POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Analysis and Benchmarking of Kubernetes Networking

Supervisor
prof. Fulvio Risso

Candidate
David Liffredo

Internship Tutor
dott. ing. Alessandro Capello

October 2020

Contents

Chapter 1

Introduction

In the last few years, the way of developing software applications has been modified, moving from a monolithic approach to a microservices approach. Instead of having a huge application that does everything, the application is divided into several independent parts, called microservices, that run autonomously and cooperate together to provide the requested service. In this new model, fundamental is the communication between microservices and the main app and consequently the network is becoming more and more important.

Another technology, which due to its characteristics fits perfectly with microservices, is becoming increasingly important and widely used today: containers. They represent a new solution in the field of software virtualization, replacing virtual machines (which also virtualize hardware) that have similar characteristics (therefore, for example, isolation and security) but are heavier. The lightweigh and a series of features (easy to compose and replace for example) make the containers perfect for hosting microservices.

However, the deployment of containers can result hard, especially when there are dependencies between them. For this reason an orchestrator that is automatically able to scale, create, delete and deploy components in a proper way is necessary, rather than installing the components manually or via scripts. The Kubernetes (K8s) framework helps users to solve these

problems, being the most popular container orchestrator nowadays.

It is an open-source system for automating deployment, scaling, and management of containerized applications. Starting from a cluster of hosts, which can be physical or virtual servers, the Kubernetes instance takes control by scheduling the containers, controlling their life making it easier to deploy containerized applications.

The basic element in Kubernetes is the pod, that hosts one or more containers and controls their lives. In the microservices pattern the communication between containers, and consequently between pods, is a key point.

Kubernetes, however, does not care about how its networking is implemented, allowing the installation of an external plugin, provided by third entity, which configures the components to allow the correct conduct of communications: this plugin is called network provider, Container Networking Interface (CNI), or simply CNI.

The CNIs, therefore, take care not only of connecting the container to the network, but more generally of the implementation and management of the entire Kubernetes networking and of the resources that influence it.

Choosing the appropriate network provider to install in the cluster may not be easy given the large amount of existing products.

1.1 Thesis goal

A Kubernetes cluster administrator, that wants to choose an appropriate CNI, could encounter difficulties due to the huge number of network providers available, without any practical feedback.

The first goal of the thesis is to create an easy to use and automatic benchmarking suite that measures performance indicators of the CNI installed in the cluster, testing it with the cluster configured in several different ways.

In particular, after having identified both ideal and daily usage scenarios for k8s clusters, a set of tests was defined to evaluate the behavior of CNIs

in such scenarios, by means of the developed benchmarking tool.

This thesis has been carried out in collaboration with Telecom Italia, that is interested in identifying the optimal CNI to install in a real k8s cluster.

The second objective is then, starting from a set of open source CNIs, to use the tool to identify the CNI with the best performance indicators.

Chapter 2

Background: Kubernetes

In order to give a basic comprehension of the context of work of this thesis, this chapter provides a brief description of Kubernetes and its main component.

2.1 Introduction

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

The cluster is the representation of Kubernetes when the platform is deployed. A cluster consists of a set of worker machines (at least one), called nodes, that run containerized applications [1]. A node can be a physical machine, for example a server, or a virtual machine deployed in a service. The node hosts the main unit that can be created in Kubernetes: the Pod.

It represents process running in the cluster and it is the smallest and essential object deployable in the cluster. It hosts one (or more) container, it shares network and storage resources between containers and manages its lifecycle, based on some information passed in the creation phase.

A Pod, in general, represents a unit of deployment: a single instance of an application in Kubernetes, which might consist of either a single container

or a small number of containers tightly coupled and that share resources.

2.2 Main components

Usually, some resources are deployed on the master node that are needed for the correct functioning of the whole framework: these objects, in literature, are classified as control plane components and they make global decisions about the cluster as well as detecting and responding to cluster events.

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

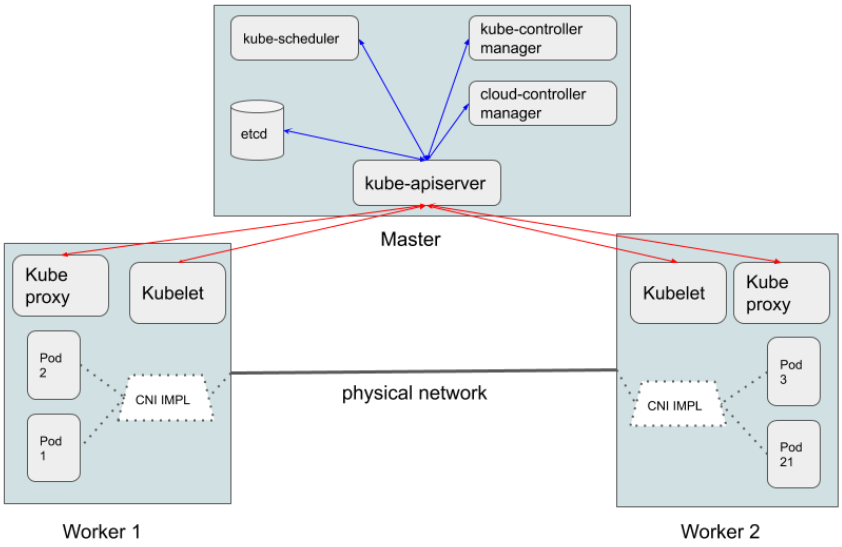

Figure 2.1. Cluster and main K8s components

In the master are deployed the following objects:

- kube-apiserver: the API Server acts as a front end for the Kubernetes control plane and exposes several Kubernetes APIs. The main implementation of a Kubernetes API server is kube-apiserver that is designed to scale horizontally: it is possible to run several kube-apiserver in the cluster, balancing requests.

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

- kube-scheduler: it is a control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on, taking into account several factors like resource requirements, constraints and so on.

- kube-controller-manager: it is a Control Plane component that runs controller processes (in theory each controller is a separate process, but they are all compiled into a single binary and run in a single process). Node controller (that checks nodes' life), replication controller (responsible for maintaining the correct number of pods for every replication controller object in the system), endpoints controller (that populates endpoints object) and service account and token controllers are included.

- cloud-controller-manager: the cloud-controller-manager only runs controllers that are specific to your cloud provider.

On the other hand, on each worker there are node components executed, that maintain running pods and provide the Kubernetes runtime environment:

- kubelet: an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

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

In user-cluster communication the main role is played by the kube-apiserver. In fact, the user supplies commands (e.g. with kubectl, the command line tool) which are translated into the Kubernetes REST API requests and sent to kube-apiserver. This object checks the received request and can do several things: if necessary, it can forward the request to another control plane component that can process the request or it can interact directly with worker nodes via the kubelet. The kubelet's job is to manage all messages directed to the node from the kube-apiserver and vice versa, so as not to allow the kube-apiserver to directly contact a pod running on the worker.

To better understand some terms used in this thesis, other K8s objects are presented below.

2.2.1 Daemonset

A Daemonset, called sometimes Daemon, ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created. This item is often used by CNIs as will be presented in the next chapter. Each network provider creates one or more Daemons objects, assigning them fundamental tasks for the correct functioning of the plugin and networking in general, so that they are automatically created on each node.

2.2.2 Service

In Kubernetes, pods are mortal and often have an unpredictable life with the consequence of difficult interaction between them. Furthermore, when created from a Deployment resource, this resource has complete control over their life (for instance it can decide to deploy a pod in one worker and then after some time moves it in another node). Having each pod its IP address is easy to think to contact through that, but several situations that can happen

in the cluster like above, can have impact on Pod life and consequently making its attributes, including IP address, unusables. So, for these reasons, contacting the pod directly becomes hard and risky.

In general, in many clusters exist several pods that live to produce data for other pods, but how can the latter be sure to always contact active pods correctly to get the information they need? For these and other situations the service resource plays an important role. The "Service is an abstraction which defines a logical set of Pods and a policy by which to access them."[2]

More simply, a service resource exposes work done by a set of "producer" pods as a network service, reachable via the service's IP address. This address, differently from that of pods, is fixed and never changes, until the service resource deletion. So, a pod that consumes this work, can contact the service directly, without controls over a specific producer pod lifecycle.

The traffic directed to service is forwarded, through an IP address translation from that of the service to a specific address of one pod that can process the request. This translation and especially the way it is done have a non-negligible impact on cluster networking and performance.

2.2.3 Network Policy

"NetworkPolicies are an application-centric construct which allows you to specify how a pod is allowed to communicate with various network "entities"..."[3]. With the Network Policies resources usage in the cluster, it is possible to allow or block the communication between pods in one direction (from pod1 to pod2) or in both directions.

A Network Policy definition has several important field, reported in the picture 2.2 :

1. it is linked to a set of pod through field called "podSelector" in the pod specification,

2. it specifies the policy types (Ingress policy, Egress policy or both),

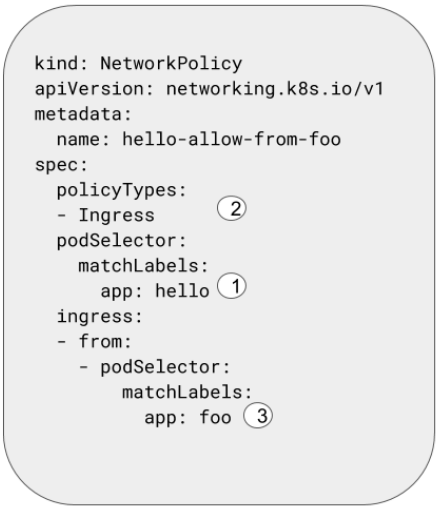

Figure 2.2. Definition of Network Policy

3. specifies also for who (pods that match the second podSelector) these policy types are allowed.

For all other pods that do not match the second podSelector the communication with the first set of pods is blocked.

The Network Policy, shown in the picture 2.2, "allows traffic to the Pods with label "app=hello" from only the Pods with label "app=foo". Other incoming traffic from Pods that do not have this label, external traffic, and traffic from Pods in other namespaces are blocked"[4].

Chapter 3

Container Networking Interface

In Kubernetes, networking is provided by the so called Container Network Interface (CNI). It consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins[5].

3.1 What is its goal and how it reaches it

Kubernetes defines three main requirements for its networking model:

1. All pods can communicate with all other pods without NAT.
2. All nodes running pods can communicate with all pods (and vice versa) without NAT.
3. IP address that a pod sees itself as is the same IP that other pods see.

These rules must be implemented by any network provider that wants to provide connectivity in Kubernetes, it does not care how. Network providers follow the Container Networking Interface (CNI) specification. This specification says, basically, how to add or remove containers to the network, all by

creating an executable plugin. Each CNI plugin, deployed as an executable invoked by Kubernetes, works on two different aspects:

- Containers management (container insertion, deletion and checks in a network).
- Precise setting of networking parameters both at host level and at single container level to allow communications to take place without errors.

Regarding the first aspect, there are three fundamental operations that each CNI must support, are fundamental:

1. the ADD operation: this operation is called when a container must be connected to the network and, taking some parameters about the new container, it configures an interface and all necessary components to correctly connect the container (usually to connect it to the network one end of a new veth pair is placed in the container, while the other end is connected directly to the host).
2. the DEL operation: it works in the opposite way to the previous one. It is used to disconnect a container from the network, releasing all the resources it uses.
3. The CHECK operation: it is important to check the container's network status and configuration (which must be, always, correct).

The second aspect, always closely related to the first, which a CNI generally deals with is the configuration and management of network parameters (IP addresses, routes installed and so on) in the host so that communication between containers is possible.

These are the minimum requirements a CNI provides, but Kubernetes networking is much more complex. In fact, it is not enough for the plugin to deal only with the management of containers, but must take into account and manage events, entities and situations that may occur in Kubernetes such as

the addition of a node to the cluster, or the management of other components that affect networking (for instance services and network policies) or the configurations necessary for connecting to the Internet.

The actual CNI (the container connection management) is therefore only a small part of the entire work provided by the network provider (and which is installed in Kubernetes).

However, unfortunately, in a misleading way the plugin installed in the cluster is very often called CNI (as well as network provider), even if it does not only perform tasks on the management of the container. In the next chapters, with the term CNI or CNI plugin will therefore mean the entire work done by the product in the Kubernetes networking.

3.2 Analysis of the main CNIs

For a cluster administrator, choosing an appropriate CNI can be difficult due to the huge number of network providers available. Some of the most used CNIs (mostly open source) are analyzed to highlight implementation differences from a theoretical point of view, and starting from this set some are tested through the benchmarking tool to underline performance differences.

3.2.1 Flannel

Flannel is one of the simplest CNIs that can be installed in the cluster. It runs a small, single binary agent called flanneld on each host (it is deployed as DaemonSet), and is responsible for allocating a subnet lease to each host out of a larger, preconfigured address space[6]. The flanneld creates some route rules in the kernel's route table to forward traffic. Flannel stores general network information (for instance network configuration or the allocated subnets) and any auxiliary data (such as the host's public IP) directly in the etcd provided by the framework.

In Flannel the encapsulation (default VXLAN[1]) is exploited to allow communication between pods. An overlay network, which runs above the host network, is created and an IP address from this overlay network is assigned to each pod deployed. To each node a set of IP addresses is given and when a pod is scheduled in this node, it receives a free IP from this set.

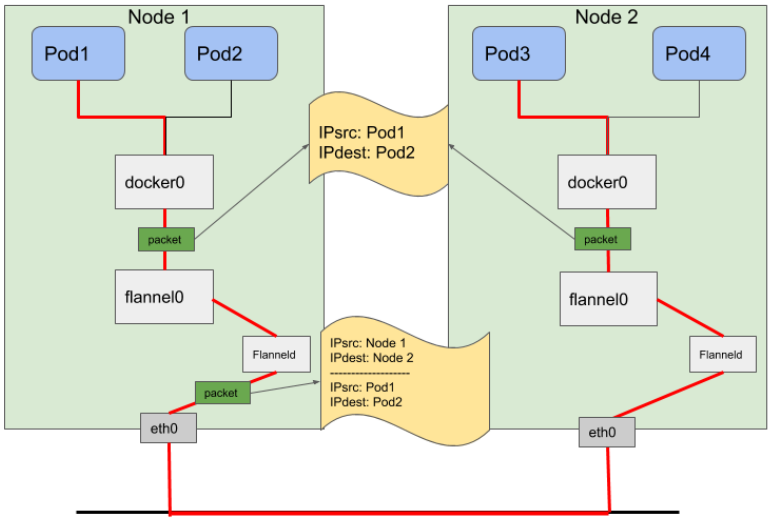

Figure 3.1. Flannel Packet flow in inter node communication

This CNI uses the default IPAM (IP Address Management) provided by the framework. The containers inside the same host can communicate with each other by the bridge "docker0" (a linux bridge that is configured by flannel), while for inter-node communication the encapsulation is mandatory. When the system realizes, using the kernel route tables, that a packet must be sent outside, the kernel forwards the packet to the flanneld process which, after contacting the etcd to find out which node to send the traffic to, adds the necessary headers to encapsulate the traffic and feed it into the network[7].

This plugin does not support both Network Policies and IPv6, while it

[1]https://en.wikipedia.org/wiki/Virtual_Extensible_LAN

exploits kube-proxy in iptables mode to allow services creation and usage.

3.2.2 Antrea

Antrea, designed by VMware[2], is a fairly recent CNI (born about two years ago) that operates at Layer 3/4 to provide networking and security services for a Kubernetes cluster, leveraging Open vSwitch to implement Pod networking and security features.

In a Kubernetes cluster, Antrea creates two main objects: a Deployment (Antrea Controller) and a DaemonSet that executes two containers (Antrea Agent and OVS daemons) on every Node. The DaemonSet also includes an init container that installs the CNI plugin - antrea-cni - on the Node and, in general, ensures that the OVS kernel module is loaded and it is chained with the portmap CNI plugin.

When installed, the network provider creates on each node an OVS bridge with several interfaces used in different kinds of communications. Antrea uses encapsulation (by default Geneve) and, as the reader can see in the picture 3.2, the OVS bridge directly forwards packets between two local Pods.

Indeed, for inter node communication, packets will be first forwarded to the tun0 port, encapsulated, and sent to the destination Node through the tunnel; then they will be decapsulated, injected through the tun0 port to the OVS bridge, and finally forwarded to the destination Pod.

