POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi Magistrale

Automatic Security in Kubernetes with Polycube

Relatore
prof. Fulvio Risso

Elis Lulja

Anno Accademico 2018-2019

To my family

Contents

List of Figures

Summary

The goal of this project is to provide automatic security features and functions to containers orchestrated by *Kubernetes* through the use of *Polycube*: from the deployment of a *Security Policy* to the automatic discovery and protection of the services involved.

Chapter 1

Introduction

Container platforms are used to package applications so that they can access a specific set of resources of the operating system running in a physical or virtual machine: in a *microservice* architecture, applications are split in various services, each of them is packaged in a separate container.

But as the number of containers and services grows larger, a system capable of managing such a situation, handling the life cycle of containers and checking their health, becomes essential.

Kubernetes is an *orchestrator* that provides the functionality of automatic deployment, management, scaling, and availability of containers. Although capable of providing the aforementioned features, it still needs a networking framework to make containers communicate with each other and the external world, as well as to protect them from unauthorized access and malicious traffic.

Polycube is a solution developed by the *Polytechnic University of Turin* that can create and destroy networking functions and make containers networking possible.

Chapter 2

Background

2.1 Kubernetes

Kubernetes is a portable and extensible platform for managing containerized workloads and services, open-sourced by Google in 2014.

It orchestrates computing, networking, and storage infrastructure on behalf of user workloads and was also designed to serve as a platform for building an ecosystem of components and tools to make it easier to deploy, scale, and manage applications.

Additionally, the Kubernetes control plane is built upon the same APIs that are available to developers and users: they can write their own controllers, such as schedulers, with their own APIs that can be targeted by a general-purpose command-line tool.

2.1.1 Nodes

Like most distributed computing platforms, a Kubernetes cluster consists of at least one *master* and multiple compute nodes, called *worker nodes*.

The master is responsible for exposing the API, scheduling the deployments and managing the overall cluster. Each node runs a container runtime, such as *Docker* or *rkt*, along with an agent that communicates with the master and additional components for logging, monitoring, service discovery and optional add-ons.

Worker nodes are the workhorses of a Kubernetes cluster: they expose compute, networking and storage resources to applications.

Finally, nodes can be virtual machines (VMs) running in a cloud or bare metal servers running within the data center.

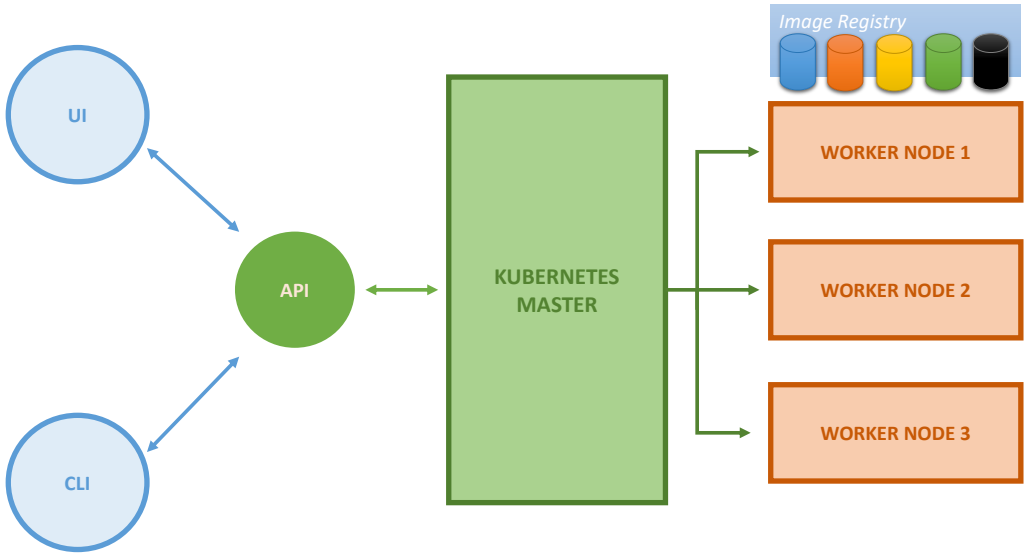

Figure 2.1. Kubernetes General Architecture

2.1.2 Pods

A pod is a collection of one or more containers and serves as Kubernetes' core unit of management.

