvisor. Since the Kubernetes cluster version in the proposed testbed is v1.24.1 in this thesis, the container runtime is containerd://1.6.6. and with integrated CRI.

Networking. There are two aspects of networking in the proposed cloud-native container-based SFC testbed on the Kubernetes cluster. The WeaveNet v2.8.1 provides the DNS *Service*s and a data plane for non-SFC communication. The separated SFC data plane is provided by the latest NSM v1.4.0 infrastructure. Many versions of the NSM releases was tested. At first, the tested version was v0.2.0 which is the earlier

release even before the first official release. The v0.2.0 release supports basic SFC implementation, but the framework can only be implemented with Helm chart version 2. With the development of NSM, the later releases improved the usability, stability, and reliability of the SFC. In addition, NSM has added more features and capabilities such as supporting different types of payloads (IP and Ethernet), latency reduction, and topology-aware scale. The later release of the NSM doesn't rely on the Helm to manage its deployment, all the binaries can be downloaded directly from the NSM GitHub repository [21]. This thesis focused on using the latest release v1.4.0 with our custom-built containers to develop an SFC.

Prometheus. The testbed used the community-maintained helm chart kube-prometheus-stack [6] release-0.11 to deploy the prometheus-operator v0.57.0 which contains (i) a Prometheus Operator which can simplify and automate the configuration of a Prometheus-based monitoring stack for Kubernetes clusters, (ii) a Prometheus server to store the collected metrics and run the Prometheus API server, (iii) an alert manager to set an alarm based on different conditions, (iv) NE on each node to monitor the environment of the two physical servers, (v) cAdvisor on each node to monitor container-related information, (vi) Grafana visualization web interface. The installation was implemented using the latest Helm 3.

Video Streaming Software. This thesis created a video streaming SFC to emulate the content distribution network for proof-of-concept. In addition, the video streaming service is latency and quality-critical and suitable for testing purpose. Specifically, this thesis used the Nginx web server, Nginx-RTMP module, and the FFmpeg video editing software. The version of the Nginx-RTMP module used in the video streaming container is 1.2.2. The Nginx web server version is 1.21.0 and the FFmpeg version is 4.4 on both video streaming and video compression containers.

The FFmpeg is a complete, cross-platform open-source audio and video framework that can perform decode, encode, transcode, mux, demux, and stream [64]. The Nginx is another open-source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. Nginx can also function as a proxy server for streaming video in many formats via Real-Time Messaging Protocol (RTMP) module [65].

4.3 NSM Framework

The basic setup for NSM included (i) one or multiple NSMgrs depending on the number of nodes, (ii) one Network Service Forwarders for each node, (iii) one *NSM Registry*, and (iv) one *Admission-Webhook*. In brief, the NSMgr plays the role of the SFC framework controller. It controls the communication between NSM components and the establishment of the SFC tunnel. Network Service Forwarder is the carrier for the SFC traffic. In this thesis, it was a VPP-type traffic forwarder due to the high performance of the VPP forwarder which can leverage the acceleration of the networking hardware such as NIC. The VPP forwarder is the only NSM component that is deployed in the data plane. The *NSM Registry* oversees registering the NSE which will be consumed by SFC later. The *Admission-Webhook* will inject the *init-container* for each NSC and NSE to create appropriate V-wires for SFC traffic. The function of each component is explained in detail in Section 2.4.

The NSM control plane and data plane components were deployed as *Pods* running inside the Kubernetes cluster. Some of them were running as *DaemonSet*, such as NSMgr and Network Service Forwarder because a *DaemonSet* ensures that all Nodes run a copy of a *Pod*. While others were running as *Deployments* like *NSM Registry*

and *Admission-Webhook*. The deployment of the NSM elements was managed by *Kustomization* YAML files which can be accessed directly from the Community GitHub [21]. Hence, only the high-level *Kustomization* file is demonstrated in Appendix D.1 to give a brief view of the NSM implementation. As shown, there are four base directories for four control plane elements, respectively. Inside each base directory, there are several resource files managed by a local *Kustomization* file. For example, line 8 in the *Kustomization* file shows a base directory "../../../apps/nsmgr" which contains other four resource files and one local *Kustomization* file as shown in Appendix D.2 to achieve the nested deployment layers. However, the details of the NSM manifest YAML files are not in the scope of the thesis, this work focuses on using the existing framework to establish SFCs in the Kubernetes cluster. Hence, this thesis provides the details of the manifest YAML files for deploying the customized video streaming SFC in the later section.

4.4 SPIRE Authentication

Authentication is important for achieving a reliable working environment. All the SFC workload created by the NSM framework needs to be authenticated to ensure that resources are reserved by the registered SFC. This had been done by implementing the SPIRE project which securely issues SVIDs to workloads and verifies the SVIDs of other workloads. The SPIRE architecture composes of (i) a SPIRE server and (ii) one or more SPIRE agents running on its dedicated namespace. The SPIRE server is the central controller for managing and issuing all identities in its configured SPIFFE trust domain. It holds registration entries that define the selectors who decide when a given SPIFFE ID should be granted. It also utilizes node attestation to automatically

```
kubectl exec -n spire spire-server-0 -- \
/opt/spire/bin/spire-server entry create \
-spiffeID spiffe://example.org/ns/spire/sa/spire-agent \
-selector k8s_sat:cluster:nsm-cluster \
-selector k8s_sat:agent_ns:spire \
-selector k8s_sat:agent_sa:spire-agent \
-node
```

Figure 4.2: The Kubernetes command for registering namespace with Spire server

```
openssl req -x509 -newkey rsa:4096 -keyout "bootstrap.key" \
-out "bootstrap.crt" -days 365 -nodes -subj '/CN=localhost' 2>/dev/null
```

Figure 4.3: The Linux command for creating a certificate and key used by the Spire server

validate agents'identities and produces SVIDs for workloads when requested by an authenticated SPIRE agent. The agent runs on every working node where the workload runs to cache the assigned SVID used by the workload. A registration entry includes a SPIFFE ID, a set of selectors, and a parent ID. The command in Fig. 4.2 presents an example of creating a registration entry for workload using the kubectl command. As the command demonstrates, the SPIRE server created the entry for the trusted domain using the properties such as SPIFFE ID, the name of the cluster, the namespace of the workload, and the service account the agent is running under.

The SVIDs are created based on the upstream authority such as the key and certification combination created by the network operator. Then, the key and certificate files can be mounted to the SPIRE server by *ComfigMap*. The command in Fig. 4.3 demonstrates a method to create the key and certificate file in the Linux system.

Similar to the NSM deployment, the manifest YAML file for deploying the SPIRE project was managed by the *Kustomization* tool. This *Kustomization* file contains

several resource files that create the namespace, service account, ConfigMaps, etc. to support the running environment of the SPIRE server and agents. This *Kustomization* file can be found in Appendix D.3 but the rest of the resource files used by the *Kustomization* file is not presented since the SPIRE project is open source and the rest of the YAML files can be accessed in the NSM GitHub repository [10].

4.5 Deploying the Prometheus Monitoring System

Kubernetes monitoring is a method of examining and reporting the health status of the cluster components, including the utilization of cluster resources such as CPU, memory, and storage. More importantly, the challenges with container-based SFC come from the managing of high availability, performance, and deployment of service functions. The Prometheus provides a query language and a robust data model that provides detailed and actionable metrics. As mentioned in Section 2.5, the following Prometheus components were deployed using Helm package manager: (i) one Prometheus server, (ii) one Prometheus Node Exporter for each node, (iii) one Prometheus operator, (iv) one Prometheus alert manager, (v) one Grafana web server, and (vi) one cAdvisor for each node.

The Prometheus components were deployed using *Deployment* to manage the life cycle of the metrics database and data collection agents in each Kubernetes node. To manage the configuration information of these *Pods* uniformly, the *ConfigMaps* were used in Kubernetes to define and manage these configurations. Fig. 4.4 demonstrates the basic configuration of the Prometheus monitoring system such as data collection intervals for the global pulling mechanism, evaluation intervals for evaluating rules of data collection, alert generation rules, and the port number of the exposed Prometheus

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
data:
  prometheus.yml: |
    global:
      scrape_interval: 15s
      evaluation_interval: 15s

