POLITECNICO DI TORINO

Master's Degree in Computer engineering

Master's Degree Thesis

Prototyping a Network Provider for Kubernetes through Disaggregated eBPF Services

Supervisors

Prof. Fulvio RISSO

Ing. Federico PAROLA

Candidate

Hamza RHAOUATI

Academic year 2020-2021

Summary

Containerisation and microservices architecture are getting momentum in nowadays ICT field. Microservices have a demand on a high number of containers which requires orchestration and interconnection. Kubernetes, an open-source container orchestration platform, has been widely adopted by cloud service providers (CSPs) for its advantages in simplifying container deployment, scalability and scheduling. Networking is one of the central components of Kubernetes, providing connectivity between different pods (group of containers) both within the same host and across hosts. Today's network infrastructure is increasingly implemented using NFV (Network Function Virtualization) technology where network services are implemented in pure software. This brings several advantages such as flexibility and cost reduction as these functions can be performed on general purpose hardware. In this context, eBPF (Extended Berkeley Packet Filter) is an excellent technology, suitable for creating network functions for fast packet processing in the Linux kernel. This thesis aims at defining and validating a modular network plugin for Kubernetes that leverages a set of disaggregated network services. This would be achieved using Polycube, an open source software framework for Linux developed at Politecnico di Torino, which is used for the creation of extremely fast network services programs and interconnect them. The modular feature of this project allows you to insert new network functions to get more features such as security, network policies and observability.

Table of Contents

List of Figures

Chapter 1

Introduction

In the past, organizations ran applications on physical servers. There was no way to define resource boundaries for applications in a physical server, and this caused resource allocation issues. For example, if multiple applications run on a physical server, there can be instances where one application would take up most of the resources, and as a result, the other applications would underperform. A solution for this would be to run each application on a different physical server. But this did not scale as resources were underutilized, and it was expensive for organizations to maintain many physical servers. As a solution, virtualization was introduced. Virtualization allows better utilization of resources in a physical server and allows better scalability because an application can be added or updated easily, reduces hardware costs, and much more. Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Lower isolation allows containers to be lighter than VMs. Containerization helps to package software, enabling applications to run unmodified in a wide range of Linux distributions and to be released and updated in an easy and fast way without downtime. Kubernetes, as a microservice orchestrator, helps developers to make sure those containerized applications run in cloud environment, providing all the resources and tools they need to work properly. Kubernetes networking allows Kubernetes components to communicate with each other and with other applications. The Kubernetes platform is different from other networking platforms because it is based on a flat network structure that eliminates the need to map host ports to container ports. The Kubernetes platform provides a way to run distributed systems, sharing machines between applications without dynamically allocating ports.

1.1 Goal of the thesis

Managing a network where containers can interoperate efficiently is very important. Kubernetes has adopted the Container Network Interface (CNI) specification for managing network resources on a cluster. This relatively simple specification makes it easy for Kubernetes to interact with a wide range of CNI-based software solutions [1]. A CNI plugin is responsible for inserting a network interface into the container network namespace (e.g., one end of a virtual ethernet (veth) pair) and making any necessary changes on the host (e.g., attaching the other end of the veth into a bridge). It then assigns an IP address to the interface and sets up the routes consistent with the IP Address Management section by invoking the appropriate IP Address Management (IPAM) plugin[2].

This thesis studies the possibility to use eBPF (Extended Berkeley Packet Filter), a novel technology that allows to run fast network functions in the Linux kernel, to prototype a network provider for kubernetes through disaggregated eBPF Services. For eBPF Services we means Network Functions such as Switch, Router, Load Balancer and NAT (Network Address Translation). Those are interconnected together to provide connectivity to containers. Thanks to eBPF and Polycube we have a secure method to ensure speed and performance, observability and security.

Chapter 2

Background

This chapter provides a description of the main elements and key technologies this thesis is based on.

First, Kubernetes is introduced, highlighting its main features giving an enphasis on the Kubernetes Network Model. Then, a detailed description of the two main technologies used in the project, eBPF and Polycube, is provided. In the end, an overview of related work studying eBPF Network Functions needed and how are interconnected.

2.1 Kubernetes: introduction

Around 2004, Google created the **Borg** [3] system, a small project with less than 5 people initially working on it. The project was developed as a collaboration with a new version of Google's search engine. Borg was a large-scale internal cluster management system, which "ran hundreds of thousands of jobs, from many thousands of different applications, across many clusters, each with up to tens of thousands of machines". It achieves high utilization by combining admission control, efficient task-packing, over-commitment, and machine sharing with process-level performance isolation. [3].

In the middle of 2014, Google presented **Kubernetes** as on open-source version of Borg. Kubernetes was created by Joe Beda, Brendan Burns, and Craig McLuckie, and other engineers at Google. Its development and design were heavily influenced by Borg and many of its initial contributors previously used to work on it. The original Borg project was written in C++, whereas for Kubernetes the Go language was chosen.

In 2015 Kubernetes v1.0 was released. Along with the release, Google set up a partnership with the Linux Foundation to form the **Cloud Native Computing Foundation** (CNCF) [4]. Since then, Kubernetes has significantly grown, achieving the CNCF graduated status and being adopted by nearly every big company. Nowadays it has become the de-facto standard for container orchestration [5, 6].

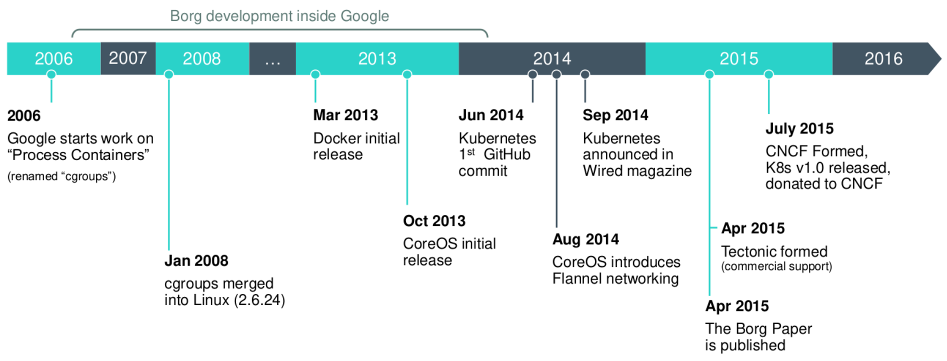

Borg development inside Google
2006
2007
2008
...
2013
2014
2015
2016
2006
Google starts work on "Process Containers"
(renamed "cgroups")
Jan 2008
cgroups merged into Linux (2.6.24)
Mar 2013
Docker initial release
Oct 2013
CoreOS initial release
Jun 2014
Kubernetes 1st GitHub commit
Aug 2014
CoreOS introduces Flannel networking
Sep 2014
Kubernetes announced in Wired magazine
Apr 2015
Tectonic formed
(commercial support)
Apr 2015
The Borg Paper is published
July 2015
CNCF Formed, K8s v1.0 released, donated to CNCF

Figure 2.1: Kubernetes History

2.2 Applications deployment evolution

Kubernetes is a portable, extensible, open-source platform for running and coordinating containerized applications across a cluster of machines. It is designed to completely manage the life cycle of applications and services using methods that provide consistency, scalability, and high availability.

What does "containerized applications" means? In the last decades, the deployment of applications has seen significant changes, which are illustrated in figure 2.2.

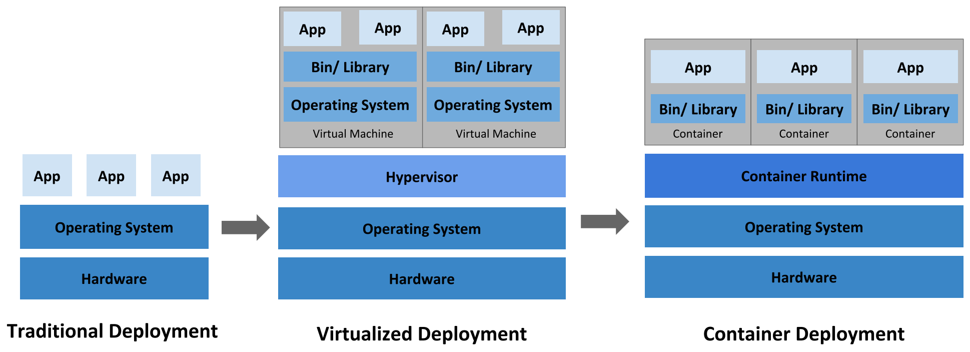


Figure 2.2: Evolution in applications deployment.

Traditionally, organizations used to run their applications on physical servers. One of the problems of this approach was that resource boundaries between applications could not be applied in a physical server, leading to resource allocation issues. For example, if multiple applications run on a physical server, one of them could take up most of the resources, and as a result, the other applications would starve. A possibility to solve this problem would be to run each application on a different physical server, but clearly it is not feasible: the solution could not scale, would lead to resources under-utilization and would be very expensive for organizations to maintain many physical servers.

The first real solution has been **virtualization**. Virtualization allows to run multiple Virtual Machines on a single physical server. It grants isolation of the applications between VMs providing a high level of security, as the information of one application cannot be freely accessed by another application. Virtualization enables better utilization of resources in a physical server, improves scalability, because an application can be added or updated very easily, reduces hardware costs, and much more. With virtualization it is possible to group together a set of physical resources and expose it as a cluster of disposable virtual machines. Isolation certainly brings many advantages, but it requires a quite 'heavy' overhead: each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

A second solution which has been proposed recently is **containerization**. Containers are similar to VMs, but they share the operating system with the host machine, relaxing isolation properties. Therefore, containers are considered a lightweight form of virtualization. Similarly to a VM, a container has its own filesystem, CPU, memory, process space etc. One of the key features of containers is that they are portable: as they are decoupled from the underlying infrastructure, they are totally portable across clouds and OS distributions. This property is particularly relevant nowadays with cloud computing: a container can be easily moved across different machines. Moreover, being "lightweight", containers are much faster than virtual machines: they can be booted, started, run and stopped with little effort and in a short time.

2.3 Container orchestrators

When hundreds or thousands of containers are created, the need of a way to manage them becomes essential; container orchestrators serve this purpose. A container orchestrator is a system designed to easily manage complex containerization deployments across multiple machines from one central location. As depicted in figure 2.3, Kubernetes is by far the most used container orchestrator. We provide a description of such system in the following.

Kubernetes provides many services, including:

- **Service discovery and load balancing** A container can be exposed using the DNS name or using its own IP address. If traffic to a container is high, a load balancer able to distribute the network traffic is provided.

- **Storage orchestration** A storage system can be automatically mounted, such as local storages, public cloud providers, and more.

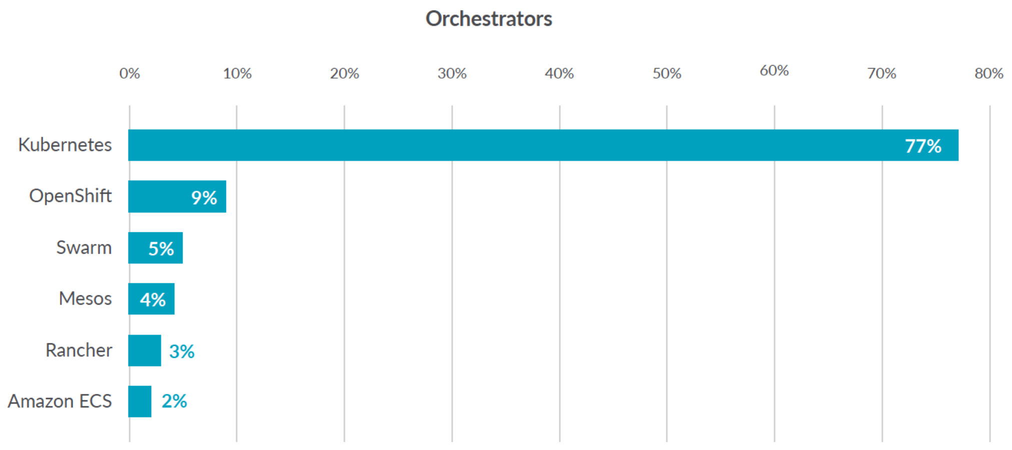


Figure 2.3: Container orchestrators use [7].

- **Automated rollouts and rollbacks** The desired state for the deployed containers can be described, and the actual state can be changed to the desired state at a controlled rate. For example, it is possible to automate the creation of new containers of 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 can be used to run containerized tasks. It is possible to set how much CPU and memory (RAM) each container needs, and automatically the containers are sized to fit in the nodes to make the best use of the resources.

- **Secret and configuration management** It is possible to store and manage sensitive information in Kubernetes, such as passwords, OAuth tokens, and SSH keys. It is possible to deploy and update secrets and application configuration without rebuilding the container images, and without exposing secrets in the stack configuration.

2.4 Kubernetes architecture

When Kubernetes is deployed, a cluster is created. A Kubernetes cluster consists of a set of machines, called **nodes**, that run containerized applications. At least one of the nodes hosts the control plane and is called **master**. Its role is to manage the cluster and expose an interface to the user. The **worker** node(s) host the **pods** that are the components of the application. The master manages the worker nodes and the pods in the cluster. In production environments, the control plane usually runs across multiple machines and a cluster runs on multiple nodes, providing fault-tolerance and high availability.

Figure 2.4 shows the diagram of a Kubernetes cluster with all the components linked together.

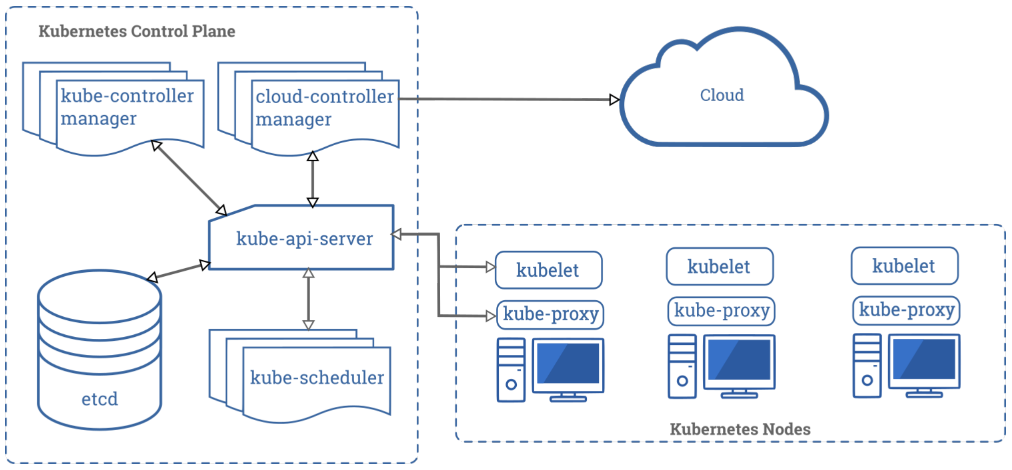


Figure 2.4: Kubernetes architecture

2.4.1 Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new pod). Although they can be run on any machine in the cluster, for simplicity, they are typically executed all together on the same machine, which does not run user containers.

API server

The API server is the component of the Kubernetes control plane that exposes the Kubernetes REST API, and constitites the front end for the Kubernetes control plane. Its function is to intercept REST request, validate and process them. The main implementation of a Kubernetes API server is `kube-apiserver`. It is designed to scale horizontally, which means it scales by deploying more instances. Moreover, it can be easily redounded to run several instances of it and balance traffic among them.

etcd

`etcd` is a distributed, consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. It is based on the Raft consensus algorithm [8], which allows different machines to work as a coherent group and survive to the breakdown of one of its members. `etcd` can be stacked in the master node or external, installed on dedicated host. Only the API server can communicate with it.

Scheduler

The scheduler is the control plane component responsible of assigning the pods to the nodes. The one provided by Kubernetes is called `kube-scheduler`, but it can be customized by adding new schedulers and indicating in the pods to use them. `kube-scheduler` watches for newly created pods not assigned to a node yet, and selects one for them to run on. To make its decisions, it considers singular and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

kube-controller-manager

Component that runs controller processes. It continuously compares the desired state of the cluster (given by the objects specifications) with the current one (read from `etcd`). 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. These controllers include:

- Node Controller: responsible for noticing and reacting when nodes go down.
- Replication Controller: in charge of maintaining the correct number of pods for every replica object in the system.
- Endpoints Controller: populates the Endpoint objects (which links Services and Pods).
- Service Account & Token Controllers: create default accounts and API access tokens for new namespaces.

cloud-controller-manager

This component runs controllers that interact with the underlying cloud providers. The `cloud-controller-manager` binary is a beta feature introduced in Kubernetes 1.6. It only runs cloud-provider-specific controller loops. You can disable these controller loops in the `kube-controller-manager`.

`cloud-controller-manager` allows the cloud vendor's code and the Kubernetes code to evolve independently of each other. In prior releases, the core Kubernetes code was dependent upon cloud-provider-specific code for functionality. In future releases, code specific to cloud vendors should be maintained by the cloud vendor themselves, and linked to `cloud-controller-manager` while running Kubernetes. Some examples of controllers with cloud provider dependencies are:

- Node Controller: checks the cloud provider to update or delete Kubernetes nodes using cloud APIs.
- Route Controller: responsible for setting up network routes in the cloud infrastructure.
- Service Controller: for creating, updating and deleting cloud provider load balancers.
- Volume Controller: creates, attaches, and mounts volumes, interacting with the cloud provider to orchestrate them.

2.4.2 Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

Container Runtime

The `container runtime` is the software that is responsible for running containers. Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

kubelet

An agent that runs on each node in the cluster, making sure that containers are running in a pod. The `kubelet` receives from the API server the specifications of the Pods and interacts with the `container runtime` to run them, monitoring their state and assuring that the containers are running and healthy. The connection with the `container runtime` is established through the Container Runtime Interface and is based on gRPC.

kube-proxy

`kube-proxy` is a network agent that runs on each node in your cluster, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, which allow network communication to your Pods from inside or outside of the cluster. If the operating system is providing a packet filtering layer, `kube-proxy` uses it, otherwise it forwards the traffic itself.

Addons

Features and functionalities not yet available natively in Kubernetes, but implemented by third parties pods. Some examples are DNS, dashboard (a web gui), monitoring and logging.

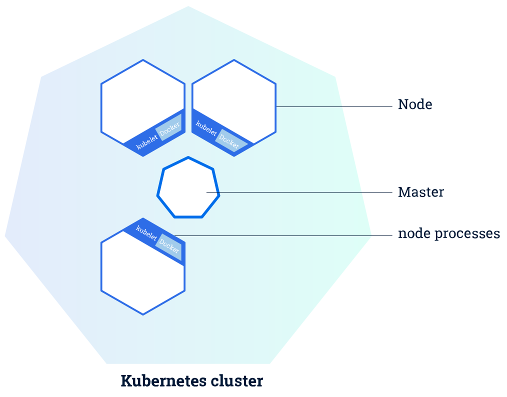


Figure 2.5: Kubernetes master and worker nodes [9].