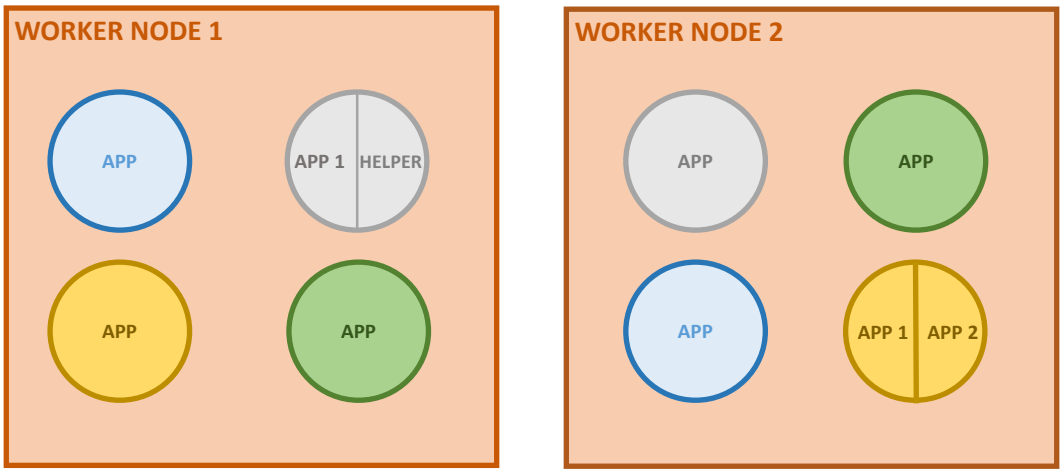

Figure 2.2. Worker nodes with running pods

Pods act as the logical boundary for containers sharing the same context

and resources. At runtime, pods can be scaled by creating dedicated resources called *replica sets*, which ensure that the deployment always runs the desired number of pods.

2.1.3 Namespaces

Kubernetes supports multiple *virtual* clusters backed by the same physical cluster, and each of them takes the name of *namespace*.

Namespaces provide a good abstraction for *environments*, i.e. when wanting to logically separate *production* applications from the ones that are currently undergoing *testing*, or applications that have a concept of *graphs*.

Pods belonging to different namespaces are only logically isolated: physical isolation is performed by other resources, such as *Network Policies*.

In the picture below, the color of each pod denotes its namespace: even if they physically run in different nodes, they seem to be running in their own cluster, separated from those in other namespaces.

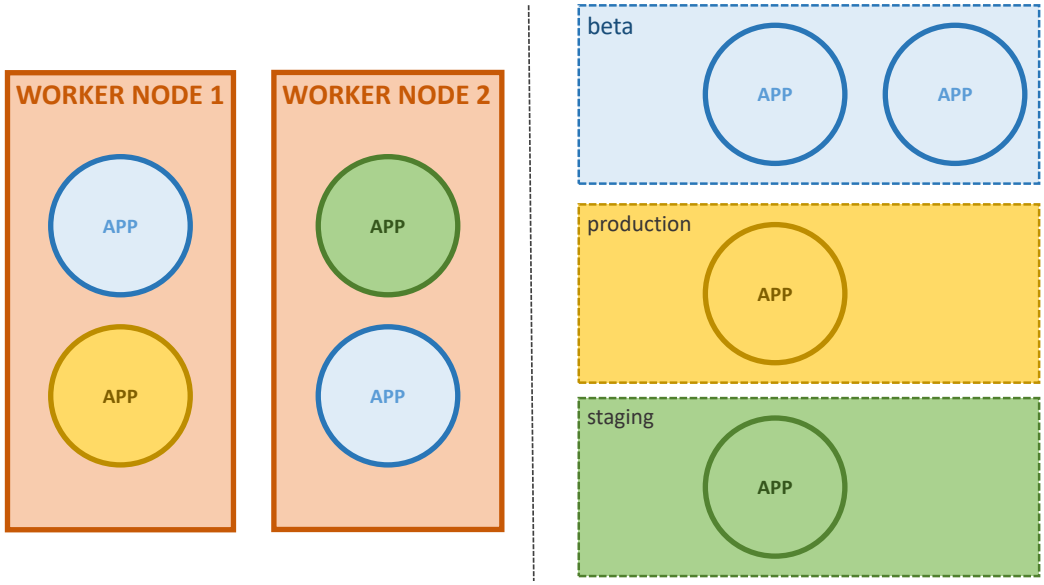

Figure 2.3. On the left: the physical reality. On the right: virtual clusters abstracted with namespaces.