    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
        - targets: ['localhost:9090']
```

Figure 4.4: The ConfigMap used by the Prometheus server to set basic rules for metrics collection

service. Similar to the video streaming service, the exposed time series dataset needs to be forwarded outside of the Kubernetes cluster for user access by creating the *NodePort* type Kubernetes *Service*. The YAML files for creating the Prometheus Deployment and Services are included in Appendix F.1 and F.2.

After finishing the configuration for the Prometheus monitoring system, the most important further step is to collect metrics that closely relate to the performance and reliability of the SFC. To that end, a Python3 script has been developed to use Prometheus API to automatically pull data from the TSDB and saved it to a separate data file such as the Comma-Separated Values (CSV) file. Based on the papers [19], [3], [49], [66], [57] investigated in the literature review, the list of information collected that is most useful includes: : (i) the latency for *Pods* initialization, (ii) the host's network device name and speed, (iii) the network throughput during the sample period, (iv) the CPU temperature, (v) the CPU and memory utilization, (vi) the HTTP request delay, (vii) the HTTP request error rate, (viii) the total number of HTTP request and its increasing rate. By leveraging the powerful PromQL built-in

functions and aggregate operations, it becomes possible to calculate the traffic pattern and predict the trends in real-time. The Python script can be found in Appendix F.3.

4.6 Customized Container-based Service

The purpose of our cloud-native SFC testbed is for developers to implement their real-life applications in a data center concretely and reliably. This thesis provides the details of the container-based video streaming SFC to emulate a video content distribution network in the data center. I decided to implement this specific use case since the video streaming service is highly on-demand and QoS-critical, especially after the global pandemic. The video streaming scenario emulated a mobile client who would like to stream a video but facing network congestion. Hence, the video size should be reduced first before the video is broadcast. To solve this issue, the client can use the proposed testbed through the web interface to establish the video streaming service. After the user finishes configuring the SFC and sent the request using the web UI, 3 *Pods* will be created and chained together to provide video streaming, firewall, and video compression services respectively at the Kubernetes cluster.

Fig. 4.5 illustrates the structure of this video streaming SFC use case. The first *Pod* is the video streaming *Pod* where the HTTP Live Streaming (HLS) services ran. A user request with the original video was sent from the first *Pod* passing the firewall *Pod* and reached the destination video compression *Pod*. After the video compression process was finished, the compressed video was sent back through the reverse route to the video streaming *Pod* passing through the firewall *Pod* again. All the customized containers running in *Pods* are created based on Dockerfile for its standardized environment, responsive deployment, and global scaling features. The

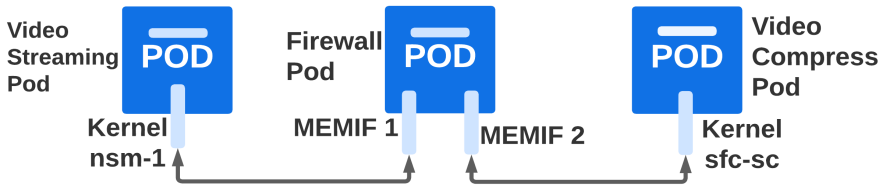


Figure 4.5: The structure of the video streaming SFC

Dockerfile is a text file that contains all the commands that will be called on the command line when the docker container is built [67]. The docker container engine will run the Dockerfile to assemble the docker image. In other words, the Dockerfile is the standard method for the developer to configure and built a docker container image. The following sections explain each customized docker container in detail, including the Dockerfiles, modules, and programming scripts loaded into the containers.

4.6.1 Video Streaming Container

The video streaming container is located at the start of the chain. The function of this docker container image is to stream video which composes of two services: (i) real-time video streaming and (ii) displaying the video on a web browser. The video streaming was achieved by using Nginx-RTMP and FFmpeg module. On the other hand, displaying the streaming video on the web interface supported by a regular Nginx webserver. One of the well-known streaming protocols is the HLS. HLS is an adaptive streaming technology allowing users to stream video that is tailored to the user's device and network conditions for the best streaming performance [65]. Nginx-RTMP is the Nginx module that integrates RTMP and HLS streaming to the Nginx web server.

A Dockerfile compiles and installs the Nginx web server, Nginx-RTMP module,

and FFmpeg from the source with default settings for HLS based on an Alpine Linux OS. The overall container image size is 64.41 MB which is publicly available from my Docker Hub repository: ziqiangwang0417/broadcastimage. The Alpine Linux is an open-source lightweight security-oriented Linux distribution for virtual networks and services [63]. One advantage of using Alpine Linux as the base image of a web server is that it is not only secure but also occupies a small number of hardware resources. The video streaming container contains the following modules with specific versions:

- Nginx 1.21.0: mainline version compiled from source
- Nginx-RTMP-module 1.2.2: compiled from source
- FFmpeg 4.4: compiled from source
- Default HLS settings

The Dockerfile for this video streaming container is in Appendix A.1. After the video streaming *Pod* started, the HLS server leveraged the Nginx-RTMP module for video streaming services by taking the input video uploaded from FFmpeg software. Hence, additional configuration was needed to get the HLS service working properly with FFmpeg. This process was done by modifying the Nginx configuration file which has placeholders for HLS features. The HLS service will read and obey the configuration written in this Nginx configuration file. The details of the configuration file can be found in Appendix B.1. As indicated in the configuration file at line 18, shown as following: *"exec ffmpeg -i rtmp://localhost:1935/stream/name -c:a libfdk_aac -b:a 128k -c:v libx264 -b:v 2500k -f flv -g 30 -r 30 -s 1280x720 -preset superfast -profile:v baseline rtmp://localhost:1935/hls/name_720p2628kbs"*, the HLS ran the Nginx-RTMP module on port 1935 with a 30-frame rate and 1280x720 resolution. Live video fragments are temporarily saved as m3u8 files in a pre-defined local directory.

Moreover, one Python script was loaded into this container by the Dockerfile to

automate the video streaming process including waiting for the video to come back from the video compression container and uploading the compressed video to the streaming server. The Python script can be found in Appendix C.1.

4.6.2 ACL-based Firewall Container

The second *Pod* in this topology is an Access Control List-based (ACL) firewall container that allows or denies traffic based on certain criteria. It is the first stop for every network package that leaves the video streaming *Pod* to the video compression *Pod*. In this ACL-based firewall implementation, network protocol and port number were used as criteria to alter the traffic. The firewall *Pod* was implemented using Golang-based VPP interfaces in its ingress and egress port to filter the incoming traffic. Some of the important GoVPP modules are listed below. The VPP binapi-generator is a VPP API library that supports abundant functions which include an ACL-based firewall with the GoVPP module [68]. The details of the Dockerfile used to build this firewall docker container are demonstrated in Appendix A.2. The container image size is 75.29 MB.

- VPP binapi-generator v0.4.0-dev
- VPP 21.06.0-9 ga41932662

In this SFC implementation, the firewall allowed ICMP traffic on any port between 0 to 65535 for both incoming and outgoing traffic just for testing its connectivity between neighbor *Pods*. Meanwhile, TCP incoming traffic was not constrained by the port number, but the TCP destination port only allowed ports 8080 and 80, because those two ports ran the Nginx web servers. The rest of the TCP traffic was blocked by the firewall since no service ran on those ports. The traffic configuration file used by the firewall was a Kubernetes *ConfigMap* object which can be found in Appendix

B.2. The *ConfigMap* file was attached as *Volume* and mounted by the firewall *Pod* when the *Pod* starts to run. By adopting *ConfigMap*, different traffic rules can be written in various *ConfigMap* files making the firewall container suitable and reusable for different use-case scenarios. Furthermore, the rules in *ConfigMap* can be easily changed without interrupting other components of the firewall which makes the firewall container more reliable.

4.6.3 Video Compression Container

The video compression *Pod* is located at the end of the chain. It monitored any incoming video files in a certain directory and compressed the video file into a smaller size. The video reduction function had been carried by FFmpeg software and automated by Linux Shell script and Python3 scripts depending on the size of the required video. To sent the compressed video back to the client *Pod* for video streaming, the Nginx web server in this container were used as a static file server, so that the compressed video could be distributed by whoever has access to the Nginx server at port 80. Hence, the Dockerfile of the video compression container contains the following required modules to complete the work:

- FFmpeg 4.4: compiled from source
- Nginx 1.21.0: mainline version compiled from source

The details of the Dockerfile can be found in Appendix A.3. The container image size is 807.4 MB since it uses Ubuntu as the base image. The container image is publicly available from my Docker Hub repository: ziqiangwang0417/reductionimage. Similar to the automation design in video streaming *Pod*, a Linux Shell script and a Python script are embedded with the container to automate the video compression function. Those programming scripts were compiled in the Dockerfile when the docker

container image was created so that it ran automatically when the *Pod* starts. The Python script used FFmpeg tools to compress the video, process the video, and send it to a local directory for further inquiry. The details of the scripts are in Appendix C.2.

4.7 Deploy the Customized Network Service Pods

All the necessary components, such as the Kubernetes cluster, NSM framework, Prometheus system, and customized container images, were ready for creating the video streaming SFC. Based on the topology mentioned in the last section, a video streaming SFC composed of 3 *Pods* wrer created with corresponding *Services*. This chapter gives detailed information regarding the implementation of the SFC.

4.7.1 Kustomization File for Video Streaming SFC

This *Kustomization* file that manages the video streaming SFC can be found in Appendix E2. As indicated in line 5, shown as following, *"namespace: ns-gq4bf"*, a namespace was created to contain the *Pods* and *Services* for this dedicated video streaming SFC so that the SFC-belonging deployments were isolated from the rest of the cluster. Most importantly, the SPIFFE authentication system assigns and examines SPIFFE tokens used by this namespace. This brought extra reliability and integrity to the system from the perspective of resource management. It also ensured the cluster resources were always assigned to the demanded verified workload.

As indicated in lines 8, 9, and 10 of the *Kustomization* file, there are three resources YAML files were deployed with the video streaming SFC for different purposes. Those manifest files can be found in Appendix B.2, E.3, and E.4, respectively. The YAML

file, "config-file.yaml" was used as the configuration file for the firewall network services as mentioned in Section 4.4.2. On the other hand, the second resource file, "sfc-sc-videoservice.yaml", declared a *NodePort*-type *Service* that exposed the video streaming traffic outside the Kubernetes cluster so that the user can access the video on a web browser. As mentioned before in Section 4.4.2, the Nginx video streaming service ran on port 8080 was mapped to a host port 30120 through the *NodePort Service* defined in this manifest file. By accessing the Nginx server IP and the port number, the streaming video was broadcast in an HLS JavaScript video player in a web browser.

Finally, the last resource YAML files, "sfc-sc-chain.yaml", declaresd the SFC traffic rules for the video streaming services. First couple of lines define the file type, the name of the SFC, and the applied namespaces for this SFC deployment, respectively. Then, the following lines in the file are the traffic steering rules. As line 15 to line 17 identified, the destination selector defined the first destination in the chain which was the application labeled as the "firewall". It means that all the traffic belonging to the SFC called "sfc-sc-chain" must pass through the firewall application first before reaching any further destination in the chain. Then from line 10 to line 14, the combination of the source and destination selector defined that any traffic that left the firewall application must go to the application labeled as "nse" which is the video compression service in this use case. All the details and labels for each *Pod* are explained in the next section with the corresponding YAML manifest files.

4.7.2 Bases and Patches

As the *Kustomization* file in Appendix E.2 shows, two *Pods* were deployed based on the base and patch structure. The first base directory in line 13 (*../../../apps/nsc-*

kernel) with the first patch file in line 18 (*patch-nsc.yaml*) was used to deploy the video streaming *Pod* that located at the start of the chain. The video streaming *Pod* contained two individual containers called "nsc" and "broadcastcon" as demonstrated in the patch file "patch-nsc.yaml" shown in Appendix E.5. The first container, "nsc", was the SFC-proxy container which provided the connections between the SFF and SFC-unaware service function such as the video streaming services. The manifest YAML files of the base container "nsc" can be found in Appendix E.1. On the other hand, the patch file "patch-nsc.yaml", not only declared the combination of the two containers but also added some significant environment variables into the SFC-proxy container. From line 11 to line 13 in file "patch-nsc.yaml", an important environment variable called "NSM_NETWORK_SERVICES" was added into the SFC-proxy container. The value of this environment variable should obey a specific format for the NSM control plane to assign the correct CNI interface to the *Pod*. The format is: type of interface://the name of the SFC/the name of the interface. For example, the environment variable used in the "patch-nsc.yaml" defines that the SFC-proxy connects to an SFC named "sfc-sc-chain" using a Kernel interface called "nsm-1".

Meanwhile, the second base in line 14 (*../../../apps/nse-kernel*) and patch file in line 19 (*patch-nse.yaml*) of the *Kustomization* file defined the video compression service located at the end of the chain. The video compression *Pod* contained two individual containers named "nse" and "nginx-ffmpeg" as demonstrated in the patch file shown in Appendix E6. Again, the "nse" was the SFC-proxy container that provides the Kernel interface for the SFC traffic and carries the traffic from the services function to SFF. The manifest YAML files of the SFC-proxy container "nse" can be found in Appendix E.7. The service function container called "nginx-ffmpeg" performed the video compression service which is not the standard network service and hence

it needs to attach to an SFC-proxy container. As demonstrated in the patch file "patch-nse.yaml" from line 11 to line 19, it also added four important environment variables into the SFC-proxy container. The "NSM_CIDR_PREFIX" defined the Classless Inter-domain Routing (CIDR) IP address range used for this SFC. The "NSM_SERVICE_NAMES" defined the name of the SFC this SFC-proxy container joined. The "NSM_LABELS" defined the network service label attached to this *Pod* so that the *NSM Registry* could find the application and register it when establishing the SFC. In this video streaming SFC example, the network service label added by this environment variable is "nse" which was the same one used in the configuration file that defines the SFC traffic rules demonstrated in line 14 (*destination_selector: aap: nes*) of Appendix E.4.

Finally, in the *Kustomization* file, line 15 (*./nse-firewall*) shows another base directory for deploying the firewall *Pod*. There were three files inside this base directory as shown in Fig. 4.6. Similar to other NSC/NSE *Pods* in the SFC, the manifest file named "patch-nse-firewall-vpp.yaml" added significant environment variables to the firewall *Pod* so that it can be recognized by the NSM manager. Like the video compression *Pod*, the same environment variable "NSM_SERVICE_NAME" in line 13 defined the name of the SFC this firewall *Pod* joined. The environmental viable "NSM_LABELS" defined the network service label attached to this *Pod* as the "firewall". This label was used by the NSM manager to select the application when establishing the SFC as demonstrated in lines 11 and 17 of Appendix E.4. The firewall was configured using the "config-file.yaml" as mentioned in Section 4.7.2 and did not need an SFC-proxy container to carry the traffic. The "config-file.yaml" was mounted to the firewall through the file called "config-patch.yaml".

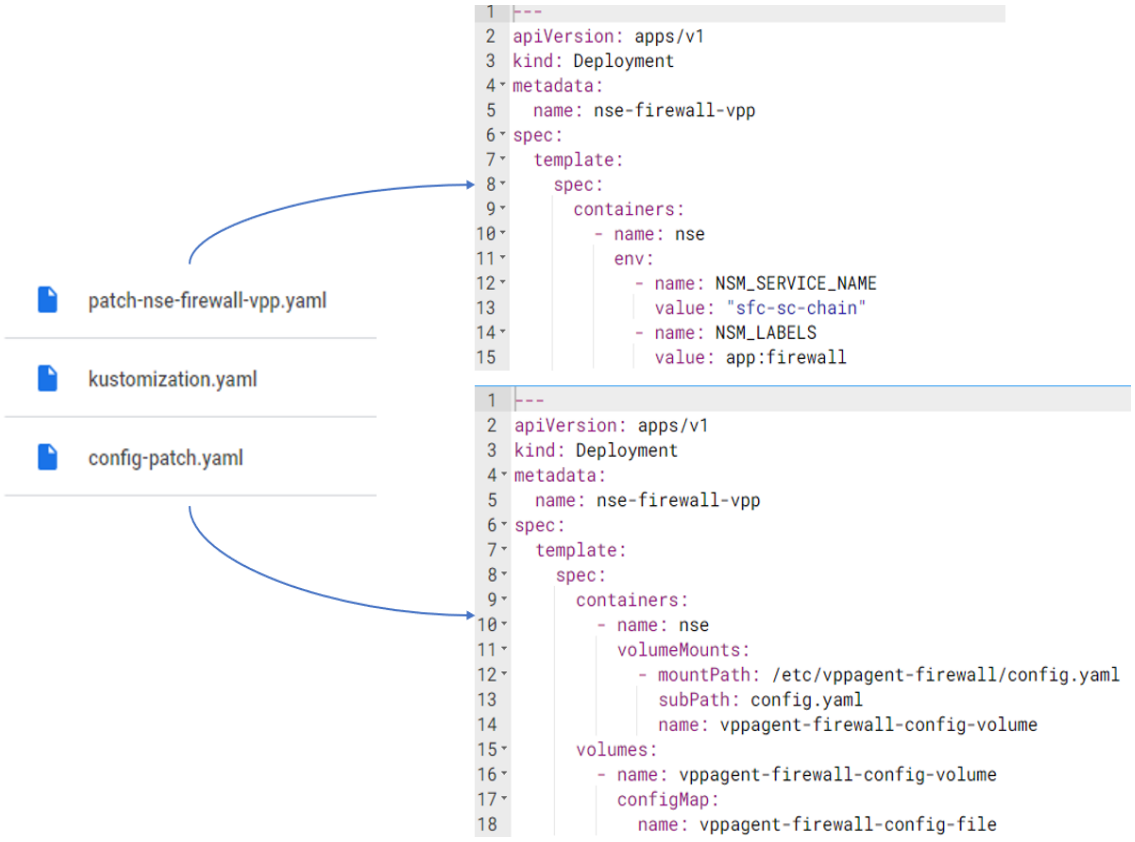


Figure 4.6: The manifests for deploying the firewall Pod used by the video streaming SFC

4.8 Chaining the Network Service Pods

The SFC demonstrated in this thesis consisted of 3 NSEs distributed in two cluster nodes. The SFC model could be represented as $NS_1(Ethernet, [NSE_1, NSE_2, NSE_3])$ since its payload type was Ethernet and consists of 3 NSEs. All the NSEs were created from the customized docker image built in Section 4.6. The $NSE_1(Kernel_1, v_1)$ located at the first node contained the function of video streaming. It had one Kernel interface that connects with V-wire #1 for sending and receiving videos. The $NSE_2([MEMIF_1, MEMIF_2], [v_1, v_2])$ was an ACL-based firewall located at the second node. It had two MEMIF interfaces which connect with V-wire #1 and V-wire

#2, respectively. The $NSE_3(Kernel_1, v_2)$ located at the end of the chain had the function of video size reduction which reduced the size of a video to ensure the user can stream video faster when facing the network bottleneck. It connected with the V-wire #2 only with one Kernel interface. The client, in this case, was integrated with the NSE_1 which makes the chain tightly organized and easier to implement.

The creation of the SFC is always started by a client sending a request and the NSMgr will search the required NSEs and communication mechanisms. During this process, a new NSE *Pod* will be registered with the *NSM Registry* if the *Pod* has the requested NSE label; also, the *Admission-Webhook* will inject the corresponding interfaces into NSC and NSEs. After the NS/SFC is declared and containers are created, V-wires will be created between $(NSE_1 and NSE_2), (NSE_2 and NSE_3)$ in the forwarder plane using either VPP traffic forwarder or Kernel traffic forwarder. It is worth noting that some of the V-wires have two different types of interfaces at the two ends.

An initial request was sent from the NSC located at the first node along with the video that would be compressed in the NSE_3 resided at the second node. Theoretically, the traffic passed through NSE_2 and reached the destination NSE_3. The traffic was steered between two physical nodes and among 3 *Pods*. Then, after the video had been processed by NSE_3, the traffic containing the compressed video was steered back from NSE_3 to NSE_1 via NSE_2. Hence, a content distribution network that uses a cloud-native SFC to distribute videos was achieved. Meanwhile, the entire workload was registered with the SPIRE agent/server to make sure that the SFC path is not compromised. Simultaneously, the Prometheus system monitored the parameters such as the *Pod* start latency, network throughput, node CPU and memory utilization, etc. The collected dataset was analyzed to validate the SFC traffic path. The result was

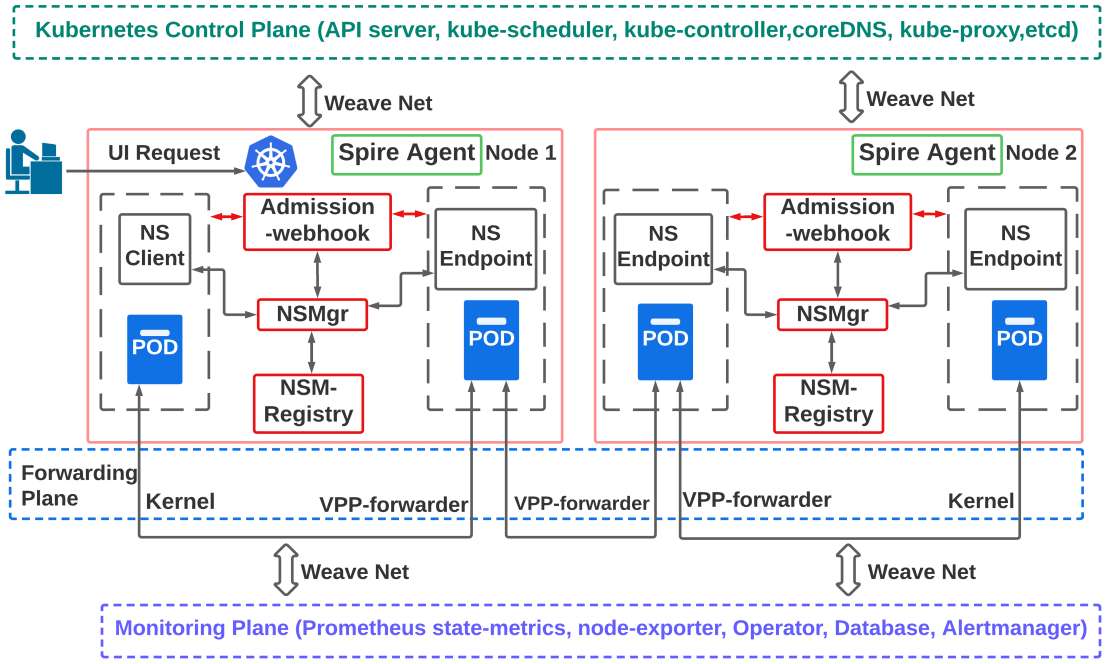


Figure 4.7: The general architecture of video streaming SFC in the Kubernetes environment

displayed in the integrated Grafana web UI. In comparison, the Prometheus and the Kubernetes traffic used the WeaveNet control plane interfaces instead of the NSM interfaces, hence the control and monitoring plane traffic does not add overheads to the SFC data plane. Fig. 4.7 demonstrates the entire SFC-based video streaming process in detail.

Chapter 5

Evaluation

The container operating system virtualization brings difficulties to the evaluation process since not many Linux Kernel tools are developed to evaluate the container system performance from the Kernel space. Especially, the containers are usually running in the isolated user space where Linux kernel tools cannot investigate. Furthermore, installing evaluation tools into each container-based microservice is time-consuming and even impossible because the container image is lightweight and tool packages may not be available. Fortunately, the Kubernetes control plane component, kubelet has integrated the cAdvisor to collect container-level metrics. Furthermore, by leveraging the Prometheus TSDB, the collected metrics are stored in the database for further access and analysis. Based on the implementation of the design presented in the previous chapters, this chapter characterizes the evaluations and the results in two different parts: performance and reliability. Together with the evaluation and the results, the employed methodologies are described in detail.

5.1 Benchmarking Methodology

To evaluate the performance of network interconnection devices, a benchmarking methodology is described in RFC 2544 [69]. Also, several specific test methods for parameters such as bandwidth, throughput, and latency defined in RFC 1242 [70] are provided together with recommended result formats. Based on those references, various tests targeting the system performance were carried out to deeply understand the potentiality of the proposed SFC testbed.

5.1.1 SFC Initialization Latency

As a baseline, the performance of the SFC testbed was evaluated in terms of the initialization latency for creating a new SFC after receiving the request from the web UI. Specifically, this experiment aims to understand how the SFC initialization latency depends on the length of the SFC path.

To figure out the relationship between elapsed time of the SFC creation and the length of SFC, two SFCs with the same functionality but different length were created. One SFC deployment has three network services as described in the video streaming example in Section 4.7. The other SFC deployment has longer path by adding three more *Pods* as middle-boxes where traffic must pass through. The brief model of the longer SFC deployment is shown in Fig. 5.1.

In this experiment, the initialization latency of an SFC is defined as the time difference between the application submission time and the time when all the requested network service *Pods* are in the ready state. Hence, the SFC initialization latency is always equal to the initialization latency of the last started *Pod* which is the video boardcast *Pod* in these two SFC deployments. The *Pod* initialization latency for each

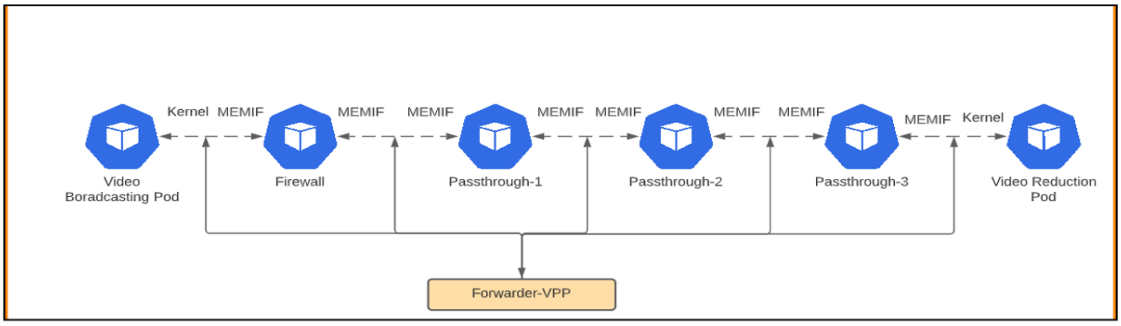


Figure 5.1: Networking architecture of the 6-Pod video streaming SFC deployment

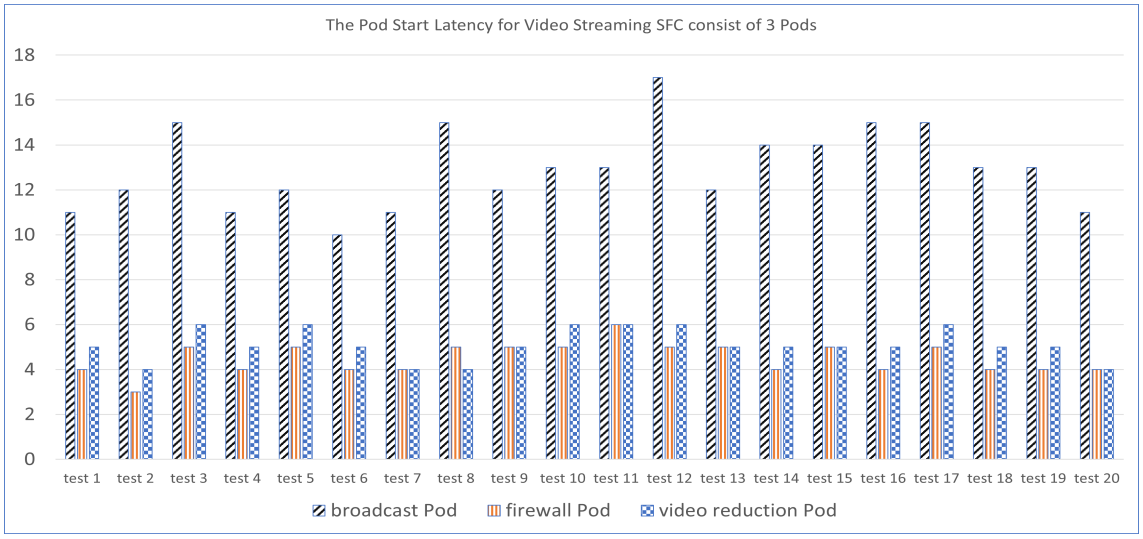


Figure 5.2: The Pod start latency for video streaming SFC with 3 Pods

SFC deployment was collected and plotted as demonstrated in Fig. 5.2 and Fig. 5.3, respectively for these two SFC deployments. As demonstrated in Fig 5.2 and Fig 5.3, each bar represents the initialization latency for a *Pod*. The initialization latency calculated in this experiment does not include the time for downloading the image from the cloud to the local container runtime. In this experiment, twenty rounds of tests had been performed for each SFC deployment. The x-axis is the traces for twenty tests and the y-axis is the *Pod* initialization latency measured in seconds.

As shown in Fig. 5.2, it is observed that the initialization latency of the SFC

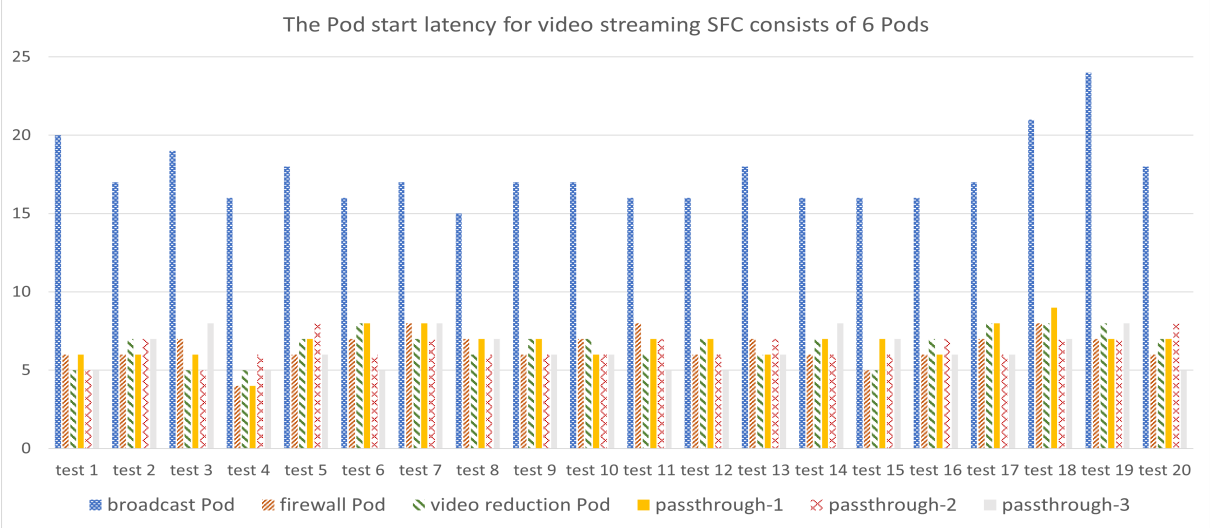


Figure 5.3: The Pod start latency for video streaming SFC with 6 Pods

varied from 10 to 17 seconds, and the average value was 12.5 seconds for the 3 *Pods* SFC deployment from twenty rounds of tests. However, as observed in Fig. 5.3, the average SFC initialization time was 17.5 seconds for 6 *Pod* SFC development. The SFC initialization latency increased 5 seconds by adding 3 more *Pods*.

The results demonstrate that the length of the SFC does not significantly affect the initialization latency of SFC as long as the container runtime caches all the container images used by network services. In fact, deploying the first *Pod*, also known as the client *Pod*, was the most time-consuming part of deploying an SFC because the client *Pod* must wait until the rest of the network service endpoints are registered with the NSMgr and deployed in the cluster, then the chaining process starts by deploying the client *Pod*. In the worst case, the 6-*Pod* SFC deployment initialization took 24 seconds, but compared to other SFC approaches using VMs with OpenStack which usually took more than 100 seconds [41], the initialization latency in the proposed container-based SFC testbed is still faster.

5.1.2 NSM Control Plane Resource Usage Performance

Even though the containerized microservices architecture brings flexibility, reliability, and scalability to the enterprise software system, moving away from the monolithic software architecture to containerized microservices architecture is bound to bring performance penalties due to increased network calls between services and container overhead. The purpose of tests in this section is to measure what system performance can be expected for the NSM control plane in a cloud-native environment. This test measures the critical metrics such as the CPU and memory usage against different NSM control plane elements during the SFC creation period. Hence, the network operator knows which NSM control plane component consumes the most system resources and understand the performance of the NSM control plane elements. During this test, 3-*Pod* video streaming SFC requests were generated to keep the NSM control plane busy.

The test results regarding the CPU and memory usage during SFC deployments can be found in Table 5.1 and Table 5.2, respectively. In Table 5.1, the column named CPU Unit Usage shows the current CPU usage for each NSM control plane component during SFC creation period. In Kubernetes, 1 CPU unit is equivalent to 1 physical CPU core, or 1 virtual core, depending on whether the node is a physical host or a virtual machine running inside a physical machine [71]. The CPU Requests column demonstrates the requested CPU when the *Pod* starts running. The CPU usage of a *Pod* cannot exceed the maximum CPU value indicated in the column named CPU Limits. Table 5.1 shows that the NSMgr can request up to 0.48 CPU, the traffic forwarder can request up to 0.53 CPU, and the *NSM Registry Pod* can only request 0.2 CPU. As shown in Table 5.1, the most CPU consuming elements were the NSMgr and the traffic forwarder (*Forwarder-VPP*) which used 143.90% and 43.29% of the

requested CPU resource, respectively. It is worth mentioning that the CPU usage can exceed the requested CPU resource as long as the CPU usage is smaller than the value indicated in the CPU Limits column.

CPU Quota					
Pod	CPU Unit Usage	CPU Requests	CPU Requests %	CPU Limits	CPU Limits%
registry-k8s	0.02	0.10	23.34%	0.20	11.67 %
nsmgr	0.12	0.28	43.29%	0.48	25.07%
forwarder-vpp	0.22	0.15	143.90%	0.53	41.11%
admission-webhook-k8s	0.00	-	-	-	-

Table 5.1: The used CPU unit of NSM control plane elements during busy time

Memory Quota					
Pod	Memory Size Usage	Memory Requests	Memory Requests %	Memory Limits	Memory Limits %
registry-k8s	21.83 MiB	40.00 MiB	54.57%	40.00 MiB	54.57%
nsmgr	198.82 MiB	140.00 MiB	142.01%	240.00 MiB	82.84%
forwarder-vpp	163.98 MiB	500.00 MiB	32.80%	500.00 MiB	32.80%
admission-webhook-k8s	13.60 MiB	-	-	-	-

Table 5.2: The memory size usage of NSM control plane elements during busy time

On the other hand, Table 5.2 demonstrates the current memory usage, requested memory and maximum memory for NSM control plane elements at column titled Memory Size Usage, Memory Requests and Memory Limits, respectively. The Memory Limits column shows that the NSMgr can request up to 240 Mebibyte (MiB) memory, the traffic forwarder can request up to 500 MiB memory and the *NSM Registry Pod* can only request up to 40 MiB memory. Table 5.2 indicates that the most memory consuming elements were still the NSMgr and the traffic forwarder which used 142.01% and 32.8% of the requested memory resource, respectively. It is also worth mentioning

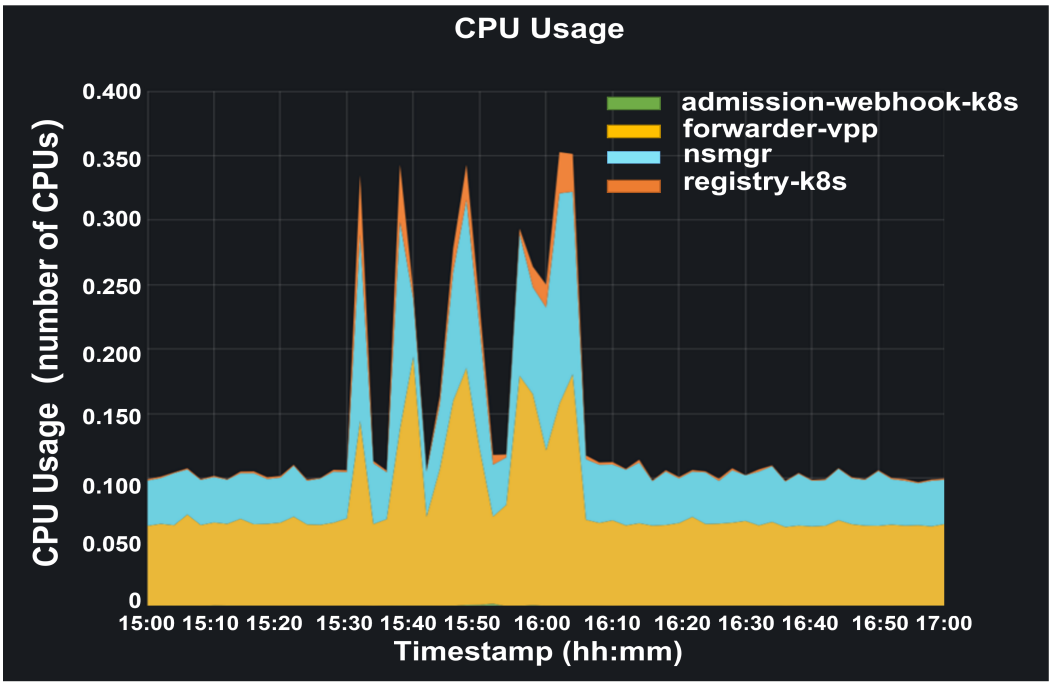


Figure 5.4: The total CPU unit usage for NSM control plane component during SFC deployment period

that even though the *NSM Registry* requested only 40 MiB memory but the memory utilization reached 54.57%.

The result of CPU and memory usage vs. time were collected and plotted as the stack graph in real-time using Prometheus web UI. The video streaming SFC use case had been deployed four times to demonstrate the CPU and memory usage performance of NSM control plane elements in both idle and busy period. From Fig. 5.4, we can observe that the total CPU usage in NSM control plane increased from roughly 0.1 CPU in idle period to 0.35 CPU during the SFC deployment period. The most CPU consuming elements were the NSMgr (blue area) and the traffic forwarder (yellow area) which indicates the same result as obtained from Table 5.1. Similarly, as shown in Fig. 5.5, during busy SFC deployment period, the total memory usage also increased from around 300 MiB to 380 MiB which is a 27% increment. Once again,

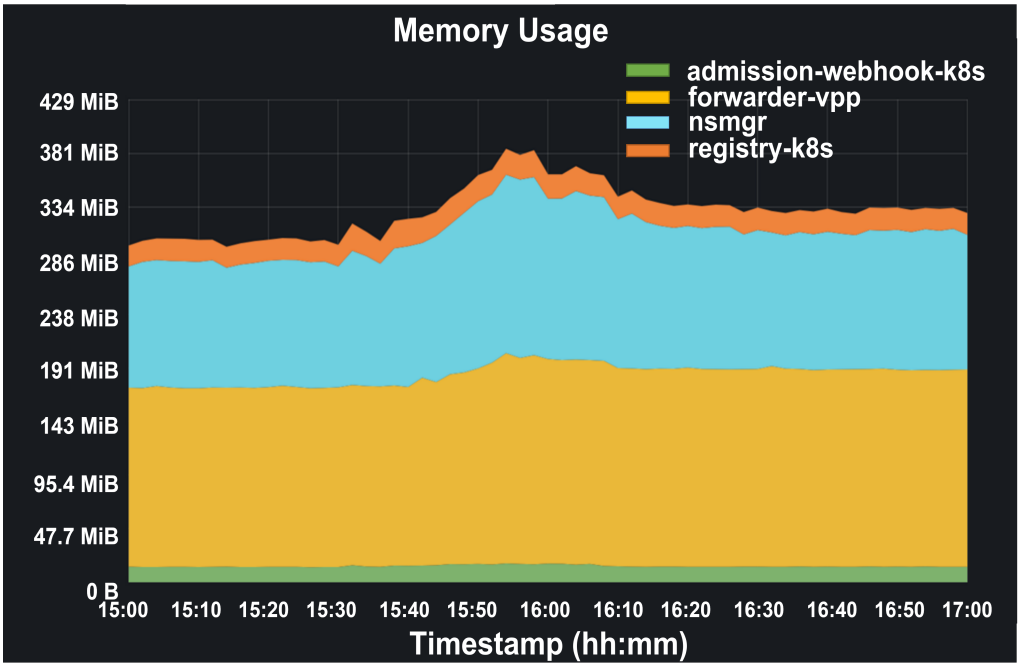


Figure 5.5: The total memory size usage for NSM control plane component during SFC deployment period

the most memory consuming elements were the NSMgr (blue area) and the traffic forwarder (yellow area). This is the same result compared to that in Table 5.2.

Moreover, it is worth noting that if there is a large amount of data flow between NSEs in the SFC, the ratio of the CPU usage will dramatically change such that the traffic forwarder uses the most CPU. Fig. 5.6 demonstrates the CPU usage of each NSM control plane element when testing the maximum network bandwidth using iperf3. The amount of CPU used by the traffic forwarder increased from 0.05 CPU to 0.5 CPU, but the CPU usage of the rest NSM components did not change, because the iperf3 container generated a large amount of traffic in the NSM network which increases the CPU usage of the traffic forwarder.

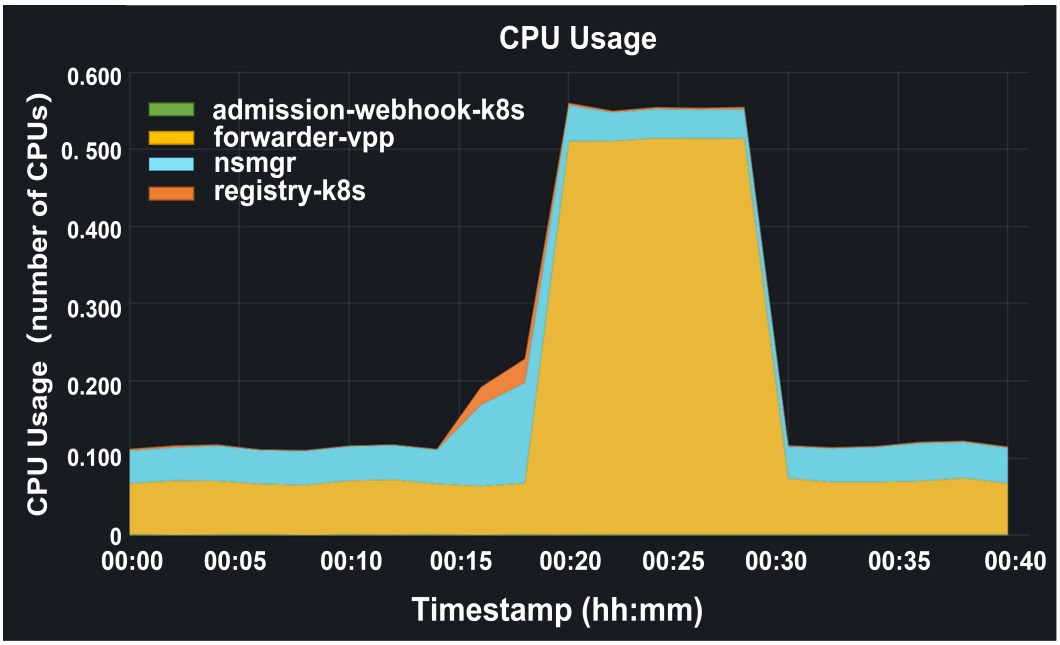


Figure 5.6: The CPU unit usage of NSM control plane elements during traffic forwarding period

5.1.3 Bandwidth Between Network Services

To have a basic understanding of the bandwidth of the SFC data plane supported by the NSM traffic forwarder. This experiment implemented the iperf3 container images which is a software tool wildly used to measure network performance. The iperf3 container uses TCP as the transport protocol by default. In this experiment, the Iperf3 sender generated TCP sessions, and sends it to the receiver which is at the other end of the chain. The TCP maximum segment size is 1460 bytes and the TCP buffer size is 128 KB. Two experiments were designed and tested to compare the different data plane bandwidths between different use cases.

The first experiment measures the network bandwidth between two default Linux Kernel interfaces provided by the WeaveNet. This test only involved an iperf3 server (receiver) and an iperf3 client (sender), hence it is the reference which represents