Regarding security features, Antrea treats the network policies resource differently than other CNIs, in an absolutely centralized way. In fact, Antrea Controller watches NetworkPolicy, Pod, and Namespace resources from the Kubernetes API and it processes several components (PodSelectors, namespaceSelectors, ipBlocks) notifying only interested nodes. Each Antrea Agent (deployed on each node) receives only the computed policies which affect Pods running locally on its Node, and directly uses the IP addresses computed by

[2]https://www.vmware.com/

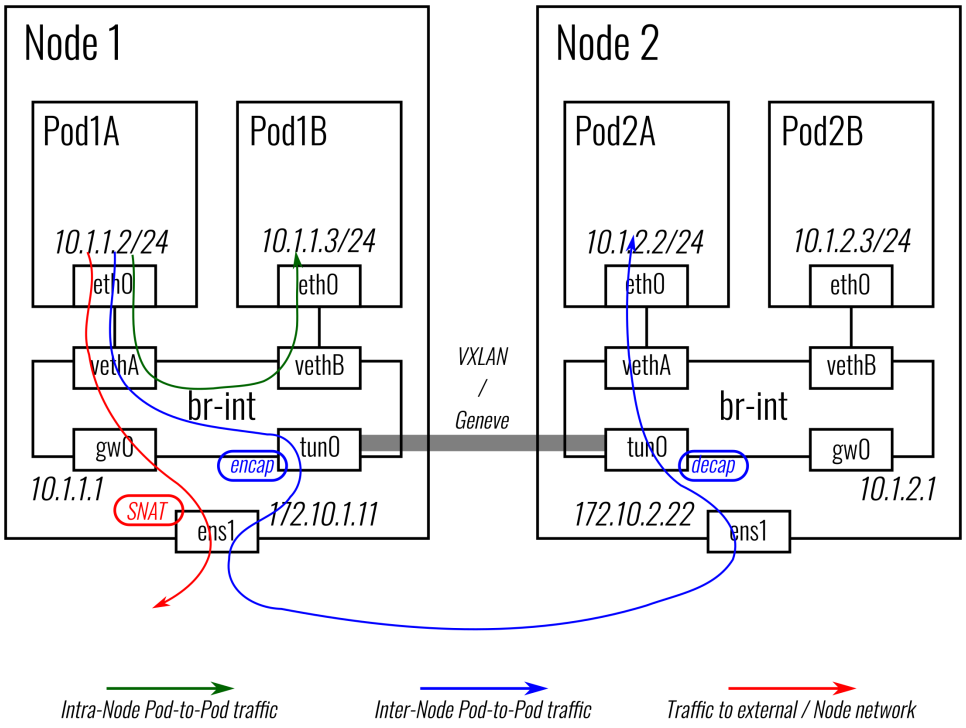

Figure 3.2. Antrea packet flow in different communications [8]

the Controller to create OVS flows enforcing the specified NetworkPolicies. In this way, only the nodes involved are notified, which can be non-secondary in clusters with many nodes.

Right now Antrea supports only IPv4 and allows encryption of inter-Node Pod traffic with IPsec tunnels when using an overlay Pod network.

3.2.3 Calico

Calico is one of the most installed and used network provider in Kubernetes clusters. It offers a complex but flexible and secure network solution, allowing the user to choose some configuration parameters. It supports both encapsulation (therefore the creation of overlay networks) and native routing (using BGP), based on user needs.

The product creates a series of components in the cluster (some daemons, an IPAM plugin) and saves all the network configurations installed in a cluster data store (which can be the etcd or something similar).

In each node are installed, among other daemons, Bird and Felix: these are the two main components of the plugin and are used, respectively, to forward and receive networking information between the nodes (forwarding and receiving BGP messages for example) and to modify the routing tables based on this information. Furthermore, Felix has the task of installing Access Control Lists (ACLs) in the Linux kernel of his node to allow only valid traffic to reach the various pods (he then implements the network policies) and notifies in case of error. The image 3.3 shows main components of Calico architecture.

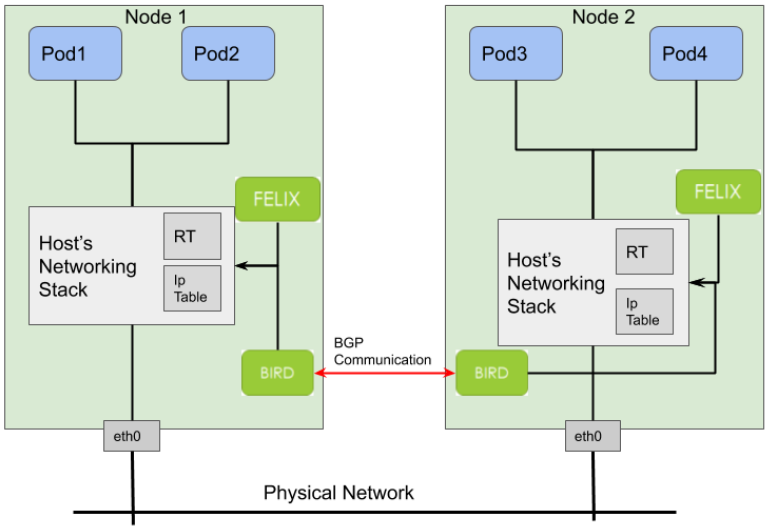

Figure 3.3. Calico architecture

Regarding packet flow, IP packets to or from a Pod are routed and firewalled by the Linux routing table and iptables infrastructure on the workload's host. For a Pod that is sending packets, Calico ensures that the host is always returned as the next hop regardless of whatever routing the Pod

itself might configure. Calico, for now, uses iptables to support services and network policies, but a trial version that takes advantage of eBPF technology (replacing iptables) is available.

Calico also introduced a new type of Network Policy. While Kubernetes network policy applies only to pods, Calico network policy can be applied to multiple types of endpoints including pods, VMs, and host interfaces. The Calico network policy is also portable in different cloud providers and it is possible to use this new network policy in addition to Kubernetes network policy, or exclusively.

An interesting aspect of Calico running in BGP mode concerns the possibility of announcing the IP addresses of the services outside the cluster. In general, it is difficult in Kubernetes accessing an internal service from outside the cluster. In a cluster created not in a cloud environment, the possibilities are the following: the installation of software (for example MetalLB) that emulate the behavior of Load type services Balancing or the creation of Node Port services (which however are available in limited numbers).

To overcome this problem, Calico allows (through specific commands) to install additional routes in the BGP messages exchanged with the border routers, so that they propagate the addresses of the internal services outside (being a command entered manually by the user, it must be consistent with cluster's addressing).

3.2.4 Contiv

Contiv-VPP is a Kubernetes CNI plugin, developed by Cisco, that employs a programmable vSwitch based on FD.io/VPP offering feature-rich, high-performance cloud native networking and services.

It deploys itself as a set of system pods, three of them installed on Kubernetes master node only and the rest of them created on all nodes (including the master node), in a very complex architecture as the image 3.4 shows.

Contiv vSwitchvSwitch is the main networking component that provides

the connectivity to Pods. It deploys on each node in the cluster and consists of two main components packed into a single container: VPP and Contiv VPP Agent.

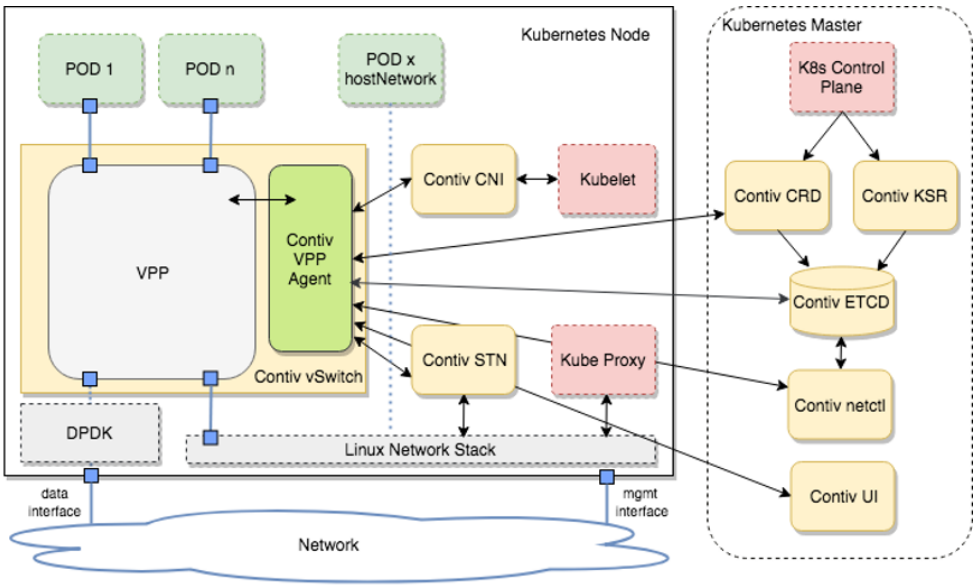

Figure 3.4. The complex architecture installed by Contiv-VPP CNI[12]

The main component is VPP. It provides Pod-to-Pod connectivity across the nodes in the cluster, as well as host-to-Pod and outside-to-Pod connectivity. While doing that, it leverages VPP's fast data processing that runs completely in userspace and uses DPDK for fast access to the network IO layer. Contiv VPP Agent, instead, is responsible for configuring the VPP according to the information gained from ETCD and requests from Contiv STN.

The components, executed in the master, have multiple objectives: first of all, Contiv VPP uses its own etcd (Contiv ETCD) instead of using the default one, and is kept updated by two other components (Contiv KSR and Contiv CRD) which respectively save information on the K8s resources and on the custom ones installed in the cluster.

In order to allow inter-node communication between PODs each node

should have one "main" VPP interface, which is unbound from the host network stack and bound to VPP. Contiv/VPP control plane automatically configures the interface either via DHCP, or with statically assigned address (if only one interface is available in the node, the Contiv STN (Steal The Nic) pod is useful and allow to use this interface as if they were two separate).

Through a series of created routing tables, the packet is correctly forwarded to the destination, whether it is local (therefore intra-node communication) or remote (therefore inter-node) after encapsulation. In fact, in order to provide inter-node connectivity via any underlay network, Contiv/VPP sets up a VXLAN tunnel overlay between each 2 nodes within the cluster (full mesh). All traffic is, in each node, checked and, if necessary, filtered if communication between the two pods cannot take place (ACLs controls).

Contiv VPP provides its own implementation of IPAM, assigning to each node both an ID and, from this ID, two different subnetwork (one for the pods created and one to well configure VPP with the node's Linux network stack).

Regarding IP translation when services are deployed, the most times load-balancing and translations between services and endpoints are done inside VPP using the high performance VPP-NAT plugin, instead of using the Linux network stack (i.e. to kube-proxy). The same happens, also, for network policy resources.

3.2.5 Cilium

Cilium exploits BPF, a Linux kernel technology, which enables the dynamic insertion of powerful security visibility and control logic within Linux itself. Because BPF runs inside the Linux kernel, Cilium security policies can be applied and updated without any changes to the application code or container configuration.

When Cilium is installed in the cluster, it creates several components on each Linux node: the most important is the Cilium Agent (deployed as a

daemon). It interacts with Kubernetes via Plugins to set up networking and security for containers running on the local server, listening to events to learn when containers are started or stopped. It creates and compiles custom BPF programs (highly efficient) which the Linux kernel uses to control all network access and all packets in/out of the container's virtual ethernet device(s), provides an API for configuring network security policies, gathers metadata about each new container that is created and interacts with the container platforms network plugin to perform IP address management (IPAM).

Cilium, in addition to the IPAM plugin provided by K8s, allows you to define for each node, through the use of a CRD, a list of addresses to be assigned to the pods assigned to that node: the list is kept updated automatically every time a pod is created or deleted.

Like Calico, Cilium also supports two different networking modes: both encapsulation with overlay (this is the default mode) and direct routing. In overlay mode, VXLAN and Geneve are baked in but all encapsulation formats supported by Linux can be enabled. In this mode, all cluster nodes form a mesh of tunnels using the UDP based encapsulation protocols. All container-to-container network traffic is routed through these tunnels.

As the image 3.5 shows, the encapsulation is done by cilium_vxlan process.

In direct routing mode, Cilium will forward all packets which are not addressed for another local endpoint to the routing subsystem of the Linux kernel. This means that the packet will be routed as if a local process would have emitted the packet.

Kubernetes is responsible for distributing the policies across all nodes and Cilium will automatically apply the policies. The plugin introduces a new kind of Network Policy, called CiliumNetworkPolicy, which is available as a CustomResourceDefinition and supports specification of policies at Layers 3-7 for both ingress and egress.

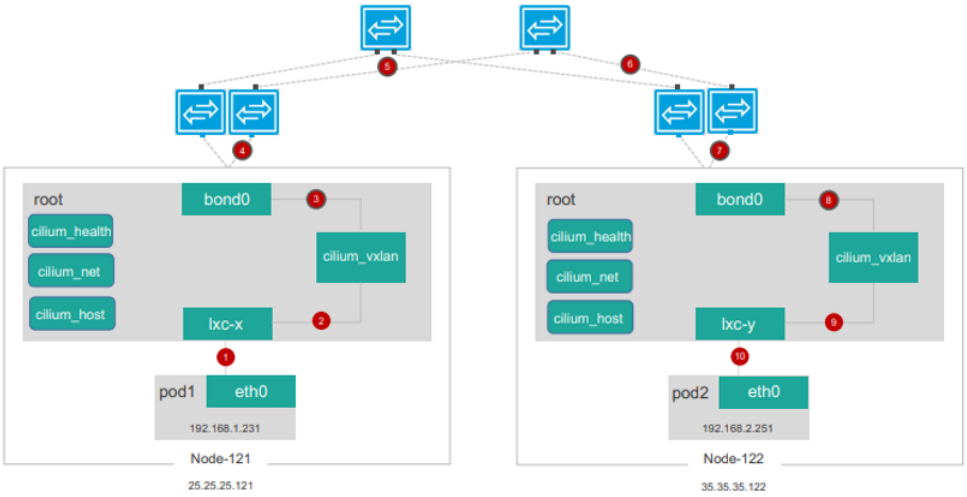

Figure 3.5. Packet Flow in Cilium, in a inter-node communication[10]

3.2.6 OVN-Kubernetes

Ovn-kubernetes is nothing more than an integration for Kubernetes of the Open Virtual Network (OVN) technology, an Open vSwitch-based software-defined networking (SDN) solution to support network abstraction[13].

The documentation of this CNI turns out to be confusing and not very thorough on the official repository, so the theoretical explanation is not very thorough.

This CNI therefore, like other solutions, is based on OVS, creating an OVS bridge on each node of the cluster.

In the "overlay" mode, OVN programs the Open vSwitch instances running inside hosts, creating a logical network amongst containers running in the cluster.

This plugin, once installed in the cluster, in turn creates various components, some on all nodes, while others only on the master (if several cluster's nodes are masters, these may be replicated on various masters). The "central" components, those installed on the master, are components that store

the user's network intent in databases (North and South Database), typical of the OVN solution. In addition, a northd daemon is also installed that translates networking intent from k8s stored in the OVN_Northbound database to logical flows in OVN_Southbound database.

In each node of the cluster, on the other hand, other components (ovs-daemons and ovn-controller) are installed that allow both the correct installation and operation of the OVS bridge (which, therefore, must interact with the master for configuration) and the forwarding of packets to the designated destination, either an intra-node or inter-node communication.

Regarding networking, as the image 3.6 illustrates, this solution creates a simple network topology with one logical switch created per k8s node and then inter-connect all those logical switches with a single logical router. Any containers created in these nodes get connected to the logical switch associated with that node as a logical port.

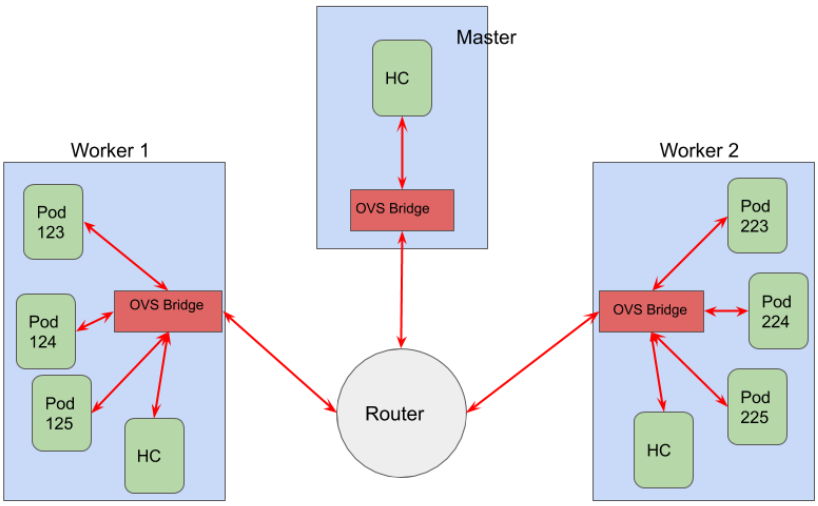

Figure 3.6. OVN-Kubernetes architecture

3.2.7 Kube-Router

Kube-Router is a CNI solution, which seeks to provide high performance without increasing complexity. In fact, compared to other plugins, it installs a single component (the kube-router daemon) to do the job, where others might install many.

It also uses different technologies and configuration to increase performance like IPVS/LVS for service proxy and direct routing between the nodes.

Kube-router is built around the concept of watchers and controllers [14]. Watchers use Kubernetes watch API to get notification on events related to create, update, delete of Kubernetes objects (like nodes, pods, endpoints, services and so on). On receiving an event from API server, watcher broadcasts events to the controller that registers to get event updates and act up on the events. Three controllers are deployed which, respectively, check service resource, network policy resource and networking routes (in this case it advertises the routes to the rest of the nodes, BGP peers, in the cluster).

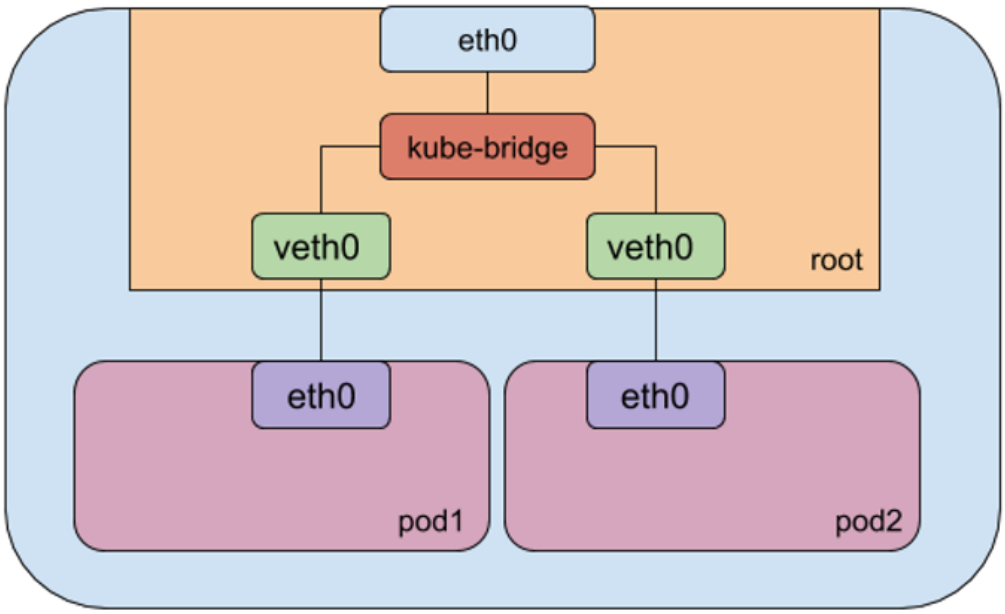

Figure 3.7. Pods in Kube-Router are connected to the network using a simple bridge[]

Kube-router chooses to implement a solution based on L3 routing solution. On the data path, inter Node Pod-to-Pod communication is done by the Node's routing stack. In fact, each kube-router instance on the Node acts as a BGP router and advertises the Pod CIDR assigned to the Node (forming full mesh in the cluster). Learned routes are injected into the local Node's routing table. In this way, a packet will be routed directly through the node's routing tables.

