POLITECNICO DI TORINO

Master of Science in Computer Engineering

Master's Degree Thesis

Enabling Seamless Autoscaling of Service Function Chains in Kubernetes

Supervisors

Prof. Fulvio RISSO

Ing. Federico PAROLA

Ing. Giuseppe OGNIBENE

Candidate

Francesco MONACO

April 2022

Abstract

Service Function Chains (SFCs) are typical components of the edge networks of telecommunications operators. They are composed by chained network functions which were originally deployed as physical appliances, subsequently as Virtual Network Functions (VNFs) and recently a new paradigm shift towards Cloud Native Network Functions (CNFs) is taking place. Kubernetes is the reference platform for running cloud native workloads, however it is not natively oriented to network functions. This thesis presents the prototype of a system to integrate SFCs in Kubernetes, with support for seamless autoscaling of chain elements. The proposed solution enables the creation of arbitrary chains starting from a logical description and a set of network functions deployments. In order to dynamically adapt to the horizontal scaling of network functions, a controller that watches and reacts to the events concerning scaling is included in the system. Traffic among network functions replicas is distributed by transparent load balancers implemented using eBPF (Extended Berkeley Packet Filter), whose logic is directly injected in the host network interfaces corresponding to the ends of pod links. Requirements for multiple interfaces in pods are satisfied using Multus CNI and a custom Container Network Interface (CNI) plugin. All this is done in a transparent way towards network functions which perceive as they are always connected to single instances of other network functions. The solution has been validated implementing a possible use case of a SFC and evaluated in terms of reaction times to the scaling of network functions. Furthermore, eBPF-based load balancer performance has been compared to other interconnections mechanisms provided by Linux Kernel. The proposed prototype supports the creation of chains for replicas that are all on the same node, future works may extend this feature to support more complex cross-node scenarios.

Table of Contents

List of Figures

Acronyms

ARP
: Address Resolution Protocol

CNI
: Container Network Interface

CNF
: Cloud Native Network Function

CRD
: Custom Resource Definition

eBPF
: extended Berkeley Packet Filter

IP
: Internet Protocol

JSON
: JavaScript Object Notation

MAC
: Media Access Control

NAT
: Network Address Translation

NFV
: Network Function Virtualization

SFC	Service Function Chain
SDN	Software Defined Networking
TCP	Transport Control Protocol
VM	Virtual Machine
VNF	Virtual Network Function

Chapter 1

Introduction

Telecommunications operators are exploring possibilities to shift their functionalities towards Cloud Native infrastructures such as Kubernetes, which promise to bring benefits mainly in terms of increased flexibility, scalability, availability, and cost saving. A substantial part of their workloads is represented by Network Functions (NFs) which are often chained together forming Service Functions Chains (SFCs).

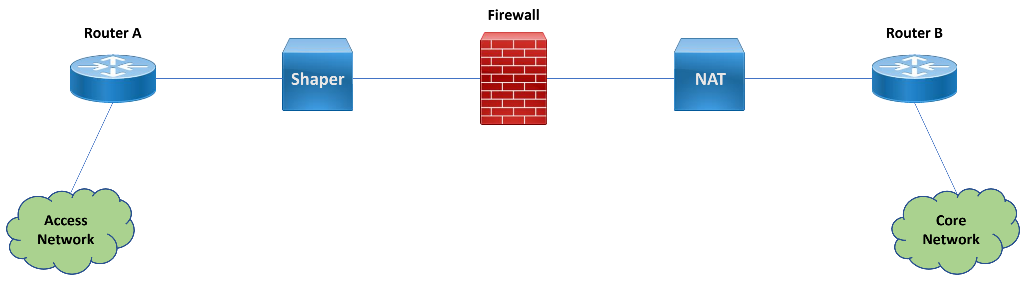


Figure 1.1: A possible Service Function Chain.

Although Kubernetes is currently de facto standard orchestrator for general-purpose workloads, it lacks some functionalities required by network functions workloads, such as chaining mechanisms and the possibility to have pod with multiple network interfaces.

Furthermore, the Service abstraction provided by Kubernetes, which enables scaling and load balancing for applications, is not suitable for network functions since typically they are not the final recipient of the network traffic. Projects such as Multus CNI and Network Service Mesh (NSM) aim to overcome some of these limitations in Kubernetes. In particular Multus enables Pod with multiple network interfaces, while NSM goes even further allowing the creation of chains, or more in general meshes of network functions, but it does not have a native logic to take advantage of the scaling features of Kubernetes.

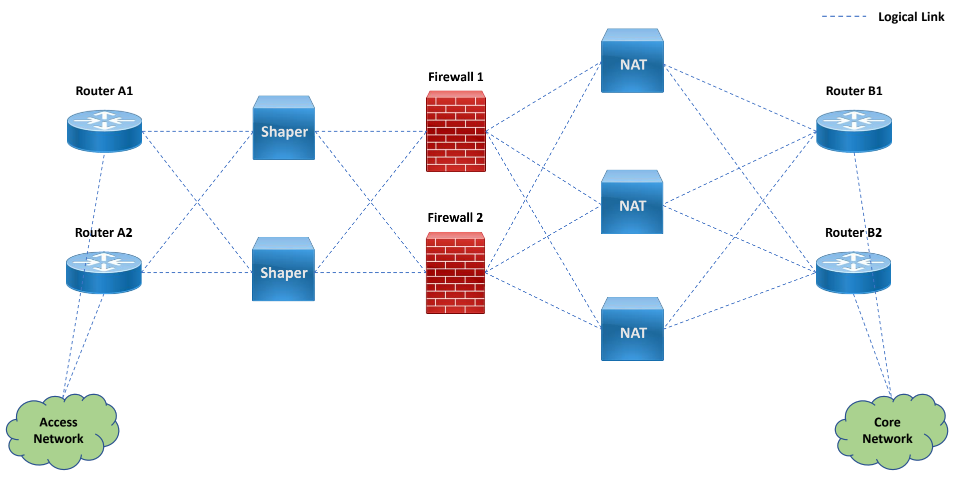


Figure 1.2: A possible scaled Service Function Chain.

This work describes the prototype of a new alternative system to integrate SFCs in Kubernetes with the aim to leverage the *autoscaling* capabilities provided by orchestrator itself. In the next chapters the architecture and the implementation of the system will be described in detail.

1.1 Goals of the thesis

All the work of this thesis was conducted with three main objectives in mind:

- Enabling seamless autoscaling of SFCs: the connectivity among replicas must be handled in a transparent way towards network functions, which do not have to notice anything different while system adapts to replicas variations.
- Allow declarative definition of SFCs: it should be possible to create a SFC with a simple description at logical level on how elements in the chain should be interconnected, without any further low-level detail.
- Allow the creation of SFCs at both L2/L3 levels: it's required the support for Network Function that works at Layer 3, namely, services that have an IP address and receive frames with destination MAC set to that of their ingress interface, or Layer 2, namely, transparent Network Functions which don't have IP addresses and receive frames with destination MAC being that of the original destination.

In the following chapters, after an overview of the technologies and tools used, it will be described how these objectives have been achieved.

Chapter 2

Background

In this chapter there is an overview of the context in which this work is placed. A brief history of the of Network functions will be presented, followed by the description of the Kubernetes architecture. In the end is proposed a related work on SFCs integration in Kubernetes.

2.1 Evolution of Network Functions

A Network function is a functional building block within a network infrastructure that has well-defined external interfaces and a well-defined functional behaviour [1]

Although the concept has never changed, NF has been implemented using various technologies over the years. In this section is shown how their evolution took place; content is freely drawn from the Telecom User Group White Paper [2].

2.1.1 Network Functions as Physical Devices

Until the 1990s, most functionality and applications for telecommunications resided primarily in physical hardware, known also as "appliances". Their software functioned more like firmware and was not easy to abstract and run on in virtual environments.

This approach was resource intensive, expensive and not flexible as it required physical intervention by operators in the event of malfunctions or changes. In this scenario, redundancy meant acquiring multiple physical instances of required appliances. Furthermore, this architecture encouraged the bundling of services and capabilities into monolithic physical systems.

2.1.2 The emergence of VNFs

In the late 1990s, the concept of virtualization for enterprise computing gained interest as enterprises were tired of excessive costs of high-powered servers. Simultaneously, there was a rapid rise in the popularity of the commodity Open Source Linux Operating System.

In 1999, VMware released its first virtualization product for x86 devices. In the early 2010s, the concept of Network Functions Virtualisation (NFV) emerged as telcos sought to virtualize their networking functionality, enhance resiliency, and take advantage of commodity hardware. With constructs such as NFV-encapsulated Virtual Network Functions (VNFs), Network Function Infrastructure (NFVI), and an orchestration component (MANO), the entire NFV hardware and software stack could be integrated and distributed across data centers.

In 2017, Open Networking Automation Platform (ONAP) emerged as the open source standard platform for orchestrating and automating physical and virtual network elements with full lifecycle management. NFV and ONAP primarily focused on creating and orchestrating virtual analogues for physical equipment, i.e. virtual boxes. This was the same pattern followed during server virtualization when initial efforts focused on replicating physical servers as virtual entities.

2.1.3 The advent of CNFs

With the arrival and spread of cloud platforms, a new paradigm shift has begun towards cloud native network functions. A cloud native network function (CNF) is a cloud native application that implements a network functionality. A CNF consists of one or more microservices developed using Cloud Native Principles and deployed as containers. An example of a simple CNF is a packet filter that implements a single piece of network functionality as a microservice. A firewall is an example of a CNF which may be composed of more than one microservice (i.e. encryption, decryption, access lists, packet inspection, etc.).

The emergence of Kubernetes as the most widely used container orchestrator allows telecommunications operators to shift their infrastructure to a de facto standard platform, saving money required to maintain custom platforms and improving performance and resiliency.

However, this transition comes with many challenges due to two main factors. On one hand there are gaps between the functionalities offered by the orchestrator and those required by network functions. On the other hand, the network functions must be entirely rewritten following the Cloud Native Principles in order to benefits from the cloud environment. Many projects under the Cloud Native Computing Foundation (CNCF) try to solve these problems.

2.2 Service Function Chains

In the edge network of telecommunication operators is common to have network functions that provide added-value services to end users. Typically, what happens is that there are several network functions in succession that create what is called a *chain*. The term *Service Function Chain* was introduced when SDN technology was used together with NFV for the implementation of composed services in virtual environments, even if the same term is commonly used for other types of implementations as well. Each network function in a SFC has a specific task to perform on traffic, they can work at any level of the OSI stack. Some examples of SFs are Firewalls, NATs, Traffic Shapers, Traffic Optimizers and so on.

SFC are defined by a list of SFs and a set of rules which determine in which order the traffic must flow among them. They can be unidirectional or bidirectional. In the first case the traffic is only forwarded in one direction and the reverse traffic is not processed, whereas in the second case the traffic must pass through the chain in both directions in a symmetric way.

With the advent of cloud technologies and CNFs, new possibilities have opened up for the implementation of more flexible chains that can benefit from cloud environments features.

2.3 Kubernetes

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, which facilitates both declarative configuration and automation [3].

When it is deployed, a cluster is obtained. A cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker nodes host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple machines and a cluster usually runs multiple nodes, providing fault-tolerance and high availability. The Figure 2.1 shows some of the main Kubernetes components.

2.3.1 Control Plane Components

The control plane's components make global decisions about the cluster, as well as detecting and responding to cluster events. 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.

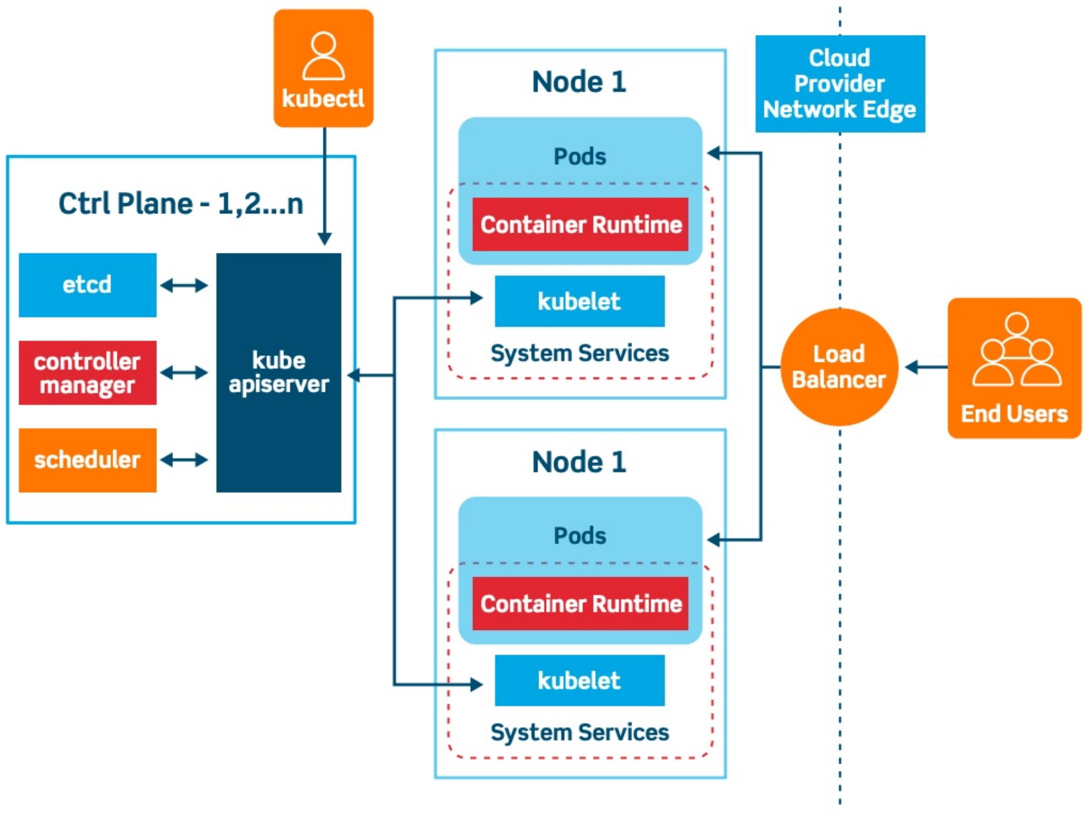


Figure 2.1: Components of a Kubernetes cluster [4].

Here is the list of components with a brief description:

- *kube-apiserver*: 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*, it is designed to scale horizontally.
- *etcd*: 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*: control plane component that embeds cloud-specific control logic. The *cloud controller manager* links cluster to cloud provider's API. The cluster does not have a *cloud controller manager* when it runs on-premises infrastructure.

2.3.2 Node Components

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. The kubelet takes a set of *PodSpecs* that are provided through various mechanisms and ensures that the containers described in those *PodSpecs* are running and healthy. The kubelet does not manage containers which were not created by Kubernetes.

- *kube-proxy*: kube-proxy is a network proxy that runs on each node in the cluster, implementing part of the Kubernetes Service concept. *kube-proxy* maintains network rules on nodes. These network rules allow network communication to Pods from network sessions inside or outside of cluster. *kube-proxy* uses the operating system packet filtering layer if there is one and it is available. Otherwise, *kube-proxy* forwards the traffic itself.

- *Container runtime*: The container runtime is the software that is responsible for running containers. Kubernetes supports *container runtimes* such as containerd, CRI-O, and any other implementation of the Kubernetes CRI

2.3.3 Workloads

A workload is an application running on Kubernetes. Whether workload is a single component or several that work together, on Kubernetes it runs inside a set of *Pods*. Pods are the smallest deployable units of computing in Kubernetes, they represent a set of running containers and they have a defined *lifecycle*. Even if it is possible to manage each Pod directly, typically their management is delegated to other resources. *Workload resources* can be used to manage the entire pod lifecycle. They configure controllers that make sure the right number of the right kind of pod are running, in order to match the state specified.

Kubernetes provides several built-in workload resources:

- *Deployment and ReplicaSet*, which represent the best choice for managing a stateless application workload on clusters, where any Pod in the *Deployment* is interchangeable and can be replaced if needed.

- *StatefulSet*, that allows to run one or more related Pods that do track state somehow. For example, if workload records data persistently, a *StatefulSet* that matches each Pod with a *PersistentVolume* can be used.

- *DaemonSet*, that defines Pods which provide node-local facilities. These might be fundamental to the operation of the cluster, such as a networking helper tool, or be part of an add-on. Every time a node that matches the specification in a *DaemonSet* is added to the cluster, the control plane schedules a Pod for that *DaemonSet* onto the new node.

- *Job and CronJob*, that define tasks which run to completion and then stop. Jobs represent one-off tasks, whereas *CronJobs* recur according to a schedule.

In the wider Kubernetes ecosystem, there are third-party workload resources that provide additional behaviours. Using a custom resource definition, is possible to add a third-party workload resource to obtain a specific behaviour that's not part of Kubernetes' core.

2.3.4 Network model

In Kubernetes, every Pod must get its own IP address, routable inside the cluster. This means users usually do not need to explicitly create links between Pods and they almost never need to deal with mapping container ports to host ports.

Kubernetes imposes the following fundamental requirements on any networking implementation:

- pods on a node can communicate with all pods on all nodes without NAT

- agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Additionally for those platforms that support Pods running in the host network (e.g. Linux):

- pods in the host network of a node can communicate with all pods on all nodes without NAT

Kubernetes IP addresses exist at the Pod scope, containers within a Pod share their network namespaces, including their IP address and MAC address. This means that containers within a Pod can all reach each other's ports on localhost. This also means that containers within a Pod must coordinate port usage, but this is no different from processes in a VM. This is called the *IP-per-pod* model.

Networking addresses four concerns:

1. Highly-coupled container-to-container communications: this is solved by Pods and *localhost* communications.
2. Pod-to-Pod communications: solved by *network providers* and their *CNI plugins.*
3. Pod-to-Service communications: this is covered by *services.*
4. External-to-Service communications: this is covered by *services.*

Kubernetes built-in network support, *kubenet*, can provide some basic network connectivity. It does not, of itself, implement more advanced features like cross-node networking or network policies. For this reason, it is more common to use third party network implementations which plug into Kubernetes using the CNI (Container Network Interface) API.

Container Network Interface is a Cloud Native Computing Foundation project that consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted [5].

2.3.5 Services

Services are an abstract way to expose an application running on a set of Pods as a network service. Kubernetes gives a single DNS name for the set of Pods and can load-balance across them.

This is necessary because Pods are non-permanent resources. Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later. This leads to a problem: if some set of *backends* Pods provides functionality to other *frontends* Pods inside the cluster, how do the *frontends* find out and keep track of which IP address to connect to, so that the *frontend* can use the *backend* part of the workload? The problem is solved by *Services.*

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them. The set of Pods targeted by a Service is usually determined by a selector.

Within the cluster the network service is usually represented as *virtual IP address*, and *kube-proxy* load balances connections to the virtual IP across the group of pods backing the service. The virtual IP is discoverable through Kubernetes DNS. The DNS name and virtual IP address remain constant for the lifetime of the service, even though the pods backing the service may be created or destroyed, and the number of pods backing the service may change over time [6].

2.3.6 Custom Resources and Operator Pattern

A custom resource is an *extension* of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects using *kubectl*, just as they do for built-in resources like Pods.

On their own, custom resources allow to store and retrieve structured data. When a custom resource is combined with a *custom controller*, custom resources provide a true declarative API. The Kubernetes declarative API enforces a separation of responsibilities. Users declare the desired state of their resources and the Kubernetes controllers keep the current state of Kubernetes objects in sync with the declared desired state. This is in contrast to an imperative API, where the user instructs a server what to do.

Custom controller can be deployed on a running cluster, independently of the cluster's lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources.

The *Operator pattern* combines custom resources and custom controllers. Custom controllers can be used to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Kubernetes is designed for automation. Out of the box, there are a lots of built-in automation from the core of Kubernetes. Kubernetes can automate deploying and running workloads, and user can automate how Kubernetes does that.