```
ubuntu@ubuntu:~/deployments-k8s/examples/use-cases/iperf-test-3$ kubectl exec iperf3 -n ns-f86jz -- /bin/sh -c 'iperf3 -c nse-kernel -t 60'
Defaulted container "iperf3" out of: iperf3, cmd-nsc, coredns, cmd-nsc-init (init)
Connecting to host nse-kernel, port 5201
[  5] local 10.32.0.17 port 36796 connected to 10.101.177.92 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  3.00 GBytes  25.8 Gbits/sec  153    582 KBytes
[  5]   1.00-2.00   sec  2.94 GBytes  25.3 Gbits/sec    0    582 KBytes
[  5]   2.00-3.00   sec  3.24 GBytes  27.8 Gbits/sec    0    699 KBytes
[  5]   3.00-4.00   sec  3.29 GBytes  28.2 Gbits/sec    0    715 KBytes
[  5]   4.00-5.00   sec  3.23 GBytes  27.7 Gbits/sec    0    715 KBytes
[  5]   5.00-6.00   sec  3.33 GBytes  28.6 Gbits/sec    0    715 KBytes
[  5]   6.00-7.00   sec  3.21 GBytes  27.6 Gbits/sec    0    715 KBytes
[  5]   7.00-8.00   sec  3.16 GBytes  27.1 Gbits/sec    0    839 KBytes
[  5]   8.00-9.00   sec  3.03 GBytes  26.1 Gbits/sec    0    839 KBytes
[  5]   9.00-10.00  sec  3.16 GBytes  27.2 Gbits/sec    0    839 KBytes
[  5]  10.00-11.00  sec  3.34 GBytes  28.7 Gbits/sec    0    839 KBytes
[  5]  11.00-12.00  sec  3.27 GBytes  28.1 Gbits/sec    0    839 KBytes
[  5]  12.00-13.00  sec  3.29 GBytes  28.3 Gbits/sec    0    839 KBytes
[  5]  13.00-14.00  sec  3.28 GBytes  28.2 Gbits/sec    0    839 KBytes
[  5]  14.00-15.00  sec  3.23 GBytes  27.8 Gbits/sec    0    945 KBytes
```