To connect containers to the network, this network provider uses the external CNI specification, not its own implementation. In fact, by default it uses CNI bridge (the simplest CNI that creates a simple bridge to which all the pods of the node are connected through a veth pair, as the picture 3.7 shows), but it is possible to change this choice via a parameter before installing the plugin.

Kube-Router, to improve performance, has decided not to manage the services through the creation and insertion of ip tables rules by the kube-proxy, but to use a different technology: IPVS. It is known to be fast than other solution as its implemented in Kernel, providing different IP load balancing techniques (NAT, TUN and LVS/DR) and supporting rich set of connection scheduling algorithms (Round-Robin, Weighted Round-Robin, Least-Connection etc.). All these features, even if they increase the complexity, improve performance managing speeder the IP address translation required when traffic is sent to services, while maintaining good performance by growing the number of services in the cluster (which is not the case in the other case).

For each network policy installed in the cluster, however, Kube-Router creates and injects a new ip table rule, a behavior common to many CNIs.

3.2.8 Polycube-K8s

This CNI is developed by the Computer Networks Group of Politecnico di Torino. It leverages eBPF and some polycube services to provide network

support for pods running in Kubernetes[15].

When installed in the cluster, it deploys four components on each node:

1. The pcn-k8s-agent interfaces with the Kubernetes API master service and dynamically reconfigures the networking components in order to create the network environment required by Kubernetes and provide connectivity to the pods.

2. The pcn-cni plugin connects new pods to the pcn-k8s networking.

3. pcn-k8switch forwards packets between pods and provides support for ClusterIP and NodePort services, while, pcn-k8sfilter is a small service that is attached to the physical interface of the nodes and performs a filtering on the incoming packets, if those packets are directed to NodePort services they are sent to the pcn-k8switch, otherwise packets continue their journey to the Linux networking stack.

Like Cilium and Calico, pcn-K8s for networking allows the user to choose the option they prefer between direct routing and encapsulation (this is the default option).

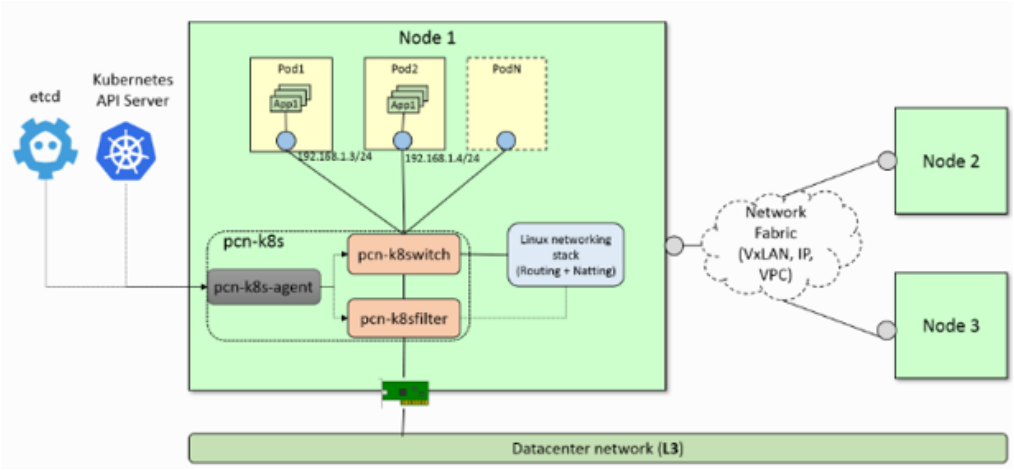

Figure 3.8. pcn-k8s node's components[16]

For Pods communication in the same host only the k8switch is exploited. This component (that works also as a router) forwards packets between pods, working at level 3. When a pod1 in the host A sends a packet to pod2 in the host B, the k8switch understands that the destination is not local and builds the encapsulation. The new packet is then forwarded to the Linux networking stack which is in charge of sending correctly the packet to destination, leveraging native routing tables and the Datacenter network fabric. When it arrives at the destination host's Linux networking stack and after the decapsulation, the inner packet (packet created by pod1) is forwarded from the Linux networking stack to the pcn-k8switch which sends the packet to the destination pod.

To correctly work with services, the lbrp (load balancer reverse proxy) is leveraged, executed in the pcn-k8switch. It recognizes the virtual IP (ClusterIP) address of the service, changes it with an endpoint's IP address of the selected service and the packet is sent regularly. The opposite translation is done when the response packet arrives to the k8switch: the Pod IP address (written in the IP source field) is translated in the previous ClusterIP address.

Regarding security, pcn-k8s CNI implements both standard kubernetes networking policy and advanced Polycube networking policies. The latter provide a more flexible, simpler and more advanced approach to filter the traffic that a pod should allow or block. They include all the features from Kubernetes Network Policies and present some additional features, with the goal to also encourage operators to write more effective policies.

3.2.9 WeaveNet

WeaveNet is another very popular and widely used CNI in clusters, although architecturally it is quite different from other network providers. It introduces in its network the concept of peer not present in the other plugins. In fact, a Weave network consists of a number of 'peers' - Weave Net routers

residing on different hosts. These peers communicate their knowledge of the topology (and changes to it) to others, so that all peers learn about the entire topology.

In this way, it creates a virtual network that connects containers across multiple hosts and enables their automatic discovery[16].

It creates a new Layer 2 network using Linux kernel features, deploying one daemon (called weave), that sets up the network and manages routing between machines (it is present also a CNI process to attach container to this network), and another one (called weave-npc) for Kubernetes Network Policy.

The plugin creates a network bridge (OVS Bridge) on the host. Each container is connected to that bridge via a veth pair, the container side of which is given an IP address and netmask supplied either by the user or by Weave Net's IP address allocator.

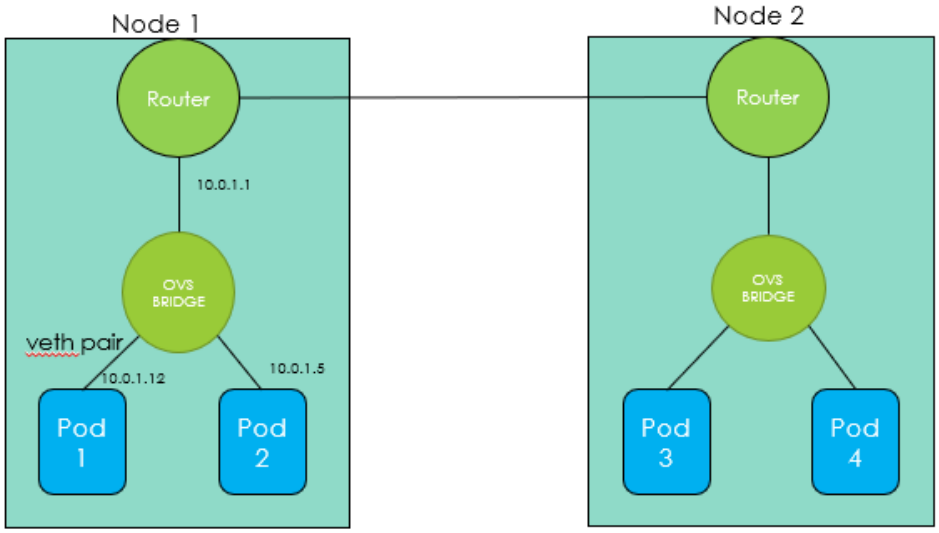

Figure 3.9. Cluster configuration with WeaveNet[16]

The network provider relies on a routing component installed on each host in the network. These routers then exchange topology information to maintain an up-to-date view of the available network landscape. When looking

to send traffic to a pod located on a different node, the weave router makes an automatic decision whether to send it via "fast datapath" or to fall back on the "sleeve" packet forwarding method.

The fast data path operates entirely in kernel space and it does not exploit routers to forward packets. Instead in the fallback sleeve method packets destined for non-local containers are captured by the kernel and processed by the Weave Net router in user space. Then, are forwarded over UDP to weave router peers running on other hosts, and injected back into the kernel which forwards them to local pods. Weave Net routers learn which peer host a particular MAC address resides on. They combine this knowledge with topology information in order to make routing decisions and thus avoid forwarding every packet to every peer.

WeaveNet is one of the CNI that allows direct encryption by setting specific parameters. It uses known solutions (e.g. Diffie-Hellman algorithm using the Go version of Bernstein's NaCl library) without adding much complexity (so it doesn't use complicated certificates or trading algorithms).

3.2.10 Amazon CNI

This network provider is presented briefly, as it cannot be used outside the Amazon Virtual Private Cluster(VPC) provided by Amazon Elastic Compute Cloud (Amazon EC2). However, it remains interesting to present it and, if possible, compare it even in a shallow way with the other CNIs. Obviously it is based on a series of products and technologies (for example Elastic Network Interfaces) introduced by the company that would require very in-depth studies and not just hints.

The network provider uses Elastic Network Interfaces on AWS and consists of two parts:

1. The CNI plugin: it is responsible for wiring the host network (for example, configuring the interfaces and virtual Ethernet pairs) and adding

the correct interface to the pod namespace.

2. The L-IPAM daemon (IPAMd): it is responsible for attaching elastic network interfaces to instances (nodes in EKS are called instance), assigning secondary IP addresses to elastic network interfaces, and maintaining a "warm pool" of IP addresses on each node for assignment to Kubernetes pods when they are scheduled.

So unlike many other CNIs, the default IPAM created by K8s is not used, but a specific one is executed (created specifically for this solution).

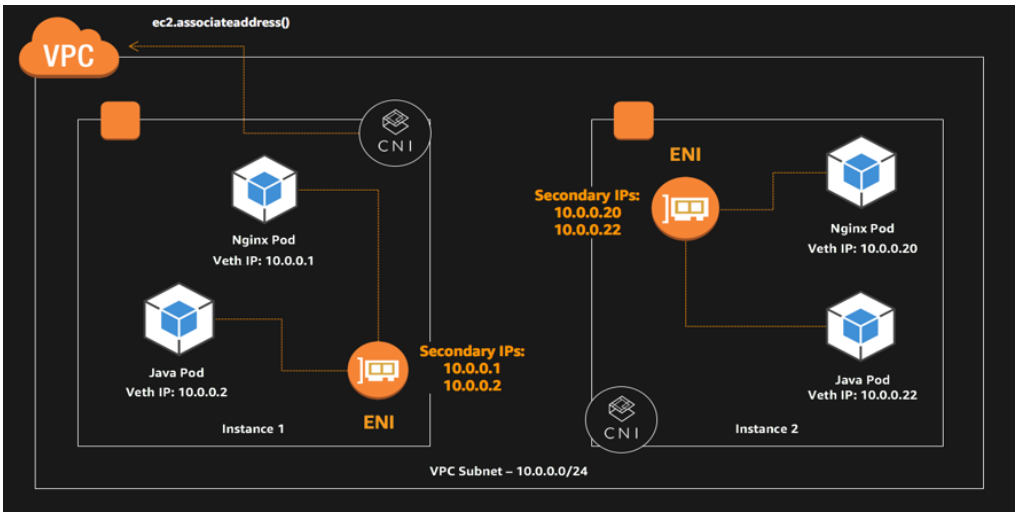

Figure 3.10. Amazon Cluster configuration: It is possible to see that each ENI has IP address of pods scheduled in its instance [17]

The predominant role in the networking, configured by this CNI, is played by the Elastic Network Interfaces (ENI) created on each node, as the image 3.10 shows. It represents a virtual network adapter and can have several network attributes, the most important of which are: a primary private IPv4 address, one or more secondary private IPv4 addresses (dedicated to pods), and an Elastic IP address (an IPv4 address public, which can be reached from the Internet). All traffic to and from the node is processed by this interface.

As for the network policies, this CNI does not offer an implementation but suggests to use in combination an additional CNI that deals with that type of resource (Calico or Cilium and so on), leaving the only task of networking to Amazon CNI.

3.3 Tested CNIs

Starting from the list of CNIs analysed in the previous section, a subset of network plugins was identified to run the developed benchmarking tool. Some of these CNIs have been monitored in multiple networking configurations: Calico and Polycube, in fact, are tested both in native routing and in VXLAN mode. In general, all the CNIs tested did not undergo any particular configuration changes (except to assign them the correct IP address space) compared to the default configuration.

Amazon CNI and Contiv VPP were not carried out for two different reasons: the Amazon CNI, while being open source, can only be used in Amazon's cloud infrastructure; for what concerns Contiv VPP, it was not possible to install it in the test environment due to the restrictions for accessing the cluster remotely.

The network providers tested are:

- Calico (Native Routing)
- Calico Vxlan
- Polycube (Native Routing)
- Polycube Vxlan
- Cilium
- OVN-Kubernetes
- Cilium

- Antrea
- WeaveNet
- Flannel

Chapter 4

Comparison between CNIs: state of art

In this chapter are presented similar works present in literature and where they differ with the job done during the thesis.

4.1 Benchmark results of Kubernetes network plugins (CNI) over 10Gbit/s network

The first is the work done by Alexis Ducastel whose goal is to find the best CNI in a pod to pod communication scenario with various protocols (TCP, UDP, FFTP, SCP).

In an article published on ITNext[18], he, first of all, analyzes the plugins chosen quickly from a theoretical point of view, focusing on some theoretical aspects (which CNIs support network policies, which allow encryption and so on) and then moves on to the presentation of the results obtained in its tests.

"The cluster is composed by three Supermicro bare-metal servers, where Kubernetes 1.14.0 is set up on Ubuntu 18.04 LTS, running Docker 18.09.2, (master deployed on the first node, server on the second node and client on the third) connected through a Supermicro 10Gbit switch. The servers are

directly connected to the switch via DAC SFP+ passive cables and are set up in the same VLAN with jumbo frames activated (MTU 9000)"[18].

All performance measurements was taken at least three times, but it is not clear how long time each measurement have taken, on six different CNIs (Calico, Cilium both in normal and encrypted configuration, Canal, Flannel, Kube-router and WeaveNet both, again, in normal and encrypted configuration).

The tests, written in bash, perform the measurement three times using different tools, averaging the three results obtained and returning it to the user. In this case only the pod to pod communication is tested and the results obtained in the case of a connection at 10 Gbits/s are very similar, as the image 4.1 shows. In fact, almost all CNIs saturate the network and there are no big differences between plugins (except in the case of CNI with encryption enabled, where time is lost for cipher and decipher).

In recent months, the article has been updated by adding two further tests (communication via a service with UDP and TCP also comes into play) and some CNIs not present in the past (Antrea and OVN-K8s) and by running only tests with UDP and TCP protocol (no more HTTP, FTP and SCP).

Surely the continuous updating and free access to the code represent a positive aspect of the work just described, while the few measurements made for the calculation of the throughput, the lack of clarity in some parts of the article (for example Calico in Vxlan or native mode?) and the few scenarios tested represent the improvement points of this work.

Unlike this work, the tests created in the thesis check the CNIs even in less than ideal configurations and in addition to the throughput they focus on the amount of CPU used and latency (utilising multiple tools). In addition, the number of measurements is different in the two benchmarkings: if in the test set just presented the number of measurements is three, this has been deemed too low and has been increased to twelve in the benchmarking that will be presented in the next chapters.

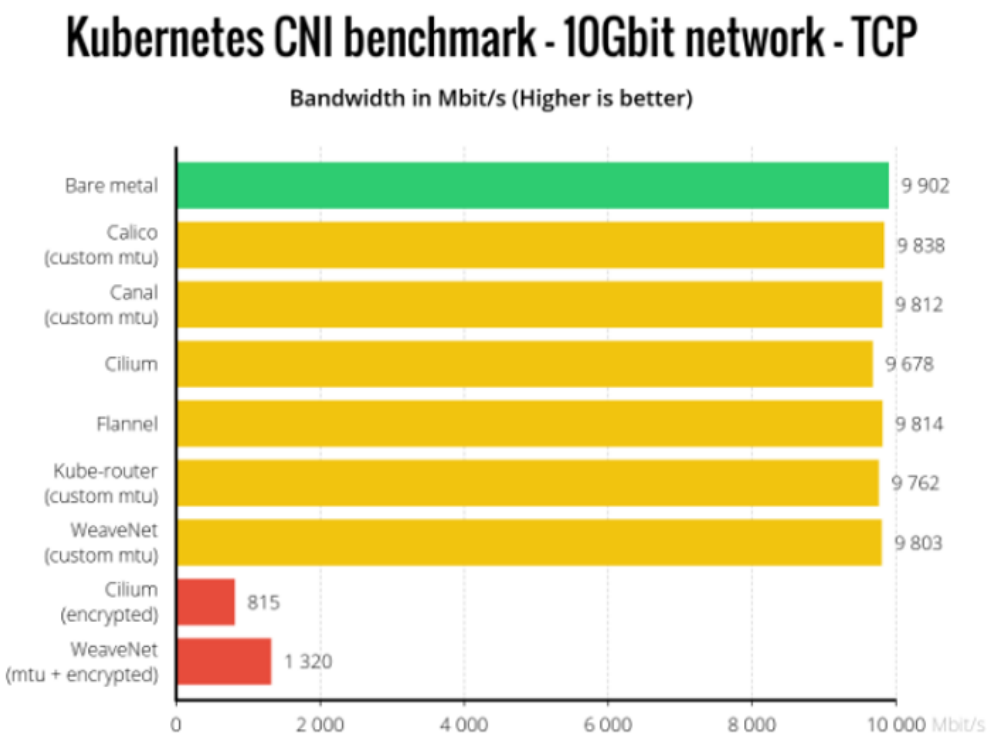

Figure 4.1. Ducastel TCP Results

4.2 Overview of Kubernetes CNI plugins performance

The second work presented is a university article[19] published in November 2019 by Narūnas Kapočius.

