POLITECNICO DI TORINO

MASTER's Degree in Computer Engineering

MASTER's Degree Thesis

Designing an eBPF-based Disaggregated Network Provider for Kubernetes

Supervisors

Prof. Fulvio RISSO

Ing. Federico PAROLA

Ing. Giuseppe OGNIBENE

Candidate

Leonardo DI GIOVANNA

April 2022

Abstract

Container orchestration plays an important role in the modern software development. Container orchestration platforms provide a base abstraction layer for simplifying the development, the deployment and the maintenance of the applications. Nowadays, the most famous and popular open-source container orchestrator is Kubernetes, a solution developed by Google. Container orchestrators are developed having in mind that handling the networking is a crucial aspect, since intrinsically distributed application components must be able to communicate to each other regardless of whether they are hosted. The component in charge of handling all the networking stuff is generally called network provider. Modern networking is increasingly implemented in software on general-purpose hardware: this allows to have much more flexibility and, at the same time, to reduce infrastructural costs. Components implementing in software the same behaviour of the traditional network appliances, called virtual network functions (VNFs), must be implemented having in mind the efficiency and the performance effectiveness. In order to reach these goals, different technologies could be used: eBPF (Extended Berkeley Packet Filter), that provides a way for executing sandboxed programs directly in the Linux kernel, is one of the most suitable one. By exploiting the above considerations, this thesis work has as its purpose that of designing and validating an eBPF-based disaggregated network provider for Kubernetes. A base virtual network topology architecture, composed mainly of standard virtual network functions (like routers, bridges and NATs), is used as base for investigating a more performant and scalable solutio; nmoreover, some improvements are considered in order to address the Kubernetes networking model compliancy. The disaggregated nature of the starting architecture is chosen and taken as primary objective since it increases the observability, the scalability and the extensibility of the solution. The development, the deployment and the interconnection of these eBPF services is made easier by leveraging Polycube, a technology built at Politecnico di Torino. A Kubernetes operator, capable of being receptive to the cluster changes and capable of reflecting them on the network infrastructure, is designed and implemented. Different performance tests are performed in order to investigate the validity of the designed solutions and to understand its limitations for future improvements.

Table of Contents

List of Figures VI

Acronyms VIII

1 Introduction 1
1.1 Goal of the thesis . . . 2

2 Background 3
2.1 Path towards container orchestration . . . 3
2.2 Kubernetes history . . . 5
2.3 Kubernetes features . . . 7
2.4 Kubernetes architecture . . . 7
2.4.1 Control plane components . . . 8
2.4.2 Nodes components . . . 10
2.5 Kubernetes objects . . . 11
2.5.1 Overview . . . 11
2.5.2 Namespaces . . . 12
2.5.3 Identifiers . . . 12
2.5.4 Status and Spec . . . 13
2.5.5 Pods . . . 13
2.5.6 ReplicaSets . . . 14
2.5.7 Deployments . . . 15
2.5.8 DaemonSets . . . 16
2.5.9 Nodes . . . 17
2.5.10 Services . . . 17
2.5.11 Endpoints . . . 18
2.6 Kubernetes operators . . . 18
2.7 The Kubernetes networking model . . . 18
2.8 The CNI specification . . . 19
2.9 eBPF (Extended Berkeley Packet Filter) . . . 20
2.9.1 Verifier . . . 20

2.9.2 Helper Functions . . . 21
2.9.3 Maps . . . 21
2.9.4 Tail calls . . . 23
2.9.5 Program Types . . . 23
2.10 BCC . . . 26
2.11 Polycube . . . 26
2.12 The VxLAN technology . . . 28

3 Architecture 30
3.1 Base architecture virtual network topology . . . 30
3.1.1 pcn-simplebridge . . . 30
3.1.2 pcn-loadbalancer-rp (pcn-lbrp) . . . 30
3.1.3 pcn-router . . . 31
3.1.4 pcn-k8sdispatcher . . . 32
3.2 The new virtual network topology . . . 34
3.2.1 pcn-k8slbrp . . . 34
3.3 The CNI plugin . . . 35
3.4 The overlay network . . . 36
3.5 The node agent . . . 36

4 Implementation 38
4.1 Automatic code generation . . . 38
4.2 pcn-k8slbrp . . . 39
4.2.1 Data Model . . . 40
4.2.2 Data plane . . . 43
4.3 pcn-k8sdispatcher . . . 43
4.3.1 Data Model . . . 43
4.3.2 Data plane . . . 46
4.3.3 Control plane . . . 46
4.4 The CNI plugin . . . 47
4.4.1 The specification requirements . . . 47
4.4.2 The CNI configuration file . . . 48
4.4.3 The implementation . . . 49
4.5 The Polykube operator . . . 60
4.5.1 The code structure . . . 61
4.5.2 Node controller . . . 61
4.5.3 Service controller . . . 64
4.5.4 Endpoints controller . . . 66
4.5.5 The recovery procedure . . . 70
4.5.6 The addresses management . . . 70
4.5.7 Environment configuration . . . 72

4.6 Network provider deployment 73

5 Evaluation 77
5.1 Tools 77
5.2 Tests overview 77
5.3 Cluster and network settings 78
5.4 Communications on the same node 79
5.4.1 TCP 79
5.4.2 UDP 80
5.5 Communications on different nodes 81
5.5.1 TCP 81
5.5.2 UDP 83

6 Conclusions 85
6.1 Future works 85

A Scripts and Commands 86

Bibliography 91

List of Figures

2.1 Evolution of the applications deployment 3
2.2 Container orchestrators usage [2] 6
2.3 Kubernetes history 6
2.4 Architectural representation of a Kubernetes cluster 8
2.5 Shared memory architecture 22
2.6 The XDP and tc hook points 23
2.7 XDP program actions 25
2.8 Polycube architecture 27
2.9 The VxLAN packet structure 29
2.10 A VxLAN example 29

3.1 The base architecture virtual network topology 31
3.2 pcn-k8sdispatcher egress traffic flow chart 32
3.3 pcn-k8sdispatcher ingress traffic flow chart 33
3.4 The new architecture virtual network topology 35
3.5 Network provider architecture 37

5.1 Same node throughput comparison for TCP traffic 79
5.2 Same node throughput comparison for TCP traffic with an increasing number of Services/Pods 80
5.3 Same node throughput comparison for UDP traffic 81
5.4 Same node throughput comparison for UDP traffic with an increasing number of Services/Pods 81
5.5 Different nodes throughput comparison for TCP traffic 82
5.6 Different nodes throughput comparison for TCP traffic with an increasing number of Services/Pods 82
5.7 Different nodes throughput comparison for UDP traffic 83
5.8 Different nodes throughput comparison for UDP traffic with an increasing number of Services/Pods 84

Acronyms

IP
: Internet Protocol

TCP
: Transport Control Protocol

UDP
: User Datagram Protoco

CNCF
: Cloud Native Computing Foundation

CNI
: Container Network Interface

VNF
: Virtual Network Function

NAT
: Network Address Translation

eBPF
: extended Berkeley Packet Filter

XDP
: eXpress Data Path

tc
: traffic control

VxLAN

Virtual eXtensible Local Area Network

Chapter 1

Introduction

The modern software development leverages micro-services architectures in order to let applications be more responsive to issues regarding flexibility, scalability and faults toleration. An application built upon these principles is composed of many services, usually deployed in different containers and connected through some sort of base network infrastructure. The amount of containers composing an application can be very high in real scenarios: from a logical point of view, the application could be composed of a very high number of logical services and each one has to be deployed using different replicas, at least in order to provide fault toleration. Each one of the container, has to be correctly configured and monitored during its life-cycle, and some mechanism that provides self-healing is needed. The fluctuations of the workload that each service will manage, have to be handled properly by scaling the amount of resources allocated to that particular service: this can be interpreted as dynamically changing the number of containers that backs the service or by allocating more resources to some of them. These problems clearly highlights the necessity of an infrastructure that automatically deploys and takes operational the containers, providing automatic configuration, self-healing and dynamic scaling capabilities. The answer to all these problems nowadays is given by Kubernetes, an open-source containers orchestrator developed by Google. If Kubernetes provides an answer to the above questions, also an answer from the network infrastructure point of view has to be given. The networking machinery must properly handle the connection between containers, providing an abstraction layer that independently provides connectivity for containers on the same host and containers on different hosts. The network infrastructures, increasingly are moving from the "hard" world to the "soft" world: this means that physical appliances now works together with virtual network functions implemented in software. Each virtual network function (VNF), following the trend discussed above for the applications, can be redounded, and many of them can be composed in different and dynamical ways. The implementation of a VNF has to be carry on

having in mind that the primary goal is the efficiency: for this reason, the virtual networking is implemented leveraging efficient and fast technologies like eBPF. Exploiting eBPF and the pluggable nature of Kubernetes, an effort can be made in order to design an efficient network provider for providing a virtual network infrastructure that the orchestrator and, at the end the applications, can leverage.

1.1 Goal of the thesis

The goal of this thesis work is to design and eBPF-based disaggregated network provider for Kubernetes. Starting from the thesis work of Ing. Hamza Rhaouati [1], an already prototyped virtual network topology, composed mainly of standard virtual network functions (like routers, bridges and NATs), was used as base for investigating a more performant and scalable solution; moreover, the architecture was extended in order to address the full Kubernetes networking model compliancy. The disaggregated nature of the starting architecture was surely maintained and taken as primary objective: indeed, it brings so many advantages like observability, scalability and extensibility. The network infrastructure was built using the eBPF technology: it allows to develop fast and efficient network functions that run directly in the Linux kernel and for this reason it was considered a must in the network provider infrastructure definition. The development, the deployment and the interconnection of these eBPF services was made easier by leveraving Polycube, a technology built at Politecnico di Torino. A communication layer, for connecting the network abstractions provided by Kubernetes (like Services) and the network infrastructure, was developed: in other terms, a Kubernetes operator, capable of being receptive to the cluster changes and capable of reflecting them on the network topology, was designed and implemented. Different performance tests were performed in order to investigate the validity of the designed solutions and to understand its limitations for future improvements.

Chapter 2

Background

This chapter provides an overview of the main components and technologies of this thesis work. First, an overview of container orchestration platforms, with a focus on Kubernetes, is provided. In the Kubernetes section, the focus will be on its networking philosophy and on abstractions leveraged during the network provider design. Then, an overview of eBPF and Polycube is provided, as they are the main enabling technologies of this work. Finally, the VxLAN technology is presented.

2.1 Path towards container orchestration

The following is an analysis of the evolution of the applications deployment. The main evolution steps are shown in Fig. 2.1.

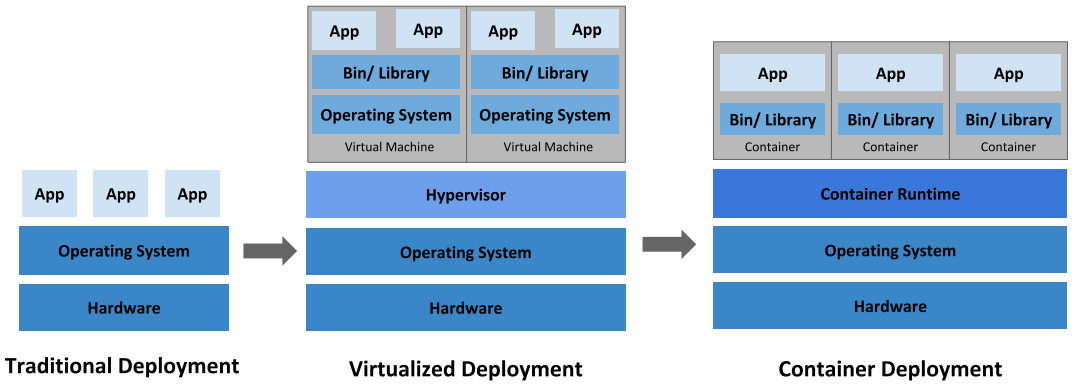

Figure 2.1: Evolution of the applications deployment

At the very beginning, application were deployed on physical server following the one-application-per-server model. In the 90', the computing technology

democratization process brought different companies to own a quite reasonable number of servers for resources sharing among different groups. Each server was, in the majority of the cases, responsible to run a single application. In this period, companies started a process of resources centralization in order to better manage the high number of servers; this process let the companies to understand the under-utilization of the owned resource. The under-utilization can be saw in terms of computing resources (most of the applications does not require an high workload for the majority of the time and old applications were not able to split their workload into different CPU core) or powering resources (taking up a server introduces an higher gap in the power consumption than exploiting it at the maximum capacity). Unfortunately, it was not possible to consolidate the deployment of different applications on the same server due to issues in:

- **performance predictability** - different applications deployed on the same server concur on resources access (CPU, RAM, network cards etc...) and this affects the single application performance
- **configuration** - different application may required incompatible server configurations like different kernel versions, different libraries versions etc...

The above two considerations let companies to stay with the one-application-per-server model for a long time.

Around the 2000s, companies started to react to this problem by leveraging virtualization. Virtualization is based on the creation of virtual hardware profiles able to abstract the concept of physical machines. Each virtual hardware profile can be used to run a different virtual server, called also virtual machine (VM), on the same physical server. The virtualization introduces different advantages; some of them are the followings:

- **isolation** - it is possible to ensure an almost complete logic partitioning of the physical server resources; this is also beneficial in term of applications security
- **consolidation** - it is possible to deploy different applications on the same physical server, avoiding resource under utilization and, at the same time, maintaining performance predictability and reducing the power consumption
- **flexibility and agility** - different virtual servers can be created, stopped, deleted, migrated, duplicated or substituted in a very flexible and rapid way: this is possible as virtual servers handling is almost decoupled from the real world

Virtualization allow to present a set of physical resources as a cluster of disposable VMs. These virtual resources can be used directly by the companies or by the companies customers (notably is the case of cloud providers).

Virtualization introduces also disadvantages. The overhead introduced by the necessity of adding a virtual hardware profile with an additional operating system acting on that, causes applications running on VMs to be slower, even if different efforts have been in order to reduce the gap from applications running of a physical machine. Furthermore, VMs startup times are not negligible at all.

In order to overcome the overheads introduced by the virtualization, efforts have been made towards the developing of containerization technologies. Containers can be seen, at very high level, as a sort of lightweight VMs which share the operating system of the host machine. Sharing the same operating system is not a major problem for most of the applications: the number of applications that requires a specific kernel version or a specific hardware profile is not so high. Containers relax the strong isolation of VMs in face of shorter bootstrap/stopping times and faster execution of the applications executed inside them. The majority of the containerization technologies leverages technologies of the Linux Kernel, such as cgroups and namespaces, in order to provide a lighter form of isolation with respect to the one provided by VMs. Also, containers facilitate the development process, letting developers to easily package their applications in a portable environment containing all the needed libraries and utilities and reducing the effort the operational teams have to do in order to make them runnable on the production environment. Containers, in this sense, meet the requirements of the modern software development.

Nowadays applications are built by leveraging (or, existing ones are migrating towards) micro-service architecture. An application built upon these principles is composed of many services, usually deployed in different containers. The amount of containers composing an application can be very high in real scenarios: from a logical point of view, the application could be composed of a very high number of logical services and each one has to be deployed using different replicas, at least in order to provide fault toleration. Each one of the container, has to be correctly configured and monitored during its life-cycle, and some mechanism that provides self-healing is needed. This arises the necessity to use a container orchestration platform (also called container orchestrator): a container orchestrator is a system designed to easily manage complex containerization deployments. As shown in Fig. 2.2, Kubernetes is by far the most used container orchestrator.

2.2 Kubernetes history

Kubernetes (commonly abbreviated with k8s) is an open-source container orchestration system platform for automating software deployment, scaling and management. The project is actually maintained by the Cloud Computing Foundation (CNCF) but it was designed by Google. Google announced for the first time

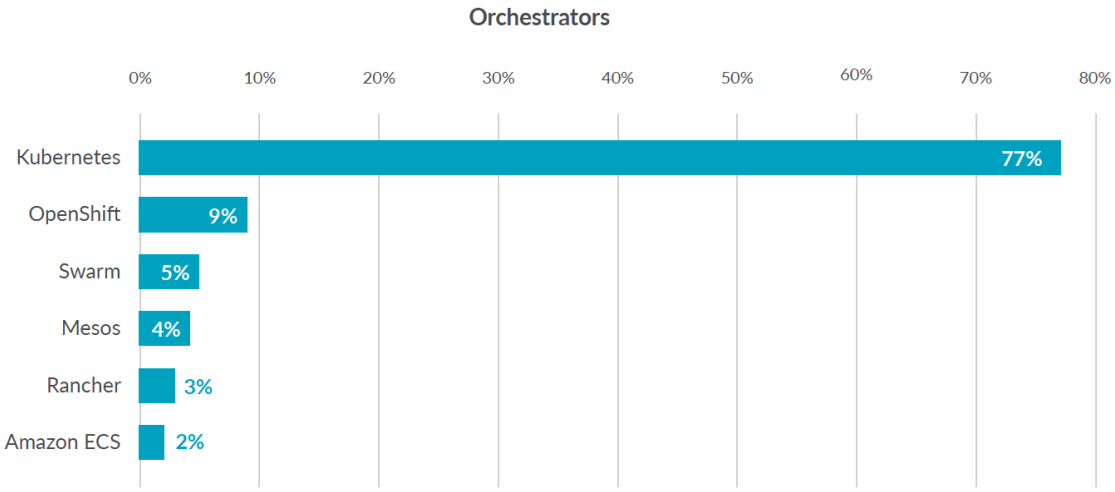

Figure 2.2: Container orchestrators usage [2]

Kubernetes in the middle of 2014. The system was heavily influenced by Google's Borg, a large-scale cluster manager used internally by Google for running hundreds of thousands of jobs. Differently from Borg, which is written entirely in C++, Kubernetes is written in Golang. The first version of Kubernetes, the 1.0, was released on July 21, 2015, contextually with a partnership set up by Google with the Linux Foundation to form the Cloud Native Computing Foundation. Kubernetes 1.0 was offered to the CNCF as a seed technology and since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company.

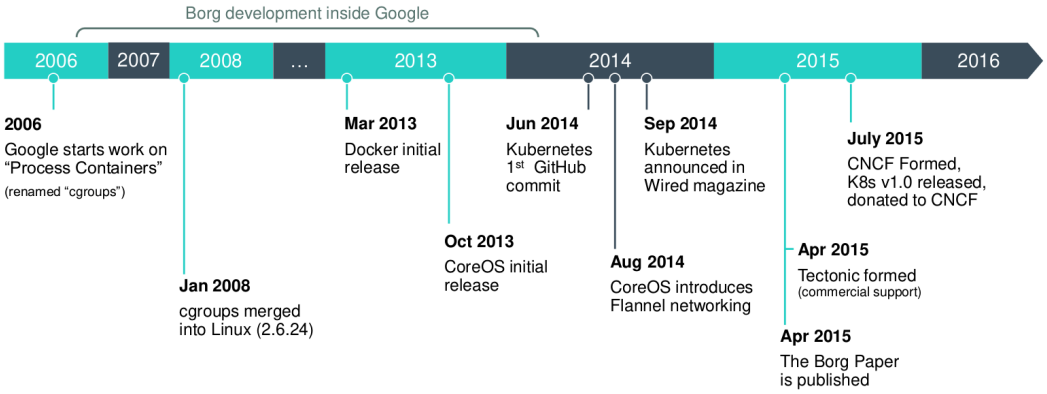

Figure 2.3: Kubernetes history

2.3 Kubernetes features

Kubernetes provides a framework to run distributed systems resiliently. It takes care of scaling and failover for applications, provides deployment patterns, and more. Kubernetes provides:

- **Service discovery and load balancing** - Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** - Kubernetes allows to automatically mount a storage system such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** - Kubernetes can use a provided description of the desired state to change the actual state of the deployed containers to the desired state at a controlled rate. For example, it is possible to automate Kubernetes to create new containers for a deployment, remove existing containers and adopt all their resources to the new container.
- **Automatic bin packing** - Kubernetes is provided with a cluster of nodes that it can use to run containerized tasks. It is possible to specify Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto the nodes to make the best use of the resources.
- **Self-healing** - Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** - Kubernetes is able to store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. Secrets and application configuration can be deployed and updated without rebuilding container images, and without exposing secrets in the stack configuration.
- **Features pluggability** - Kubernetes is not monolithic, and solutions for logging, monitoring, alerting solutions are optional and pluggable. Also network providers and container runtimes are easily replaceable.

2.4 Kubernetes architecture

When Kubernetes is deployed a **Kubernetes cluster** is created. An architectural representation of a Kubernetes cluster is provided in Fig. 2.4.

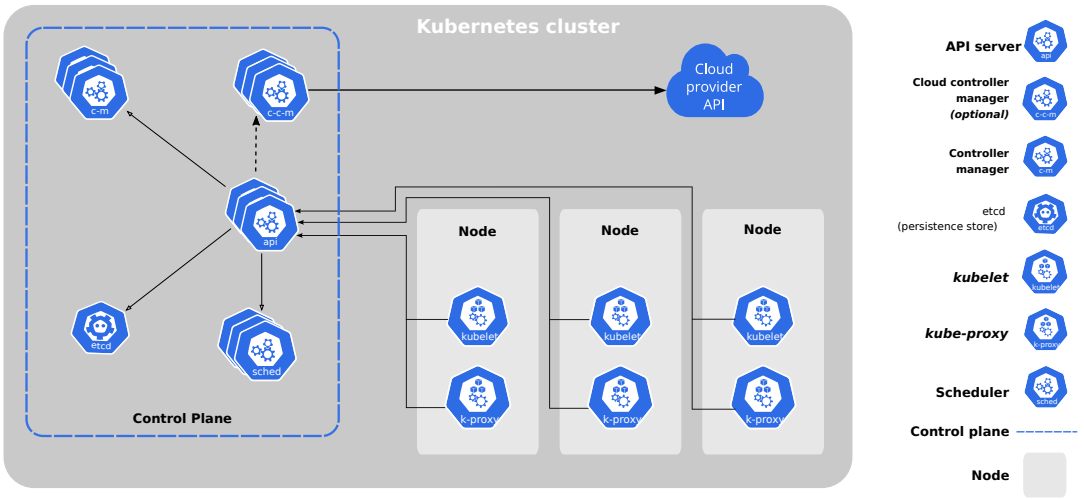

Figure 2.4: Architectural representation of a Kubernetes cluster

A k8s cluster is composed of a set of worker machines, called **nodes**, that run containerized applications. Every cluster has at least one worker node. A worker node hosts **pods**, that are sets of running containers in the cluster. The control plane is the logical component representing the container orchestration layer: it exposes the API and the interfaces to define, deploy, and manage the containers lifecycle. The control plane manages the worker nodes and the pods of the cluster. In production environments, it is deployed on multiple nodes in order to achieve fault-tolerance and high availability. For the same reason, the cluster is usually composed of multiple nodes.