```
[  5]  55.00-56.00  sec  3.27 GBytes  28.1 Gbits/sec    0   3.19 MBytes
[  5]  56.00-57.00  sec  3.30 GBytes  28.3 Gbits/sec    0   3.19 MBytes
[  5]  57.00-58.00  sec  3.30 GBytes  28.3 Gbits/sec    0   3.19 MBytes
[  5]  58.00-59.00  sec  3.30 GBytes  28.4 Gbits/sec    0   3.19 MBytes
[  5]  59.00-60.00  sec  3.20 GBytes  27.5 Gbits/sec    0   3.19 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-60.00  sec   191 GBytes  27.4 Gbits/sec  153             sender
[  5]   0.00-60.04  sec   191 GBytes  27.4 Gbits/sec                  receiver

iperf Done.
```

Figure 5.7: The iperf3 bandwidth test result using WeaveNet interfaces

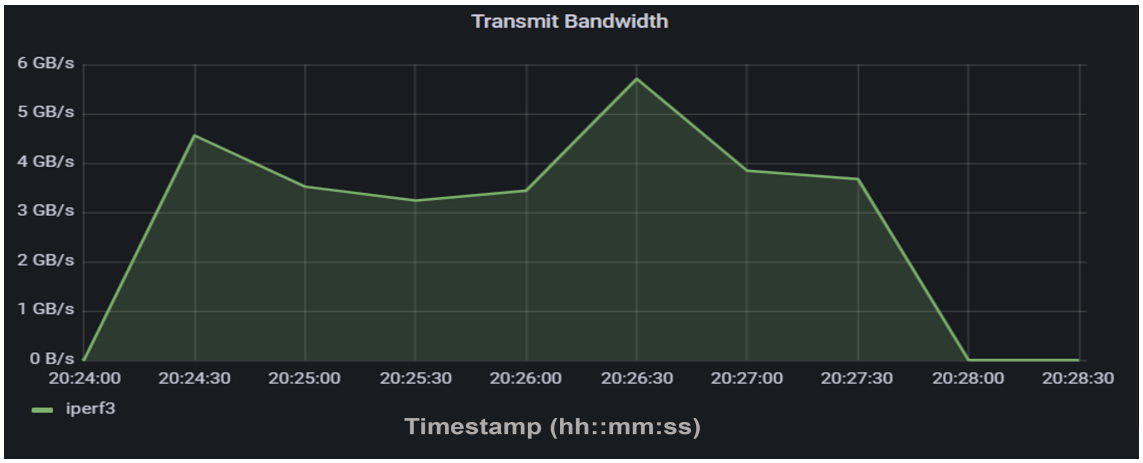


Figure 5.8: The iperf3 bandwidth test result using WeaveNet interfaces visualized in Prometheus

the bandwidth of Kubernetes' non-SFC data plane when comparing the bandwidth performance supported by the NSM interfaces. Due to the size of the table, only the first 15 rows and the last 5 rows are shown in Fig. 5.7. As demonstrated in the test results, the performance of the bandwidth was reached 27.4 Gigabits/second (Gbps) between the default CNIs provided by the WeaveNet.

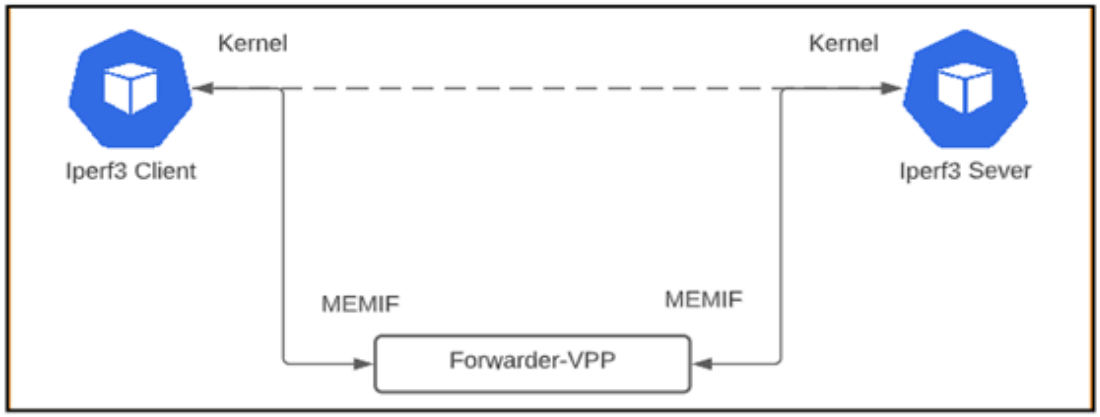


Figure 5.9: The networking architecture of the baseline network bandwidth test deployment

In a more realistic scenario, the iperf3 container cannot always be used with user applications to collect the real-time network bandwidth. Instead, the real-time network bandwidth can be collected through the Prometheus monitoring system in the proposed testbed. The traffic passing through the ingress and egress port of every NSE is queried from the Prometheus database and the average throughput is calculated relying on the PromQL build-in functions. Similar bandwidth test result was obtained from the Grafana UI as demonstrated in Fig. 5.8. The x-axis is the system time, while the y-axis is the bandwidth of the iperf3 transmit interface. The unit of y-axis is Gigabytes/second (GBps) instead of Gigabits/second (Gbps) used in Iperf3 output message. As Fig. 5.8 demonstrats, the average bandwidth of the default CNIs also reached approximately 3.4 GBps (27.4Gbps) during the test period.

The second experiment is the baseline test which only involved an iperf3 server, iperf3 client, and the NSM traffic forwarder to measure the maximum bandwidth provided by NSM interfaces. The *Pods* that ran iperf3 containers established network connections using an NSM traffic forwarder to measure the maximum bandwidth