The Lithuanian researcher focuses his studies on showing how much the throughput of various CNIs is affected by running Kubernetes on virtual machines (therefore in a virtual environment) compared to installing the framework directly on the bare metal server.

So the tests he wrote and the results obtained by the latter tend to show the deterioration in performance when the various plugins are installed in a virtual environment, compared to the bare metal environment.

Test environment was adapted to measure Kubernetes CNI plugins performance in virtualized and physical infrastructure which results were compared to physical infrastructure baseline performance results.

It builds 3 different scenarios:

1. Kubernetes cluster in physical environment (K8s running directly on the physical OS).

2. Kubernetes cluster in virtualized environment (VMware ESXi type 1 hypervisor was used where VMware vSphere tool set was used for provisioning virtual machines).

3. two bare metal directly connected for baseline results (only plain OS with tools needed to complete measurements were installed).

In all three test cases servers were running CentOS 7.6 OS. Kubernetes clusters were created using Docker 18.09.1 and Kubernetes 1.14.1.

Kubernetes cluster consisted of 3 nodes – 2 worker nodes and one master node. During each measurement, taken 3 times a day for one week, Kubernetes master node was set up in a different datacenter.

He tests nine CNIs in these scenarios (Canal, Calico, Romana, Kopeio [1], Flannel, Cilium, KubeRouter, WeaveNet, TungsteinFabric[2]) and, in general, all plugins throughput is significantly less compared to bare metal results in both virtual and physical cluster.

In fact, first of all, he illustrates the results obtained with 1500 bytes messages (MTU = 1500 Bytes) in the two scenarios with Kubernetes active, and then shows in a graph how much worses each CNI in the virtual environment, for the TCP protocol and for HTTP.

Beyond this, it reports through graphs the worsening occurred with jumbo frames (MTU = 9000 Bytes) both with TCP and HTTP.

[1]https://github.com/kopeio/networking

[2]https://tungsten.io/

For all CNI plugins, the TCP throughput deteriorates in the virtual environment, in 1500 MTU test case, as the picture 4.2 illustrates, throughput decreased significantly more compared to 9000B MTU test case.

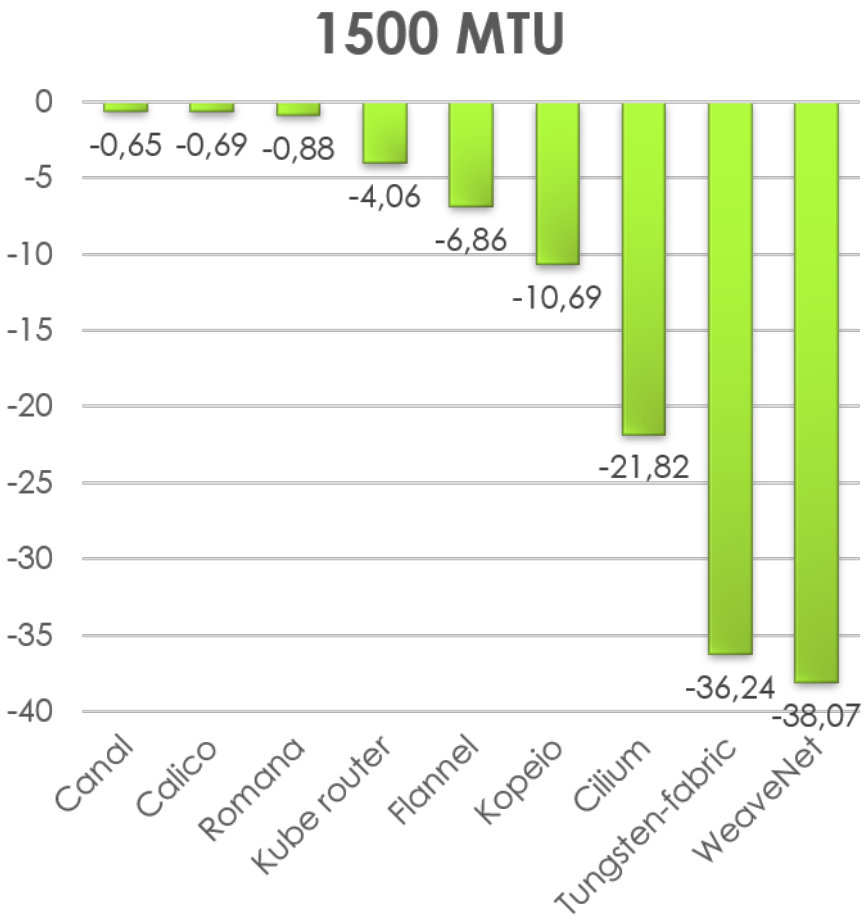

Figure 4.2. Performance decrease in K8s/Vm/Bare Metal (TCP), graphs taken from the article

In general, this article compares the plugins, but without creating a benchmarking tool since its goal is different: to see how much the throughput decreases when Kubernetes is run on virtual machines running on servers, compared to the direct installation on bare metal.

If compared to the previous work, the general configuration information is reported in more detail, on the other hand the tests used and also part of

the results (only the differences in percentage are reported) remain unusable and hidden to the reader.

The difference with the thesis developed lies in the objective: although there is a comparison between the CNIs and the different behaviors are highlighted, in this case the interest is focused on the differences between physical and virtual environment.

4.3 Tests in Kubernetes repository

In the Kubernetes repository present on github [20], there are tests that go to calculate the throughput, in various scenarios (very similar to those used in this tool and presented in the next chapter). Here too, the same two tools (netperf and iperf) are used to calculate the behavior of the plugin in pod to pod inter-node and intra-node communications and in communications via service, but with some differences.

In these tests, the MTU of the messages is statically changed several times and therefore the results returned to the user give information on the variation of the throughput as the size of the packets sent increases.

On the other hand, however, each measurement is made only once as good without carrying out more accurate checks (even just repeating the measurement a few times). In addition, the tests written with netperf when the service resource comes into play present an error in the commands given to the tool, which does not allow the correct measurement of the throughput (the result is always zero).

However, these tests represent a good basis for calculating the performance of the CNI installed in the cluster, without however returning useful information on the amount of CPU used by the two nodes, or other network characteristics such as latency.

Chapter 5

CNI benchmarking suite

This chapter represents the main core of the thesis. First of all, the network features monitored by the benchmarking suite are presented. Then, the whole suite is explained to the reader from both an architectural and implementation point of view (which scenarios are used in the tests and how they are implemented), with a quick presentation, in the middle section, of the network tools used. In general, this suite stresses the CNI behaviour in several ideal and real scenarios, highlighting how the implementation choices affect the network performance.

5.1 Key Performance Indicators

The CNI benchmarking tool measures a set of Key Performance Indicators (KPIs), chosen to analyze how the plugins behave in several test scenarios, and to understand the differences between the selected CNIs. The chosen KPIs are the most used benchmarks to evaluate the performance of a network infrastructure:

- Throughput: it "measures how many packets per unit of time arrive at their destinations successfully. Throughput is usually measured in bits per second, but it can also be expressed in data (packets ndr) per second."[24]

- Node's CPU usage in percentage: the CPU used (in percentage) on each worker to reach the measured throughput.

- Latency: in this benchmarking suite, the latency is measured as the time that elapses between opening the connection and receiving the first response byte, called also time to first byte (TTFB).

The throughput, in the tests proposed in this work, is computed both with UDP protocol and TCP protocol, binding the tool used to a specific worker's core both client side and server side, while the latency only in the TCP case.

Using them to inspect the functioning of the CNI is interesting as it allows both to see the effects of the implementation choices at the network level effectuated by the network provider (who uses encapsulation and who instead uses direct routing) and how the CNI acts in front of a series of K8s resources that affect the network.

What are these resources? And how do they affect the cluster networking and performance?

Services and Network Policies are K8s resources widely used, since they add important features to communication as, respectively, reliability and security. In fact, the former, as said before in the section 2.2.2, allows pods to avoid both check on pods' IP address fairness and communication with not existent pods. The latter, again described in the previous section 2.2.3, allows construction of security rules in communications between pods.

Their implementation (leveraging creation of ip table rules or eBPF programs and so on), mostly when the number of these resources deployed is high, has no-negligible impact on the networking and influences performance.

Each CNI presented before has its implementation of services and the tool, proposed in this thesis, tries to bring out, in several scenarios with services, the CNI behaviour and its influence on network features with a series of related but different tests.

Network policies, also, affect the network, binding communications (as

already mentioned, only some are allowed). The CNI has the task of implementing, as they see fit, this resource and the proposed solutions, as already shown in chapter 3, are different. This benchmarking tool verifies, with an appropriate scenario, how the CNI implementation influences the networking and the throughput, to give to users some information about CNI installed and its security implementation.

5.2 Test Suite Structure

After a presentation of Key Performance Indicators monitored and K8s resources involved, in this section the test suite will be exposed from an architectural point of view.

Each proposed scenario tries to reproduce a possible situation, both ideal and real, findable in a cluster to observe how the CNI behaves. This structure has been chosen to have not only results on ideal situations, hard to find everyday, but in common and detectable cases in almost every cluster.

It is possible to divide tests proposed in two main groups:

- Pod to Pod communication (PTP Communication)
- Pod to Service to Pod communication (PSP Communication)

To make the explanation easier and clearer, the two pods (client side and server side) responsible for making the measurements will be called Clientpod and Serverpod respectively, to distinguish them from other standard and unused pods that can be created in the cluster.

5.2.1 Pod to Pod Scenarios

The first aspect that has been chosen to monitor to highlight the difference in behavior between CNIs is the Pod to Pod communication. This has been observed with the cluster in two very different configurations: empty, apart the two pods, and various pods and network policies installed.

In this way the pod to pod communication is tested both in the ideal condition, when no one can disturb the exchanged traffic and also in a situation common to many clusters, when some restrictions between pods are present.

In addition to the aspects just described, another configuration was examined. Generally the K8s scheduler assigns pods to nodes in a reasonable way (e.g. it does not distribute pods to nodes with not available resources and distributes them across all nodes) and consequently most communications are inter-node (for example sender pod in node1 and receiver pod in node2), but sometimes, especially in small clusters, it is very common that there is communication between two pods created on the same node.

In this test suite both scenarios are covered, for each kind of test, starting from the easiest to the most complicated (in each scenario they are presented together even if they represent two sub-scenarios distinct).

Simplest PTP Communication

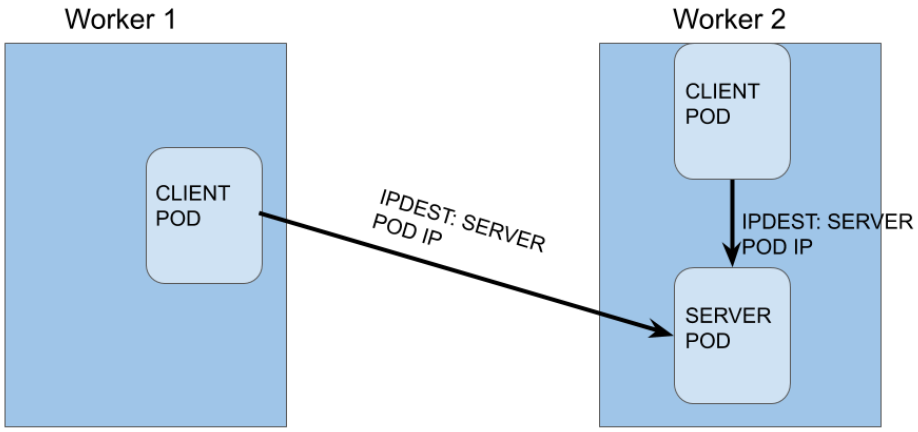

Figure 5.1. Representation of Pod to Pod communication

The first scenario presented examines throughput and CPU usage when

pods are deployed both on different worker and on the same, as it can see in the picture 5.1.

In the Clientpod the container executes the client side of the tool chosen, while in the other hand the container in the Serverpod the server side. This is an ideal cluster scheme, where only two pods are presents, difficult to find in reality.

PTP Communication with other Pods and Network policies

The previous scenario (and its two sub-scenarios), as said before, are a corner case/ borderline case, being created only two pods. Normally, there are a lot of pods deployed in different nodes, maybe, with network policies installed and all these objects can influence the communication between above previous pods and the cluster networking in general.

The second scenario created tries to evaluate CNIs behaviour in daily situations, when several network policies are installed for different pods. This to observe how much influence has the network policies solution adopted by a CNI on the network throughput and the workers' CPU.

The two sub-scenarios introduced in the previous scenario are present here, as the picture shows 5.2.

As can also be seen from this image, the differences between this scenario and the previous one are in the number of object created in the cluster. In this second case, there are several pods in the cluster and a huge number of network policies are installed.

This last wants to show in a clearly way how much the network policies and their CNIs implementation influence the throughput in a pod to pod communication, being able to increase a priori the number of policies and pod installed in the cluster (in order to notice the difference in behavior as the installed resources increase).

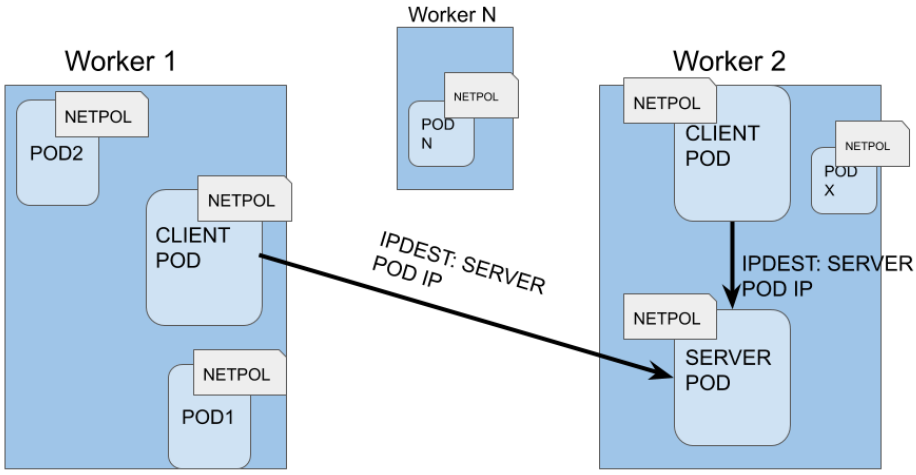

Figure 5.2. Representation of Pod to Pod communication with several pods and network policies installed

5.2.2 Pod to Service to Pod Scenarios

The services are the second resources that have a non-negligible impact on the cluster networking. In this kind of communication the ClientPod contacts the service that redirects the traffic to correct ServerPod.

Like for the network policies, scenarios are proposed that could highlight the different implementations offered by the CNI, always starting from an ideal case (or almost) to arrive at more complex but everyday scenarios.

PSP communication with a service

This is, once again, an ideal situation, difficult to find in reality, in which there is only one service and two pods distributed in the cluster.

The things said above for inter-node communication and intra-node communication are identical here: the ClientPod that contacts the ServerPod through the service is deployed one time in different node, while the second time in the same node of the ServerPod.

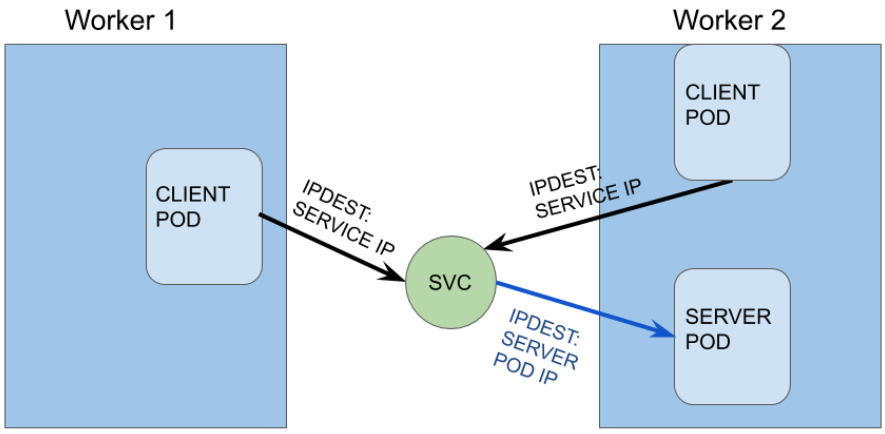

Figure 5.3. Representation of Pod to Service to Pod communication with one service

The only difference introduced between the situation described in the picture 5.1 and that one in 5.3 is the presence of service that forwards the client traffic to the correct backend pod, influencing the throughput and CPU usage based on how the the traffic redirection (IP translation) is done by the plugin .