In the following, a more detailed description of the control plane and nodes components is given.

2.4.1 Control plane components

Control plane components make global decisions about the cluster: for instance, scheduling, detecting and responding to cluster events (such as starting a new pod when a deployment `replicas` field is unsatisfied) are tasks handled by these components. Control plane components can be run on any machine in the cluster, but usually, dedicates nodes are used for this purpose: these nodes typically are not used for running user containers. The following sections describe each control plane component.

API server

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The main implementation of a Kubernetes API server is **kube-apiserver**, that is designed to scale horizontally (scale by deploying more instances). The Kubernetes API server validates and configures data for the API objects which include pods, services and others. It services REST operations and provides the frontend to the cluster's shared state through which all other components interact.

etcd

etcd is a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines. It gracefully handles leader elections during network partitions and can tolerate machine failure, even in the leader node [3]. etcd is used by Kubernetes as backing store for all cluster data. Only the API server can communicate with this component.

Scheduler

In Kubernetes, scheduling refers to making sure that a pod is matched to a node. The scheduler is the control plane component responsible of assigning the pods to nodes. The scheduler implementation provided by Kubernetes is called **kube-scheduler**. It is possible to use a custom scheduler implementation by adding it and indicating on the pod specification to use it. The scheduler makes decision based on singular and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

Controller manager

The controller manager is the control plane component responsible of running controller processes. A controller is a control loop that watches the shared state of the cluster through the API server (that in turn will read it from the etcd) and makes changes attempting to move the current state towards the desired state. **kube-controller-manager** is the default implementation provided by Kubernetes. 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. The following is a list of some types of controllers:

- **Node Controller** - responsible for noticing and responding when nodes go down

- **Job Controller** - watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion
- **Endpoints Controller** - populates the Endpoints object (that is, joins Services and Pods)
- **Service Account & Token Controller** - create default accounts and API access tokens for new namespaces

Cloud controller manager

The cloud controller manager is a controller plane component responsible of running controllers that embed cloud-specific control logic. The cloud controller manager lets link the cluster into a cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with the cluster. On an on-premises Kubernetes cluster, the cloud controller manager is not present. As in the case of the controller manager, the cloud controller manager combines different control loops in a single binary. The following are some types of controllers that could have cloud provider dependencies:

- **Node Controller** - for checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- **Route Controller** - for setting up routes in the underlying cloud infrastructure
- **Service Controller** - for creating, updating and deleting cloud provider load balancers

2.4.2 Nodes components

Nodes components are components running on each node of the Kubernetes cluster. In the following, a brief description of each component is provided.

Kubelet

Kubelet is the primary node agent that runs on each node of the Kubernetes cluster. It takes a set of PodSpecs (a PodSpec is a YAML or JSON object that describes a pod) that are provided through various mechanisms (primarily through the API server) and ensures that the containers described in those PodSpecs are running and healthy. A Kubelet agent waits for the Pod to be assigned to the node where it is deployed before starting the latter described process. Containers that are not created inside the Kubernetes context are not managed by Kubelet. Kubelet interacts with the container runtime in order to achieve its goal. The communication is based on gRPC.

kube-proxy

kube-proxy is a network proxy that runs on each node of the cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes: these network rules allow network communication to your Pods from network sessions inside or outside of your cluster. It can be replaced by other custom solutions.

Container runtime

The container runtime is the software that is responsible for running containers. Kubernetes supports container runtimes which are compliant with the Kubernetes Container Runtime Interface such as containerd and CRI-O.

Addons

Addons extend the functionality of Kubernetes. Some examples are DNS, dashboard (a web GUI), monitoring and logging addons.

2.5 Kubernetes objects

The following is an analysis of the Kubernetes objects and related concepts with a focus on the ones used extensively in this thesis work.

2.5.1 Overview

A Kubernetes object is a persistent entity in the Kubernetes system. Objects are used to describe the actual and the desired state of the cluster. Once a Kubernetes object is created, Kubernetes will work in order to ensure that the object exists. By creating an object it is possible to tell to Kubernetes which is the shape of the desired cluster's workload. In order to create, read, update and delete objects, it is necessary to interact with the API server through the exposed HTTP API: this can be done for example by using a command line interface such as **kubectl**. The exposed endpoint, used to access to an object, is called **resource**. The followings operations are allowed on a resource:

- **Create** - used for creating a resource. Once a resource is created, Kubernetes starts to move the cluster state towards the desired state for the new created resource
- **Read** - used for retrieving a resource. The read operation has 3 variants:
  - **Get** - used for retrieving a resource by name

  - **List** - used for retrieving a list of resource objects of a specific type within a specific namespace. For this variant, it is possible to filter the results by using a selector
  - **Watch** - used for starting a streaming session on which results for one or more objects are retrieved as updatings on the objects happen

- **Update** - used for updating a resource. This operation has 2 variants:
  - **Replace** - used for replacing the entire object spec with the provided one
  - **Patch** - used for applying changes to a specific field

- **Delete** - used for deleting a resource. Depending on the resource, child objects are automatically deleted contextually to the parent deletion.

2.5.2 Namespaces

Namespaces provides a mechanism for isolating groups of resources within a single cluster. By using Namespaces, it is possible to divide the cluster resources between multiple users or teams. Namespace scoping can be enabled only for namespaced object (e.g. Pods, Deployments, Services etc...): cluster-wide objects, like Nodes or PersistentVolumes, cannot be scoped inside a Namespace. By default, a Kubernetes cluster starts with four Namespaces:

- **default** - this Namespace is used as the default choice for namespaced objects with no other Namespace associated
- **kube-system** - this Namespace is usually reserved to objects created by the Kubernetes system or to objects linked to the cluster functioning
- **kube-public** - this Namespace is conceived for exposing cluster public information: for this reason, it is accessible from all the cluster users (even the non-authenticated ones)
- **kube-node-lease** - this Namespace is used for scoping Lease objects associated to the various cluster nodes; node leases allow kubelet to send heartbeats in order to allow the control plane to detect a node failure

2.5.3 Identifiers

Each object in the cluster is uniquely identified by a system-generated **UID**. An UID is a UUID: once an UID is assigned to an object in the cluster, it will not be reused for any future object in the entire cluster lifetime, even if the object associated to the UID is deleted.

Each object has also a **name**. Only one object of a given kind (for example Pod) can have a given name at a time. However, differently from the UID, if the object is deleted, it is possible to create an object of the same kind, inside the same Namespace, using the same name.

In order to group and organize subsets of objects, labels can be used. **Labels** are key-value pairs used to assign identifying attributes to objects: these attributes, can be meaningful and relevant from the users point-of-view, but they have not a direct semantic for the core system. Labels are directly attached to objects in the `metadata` field. The key part of a label must be unique for a given object. Labels are exploited by **Label Selectors**, the core grouping primitive in Kubernetes: by using Label Selectors, labels can be used to select objects and to find collections of objects that satisfy certain conditions.

In order to attach non-identifying information/metadata to objects, it is possible to use **Annotations**. Annotations can contains small or large, structured or unstructured metadata and are not limited in the number of characters that are allowed, as in the case of labels.

2.5.4 Status and Spec

The majority of the Kubernetes objects includes two nested object fields: the **status** and the **spec**. The `status` field contains information regarding the actual object state. The `spec` field contains information regarding the desired object state. The Kubernetes control plane continuously work in order to move the actual objects state to the desired one. Besides these information, Kubernetes object contains also metadata.

2.5.5 Pods

A **Pod** is the smallest deployable unit of computing in Kubernetes. It is composed by one or more containers, whose descriptions are co-located in the same Pod single object. Containers inside a single Pod share isolated storage and network resources: this context is realized by using a set of technologies like Linux namespaces or cgroups. Containers belonging to the same Pod are always deployed on the same node. A Pod is assigned to a node by the scheduler. Logically, a Pod models an application-specific "logical host": this means that application containers inside a single Pod have to be thought as applications executed on the same physical or virtual machine.

The listing in 2.1 is an example of a YAML manifest containing the description of a single-container Pod.

Listing 2.1: Pod YAML manifest example [4]

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

2.5.6 ReplicaSets

A **ReplicaSet** is the most basic Kubernetes abstraction describing the concept of a stable set of Pods running at any given time. The main fields composing a ReplicaSet description are the followings:

- the `selector` field, used for identifying the Pods owned by the ReplicaSet
- the `replicas` field, defining the number of replicas (or Pods) the ReplicaSet has to try to enforce by adding or deleting Pods
- the `template` field, containing the Pod template with the information that have to be used in order to create new Pods for meeting the desired number of replicas requirement.

Even if using a ReplicaSet allows to maintain a certain number of replica at any given time for a given Pod specification and updating this number in an easy way, very often, the higher-level Deployment concept is used in place of it.

The listing in 2.2 is an example of a YAML manifest containing the description of a ReplicaSet used for deploying 3 php-redis replicas.

Listing 2.2: ReplicaSet YAML manifest example [5]

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
```

```
14      metadata:
15        labels:
16          tier: frontend
17      spec:
18        containers:
19        - name: php-redis
20          image: gcr.io/google_samples/gb-frontend:v3
```

2.5.7 Deployments

A **Deployment** provides a declarative way to update ReplicaSets and Pods. Once a Deployment is created, an associated ReplicaSet is created in order to manage the required number of Pod replicas. The Deployment Controller moves the cluster state toward the desired Deployment state at a controlled rate: this is the reason why most of the applications are deployed inside Deployments and not directly using ReplicaSets. Like in the case of ReplicaSets, selector are used to identify the associated Pods through labels.

The listing in 2.3 is an example of a YAML manifest containing the description of a Deployment for deploying 3 nginx replicas.

Listing 2.3: Deployment YAML manifest example [6]

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    labels:
6      app: nginx
7  spec:
8    replicas: 3
9    selector:
10     matchLabels:
11       app: nginx
12   template:
13     metadata:
14       labels:
15         app: nginx
16     spec:
17       containers:
18       - name: nginx
19         image: nginx:1.14.2
20         ports:
21         - containerPort: 80
```

2.5.8 DaemonSets

A **DaemonSet** is an high-level abstraction used for ensuring that all the cluster nodes (or some of them, depending for example on taints and tolerations) runs a copy of the Pod template specified in the DaemonSet spec. If a node is removed from the cluster, the Pods associated to the DaemonSet are garbage collected. DaemonSets are typically used for running node daemons for storage, logs collection and monitoring.

The listing in 2.4 is an example of a YAML manifest containing the description of a DaemonSet for deploying on each node of the cluster a fluentd-elasticsearch replica.

Listing 2.4: DaemonSet YAML manifest example [7]

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

2.5.9 Nodes

A **Node** object is a Kubernetes logical representation of a cluster node. A Node object can be manually created by a user or by a kubelet on the associated node which self-registers to the control plane. Regardless the way in which a Node object is created, after the creation, the control plane is responsible for checking if it is valid: for example, the reachability and the healthiness of the node are checked (and also periodically re-verified). The object contains useful regarding the node configuration and its status; notable examples are:

- the `podCIDR` field in the `spec`, determining the addresses range from which the node Pods addresses must be extracted
- the `addresses` list contained in the `status` field, containing the hostname and addresses associated to the node interfaces

2.5.10 Services

A **Service** is an abstract way to expose an application running on a set of Pods as a network service. Pods are ephemeral by nature, and they can created or destroyed in any moment: for example, a Deployment can create or destroy Pods to meet the requirement on the desired number of replica. As the number of backend Pods exposing a particular network service may change during time, a mechanism for reaching the service from a frontend client has to be provided by the infrastructure: the Service abstraction is provided to solve this problem. Pods are associated to a Service through the label-selector mechanism. Different types of Service are available:

- **ClusterIP** - the Service is only accessible from within the cluster; it is suitable for the Pod-to-Service internal communications through the Service's associated virtual IP (called ClusterIP); this is the default type
- **NodePort** - besides a ClusterIP assigned for internal Pod-to-Service communications, a static port is assigned for enabling Service reachability from outside the cluster by contacting `<NodeIP>:<NodePort>`
- **LoadBalancer** - the Service is exposed using a cloud provider's load balancer; the external load balancer will route the traffic to associated NodePort and ClusterIP services, contextually created with the LoadBalancer Service
- **ExternalName** - the Service is mapped to an external one so that local apps can access it

The Service objects contains a list of one or more ports, each one representing a port exposed by the Service. If there is more than one port, it is mandatory to specify a name.

The listing in 2.5 is an example of a YAML manifest containing the description of a ClusterIP service which redirects requests coming from the port 80 to port 9376 of any Pod with the `app=MyApp` label.

Listing 2.5: Service YAML manifest example [8]

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

2.5.11 Endpoints

An **Endpoints** object is automatically created after the creation of a Service object on which a selector is specified. This object represents a collection of endpoints that implement the actual service. The object contains a list of `EndpointsSubsets`: each of them retains information about a set of addresses that share a common set of ports. The object contains also information about the readiness of the service backends.

2.6 Kubernetes operators

Operators are application-specific controllers that enable the extension of the Kubernetes automation infrastructure addressing application-specific problems. Operators exploit two core concept of Kubernetes: controllers and resources. Custom resources can be defined, and custom controllers can handle these resources by using the same mechanisms offered for the Kubernetes built-in resources.

2.7 The Kubernetes networking model

Kubernetes dictates some fundamental requirements that must be followed by any networking implementation. The basic assumption is that every Pod gets its own IP address: this creates a backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming,

service discovery, load balancing, application configuration, and migration [9]. This allocation strategy is called *IP-per-Pod model*. The requirements imposed to any networking implementation are the followings:

- 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

For those platforms that support Pods running in the host network, the following additional requirements is given:

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

Container inside the same Pod are always co-located on the same node and shares the same network namespace: this allows containers of the same Pod to communicate through localhost. Of course, this implies that processes inside the same Pod must coordinate the port usage. Kubernetes networking allows Pods to request to forward traffic reaching a specific host port to the Pods itself: this feature is not so much used.

The above considerations must be taken into account by each networking implementation during the handling of the following supported types of communications:

- **Highly-coupled container-to-container communications** - as already discussed, container belonging to the same Pods can communicate through the loopback interface of the network namespace to which they belong
- **Pod-to-Pod communications** - this involves both Pods running on the same node and Pods running on different nodes
- **Pod-to-Service communications**
- **External-to-Service communications**

The last two types of communication are highly related with the concept of Kubernetes Service.

2.8 The CNI specification

CNI (Container Network Interface), is a Cloud Native Computing Foundation project, consisting 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. Because of this focus, CNI has a wide range of support and the specification is simple to implement [10]. Kubernetes network plugins must adhere to the CNI specification: this choice is made in order to guarantee interoperability among all the network solutions. Once a Pod is created with an own network namespace, Kubernetes calls the CNI plugin in order to create the proper network interface in the network namespace and configure the networking for allowing incoming and outcoming communications. The container runtime is responsible of passing a set of arguments through environment variable and through the standard input to the CNI plugin, as prescribed by the specification.

2.9 eBPF (Extended Berkeley Packet Filter)

Extended Berkeley Packet Filter (eBPF) [11] represents the enhanced version of the Berkeley Packet Filter (BPF). It was proposed by Alexei Staravoitov in 2013 and it was introduced in version 3.18 of the Linux Kernel, making the original version, which is being referred to as "classic" BPF (cBPF) these days mostly obsolete. eBPF is a recent technology that enables flexible data processing thanks to the capability to inject new code in the Linux kernel at run-time, which is triggered each time a given event occurs. cBPF was born in 1992 and was a very simple VM used to perform in-kernel packet filtering (a famous example of application is tcpdump).

eBPF is very promising due to some characteristics that can hardly be found all together, such as the capability to execute code directly in the vanilla Linux kernel (hence without the necessity to install any additional kernel module), the possibility to compile and inject dynamically the code, the capability to support arbitrary service chains and the integration with the Linux eXpress Data Path (XDP) for early (and efficient) access to incoming network packets. At the same time, eBPF is known for some limitations such as limited program size, limited support for loops, and more, which may impair its capacity to create powerful networking programs.

In the following pages, some important features of eBPF are discussed.

2.9.1 Verifier

A very important concept of eBPF is safety. Since an eBPF program can be loaded at runtime in the kernel, it is checked by the verifier, which ensures that the given program cannot harm the system. The following is a list of constraints imposed by the verifier to the eBPF programs:

- the program must not have infinite loops

- the program must not use uninitialized variables or access out-of-bounds memory
- the program must be of a certain size that meets the system requirements
- the program must have a finite complexity. The verifier will evaluate all possible execution paths and must be able to complete the analysis within appropriate limits

2.9.2 Helper Functions

Technically, creating code that does complex operations with eBPF could be critical, since the C of the eBPF is limited. The solution that comes to the aid of developers are helpers. **Helpers** are native software functions in the Linux kernel that can be called within an eBPF program. While the eBPF code can be injected on demand, the helpers must be compiled apriori into the Linux kernel: this means that it is possible to use the helpers already listed in the Linux kernel. In case the already provided helpers are not enough, it is possible to define new ones and add them to the kernel. In brief, helper functions allow eBPF programs to consult a kernel-defined set of function calls to retrieve and send data to and from the kernel. The support functions available may differ for each type of BPF program.

2.9.3 Maps

Traditional BPF was stateless: eBPF overcomes this problem by providing mechanisms to save the state within a memory. The main purpose is to export data from the kernel to userspace, push data from userspace to the kernel or share data between different eBPF programs. The problem with having shared memory is concurrency: eBPF found a solution to this problem by defining a non-generic typed memory data structure called map. A **map** can assume a particular form (vector, hashmap, table etc...) and it is possible to have more than one simultaneously. If maps are used, the concurrency is directly managed by the system and programmers do not have to take care of it. Maps can also be nested, i.e. it is possible to have maps of various types of maps. In brief, maps are efficient key/value stores that reside in kernel space.

Maps can be accessed from a BPF program in order to keep state among multiple BPF program invocations. They can also be accessed through file descriptors from user space and can be arbitrarily shared with other BPF programs. BPF programs which share maps are not required to be of the same program type. The sharing mechanism is shown in Fig. 2.5.

There are different types of maps, which have behaviors and structures that distinguish them. The following is a list of generic maps:

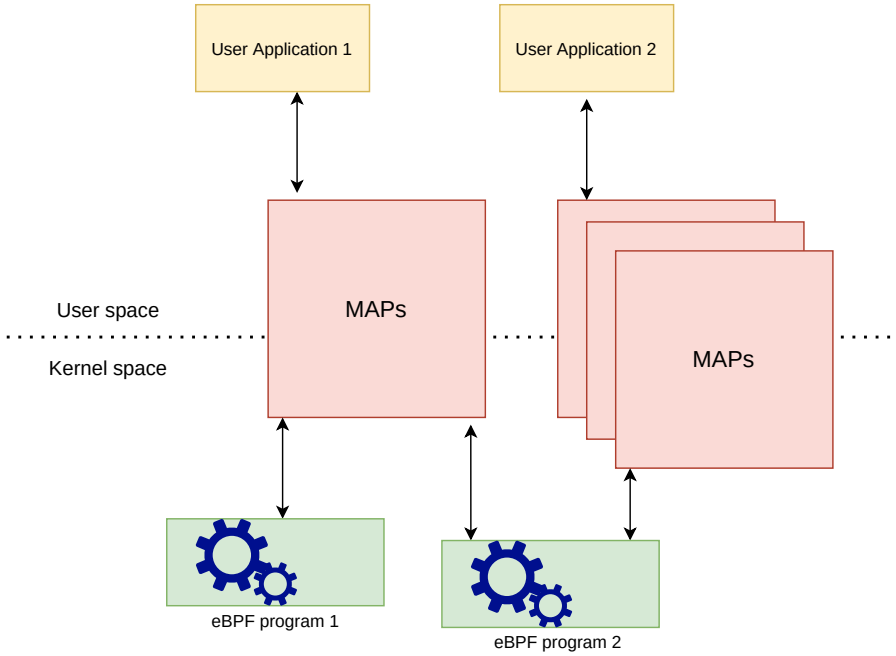

Figure 2.5: Shared memory architecture

- `BPF_MAP_TYPE_HASH`
- `BPF_MAP_TYPE_ARRAY`
- `BPF_MAP_TYPE_PERCPU_HASH`
- `BPF_MAP_TYPE_PERCPU_ARRAY`

There are also non-generic maps, some of which are:

- `BPF_MAP_TYPE_PROG_ARRAY`
- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`
- `BPF_MAP_TYPE_PERF_EVENT_ARRAY`
- `BPF_MAP_TYPE_STACK_TRACE`

Some maps also have a `PERCPU` version that allows to have different instances of the same table for each CPU core. The `PERCPU` versions are intended for performance improvement. No synchronization mechanism is needed and maps can also be cached for a further increase in access speed.

2.9.4 Tail calls

The concept of **tail call** can be used to overcome the size limit of an eBPF program. In practice, thanks to tail calls, an eBPF program can call another without going back to the old program: this is different from what happens usually with a normal function call, on which the program control returns to the caller after the function ending. A tail call has a minimal overhead and it is implemented as a long jump, reusing the same stack frame.