2.5 Kubernetes objects

Kubernetes defines several types of objects, which constitutes its building blocks. Usually, a K8s resource object contains the following fields [10]:

- `apiVersion`: the versioned schema of this representation of the object;
- `kind`: a string value representing the REST resource this object represents;
- `ObjectMeta`: metadata about the object, such as its name, annotations, labels etc.;
- `ResourceSpec`: defined by the user, it describes the desired state of the object;
- `ResourceStatus`: filled in by the server, it reports the current state of the resource.

The allowed operations on these resources are the typical CRUD actions:

- **Create**: create the resource in the storage backend; once a resource is created, the system applies the desired state.
- **Read**: comes with 3 variants
  - **Get**: retrieve a specific resource object by name;
  - **List**: retrieve all resource objects of a specific type within a namespace, and the results can be restricted to resources matching a selector query;
  - **Watch**: stream results for an object(s) as it is updated.
- **Update**: comes with 2 forms
  - **Replace**: replace the existing spec with the provided one;
  - **Patch**: apply a change to a specific field.
- **Delete**: delete a resource; depending on the specific resource, child objects may or may not be garbage collected by the server.

In the following we illustrate the main objects needed in the next chapters.

2.5.1 Label & Selector

Labels are key-value pairs attached to a K8s object and used to organize and mark a subset of objects. Selectors are the grouping primitives which allow to select a set of objects with the same label.

2.5.2 Namespace

Namespaces are virtual partitions of the cluster. By default, Kubernetes creates 4 Namespaces:

- **kube-system**: it contains objects created by K8s system, mainly control-plane agents;
- **default**: it contains objects and resources created by users and it is the one used by default;
- **kube-public**: readable by everyone (even not authenticated users), it is used for special purposes like exposing cluster public information;
- **kube-node-lease**: it maintains objects for heartbeat data from nodes.

It is a good practice to split the cluster into many Namespaces in order to better virtualize the cluster.

2.5.3 Pod

Pods are the basic processing units in Kubernetes. A pod is a logic collection of one or more containers which share the same network and storage, and are scheduled together on the same pod. Pods are ephemeral and have no auto-repair capacities: for this reason they are usually managed by a controller which handles replication, fault-tolerance, self-healing etc.

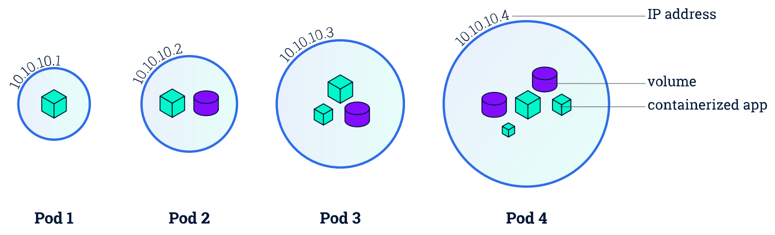


Figure 2.6: Kubernetes pods [9].

2.5.4 ReplicaSet

ReplicaSets control a set of pods allowing to scale the number of pods currently in execution. If a pod in the set is deleted, the ReplicaSet notices that the current number of replicas (read from the `Status`) is different from the desired one (specified in the `Spec`) and creates a new pod. Usually ReplicaSets are not used directly: a higher-level concept is provided by Kubernetes, called **Deployment**.

2.5.5 Deployment

Deployments manage the creation, update and deletion of pods. A Deployment automatically creates a ReplicaSet, which then creates the desired number of pods. For this reason an application is typically executed within a Deployment and not in a single pod. The listing 2.1 is an example of deployment.

Listing 2.1: Basic example of Kubernetes Deployment [9].

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

The code above allows to create a Deployment with name `nginx-deployment` and a label `app`, with value `nginx`. It creates three replicated pods and, as defined in the `selector` field, manages all the pods labelled as `app:nginx`. The template field shows the information of the created pods: they are labelled `app:nginx` and launch one container which runs the nginx DockerHub image at version 1.7.9 on port 80.

2.5.6 Service

A Service is an abstract way to expose an application running on a set of Pods as a network service. It can have different access scopes depending on its ServiceType:

- **ClusterIP**: Service accessible only from within the cluster, it is the default type;
- **NodePort**: exposes the Service on a static port of each Node's IP; the

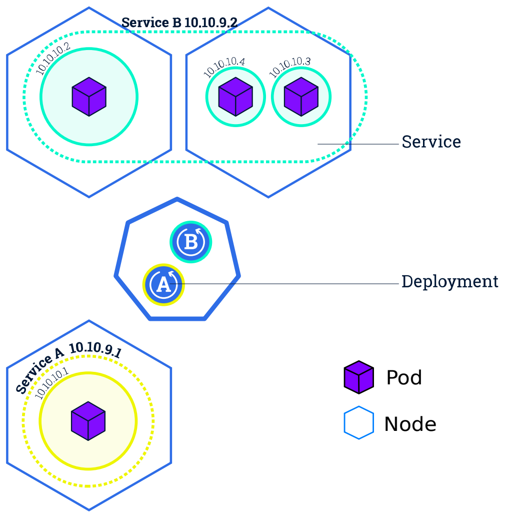


Figure 2.7: Kubernetes Services [9].

NodePort Service can be accessed, from outside the cluster, by contacting `<NodeIP>:<NodePort>`;

- **LoadBalancer**: exposes the Service externally using a cloud provider's load balancer;
- **ExternalName**: maps the Service to an external one so that local apps can access it.

The following Service is named `my-service` and redirects requests coming from TCP port 80 to port 9376 of any Pod with the `app=MyApp` label.

Listing 2.2: Basic example of Kubernetes Service [9].

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

2.6 Kubernetes Operator

A Kubernetes operator is an application-specific controller that extends the functionality of the Kubernetes API to create, configure, and manage instances of complex applications on behalf of a Kubernetes user.

It builds upon the basic Kubernetes resource and controller concepts, but includes domain or application-specific knowledge to automate the entire life cycle of the software it manages.

Controllers are the core abstraction used to build Kubernetes. Once you've declared the desired state of your cluster using the API server, controllers ensure that the cluster's current state matches the desired state by continuously watching the state of the API server and reacting to any changes. Controllers operate using a simple loop that continuously checks the current state of the cluster against the desired state of the cluster. If there are any differences, controllers perform tasks to make the current state match the desired state. In pseudo-code:

Listing 2.3: Controller loop example

```
while true:
 X = currentState()
 Y = desiredState()

  if X == Y:
    return  # Do nothing
  else:
    do(tasks to get to Y)
```

For example, when you create a new Pod using the API server, the Kubernetes scheduler (a controller) notices the change and makes a decision about where to place the Pod in the cluster. It then writes that state change using the API server (backed by etcd). The kubelet (a controller) then notices that new change and sets up the required networking functionality to make the Pod reachable within the cluster. Here, two separate controllers react to two separate state changes to make the reality of the cluster match the intention of the user.

2.7 CNI - the Container Network Interface

CNI (Container Network Interface), a Cloud Native Computing Foundation project, consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted. Kubernetes first creates a container without a network interface and then calls a CNI plug-in. The plug-in configures container networking and returns information about allocated network interfaces, IP addresses, etc. The parameters that Kubernetes sends to a CNI plugin, as well as the structure of the response must satisfy the CNI specification, but the plug-in itself may do whatever it needs to do its job.

2.8 eBPF (Extended Berkeley Packet Filter)

Initially proposed by Alexei Staravoitov in 2013, Extended Berkeley Packet Filter (eBPF) [11] is a virtual machine integrated into the Linux Kernel that allows to execute custom bytecode injected at runtime in an event-based way. eBPF was introduced in Kernel 3.18 and is the evolution of the classic Berkeley Packet Filter (cBPF), once simply known as BPF.

BPF was born in 1992 and was a very simple VM used to perform in-kernel packet filtering. BPF provides on some Unix-like OSes a raw interface to data link layers in a protocol-independent fashion, and the potential to operate with custom code on the intercepted packets. All packets on the network, even those intended for other hosts, are accessible through this mechanism, provided that the network driver support promiscuous mode. BPF roughly offers a service similar to raw sockets, but it provides packet access through a file interface ratherthan a network interface. The network TAP, a component in the lower layers of the networking stack, copied packets received by network interfaces to the BPF filter, where injected bytecode decided whether the packet needed to be sent to the user space. Matching packets were inserted into a buffer, that could be read by a user space program, such as Tcpdump, through a dedicated API.

eBPF extends BPF making it general-purpose, and now program can operate and modify the packet content, hence enabling a new breed of applications such as bridging, routing, NATting, and more. The “Classic” BPF is not used anymore,and legacy applications are adapted from the BPF bytecode to the eBPF. It is an interesting technology not only for packet processing, but also for other aspects like security management and kernel monitoring.

This technology allows to extend the Linux kernel in a easy way, without writing kernel modules that must be builded inside the kernel.

Main features of eBPF are discussed in following sections.

C-based programming

eBPF code can be written in (a restricted version of) C, which allows for easier program development and more powerful functionalities with respect to bare assembly.

2.8.1 vCPU Architecture

BPF is a general purpose RISC instruction set and was originally designed for the purpose of writing programs in a subset of C which can be compiled into BPF instructions through a compiler back end (e.g. LLVM), so that the kernel can later on map them through an in-kernel JIT compiler into native opcodes for optimal execution performance inside the kernel. BPF consists of eleven 64 bit registers with 32 bit subregisters, a program counter and a 512 byte large BPF stack space. Registers are named r0 - r10. The operating mode is 64 bit by default, the 32 bit subregisters can only be accessed through special ALU (arithmetic logic unit) operations. The 32 bit lower subregisters zero-extend into 64 bit when they are being written to.

Register r10 is the only register which is read-only and contains the frame pointer address in order to access the BPF stack space. The remaining r0 - r9 registers are general purpose and of read/write nature.

2.8.2 Safety

eBPF allows the injection of custom code at runtime that work in concert with the kernel, making use of existing kernel infrastructure (e.g. drivers, netdevices, tunnels, protocol stack, sockets) and tooling (e.g. iproute2) as well as the safety guarantees which the kernel provides. Unlike kernel modules, BPF programs are verified through an in-kernel verifier in order to ensure that they cannot crash the kernel, always terminate, etc.

2.8.3 Helpers

Helpers are sets of functions pre-compiled and ready to be used inside the Linux kernel. eBPF programs can call shuch functions, that are executed outside the eBPF context and therefore are not subject to its constraints. Available helper functions may differ for each BPF program type, for example, BPF programs attached to sockets are only allowed to call into a subset of helpers compared to BPF programs attached to the tc layer. Encapsulation and decapsulation helpers for lightweight tunneling constitute an example of functions which are only available to lower tc layers, whereas event output helpers for pushing notifications to user space are available to tc and XDP programs.

Each helper function is implemented with a commonly shared function signature similar to system calls. The signature is defined as:

Listing 2.4: Helper function signature

```
u64 fn(u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)
```

2.8.4 Maps

An eBPF program is triggered by a packet received by the virtual CPU. To store the packet in order to process it, eBPF defines a volatile “packet memory”, which is valid only for the current packet: this means there is no way to store information needed across subsequent packets. eBPF defines the concept of state with a set of memory areas, which are called maps. Maps are efficient key/value data structures that reside in kernel space. They can be accessed by eBPF programs through dedicated helper functions. They can be shared between different programs and can also be accessed by the user programs, providing an efficient way to exchange data between kernel and user space. The kernel guarantees safe concurrent access to maps using the Read-Copy-Update mechanism. An important side effect of using maps is that the state of the program is decoupled from the code. Instructions are in the program, the data used by such instructions are in the maps.

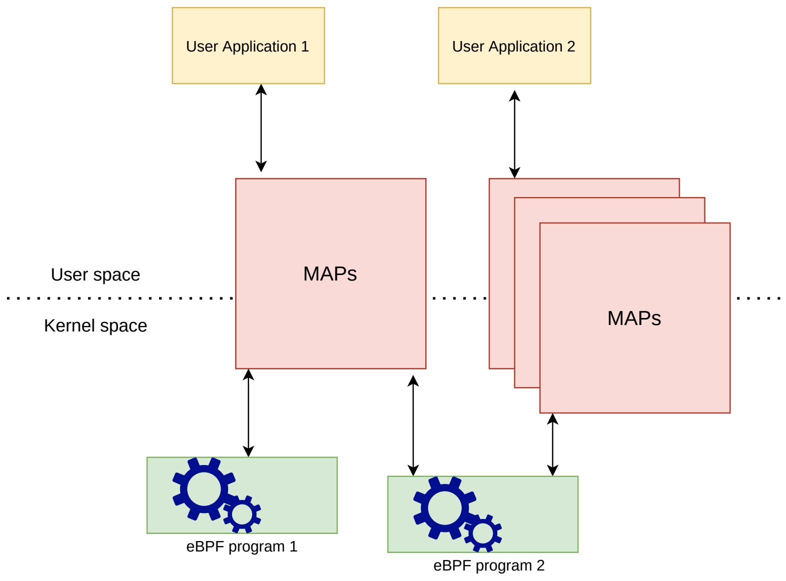


Figure 2.8: Shared memory: architecture.

There are different types of maps, which have behaviors and structures that distinguish them. There are both generic maps, some of which are:

- 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 you to have different instances of the same table for each CPU core, which leads to an improvement in performance. No synchronization mechanism is needed and maps can also be cached for a further increase in access speed.

2.8.5 Tail Calls

The mechanism of tail calls allows an eBPF program to call another one with a minimum overhead. Unlike function calls, tail calls transfer the control to a different program and never return. Service chains can be created exploiting direct virtual links between two eBPF programs or tail calls.

In order to avoid undefined execution time the number of consecutive nested calls is limited to 32.

This allows developers to overcome the program size limitation in the JIT compiler: starting from one big program, this can be split in multiple modules,perhaps functionally distinct such as header parsing, ingress filtering, forwarding, and analytics.

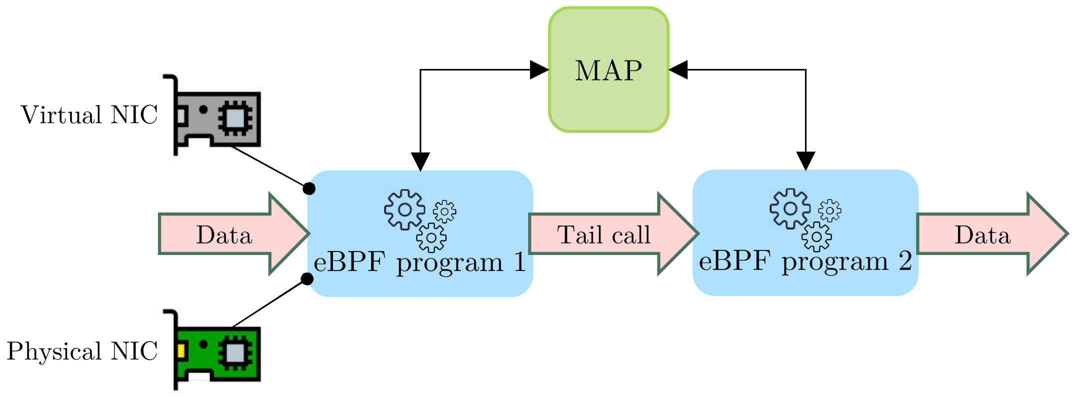


Figure 2.9: Example of a service chain implemented using tail calls.

2.8.6 Hooks

The execution of eBPF programs is triggered by specific kernel events that take the name of Hook Points. Different kernel events are handled by different program types, each one invoked with specific metadata carrying information about execution context. Hook points include all possible kernel events, for example packet reception, system calls invocation, page fault, etc.

There are two program types related to packet processing: eXpress Data Path and Traffic Control.

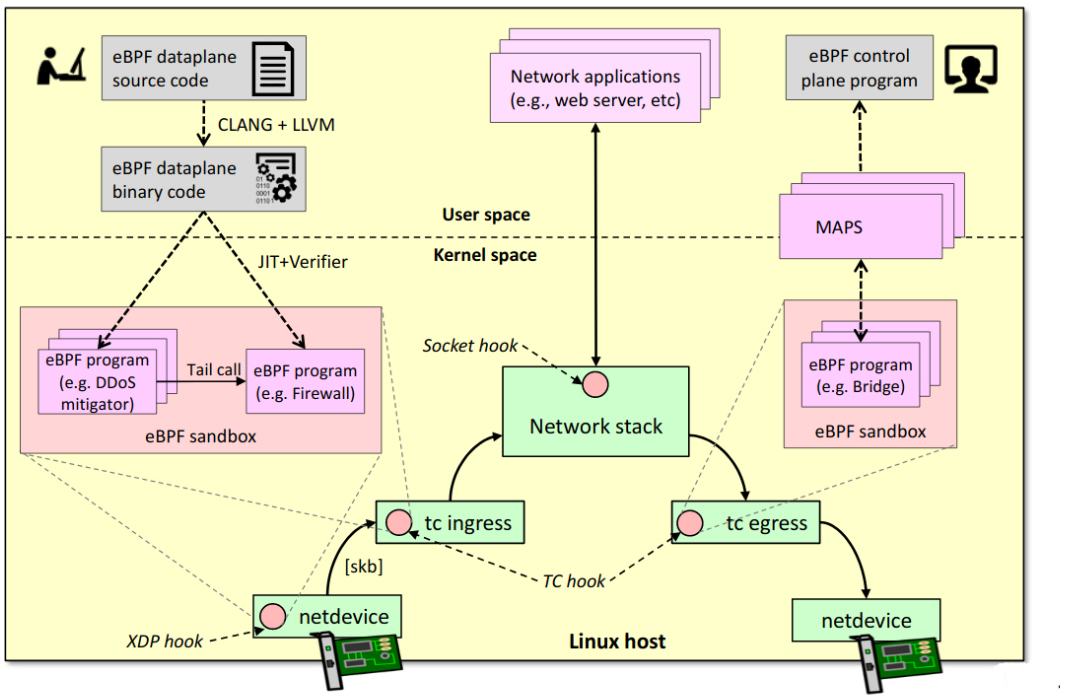

eBPF dataplane source code
CLANG + LLVM
eBPF dataplane binary code
JIT+Verifier
Network applications (e.g., web server, etc)
eBPF control plane program
MAPS
User space
Kernel space
eBPF program (e.g. DDoS mitigator)
Tail call
eBPF program (e.g. Firewall)
eBPF sandbox
Socket hook
Network stack
eBPF program (e.g. Bridge)
eBPF sandbox
tc ingress
tc egress
[skb]
TC hook
netdevice
XDP hook
Linux host
netdevice

Figure 2.10: Graphical representation of XDP and TC hook points.

eXpress Data Path (XDP)

ThXDP (eXpress Data Path) is a programmable, high performance packet processor in the Linux networking data path; it provides an additional hook to be used with eBPF programs to intercept packets in the driver space of the network adapter, before they are manipulated by the Linux kernel. The main advantage of this early processing is that it avoids the overhead and the memory consumption added by the kernel to create the socket buffer (*skb* data structure) which wraps the packet for standard Linux processing in TC mode. XDP runs in the lowest layer of the packet processing stack, as soon as the NIC driver realizes a packet has arrived. However, packets here are not delivered to userspace, but to the injected eBPF program executed in kernel space. One of the main use cases is pre-stack processing for filtering or DDOS mitigation.