```
ubuntu@ubuntu:~/deployments-k8s/examples/use-cases/iperf-test-2$ kubectl exec iperf3 -n ns-f8
6jz -- /bin/sh -c 'iperf3 -c 172.16.1.100 -t 200'
Defaulted container "iperf3" out of: iperf3, cmd-nsc, coredns, cmd-nsc-init (init)
Connecting to host 172.16.1.100, port 5201
[  5] local 172.16.1.101 port 46452 connected to 172.16.1.100 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec   276 MBytes  2.32 Gbits/sec    5   2.21 MBytes
[  5]   1.00-2.00   sec   266 MBytes  2.23 Gbits/sec   11   1.62 MBytes
[  5]   2.00-3.00   sec   224 MBytes  1.88 Gbits/sec    0   1.62 MBytes
[  5]   3.00-4.00   sec   184 MBytes  1.54 Gbits/sec    0   1.93 MBytes
[  5]   4.00-5.00   sec   209 MBytes  1.75 Gbits/sec    1   1.84 MBytes
[  5]   5.00-6.00   sec   158 MBytes  1.32 Gbits/sec    0   1.99 MBytes
[  5]   6.00-7.00   sec   238 MBytes  1.99 Gbits/sec    1   2.27 MBytes
[  5]   7.00-8.00   sec   180 MBytes  1.51 Gbits/sec    0   2.43 MBytes
[  5]   8.00-9.00   sec   240 MBytes  2.01 Gbits/sec    0   2.60 MBytes
[  5]   9.00-10.00  sec   235 MBytes  1.97 Gbits/sec    1   2.75 MBytes
[  5]  10.00-11.00  sec   212 MBytes  1.78 Gbits/sec    2   2.88 MBytes
[  5]  11.00-12.00  sec   234 MBytes  1.96 Gbits/sec    1   2.99 MBytes
[  5]  12.00-13.00  sec   240 MBytes  2.01 Gbits/sec    8   3.01 MBytes
[  5]  13.00-14.00  sec   274 MBytes  2.30 Gbits/sec    6   3.03 MBytes
[  5]  14.00-15.00  sec   201 MBytes  1.69 Gbits/sec    8   3.03 MBytes
   ⋮
[  5] 195.00-196.00 sec   169 MBytes  1.42 Gbits/sec    0   2.81 MBytes
[  5] 196.00-197.00 sec   229 MBytes  1.92 Gbits/sec    0   2.82 MBytes
[  5] 197.00-198.00 sec   189 MBytes  1.58 Gbits/sec    0   2.88 MBytes
[  5] 198.00-199.00 sec   210 MBytes  1.76 Gbits/sec    3   3.00 MBytes
[  5] 199.00-200.00 sec   175 MBytes  1.47 Gbits/sec    7   3.03 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-200.00 sec  42.8 GBytes  1.84 Gbits/sec  841             sender
[  5]   0.00-200.08 sec  42.8 GBytes  1.84 Gbits/sec                  receiver
```

Figure 5.10: The iperf3 test result for the baseline bandwidth test

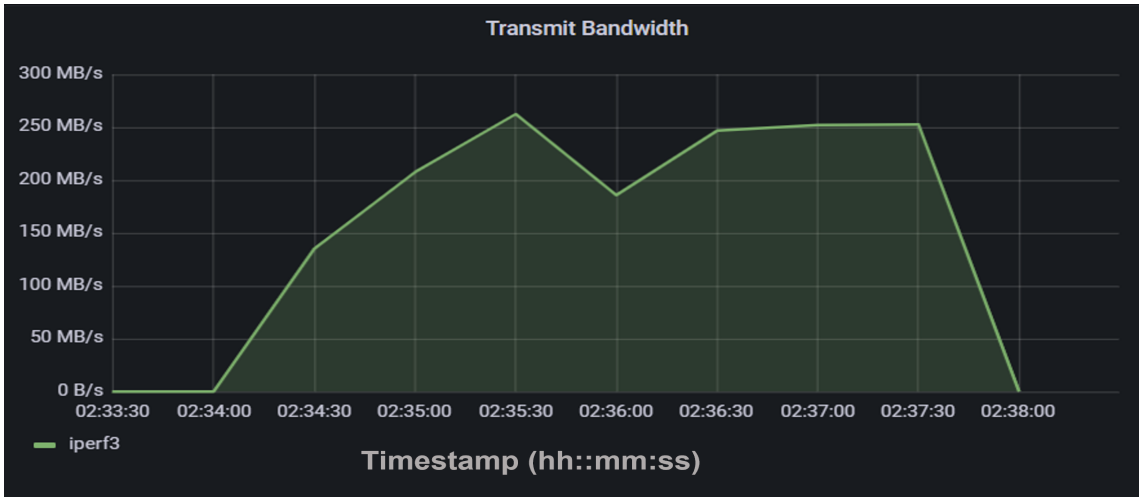


Figure 5.11: The Prometheus UI shows the baseline bandwidth test results

between the two NSM interfaces as demonstrated in Fig. 5.9.

The iper3 tool generated the breakdown table for the bandwidth test as demonstrated in Fig. 5.10. It shows that the average bandwidth of the NSM data plane was 1.84 Gbps (230 MB/s) which is 15 times smaller than the network bandwidth

supported by the WeaveNet Linux Kernel interface. Again, the same test result was obtained from the Grafana web UI as demonstrated in Fig. 5.11. The y-axis demonstrates that the average bandwidth is roughly 230 MB/s in the NSM transmit interface.

5.1.4 Latency Between Network Services

A slow network is unacceptable for time-critical applications such as video streaming services. High latency is increasingly problematic as networks grow bigger, since having more connections means more points where delays and issues can occur. These risks become greater as end-users connect with remote cloud servers and various network services across multiple domains. It's important to measure the network latency between network services in the proposed testbed so that the network operator can plan the network application accordingly. Network latency is the time taken for data or a request to go from the source to the destination. Network latency was measured in milliseconds. The Round-Trip Time (RTT) was obtained from the ICMP ping test including echo and echo reply messages. The packet size for each echo request was set to 1500 bytes instead of the default 32 bytes to increase the traffic load. Similar to the bandwidth test, the following three experiments were designed based on ICMP ping test to compare the network latency for three different use cases:

- The first RTT was measured from a simple deployment that contained only a client *Pod* (the sender), a server *Pod* (receiver), and an NSM traffic forwarder to obtain the baseline latency between two *Pods* which adopted the NSM CNIs.
- The second RTT instance was measured from the client *Pod* of the video streaming SFC mentioned in Section 4.7 to obtain the network latency if there is a firewall NSE in the middle of the chain.

• Compared to the second RTT measurement experiment, the third RTT instance was measured with the longer 6-*Pod* video streaming SFC deployment mentioned in Section 5.1.1 that involves three more passthrough *Pods*. Even though the passthrough *Pods* only forward the traffic from ingress to egress port, the RTT should increase not only because the SFC path of the 6-*Pod* video streaming SFC is longer than that of the 3-*Pod* SFC, but also because the processing time for redirecting traffic increases.

Each experiment sent one hundred ICMP packets and the test results are shown in Fig. 5.12. The y-axis is the RTT in seconds, and the x-axis is the trace of one hundred packets. The result shows that the average RTT was only 0.438 ms for the baseline deployment. For the 3-*Pod* video streaming SFC, the average RTT increased to 78.836 ms since there was a firewall VNF in between. The worst case is the 6-*Pod* video streaming SFC where an average RTT was 280.676 ms since there were three more passthrough *Pods*, hence the network latency was trebled. The RTT of the 6-*Pod* video streaming SFC also varied dramatically from 150 to 400 ms. These test results demonstrate that the network latency incrementation is close to a linear relationship with the number of elements in the SFC path. It also shows that the RTT becomes unstable when the length of the SFC path increases.

5.2 Reliability Results

In this thesis, a novel fault management mechanism for SFC was proposed that can rapidly and dynamically recover an SFC from a failed NSE. In the proposed testbed, no backup NSE or backup Service Function Path needs to be established in advance. The Network Service Mesh system monitors all the registered NSEs used by an SFC and if there are one or more NSEs out of service, it dynamically creates new NSE(s) with

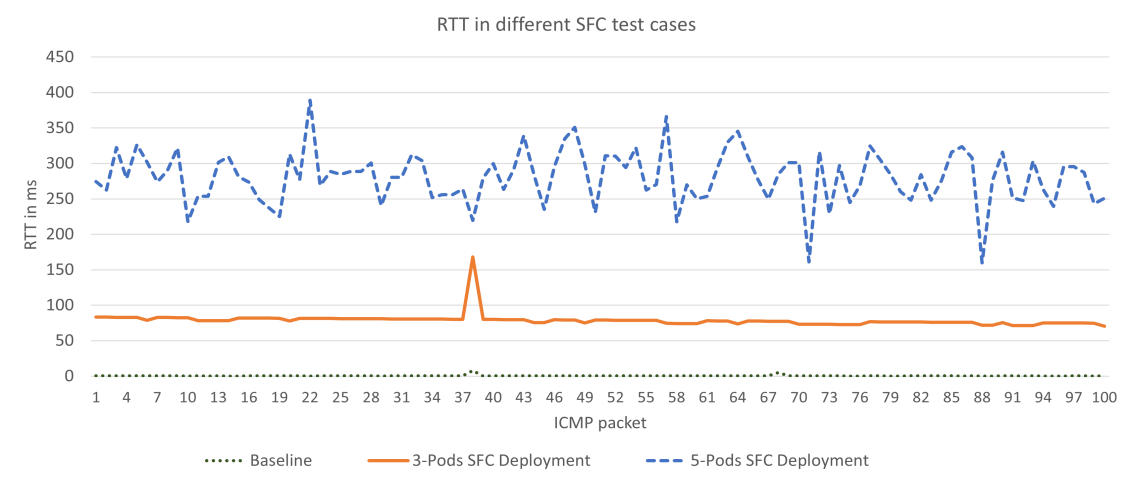


Figure 5.12: The RTT test results from different VNF deployment cases

the same functionality as that of the failed one(s) for replacement. Fault detection for SFC is managed as the health monitoring for *Pods* and virtual links between *Pods*. The recovery includes two parts: *Pod* recovery and SFC re-configuration. The healthiness and liveness probes used by the Kubernetes cluster detect the *Pod*'s failure and schedule a new *Pod* to replace the faulty container-based NSE, while the NSM control plane re-configures the SFC connections.

In real life scenarios, faults can be caused by issues such as CPU/memory overload or unreachable network. However, the fault in this test was simulated by manually deleting one or more components consumed by an SFC. The ICMP ping test was used to check the connectivity of the SFC path. The faulty components included NSE (e.g., firewall and video compression *Pod*), NSMgr, *NSM Registry*, VPP-forwarder, or any combinations of those. In this test, seven fault scenarios were selected to test the reliability of the SFC framework. Four scenarios involve a failure of one component including NSE, NSM traffic forwarder, NSMgr and NSM Regsiter. Two scenarios involve a failure of two components including NSMgr with NSE and NSMgr with traffic

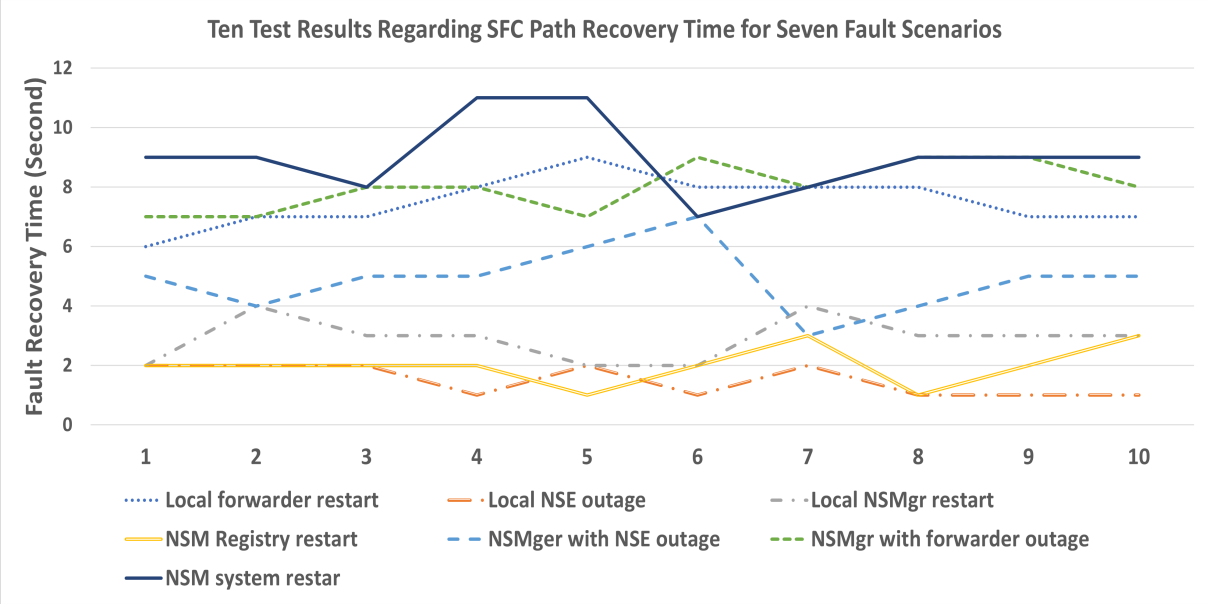


Figure 5.13: Ten Test Results Regarding SFC Path Recovery Time for Seven Fault Scenarios

forwarder. One extreme failure scenario involves all NSM control plane components. For each fault scenario, ten rounds of tests had been conducted to collect and calculate the average SFC path recovery time. Fig.5.13 shows the SFC path recovery time for all seven fault scenarios from ten tests. The y-axis is the fault recovery time in seconds and the x-axis is the trace of ten tests.

Table 5.3 illustrates the average recovery time against different fault scenarios. As shown in Table 5.3, the longest service downtime occurred when the entire NSM system restarted. It took nine seconds on average to recover the SFC connection. The result is expected as the NSM system restart involves all the elements of the deployed SFC including both control plane and data plane. This nine seconds recovery time is still quite fast comparing to the bootup time of VM which usually takes half a minute for a basic Ubuntu image [72]. ten seconds is usually the limit for keeping

the user's attention on the task for a web-based application, hence the nine seconds recovery time for restarting the entire SFC control plane and services is acceptable [73]. Furthermore, the traffic forwarder took most of the time to reconnect the data plane elements during the SFC path recovery process. The average recovery time was eight seconds for traffic forwarder restart. Meanwhile, the NSMgr and *NSM Registry* recovery took only three seconds and two seconds on average, respectively. Despite the recovery performance of the NSM component, NSE cloud be quickly resumed from any fault, as short as two seconds, to ensure consistent service as demonstrated in Table 5.3.

Fault Scenario	local forwarder outage	local NSE outage	local NSMgr restart	NSM Registry restart	NSMgr with NSE outage	NSMgr with Forwarder outage	NSM system restart
recover time (second)	8	2	3	2	5	8	9

Table 5.3: The average service recovery time against different fault scenarios

Chapter 6

Conclusion and Future Work

The work in this thesis addressed the difficulties of developing service function chaining in a cloud-native environment. A solution was proposed based on Kubernetes, Network Service Mesh, and Prometheus for users to interact and allocate a traffic steering SFC dynamically while considering the hardware resources of the underlying NFV infrastructure. The self-healing feature supported by Kubernetes and Network Service Mesh facilitates reliability of the SFC running on the testbed. Moreover, the SPIRE project was implemented in the cluster to provide workload authentication. Benefiting from various metrics exporters, the Prometheus monitoring system validated the container-based SFC traffic and ensured that the integrity of the SFC path was not compromised. In addition, hardware and network metrics such as network throughput and resource utilization were visualized using Grafana web UI. A user-friendly web interface was developed as the front-end of the testbed to allow the users to configure customized SFC easily. To demonstrate the proof-of-concept, a real-life scenario was built to emulate the video streaming use case and used to conduct various performance tests.

The Network Service Mesh is a promising open-source project to provide complicated layer 2 and layer 3 services that the Kubernetes cluster needs. Integrated with the most state-of-art open-source monitoring system, Prometheus, the network operator can investigate the traffic path and the available hardware resources. As demonstrated in the test result, the deployed SFC connectivity can be quickly recovered from service interruptions as short as 2 seconds. The SFC framework can also resist unexpected changes, but it will take a bit longer to recover. Furthermore, the CPU and memory usage of the NSM control plane elements indicate that the performance of the SFC framework is more sensitive to CPU interference rather than memory interference. The bandwidth and latency of the SFC data plane can also be monitored in real time. Unfortunately, the bandwidth of the NSM-based SFC data plane is worse than the default K8s data plane provided by WeaveNet. Moreover, when the length of the SFC path increases, the performance of the bandwidth and the latency become unacceptable compared to the default CNIs.

In the future, the collected metrics such as SFC request rate, timestamp, CPU, and memory usage information can be used to train a Machine Learning (ML) model that can perform SFC prediction and autoscaling. ML has emerged as a important solution for network modeling of the self-driven network. More importantly, it can foresee and eliminate issues before it happens. Investigation of ML for dynamic SFC deployment with effective resource management is one of the important research directions.

Appendix A

This section contains all the Dockerfiles for containers used in video streaming SFC.

A.1 Video Streaming Docker Container

