POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Integrating VNF Service Chains in Kubernetes Cluster

Supervisor
Prof. Fulvio Risso

Candidate
Raffaele Giuseppe Trani

Academic Year 2019-2020

Dedicated to
my family

Contents

Abstract

In recent years, the so called Cloudification is spreading more and more and, in network world, it is bringing the Network Function Virtualization (NFV) paradigm to life. With Cloudification, the abstraction given by NFV is brought at another level: the integration of this two trends means that network functions would not be deployed as virtual machines on physical servers, but as containers, the basic element in a cloud environment, possibly deployed on same virtual machine. In this respect, telecom operators need to adopt to this evolution, taking in account all the problems related to it.

In first instance, this thesis provides an analysis of Kubernetes (the most popular orchestrator for cloud native services) networking system, which is provided by the so called Container Network Interface (CNI). It consists of set of APIs to define a network configuration to be applied inside Kubernetes' cluster, in order to provide network connectivity to elements in the cluster. There are different implementations of the CNI and some of them were analyzed to understand if, with their functionalities, they can provide the necessary requirements to integrate VNFs in Kubernetes environment. From the analysis, it comes out that CNIs do not provide the features required for the integration, so other solutions were considered and studied.

The most relevant and interesting existing solution is Network Service Mesh project. In order to solve these problems and make VNFs work in a Kubernetes environment, Network Service Mesh (NSM) project proposes to allow chaining between pods that implements VNFs by building "virtual wires" that connect them and allow the traffic to flow inside the established chain.

A deep analysis of the implementation of this project was carried out, understanding how it works in terms of control plane and data plane functionalities.

To understand the potentialities of NSM's proposed solution, several performance tests were carried out. To provide a basis of comparison, different scenarios have been deployed, which do not necessarily includes NSM and Kubernetes. These scenarios were manually created and tested, so it was possible to provide a comparison not only in terms of performance, but also in terms of specific configurations and requirements that were needed to deploy each scenario.

Chapter 1

Introduction

The world of telecommunications and networking is moving from Virtualization to Cloudification. Virtualization was born out of a desire to consolidate individual servers running disparate applications onto fewer servers by implementing these applications on virtual machines (VMs) in a hypervisor environment. In this optic, the process of Network Function Virtualization (NFV) took place: network functions were moved from physical dedicated hardware blocks to VMs running on common servers (properly implementing the chaining between them), becoming in this way Virtual Network Functions (VNFs).

The introduction of NFV technologies brought many benefits in terms of scalability and flexibility, but there are also some drawbacks, e.g., in terms of performance, compared to the solutions based on purpose-built hardware tightly coupled with network function software. In these terms, Cloudification, and more specifically containers, can represent a better solution of network functions. With Cloudification, network functions would not be deployed as virtual machines on physical servers, but as containers, the basic element in a cloud environment, which can be deployed on same virtual machine. Containers are a much lighter way of deploying multiple workloads on the same host operating system; they also use far less memory and start up more quickly than VMs. The advantages of containers, compared to VMs, significantly contribute to the efficiency gains needed by NFV.

For these reasons, telecom operators are interested in the potential of containers for NFV and in last years some projects have already started developing the so called "Cloud-native Network Functions"(CNFs), i.e. network functions developed appropriately to work as containers in cloud environment. Besides the creation, CNFs need to be integrated in cloud environment: this aspect is nowadays the most problematic one, and telecom operators are trying to address the problems raised by requirement.

Kubernetes (K8s) is nowadays the most used and popular orchestrator for cloud native services. It is an open source container orchestration engine for automating

deployment, scaling, and management of containerized applications. With the intent to understand if Kubernetes' functionalities allow the integration of VNFs, K8s' networking was initially analyzed in this thesis. The analysis did not bring to a solution, meaning that Kubernetes does not support natively integration of NFV, mainly for two reasons:

- lack of multiple network interfaces' configuration on containers: VNFs requires two network interfaces to properly work. Kubernetes should configure these interfaces in containers that implement VNFs, but it does not provide this behavior.
- unsupported service chain implementation: each VNF provides a different network function, which in most cases needs to to be linked to other network functions. In this way a service chain, i.e. a chain of VNFs, is created, so that traffic is elaborated by all the VNFs of the chain in the precise order with which VNFs are disposed in the chain. Again, Kubernetes does not natively support this behavior.

In the researches carried out to find a possible implemented solution, the project Network Service Mesh (NSM) was the most valid and, for this reason, the one that was analyzed in depth in this thesis. To understand how this solution behaves also in terms of performance, various tests were carried out, and also comparison with different scenarios which do not involve NSM is provided in following chapters.

Chapter 2

Background

In order to give a basic comprehension of the context of work of this thesis, this chapter provides a brief description of technologies that were used in this thesis.

2.1 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.

Figure 2.1 shows a graphic representation of main elements constitutive elements in Kubernetes:

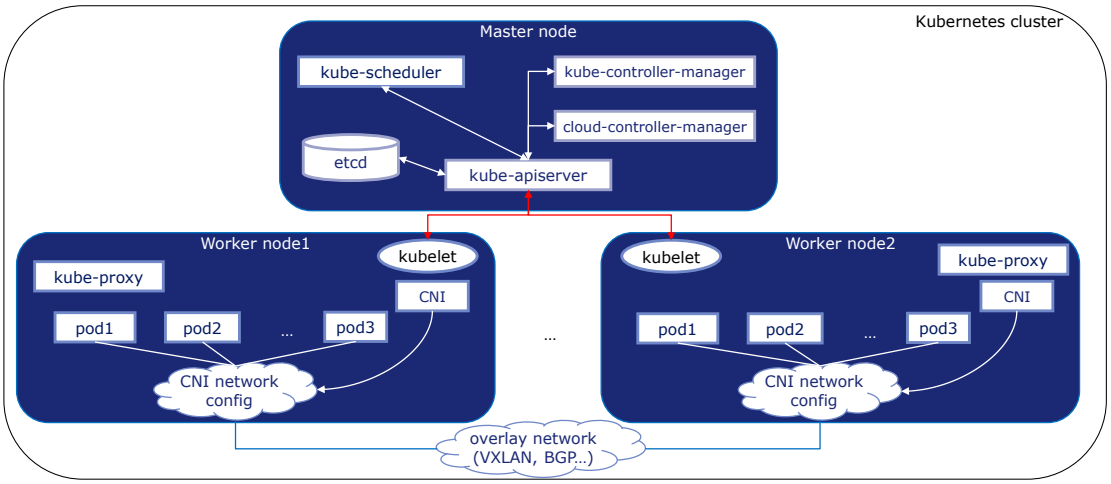

Figure 2.1: Kubernetes main constitutive elements

First of all, when Kubernetes is deployed, it is represented by a cluster. A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications [1]. A node can be arbitrarily a physical machine (i.e. a server), or a virtual machine in a server.

Every cluster is always composed by a master node and at least one worker node. The worker node(s) host the basic element in Kubernetes, which is called Pod. A Pod is the basic execution unit of a Kubernetes application: it is the smallest and simplest unit in the Kubernetes object model that can be created or deployed. A Pod represents processes running on the cluster. A Pod encapsulates an application's container (or, in some cases, multiple containers), storage resources, a unique network IP, and options that govern how the container(s) should run. A Pod represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container or a small number of containers that are tightly coupled and that share resources.

Master node hosts pods in charge of Kubernetes control plane functionalities. The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod when a deployment's replicas field is unsatisfied). Control Plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all Control Plane components on the same machine, and do not run user containers on this machine. Here follows a brief description of master node components:

- **kube-apiserver**: the API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

- **etcd**: this element is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

- **kube-scheduler**: Control Plane component that watches for newly created pods with no assigned node, and selects a node for them to run on.

- **kube-controller-manager**: Control Plane component that runs controller processes. Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

- **cloud-controller-manager**: this element runs controllers that interact with the underlying cloud providers.

On the other side, node components run on every node, maintaining running pods and providing the Kubernetes runtime environment:

- **kubelet**: an agent that runs on each node in the cluster. It makes sure that containers are running in a pod.
- **kube-proxy**: kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. Kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.
- **Container Network Interface (CNI)**: consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins [2]. It is in charge of providing all kinds of connectivity in the cluster: intra-node pod-to-pod connectivity, inter-node pod-to-pod connectivity, outside cluster-to-pod connectivity and vice versa. In the interest of this thesis, further analysis of CNI will be provided in next chapters.

When interacting with the cluster, user gives specific command, which are translated in Kubernetes REST API requests that are sent to kube-apiserver on master node. Depending on the kind of request, the kube-apiserver can interact with other control plane components on master node, for example forwarding the request to them so that they can elaborate it, or interact directly with worker nodes. The interaction with worker nodes is made possible by action of the kubelet, which receives and reply to messages from kube-apiserver. There is never direct interaction from kube-apiserver to pods deployed in worker nodes, communication is always mediated by the kubelet.

2.2 NFV

Network functions (NFs) underwent a deep transformation under last decades. First of all, it useful to define what is a network function: in its traditional implementation, in consists in a physical block (i.e. a hardware component) to which two network interfaces are associated. One interface is used to get the internet traffic inside the network function and the other is used to get traffic out of it. The heart of the network function is the software that implement the functionalities of it: it runs inside the hardware box associated to the network function and it elaborates the traffic that gets inside the network function. Once the traffic is elaborated, it gets out from the network function and continues its journey (possibly going to another network function).

In the first version of network functions, this structure was strictly respected: NFs were composed by a dedicated hardware component on which software implementation of specific network function run. Service chains of network functions were created by physically placing each "box" of a network function one after another and physically connecting interfaces of these boxes together: in this way traffic passed through each network function in the order they where physically placed. Figure 2.2 gives a graphic representation of this scenario:

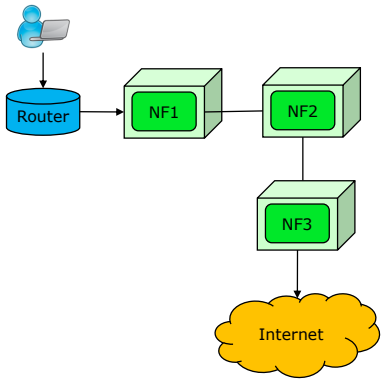

Figure 2.2: Service chain of network functions

This implementation presents some problems in terms of management and control of network functions. Moreover, most of the times dedicated hardware and implemented software were proprietary, i.e. they were produced and provided by the same vendor, making interoperability of network functions provided by different vendors hard. As software implementation was specific for hardware component, it was also difficult to have hardware of a vendor with a software of another vendor,

making telecom operators' choice to buy from a vendor binding.

To solve this situation, Network Function Virtualization (NFV) caught on. It basically consists of decoupling hardware dedicated components from software implementation of network function. In this way, network functions are deployed as Virtual Machines on a common hardware infrastructure, which can be a physical server, and, for this reason, they are called Virtual Network Functions: the environment of work of each VNF is not a specific hardware for each of it, but more VNFs can be deployed and executed on same machine. Each VNF has its own VM where it is executed and can be linked together with other VNFs on the same server. If VNF needs to receive or send traffic outside the server, one interface of the VM in which it is executed is linked to a physical NIC of the server. If VNFs that are deployed on different physical servers need to be chained together, network components (such as switches or routers) can be specialized to provide the required chaining. Figure 2.3 fives a graphic representation of the described scenario:

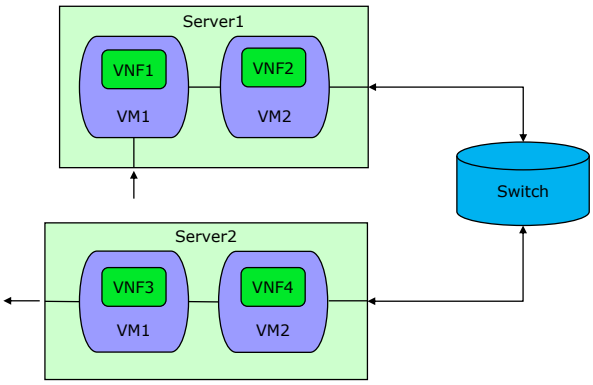

Figure 2.3: Service chain of virtual network functions

Network Function Virtualization solved the problem of strict binding between hardware and software components of a Network Function. On the other hand, providing the right intelligence to other components on the network to allow chaining of VNFs on different servers and the management itself of these VNFs introduced the necessity of a management unity that provides proper functionalities to implement this behavior. For this reason, Software Defined Networking (SDN) concept is strictly related to NFV: briefly, it allows to create a centralized control plane unity that can manage and gives proper instructions to network components which implements the data plane. For NFV, it means that SDN can be properly used to instruct network components involved in chaining of VNFs to forward traffic between elements involved in the chain (which can be other network components or servers where VNFs are deployed).

2.3 VPP

VPP technology has been used in this thesis and, for this reason, a description of this technology is provided in this section.

The VPP platform is an extensible framework that provides out-of-the-box production quality switch/router functionality. It is the open source version of Cisco's Vector Packet Processing (VPP) technology: a high performance, packet-processing stack that can run on commodity CPUs [3]. VPP is a key component of FD.io - The Fast Data Project of Linux Foundation. VPP's name comes from the fundamental way of processing packets. Regular scalar packet processing processes one packet at the time; in contrast VPP processes packets in vectors, which are essentially lists of packets. This reduces overhead and increases cache performance. VPP is especially attractive for call session data storage applications, since it also has an user space TCP stack.

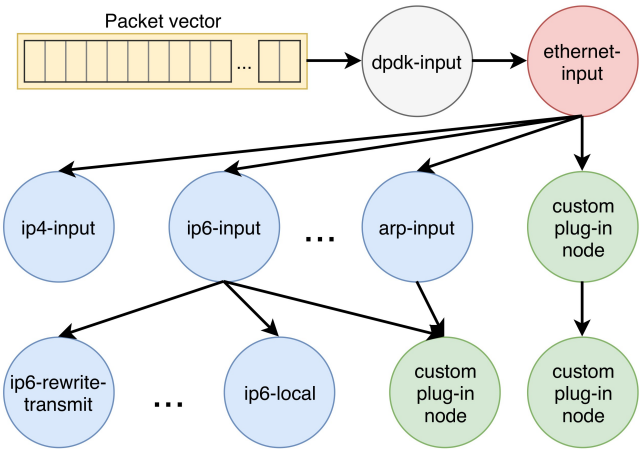

Figure 2.4: Example of VPP graph processing [4]

The core of VPP is the packet processing graph illustrated in figure 2.4. Vectors of packets come into the system and are processed according to the graph. If packets are not all the same, the vector is split in different nodes. All the basic processing in VPP is done in vSwitch/vRouter, but due to the high modularity users can create their own nodes to process the packets. As is apparent from figure 2.4, DPDK is a common way to attach the NICs in the VPP.

One of the main performance gains of the VPP comes from better use of caches. Modern CPUs have multiple caches, which differ in latency and size. The L1 cache is the smallest and fastest, so ideally it is used as much as possible while running a program. The L1 cache is divided into instruction cache and data cache. In

case of packet processing the instruction cache is the most important factor. If a packet is processed in a scalar manner, so many instructions are used for processing the packet that when the next one starts the needed instructions are not in the L1 cache anymore. This is an example of cache trashing. With vector packet processing, where processing is done for all packets in the vector, one node at the time, L1 cache usage is much more efficient, which leads to better performance.

VPP is also provided with a functional Host-stack, which can be used through many different APIs, but it is also under refactoring and the APIs might be changing in the process. In the current VPP version there are three main ways to communicate with the Host-stack from a third-party application. Lowest level of these is a binary API, which requires considerable integration work for it to be used with the application, and it is not intended to be used directly for that. VPP Communication Library (VCL) offers a higher-level API to build software against. VCL is currently the recommended way of using the VPP Host-stack. The last option is using VCL with LD_PRELOAD. This method does not require integration work, but does have a very limited compatibility with applications.

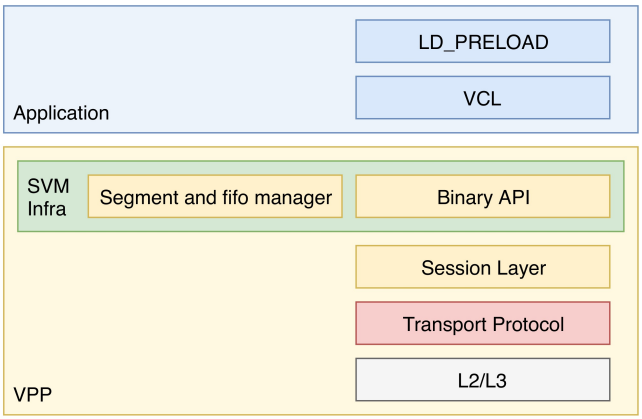

Figure 2.5: VPP host stack [4]

Raw session layer binary API is the lowest level API provided in the VPP Host-stack. It does not provide support for asynchronous communication. It is not meant to be used for integrating applications to VCL. This can be done if the limited functionality is not a problem from the standpoint of the application, but it is not recommended and there are no guarantees on stability of the API. VPP Communication Library (VCL) API is the most modern and recommended way to leverage the VPP Host-stack. It does provide its own implementations of Linux I/O event notification facility epoll which provides more support for integration work. VCL also has a POSIX API which can be used through LD_PRELOAD.

LD_PRELOAD allows a user to load any libraries before others when starting applications in Linux. This enables the user to replace any function provided by other libraries. This way LD_PRELOAD can be used to link an application to VCL without changing the application binary as long as it uses the POSIX API. This is a key point in VPP technology as it is the only acceleration framework which supports integration without changing the software.

In the context of this thesis, VPP Host-Stack and VPP application are useful in the deployment of Virtual Network Functions in Kubernetes environment. VPP acts in the pod which implements VNF as VPP Agent. The VPP Agent is a Go implementation of a control/management plane for VPP based cloud-native Virtual Network Functions. The VPP Agent is built on top of CN Infra, a framework for developing cloud-native VNFs (CNFs) [5].

The VPP Agent can be used as-is as a management/control agent for VNFs based on off-the-shelf VPP, or as a framework for developing management agents for VPP-based CNFs. The VPP Agent is basically a set of VPP-specific plugins that use the CN-Infra framework to interact with other services/microservices in the cloud. The VPP Agent exposes VPP functionality to client apps via a higher-level model-driven API. Clients that consume this API may be either external (connecting to the VPP Agent via REST, gRPC API, Etcd or message bus transport), or local Apps and/or Extension plugins running on the same CN-Infra framework in the same Linux process.

Figure 2.6 shows the VPP Agent in context of a cloud-native VNF, where the VNF's data plane is implemented using VPP/DPDK and its management/control planes are implemented using the VNF agent:

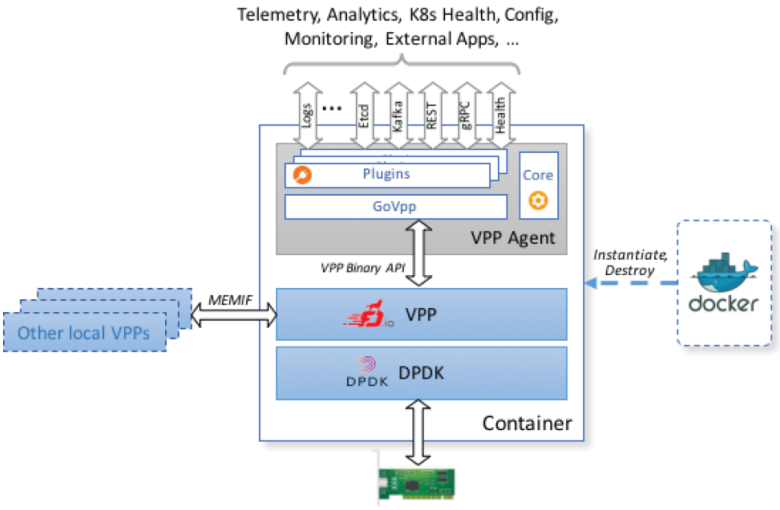

Figure 2.6: Example of VPP Agent [5]

2.3.1 DPDK

VPP is strongly integrated with DPDK technology. DPDK is the Data Plane Development Kit that consists of libraries to accelerate packet processing workloads running on a wide variety of CPU architectures. DPDK consists of a set of libraries and drivers for fast packet processing. It runs mostly in Linux user space and supports a wide variety of hardware. One of the key ideas in DPDK is poll mode driver (PMD). In its essence, PMD means moving from traditional interrupt-centric way of sending to constantly polling for new packets instead. Polling is not a new invention, but only recent advancements on hardware has made it a plausible candidate for sending and receiving traffic. To do efficient polling-based transmit, at least one CPU core has to be fully allocated to PMD only. Any interrupt by other applications or kernel slows DPDK down and causes jitter. With PMD packets are directly pulled to the user space from the rx_ring buffer of the NIC. Also transmit is done directly from the user space to tx_ring buffer. Figure 2.7 shows the PMD receiving flow:

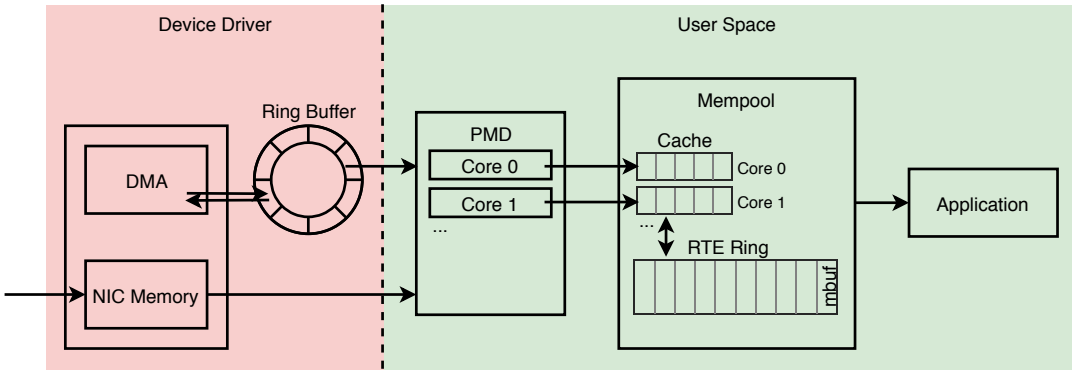

Figure 2.7: DPDK PMD packet processing [4]

DPDK includes three major libraries: `Mbuf`, `Mempool` and `Ring`. The mbuf library provides the structure to save packets to in DPDK. It can be generalized to be corresponding to the sk_buff structure of the Linux kernel. Mbufs are stored in mempool which does allocation of the space from memory. Mempool handles are usually based on a ring buffer implemented in the ring library. Mempool also implements a small cache which has a table of pointers for each core. This way, recent packets are even faster to process further. Multiple DPDK cores can share mempool, but caches are only accessed by the core which owns the cache table. If there is only one interface which DPDK is receiving data from, it only needs one core. However, if there are multiple interfaces or NIC rx/tx buffers, each needs its own processing core.