Kubernetes' *operator pattern* concept allow to extend the cluster's behaviour without modifying the code of Kubernetes itself by linking controllers to one or more custom resources. *Operators* are clients of the Kubernetes API that act as controllers for a *Custom Resource*.

2.4 Related Work

Network Service Mesh (NSM) is a Cloud Native Computing Foundation sandbox project [7], that can be used to integrate SFCs in Kubernetes.

NSM try to solve a fundamental problem in Runtime Domains (like Kubernetes): allow individual workloads, wherever they are running, to connect securely to

Network Services that are independent of where they run.

A *Network Service* in NSM is something that provides some features in terms of *Connectivity*, *Security* and *Observability* to the traffic that they handle. The payload of these services can be either IP or Ethernet. For this reason, NSM is complementary to traditional Service Meshes like Istio and Linkerd which focus mainly on L4-L7 payloads.

Examples of Network Services can be a simple distributed vL3 that allows the workloads in the domain to communicate via IP, or a more complex composition of services like a *Service Function Chain*. NSM allows workloads to consume, more generally, Network Services that are provided by a graph of Endpoints composed together.

In NSM, *Network Service Clients* (NSCs) are workloads that desire to be connected to a Network Service. *Network Service Endpoints* (NSEs) are the element that provide Network Services. Both may be Pods but also VMs or physical devices.

Clients are connected to Endpoints through *Virtual Wires* (vWire), each vWire carries traffic for exactly one Network Service.

Forwarders are the data plane components that enable vWire abstraction, NSM provides some reference forwarders based on different technologies such as VPP, OvS, SR-IOV.

Although NSM supports chaining mechanisms for network services, it does not natively provide any *load balancing* mechanism among replicas that takes advantage of the *scaling* of the NSEs in the chain. This means that even if a new replica of a network function in a Network Service is instantiated, this will not be leveraged and traffic will follow the original data path, it only may be selected for other new chains or if the current used replica fails.

The main limitation comes from Forwarders which have traditionally been *Point-to-Point* (P2P), in other words, they were designed to transfer packet from a single ingress to a single egress. Starting with version v1.1.0 [8], NSM introduced support for *Point-to-Multi-Point* (P2MP) forwarders which can manage connections with multiple replicas. However, reference forwarders are still P2P only, this means that currently there is no provided implementation for a P2MP forwarder.

Chapter 3

Foundation technologies

In this chapter the main technologies used for the implementation of the prototype are presented. The first one to be presented is eBPF, that is the technology on which the data plane of the load balancers is based. Netlink is mentioned next, along with two libraries based on its interface which have been used for the manipulation of network interfaces. Then Multus is introduced, a meta-CNI that allows the execution of multiple CNIs and enables Pods with multiple interfaces. In the end, is presented the Kopf framework which was used to implement a Kubernetes operator.

The content of the following sections is mainly drawn from official documentations.

3.1 eBPF (extended Berkeley Packet Filter)

eBPF (Extended Berkeley Packet Filter) is a highly flexible and efficient virtual machine in the Linux kernel that allows to execute bytecode at various hook points in a safe manner [9]. eBPF programs are event-driven and are run when the kernel or an application passes a certain hook point.

It is an evolution of the “classic” BPF (cBPF), which was one of the first packet filter implementation in kernel. The new eBPF instead, has an instruction set that is generic and flexible that make it usable in many other scenario apart from networking. eBPF does not have only a wider instruction set, but it has a whole new infrastructure that expand kernel functionalities.

3.1.1 eBPF vCPU

The vCPU is based on a *general-purpose* RISC instruction set. It is possible to write programs in a subset of C language which are then compiled into vCPU instructions through a compiler. However, those instructions can be mapped to native opcodes by a JIT compiler in the kernel for optimal execution performance.

eBPF consists of eleven 64 bit registers with 32 bit subregisters, a program counter and a 512 byte large eBPF stack space. Registers are named r0 - r10. The operating mode is 64 bit by default, the 32 bit subregisters can only be accessed through special ALU (arithmetic logic unit) operations. The 32 bit lower subregisters zero-extend into 64 bit when they are being written to. Register r10 is the only register which is read-only and contains the frame pointer address in order to access the eBPF stack space. The remaining r0 - r9 registers are general purpose and of read/write nature.

Upon entering execution of a BPF program, register r1 contains the context for the program. The context is the input argument for the program, similar to *argc/argv* pair for a typical C program. The context is defined by the program type, for example, a networking program can have a kernel representation of the network packet (skb) as the input argument.

The maximum instruction limit per program is restricted to 4096 eBPF instructions, which, by design, means that any program will terminate quickly. For kernel newer than 5.1 this limit was lifted to 1 million eBPF instructions.

3.1.2 Verifier

eBPF works together with kernel, for this reason it has to provide the same safety guarantees which kernel provides. eBPF programs are verified through an in-kernel verifier in order to ensure that they cannot crash the kernel, always terminate not affecting kernel's stability.

Although the instruction set contains forward as well as backward jumps, the in-kernel BPF verifier will forbid loops so that termination is always guaranteed. This means that from an instruction set point of view, loops can be implemented, but the verifier will restrict that.

Furthermore, verifier ensure that every possible memory access by an eBPF program is safe. It verifies that inside the code there are the needed checks to avoid invalid memory accesses, preventing out of bound accesses or null pointer dereferencing.

3.1.3 Helpers

Helper functions are a concept which enables eBPF programs to consult a core kernel defined set of function calls in order to exchange data with kernel. Available helper functions may differ for each eBPF program type. Each helper function is implemented with a commonly shared function signature similar to system calls.

They allow to enrich eBPF programs with more complex operations that overcome verifier limitations. However, since they are statically defined in the kernel, they cannot be extended or modified.

3.1.4 Maps

Maps are efficient key/value stores that reside in kernel space. They can be accessed from an eBPF program in order to keep state among multiple eBPF program invocations. They can also be accessed through file descriptors from user space and can be arbitrarily shared with other eBPF programs or user space applications.

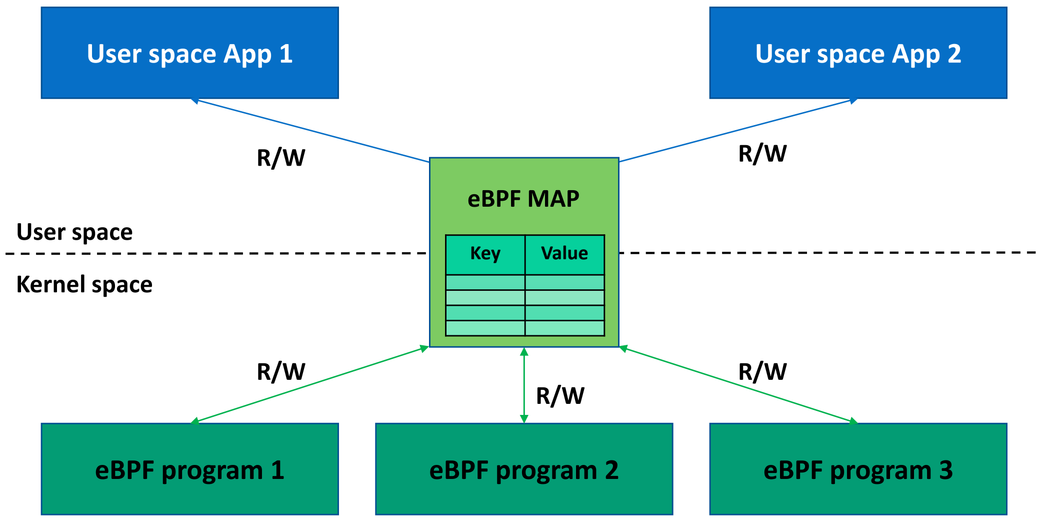


Figure 3.1: eBPF map accessed by multiple programs.

Map implementations are provided by the core kernel. There are generic maps that can read or write arbitrary data, but there are also a few non-generic maps that are used along with special helper functions. Generic maps all use the same common set of eBPF helper functions in order to perform lookup, update or delete operations while implementing a different backend with differing semantics and performance characteristics.

They can be divided in two categories:

- *per-CPU*: each map is replicated for each CPU, and each CPU reads or writes only from its specific instance
- *shared*: there is only a single instance of the map that is shared across all the program instances that refer to it

It is important to note that shared eBPF maps can be accessed concurrently not only from different programs, but also from user space programs and even from other instances of the same eBPF program running on different CPU cores. To avoid interference problems is possible to use spinlocks primitives provided by eBPF itself. At the same time per-CPU maps, does not have to handle synchronization at all as there is no concurrency, and this brings as a result an improvement in performance.

3.1.5 Object Pinning

eBPF maps and programs act as a kernel resource and can only be accessed through file descriptors, backed by anonymous inodes in the kernel. With this approach there are advantages and disadvantages: user space applications can make use of most file descriptor related APIs, but at the same time, file descriptors are limited to a processes' lifetime, which makes options like map sharing rather difficult to carry out.

Thus, this brings a number of complications for certain use cases such as iproute2, where TC or XDP sets up and loads the program into the kernel and terminates itself eventually. With that, also access to maps is unavailable from user space side, where it could otherwise be useful, for example, when maps are shared between ingress and egress locations of the data path. Also, third party applications may wish to monitor or update map contents during eBPF program runtime.

To overcome this limitation, a minimal kernel space BPF file system has been implemented, where BPF map and programs can be pinned to. This process is called *object pinning*. The BPF system call has therefore been extended with two commands which can pin (BPF_OBJ_PIN) or retrieve (BPF_OBJ_GET) a previously pinned object.

3.1.6 Program Types

eBPF supports many different kind of program types. As concerns networking, there are two main program types: XDP eBPF and TC eBPF.

eXpress Data Path (XDP)

XDP stands for eXpress Data Path and provides a framework for eBPF that enables high-performance programmable packet processing in the Linux kernel. It runs the eBPF program at the earliest possible point in software as soon as the network driver receives the packet. At this point the driver has only picked the packet, but it has not made yet any allocation for the skb structure or any other expensive operation.

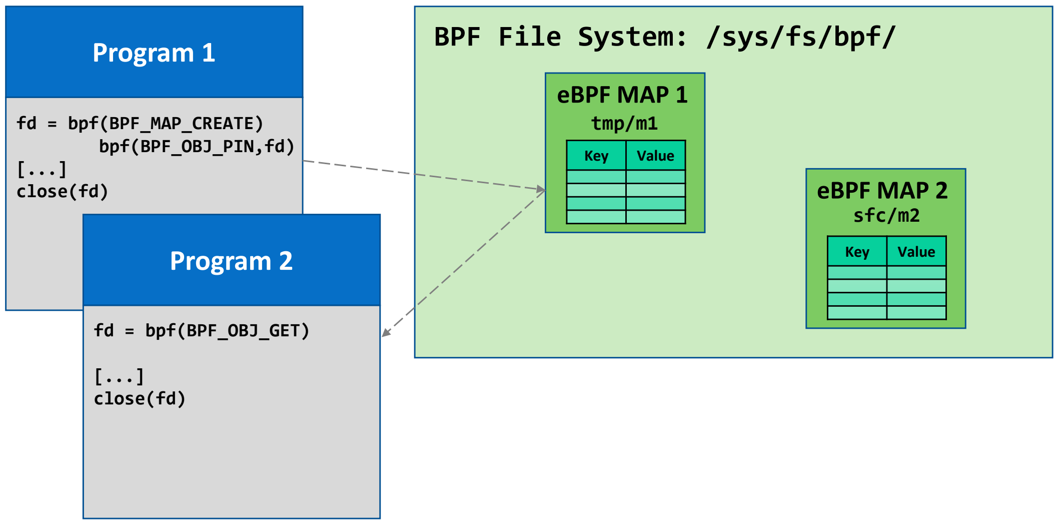


Figure 3.2: An example of object pinning on BPF file system.

XDP does not bypass the kernel, it works together with it and can benefit of all the advantages of kernel space. Since it includes support of eBPF, it benefits from all its programmability and safety features. eBPF allows to access packet data directly, and the framework contains XDP action codes which can be used by the eBPF code to instruct the driver how to proceed with the packet (i.e. DROP, REDIRECT, etc.)

It is important to point out that XDP supports only the *ingress* point of networking data path, so it can be triggered only on the incoming traffic on an interface.

Traffic Control (TC)

eBPF can be used also in the kernel Traffic Control (TC) layer in the networking data path. In TC the eBPF input context is a `sk_buff`, because when the networking stack receives a packet after the XDP layer, it wraps it in this structure with some additional metadata about the packet itself. This additional information is provided as context for the eBPF program which can benefit of a wider set of helpers, however it costs in term of performance since the `sk_buff` obtained after the parsing of the packet.

Differently from XDP, TC eBPF programs can be triggered out of *ingress* and also *egress* points in the networking data path. Furthermore, TC eBPF programs does not require driver changes since their hook points are in layers that are generic to all interfaces. Changes are only required when TC eBPF offloading is desired.

eBPF programs in the TC layer are run from the *cls_bpf* classifier, however this name can be misleading because programs here are fully capable of not only of reading packets but also of modifying them. They can also decide how the packet processing should continue returning an action code. Both the TC ingress and the egress hook, where *cls_bpf* itself can be attached to, are managed by a pseudo *queuing discipline* called *sch_clsact*.

A queuing discipline is a network scheduler. Scheduling is the mechanism by which packets are arranged between input and output of a particular queue [10]. The *sch_clsact* qdisc is not a real queuing discipline, it is only a convenient object where filter can be attached to, in order to process packets incoming and outgoing from an interface.

BPF Compiler Collection

BPF Compiler Collection (BCC) is a toolkit for creating efficient kernel tracing and manipulation programs. BCC makes eBPF programs easier to write, with kernel instrumentation in C, and front-ends in Python and LUA. It is suited for many tasks, including performance analysis and network traffic control [11].

On one hand it provides *macros* to simplify the writing of eBPF C code, on the other hand it supports the creation of user space programs in Python. In the latter case, it makes it possible to write short but expressive programs with all the high-level languages advantages of Python that are missing with C. For example, is possible to handle eBPF maps as normal *dictionaries*, making them easy to use. Underneath there are always calls to eBPF helpers, but they are hidden to the programmer that uses the library.

3.2 Netlink

Netlink is a socket family used to transfer information between kernel and user-space processes. It consists of a standard sockets-based interface for user space processes and an internal kernel API for kernel modules [12].

Using Netlink, an application can send/receive information to/from different kernel features, such as networking, to check their current status and to control them.

Rtnetlink is a specialized netlink family that allows to manage networking environment [13]. Most used network utilities such as *iproute2* are based on it. Functionalities that can be controlled with rtnetlink are network routes, IP addresses, link parameters, neighbour setups, queueing disciplines, traffic classes and packet classifiers.

3.2.1 Pyroute2

Pyroute2 is a pure Python *netlink* library [14], it consists of a common netlink messages coder/decoder and a number of protocol-specific modules. It may be used to work with existing protocols such as RTNL, as well as to create new netlink protocols.

Pyroute2 provides a low-level *IPRoute* module that can be used to manage Linux network configuration. The IPRoute class is a 1-to-1 *rtnetlink* mapping providing access to rtnetlink as is.

This enables development of Python programs that can obtain network configuration and alter it. For example, is possible to add a *queuing discipline* and a *filter* to an interface, or to change attributes of a *link*, or to modify the *routing table*.

3.2.2 netlink (go package)

The go netlink package provides a simple *netlink* library [15]. It can be used to add and remove *interfaces*, set ip *addresses* and *routes*, configure *ipsec* etc. The library attempts to provide an api that is loosely modeled on the CLI provided by *iproute2*.

As it is a project still under development, some netlink functions are not yet available in the high-level interface provided. Nonetheless, it is widely used in go-based applications that have to manipulate the network configuration, such as most of the CNI plugins for Kubernetes.

3.3 Multus CNI

Multus CNI is a container network interface (CNI) plugin for Kubernetes that enables attaching multiple network interfaces to pods [16]. Typically, in Kubernetes each pod only has one network interface, with Multus is possible to create a *multi-homed* pod that has multiple interfaces. This is accomplished by Multus acting as a "meta-plugin", a CNI plugin that can call multiple other CNI plugins.

In the Figure [3.3] there is a Pod with three interfaces. The eth0 is the default interface that connects to Kubernetes Cluster Network, which is used to communicate with API or with other Pods in the network. Net0 and Net1 are additional interfaces that connect the Pod to other networks using other CNI plugins.

To enable configuration of multiple network interfaces, Multus provides a Custom Resource Definition called *NetworkAttachmentDefinition*. This custom resource must be used to specify the network configuration object of the other CNI plugins for additional interfaces. The NetworkAttachmentDefinition can contain a whole JSON CNI configuration object or a name corresponding to the *name* field of a config file in the CNI configuration directory.

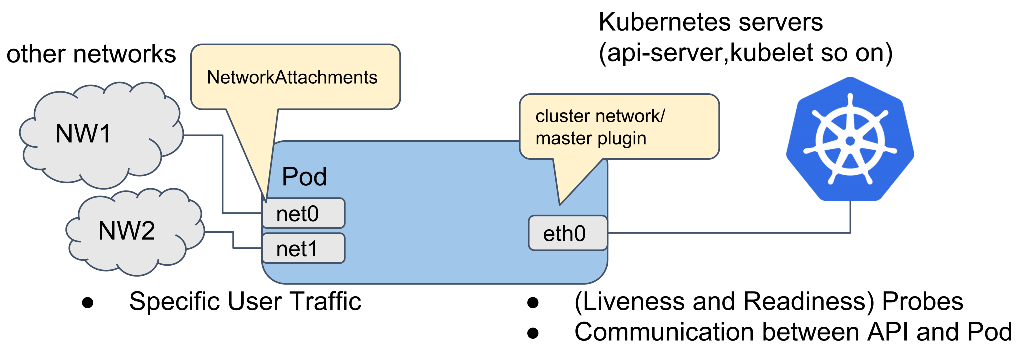


Figure 3.3: A pod with multiple attached interfaces [16].

When a NetworkAttachmentDefinition object is created and committed to the API it can be used to attach additional interfaces to Pods. To enable this behaviour is necessary to add a specific *annotation* to Pod metadata:

```
k8s.v1.cni.cncf.io/networks:  {NetAttachDefName}@{ifName}
```

In this way, when Pod will be scheduled and the Container Runtime will call the ADD command on the CNI plugin, Multus plugin will fetch Pod object from API server and search for the *networks* annotation: for each value of the annotation, it will *delegate* the add to the corresponding CNI plugin providing the configuration object specified in the NetworkAttachmentDefinition and the interface name as arguments.

In the Figure 3.4 is showed the typical workflow:

It is important to specify that the default network plugin is always executed, there is no need to specify it with additional annotations.

3.4 Kubernetes Operator Pythonic Framework (Kopf)

Kopf (Kubernetes Operator Pythonic Framework) is a framework and a library to simplify development of Kubernetes operators in Python code [17].

It provides toolkit to run the operator, to talk to the Kubernetes cluster, and to marshal the Kubernetes events into the pure Python functions of the Kopf-based operator, and the libraries to assist with a limited set of common tasks for manipulating the Kubernetes objects [18].