PSP communication with a growing number of services

To simulate a situation common to many clusters, this structure can have several sub-scenario, reported in a schematic way in the picture 5.4. As the reader can see, normally in the cluster various services and pods can exist in addition to the previous ones. These scenarios are interesting to detect if and how the CNI plugins scale as the number of services increases.

In these cases, the work done by the kube-proxy and more general the solution proposed by each CNI could have a non-negligible impact and for these reason scenarios are present in the test suite.

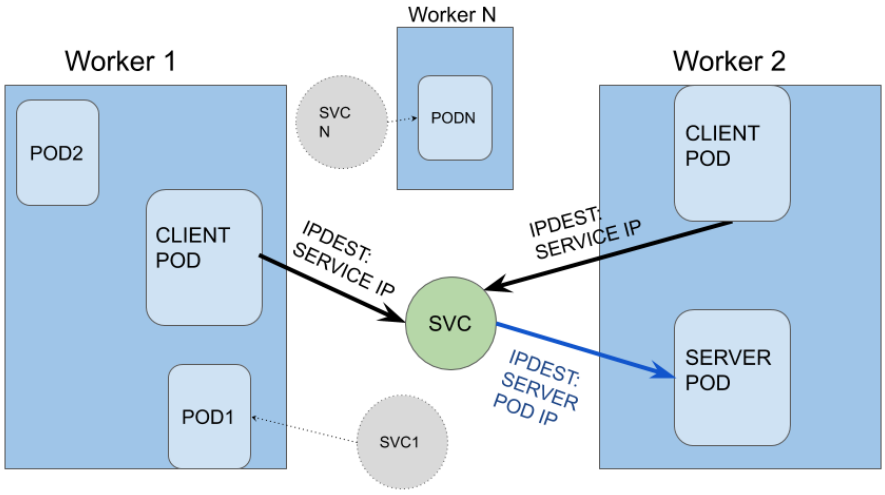

Figure 5.4. Representation of Pod to Service to Pod communication with several service

PSP communication with growing number of services and server replica

In the pod-to-service-to-pod communication just explained, there is a simplification: the service exposes an application (the application tool for calculating the throughput for example) running only on a single Serverpod. Many times, however, there are several pods behind the service: hence a further scenario has been created to evaluate the CNIs behaviour. This situation checks how CNIs implement the IP translation from serviceIP to one of possible podIP (not only one) and computes another network features never analysed before: the latency.

By latency, as already mentioned above, is computed the time between the opening of the connection by the client and the reception of the first response byte from the server, called also Time to First Byte (TTFB).

Also in this scenario, the test can be performed with an increasing number of services in the cluster, but in addition to the previous scheme, it is now

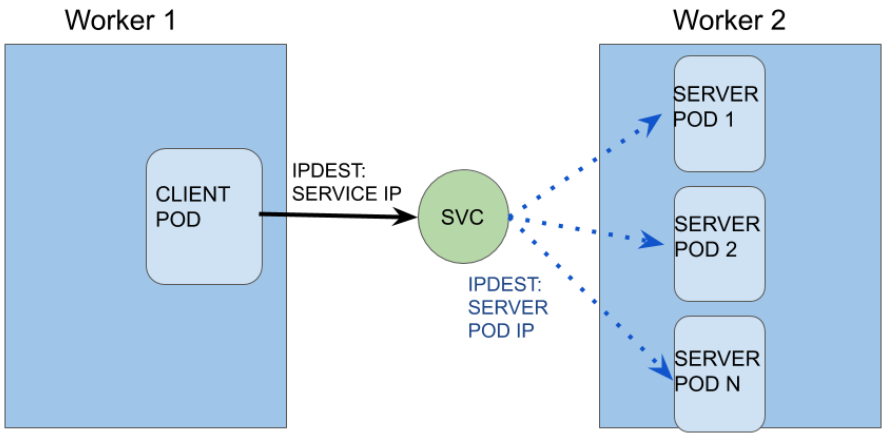

Figure 5.5. Representation of Pod to Service to Pod communication with n Serverpod Replica

possible to set the number of ServerPods in advance. In the other hand in this case, only the pod to pod inter-node communication through service is examined. The intra-node communication is skipped to avoid the creation of an huge number of pods (being a free parameter the number of replica) on the same node that could cause damage to the cluster.

5.3 Tools used

In this section the tools used to achieve measurations are briefly presented with also a parameter list to know to well understand test implementations exposed in the next section.

5.3.1 Iperf3

Iperf, sometimes referred as iperf3, "is a tool for active measurements of the maximum achievable bandwidth on IP networks"[21]. Redesigned from scratch, it separates from previous versions, presented below, adding some

features taken from other tools such as netperf and nuttcp that were not present in the basic version.

The main parameters to know to understand tests are:

- -s: to start the server side of the tool.
- -c IPServer: to start the client side of the tool followed with the server address IP to contact.
- -p: the port used by the tool
- -M: set the TCP Maximum Segment Size (MSS)
- -N: TCP no delay used
- -Z: "Zero Copy" method for sending data (this method consumes less CPU).
- -A coreClient,coreServer: binding the traffic to specified core in client side and server side.
- -u: set UDP protocol rather than TCP.
- -b 0: generally -b "Set target bandwidth to n bits/sec"[21], -b 0 sets unlimited target bandwidth for UDP (the default is 1 Mbits/sec).

5.3.2 Iperf2

Iperf2 is a relative of the previous tool (as the name suggests), but not the previous version, being iperf3 rebuilt from scratch. Therefore, it has a different parameters and behaviour (the first is single thread, while this is multi-thread). In the CNI benchmarking suite, iperf2 is used to compute throughput in the PSP scenarios, and the "-b" has a different configuration (the value 0 does not mean "unlimited bandwidth" and a right value must be chosen).

5.3.3 Netperf

"Netperf is a benchmark that can be use to measure various aspect of networking performance. The primary foci are bulk (aka unidirectional) data transfer and request/response performance using either TCP or UDP.."[22].

It is used to provide an alternative tool to Iperf3 to the user to evaluate the performance of the CNI in the cluster.

As for iperf3, the fundamental parameters used in the benchmarking suite during the measurements will be reported:

- netserver: to start the server side of the tool.
- -H IPServer: to start the client side of the tool followed with the server address IP to contact.
- -p: the port used by the tool
- -D: TCP no delay used
- -Z: "Zero Copy" method for sending data (this method consumes less CPU).
- -T coreClient,coreServer: binding the traffic to specified core in client side and server side.
- -t UDP_STREAM: sets UDP protocol rather than TCP.
- -i maxIt, minIt: it sets the number of maximum and minimum iterations to find the measure in the confidential interval

Differently from iperf3, with this tool it is not possible to modify the Maximum Segment Size (MSS) of packet send in the network.

For both tools, the several communication created are bind to specific core of workers.

5.3.4 Curl

The last tool used in this benchmarking tool is curl. It is a famous command line tool to transfer data, used in many areas. In our case, curl it is used to find the speed of moving one file from one pod to another and Time to First Byte (TTFB).

5.4 Tests Implementation

In this section, implementation choices will be explained for each scenarios, underlining tools used with their parameters, the main variables used by

tests, the cluster set up and how are the K8s objects created. It is divided as follows: the first subsection presents briefly language used and how is realized the benchmarking suite from code point of view, the second explains common behaviours to all tests and next show particular single test configurations.

5.4.1 Benchmarking suite code structure

The benchmarking suite is written in go [1] programming language exploiting the client-go[2] library that allows to talk with the cluster.

The picture 5.6 shows the code organization of the benchmarking suite.

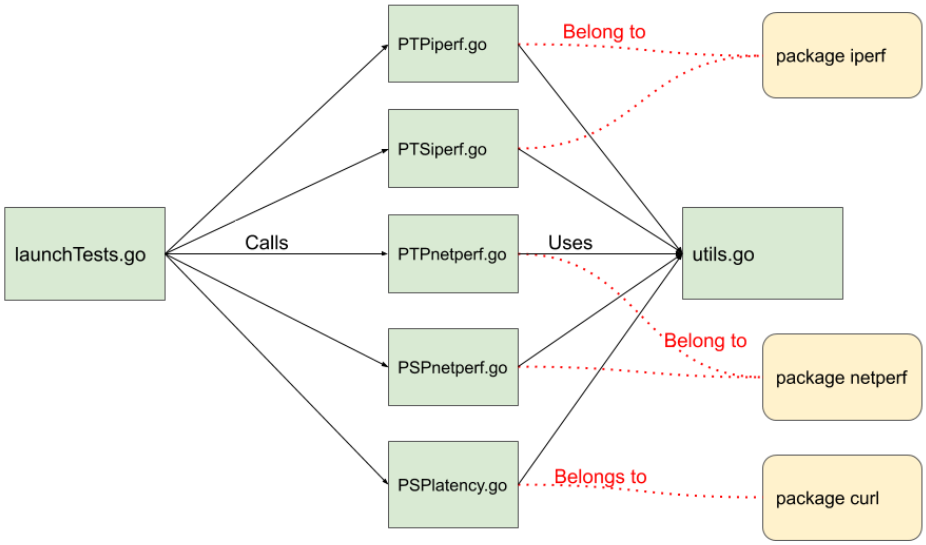

Figure 5.6. Code Structure

The "launchTests.go", which is the main class, calls all tests to cover scenarios described in the previous section and receives results from them. These results, as well as being shown on the screen, are written in a file that is saved in the host where the benchmarking is executed.

[1] https://golang.org/

[2] https://github.com/kubernetes/client-go

The tests called by this class are grouped into packages according to the tool used. So, in the "netperf" package are present the tests that covered the previous scenarios using the netperf tool and so on. In addition, a test is configured to represent a specific scenario, based on parameters passed to it: for example, the "iperfService.go", depending on a given value which indicates number of services in the cluster, brings back results for that specific cluster configuration.

The "utils" package, on the other hand, has a series of functions which, correctly configured through the parameters, are used by all the other tests (for example to delete the namespace or a Network Policies).

5.4.2 General implementation

Some general information regarding the whole benchmarking suite are reported here, before analyzing each scenario singularly.

The first aspect to highlight concerns the creation of pod in a specific node: in fact, in this test suite in some scenarios ClientPod and ServerPod must be deployed on different nodes, in other in the same. To cover both cases the NodeSelector field of the pod's specification is used, a field that permits to create a pod in a node that matches the NodeSelector.

In the picture 5.7, an example of NodeSelector is presented.

The pod1 is scheduled only on the node X due to the label set in the node that matches the field of nodeSelector present in the pod1 (the same of pod3). The pod2 that has not any label can be deployed on both server.

In all tests a boolean flag, based on its value and passed as a parameter, allows to choose which of two scenarios (inter-node communication or intra-node communication) set.

Another detail to point out is the creation of ClientPod and ServerPod. In the tests, that will be described, the Clientpod, which acts as a client for the specific tool (for example iperf3), is generated by a K8s job resource and, based on the application it hosts, executes the tool one or more times

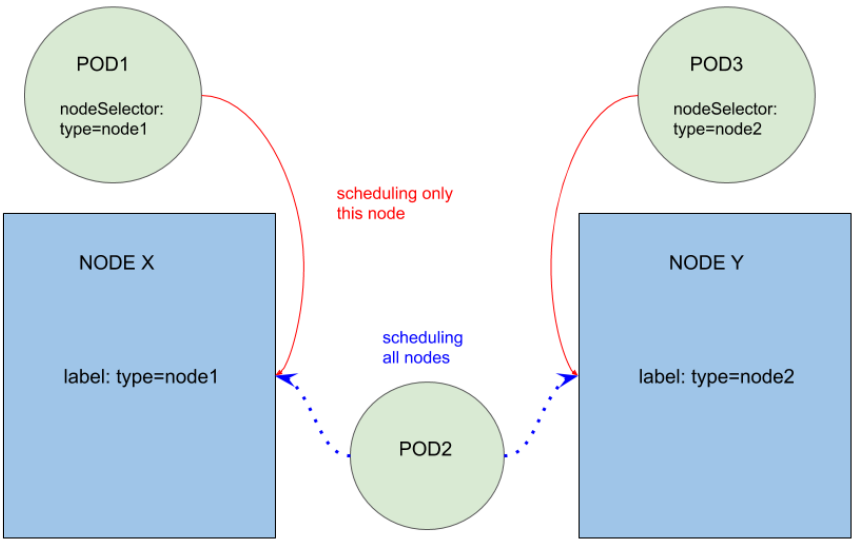

Figure 5.7. NodeSelector example

to compute measures.

But what is a job in K8s? "A Job creates one or more Pods and ensures that a specified number of them successfully terminate. As pods successfully complete, the Job tracks the successful completions."[23]. When the status of pod is "Completed", its logs are read to obtain measures done. Instead, the server side is generated from a K8s deployment resource where the number of replica of the ServerPod is always disposed to one, except in one case, which will be reported later.

The third feature common to all test cases are the number of measurements done, based on tool used.

In each test, where iperf2/iperf3 are used, twelve measures are made and the output given to the user is the average of these without the best and the worst one.

With netperf things are different: being the tool structured differently also the test has a dissimilar behaviour. In fact, the tool itself reports a measure confidence interval and an error range that represents how much the measured

value deviates from the real one (in percentage). If the output of test does not have an error range superior to the default one, the measure is considered good, otherwise the test is carried out again. This flow is repeated, in the worst case, five times and as result, if the error range continues to be greater than the default, the measure with the best range error is taken (so, the smallest).

The segment size sent is another difference between iperf3 and netperf in TCP mode: in fact, while with iperf3 it is possible to set any size, with netperf this is not possible

Curl is used in only a scenario, when the number of Serverpod Replica is major of one, as Iperf and Netperf fail to work in this case. In this case, six measurements are taken and the four centrals are used to calculate the average. In addition to the average speed of downloading/moving a file between two pods, in this case the latency average is also measured, understood as already explained above Time to First Byte (TTFB).

The last aspect is the namespace created before measurement execution. For each test a specific namespace is created, to isolate test execution from the other namespaces and pods present in the cluster. As soon as the tests are terminated or some error occurs, the namespace is deleted to return the cluster to its initial conditions.

5.4.3 PTP Communication

As described in the previous section, for Pod to Pod Communication two main scenarios are considered: the first is the ideal scheme (as the picture 5.1 remembers) where only two pods are created in the cluster. Nothing from implementation point of view must be added, from what was said previously. In fact, being the simplest cluster configuration ever, nothing else needs to be configured besides the two aforementioned pods.

The second written test, for this scenario 5.2, requires some further clarification. Starting from the previous pod to pod communication, several silent

pods are added to the cluster. Through a configurable parameter it is possible to decide the number of pods to create, in addition to the other two in charge of the measurement, to avoid that the cluster is in the ideal conditions as before. After the creation of aforesaid pods, several network policies are installed that only allow Clientpod and Serverpod to send and receive traffic with each other and with other newly created pods.

By default, the benchmarking suite carries out this last test two times with initially 1k pods (and 2K network policies) in the cluster and then with 10K pods (and 20K network policies).

Being interested, in these two test, to TCP and UDP traffic, netperf and iperf3 tools are exploited to measure the throughput and the CPU usage both for intra-node communication and inter-node communication.

Next tables 5.1 and 5.2 list the commands used with these tools.

Protocol	Iperf3 server	Iperf3 client
TCP	iperf3 -s -p 5002 -V	iperf3 -c PODIP -p 5002 -V -N -t 10 -Z -A 1,2 -M 1448
UDP	iperf3 -s -p 5003 -V	iperf3 -c PODIP -u -b 0 -p 5003 -V -N -t 10 -Z -M 1448

Table 5.1. The Iperf3's commands used in TCP and UDP mode PTP communication

Protocol	Netperf server	Netperf client
TCP	netserver -p 15001 -v 2 -d	netperf -H PODIP -T 1,2 -i 30,2 -j -p 15001 -c -C – -D
UDP	netserver -p 15003 -v 2 -d	netperf -t UDPSTREAM -H PODIP -i 30,2 -p 15003 -v 2 -c -C -T 1,2 – -R 1 -D

Table 5.2. The Netperf's commands used in TCP and UDP mode PTP communication

5.4.4 PSP Communication

Tests written, for when there are one or more services in the cluster, as reported in these pictures 5.3 and 5.4, have some more complexity than those in the previous section. First of all the calculating tools used have had problems with services and UDP mode, in fact, while for netperf only TCP mode is used, to have a response for throughput when services are deployed and UDP protocol used, iperf2 was used instead of iperf3. Using this last tool, the tests lose the CPU usage, not being returned by tool.

As mentioned above, the same test is run several times, with different parameters each time, to calculate the throughput and when possible the CPU usage (in percent) The test is performed with the number of services present growing (1, 10, 100, 1000 and 10000), but this number can be chosen a priori. Consequently, also the number of pods created in the cluster grows along. In fact, before creating the Clienpod, the Serverpod and the service that connect the two, a check on a parameter passed from the main class to test is done. This parameter indicates the number of services and pods that must be present in the cluster during the measurements and when major of one, the correspondent number of services and pods are deployed.

A further parameter, as for the previous tests, tells where the Clientpod must be created, to test both inter-node communication and intra-node communication.

The following tables, as for the previous section, show the command with specific parameter executed in the pods both for client and server.

Protocol	Iperf3 server	Iperf3 client
TCP	iperf3 -s -p 5001 -V	iperf3 -c SERVICEIP -p 5001 -V -N -t 10 -Z -A 1,2 -M 1448

Table 5.3. The iperf3's commands used in TCP in PSP communication

The reader can observe the differences between table 5.1 and table 5.4. In

both case the throughput in UDP is computed, but in Pod to Pod communication with iperf3, while in this case with iperf2 and can also observe that no UDP mode is used with Netperf (see table 5.5).