The return code of the program defines how the packet must be processed by the kernel. It can be dropped (`XDP_DROP` or `XDP_ABORTED`), can be redirected to another interface using helper functions `bpf_redirect()` and `bpf_redirect_map()` that return code `XDP_REDIRECT`, can be sent back to the same interface (`XDP_TX`) or can continue is path in the networking stack (`XDP_PASS`).

Traffic Control (TC)

This program type allows to process packets in the Traffic Control layer of the networking stack.

At this point the packet has been parsed and copied in a data structure called socket buffer and additional metadata are available to the eBPF program such as the protocol, the priority, the reception timestamp, VLAN associated metadata and layer 3 and 4 information.

While TC programs can't achieve XDP performance they come with some advantages:

- No driver support is required, allowing this kind of programs to be attached to any interface.
- Unlike XDP programs, TC ones can process packets in the egress path of the networking stack.
- Thanks to additional metadata available, a richer set of helpers is provided, allowing to perform complex operations like handling VLAN encapsulation or updating L3 and L4 checksums.

2.9 BCC

BCC (BPF Compiler Collection) is a toolkit for creating efficient kernel tracing and manipulation programs. It provides macros and structures to simplify the writing of eBPF C code, and includes frontends in Python and LUA to interact with eBPF programs in user space. While being mostly focused on tracing its infrastructure can also be used for network traffic management.

2.10 Polycube

Polycube [12] is an open source framework that enables the creation of fast and efficient in-kernel network functions chain based on eBPF and XDP technologies.

Polycube provides the user with a set of services, such as router, firewall, bridge, etc., that can be dynamically connected and configured to provide custom connectivity to namespaces, containers, virtual machines, and physical hosts. From the developer perspective, Polycube provides the infrastructure to create complete network functions, simplifying the implementation of control and user planes and the management of the interaction between the two. Two standalone applications are also available: *pcn-iptables*, a faster clone of iptables, and *pcn-k8s*, a CNI network plugin for Kubernetes.

Polycube adopts a centralized architecture, in which all management tasks are carried out by a userspace daemon, called *polycubed*. Interaction with the system can happen using a command line interface, called *polycubectl*, or through a RESTful API.

A schematic representation of Polycube architecture is shown in 2.11, main aspects are discussed in the following sections.

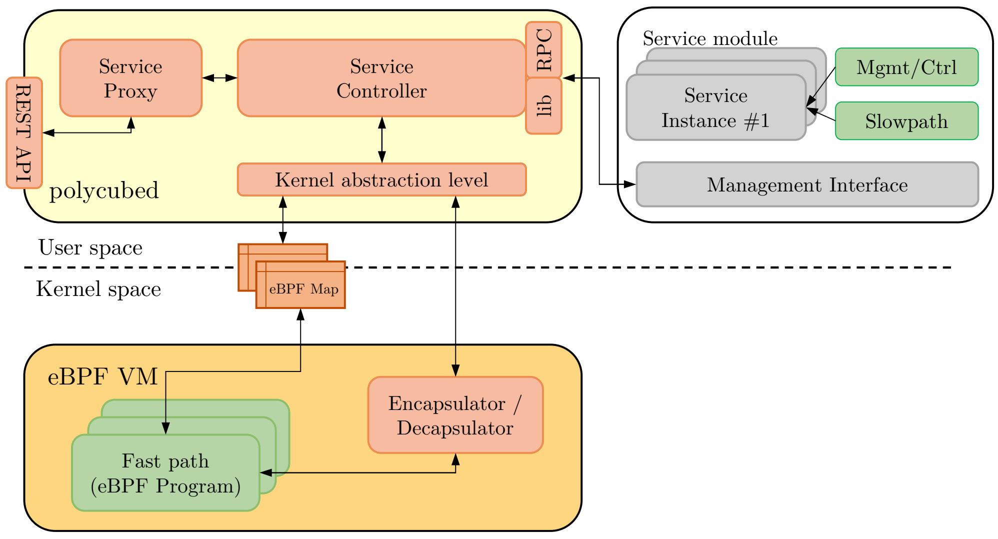


Figure 2.11: Simplified Polycube architecture.

2.10.1 Services

Each Polycube service is made up of a `control plane` and a `data plane`. The data plane is responsible for per-packet processing and forwarding, while the control and management plane is in charge of service configuration and non-dataplane tasks (e.g., routing protocols). Although this separation between the control and data plane is common in many network functions architectures, Polycube provides a clear separation between these components; each service is composed of a set of standard parts that make it easier for the programmers to implement the desired behavior, while Polycube takes care of creating all the surrounding glue logic, handling all the interactions and communications between the different components.

Data Plane

The data plane is responsible of the processing and forwarding of single packets and is composed by a fast and a slow path.

The *fast path* is executed at kernel level by or more eBPF programs and performs basic tasks like packet parsing and mangling and maps update.

The *slow path* is executed in user space and handles all that packets that require a more complex processing, like generating an Arp Reply. Although eBPF offers the possibility to perform some complex and arbitrary actions on packets, it suffers from some well-known limitations due to his restricted virtual machine, which however are necessary to guarantee the integrity of the system. Those limitations may impair the flexibility of the network function, which may not be able to perform complex actions directly in the eBPF fast path or could slow down its execution, adding more instructions in the fast path to handle exceptional cases. To overcome those limitations, Polycube introduces an additional data plane component that is no longer limited by the eBPF virtual machine and it can hence execute arbitrary code. The slow path module is executed in user space and interacts with the eBPF fast path using a set of components provided by the framework. The eBPF fast path program can redirect packets (with custom meta-data) to the slow path. Similarly, the slow path can send packets back to the fast path; in this case, Polycube provides the possibility to inject the packet into the ingress queue of the network function port, simulating the reception of a new packet from the network, or into the egress queue, hence pushing the packet out of the network function.

Control plane

The control plane of a virtual network function is the place where out-of-band tasks, needed to control the data plane and to react to possible complex events (e.g., Routing Protocols, Spanning Tree) are implemented. It is the point of entry for external players (e.g. service orchestrator, user CLI) that need to access service's resources, modify (e.g., for configuration) or read service parameters (e.g., reading statistics) and receive notifications from the service fast path or slow path.

Polycube defines a specific control and management module that performs the previously described functions. It exposes a set of REST APIs used to perform the typical CRUD (create-read-update-delete) operations on the service itself; these APIs are automatically generated by the framework starting from the service description, removing this additional implementation overhead to the programmer. Each Rest request passes through `polycubed` which dispatches it to the corresponding service control plane.

2.10.2 Cubes

Cubes are instances of Polycube services, that can be connected together to create complex service chains.

Data plane of services can be instantiated both as a TC and as a XDP (XDP_DRV for native XDP and XDP_SKB for generic XDP) program. To help the developer writing code that is not bound to the program type Polycube provides unified functions to perform tasks that needs a different implementation for TC and XDP programs, such as VLAN encapsulation and packet checksum update.

Two kinds of cubes are available:

Standard Cubes

Standard cubes have forwarding capabilities and can be used to implement services such as a Router or a Bridge.

Polycube introduces the concept of port, a connection point that can link a standard cube to another cube or to a network device. Information about the port on which the packet was received is carried in packet metadata.

Besides dropping the packet and sending it to the kernel networking stack with `RX_DROP` and `RX_OK` return codes, a standard cube can redirect it to another port using the `pcn_pkt_redirect()` function. The code of this function is dynamically generated every time a new port is connected to the cube, in order to either perform a tail call to the eBPF code of a peer cube, or to invoke the `bpf_redirect()` helper to send the packet out of an interface.

Transparent Cubes

Transparent cubes do not have forwarding capabilities and have to be attached to a port of an existing standard cube or to a network device. They process packets flowing in or out the entity they are bound to through their set of ingress and egress programs, and can be used to implement services like a firewall or a NAT.

Cubes of this type inherit the parameters of the port they are attached to (MAC, IPv4, etc.), multiple instances can be connected to the same port implementing a stack of functions.

Polycube wrapper code allows to correctly link programs in the ingress and egress chain of an interface. Every time a transparent cube is attached or removed this code can be updated injecting a new version of the program, to connect the cube to the correct next entity.

When an instance of transparent cube lets the packet pass with return code `RX_OK`, three situations may occur:

- There is another cube (transparent or standard) in the chain: in this case its eBPF code is executed with a tail call.
- The next entity is a networking device: in this case the packet is redirected using `bpf_redirect()` helper.
- The next entity is the networking stack of the host: the packet proceeds with return code `XDP_PASS` or `TC_ACT_OK`.

In the next figure 2.12 is shown an example of topology made by Transparent Cubes (purple) and Standard Cubes (yellow).

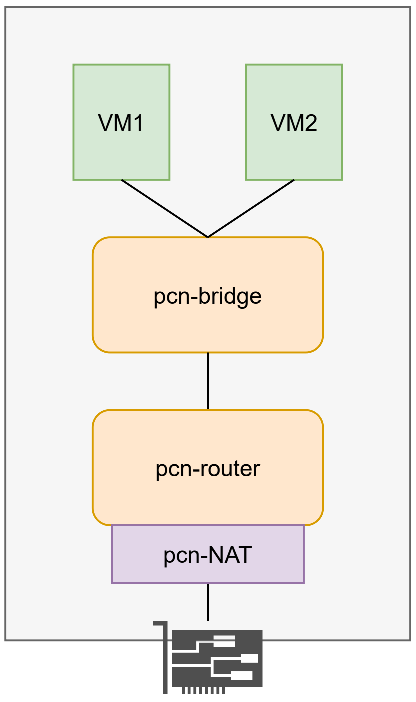


Figure 2.12: Example of transparent and standard cubes chain

2.10.3 Polycube Daemon

The Polycube System Daemon polycubed provides a kernel abstraction layer that is used by the different services. It exposes a configuration mechanism of the different service instances through a rest API server. Users can interact with it using polycubectl. It manages the lifecycle of cubes, handling their creation, update, connection, and deletion.

Polycubed is based on the BCC toolkit, used to manage the interaction among the user space component of services and their in-kernel data path.

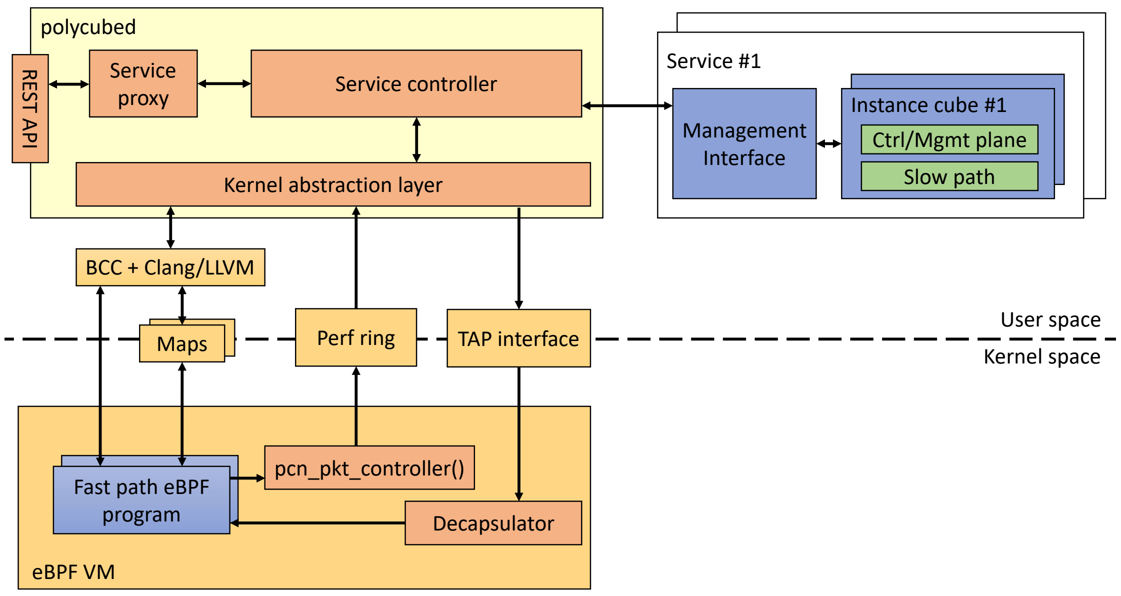


Figure 2.13: Polycubed architecture.

2.11 Virtual Extensible LAN

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) [13]. 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 [14]. With a 24-bit segment ID, aka VXLAN Network Identifier (VNI), VXLAN allows

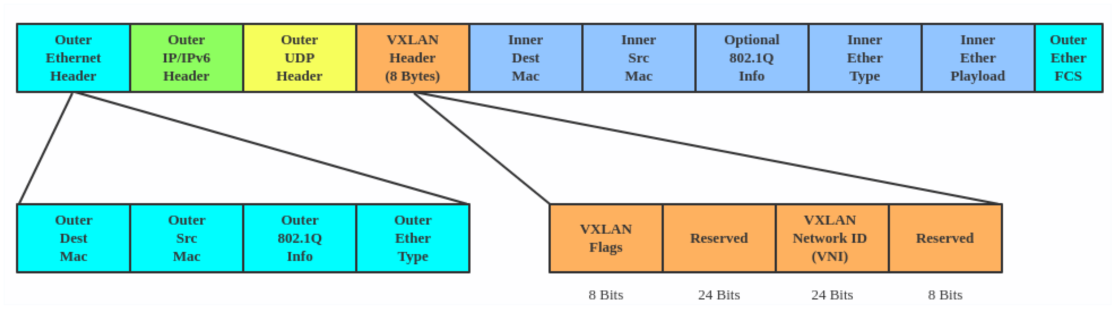
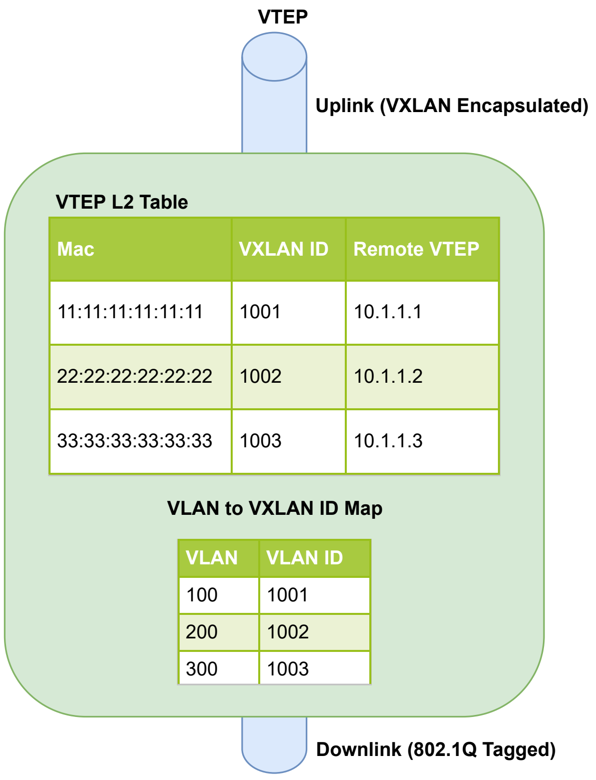


Figure 2.14: VXLAN packet.

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 separate 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.

Using the diagram 2.15 for reference 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 multicast groups. Each VXLAN ID has an assigned IP multicast group to use for traffic flooding (the same multicast group can be



Mac	VXLAN ID	Remote VTEP
11:11:11:11:11:11	1001	10.1.1.1
22:22:22:22:22:22	1002	10.1.1.2
33:33:33:33:33:33	1003	10.1.1.3

VLAN	VLAN ID
100	1001
200	1002
300	1003

Figure 2.15: VXLAN example.

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.

Chapter 3

Cluster Networking

Kubernetes was built to run distributed systems over a cluster of machines. The very nature of distributed systems makes networking a central and necessary component of Kubernetes deployment, and understanding the Kubernetes Networking Model will allow to know which NF (Network Functions) are needed and how to interconnect them. There are four distinct networking problem to address: [15]

- **Highly-coupled container-to-container communications**: this is solved by Pods and `localhost` communications.
- **Pod-to-Pod communications**: this will be explained in next sections.
- **Pod-to-Service communications**: this is covered by services.
- **External-to-Service communications**: this is covered by services.

3.1 The Kubernetes Networking Model

Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where Pods can be treated much like VMs. Kubernetes makes opinionated choices about how Pods are networked. In particular, Kubernetes dictates the following requirements on any networking implementation:

- all Pods can communicate with all other Pods without using NAT (Network Address Translation)
- all Nodes can communicate with all Pods without NAT.
- the IP that a Pod sees itself as is the same IP that others see it as.

Given these constraints, we are left with cluster networking problems shown in the previous section 3.

3.1.1 Container-to-Container Networking

Network communication in virtual machine is tipically viewed as interacting directly with an Ethernet device. In reality each running process communicates within a network namespace [16] that provides a logical networking stack with its own routes, firewall rules, and network devices, which means it provides a brand new network stack for all the processes within the namespace. In essence network namespaces provide isolation of the system resources associated with networking.
Network namespaces can be created using the **ip** command.

Listing 3.1: Basic example of creating a network namespace called ns1.

```
 \$ ip netns add ns1
```

When the namespace is created, a mount point for it is created under /var/run/netns

Listing 3.2: Basic example of creating a network namespace called ns1.

```
\$ ls /var/run/netns
ns1
\$ ip netns
ns1
```

By default, Linux assigns every process to the root network namespace so that they can be reachable from external 3.1.

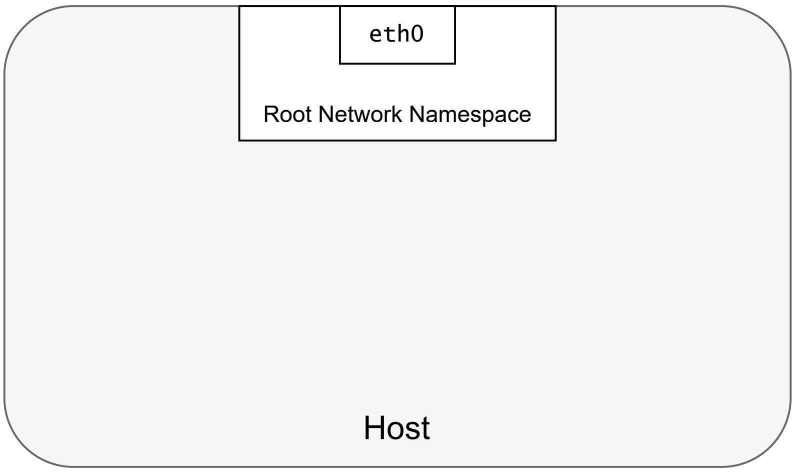


Figure 3.1: Root network namespace

A Pod is made up as a group of Docker container that share a network namespace. Containers inside a Pod all have the same ip address and same port space assigned through the network namespace, and can find each other via `localhost` since they reside in the same namespace. So each Pod has its own network namespace. Applications within a Pod also have access to shared volumes which are part of the Pod and can be mounted by applications themselves.

3.1.2 Pod-to-Pod Networking

In Kubernetes, every Pod has its own IP address and each Pod must communicate with others without NAT, wether the Pod is deployed on the same physical Node or in a different one.