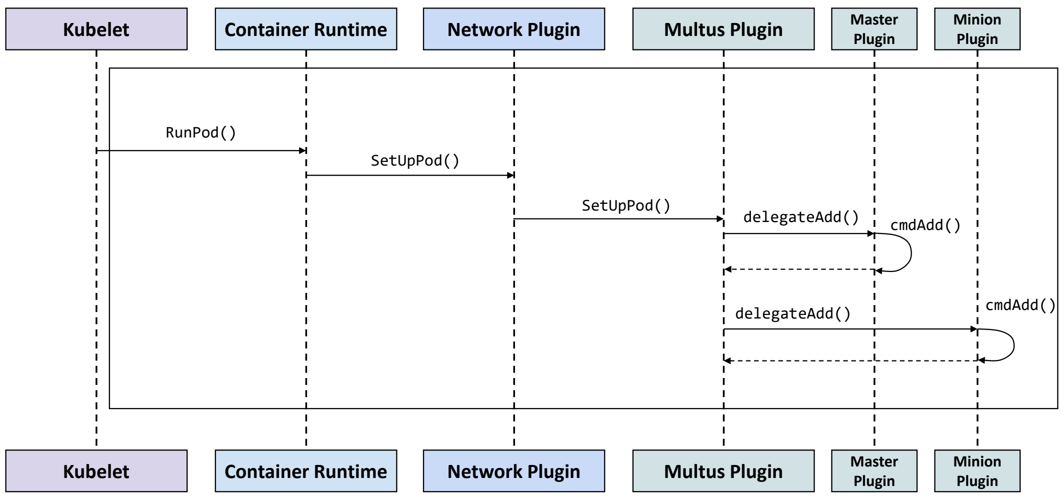


Figure 3.4: Multus network workflow in Kubernetes [16].

Any operator built with Kopf is based on *handlers*.

In the following sections key features of Kopf are described.

3.4.1 Handlers

Kopf operators differently from other operators built with other frameworks like *Operator SDK* and *Kubebuilder*, is not based by default on a *reconcile loop*. The idea behind reconciliation loop is that operator logic should reconsider the whole resource state and check if it matches the desired state, independently from the nature of the event that triggered the reconciliation itself. This approach is referred to as *level-based* as opposed to *edge-based* triggering which idea, instead, is to receive an event and react to a state variation.

Kopf embraces this second approach, providing a set of high-level handlers that react to *state changes*. Whenever an event happens for a watched object, Kopf framework detects what actually happened, in particular it can determine if the object was just created, if it was edited and which fields specifically were edited, if it was deleted or marked for deletion. When the cause is determined, it triggers the appropriate handler passing all the needed information.

However, Kubernetes only notifies *when* something is changed in a resource, but it does not specify *what* changed. In order to enable the state-changing handlers Kopf needs to keep track of the *state* of the handlers, to do that it stores needed information inside resources under special annotations or status fields. All this complexity is hidden, and this behaviour is completely transparent to the developer which has only to write the handlers logic for each desired state-changing handler.

Along with state-changing handlers Kopf provides two other special kinds:

- *Event-watching handlers*: they can be considered as low-level handlers for events that are received from K8s “as is”; they are the equivalent of the informers. With this kind of handler events can be intercepted and handled silently, without storing the handlers’ status on objects.
- *Resuming handlers*: they are executed only when an operator restarts and detects an object that existed before. With resuming handler, in addition to creation/update/deletion handlers no object will be left unattended even if it does not change over time.

To register a handler for an event, the `@kopf.on` decorator must be put on the handler function definition.
All available options are: `@kopf.on{event,create,update,delete,resume}`

Kopf provides a lot of *kwargs* for handlers functions, regarding configuration, logging, indexes, but most importantly resource related arguments. In order to obtain resource object related to the event in the handler, it is sufficient to declare an argument *body.* In an analogous way it is possible to declare other arguments such as *metadata*, *spec*, *status* and so on, which provide access to specific part of the resource.

3.4.2 Filters

Handlers can be restricted to be executed only when a resource match certain criteria. This is useful to avoid not necessary calls of the handlers on each event regarding the resource. Filters can be specified as special parameters inside the decorators for handlers functions.

Kopf provides various kinds of checks:

- Specific values: with Python literals values
- Presence of values: with special markers `kopf.PRESENT`/`kopf.ABSENT`.
- Per-value callbacks: with a callable object which returns true/false.
- Whole-body callbacks: with a callable object which returns true/false.

Those checks can be applied in multiple ways, even combined. Metadata is the most commonly filtered aspect of the resources, but also other specific fields in the spec or in the status can be checked. For update handlers, Kopf provide the possibility to check separately old and new values with different filters allowing an improved filtering precision.

3.4.3 Indices

Indexers automatically maintain in-memory overviews of resources, commonly named *indices*, grouped by keys that are usually calculated based on these resources.

Indices can be used for cross-resource awareness, sometimes when a resource is changed, it may be necessary to get all the information about all resources of another kind. Instead of requesting those resources to the Kubernetes API, which is an expensive operation, with indices is possible retrieve those information from a local store.

Kopf automatically handles all the operations needed to create and update indices to reflect the state of the resources in the cluster. The developer has only to specify a mapping function for the index that map a resource to a dictionary entry, optionally it can also accept a filter on the resources that should be considered in equivalent way to handlers. When a resource is created or starts matching the filters, it is processed by all relevant indexing functions, and the result is put into the indices. When a previously indexed resource is deleted or stops matching the filters, all associated values are removed.

To declare an index the `@kopf.index` decorator must be put on an indexing function. Indices are available to all handlers functions as *kwargs* named exactly same as the indices, and they can be used as traditional dictionaries.

3.4.4 Peering

Usually, one and only one operator instance should be deployed for a resource. However, for redundancy purposes many replicas may be deployed, but this can cause *collisions*.

To prevent collisions, Kopf by default assign a priority to each operator instance, and when an operator notices other operators with higher priority it pauses until those stop working. All running operators communicate with each other via *peering objects*, so they know about each other.

However, in some specific cases operators must be deployed as DaemonSet to allow per-node actions. In this scenario, multiple operators must be active and react to the same resources events. To enable this behaviour the *standalone mode* for Kopf operator is required. The standalone mode prevents an operator from peering and talking to other operators. Even if other operators with higher priority will start handling the same objects the operator will not stop.

When multiple operators react to same resource events, in particular when state-changing handlers are involved, names of the annotations or status fields used as handlers' state store must be customized with node-specific information. This is a fundamental point to avoid collision of different operators that handle the same object and store their handler's state on the object itself. Kopf allows to configure these aspects through settings that can be specified in the operator *startup* handler.

3.4.5 Eventual consistency

Kopf provides eventual consistency of handling. When arbitrary errors occur during handler execution it will retry later until it succeeds. The framework provides several configuration options that include custom limits for the number of attempts, specifying unrecoverable exception, backoff time between arbitrary errors and timeouts for handlers.

Chapter 4

Prototype Architecture

This chapter describes the general prototype architecture and provides an overview of each component. All the implementation details will be explained in the next chapter.

4.1 General Architecture

The system was conceived to be fully integrated with Kubernetes. For this reason, some of prototype components have been designed to leverage the extensibility features of the aforementioned platform. Further detail will be provided for each component.

The prototype architecture is composed of:

- *CNI plugin*: this is the component that configures the network interfaces of network functions involved in the chain.
- *Load Balancers*: these are the components that implement the cross-connections between network functions and that enable the split of the traffic among different replicas in the chain.
- *Operator*: this is the element that allows chain creation and enables the seamless adapting features of the system.

From the combined action of these elements, the system is able to create and manage the chains of network functions. The CNI plugin configures the desired interfaces on the network function and the Operator connect those interfaces to the load balancers according to the chain definition. The behaviour of these elements is configured by means of user-defined resources in the cluster.

In Figure 4.1 is depicted a possible scenario in a Kubernetes cluster, in which all the architectural elements are present.

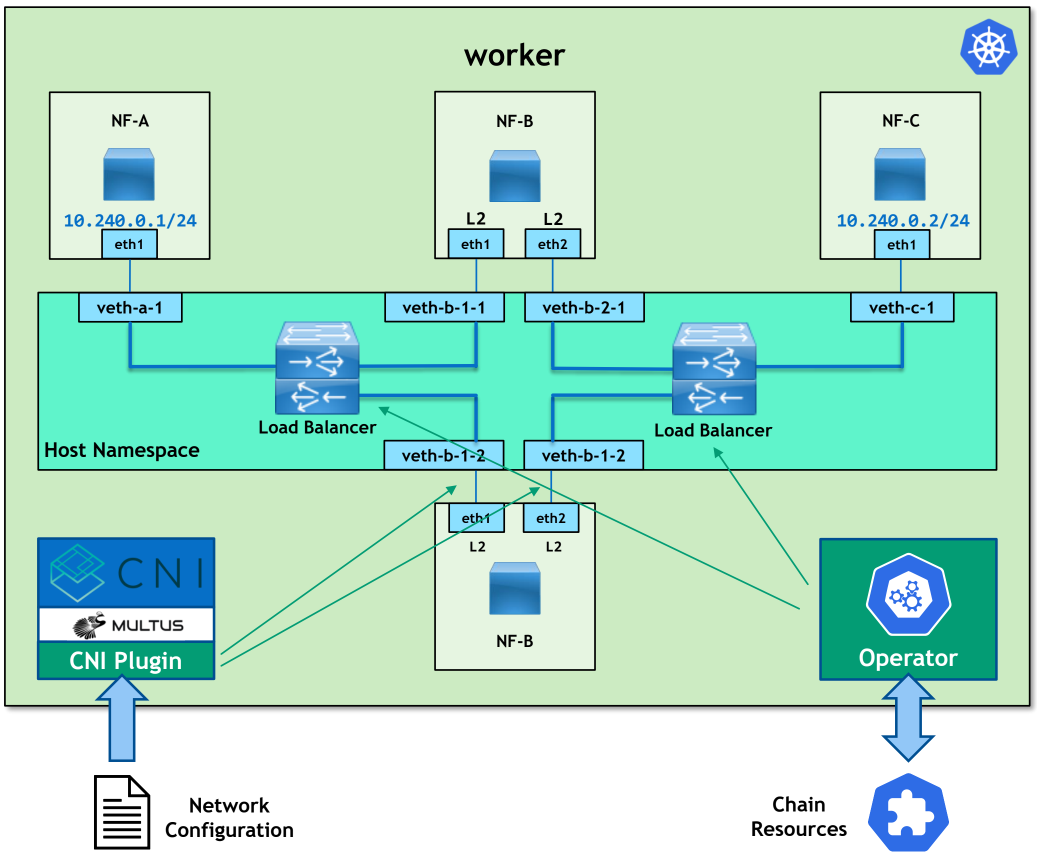


Figure 4.1: System components on a Kubernetes node.

4.2 CNI plugin

The CNI plugin is in charge of configuring all the necessary additional interfaces on pods. It is suitable for configuring only interfaces involved with the chains, for this reason it must be used in conjunction with the main plugin of the cluster network provider. Hence, Multus CNI is required, in order to be able to use the custom plugin together with the main plugin.

The plugin supports configurations for interfaces both at L2/L3 level. This means that it can add interfaces that have IP addresses, but at the same time it allows to simply create interfaces that does not have an IP and provide a simple L2 link. This last aspect is useful when transparent network functions that do not have an IP address are in the chain.

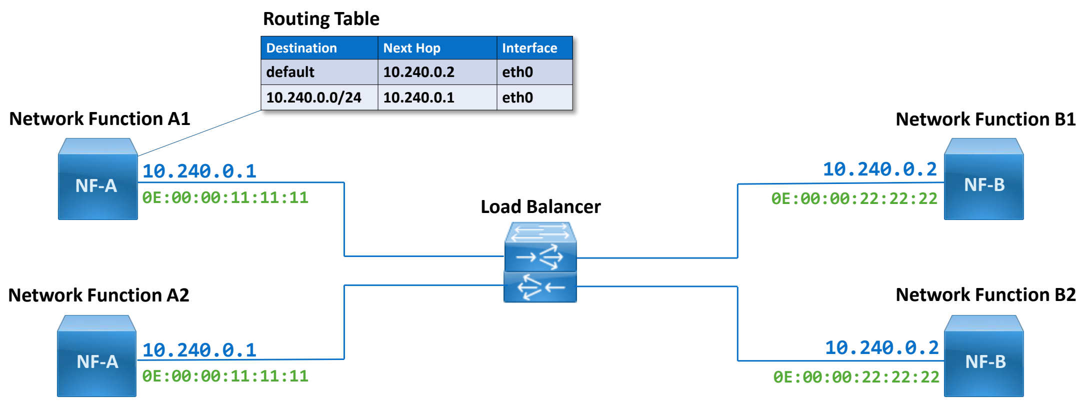
It is important to point out that this plugin does not provide itself cross-connection mechanism between NFs. All the interfaces configured by this plugin, provide a link between Pods and their host. How the traffic is then forwarded between chain elements is in charge of load balancers.

4.2.1 IP-MAC address management

As regards the management of the IP and MAC addresses assigned by the custom CNI plugin to pod interfaces, a very specific choice has been made in order to guarantee the transparency of the solution towards network functions and to keep the load balancing logic simple.

What is normally expected from CNIs is that they configure interfaces with unique addresses for each pod, so that all kind of communication in the cluster are enabled without conflicts. This is certainly the kind of approach needed to ensure basic cluster networking, but it may not be the best one for managing addressing in scaling chains of network functions. In other words, when there are multiple replicas of the same network function, it may be more convenient to assign each interface involved in the chain the same addresses.

The proposed solution is based on this alternative approach. As shown in Figure 4.2 each replica has same MAC and IP addresses on the interfaces linked to the load balancer.



Routing Table

Destination	Next Hop	Interface
default	10.240.0.2	eth0
10.240.0.0/24	10.240.0.1	eth0

Figure 4.2: Network function replicas with same addresses configured.

From *NF-A* replicas point of view the next hop for the default route is only one, in this way the presence of other replicas is hidden to network functions, which behave as if they were linked to a singular instance. On the contrary, if there had been several IP addresses on the interfaces, there would have been more entries in the routing table and as a consequence transparency on the number of replicas would have been lost. Moreover, in this in this hypothetical scenario there would be no need at all for an additional load balancing logic, but an alternative one would have to be configured directly on the network functions.

In addition to IP address, also MAC address is the same for each replica. The reason behind that comes from the need to be able to manage the ARP protocol and L2 addressing between network functions. Without the same MAC addresses there is a conflict between the load balancing logic and the L2 addressing. The load balancing logic works on the quintuple of the packets session and do not consider the L2 addresses, but at the same time these packets have a specific MAC address destination of a replica, which is obtained through an ARP request. In other words, each network function sends packets with the destination MAC address set to the value obtained through ARP, but then these packets are distributed by the load balancers among the various replicas regardless of their MAC address. This means that traffic could be forwarded to replicas that have a MAC address on the interface which is different from the destination of the packets.

As mentioned above, a possible way to reconcile ARP management, L2 addressing and load balancing logic is, to assign the same MAC addresses to all replicas. In this way it does not matter which replica answer to the ARP requests, the result will be always the same and as a consequence traffic can be split among replicas without the risk of packet drops caused by L2 addresses mismatch.

4.3 Load Balancers

The load balancer is the component that is in charge of implementing *cross-connections* among NFs in SFCs. They are completely *transparent* elements, which act without any interaction with network functions. They only provide *forwarding* and *load balancing* logic to forward packet among NFs replicas.

Load balancers are *two-sided* and *bidirectional*. They are deployed for each pair of NFs in a chain, and they deliver traffic on both directions, namely traffic going from *NF-A* to *NF-B* replicas and traffic going from *NF-B* to *NF-A* replicas. Internally they have information about interfaces attached on one side and on the other side and they only forward traffic between interfaces of opposite sides.

Since they are bidirectional there is no real distinction between the two sides, but to disambiguate sometimes one side it is called *frontend* and the other *backend*, even though there is no difference.

Load balancers work on *sessions*; they keep track of all the handled session flows and involved interfaces. The first time that a packet of an untracked session is received, an egress interface is selected and information about interfaces and session flow is registered. The following times the interfaces are selected retrieving recorded information related to the session. This is necessary because some connection-oriented protocols such as TCP can have problem if sessions' packets go on different paths.

In Figure 4.3 is shown the information available to the load balancers when a packet from a new session arrives:

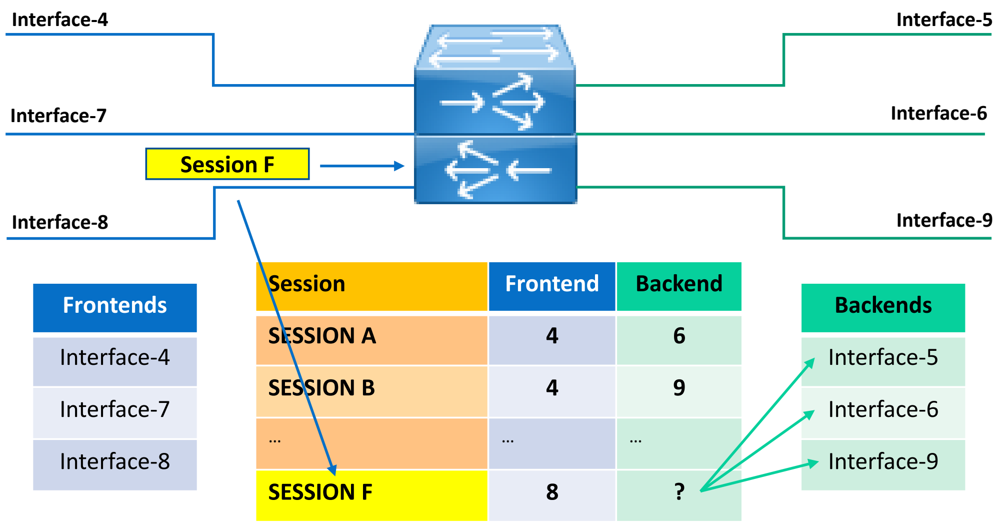


Figure 4.3: A load balancer receiving a packet for a new session.

However, Load balancers does not have the capability to understand which are the specific frontend and backend interfaces to handle, they require an external component that act as a *control plane* and injects all the needed information. This is one of the Operator's jobs.

4.4 Operator

The operator is a Kubernetes Operator; it is the component that contains all the logic needed to create chains and to adapt the system to the variation of NF replicas. This means that on one hand it has to transform logical description of the chain into real cross-connections between NFs while on the other hand it has to manage those cross-connections in order that they adapt to cluster changes.

Logical description of the chains is provided by means of a Kubernetes *Custom Resource*, the operator watches events related to those resources and reacts accordingly. It handles the full *lifecycle* of chains, from the creation to the deletion. This includes the creation and the handling of cross-connections between elements in the chain.

As concerns cross-connections, it acts as a control plane for load balancers making them consistent with the chain state. In order to do that, operator watches all the events from the cluster that involve NFs pod in handled chains. When it receives those events, it reacts to them provoking a *reconciliation* between the chain state and information injected in involved load balancers. If the event was the deletion of a Pod replica, this means that its interfaces must be removed from the load balancers it was connected to, on the contrary if the event was the creation of a new Pod this means that its interfaces must be added to the load balancers it will be connected to.

Chapter 5

Prototype implementation

In this chapter is presented the implementation of the architectural elements described in the previous sections. For each element are provided reasons behind implementation choices and detailed descriptions of main functions in the code.

Different programming languages have been used: Go for the CNI plugin, Python for the Operator and C for load balancers data plane. Furthermore, YAML and JSON were used as data-serialization languages for Kubernetes resources and CNI configuration files, respectively.

5.1 CNI plugin: sfc-ptp

The custom CNI plugin, called sfc-ptp, has been implemented according to the model provided by the reference plugins of CNI project. In particular it was inspired by the ptp plugin for the code structure, even though the custom CNI enables different kind of configurations. Moreover, it was designed to be used in a Kubernetes environment in conjunction with Multus CNI, so it is not suitable for more general contexts.

In addition to the creation of the *veth* pairs between Pod and host, this plugin stores some information related to the configured interfaces within the Pod resources in the cluster. This is essential for the operator to manage correctly cross-connections between chain elements.

5.1.1 Plugin Configuration

Like all plugins that follow the CNI specification, it accepts a configuration object written in JSON, which describes the network configuration desired. This configuration object is passed to the plugin at runtime. A part from required and well-known fields, the plugin defines additional fields necessary for the management of the chains by the prototype.