```
ARG NGINX_VERSION=1.21.0
ARG NGINX_RTMP_VERSION=1.2.2
ARG FFMPEG_VERSION=4.4


##############################
# Build the NGINX-build image.
FROM alpine:3.13 as build-nginx
ARG NGINX_VERSION
ARG NGINX_RTMP_VERSION

# Build dependencies.
RUN apk add --update \
  build-base \
  ca-certificates \
  curl \
  gcc \
  libc-dev \
  libgcc \
  linux-headers \
  make \
  musl-dev \
  openssl \
  openssl-dev \
  pcre \
  pcre-dev \
  pkgconf \
  pkgconfig \
  zlib-dev

# Get nginx source.
RUN cd /tmp && \
```

```
  wget https://nginx.org/download/nginx-${NGINX_VERSION}.tar.gz && \
  tar zxf nginx-${NGINX_VERSION}.tar.gz && \
  rm nginx-${NGINX_VERSION}.tar.gz

# Get nginx-rtmp module.
RUN cd /tmp && \
  wget https://github.com/arut/nginx-rtmp-module/archive/v${NGINX_RTMP_VERSION}.tar.gz
&& \
  tar zxf v${NGINX_RTMP_VERSION}.tar.gz && rm v${NGINX_RTMP_VERSION}.tar.gz

# Compile nginx with nginx-rtmp module.
RUN cd /tmp/nginx-${NGINX_VERSION} && \
  ./configure \
  --prefix=/usr/local/nginx \
  --add-module=/tmp/nginx-rtmp-module-${NGINX_RTMP_VERSION} \
  --conf-path=/etc/nginx/nginx.conf \
  --with-threads \
  --with-file-aio \
  --with-http_ssl_module \
  --with-debug \
  --with-cc-opt="-Wimplicit-fallthrough=0" && \
  cd /tmp/nginx-${NGINX_VERSION} && make && make install

##############################
# Build the FFmpeg-build image.
FROM alpine:3.13 as build-ffmpeg
ARG FFMPEG_VERSION
ARG PREFIX=/usr/local
ARG MAKEFLAGS="-j4"

# FFmpeg build dependencies.
RUN apk add --update \
```

```
  build-base \
  coreutils \
  freetype-dev \
  lame-dev \
  libogg-dev \
  libass \
  libass-dev \
  libvpx-dev \
  libvorbis-dev \
  libwebp-dev \
  libtheora-dev \
  openssl-dev \
  opus-dev \
  pkgconf \
  pkgconfig \
  rtmpdump-dev \
  wget \
  x264-dev \
  x265-dev \
  yasm

RUN echo http://dl-cdn.alpinelinux.org/alpine/edge/community >> /etc/apk/repositories
RUN apk add --update fdk-aac-dev

# Get FFmpeg source.
RUN cd /tmp/ && \
  wget http://ffmpeg.org/releases/ffmpeg-${FFMPEG_VERSION}.tar.gz && \
  tar zxf ffmpeg-${FFMPEG_VERSION}.tar.gz && rm ffmpeg-${FFMPEG_VERSION}.tar.gz

# Compile ffmpeg.
RUN cd /tmp/ffmpeg-${FFMPEG_VERSION} && \
```

```
  ./configure \
  --prefix=${PREFIX} \
  --enable-version3 \
  --enable-gpl \
  --enable-nonfree \
  --enable-small \
  --enable-libmp3lame \
  --enable-libx264 \
  --enable-libx265 \
  --enable-libvpx \
  --enable-libtheora \
  --enable-libvorbis \
  --enable-libopus \
  --enable-libfdk-aac \
  --enable-libass \
  --enable-libwebp \
  --enable-postproc \
  --enable-avresample \
  --enable-libfreetype \
  --enable-openssl \
  --disable-debug \
  --disable-doc \
  --disable-ffplay \
  --extra-libs="-lpthread -lm" && \
  make && make install && make distclean

# Cleanup.
RUN rm -rf /var/cache/* /tmp/*

##########################
# Build the release image.
FROM alpine:3.13
```

```
127
128 # Set default ports.
129 ENV HTTP_PORT 80
130 ENV HTTPS_PORT 443
131 ENV RTMP_PORT 1935
132
133 RUN apk add --update \
134   ca-certificates \
135   gettext \
136   openssl \
137   pcre \
138   lame \
139   libogg \
140   curl \
141   libass \
142   libvpx \
143   libvorbis \
144   libwebp \
145   libtheora \
146   opus \
147   rtmpdump \
148   x264-dev \
149   x265-dev
150
151 RUN apk add --no-cache python3 py3-pip
152 COPY --from=build-nginx /usr/local/nginx /usr/local/nginx
153 COPY --from=build-nginx /etc/nginx /etc/nginx
154 COPY --from=build-ffmpeg /usr/local /usr/local
155 COPY --from=build-ffmpeg /usr/lib/libfdk-aac.so.2 /usr/lib/libfdk-aac.so.2
156 COPY . .
```

```
157
158 # Add NGINX path, config and static files.
159 ENV PATH "${PATH}:/usr/local/nginx/sbin"
160 ADD nginx.conf /etc/nginx/nginx.conf.template
161 RUN mkdir -p /opt/data && mkdir /www
162 ADD static /www/static
163
164 EXPOSE 1935
165 EXPOSE 80
166
167 #COPY start.sh /
168 #RUN chmod +x /start.sh
169
170 CMD envsubst "$(env | sed -e 's/=.*//' -e 's/^/\$/g')" < \
171   /etc/nginx/nginx.conf.template > /etc/nginx/nginx.conf && \
172   nginx & \
173   python3 broadcast.py
```

A.2 Firewall Container

```
ARG VPP_VERSION=v22.06-rc0-147-gb2b1a4ad2
FROM ghcr.io/edwarnicke/govpp/vpp:${VPP_VERSION} as go
COPY --from=golang:1.18.2-buster /usr/local/go/ /go
ENV PATH ${PATH}:/go/bin
ENV GO111MODULE=on
ENV CGO_ENABLED=0
ENV GOBIN=/bin
RUN rm -r /etc/vpp
RUN go install github.com/go-delve/delve/cmd/dlv@v1.8.2
ADD https://github.com/spiffe/spire/releases/download/v1.2.2/spire-1.2.2-linux-x86_64-glibc.tar.gz .
RUN tar xzvf spire-1.2.2-linux-x86_64-glibc.tar.gz -C /bin --strip=2 \
spire-1.2.2/bin/spire-server spire-1.2.2/bin/spire-agent

FROM go as build
WORKDIR /build
COPY go.mod go.sum ./
COPY ./internal/imports ./internal/imports
RUN go build ./internal/imports
COPY . .
RUN go build -o /bin/app .

FROM build as test
CMD go test -test.v ./...

FROM test as debug
CMD dlv -l :40000 --headless=true --api-version=2 test -test.v ./...

FROM ghcr.io/edwarnicke/govpp/vpp:${VPP_VERSION} as runtime
COPY --from=build /bin/app /bin/app
ENTRYPOINT [ "/bin/app" ]
```

A.3 Video Compression Container

```
FROM ubuntu/nginx:latest

# Compile and install fresh ffmpeg from sources:
# See: https://trac.ffmpeg.org/wiki/CompilationGuide/Ubuntu
RUN apt-get update -qq && apt-get -y install \
      autoconf \
      automake \
      build-essential \
      cmake \
      git-core \
      libass-dev \
      libfreetype6-dev \
      libsdl2-dev \
      libtool \
      libva-dev \
      libvdpau-dev \
      libvorbis-dev \
      libxcb1-dev \
      libxcb-shm0-dev \
      libxcb-xfixes0-dev \
      pkg-config \
      texinfo \
      wget \
      zlib1g-dev \
      nasm \
      yasm \
      libx265-dev \
      libnuma-dev \
      libvpx-dev \
      libmp3lame-dev \
      libopus-dev \
      libx264-dev \
      libfdk-aac-dev
```

```
RUN mkdir -p ~/ffmpeg_sources ~/bin && cd ~/ffmpeg_sources && \
    wget -O ffmpeg-4.2.2.tar.bz2 https://ffmpeg.org/releases/ffmpeg-4.2.2.tar.bz2 && \
    tar xjvf ffmpeg-4.2.2.tar.bz2 && \
    cd ffmpeg-4.2.2 && \
    PATH="$HOME/bin:$PATH" PKG_CONFIG_PATH="$HOME/ffmpeg_build/lib/pkgconfig" ./configure \
      --prefix="$HOME/ffmpeg_build" \
      --pkg-config-flags="--static" \
      --extra-cflags="-I$HOME/ffmpeg_build/include" \
      --extra-ldflags="-L$HOME/ffmpeg_build/lib" \
      --extra-libs="-lpthread -lm" \
      --bindir="$HOME/bin" \
      --enable-libfdk-aac \
      --enable-gpl \
      --enable-libass \
      --enable-libfreetype \
      --enable-libmp3lame \
      --enable-libopus \
      --enable-libvorbis \
      --enable-libvpx \
      --enable-libx264 \
      --enable-libx265 \
      --enable-nonfree && \
    PATH="$HOME/bin:$PATH" make -j8 && \
    make install -j8 && \
    hash -r
RUN mv ~/bin/ffmpeg /usr/local/bin && mv ~/bin/ffprobe /usr/local/bin && mv ~/bin/ffplay /usr/local/bin

RUN apt-get --yes install openssl  libssl-dev

RUN apt-get --yes install python3 pip

ADD my-script.sh /
RUN chmod +x /my-script.sh
COPY my-script.sh .
COPY nginx.conf .
RUN mkdir -p /input
RUN mkdir -p /inputnone
COPY videoreduction.py .
```

```
COPY videonone.py .
COPY encanto.mp4 .
COPY LUCA.mp4 .
COPY vidsoul1.mp4 .

#CMD python3 test.py
#CMD ["/usr/sbin/sshd","-D"]
#EXPOSE 1935
EXPOSE 80


ENTRYPOINT ["/bin/bash","/my-script.sh"]
```

Appendix B

This Appendix section provides configuration information regarding software used in this thesis.

B.1 Nginx-RTMP Module Configuration

These are the modified configuration files named nginx.conf for Nginx Server to stream video using Nginx-RTMP and HLS.

```
daemon off;

error_log /dev/stdout info;

events {
    worker_connections 1024;
}

rtmp {
    server {
        listen 1935;
        chunk_size 4000;

        application stream {
            live on;

            exec ffmpeg -i rtmp://localhost:1935/stream/$name
              -c:a libfdk_aac -b:a 128k -c:v libx264 -b:v 2500k -f flv -g 30 -r 30 -s 1280x720 \
               -preset superfast -profile:v baseline rtmp://localhost:1935/hls/$name_720p2628kbs
              -c:a libfdk_aac -b:a 128k -c:v libx264 -b:v 1000k -f flv -g 30 -r 30 -s 854x480 \
               -preset superfast -profile:v baseline rtmp://localhost:1935/hls/$name_480p1128kbs
              -c:a libfdk_aac -b:a 128k -c:v libx264 -b:v 750k -f flv -g 30 -r 30 -s 640x360 \
               -preset superfast -profile:v baseline rtmp://localhost:1935/hls/$name_360p878kbs
              -c:a libfdk_aac -b:a 128k -c:v libx264 -b:v 400k -f flv -g 30 -r 30 -s 426x240 \
               -preset superfast -profile:v baseline rtmp://localhost:1935/hls/$name_240p528kbs
              -c:a libfdk_aac -b:a 64k -c:v libx264 -b:v 200k -f flv -g 15 -r 15 -s 426x240 \
               -preset superfast -profile:v baseline rtmp://localhost:1935/hls/$name_240p264kbs;
        }
```

```
        application hls {
            live on;
            hls on;
            hls_fragment_naming system;
            hls_fragment 5;
            hls_playlist_length 10;
            hls_path /opt/data/hls;
            hls_nested on;

            hls_variant _720p2628kbs BANDWIDTH=2628000,RESOLUTION=1280x720;
            hls_variant _480p1128kbs BANDWIDTH=1128000,RESOLUTION=854x480;
            hls_variant _360p878kbs BANDWIDTH=878000,RESOLUTION=640x360;
            hls_variant _240p528kbs BANDWIDTH=528000,RESOLUTION=426x240;
            hls_variant _240p264kbs BANDWIDTH=264000,RESOLUTION=426x240;
        }
    }
}

http {
    root /www/static;
    sendfile off;
    tcp_nopush on;
    access_log /dev/stdout combined;

    # Uncomment these lines to enable SSL.
    # ssl_ciphers         HIGH:!aNULL:!MD5;
    # ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    # ssl_session_cache   shared:SSL:10m;
    # ssl_session_timeout 10m;
```

```
59
60      server {
61          listen 80;
62
63          # Uncomment these lines to enable SSL.
64          # Update the ssl paths with your own certificate and private key.
65
66          # listen 443 ssl;
67          # ssl_certificate     /opt/certs/example.com.crt;
68          # ssl_certificate_key /opt/certs/example.com.key;
69
70          location /hls {
71              types {
72                  application/vnd.apple.mpegurl m3u8;
73                  video/mp2t ts;
74              }
75              root /opt/data;
76              add_header Cache-Control no-cache;
77              add_header Access-Control-Allow-Origin *;
78          }
79
80          location /live {
81            alias /opt/data/hls;
82            types {
83                application/vnd.apple.mpegurl m3u8;
84                video/mp2t ts;
85            }
86            add_header Cache-Control no-cache;
87            add_header Access-Control-Allow-Origin *;
```

```
88          }
89
90          location /stat {
91              rtmp_stat all;
92              rtmp_stat_stylesheet stat.xsl;
93          }
94
95          location /stat.xsl {
96              root /www/static;
97          }
98
99          location /crossdomain.xml {
100             default_type text/xml;
101             expires 24h;
102         }
103     }
104 }
```

B.2 Firewall Configuration File

The firewall configuration file defines the traffic rules for this ACL-based firewall.

```yaml
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: vppagent-firewall-config-file
data:
  config.yaml: |
    allow icmp:
      ispermit: 1
      proto: 1
      srcportoricmptypelast: 65535
      dstportoricmpcodelast: 65535
    forbid tcp8080:
      proto: 6
      srcportoricmptypelast: 65535
      dstportoricmpcodefirst: 8080
      dstportoricmpcodelast: 8080
    allow tcp80:
      ispermit: 1
      proto: 6
      srcportoricmptypelast: 65535
      dstportoricmpcodefirst: 80
      dstportoricmpcodelast: 80
```

Appendix C

This Appendix section provides Python3 scripts and Linux scripts used in this thesis for automating processes.

C.1 Broadcast.py

The function of this script monitoring the input folder and upload any video in the input folder to the HLS streaming server using FFmpeg.