2.3.2 MEMIF interfaces

As VPP technology purely works in user space, it uses MEMIF as network interfaces. Shared memory packet interface (MEMIF) provides high performance packet transmit and receive between user application and Vector Packet Processing (VPP) or multiple user applications. Using `libmemif`, user application can create shared memory interface in master or slave mode and connect to VPP or another application using libmemif. Once the connection is established, user application can receive or transmit packets using libmemif API [6].

In order to create memif connection, two memif interfaces, each in separate process, are needed. One interface in master role and other in slave role. It is not possible to connect two interfaces in a single process. Each interface can be connected to one interface at same time, identified by matching id parameter.

Memif driver uses unix domain socket to exchange required information between memif interfaces. Socket file path is specified at interface creation. If socket is used by master interface, it's marked as listener socket (in scope of current process) and listens to connection requests from other processes. One socket can be used by multiple interfaces. One process can have slave and master interfaces at the same time, provided each role is assigned unique socket.

Slave is producer and master is consumer. Memory regions, are mapped shared memory files, created by memif slave and provided to master at connection establishment. Regions contain rings and buffers. Rings and buffers can also be separated into multiple regions. For no-zero-copy, rings and buffers are stored inside single memory region to reduce the number of opened files.

Slave interface attempts to make a connection on assigned socket. Process listening on this socket will extract the connection request and create a new connected socket (control channel). Then it sends the "hello" message, containing configuration boundaries. Slave interface adjusts its configuration accordingly, and sends "init" message. This message among others contains interface id. Driver uses this id to find master interface, and assigns the control channel to this interface. If such interface is found "ack" message is sent. Slave interface sends "add region" message for every region allocated. Master responds to each of these messages with "ack" message. Same behavior applies to rings. Slave sends "add ring" message for every initialized ring. Master again responds to each message with "ack" message. To finalize the connection, slave interface sends "connect" message. Upon receiving this message master maps regions to its address space, initializes rings and responds with "connected" message. Disconnect can be sent by both master and slave interfaces at any time, due to driver error or if the interface is being deleted.

2.3.3 TUN/TAP interfaces

TUN/TAP interfaces are a feature offered by Linux that can do user space networking, that is, allow user space programs to see raw network traffic (at the ethernet or IP level) and elaborate it if necessary [7].

TUN/TAP interfaces are software-only interfaces, meaning that they exist only in the kernel and, unlike regular network interfaces, they have no physical hardware component (i.e. there is not a physical "wire" connected to them). A TUN/TAP interface can be considered as a regular network interface with the difference that, when the kernel decides to send data "on the wire", it instead sends data to some user space program that is attached to the interface (using a specific procedure). When the program attaches to the TUN/TAP interface, it gets a special file descriptor, reading from which gives it the data that the interface is sending out. In a similar fashion, the program can write to this special descriptor, and the data (which must be properly formatted) will appear as input to the TUN/TAP interface. To the kernel, it would look like the TUN/TAP interface is receiving data "from the wire".

The difference between a TAP interface and a TUN interface is that a TAP interface outputs (and must be given) full ethernet frames, while a TUN interface outputs (and must be given) raw IP packets (and no ethernet headers are added by the kernel). Whether an interface functions like a TUN interface or like a TAP interface is specified with a flag when the interface is created.

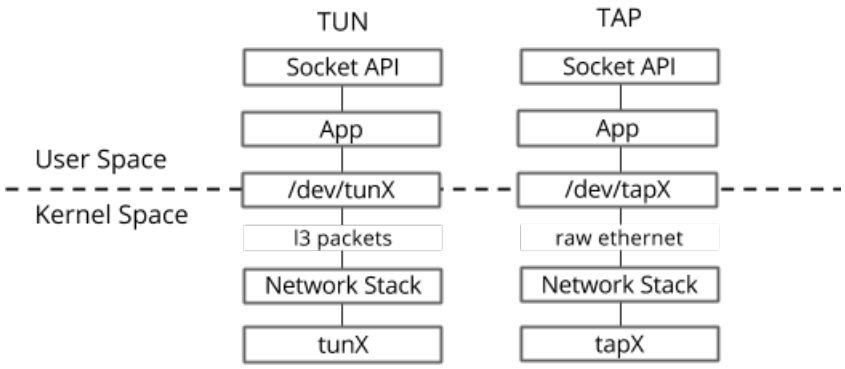

Figure 2.8: Example of TUN and TAP interfaces [8]

The interface can be transient, meaning that it's created, used and destroyed by the same program; when the program terminates, even if it doesn't explicitly destroy the interface, the interfaces ceases to exist. Another option is to make the interface persistent; in this case, it is created using a dedicated utility and then normal programs can attach to it; when they do so, they must connect using the

same type (TUN or TAP) used to originally create the interface, otherwise they will not be able to attach.

Once a TUN/TAP interface is in place, it can be used just like any other interface, meaning that IP addresses can be assigned, its traffic can be analyzed, firewall rules can be created, routes pointing to it can be established, etc.

Chapter 3

Networking in Kubernetes

As first step in trying to solve this problem, a study of already existing tools was carried out, to see if already implemented and working solutions were present. In particular, as the problem consists of integration of VNFs in Kubernetes environment, K8s' networking system was analyzed.

In Kubernetes, networking is provided by the so called Container Network Interface (CNI). It consists of a set of APIs that are introduced in K8s cluster and are used to provide network connectivity to pods of the cluster. Apart from specific cases, one CNI can be used inside the cluster: once the CNI is applied with specific command, the network configuration provided by that CNI is established to all elements of the cluster.

There are some specific APIs that compose the CNI which are fundamental for correct implementation of the CNI. This APIs are grouped together under under a unique file called CNI plugin. A CNI plugin is responsible for inserting a network interface into the container network namespace (e.g. one end of a veth pair) and making any necessary changes on the host (e.g. attaching the other end of the veth into a bridge). It should then assign the IP to the interface and setup the routes consistent with the IP Address Management section by invoking appropriate CNI plugin [9]. In this sense, different CNI plugin exist and must be invoked in a precise order to allow the correct configuration of the network interface: the output of a CNI plugin application can be given as input to another specific CNI plugin.

The operations that CNI plugins must support are:

- **ADD**: this function adds container to network. It takes some parameters about container (container ID, network namespace path, network configuration etc.) as input and, as result, it configures a network interface and all necessary components in the container so that it can correctly use the network configuration provided: this also means that, if network configuration provides a specific network policy, the pod must respect the policy).

- **DEL**: this function deletes container from network. As ADD operation, it takes some parameters (which should be the same as those passed to the corresponding add operation) as input and should release all resources held by the supplied container ID in the configured network. As result, pod is not present on the network configuration, i.e. it cannot reach other pods and it cannot be reached by other pods.
- **CHECK**: this operation checks container's networking is as expected (i.e. it respects configuration and policies established in the network).

Some notes about CNI plugins:

- The container runtime must create a new network namespace for the container before invoking any plugins.
- The runtime must then determine which networks this container should belong to, and for each network, which plugins must be executed.
- The network configuration is in JSON format and can easily be stored in a file. The network configuration includes mandatory fields such as "name" and "type" as well as plugin (type) specific ones. The network configuration allows for fields to change values between invocations.
- The container runtime must add the container to each network by executing the corresponding plugins for each network sequentially.
- Upon completion of the container life cycle, the runtime must execute the plugins in reverse order (relative to the order in which they were executed to add the container) to disconnect the container from the networks.
- The container runtime must order ADD and DEL operations for a container, such that ADD is always eventually followed by a corresponding DEL. DEL may be followed by additional DELs but plugins should handle multiple DELs permissively (i.e. plugin DEL should be idempotent).
- A runtime must not call ADD twice (without a corresponding DEL) for the same (network name, container id, name of the interface inside the container). This implies that a given container ID may be added to a specific network more than once only if each addition is done with a different interface name.

CNI can also provide a specific behavior also for network policy implemented in the cluster. Kubernetes imposes that the basic policy that each CNI must provide is that all pods can reach each other inside the same cluster. CNI must implement this behavior, but CNI can provide more elaborated network policy associated to the network configuration.

3.1 Analysis of different CNIs

Nowadays there are different kind of CNIs, each one providing a particular behavior to allow networking inside the cluster. The one described in following sections has been more specifically analyzed to properly understand their different behaviors and see if they might provide useful function for integration of VNFs in Kubernetes.

To properly understand next sections, it is necessary to point out the meaning of these concepts [10]:

- **Layer-2 networking**: The "data link" layer of the OSI (Open Systems Interconnection) networking model. Layer 2 deals with delivery of frames between two adjacent nodes on a network. Ethernet is a noteworthy example of Layer-2 networking, with MAC represented as a sublayer.

- **Layer-3 networking**: The "network" layer of the OSI networking model. Layer 3's primary concern involves routing packets between hosts on top of the layer 2 connections. IPv4, IPv6, and ICMP are examples of Layer 3 networking protocols.

- **VXLAN**: this term stands for "virtual extensible LAN". Primarily, VXLAN is used to help large cloud deployments scale by encapsulating Layer-2 Ethernet frames within UDP datagrams. VXLAN virtualization is similar to VLAN, but offers more flexibility and power (VLANs were limited to only 4,096 network IDs). VXLAN is an encapsulation and overlay protocol that runs on top of existing networks.

- **Overlay network**: An overlay network is a virtual, logical network built on top of an existing network. Overlay networks are often used to provide useful abstractions on top of existing networks and to separate and secure different logical networks.

- **Encapsulation**: Encapsulation is the process of wrapping network packets in additional layer to provide additional context and information. In overlay networks, encapsulation is used to translate from the virtual network to the underlying address space to route to a different location (where the packet can be de-encapsulated and continue to its destination).

- **BGP**: Stands for "border gateway protocol" and is used to manage how packets are routed between edge routers. BGP helps figure out how to send a packet from one network to another by taking into account available paths, routing rules, and specific network policies. BGP is sometimes used as the routing mechanism in CNI plugins instead of encapsulated overlay networks.

3.1.1 Bridge CNI

The first basic implementation of CNI is bridge CNI. With bridge CNI plugin, all containers (on the same host) are plugged into a bridge (virtual switch) that resides in the host network namespace. The containers receive one end of the veth pair with the other end connected to the bridge. An IP address is only assigned to one end of the veth pair (one residing in the container).

The bridge created by the CNI can be configured to work in two different way:

- **L2-only mode**: in this case the bridge would need to be bridged to the host network interface. Inter-node communication between pods happens through ARP and L2 switching, and it is based on underlay physical network.

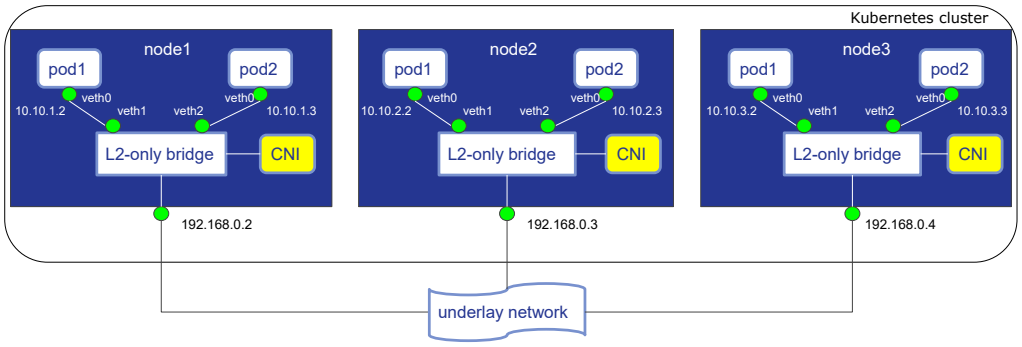

Figure 3.1: L2-only mode bridge

- **L3 mode**: an IP is also assigned to the bridge, turning it into a gateway for the pods. Also in this case Inter-node communication between pods is based on underlay physical network, but it will use L3 routing instead of L2 switching.

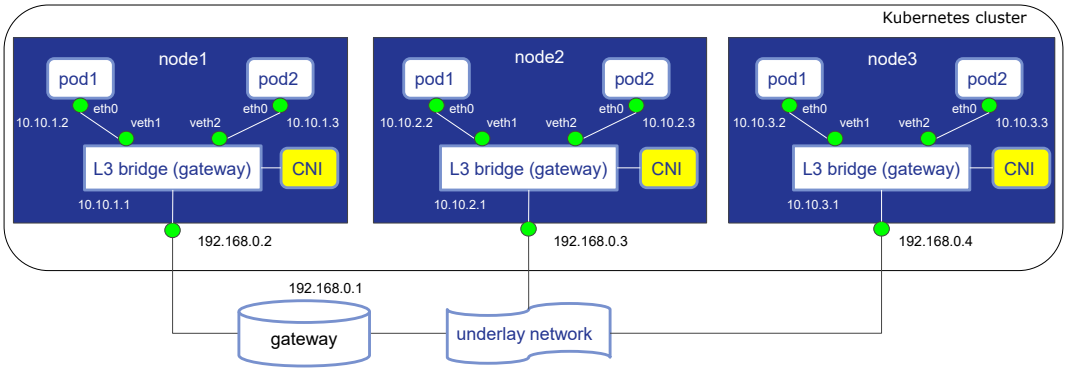

Figure 3.2: L2-only mode bridge

The network configuration specifies the name of the bridge to be used. If the bridge is missing, the plugin will create one on first use and, if gateway mode is used, assign it an IP that was returned by IPAM plugin via the gateway field.

Speaking about network configuration, it is represented by a YAML file: when the specific CNI must be installed in Kubernetes cluster, it is installed applying the proper command in Kubernetes. In this case, the network configuration can be as the following example [11]:

```
{
 "cniVersion": "0.3.1",
 "name": "mynet",
 "type": "bridge",
 "bridge": "mynet0",
 "isGateway": true,
 "isDefaultGateway": true,
 "ipMasq": true,
 "ipam": {
     "type": "host-local",
     "subnet": "10.10.0.0/16"
 }
}
```

Each field has a different meaning and is useful to provide detailed information to the network configuration:

- name: the name of the network.
- type: defines the type of CNI that is to be applied.
- bridge: name of the bridge to use/create.
- isGateway: assign an IP address to the bridge. Defaults to false.
- isDefaultGateway: Sets isGateway to true and makes the assigned IP the default route. Defaults to false.
- ipMasq: set up IP Masquerade on the host for traffic originating from this network and destined outside of it. Defaults to false.
- ipam (dictionary, required): IPAM configuration to be used for this network. For L2-only network, create empty dictionary.

Bridge CNI does not explicitly manage inter-node communication. In facts, as shown in previous figures, networking between different nodes relies on underlay network, i.e. to network elements that lead from a node to another.

3.1.2 Flannel CNI

Flannel is one of the most mature examples of networking fabric for container orchestration systems, intended to allow for better inter-container and inter-host networking. Flannel can use the Kubernetes cluster's existing etcd cluster to store its state information using the API to avoid having to provision a dedicated data store [10].

Flannel CNI is responsible for providing a layer 3 IPv4 network between multiple nodes in a cluster. Within a single Kubernetes node, it runs as a DaemonSet called flanneld, which is responsible for creating the CNI network configuration for communication between pods inside the same node. In particular, it spans a virtual bridge and a virtual router inside the node: these two elements collaborate to provide communication of pods inside the same node.

As CNI specifics impose, also Flannel CNI comes with a CNI plugin, which implements the behavior of fanneld. When flanneld is started, it outputs a /run/flannel/subnet.env file that looks like this [12]:

```
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.17.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=true
```

This information reflects the attributes of flannel network on the node. The flannel CNI plugin uses this information to configure another CNI plugin, such as bridge plugin. Given the following network configuration file and the contents of /run/flannel/subnet.env above,

```
{
  "name": "mynet",
  "type": "flannel"
}
```

the flannel plugin will generate another network configuration file:

```
{
  "name": "mynet",
  "type": "bridge",
  "isGateway": true,
  "ipam": {
     "type": "host-local",
     "subnet": "10.1.17.0/24"
  }
}
```

It will then invoke the bridge plugin, passing it the generated configuration. Same procedure (with appropriate parameters) is applied to configure router inside each node. These two elements are linked together to provide correct traffic routing inside each node.

Considering communication between pods across different nodes, large internal network is created that spans across every node within the cluster. Within this overlay network, each node is given a subnet to allocate IP addresses internally. Fanneld provides encapsulation functionalities. When virtual bridge configured on a node receives traffic sent from a pod in the same node that needs to be forwarded to a pod on a different node, that traffic is first sent to flanneld, wich encapsulates it for routing to appropriate destination. The router receives this traffic and, thanks to encapsulation parameters (i.e. IP addresses), it understands that the traffic needs to be sent on the overlay network so that it reaches appropriate node.

Flannel has several different types of backends available for encapsulation and routing. The default and recommended approach is to use VXLAN, as it offers both good performance and is less manual intervention than other options.

Figure 3.3 gives a graphic representation of how Flannel CNI is deployed in Kubernetes:

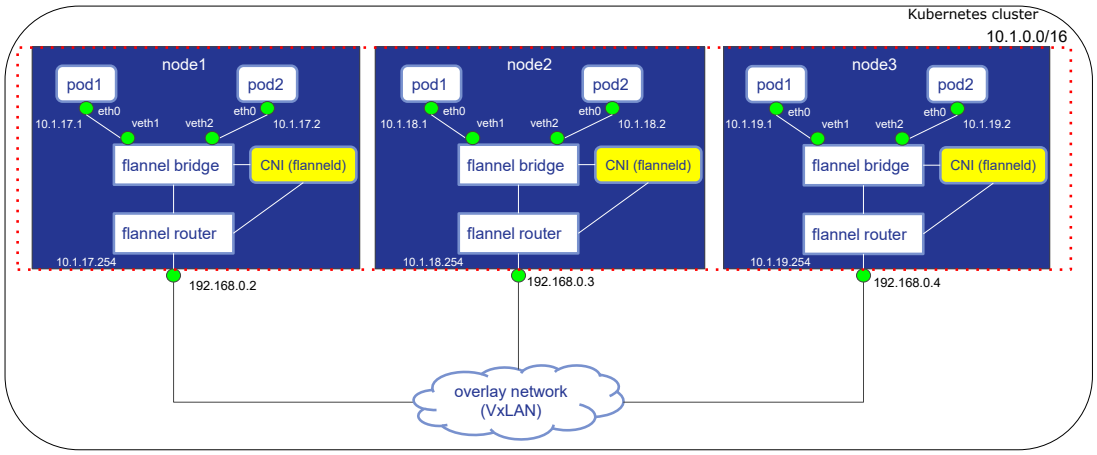

Figure 3.3: Flannel CNI

Speaking about network policy, Flannel does not offers any specific network configuration to provide a different policy from the standard one (i.e. each pod can communicate and reach each other pod inside the cluster).