2.9.5 Program Types

There are different types of eBPF programs. The type of the eBPF program is defined by the specific type of events on which the program has to be triggered. Each type of event defines an **Hook Point**, which can be interpreted as a point of which an eBPF program can be attached. The number of available Hook Points is quite high. Hook Points are located at different levels in the Linux networking stack: an event can be captured as soon as it exits the card network, as soon as the Operating System comes into play, in sockets or at the RAW level (traditional BPF). The metadata associated with packets and the allowed actions change according to the hook used. For networking purposes, program execution starts when a packet arrives. Two of the main types for networking are **eXpress Data Path** (XDP) and **traffic control** (tc). A visualization of the XDP and tc Hook Points is shown in Fig. 2.6.

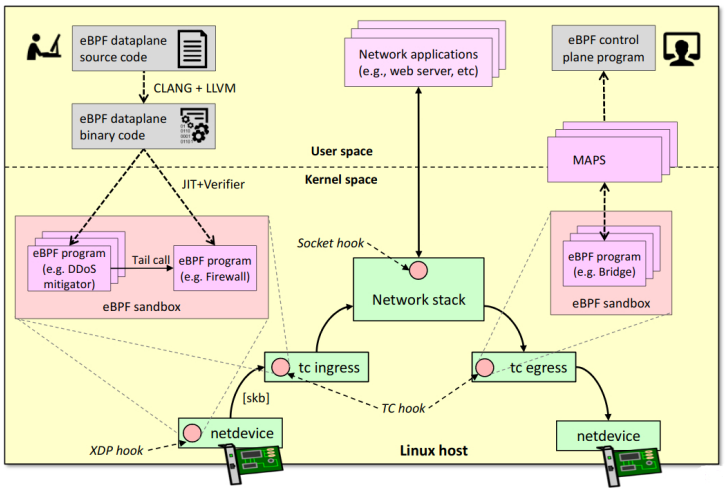

Figure 2.6: The XDP and tc hook points

XDP (eXpress Data Path)

XDP [12] provides a mechanism to run an eBPF program at the lowest possible level in the Linux networking stack. An eBPF program attached to the XDP Hook Point is triggered immediately after the reception of a packet. No expensive operations are performed at this level of the network stack: indeed, the program is triggered before any allocation of kernel metadata structures such as skb, spending fewer CPU cycles for packet processing than conventional stack delivery. On the other hand, the information provided to the program at this level are poor.

XDP has three operating modes:

- **Driver (or Native) mode** - the network card driver must support this model. Notice that, a device that runs in XDP Driver can redirect a packet only to another device running XDP Driver
- **Offloaded mode** - the BPF program is offloaded directly into the NIC instead of running on the host CPU. This is usually implemented in so-called SmartNICs
- **Generic (or SKB) mode** - in this case, XDP can also be used with drivers that do not offer native support. Note that, a device that runs in XDP Generic can redirect a packet to all the devices that run in XDP Generic

Possible use cases are: early packet discarding (for example for DDoS mitigation), firewalling, load balancing and forwarding. After the XDP program has been executed, an appropriate value must be returned:

- `XDP_DROP` - the packet is dropped directly at the driver level, without wasting any other resources
- `XDP_PASS` - the packet can keep going up the network stack. In this case, the CPU that is processing the packet allocates an skb, populates it and passes it forward to the GRO (Generic Receive Offloload) engine
- `XDP_TX` - the packet just received is modified and/or checked and then is sent back to the same NIC it came from. This type of action is used for example to make load balancer
- `XDP_REDIRECT` - the packet is redirected to another NIC
- `XDP_ABORTED` - indicates an exception; in this case the behavior is the same as `XDP_DROP` except that `XDP_ABORTED` passes `trace_xdp_exception` tracepoint which can be further monitored to detect incorrect behavior

A graphical representation of the possible XDP program actions is shown in Fig. 2.7.

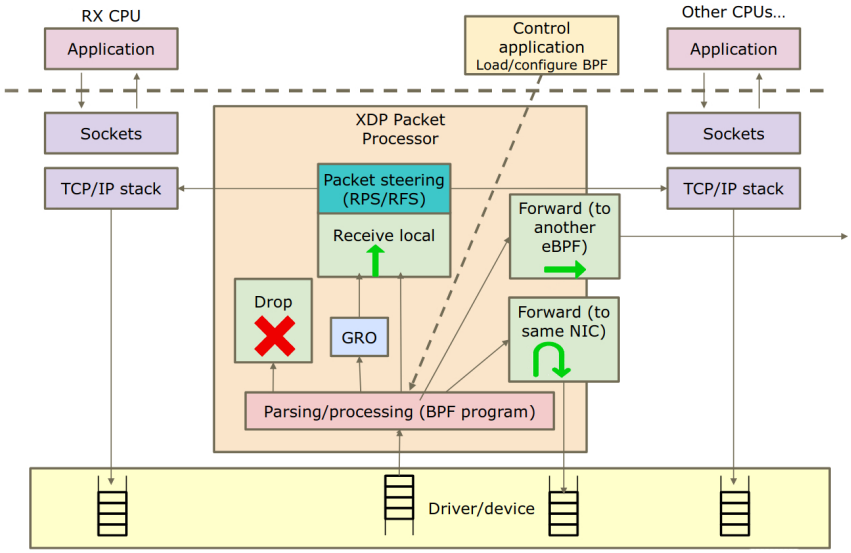

Figure 2.7: XDP program actions

Obviously, XDP has not only advantages; some limitations are the followings:

- with XDP it is possible to use only a limited number of helpers (compared for example to those that can be called with tc)
- XDP Driver mode limitations - nowadays, most XDP-enabled drivers use a specific memory model (e.g., one packet per page) to support XDP on their devices. Among the different actions allowed in XDP, there is the possibility to redirect the packet to another physical interface (`XDP_REDIRECT`). While this action is currently possible within the same driver, in our understanding, it is not possible between interfaces of different drivers
- Generic XDP limitations - in case XDP is used without driver support, the XDP program is executed immediately after the skb allocation and therefore in practice loses the advantages that come from the Driver mode. Although it must be said that it continues to perform better than tc

tc (traffic control)

tc programs intercept data when it reaches the traffic control function of the kernel, in RX or TX mode. Compared to XDP, a tc program has the following characteristics [11]:

- the BPF input context is a sk_buff, not a xdp_buff. When the kernel networking stack receives a packet, after the XDP layer, it allocates a buffer and parses the packet to store metadata about the packet: this representation is known as the sk_buff
- tc BPF programs can be triggered out of ingress and also egress points in the networking data path; XDP program can only be triggered on the ingress points
- tc BPF programs do not require any driver changes since they are run at Hook Points in generic layers in the networking stack: this means that they can be attached to any type of networking device

2.10 BCC

BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. Thanks to BCC it is possible to write eBPF programs in a much easier way: this is made possible thanks to the kernel instrumentation in C (which includes a C wrapper around LLVM) and frontends in Python and Lua. It is suitable for many tasks, including performance analysis and network traffic control.

2.11 Polycube

Polycube [13] is an open-source software framework, based on eBPF, that enables the creation of arbitrary and complex network function chains. A powerful in-kernel data plane and a versatile user-space control plane with strong isolation, persistence and composability characteristics can be used in every function. In addition, a common model for each network function control and management strategy simplifies the manageability and speeds up the implementation of new network services.

Polycube architecture main points are:

- Polycubed is a service independent daemon that allows to control the entire polycube service, starting from startup, through configuration and stopping all available network functions, i.e. services. This module mainly acts as a proxy, that is, it receives a request from its REST interface, forwards it to the appropriate service instance and responds to the user. Polycube supports both local services which are implemented as shared libraries, which are installed on the same server as polycubed and whose interaction occurs through direct calls, but also remote services which are implemented as remote daemons even running on a different machine, which communicate with polycubed via gRPC

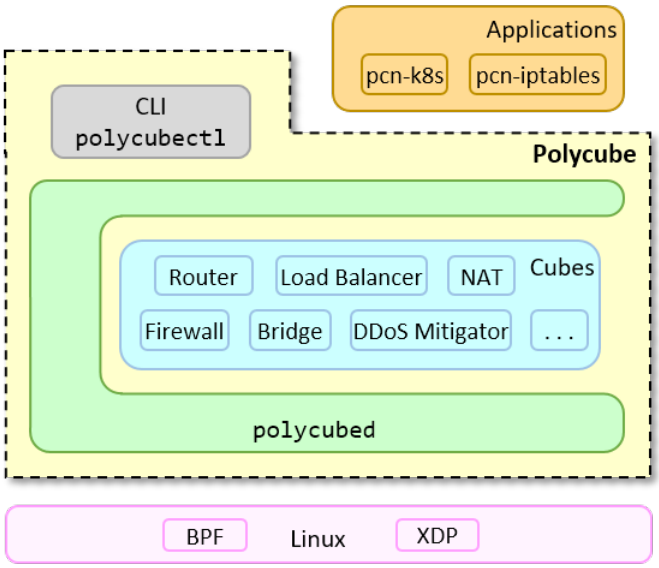

Figure 2.8: Polycube architecture

- A Polycube service can be seen as a plug-in that can be installed and started at runtime. Each service must implement a specific interface to be recognized and controlled by the polycubed daemon. Each network function is implemented as a separate module and multiple versions of the same function can coexist, this is because there may be cases where a simple and fast version is needed and therefore a reduced set of functions may suffice, but, there may be cases where the full but slower version is needed. Each implementation of the service includes the datapath (Data Plane), i.e. the eBPF code to be injected into the kernel, the control/management plane, which defines the primitives that allow you to configure the behavior of the service, and the slow path, which manages packets that cannot be fully processed in the kernel

- Polycubectl represents the command line interface (CLI) which is independent of the service and which allows to control the entire system, such as starting/stopping services, creating service instances and configuring/querying the extension of each individual service. This module cannot know a priori which service it will have to control and therefore its internal architecture is service-agnostic, i.e. it is able to interact with any service through a well-defined control/management interface that must be implemented by each service. To facilitate the programmer life in the creation of Polycube services, there is a set of tools for automatic code generation, capable of creating the skeleton of the control/management interface starting from the YANG (Yet Another Next Generation) model of the service itself

2.12 The VxLAN technology

Virtual Extensible LAN (VxLAN) is a network virtualization technology that attempts to address the scalability problems associated with large cloud computing deployments. It uses a VLAN-like encapsulation technique to encapsulate OSI layer 2 Ethernet frames within layer 4 UDP datagrams, using 4789 as the default IANA-assigned destination UDP port number. VxLAN endpoints, which terminate VxLAN tunnels and may be either virtual or physical switch ports, are known as **VxLAN tunnel endpoints** (VTEPs) [14]. VxLAN is an overlay network to carry Ethernet traffic over an existing (highly available and scalable) IP network while accommodating a very large number of tenants. It is defined in RFC7348 [15]. The structure of a VxLAN packet is shown in Fig. 2.9. With a 24-bit segment ID, aka **VxLAN Network Identifier** (VNI), VxLAN allows up to 2^{24} (16,777,216) virtual LANs, which is 4,096 times the VLAN capacity. The full Ethernet Frame (with the exception of the Frame Check Sequence: FCS) is carried as the payload of a UDP packet. VxLAN tunnel endpoints has two logical interfaces: an uplink and a downlink. The uplink is responsible for receiving VxLAN frames and acts as a tunnel endpoint with an IP address used for routing VxLAN encapsulated frames. These IP addresses are infrastructure addresses and are separated from the tenant IP addressing for the nodes using the VxLAN fabric. VxLAN frames are sent to the IP address assigned to the destination VTEP; this IP is placed in the Outer IP Destination Address. The IP of the VTEP sending the frame resides in the Outer IP Source Address. Packets received on the uplink are mapped from the VxLAN ID to a VLAN and the Ethernet frame payload is sent as an 802.1Q Ethernet frame on the downlink. During this process the inner MAC Source Address and VXLAN ID is learned in a local table. Packets received on the downlink are mapped to a VXLAN ID using the VLAN of the frame. A lookup is then performed within the VTEP L2 table using the VXLAN ID and destination MAC; this lookup provides the IP address of the destination VTEP. The frame is then encapsulated and sent out the uplink interface.

The diagram shown in 2.10 is used as a reference for illustrating the following example. A frame entering the downlink on VLAN 100 with a destination MAC of 11:11:11:11:11:11 will be encapsulated in a VxLAN packet with an outer destination address of 10.1.1.1. The outer source address will be the IP of this VTEP (not shown) and the VxLAN ID will be 1001. In a traditional L2 switch a behavior known as flood and learn is used for unknown destinations (i.e. a MAC not stored in the MAC table). This means that if there is a miss when looking up the MAC the frame is flooded out all ports except the one on which it was received. When a response is sent the MAC is then learned and written to the table. The next frame for the same MAC will not incur a miss because the table will reflect the port it exists on. VxLAN preserves this behavior over an IP network using IP

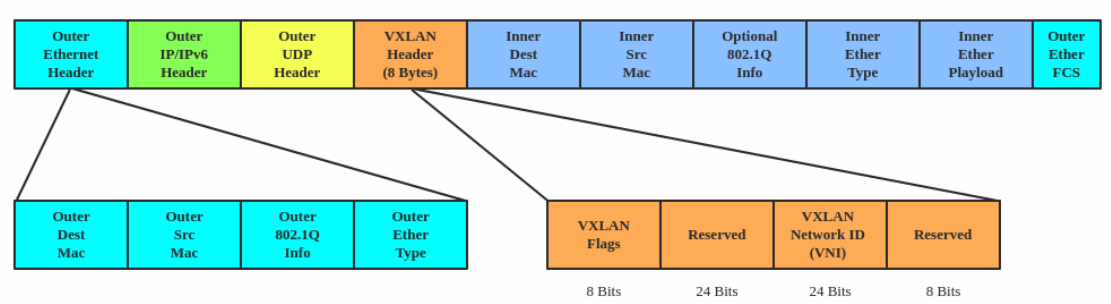

Figure 2.9: The VxLAN packet structure

multicast groups. Each VxLAN ID has an assigned IP multicast group to use for traffic flooding (the same multicast group can be shared across VXLAN IDs). When a frame is received on the downlink bound for an unknown destination, it is encapsulated using the IP of the assigned multicast group as the Outer DA; it's then sent out the uplink. Any VTEP with nodes on that VxLAN ID will have joined the multicast group and therefore receive the frame. This maintains the traditional Ethernet flood and learn behavior.

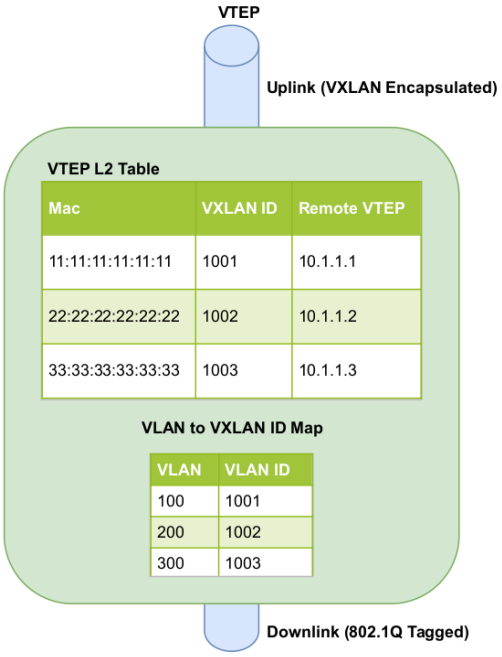

Figure 2.10: A VxLAN example

Chapter 3

Architecture

This chapter contains an in-depth explanation of the architecture of the network provider. First, a description of the base architecture virtual network topology proposed in [1] is given: each topology component is investigated and some changes are proposed in order to overcome some efficiency and scalability problems or to add new features. Lastly, a description of the CNI plugin, the overlay networking and the node agent is provided.

3.1 Base architecture virtual network topology

The base architecture virtual network topology is shown in 3.1. It exploits three standard Polycube services (`pcn-simplebridge`, `pcn-router` and `pcn-lbrp`) and one custom Polycube service (the `pcn-k8sdispatcher`) that is built on purpose for the work. In the following a description of each of these services is provided.

3.1.1 pcn-simplebridge

The **pcn-simplebridge** service implements a fast and simple Ethernet bridge. It is used in order to allow communication between the Pods on the same node and in order to give them access to their default gateway.

3.1.2 pcn-loadbalancer-rp (pcn-lbrp)

The **pcn-loadbalancer-rp** service (or simply `pcn-lbrp`) implements a Reverse Proxy Load Balancer. Its design is inspired by Maglev, the Google Load Balancer [16]. The number of exported network interfaces is limited to 2: a frontend port and a backend port. The frontend port is facing towards the client that want to contact a virtual service, whereas the backend port is facing towards the virtual service

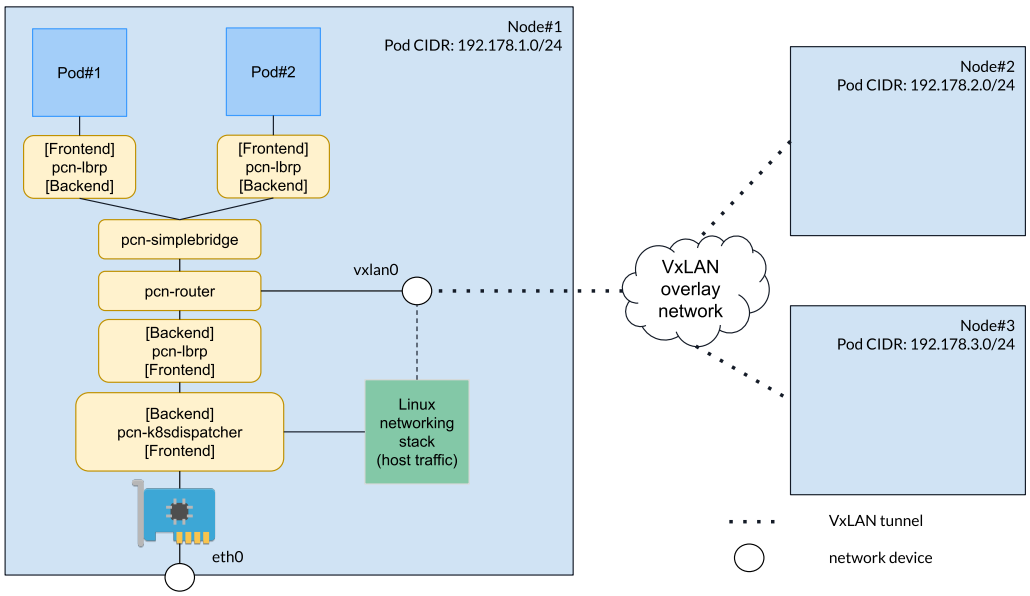

Figure 3.1: The base architecture virtual network topology

backends. If a packet is directed to a virtual service, it is hashed to determine which is the correct backend; the hashing function guarantees that all packets belonging to the same TCP/UDP session will always be terminated to the same backend server. A single load balancer is placed between the router and the k8sdispatcher in order to allow external clients to reach NodePort Services. Furthermore, a load balancer is placed between each Pod and the simplebridge: this is done in order to allow Pods to reach Service's backends Pods through the Service's Cluster IP.

3.1.3 pcn-router

The **pcn-router** service implements an eBPF router and so, it is responsible of performing routing and forwarding of packets. It only supports IPv4 and static routing. The router connects to other elements with a set of ports, each one identified by a name and configured with a primary IP address, a list of secondary IP addresses and a MAC address. The static routing table can be configured with a set of routes, specifying destination network CIDR (address and prefix length), next hop and path cost. This service can be attached also to physical or virtual interfaces. If the port is connected to a network interface of the host, MAC and IP addresses of the router port has to be the same of the host interface. The router is the default gateway for Pods in the same node: it connects Pods to the external world and to other Pods, enabling Pod to Pod communication across nodes thanks

to VxLAN [15] tunnel.

3.1.4 pcn-k8sdispatcher

The **pcn-k8sdispatcher** service was originally conceived and developed in the work [1]. The service provides an eBPF implementation of a custom NAT: it performs different actions depending on the type and on the direction of the traffic.

For Egress Traffic, the flow chart in 3.2 can be used to explain the functioning of the service.

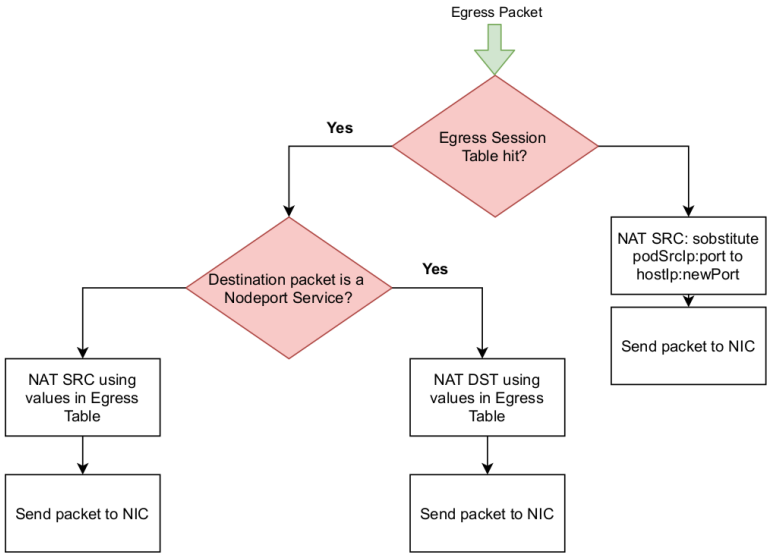

Figure 3.2: pcn-k8sdispatcher egress traffic flow chart

The Egress Traffic is the traffic generated by Pods and directed to the external world. This traffic can be generated by an internal Pod that want to contact the external world or as a response to an external world request. For this traffic, the service maintains an egress session table containing information about the active egress sessions. If a Pod want to contact the external world, no active egress session will be present in the table: in this scenario, the service performs SNAT, replacing the address of the Pod with the address of the node, and adds an entry to the egress session table. If the outgoing traffic is generated as a response to an external request, an egress session table hit will happen: in case of traffic generated as a response for a request to a NodePort Service, DNAT is performed; otherwise SNAT is performed. In both the latter two cases, NAT is performed using the information retrieved from the egress session table.