It is important to point out that the plugin configuration for sfc-ptp is intended to be for a single interface of a specific network function. Usually for other CNI there is only one configuration object for each node in the cluster, but since this plugin configures additional interfaces with specific parameters, it needs different configuration objects for each type of interface.

The most important configuration fields are listed below:

- `name` (`string`, `required`): the name of the network configured
- `type` (string, required): matches the name of the CNI plugin binary, must be sfc-ptp for this plugin
- `mtu` (`integer`, `optional`): explicitly set MTU to the specified value. Defaults to value chosen by the kernel.
- `ipam` (`dictionary`, `optional`): IPAM configuration to be used for this network. Although any type of IPAM plugin is supported, the use of the *static* IPAM plugin is required. If no configuration is specified, the interface configured will be L2 only.
- `macGenerationString` (`string`, `required`): a string that is used to generate a MAC address for the pod interface. Equal strings generate equal MAC addresses.
- `hostInterfacePrefix` (`string`, `optional`): the prefix of the network interface name that will be created in the host side. It can be useful in case of debugging. If not specified, a random name will be given to the interface.
- `dataDir` (`string`, `optional`): path to a directory where the plugin can store/retrieve information necessary for interface configuration. If not specified, a default location is chosen.
- `kubeconfig` (`string`, `optional`): path to a kubeconfig file that is used by the plugin to authenticate to the API server, in order to be able to read and update Pod resources. If not specified, the default path for the Multus kubeconfig file will be selected.

In the following listing is provided an example of configuration for the sfc-ptp plugin.

```
{
    "cniVersion": "0.3.1",
    "name": "net1",
    "type": "sfc-ptp",
    "ipam": {
        type": "static",
        "addresses": [
            {
                "address": "10.240.0.1/24",
                "gateway": "10.240.0.254"
            }
        ],
        "routes": [
            {"dst": "0.0.0.0/0", "gw": "10.240.0.254"}
        ]
    },
    "hostInterfacePrefix": "veth-client",
    "macGenerationString": "client",
    "dataDir": "/var/lib/cni/sfc-ptp"
}
```

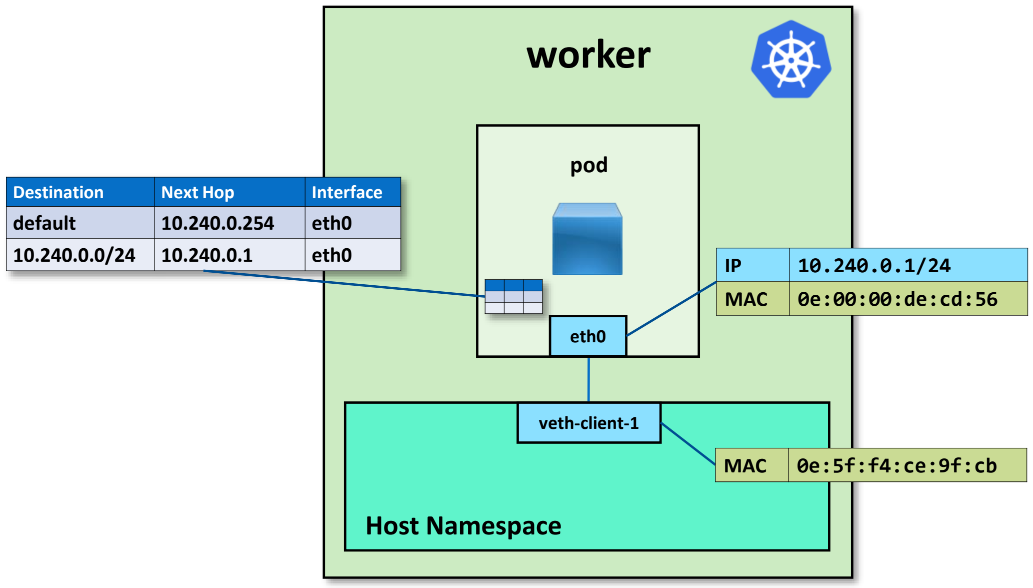
Listing 5.1: Configuration example for the sfc-ptp plugin

When the sfc-ptp plugin is executed providing the previous configuration file and the additional required arguments for the target pod, it will configure a veth pair between the pod and the host. In this specific case, through the static IPAM plugin will be configured an IP address and a default route in the pod. Furthermore, the value of `hostInterfacePrefix` specified will be used to generate a name for the end of the veth pair that resides in the host. To do this, the CNI plugin simply add an increasing index to the prefix making each name unique. For each different prefix, the CNI plugin keeps track of the last index used in specific files under the specified `dataDir` path. In Figure 5.1 is shown the result in a scenario where the plugin was invoked in a node of a Kubernetes cluster to configure a Pod network.

It is important to note that the sfc-ptp plugin does not enable any kind of communication between pods or between pods and host network. From the host point of view there is visibility only on its veth end, but there is no additional information about the address configuration of the other end and consequently there are no routes configured to reach those addresses.

For this reason, the sfc-ptp plugin can be only used in conjunction with another CNI plugin that satisfy the networking requirements of Kubernetes, thus allowing the pod to communicate across the cluster network. In order to do that Multus CNI is used, so that the main plugin is always executed for each Pod in the cluster, while the sfc-ptp is only executed when it is required by network functions pods who need additional interfaces.

In this way traditional traffic can be handled by cluster networking, while traffic among network functions in the chains, namely, traffic that passes through sfc-ptp interfaces can be handled differently. In particular, in the proposed solution, SFCs traffic is forwarded between network function pods through load balancers. Without the load balancers forwarding logic there would be no forwarding at all



Destination	Next Hop	Interface
default	10.240.0.254	eth0
10.240.0.0/24	10.240.0.1	eth0

IP	10.240.0.1/24
MAC	0e:00:00:de:cd:56

MAC	0e:5f:f4:ce:9f:cb

Figure 5.1: Pod connected with host through a veth pair created by *sfc-ptp*.

and consequently there would be no connectivity in the chain.

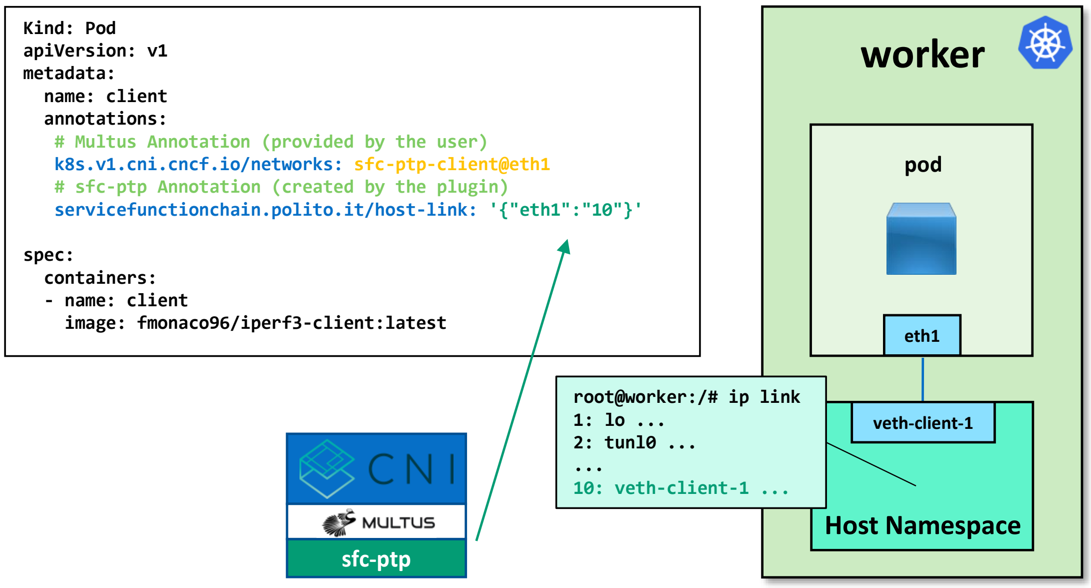
Summing up, the sfc-ptp plugin only creates a veth pair between a Pod and its host, and configures some specific aspects on the interfaces, but it does not enable or configure any forwarding mechanism in the host. Without additional logic all the traffic is dropped by the host immediately upon receiving it from its veths ends.

5.1.2 Host-link Pod annotation

As mentioned before, the sfc-ptp plugin saves some information about the configured interfaces inside the Pod resources. In particular, it is stored in a specific annotation created on purpose: `servicefunctionchain.polito.it/host-link`

The value of this annotation is intended to be a JSON string, whose properties are the *names* of the interfaces in the pod, while the values are the *indexes* of the corresponding interfaces on the host side. Therefore, through this annotation it is possible to retrieve, for each pod interface added through the sfc-ptp plugin, the index that identifies the other end of the link inside the host. The Figure 5.2 shows in detail how the annotation is created.

The reason this annotation was introduced is due to the fact that when the operator observes an event concerning a network function pod, it must have details of the involved interfaces, both on the pod side and on the host side, to proceed with his work. Without this annotation, the operator would have no way of



```
Kind: Pod
apiVersion: v1
metadata:
  name: client
  annotations:
   # Multus Annotation (provided by the user)
   k8s.v1.cni.cncf.io/networks: sfc-ptp-client@eth1
   # sfc-ptp Annotation (created by the plugin)
   servicefunctionchain.polito.it/host-link: '{"eth1":"10"}'

spec:
  containers:
  - name: client
    image: fmonaco96/iperf3-client:latest
```

```
root@worker:/# ip link
1: lo ...
2: tunl0 ...
...
10: veth-client-1 ...
```

Figure 5.2: Host-link annotation configured in a Pod resource.

obtaining this information, because within the Pod resource there is no reference to its network namespace. On the contrary, the CNI plugin has all this information available during its execution. Therefore, by adding the necessary details to the resource it can make them easily available to the operator.

It is important to point out that in order to do that, the sfc-ptp plugin needs a kubeconfig file configured with a *subject* that has the right *permissions* to read and write Pod resources.

5.1.3 Host interface name

As anticipated in the description of the configuration file for sfc-ptp plugin, there is an optional field named `hostInterfacePrefix` in which a prefix name for the host interface can be specified. The reason this field was introduced is that it can be used to simplify debugging in development. What CNIs typically do is configure links between pods and host, giving arbitrary names to the host ends. In figure 5.3 is depicted a common scenario.

From the host network point of view, it is not easy to understand which pod each interface corresponds to, even with the additional information that can be obtained by inspecting those interfaces. Therefore, the `hostInterfacePrefix` field was introduced among configuration options: by specifying a prefix it is possible to have recognizable interfaces. The reason there is a prefix name and not simply a name

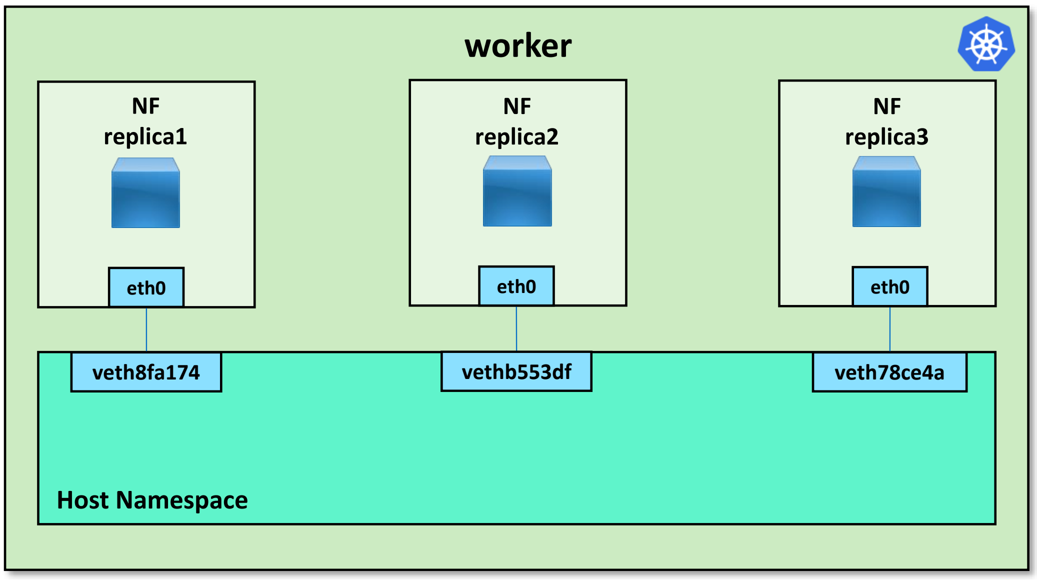


Figure 5.3: Veth interfaces with arbitrary names on the host side.

is due to the fact that there can be multiple replicas of the same network function pod that are configured executing the sfc-ptp with the same configuration object. To have a unique name an index must be added to the prefix. This behaviour is shown in the Figure 5.4, where as prefix is used *"veth-nf"*.

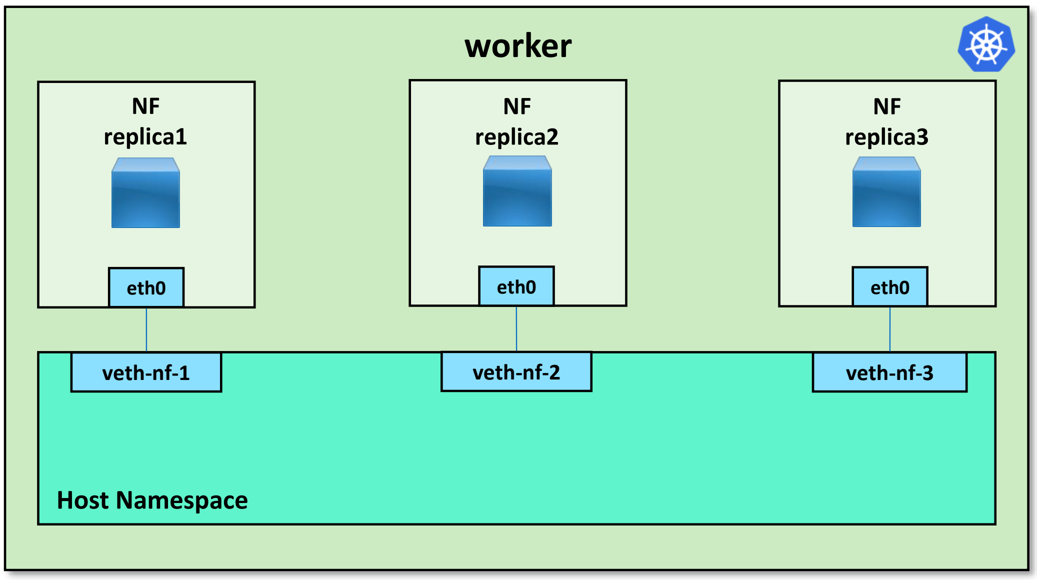


Figure 5.4: Veth interfaces with *"veth-nf"* set as `hostInterfacePrefix`.

5.1.4 Implementation Code

The sfc-ptp plugin is entirely written in Go. Golang is currently widely used to write code for applications in cloud environment. Therefore, CNI plugins, like most Kubernetes related software, are commonly written using this language. However, the CNI specification is language agnostic, so basically a CNI plugin can be written using any language that generates an executable.

The choice of Go was also motivated by the fact that there are libraries, provided by the developers of the CNI project themselves, which facilitate the writing of new CNI plugins. These, in turn, exploit the *netlink* package for interface management.

The plugin implements the ADD and DEL operations but not CHECK as it is not yet exploited in Kubernetes. Hereinafter, the most important pieces of code for ADD and DEL operations are shown. For the sake of simplicity, code about error handling and creation of result object is not included.

ADD Operation

In the ADD operation the CNI plugin receives a JSON configuration object and some environments parameters as arguments. Its job is to create the desired interface inside a pod, and to output a JSON-serialized result on the standard output. Moreover, it has to store some information of the configured interface inside the annotations of the Pod resource, in order to be available to the operator which has to work on these interfaces.

In Listing 5.2 is shown the main code of the ADD. First of all, the *name* for the host interface and the MAC address of the pod interface are created. Then, code is divided between L3 or L2 only configuration for the link. In the L3 case it is also executed the ADD operation on the IPAM plugin, in order to obtain the IP configuration to apply to the pod interface. In the end, there is a call to the function which updates the Pod annotation with information related to the interface configured.

```go
func cmdAdd(args *skel.CmdArgs) error {
    // Parse configuration from stdin
    conf, err := parseConfig(args.StdinData)
    // Create host veth name from the prefix (if specified)
    hostVethName, err := createHostVethName(conf.HostInterfacePrefix,
     conf.DataDir)
    // Create pod veth address from the given string
    podVethMac := createPodVethMac(conf.MacGenerationString)
    var result *current.Result
    // Check if IPAM configuration is required
    isLayer3 := conf.IPAM.Type != ""
    if isLayer3 {
        // Run the IPAM plugin and get back the config to apply
        r, err := ipam.ExecAdd(conf.IPAM.Type, args.StdinData)
        // Convert the IPAM result into the current Result type
        result, err = current.NewResultFromResult(r)
        // Get pod network namespace
        netns, err := ns.GetNS(args.Netns)
        // Setup L3 veth
        _, _, err = setupL3Veth(netns, args.IfName, hostVethName,
    conf.HostInterfacePrefix, conf.MTU, podVethMac, result)
    } else {
        // Layer 2 only
        // Get pod network namespace from args
        netns, err := ns.GetNS(args.Netns)
        // Setup L2 veth
        hostInterface, podInterface, err := setupL2Veth(netns, args.
    IfName, hostVethName, conf.HostInterfacePrefix, conf.MTU,
    podVethMac)
    }
    // Set Pod Network Interface
    err = setPodNetworkInterface(conf.Kubeconfig, args.IfName,
    hostInterface.Name, k8sArgs)
    ...
    // Return result
    return types.PrintResult(result, conf.CNIVersion)
}
```

Listing 5.2: ADD operation code.

The interface name is built from the `hostInterfacePrefix` if provided, while the pod veth MAC is always obtained from the given string. In the first case the string is used as a base for the name, in the second case the string is hashed in order to obtain random digits which are then properly combined to get a MAC address. Listing 5.3 shows how it is created an unique name from the prefix, while Listing 5.4 shows how the MAC is generated.

```go
func createHostVethName(hostInterfacePrefix string, dataDir string) (
    string, error) {
    ...
    // Get path for the handled interface prefix
    // and create all necessary folders
    dir := filepath.Join(dataDir, hostInterfacePrefix)
    if err := os.MkdirAll(dir, 0755); err != nil {
        return "", err
    }
    // Create file lock
    lk, err := disk.NewFileLock(dir)
    ...
    // Acquire lock on file lock to avoid interference
    // between simultaneous plugin calls
    lk.Lock()
    defer lk.Unlock()
    // Read last index for this host interface name from file
    var index int
    indexFilePath := filepath.Join(dir, "index")
    data, err := os.ReadFile(indexFilePath)
    if err != nil {
        ...
    } else {
        // File exist, get last used index
        index, err = strconv.Atoi(string(data))
        ...
    }
    // Open file to update last used index
    indexFile, err := os.OpenFile(indexFilePath, os.O_WRONLY|os.
   O_CREATE|os.O_TRUNC, 0644)
    ...
    // Write new last used index
    index += 1
    _, err = indexFile.WriteString(strconv.Itoa(index))
    ...
    // Return host veth name (hostInterfacePrefix-{index})
    return hostInterfacePrefix + "-" + strconv.Itoa(index), nil
}
```

Listing 5.3: Host interface name creation.

A progressive index is read from a file, then it is incremented and added to the interface prefix. Finally, the updated index is stored in the same file. It is important to highlight that file is accessed in mutual exclusion, in order to avoid race conditions between multiple execution of the plugin. The file path is determined by combining the `dataDir` path and the prefix.

```
...
// MAC prefix that will be combined with hashed value
var macPrefix string = "0E:00:00"
...
func createPodVethMac(macGenerationString string) string {
    // Hash macGenerationString to make a MAC suffix
    hash := hashString(macGenerationString)
    return fmt.Sprintf("%s:%s:%s:%s", macPrefix, hash[:2], hash[2:4],
     hash[4:6])
}
```

Listing 5.4: Pod interface MAC creation.