3.1.3 Calico CNI

While Flannel is positioned as the simple choice, Calico is best known for its performance, flexibility, and power. Calico does not concern itself only with providing network connectivity between nodes and pods, but also with network security and administration [10]. All these characteristics are made possible by the fact that Calico implements its own network policy, which substitutes the Kubernetes default one.

An example of Calico CNI's YAML file which is applied in installation of this CNI is the following [13]:

```
{
    "name": "any_name",
    "cniVersion": "0.1.0",
    "type": "calico",
    "policy": {
      "type": "k8s"
    },
    "kubernetes": {
        "kubeconfig": "/path/to/kubeconfig"
    },
    "ipam": {
        "type": "calico-ipam"
    }
}
```

Differently from previous described CNIs, as Calico supports network policy specification, the field "policy" is introduced in this file. The value of this field represents the type of policy that is applied in the Kubernetes cluster. This field needs to be followed by "kubernetes" field, whose value indicates the path where to find the configuration that need to be applied for network policy.

To provide network connectivity to pods within the same node, Calico CNI installs three fundamental elements in each node of the cluster:

- **Calico node**: This is the agent that will run inside each node. It includes Bird, Felix and a few other helper processes.
- **Bird**: it is a per node BGP daemon that exchanges route information with BGP daemons running on other nodes. It in fact interacts with routing table of the host (or WM) where the node is running.
- **Felix**: it is another per-node daemon that is used to configure routes and enforce network policies on the node it is running. It interacts with IP tables of the host (or VM) where the node is running, implementing the required network policy.

These elements collaborate to provide the network policy that is specified in the YAML file of the network configuration. Figure 3.4 gives a graphic representation of Calico CNI deployed in a cluster:

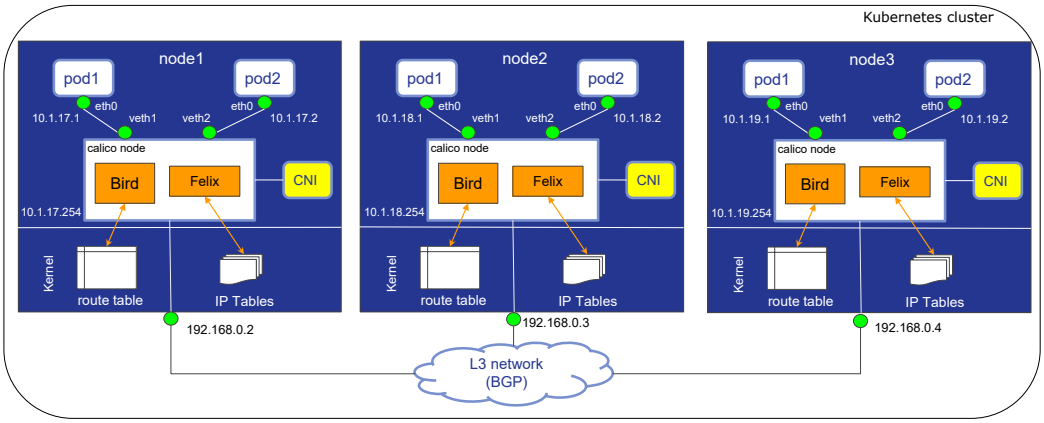

Figure 3.4: Calico CNI

Speaking about inter-node network connectivity, unlike Flannel, Calico does not use an overlay network. Instead, Calico configures a Layer-3 network that uses the BGP routing protocol to route packets between nodes. This means that packets do not need to be wrapped in an extra layer of encapsulation when moving between nodes. The BGP routing mechanism can direct packets natively without an extra step of wrapping traffic in an additional layer of traffic. In this optic, calico node and its internal elements are in charge of appropriately routing traffic to destination node. In particular, Bird component, thanks to routing messages exchanged with other Bird components in other nodes, knows where to route the traffic inside the cluster and actually sends traffic outside the node.

As mentioned before, network policy customization is a key feature in Calico CNI. Once specific network policy rules are defined, they are applied within the cluster where the CNI is applied: this means that it is possible to configure powerful rules describing how pods should be able to send and accept traffic (such as namespace isolation specific rules), improving security and control over your networking environment.

Moreover, an intrinsic characteristic of Calico CNI is that it allows for more conventional troubleshooting when network problems arise. While encapsulated solutions using technologies like VXLAN work well, the process manipulates packets in a way that can make tracing difficult. With Calico, the standard debugging tools have access to the same information they would in simple environments, making it easier for a wider range of developers and administrators to understand behavior.

3.1.4 Multus CNI

As the name suggests, Multus CNI allows K8s pods to be multi-homed. The Multus CNI plugin can be used in conjunction with other CNI plugins (e.g. Flannel, Calico, etc.). The Multus CNI plugin can also work with different internet protocol address management (IPAM) configurations and networks [14].

In its basic operation, Multus CNI groups multiple plugins into delegates and invokes each plugin in sequential order according to the CNI configuration file. A possible configuration file of Multus CNI can be represented by following code:

```
{
  "cniVersion": "0.1.0"
  "type": "multus",
  "delegates": [
    {
        "type":"flannel",
        "masterplugin": true,
        "delegate": {
            "isDefaultGateway": true
            "ipam": {
                "type": "host-local",
                "subnet": "10.1.17.0/24"
            }
        }
    }
    {
        "type":"calico",
        "masterplugin": true,
        "delegate": {
            "policy": {
                "type": "k8s"
            },
            "kubernetes": {
                "kubeconfig": "/path/to/kubeconfig"
            }
        }
    }
  ]
}
```

When the Multus CNI plugin is enabled, a “master plugin” gets instantiated. This master plugin is responsible for managing the “main” interface of the pod, i.e. it identifies the primary network and sets the default route via this network. This is the only network configuration option of the Multus plugin. The other plugins

are referred to as minion plugins (indicated in the code as “delegates”): they are responsible for configuring other interfaces in the pod. Naturally, there can only be one master plugin, and this is specified in the Multus configuration file using the tag “"masterplugin": true” in the minion plugins’ configuration, as shown in the example code above.

When a pod is scheduled in a node, the kubelet of that node activates to instantiate the pod. In the process of deployment, it invokes the master plugin of Multus CNI, which acts like a “meta plugin” for other minion plugins: besides configuring default network interface in the pod , it calls minion plugins, which are in charge of configuring proper network interfaces in the pod: in the example, Flannel plugin configures a veth pair between the pod and flannel bridge, while Calico plugin configures a veth pair (along with correct entries inside routing table and IP tables for implementation of required network policy) between pod and Calico node. In this way, the pod is able to work with both implementation of CNIs. Note that a “annotations” field in the YAML file of the pod must be provided, otherwise this process cannot work properly. Referring to Multus CNI configuration file above, an example of this file can be represented as follow:

```
annotations:
    k8s.v1.cni.cncf.io/networks: flannel, calico
```

In this way, the kubelet knows that the pod must be deployed with interfaces related to the network configurations specified in this field.

Figure 3.5 provides a graphic representation of the scenario:

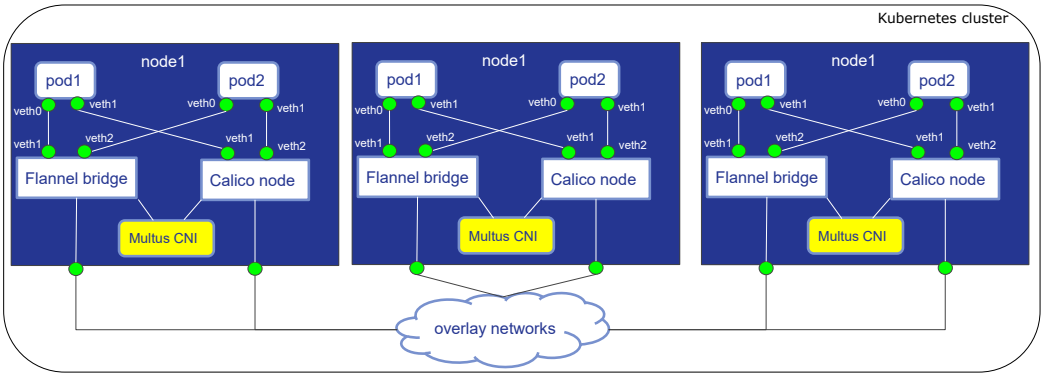

Figure 3.5: Multus CNI

In terms of inter-node communication, Multus CNI allows configuration of different implementations, on the basis of minion CNIs. In this way, even in inter-node communication each CNI can work separately.

3.1.5 Contiv-VPP CNI

Contiv-VPP CNI is a Kubernetes network plugin that uses FD.io VPP to provide network connectivity between pods in a K8s cluster. It deploys itself as a set of system pods, some of them on the master node, and some of them on each node in the cluster [15].

Contiv-VPP is fully integrated with K8s via its components, and it automatically reprograms itself upon each change in the cluster via K8s API.

The main component of the solution, VPP, runs within the contiv-vswitch pod on each node in the cluster and provides pod-to-pod connectivity across the nodes in the cluster, as well as host-to-pod and outside-to-pod connectivity. While doing that, it leverages VPP's fast data processing that runs completely in user space and uses DPDK for fast access to the network IO layer.

Kubernetes services and policies are also reflected into VPP configuration, which means they are fully supported on VPP, without the need of forwarding packets into the Linux network stack (i.e. to kube-proxy).

Contiv/VPP consists of several components, each of them packed and shipped as a container. First three following components are deployed on Kubernetes master node only, while the other are part of woker nodes:

- **Contiv KSR**: it is an agent that subscribes to K8s control plane, watches K8s resources and propagates all relevant cluster-related information into the Contiv ETCD data store. Other Contiv components do not access the k8s API directly, they subscribe to Contiv ETCD instead.

- **Contiv CRD**: this element handles K8s Custom Resource Definitions defined in k8s API and processes them into configuration in Contiv ETCD. Currently it covers Contiv-specific configuration of individual k8s nodes such as IP address and default gateway, etc. Apart from this functionality, it also runs periodic validation of the topology, and exports the results as another CRD entry. The contiv-netctl tool which sits in the same Docker container can be used to explore runtime state of the cluster, such us current IPAM assignments, VPP state etc., or to execute a debug CLI on any of the VPPs in the cluster.

- **Contiv ETCD**: Contiv/VPP uses its own instance of ETCD database for storage of k8s cluster-related data reflected by KSR, which are then accessed by Contiv vSwitch Agents running on individual nodes. Apart from the data reflected by KSR, ETCD also stores persisted VPP configuration of individual vswitches, as well as some more internal metadata.

- **Contiv vSwitch** : vSwitch is the main networking component that provides the connectivity to pods. It deploys on each node in the cluster, and consists of two main components packed into a single container: VPP and Contiv VPP

Agent. VPP is the data plane software that provides the connectivity between pods, host Linux network stack and data-plane NIC interface controlled by VPP. Pods are connected to VPP using TAP interfaces wired between VPP and each pod, host network stack is connected to VPP using another TAP interface connected to the host network namespace, data-plane NIC is controlled directly by VPP using DPDK. Note that this means that this interface is not visible to the host Linux network stack, and the node either needs another management interface for k8s control plane communication, or STN (Steal The NIC) deployment must be applied. Contiv VPP Agent is the control plane part of the vSwitch container. It is responsible for configuring the VPP according to the information gained from ETCD and requests from Contiv STN.

- **Contiv STN (Steal The NIC) Daemon**: as already mentioned, the default setup of Contiv/VPP requires two network interfaces per node: one controlled by VPP for data facing PODs and one controlled by the host network stack for k8s control plane communication. In case that your k8s nodes do not provide two network interfaces, Contiv/VPP can work in the single NIC setup, when the interface will be "stolen" from the host network stack just before starting the VPP and configured with the same IP address on VPP, as well as on the host-VPP interconnect TAP interface, as it had in the host before it.

Figure 3.6 provides a graphic representation of the scenario:

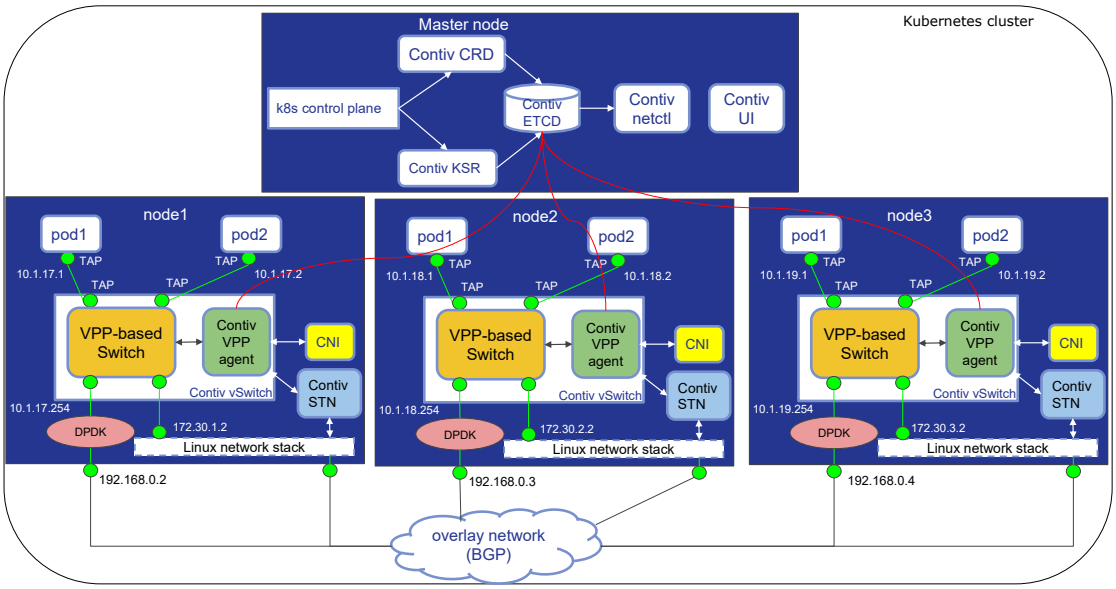

Figure 3.6: Contiv-VPP CNI

3.2 Limitations

Even if all described CNI do provide network connectivity inside Kubernetes cluster and, in some cases, can also provide security and administration defining specific network policies, they are not able to provide necessary behavior to allow Virtual Network Functions' integration inside Kubernetes environment. As mentioned in the introduction, the two main features needed to allow this integration are the following:

- **Multiple network interfaces on single pod**: a generic VNF needs two interfaces to work properly, one for traffic to get inside the VNF and one to get outside of it. In this optic, since in Kubernetes environment a VNF would be implemented in a pod, the latter needs to have the two interfaces mentioned before, to allow correct behavior of the VNF.

- **Service chaining implementation**: VNFs need to be linked together to implement a service chain. The chain defines also the order of the VNFs, so that traffic passes through them in the established order. Kubernetes does not provide this behavior, which should be provided by an external element.

Figure 3.7 gives a graphic representation of the described concepts:

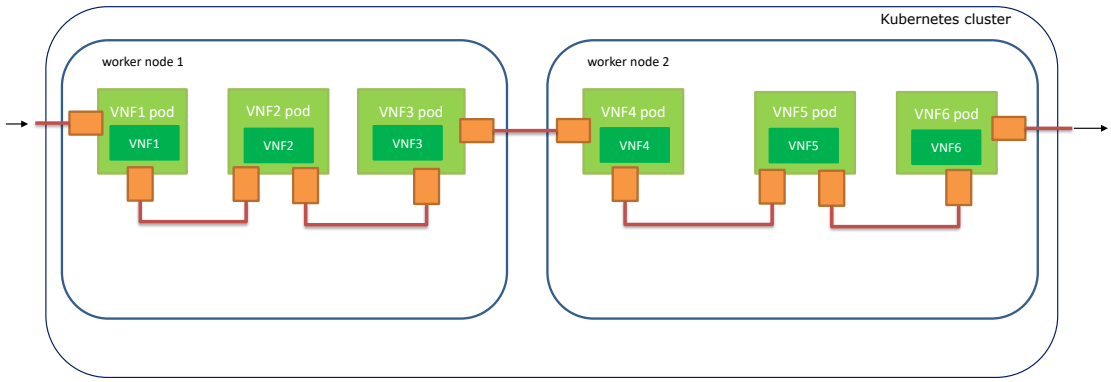

Figure 3.7: Service chain of VNFs in K8s

Apart from Multus CNI, all the described CNIs does not provide the first feature. In fact, as CNI specifics point out, a pod can be added to a network configuration only once, which means that only one interface can be configured inside a pod. Multus solves this problem assigning different network configurations in the cluster, in this way the pod is added to different network configurations and each injected

interface is separated from the other and has different associated parameters (i.e. different interface id, IP address etc.).

However, neither Multus, nor other CNIs provide any implementation for the service chain. Each CNI in fact provides connectivity between pods and deploys necessary elements to allow routing not only inside the node but also between the nodes, but it does not establish how links between pods need to be created to implement a chain of VNFs (i.e. of pods). In most cases links are composed by veth pairs created between each pod and a virtual network element (e.g. a virtual bridge) and routing rules are applied to the network configuration to allow correct routing, but this configuration cannot be specialized to make traffic traverse specific links in specific order as a service chain requires.

In this sense, Kubernetes-related resources do not provide in any way a possible integration of VNFs in Kubernetes environment without any further specific implementations.

Chapter 4

Network Service Mesh

This chapter illustrates the project Network Service Mesh [16]. Its main aim is to allow chaining of VNFs in a Kubernetes environment, which represents the main existing solution to the problem analyzed in this thesis.

4.1 Main concepts

NSM offers a system of "virtual wiring" for communication between pods. It is based on the following concepts, fundamental to understand how it works. Figure 4.1 presents a a graphical representation of the following concepts.

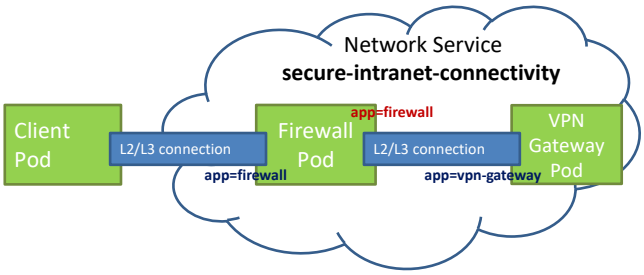

Figure 4.1: Main concepts in Network Service Mesh

Network Service

The first concept that NSM introduces is the Network Service.

First of all, it is implemented in Kubernetes as a Custom Resource Definition (CRD). Briefly, besides Kubernetes basic installation resources (such as services), Kubernetes allows to define a resource which is custom, so it is built ad-hoc and implements a specific function. Once the CRD is defined and installed in Kubernetes, it will accept any request of creation of the new defined resource. NSM defines Network Service as a CRD and installs it in Kubernetes.

A Network Service represents the logical implementation of a chain of VNFs implemented as pods in the cluster. If, like in figure 4.1, a chain of VNFs is composed by a firewall and a VPN gateway, the Network Service would represent in NSM's world the chaining of these virtual network functions: it is important to note that a Network Service also specifies the order of VNFs, so traffic that traverses the chain represented by the Network Service will follow the order of the VNFs defined in the Network Service.

Cross-connection

Another important concept to understand in order to correctly comprehend how NSM works is what in this project is defined as cross-connection. It represents the virtual Wire created between the pods involved in the chain. Referring to figure 4.1, each link between two pods is represented by a cross connection. As we will see in detail later, it is a prerogative of NSM's control plane elements to build each cross-connection, to allow proper communication between pods. It is composed by two interfaces, each of which is configured and injected in pods involved in the communication. NSM supports L2/L3 cross-connections: it means that ethernet frames or IP packets will traverse these connections. Once the cross-connection is established, traffic travels through it to go from one pod to another.

4.2 Main elements

To properly create cross-connections and allow communications between pods, NSM provides both control plane and data plane elements. Figure 4.2 gives a graphical representation of these elements.

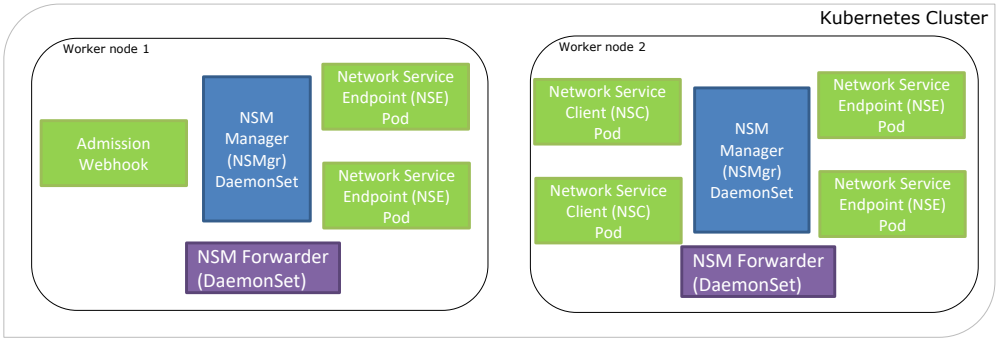

Figure 4.2: Main elements involved in Network Service Mesh

Network Service Client

The first element involved in Network Service Mesh environment is the Network Service Client. It is deployed as a pod in Kubernetes environment and its main aim is to require a cross-connection to a specific Network Service. To provide this behavior, it is composed by a NSM specific container (which can be a different container from the main one) whose main aim is to request a cross-connection to NSM Manager. This request is sent to the Manager using gRPC[1] technology: it includes all necessary parameters needed to allow the Manager understand which Network Service is required. Once the request is sent, the container awaits to receive a reply from the Manager: if, after a given timeout, it does not receives any reply, it tries to send the request again for a specific amount of time. If, after all the attepts, it does not receive any reply, or it receives a reply which however contains an error message, the container reaches an error state and it needs to be restarted to retry again to connect to the Network Service.