For Ingress Traffic, the flow chart in 3.3 can be used to explain the functioning of the service.

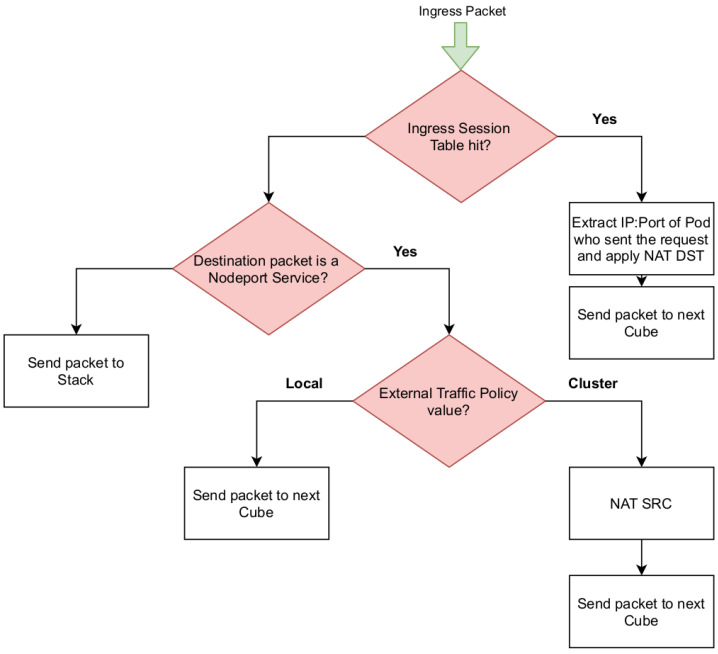

Figure 3.3: pcn-k8sdispatcher ingress traffic flow chart

The Ingress Traffic can be differentiated in traffic directed to the host (either directly or because it needs VxLAN processing) and traffic directed to Pods. The traffic directed to Pods can be the traffic generated by an external host trying to contact a NodePort service or the return traffic generated by an external host in order to provide a response to an internal Pod request. The service uses an ingress session table containing all the active ingress sessions. If a session table hit happens, the service performs DNAT by replacing `<NodeIP>:<Port>` of the packet with `<PodIP>:<PodPort>` extracted from the ingress session table entry. If no session table are associated with the incoming packet, the service try to determine if a NodePort rule matches the packet characteristics. In case of no NodePort rule matching, the packet is sent to the Linux stack for further processing. In case of NodePort rule matching, different actions are applied according to the `ExternalTrafficPolicy` of the Kubernetes NodePort Service associated to the rule. If the policy is `LOCAL`, the traffic is allowed to reach only backend Pods located on the current node: in this case the packet can proceed towards the Pod without modifications. In case the policy is `CLUSTER`, the packet can also reach backend Pods located on other nodes: since later in the chain the packet will be

processed by a load balancer and the return packet will have to transit through the same load balancer, SNAT is applied by replacing the source IP address with a specific reserved address belonging to the Pod CIDR of the node on which the k8sdispatcher is deployed. In this way the two nodes (the one that receives the request and the one running the selected backend Pod) will exchange the packets of the flow over the VxLAN interconnect. In this case, a corresponding session entry is stored into the ingress sessions table.

3.2 The new virtual network topology

The base architecture virtual network topology allows a network provider built on top of it to be partially compliant to the Kubernetes networking dictates: indeed, the architecture does not handle the communication between an host process and a Pod deployed on the same node. Moreover, the architecture suffers of efficiency and scalability problems due to the following two reasons:

- a new load balancer must be created for each Pod and each time the relative eBPF code must be loaded in the Linux kernel
- each Kubernetes Service or related Endpoints variations must be reflected upon each load balancer

In order to deal with the above problems, the new topology shown on Fig. 3.4 has been conceived. First, a new port on the router is created and connected to a virtual Ethernet pair for enabling communication between Pods and host processes running on the same node: this will be called **polykube veth pair** in the following discussions. Finally, the simplebridge and the per-Pod load balancers are removed and substituted by the instance of a new custom service, called `pcn-k8slbrp`: this new component overcomes the interfaces number limitation of the `pcn-lbrp` service. In the following, a description of the `pcn-k8slbrp` service is provided.

3.2.1 pcn-k8slbrp

The **pcn-k8slbrp** service is developed specifically for this project. It provides an implementation of a Reverse Proxy Load Balancer. It is inspired by the functioning of the already existing `pcn-lbrp` service but extends its architecture in order to increase the number of supported ports. The purpose of this service is to perform the necessary translation of the Kubernetes Service's associated IP and port with the ones of a real Service's backend Pod, according to the load balancing logic. A k8slbrp port can be a `FRONTEND` or a `BACKEND` port. Depending of the port mode, the k8slbrp can support one or more frontend ports. Two port modes are supported: `SINGLE` and `MULTI`. In `SINGLE` port mode, the load balancer supports only a single

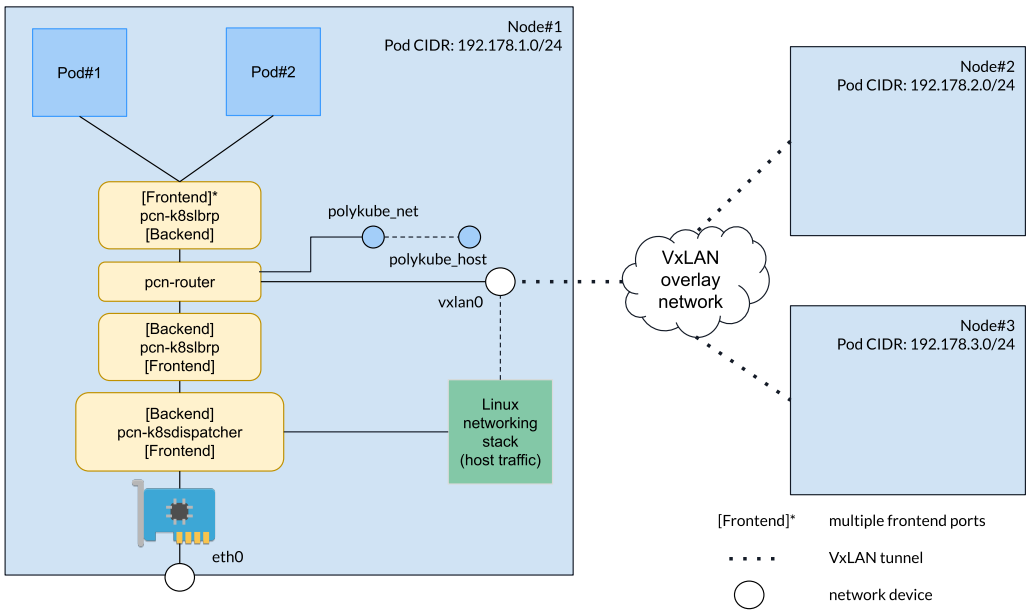

Figure 3.4: The new architecture virtual network topology

`FRONTEND` port; in `MULTI` port mode, multiple `FRONTEND` ports are supported. In both cases, only a `BACKEND` port is supported. Regardless the port mode, if a packet coming from a `FRONTEND` port is directed to a Service, DNAT is performed on it; the corresponding SNAT is performed for packets coming from backends and on the way back to clients. Like in the case of the `pcn-lbrp` service, packet are hashed to determine which is the correct backend; all packets belonging to the same TCP/UDP session will always be terminated to the same backend server. The translation performed is from `<vip>:<vport>` to `<realip>:<realport>`.

The topology is made up 2 different `pcn-k8slbrp` cubes: the **internal k8slbrp** and the **external k8slbrp**. The internal k8slbrp cube is configured in `MULTI` port mode and is placed between the Pods and the router: it allows to connect Pods to the network topology and enables Kubernetes ClusterIP Services communications. The external k8slbrp is configured in `SINGLE` port mode and is placed between the router and the k8sdispatcher: like the lbrp present in the base architecture, it enables external client to reach NodePort Services exposed by internal backend Pods.

3.3 The CNI plugin

In order to handle the attachment of the Pods to the network topology, a CNI plugin has to be conceived. The CNI plugin is invoked by the container runtime each time

a Pod is created, destroyed or checked against connectivity reachability. Each time a Pod is created, the CNI plugin must create a virtual Ethernet pair connecting the Pod to the internal k8slbrp and configure the Pod's network namespace in order to allow processes running inside the Pod to reach the Pods Default Gateway. Each time a Pod is destroyed, the corresponding network interfaces and the connection to the internal k8slbrp must be destroyed as well. The convention used by the CNI has to be carefully evaluated, since they could be exploited by other network provider components.

3.4 The overlay network

For communications involving different nodes, an overlay networking solution is chosen: this allows to implement the virtual networking without relying on a specific network technology which may not be given for a specific cluster deployment. The VxLAN encapsulation protocol is used without requiring multicast support or the needing of physical switches that actively partecipate in IGMP snooping: this independence is achieved by configuring in a proper way the Linux bridge forwarding database.

3.5 The node agent

The network provider must handle all the tasks related to the networking of the cluster. Since the virtual network topology of each node has to be deployed and kept in sync with the cluster status, an agent on each node is needed.

The agent is responsible of ensuring the network topology on the node on which is deployed and ensuring the intra- and the inter-node connectivity: this means creating, connecting and checking the configuration of each Polycube cube and creating and checking the VxLAN inter-node connectivity. The agent must ensure the connectivity between host processes and Pods running on the same node: this means ensuring the presence and the configuration of the virtual Ethernet pair on the node and the right configuration on the host routing table for allowing host processes to contact Pods; moreover, also the topology router routing table must be correctly configured for allowing Pods to contact host processes on the same node.

The agent need to be able to recover from failure of various kind: this involves failure in the network topology or failure in the VxLAN interconnection. The agent has to receipt the changes on the Kubernetes networking infrastructure through the Kubernetes control plane (specifically, from the API server), such as new node added to the cluster or new created services, and propagate them on the network topology. The following is a description of the resources that must be watched:

- **nodes** - each time a Node object is created, the agent must configure the topology router and the host Linux bridge forwarding database for enabling inter-node communications through the VxLAN interface
- **services** - each time a Service object is created, the internal and the external k8slbrp and also the k8sdispatcher has to be configured properly according to the Service type
- **endpoints** - the information contained in the Endpoints object associated to a Service object are useful to determine the backends Pods associated to the Service; these information are used for configuring the internal and the external k8slbrp for enabling service resolution

The agent running on each node communicates with the Kubernetes API server through its exposed REST API. In order to handle the network topology, the agent interact with the Polycube daemon through the REST API: this daemon, like the agent, must be present on each node of the cluster. The communication between the agent and the Polycube daemon is realized through the loopback interface of the node. The image in 3.5 shows the final representation of the architecture of the network provider.

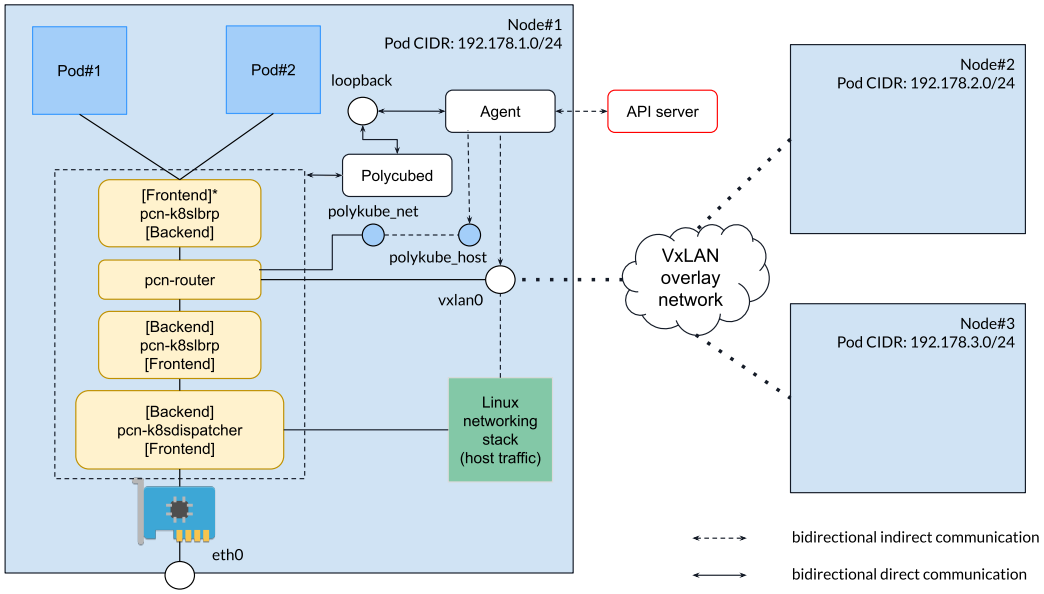

Figure 3.5: Network provider architecture

Chapter 4

Implementation

This chapter explains how concepts shown in the Chap. 3 have been implemented in details, which programming languages and which workarounds have been used in order to solve the problems encountered. The `pcn-router` is a standard service, already available in the Polycube framework. The `pcn-k8slbrp` service is built specifically for the project. The `pcn-k8sdispatcher` service code has been cleaned up and optimized in order to efficiently work in couple with the agent. A custom CNI plugin has been implemented as well as an operator for watching Kubernetes resources and reflecting their changes on the network topology. C++ has been used for implementing the `pcn-k8slbrp` control plane and revising the `pcn-k8sdispatcher` one. C has been used for working with the services data plane implementations. The YANG data modelling is used to describe the services structure. The CNI plugin and the operator have been implemented using Golang.

4.1 Automatic code generation

Polycube provides an automatic code generation utility, called `polycube-codegen`, that can be used to generate a stub from a YANG datamodel. It is composed by `pyang` and `swagger-codegen`: `pyang` parses the YANG datamodel and creates an intermediate JSON that is compliant with the OpenAPI specifications [17]; `swagger-codegen` starts from the JSON file produced by `pyang` and creates a C++ code skeleton that actually implements the service. Among the services that are automatically provided, the C++ skeleton handles the case in which a complex object is requested, such as the entire configuration of the service, which is returned by disaggregating the big request into a set of smaller requests for each leaf object. Hence, this leaves to the programmer only the responsibility to implement the interaction with leaf objects. The following is a description of the structure generated by `polycube-codegen`:

- **src/base/{resource-name}Base.[h,cpp]** - one base class is generated for every resource defined in the data-model (including the service itself), this classes define the interface that must be implemented to be compliant with the management API
- **src/api/{service-name}Api.[h,cpp]** and **src/api/{service-name}ApiImpl.[h,cpp]** - implements the shared library entry points to handle the different request for the rest API endpoints. The developers does not have to modify it
- **src/serializer/** - this folder contains one JSON object class for each object used in the control API. These classes are used to performs the marshalling/unmarshalling operations
- **src/{resource-name}.[h,cpp]** - these classes implement the corresponding interface of the base directory, they provide a standard implementation for some of the methods, while others must be written by the programmer to define the actual behaviour of the service
- **src/{service-name}-lib.c** - this is used to compile the service as shared library
- **src/{service-name}_dp.c** - this contains the fast path code for the service
- **.swagger-codegen-ignore** - this file is used to prevent files from being overwritten by the generator if a re-generation is performed

Each Polycube service must follow a specific convention for generating the REST APIs: this is needed in order to interact with them by using polycubectl or other REST clients specifically built for the purpose. This convention is exploited by `polycube-codegen`, which supports the generation of client stubs in different programming languages. Since both the CNI plugin and the operator need to interact with the services control plane, this features is used in order to generate client libraries for the `pcn-router`, the `pcn-k8slbrp` and the `pcn-k8sdispatcher`.

4.2 pcn-k8slbrp

The **pcn-k8slbrp** service is designed in order overcome the limitations of the standard `pcn-loadbalancer-rp` service, already provided in the Polycube framework. Specifically, this standard load balancer implementation only supports two ports: a frontend and backend port. First, starting from the `pcn-loadbalancer-rp` implementation, an effort has been made in order to support the creation of more than one frontend port. Once the multiple frontend ports feature has been implemented,

the port mode concept has been introduced: this allow to fallback to the default `pcn-loadbalancer-rp` implementation in case no more than one frontend port is needed (like in the case of the external load balancer): this lets understand that the port mode feature has been introduced only for optimization reasons.

4.2.1 Data Model

Traffic coming from the unique backend port must be redirected to the proper frontend port. In order to select the right port, it is enough to associate to each frontend port the IP address of the connected Pod: in this way, the packet can be redirected to the port whose associated IP is the destination IP of the packet. The IP address is associated during the port creation. The `pcn-k8slbrp` data model snippet in 4.1 reflects this choice.

Listing 4.1: pcn-k8slbrp data model - ports definition

```
1  uses "polycube-standard-base:standard-base-yang-module" {
2    augment ports {
3      leaf type {
4        type enumeration {
5          enum FRONTEND { description "Port connected to the clients
       "; }
6          enum BACKEND { description "Port connected to the backend
       servers"; }
7        }
8        mandatory true;
9        description "Type of the LB port (e.g. FRONTEND or BACKEND)";
10     }
11     leaf ip {
12       type inet:ipv4-address;
13       description "IP address of the client interface (only for
       FRONTEND port)";
14       polycube-base:cli-example "10.10.1.1";
15     }
16   }
17 }
```

The port mode is simple to implement. If the `pcn-k8slbrp` cube is created in `SINGLE` port mode, an IP address is not needed on the single allowed frontend port. The port mode is introduced in the data model in the way shown in 4.2.

Listing 4.2: pcn-k8slbrp data model - port mode definition

```
1  leaf port_mode {
2    type enumeration {
3      enum SINGLE;
4      enum MULTI;
5    }
```

```
6       default MULTI;
7       description "K8s lbrp mode of operation. 'MULTI' allows to manage
         multiple FRONTEND port. 'SINGLE' is optimized for working with a
        single FRONTEND port";
8     }
```

The YANG data model, as shown in 4.3 presents two other important concepts: **services** and **backends**.

Listing 4.3: pcn-k8slbrp data model - services and backends definition

```
1   list service {
2     key "vip vport proto";
3     description "Services (i.e., virtual ip:protocol:port) exported
      to the client";
4     leaf name {
5       type string;
6       description "Service name related to the backend server of the
      pool is connected to";
7       polycube-base:cli-example "Service-nigx";
8     }
9
10    leaf vip {
11      type inet:ipv4-address;
12      description "Virtual IP (vip) of the service where clients
      connect to";
13      polycube-base:cli-example "130.192.100.12";
14    }
15
16    leaf vport {
17      type inet:port-number;
18      description "Port of the virtual server where clients connect
      to (this value is ignored in case of ICMP)";
19      polycube-base:cli-example "80";
20    }
21
22    leaf proto {
23      type enumeration {
24          enum ICMP;
25          enum TCP;
26          enum UDP;
27          enum ALL;
28      }
29      mandatory true;
30      description "Upper-layer protocol associated with a
      loadbalancing service instance. 'ALL' creates an entry for all the
       supported protocols";
31    }
32
33    list backend {
```

```
34        key "ip";
35        description "Pool of backend servers that actually serve
    requests";
36        leaf name {
37          type string;
38          description "name";
39          polycube-base:cli-example "backend1";
40        }
41
42        leaf ip {
43          type inet:ipv4-address;
44          description "IP address of the backend server of the pool";
45          polycube-base:cli-example "10.244.1.23";
46        }
47
48        leaf port {
49          type inet:port-number;
50          description "Port where the server listen to (this value is
    ignored in case of ICMP)";
51          mandatory true;
52          polycube-base:cli-example "80";
53        }
54
55        leaf weight {
56          type uint16;
57          description "Weight of the backend in the pool";
58          polycube-base:cli-example "1";
59        }
60      }
61    }
```

A **service** is identified by the triple `<vip>:<vport>:<proto>`. Each service represents a virtual service exposed by a certain number of backends Pod. The supported protocol types are TCP, UDP and ICMP. The ALL protocol type is a shortcut for the latter three types. It is possible to associate to each service also a name: this doesn't need to be unique among services, and so can be used to easily group them (more details will be given in later sections).

Each service have a list of **backends**. Given a service, a backend is identified by its IP. The port number is mandatory. The backend name doesn't need to be unique among the service backends and can be used to perform backends grouping. Lastly, a weight can be associated to each backend: its purpose is to determine how the traffic has to be splitted among the service backends. The default weight value is 1.

4.2.2 Data plane

Traffic coming from a backend port is only allowed to go to a frontend port. If the load balancer is configured in `MULTI` port mode, the proper frontend port is chosen by using the destination IP of the packet as a key of an eBPF hash map called `ip_to_frontend_port`: if an entry is found in the map, the obtained frontend port number is used to forward the packet; if an entry is not found, the packet is dropped.

Traffic coming from a frontend port is only allowed to go to the backend port. This solution simplifies the implementation but introduces the following problem: if a Pod want to reach another Pod inside the same Pod CIDR, it will try to obtain its MAC address through an ARP request; since the internal load balancer is conceived to be connected through its backend port to the router, each of these ARP request will be received by the router and will be discarded. The solution to the problem requires to configure a /32 address on each Pod: in this way, for all the traffic types, each Pod will try to reach all the other destinations through its default gateway.

If the packet is not an IPv4 packet and comes from a backend port, this is flooded to all the frontend port. If the packet comes from a frontend port and is an ARP reply, the original ARP reply is sent to the backend port and a second ARP reply, using the virtual source IP associated to the src IP rewriting feature of the load balancer, will be sent as well to the backend port. This latter feature is inherited by the `pcn-loadbalancer-rp` implementation.

4.3 pcn-k8sdispatcher

The **pcn-k8sdispatcher** is a service developed in order to allow the internal virtual topology to communicate with the external world: for example, it is responsible to perform NAT during Pod to Internet communications. The service, already designed in [1] and implemented in [18], has been cleaned up and optimized for working together with the operator.