The MAC address obtained with this procedure will be the same for equal prefixes. Hence, replicas with the same plugin configuration will have the same MAC address, as it is required by the solution.

The following listing show the specialized function for configuring the veths for the

L3 case. The setup is done starting from parameters of the configuration object and from those derived in the previous functions. The function is based on the library function `ip.SetupVethWithName()` which creates both veth devices and moves the host-side veth into the provided host namespace. The code is executed in the context of the pod namespace. The function for the L2 case is similar with the only difference that for the L3 case there is also the configuration of IP addresses on the interface. This is also done through a library function `ipam.ConfigureIface()`.

```go
func setupL3Veth(netns ns.NetNS, ifName string, hostIfname string,
    hostInterfacePrefix string, mtu int, mac string, pr *current.
    Result) (*current.Interface, *current.Interface, error) {
    ...
    err := netns.Do(func(hostNS ns.NetNS) error {
        hostVeth, contVeth0, err := ip.SetupVethWithName(ifName,
    hostIfname, mtu, mac, hostNS)
        ...
        // Configure pod veth with given IPAM configuration
        if err = ipam.ConfigureIface(ifName, pr); err != nil {
            return err
        }
        return nil
    })
    ...
}
```

Listing 5.5: L3 veth pair creation.

Finally, in the last listing is shown how the host interface index corresponding to the configured interface in the Pod is stored in the annotations of the Pod resource. The code operates as follow:

1. The index of the host interface is obtained from its attributes
2. A Kubernetes client is instantiated to communicate with the API server
3. The resource of the current Pod is requested to the API server
4. A map that will contain the information about interfaces is created
5. If the resource has already the annotation set the value is unmarshalled in the map
6. The current pod interface is added to the map with the corresponding host index
7. The map is marshalled into the annotation
8. The updated Pod is sent with an update request to the API server

9. If the request fails, the steps starting from the third onwards are repeated

```
func setPodNetworkInterface(kubeconfigPath string, ifName string, hostIfname string, k8sArgs *K8sArgs) error {
    // Get host link index
    hostLink, err := netlink.LinkByName(hostIfname)
    index := hostLink.Attrs().Index
    // Create Kubernetes client to interact with the API server
    restClientConfig, err := clientcmd.BuildConfigFromFlags("", kubeconfigPath)
    clientset, err := kubernetes.NewForConfig(restClientConfig)
    coreClient := clientset.CoreV1()
    // Get details about the current Pod from arguments
    name := string(k8sArgs.K8S_POD_NAME)
    namespace := string(k8sArgs.K8S_POD_NAMESPACE)

    // Set the host-link annotation
    resultErr := retry.RetryOnConflict(retry.DefaultRetry, func() error {
        // Get Pod object from the API server
        pod, err := clientset.CoreV1().Pods(namespace).Get(context.TODO(), name, metav1.GetOptions{})
        // Create link map [podIfname] -> hostIfindex
        linkMap := map[string]string{}
        // Check if pod has host-link annotation already
        _, ok := pod.Annotations[linkAnnotation]
        if ok {
            err = json.Unmarshal([]byte(pod.Annotations[linkAnnotation]), &linkMap)
        }
        // Add link to map
        linkMap[ifName] = strconv.Itoa(index)
        // Marshal map to JSON
        jsonData, err := json.Marshal(linkMap)
        // Try to add annotation
        pod.Annotations[linkAnnotation] = string(jsonData)
        _, err = coreClient.Pods(namespace).UpdateStatus(context.TODO(), pod, metav1.UpdateOptions{})
        return err
    })
    ...
    return nil
}
```

Listing 5.6: Set up of the host-link annotation in the Pod.

DEL Operation

As in the ADD operation, DEL receives a JSON configuration object and some environments parameters as arguments. Its job is to clean-up the interfaces created with a previous ADD. In the case of L3 configuration it is executed also the DEL operation of the IPAM plugin. For the link deletion is used the library function `ip.DelLinkByNameAddr()`.

```go
func cmdDel(args *skel.CmdArgs) error {
    // Parse configuration from stdin
    conf, err := parseConfig(args.StdinData)
    ...
    // Check if IPAM clean-up is required
    isLayer3 := conf.IPAM.Type != ""
    if isLayer3 {
        if err := ipam.ExecDel(conf.IPAM.Type, args.StdinData); err
    != nil {
            log.Printf("ipam.ExecDel returned error %v", err)
            return err
        }
    }
    ...
    // Delete can be called multiple times
    // if the device is already removed it is not an error
    err = ns.WithNetNSPath(args.Netns, func(_ ns.NetNS) error {
        var err error
        _, err = ip.DelLinkByNameAddr(args.IfName)
        if err != nil && err == ip.ErrLinkNotFound {
            return nil
        }
        return err
    })
    ...
    return nil
}
```

Listing 5.7: DEL operation code.

Since the DEL operation, unlike the ADD, can be called multiple times for the same configuration, it may happen that there is nothing left to clean-up after the first run. For this reason, it is not considered an error when the interface is not found.

5.1.5 Usage with Multus

As stated previously, the sfc-ptp is designed to be used with Multus. In this section is presented an example configuration of a Kubernetes *Deployment* in which Pods have two additional interfaces. In particular each pod is an instance of an iptables firewall, which uses the additional interfaces as ingress and egress respectively.

First of all, the plugin configuration for the two interfaces must be created, and the objects must be included in a *NetworkAttachmentDefinition* as required by Multus.

Here is shown the configuration object for one of the two interfaces in a NetworkAttachmentDefinition resource.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sfc-ptp-firewall-1
spec:
  config: '{
      "cniVersion": "0.3.1",
      "name": "sfc-ptp",
      "type": "sfc-ptp",
      "hostInterfacePrefix": "veth-fw-1",
      "macGenerationString": "firewall1"
  }'
```

Listing 5.8: NetworkAttachmentDefinition for a L2 firewall interface

In this case the interface of the firewall is pure L2, so there is no IPAM configuration specified. The *NetworkAttachmentDefinition* of the second interface is equivalent, only the `hostInterfacePrefix` and the `macGenerationString` in the configuration change in `veth-fw-2` and `firewall2` respectively.

Once these resources are defined, they can be referenced in pods definition. In Listing 5.9 is shown how *NetworkAttachmentDefinitions* are used. They are values of a special annotation required by Multus that is `k8s.v1.cni.cncf.io/networks`.

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: firewall-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      servicefunctionchains.polito.it/networkFunction: firewall
  template:
    metadata:
      annotations:
        k8s.v1.cni.cncf.io/networks:
          sfc-ptp-firewall-1@eth1, sfc-ptp-firewall-2@eth2
      labels:
        servicefunctionchains.polito.it/networkFunction: firewall
    spec:
```

```
    containers:
    - name: firewall
      image: fmonaco96/iptables-firewall:latest
      securityContext:
        privileged: true
```

Listing 5.9: YAML Manifest of a firewall deployment.

When the two pods of this deployments will be scheduled, and Multus CNI will be called, a default interface will be added using the main plugin and two other interfaces will be added with the sfc-ptp as specified in the *NetworkAttachment-Definitions* referenced.

In Figure 5.5 is depicted the final result of this configuration. For the sake of clarity, the interface handled by main plugin is not shown.

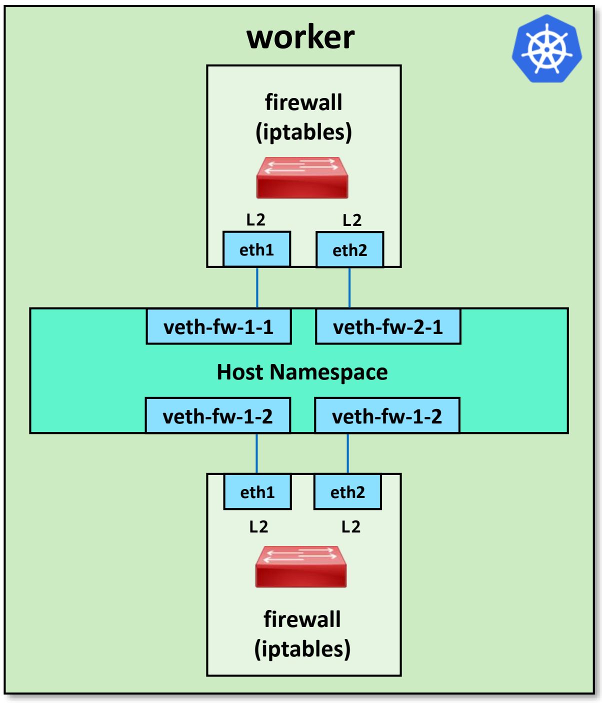


Figure 5.5: Firewall deployment with two replicas configured.

At the same time, the `host-link` annotations in the Pod resources will be appropriately set to reflect the respective configured interfaces.

5.2 Load Balancers

The Load Balancers data plane implementation is based on eBPF technology. The code, written in C, uses the macros provided by BCC for the declaration and access to the eBPF maps. The control plane, instead, is included in the operator logic and will be presented in the next section. Therefore, how the interfaces maps are filled is not covered here but only how they are used.

The eBPF program is injected directly into the TC layer of the host interfaces involved in the chain, in particular on the ingress path. Therefore, each interface has its own program, but at the same time, related data structures are shared among these. In this way, even if there are several instances of the load balancer program there is a single common state. For the sake of clarity, in all the figures the load balancer is represented as a single object linked to replicas' interfaces but actually, as mentioned before, each interface has its own program instance while the state is shared among them.

In Figure 5.6 is represented a chain of network functions connected through load balancers. The code is injected into the interfaces from the host namespace side.

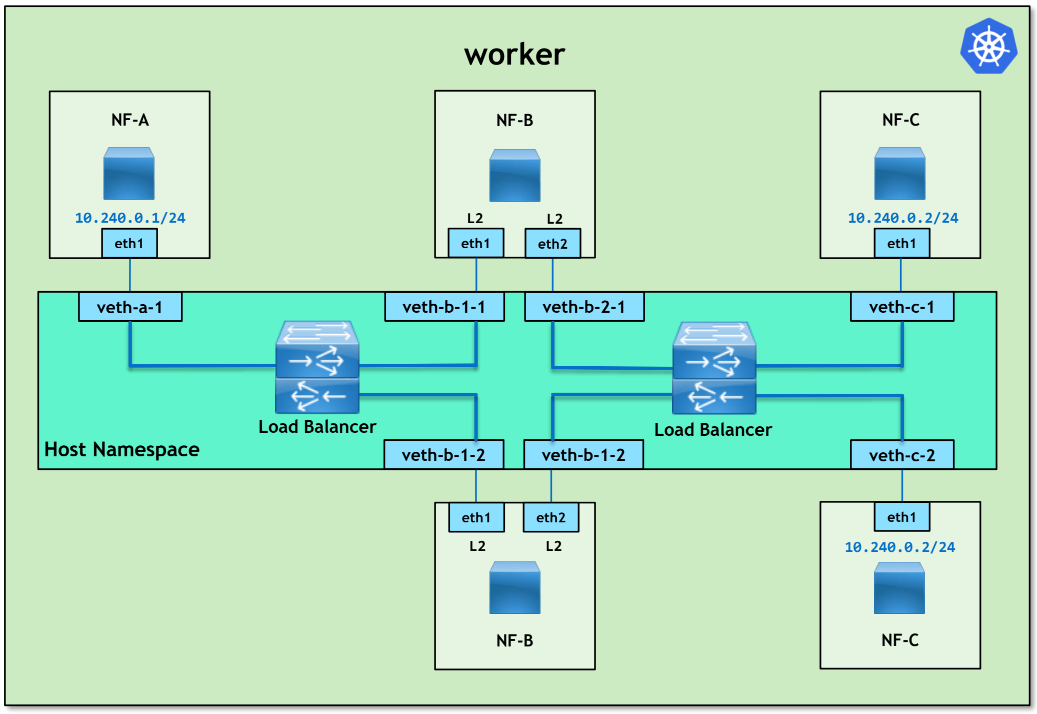


Figure 5.6: Load balancers between network functions in a chain.

A packet arriving from pod interfaces is intercepted as soon as it arrives at the host interface ingress. Once processed, the packet is redirected directly to the egress of the host interface selected by the load balancing logic, and from here it will follow the link to the pod.

5.2.1 Data structures

The load balancer data structures are six eBPF maps:

- *sessions*: `BPF_MAP_TYPE_LRU_HASH` that act as a session table, here is stored for each session the information about interfaces of the involved network function replicas, in order that traffic for a session is always forwarded to the same replicas.
- *frontends_interfaces*: `BPF_MAP_TYPE_ARRAY` in which are stored the indexes of the frontend interfaces, namely, indexes of replicas interfaces on the *frontend* side of the load balancer.
- *num_frontends*: `BPF_MAP_TYPE_ARRAY` composed by single element that store the number of frontends interfaces in the *frontends_interfaces* map.
- *backends_interfaces*: `BPF_MAP_TYPE_ARRAY` in which are stored the indexes of the backend interfaces, namely, indexes of replicas interfaces on the *backend* side of the load balancer.
- *num_backends*: `BPF_MAP_TYPE_ARRAY` composed by single element that store the number of backends interfaces in the *backends_interfaces* map
- *arp_cache*: `BPF_MAP_TYPE_LRU_HASH` that is used as a cache for the ARP replies, in order to simplify the ARP protocol management across the load balancer. Since ARP packets do not have a session, it is not possible to use the usual load balancer logic. With this cache, only the first time that a replica performs an ARP request the packet is forwarded in order to get a reply. Once the reply has been received it is stored in this cache and load balancer will answer directly to following requests.

In the following Listing is shown how those maps are declared, it is important to highlight that they are all *pinned* maps.

```
BPF_TABLE_PINNED("lru_hash", struct session, struct
    session_interfaces, sessions, MAX_SESSIONS, "/sys/fs/bpf/nsc/
    __LB_NAME__/session_map");

BPF_TABLE_PINNED("array", __u32 , __u32, num_backends, 1, "/sys/fs/
    bpf/nsc/__LB_NAME__/num_backends_map");
```

```

BPF_TABLE_PINNED("array", __u32 , __u32, backends_interfaces,
    MAX_INTERFACES, "/sys/fs/bpf/nsc/__LB_NAME__/
    backends_interfaces_map");

BPF_TABLE_PINNED("array", __u32 , __u32, num_frontends, 1, "/sys/fs/
    bpf/nsc/__LB_NAME__/num_frontends_map");

BPF_TABLE_PINNED("array", __u32 , __u32, frontends_interfaces,
    MAX_INTERFACES, "/sys/fs/bpf/nsc/__LB_NAME__/
    frontends_interfaces_map");

BPF_TABLE_PINNED("lru_hash", __u32, struct arp_reply_frame, arp_cache
    , MAX_ARP_CACHE, "/sys/fs/bpf/nsc/__LB_NAME__/arp_cache_map");
```

Listing 5.10: Load Balancer eBPF maps declaration.

The path of the BPF file system, where the maps are pinned, is specified as a template in the source code. The string `__LB_NAME__` is replaced at runtime by the operator with the real name of the load balancer, before compiling the code. Each load balancer has its own set of maps pinned to an exclusive path in the file system. The use of pinned maps allows the operator to never lose the reference to the state, this is particularly useful when it has to restore after an unexpected crash.

In Listing 5.11 are shown the definitions of *structs* that are used as keys and values in the *sessions* map and in the *arp_cache map*

```
// Session quintuple
struct session {
  __be32 ip_src;    // Source IP
  __be32 ip_dst;    // Destination IP
  __be16 port_src;  // Source Port
  __be16 port_dst;  // Destination Port
  __u8 proto;       // Protocol Type
} __attribute__((packed));

// Indexes of interfaces
// involved with a session
struct session_interfaces {
    __u32 frontend;
    __u32 backend;
} __attribute__((packed));

// Arp reply buffer struct
struct arp_reply_frame {
    unsigned char bytes[ARP_FRAME_LEN];
} __attribute__((packed));
```

Listing 5.11: Structs definitions

The `struct session` is used as key in the `sessions` map, while value is a `struct session_interfaces`. It is essential to underline that fields of the `struct session_interfaces`, that are `frontend` and `backend`, must not be considered strictly as `ingress` and `egress` interface, respectively. In other words, a fronted can be both an ingress and an egress interface and the same is true for a backend. That depends on the direction of the traffic: if the packet of a known session is received from a `frontend` interface the load balancer will use the stored `backend` interface associated to the session as `egress`. Similarly, in the opposite direction when a packet is received from a `backend` interface, the stored `frontend` interface will be considered as the `egress`.

The `struct arp_reply_frame` is used as value in the `arp_cache` map, it stores the entire frame for ARP replies. Since the target IP is used as the key in the map, there is no additional struct but a simple integer on 32 bits is sufficient.

5.2.2 Data plane implementation

The data plane is composed by two eBPF programs, one for the frontend interfaces and the other for the backend interfaces. The two programs are absolutely equivalent with the only exception that the one on the backend side has an additional action that simply swaps the session parameters so that the quintuple considered is equivalent in both directions.

Since the program is executed in the TC hook of the networking stack, the context in the eBPF program will be the struct `__sk_buff`. This struct contains the pointer to the packet buffer and other useful metadata such as the index of the interface from which the packet was received.

In Listing 5.12 are shown the main lines of code that implements the load balancing logic for packets arriving from a frontend interface. For the sake of brevity, all mandatory checks on packet size and pointer validity are omitted. First of all, pointers relative to the memory of the packet and the index of the interface from which packet was received, are retrieved from the context `__sk_buff`. After that, the packet is parsed in order to get session parameters which are stored in a struct. At this point, is possible to start the load balancing operations.

```
int handle_loadbalance_fe(struct __sk_buff *ctx) {
    // Retrieve pointers to the begin and end of the packet buffer
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    // Get ingress interface index
    __u32 ingress_ifindex = ctx->ifindex;
    ...
    // Session parameters
    struct session session_key = {};

    int success = get_packet_session(data,data_end,&session_key);
    ...
    // Load balancing
    // Check if session is present in session table
    struct session_interfaces new_interfaces = {};
    struct session_interfaces *interfaces = sessions.lookup(&
session_key);

    if(!interfaces){
        // Session is not present in the table
        // Calculate index of backend interface

        // Get num of backend interfaces
        unsigned int zero = 0;
        __u32 *n_backends = num_backends.lookup(&zero);
        ...
        // Calculate backend index
        __u32 hash = jhash((const void *)&session_key, sizeof(struct
session), JHASH_INITVAL);
        __u32 backend_index = hash % (*n_backends);
        // Get egress interface index
        __u32 *egress_ifindex = backends_interfaces.lookup(&
backend_index);
        ...
        // Create a new entry
        new_interfaces.frontend = ingress_ifindex;
        new_interfaces.backend = *egress_ifindex;
        interfaces = sessions.lookup_or_try_init(&session_key,&
new_interfaces);
        ...
    }
    return bpf_redirect(interfaces->backend,0);
}
```

Listing 5.12: Load balancer implementation for frontends interfaces.

After getting the session parameters, it is checked whether the session is present in the session map or not. If it is already present, the index of the egress interface is simply taken from the field of the entry value. If the session does not have an entry yet, the index of an egress interface is drawn, and it is saved in the session table. In both cases, if there are no errors, the last statement calls the `bpf_redirect()`, which redirects the packet to the egress of the specified interface.

The selection of the egress interface for a new session is done by hashing the session parameters and by calculating the modulo against the number of the available interfaces. The result is used as index to get the corresponding interface in the interfaces array map. The selected interface is then stored, together with the interface from which the packet was received, in a `session_interfaces` struct. This struct, in turn, is inserted into the session map with the corresponding session key obtained from the packet in order that it will be enough a simple lookup to forward the following packets.