Pod-to-Pod: same Node

Namespaces can be attached using a Linux Virtual Ethernet Device [17]. They can be used to tunnels betweeen network namespaces to create a bridge to a network service in another namespace. To connect Pod namespaces, we can assign one side of the veth pair to the Pod's netwoork namespace and the other one to the root network namespace 3.2.

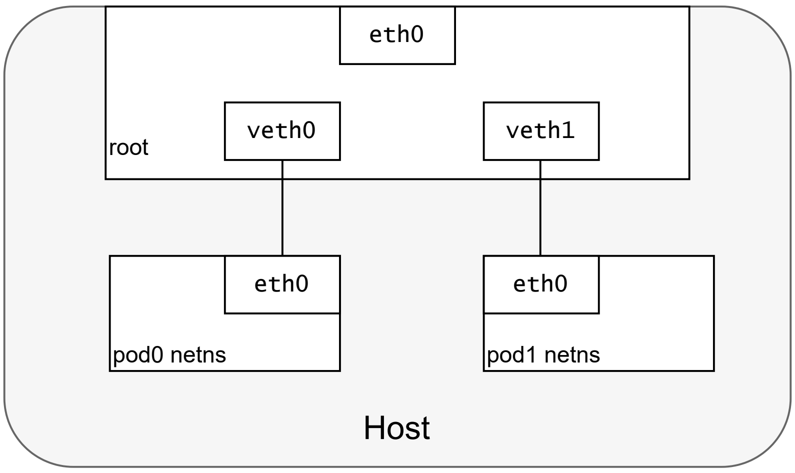


Figure 3.2: Attach network namespaces to root namespace

Now, we want Pods to talk to each other, and for this we can use a bridge because Kubernetes allocates one subnet per Node and so Pods in the same host are in the same subnet and then they will communicate with Layer 2 networking 3.3. The bridge maintains a forwarding table to know where destination is. When bridge receives a packet, it saves in the forwarding table the source port and MAC source of the packet and then if it knows where destination is, it will forward the packet through destination port, else will broadcasts the frame out to all connected devices except the original sender.

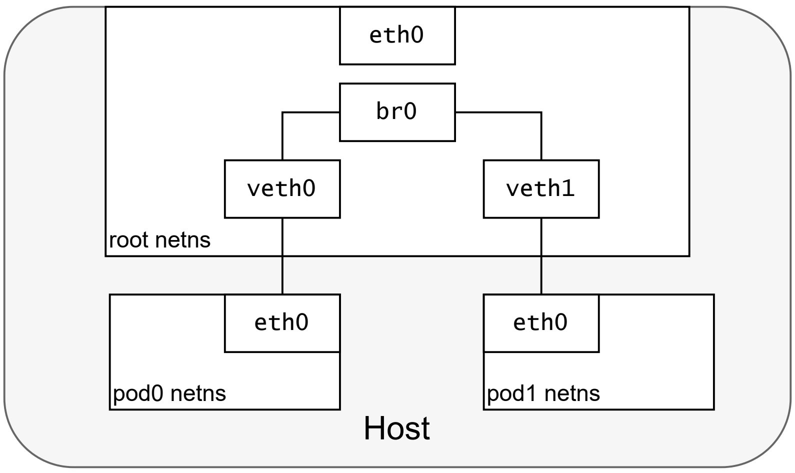


Figure 3.3: Connecting pods to a Linux Bridge

Packet path: Pod-to-Pod, same Node

Given the previous topology 3.3, lets assume communication from Pod1 to Pod2. Pod1 sends packet to its interface eth0, which is the default interface for the Pod, and this one is directly attached to the Linux Bridge br0 and the latter one will send it to veth1 interface that is directly attached to Pod2. So with this strategy Pods in the same host can communicate without NAT.

Packet path: Pod-to-Pod, different Nodes

Kubernetes networking model requires that Pod IPs are reachable across all nodes without NAT. Generally, every node has assigned its own Pod CIDR block. Once traffic with destination one of the IPs of the CIDR block reaches the Node, this one must send it to the right Pod. So the key is to assuming that `the network can route traffic in a CIDR block to the correct node`.
Lets assume traffic from Pod1 in the host1 to Pod2 in the host2. The packet begins by being sent through Pod 1's Ethernet device which is paired with the virtual Ethernet device in the root namespace. Ultimately, the packet ends up at the root namespace's network bridge and from thatwe assume that the packet will arrive to eth0 in the root namespace that is the Default Gateway for Pods. At this point the route leaves the Node and enters the network. We assume for now that the network can route the packet to the correct Node based on the CIDR block assigned to the Node. The packet enters the root namespace of host2, where it is routed through the bridge to the correct virtual Ethernet device. Finally, the route completes by flowing through the virtual Ethernet device's pair residing within Pod 2's namespace. Generally speaking, each Node knows how to deliver packets to

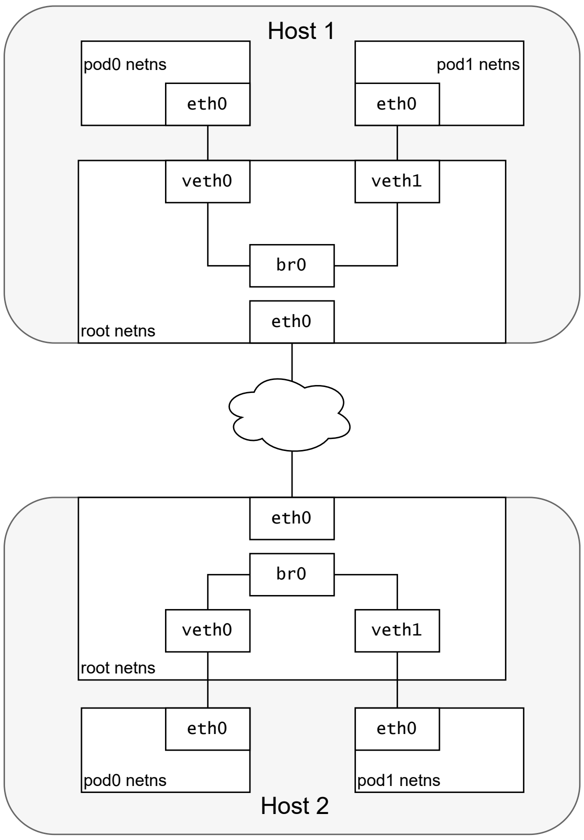


Figure 3.4: Routing traffic between Pods on different Nodes.

Pods that are running within it. Once a packet reaches a destination Node, packets flow the same way they do for routing traffic between Pods on the same Node.

3.1.3 Pod-to-Service Networking

Kubernetes Pods are created and destroyed to match the state of your cluster. Pods are nonpermanent resources. If you use a Deployment to run your app, it can create and destroy Pods dynamically.

Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later. [18] So because of ephemerality of Pods **Services** are built into Kubernetes to address this problem. Services act as an abstraction over Pods and assign a single virtual IP address to a group of Pod IP addresses. Any traffic addressed to the virtual IP of the Service will be routed and `load-balanced` to the set of Pods that are associated with the virtual IP. This allows the set of Pods associated with a Service to change at any time — clients only need to know the Service's virtual IP, which does not change. When a Pod contact a service something must substitute the service ip with one of Pods behind the service performing NAT and Load-balancing. This actor can be implemented using:

- **Iptables**: is a user-space program providing a table-based system for defining rules for manipulating and transforming packets using the netfilter framework. In Kubernetes, iptables rules are configured by the kube-proxy controller that watches the Kubernetes API server for changes.
- **IPVS (IP Virtual Server)**: is also built on top of netfilter and implements transport-layer load balancing as part of the Linux kernel. IPVS can direct requests for TCP- and UDP-based services to the real performin load-balancing, and make services of the real servers appear as virtual services on a single IP address.
- **eBPF**: with eBPF can be implement efficient code to allow translation and load-balancing of Service ip

In Kubernetes translation of Service IP and Load-balancing is performed by **kube-proxy** [19] using Iptables or IPVS.

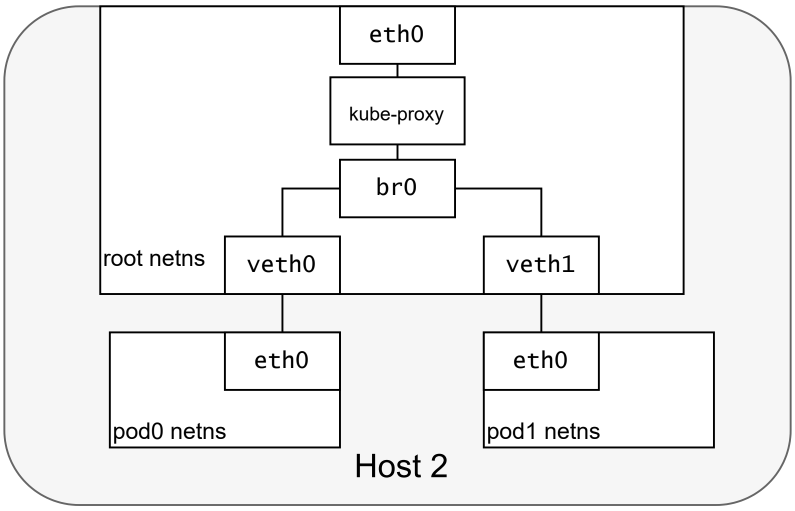


Figure 3.5: Routing traffic from Pod to Service.

When Pod1 contact a service the life of packet begins in the same way as before. Since Service IP is not in the same Pod's subnet the packet will go to node's external interface, and then kube-proxy will choose randomly one of Pods behind the service, because is the firts time that Pod1 contacts the service, and rewrite the destination IP of the packet and saves the session. On the return path, the IP address is coming from the backend Pod. In this case kube-proxy again will rewrite the IP source to replace the Pod IP with the Service IP. With this strategy the sender Pod1 believes to communicate always with the Service IP. Kube-proxy mantains a session so the previously selected backend Pod will be always chosen for Pod1 as long as the session is alive.

3.1.4 Internet-to-Service Networking

Internet-to-service networking is required for getting traffic from a Kubernetes Service out to the Internet, and get traffic from the Internet to Services.

Egress traffic

Routing traffic from a Node to the public Internet is network specific and really depends on how your network is configured to publish traffic. When pod send traffic to public Internet, a Nat rule must be added to change the Ip source of Pod because this one is not reachable in Public Internet. So in this case we can use one of solutions shown previously 3.5

Ingress traffic

Ingress — getting traffic into your cluster — is a surprisingly tricky problem to solve. Again, this is specific to the network you are running, but in general, Ingress is divided into two solutions that work on different parts of the network stack: a Service LoadBalancer and an Ingress controller. When you create a Kubernetes Service you can optionally specify a LoadBalancer to go with it. The implementation of the LoadBalancer is provided by a cloud controller that knows how to create a load balancer for your service. Once your Service is created, it will advertise the IP address for the load balancer. Layer 7 network Ingress operates on the HTTP/HTTPS protocol range of the network stack and is built on top of Services. The first step to enabling Ingress is to open a port on your Service using the NodePort Service type in Kubernetes. If you set the Service's type field to NodePort, the Kubernetes master will allocate a port from a range you specify, and each Node will proxy that port (the same port number on every Node) into your Service. That is, any traffic directed to the Node's port will be forwarded on to the service using iptables rules. This Service to Pod routing follows the same internal cluster load-balancing pattern.

Chapter 4

Prototype Architecture

This chapter will explain the general architecture and the topology of CNI network plugin for Kubernetes. It gets into more details on functions carried out by the base modules that together solves networking problems seen in the previous chapter 3.

4.1 General Architecture

The prototype of the network plugin is designed to be made up by standard Network Functions built using Polycube 2.10. A set of smaller and standard modules carrying out specific functions are linked together to obtain Cluster Networking connectivity in Kubernetes cluster. The integration with Linux kernel provided by the eBPF technology allows to deploy the solution in a traditional data center scenario, where it can either route the packets towards different machines or external networks, or forward them to other functions running on the same host in containers or virtual machines, all of this without requiring a specific support from the virtualization infrastructure. Having a network Plugin made up by Disaggregated eBPF Services have some advantages:

- **Modularity**: in addition to the essential services required for the functioning of the plugin, others can be added to give more functionality like security, network policies, alerts and more.
- **Performance**: thanks to eBPF we can write services that processes packets closer to the NIC for fast packet processing.
- **Scalability:** This network plugin uses standards based network protocols trusted worldwide by the largest internet carriers.

Polycube provides the infrastructure to build and link the services together. Every service is composed by an in-kernel data plane and a user space control /management plane, is described by a YANG data model and can be accessed by a RESTful API, directly with HTTP requests or through the Polycube CLI. eBPF dataplane of those services are interconnected from Pods to your data and tunnel interfaces. This allows Services to spot workload packets early and handle them through a fast-path that bypasses iptables and other packet processing that the kernel would normally do.

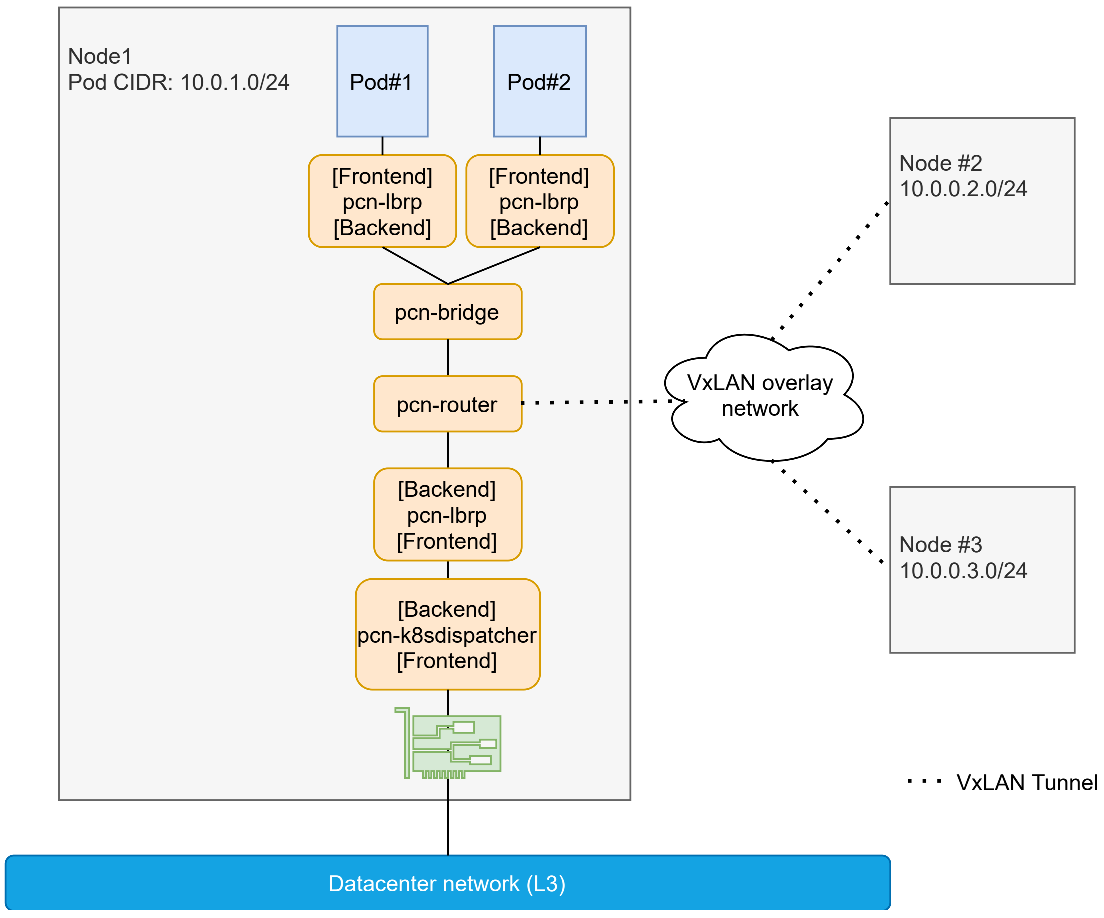


Figure 4.1: Modular Network Plugin topology

The topology shown in 4.1 is made up by four different Polycube Services: **pcn-bridge**, **pcn-router**, **pcn-loadbalancer-rp** and **pcn-k8sdispatcher**. Each one of those services implements a specific part of interconnection between Pods, Services, and the external world.

4.2 Pcn-bridge

This service implements a fast and simple Ethernet bridge. It is responsible for interconnection between Pods in the same Host 3.1.2, and interconnection between Pods and the Default Gateway. Transparent bridging uses a table called the forwarding table to control the forwarding of frames between network segments. The table starts empty and entries are added as the bridge receives frames. If a destination address entry is not found in the table, the frame is flooded to all other ports of the bridge, flooding the frame to all segments except the one from which it was received. By means of these flooded frames, a host on the destination network will respond and a forwarding database entry will be created. Both source and destination addresses are used in this process: source addresses are recorded in entries in the table, while destination addresses are looked up in the table and matched to the proper segment to send the frame to.It supports up to 1024 hosts. This service is already available in Polycube and thanks to eBPF performance is much better than a Linux Bridge [20].

4.3 Pcn-router

Router module 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 addressesand a MAC address. The static routing table can be configured with a set of routes, specifying destination network CIDR (address and prefix lenght), next hop and path cost. This service can be attached also to phisycal or virtual interfaces. If the port is connected to a network interface of the host Mac and Ip of the router port is the same of the physical one. In this scenario the Router is Default gateway for Pods in the same node connecting Pods to external world and to other Pods performing Pod to Pod communication across nodes thanks to VxLAN [14] tunnel.

4.4 Pcn-loadbalancer-rp

This service implements a Reverse Proxy Load Balancer. According to the algorithm, incoming IP packets from FRONTEND port are delivered to the real backend servers by replacing their IP destination address with the one of the real server, chosen by the load balancing logic. Hence, IP address rewriting is performed in both directions, for traffic coming from the Internet and the reverse. Packet are 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. This module is responsible for performing the translations of Kubernetes Service virtual IP with one of PODs behind it 3.5. There is a pcn-lbrp for each Pod and one between pcn-router and pcn-k8sdispatcher to allow Nodeport Service Communication. Traffic is forwarded to backend services after performing an IP address rewriting in the packet (from vip:port to the selected realip:port); the vip virtual IP address and the IP address of the actual servers should belong to different IP networks. This service exports two network interfaces: - Frontend port: connects the LB to the clients that connect to the virtual service, likely running on the public Internet - Backend port: connects the LB to to backend servers

4.5 Pcn-k8sdispatcher

This service is built specifically for this project. It is a custom eBPF Nat. It performs source Nat for packets coming from Pods: it transparently changing the source IP address of an end route packet and performing the inverse function for any replies. A session table is used to track opened communications 4.2.