4.3.1 Data Model

Kubernetes associates to the NodePort Services a port range: this is used in order to discriminate packets directed to the network stack of the node from packets directed to the services. This port range is customizable and the k8sdispatcher must be aware of it in the datapath. Also, the k8sdispatcher must be aware of the IP address configured on the physical interface connected to the node: this is the IP address associated to the Kubernetes node and together with the port range allows to identify a packet eligible to be checked against the registered NodePort

services. This two information (the NodePort range and the IP address associated to the frontend port) are defined in the data model as shown in 4.4.

Listing 4.4: pcn-k8sdispatcher data model - ports and NodePort range definition

```
1     uses "polycube-standard-base:standard-base-yang-module" {
2         augment ports {
3             leaf type {
4                 type enumeration {
5                     enum BACKEND { description "Port connected to cni
      topology"; }
6                     enum FRONTEND { description "Port connected to
    NIC"; }
7                 }
8                 mandatory true;
9                 description "Type of the k8sdispatcher port (e.g.
    BACKEND or FRONTEND)";
10            }
11            leaf ip {
12                type inet:ipv4-address;
13                description "IP address of the node interface (only
    for FRONTEND port)";
14                polycube-base:cli-example "10.10.1.1";
15            }
16        }
17    }
18
19    leaf nodeport-range {
20        type string;
21        description "Port range used for NodePort services";
22        default "30000-32767";
23        polycube-base:cli-example "30000-32767";
24    }
```

It is not enough for an incoming packet to have destination address equal to the one configured on the external interface and destination port in the NodePort range: in order to not be sent to the host stack, the packet must match one of the NodePort rules registered on the `pcn-k8sdispatcher` instance. A NodePort rule is identified by a port and a protocol type, as shown in 4.5.

Listing 4.5: pcn-k8sdispatcher data model - NodePort rule

```
1     list nodeport-rule {
2         key "nodeport-port proto";
3
4         leaf nodeport-name {
5             type string;
6             description "An optional name for the NodePort rule";
7         }
8         leaf nodeport-port {
```

```
9             type inet:port-number;
10            description "Destination L4 port number";
11        }
12        leaf proto {
13            type string;
14            description "L4 protocol";
15        }
16        leaf service-type {
17            type enumeration {
18                enum CLUSTER { description "Cluster wide service"; }
19                enum LOCAL { description "Local service"; }
20            }
21            mandatory true;
22            description "Denotes if this Service desires to route
    external traffic to node-local or cluster-wide endpoint";
23        }
24    }
```

A NodePort rule is associated one-to-one with a NodePort Service's `ServicePort` configured on the cluster. In order to easily identify all the NodePort rules associated to a NodePort Service, a name can be configured on it. The `service-type` stores the information of the `externalTrafficPolicy` of the NodePort service: if it is set to `CLUSTER`, then SNAT is performed before sending the packets to the next cube. The `internal-src-ip` address, whose definition is shown in 4.6, is used to perform the aforementioned NAT operation.

Listing 4.6: pcn-k8sdispatcher data model - Internal Source IP

```
1     leaf internal-src-ip {
2         type inet:ipv4-address;
3         description "Internal src ip used for services with
    externalTrafficPolicy=CLUSTER";
4         polycube-base:cli-example "10.10.1.1";
5         mandatory true;
6     }
```

In order to store the information about the ingress and the egress sessions, the concept of Natting rule is defined in the data model. A Natting rule is identified by the quintuple (IPSrc, IPDst, PortSrc, PortDst, Proto). The data model snippet is shown in 4.7.

Listing 4.7: pcn-k8sdispatcher data model - Natting rule

```
1     list natting-rule {
2         key "internal-src internal-dst internal-sport internal-dport
    proto";
3
4         leaf internal-src {
5             type inet:ipv4-address;
```

```
6               description "Source IP address";
7           }
8           leaf internal-dst {
9               type inet:ipv4-address;
10              description "Destination IP address";
11          }
12          leaf internal-sport {
13              type inet:port-number;
14              description "Source L4 port number";
15          }
16          leaf internal-dport {
17              type inet:port-number;
18              description "Destination L4 port number";
19          }
20          leaf proto {
21              type string;
22              description "L4 protocol";
23          }
24          leaf external-ip {
25              type inet:ipv4-address;
26              description "Translated IP address";
27          }
28          leaf external-port {
29              type inet:port-number;
30              description "Translated L4 port number";
31          }
32      }
```

4.3.2 Data plane

The service dataplane has been cleaned up from the already existing implementation. Moreover, a better ARP handling has been implemented. If a packet coming from the backend port is an ARP request, it is allowed to go to the frontend port: this is needed in order to allow the Polycube router to send ARP request packets for inferring the cluster nodes and the cluster Default gateway MAC addresses. ARP packets coming from the backend port other than requests are dropped. If a packet coming from the frontend port is an ARP reply, it is duplicated: a copy is sent to the Linux host network stack and a copy is sent to the next cube. If a packet coming from the frontend port is an ARP request, it is sent to the stack. ARP packets coming from the frontend port other than requests or reply are dropped.

4.3.3 Control plane

NodePort rules can be created, updated or destroyed by performing specific REST calls to the Polycube daemon; Natting rules instead, can be only retrieved for

debugging purposes since only the data plane code can add them.

4.4 The CNI plugin

Network plugins in Kubernetes come in a few flavors: **CNI plugins** and the **Kubenet plugin**. CNI plugins must adhere to the Container Network Interface (CNI) specification, a specification of the Cloud Native Computing Foundation (CNCF). The kubelet has a single default network plugin, and a default network common to the entire cluster. It probes for plugins when it starts up, remembers what it finds, and executes the selected plugin at appropriate times in the Pod lifecycle (this is only true for Docker, as CRI manages its own CNI plugins).

In order to instruct Kubernetes to use a CNI plugin, it is possible to pass to the kubelet the `--network-plugin=cni` command-line option. Kubelet reads a file from `--cni-conf-dir` (default `/etc/cni/net.d`) and uses the CNI configuration from that file to set up each Pod's network. If there are multiple CNI configuration files in the directory, the kubelet uses the configuration file that comes first by name in lexicographic order. A CNI configuration file must match the CNI specification, and any required CNI plugins referenced by the configuration must be present in `--cni-bin-dir` (default `/opt/cni/bin`). In addition to the CNI plugin specified by the configuration file, Kubernetes requires the standard CNI `lo` plugin, at minimum version 0.2.0. Kubernetes follows the v0.4.0 release of the CNI specification. For this project, also the `host-local` IPAM plugin is required in order to manage the assignment of the IPv4 addresses to the Pods deployed on a specific node.

4.4.1 The specification requirements

The CNI specification prescribes the CNI plugin to implement the following operations for the following purposes:

- `ADD` - add a container to network or apply modifications; if the plugin is invoked requiring an `ADD` operation, the plugin should either create the specified interface inside the specified container namespace or adjust its configuration

- `DELETE` - remove a container from network or un-apply modifications; if the plugin is invoked requiring a `DELETE` operation, the plugin should either delete the specified interface inside the specified container namespace or undo any modifications applied in the plugin's `ADD` functionality

- `CHECK` - check that the container networking is as expected; by invoking the CNI plugin requiring the `CHECK` operation, the runtime is able to probe the status of an existing container networking

- `VERSION` - show the CNI plugin version

Each operation must be able to handle the following list of environment parameters. Depending on the operation, some of them are required and other are optional:

- `CNI_COMMAND` - indicates the desired operation (`ADD`, `DEL`, `CHECK` or `VERSION`)
- `CNI_CONTAINERID` - a unique plaintext identifier for a container, allocated by the runtime; in Kubernetes, most of the times this is the container ID associated to the pause container of the Pod
- `CNI_NETNS` - indicates the isolation domain of the container; if network namespaces are used as isolation domains, this is the path to the network namespace (e.g. `/run/netns/[nsname]`)
- `CNI_IFNAME` - indicates the name of the target interface inside the container; in case of an ADD operation, this is the name of the interface that has to be created inside the container
- `CNI_ARGS` - alphanumeric key-value pairs separated by semicolons representing extra arguments that has to be passed to the plugin
- `CNI_PATH` - list of paths other than the default one to search for CNI plugins

Besides the environment parameters, a plugin configuration, inferred from the CNI configuration file, is passed to the CNI plugin in a JSON format through the standard input. This configuration may contain a `prevResult` field, describing the output of previous invocations of the plugin (the presence of this field is linked to the specific operation). It is possible to insert custom fields in the CNI configuration file, and these will be passed as they are to the CNI plugin: this can be exploit to customize the behaviour of the plugin.

4.4.2 The CNI configuration file

The CNI specification requires the presence of a configuration file in `/etc/cni/net.d`. In 4.8, an example of a possible configuration file, designed to work properly with the current CNI plugin implementation, is provided. As discussed in more details in the following sections, the CNI configuration is not placed in a static file before the network provider bootstrap, but is generated dynamically the first time the network topology is created (using information extracted from it); during the topology recovery procedures, some information, previously extracted for populating the configuration, are re-enforced on the cubes configuration.

Listing 4.8: CNI configuration file example

```
{
    "cniVersion": "0.4.0",
    "name": "mynet",
    "type": "polykube-cni-plugin",
    "mtu": 1450,
    "intK8sLbrp": "ikl0",
    "gateway": {
        "ip": "192.178.1.254",
        "mac": "6a:5e:56:e1:69:9a"
    },
    "ipam": {
        "type": "host-local",
        "ranges": [
            [
                {
                    "subnet": "192.178.1.0/24",
                    "rangeStart": "192.178.1.2",
                    "rangeEnd": "192.178.1.253",
                    "gateway": "192.178.1.254"
                }
            ]
        ],
        "dataDir": "/var/lib/cni/networks/mynet",
        "resolvConf": "/etc/resolv.conf"
    }
}
```

As it is possible to see from the configuration file, the Pod's default gateway IPv4 and the MAC addresses are provided to the CNI plugin. The file specifies also the name of the internal k8slbrp and the MTU that has to be configured on the veth: this latter value is chosen in order to avoid possible packets fragmentation due to the encapsulation technology used during the path towards the destination. The `host-local` IPAM plugin is invoked by the CNI plugin in order to retrieve an IP configuration, so parameters for instructing the `host-local` IPAM plugin are specified as well. The reason why the ranges of available IPv4 addresses go from .2 to .253 will be explained in the following sections.

4.4.3 The implementation

At high level, the CNI plugin must be able to handle the connection of the Pod with the k8slbrp and handle its network namespace routing and forwarding. The executable of the plugin is placed in the `/opt/cni/bin` folder by the operator. The following subsections provide an overview of the main steps followed during the execution of each operations. The code snippets are shrinked and readapted

starting from the real implementation, in order to not overload too much the visualization; examples are the omitted logging lines.

The ADD operation

The `ADD` operation is responsible of:

- creating a veth pair, assigning the required `CNI_IFNAME` name to the container end
- pushing the container end inside the Pod network namespace
- configuring the routing table of the network namespace with the Pod's default gateway IPv4 address
- configuring the ARP cache of the network namespace with the Pod's default gateway MAC address
- creating a frontend port on the internal k8slbrp, assigning to it the IPv4 address allocated by the IPAM module and connecting it to the veth pair host end

The code in 4.9 is an extract of the main operations carried out by the plugin when it is invoked with `CNI_COMMAND` environment variable set to `ADD`.

Listing 4.9: CNI plugin ADD operation main function

```
1  func cmdAdd(args *skel.CmdArgs) error {
2      // parsing configuration
3      conf, err := loadNetConf(args.StdinData)
4      if err != nil {
5          return fmt.Errorf("failed to parse netconf: %v", err)
6      }
7
8      // parsing prevResult, if present
9      var prevResult *current.Result
10     if conf.PrevResult != nil {
11         if prevResult, err = current.NewResultFromResult(conf.
       PrevResult); err != nil {
12             return fmt.Errorf("failed to convert prevResult into
       current version: %v", err)
13         }
14     }
15
16     // getting netns handle
17     netns, err := ns.GetNS(args.Netns)
18     if err != nil {
19         return fmt.Errorf("failed to open %q netns: %v", args.Netns,
       err)
```

```
20      }
21      defer netns.Close()
22
23      // checking if the specified iface already exists in the
        specified netns
24
25      // ... //
26
27      // getting IP from ipam plugin. Even if the returned IPv4 has a
        /24 prefix length, eventually a /32 will
28      // be configured on the container interface
29      addr, err := allocIP(conf.IPAM.Type, args.StdinData)
30      if err != nil {
31          return fmt.Errorf("failed to get ip through ipam plugin: %v",
         err)
32      }
33
34      // setting up the veth pair
35      contIfaceName := args.IfName
36      ipHexStr := hex.EncodeToString(addr.IP)
37      hostIfaceName := utils.GetHostIfaceName(contIfaceName, ipHexStr)
38
39      hostIface, contIface, err := setupVeth(netns, contIfaceName,
        hostIfaceName, conf.MTU)
40      if err != nil {
41          return fmt.Errorf("failed to setup veth pair: %v", err)
42      }
43
44      // configuring netns
45      if err := configureNetns(netns, contIfaceName, addr, &conf.Gw);
        err != nil {
46          return fmt.Errorf("failed to configure the %q netns: %v",
        args.Netns, err)
47      }
48
49      // creating k8slbrp port and connecting it to hostInterface
50      k8sLbrpName := conf.K8sLbrpName
51      k8sLbrpPortName := ipHexStr
52      port := k8slbrp.Ports{
53          Name:  k8sLbrpPortName,
54          Type_: "frontend",
55          Ip_:   addr.IP.String(),
56          Peer:  hostIfaceName,
57      }
58
59      if resp, err := k8sLbrpAPI.CreateK8sLbrpPortsByID(context.TODO(),
        k8sLbrpName, k8sLbrpPortName, port); err != nil {
60          return fmt.Errorf(
```

```
61              "failed to create and connect %q internal k8s lbrp %q
    port: %v", k8sLbrpName, k8sLbrpPortName, err,
62          )
63      }
64
65      // setting up the plugin result
66      result := &current.Result{}
67      if prevResult != nil {
68          result = prevResult
69      }
70      // The contIface will be placed at index len(result.Interfaces),
71      // so the hostIface will be placed at index len(result.Interfaces
    ) + 1
72      contIp := &current.IPConfig{
73          Interface: current.Int(len(result.Interfaces)), // 0 if the
    plugins is unchained
74          Address:   *addr,
75          Gateway:   conf.Gw.IP,
76      }
77      contRoute := &types.Route{
78          Dst: net.IPNet{
79              IP:   net.IPv4zero,
80              Mask: net.IPv4Mask(0, 0, 0, 0),
81          },
82          GW: conf.Gw.IP,
83      }
84      // The order is important! The contIface must precede the
    hostIface,
85      // since this will be exploited in the DEL and CHECK operations
86      result.Interfaces = append(result.Interfaces, contIface,
    hostIface)
87      result.IPs = append(result.IPs, contIp)
88      result.Routes = append(result.Routes, contRoute)
89
90      return types.PrintResult(result, conf.CNIVersion)
91  }
```

The container interface name is constraint by the `CNI_IFNAME` environment variable value. The name of the host end of the veth pair is chosen by taking into account the fact that the operator must be able to infer it starting from the information extracted from the Pod. The `utils.GetHostIfaceName` function is used to produce a suitable name for the interface, starting from container interface name and from the hexadecimal representation of the IPv4 address retrieved from the IPAM module. The obtained name is in the form `trunc(<contIface>)_<ipHexStr>`, where `trunc(<contIface>)` is the container interface name possibly truncated if the length of `<contIface>_<ipHexStr>` is higher than 15 character: this takes into account the constraint on the Linux interfaces name length.

The `setupVeth` function is responsible of creating the veth pair and pushing the container side inside the network namespace. The code is shown in 4.10.

Listing 4.10: setupVeth function used in the ADD operation

```
// setupVeth creates a veth pair using the provided ends names and puts the container end inside the provided netns. It
// returns a couple containing the container interface and the host interface info.
func setupVeth(netns ns.NetNS, contIfName, hostIfName string, mtu int) (*current.Interface, *current.Interface, error) {
    contIface := &current.Interface{}
    hostIface := &current.Interface{}

    err := netns.Do(func(hostNS ns.NetNS) error {
        // creating the veth pair in the container and moving host end into host netns
        hostVeth, contVeth, err := ip.SetupVethWithName(contIfName, hostIfName, mtu, "", hostNS)
        if err != nil {
            return err
        }
        contIface.Name = contVeth.Name
        contIface.Mac = contVeth.HardwareAddr.String()
        contIface.Sandbox = netns.Path()
        hostIface.Name = hostVeth.Name
        return nil
    })
    if err != nil {
        return nil, nil, err
    }

    // need to lookup hostVeth again as its index has changed during ns move
    hostVeth, err := netlink.LinkByName(hostIface.Name)
    if err != nil {
        return nil, nil, fmt.Errorf("failed %q lookup: %v", hostIface.Name, err)
    }
    hostIface.Mac = hostVeth.Attrs().HardwareAddr.String()

    return hostIface, contIface, nil
}
```

The k8slbrp frontend port is created following similar considerations to the ones made for the host interface. The name of the port is set to the hexadecimal representation of the IPv4 address associated to the container interface. The IP attribute of the port is set to the IPv4 address value.

The last notably thing is the positions of the container and the host interface in

produced plugin result. The container interface precedes the host one because this assumption will be used in the `DEL` and in the `CHECK` operations code.

The DEL operation

The `DEL` operation is responsible of:

- calling the IPAM module in order to release the IPv4 address assigned to the Pod
- deleting the k8slbrp frontend port associated with the Pod
- deleting the veth pair previously created for the Pod
- cleaning up the routing table and the ARP cache of the Pod's network namespace

The code in 4.11 is an extract of the main operations carried out by the plugin when it is invoked with `CNI_COMMAND` environment variable set to `DEL`.

Listing 4.11: CNI plugin DEL operation main function

```
1  func cmdDel(args *skel.CmdArgs) error {
2      // parsing configuration
3      conf, err := loadNetConf(args.StdinData)
4      if err != nil {
5          return err
6      }
7
8      // parsing prevResult
9      if conf.PrevResult == nil {
10         return errors.New("missing configuration: prevResult must be
       specified")
11     }
12     prevResult, err := current.NewResultFromResult(conf.PrevResult)
13     if err != nil {
14         return fmt.Errorf("failed to convert prevResult into current
       version: %v", err)
15     }
16
17     // extracting the container interface with its own ip
       configurations
18     contIfaceConf, _, err := getIfaceConfs(prevResult, args.IfName,
       args.Netns)
19     if err != nil {
20         return fmt.Errorf("unexpected prevResult: %v", err)
21     }
22     contIP := contIfaceConf.IPConf.Address.IP
23
```

```
24      // releasing IP address
25      if err := ipam.ExecDel(conf.IPAM.Type, args.StdinData); err !=
    nil {
26          return fmt.Errorf("DEL operation failed on ipam plugin: %v",
    err)
27      }
28
29      // deleting k8slbrp port
30      k8sLbrpName := conf.K8sLbrpName
31      k8sLbrpPortName := hex.EncodeToString(contIP)
32      if resp, err := k8sLbrpAPI.DeleteK8sLbrpPortsByID(
33          context.TODO(), k8sLbrpName, k8sLbrpPortName,
34      ); err != nil && resp.StatusCode != 409 {
35          return fmt.Errorf(
36              "failed to delete %q port on %q internal k8s lbrp: -
    error: %s, response: %+v",
37              k8sLbrpPortName, k8sLbrpName, err, resp,
38          )
39      }
40
41      // deleting netns iface and related stuff (routes, arpentry, etc
    ...)
42      if args.Netns != "" {
43          // There is a netns so try to clean up. Delete can be called
    multiple times
44          // so don't return an error if the device is already removed.
45          if err := ns.WithNetNSPath(args.Netns, func(_ ns.NetNS) error
     {
46              if err = ip.DelLinkByName(args.IfName); err != nil && err
     != ip.ErrLinkNotFound {
47                  // if there is an error different from ip.
    ErrLinkNotFound, returns error
48                  return err
49              }
50              return nil
51          }); err != nil {
52              // if netns is not found, continue anyway.
53              if _, notFound := err.(ns.NSPathNotExistErr); !notFound {
54                  return fmt.Errorf("failed to delete iface %q into
    netns %q: %v", args.IfName, args.Netns, err)
55              }
56          }
57      }
58
59      return nil
60  }
```

In order to complete its job, the `DEL` operation need to have access to the IPv4 address configured on the container veth pair end. This information need to

be extracted from the prevResult provided by the runtime: the `getIfaceConfs` function is used for this purpose. The code of the `getIfaceConfs` function is provided in 4.12.