```python
import os, time

destination = "/output/"
path_to_watch = "/input/"
before = dict ([(f, None) for f in os.listdir (path_to_watch)])

while 1:
    time.sleep (1)
    after = dict ([(f, None) for f in os.listdir (path_to_watch)])
    added = [f for f in after if not f in before]
    if added:
        print ("Added: ", ", ".join (added))
        os.system('ffmpeg ' + '-re ' + '-i ' + path_to_watch + ''.join(added) + \
          ' -vcodec' + ' copy ' + '-loop ' \
          + '-1 ' + '-c:a ' + 'aac ' + '-b:a ' + '160k ' + '-ar ' + '44100 ' + '-strict'
        print("File copied: ", ''.join(added))
    before = after
```

C.2 Video_compression.sh

Linux Shell script and Python script for automating the video compression function.

```bash
#!/bin/bash
envsubst "$(env | sed -e 's/=.*//' -e 's/^/\$/g')" \
< /etc/nginx/nginx.conf.template > /etc/nginx/nginx.conf &
nginx &
python3 videoreduction.py &
python3 videonone.py
```

C.3 Video_reduction.py

```
import os, time

destination = "/var/www/html/"
path_to_watch = "/input/"
before = dict ([(f, None) for f in os.listdir (path_to_watch)])

while 1:
  time.sleep (1)
  after = dict ([(f, None) for f in os.listdir (path_to_watch)])
  added = [f for f in after if not f in before]
  if added:
      print ("Added: ", ", ".join (added))
      #os.system('cp '+ path_to_watch + ''.join(added) + ' ' + destination)
      os.system('ffmpeg ' + '-i ' + path_to_watch + ''.join(added)
      + ' -crf ' + '40 ' + destination + ' '.join(added))
      print("File copied: ", ''.join(added))
  before = after
```

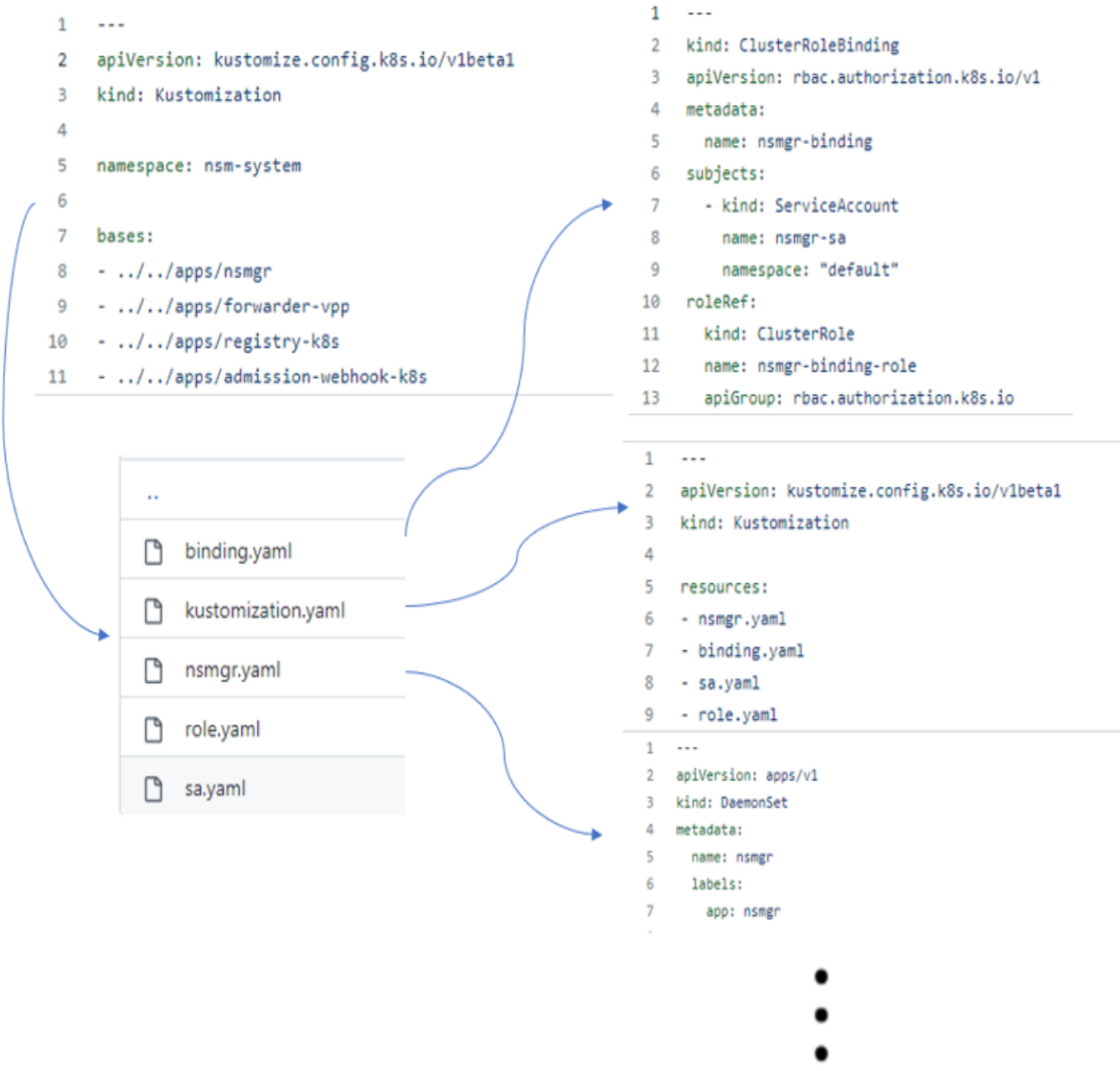
Appendix D

This Appendix section provides manifest YAML files for deploying NSM framework objects including NSMgr, Network Service Forwarder, NSM Registry, and Admin-Webhook.

D.1 NSM Framework Kustomization YAML File

```
1  ---
2  apiVersion: kustomize.config.k8s.io/v1beta1
3  kind: Kustomization
4
5  namespace: nsm-system
6
7  bases:
8  - ../../apps/nsmgr
9  - ../../apps/forwarder-vpp
10 - ../../apps/registry-k8s
11 - ../../apps/admission-webhook-k8s
```

D.2 YAML File to Deploy NSM Control Plane Component-NSMgr



D.3 Kustomization YAML File for Deploying SPIRE Project

```yaml
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: spire

generatorOptions:
  disableNameSuffixHash: true

secretGenerator:
- name: spire-secret
  files:
  - bootstrap.key
  - bootstrap.crt

resources:
- spire-namespace.yaml
- agent-account.yaml
- agent-cluster-role.yaml
- agent-configmap.yaml
- agent-daemonset.yaml
- server-account.yaml
- server-cluster-role.yaml
- server-configmap.yaml
- server-service.yaml
- server-statefulset.yaml
- spire-bundle-configmap.yaml
```

Appendix E

This Appendix section contains manifest YAML files for deploying network service *Pods* such as NSC, NSE, NS, *Services*, and its corresponding patches.

E.1 NSC Original Manifest Files

kustomization.yaml

```
1  ---
2  apiVersion: kustomize.config.k8s.io/v1beta1
3  kind: Kustomization
4
5  resources:
6  - nsc.yaml
```

nsc.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nsc-kernel
  labels:
    app: nsc-kernel
spec:
  selector:
    matchLabels:
      app: nsc-kernel
  template:
    metadata:
      labels:
        app: nsc-kernel
        "spiffe.io/spiffe-id": "true"
    spec:
      containers:
        - name: nsc
          image: ghcr.io/networkservicemesh/ci/cmd-nsc:801ebc5
          imagePullPolicy: IfNotPresent
          env:
            - name: SPIFFE_ENDPOINT_SOCKET
              value: unix:///run/spire/sockets/agent.sock
            - name: NSM_LOG_LEVEL
              value: TRACE
            - name: NSM_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
          volumeMounts:
            - name: spire-agent-socket
              mountPath: /run/spire/sockets
              readOnly: true
            - name: nsm-socket
              mountPath: /var/lib/networkservicemesh
              readOnly: true
          resources:
            requests:
```

```
            cpu: 100m
            memory: 40Mi
          limits:
            memory: 80Mi
            cpu: 200m
      volumes:
        - name: spire-agent-socket
          hostPath:
            path: /run/spire/sockets
            type: Directory
        - name: nsm-socket
          hostPath:
            path: /var/lib/networkservicemesh
            type: DirectoryOrCreate
```

E.2 The Kustomization File for Deploying the Video Streaming SFC

```yaml
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

namespace: ns-gq4bf

resources:
- config-file.yaml
- sfc-sc-videoservice.yaml
- sfc-sc-chain.yaml

bases:
- ../../../apps/nsc-kernel
- ../../../apps/nse-kernel
- ./nse-firewall

patchesStrategicMerge:
- patch-nsc.yaml
- patch-nse.yaml
```

E.3 The sfc-sc-videoservice.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: sfc-sc-videoservice
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
    nodePort: 30120
  - port: 1935
    targetPort: 1935
    protocol: TCP
    name: rtmp
    #nodePort: 30011
  selector:
    app: nsc-kernel
```

E.4 The sfc-sc-chain.yaml

```yaml
---
apiVersion: networkservicemesh.io/v1
kind: NetworkService
metadata:
  name: sfc-sc-chain
  namespace: nsm-system
spec:
  payload: ETHERNET
  matches:
    - source_selector:
        app: firewall
      routes:
        - destination_selector:
            app: nse
    - routes:
        - destination_selector:
            app: firewall
```

E.5 patch-nsc.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nsc-kernel
spec:
  template:
    spec:
      containers:
        - name: nsc
          env:
            - name: NSM_NETWORK_SERVICES
              value: kernel://sfc-sc-chain/nsm-1
            - name: NSM_REQUEST_TIMEOUT
              value: 75s
        - name: broadcastcon
          image: broadcastimage:latest
          imagePullPolicy: IfNotPresent
      nodeSelector:
        kubernetes.io/hostname: ubuntu
```

E.6 patch-nse.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nse-kernel
spec:
  template:
    spec:
      containers:
        - name: nse
          env:
            - name: NSM_CIDR_PREFIX
              value: 172.16.1.100/31
            - name: NSM_SERVICE_NAMES
              value: "sfc-sc-chain"
            - name: NSM_REGISTER_SERVICE
              value: "false"
            - name: NSM_LABELS
              value: "app:nse"
        - name: nginx-ffmpeg
          image: reductionimage:latest
          imagePullPolicy: IfNotPresent
      nodeSelector:
        kubernetes.io/hostname: ubuntu153

```

E.7 NSE Original Manifest Files

kustomization.yaml

```yaml
---
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- nse.yaml
```

nse.yaml

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nse-kernel
  labels:
    app: nse-kernel
spec:
  selector:
    matchLabels:
      app: nse-kernel
  template:
    metadata:
      labels:
        app: nse-kernel
        "spiffe.io/spiffe-id": "true"
    spec:
      containers:
        - name: nse
          image: ghcr.io/networkservicemesh/ci/cmd-nse-icmp-responder:c77cb2c
          imagePullPolicy: IfNotPresent
          env:
            - name: SPIFFE_ENDPOINT_SOCKET
              value: unix:///run/spire/sockets/agent.sock
            - name: NSM_NAME
              valueFrom:
```

```
27                fieldRef:
28                  fieldPath: metadata.name
29            - name: NSM_LOG_LEVEL
30              value: TRACE
31            - name: NSM_CONNECT_TO
32              value: unix:///var/lib/networkservicemesh/nsm.io.sock
33          volumeMounts:
34            - name: spire-agent-socket
35              mountPath: /run/spire/sockets
36              readOnly: true
37            - name: nsm-socket
38              mountPath: /var/lib/networkservicemesh
39              readOnly: true
40          resources:
41            requests:
42              cpu: 100m
43              memory: 40Mi
44            limits:
45              memory: 80Mi
46              cpu: 200m
47      volumes:
48        - name: spire-agent-socket
49          hostPath:
50            path: /run/spire/sockets
51            type: Directory
52        - name: nsm-socket
53          hostPath:
54            path: /var/lib/networkservicemesh
55            type: DirectoryOrCreate
```

Appendix F

This Appendix section contains manifest YAML files for the Prometheus monitoring system including the Deployment and *Service*.

F.1 Service for Prometheus database

```
1 apiVersion: v1
2 kind: "Service"
3 metadata:
4   name: prometheus
5   labels:
6     name: prometheus
7 spec:
8   ports:
9   - name: prometheus
10    protocol: TCP
11    port: 9090
12    targetPort: 9090
13  selector:
14    app: prometheus
15  type: NodePort
```

F.2 Deployment for Prometheus components

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    name: prometheus
  name: prometheus
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus
    spec:
      containers:
      - name: prometheus
        image: prom/prometheus:v2.2.1
        command:
        - "/bin/prometheus"
        args:
        - "--config.file=/etc/prometheus/prometheus.yml"
        ports:
        - containerPort: 9090
          protocol: TCP
        volumeMounts:
        - mountPath: "/etc/prometheus"
          name: prometheus-config
      volumes:
      - name: prometheus-config
        configMap:
          name: prometheus-config
```

F.3 Python3 Scripts for Metrics Collection

```python
import csv
import time
import requests

"""
A simple program to print the result of a Prometheus query as CSV.
"""
# NIC Info instance value
nic_rx = 'node_network_receive_bytes_total{job="Ciena_Ottawa"}!=0'
nic_tx = 'node_network_transmit_bytes_total{job="Ciena_Ottawa"}!=0'
#nic_speed = 'node_network_speed_bytes{job="Ciena_Ottawa"}/(1024*1024*1024/8)!=0'
nic_speed = 'node_network_speed_bytes{instance=~"162.244.229.51:9100.*"}*8'



nic_list = (nic_rx, nic_tx, nic_speed)
# NIC Info instance value
nic_rx_range = 'delta(node_network_receive_packets_total{job="Ciena_Ottawa"}[1m])!=0'
nic_tx_range = 'delta(node_network_transmit_packets_total{job="Ciena_Ottawa"}[1m])!=0'
nic_rx_increase = 'irate(node_network_receive_packets_total{job="Ciena_Ottawa"}[1m])!=0'
nic_tx_increase = 'irate(node_network_transmit_packets_total{job="Ciena_Ottawa"}[1m])!=0'
network_throughput = 'irate(node_network_transmit_bytes_total{instance=~"162.244.229.51:9100.*"}[1m])'
Packet_losses = 'irate(node_netstat_Tcp_RetransSegs{instance=~"162.244.229.51:9100.*"}[1m])'

nic_list_changes = (nic_rx_range,nic_tx_range,nic_rx_increase,nic_tx_increase, network_throughput , Packet_losses)
# NIC Info range values
nic_rx_5 = 'node_network_receive_bytes_total{job="Ciena_Ottawa"}[5m]'
nic_tx_5 = 'node_network_transmit_bytes_total{job="Ciena_Ottawa"}[5m]'
nic_list_5min=(nic_rx_5,nic_tx_5)
#CPU info

number_of_cpu = 'count without(cpu, mode) (node_cpu_seconds_total{mode="idle", job="Ciena_Ottawa"})'
avg_cpu_core_temperature = 'avg (node_hwmon_temp_celsius{instance=~"162.244.229.51:9100",job="Ciena_Ottawa"})'
cpu_utilization = '1 - avg without(cpu) (rate(node_cpu_seconds_total{job="Ciena_Ottawa",mode="idle"}[2m]))'
NVMe_total_util ='irate(node_disk_io_time_seconds_total{instance=~"162.244.229.51:9100.*", device=~"nvme.*"}[5m])'
NVMe_transfer_bytes ='irate(node_disk_read_bytes_total{instance=~"162.244.229.51:9100.*", device=~"nvme.*"}[5m])'
storage_count = 'node_disk_io_time_seconds_total{instance=~"162.244.229.51:9100.*", device=~"nvme[0-7]n1"}'
goodput = 'irate(node_disk_written_bytes_total{instance=~"162.244.229.51:9100.*", device=~"nvme.*"}[1m])'