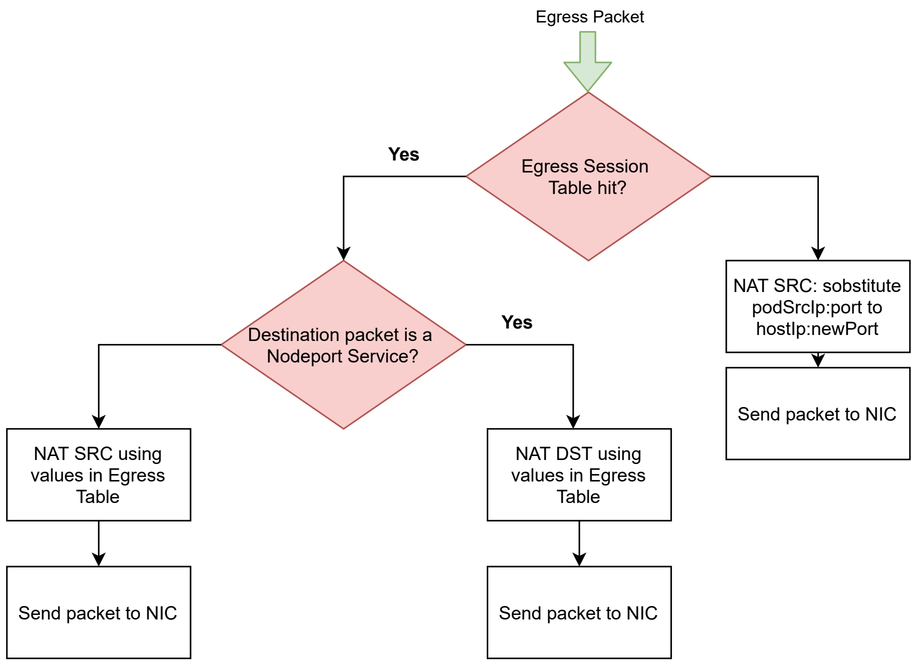

Egress Packet
Egress Session Table hit?
Yes
Destination packet is a Nodeport Service?
Yes
NAT SRC: sobstitute podSrcIp:port to hostIp:newPort
Send packet to NIC
NAT SRC using values in Egress Table
Send packet to NIC
NAT DST using values in Egress Table
Send packet to NIC

Figure 4.2: pcn-k8sdispatcher: egress schema

When a packet arrives into the Node this cube checks if the packet belongs to an opened session or if it is direct to a Nodeport Service after checking `ExternalTrafficPolicy` value of Service itself. If so the packet follows the Kubernetes network topology and it is properly modified, if not it is sent to Linux stack networking. The externalTrafficPolicy is a standard Service option that defines how and whether traffic incoming the node is load balanced. Cluster is the default policy and the traffic can be spread across all backend Pods, Local instead choose only Pods inside the node and is often used to preserve the source IP of the traffic. Local effectively disables load balancing on the cluster node so that traffic that is received by a local Pod sees the original source IP address 4.3.

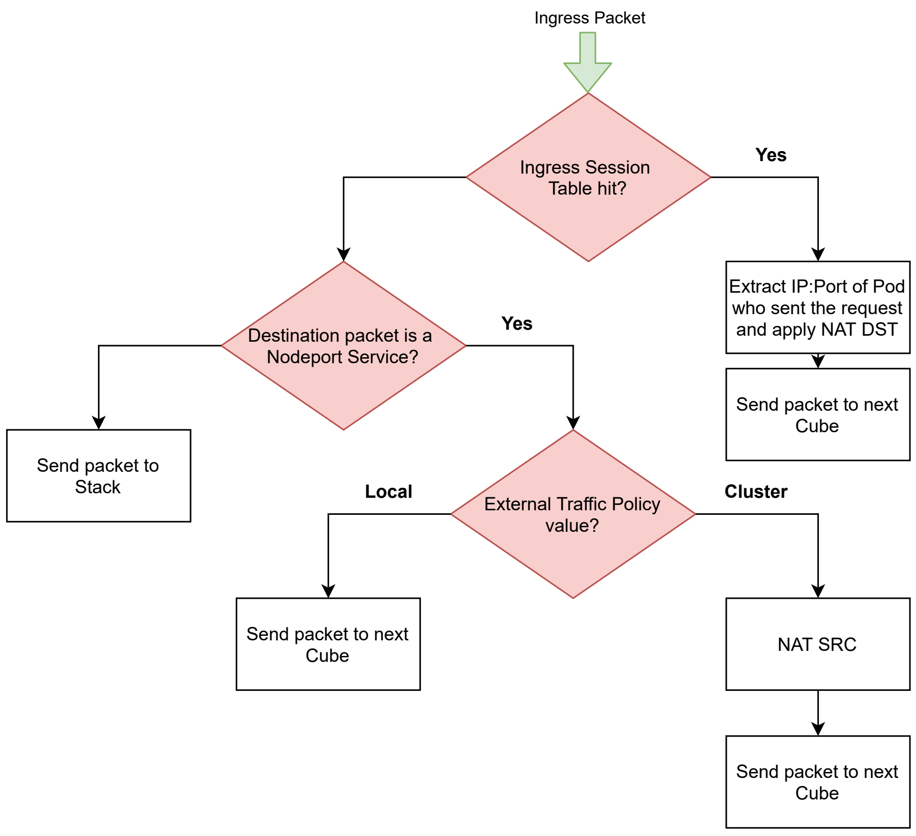

Ingress Packet
Ingress Session Table hit?
Yes
Extract IP:Port of Pod who sent the request and apply NAT DST
Send packet to next Cube
Destination packet is a Nodeport Service?
Yes
Send packet to Stack
Local
External Traffic Policy value?
Cluster
Send packet to next Cube
NAT SRC
Send packet to next Cube

Figure 4.3: pcn-k8sdispatcher: ingress schema

4.6 Integration with Kubernetes

All Services shown in previous section must react to some Kubernetes events in order to have connectivity between all components. For this purpose a `pcn-k8s operator` is designed to watch some Kubernetes resources and communicate with Polycube Daemon to update rules of services accordingly. `pcn-k8s operator` watches following resources:

- **Service**: every time a service is created or deleted, `pcn-loadbalancer-rp` translation rules must be updated accordingly.

- **Node**: when a new Node join the cluster, all services must be created and synchronized to provide connectivity between existing Services and Nodes.
- **Pods**: each time new Pod is created, a `pcn-loadbalancer-rp` is created and attached to Pod's interface, then the operator update it's rules in order to provide Service connectivity for that Pod.

`pcn-k8s` operator watch kubernetes resources and update Polycube services using REST protocol. Thanks to this operator cluster networking 3 can be satisfied.

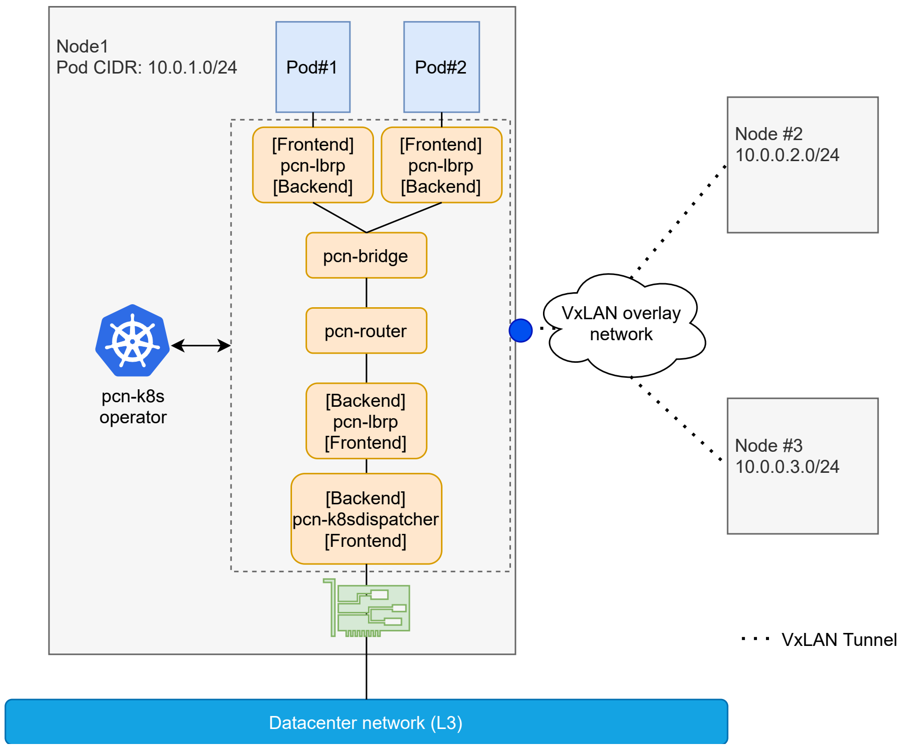


Figure 4.4: Interoperability between K8s operator and Polycube

Chapter 5

Prototype Implementation

As already described in the previous chapter, the chosen architecture is designed to work with Kubernets. This chapter explains how concepts shown in the Prototype Architecture 4 section has been implemented in details, which programming languages are used and which workarounds used to solve problems encountered.

Router, Load Balancer Reverse Proxy, Bridge and Router are standard cubes already available in the Polycube framework that can be connected to interfaces or other services through a Port. K8s-dispatcher is built specifically for the project, it is a standard cube and it will be attached between Host NIC and a pcn-loadbalancer-rp instance.

Programming languages used in this project are `C++` for the implementation of Control Plane of the service, `C` for the Data Plane, `Go` for CNI plugin and pcn-k8s operator and the `YANG` data modelling is used to describe the Service structure.

5.1 Automatic Code Generation

Polycube provides an automatic code generation called `polycube-codegen` used to generate a stub from a YANG datamodel. It is composed by pyang, that parses the YANG datamodel and creates an intermediate JSON that is compliant with the OpenAPI specifications [21]; and swagger-codegen starts from the latter file 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.

- `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`: Is used to compile the service as shared library.

- `src/{service-name}_dp.c`: 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.

Since each Polycube service uses a specific convention for generate the REST APIs, in order to interact with them it can be used `polycubectl` or creating a client programs specific for each service. `polycube-codegen` supports the generation of client stubs in different programming languages including `golang`. For this project have been generated Go clients of all Polycube services used, in order to be controlled by the operator.

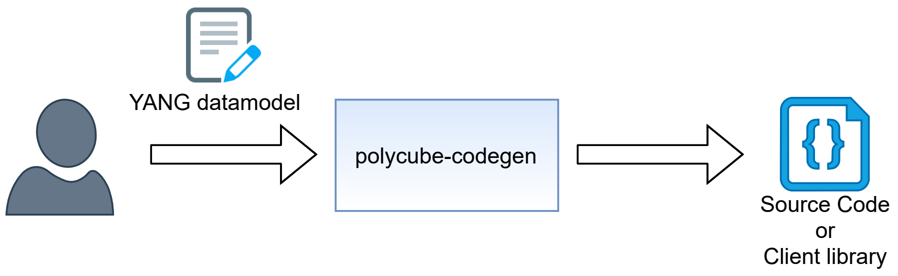


Figure 5.1: Generating code with polycube-codegen

5.2 K8sdispatcher

k8sdispatcher is a Polycube service made to allow POD to internet communication and Service to POD communication. As explained in the previous section, the generated classes reflect the structure of the node tree present in the YANG model. Since the main structure is a tree, each class represents a node and contains all the properties of the node itself and references to its children nodes though the use of C++ shared pointers. Each generated class receive in its constructor a configuration object which may contain a value for any property of the class itself; this is mainly used by this service to be instantiated with a pre-defined configuration which has to be applied to initialize the service.

5.2.1 Data Model

The `Data Model` of K8s-dispatcher defines a set of Nodeport-rules and Natting-rules. Nodeport-rules can be created, updated or destroyed by Rest-calls, Natting-rules instead can be only getted for debugging purposes because only dataplane_code can add natting rules. Nodeport-rules can be Cluster type or Local type.

Listing 5.1: pcn-k8sdispatcher data model: definition of nodeport-rule and nodeport-range

```
  list nodeport-rule {
        key "nodeport-port proto";

        leaf internal-src {
            type inet:ipv4-address;
            description "Source IP address";
        }
        leaf nodeport-port {
            type inet:port-number;
            description "Destination L4 port number";
        }
        leaf proto {
            type string;
            description "L4 protocol";
        }
        leaf service-type {
        type enumeration {
          enum CLUSTER { description "Cluster wide service"; }
          enum LOCAL { description "Local service"; }
        }
        mandatory true;
        description "Denotes if this Service desires to route
    external traffic to node-local or cluster-wide endpoint";
      }
    }

  leaf nodeport-range {
    type string;
    description "Port range used for NodePort services";
    default "30000-32767";
    polycube-base:cli-example "30000-32767";
  }
```

Local type is used to maintain client Ip and so the traffic can't be offloaded to pods external to the node. In this case Loadbalancing is performed inside the node. Cluster type instead, Loadbalancing is performed using all Pods covered by the Service, and to do so src Ip of packet must be modified so reply return where translation is performed.

Listing 5.2: pcn-k8sdispatcher data model: definition of natting-rule

```
list natting-rule {
        key "internal-src internal-dst internal-sport internal-dport
    proto";
        leaf internal-src {
            type inet:ipv4-address;
            description "Source IP address";
        }
        leaf internal-dst {
            type inet:ipv4-address;
            description "Destination IP address";
        }
        leaf internal-sport {
            type inet:port-number;
            description "Source L4 port number";
        }
        leaf internal-dport {
            type inet:port-number;
            description "Destination L4 port number";
        }
        leaf proto {
            type string;
            description "L4 protocol";
        }
        leaf external-ip {
            type inet:ipv4-address;
            description "Translated IP address";
        }
        leaf external-port {
            type inet:port-number;
            description "Translated L4 port number";
        }
    }
```

Ports can be of type Frontend or Backend. The first one must be attached to Host NIC, and thanks to a callback in the Control Plane it retrieves NIC's IP and MAC and stores it in the eBPF code that will be used for natting. It also defines an internal IP used for Services with `ExternalTrafficPolicy = Cluster`.

5.2.2 Dataplane Class

The `Dataplane` class represents the configuration of the dataplane: it stores the eBPF code, services and NAT configurations and ports. The main aspects of the Data Plane is described in the previous chapter 4.5. Informations of Nodeport services exposed by Kubernetes are stored in an eBPF maps. The key type is a structure called dp_k that saves service Ip, port and protocol. If service has `ExternalTrafficPolicy = Cluster`, source IP of packet is traduced so reply packet returns where traduction is performed.

Listing 5.3: Nodeport Service Map

```
struct dp_k {
  u32 mask;
  __be32 external_ip;
  __be16 external_port;
  uint8_t proto;
};
struct dp_v {
  __be32 internal_ip;
  __be16 internal_port;
  uint8_t entry_type;
};

BPF_F_TABLE("lpm_trie", struct dp_k, struct dp_v, dp_rules, 1024,
BPF_F_NO_PREALLOC);
```

Ingress Natting rules are saved into **dp_rules** map. Each time a new nodeport service is created, `pcn-k8s` operator performs a REST call to invoke function `K8sdispatcher::addNodeportRule` which adds a new nodeport rule to eBPF datapath. Every time a new packet comes from the host NIC first of all is checked if packet belongs to a existing session. If so the packet is NATTED and then sent to `BACKEND` port. If not destination packet, port, and protocol are used to create dp_k variable used to check if it belongs to a nodeport service. If so thanks to entry_type value of dp_v struct is used to traduce it correctly. `dp_v.entry_type` can be:

- `NAT_SRC`: SrcIp, and SrcPort are translated. It is used for egress packets coming from Pods.

- `NAT_DST`: DstIp, and DstPort are translated. It is used for ingress packets that are going to Pods or for egress Pods packets with Cluster Nodeport_Service destination.

- `NODEPORT_CLUSTER`: SrcIp and SrcPort of Ingress Cluster Nodeport Service packets are translated with internal_ip and new port to mantain client-server session also if Pod in other node is chosen.

Listing 5.4: Session tables

```
struct st_k {
  uint32_t src\_ip;
  uint32_t dst\_ip;
  uint16_t src_port;
  uint16_t dst_port;
  uint8_t proto;
};
struct st_v {
  uint32_t new_ip;
  uint16_t new_port;
  uint8_t originating_rule_type;
};
BPF_TABLE("lru_hash", struct st_k, struct st_v, egress_session_table,
NAT_MAP_DIM);
BPF_TABLE("lru_hash", struct st_k, struct st_v, ingress_session_table
    ,
NAT_MAP_DIM);
```

5.2.3 Control Plane

The control plane is implemented by three main classes: `Pcn-k8sdispatcher`, `Nodeport-rule` `Natting-rule`.

`Pcn-k8sdispatcher` is the access point of the service, it holds a `unordered_map<NodeportKey, NodeportRule>` that saves Noodeport rules in control plane and inject them in maps read by eBPF datapath. It also saves Nodeport-range ports, ip and mac of host NIC and methods mapped to Rest API.

`Nodeport-rule` saves a natting rule. The class contains: *service-ip*, *protocol*, *service-port*, *service-type* with getters and setters methods.

`Natting-rule` saves informations about a session: *src-ip*, *src-port*, *dst-ip*, *dst-port*, *protocol* and getter and setters methods.

5.3 CNI Network Plugin

The Container Network Interface(CNI) is a Cloud Native Computing Foundation projects. With the CNI, we have a unified interface for network services and we should only implement our network plugin once, and it should works everywhere which support the CNI. 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. When container runtime starts a container reads a json file and then network plugin do the magic to connect container. 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. The 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). If there are multiple CNI configuration files in the directory, the kubelet uses the configuration file that comes first by name in lexicographic order.

5.3.1 How it works

In CNI specifiction, there're three method we need to implement for our own plugin.

- `Add`: will be invoked when the container has been created. The plugin should prepare resources and make sure that container with network connectivity.
- `DELETE`: will be invoked when the container has been destroyed. The plugin should remove all allocated resources.
- `VERSION`: shows the version of this CNI plugin.

For each method, the CNI interface will pass the following information into the plugin:

- `ContainerID`: it is the target ContainerID.
- `Netns`: Network Namespace path of the container.
- `IfName`: Interface name should be created in the container.
- `Path`: The current working PATH, you should use it to execute other CNI.
- `StdinData`: Configuration file of the CNI.

First step is to write configuration file of CNI plugin that container runtime will pass to the plugin under StdinData value. When container runtime creates a container, it calls CNI network plugin passing config file and informations shown previously. The CNI network plugin is a binary, not a daemon. This means that every information that is necessary in the future must be saved in an external store or Database. After the CNI plugin is invoked it can invoke multiple IPAM (IP Address Management) plugin taking its relevant informations from configuration file. This is also a binary invocation and not a daemon. The IPAM will reply with a configuration that is used by the network plugin. After the CNI network plugin uses response from IPAM and network configuration to set up networking for the container.