2.1.4 Services

Services group a set of Pod endpoints into a single resource, solving the problem of knowing the IP address of every single one of them.

The image below serves as a good example of this: *frontend* pods do not need to know the IP address of every single *backend* pod, as they only need to know the *Service* applied to them.

The frontend only needs to query the Service's address, and a backend pod will be chosen as its peer, independently of the worker node it is running in.

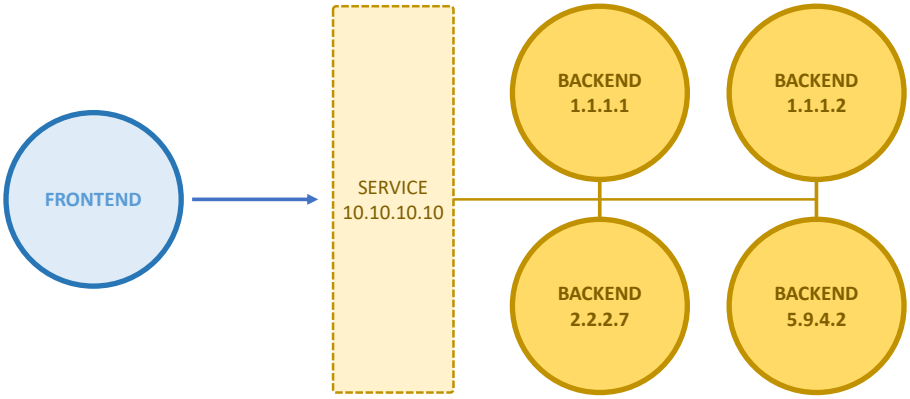

Figure 2.4. A Service resource knows the addresses of all the pods that expose the same functionality to other pods.

2.1.5 Kubernetes Network Policies

A *Kubernetes Network Policy* is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

Pods can communicate with each other by default, but as soon as a network policy is deployed and applied to them, they become isolated and can only communicate with allowed peers.

Network Policies need a network plugin in order to be fully functional: this is a situation where *Polycube* comes to the rescue.

2.2 Polycube

The creation of network functions in *eBPF* can result very challenging to end developers.

Polycube was born to provide a new software architecture and overcome these struggles: network functions can be managed and created on a centralized and more simpler way.

The tools already provided by it can be used to create complex services and scenarios.

Each network function created by Polycube is called *cube*: these are similar to plug-ins and can be instantiated and destroyed on demand. Additionally, they can be combined to create more powerful chains.

2.2.1 Firewall Cubes

Firewall cubes are transparent services that drop or forward packets according to certain criteria specified in the given rules list.

Matching is done by checking the following fields:

- IPv4 addresses: source and destination addresses can be specified in a *CIDR* notation.
- Level 4 Protocol: traffic can be filtered according to the transport protocol they are being sent with. *TCP*, *UDP*, and *ICMP* are supported.
- Level 4 Ports: both destination and source port can be specified.
- TCP Flags
- Connection Status: the firewall can discriminate packets according to the connection status they belong to.

The criteria are not mutually exclusive and empty rule fields are interpreted as wild cards and the according value in the packet won't be considered in the matching process in this case.

As for rule insertion, they are not applied until their processing is considered to be ready, thus providing *atomicity* features: only when all rules have been successfully processed and applied the policy can be considered to be effectively enforced.

Two ways of inserting rules are provided: the *Interactive Mode*, in which results can be seen as soon as the rule is inserted, and the *Transaction Mode*, in which rules must be manually applied to be effective. The latter is the best choice in case of multiple rules insertion, as results are applied only when all of them are inserted and, thus, avoiding unnecessary delays.