ARP handling

Before the load balancing, there is a specific section of the code dedicated to the management of the ARP protocol which is executed only when protocol-related packets are processed. The reasons behind this protocol-specific handling have already been presented in the section discussing IP/MAC addressing management.

First of all, when an ARP packet is received it is checked whether it is a request or a response. If it is a request, it is checked whether for the target ip there is an entry in the `arp_cache` map. If there is a cache hit, the incoming packet is overwritten with the cached reply, and it is redirected to the same interface it came from. Otherwise, if there is a cache miss nothing is done, and the execution proceeds with the load balancing logic.

Instead, when it is processed an ARP reply, this is immediately stored in the cache in order to be available for the future requests. After that the execution proceeds with the usual load balancing logic which will forward the ARP reply to the interface from which the previous request arrived. This happens only for the first packet, for the following packets the cache comes into play.

In Listing 5.13 is shown the code that handles ARP packets.

```
...
// Check if packet is an ARP packet
...
if(eth->h_proto == htons(ETH_P_ARP)){
    // Get pointer to the ARP header
    struct arphdr *arph = (struct arphdr*)((void*)eth + sizeof(struct
    ethhdr));
    ...
    // Get target and sender IP
    __be32 target_ip = arph->ar_tip;
    __be32 sender_ip = arph->ar_sip;

    // Check if it is an ARP request
    if(arph->ar_op == htons(ARPOP_REQUEST)){
        // ARP request

        // ARP reply frame
        struct arp_reply_frame *arp_reply;
        struct arp_reply_frame *arp_buffer = data;

        // Check if the reply for target IP is in the cache
        arp_reply = arp_cache.lookup(&target_ip);
        if(arp_reply){
            // Cache hit: respond with cached reply
            *arp_buffer = *arp_reply;
            return bpf_redirect(ingress_ifindex,0);
        }

    } else if (arph->ar_op == htons(ARPOP_REPLY)){
        // ARP reply
        // Add reply to cache
        struct arp_reply_frame *arp_buffer = data;
        arp_cache.update(&sender_ip,arp_buffer);
    }
...
```

Listing 5.13: ARP protocol handling

It is important to highlight that when the ARP packets are handled, a dummy session is created. This is necessary because session parameters are required by load balancing logic. The parameters of the session are created as reported in the following listing.

```
static inline int get_packet_session(void *data, void* data_end ,
    struct session *session_key){
    // Interpret the first part of the packet as an ethernet header
    struct ethhdr *eth = data;
    ...
    // Only IP and ARP are handled
    switch(eth->h_proto) {

        case htons(ETH_P_IP): // ipv4 packet
        ...

        case htons(ETH_P_ARP): // arp packet
        {
            // Get pointer to the ARP header
            struct arphdr *arph = (struct arphdr*)((void*)eth +
    sizeof(struct ethhdr));
            if ( (void*)arph + sizeof(struct arphdr) > data_end){
                return 0;
            }
```

```
                session_key->ip_src = arph->ar_sip;
                session_key->ip_dst = arph->ar_tip;
                // Since ETH_P_ARP is on two bytes
                // We use 0 as protocol
                session_key->proto = 0;
                // We don't have ports here
                session_key->port_src = 0;
                session_key->port_dst = 0;
                break;
            }

        default:
            return 0;
    }
    return 1;
}
```

Listing 5.14: Creation of a dummy session for ARP packets

The IP addresses are taken from the packet, while ports and protocol are set to zero. This dummy session is then treated by the load balancer like all the others, although only the first packets actually propagate through the chain.

5.3 Operator: sfc-operator

The sfc-operator is the element that implements all the logic that handles the full life cycle of the service function chains. That includes the management of the load balancers and their control and data plane. It implements the Operator Pattern of Kubernetes acting as a controller for custom resources.

The sfc-operator has been developed using *Kopf* framework, therefore it is entirely written in Python. The advantage of using Python as a language is the possibility of directly exploiting the BCC toolkit as well. In this way, it was possible to create a single program that can deal with the management of events concerning Kubernetes resources but also with the practical aspects concerning the configuration of the data plane of load balancers. On one hand, events are managed through Kopf-based handlers. On the other hand, control plane of the load balancers is implemented with the support of a *LoadBalancer* class, which provides methods to manipulate their data structures but also to manage their life cycle.

The operator job can be summarized as follows: it handles events involving chain resources and configures the cross-connections through the LoadBalancer class instances, in order that the desired state of the chain is reflected in the cluster.

5.3.1 Deployment

The sfc-operator is deployed as a *DaemonSet* whose pods run in the host network namespace with special privileges. The reason it is deployed as a DaemonSet is that it is necessary that each node has its own instance of the operator, which handles the resources related to the node itself. Moreover, since it has to perform operations on host network interfaces it must be able to interact with the host network namespace, where it has full visibility of the host ends of veth links to pods. Furthermore, it needs to be privileged because it must be able to apply changes to the network stack, for example injecting eBPF programs into interfaces. Finally, it requires two *hostPath* to be mounted as volumes: the first one must contain the *header files* of the kernel used by the node, the second one must be a path in a mounted *BPF file system.* The first is needed because the operator through BCC has to compile the C code into eBPF byte code, the latter is needed because the load balancers are based on pinned maps which require a file system configured to support that.

Regarding RBAC permissions, sfc-operator has its own *ServiceAccount* associated with a *ClusterRole*, that defines read/write permissions needed to access the handled resources. Since it is deployed as a DaemonSet, the Kopf operator must be started in *standalone* mode in order to allow the execution of all instances.

5.3.2 Handled Resources

The sfc-operator handles three different Kind of resources:

- *ServiceFunctionChain*: a Custom Resource that represent the logical definition of the desired chain of network functions.
- *LoadBalancer*: a Custom Resource that represent an instance of a load balancer between two network functions in the chain.
- *Network Function (Pod)*: a Pod resource with a special label which specifies what kind of network function it is running.

ServiceFunctionChain

This resource represents the desired state of the chain, describing its composition and how network functions are linked together.

The ServiceFunctionChain `spec`, that represent the description of the chain, is defined by the following fields:

- `targetNodeName` (`string`): specifies the name of the node on which the chain is created and therefore the node where all its network functions are deployed.

- `networkFunctions` (`[]NetworkFunction`): list of the network functions that compose the chain. Each NetworkFunction is characterized by:
  - `name` (`string`): specifies the name of the network function
  - `links` (`map[string][string]`): map of network function links where the key is the name of the linked network function, and the value is the name of the interface of the current network function to which it is logically connected.

By applying a manifest for this resource, a user provides the logical description of the chain to the operator, which can proceed with the instantiation of the cross-connections among network functions specified.

LoadBalancer

The LoadBalancer resource, differently from the ServiceFunctionChain, is typically not used directly by users, but instead it is created by the operator. This resource represents a physical instance of a deployed load balancer that cross-connect network functions. For each resource there is a corresponding load balancer with its control and data plane.

The LoadBalancer `spec` is defined as follows:

- `targetNodeName` (`string`): specifies the name of the node on which the load balancer is deployed.
- `frontend` (`Frontend`): describes the type of network function on the frontend side of the load balancer.
  - `name` (`string`): specifies the name of the network function.
  - `interface` (string): specifies the name of the interface of the network function pods linked to the load balancer.
- `backend` (`Backend`): describes the type of network function on the backend side of the load balancer.
  - `name` (`string`): specifies the name of the network function
  - `interface` (`string`): specifies the name of the interface of the network function pods linked to the load balancer.

The instances of this resource are created by the operator during the processing of the ServiceFunctionChain manifests. For each pair of network functions linked together, a LoadBalancer resource is created. After that, it will be the operator itself to proceed with the instantiation of the real load balancer compiling and injecting the eBPF code in the kernel. The handlers involved in these operations will be described in the dedicated section.

Network Function (Pod)

Service chains are made up of network functions, in this context they are deployed as Pods. In order to distinguish the Pods representing network functions from those representing generic workloads, a specific label is added to the metadata:

- `servicefunctionchain.polito.it/networkFunction`: its value specifies the name of the network function that the Pod runs.

The value specified in this label is the same that is used in the definition of the *ServiceFunctionChain* and *LoadBalancer* and it must be the same for all the pods that run the same type of network function. With this label, the operator is capable of recognizing network functions, and it can react to events that involve these resources.

In addition to this label is necessary to explicitly specify in the Pod `spec`, the *nodeName* of the node where the pod should be scheduled. The name must be the same not only for all the replicas of the same network function, but also for all the replicas of other network functions in the chain. The same node name must be used in the definition of *ServiceFunctionChain*. This is necessary because currently operators only manage resources if they are all on the same node.

5.3.3 Indices

Before starting with the description of the handlers, some auxiliary data structures are shown, that are the Kopf *indices*. They are used to index load balancers by network function they are linked to, and to index pod by the type of network function that they are running. Namespaces of the resources are also considered in all the indices.

LoadBalancers indices

The load balancer indices are two: one indexes load balancers resources by frontend network function and the other one by backend network function. These are particularly useful when it is necessary to understand which load balancers need to be updated when an event involves a specific network function. What they provide is a dictionary that has the namespace and the name of a network function as key and list of objects, composed by the load balancers name and their *frontend/backend* interface name, as values. This information is extracted from the LoadBalancers resources. The definition of the indices is presented in the Listing 5.15.

```
@kopf.index('loadbalancers', field='spec.targetNodeName', value=node_name)
def lb_by_frontend_index(name, namespace, spec,**_):
    nf_name = spec['frontend']['name']
    fe_interface = spec['frontend']['interface']
    return { (namespace, nf_name): {'name': name, 'interface': fe_interface} }
@kopf.index('loadbalancers', field='spec.targetNodeName', value=node_name)
def lb_by_backend_index(name, namespace, spec,**_):
    nf_name = spec['backend']['name']
    be_interface = spec['backend']['interface']
    return { (namespace, nf_name): {'name': name, 'interface': be_interface} }
```

Listing 5.15: LoadBalancers indices definition.

Network Function Pods index

The network function pod index is useful when it is needed to get the list of all the pods that are running a particular kind of a network function. This happens for example when a load balancer is created to cross-connect two network functions in a chain, and it is necessary to link the replicas of the network functions to its frontend and backend.

This index not only stores the list of pod names for each network function type, but also the details about its interfaces which are retrieved from the `host-link` annotation. The content of the annotation is loaded as a JSON object in order to be accessed like a dictionary, and it is stored along with the name of the pod to which it belongs. As a result, for each network function type in a namespace the index provides a list of objects, each one representing a specific pod instance and its interfaces.

The definition of the index is presented in the Listing 5.16.

```
@kopf.index('pods',
    annotations={'servicefunctionchain.polito.it/host-link': kopf.PRESENT},
    labels={'servicefunctionchain.polito.it/networkFunction': kopf.PRESENT},
    field='spec.nodeName', value=node_name)
def pod_by_networkfunction(name, namespace, annotations, labels,**_):
    nf_name = labels['servicefunctionchain.polito.it/networkFunction']
    return { (namespace, nf_name): {'name': name, 'interfaces': json.loads(annotations['servicefunctionchain.polito.it/host-link'])}}
```

Listing 5.16: Network Function Pod index definition.

5.3.4 Handlers

The elements that enable the operator functionalities are the handlers. They are used to manage all the resources described in the previous section. It is important to point out that all the handlers apply filters on the resources to be managed, in particular each handler manages only the resources that belong to the node on which it is running.

ServiceFunctionChain creation handler

This handler is executed only once, when a new ServiceFunctionChain resource is created. It operates as follows: it iterates over the links of each network function specified in the chain, and for each link it creates a LoadBalancer resource. This resource is *adopted* by the current *ServiceFunctionChain*, which means that it will inherit the namespace and it will have a *ownerReference* to it. The latter enables the mechanism of cascading deletion, meaning that Kubernetes will automatically delete the LoadBalancer resources when their owner, a ServiceFunctionChain instance, is deleted. Lastly, it pushes this new resource to the API server.

In the following Listing is shown the code snippet that creates the LoadBalancers.

```python
@kopf.on.create('servicefunctionchain', field='spec.targetNodeName',
    value=node_name)
def create_nsc_fn(spec, name,namespace,**_):
...
for target_name, current_ifname in current_nf['links'].items():
    # Get link from the target NF corresponding to the current NF
    target_ifname = nf_dict[target_name]['links'][current_name]
    # Create load balancer name
    lb_name = "lb-"+current_name[:min(len(current_name),
   NF_PREFIX_NAME_LEN)] + "-" + target_name[:min(len(target_name),
   NF_PREFIX_NAME_LEN)]
    # Create the load balancer manifest from a template
    lb_text = lb_template.format(name=lb_name, ...)
    lb_data = yaml.safe_load(lb_text)

    # Make this loadbalancer a child of current chain object
    # (set owner reference... enable cascade deletion)
    kopf.adopt(lb_data)
    try:
        # Create the loadbalancer resource using K8s client
        api.create_namespaced_custom_object(
            group='servicefunctionchain.polito.it',
            version='v1',
            namespace=namespace,
            plural='loadbalancers',
            body=lb_data
        )
    except client.ApiException as err:
        ...
```

Listing 5.17: ServiceFunction creation handler.

The name of the LoadBalancer is built from the names of the linked network functions while the manifest is created customizing a template.

LoadBalancer creation and resuming handler

This handler is called when a LoadBalancer resource is created and when the operator is resumed after a downtime. The task of this handler is to instantiate a LoadBalancer object of the homonymous Python class, which provides an abstraction for control and data plane management of eBPF load balancers. This class will be presented in detail in the next subsection.

After the creation of the object, it is ensured that the interfaces of all the network function replicas, which the load balancer must connect, are properly configured in its data plane in order to enable the forwarding. In order to get all the network functions involved, the handler uses the pod index. As mentioned in the index description, for each pod are available the details of its links and this allows the operator to know which interfaces he has to work on. In the end, the object created is added to a global dictionary in order to be accessible, when it is necessary, in other handlers that have to perform actions on it.

```
@kopf.on.resume('loadbalancers', deleted=True ,field='spec.
    targetNodeName', value=node_name)
@kopf.on.create('loadbalancers', field='spec.targetNodeName', value=
    node_name)
def create_lb_fn(pod_by_networkfunction: kopf.Index, spec, name,
    namespace,**_):
     # Get interface names
     fe_interface = spec['frontend']['interface']
     be_interface = spec['backend']['interface']
     # Create loadbalancer object
     loadbalancer = sfc.LoadBalancer(name)

     # Get all frontends network functions to link
     fe_pods = pod_by_networkfunction.get((namespace, spec['frontend'
    ]['name']), [])
     for fe_pod in fe_pods:
         # Get corresponding frontend interface index
         ifindex = fe_pod['interfaces'].get(fe_interface)
         # Ensure interface in load balancer frontend interfaces
         loadbalancer.ensure_frontend_interface(int(ifindex))

     # Get all backends network functions to link
     be_pods = pod_by_networkfunction.get((namespace, spec['backend'][
    'name']), [])
     for be_pod in be_pods:
         # Get corresponding backend interface index
         ifindex = be_pod['interfaces'].get(be_interface)
         # Ensure interface in load balancer backend interfaces
         loadbalancer.ensure_backend_interface(int(ifindex))

     # Add loadbalancer to dictionary
     loadbalancers_dict[name] = loadbalancer
```

Listing 5.18: LoadBalancer creation and resuming handler.

After the creation of the LoadBalancer object, as mentioned before it is ensured that all the interfaces of the frontend and backend network functions are correctly configured, which means that the eBPF code has been injected in them, and that their index number has been added to the load balancer data structures. The reason it is ensured that they have already been configured, instead of configuring them directly, derives from the fact that this handler is called both in case of load balancer creation and in case of operator resume after a downtime. In the latter case it is not clear which interfaces have already been configured and which are not. Therefore, it is necessary to process them all again and add only the missing ones, and this is the behaviour that the "ensure" methods of the LoadBalancer class provide.

LoadBalancer deletion handler

The LoadBalancer deletion handler is in charge of performing cleaning up operations on the load balancers. This task is delegated to a method of the corresponding Python object. After that, the deleted load balancer is removed from the dictionary.

```
@kopf.on.delete('loadbalancers', field='spec.targetNodeName', value=
   node_name)
def delete_lb_fn(spec,name,**_):
    # Cleanup loadbalancer and delete from dictionary
    lb = loadbalancers_dict.get(name)
    if lb:
        lb.cleanup()
        del loadbalancers_dict[name]
```

Listing 5.19: LoadBalancer deletion handler.

NetworkFunction Pod update and deletion handlers

The Network Function Pod handlers allow the operator to intervene in two key moments of the Pod life cycle: when it starts running and when it is going to be deleted. It is important to highlight that these are the handlers that enable the seamless adapting features of the solution to the scaling of network function replicas. When a new replica starts running the operator catch this status update and ensures that the pod is included in the chain, so that it can be selected by the load balancers. On the other hand, when a replica is going to be deleted the operator proceeds with the removal of the replica from the chain, so that traffic no longer passes through that pod.

The operations performed by the two handlers are very similar, the only difference is that in the first case pod interfaces are added to the load balancers involved, while in the second case they are removed. To be precise, the interfaces involved in the load balancer logic are those on the host namespace side and not those on the pod side, as explained previously. In order to do their operations for each Pod, the handlers need to get the indexes of the corresponding interfaces on the host side, and the lists of the load balancers that must be updated. The former are obtained from the `host-link` annotation in the pod resource, while the latter from the load balancers indices.

After that, for each load balancer frontend/backend that must be updated, it is necessary to determine with which specific host interface it is associated. To do this, it is sufficient to obtain the *interface* name string from the object obtained from the index and get the value corresponding to this name inside the dictionary obtained from the `host-link` annotation. Once the host interface index has been found, it is possible to proceed with the addition or removal operations.

In the Listing 5.20 is presented the code that implements the *update* handler. This, unlike the *creation* handler, enables the definition of special filters that allow operator to react exactly when the pod changes from the *Pending* to the *Running* state. This prevents the pod from being added to the chain before it is actually ready to receive traffic.

```
@kopf.on.update('pods',
    labels={'servicefunctionchain.polito.it/networkFunction': kopf.
   PRESENT},
    annotations={'servicefunctionchain.polito.it/host-link': kopf.
   PRESENT},
    field='status.phase', old='Pending', new='Running',
    when=lambda spec, **_: spec.get('nodeName', '') == node_name)
def update_pod_fn(lb_by_frontend_index: kopf.Index,
   lb_by_backend_index: kopf.Index, namespace, annotations, labels,**
   _):
    # Get network function name from label
    nf_name=labels['servicefunctionchain.polito.it/networkFunction']
    # Get network function interfaces
    interfaces=json.loads(annotations['servicefunctionchain.polito.it
   /host-link'])

```

```
    # Search using index the involved loadbalancers
    lb_to_add_fe=lb_by_frontend_index.get( (namespace, nf_name), [])
    lb_to_add_be=lb_by_backend_index.get( (namespace, nf_name), [])
    # For each loadbalancer involved on the frontend
    for lb_cr in lb_to_add_fe:
        # Get lb name
        lb_name = lb_cr['name']
        # Get the interface name attached to the frontend
        fe_ifname = lb_cr['interface']
        # Get the ifindex corresponding to the ifname
        ifindex = interfaces.get(fe_ifname)
        # Get LoadBalancer object from the dictionary
        lb_obj: sfc.LoadBalancer = loadbalancers_dict.get(lb_name)

        if lb_obj and ifindex:
            # Ensure interface ifindex is among lb frontends
            lb_obj.ensure_frontend_interface(int(ifindex))
    # For each loadbalancer involved on the backend
    for lb_cr in lb_to_add_be:
        # Get lb name
        lb_name = lb_cr['name']
        # Get the interface name attached to the backend
        be_ifname = lb_cr['interface']
        # Get the ifindex corresponding to the ifname
        ifindex = interfaces.get(be_ifname)
        # Get LoadBalancer object from the dictionary
        lb_obj: sfc.LoadBalancer = loadbalancers_dict.get(lb_name)

        if lb_obj and ifindex:
            # Ensure interface ifindex is among lb backends
            lb_obj.ensure_backend_interface(int(ifindex))
```

Listing 5.20: Network Function Pod update handler.