Listing 4.12: getIfaceConfs function used in the DEL and CHECK operations

```
// getIfaceConfs scans the prevResult.Interfaces in order to find the
    expected container and host interface created
// during the ADD operation. If the two interfaces are found, they
    are returned in association with their IPConf
func getIfaceConfs(prevResult *current.Result, contIfName, netnsName
    string) (*IFaceConf, *IFaceConf, error) {
    var contIfaceConf, hostIfaceConf *IFaceConf
    // scanning all prevResult interfaces
    for i, iface := range prevResult.Interfaces {
        // is this the container interface?
        if iface.Name == contIfName && (netnsName == "" || netnsName
    == iface.Sandbox) {
            contIfaceConf = &IFaceConf{
                ResultIndex: i,
                Interface:   iface,
            }
            // the host interface is placed immediately after the
    container interface
            hostIfaceConf = &IFaceConf{
                ResultIndex: i + 1,
                Interface:   prevResult.Interfaces[i+1],
            }
            break
        }
    }
    // if the two interfaces were not found return an error
    if contIfaceConf == nil {
        return nil, nil, errors.New("unexpected interfaces: wrong or
    missing")
    }
    // scanning prevResult.IPs in order to associate the container
    interface to its ip configuration
    for _, ipConf := range prevResult.IPs {
        if *ipConf.Interface == hostIfaceConf.ResultIndex {
            ipConf.Address.IP = ipConf.Address.IP.To4()
            if ipConf.Address.IP == nil {
                return nil, nil, errors.New("unexpected non IPv4
    address configured on interface")
            }
            hostIfaceConf.IPConf = ipConf
        }
        if *ipConf.Interface == contIfaceConf.ResultIndex {
            ipConf.Address.IP = ipConf.Address.IP.To4()
            if ipConf.Address.IP == nil {
```

```
37                  return nil, nil, errors.New("unexpected non IPv4
    address configured on interface")
38              }
39              contIfaceConf.IPConf = ipConf
40          }
41      }
42      // checking that an ip configuration for the container interface
    and no configuration
43      // for the host interface were found
44      if contIfaceConf.IPConf == nil || hostIfaceConf.IPConf != nil {
45          return nil, nil, errors.New("unexpected ip configurations:
    wrong or missing")
46      }
47      return contIfaceConf, hostIfaceConf, nil
48  }
```

Thee `getIfaceConfs` function is written taking into account that the runtime could not provide the `CNI_NETNS` environment parameter in the DEL operation. The objective is to extract the container and the host interface information from the `prevResult` and store them in a structure, together with their configured IPs. The insertion order of the interfaces used during the `ADD` operation is exploited: the host interface is assumed to be immediately after the container interface. The function verifies that an IPv4 configuration is provided for the container interface and no IPv4 configuration is provide for the host interface.

The CHECK operation

The `CHECK` operation is responsible of:

- calling the IPAM module in order to check the IPv4 address assigned to the Pod
- checking the presence and the IPv4 configuration of the container veth pair end in the Pod's network namespace
- checking the networking configuration of the Pod's network namespace
- checking the presence of the host veth pair end in the root network namespace
- checking the presence and the configuration of the k8slbrp port associated to the Pod

The code in 4.13 is an extract of the main operations carried out by the plugin when it is invoked with `CNI_COMMAND` environment variable set to `CHECK`.

Listing 4.13: CNI plugin CHECK operation main function

```
func cmdCheck(args *skel.CmdArgs) error {
    // parsing configuration
    conf, err := loadNetConf(args.StdinData)
    if err != nil {
        return err
    }

    // checking the presence of prevResult (its presence is made mandatory by the CNI specification and parsing it
    // in order to check the container networking)
    if conf.PrevResult == nil {
        return errors.New("missing configuration: prevResult must be specified")
    }
    prevResult, err := current.NewResultFromResult(conf.PrevResult)
    if err != nil {
        return fmt.Errorf("failed to convert prevResult into current version: %v", err)
    }

    // CHECK on ipam plugin
    err = ipam.ExecCheck(conf.IPAM.Type, args.StdinData)
    if err != nil {
        return fmt.Errorf("CHECK operation failed on ipam plugin: %v", err)
    }

    // getting netns handle
    netns, err := ns.GetNS(args.Netns)
    if err != nil {
        return fmt.Errorf("failed to open %q netns: %v", args.Netns, err)
    }
    defer netns.Close()

    // extracting the container interface and the host interface with their own IP configurations
    contIfaceConf, hostIfaceConf, err := getIfaceConfs(prevResult, args.IfName, args.Netns)
    if err != nil {
        return fmt.Errorf("unexpected prevResult: %v", err)
    }

    // checking args.Netns netns interface and routes
    if err := netns.Do(func(_ ns.NetNS) error {
        if err := checkIface(nlog, args.Netns, contIfaceConf); err != nil {
            return err
        }
```

```
42
43          // checking that routes are correctly configured
44          if err := ip.ValidateExpectedRoute(prevResult.Routes); err !=
        nil {
45              return fmt.Errorf("failed %q netns routes checking: %v",
        args.Netns, err)
46          }
47          return nil
48      }); err != nil {
49          return err
50      }
51
52      // checking root netns interface
53      if err := checkIface(nlog, "root", hostIfaceConf); err != nil {
54          return err
55      }
56
57      // checking k8slbrp port connection
58      k8sLbrpName := conf.K8sLbrpName
59      contIP := contIfaceConf.IPConf.Address.IP
60      k8sLbrpPortName := hex.EncodeToString(contIP)
61      k8sLbrpPortPeer := hostIfaceConf.Interface.Name
62      if err := checkK8sLbrpPort(k8sLbrpName, k8sLbrpPortName,
        k8sLbrpPortPeer, contIP.String()); err != nil {
63          return fmt.Errorf("failed %q k8slbrp port checking: %v",
        k8sLbrpName, err)
64      }
65      return nil
66  }
```

Also in this case, the `getIfaceConfs` function is used to retrieve the relevant interfaces information.

The `checkIface` function is used for checking the interface existence and their IPv4 configuration. The function code is provided in 4.14.

Listing 4.14: checkIface function used in the CHECK operation

```
1  // checkIface verifies that the provided interface is correctly
       configured.
2  func checkIface(..., netnsName string, iface *IFaceConf) error {
3      ifaceName := iface.Interface.Name
4
5      // obtaining interface corresponding link
6      link, err := netlink.LinkByName(ifaceName)
7      if err != nil {
8          if _, notFound := err.(netlink.LinkNotFoundError); notFound {
9              return fmt.Errorf("%q iface doesn't exist into %q netns:
       %v", ifaceName, netnsName, err)
10         }
```

```
11          return fmt.Errorf("failed %q iface lookup into %q netns: %v",
        ifaceName, netnsName, err)
12      }
13
14      // if no ip configuration are expected to be configured on link,
        simply return
15      if iface.IPConf == nil {
16          return nil
17      }
18
19      // obtaining addresses configured on link
20      addrs, err := netlink.AddrList(link, netlink.FAMILY_V4)
21      if err != nil {
22          return fmt.Errorf("failed %q iface addresses lookup into %q
        netns: %v", ifaceName, netnsName, err)
23      }
24
25      // checking if the ip addresses are correctly configured on link
26      for _, addr := range addrs {
27          if addr.IPNet.String() == iface.IPConf.Address.String() {
28              return nil
29          }
30      }
31      return fmt.Errorf("%q iface ip misconfiguration into %q netns: %v
        ", ifaceName, netnsName, err)
32  }
```

4.5 The Polykube operator

As discussed in the previous chapter, an agent is needed to create the proper network infrastructure on each node of the cluster. Besides the tasks related to setting up the virtual network topology and the proper interfaces, the agent must react to the Kubernetes cluster changes: so the agent must be a Kubernetes operator. In order to easily develop the operator, the Operator SDK has been used. The Operator SDK provides the tools to build, test, and package Operators [19] written in Golang. It leverages the facilities offered by the `controller-runtime` library to handle resources watching, caching, events rescheduling and all the tasks related to operator building. The operator must starts a set of controller: each one of them is associated with a primary resource, but could also be associated to other secondary resources. In the following sections an in-depth overview of the implementation details of the operator is given. The Polykube operator project is available in [20].

4.5.1 The code structure

The Polykube operator code is organized in the following packages:

- **cni** - containing the code related to the CNI plugin implementation as well as the scripts needed to install/uninstall it from the node
- **controllers** - containing the code related to the controllers implementation
- **node** - containing all the code related to manage the specific node; specifically, it contains the code for retrieving environment information and information related to the Kubernetes node (e.g: Pod CIDR or node external interface and node default gateway) as well as the code for managing the VxLAN interface and the polykube veth pair
- **polycube** - containing all the code related to manage the network virtual topology; specifically, it contains the generated client libraries for interacting with the polycubed REST APIs, the code for ensuring that the proper Polycube cubes exist and are connected and the code for managing the cubes configuration in reaction to the cluster events
- **types** - containing the definition of the Golang structures that do not conceptually belong to any of the other packages
- **utils** - containing some utility function used by the other packages

4.5.2 Node controller

The **Node controller** code reconciliation loop is implemented by the `Reconcile` function of the `NodeReconciler` struct. Before a node event can trigger the execution of the `Reconcile` function, it is examined by the opportune predicate returned by the `nodeControllerPredicate` function. Whatever is the type of the event (creation, updating of deletion), node events can trigger the reconciliation loop only if they regard a node different from the one on which the operator instance is deployed. The creation or the deletion of a control plane node doesn't trigger the reconciliation loop if the deployment of the network provider is not allowed in control plane nodes. Updating events trigger a reconciliation if a not ready node becomes ready or vice versa. Also, updating events pass the predicate if a normal node becomes a control plane one or a control plane one becomes a normal one.

The `NodeDetailMap` map is created is order to store the information regarding the nodes. For each node correctly retrieved by the API server, a `NodeDetail` struct is built: it contains the IPv4 address of the node physical interface connected to the Kubernetes cluster, its Pod CIDR and its Vtep IPv4 address and prefix

length. The `NodeDetailMap` is indexed by the `NamespacedName` extracted from the request object.

At high level, the actions performed by the Node controller are the following:

- if the node has been *deleted*, is *not ready* or it is become a *control plane node*, the route towards its Pod CIDR through the VxLAN interface must be deleted (if present) from the Polycube router, and the Linux bridge forwarding database entry related to it must be removed (if present); in case the node has been deleted, the information needed in order to perform the described cleanup are taken from the `NodeDetailMap`
- in all the other cases, the controller has to enforce the route towards the Pod CIDR through the VxLAN interface in the Polycube router and the Linux bridge forwarding database entry; before trying to enforce these configurations, a check is performed in order to verify if they are already enforced

The `Renconcile` function of the `NodeReconciler` is shown in 4.15.

Listing 4.15: the Node controller reconciliation loop

```
1  func (r *NodeReconciler) Reconcile(ctx context.Context, req ctrl.
       Request) (ctrl.Result, error) {
2      nId := req.NamespacedName.String()
3
4      n := &corev1.Node{}
5      if err := r.Get(ctx, req.NamespacedName, n); err != nil {
6          // if the error is different from not found returns error...
7          if !apierrors.IsNotFound(err) {
8              return ctrl.Result{}, err
9          }
10         nodeDetail, found := NodeDetailMap[nId]
11         if !found {
12             return ctrl.Result{}, nil
13         }
14         // the node object was deleted, so it has to be removed from
       the infrastructure
15         if err := polycube.DeleteRouterRoute(nodeDetail.PodCIDR,
       nodeDetail.VtepIPNet.IP); err != nil {
16             return ctrl.Result{}, err
17         }
18         if err := node.DeleteFdbEntry(nodeDetail.IP); err != nil {
19             return ctrl.Result{}, err
20         }
21         delete(NodeDetailMap, nId)
22         return ctrl.Result{}, nil
23     }
24
25     nodeDetail, err := buildNodeDetail(n)
```

```
26      if err != nil {
27          return ctrl.Result{}, err
28      }
29
30      _, isControlPlaneNode := n.Labels[controlPlaneLabel]
31      if (isControlPlaneNode && !node.Env.IsCPNodesDeployAllowed) || !
    node.IsReady(n) {
32          if err := polycube.DeleteRouterRoute(nodeDetail.PodCIDR,
    nodeDetail.VtepIPNet.IP); err != nil {
33              return ctrl.Result{}, err
34          }
35          if err := node.DeleteFdbEntry(nodeDetail.IP); err != nil {
36              return ctrl.Result{}, err
37          }
38          return ctrl.Result{}, nil
39      }
40
41      // the following will create or update the entry related to the
    node inside the map: this
42      // is needed in order to handle a possible future node deletion
43      NodeDetailMap[nId] = nodeDetail
44
45      routeExist, err := polycube.CheckRouterRouteExistence(nodeDetail.
    PodCIDR, nodeDetail.VtepIPNet.IP)
46      if err != nil {
47          return ctrl.Result{}, err
48      }
49      if !routeExist {
50          if err := polycube.CreateRouterRoute(nodeDetail.PodCIDR,
    nodeDetail.VtepIPNet.IP); err != nil {
51              return ctrl.Result{}, err
52          }
53      }
54
55      entryExist, err := node.CheckFdbEntryExistence(nodeDetail.IP)
56      if err != nil {
57          return ctrl.Result{}, err
58      }
59      if !entryExist {
60          if err := node.CreateFdbEntry(nodeDetail.IP); err != nil {
61              return ctrl.Result{}, err
62          }
63      }
64
65      return ctrl.Result{}, nil
66  }
```

4.5.3 Service controller

The **Service controller** is implemented through the `ServiceReconciler` and its `Reconcile` function. All the events regarding Services must trigger the reconciliation loop and, for this reason, no predicate are implemented for filtering events.

The `ServiceDetail` struct is built in order to store in an efficient way the information extracted from a Kubernetes Service Object. This struct is generic and can store information for both the two supported types of Kubernetes Services: ClusterIP Services and NodePort Services. The structure is built upon the concept of `FrontendSet`: a `FrontendSet` represents a set of `Frontend`; a `Frontend` is a struct representing a network service in the form `<VirtualIP>:<VirtualPort>:<ProtocolType>`. In 4.16, the definition of these latter concepts is shown. In order to easily work with the `FrontendSet` struct, some utility functions are built for adding elements to it or for checking if it contains a particular `Frontend`.

Listing 4.16: the ServiceDetail and the FrontendSet definitions

```
type Frontend struct {
    Vip   string
    Vport int32
    Proto string
}

type FrontendsSet map[Frontend]struct{}

type ServiceDetail struct {
    ServiceId              string
    NodePortFrontendsSet   FrontendsSet
    ClusterIPFrontendsSet  FrontendsSet
    ExternalTrafficPolicy  string
    InternalTrafficPolicy  string
}
```

Besides the `FrontendSets` for the NodePort Services and the ClusterIP Services, the `ServiceDetail` struct contains also the information on the internal and the external traffic policies.

Throughout the Service reconciliation loop, each Service is identified by the `NamespacedName` extracted from the related request object: this identifier will be denoted by name `serviceId` during the following discussion. At high level, the actions performed by the Service controller are the followings:

- if the Service is of a type different from ClusterIP or NodePort, no action are performed
- if the Service has been deleted, a different cleanup action is performed depending on the Service type: if the Service is of type ClusterIP, all the internal

k8slbrp services having names equal to the `serviceId` are deleted; if the Service is of type NodePort, all the internal k8slbrp services, all the external k8slbrp services and all the NodePort rules having names equal to `serviceId`, are deleted

- in all the other cases, the controller has to enforce the proper services on the internal and external k8slbrp and the proper NodePort rules on the k8sdispatcher: like in the previous case, services or rules are enforced on the proper components depending on the Service type

The `Renconcile` function of the `ServiceReconciler` is shown in 4.17.

Listing 4.17: the Service controller reconciliation loop

```
1  func (r *ServiceReconciler) Reconcile(ctx context.Context, req ctrl.
       Request) (ctrl.Result, error) {
2      sId := req.NamespacedName.String()
3
4      s := &corev1.Service{}
5      if err := r.Get(ctx, req.NamespacedName, s); err != nil {
6          // if the error is different from not found returns error...
7          if !errors.IsNotFound(err) {
8              return ctrl.Result{}, err
9          }
10         // the following work for both the internal and the external
       load balancers.
11         // In case o services other than ClusterIP and NodePort ones,
        the following
12         // will not delete anything
13         if err := polycube.CleanupK8sLbrpsServicesById(sId); err !=
       nil {
14             return ctrl.Result{}, err
15         }
16         if err := polycube.CleanupK8sDispatcherNodePortRulesById(sId)
       ; err != nil {
17             return ctrl.Result{}, err
18         }
19         return ctrl.Result{}, nil
20     }
21
22     st := s.Spec.Type
23     // not able to handle services other than ClusterIP and NodePort
       ones
24     if st != corev1.ServiceTypeClusterIP && st != corev1.
       ServiceTypeNodePort {
25         return ctrl.Result{}, nil
26     }
27
28     nodeIP := node.Conf.ExtIface.IPNet.IP
```

```
29      sd := buildServiceDetail(s, sId, nodeIP)
30
31      if err := polycube.SyncK8sLbrpServices(sd); err != nil {
32          return ctrl.Result{}, err
33      }
34
35      if err := polycube.SyncK8sDispatcherNodePortRules(sd, nodeIP);
        err != nil {
36          return ctrl.Result{}, err
37      }
38
39      return ctrl.Result{}, nil
40  }
```

The function responsible of building the `ServiceDetail` struct associated to the Service object is `buildServiceDetail`. This function creates a `Frontend` struct for each possible couple (ServicePort, ClusterIP) extracted from the Service object. The obtained `Frontends` are stored in the `ClusterIPFrontendsSet` field of the `ServiceDetail` struct. If the Service is of type NodePort, a `Frontend` struct is created for the `NodePort` field of each `ServicePort` of the Service object: these `NodePort` ports are associated to the IP address configured on the physical interface of the node. The obtained `Frontends` are stored in the `NodePortFrontendsSet` field of the `ServiceDetail` struct.

4.5.4 Endpoints controller

The **Endpoints controller** is implemented through the `EndpointsReconciler` and its `Reconcile` function. Besides watching for events regarding Endpoints, the Endpoints controller watches also the events regarding Services in order to take into account the changes made upon their traffic policies: this is done in order to clearly separate in two different controllers the handling of the topology services and the handling of the related backends.

Before an Endpoints or a Service event can trigger the execution of the `Reconcile` function, the event is examined by the opportune predicate returned by the `endpointsControllerPredicate` function. Whatever is the type of object to which the event is referring, if it is a deletion event, it does not trigger the reconciliation loop: this is done because the deletion of a Service is already handled by the Service controller, and the deletion of an Endpoints object is always linked to the deletion of the corresponding Service object (and so it is also handled by the Service controller). The creation events are handled only if they refer to Endpoints object. An updating event can trigger the reconciliation loop only if the event is referred to an Endpoints object or some changes were made on the Service object traffic policies.

The `EndpointsDetail` struct is built in order to store in an efficient way the information extracted from a Kubernetes Endpoints Object. The snippet in 4.18 shows how this struct ,together with related types and functionatilies, are defined.

Listing 4.18: the EndpointsDetail, the ServiceToBackends and the BackendsSet definitions

```
type Backend struct {
    Ip      string
    Port    int32
    Weight int32
}
type BackendsSet map[Backend]struct{}

func (bs BackendsSet) Contains(b Backend) bool {
    _, ok := bs[b]
    return ok
}
func (bs BackendsSet) Add(b Backend) {
    bs[b] = struct{}{}
}

type ServiceToBackends map[string]BackendsSet

func (stb ServiceToBackends) Add(s string, b Backend) {
    if stb[s] == nil {
        stb[s] = make(BackendsSet)
    }
    stb[s][b] = struct{}{}
}

func (stb ServiceToBackends) GetBackendsSet(s string) BackendsSet {
    return stb[s]
}

type EndpointsDetail struct {
    EndpointsId                 string
    ClusterIPServiceToBackends  ServiceToBackends
    NodePortServiceToBackends   ServiceToBackends
}
```

The `EndpointsDetail` struct is generic and can store information regarding the endpoints of a ClusterIP Service as well as the endpoints of a NodePort Service. If the Service is of type ClusterIP, only the `ClusterIPServiceToBackends` field will be populated (if any backend is present). If the Service is of type NodePort, both the `ClusterIPServiceToBackends` and the `NodePortServiceToBackends` fields will be populated (if any backend is present). The `EndpointsDetail` struct is built in two steps:

- in the first step, a `ServiceToBackends` struct is built by using the `buildServiceToBackends` function

- in the second step, the `EndpointsDetail` struct is built by combining the information extracted from the Endpoints related Service object and the information organized inside the already built `ServiceToBackends`; this second step is achieved by using the `buildEndpointsDetail` function

The `buildServiceToBackends` function creates a `BackendsSet` struct for each `EndpointPort` contained in the `EndpointSubset` list of Endpoints object. An `EndpointPort` is associated with a unique `ServicePort`: this association will be exploited by the `buildEndpointsDetail` function. An `EndpointPort` must be identified by a name if more than one port is present; if only one port is present, then the name is optional. By using the convention of identifying the single unnamed `EndpointPort` with an hyphen, the `ServiceToBackends` maps an `EndpointPort` name with the related set of `Backends`. A `Backend` struct contains also the information related to the weight of the backend in the load balancing logic. The convention used is to prefer backends Pods deployed on the same node, so 3 is assigned as a weight to them and 1 to Pods deployed on other nodes.

The `ServiceToBackends` struct is provided to the `buildEndpointsDetail` function, together with the Service object related to the Endpoints object.

For each couple `(ClusterIP, ServicePort)` extracted from the Service object, a triple `<vip>:<vport>:<proto>` is created. For each triple, a `BackendsSet` is created: this is populated with the `Backends` of the `EndpointsPort` associated with the `ServicePort`. The association between triples and `BackendsSet` are stored in the `ClusterIPServiceToBackends` field of the `EndpointsDetail`.

If the Service is of type NodePort, also the `NodePortServiceToBackends` field is populated: for each `ServicePort`, the triple `<nodeip>:<nodeport>:<proto>` is created; for each triple, a `BackendsSet` is created and populated in a way similar to the one described above.

In both cases, `buildEndpointsDetail` function inserts a backends in the set of a particular service only if it is policy-compliant: for example, if a backend is deployed on a node different from the local node and the Service object has `internalTrafficPolicy=LOCAL`, then the backend will not be considered as it is not possible for a Pod to reach a backend of the Service in another node.

Throughout the Endpoints reconciliation loop, each Endpoints is identified by the `NamespacedName` extracted from the related request object: this identifier will be denoted with `epsId` during the following discussion. At high level, the actions performed by the Endpoints controller are the followings (notice that this are the actions performed only if the event passes the predicate):

1. the Endpoints object is retrieved

2. the Service object related to the Endpoints object is retrieved
3. if the Service object type if different from ClusterIP and NodePort no action are performed
4. an `EndpointsDetail` struct is built
5. the load balancers services backends are synced in order to reflect the backends list described in the Endpoints object

The `Renconcile` function of the `EndpointsReconciler` is shown in 4.19.