The process described above starts as soon as the pod is deployed in K8s, so that the cross-connection is immediately established and the main container of the NSC can communicate to the pods implementing VNFs involved in the required Network Service as soon as possible.

[1] gRPC is a protocol used to exchange messages over Linux sockets

Network Service Endpoint

This element is in charge of implementing a VNF specified in a Network Service. Referring to figure 4.1, it would be the firewall pod (as well as the VPN gateway pod) belonging to the Network Service named "secure-intranet-connectivity".

As for Network Service Client, also this pod can be composed by two containers, one implementing NSM control plane functionalities, the other implementing the required virtual network function. The first container is the one in charge of providing NSM control plane functionalities to the pod, in particular it receives request of cross-connection forwarded by NSM Manager from NSC. Once it receives the request, in may behave in two different ways: if it is not the last hop of the service chain it belongs to, it will create a new request in order to create a cross-connection to the next hop of the chain and, once it receives reply, it will reply to NS Client's original request; otherwise, if it is the last hop of the chain, it will immediately elaborate the request and create appropriate reply which it then sends to NSM Manager. This behavior is made possible by the particular implementation of its functionalities: it is in fact divided in further Endpoints, each of one representing one specific functionality. In order to implement one behavior rather than another one, it is sufficient to add or remove specific endpoints in the ones that compose the implementation. On the other side, the other container implements the virtual network function, providing the specific required behavior. It is worth to note that these two containers can work concurrently in the pod: for this reason, it may be necessary for them to coordinate, or eventually the VNF container may need additional configuration, so that the pod implementing the VNF can work properly with cross-connection built by NSM container.

However, as it will be described in next chapter, there can be NSEs that present only one container, which implements NSM control plane functionalities and includes also VNF implementation. In this case, the part of the container implementing the virtual network function does not need further configuration, as it receives direct instructions from NSM part of the container.

Network Service Manager

This element is fundamental in the control plane of Network Service Mesh environment. Its main aim is to receive and transmit all messages involved in control plane communications between elements involved in the construction of a cross-connection. These messages are implemented by NSM using gRPC technology.

First of all, it is implemented as a DaemonSet in Kubernetes environment. Briefly, in K8s when a resource is defined as DaemonSet, a pod which implements this resource is deployed in each node of the cluster. In this way, each node has its own Network Service Manager, which guarantee that each pod in a node that is part of NSM will receive control plane messages. NSM Managers can interact with

each other to exchange both direct messages and messages that need to be sent to other NSM elements in the node.

This element is composed by three different containers, which coordinate to provide proper functionalities:

- **nsmd**: this container is the heart of the NSM control plane implementation of Network Service Manager. It is in charge of elaborating all kind of requests involved in the cross-connections construction. For this reason, nsmd contains several smaller components, each of which interacts with a particular NSM elements. For sake of simplicity, only main components are described.

  The first fundamental component is the one in charge of elaborating requests from Network Service Client and Network Service Endpoint. This element is composed by a series of functionalities that analyze parameters specified in the request and, if necessary, modify it, to allow proper forwarding behavior of the request.

  Another fundamental component is the one that communicates with Network Service Forwarder: this component, starting from other requests coming from NSC and NSE, builds a request with specific parameters needed by the forwarder to work properly. It also receives and elaborates replies coming from the forwarder.

  Another fundamental component is the one in charge of monitoring cross-connection. Precisely, it registers to the Network Service Forwarder and intercepts cross-connection related messages coming from it. This is necessary to provide properly behavior to get updates about cross-connections between pods registered with NSM Manager.

- **nsmd-k8s**: this container is in charge of everything in NSM Manager concerning local and remote registries. It also communicates with Network Service Endpoints that are deployed on its same node. In particular, when the NSE starts running, it registers to its NSM Manager: the communication of this registration happens with this container, which provides necessary functionalities to register the NSE. It also communicates internally with nsmd container, to provide interaction with data stored locally and NSM managers in other K8s nodes for data stored remotely. For instance, every time nsmd needs to check if an Network Service Endpoint is deployed on the same node or in another node, it will ask the nsmd-k8s to access local registries and check if there is a NSE registered locally: in positive case, it returns immediately the registration associated to the endpoint, in negative case it will contact other NSM Managers to understand where the NSE is deployed (i.e. which NSM manager the NSE is registered with).

- **nsmdp**: this last container is in charge of checking that all the elements involved in NSM Manager functions are working properly. To do so, it interacts with them, periodically interrogating their state and, in case something is wrong, it provides necessary functionalities to correctly react to the event.

Network Service Forwarder

This element's main aim is to implement NSM data plane functionalities. As communication between pods is provided with cross-connections, it is in charge of configuring interfaces and building cross-connection between involved pods. To provide this behavior, it receives request with specific parameters about cross-connection from the NSM Manager: the forwarder analyzes the request and, if it supports the specified parameters, it builds the cross-connection. Once the cross-connection is built, the forwarder advertises the NSM Manager about it, so that the latter can replies to the Network Service Client, which will then be able to use the cross-connection to communicate with the other pod.

Network service Mesh provides two different implementation of this forwarder:

- **VPP Forwarder**: this implementation is the default one provided by NSM. In this case the forwarder behaves differently depending on the technology of the cross-connection indicated in the request that the forwarder receives from the NSM Manager. This forwarder can in fact support both "standard" technology and VPP technology and it configures different kind of interfaces basing on the technology that is implemented in the pods involved in the communication. In the analyzed cases depicted in next chapter more details about this behavior are described.

- **Kernel Forwarder**: differently from the previous implementation, in this case the forwarder supports only the construction of cross-connection which consists of a veth pair between the two pods involved in the communication.The set up of the cross-connection, the forwarder uses `NetLink` Linux library, which not only injects veth interfaces in pods, but it also configures appropriate routing table's entries to allow the reachability between involved pods. This behavior that, if the request of creation of cross-connection specifies parameters which are related to VPP technology, or in general to other technology rather than the "standard" one, the forwarder will not be able to satisfy the request and will reject it.

Admission webhook

In a Kubernetes cluster, normally, when the api-server receives a request of pod creation, it will immediately elaborate the request, sending it to the proper elements

inside the master node. However, the api-server can be configured to work in a slightly different way: when the api-server receives a request of pod's creation, instead of elaborating immediately the request, it will previously send the request to a chain of controllers (called webhooks), which are basically pods that analyze the request and can perform some action on the request itself, modifying it or even rejecting it. This controllers can be created created and configured manually, so that requests can receive custom analysis.

In this optic, NSM provides its own mutating admission webhook: this elements intercepts pod creation request to api-server and, on the basis of its internal configuration, it can modify the request, inject specific code in the YAML associated to the request.

The analysis of the request happens as follows [17]. The YAML file of the request can have in its metadata section annotations, which are key value pairs (similar to labels):

```
apiVersion: v1
kind: Pod
metadata:
  name: icmp-responder-nsc
  annotations:
    ns.networkservicemesh.io: icmp-responder
```

The annotation key, in this case, is ns.networkservicemesh.io, the value is icmp-responder. The basic form of the value is a comma-delimited list of urls for the network service:

```
${nsname}/${optionally interface to attach the ns to}?${optional list of labels}$
```

So, for example:

```
secure-intranet-connectivity/eth2?app=firewall&version=2
```

would imply a network service named secure-intranet-connectivity connected on eth2, with labels: app=firewall and version=2.

Merging this example to the full YAML above, we can let the client connect with two network services simultaneously:

```
apiVersion: v1
kind: Pod
metadata:
  name: icmp-responder-nsc
  annotations:
    ns.networkservicemesh.io:
      icmp-responder,secure-intranet-connectivity/eth2?app=firewall&version=2
```

If and only if the pod has the ns.networkservicemesh.io annotation exists, and is of the right form, and if the admission webhook is installed, then it will inject an InitContainer (according to webhook's configuration):

```
initContainers:
  - name: nsc
    image: ${REPO}/${INITCONTAINER}:${TAG}
    imagePullPolicy: IfNotPresent
    env:
      - name: NS_NETWORKSERVICEMESH_IO
        value: ${value of annotation}
    resources:
      limits:
        networkservicemesh.io/socket: 1
```

The ${INITCONTAINER} is based on the SDK's NSMClientList which parses ${NS_NETWORKSERVICEMESH_IO} and spawns the needed number of clients. ${REPO}, ${INITCONTAINER}, and ${TAG} are specifiable for the mutating admission webhook container, defaulting to REPO=networkservicemesh, INITCONTAINER=nsc, TAG=latest.

Once the pod is deployed, the InitContainer injected by the mutating admission webhook will be executed: its main aim is to send a request with the specified labels to connect to the proper Network Service.

This element is therefore necessary to allow a client pod be able to communicate with other pods involved in NSM. It is important to note that also NSEs' YAML are analyzed by this element: however, in this case the admission webhook does not find any particular annotation and it does not inject any specific code in the YAML file.

4.3 NSM use case

To provide a better understanding of how NSM works, here follows a detailed description of all the steps to deploy a chain of VNFs, from the definition of pods' YAML files involved in the chain, to how the cross-connection construction system works.

4.3.1 YAML files definition

The first step in the deployment of a service chain is the correct definition of YAMLs files of both the Network Service that represents the chain of VNFs and the pods that implement the chain itself.

Network Service's YAML

The first element that needs to be defined in service chain's deployment is the Network Service, whose definition is represented by a YAML. Figure 4.3 gives an example of Network Service's YAML:

```yaml
apiVersion: networkservicemesh.io/v1alpha1
kind: NetworkService
metadata:
  name: example
spec:
  payload: IP
  matches:
    - match:
      sourceSelector:
        app: firewall
      route:
        - destination:
          destinationSelector:
            app: vpn-gateway
    - match:
      route:
        - destination:
          destinationSelector:
            app: firewall
```

Figure 4.3: YAML of Network Service

Following information are provided in this file:

- "kind" label: as mentioned before, Network Service is defined in K8s as CRD, so the "kind" label indicates that this file represents the implementation of a Network Service.

- "name" label: the value of entry simply indicates the name of the Network Service, which will be indicated in each request or reply to refer to that particular Network Service.
- "payload" label: value of this entry indicates the kind of traffic that is supported by cross-connections involved with this Network Service. This means that, once cross-connections between pods that implement VNFs of the Network Service are created, the traffic that traverses these links will be of the kind specified in this label.
- "matches" label: this part of the YAML indicates which VNFs are associated to the Network Service and mostly in which order VNFs are chained. Each "math" label consists of a "sourceSelector" and a "destinationSelector" label: the value of the key-value pair of this labels represent respectively the source and the destination of a pair of VNFs that composes the Network Service. Taking as example the first match in figure 4.3, the source of the pair is the pod implementing the firewall, while the destination is the pod implementing the VPN gateway: in the order of the VNFs, firewall pod is always before VPN-gateway, meaning that, in the chaining of VNFs, a cross-connection is built between these two pods and traffic will traverse it to go from firewall to VPN-gateway in the specified order. A match may not have the sourceSelector label: this means that each request of cross-connection which does not contain any value for the "app" key is directed to the pod implementing the specified VNF in the destinationSelector of this match. This is useful to point out which is the first VNF that receives the traffic in the Network Service.

This YAML file gives all the necessary information to understand how the Network Service is implemented. Each NSM Manager registers this information so that, each time a NSM Manager receives a cross-connection request, depending on labels contained in the request it knows the destination of the request.

Network Service Endpoint's YAML

To provide a correct deployment of the service chain, it is also necessary to define Network Service Endpoints' YAML files. Figure 4.4 gives an example of one possible NSE's YAML:

```yaml
apiVersion: apps/v1
kind: Pod
spec:
  containers:
    - name: sidecar-nse
      image: raffaeletrani/sidecar-nse:master
      imagePullPolicy: IfNotPresent
      env:
        - name: ENDPOINT_NETWORK_SERVICE
          value: "example"
        - name: ENDPOINT_LABELS
          value: "app=firewall"
        - name: IP_ADDRESS
          value: "172.16.1.0/24"
        - name: CLIENT_NETWORK_SERVICE
          value: "example"
        - name: CLIENT_LABELS
          value: "app=firewall"
      resources:
        limits:
          networkservicemesh.io/socket: 1
    - name: firewall-container
      image: raffaeletrani/firewall_container:k8s
      command: ['/bin/sh', '-c', 'sleep infinity']
metadata:
  name: firewall
  namespace: default
```

Figure 4.4: YAML of Network Service Endpoint

The YAML file must be provided with correct information to make the NSE work correctly with NSM. For this purpose, necessary functions to allow the pod behave properly (i.e. receive, elaborate and reply to cross-connections requests) in NSM environment are to be implemented in a container that runs inside the pod. In figure 4.4, the container that satisfies this necessity is the one called "sidecar-nse". First task of this element is to register with NSM Manager of the node in which the pod is scheduled: in this way it advertises its presence to NSM Manager, also giving information about itself. This information, together with other ones fundamental to implement the correct behavior in NSM, must be provided to the container as "env" key-value pairs, as figure 4.4 shows:

- ENDPOINT_NETWORK_SERVICE: this value indicates the name of the Network Service this NSE belongs to.
- ENDPOINT_LABELS: this value indicates the VNF that the NSE implements in the Network Service. This and the previous value must be specified in the registration to the NSM Manager.
- IP_ADDRESS: this value is necessary when the NSE is involved in a cross-connection where IP addresses are required. It in fact represents the IP addressing from which IP addresses are taken when assigning them to network interfaces associated to a cross-connection. Further details will be provided later.
- CLIENT_NETWORK_SERVICE: if the NSE is not the last of the chain, it will require a cross-connection to the next hop of the chain. This value, as well as the next one, represents the Network Service that is associated with the cross-connection request.
- CLIENT_LABELS: this value is inserted in the cross-connection request and it indicates the NSE that is making the request. In reference to figure 4.3, it corresponds to the sourceSelector of a match.

Once the registration is completed, this container's aim is to receive requests from NSM Manager, elaborate them (i.e. send a request to the next hop of the chain if it is not the last one) and reply to it.

As we are talking about a Network Service Endpoint, also VNF functionalities must be provided in the pod. Depending on the implemention choice, the VNF can run as a different container in the pod, or can be implemented within the NSM container, integrating its functions with the ones provided by this container. In figure 4.4, VNF is implemented in a specific container, different from the NSM's one.

Network Service Client YAML

To make a generic pod behave like a Network Service Client, meaning it can interact with NSM and request a cross-connection to a Network Service, specific configuration of its YAML must be provided. As example of a YAML file is can be seen in left part of figure 4.5. The specific part of the YAML code that must be provided is the "annotations" value: this key-value pair is fundamental to provide the correct behavior. As explained in Admission Webhook section, this element intercepts the YAML files of a deployment and, depending on the presence of the "annotations" value, it will inject the code related to the initContainer provided by NSM, to allow the pod require a cross-connection. As result, the final YAML file

that will be elaborated by master node's elements is the one in the right part in figure 4.5. It is worth noticing that the key-value pair of the "annotations" value is used and replicated by the Admission Webhook as variable that is passed to the initContainer, so that the proper Network Service is required.

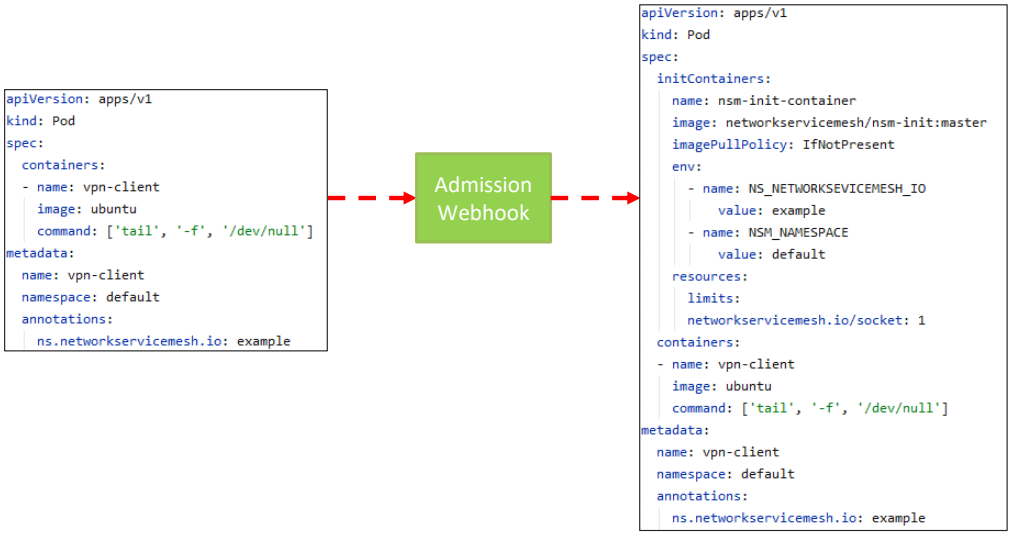

Figure 4.5: YAML of Network Service Client

4.3.2 Pods' deployment

After the definition of pod's YAML files involved, the service chain can be deployed in Kubernetes environment. To provide an automatic installation, Network Service Mesh uses Helm's[2] charts. As first part of Helm deployment, pods will be scheduled in cluster's nodes and after that they start performing their functions. Figure 4.6 shows what happens once pods are deployed in the cluster:

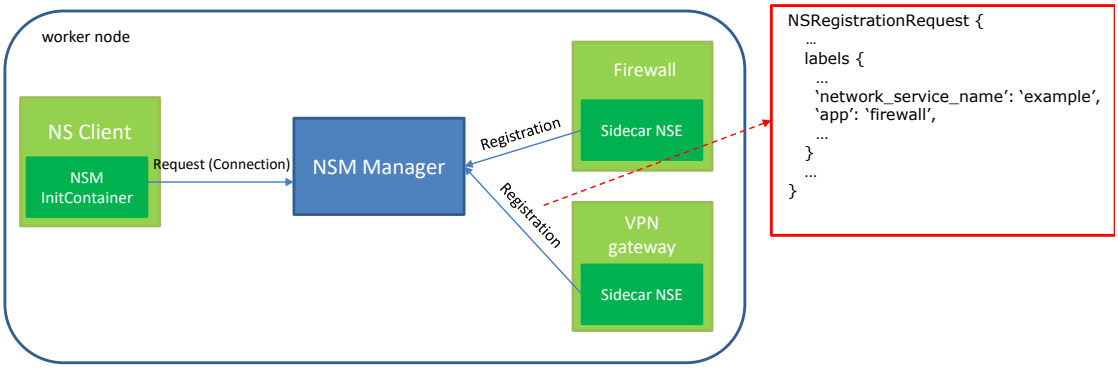

Figure 4.6: Pods' first functions executed during Helm deployment

First step that NSEs's sidecar containers perform is register with NSM Manager of the node in which it is deployed. This registration consists of a gRPC message called "NSRegistrationRequest" sent to the NSM Manager, which contains different information about the pod's role in NSM. The most important part of this request is the one indicated in figure 4.6 as "labels": the first key-value pair pointed out in the figure indicates the name of the Network Service, and the second part indicates the specific VNF that this pod implements in the Network Service. It is important to note that this two key-value pairs are the ones that are passed to the sidecar as variables in the definition of the YAML file of the pod in which the container runs. If the registration is successful, NSM Manager will know that in its node there is a pod implementing a VNF function of that specific Network Service and it will send requests directed to that pod if necessary.

On the other side, in the pod implementing Network Service Client initContainer starts working and, as his main aim is to build it sends a request of cross-connection to the NSM Manager. As this is part of the actual cross-connection construction, more details are provided in the following section.