Protocol	iperf2 server	iperf2 client
UDP	iperf -s -u -p 5003	iperf -c SERVICEIP -u -b 10000G -p 5003 -V -i 1 -t 10

Table 5.4. The iperf2's commands used in UDP mode PSP communication

Protocol	Netperf server	Netperf client
TCP	netserver -p 15001 -v 2 -d	netperf -H PODIP -T 1,2 -i 30,2 -j -p 15001 -c -C – -D

Table 5.5. The netperf's commands used in TCP mode PSP communication

Completely different is the test implementation for the last scenario, reported in this picture 5.5. For this cluster configuration, being the number of replica of Serverpod major than one, the tools used in previous tests are useless. To test also this kind of situation, the curl tool is used. In each ServerPods is created, by an init container, a 10MB file that, later, will be requested and transferred to the Clientpod. In this case, in addition to the network speed, latency is calculated during the file transfer, defined in this case as time to first byte (TTFB).

But, what is an init container? An init container is a container executed in a pod before the "app" container. This kind of container is exploited to do several works for the "app" container: in this case a file is created that is processed by the container containg curl, but in general it can be used to setup specific configuration, it offer "a mechanism to block or delay app container startup until a set of preconditions are met"[25] and other actions useful for the application.

This test can be set up as the user want. The default configuration executes this tests several times, growing the number of services present in the cluster and, also, the number of replica of ServerPod. While for the number of services present in the cluster the same mechanism presented before are used, to set the number of Serverpod's replicas another parameter, passed to the test function, is used and written in the appropriate field in the Deployment declaration. The default configuration runs this test with 10, 20, 50 and 100 ServerPod's replicas (the upper bound is 100 to avoid problem on worker node in test environment) and a growing number of services in the cluster (1,100,1000 and 10000 services and pods).

The command executes several times from Client Pod is the following:

Protocol	server tool	client tool
HTTP	nginx	curl http://ServiceIP:8080 -o dev/null -w TTFB: %{time_starttransfer}

Table 5.6. The curl's commands used when multiple ServerPod replica are created

The next table reports information on docker image used in different tests:

Tool	Server	Client
iperf3	networkstatic/iperf3	networkstatic/iperf3
netperf	leannet/k8s-netperf	leannet/k8s-netperf
iperf2	leannet/k8s-netperf	leannet/k8s-netperf
curl	nginx	networkstatic/iperf3

Table 5.7. Table that represent for each tool the docker image used both client side and server side

Chapter 6

Test Environment and background works

This chapter illustrates the test environment where tests have been executed and it presents some background works done to run every test in a proper way.

6.1 Test Environment

The test cluster, belonging to a laboratory at Polytechnic of Turin, where tests have been deployed and executed, is a bare metal cluster, composed of three nodes: one master and two worker nodes, each one deployed in a different physical server. These servers are in the same L2 network (as the reader can see in the picture 6.1).

The master is deployed in a desktop Computer with Intel(R) Core(™) i5-3450S CPU @ 2.80GHz processor, 8GiB DIMM DDR3 Synchronous 1333 MHz RAM and 82579V Gigabit Network Connection (1Gbits/s). The OS is Ubuntu 18.04 with the 4.15 0-70-generic Linux Kernel version and the K8s version is 1.16.3.

The two workers, instead, are situated in Dell Precision Tower 3620 (0687), with Intel Xeon E3-1245 v5 3.50Hz processor, 32 GB RAM e 1TB HGST Disk and Ethernet Controller XL710 40GbE for QSFP+. In both servers are

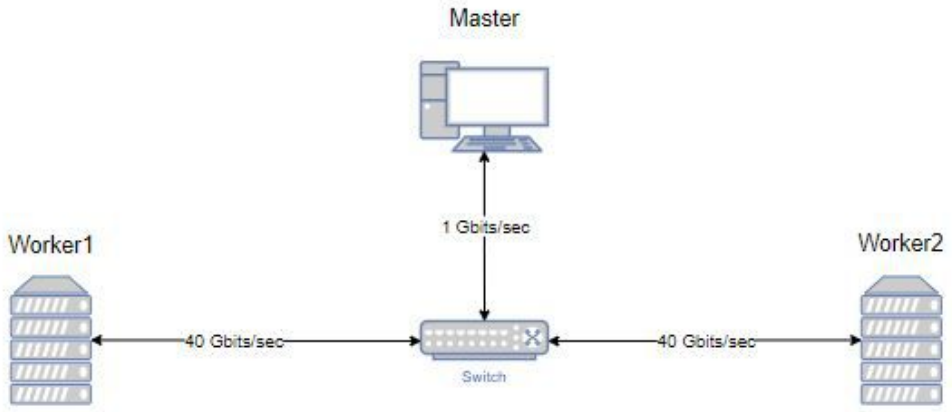

Figure 6.1. Representation of Test Environment

installed Ubuntu 18.04, the Linux kernel 4.15 0-101-generic and Kubernetes 1.16.3. The workers are linked with 40 Gbits Ethernet cable through a switch (Cisco 40Gbps).

6.2 K8shaper

K8s cluster shaper (K8shaper) is a framework, part of a larger project developed at Politecnico of Turin called LiqoTech, written in go language that allows the user to create a cluster of many virtual nodes as he wants.

This framework relies heavily on virtual-kubelet[1]. The virtual kubelet emulates the real kubelet process, creating a node resource and managing it, but in reality it is a pod running in one of the real nodes. From the cluster point of view, for each running virtual-kubelet pod a normal node is seen, so in this way it is possible to increment the number of workers present in the cluster and pods can be deployed on them.

However, the aforementioned pods have some limitations compared to those that are executed on real workers. First of all, for these pods the

[1]Virtual Kubelet, https://github.com/virtual-kubelet/virtual-kubelet

resources will never be allocated (being pods deployed on virtual nodes) and consequently their containers never started (the image never pulled). In addition, some vital information for proper operation are missing. Due to these limitations, pods will never reach a complete and correct Running phase, but a new status called "Mocked". A positive aspect is that a lot of pods will be able to be created without consuming resources, but in the other hand, they will not receive, send traffic or do anything else.

Why has K8shaper been used in this thesis? Kubernetes enforces a limit to number of pods schedulable on a node, by default 110. This configuration is modifiable with some commands, but, everytime that a new pod must be deployed, various checks on node resources are done and, if no enough resources are available, that pod stays in Pending status until something changes. For these reasons in small cluster it is difficult to create a large number of pods and related services and the test cluster, presented before, falls into this situation. In fact, some of the tests explained above require that a high number of pods be installed in the cluster, which cannot be scheduled only on the two real nodes.

K8shaper, with some code modifications, fits perfectly helping to create many virtual nodes, where pods can be deployed without consuming resources. In the tests presented before, there are some scenarios where the cluster has been dirtied for not having it in ideal conditions, through the creation of an huge number of pods and services. In this way, using K8shaper, a large part of these unused pods can be created without causing major problems to the two real workers and to the pods performing the measurements.

In the pictures 6.2 and 6.3 , it is possible to see an example of K8shaper and how works.

6.2.1 IPAM in K8shaper

As said before, nodes created by K8shaper instantiate pods that have some limitations. In order to be able to use this solution, to carry out the prepared

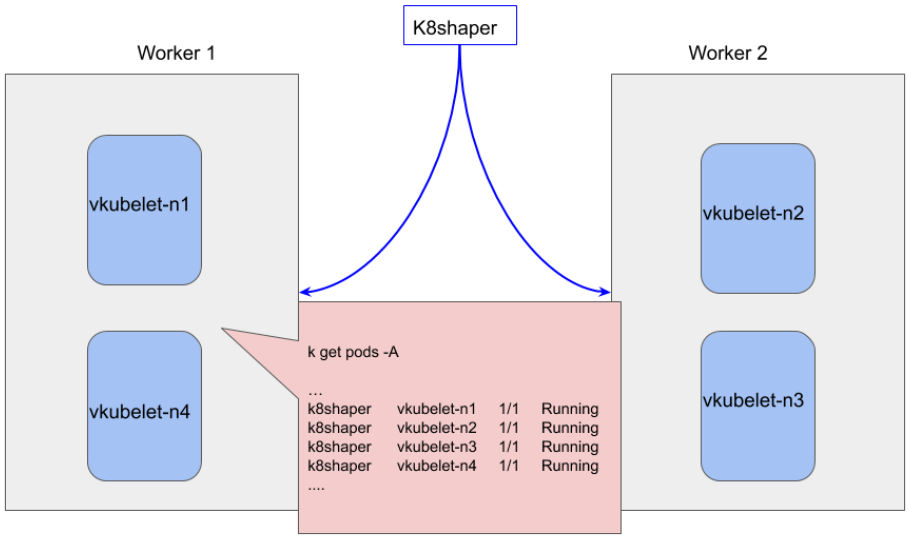

Figure 6.2. Pods created in the cluster, called vkubelet-nx which are seen as nodes by cluster

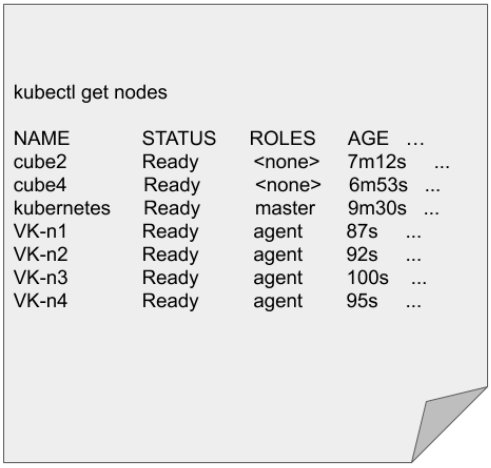

Figure 6.3. List of nodes present in the cluster. The reader can see the vk-nodes

tests, some changes have been made to the K8shaper code trying to decrease the limitations and obtain a pod image/version appropriate for tests.

The first major problem encountered was the IP address management.

Kubernetes leaves this management to each CNI and normally the CNI assigns one or more IP subnet (specified in the Pod Cidr field), taken from the cluster IP network, to each nodes present in the cluster. When a pod is scheduled on a node, it receives an IP address from that set of IP given to node.

With virtual kubelet the flow of IP assigning increases in complexity. First of all, the virtual kubelet is a pod deployed on a node, and it has an IP belonging to the node's subnet. When the pod is up and running, the control plane sees it as a node and assigns it an IP subnet coming from the network assigned to the cluster (specified in the Pod Cidr field).

Initially, in the K8shaper project to each pod created on a virtual node a fake IP was assigned, the same for all virtual pods, not being interested to pod networking. This configuration has been modified to make the project useful for the benchmarking suite. Indeed, creating a pod with correct IP makes easier the routing tables construction for real workers, avoiding overlapping IP conflicts and also the endpoints resources creation when the pod is associated with a services.

The work carried out can be divided into three parts that are briefly explained:

1. Get IP subnetwork allocated for virtual node:

   After the virtual-kubelet pod reaches the Running phase, it creates the node resource, the control plane of the cluster sees additional node and assigns it an IP subnetwork. The virtual-kubelet, since it becomes "Running", waits until the assignation is done to initialize correctly each pod allocated on virtual node.

   From implementation point of view, in the virtual kubelet execution, new thread, used to listen for events "Node Updates", is started, to notify the main thread when the subnet is assigned to node managed virtual-kubelet (here the "chan" go object is exploited to pass value between

different threads).

2. Assign a correct IP to each pod scheduled:

   Now, being available the subnet allocated for node, a thread safe mechanism must be used to give a correct address to pod and and also the release of the address, when a pod is deleted, must not be overshadowed, but controlled. Why thread-safe? Because it is possible that two pods from two different threads come in the same instant to request IP address and no mistake must be made (for example to assign the same IP or IP already used).

   A go library, found in the web, called "go-ipam"[2] has been used to implement the IP address management. It receives in node initialization the Pod CIDR assigned to it and is used everytime a pod is created or deleted. This go library affirms to be thread-safe, implementing a mechanism called Optimistic concurrency control (OCC)[3]

3. Update status pod in Running

   The last change made to the code is to apply to pod the "Running" status and not "Mocked" after it has received the correct IP. The endpoint object, essential for the correct functioning of communication through the service, is set correctly. Indeed, only when the pod is in the Running state is its IP added to the list of IPs that the service acts as a proxy, (remaining in the Mocked state this would never have happened). This passage, from "Mocked" status to "Running", has been implemented through a boolean flag passed in virtual kubelet construction phase and checked in pod creation phase.

As mentioned before, K8shaper allows to run the benchmarking suite with high number of pods scheduled in the test cluster, but unfortunatly this

[2]https://github.com/metal-stack/go-ipam

[3]https://en.wikipedia.org/wiki/Optimistic_concurrency_control

solution, as the reader can see in the next chapter, is not compatible with all CNIs to test.

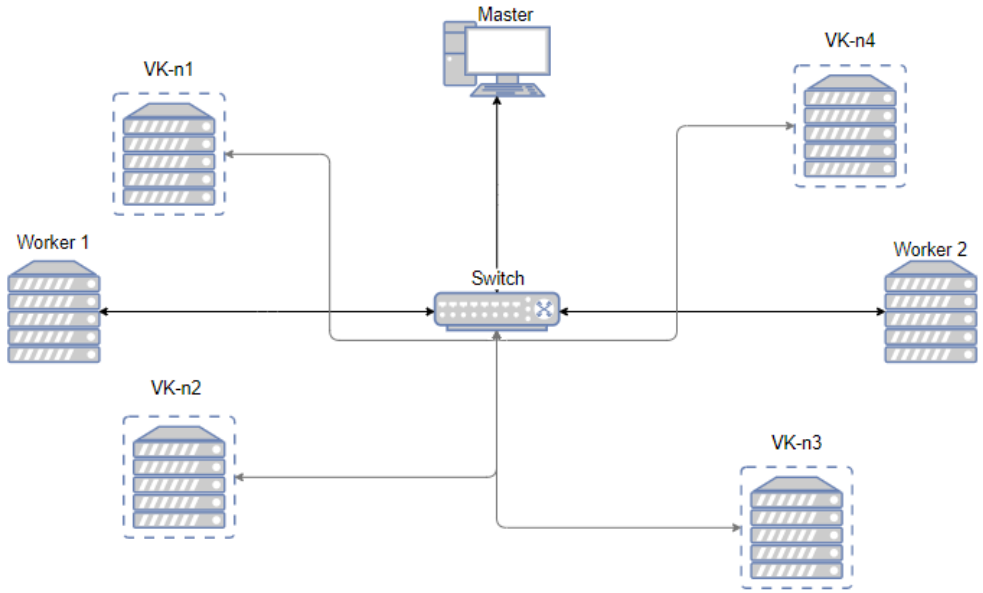

Figure 6.4. A cluster representation with four virtual-kubelet pods created

Chapter 7

Results Obtained

In this chapter two kinds of results obtained by benchmarking are presented in different sections. The former shows the outcomes reached/obtained running the test suite in the cluster illustrated in the chapter 6, while the latter reports if the tool is useful to spotlight the differences between CNIs with its pros and cons.

7.1 Experimental Results

In the next pages, test results are reported, without any ranking.

The next flowcharts show for each test the throughput reached up from specified CNI in Gbits/s with iperf3 and iperf tools, while those obtained with netperf are not disclosed both for sake of space and tool measure reliability.

Again for reasons of space, the amount of CPU used by the plugins in the various tests and the latency in communications will be examined at a high level.

7.1.1 Problems encountered

In the next flowcharts, Polycube's results, when the CNI is configured in native routing, are not comparable with the others for the following reason.

Polycube CNI with native routing configuration has been tested in a different period with respect to other CNIs, being added successively after seeing the outcomes of VxLAN configuration. Its results, however, cannot be compared with the others, due to some problems not resolved happening in the cluster. In fact, in this second measurement period a throughput decrease has been verified, caused by an event not understood, that seems to have created an upper limit for all CNIs about 21/22 Gbits/s for throughput in pod to pod communication, when pods are scheduled in different workers.

Another problem encountered while running the tests was the incompatibility between some CNIs and K8shaper. In fact, if Calico, WeaveNet, Flannel and Cilium had no problems in relating to this product, the other CNIs who for routing table reasons, who for reasons of internal K8s resources (namespace never completely deleted) were not able to be tested with a high number of pods and nodes in the real cluster.

For these reasons, CNIs tested are fewer than those with a small number of resources deployed in the cluster and consequently results reported in graphs.

The last issue to report concerns the tests with 10K pods created in the cluster. Initially, due to the large time taken to configure the cluster scenario (to create that number of pods it takes about 40 minutes and the same time is used in the delete phase) it was decided to carry out these measurements only with the three CNIs that they would have performed better in the previous scenario (1000 pods in the cluster).

Unfortunately, it was later noticed that with K8shaper these tests, even if performed, did not go well with Calico (in both modes).

7.1.2 Cni behaviour in a specific Test Scenario

Pod to Pod Communication TCP mode

In the simplest scenario 5.1, tested by benchmarking, both in inter-node and intra-node communication are evident the differences between CNIs, as the

graphs 7.1 and 7.2 report.