Listing 4.19: the Endpoints controller reconciliation loop

```
1  func (r *EndpointsReconciler) Reconcile(ctx context.Context, req ctrl
       .Request) (ctrl.Result, error) {
2      epsId := req.NamespacedName.String()
3
4      eps := &corev1.Endpoints{}
5      if err := r.Get(ctx, req.NamespacedName, eps); err != nil {
6          // if the endpoints resource is not found, it means that the
       associated
7          // service was deleted: in this case, let the service
       controller handle
8          // the cleanup
9          return ctrl.Result{}, client.IgnoreNotFound(err)
10     }
11
12     s := &corev1.Service{}
13     if err := r.Get(ctx, req.NamespacedName, s); err != nil {
14         // if the service resource is not found, the service
       controller will handle
15         // the cleanup
16         return ctrl.Result{}, client.IgnoreNotFound(err)
17     }
18
19     st := s.Spec.Type
20     // not able to handle services other than ClusterIP and NodePort
       ones
21     if st != corev1.ServiceTypeClusterIP && st != corev1.
       ServiceTypeNodePort {
22         return ctrl.Result{}, nil
23     }
24
25     serviceToBackends := buildServiceToBackends(eps)
26     endpointsDetail := buildEndpointsDetail(s, epsId,
       serviceToBackends)
27
28     needResync, err := polycube.SyncK8sLbrpsServicesBackends(
       endpointsDetail)
```

```
29      if err != nil {
30          return ctrl.Result{}, err
31      }
32      if needResync {
33          return ctrl.Result{Requeue: true}, nil
34      }
35
36      return ctrl.Result{}, nil
37  }
```

4.5.5 The recovery procedure

Two types of failure can occur during the network provider life cycle: a crash of the `polycubed` daemon (and a consequent crash of the topology) or a crash of the `polykube-operator`. The `polykube-operator` detects that the Polycube daemon is unreachable through a polling procedure: it performs an HTTP request each 10 seconds and, if the request fails, it start the recovery procedure. On the other hand, if the operator crashes, it is able to restart and analyze the status of the node in order to recover the operating status. Both the recovery procedure are handled by the `ensureDeployment` function: indeed this function is called each time the operator starts its execution and also each time the polling procedure detect a failure in the Polycube daemon. The main steps followed by the function are the followings:

1. ensuring the connection to the polycubed by setting up a polling procedure with an exponential backoff delay between one request and another
2. ensuring that all the cubes composing the topology are up and runnning
3. ensuring that all the cubes are connected to each other and to the networking interfaces in the proper way
4. ensuring the presence of the CNI configuration file and the compliance of its information with the one extracted from the topology; specifically, if `ensureDeployment` is called again after the operator initialization, the polycube router MAC address is set to be equal to previously retrieved one
5. ensuring that previously deployed Pods are connected to the topology; this is made possible by exploiting the naming conventions used by the CNI plugin

4.5.6 The addresses management

The agent is responsible of evaluating, starting from the information extracted from the environment (the API server, the cluster node on which is deployed or

the environment variables), the IPv4 configurations that must be enforced during the network provider deployment.

The node address

The main node physical interface connecting the node to the Kubernetes cluster can be retrieved from the API server. This address is used to define a Polycube router route for allowing Pods to contact, through the polycube veth pair, host processes listening on the node physical interface address.

Pods default gateway

Starting from the Pod CIDR assigned to the node (retrieved from the API server), the agent evaluates the IPv4 address that must be assigned to the Polycube router port corresponding to the Pods default gateway. The used convention is to assign the last address other than the broadcast one: for example, if the Pod CIDR is `192.178.1.0/24`, than the chosen address will be `192.178.1.254/24`.

The virtual Pod address

An IPv4 address must be reserved for allowing the k8sdispatcher to handle properly requests for NodePort Services with `externalTrafficPolicy=CLUSTER`. This address is called "the virtual Pod address", even if it is not assigned to any Pod, but only configured during the k8sdispatcher creation. The convention is to assign the first address of the Pod CIDR other than the network ID: for example, if the Pod CIDR is `192.178.1.0/24`, than the chosen address will be `192.178.1.1`.

The Vtep address

The IPv4 address that must be configured on the VxLAN tunnel endpoint is the most complex to evaluate. The complexity is due to the fact that, for designing choices, it was preferred to not conceive a distributed IPAM system. The address is calculated using 3 CIDRs: the Cluster CIDR, the Pod CIDR and the Vtep CIDR. The Cluster CIDR is the CIDR from which the Pod CIDRs of all the nodes are extracted. The Cluster CIDR must be provided from the user during the configuration of the network provider manifests, since it cannot be inferred easily from the cluster APIs. The Vtep CIDR is chosen by the user and has to be provided in the configuration manifests by the user as well. During the following explanation, for a visualization purpose, the following setting can be assumed: `ClusterCIDR=192.178.0.0/16`, `PodCIDR=192.178.1.0/24`, `VtepCIDR=10.18.0.0/16`. The procedure is the following:

1. the portion `p` of the Pod CIDR that specifically identifies the node is extracted: in the previous example it is composed by the bits of the second least significant byte (the .1. portion)

2. the length of `p` is verified in order to check if it is not greater than the length of the host identifier portion of the Vtep CIDR; in the example the length of `p` is 8 and it is not greater than 16, the length of the host identifier of `10.18.0.0/16`

3. the Vtep IPv4 address is calculated by placing `p` at the end of the Vtep CIDR; in the previous example, the obtained Vtep address is `10.18.0.1/16`

The described procedure allows each node to have a different IPv4 address for the VxLAN tunnel endpoint, without the necessity to store information regarding the already allocated addresses.

4.5.7 Environment configuration

In order to give the operator access to the information that cannot be retrieved by contacting the API server or inferred from the node configuration, or in order to customize the behaviour of the network provider, configuration information must be provided. The configuration information are provided through environment variables to the `polykube-operator` container: the majority of these environment variables are extracted by the platform from a specifically deployed **ConfigMap**. The ConfigMap must be called `polykube-cfg` and must be deployed in the `kube-system` namespace: a possible example is shown in 4.20.

Listing 4.20: The network provider config map

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: polykube-cfg
  namespace: kube-system
data:
  vxlanIfaceName: "vxlan0"
  polykubeVethPairNamePrefix: "polykube"
  vtepCidr: "10.18.0.0/16"
  polykubeVethPairCidr: "172.18.0.0/30"
  clusterCidr: "192.178.0.0/16"
  nodePortRange: "30000-32767"
  cniConfFilePath: "/host/etc/cni/net.d/00-polykube.json"
  mtu: "1450"
  intK8sLbrpName: "ikl0"
  routerName: "r0"
  extK8sLbrpName: "ekl0"
```

```
18   k8sDispName: "k0"
19   cubesLogLevel: "INFO"
20   # if the following is set to true, the following toleration has to
       be
21   # added to the daemonset tolerations:
22   # - effect: NoSchedule
23   #   key: node-role.kubernetes.io/master
24   isCPNodesDeployAllowed: "true"
```

The operator is able to default the majority of the configuration information, but some of them are required. An example of required information is the `clusterCidr` field. The ConfigMap allows to customize different aspects of the polycube topology, such as the cubes name and the cubes log level. The `isCPNodesDeployAllowed` flag allows to specify if the network provider has to be deployed also on control plane nodes.

Other environment variables, such as the Operator Pod name or the Node name, are directly extracted from the DaemonSet object used for deploying the network provider.

4.6 Network provider deployment

As already discussed, an instance of the Polykube Operator and an instance of the Polycube image must be deployed on each node. By exploiting the concept of DaemonSet, it is possible to realize this setting in an declarative way. A stripped version of the DaemonSet manifest is shown in 4.21.

Listing 4.21: The network provider DaemonSet manifest

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: polykube
5    namespace: kube-system
6  spec:
7    selector:
8      matchLabels:
9        k8s-app: polykube
10   template:
11     metadata:
12       labels:
13         k8s-app: polykube
14     spec:
15       priorityClassName: system-node-critical
16       serviceAccountName: polykube-ctrl-mgr-sa
17       hostNetwork: true
18       containers:
```

```
19        - name: polycubed
20          image: ekoops/polycube:latest
21          securityContext:
22            privileged: true
23          command: ["polycubed",
24                    "--loglevel=INFO",
25                    "--addr=0.0.0.0",
26                    "--logfile=/host/var/log/polycubed.polykube.log"]
27          volumeMounts:
28            # ...
29        - name: polykube-operator
30          image: ekoops/polykube-operator:latest
31          securityContext:
32            privileged: true
33          command: ["/polykube"]
34          lifecycle:
35            postStart:
36              exec:
37                command:
38                  - "/cni-install.sh"
39            preStop:
40              exec:
41                command:
42                  - "/cni-uninstall.sh"
43          env:
44            - name: POD_NAME
45              valueFrom:
46                fieldRef:
47                  fieldPath: metadata.name
48            - name: NODE_K8S_NAME
49              valueFrom:
50                fieldRef:
51                  fieldPath: spec.nodeName
52            - name: VXLAN_IFACE_NAME
53              valueFrom:
54                configMapKeyRef:
55                  name: polykube-cfg
56                  key: vxlanIfaceName
57            # ...
58          volumeMounts:
59            # ...
60      volumes:
61        # ...
62      tolerations:
63        - effect: NoSchedule
64          key: node.kubernetes.io/not-ready
65        #  Uncomment/Comment the following toleration in order to
      allow/disallow deployment on master node
66        - effect: NoSchedule
```

```
67            key: node-role.kubernetes.io/master
68          - effect: NoSchedule
69            key: node.cloudprovider.kubernetes.io/uninitialized
70            value: "true"
71          - key: CriticalAddonsOnly
72            operator: "Exists"
```

As it is possible to see, the concept of privileged Pod is used in order to allow containers to access the host interfaces and the host root network namespace. Among other things, tolerations are used for deploying the network provider also on the master node. The Pod is also associated with a specific ServiceAccount: this is done for enabling access to the required information through the API server. The ServiceAccount links the Pod with the Role shown in 4.22.

Listing 4.22: The Role manifest

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: ClusterRole
3  metadata:
4    creationTimestamp: null
5    name: polykube-ctrl-mgr-role
6  rules:
7  - apiGroups:
8    - ""
9    resources:
10   - nodes
11   verbs:
12   - get
13   - list
14   - watch
15   - update
16 - apiGroups:
17   - ""
18   resources:
19   - nodes/finalizers
20   verbs:
21   - update
22 - apiGroups:
23   - ""
24   resources:
25   - nodes/status
26   verbs:
27   - get
28   - list
29 - apiGroups:
30   - ""
31   resources:
32   - services
33   verbs:
```

```
34    - get
35    - list
36    - watch
37  - apiGroups:
38    - ""
39    resources:
40    - services/status
41    verbs:
42    - get
43  - apiGroups:
44      - ""
45    resources:
46      - endpoints
47    verbs:
48      - get
49      - list
50      - watch
51  - apiGroups:
52      - ""
53    resources:
54      - endpoints/status
55    verbs:
56      - get
57  - apiGroups:
58      - ""
59    resources:
60      - pods
61    verbs:
62      - list
```

Chapter 5

Evaluation

This chapter analyzes the network performance of the proposed network provider. To assess the overhead of the disaggregated approach, the throughput provided by the network provider under different circumstances has been measured.

5.1 Tools

The tests have been run using the `iPerf3` tool. iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks [21]. Tests have been conducted by using the default parameters and instructing iPerf3 to use zero-copy methods for sending data. A zero-copy method allows to reach higher throughput values with less CPU usage: this is useful to see better the performance differences. In order to use the zero-copy approach, the `-Z` flag is passed to the iPerf3 client instance.

5.2 Tests overview

The client and the server instances have always been executed inside Pods. Both TCP and UDP communications have been tested. Tests have been performed both for communication involving entities on the same node and on different nodes. The following communication scenarios have been considered:

- **Container-to-Container (localhost)** - both client and server run within the same Pod and use the loopback interface to communicate; this represents a baseline of the throughput that can be achieved between two Pods, since no additional networking is required
- **Pod-to-Pod** - a client in a Pod accesses the server through the Pod's IP, showing the performance of the base networking without load balancing

- **Pod-to-Service** - the client accesses the server using a ClusterIP service, to evaluate the performance of the load balancer as well as the L3 routing

The Pod-to-Pod and Pod-to-Service tests have been performed with Pods running both on the same machine and on different nodes, with the latter adding the overhead of the VxLAN encapsulation and of the physical network (link speed, PCI bus). The Container-to-Container tests cannot be performed for container running on different nodes, since containers belonging to the same Pod are always scheduled on the same node, inside the same network namespace.

To have a comparison with another eBPF solutions, the **Cilium** and the **Calico** providers have been tested and the results have been compared with the ones obtained by the proposed solution. Cilium has been deployed using its eBPF kube-proxy replacement: this enables the provider to leverage all the benefits of eBPF without being penalized by the overhead introduced by kube-proxy in services management. The XDP acceleration has been disabled in order to enable a fair comparison, since the proposed network provider doesn't make use of XDP. Similar consideration can be done for Calico: the eBPF dataplane has been enabled in order to completely replace kube-proxy.

In order to compare the approach used by the proposed solution with the one proposed by traditional solutions that do not use eBPF, also **Flannel** has been tested. Flannel leverages kube-proxy for providing full Kubernetes cluster connectivity.

The throughput performances of the tested solutions can be negatively influenced by the number of Services/Pod currently deployed on the cluster when tests are performed. In order to evaluate how strong can be this influence, Pod-to-Service tests have been conducted in scenarios with 0, 10 or 100 additional Kubernetes Services. Each additional Kubernetes Service is deployed together with an associated backend Pod.

Each setting has been tested 10 times and from the obtained samples, the mean and the standard deviation has been evaluated in order to realize significant plots.

5.3 Cluster and network settings

A cluster composed by two physical nodes has been deployed for testing: this is needed for testing both communications involving entities on a single node and communications involving entities deployed on different nodes. The two nodes will be called **nodeA** and **nodeB** in the following discussion. nodeA and nodeB are installed in the same sub-network and are connected through a 40G Switch: this is needed in order to avoid performances to be penalized by the physical network. Both nodeA and nodeB are equipped with:

- **CPU** - Intel(R) Xeon(R) CPU E3-1245 v5 @ 3.50GHz

- **Memory** - 64 GB DDR4 2400 MT/s
- **NIC** - 40000Mb/s port
- **OS** - Ubuntu 18.04
- **Kernel version** - 5.4.0-96-generic

The control plane is deployed on nodeA. Tests involving a single node are conducted by deploying the iPerf3 client and server on the nodeB: this is done to avoid performance penalties induced by the execution of the control plane. Tests involving different nodes are conducted by placing the client on nodeA and the server on nodeB.

5.4 Communications on the same node

5.4.1 TCP

Results for TCP communications involving a client and a server deployed on the same node are shown in Fig. 5.1.

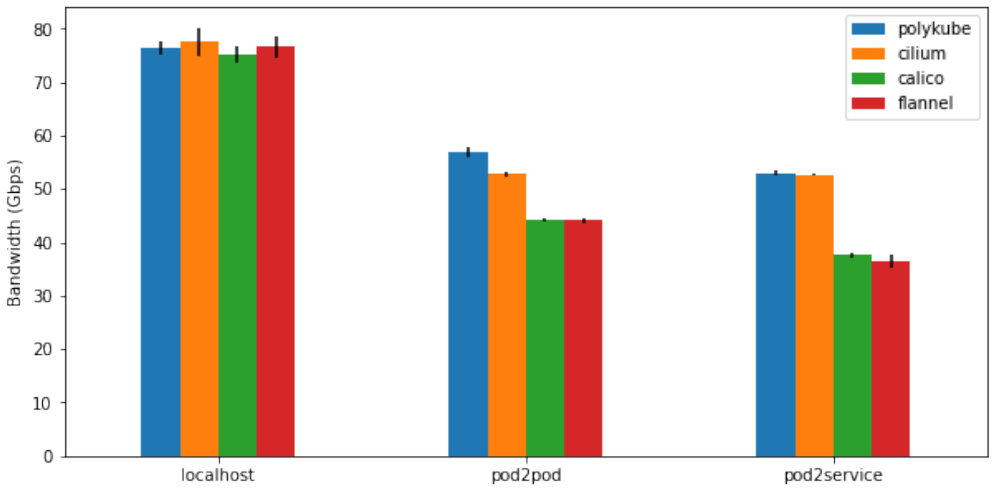

Figure 5.1: Same node throughput comparison for TCP traffic

As expected, the baseline performances, represented by the communications of a client and a server deployed on the same Pod, are the highest. They are quite similar among them. Performances reached in the Pod-to-Service communications are lower than performances reached in the Pod-to-Pod communications: this is an expected trend, since some form of translation must be performed for packets directed to a virtual IP, before reaching the real backend Pod (this is also true for the

reverse path). An exception to the previous consideration is represented by Cilium: the Pod-to-Service communications seem to reach similar throughput values. The proposed solution reaches the highest performances in both the Pod-to-Pod and the Pod-to-Service cases.

The results in 5.2 show how the providers handle an increasing number of Kubernetes Services (and relative backend Pods) in the already discussed scenario.

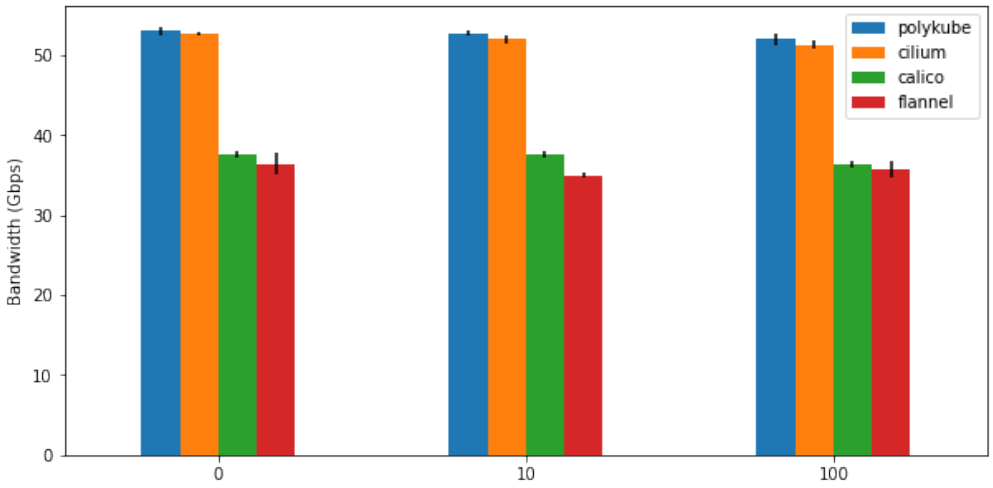

Figure 5.2: Same node throughput comparison for TCP traffic with an increasing number of Services/Pods

As the plot shows, increasing the number of Services does not influence too much the performances of the tested providers.

5.4.2 UDP

The results for UDP communications involving a client and a server deployed on the same node are shown in Fig. 5.3.

This scenario is similar to the previous one: also in this case, the proposed solution reaches the highest performances, even if the throughput differences are much lower than the ones shown in the TCP scenario.

Following the same approach used for TCP communications, the results in 5.4 show how the providers handle an increasing number of Kubernetes Services (and relative backends Pods) in UDP communications involving entities deployed on the same node.

As the plot shows, increasing the number of Services does not influence too much the performances of the tested providers: only a negligible performances drop can be point out for Cilium in case of 100 additional Services/Pods.

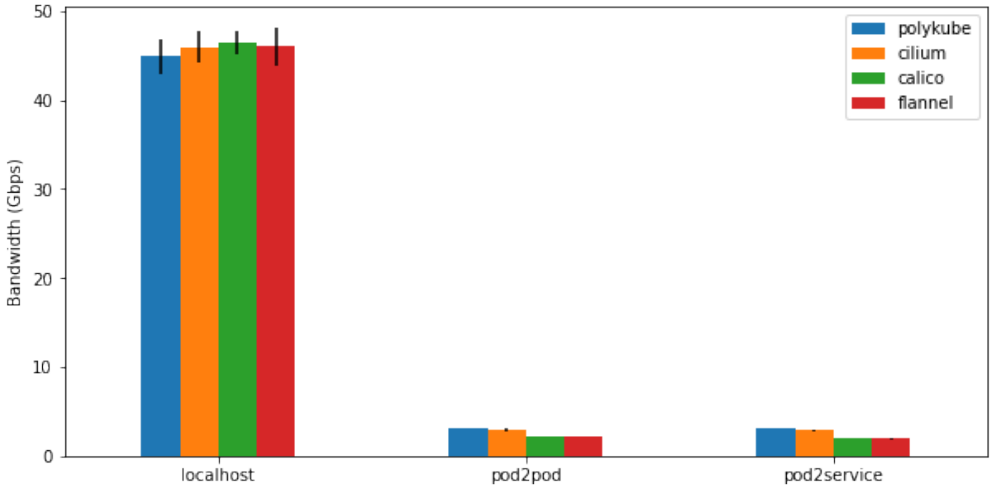

Figure 5.3: Same node throughput comparison for UDP traffic

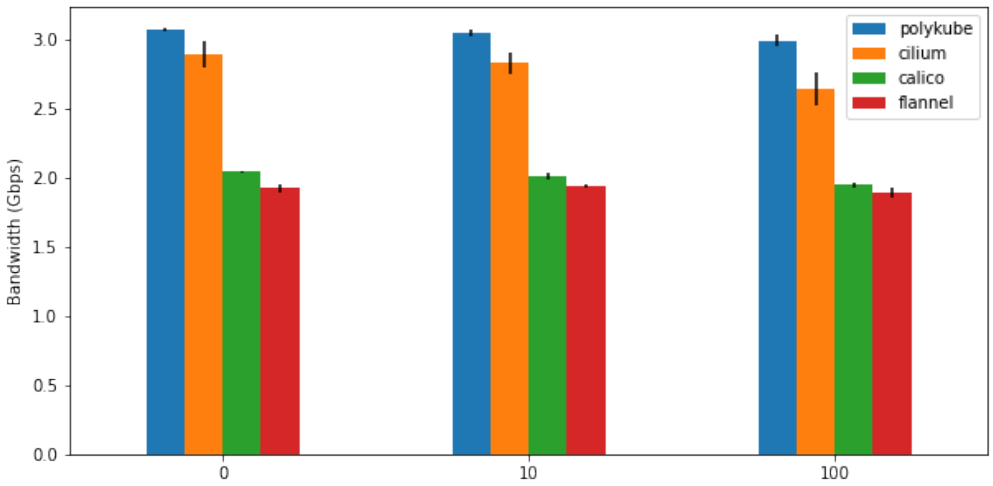

Figure 5.4: Same node throughput comparison for UDP traffic with an increasing number of Services/Pods

5.5 Communications on different nodes

5.5.1 TCP

Results for TCP communications involving a client and a server deployed on different nodes are shown in Fig. 5.5.

The results are different with respect to the one seen for the communications inside the same node. The performances of the proposed solution and the ones

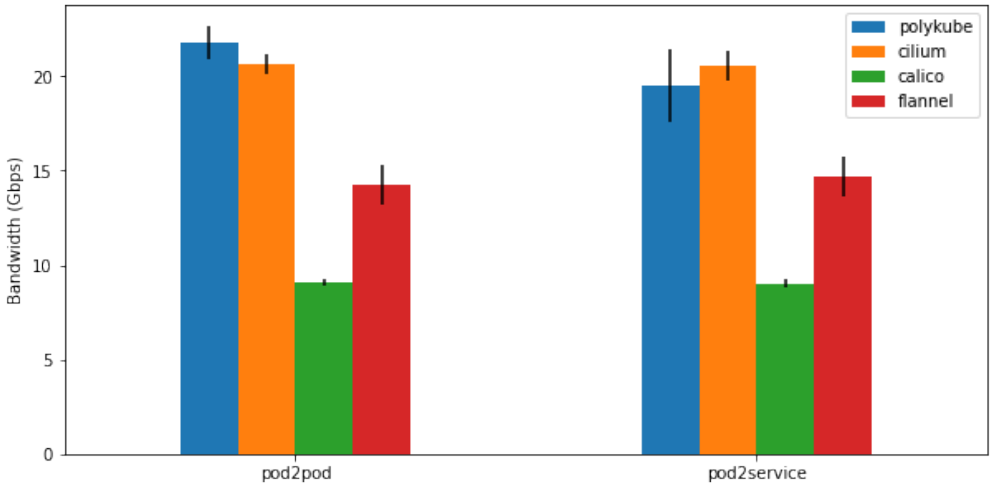

Figure 5.5: Different nodes throughput comparison for TCP traffic

reached by Cilium are the highest. Cilium reaches higher throughput values than the proposed solution on the Pod-to-Service communications, but lower values on the Pod-to-Pod ones. However, the high standard deviations suggest that the performances are not so different between the two.

The results in 5.6 show how the providers handle an increasing number of Kubernetes Services (and relative backend Pods) in the already discussed scenario.

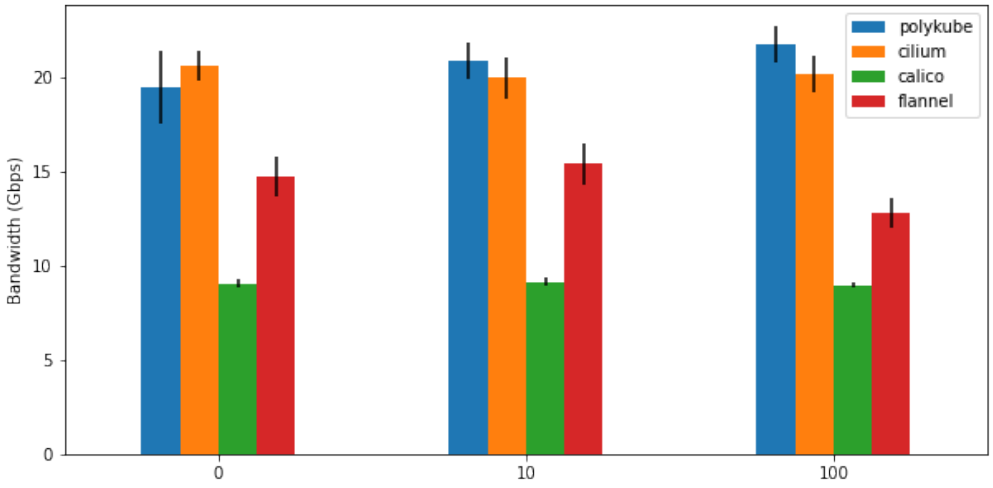

Figure 5.6: Different nodes throughput comparison for TCP traffic with an increasing number of Services/Pods

The bars plot confirm that the performances reached by both the proposed

solution and Cilium are the highest. The two providers performances are not so different: indeed, even if the average throughput value is lower in the scenario with 0 additional Services/Pods for the proposed solution, it tends to be higher in the scenarios with 10 or 100 additional Services/Pods; moreover, notice that the standard deviation is lower in these latter two cases. It is interesting to notice how Flannel performances decrease with 100 additional Services/Pods.

5.5.2 UDP

The results for UDP communications involving a client and a server deployed on different nodes are shown in Fig. 5.7.

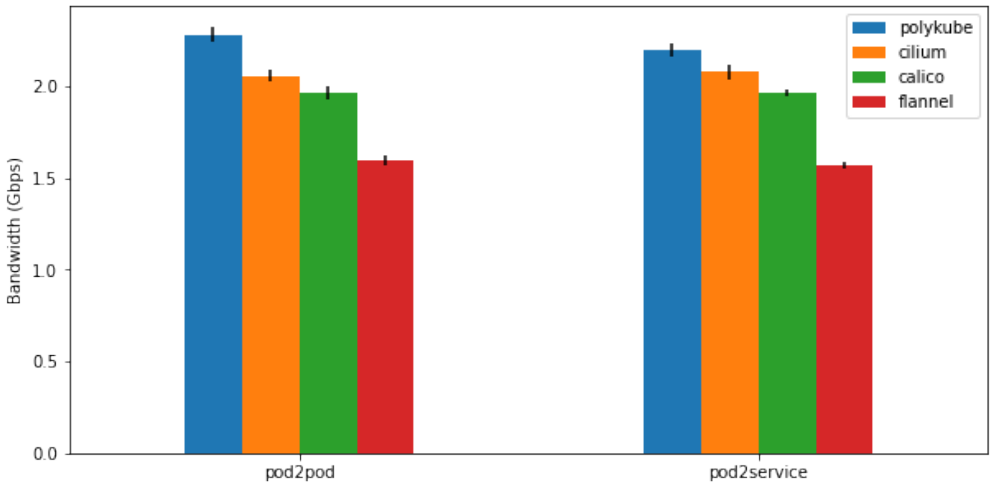

Figure 5.7: Different nodes throughput comparison for UDP traffic

This scenario is similar to the one involving UDP communications between entities on the same node. Also in this case, the proposed solution reaches the highest performances, followed by Cilium, Calico and Flannel.

The latter plot shown in 5.8 analyzes how the providers handle an increasing number of Kubernetes Services (and relative backends Pods) in UDP communications involving entities deployed on different nodes.

As expected, increasing the number of Services does not influence too much the performances of the tested providers.

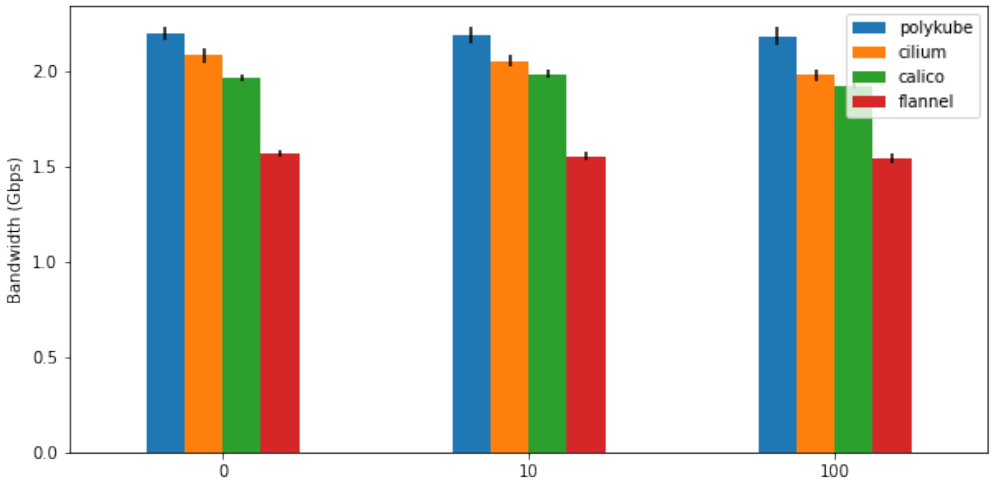

Figure 5.8: Different nodes throughput comparison for UDP traffic with an increasing number of Services/Pods

Chapter 6

Conclusions

The purpose of this thesis work is to present an high performance Kubernetes network provider built upon the eBPF technology by exploiting the Polycube framework. All the tasks regarding the deployment, the connection, the healing and the configuration of the virtual network topology have been demanded to an operator, deployed on each node of the Kubernetes cluster, specifically built for this project. The proposed virtual network topology architecture is made up three Polycube services: `pcn-k8slbrp`, `pcn-router` and `pcn-k8sdispatcher`. The enhancements proposed on the base virtual network topology solve the efficiency and scalability problems of the base architecture; moreover, they allow the communication between host processes and Pods deployed on the same node. The modular feature of the project allows to update existing services or insert new network functions to get more features such as security, network policies and/or observability.

6.1 Future works

The choice of Open Source components, their modularity and their ease of integration, leave wide space for future work, both from an implementation and research point of view. Some of the future works that can be based on this thesis work can be the following. The support for Kubernetes Network Policies can be introduced by specifically building a Polycube service able to understand and enforce them on the traffic. The actual inter-node communication is based on the VxLAN technology: one interesting improvement could be introducing the support for Direct Routing. Other features could go in the direction of extending the observability and the security exploiting also the XDP hook points. Finally, conformance tests could be conducted in order to ensure that, by using the proposed solution, the Kubernetes cluster is properly configured and that its behavior conforms to official Kubernetes specifications.

Appendix A

Scripts and Commands

In order to perform the evaluation discussed in 5, a specific folder structure and a file naming convention have been defined: each JSON file output by iPerf3, in order to be correctly processed, must have a name following this convention and reside in a path compliant to this structure. The defined convention is the following: `{scope}_{proto}/{provider}/{mode}/{mode}_{proto}_{#}.json`. In this template:

- `{scope}` can assume the value `same_node` or `diff_nodes`
- `{proto}` can assume the value `tcp` or `udp`
- `{provider}` can assume the value
  - `localhost`, `pod2pod`, `pod2service` if `scope=same_node`
  - `pod2pod`, `pod2service` if `scope=diff_nodes`
- `{#}` can assume values from 0 to 9 (it represent the specific measurement instance, since the measurement is repeated 10 times for each setting)

In the specific case of `mode=pod2service`, the convention is different:
`{scope}_{proto}/{provider}/{mode}/{n}/{mode}_{proto}_{#}.json`
Here, `{n}` can assume value 0, 10 or 100: it represents the number of additional Kubernetes Services/Pods considered during the specific test.

The scripts shown in A.1, A.2, A.3 are used to produce the results that follow the described structure.

Listing A.1: localhost.sh