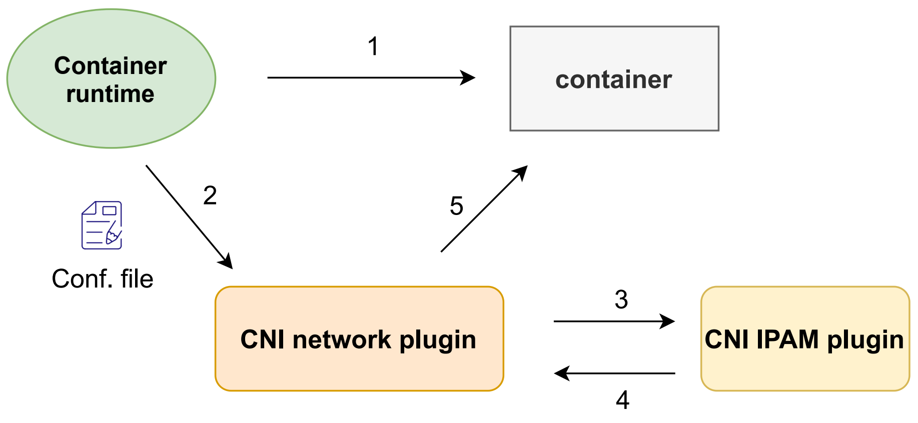


Figure 5.2: CNI network plugin setup/cleanup networking of container

5.3.2 Implementation

Configuration file of CNI contains useful information for our CNI like `Maximum Transfer Unit` and subnet used by IPAM that in this case is node `Pod Cidr`. C

Listing 5.5: CNI configuration file

```
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "polycube",
    "mtu": %s,
    "ipam": {
        "type": "host-local",
        "subnet": "%s",
        "rangeStart": "%s",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

configuration file is created by a go program before pcn-k8s operator starts. This program fills subnet by Node Pod CIDR that is retrieved by the Spec.PodCIDR value in the resource Node. Mtu is taken by a config map previously deployed. This json file is passed every time CNI plugin is called. IPAM (IP Address Management) provides some method to handle the IP/Route management. `host-local` IPAM plugin allocates IP addresses out of a set of address ranges. It stores the state locally on the host file-system, therefore ensuring uniqueness of IP addresses on a single host.

ADD function

Runtime container invokes ADD function every time a new container is created. In Kubernetes the `Kubelet` invokes ADD function every time a new Pod is created. In this project the add functions creates a veth-pair and allocates an IP for Pod retrieved by host-local IPAM plugin.

Listing 5.6: Creation of veth pair

```
func setupVeth(netns ns.NetNS, ifName string, mtu int) (*current.
    Interface, *current.Interface, error) {
     contIface := &current.Interface{}
     hostIface := &current.Interface{}

     err := netns.Do(func(hostNS ns.NetNS) error {
          // create the veth pair in the container and move host end
    into host netns
          hostVeth, containerVeth, err := ip.SetupVeth(ifName, mtu,
    hostNS)
          if err != nil {
               return err
          }
          contIface.Name = containerVeth.Name
          contIface.Mac = containerVeth.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 to lookup %q: %v",
    hostIface.Name, err)
    }
    hostIface.Mac = hostVeth.Attrs().HardwareAddr.String()

    return hostIface, contIface, nil
}

{
// run IPAM plugin and get back the config to apply
    r, err := ipam.ExecAdd(conf.IPAM.Type, args.StdinData)
    if err != nil {
        return err
    }

    // Invoke ipam del if err to avoid ip leak
    defer func() {
        if err != nil {
            ipam.ExecDel(conf.IPAM.Type, args.StdinData)
        }
    }()

    // Convert whatever the IPAM result was into the current Result
    type
    result, err := current.NewResultFromResult(r)

    ...

    hostInterface, containerInterface, err := setupVeth(netns, args.
    IfName, conf.MTU)
}
```

After that a new `pcn-loadbalancer-rp` is created with name "lbp-<PodIP>" and connected between Pod and `pcn-simplebridge`.

Listing 5.7: Creation of Load balancer

```
func createLbrp(ip string) (string , error) {
    // create lbrp with pod ip so it can be referenced by operator
    name := "lbrp-" + ip


    lbrpPortBackend := lbrp.Ports{
        Name: "to_switch",
        Type_: "BACKEND",
    }
    lbrpPortFrontend := lbrp.Ports{
        Name: "to_pod",
        Type_: "FRONTEND",
    }
    lbrpPorts := []lbrp.Ports{lbrpPortFrontend,lbrpPortBackend}
    lb := lbrp.Lbrp{
        Name: name,
        Ports: lbrpPorts,
        Loglevel: "DEBUG",
    }
    if response, err := lbrpAPI.CreateLbrpByID(context.TODO(), name,
   lb); err != nil {
        log.Errorf("An error occurred while trying to create lbrp %s:
    error: %s, response: %+v", name, err, response)
        return "",err
    }
    log.Infof("lbrp %s successfully created", name)


    return name, nil
}
```

This name is used this name is used in such a way `pcn-k8s` operator can retrieve the loadbalancer easily when POD is created and add services rules to give service connectivity to the Pod.

DELETE function

`Kubelet` invokes DELETE function every time a Pod is deleted. DELETE function is used to remove all resources allocated for that POD. So k8s-switch port and the pod's pcn-loadbalancer-rp are deleted. host-local IPAM plugin is invoked with it's delete function to remove allocation of that IP so it can be used again.

VERSION function

Version function returns version supported by the CNI PLUGIN.

5.4 Pcn-k8s Kubernetes Operator

`pcn-k8s` leverages Polycube services shown in 4 to provide network support for pods running in Kubernetes. It supports the cluster Kubernetes networking model 3, ClusterIP and NodePort services. This operator is synchronised with the Kubernetes API server and dynamically reconfigures the networking components in order to keep connectivity. **Overlay networking** VxLAN is used to connect nodes. This is useful when nodes are on different subnets and the user does not have direct control over the physical network. Basically, a controller will periodically match the state of the system to the to-be state. For that purpose, several functionalities are required.

- We need to be able to keep track of the state of the system. This is done based on an event-driven processing and handled by **informers** that are able to subscribe to events and invoke specific handlers and **listers** that are able to list all resources in a given Kubernetes cluster
- We need to be able to keep track of the state of the system. This is done using **object stores** and their indexed variants **indexer**
- Ideally, we should be able to process larger volumes using multi-threading, coordinated by **queues**

5.4.1 Queues and concurrency

The ability to easily create threads (called go-routines) in Go and the support for managing concurrency and locking are one of the key differentiators of the Go programming language, and of course the Kubernetes client library makes use of these features. Essentially, a queue in the Kubernetes client library is something that implements an interface that contains the following methods:

- **Add** adds an object to the queue
- **Get** blocks until an item is available in the queue, and then returns the first item in the queue
- **Done** marks an item as processed

Internally K8s working queue uses Go maps. THe keys of these maps are arbitrary objects. One of these maps is called the dirty set, this map contains all elements that make up the actual queue, i.e. need to be processed. The second map is called the processing set, these are all items which have been retrieved using Get, but for which Done has not yet been called. As maps are unordered, there is also an array which holds the elements in the queue and is used to define the order of processing. Note that each of the maps can hold a specific object only once, whereas the queue can hold several copies of the object.

If we add something to the queue, it is added to the dirty set and appended to the queue array. If we call Get, the first item is retrieved from the queue, removed from the dirty set and added to the processing set. Calling Done will remove the element from the processing set as well, unless someone else has called Add in the meantime again on the same object – in this case it will be removed from the processing set, but also be added to the queue again. Reading from queue channels is used to synchronize writers and readers.

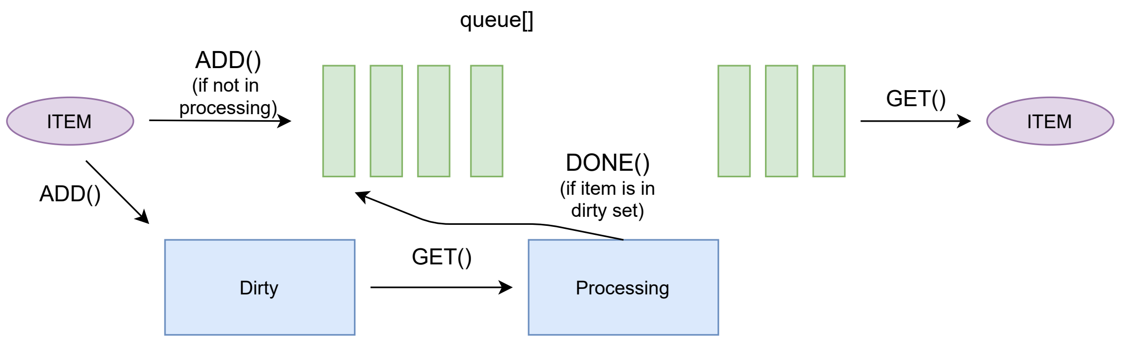


Figure 5.3: Working queue scheme

5.4.2 Informers, Reflectors and Indexers

The client-go package comes with a subpackage that makes getting events easy: `k8s.io/client-go/tools/cache`. It allows us to easily add functions that will be called when certain events come in. It also allows us to store all of the objects in memory easily which is called a Store. However, there's another package that ties the concepts the cache package provides into one: the `k8s.io/client-go/informers` package. It comes with a simple factory for all Kubernetes resources that nearly mirrors the kubernetes.Interface type. An **informer** defined in the base controller inside package cache pops objects from the Delta Fifo queue. The job of this base controller is to save the object for later retrieval, and to invoke our controller passing it the object.

A **reflector**, which is defined in type Reflector inside package cache, watches the Kubernetes API for the specified resource type (kind). The function in which this is done is `ListAndWatch`. The watch could be for an in-built resource or it could be for a custom resource. When the reflector receives notification about existence of new resource instance through the watch API, it gets the newly created object using the corresponding listing API and puts it in the Delta Fifo queue inside the `watchHandler` function.

An **indexer** provides indexing functionality over objects. It is defined in type Indexer inside package cache. A typical indexing use-case is to create an index based on object labels. Indexer can maintain indexes based on several indexing functions. Indexer uses a thread-safe data store to store objects and their keys. There is a default function named MetaNamespaceKeyFunc defined in type Store inside package cache that generates an object's key as <namespace>/<name> combination for that object.

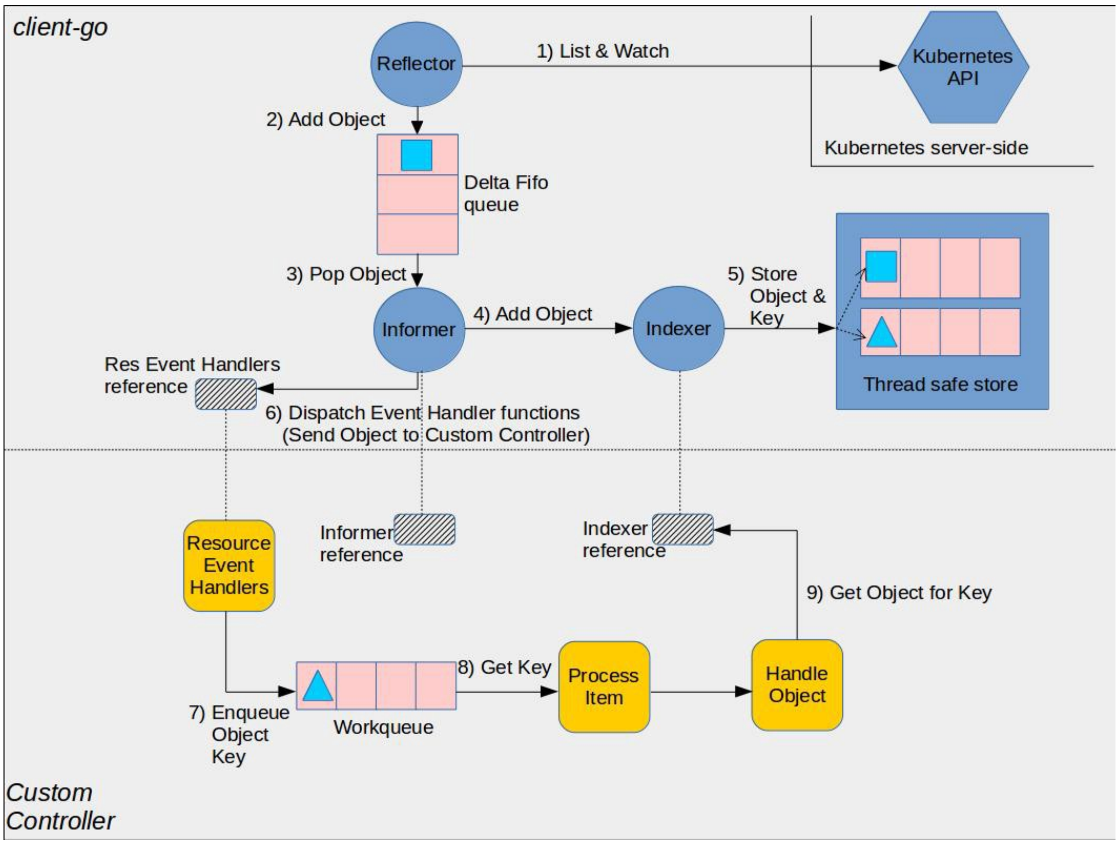


Figure 5.4: Client-go custom controller components [22]

5.4.3 How it works

`pcn-k8s` is a daemonset so there is one pcn-k8s for node. When the daemonset is deployed polycubed is started. The controller waits for it to becomes ready and then creates informers used to watch three Kubernetes resources: `Nodes`, `Pods` and `Endpoints`. Node controller allows to update the VXLAN table every time a new Node is added to the cluster or deleted. Subnet IP of VXLAN is taken from the configmap.

Listing 5.8: Polycube configmap

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: polycube-config
  namespace: kube-system
data:
  # Mtu to be configured in the pods.
  # If the all nodes are running on the same datacenter, 1500 can be
    used
  # otherwise 1450 has to be used due to the tunneling overhead
  mtu: "1450"

  # ServiceClusterIP range, should be the same as "service-cluster-ip
    -range"
  # passed to the api server
  serviceClusterIPRange: "10.96.0.0/12"

  # Range used for node port services, if should modify it if you
    specified
  # the "--service-node-port-range" flag. Default is "30000-32767"
  serviceNodePortRange: "30000-32767"

  # range used for the VTEPs on the overlay network. Choose any
  # non conflicting /16 range.
  vtepsRange: "10.18.0.0/16"
---
```

When a new Pod is created, a pcn-loadbalaner-rp is created for that Pod. This allow pod-to-service communication. Pod controller is used to fill existing Services in the new loadbalancer-rp of that Pod when it is created. The set of Pods targeted by a Service is usually determined by a selector and they are called Endpoints. So when a new service is added to the Loadbalancer, also Endpoints are inserted in the backend table for that Service. In such a way when the Pod contact a service the pcn-loadbalancer-rp check if Destinatio IP is a service. If yes it traduce it with one of the Endpoints.

Endpoint controller is responsible of checking service updates. Endpoint are Service backend Pods. From an Endpoint object, the service that cover it can be retrieved from the object itself. In this way we can observe also the Services. The controller is notified every time a new endpoint is created, deleted or updated. If a new Endpoint is created first of all is checked which service belongs to: ff it is a new Service all loadbalancer-rp in the node receives a request for creating a new service with endpoints covered by it. If not Endpoint is added to array of backends for that service. When there is no more Endpoints for a Service, this one is deleted from Services map of all LoadBalancers. If the service has externalTrafficPolicy=`Local`, each ingress Node LoadBalancer has only backends contained in that Node.

Chapter 6

Results

This chapter analyzes the network performance of the proposed architecture compared to similar scenario without the CNI. To test the performances in a single Node the machine used is equipped with:

- `CPU`: Intel Core i7-6700 @ 3.40GHz x 8
- `Memory`: 32 GB DDR4 2133 MT/s
- `NIC`: 1 Gbit/s port
- `Os`: Ubuntu 20.04
- `Kernel version`: 5.8.0-59-generic

The program used to test the network is iperf3 [23]. iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks. It supports tuning of various parameters related to timing, buffers and protocols (TCP, UDP, SCTP with IPv4 and IPv6). For each test it reports the bandwidth and other parameters. Tests are performed in TCP and UDP changing the length of buffers to read or write with `-l` flag, and socket buffer sizes to the specified value in `-w` flag.

6.1 Communication in the same Node

First test proposed is iPerf3 client-server communication in the same node. After evaluation of localhost performance, this one is compared with direct Network namespaces communication in the same Node and then Network namespace-to-service that has a backend in the same Node.

6.1.1 Localhost

The first test done is launch an iPerf server and client in Localhost.

TCP

Following plots shows TCP benchmark varying the TCP window size: 0 (Default), 64KB, 300KB.

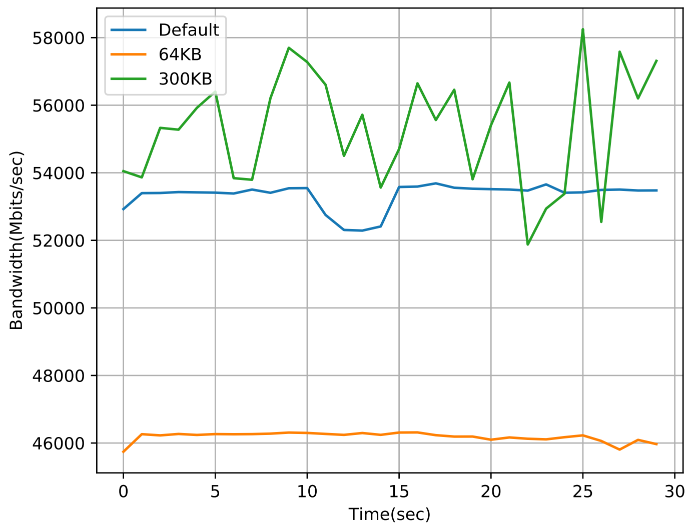


Figure 6.1: TCP bandwidth in Localhost scenario varying TCP window size

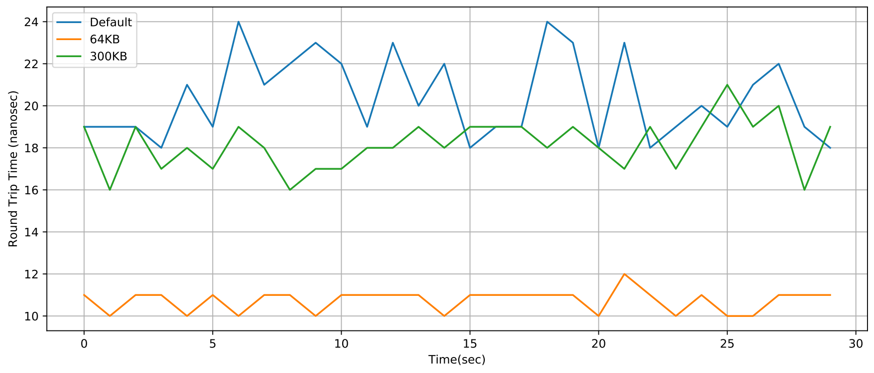


Figure 6.2: TCP RTT in Localhost scenario varying TCP window size

The Default value depends from host machine, kernel and OS. Observing file located in /proc/sys/net/ipv4/tcp_rmem it has those three values: 4096 131072 6291456 that are minimum window, default window and maximum window respectively. Default value is not easy to study because it depends by Operating System itself, but greater is the value of the TCP window size greater is the bandwidth and greater the RTT value.