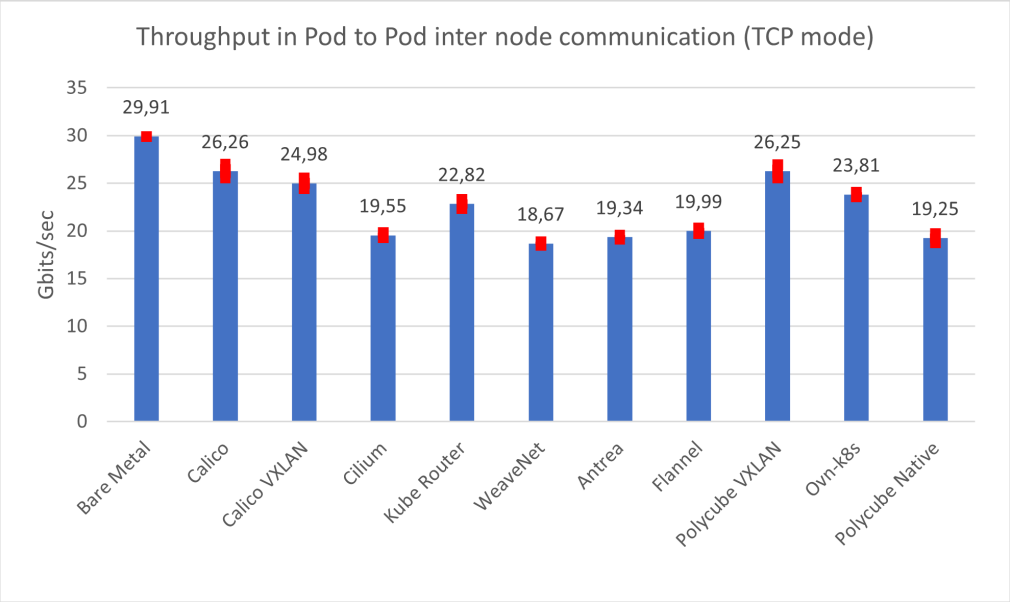

Figure 7.1. CNIs throughput with Iperf3 in TCP mode, in inter-node communication

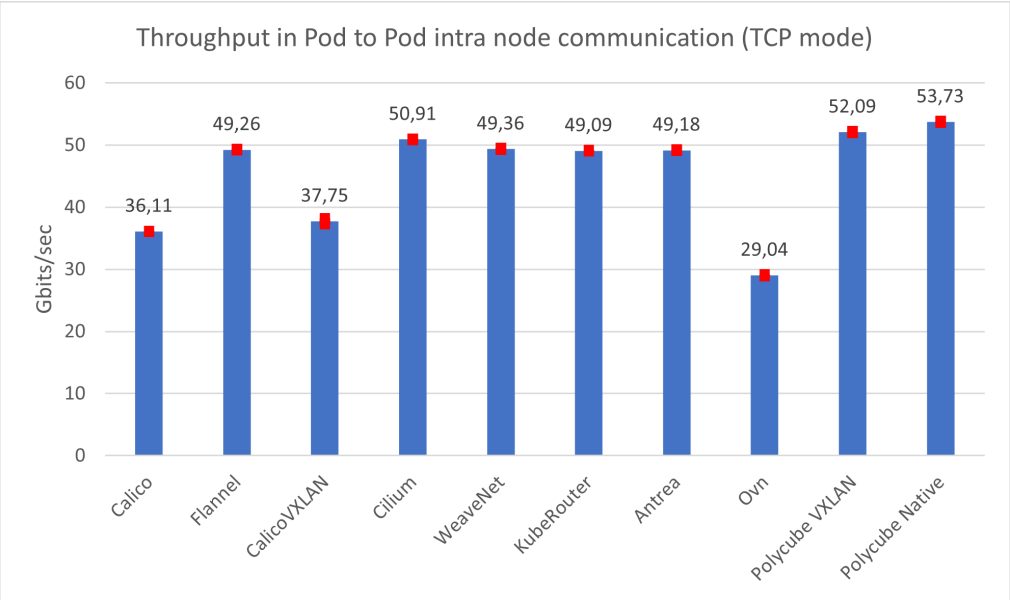

Figure 7.2. CNIs throughput with Iperf3 in TCP mode, in intra-node communication

Some CNI as Calico (in both modes), KubeRouter and Polycube Vxlan have a better behaviour then others when pods are deployed in different nodes.

Instead, in the intra-node communication , even if all CNIs have a performance increase, Calico increases less than others, while Polycube maintains the lead, followed by Cilium.

Pod to Pod Communication UDP mode

In the case of UDP, on the other hand, the outcomes reported by the execution of the tests, that the reader can see in the pictures 7.3 and 7.4, are all quite similar.

This protocol has not been tested for OVN: in fact, despite the transition to a more recent kernel (v 5.0.0) on the machines required by the CNI to work correctly, the installation was not completed due to small imperfections present in the plugin, making UDP communication between pods on different workers impossible or with errors.

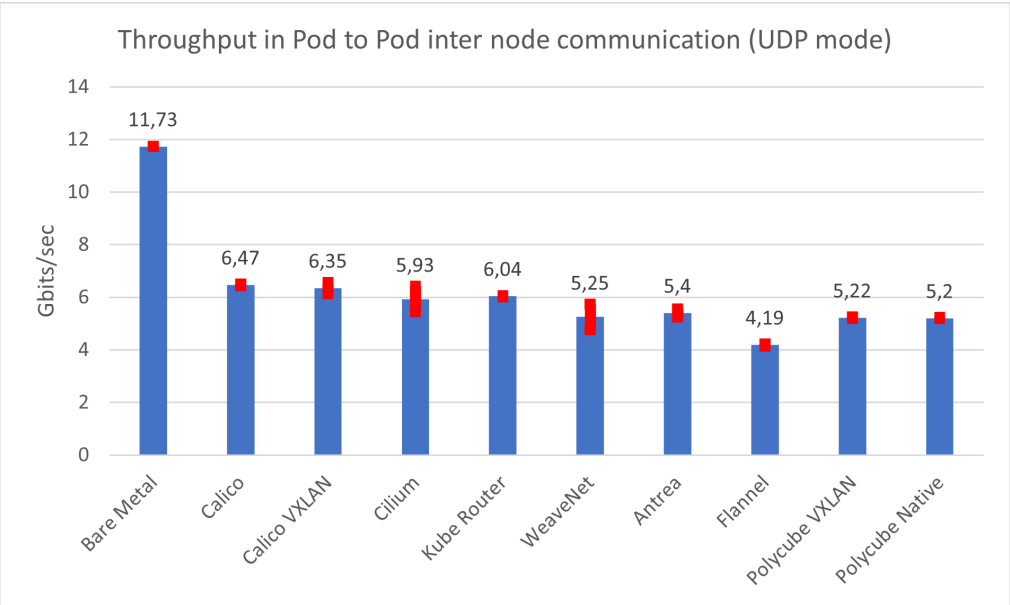

Figure 7.3. CNIs throughput with Iperf3 in UDP mode, inter-node communication

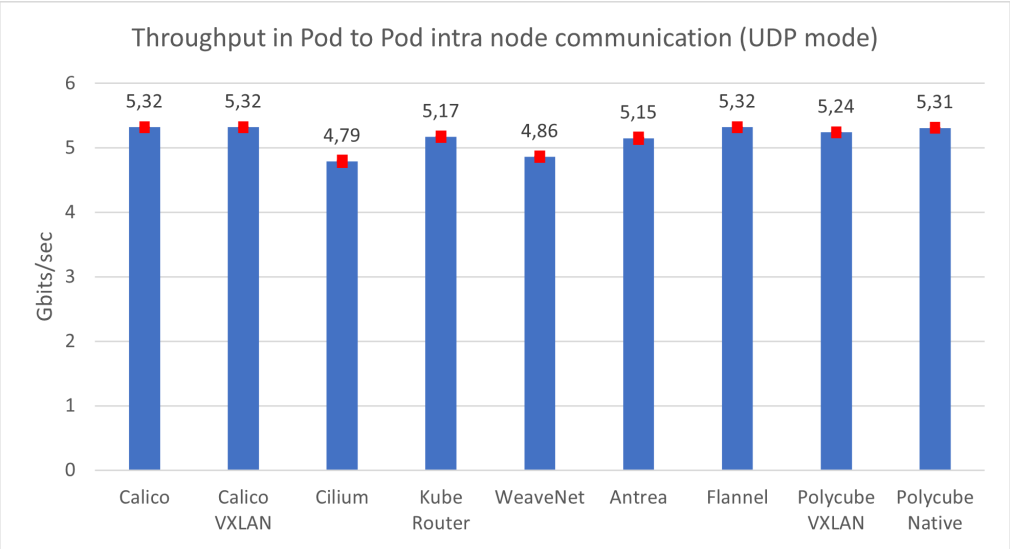

Figure 7.4. CNIs throughput with Iperf3 in UDP mode, intra-node communication

Pod to Pod Communication TCP mode and Network Policies

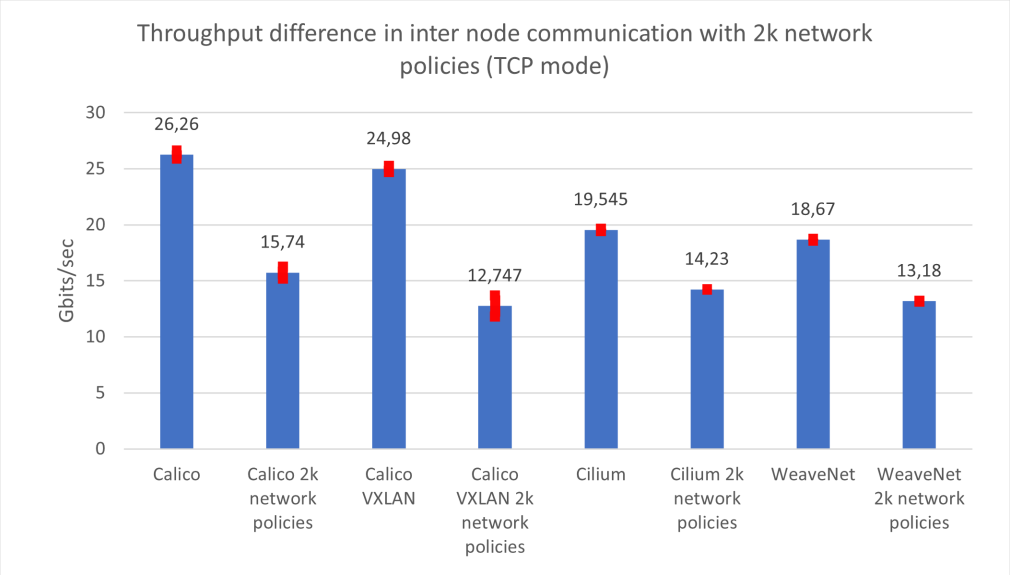

Figure 7.5. CNIs throughput with Iperf3 in TCP mode and several Network Policies are created

As the image 7.5 shows, with several network policies installed in the cluster and using TCP protocol, the reader can notice a worsening in the CNIs, some in a more evident way, some in a slightly more mitigated way due, in part, to the technologies used in the plugin: for example Cilium using eBPF scales better than those who only use the ip tables rules to implement the network policies.

Pod to Pod Communication UDP mode and Network Policies

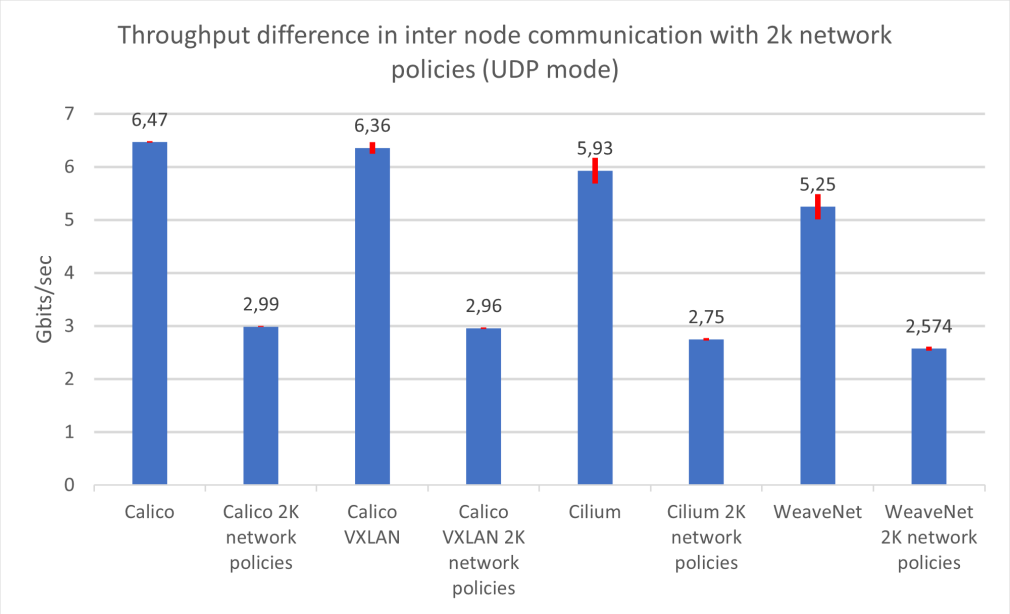

Figure 7.6. CNIs throughput with Iperf3 in UDP mode and several Network Policies are created

As the image 7.6 shows, with several network policies installed in the cluster and using UDP protocol, the throughput is halved approximately, without however that any CNI stands out, but with very similar results.

Pod to Service to Pod Communication TCP mode

Another difference, which the benchmarking tool tries to highlight, is how the CNI works with services, as their number increases within the cluster,

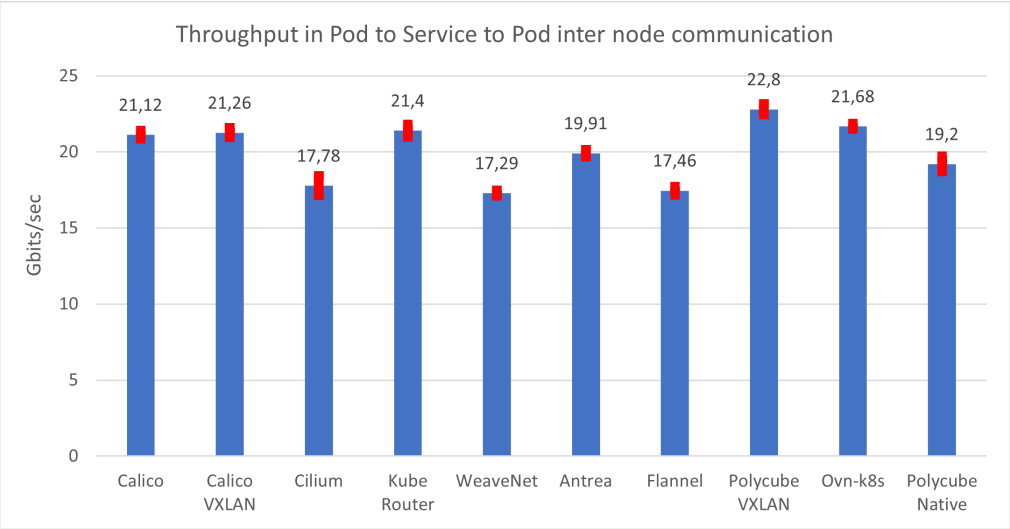

Figure 7.7. CNIs throughput with Iperf3 and a service deployed in the cluster

in order to expose the effects produced by the implementation choices of the various network providers.

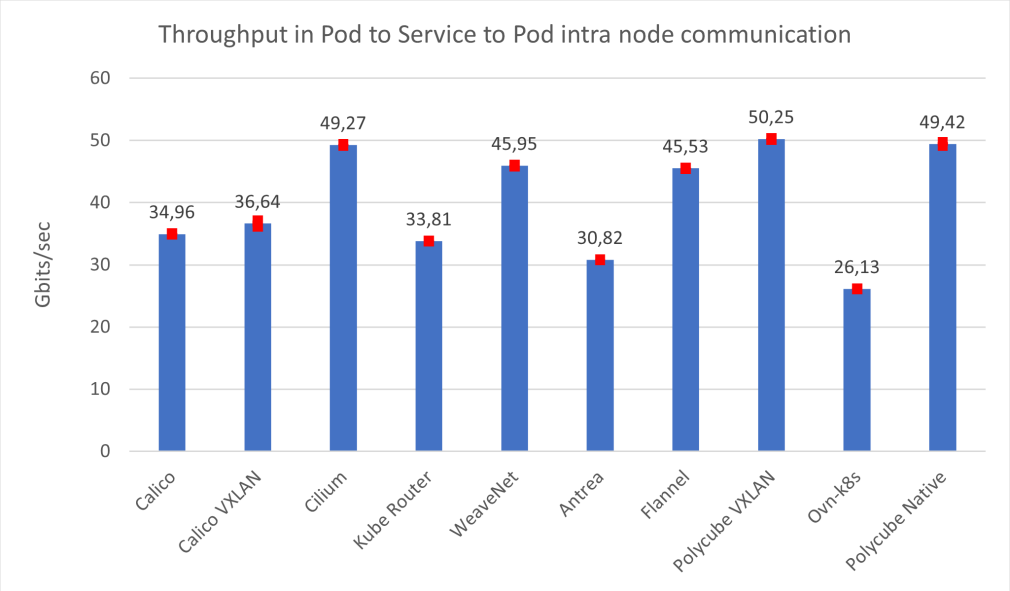

Figure 7.8. CNIs throughput with Iperf3 in TCP mode with one service in intra-node communication

As the reader can see in these graphs 7.7 and 7.8, both for the intra-node

and the inter-node communication a performance decrease happened. In both cases, plugins perform worse than the simpler pod-to-pod communication. It is possible to notice that some CNI have a small decrease, others have a more marked one if compared to results shown in 7.1 and 7.2 (for example Kube-Router and Antrea).

Unfortunately, it is not possible to show the behavior as the number of services increased for all the CNIs involved (as already explained above and as happened for the Network Policies). The following picture 7.9 shows how much the CNIs throughput deteriorates when in the cluster is created one service or when 1000 services are deployed. Here too, as already specified above, Cilium and WeaveNet mitigate the impact of the large number of services deployed, while the other CNIs worsen some by about 50% (Flannel) or by 33% (Calico).