[2]It is a Kubernetes package manager which allows to define a group of Kubernetes resources and automatically deploy them in the K8s' cluster. Github reference https://github.com/helm/helm

4.3.3 Building the cross-connection

Once pods involved in service chain implementation are deployed and VNFs' NSM-related sidecar containers end their registration with the NSM Manager, cross-connections construction begins. It is worth noticing that this event takes place within the Helm's deployment, i.e. the Helm command used to install pods involved in the deployment will not terminate until all cross-connections between pods are built. Main steps of building the cross-connection are the following:

1. The very first action of this process is performed by NSM initContainer in NS Client pod: it sends a gRPC message to the NSM Manager containing the cross-connection request to a Network Service. Figure 4.7 gives an example of a the request:

Figure 4.7

2. When NSM Manager receives the cross-connection request, it analyzes its parameters, in particular it tries to find the Network Service specified in the request and, if it exists, it analyzes label "app" to understand which VNF of the Network Service is required. In figure 4.7 this label is empty meaning that, referring to figure 4.3, the request needs to be sent to a pod which implements a firewall. NSM Manager so tries to understand if the requested VNF is implemented by a pod (i.e. an NSE) that is present in its same node: to do so it checks its local registry to find out if there is a NSRegistration that corresponds to a NSE that implements the required VNF and, if so, it will forward the request to it. Otherwise, it forwards the request to NSM Managers of other nodes, requiring where the NSE is scheduled. The NSM Manager that receives the request behaves in the same way, until the request sent by the first NSM is satisfied.

3. Once the correct pod receives the request, it analyzes the request, to understand if the parameters associated to the request (i.e. specific required technology of cross-connection) can be satisfied. In positive case, it creates a reply to the request and it sends it back to the NSM Manager. figure 4.8 provides an example of the reply:

```
NSRequest {
  …
  labels {
    …
    'ns_name':'example',
    'app': '',
    …
  }
  …
}
```

Figure 4.8

As it can be seen, in this case the reply contains also parameters associated to the IP addresses that need to be assigned to the interfaces involved in the cross-connection.

4. NSM Manager receives the reply from NSE and elaborates it. In particular, it extracts specific parameters, with the aim to create a request to be forwarded to the NSM forwarder. This part of the process is called "programming of the NSM forwarder": the forwarder receives specific instructions and parameters about the cross-connection that needs to be built. The request specify the technology that is used by pods that must by cross-connected: depending on this parameter, proper interfaces are configured in involved pods.

5. The NSM forwarder injects required interfaces in pods that need to communicate. Depending on the technology in may also create interfaces on itself and properly configure them to receive traffic from interfaces injected in pods and forward it between them. Next chapter will give more details about this aspect.

6. Once the interfaces are injected and properly configured, forwarder creates the cross-connection between involved pods. It registers it locally, so that it can monitor (with the help of NSM Manager) all created cross-connections.

7. Once the cross-connection is created, it advertises the NSM Manager that the process is done: the Manager may interrogate the forwarder to check if all cross-connections are still working properly, or it can also receive information from the forwarder about cross-connections' state.

8. As last step of this process, once the NSM Manager receives confirmation about creation of the proper cross-connection, it advertises the NSM initContainer running in NS Client pod that the cross-connection is created and it can be used to communicate with NSEs involved in the required Network Service. As the initContainer accomplished its job, it stops working and the main container of NS Client starts running.

Cross-connection on same node - one hop

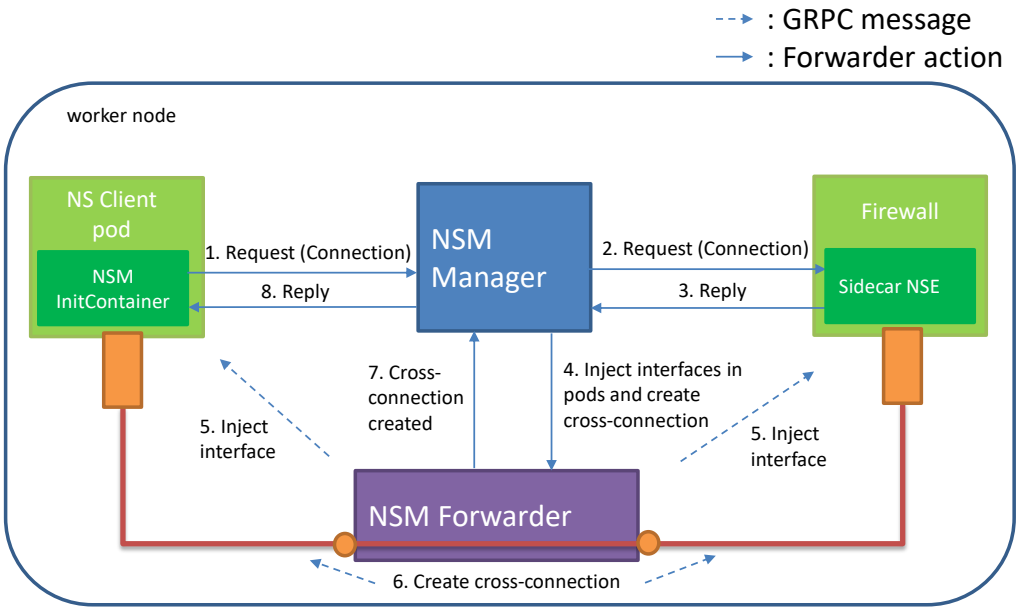

Figure 4.9: Cross-connection to one Network Service Endpoint

Figure 4.9 gives a graphic representation of the steps explained above in the creation of the cross-connection. As the example is provided in a single K8s' worker node, the NSM Manager does not have to communicate with NSM Manager on other nodes to understand where the NSE implementing the firewall is scheduled.

Cross-connection on same node - two hops

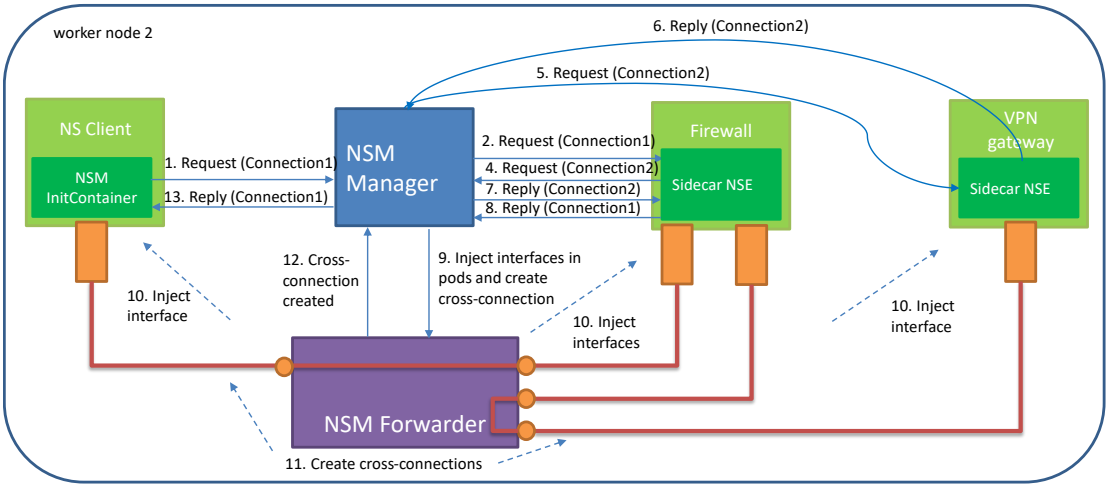

Figure 4.10: Cross-connection to service chain of two NSEs

Figure 4.10 gives a graphic representation of the process of building the cross-connection between more than two pods: it this case there is one NS Client requiring a cross-connection with a Network Service which consists of two VNFs. For this reason, the process is slightly different from the previous case: once the NSM sidecar running inside firewall pod receives the NSRequest from NSM Manager, it will not immediately reply to the NSM Manager, but it sends to it another request of cross-connection: this request is labelled so that the NSM Manager will understand that it is directed to next hop of the service chain implemented in the Network Service. In reference to figure 4.7, the request contains a label referring to the same Network Service required in the previous request (i.e. the Network Service firewall pod and VPN gateway pod belongs to), but as value of label "app" it has the name of the NSE, in this case the firewall, which is making the request. In this way, referring to figure 4.3, the NSM Manager understands the the sourceSelector of the request is the firewall and, basing on the matches of the Network Service's YAML file, the destinationSelector is the VPN gateway. As the previous case, because all pods are deployed in same node, the request is not sent to any NSM Manager on other nodes, but it is forwarded directly to the pod that implements the VPN gateway. The NSM sidecar running inside VPN gateway pod behaves as explained before, replying to NSM Manager in case parameters of NSRequest are supported. NSM Manager then replies to firewall pod's request, which then replies to NS Client NSRequest and from this point on, the process goes on in the same way as the previous case.

Cross-connection on different nodes

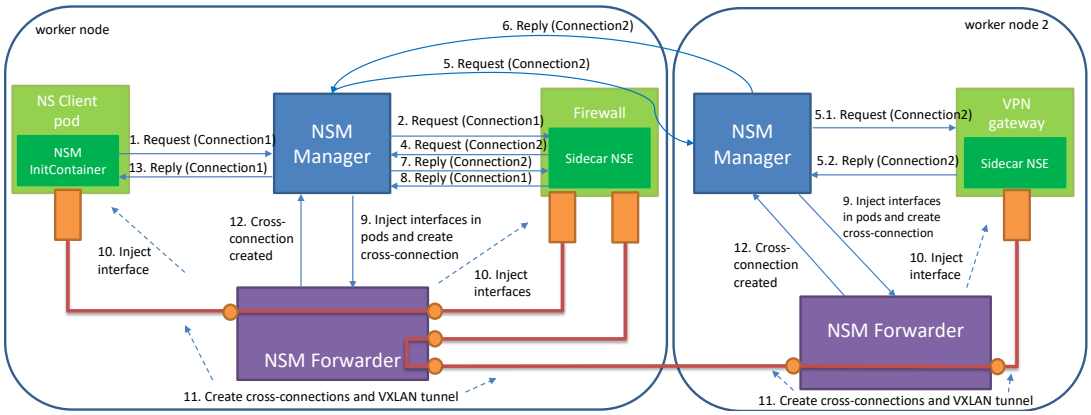

Figure 4.11: Cross-connection to service chain on different nodes

Figure 4.11 gives a graphic representation of the process of building the cross-connection between more than two pods that are deployed on two different nodes. The main difference with the previous case is the forwarding of the NSRequest created by firewall pod: once NSM Manager on same node of firewall pod receives the request, it understands that the VPN gateway pod is not scheduled in its same node, so it sends the request to other NSM Managers. NSM Managers receives the request, analyze it and, if required pod is scheduled in same node, it will forward the request to it. NSM sidecar running inside VPN gateway pod processes the request in the usual way and replies to the NSM Manager, which will forward the reply to NSM Manager of the initial node. NSM Managers then properly program the forwarder to allow communication between pods on different nodes: forwarders of the involved nodes understand that a inter-node communication is required, so they configure a VXLAN tunnel between them so that traffic can be forwarded from one node to the other. In this case, as the process requires more elaboration, it may takes more time to build all the required cross-connections, but the result is always the same: pods involved in the communication can send traffic using the interfaces injected by forwarders and traffic will traverse the created cross-connections.

4.4 Development of NSM applications

The Network Service Mesh project shows up as a "plug-and-play" system, i.e. if a VNFs provider with already developed and working virtual network functions wants to use NSM to integrate its VNFs in Kubernetes, it simply needs to deploy NSM main components in Kubernetes and its VNFs, properly configure them to work together and the system will automatically works in the desired way. This means that NSM would adapt to any kind of VNF service chain that needs to be deployed in Kubernetes.

To understand if this behavior is actually supported, different combinations of NSM with VNFs were deployed and tested. Results are summarized in next subsections.

4.4.1 NSM with already developed VNFs

As previously said, NSM claims that VNFs service chains can be deployed together with NSM in Kubernetes and the system should work in the desired way. This feature is true, but it actually has some requirements that need to be taken in account for this kind of deployment.

Let us pretend that, in a simplified scenario, an already developed VNF has been properly containerized and it can be deployed as a pod in Kubernetes. Before deploying the VNF, NSM main components are deployed as pods in K8s cluster. As explained in previous sections, the VNF is represented in NSM as part of a Network Service Endpoint: for this reason the container implementing the VNF is deployed as part of the pod which represents the NSE and it cooperates with the container implementing the NSE. This cooperation can be although problematic. The first problem comes with the configuration of interfaces by NSM control plane elements in the NSE in order to allow the NSE be part of the service chain. As the VNF container implements the VNF functionalities, it expects to have two already configured network interfaces that can be used to receive and send traffic. If these interfaces need to have particular requirements (e.g. specific technology, IP address, working method etc.), this last may not be satisfied by the configured interfaces that are provided by NSM. It is true that specific variables can be passed to the NSM container of VNF pod so that the configured interfaces satisfies some requirements, but the behavior of the VNF container may be modified to correctly adapt to the interfaces provided by NSM. In this respect, better behavior can be obtained by VNFs that do not start working with specific configured interfaces as they start working, but accept configured interfaces and start providing proper behavior once interfaces are configured.

Another aspect that must be taken in account when working with NSM is the fact that NSM implements by itself some functionalities that, in some cases, may

be already implemented (or need to be implemented) by the VNF container. For example, in most cases the interfaces' IP assignment is provided by the NSM container of VNF pod and this behavior is strictly required by NSM to work properly: this can be problematic if the VNF requires to be the one in charge of assigning IP addresses, or some of its functionalities are in some way involved with dynamic IP assignment.

As described in next sections, NSM supports VPP technology 2.3 and provides a substantial integration of this technology in its components. In particular, the NSM forwarder in its default implementation presents VPP technology. NSM also uses VPP inside NSE pods, integrating its functionalities in the NSM container of the pod. In this case, the integration of "VPP NSE" with a VNF container within the same pod can be problematic. The first problem is related to the fact that VPP works and configure MEMIF interfaces in VNF pod. Nowadays VNFs generally work with network interfaces and, as MEMIF are basically portion of shared memory, the VNF would not be able to see and interact with this interface. Moreover, when VPP technology is used, the VPP Agent is in charge of interacting with packets received by MEMIF interfaces: this means that VNF container is not able to elaborate received packets. In this respect, the VNF container can not work properly and the only way to integrate VNF inside the pod is to rewrite the VNF functionalities appropriately to work inside the VPP implementation.

It is important to note that, as NSM is an open source project, it allows to create specific NSM containers that will run with VNF containers, i.e. it is possible to define specific Network Service Endpoints that can correctly implement the required behavior for the specific features of VNFs. In the tested scenarios of next chapter, specific NSM container implementations were created, to satisfy the requirements of the deployed VNFs containers. It is in fact necessary to use the appropriate NSE depending on the "position" of the VNF inside the chain: if the VNF is not the last hop, the NSE container needs to require a cross-connection to the next hop, so this functionality must be provided, differently from the case where the VNF is the last hop, where this functionality is not needed. This feature is also very important in those situations where the features of the VNFs cannot be modified, so a specific implementation of the NSE is required.

4.4.2 Ad-hoc VNFs with NSM

In the scenario depicted below, NSM can integrate already developed VNFs, but with some constraints. A possible alternative to avoid apposite adjustments in VNF implementation or in NSE implementation could be building ad-hoc VNFs with NSM. This of course would mean to implement VNF from scratch, but can be a possible solution when requirements of integration between developed VNF and NSM cannot be satisfied. Note that this solution can be seriously taken in account

when containerization of VNF means creating a Cloud-native Network Function, i.e. the re-implementation of the VNF as container.

As NSM supports both standard and VPP technology, ad-hoc VNFs can be differentiated in two cases, the first implementing a VNF with standard technology, the second with VPP technology.

In the first case, the approach may be divided into two further cases. In the first approach the VNF can be implemented as a separate container, differentiating it from the container implementing NSM functionalities in the NSE pod. In this case, the VNF container must be aware that interfaces are configured in the pod after NSM container completes the communication with other NSM components. Moreover, the two containers must cooperate such that, in case NSM container takes over events of rebuilding or updating of cross-connections, the VNF container must be advertised about these events, so that it can update or modify its behavior on the basis of the new (or updated) cross-connections. On the other hands, in the second approach the VNF functionalities are integrated within the container implementing NSM functionalities in the NSE pod. As explained before, NSM functionalities are divided in so called Endpoints, each one implementing one specific functionality. In this case, VNF functionalities would be implemented as Endpoint along with NSM functionalities. In this way, the part of the container implementing the virtual network function does not need further configuration, as it receives direct instructions from NSM part of the container. Moreover, in case of update or rebuilding of cross-connections the pod is involved in, as VNF functionalities are integrated with NSM functionalities, the former are automatically aware of the update and do not need particular implementations or manual reconfiguration.

In the second case, the possible division explained in the previous case is not possible. In fact, as explained in previous subsection, when using VPP technology the interfaces configured by NSM in the VNF pod are MEMIF interfaces. For this reason, if the VNF was implemented as a separate container, it would not be able to elaborate packets received by MEMIF interfaces, only the VPP Agent is able to manage interfaces and packets. In this respect, the only way to provide VNF functionalities with VPP technology is to integrate them with VPP functionalities, i.e. inside the VPP Agent. Also VPP Agent functionalities, as NSM functionalities inside the NSE, are implemented as specific functions that are chained to work together and can be added and removed to give different behavior. In this respect, VNF functionalities become part of those functions and are integrated with the ones that implement the VPP Agent: packets received by MEMIF interfaces can be configured with specifics provided by VNF functionalities and, more in general, they can elaborate packets received by these interfaces.

Chapter 5

Benchmarking NSM

After a deep analysis of the Network Service Mesh project, various tests of performance were carried out, to understand also the potentiality of the proposed solution with respect to other technologies. For this purpose, different scenarios were deployed. Figure 5.1 presents a general graphical representation of the deployed scenarios:

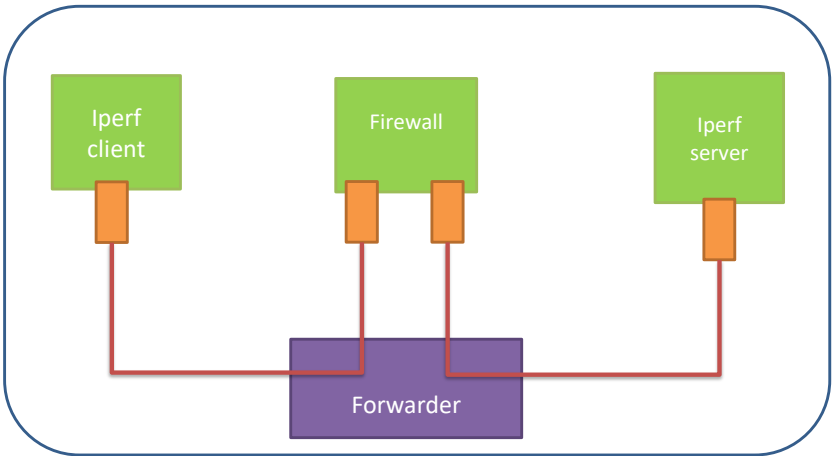

Figure 5.1: Representation of implemented service chain

Some considerations on figure 5.1:

- The application used to run the tests is iperf3[1]. For this reason, the head and the tail of the chain are respectively iperf client and iperf server.

[1]https://iperf3.fr/

- Depending on the scenario, Kubernetes may or may not be the environment of the deployment (an explanation of the choice will be given per each scenario).
- As described in previous chapter, NSM supports both communication between pods that are both on the same Kubernetes node and on different nodes, but for different reasons tests reported below refers to scenarios that involve only one node. In this respect, also scenarios that do not involve Kubernetes were tested on the same physical server.
- The chain is such simple in order to test a very basic implementation. In a more realistic scenario, the connection can come from outside of the deployed chain (for example from a user) and it can be directed outside the chain (for example, on the public network).

5.1 Tested scenarios

5.1.1 First scenario: baseline

In order to have a basis of comparison in terms of performance, first scenario does not involve Kubernetes environment, and can be considered a baseline for the following ones. It consists in the communication between different Linux network namespaces.

Case 1: two namespaces

The first case consists of communication of two different Linux network namespaces.

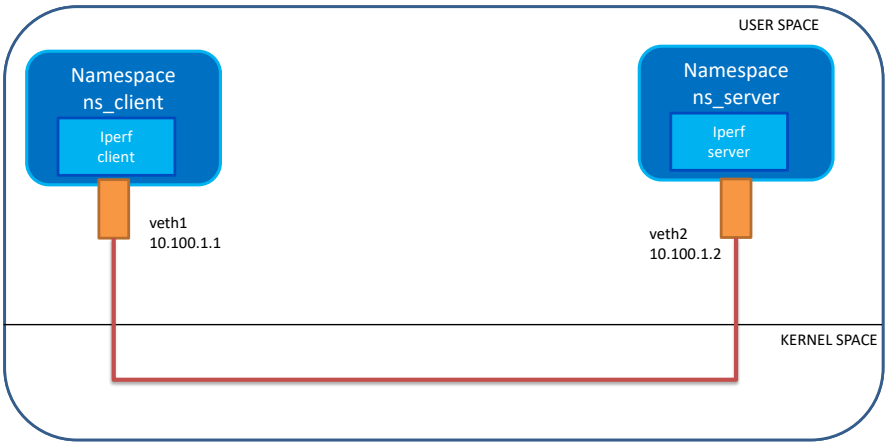

Figure 5.2: First scenario, first case

First namespace runs iperf client and second namespace runs iperf server. To allow communication between namespaces, one veth pair is configured, in particular one veth interface is configured on iperf client namespace and the other in iperf server namespace. With this configuration, traffic generated by iperf instances goes directly from one namespace to the other.