```
1 #!/bin/bash
2
3 PROV=$1
```

```
4
5  iperf3 -s localhost &
6  pid=$!
7
8  function cleanup {
9      set +e
10     kill $pid
11 }
12 trap cleanup EXIT
13 set -x
14 set -e
15
16 TCP_FOLDER=same_node_tcp/$PROV/localhost
17 UDP_FOLDER=same_node_udp/$PROV/localhost
18
19 # LOCALHOST
20 mkdir -p $TCP_FOLDER
21 mkdir -p $UDP_FOLDER
22 for i in {0..9}; do
23 ## TCP
24     iperf3 -c localhost -t 10 -Z -J > $TCP_FOLDER/localhost_tcp_${i}.
       json
25     sleep 5
26 ## UDP
27     iperf3 -c localhost -u -b 0 -t 10 -Z -J > $UDP_FOLDER/
       localhost_udp_${i}.json
28     sleep 5
29 done
```

Listing A.2: pod2pod.sh

```
1  #!/bin/bash
2  set -x
3  set -e
4
5  SERVER_IP=$1
6  PREFIX=$2
7  PROV=$3
8
9  TCP_FOLDER=${PREFIX}_tcp/$PROV/pod2pod
10 UDP_FOLDER=${PREFIX}_udp/$PROV/pod2pod
11
12 # Pod to pod
13 mkdir -p $TCP_FOLDER
14 mkdir -p $UDP_FOLDER
15
16 for i in {0..9}; do
17 ## TCP
```

```
18          iperf3 -c $SERVER_IP -t 10 -Z -J > $TCP_FOLDER/pod2pod_tcp_${
    i}.json
19          sleep 5
20 ## UDP
21          iperf3 -c $SERVER_IP -u -b 0 -t 10 -Z -J > $UDP_FOLDER/
    pod2pod_udp_${i}.json
22          sleep 5
23 done
```

Listing A.3: pod2service.sh

```
1  #!/bin/bash
2  set -x
3  set -e
4
5  CLUSTER_IP=$1
6  PREFIX=$2
7  PROV=$3
8  SVC_NUM=$4
9
10 TCP_FOLDER=${PREFIX}_tcp/$PROV/pod2service/$SVC_NUM
11 UDP_FOLDER=${PREFIX}_udp/$PROV/pod2service/$SVC_NUM
12
13 # Pod to service
14 mkdir -p $TCP_FOLDER
15 mkdir -p $UDP_FOLDER
16
17 for i in {0..9}; do
18 ## TCP
19      iperf3 -c $CLUSTER_IP -t 10 -Z -J > $TCP_FOLDER/pod2service_tcp_$
    {i}.json
20      sleep 5
21 ## UDP
22      iperf3 -c $CLUSTER_IP -u -b 0 -t 10 -Z -J > $UDP_FOLDER/
    pod2service_udp_${i}.json
23      sleep 5
24 done
```

The Python script in A.4 contains the definition of the functions and some call example for generating the bars plots in a Jupyter Notebook environment

Listing A.4: Bars plots script

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import re
4 import json
5 import os
6 import sys
7 import pandas as pd
```

```
8
9  path="./test_folder"
10 path=os.path.abspath(path)
11
12 providers = ["polykube", "cilium", "calico", "flannel"]
13
14 def get_per_mode_data(path, scope, proto):
15     data = {}
16     folder = f"{scope}_{proto}"
17     modes = ["pod2pod", "pod2service"] if scope == "diff_nodes" else
       ["localhost", "pod2pod", "pod2service"]
18
19     for provider in providers:
20         data[provider] = {"means": [], "stds": []}
21         for mode in modes:
22             bandwidths = []
23             for i in range(10):
24                 filename = f"{mode}_{proto}_{i}.json"
25                 if mode != "pod2service" and mode != "ext2int":
26                     file_path = os.path.join(path, folder, provider,
       mode, filename)
27                 else:
28                     file_path = os.path.join(path, folder, provider,
       mode, "0", filename)
29                 with open(file_path, "rb") as f:
30                     decodedjson = json.load(f)
31                     key = "sum_received" if proto == "tcp" else "sum"
32                     bandwidths.append(decodedjson["end"][key]["
       bits_per_second"]/1e9)
33             data[provider]["means"].append(np.mean(bandwidths))
34             data[provider]["stds"].append(np.std(bandwidths))
35     return data, modes
36
37 def get_per_load_data(path, scope, proto, mode):
38     data = {}
39     folder = f"{scope}_{proto}"
40     loads = ["0", "10", "100"]
41
42     for provider in providers:
43         data[provider] = {"means": [], "stds": []}
44         for load in loads:
45             bandwidths = []
46             for i in range(10):
47                 filename = f"{mode}_{proto}_{i}.json"
48                 file_path = os.path.join(path, folder, provider, mode
       , load, filename)
49                 with open(file_path, "rb") as f:
50                     decodedjson = json.load(f)
51                     key = "sum_received" if proto == "tcp" else "sum"
```

```
52                      bandwidths.append(decodedjson["end"][key]["
    bits_per_second"]/1e9)
53              data[provider]["means"].append(np.mean(bandwidths))
54              data[provider]["stds"].append(np.std(bandwidths))
55      return data, loads
56
57  def plot_data(data, index, color="black"):
58      means = {}
59      stds = {}
60      for provider in providers:
61          means[provider] = data[provider]["means"]
62          stds[provider] = data[provider]["stds"]
63      df = pd.DataFrame(means, index=index)
64      ax = df.plot.bar(rot=0, yerr=stds, figsize=(10, 5))
65      ax.tick_params(colors=color, which='both')
66      ax.set_ylabel("Bandwidth (Gbps)")
67      ax.yaxis.label.set_color(color)
68
69  # example 1
70  scope = "same_node"
71  proto = "tcp"
72  per_mode_data, modes = get_per_mode_data(path, scope, proto)
73  plot_data(per_mode_data, modes, "black")
74
75  # example 2
76  scope = "diff_nodes"
77  proto = "udp"
78  mode = "pod2service"
79  per_load_data, loads = get_per_load_data(path, scope, proto, mode)
80  plot_data(per_load_data, loads, "black")
```

The `get_per_mode_data` was used to extract the data needed for producing plots collecting together the various mode tests. The `get_per_load_data` was used to extract the data needed for producing plots collecting together the various pod2service tests varying the number of additional Kubernetes Services/Pods.

Bibliography

[1] Hamza Rhaouati. «Prototyping a Network Provider for Kubernetes through Disaggregated eBPF Services». Luglio 2021. URL: http://webthesis.biblio.polito.it/19112/ (cit. on pp. 2, 30, 32, 43).

[2] Eric Carter. *Sysdig 2019 Container Usage Report: New Kubernetes and security insights*. Oct. 2019. URL: https://sysdig.com/blog/sysdig-2019-container-usage-report/ (cit. on p. 6).

[3] etcd Authors. *etcd - Homepage*. URL: https://etcd.io/ (cit. on p. 9).

[4] The Kubernetes Authors. *Pods | Kubernetes*. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (cit. on p. 13).

[5] The Kubernetes Authors. *ReplicaSet | Kubernetes*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/ (cit. on p. 14).

[6] The Kubernetes Authors. *Deployments | Kubernetes*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (cit. on p. 15).

[7] The Kubernetes Authors. *DaemonSet | Kubernetes*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/ (cit. on p. 16).

[8] The Kubernetes Authors. *Service | Kubernetes*. URL: https://kubernetes.io/docs/concepts/services-networking/service/ (cit. on p. 18).

[9] The Kubernetes Authors. *Services, Load Balancing, and Networking | Kubernetes*. URL: https://kubernetes.io/docs/concepts/services-networking/ (cit. on p. 19).

[10] containernetworking. *CNI - the Container Network Interface*. URL: https://github.com/containernetworking/cni (cit. on p. 20).

[11] The Cilium Authors. *BPF and XDP Reference Guide*. URL: https://docs.cilium.io/en/v1.11/bpf/ (cit. on pp. 20, 25).

[12] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. «Creating complex network services with ebpf: Experience and lessons learned». In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, pp. 1–8 (cit. on p. 24).

[13] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, Jianwen Pi, and Aasif Shaikh. «A Service-Agnostic Software Framework for Fast and Efficient in-Kernel Network Services». In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–9 (cit. on p. 26).

[14] *Virtual Extensible LAN*. URL: `https://en.wikipedia.org/wiki/Virtual_Extensible_LAN` (cit. on p. 28).

[15] Mallik Mahalingam, Dinesh Dutt, Kenneth Duda, Puneet Agarwal, Larry Kreeger, T. Sridhar, Mike Bursell, and Chris Wright. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. Aug. 2014. DOI: `10.17487/RFC7348`. URL: `https://www.rfc-editor.org/info/rfc7348` (cit. on pp. 28, 32).

[16] Daniel E. Eisenbud et al. «Maglev: A Fast and Reliable Software Network Load Balancer». In: *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI'16. Santa Clara, CA: USENIX Association, 2016, pp. 523–535. ISBN: 9781931971294 (cit. on p. 30).

[17] Swagger Authors. *OpenAPI Specification*. URL: `https://swagger.io/specification/` (cit. on p. 38).

[18] Hamza Rhaouati. *pcn-k8sdispatcher implementation*. URL: `https://github.com/ReddaHawk/polycube/tree/hmz/k8s-operator/src/services/pcn-k8sdispatcher` (cit. on p. 43).

[19] OperatorSDK Authors. *Operator SDK*. URL: `https://sdk.operatorframework.io/` (cit. on p. 60).

[20] Leonardo Di Giovanna. *Polykube Operator implementation*. URL: `https://github.com/ekoops/polykube-operator` (cit. on p. 60).

[21] iPerf Authors. *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. URL: `https://iperf.fr/` (cit. on p. 77).

Acknowledgements

At the end of such a long path, I can find a moment to dedicate entirely to the people who supported, accompanied or even only observed it for a while.

To Professor Fulvio Risso, who fueled my passion for networking during the Cloud Computing course. His passion and ability to explain complex concepts in such a simple way have rekindled in me the love of knowledge that, perhaps with the passage of time, had been replaced by the desire to finish university as soon as possible.

To Giuseppe Ognibene and Federico Parola, who endured my outbursts and offered their expertise to help me finish this thesis work. To my colleagues/friends Giuseppe, Andrea, Riccardo, Valerio, Gianmarco and Vincenzo, who shared with me the anxieties and joys of the university world.

To my friends Gaetano, Fabio, Domenico, Domenico, Giuseppe, Baldo and Salvatore, who have always supported me emotionally and have always believed in my abilities.

Last but not least, my best thanks go to my family. To my parents, that always supported my passion and did everything they could to allow me to pursue my dreams. To my gradparents, that together with my parents taught me the value of love and goodness of mind.

Leonardo Di Giovanna