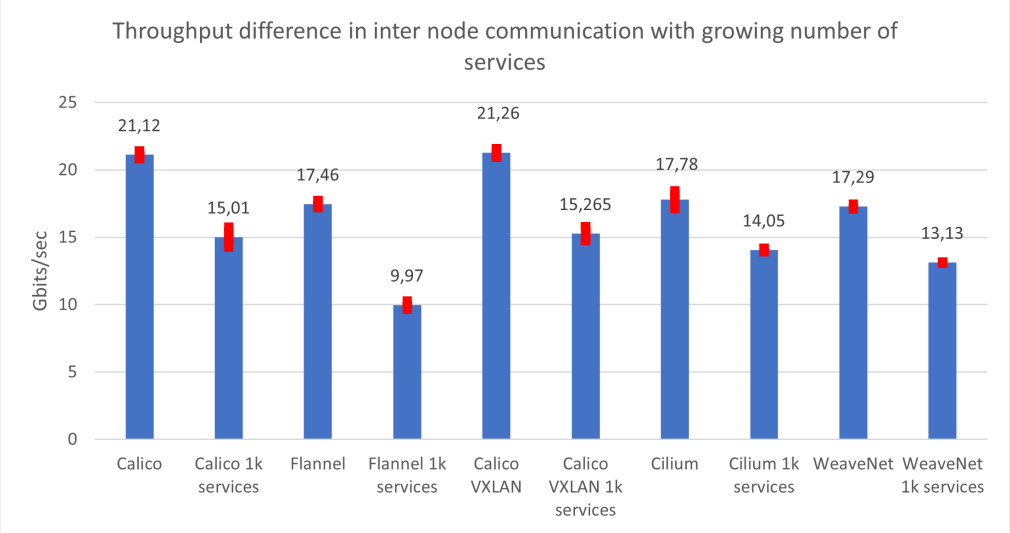

Figure 7.9. CNIs throughput differences with Iperf3 in TCP mode with one and 1000 services

7.1.3 General CNI behavior

From what the tests carried out report, it can be seen how some CNIs have (on average) a better throughput than others for TCP protocol when pods are deployed in different nodes. To the former group belong CNIs like Calico (in both BGP and VXLAN mode), Polycube and Kube-Router (also OVN-k8s), while to the latter Flannel, Cilium, Antrea and Weave Net.

The behaviour changes when the pods are scheduled in the same node: in this case CNIs that use eBPF have throughput higher (Cilium and Polycube), but also Flannel and WeaveNet result to work well in this scenario, while Calico worsens in both configurations his performance.

Although as the number of network policies or services present in the cluster increases, some CNIs respond better than others, it is also true that the best throughput achieved during the tests belongs to Calico which therefore seems to be the best even in these cases (despite having a loss greater, for example, in Cilium).

Regarding UDP protocol, CNIs have a close throughput: Calico (always in both configurations) takes the lead in scenario without services, while in the other cases it is Polycube to have a higher throughput. However, all these outcomes, in general, result much more similar than in the TCP case.

As for CPU utilization, the tool used has a non-negligible impact and risks obscuring the amount of CPU used by the CNI during its work. For example in a Pod to Pod communication with UDP the client CPU usage is on average for all CNIs 99%, while in a bare metal communication between the two workers is about 97% (so the plugin's impact is very low).

In addition, in some situations (all UDP tests and also intra - TCP tests) the CNIs outcomes are practically identical both client side and server side and so not very interesting.

More interesting are those of client side TCP: in this case the CNIs that use less CPU are those having performance less high (WeaveNet and Cilium)

and those that have better results utilizes more CPU (Polycube and Cilium), however, remaining very far from complete saturation (in one case about 25% of the core is used and in the other about 35%). When services come into play differences between CNIs in CPU usage diminish with some anormal behaviors (for example: Antrea, now, consumes more CPU than others). Ovn-k8s, focusing only on numbers, is the worst CNI for client CPU usage, but perhaps this is due to imperfect installation.

For the server side, Antrea's and WeaveNet's behavior is to put in evidence. In inter-node communication with UDP the CPU employed by these two CNIs is 1/10 compared to others (1% versus 10%). In the other scenarios in general, CNIs use the same amount of CPU on average with no major differences. In TCP tests in pod to pod communication inter-node some CNIs (WeaveNet, Flannel, Antrea and Polycube) have about the same attitude(45%), while KubeRouter is the worst (60%). If instead pods are created in the same node, Polycube (46%) differs in the smaller amount of CPU used when most CNIs have close outcomes (55%).

The latency is the last indicator monitored in this benchmarking suite: in this scenario, the test results show that some CNIs manage the growth in the number of services in the cluster better than others. For example, the following image 7.10 shows the latency when the number of ServerPod replicas is equal to 100, and the services first only one and then 1000.

On the other hand, the growth in the number of Server Pod replicas seems in most cases to have no impact on latency as can be seen from the following graph 7.11.

7.2 CNI benchmarking results

The thesis work carried out, after a theoretical analysis, focused on the creation of a test suite which, through specific KPIs, wants to highlight the behaviour of CNI installed, finding differences with other plugins that can

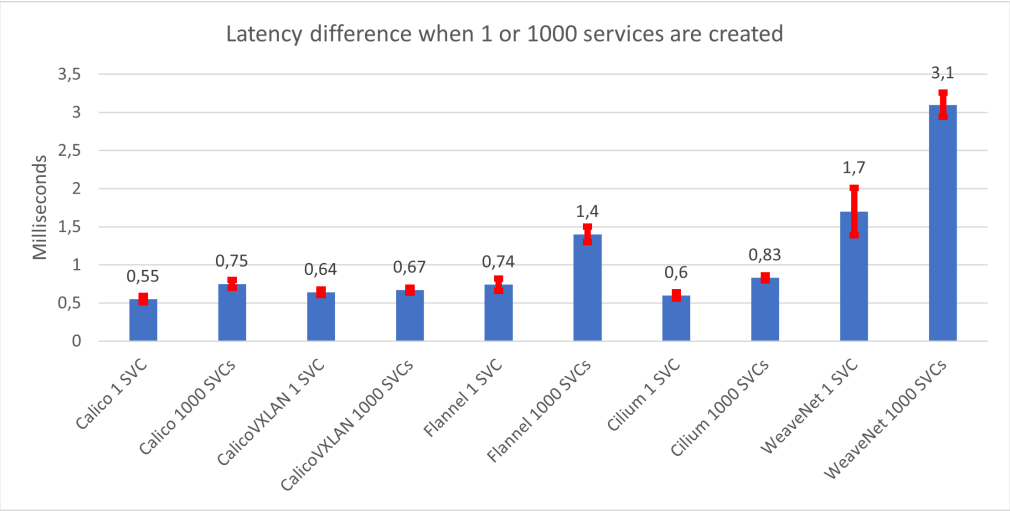

Figure 7.10. CNIs latency differences with one service and 1000 services when 100 ServerPods are deployed

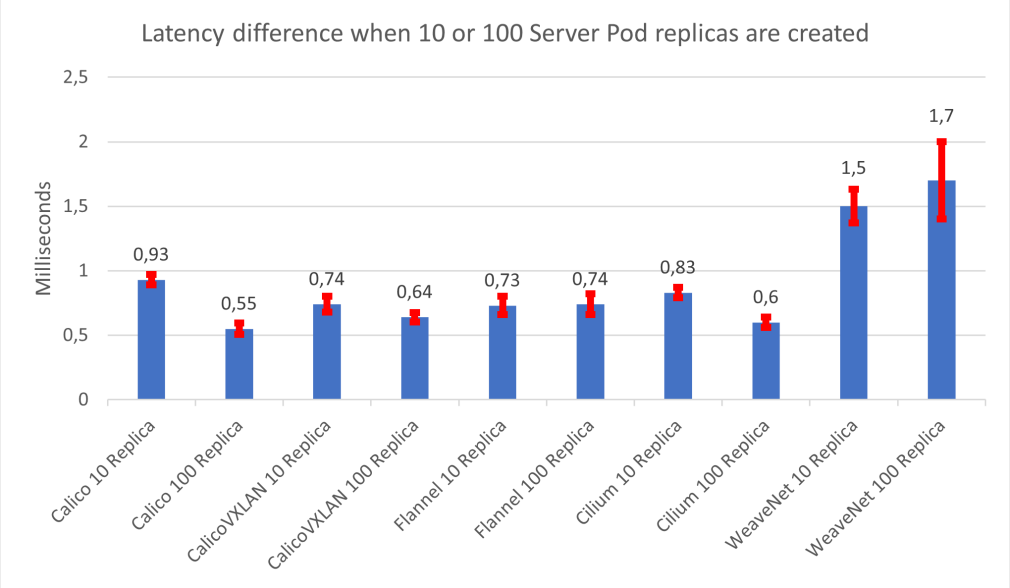

Figure 7.11. CNIs latency differences when 1 service is deployed and 10 or 100 ServerPods are created

be installed in the cluster. But, are the chosen KPIs adequate to achieve the tool's goal? And what are the pros and cons of this benchmarking?

In this section we will try to answer these questions, having in mind the

results, presented in the previous section, obtained by running the tool in the real cluster to demonstrate if a benchmark tool is useful and if that created in this thesis can be taken into consideration.

The benchmarking tool, as the results of the previous section demonstrate, is useful for getting practical information on CNIs. Both the tests that monitor the throughput and those for latency highlight how a CNI behaves in various scenarios, reporting useful results to the cluster administrator in choosing the best plugin to install in the cluster according to the need.

The KPIs chosen in most cases differentiate the plugins by showing how some behave in general in a better way than others from various points of view.

Furthermore, the test suite is completely automatic and does not require much effort to be launched: once execution has begun, all possible scenarios are executed sequentially without the user having to follow the process of creating each individual test. In fact, theoretically, any user after starting the tool can wait for its conclusion to then read the file created where he finds the results, without having to follow all the steps performed by the various tests. Inside the file it is possible to find in addition to the average, also the outputs of the single iterations of the tests that can be inspected in case something is not clear.

This solution, on the other hand, has some downsides. The first of these is time: by running the instrument automatically, about 60 cascade tests are performed which take many hours to run all. Some scenarios to be configured correctly, always referring to the cluster in which the benchmarking was performed, required several minutes each and if performed several times with different tools the time taken is very high: for example in tests that require the creation of 1000 pods the waiting time for the result is about 15 minutes each using the iperf only. Netperf also represents the biggest unknown in the equation of time: the time taken by the tool to calculate the throughput can vary a lot from a few minutes up to 35/40, depending on how long it takes

to find the measure in the interval of confidence required.

The second negative aspect of the suite created during the thesis is connected to this: to find an alternative to iperf, as already mentioned above, it was decided to perform the measurements in the same scenarios using the netperf tool. Some of the measurements made, however, with this second application are often unclear or with a large error range (even close to 30%), as well as lengthening the waiting time for test results.

The tool, for all its tests to be carried out correctly, needs the cluster to be large enough to contain all the resources that will be created because tests where many pods are created, otherwise, could cause problems.

Chapter 8

Conclusion and future works

In general, each network provider has its strengths and weaknesses and the choice of an appropriate CNI is not so simple due to the huge number of plugins available.

The presence of an automatic benchmarking tool, which stresses the CNI installed in the cluster in different scenarios, ideal and not, and returns information to the user simplifying the correct plugin selection, is therefore useful.

This benchmarking suite helps differentiate the performance of the various CNIs, despite having some obvious limitations.

In fact, it clearly distinguishes the performance achieved by the various CNIs in different scenarios, highlighting how, in general, some network providers behave better than others. Moreover, the automatic execution of the tests, the ease in launching the tool and the file created containing results represent excellent features of benchmarking tool, allowing even non-expert users to run it and analyze carefully the outcomes obtained.

One of the major weakness lies in the applications chosen to measure key performance indicators: although they are, in general, the most famous and used tools to measure these features, they are easily influenced by various

factors, even uncontrollable, with the consequent output of the test not very significant.

An intrinsic features is the waiting time for the results. Although the tests and measurements performed are many, the time spent running the entire test suite may be too high, especially for users interested in specific test results. This feature can be easily improved with small changes that can be made in the future.

The tool created, in addition, focuses only on certain features proposed by the CNIs, which may not be interesting for some users who would like additional information on other aspects (encryption for example).

Some tests, as they are structured, require an adequate number of nodes on which scheduling all the pods created. This, although not a real disadvantage, can be problematic for running the tool in small clusters, as those tests will never run correctly.

As for the second objective of the thesis, namely which was the CNI that performed best in the scenarios present in the benchmarking, it is clear that Calico and Cilium represent the best choices, combining the performances obtained in the various tests with good reliability and safety. Other CNIs, on the other hand, fail in one of two cases: Polycube, for example, does not always associate total reliability with excellent performance, while, instead, Weavenet, although reliable, does not have good throughput.

The CNI benchmarking tool certainly has many improvements that can be made, being a first version. As a future work, besides including more indicative and evidential scenarios, it can certainly be useful to create a filter that allows the user to parameterize the tests in a personalized way. In addition to running the tests with the default parameters, inserting the possibility for the user to configure the tests as he wishes (for example by setting the MTU size or the number of services and pods created in the cluster) can be an interesting improvement.

In fact, these changes would make the tests shorter and more interesting

for the user, being able to test one or more scenarios according to his needs, perhaps not being interested in all the tests.

Further future work could focus on finding better performing and less influential tools that can be used in benchmarking, in order to make the results more and more reliable.

Bibliography

[1] Kubernetes documentation, https://kubernetes.io/docs/concepts/overview/components/

[2] Services in Kubernetes, https://kubernetes.io/docs/concepts/services-networking/service/

[3] Network Policies, https://kubernetes.io/docs/concepts/services-networking/network-policies/

[4] Network Policies Example, https://cloud.google.com/kubernetes-engine/docs/tutorials/network-policy

[5] Container Network Interface, https://github.com/containernetworking/cni

[6] Flannel, https://github.com/coreos/flannel

[7] Networking in Flannel, https://blog.laputa.io/kubernetes-flannel-networking-6a1cb1f8ec7c

[8] Antrea, https://github.com/vmware-tanzu/antrea/blob/master/docs/architecture.md

[9] Cilium, https://docs.cilium.io/en/v1.7/concepts/overview/

[10] Cilium Flow, https://static.sched.com/hosted_files/onsna19/2f/Packet_Walks_In_Kubernetes.pdf

[11] Calico, https://docs.projectcalico.org/about/about-calico

[12] Contiv-VPP, https://github.com/contiv/vpp/tree/master/docs

[13] Ovn-kubernetes,https://github.com/ovn-org/ovn-kubernetes/blob/master/README_MANUAL.md

[14] KubeRouter, https://www.kube-router.io/docs/

[15] Polycube-Kubernetes, https://polycube-network.readthedocs.io/en/latest/components/k8s/pcn-kubernetes.html

[16] WeaveNet, https://www.weave.works/docs/net/latest/overview/

[17] Amazon CNI, https://github.com/aws/amazon-vpc-cni-k8s

[18] Benchmark results of Kubernetes network plugins (CNI) over 10Gbit/s network (Updated: April 2019), https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4

[19] Overview of Kubernetes CNI plugins performance, https://doi.org/10.3846/mla.2020.11454

[20] Tests Kubernetes Repository, https://github.com/kubernetes/perf-tests/tree/master/network/benchmarks/netperf

[21] Iperf3, https://github.com/esnet/iperf

[22] Netperf, https://hewlettpackard.github.io/netperf/doc/netperf.html

[23] Jobs in Kubernetes, https://kubernetes.io/docs/concepts/workloads/controllers/job/

[24] Throughput Definition, https://www.dnsstuff.com/network-throughput-bandwidth

[25] Init Container, https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

[26] Virtual Kubelet, https://github.com/virtual-kubelet/virtual-kubelet

Acknowledgements

E' giusto prima della fine di questo percorso ringraziare. Ringraziare tutte le persone che mi sono state vicine, condividendo con me momenti felici, ansie e fatiche.

Per prima cosa vorrei ringraziare il professor Risso, mio relatore, per la passione trasmessami durante le lezioni e per l'opportunità datami con questa tesi, incoraggiandomi a migliorare sempre più. In questi mesi ho imparato molto sia dal punto di vista professionale sia dal punto di vista umano, grazie veramente.

Grazie ad Alessandro Capello, ingegnere di Telecom, per essermi stato accanto in questo periodo, non facendomi mai mancare il suo supporto, la sua disponibilità e le sue conoscenze necessarie per la realizzazione della tesi.

Grazie ad Alex, Mattia e Raffaele per la disponibilità, la competenza e gli aiuti (innumerevoli!) che ho ricevuto e che hanno reso possibile lo svolgimento di questa tesi.

Un grosso grazie va alla mia meravigliosa famiglia per l'amore e il tifo che ho sempre sentito, e soprattutto alla mia mamma e al mio papà, che mi hanno trasmesso la passione per la cultura e per la tecnologia, standomi accanto (anche se lontani) dal primo fino all'ultimo giorno, facilitando i miei anni univeristari.

Grazie agli amici, sia a quelli del "venerdi sera" (Alex, Luca e Simone in primis) sia a quelli incontrati sui banchi (soprattutto i due Francesco ed i due Giacomo) per le risate, le bevute, le ansie ed i sogni condivisi in questi anni.

Grazie a Claude, fratello più che amico, per aver sempre creduto in me, strappandomi un sorriso anche nei momenti più bui.

Grazie, infine, ad Alessia. Per aver condiviso con me questo viaggio più di chiunque altro, per le insalate di Millikan, per i ripassi a tarda notte prima degli esami, per i momenti di svago, per avermi sopportato ed aiutato quando nessun altro lo avrebbe fatto, Grazie.