Case 2: "vanilla" service chain

To see how performance changes in a more chain-oriented context, the second case consists of communication of three different Linux network namespaces. This case is divided into three sub-cases. Figure 5.3 shows a graphic representation of first sub-case.

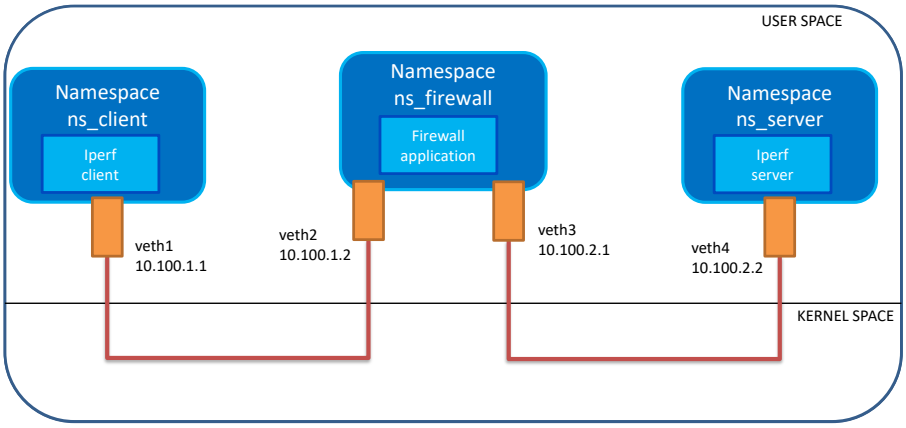

Figure 5.3: First scenario, second case, iptables rules as firewall

The first namespace runs the instance of iperf client, the middle namespace runs iptables commands as implementation of a firewall application, the last namespace runs the instance of iperf server. To allow communication between namespaces, veth pairs are configured in different namespaces. In particular, IPs are assigned to virtual ethernet interfaces such that each pair is isolated from the other one. In this respect, the forwarding of the traffic is provided by manually configuring the correct entries in the routing tables of each namespace: in the iperf client's namespace the routing entry to reach the IP addressing given to veth pair between firewall and iperf server is added, and vice versa in the iperf server's namespace. Always in terms of routing, as the firewall is involved in both veth pairs with iperf client and iperf server, no manual configuration of routing table's entries was needed for it: once the veth pairs are configured, appropriate entries in firewall namespace's routing table are configured. After the overall configuration, traffic flows from iperf client namespace to iperf server namespace, passing through firewall namespace.

Also another implementation of this case was tested. In this second sub-case, the firewall is implemented as a transparent firewall, which consists of a Linux bridge with iptables configured in it. Figure 5.4 graphically represents this sub-case:

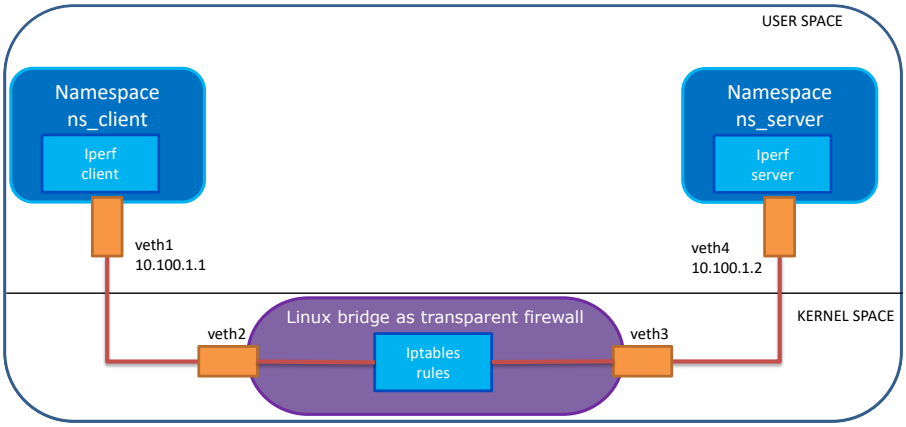

Figure 5.4: First scenario, second case, Linux bridge with iptables rules as transparent firewall

Differently from previous sub-case, as it is necessary to implement a transparent firewall, the veth interfaces configured in the Linux bridge do not have an IP address. In this respect, only one IP addressing is used to assign IP addresses to iperf client and iperf server namespace. As the bridge implements a transparent firewall, the two namespaces see each other as directly attached, so no specific routing table entries were to be configured in namespaces. Firewall implementation is represented in this case by iptables inside Linux bridge: packets received by interfaces are checked with iptables rules and, if checks are positive, they are forwarded to the other interface.

Last implementation of this case, which represents the third sub-case, presents the firewall application implemented as user application in firewall namespace. The user application not only provides the firewall functionalities, but it also forwards traffic between the two interfaces of firewall's namespace, implementing what can be call a transparent firewall. To allow this particular implementation, veth interfaces on firewall namespace do not have an IP address, there is only one IP addressing between client and server namespace. Figure shows graphic representation of this sub-case:

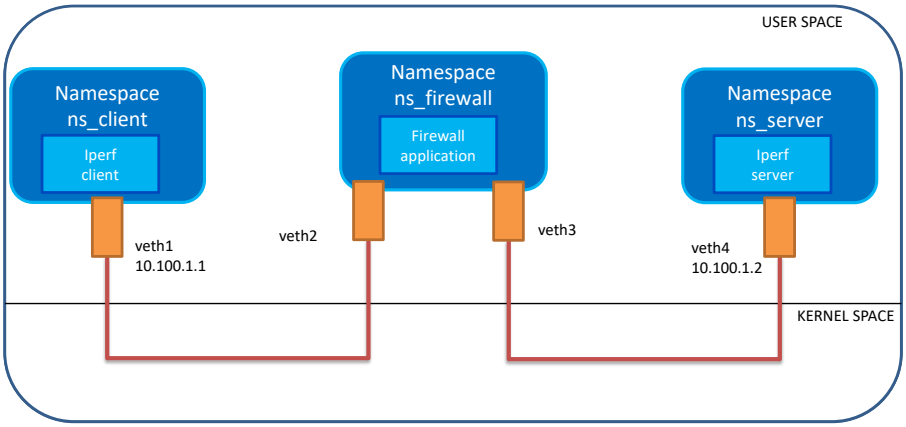

Figure 5.5: First scenario, second case, user application as firewall

Differently from previous sub-case, as client and server namespaces' interfaces have IP addresses belonging to the same IP addressing, routing table entries are automatically configured in respective routing tables once the veth interfaces are configured in the namespaces. On the other side, as firewall application is in charge of forwarding traffic, firewall namespace's routing table does not have any entry.

5.1.2 Second scenario: NSM Forwarder without VPP in chain

The second scenario is deployed in a Kubernetes worker node and involves the use of Network Service Mesh solution. It consists of a basic chain of pods:

- First pod runs two containers. The first container is the initContainer injected by NSM Admission Webhook: it requires a connection to the Network Service that (in this case) represents the chain between firewall pod and iperf server pod. The second container implements the iperf client, which will send traffic to iperf server to test the performance of the chain.
- Second pod runs two containers as well. The first one is the sidecar used to allow the pod work properly in NSM, in particular its function is to register with the NS Manager present on the same node, to advertise which function of the Network Service it belongs to it implements. This container will receive control plane request of cross-connection creation and will reply to them. The second container runs the application that implements firewall functions.
- Third pod runs two containers as well. The first is the same sidecar described in firewall pod; the second container implements the iperf server: it will receive traffic from iperf client and it will send traffic back to it.

This scenario is divided into two cases. The reason of this splitting is related to the fact that NSM provides two different implementation of the forwarder, so same implementation of pods with each type of forwarder was tested, to understand the difference between them.

Case 1: NSM Kernel Forwarder

The first case is implemented with NSM kernel forwarder: this element configures veth pair between pods involved in communications. As explained in the previous chapter, to provide this behavior, it uses `NetLink` Linux library, which provides necessary functions to properly configure and inject veth interfaces in involved pods. As shown in figure 5.6, in this case the forwarder is not directly involved in the communication, it just provides the cross-connection between pods involved in the communications, it does not configures further interfaces (as it happens in next case) to allow the forwarding.

As in second case of first scenario, also this case is divided in two sub-cases. The first one is represented in figure 5.6.

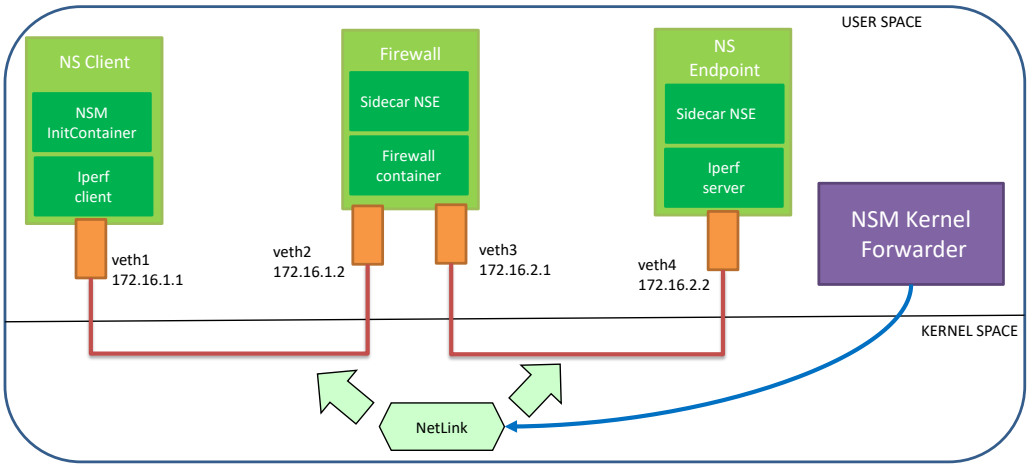

Figure 5.6: Second scenario, first case, iptables rules as firewall

Also in this case IP addressing assigned to each veth pair is different and veth pairs are isolated from each other. For this reason, it was necessary to manually configure routing tables entries in iperf client and iperf server, to allow each of them to reach the other one. On the other hand, always in the same way as previous scenario, no manual configuration of routing table inside firewall pod was needed, i.e. once the veth pairs were configured, automatic routing tables inside firewall pod were injected, so that it could reach both veth pairs. In this sub-case, the firewall application consists of iptables rules injected in the kernel.

The second sub-case, as previous scenario, implements a transparent firewall.

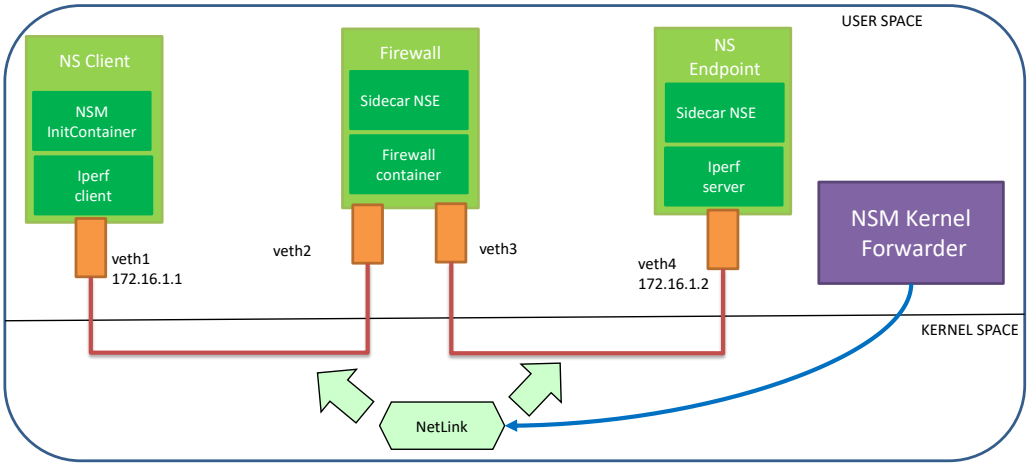

Figure 5.7: Second scenario, first case, user application as firewall

As figure 5.7 shows, firewall pod's veth interfaces do not have an IP address, so that iperf client and iperf server pod can see each other as "directly" connected. In this respect, firewall pod also provides forwarding functionalities, i.e. it forwards traffic between interfaces. The implementation of firewall and forwarding functionalities are provided by a user application built expressly to allow the firewall behave as a transparent firewall: it listens to packets received on both interfaces, once a packet is received from an interface it checks its parameters and, if checks are positive, it properly forwards it to the other interface. Note that, as NSM provides a configuration of this case as in first figure, appropriate commands were needed to create this configuration.

Case 2: NSM VPP Forwarder

The second case is implemented with NSM VPP forwarder. In this case, this element is involved in the forwarding of the traffic. In fact, this element configures TAP interfaces[2] pairs between each pod and itself, and configures the self-injected TAP interfaces so that traffic flows correctly from one interface to another to allow proper communication between pods.

[2]Section 2.3.3

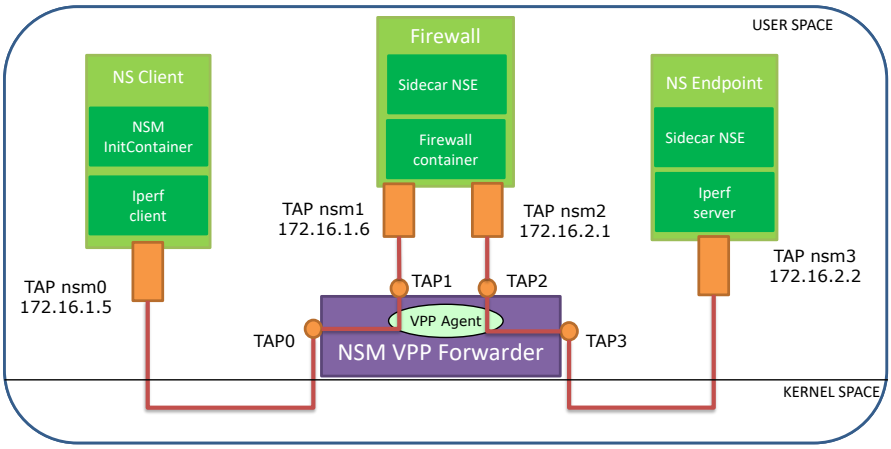

Figure 5.8: Second scenario, second case

Pods will not be aware of the presence of TAP interfaces configured in the forwarder, they will consider the link to be direct to the TAP interface configured in the other pod. As in the previous case, IP addressing assigned to TAP interfaces pair is different and TAP interfaces pairs are isolated from each other. Also in this case it was therefore necessary to manually configure routing table entries in each involved pod.

It is important to point out a note about the figure above. The TAP pair between iperf client pod and NSM VPP forwarder, as well as the one between iperf server and the forwarder, is partially represented in Kernel space. This because, while the TAP interface configured in the forwarder is in user space and is not present in the Kernel, the TAP interface configured in the NS Client exists in the network namespace of the pod and so it belongs to Kernel network namespace of NS Client pod. For this reason, the connection actually passes in kernel space and returns in user space. For sake of clarity in the figure, TAP pairs between firewall pod and forwarder are "direct" (i.e. the link does not pass in kernel space), but the implementation of the link happens in the very same way as with other pods.

As previous case, also this case is divided in two sub-cases. The first sub-case is already represented in figure 5.8: in this sub-case firewall application is implemented as iptables rules, so there was no need to create a particular configuration (i.e. the one provided by NSM is maintained as it is). The second sub-case is represented in figure 5.9:

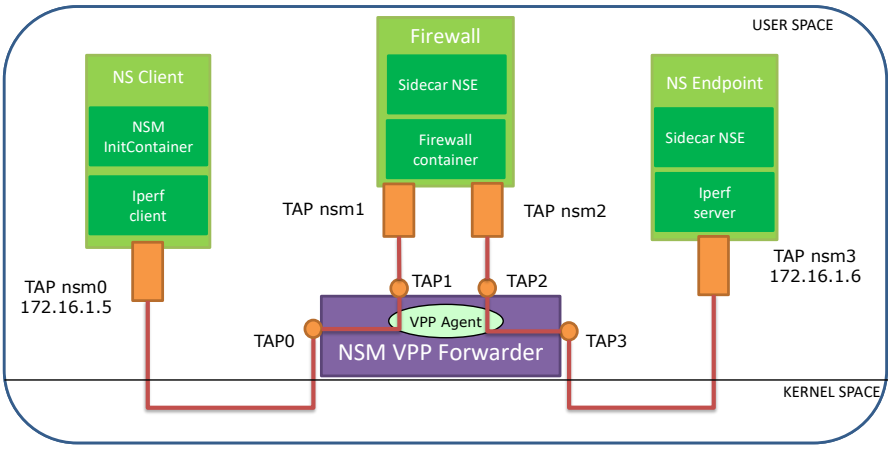

Figure 5.9: Second scenario, second case

In this sub-case, the firewall is implemented as a transparent firewall, i.e. it provides both firewall and traffic forwarding functionalities. As in first case of this scenario, in this sub-case the firewall is implemented as user application running in firewall pod. To provide this behavior, specific commands were applied to configure the system to work in this way.

5.1.3 Third scenario: NSM Forwarder with VPP in chain

The third scenario, as the previous one, is deployed in a Kubernetes worker node and involves the use of Network Service Mesh solution. It consists of the same chain of pods as the previous one, but with some modifications. Also in this case, the scenario is divided into two different cases. The specific differences are explained in the description of the two cases.

Case 1: VPP firewall

In the first case, pods involved in the chain are implemented as follows:

- iperf client and iperf server maintain the same characteristics as the previous scenario, so they have both two containers, one implementing NSM control plane functions and the other implementing iperf respective application.
- firewall pod, differently from the previous scenario, runs only one container. The reason of this choice is that in this case VPP technology is used: as VPP works with MEMIF interfaces and implements a VPP agent to manage and configure this interfaces, code implementing firewall application must be embedded in code implementing VPP agent. Moreover, to make the VPP agent

correctly collaborate with NSM and receive the proper configuration of MEMIF interfaces, code implementing VPP agent is embedded in code implementing the sidecar which provides proper NSM control plane functionalities to pod. In this way, NSM and VPP agent functionalities are bound together, automating the processing of NSM control plane requests and processing of traffic by firewall functionalities.

Figure 5.10 gives graphical representation of this case:

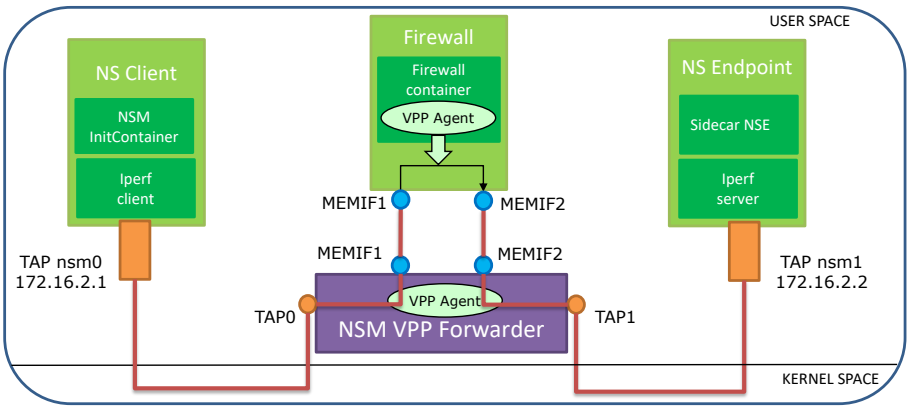

Figure 5.10: Third scenario, first case

In this case NSM VPP forwarder works as adaption layer: it configures TAP interfaces pairs with pods not including VPP technology, while it configures MEMIF interfaces with firewall pod, which includes VPP technology. Self-injected interfaces are configured so that traffic will be translated and forwarded from one kind of interface to another.

Other two notes about this case:

- TAP interfaces injected in iperf client and iperf server pods receive an IP address from the same IP addressing. This means that, differently from the previous scenarios, there is no need to manually configure routing table entries in each pod to allow reachability to the other pod. Each pod will see its TAP interface as linked directly to the other TAP interface configured in the other pod, it will not be aware of all the intermediate processing.

- Differently from the second case of the previous scenario, the graphic representation provided in figure 5.10 is the actual implementation of the scenario: as TAP interfaces configured in iperf client and iperf server pod are part of the network namespace of the pods, the part of TAP pairs is represented in kernel space, meaning that traffic passes also in kernel space. On the other hand, as MEMIF interfaces are present only in user space, cross-connections

between forwarder and firewall pod are strictly in user space and traffic does not cross kernel space, as represented in figure.