In the definition, there are also some additional filter that restrict the execution only to pods that match specific conditions. In particular are handled only update events of pods with the `networkFunction` label and with the `host-link` annotation. Similarly to the other cases, there is also the requirement that pods must be on the same node as the operator. The combination of all these filters ensures that only pods that have been correctly configured and started will be handled.

The *deletion* handler, as mentioned before, has the same structure as the *update* handler, with the only difference that it is executed when a pod is about to be deleted. Its job is to remove the pod interfaces from load balancers as soon as possible to prevent traffic from being forwarded to it after its termination.

```
@kopf.on.delete('pods',
    labels={'servicefunctionchain.polito.it/networkFunction': kopf.
   PRESENT},
    annotations={'servicefunctionchain.polito.it/host-link': kopf.PRESENT},
    when=lambda status, spec, **_: spec.get('nodeName', '') == node_name)
def delete_pod_fn(lb_by_frontend_index: kopf.Index, lb_by_backend_index: kopf.Index, namespace, annotations, labels,**_):
    # Get network function name from label
    nf_name = labels['servicefunctionchain.polito.it/networkFunction']
    # Get network function interfaces
    interfaces = json.loads(annotations['servicefunctionchain.polito.it/host-link'])

    # Search using index the involved loadbalancers
    lb_to_del_fe=lb_by_frontend_index.get( (namespace, nf_name), [])
    lb_to_del_be=lb_by_backend_index.get( (namespace, nf_name), [])
    # For each loadbalancer involved on the frontend
    for lb_cr in lb_to_del_fe:
        # Get lb name
        lb_name = lb_cr['name']
        # Get the interface name attached to the frontend
        fe_ifname = lb_cr['interface']
        # Get the ifindex corresponding to the ifname
        ifindex = interfaces.get(fe_ifname)
        # Get LoadBalancer object from the dictionary
        lb_obj: sfc.LoadBalancer = loadbalancers_dict.get(lb_name)

        if lb_obj and ifindex:
            # Delete interface ifindex from lb frontends
            lb_obj.del_frontend_interface(int(ifindex))
    # For each loadbalancer involved on the backend
    for lb_cr in lb_to_del_be:
        # Get lb name
        lb_name = lb_cr['name']
        # Get the interface name attached to the backend
        be_ifname = lb_cr['interface']
        # Get the ifindex corresponding to the ifname
        ifindex = interfaces.get(be_ifname)
        # Get LoadBalancer object from the dictionary
        lb_obj: sfc.LoadBalancer = loadbalancers_dict.get(lb_name)

        if lb_obj and ifindex:
            # Delete interface ifindex from lb backends
            lb_obj.del_backend_interface(int(ifindex))
```

Listing 5.21: Network Function Pod deletion handler.

5.3.5 LoadBalancer Class

The LoadBalancer class implements an interface for controlling the eBPF data plane of the load balancers instances. Moreover, it provides methods to handle the full lifecycle of the eBPF programs, from the code compiling and injection into interfaces to the clean-up of allocated resources. In order to that, it relies on *BCC* and *pyroute2* libraries.

Each instance of the class has a one-to-one correspondence with an instance of a load balancer data plane. When an object is created, a relative eBPF program is immediately compiled and its maps instantiated, and thanks to the abstraction of BCC is possible to embed their handles as normal instance variables. These variables are then used by the methods to perform operations on the corresponding objects. For example, it is possible to inject the eBPF program associated to the load balancer into a given interface but it is also possible to add a given interface index in a map in order to be used by load balancing logic. The operations for injecting the eBPF programs are done by means of the `tc` method of the `IPRoute` class from *pyroute2* library which provides an interface to interact with the *traffic control* layer. Instead, the manipulation of the eBPF maps is done directly by exploiting the abstractions of BCC, which allows the use of these maps as if they were python dictionaries.

The sfc-operator by means of the methods of this class can act as a control plane for load balancers configuring their behaviour. Hereinafter, the structure of the class and the most important methods are shown.

Class structure

The LoadBalancer class has only one class variable that is a *lock*, that is used only in the `__init__` method. Apart from this, it defines a set of methods that manage the operations concerning the load balancers, from the initialization to the clean-up. The methods can be divided between those that act on the *frontend* side of the load balancer and those that act on the *backend* side, but their logic is absolutely equivalent. In general, these methods work on the index of the interfaces, which is the identification number that the kernel assigns to each interface.

Furthermore, the class is entirely thread-safe, so it is possible for several threads to work on the same instances without the risk of destructive interference.

Initialization method

The `__init__` method is in charge of the instantiation of the eBPF program and of the definition of the related instance variables in the object. First of all, it stores the metadata associated with the load balancer, such as the name. Moreover, it creates the re-entrant lock that will be used in all the methods to ensure thread safety.

Finally, through the `BPF` class it instantiates an object that represent the eBPF program instance. This object is created providing the source code of the load balancer data plane, which was described in a dedicated section. It is important to point out that the source code is a template, so it has to be specialized and this is done using the name of the current instance. Once the `BPF` object has been created, all the needed handles for functions and maps in the program are extracted from it and stored as instance variables, in order to be easily used by other methods.

In the following listing is shown the code of the `__init__` method:

```python
def __init__(self, name: str):
    self.init_lock.acquire()
    try:
        # Save loadbalancer name
        self.name = name
        # Create lock object
        self.lock = threading.RLock()

        # Create bpf object
        dp_source = dp_template.replace('__LB_NAME__',name)
        self.bpf_handle = BPF(text=dp_source)
        # Get functions
        self.frontend_lb_fn = self.bpf_handle.load_func("
    handle_loadbalance_fe", BPF.SCHED_CLS)
        self.backend_lb_fn = self.bpf_handle.load_func("
    handle_loadbalance_be", BPF.SCHED_CLS)
        # Get maps
        self.session_map = self.bpf_handle.get_table("sessions")
        self.frontends_map = self.bpf_handle.get_table("
    frontends_interfaces")
        self.num_frontends_map = self.bpf_handle.get_table("
    num_frontends")
        self.backends_map = self.bpf_handle.get_table("
    backends_interfaces")
        self.num_backends_map = self.bpf_handle.get_table("
    num_backends")
    finally:
        self.init_lock.release()
```

Listing 5.22: LoadBalancer instance initialization

Another thing, that is important to underline is that maps handles do not necessarily refer to new instances, but it is possible that they refer to pre-existing instances instead. This derives from the fact that the maps are pinned, so it is possible to recover their state as long as they are mounted on the BPF file system. The consequent advantage is that, after a possible crash of the program, it is possible to recreate the LoadBalancer objects and automatically obtain a reference to the previously created maps.

As shown in the previous listing, even the init requires a lock, because the BPF class relies on modules that must be used in mutual exclusion.

Methods for adding interfaces

When it is necessary to add an interface to a load balancer, there are three operations that must be done:

1. The *clsact* qdisc must be added to the interface, in order to attach a filter
2. The eBPF function code must be injected as a *filter* into the ingress hook of the interface
3. The interface index number must be added to the corresponding load balancer map

As soon as the eBPF program is injected in the interface, all the traffic that will be sent on that interface will be processed by the load balancer logic. On the other hand, as soon as the interface is added to the map of interfaces, it can be selected as an *egress* interface by the load balancer logic for packets of a session.

Here is shown the implementation of the *add* for a backend interface:

```
def add_backend_interface(self, ifindex):
    self.lock.acquire()
    try:
        # Add appropriate qdisc to the interface
        # But first check if the old one has to be cleaned up
        qdiscs = iproute.get_qdiscs(ifindex)
        for nlmsg in qdiscs:
            tca_kind = nlmsg.get_attr('TCA_KIND')
            if(tca_kind == "clsact"):
                # Cleanup old qdisc (this will clean all filters)
                iproute.tc("del", "clsact", ifindex)
        iproute.tc("add","clsact", ifindex)

        # Add filter to the interface
        iproute.tc("add-filter", "bpf", ifindex, ":1", fd=self.
    backend_lb_fn.fd, name=self.backend_lb_fn.name,parent="ffff:fff2",
     direct_action=True,classid=1)

        # Get current number of backends
        num_be = self.num_backends_map[0].value

        # Add backend interface to map
        be_leaf = self.backends_map.Leaf(ifindex)
        self.backends_map[num_be] = be_leaf
        # Update number of backends
        num_be_leaf = self.num_backends_map.Leaf(num_be+1)
        self.num_backends_map[0] = num_be_leaf
    finally:
        self.lock.release()
```

Listing 5.23: Method to add a backend interface

If a clsact qdisc is already present, this is deleted before adding the new one, so that any pre-existing filters are deleted. The new filter is added loading the eBPF function corresponding to the backend side on the *ingress* hook of the interface.

Other methods related with the add, are those that *ensure* that a given interface is linked to the load balancer. These check if the index of a specific interface is present in a load balancer map. If it is already present, they do nothing, if it is not, then they call the corresponding add method.

In the following Listing is shown the code of the method that ensures that a backend interface is added to the load balancer:

```
def ensure_backend_interface(self,ifindex):
    self.lock.acquire()
    try:
       # Get backend interfaces
        num_be = self.num_backends_map[0].value
        be_interfaces = list(map(lambda ifindex: ifindex.value,self.
    backends_map.values()[:num_be]))

        # Check if ifindex is already in the map
        if ifindex in be_interfaces:
            return

        # Add interface to frontends
        self.add_backend_interface(ifindex)
    finally:
        self.lock.release()
```

Listing 5.24: Method to ensure that a backend interface is added

Methods for removing interfaces

When an interface must be removed from a load balancer, there are three operations that must be done:

1. The *clsact* qdisc must be removed from the interface
2. The interface index must be removed from the corresponding load balancer map
3. All the entries that included the interface in the session table must be removed from it

First of all, the clsact qdisc is removed, in order that all the attached eBPF filters are implicitly removed too. After that, the interface must be removed from the maps, in order that the load balancer no longer selects it as a possible egress. Finally, all sessions that were forwarded on that interface must be reassigned to other interfaces. This behaviour is enabled by simply removing all the entries that included the deleted interface.

Here is shown the implementation of the del for a frontend interface:

```python
def del_frontend_interface(self, ifindex):
    self.lock.acquire()
    try:
        # Delete qdisc if present (this will remove all filters)
        qdiscs = iproute.get_qdiscs(ifindex)
        for nlmsg in qdiscs:
            tca_kind = nlmsg.get_attr('TCA_KIND')
            if(tca_kind == "clsact"):
                iproute.tc("del", "clsact", ifindex)

        # Get current number of frontends
        num_fe = self.num_frontends_map[0].value

        # Search index of the interface in the map
        index = None
        found = False
        for i in range(num_fe):
            if(self.frontends_map[i].value == ifindex):
                index = i
                found = True
                break
        if(not found):
            return

        # Get last ifindex in the map
        last_ifindex = self.frontends_map[num_fe-1].value
        # Substitute the deleted ifindex with the last ifindex
        fe_leaf = self.frontends_map.Leaf(last_ifindex)
        self.frontends_map[index] = fe_leaf
        # Update number of frontends
        num_fe_leaf = self.num_frontends_map.Leaf(num_fe-1)
        self.num_frontends_map[0] = num_fe_leaf

        # Cleanup of sessions
        self.cleanup_sessions(ifindex)
    finally:
        self.lock.release();
```

Listing 5.25: Method to delete a frontend interface

Methods for cleaning up LoadBalancer

Finally, the LoadBalancer class provide some methods to clean-up all the involved resources. They allow to clear specific maps such as frontends and backends but also to unpin them from the file system in order that kernel resources can be fully released. Along with the cleaning of the maps, the cleaning of the interfaces is obviously done, from which the *clsact qdisc* and consequently all the *filters* are removed. These methods are useful when a LoadBalancer resource has to be finalized because it is going to be deleted.

In the following listing is shown the method that finalize the LoadBalancer object:

```python
def cleanup(self):
    self.lock.acquire()
    try:
        # Clear all frontend and backend interfaces
        # (to remove eBPF filter from interfaces)
        self.clear_frontends()
        self.clear_backends()


        # Delete folder of pinned eBPF maps (unpin of eBPF maps)
        shutil.rmtree(NSC_BPFFS_PATH+'/'+self.name,ignore_errors=
    False)
    finally:
        self.lock.release()
```

Listing 5.26: Method to clea up LoadBalancer object

To clear maps are called specialized methods that simply delete all the entries and for each entry they proceed with the clean-up of the associated interface, which consists in eliminating the added clsact qdisc.

Chapter 6

Evaluation

In this chapter is provided a performance evaluation of the system based on different kind of tests. The evaluations are made on the performance of the load balancers and on the reaction time of the overall system to the occurrence of specific events. First of all, a description of the tests environment will be given, then all tested scenarios will follow.

6.1 Testbed characteristics

Tests were conducted on a desktop machine on which was deployed a Kubernetes cluster composed of two nodes: one master node and one worker node. To deploy the cluster Kind[1] was used, it allows to run a Kubernetes cluster using Docker containers as nodes. As main benchmarking tool was used iperf3.

Machine characteristics are listed below:

- Processor: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz with 4 cores and Hyper-threading enabled
- Memory: 16 GB of DRAM
- Kernel Version: Ubuntu 20.01.1
- Kernel Release: 5.11.0-41-generic

Software versions:

- Kind: 0.11.1

[1]https://kind.sigs.k8s.io/

- Kubectl client: 1.21.2
- Kubernetes API: 1.21.1
- Docker Engine: 20.10.7
- iperf3: 3.7

Deployments

In the tests different kinds of Kubernetes deployments have been used, which are based on three Docker images:

- *iperf3-client*: an image with iperf3 installed and some scripts for testing. By default, it sleeps indefinitely to allow a shell to be attached.
- *iptables-firewall*: an image that contains an init script which set up a bridge between two interfaces and install iptables rules on it.
- *iperf3-server*: an image with iperf3 installed which by default starts iperf3 in server mode and waits for connections.

All these images are based on Alpine Linux, which is an extremely lightweight distribution. It is important to highlight that for the tests conducted, there were no special rules configured on the firewall, it simply accepted all traffic in transit.

6.2 Load Balancer performance

In this section is analyzed the load balancer performance. First, it will be compared to other Kernel interconnection mechanisms and then will be evaluated how performance changes as handled interfaces increase.

6.2.1 Performance comparison

The load balancer performance has been evaluated and compared to other two interconnection mechanisms provided by Linux kernel that are *virtual ethernet pairs* and *bridges*. The tests have been conducted in two different scenarios:

- A chain composed of a client pod and a server pod
- A chain composed of a client pod a transparent firewall pod and a server pod

For each scenario there is a specific sub-case for each cross-connection mechanism. All cases for both scenarios are shown in Figures 6.1-6.6:

First scenario sub-cases

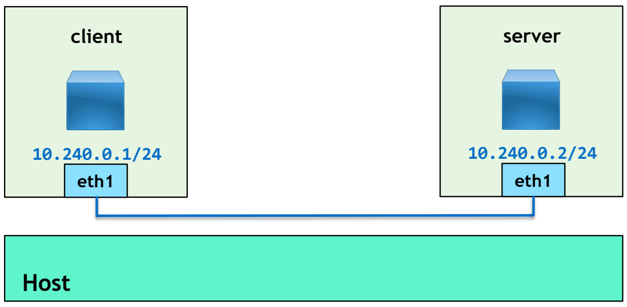


Figure 6.1: Client-Server connected through a veth pair.

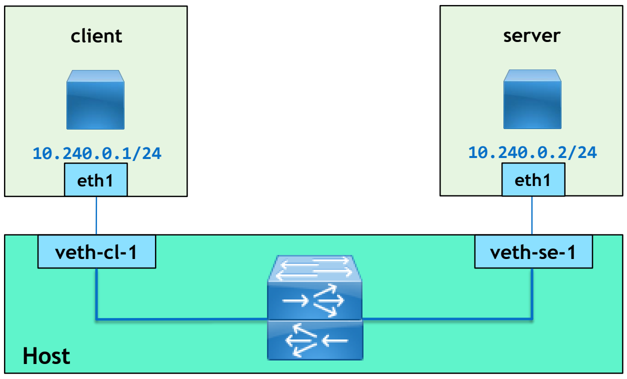


Figure 6.2: Client-Server connected through a load balancer.

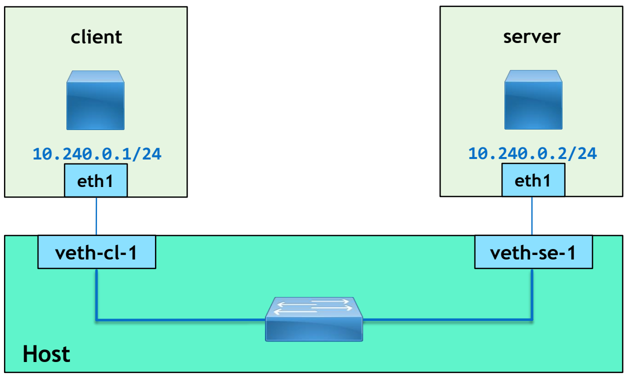


Figure 6.3: Client-Server connected through a linux bridge.

Second scenario sub-cases

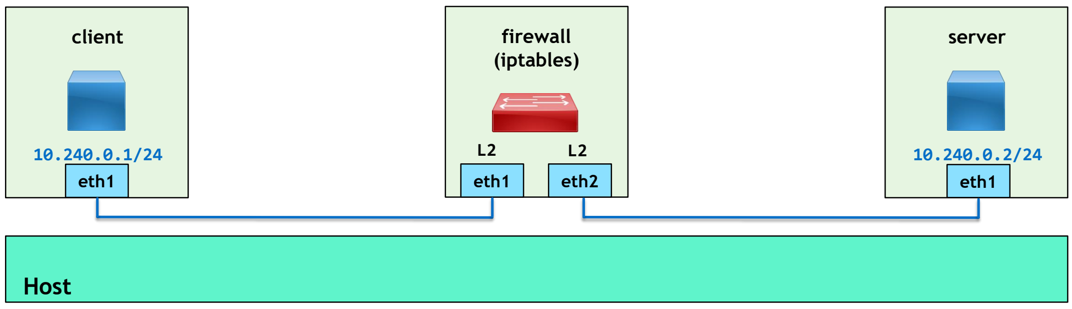


Figure 6.4: Client-Firewall-Server connected through veth pairs

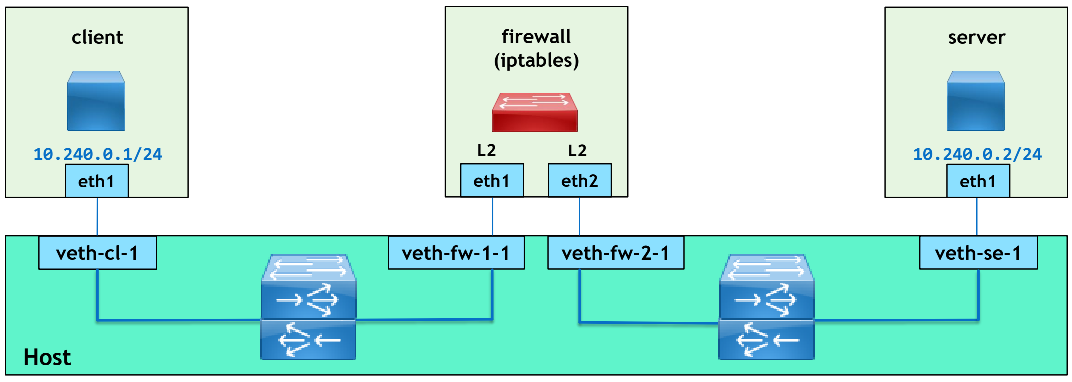


Figure 6.5: Client-Firewall-Server connected through load balancers.