Following plots shows TCP benchmark varying the length of buffers to read or write. iPerf works by writing an array of len bytes a number of times. Default is 128 KB for TCP, 8 KB for UDP.

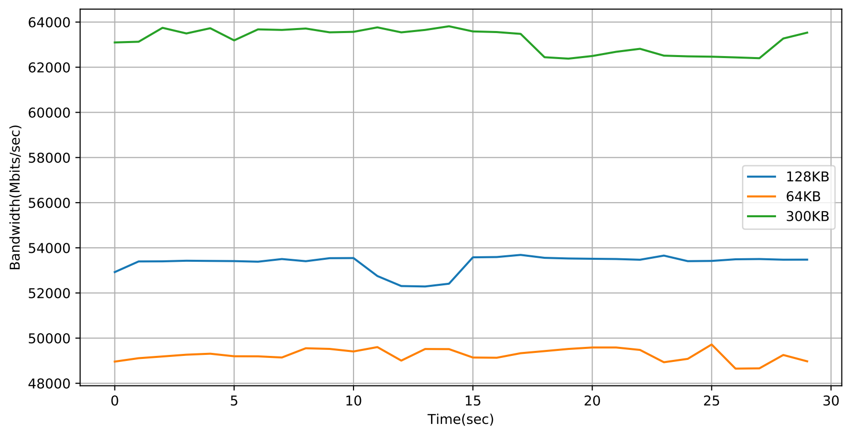


Figure 6.3: TCP bandwidth in Localhost scenario varying buffer length

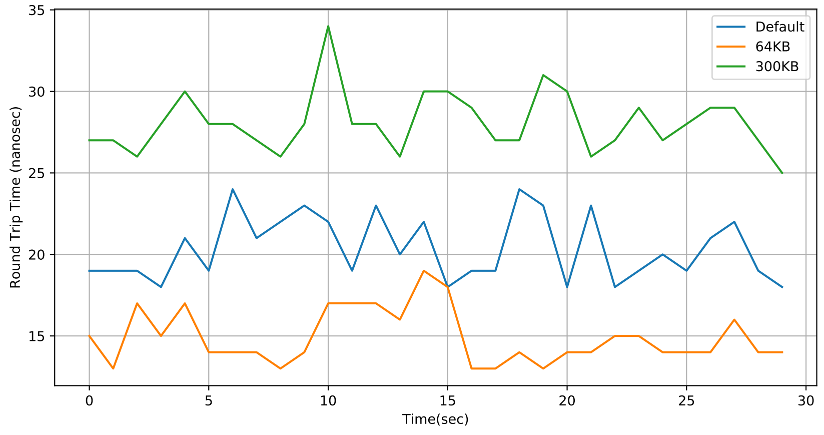


Figure 6.4: TCP RTT in Localhost scenario varying buffer length

Increasing the buffer length we have overall better bandwidth than before, but RTT is quite worst than before.

6.1.2 UDP

For UDP same test is done similar then before. -l flag instead of changing the TCP windows size, in this case changes the socket buffer size.

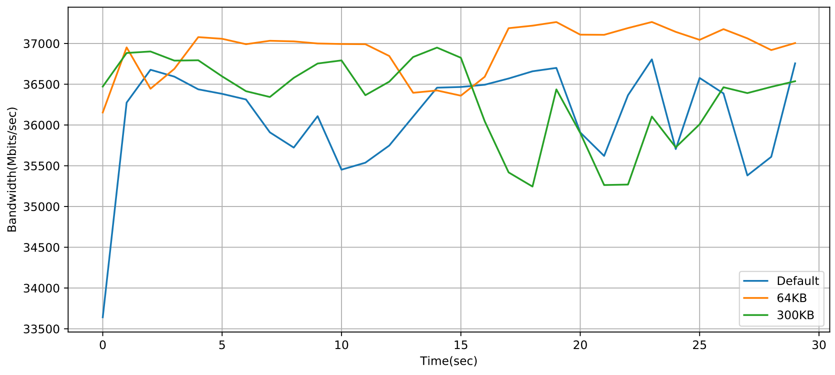


Figure 6.5: UDP bandwidth in Localhost scenario varying socket buffer size

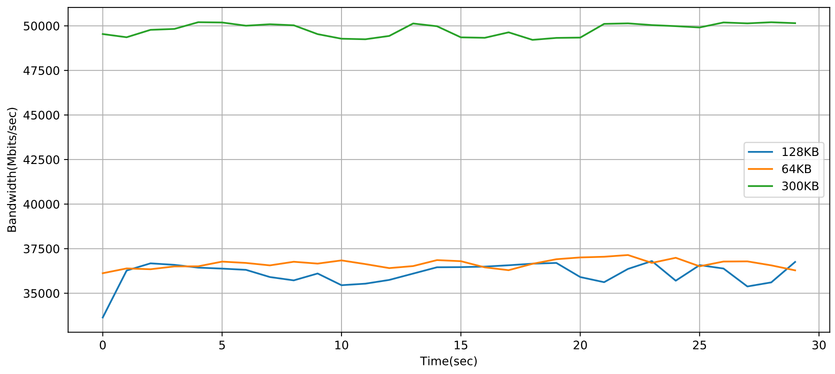


Figure 6.6: UDP bandwidth in Localhost scenario varying buffer length

Each frame goes through several buffers as you send it: The application buffer, The Protocol Buffer, The Software interface buffer and the Hardware interface buffer. As we start stressing the stack by sending high speed data, buffers will fill up and either block or lose data. Also strategies for timeliness and polling can impact performances. For example, by using a larger buffer and poll less often we can get much better performance while sacrificing latency.

TCP is optimized for high speed bulk transfers while UDP is optimized for low latency in the Linux kernel. This has an impact on buffer sizes and how data is polled and handed over. Sending high speed data over UDP is usually a bad idea, unless a congestion control is implemented. TCP protects the from congestion collapses. UDP is preferred when small amounts of data is exchanged or high timeliness is required.

6.1.3 Pod to Pod same Node

In this scenario two Pods and pcn-simplebridge are involved because Pods are in the same subnet.

TCP

Following plots shows TCP benchmark done like in Localhost scenario

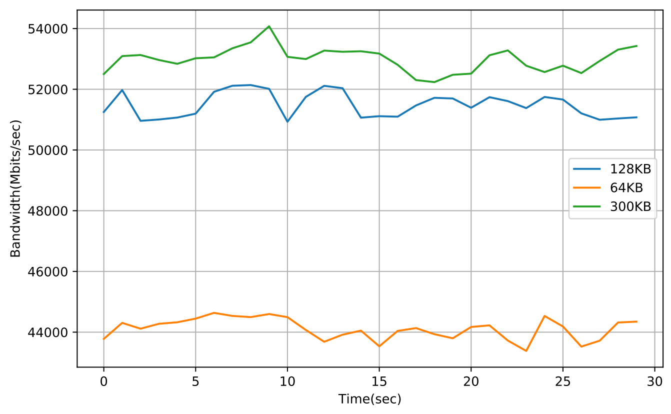


Figure 6.7: TCP bandwidth in Pod-to-Pod in the same node scenario varying TCP window size

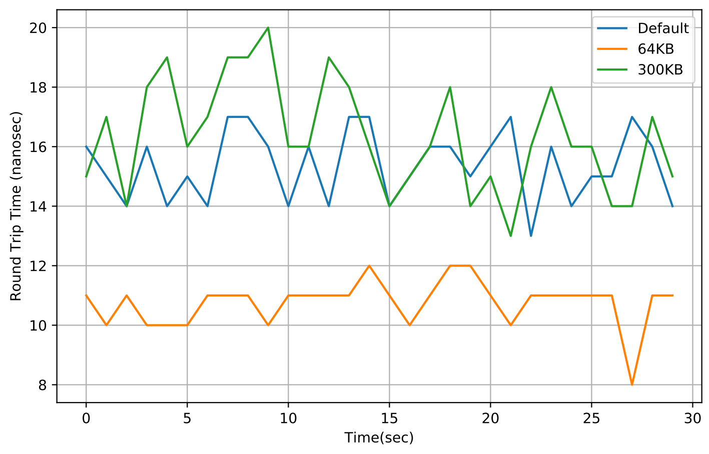


Figure 6.8: TCP RTT in Pod-to-Pod in the same node scenario varying TCP window size

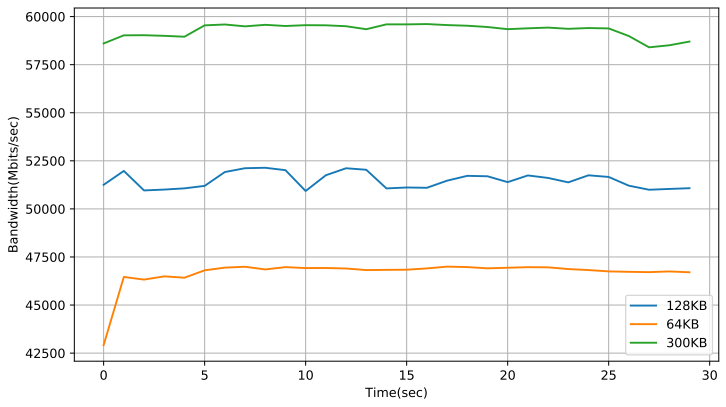


Figure 6.9: TCP bandwidth in Pod-to-Pod in the same node scenario varying buffer length

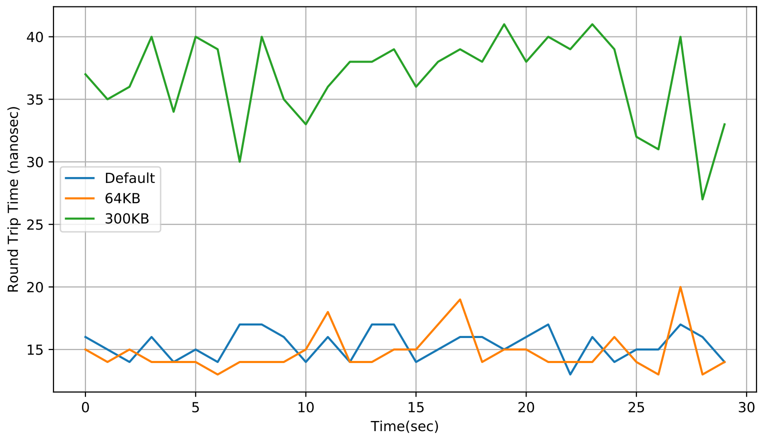


Figure 6.10: TCP RTT in Pod-to-Pod in the same node scenario varying buffer length

UDP

While TCP performance is similar to localhost scenario UDP it is more worst. This is caused by MTU set in veth-pair interfaces: localhost has a MTU equal to 64 KB while Pod MTU is equal to 1450 B because VXLAN has an header of 50 bytes. So another test is performed changing the Veth-pair MTU to 32 KB and not 64 KB because the latter is more than the maximum supported by veth-pair itself. With

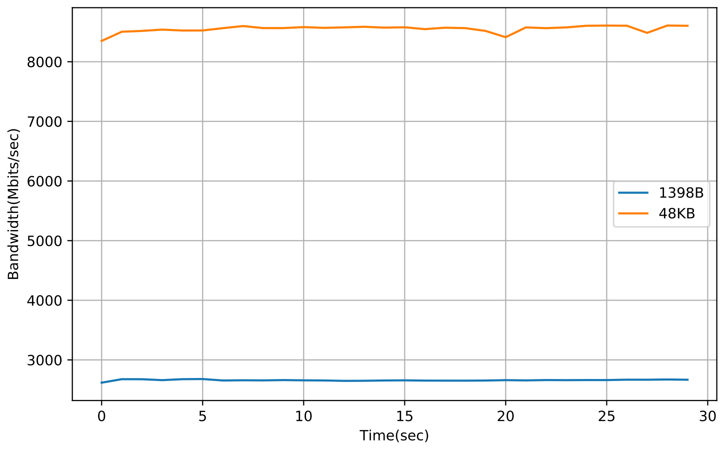


Figure 6.11: UDP bandwidth in Pod-to-Pod in the same node scenario varying socket buffer size

this larger MTU, UDP bandwidth becomes more acceptable like shown in the plot 6.12.

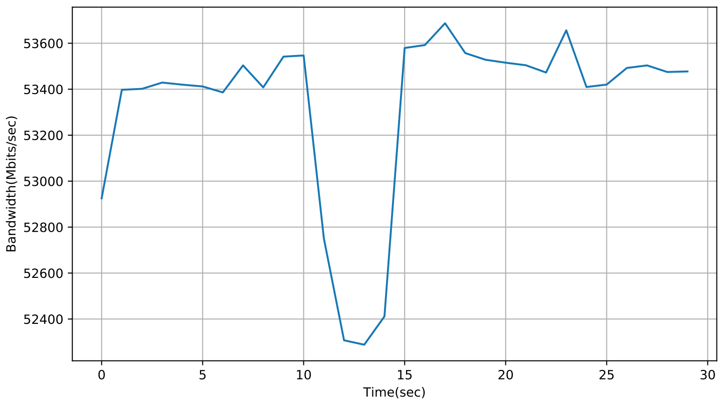


Figure 6.12: UDP bandwidth in Pod-to-Pod in the same node scenario with Pods MTU = 32KB

6.1.4 Pod to Service same Node

TCP

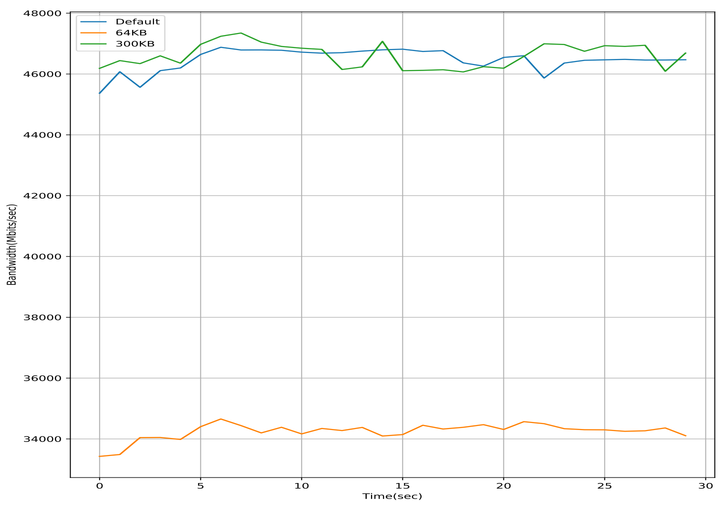


Figure 6.13: TCP bandwidth in Pod-to-Svc in the same node scenario varying TCP window size

UDP

In average Pod to Service they are bit worse because of the packet translation from svcIP:svcPort to backendIP:backendPort and vice versa performed by pcn-loadbalancer-rp.

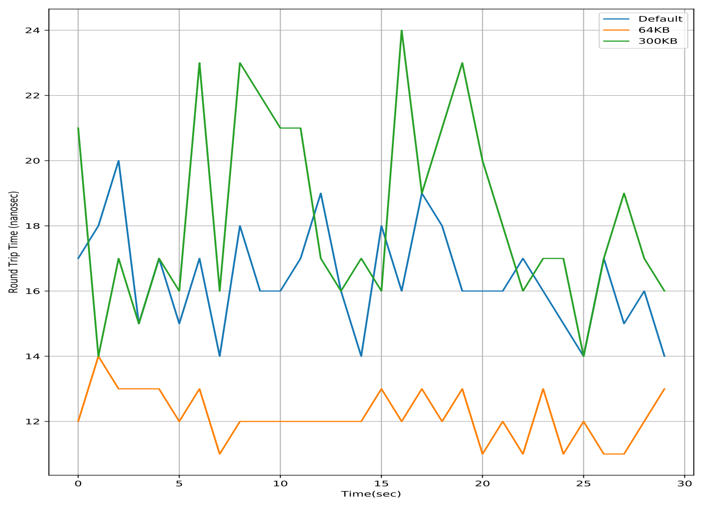


Figure 6.14: TCP RTT in Pod-to-Svc in the same node scenario varying TCP window size

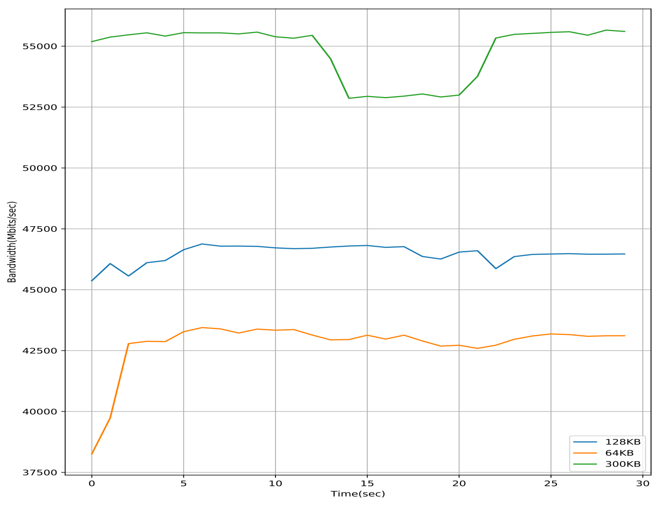


Figure 6.15: TCP bandwidth in Pod-to-Svc in the same node scenario varying buffer length

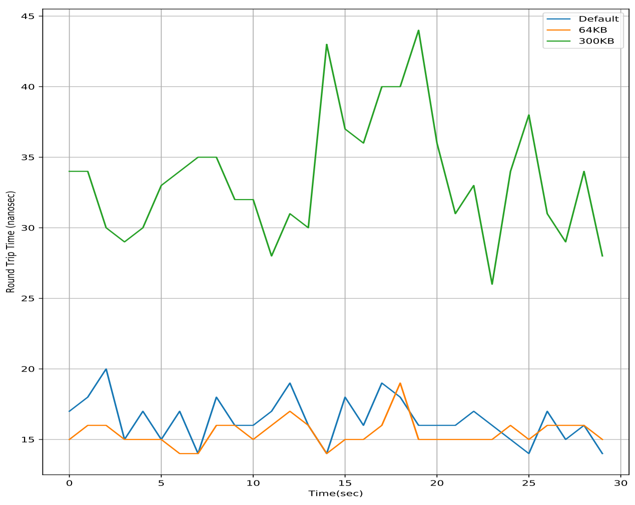


Figure 6.16: TCP RTT in Pod-to-Svc in the same node scenario varying buffer length

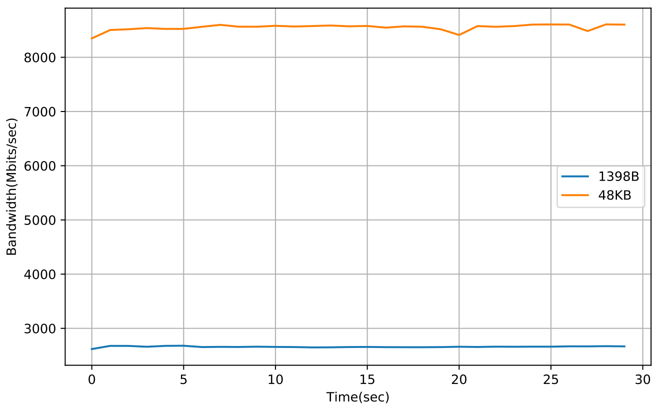


Figure 6.17: UDP bandwidth in Pod-to-Svc in the same node scenario varying socket buffer size