#'irate(node_disk_io_time_seconds_total{job="Ciena_Ottawa"}[5m])'
```

```
#cpu_
cpu_list = (number_of_cpu, avg_cpu_core_temperature, cpu_utilization, NVMe_total_util, NVMe_transfer_bytes, storage_count, goodput)
#params = 'irate(process_cpu_seconds_total[1h])'
#params = 'process_cpu_seconds_total[1m]'
#__name__!=""


def extractor_instance_value(nic_list):
    for i in nic_list:
        response = requests.get('http://165.124.33.158:9091/api/v1/query',
            params={'query':i})
        results = response.json()['data']['result']

    # Build a list of all labelnames used.
        labelnames = set()
        for result in results:
            labelnames.update(result['metric'].keys())

    # Canonicalize
        labelnames.discard('__name__')
        labelnames = sorted(labelnames)

    #writer = csv.writer(sys.stdout)
    # Write the header,
        filename = "instance "+i[:24]+".csv"
        with open(filename, 'a') as f:
            writer = csv.writer(f)
        #writer.writerow(['query_name', 'timestamp', 'value'] + labelnames)

    # Write the samples.
            for result in results:
                if result['metric'].get('__name__', ''):
                    l = [result['metric'].get('__name__', '')] + result['value']
                    for label in labelnames:
                    #print(result['metric'].get(label, ''))
                        l.append(result['metric'].get(label, ''))
                    writer.writerow(l)
                else:
```

```
                l = result['value']
                for label in labelnames:
                    # print(result['metric'].get(label, ''))
                    l.append(result['metric'].get(label, ''))
                writer.writerow(l)


def extractor_range_value(nic_list_range):
    for i in nic_list_range:
        response = requests.get('http://165.124.33.158:9091/api/v1/query',
            params={'query':i})
        results = response.json()['data']['result']

    # Build a list of all labelnames used.
        labelnames = set()
        for result in results:
            labelnames.update(result['metric'].keys())

    # Canonicalize
        labelnames.discard('__name__')
        labelnames = sorted(labelnames)

    #writer = csv.writer(sys.stdout)
    # Write the header,
        filename = "range "+i[:24]+".csv"
        with open(filename, 'a') as f:
            writer = csv.writer(f)
        #writer.writerow(['query_name', 'timestamp', 'value'] + labelnames)

    # Write the samples.
            for result in results:
                l = [result['metric'].get('__name__', '')] + result['values']
                for label in labelnames:
                    l.append(result['metric'].get(label, ''))
                writer.writerow(l)

```

```
def extractor_range_value(cpu_list):
    for i in cpu_list:
        response = requests.get('http://165.124.33.158:9091/api/v1/query',
            params={'query':i})
        results = response.json()['data']['result']

    # Build a list of all labelnames used.
        labelnames = set()
        for result in results:
            labelnames.update(result['metric'].keys())

    # Canonicalize
        labelnames.discard('__name__')
        labelnames = sorted(labelnames)

    #writer = csv.writer(sys.stdout)
    # Write the header,
        filename = "range "+i[:24]+".csv"
        with open(filename, 'a') as f:
            writer = csv.writer(f)
        #writer.writerow(['query_name', 'timestamp', 'value'] + labelnames)

    # Write the samples.
            for result in results:
                l = [result['metric'].get('__name__', '')] + result['values']
                for label in labelnames:
                    l.append(result['metric'].get(label, ''))
                writer.writerow(l)

#extractor_range_value(nic_list_5min)

while (1):
    extractor_instance_value(nic_list)

    extractor_instance_value(nic_list_changes)

    extractor_range_value(nic_list_5min)

    extractor_instance_value(cpu_list)
```

Bibliography

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2015.

[2] ETSI. Network functions virtualisation (nfv); architectural framework. `https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf`.

[3] Ashish Lingayat, Ranjana R. Badre, and Anil Kumar Gupta. Performance evaluation for deploying docker containers on baremetal and virtual machine. In *2018 3rd International Conference on Communication and Electronics Systems (ICCES)*, pages 1019–1023, 2018.

[4] The Kubernetes Authors. What is kubernetes? `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`, 06.04.2022. Accessed: 2022-7-10.

[5] Basic examples. `https://github.com/networkservicemesh/deployments-k8s/tree/main/examples/basic`. Accessed: 2022-06-17.

[6] kube-prometheus-stack. `https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack`. Accessed: 2022-06-13.

[7] Hadi Razzaghi Kouchaksaraei and Holger Karl. Service function chaining across openstack and kubernetes domains. In *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, DEBS '19, page 240–243, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Shih-Ying Song and Fuchun Joseph Lin. Dynamic fault management in service function chaining. In *2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1477–1482, 2020.

[9] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad. A survey on service function chaining. *Journal of Network and Computer Applications*, 75:138–155, 2016.

[10] Spire. `https://github.com/networkservicemesh/deployments-k8s/tree/main/examples/spire`. Accessed: 2022-06-14.

[11] What is cloud native. `https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition`, 04.07.2022. accessed: 01.09.2016.

[12] Chris Richardson. What are microservices? `https://microservices.io/index.html`. Accessed: 2022-06-06.

[13] Eric Knorr. The 2020 idg cloud computing survey. `https://www.infoworld.com/article/3561269/the-2020-idg-cloud-computing-survey.html`, 08.06.2020. Accessed: 2022-06-08.

[14] Adel Bouridah, Ilhem Fajjari, Nadjib Aitsaadi, and Hacene Belhadef. Optimized scalable sfc traffic steering scheme for cloud native based applications. In *2021 IEEE 18th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6, 2021.

[15] Scott van Kalken of F5. What are namespaces and cgroups, and how do they work? https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work, 07.21.2021. Accessed: 2022-06-06.

[16] Raffaele Giuseppe Trani. Integrating vnf service chains in kubernetes cluster, 2020. https://webthesis.biblio.polito.it/14516/1/tesi.pdf.

[17] Brunstrom A. Petlund A. Hurtig, P. and M. Welzl. TCP and Stream Control Transmission Protocol (SCTP) RTO Restart. RFC 7765, February 2016.

[18] Anna Engelmann, Wolfgang Bziuk, and Admela Jukan. Bounding reliability in service function chaining. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 413–418, 2020.

[19] Davide Borsatti, Gianluca Davoli, Walter Cerroni, Chiara Contoli, and Franco Callegati. Performance of service function chaining on the openstack cloud platform. In *2018 14th International Conference on Network and Service Management (CNSM)*, pages 432–437, 2018.

[20] Prometheus Authors. Overview. https://prometheus.io/docs/introduction/overview/. Accessed: 2021-6-23.

[21] Github deploymentsk8s. https://github.com/networkservicemesh/deployments-k8s. Accessed: 2022-06-17.

[22] Key concepts. https://networkservicemesh.io/docs/concepts/architecture/, author = The Network Service Mesh authors, note = Accessed: 2022-7-10, year = 06.04.2022.

[23] The SPIFFE authors. Spiffe overview. https://spiffe.io/docs/latest/spiffe-about/overview/. Accessed: 2021-6-23.

[24] Yong Li and Min Chen. Software-defined network function virtualization: A survey. *IEEE Access*, 3:2542–2553, 2015.

[25] Onos overview. https://opennetworking.org/onf-sdn-projects/. Accessed: 2022-6-11.

[26] Ryu SDN Framework Community. Build sdn agilely. https://ryu-sdn.org/, 2017. Accessed: 2022-6-10.

[27] OpenDaylight Project. Platform overview. https://www.opendaylight.org/about/platform-overview. Accessed: 2022-6-23.

[28] Red Hat. Overview. http://skydive.network/documentation/. Accessed: 2022-6-23.

[29] The Jaeger Authors. Getting started. https://www.jaegertracing.io/docs/1.36/getting-started/. Accessed: 2022-6-23.

[30] Linux Foundation 2018-2021. What is the vector packet processor (vpp). https://s3-docs.fd.io/vpp/22.02/. Accessed: 2022-1-28.

[31] Andreas Berl, Hermann De Meer, Helmut Hlavacs, and Thomas Treutner. Virtualization in energy-efficient future home environments. *IEEE Communications Magazine*, 47(12):62–67, 2009.

[32] Kubernetes Blog. Don't panic: Kubernetes and docker. https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/, 02.12.2020. Accessed: 2022-6-16.

[33] containerd Authors. containerd overview. https://containerd.io/docs/. Accessed: 2022-6-06.

[34] Docker Inc. Swarm mode overview. https://docs.docker.com/engine/swarm/. Accessed: 2022-6-23.

[35] Kubernetes Blog. Container runtime interface (cri). https://kubernetes.io/docs/concepts/architecture/cri/, 10.01.2020. Accessed: 2022-6-16.

[36] The Kubernetes Authors. Pods. https://kubernetes.io/docs/concepts/workloads/pods/, 10.01.2020. Accessed: 2022-6-16.

[37] The Kubernetes Authors. Service. https://kubernetes.io/docs/concepts/services-networking/service/, 06.18.2022. Accessed: 2022-6-16.

[38] The Kubernetes Authors. Secret. https://kubernetes.io/docs/concepts/configuration/secret/, 05.14.2022. Accessed: 2022-6-16.

[39] The Kubernetes Authors. Configmaps. https://kubernetes.io/docs/concepts/configuration/configmap/, 05.04.2022. Accessed: 2022-6-16.

[40] The Kubernetes Authors. Volumes. https://kubernetes.io/docs/concepts/storage/volumes/, 06.19.2022. Accessed: 2022-6-20.

[41] Boutheina Dab, Ilhem Fajjari, Mathieu Rohon, Cyril Auboin, and Arnaud Diquélou. An efficient traffic steering for cloud-native service function chaining. In *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, pages 71–78, 2020.

[42] Contiv-vpp kubernetes network plugin. https://contivpp.io/docs/concepts/what-is-contiv-vpp/. Accessed: 2022-3-02.

[43] Ovn4nfv k8s plugin - network controller. https://github.com/opnfv/ovn4nfv-k8s-plugin, 06.10.2020. Accessed: 2022-6-16.

[44] The Network Service Mesh authors. Architecture. https://networkservicemesh.io/docs/concepts/architecture/. Accessed: 2022-6-16.

[45] Nitin Sukhija and Elizabeth Bautista. Towards a framework for monitoring and analyzing high performance computing environments using kubernetes and prometheus. In *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 257–262, 2019.

[46] Helm Authors. Helm architecture. https://helm.sh/docs/topics/architecture/. Accessed: 2022-06-06.

[47] Roberto Morabito and Nicklas Beijar. A framework based on sdn and containers for dynamic service chains on iot gateways. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, page 42–47, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Pox manual current documentation. https://noxrepo.github.io/pox-doc/html/. Accessed: 2022-06-15.

[49] Sergio Livi, Quentin Jacquemart, Dino Lopez Pacheco, and Guillaume Urvoy-Keller. Container-based service chaining: A performance perspective. In *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, pages 176–181, 2016.

[50] Sevil Dräxler, Holger Karl, Manuel Peuster, Hadi Razzaghi Kouchaksaraei, Michael Bredel, Johannes Lessmann, Thomas Soenen, Wouter Tavernier, Sharon Mendel-Brin, and George Xilouris. Sonata: Service programming and orchestration for virtualized software networks. In *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 973–978, 2017.

[51] Service function chaining extension for openstack networking. `https://docs.openstack.org/networking-sfc/latest/`, 06.10.2019. Accessed: 2021-09-07.

[52] Cong-Phuoc Hoang, Ngoc-Thanh Dinh, and YoungHan Kim. An extended virtual network functions manager architecture to support container. In *Proceedings of the 2018 International Conference on Information Science and System*, ICISS '18, page 173–176, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Tacker - openstack nfv orchestration. `https://wiki.openstack.org/wiki/Tacker`. Accessed: 2022-06-06.

[54] Service function chaining extension for openstack networking. `https://docs.openstack.org/networking-sfc/rocky/`, 09.29.2018. Accessed: 2022-06-06.

[55] Project description. `https://docs.openstack.org/kuryr-kubernetes/latest/readme.html`, 05.18.2016. Accessed: 2022-06-06.

[56] Alessio Giorgetti, Davide Scano, Javad Chamanara, Mustafa Albado, Edgard Marx, Sean Ahearne, Andrea Sgambelluri, Francesco Paolucci, and Filippo Cugini. Kubernetes orchestration in sdn-based edge network infrastructure. In *Optical Fiber Communication Conference*, pages M3Z–3. Optica Publishing Group, 2022.

[57] Michel Gokan Khan, Saeed Bastani, Javid Taheri, Andreas Kassler, and Shuiguang Deng. Nfv-inspector: A systematic approach to profile and analyze virtual network

functions. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–7, 2018.

[58] Rami Akrem Addad, Diego Leonel Cadette Dutra, Tarik Taleb, and Hannu Flinck. Ai-based network-aware service function chain migration in 5g and beyond networks. *IEEE Transactions on Network and Service Management*, 19(1):472–484, 2022.

[59] Felipe Ramos, Eduardo Viegas, Altair Santin, Pedro Horchulhack, Roger R. dos Santos, and Allan Espindola. A machine learning model for detection of docker-based app overbooking on kubernetes. In *ICC 2021 - IEEE International Conference on Communications*, pages 1–6, 2021.

[60] Yixin Bao, Yanghua Peng, and Chuan Wu. Deep learning-based job placement in distributed machine learning clusters. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pages 505–513, 2019.

[61] Nikunj Parekh, Swathi Kurunji, and Alan Beck. Monitoring resources of machine learning engine in microservices architecture. In *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 486–492, 2018.

[62] Tania Lorido-Botrán, Jose Miguel-Alonso, and Jose Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12, 12 2014.

[63] Rob Ewaschuk. Monitoring distributed systems. `https://sre.google/sre-book/monitoring-distributed-systems/`. Accessed: 2022-06-11.

[64] About ffmpeg. `https://ffmpeg.org/about.html`. Accessed: 2022-06-07.

[65] Christopher Mueller of BITMOVIN. Scalable live video streaming with nginx plus and bitmovin. https://www.nginx.com/blog/scalable-live-video-streaming-nginx-plus-bitmovin/, 03.16.2016. Accessed: 2022-06-07.

[66] Michel Gokan Khan, Javid Taheri, Mohammad Ali Khoshkholghi, Andreas Kassler, Carolyn Cartwright, Marian Darula, and Shuiguang Deng. A performance modelling approach for sla-aware resource recommendation in cloud native network functions. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 292–300. IEEE, 2020.

[67] Docker Inc. Dockerfile reference. https://docs.docker.com/engine/reference/builder/. Accessed: 2022-06-10.

[68] Linux Foundation. Go api. https://s3-docs.fd.io/vpp/22.06/interfacing/go/index.html. Accessed: 2022-06-12.

[69] Benchmarking Methodology for Network Interconnect Devices. RFC 2544, March 1999.

[70] Scott O. Bradner. Benchmarking Terminology for Network Interconnection Devices. RFC 1242, July 1991.

[71] The Kubernetes Authors. Resource management for pods and containers. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, 6.11.2022.

[72] Yuqing Qiu, Chung-Horng Lung, Samuel Ajila, and Pradeep Srivastava. Experimental evaluation of lxc container migration for cloudlets using multipath tcp. *Computer Networks*, 164:106900, 2019.

[73] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.