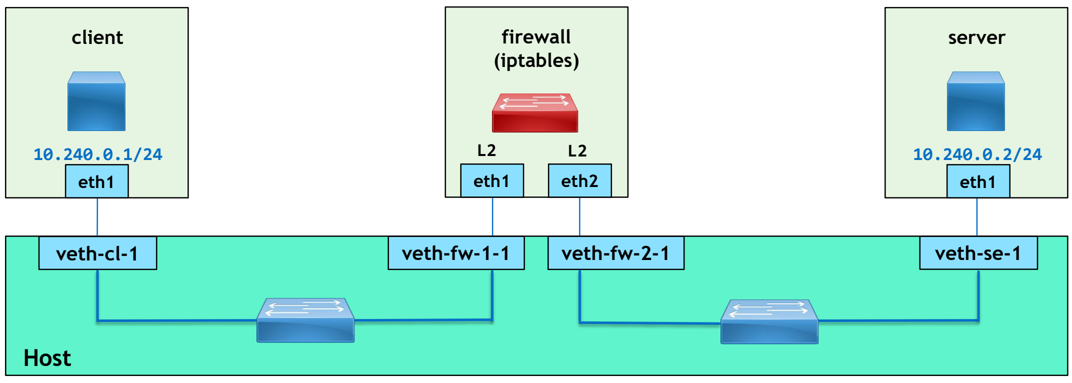


Figure 6.6: Client-Firewall-Server connected through linux bridges.

Results

Results in Figure 6.7 shows that the load balancer performance is in an intermediate position between that of veth and bridge. Since load balancer introduces additional logic compared to the simple veth is comprehensible that it introduces higher latencies and as a result a lower throughput. However, thanks to the efficiency and high performance provided by eBPF technology, performance is only slightly lower than veth, even though load balancer introduces more complex logic. On the other hand, the load balancer outperforms noticeably the linux bridge.

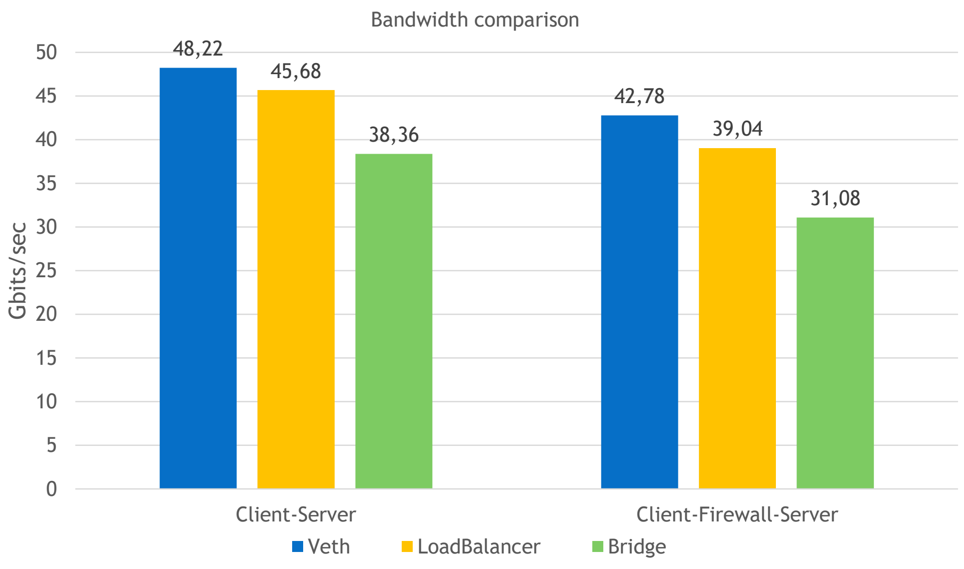


Figure 6.7: Performance comparison between scenarios' subcases.

6.2.2 Performance as replicas increase

With the following tests is evaluated the variation in performance of load balancer as number of replicas increases. The chain deployed for this test is again composed by three kind of pod: a client, a firewall and a server. Client and server are always based on iperf3 and again the firewall is based on a bridge internal to the pod that connects the two interfaces.

The figure 6.8 shows the tested scenario. The firewall is the pod that is scaled in order to have an increasing number of replicas.

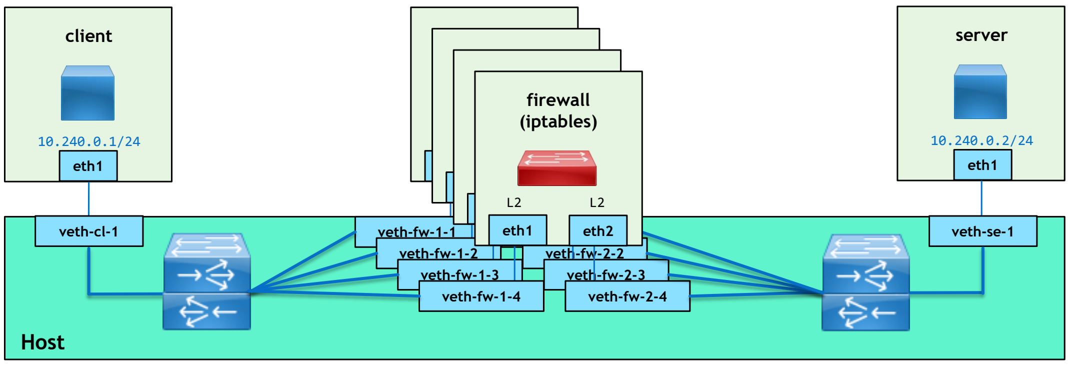


Figure 6.8: Chain with multiple firewall replicas.

Results

From the results in Figure 6.9 it turns out that number of replicas does not affect load balancer performance. This result is not surprising considering how the load balancer logic is implemented. Its forwarding operations are based on a session table, which is implemented via a hash table and its performance is independent of the number of interfaces. The only operation in which the number of replicas is explicitly involved is the one that selects the egress interface for the first packet of a new session, but since this is simply a modulo operation that occurs for a single packet, the effect is absolutely negligible.

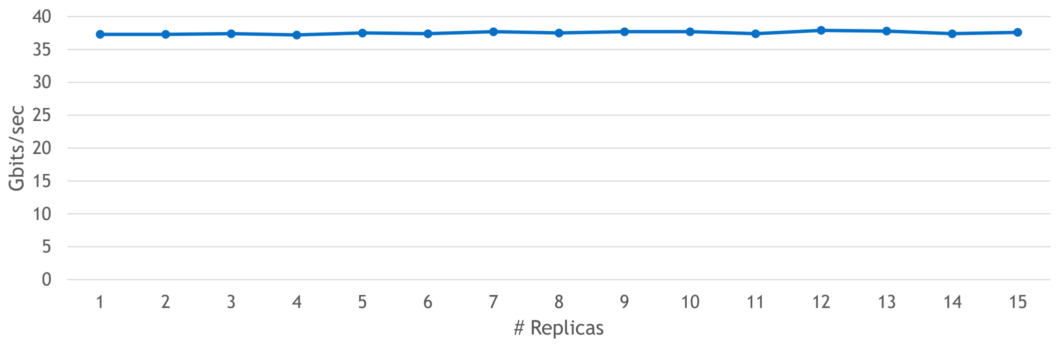


Figure 6.9: Performance as firewall replicas increase.

6.3 System Reaction time

In this section is evaluated the reaction time of the system following the creation or deletion of a network function replica in a chain. In order to do this, it is measured how much time passes from the change of state of a replica to when the operator notices it, and also how long it takes to the operator to configure/clean-up the interfaces and update the load balancers involved with the event.

For the tests, a generic network function with two network interfaces was used. For this reason, the add or remove operations on the interfaces are performed twice for each event.

6.3.1 New Replica

Here is analyzed the reaction time to a new replica running event for a network function deployment in a chain. This test evaluates how long it takes from when a pod starts running until it is fully included in the chain. When the operator notices that there is a new replica up and running, it starts the handler which has to find all the load balancers that need to be updated in order to consider the new replica. Once found, it proceeds with all the necessary operations, starting from obtaining the indexes of the network function interfaces on the host namespace side, up to the injection of the eBPF code in those interfaces and the updating of the load balancers maps.

For the test, first of all, it was measured how much time passes from the start of the Pod execution to when the operator notices it. After that, the total activity time of the operator was measured, which is mainly composed of the time needed to add the clsact qdiscs and the eBPF programs to the interfaces. The time taken to search for the load balancers involved and to update the eBPF maps is not shown in the results. The reason is that its contribution to total uptime is so low that can be considered negligible.

Results in Figure 6.10 show that most of the delay is introduced by the propagation in the cluster of the event regarding pod running status. The time required by the operator activity is much less compared to it. This means that the delay introduced by operator is less significant than the latency introduced by the generation and propagation of events in the cluster. From the chart it can be seen that the most time-consuming operation is the injection of the eBPF filter with the load balancing logic into the ingress TC hook of an interface.

It is important to point out that the operator could react to the pod *creation* event, which means even before it is actually running. However, this would result in the pod being prematurely added to the chain. After a pod resource is created, it takes some time to set up its sandbox and start up its containers. Consequently, when the operator would react to the creation, there could be the risk of trying

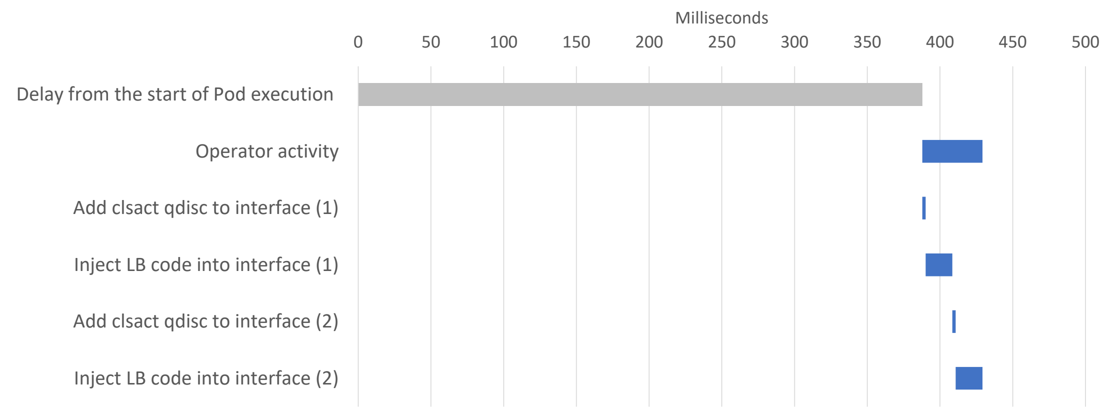


Figure 6.10: Reaction time composition for a new replica event.

to include in the chain a network function that does not yet have the interfaces configured or that is not yet ready to receive traffic. For this reason, the operator only reacts when it is sure that the pod is running.

6.3.2 Replica deleted

The opposite case to the previous is the deletion of a replica. In this case, the operator reacts as soon as it detects that a pod is going to be deleted. This test measures how long it takes from when the pod begins its graceful termination to when it is successfully removed from the chain. The operator acts by removing the clsact qdisc previously added in the interfaces and updates the load balancers maps so that they no longer select the interfaces of the pod that is about to be deleted. As in the previous case, the time to find the load balancers involved and to update the maps is not shown in the results as it was negligible.

The tests measured how much time passes from the beginning of the pod termination to when the operator reacts to it. In addition to this, the total activity time for the operator was also evaluated, which in this case is mainly composed of the time required for the removal of the clsact qdisc from the interfaces of the network function. It is also important to specify that the pod used in testing took about 0.5 seconds to gracefully terminate. The obtained results are shown in Figure 6.11.

From the diagram it can be seen that the total activity time of the operator is similar to the previous one. In this case the main contribution is given by the removal of the clsact qdisc from the interfaces, which takes more time than its addition. This derives from the fact that when the qdisc is deleted, all the associated filters are also implicitly removed, which in this case involves the removal

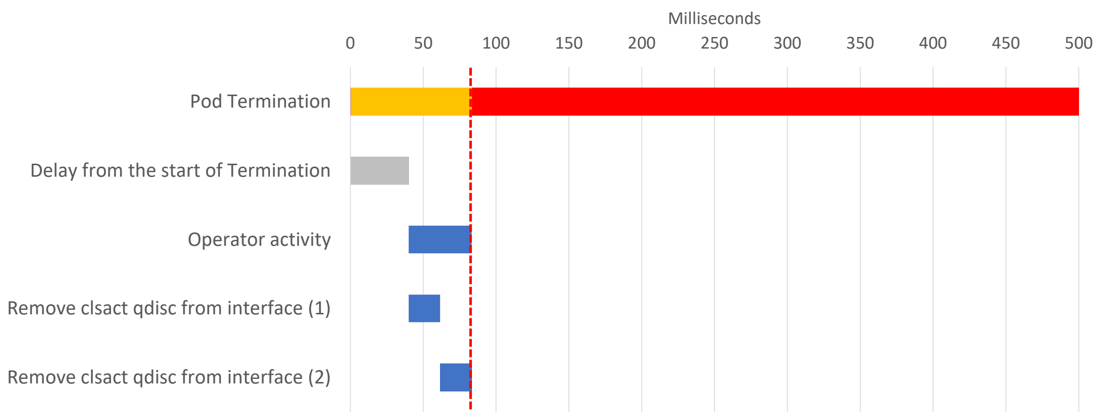


Figure 6.11: Reaction time composition for a terminating replica event.

of the eBPF programs previously injected. Even in this case the operator reacts after a slight delay as evidenced by the grey bar in the graph, this is always due to event propagation delays in the cluster.

However, a very important aspect to consider in this scenario is the time it takes for the pod to terminate. As mentioned previously, in the tests it was used a pod which gracefully terminates in 0.5 seconds, however other pods may take less and others longer. What is important is that the operator finishes his operations before the pod terminates. If a pod terminates earlier, there may be temporary packet losses in the chain because load balancers try to send traffic to replicas that no longer exist, because they haven't been updated yet. For the case considered, the dashed red line in the chart highlights the moment from which the replica is no longer part of the chain. If the pod had been terminated immediately in 10 ms, there would have been a time window in which packets would have been dropped.

Since there are unavoidable propagation delays of events and the operator's actions are not instantaneous, it is needed to force the network functions not to terminate too fast to prevent these problems.

Chapter 7

Conclusions

This thesis has presented the prototype of a system to integrate Service Function Chains in Kubernetes, enabling seamless autoscaling of network functions through the deployment of eBPF-based load balancers among replicas. The main focus was on trying to bring the benefits of Cloud Native platforms to network workloads and the proposed solution has shown a possible approach in this direction.

The system was composed by elements designed to be natively integrated into Kubernetes environment, leveraging the extensibility features of the aforementioned platform. This was the case of the custom CNI that configures network interfaces of NFs, and of the Operator which act as controller of system functionalities.

As concerns load balancers, the use of eBPF allowed the integration of their logic directly into the networking data path, with advantages in terms of performance and transparency towards network functions.

The evaluation of the prototype showed that the overhead of the additional load balancing logic is low, and this is mainly due to efficiency of eBPF technology. Moreover, the analysis of reaction time to replicas creation/deletion showed that system quickly reacts to their variations, highlighting that most of the delay is due to the propagation delays of events themselves.

7.1 Future Works

The solution, being a prototype, has some limitations which can be overcome by future works. The main objective of this work was to create a proof of concept of a system capable of managing chains with a variable number of replicas, therefore it was developed considering simplified scenarios in order to obtain results on feasibility in a short time.

Possible improvements include:

- Cross-node chains: at the moment, chain creation is made per-node, therefore only replicas on the specified node are considered. To enable cross-node chains is necessary to add a tunneling logic that handles links across nodes, and the operators must consider also remote replicas while configuring load balancing logic.
- Weighted Load Balancing: currently load balancing on replicas is made using an index obtained by a number derived from the hash of a session quintuple and from the number of replicas. This logic does not consider any additional parameter. It might be beneficial to consider network load metrics of the available replicas, in order to select less stressed ones.
- Support for different technologies: at present the system uses eBPF-based load balancers that work on *veth* interfaces. In future works could be explored the possibility of implementing the same system using different technologies that enable the same functionalities using user space resources as an alternative. This would allow the solution to have more integration possibilities with existing systems and would require fewer privileges than a solution totally based on kernel mechanisms.

Bibliography

[1] Network Functions Virtualisation (NFV) ETSI Industry Specification Group (ISG). *Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV*. URL: `https://www.etsi.org/deliver/etsi_gr/NFV/001_099/003/01.06.01_60/gr_nfv003v010601p.pdf`. (accessed: 02:2022) (cit. on p. 4).

[2] CNCF Telecom User Group. *Cloud Native Thinking for Telecommunications*. URL: `https://github.com/cncf/telecom-user-group/blob/master/whitepaper/cloud_native_thinking_for_telecommunications.md`. accessed: 02:2022 (cit. on p. 4).

[3] The Kubernetes Authors. *What is Kubernetes?* URL: `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. accessed: 02:2022 (cit. on p. 6).

[4] Vamsi Chemitiganti. *Kubernetes Concepts and Architecture*. URL: `https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/`. accessed: 02:2022 (cit. on p. 7).

[5] The CNI authors. *CNI*. URL: `https://www.cni.dev/`. accessed: 02:2022 (cit. on p. 10).

[6] Tigera Inc. *About Kubernetes Networking*. URL: `https://projectcalico.docs.tigera.io/about/about-k8s-networking`. accessed: 02:2022 (cit. on p. 11).

[7] The Network Service Mesh authors. *Network Service Mesh*. URL: `https://networkservicemesh.io/`. accessed: 02:2022 (cit. on p. 11).

[8] The Network Service Mesh authors. *Release v1.1.0*. URL: `https://networkservicemesh.io/docs/releases/v1.1.0/`. accessed: 02:2022 (cit. on p. 12).

[9] Cilium Authors. *BPF and XDP Reference Guide*. URL: `https://docs.cilium.io/en/stable/bpf/`. accessed: 02:2022 (cit. on p. 13).

[10] Martin A. Brown. *Traffic Control HOWTO*. URL: `https://tldp.org/HOWTO/Traffic-Control-HOWTO/components.html#c-qdisc`. accessed: 02:2022 (cit. on p. 18).

[11] The BCC Authors. *BPF Compiler Collection*. URL: `https://github.com/iovisor/bcc`. (accessed: 02:2022) (cit. on p. 18).

[12] The Linux Manual Authors. *netlink(7)*. URL: `https://linux.die.net/man/7/netlink`. (accessed: 02:2022) (cit. on p. 18).

[13] The Linux Manual Authors. *rtnetlink(7)*. URL: `https://linux.die.net/man/7/rtnetlink`. (accessed: 02:2022) (cit. on p. 18).

[14] pyroute2 Authors. *pyroute2 netlink framework*. URL: `https://pyroute2.org/`. (accessed: 02:2022) (cit. on p. 19).

[15] netlink Authors. *netlink - netlink library for go*. URL: `https://github.com/vishvananda/netlink`. (accessed: 02:2022) (cit. on p. 19).

[16] The Multus CNI Authors. *Multus CNI*. URL: `https://github.com/k8snetworkplumbingwg/multus-cni`. (accessed: 02:2022) (cit. on pp. 19–21).

[17] Sergey Vasilyev. *Kubernetes Operator Pythonic Framework*. URL: `https://github.com/nolar/kopf`. (accessed: 02:2022) (cit. on p. 20).

[18] Sergey Vasilyev. *Kopf: Kubernetes Operators Framework*. URL: `https://kopf.readthedocs.io/en/stable/`. (accessed: 02:2022) (cit. on p. 20).