6.2 Communication in different Nodes

In this scenario communication cross Nodes are tested. For those tests 2 servers are used connected directly at 40G/s speed. Server1 hardware specs are:

- `CPU`: Xeon E3-1200 v5/E3-1500 v5/6th Gen Core Processor (x16)

- `Memory`: 16 GB
- `NIC`: 40 Gbit/s port
- `Os`: Ubuntu 18.04
- `Kernel version`: 5.8.0-59-generic

Server2 hardware specs are:

- `CPU`: Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz (x14)
- `Memory`: 187 GB
- `NIC`: 40 Gbit/s port
- `Os`: Ubuntu 18.04
- `Kernel version`: 5.8.0-59-generic

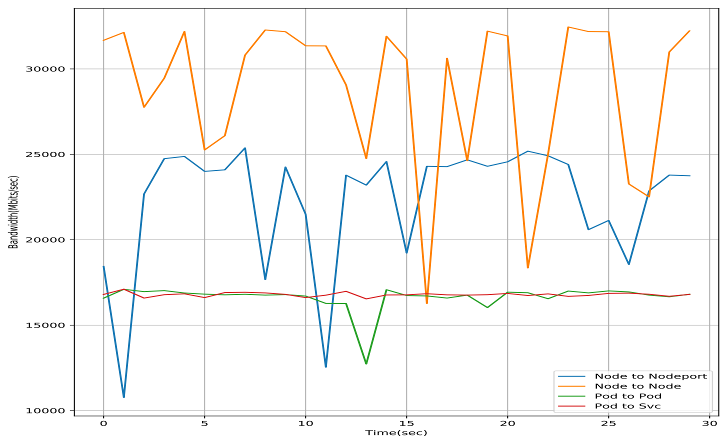


Figure 6.18: TCP bandwidth Node to Node scenario with socket buffer size = 300KB

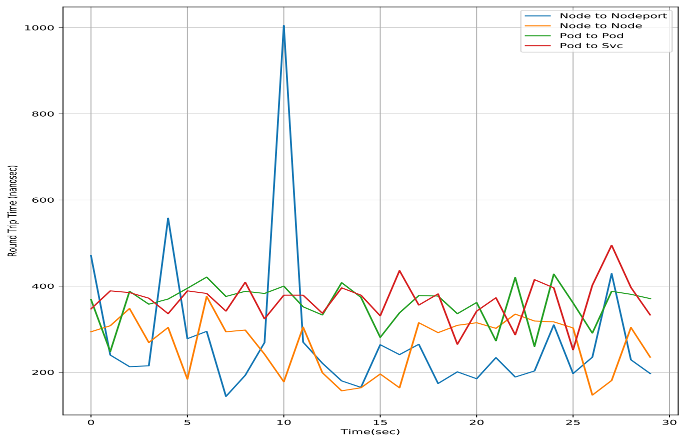


Figure 6.19: TCP RTT Node to Node scenario with socket buffer size = 300KB

6.3 Comparison with different CNIs

In this section different CNIs are compared in Pod to Pod/Svc communication in the same Node. Testing in different Nodes is not performed because this project doesn't support direct routing for now. To test the CNIs a single Kubernetes node is setup with Kubeadm while this project is tested with script shown here A.1.

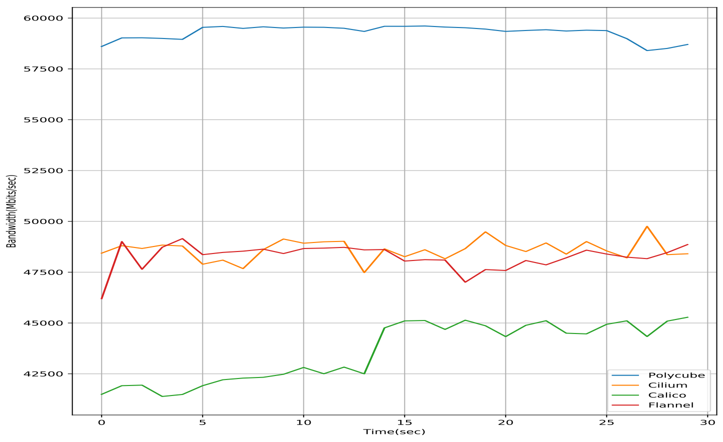


Figure 6.20: TCP bandwidth in Pod to Pod scenario with socket buffer size = 300KB

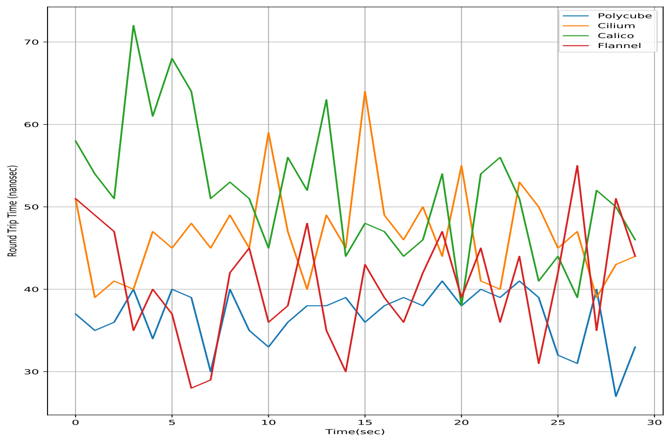


Figure 6.21: TCP RTT in Pod to Pod scenario with socket buffer size = 300KB

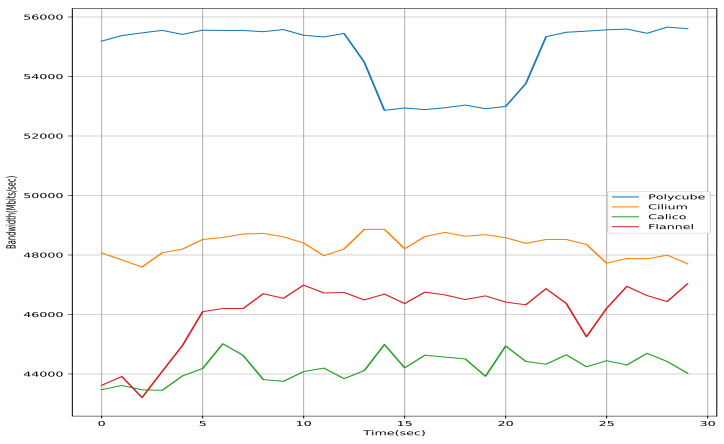


Figure 6.22: TCP bandwidth in Pod to Svc scenario with socket buffer size = 300KB

Polycube performance is much better than others CNI. CNIs values may be afflicted by utilization of Pod container while Polycube test is done with Network namespace. When `pcn-k8s` is ready this project will be tested with kubeadm too.

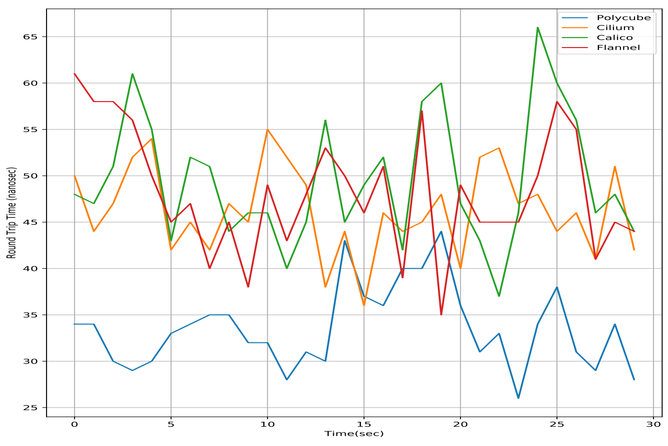


Figure 6.23: TCP RTT in Pod to Svc scenario with socket buffer size = 300KB

Chapter 7

Conclusions

This thesis has presented a prototype of high-performance modular CNI network plugin for Kubernetes running into the Linux kernel thanks to the use of eBPF/XDP.

The architecture proposed is composed by four Polycube network services: pcn-loadbalaner-rp, pcn-simplebridge, pcn-router, pcn-k8sdispatcher. All those actors connected together and coordinated by pcn-k8s operator allows cluster-networking.

In the implementation phase different challenges have been faced due to the cluster networking rules. Evaluate which existing Polycube services can be used, how to interconnect them and which functionalities needed is not provided and then, create a new Service for it.

The modular feature of this project allows you to update existing services or insert new network functions to get more features such as security, network policies and/or observability.

7.1 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 researc hpoint of view. Some of the future works that can be based on this thesis work can be the following. Having an indipendent Loadbalancer-rp for each pod cause to update each one of them every time a Service update comes. This can be overcome using eBPF Pinned Maps or create a loadbalancer-rp with multiple ports. The first solution consist of convert Service map and Backend map from BPF_TABLE to BPF_TABLE_PINNED. Pinned Maps consist of a map that is created by the first service that use it and the others can read it. With this strategy we can update only one Loadbalancer and the others will see the updates. The second one consist to create one loadbalancer-rp ad hoc for this project: this new service must support multiple ports and acts also as a switch. When a packet comes in the other ports acts as a backend port. If the destination is one of them not only Port and IPis changed but also MAC address. In fact with the current topology when a Loadbalancer choose a Pod in the same node the packet flows to the router and then comes back to the backend Pod, because Mac destination address is Default Gateway. Another feature to introduce is support for Direct Routing and Network Policies.

Appendix A

Scripts and Commands

To perform polycube network plugin tests in 6 topology testing is created with the following linux script:

Listing A.1: Topology creation

```
#!/bin/bash

function cleanup {
    set +e
    polycubectl del br1ns
    polycubectl del r1
    polycubectl del k1
    polycubectl del lb1
    polycubectl del lb2
    polycubectl del lb3
    sudo ip link del vxlan0
    sudo bridge fdb delete to 00:00:00:00:00:00 dst 192.168.0.191 dev
     vxlan0
    for i in `seq 1 2`; do
    sudo ip link del veth${i}root
    sudo ip netns del ns${i}
    #sudo ip link del pc_veth_pc
    done
}
trap cleanup EXIT
set -x
set -e
# Create 2 namespaces and the related veth
for i in `seq 1 2`; do
    # Create new namespace
    sudo ip netns add ns${i}
    # Add new veth interface
    sudo ip link add veth${i}root type veth peer name veth${i}ns
    sudo ip link set dev veth${i}root mtu 1450
    sudo ip link set dev veth${i}ns mtu 1450
    sudo ip link set veth${i}ns netns ns${i}
    # Enable veth on both root and newly created namespace
    sudo ip netns exec ns${i} ip link set dev veth${i}ns up
    sudo ip link set dev veth${i}root up
    # Set IP address to the namespace's end of the virtual interface
    sudo ip netns exec ns${i} ip addr add 10.10.7.${i}/24 dev veth${i}ns
    sudo ip netns exec ns${i} ip route add default via 10.10.7.254 dev veth${i}ns
    echo "Created ns${i} and veth${i} with IP 10.10.7.${i}/24"
done
# Create simple bridge br1ns
polycubectl simplebridge add br1ns
polycubectl lbrp add lb2 #loglevel=TRACE
# Create lbrps
polycubectl lbrp lb2 ports add port1 type=FRONTEND
polycubectl lbrp lb2 ports add port2 type=BACKEND
polycubectl lbrp add lb3 #loglevel=TRACE
polycubectl lbrp lb3 ports add port1 type=FRONTEND
polycubectl lbrp lb3 ports add port2 type=BACKEND
# Create and connect port 1
polycubectl br1ns ports add toveth1
polycubectl connect lb2:port1 veth1root
polycubectl connect lb2:port2 br1ns:toveth1
# Create and connect port 2
polycubectl br1ns ports add toveth2
polycubectl connect lb3:port1 veth2root
polycubectl connect lb3:port2 br1ns:toveth2
# Create and connect bridge port to router
polycubectl br1ns ports add to_router
# Create router
polycubectl router add r1 #loglevel=TRACE
# Create router port to bridge
polycubectl r1 ports add to_br1ns ip=10.10.7.254/24
# Create router port to physical interface
polycubectl r1 ports add to_internet mac=a0:8c:fd:ce:cb:01 ip=192.168.0.92/23
polycubectl connect r1:to_br1ns br1ns:to_router
#polycubectl connect r1:to_internet wlp1s0
polycubectl lbrp add lb1 loglevel=TRACE
polycubectl lbrp lb1 ports add port1 type=FRONTEND
polycubectl lbrp lb1 ports add port2 type=BACKEND

# Create K8dsispatcher
polycubectl k8sdispatcher k1 internal-src-ip=3.3.1.1 #loglevel=TRACE
polycubectl k1 ports add to_int type=FRONTEND
polycubectl k1 ports add to_router type=BACKEND
polycubectl connect r1:to_internet lb1:port2
polycubectl connect lb1:port1 k1:to_router
polycubectl connect k1:to_int eno1
# Add default route
polycubectl r1 route add 0.0.0.0/0 192.168.0.254
#polycubectl r1 address address=192.168.0.254 mac=00:62:ec:7d:72:71
    interface=to_internet
polycubectl r1 arp-table add 192.168.0.254 mac=00:62:ec:7d:72:71
    interface=to_internet
# Create vxlan interface
sudo ip link add vxlan0 type vxlan id 42 dev eno1 dstport 0
sudo bridge fdb append to 00:00:00:00:00:00 dst 192.168.0.191 dev
    vxlan0
sudo ip link set dev vxlan0 up
sudo ip addr add 10.18.0.1/16 dev vxlan0
polycubectl r1 ports add to_vxlan
polycubectl connect r1:to_vxlan vxlan0
polycubectl r1 route add 10.10.8.0/24 10.18.1.1

read -p "Press ENTER to delete current configuration..."
```

Following commands show how to add services to loadbalancer-rp and nodeport natting rule to k8sdispatcher

Listing A.2: Services and nodeport rule

```
    # Create service in lb2
    polycubectl lbrp lb2 service add 110.18.0.1 5201 ALL name=service
    # Add backend
    polycubectl lbrp lb2 service 110.18.0.1 5201 TCP backend add
   10.10.8.2 port=5201 name=B1 weight=10
    # Add nodeport rule to k8sdipatcher k1
    curl -d '{"internal-src": "13.13.13.13", "proto": "tcp", "
   nodeport-port": 30000, "service-type": "CLUSTER"}' -H "Content-
   Type: application/json" -X POST http://localhost:9000/polycube/v1/
   k8sdispatcher/k1/nodeport-rule/30000/tcp
```

All iperf3 commands are runned with -J flag to obtain a json file containing all output values. After that a python script is used to read json and generate a Plot.

Listing A.3: Python script

```
import re
import matplotlib.pyplot as plt
import numpy as np
import json

time_list = []
rate_list = []
rtt_list = []

```

```

f = open('./nodetonode/nodeport-tcp-default.json','r')# iperf-log.txt
    is the iperf log file name

data = json.load(f)
for i in data['intervals']:
    rate_list.append(i['streams'][0]["bits_per_second"]/1e6)
    time_list.append(i['streams'][0]["start"])
    rtt_list.append(i['streams'][0]["rtt"])

f.close

plt.figure()
plt.plot(time_list, rate_list, label = "label")
plt.xlabel('Time(sec)')
plt.ylabel('Bandwidth(Mbits/sec)')
plt.legend()
plt.grid()
plt.show()
```

Bibliography

[1] The ubuntu Authors. *CNI overview.* URL: `https://ubuntu.com/kubernetes/docs/cni-overview`. (accessed: 06.2021) (cit. on p. 2).

[2] Kedar Vijay Kulkani. *A brief overview of the Container Network Interface (CNI) in Kubernetes.* URL: `https://www.redhat.com/sysadmin/cni-kubernetes`. (accessed: 06.2021) (cit. on p. 2).

[3] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. «Large-scale cluster management at Google with Borg». In: *Proceedings of the European Conference on Computer Systems (EuroSys).* Bordeaux, France, 2015 (cit. on p. 3).

[4] Ferenc Hámori. *The History of Kubernetes on a Timeline.* June 2018. URL: `https://blog.risingstack.com/the-history-of-kubernetes/` (cit. on p. 3).

[5] Steven J. Vaughan-Nichols. *The five reasons Kubernetes won the container orchestration wars.* Jan. 2019. URL: `https://blogs.dxc.technology/2019/01/28/the-five-reasons-kubernetes-won-the-container-orchestration-wars/` (cit. on p. 4).

[6] Kalyan Ramanathan. *5 business reasons why every CIO should consider Kubernetes.* Oct. 2019. URL: `https://www.sumologic.com/blog/why-use-kubernetes/` (cit. on p. 4).

[7] 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).

[8] Diego Ongaro and John Ousterhout. «In search of an understandable consensus algorithm». In: *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14).* 2014, pp. 305–319 (cit. on p. 8).

[9] *Kubernetes official documentation.* URL: `https://kubernetes.io/docs/home/` (cit. on pp. 10, 12–14).

[10] *Kubernetes API official documentation.* URL: `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.17/` (cit. on p. 10).

[11] The Cilium Authors. *BPF and XDP Reference Guide*. URL: https://docs.cilium.io/en/v1.8/bpf/. (accessed: 06.2021) (cit. on p. 17).

[12] 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. 22).

[13] *Virtual Extensible LAN*. URL: https://en.wikipedia.org/wiki/Virtual_Extensible_LAN (cit. on p. 27).

[14] M. Mahalingam. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. URL: https://datatracker.ietf.org/doc/html/rfc7348. (accessed: 06.2021) (cit. on pp. 27, 39).

[15] Kubernetes Authors. *Cluster Networking*. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/. (accessed: 06.2021) (cit. on p. 30).

[16] Linux manual page Authors. *network_namespaces - overview of Linux network namespaces*. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html. (accessed: 06.2021) (cit. on p. 31).

[17] Linux manual page Authors. *veth - Virtual Ethernet Device*. URL: https://man7.org/linux/man-pages/man4/veth.4.html. (accessed: 06.2021) (cit. on p. 32).

[18] Kubernetes Authors. *Service*. URL: https://kubernetes.io/docs/concepts/services-networking/service/. (accessed: 06.2021) (cit. on p. 34).

[19] Kubernetes Authors. *Service IP addresses*. URL: https://kubernetes.io/docs/concepts/services-networking/service/#ips-and-vips. (accessed: 06.2021) (cit. on p. 35).

[20] The Linux Foundation Authors. *bridge*. URL: https://wiki.linuxfoundation.org/networking/bridge. (accessed: 06.2021) (cit. on p. 39).

[21] Swagger Authors. *OpenAPI Specification*. URL: https://swagger.io/specification/. (accessed: 06.2021) (cit. on p. 43).

[22] *sample-controller github repo*. URL: https://github.com/kubernetes/sample-controller/blob/master/docs/controller-client-go.md (cit. on p. 56).

[23] *Iperf3 official website*. URL: https://iperf.fr/ (cit. on p. 58).