Case 2: VPP technology across the chain

In the second case, to test the potentiality of VPP technology, VPP is deployed in every element of the chain. Here follows a description of involved pods:

- Differently from the previous scenarios, iperf client pod runs one container. The reason is the following: as iperf application needs to see the MEMIF interface configured by VPP agent, it cannot run in the pod as a separate container, but it must be integrated in the VPP itself. VPP does not natively support iperf application, so VPP agent required specific manual configuration.
- firewall pod, as the previous case, runs one container, which embeds NSM control plane functionalities and VPP agent implementation, the latter of which includes firewall implementation.
- As for iperf client pod, also iperf server pod runs one container, for the same previous explained reason. Also in this case, as VPP does not natively support iperf application, VPP agent required specific manual configuration.

Figure 5.11 gives graphical representation of this case:

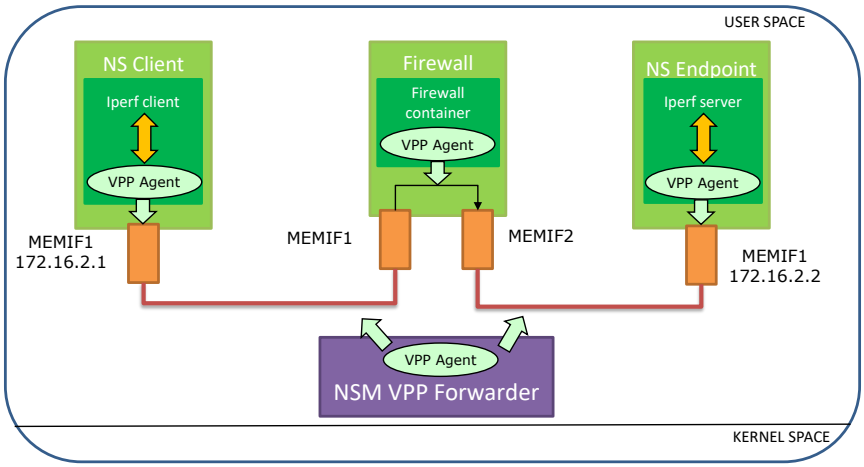

Figure 5.11: Third scenario, second case

Differently from the previous case, NSM VPP forwarder behaves in a different way: as configured interfaces on communicating pods are all MEMIF interfaces, the forwarder works like a MEMIF proxy, which means that it does not self-inject any kind interface, it simply allows the traffic to be proxied from one MEMIF interface

to another. Moreover, as the previous case, MEMIF interfaces configured on iperf client and server pods do not need manual configuration to be able to reach each other, they see the two MEMIF interfaces as part of the same link, without being aware of intermediate processing.

5.1.4 Forth scenario: eBPF

To compare NSM performance with eBPF technology, the last deployed scenario involves Polycube project. Polycube is an open source software framework for Linux that provides fast and lightweight network functions, called *cubes*, which can be composed to build arbitrary service chains and, in this scenario, thet provide custom network connectivity to namespaces. Network functions are based on recent BPF and XDP Linux kernel technologies.

As Polycube provides different possible configuration to compare previous scenarios with, also this scenario is divided into two cases.

Case 1: transparent firewall

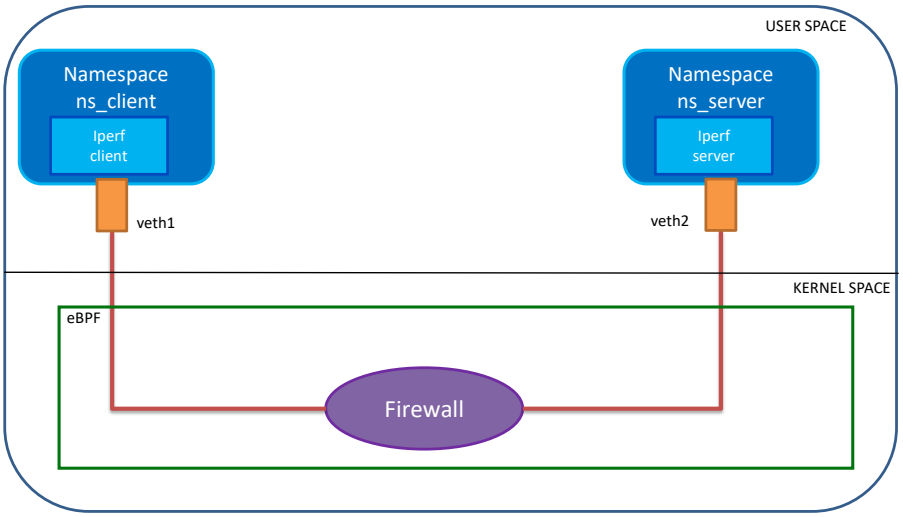

Figure 5.12: Fourth scenario, first case

In the first case of this scenario, two namespaces are involved in the communication: the first one implements an instance of iperf client, the second one implements an instance of iperf server. The only element involved in communication is Polycube firewall. It is implemented as a transparent virtual network function: it attaches to an interface involved in the communication (in this case, it is attached to veth1 interface of iperf client's namespace) and it analyzes incoming and outgoing traffic

of the specific interface. To provide a correct behavior, simple rules are applied to it, so that traffic can flows correctly from one namespace to the other.

Case 2: transparent firewall with "pbforwarder"

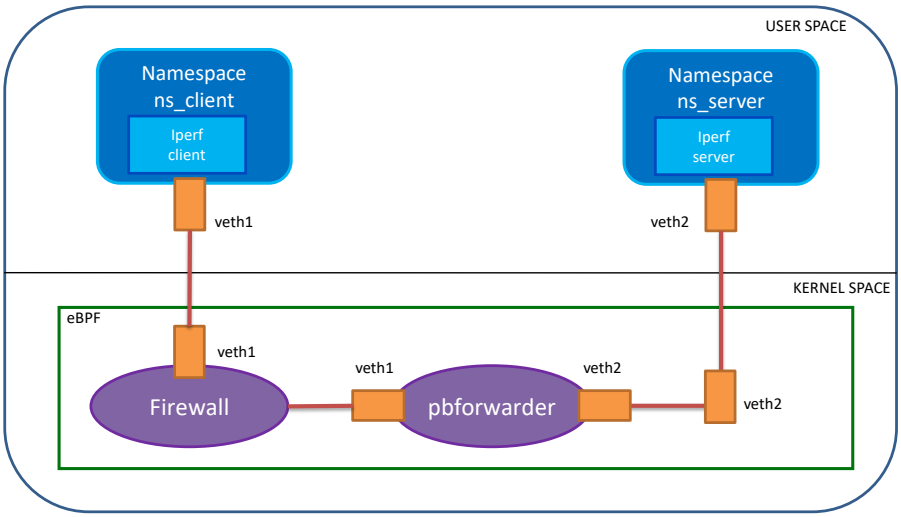

Figure 5.13: Fourth scenario, second case

In the second case of this scenario, a more complex chain of Polycube cubes is implemented. Veth pairs are created between namespaces involved in the communication and host's namespace (i.e. in the kernel of the host in which namespaces are deployed). Firewall transparent cube is now attached to the kernel side veth interface involved in the veth pair between iperf client's namespace and host's namespace. To allow forwarding between veth pairs, the *Polycube pbforwarder*[3] is involved in the chain: this cube is able to forward traffic between its interfaces depending on specific forwarding rules that are configured in the cube itself. It this case it creates veth1, which is the veth peer to veth interface configured in the host's namespace for veth pair with iperf client's namespace; the same behavior is applied to veth2, peering to kernel's veth interface for veth pair with iperf server namespace. With forwarding rules configured in pbforwarder, traffic is sent from veth1 to veth2 and vice versa.

[3]https://polycube-network.readthedocs.io/en/latest/services/pcn-pbforwarder/pbforwarder.html

5.2 Tests

The described scenarios were deployed and tested on a server available in network laboratory at Polytechnic of Turin, where this thesis was carried out. Here follows hardware and software information about environment of work:

- Environment tests' characteristics:
  - Ubuntu version: 18.04
  - Kubernetes version: 1.16.3
  - CNI used: Polycube CNI. However, the specific CNI in use does not affect the result of the Kubernetes tests, as the traffic remains confined in the NSM portion of the network.
  - Kubernetes nodes deployed: one master node and two worker nodes, each one in a different physical server
  - Worker nodes hardware characteristics:
    - Processor: Intel Xeon 3.50Hz
    - Memory: 32 GB RAM
    - Disk: 1TB HGST
- Application level characteristics:
  - Firewall applications used:
    - Iptables rules
    - Firewall user application[4]
  - Container image for iperf3: networkstatic/iperf3
  - Container image for firewall: raffaeletrani/firewall_container:k8s. This Docker image contains firewall user application and has kernel network privileges to implement iptables rules as firewall.
  - VPP configuration to work with iperf3: https://github.com/RaffaeleTrani/VPP-iperf3-config
  - NSM Git repository version: January 28th, 2020
  - Helm version: 2.15.0

Iperf3 tests were carried out with following characteristics:

- Transport protocol layer used: TCP

[4]https://github.com/RaffaeleTrani/firewall

- Interval of time of TCP transmission: 60 seconds
- Number of bytes per TCP block: 128 KBytes

Tests generates a single TCP session where packets, in Linux, are always forwarded on the same CPU. For this reason, the measured throughput refers to traffic managed by a single CPU; with a more diversified traffic, higher results could be obtained.

Note that the choice of TCP is not mandatory in these tests. In fact, it is also possible to execute these tests with UDP. Main reason why TCP was chosen is because the aim of this tests is not to obtain absolute performance of the system (i.e. considering also best behavior of environment hardware components), but to obtain an effective comparison between the different deployed scenarios.

5.2.1 First scenario: baseline

Case 1: two namespaces

Figure 5.19 shows obtained results of this case. For sake of space, only first eight seconds and last four seconds of the tests are explicitly reported. Also a summary of results and CPU utilization are provided:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	8.71 GBytes	74.8 Gbits/sec	0	396 KBytes
1.00-2.00 sec	8.73 GBytes	75.0 Gbits/sec	0	396 KBytes
2.00-3.00 sec	8.71 GBytes	74.9 Gbits/sec	0	396 KBytes
3.00-4.00 sec	8.71 GBytes	74.8 Gbits/sec	0	396 KBytes
4.00-5.00 sec	8.54 GBytes	73.4 Gbits/sec	0	396 KBytes
5.00-6.00 sec	8.58 GBytes	73.7 Gbits/sec	0	396 KBytes
6.00-7.00 sec	8.49 GBytes	73.0 Gbits/sec	0	467 KBytes
7.00-8.00 sec	8.68 GBytes	74.6 Gbits/sec	0	467 KBytes
...	...	...	...	...
56.00-57.00 sec	8.70 GBytes	74.8 Gbits/sec	0	2.33 MBytes
57.00-58.00 sec	8.58 GBytes	73.7 Gbits/sec	0	3.50 MBytes
58.00-59.00 sec	8.84 GBytes	76.0 Gbits/sec	0	3.50 MBytes
59.00-60.00 sec	8.69 GBytes	74.7 Gbits/sec	0	3.50 MBytes

Summary results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	527 GBytes	75.5 Gbits/sec	0

CPU Utilization: local/sender 96.1%, remote/receiver 93.7%

Figure 5.14: Results of iperf3 tests

CPU utilization indicates both iperf client (sender) and iperf server (receiver) utilization of the CPU in which they are scheduled as processes. The percentage refers to the the quantity of resources of the CPU that are dedicated to the iperf process and also to forward traffic generated by iperf.

Case 2: "vanilla" service chain

As explained in previous section, in this case are reported three further sub-cases: the difference between them is how the firewall is implemented. In the first sub-case the firewall consists of iptables rules configured in the firewall namespace.

Iptables as firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	7.71 GBytes	66.2 Gbits/sec	0	308 KBytes
1.00-2.00 sec	7.90 GBytes	67.9 Gbits/sec	0	322 KBytes
2.00-3.00 sec	7.63 GBytes	65.5 Gbits/sec	0	322 KBytes
3.00-4.00 sec	7.70 GBytes	66.1 Gbits/sec	0	342 KBytes
4.00-5.00 sec	7.56 GBytes	64.9 Gbits/sec	0	368 KBytes
5.00-6.00 sec	7.69 GBytes	66.1 Gbits/sec	0	407 KBytes
6.00-7.00 sec	7.62 GBytes	65.5 Gbits/sec	0	407 KBytes
7.00-8.00 sec	7.63 GBytes	65.6 Gbits/sec	0	532 KBytes
...	...	...	...	...
56.00-57.00 sec	7.78 GBytes	66.8 Gbits/sec	0	3.35 MBytes
57.00-58.00 sec	7.67 GBytes	65.9 Gbits/sec	0	3.35 MBytes
58.00-59.00 sec	7.64 GBytes	65.7 Gbits/sec	0	3.35 MBytes
59.00-60.00 sec	7.74 GBytes	66.5 Gbits/sec	0	3.35 MBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	465 GBytes	66.5 Gbits/sec	0

CPU Utilization: local/sender 100%, remote/receiver 96.3%

Figure 5.15: Results of iperf3 tests with iptables as firewall

In the second sub-case the firewall is implemented as kernel transparent firewall and consists of a Linux bridge with iptables rules configured in it provide firewall functionalities.

Linux bridge as transparent firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	6.95 GBytes	59.7 Gbits/sec	0	464 KBytes
1.00-2.00 sec	6.97 GBytes	59.9 Gbits/sec	0	464 KBytes
2.00-3.00 sec	6.96 GBytes	59.8 Gbits/sec	0	464 KBytes
3.00-4.00 sec	6.71 GBytes	57.6 Gbits/sec	0	464 KBytes
4.00-5.00 sec	6.60 GBytes	56.7 Gbits/sec	0	665 KBytes
5.00-6.00 sec	6.94 GBytes	59.6 Gbits/sec	0	699 KBytes
6.00-7.00 sec	6.90 GBytes	59.3 Gbits/sec	0	699 KBytes
7.00-8.00 sec	6.86 GBytes	58.9 Gbits/sec	0	699 KBytes
...	...	..	...	...
56.00-57.00 sec	6.91 GBytes	59.4 Gbits/sec	228	2.54 MBytes
57.00-58.00 sec	6.81 GBytes	58.5 Gbits/sec	0	2.54 MBytes
58.00-59.00 sec	6.94 GBytes	59.6 Gbits/sec	0	2.54 MBytes
59.00-60.00 sec	6.82 GBytes	58.5 Gbits/sec	0	2.54 MBytes

Summary results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	409 GBytes	58.6 Gbits/sec	228

CPU Utilization: local/sender 99.9%, remote/receiver 97.4%

Figure 5.16: Results of iperf3 tests with Linux bridge as transparent firewall

In the third sub-case the firewall is implemented as a transparent firewall by a user application, which takes packets from one interface, analyzes it and sends the packet to the other interface.

User application as firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	128 MBytes	1.07 Gbits/sec	0	1.35 MBytes
1.00-2.00 sec	147 MBytes	1.24 Gbits/sec	0	1.35 MBytes
2.00-3.00 sec	154 MBytes	1.29 Gbits/sec	0	1.35 MBytes
3.00-4.00 sec	138 MBytes	1.16 Gbits/sec	0	1.35 MBytes
4.00-5.00 sec	143 MBytes	1.20 Gbits/sec	0	1.35 MBytes
5.00-6.00 sec	154 MBytes	1.29 Gbits/sec	0	1.35 MBytes
6.00-7.00 sec	149 MBytes	1.24 Gbits/sec	0	1.35 MBytes
7.00-8.00 sec	139 MBytes	1.17 Gbits/sec	0	1.35 MBytes
...	...	...	...	...
56.00-57.00 sec	154 MBytes	1.29 Gbits/sec	0	1.35 MBytes
57.00-58.00 sec	154 MBytes	1.29 Gbits/sec	0	1.35 MBytes
58.00-59.00 sec	153 MBytes	1.29 Gbits/sec	0	1.35 MBytes
59.00-60.00 sec	154 MBytes	1.29 Gbits/sec	0	1.35 MBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	8.79 GBytes	1.26 Gbits/sec	0

CPU Utilization: local/sender 28.4%, remote/receiver 43.8%

Figure 5.17: Results of iperf3 tests with user application as firewall

5.2.2 Second scenario: NSM forwarder without VPP in chain

As in the second case of the first scenario, also both cases of this scenario were tested with two different implementation of the firewall application, the first one as iptables rules, the second one as user application.

Case 1: NSM kernel forwarder

Iptables as firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	6.99 GBytes	60.0 Gbits/sec	0	857 KBytes
1.00-2.00 sec	7.11 GBytes	61.1 Gbits/sec	0	857 KBytes
2.00-3.00 sec	7.11 GBytes	61.0 Gbits/sec	0	903 KBytes
3.00-4.00 sec	7.14 GBytes	61.3 Gbits/sec	0	903 KBytes
4.00-5.00 sec	7.14 GBytes	61.3 Gbits/sec	0	903 KBytes
5.00-6.00 sec	6.99 GBytes	60.1 Gbits/sec	0	903 KBytes
6.00-7.00 sec	7.04 GBytes	60.5 Gbits/sec	0	1.37 MBytes
7.00-8.00 sec	7.22 GBytes	62.0 Gbits/sec	0	1.37 MBytes
...	...	...	...	...
56.00-57.00 sec	7.17 GBytes	61.6 Gbits/sec	0	3.09 MBytes
57.00-58.00 sec	7.17 GBytes	61.6 Gbits/sec	0	3.09 MBytes
58.00-59.00 sec	7.03 GBytes	60.4 Gbits/sec	0	3.09 MBytes
59.00-60.00 sec	7.13 GBytes	61.2 Gbits/sec	0	3.09 MBytes

Summary results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	427 GBytes	61.1 Gbits/sec	4

CPU Utilization: local/sender 99.6%, remote/receiver 74.0%

User application as firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	315 MBytes	2.64 Gbits/sec	0	3.13 MBytes
1.00-2.00 sec	345 MBytes	2.89 Gbits/sec	0	3.13 MBytes
2.00-3.00 sec	344 MBytes	2.88 Gbits/sec	0	3.13 MBytes
3.00-4.00 sec	340 MBytes	2.85 Gbits/sec	0	3.13 MBytes
4.00-5.00 sec	342 MBytes	2.87 Gbits/sec	0	3.13 MBytes
5.00-6.00 sec	342 MBytes	2.87 Gbits/sec	0	3.13 MBytes
6.00-7.00 sec	306 MBytes	2.57 Gbits/sec	11	2.52 MBytes
7.00-8.00 sec	334 MBytes	2.80 Gbits/sec	0	3.03 MBytes
...	...	...	...	...
56.00-57.00 sec	346 MBytes	2.90 Gbits/sec	0	3.04 MBytes
57.00-58.00 sec	345 MBytes	2.89 Gbits/sec	0	3.04 MBytes
58.00-59.00 sec	344 MBytes	2.88 Gbits/sec	0	3.04 MBytes
59.00-60.00 sec	334 MBytes	2.80 Gbits/sec	0	3.04 MBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	19.0 GBytes	2.73 Gbits/sec	305

CPU Utilization: local/sender 22.4%, remote/receiver 13.4%

Figure 5.18: Results of iperf3 tests

Case 2: NSM VPP forwarder

Iptables as firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	261 MBytes	2.19 Gbits/sec	293	788 KBytes
1.00-2.00 sec	160 MBytes	1.34 Gbits/sec	0	928 KBytes
2.00-3.00 sec	189 MBytes	1.58 Gbits/sec	0	1.05 MBytes
3.00-4.00 sec	174 MBytes	1.46 Gbits/sec	74	877 KBytes
4.00-5.00 sec	150 MBytes	1.26 Gbits/sec	5	738 KBytes
5.00-6.00 sec	125 MBytes	1.05 Gbits/sec	33	645 KBytes
6.00-7.00 sec	114 MBytes	954 Mbits/sec	20	407 KBytes
7.00-8.00 sec	93.8 MBytes	786 Mbits/sec	0	554 KBytes
...	...	...	...	...
56.00-57.00 sec	200 MBytes	1.68 Gbits/sec	0	1.11 MBytes
57.00-58.00 sec	148 MBytes	1.24 Gbits/sec	53	666 KBytes
58.00-59.00 sec	142 MBytes	1.20 Gbits/sec	0	817 KBytes
59.00-60.00 sec	139 MBytes	1.16 Gbits/sec	54	704 KBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	8.67 GBytes	1.24 Gbits/sec	1304

CPU Utilization: local/sender 4.3%, remote/receiver 23.0%

User application as firewall:

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	65.3 MBytes	548 Mbits/sec	461	1.22 MBytes
1.00-2.00 sec	88.8 MBytes	744 Mbits/sec	0	1.31 MBytes
2.00-3.00 sec	71.2 MBytes	598 Mbits/sec	349	1010 KBytes
3.00-4.00 sec	71.2 MBytes	598 Mbits/sec	0	1.04 MBytes
4.00-5.00 sec	72.5 MBytes	608 Mbits/sec	0	1.08 MBytes
5.00-6.00 sec	72.5 MBytes	608 Mbits/sec	0	1.12 MBytes
6.00-7.00 sec	76.2 MBytes	640 Mbits/sec	0	1.17 MBytes
7.00-8.00 sec	86.2 MBytes	724 Mbits/sec	0	1.22 MBytes
...	...	...	...	...
56.00-57.00 sec	72.5 MBytes	608 Mbits/sec	0	1.02 MBytes
57.00-58.00 sec	72.5 MBytes	608 Mbits/sec	0	1.07 MBytes
58.00-59.00 sec	65.0 MBytes	545 Mbits/sec	97	824 KBytes
59.00-60.00 sec	56.2 MBytes	472 Mbits/sec	0	925 KBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	4.41 GBytes	631 Mbits/sec	1905

CPU Utilization: local/sender 3.3%, remote/receiver 8.1%

Figure 5.19: Results of iperf3 tests

5.2.3 Third scenario: NSM forwarder with VPP in chain

Case 1: VPP firewall

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	258 MBytes	2.16 Gbits/sec	183	1.57 MBytes
1.00-2.00 sec	266 MBytes	2.23 Gbits/sec	0	1.69 MBytes
2.00-3.00 sec	258 MBytes	2.16 Gbits/sec	0	1.80 MBytes
3.00-4.00 sec	255 MBytes	2.14 Gbits/sec	36	1.39 MBytes
4.00-5.00 sec	271 MBytes	2.28 Gbits/sec	0	1.52 MBytes
5.00-6.00 sec	272 MBytes	2.29 Gbits/sec	0	1.64 MBytes
6.00-7.00 sec	265 MBytes	2.22 Gbits/sec	0	1.75 MBytes
7.00-8.00 sec	275 MBytes	2.31 Gbits/sec	0	1.87 MBytes
...	...	...	...	...
56.00-57.00 sec	262 MBytes	2.20 Gbits/sec	0	1.68 MBytes
57.00-58.00 sec	259 MBytes	2.17 Gbits/sec	0	1.80 MBytes
58.00-59.00 sec	261 MBytes	2.19 Gbits/sec	0	1.89 MBytes
59.00-60.00 sec	259 MBytes	2.17 Gbits/sec	0	1.96 MBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	14.7 GBytes	2.10 Gbits/sec	2282

CPU Utilization: local/sender 6.6%, remote/receiver 32.6%

Figure 5.20: Results of iperf3 tests

Case 2: VPP technology across the chain

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	49.6 MBytes	416 Mbits/sec	0	0.00 Bytes
1.00-2.00 sec	54.2 MBytes	455 Mbits/sec	0	0.00 Bytes
2.00-3.00 sec	54.1 MBytes	454 Mbits/sec	0	0.00 Bytes
3.00-4.00 sec	54.0 MBytes	453 Mbits/sec	0	0.00 Bytes
4.00-5.00 sec	54.0 MBytes	453 Mbits/sec	0	0.00 Bytes
5.00-6.00 sec	54.2 MBytes	455 Mbits/sec	0	0.00 Bytes
6.00-7.00 sec	54.2 MBytes	455 Mbits/sec	0	0.00 Bytes
7.00-8.00 sec	54.2 MBytes	454 Mbits/sec	0	0.00 Bytes
...	...	...	...	...
56.00-57.00 sec	54.1 MBytes	454 Mbits/sec	0	0.00 Bytes
57.00-58.00 sec	54.3 MBytes	456 Mbits/sec	0	0.00 Bytes
58.00-59.00 sec	54.3 MBytes	455 Mbits/sec	0	0.00 Bytes
59.00-60.00 sec	54.3 MBytes	455 Mbits/sec	0	0.00 Bytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	3.17 GBytes	454 Mbits/sec	0

CPU Utilization: local/sender 100.0%, remote/receiver 100.0%

Figure 5.21: Results of iperf3 tests

5.2.4 Forth scenario: eBPF

Case 1: transparent firewall

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	8.20 GBytes	70.4 Gbits/sec	0	283 KBytes
1.00-2.00 sec	8.13 GBytes	69.8 Gbits/sec	0	419 KBytes
2.00-3.00 sec	8.05 GBytes	69.2 Gbits/sec	0	485 KBytes
3.00-4.00 sec	7.91 GBytes	67.9 Gbits/sec	0	485 KBytes
4.00-5.00 sec	8.08 GBytes	69.4 Gbits/sec	0	485 KBytes
5.00-6.00 sec	7.93 GBytes	68.1 Gbits/sec	0	485 KBytes
6.00-7.00 sec	7.87 GBytes	67.6 Gbits/sec	0	485 KBytes
7.00-8.00 sec	7.97 GBytes	68.5 Gbits/sec	0	485 KBytes
...	...	...	...	...
56.00-57.00 sec	7.92 GBytes	68.1 Gbits/sec	0	4.20 MBytes
57.00-58.00 sec	8.07 GBytes	69.3 Gbits/sec	0	4.20 MBytes
58.00-59.00 sec	7.92 GBytes	68.1 Gbits/sec	0	4.20 MBytes
59.00-60.00 sec	8.05 GBytes	69.1 Gbits/sec	0	4.20 MBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	479 GBytes	68.6 Gbits/sec	0

CPU Utilization: local/sender 99.9%, remote/receiver 95.7%

Figure 5.22: Results of iperf3 tests

Case 2: transparent firewall with pbforwarder

Interval	Transfer	Bandwidth	Retr	Cwnd
0.00-1.00 sec	7.43 GBytes	63.8 Gbits/sec	0	494 KBytes
1.00-2.00 sec	7.47 GBytes	64.2 Gbits/sec	0	519 KBytes
2.00-3.00 sec	7.36 GBytes	63.2 Gbits/sec	0	574 KBytes
3.00-4.00 sec	7.27 GBytes	62.4 Gbits/sec	0	574 KBytes
4.00-5.00 sec	7.33 GBytes	62.9 Gbits/sec	0	574 KBytes
5.00-6.00 sec	7.32 GBytes	62.8 Gbits/sec	0	574 KBytes
6.00-7.00 sec	7.34 GBytes	63.0 Gbits/sec	0	734 KBytes
7.00-8.00 sec	7.23 GBytes	62.1 Gbits/sec	0	734 KBytes
...	...	...	...	...
56.00-57.00 sec	7.26 GBytes	62.3 Gbits/sec	0	2.01 MBytes
57.00-58.00 sec	7.30 GBytes	62.7 Gbits/sec	0	2.01 MBytes
58.00-59.00 sec	7.30 GBytes	62.8 Gbits/sec	0	2.12 MBytes
59.00-60.00 sec	7.25 GBytes	62.2 Gbits/sec	0	2.12 MBytes

Summary Results:

Interval	Transfer	Bandwidth	Retr
0.00-60.00 sec	437 GBytes	62.6 Gbits/sec	4086

CPU Utilization: local/sender 99.9%, remote/receiver 97.8%

Figure 5.23: Results of iperf3 tests

5.3 Final discussion

Figure 5.24 gives a graphic representation of the comparison between obtained bandwidth in all different scenarios. Label such as "1S2C-usr-app" means that this value refers to the first case (1C) of the second scenario (1S) and usr-app means that the implementation of the firewall was provided by a user application.

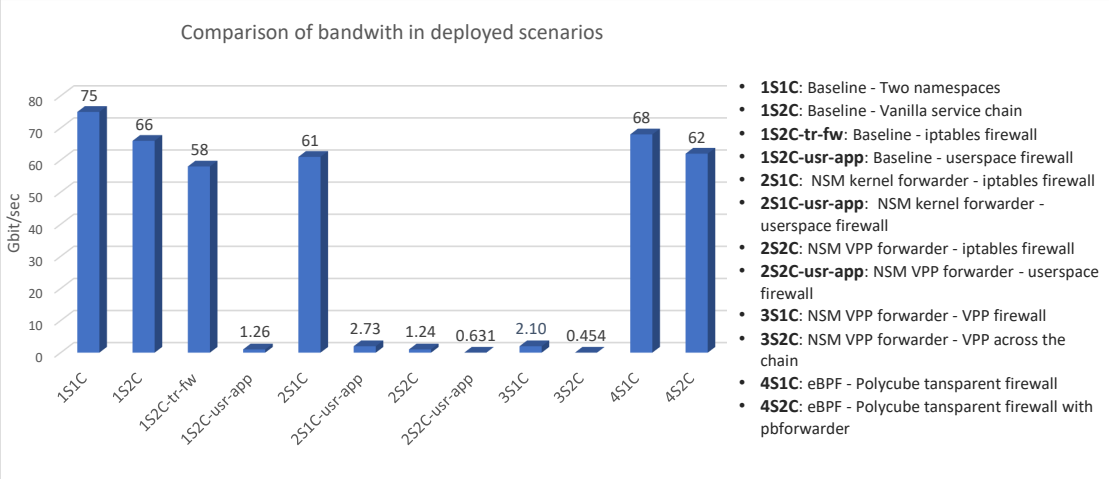

Figure 5.24: Comparison graph between bandwidth in scenarios

It is necessary to point out some notes about obtained results:

- high performance was obtained in first and fourth scenario where virtual ethernet interfaces were used for communication between pods. Also first case of second scenario involves veth pairs in communication of involved pods, in fact performance is comparable to other mentioned scenarios.

- In scenarios where firewall application was implemented both as iptables rules and as user application, better performance was obtained in the former case. The reason of this difference is associated to the route and the forwarding of the traffic in the two cases: in the first case, as firewall is implemented as iptables rules that are implemented directly in kernel space of the specific namespace (or, in Kubernetes environment, of the pod) and forwarding is implemented as well in kernel space by routing table, traffic does not move from kernel space to user space, its processing is entirely in kernel space. In the second case, the firewall application (that implements also the forwarding) runs in user space, so the processing of the traffic happens in user space: this

means that traffic needs to move from kernel space when it enters inside the namespace or in the pod where the application runs (through first interface), to user space when it needs to be elaborated from firewall user application and then back to kernel space when it exits from the namespace or the pod (through second interface) to go to destination. This switch between kernel and user space, together with the fact that kernel space has better performance in forwarding traffic, generates a overhead in all traffic processing, impacting on obtained performance.

- low performance are obtained in scenarios where VPP technology is used in NSM forwarder and pods implementing the service chain do not runs with the same technology. In this case, performance is low due to TAP interfaces that the forwarder configures in involved pods and itself for communication. This kind of interface seems to be not optimized for performance and, as result, obtained numbers are low compared to other scenarios where this kind of interface is not used.

- even worse performance is obtained in the case in which VPP is used across all the chain (i.e. both pods that implement the service chain and NSM forwarder runs with this technology). However, the reason is different from the previous case; in fact here MEMIF interfaces are configured in pods involved in communication. As explained in the description of this case, VPP does not support native integration with iperf application, so specific manual configuration of both iperf server and iperf client had to be provided to allow this case work properly. This configuration is adapted from the one provided by VPP documentation and it is possible that some parameters do not have the best values to obtain best performance and they may impact on results. Furthermore, VPP developers claim that current implementation of VPP is not optimized for performance with standard application such as iperf. Nevertheless, as NSM project is deeply involved with VPP technology, tests of performance were executed also with this technology and results are reported as well.

In terms of configuration of different scenarios to work correctly, the situation is different respect to performance results:

- In Kubernetes scenarios, speaking about configuration of pods that were involved in communication, it was sufficient for both VPP and non VPP scenarios to write YAML's templates of deployed pods and with few specific commands deploy NSM control plane pods and the ones involved in the communication. On the other hands, in VPP scenarios firewall functionalities had to be specifically implemented to work with VPP Agent inside VPP Agent

itself, while in non VPP scenarios the implementation of the firewall with non-specific methods (i.e. iptables rules or user application).

- Speaking about configuration of the application to run the tests (i.e. iperf3), non VPP scenarios require specific commands to allow proper communication of pods through cross-connections created by NSM control plane. In particular, routing table's entries needed to be manually configured in pods to allow reachability from iperf client to iperf server. On the other side, scenarios that involve VPP in communicating pods do not require further manual configuration to communicate through NSM's cross-connections and traffic can immediately traverse the chain from the head to the tail.

- In terms of configuration of elements to work with iperf application, besides installing proper Linux packet to execute iperf3 command, non VPP scenarios did not require any further specific configuration. VPP scenarios required instead specific command and configuration to allow iperf interact properly with VPP Agent.

Finally, CPU utilization needs to be commented as well. Iperf3 assigns one thread to iperf server and one thread to iperf client. These threads, if not specified, are scheduled by the Operating System arbitrarily to two different CPUs of the ones available in the machine where iperf3 is running. Analyzing how this scheduling happens, during tests these two threads can be rescheduled in different CPUs depending on different factors (load on each CPU, different priority of running processes, etc.) and the load of a thread can be distributed uniformly on more CPUs if Operating System deems it necessary. Reported data show that CPU utilization in most of the cases is 100%: as iperf communication consists of transmission and processing of TCP traffic, most of the CPU is used from Kernel to forward traffic, while the remaining part is dedicated to the actual iperf thread which elaborates and send (or receive) the traffic. This means that the two threads use the totality of available resources of the CPU on which they are scheduled, always giving best effort in communication. Note that Operating System always tries not to schedule iperf client and iperf server's threads on same CPU, so that they do not risk to get less resources of the CPU where they are running.

In those cases were CPU utilization is low, the main difference with previous situation is how the forwarding of traffic happens. When the forwarding is provided by the kernel (i.e. with routing tables), most of the CPU is used to provide this forwarding. In those cases where the forwarding is provided by a user application, the kernel does not consume CPU resources to forward traffic and, for this reason, the CPU is only used by the user application and the iperf thread which elaborate and forward TCP traffic. As kernel is consistently faster than user application in providing traffic forwarding, better performance is obtained in those cases.

Chapter 6

Conclusions and Future Works

Network Service Mesh project is nowadays the most promising solution to allow integration of Virtual Network Functions in Kubernetes environment. After the analysis carried out in this thesis, following positive aspects can be pointed out:

- Effective chaining of pods: once the chain is established and pods implementing VNFs are deployed, the cross-connections really allow communication between them. Also the order of VNFs in the chain is respected, i.e. if a VNF is defined to be chained before another one in the Network Service implementation, the traffic will flow inside the first VNF and then inside the second VNF.
- Supported technologies: NSM supports standard technology and VPP technology in its data plane component (i.e. the forwarder) and in pods that need to communicate. It also provides an already implemented support for VXLAN tunneling in inter-node communication. Moreover, it provides APIs for forwarder implementation, which means that own specific implementation of the forwarder can be created and deployed.
- Self-healing system: NSM comes with a full working monitoring system, which does not only concerns with cross-connections' state, but also with pods involved in NSM environment. Specifically, if a cross-connection for some reasons fails, the NSM forwarder that created it advertises this event and sends a message to NSM Manager, which restarts the process of creation of cross-connection to substitute the failed one. On the other hand, if pod implementing a VNF stops working, the NSM Manager advertises it and tries to create a cross-connection with the replica pods that implement the same VNF or, if no replicas are present, with the new same pod as soon as it is recreated.

Besides positive aspects, some other elements of this solution should be improved and developed:

- Integration with VPP technology: VPP is becoming more and more used in network world and it is properly used by NSM to allow more automatic configuration of communication between pods. Nevertheless, it is still under deep development and NSM should collaborate and better integrate VPP technology in its solution, mostly in terms of performance where, as it can be seen from obtained results, it does not behave as standard and older technologies.

- Supported technologies for communication: as said before, NSM does support more than one technology in communication of pods and also in the forwarder. However, NSM claims that its solution can support and work with any desired technology (i.e. SR-IOV for intra-node communication, BGP or MPLS for inter-node communication), but functionalities in control plane elements still need to be developed for other kind of technologies rather than the ones already supported.

- Support for dynamic change of service chain: in VNF service chain is sometimes necessary to dynamically change the chain (e.g. in order to send the traffic to a VNF rather then to another one). NSM does not supports this aspect, but its developers are working on it to provide this behavior.

- Load balancing: speaking about a more specific aspect, NSM does not provide load balancing between NSEs that are replicas of same VNF implementation. This means that if a NSM Manager receives more requests of cross-connection and it knows that a pod implementing the required VNF is present on same node, it will send all requests to it, not balancing requests to other possible pods implementing same VNFs.

Performance tests were executed on one physical server and, in Kubernetes cases, in one worker node. As future work of this thesis, to complete the benchmarking of NSM same scenarios will be deployed in different Kubernetes nodes on same physical machine, and on different Kubernetes nodes on different physical machines.

For what concerns Network Service Mesh future works, developers are mainly focusing, besides improving already provided features, on inter-cluster communication of pods. This is an important and more and more relevant aspect that telecom operators are requiring: it is frequent that one Virtual Network Function runs as a pod in a Kubernetes cluster and another VNF the first one needs to be chained to is running as another pod in another Kubernetes cluster (possibly in another physical server). NSM is working on providing the possibility of this communication.

Bibliography

[1] *Kubernetes documentation*. https://kubernetes.io/docs/home/. Feb. 2019.

[2] *Container Network Interface*. https://github.com/containernetworking/cni. 2019.

[3] *What is VPP*. https://wiki.fd.io/view/VPP/What_is_VPP%3F. 2010.

[4] *VPP*. https://aaltodoc.aalto.fi/handle/123456789/34747. 2018.

[5] *VPP Agent*. https://github.com/ligato/vpp-agent. 2020.

[6] *MEMIF interface*. https://doc.dpdk.org/guides/nics/memif.html. 2019.

[7] *TUN/TAP interfaces*. https://backreference.org/2010/03/26/tuntap-interface-tutorial/. 2010.

[8] *TUN, TAP and Veth - Virtual Networking Devices Explained*. https://www.fir3net.com/Networking/Terms-and-Concepts/virtual-networking-devices-tun-tap-and-veth-pairs-explained.html. 2017.

[9] *Container Network Interface Specification*. https://github.com/containernetworking/cni/blob/master/SPEC.md. 2020.

[10] *Comparison between different CNI*. https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/. 2019.

[11] *Bridge CNI*. https://github.com/containernetworking/plugins/blob/master/plugins/main/bridge/README.md/. 2019.

[12] *Flannel CNI plugin*. https://github.com/containernetworking/plugins/tree/master/plugins/meta/flannel. 2019.

[13] *Calico CNI*. https://docs.projectcalico.org/v3.10/reference/cni-plugin/configuration. 2019.

[14] *Multus CNI*. https://builders.intel.com/docs/networkbuilders/enabling_new_features_in_kubernetes_for_NFV.pdf. 2019.

[15] *Contiv-VPP CNI*. https://github.com/contiv/vpp/blob/master/docs/ARCHITECTURE.md. 2019.

[16] *Network Service Mesh*. https://networkservicemesh.io/. 2020.

[17] *NSM Admission Webhook*. https://github.com/networkservicemesh/networkservicemesh/blob/master/docs/spec/admission.md. 2020.

Appendix A

Automatic scripts

To make the process of performance testing automatic, a suite of scripts[1] was developed. Once the script starts, it creates elements involved in the test, executes the test and prints results of the test as output. At the end, it deletes created elements, so that the environment of execution is not affected by the test.

Scripts are written in bash and uses iperf3 tool to obtain performance results. Also Helm deployments' templates[2] for Kubernetes related tests were manually created.

Tests were executed following this steps:

- reboot server where test was to be launched
- run command that starts the automatic script for test
- take note of the printed results

This steps guarantee that obtained performance results were reliable, which means that they were not influenced by any possible factor related to other running processes on servers where tests were executed.

[1] https://github.com/RaffaeleTrani/scripts

[2] Each Helm deployment can be found at appropriate Git repository at https://github.com/RaffaeleTrani

Acknowledgements

Lascio le ultime righe di questa tesi in italiano per ringraziare chi mi ha sostenuto durante questo lungo percorso che mi ha portato a questo traguardo.

Il ringraziamento principale va alla mia famiglia e in particolare ai miei genitori, che mi hanno dato sempre un grande supporto anche nei momenti difficili di questo percorso.

Ringrazio i miei parenti e tutti gli amici che, a modo loro, mi hanno incoraggiato e sostenuto nel portare a termine questo percorso.

Ringrazio i miei colleghi nonchè amici che mi hanno accompagnato e con il quale ho condiviso gioie e dolori per raggiungere questo traguardo.

Ringrazio Alex e tutti i ragazzi del laboratorio 9 per l'aiuto datomi durante lo sviluppo di questa tesi.

Infine, ringrazio il mio relatore, il professore Fulvio Risso, e Alessandro D'Alessandro, ingegnere dell'azienda TIM con il quale ho collaborato, per avermi dato l'opportunità di realizzare questo progetto e per il supporto costante e gli stimoli che mi hanno permesso di ampliare le mie conoscenze e portare a termine questa tesi.

Raffaele