A *Default Action* can also be specified for those situations where a packet matches no rules: in this case it can either be dropped or forwarded anyway by setting the default action to *Drop* or *Forward* respectively.

Lastly, a *Statistics* feature is also available: the number of packets and bytes forwarded or dropped is presented, divided by the rule that matched the packet.

Chapter 3

Standard Kubernetes Network Policies

Cluster security is not handled by Kubernetes itself: security policies are correctly added to the cluster but totally ignored if no network plugin – or, anyway, any security provider – is installed. In this project, this job is entrusted to Polycube.

This chapter focuses on the network policies created and provided by Kubernetes, providing insights and features that are common to the *Polycube Network Policies*, which will be presented in Chapter 5.

3.1 Features

Securing the cluster means protecting pods from unauthorized access or preventing such pods from performing malicious requests or, more generally, that should not be allowed. The way to define this behavior is by creating policies.

The *Kubernetes Network Policies*, introduced in Chapter 1, are supported by Polycube.

Such policies provide a way to define peers that are allowed to instantiate communications, and can be either internal to the cluster or located in the external world. A very simple example is to use namespaces – introduced in Section 2.1.3 – to define different environments, such as `testing` or `production` and use policies to prevent pods from a namespace to access – and, potentially, mess things – resources on the other namespace.

Communications can be filtered by defining rules for both incoming and

outgoing connections and the transport protocol used: both *TCP* and *UDP* are supported. Only *IPv4* traffic is checked: anything that is not *IP* is forwarded regardless.

As soon as a network policy is deployed, pods are automatically protected and the appropriate measures are taken in case of cluster events: pods are automatically discovered as they scale up or down and protected with the appropriate rules.

In case of crashing pods, Kubernetes will instantiate them again and Polycube will protect them without any manual intervention.

Multiple policies can be applied for the same pod at the same time, sparing operator the headache of manually integrating new rules into already deployed policies.

3.2 Structure of a Standard Kubernetes Network Policy

This section focuses on the correct steps to take in order to define an effective policy (which have to be created in a YAML file), along with heads-up in using its features.

3.2.1 Common fields

Like all resources, Network Policies share a common structures with all other resources used by Kubernetes. These are:

- `kind`: it specifies the resource type that is going to be deployed.
- `apiVersion`: the resource's targeted API version.
- `metadata`: a collection of fields used to identify the resource.
- `spec`: data unique to the resource and useful to perform actions with it.

For Kubernetes Network Policies, the `kind` and `apiVersion` fields are defined like this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

3.2.2 Name and namespace

The name of the policy is unique throughout a namespace, not in the entire cluster.

The namespace defines the validity of the policy and where the pods to be protected can be found. If left blank, the policy will be deployed in the `default` namespace, but if the pod to protect is not in `default`, then it must be filled accordingly: forgetting to do so will make the network plugin select the wrong pods to protect and to restrict, or even fail to find them.

The name of the policy and its namespace validity must be defined in the policy's metadata:

```
metadata:
  name: test-network-policy
  namespace: my-namespace
```

3.2.3 The policy type

The `PolicyTypes` field, contained inside *spec*, defines the directions to filter: if filled, it accepts at least one value, i.e. when only restricting one connection direction – *Ingress* for incoming connections or *Egress* for the outgoing ones – and at most two values when restricting them both.

Below, an example of a policy targeting both directions:

```
spec:
  policyTypes:
  - Ingress
  - Egress
```

Caveats

The `PolicyTypes` field is actually used when wanting to restrict *Egress* connections: when this field is omitted, the default direction is *Ingress*.

As a result, if the policy is indeed intended to be ingress only, this field may be totally ignored and left blank, but if the policy is intended to be an egress only or to cover both directions, then the *Egress* value must be included in it.

Forgetting to do so may lead to subtle and naive errors, which in turn will make any network plugin fail to parse the policy or, even worse, parse it incorrectly: in this case, in fact, the parsing process is going to be performed anyways and no errors will be shown because the network plugin thinks the policy is ingress-only, making this error even subtler to debug.

3.2.4 Selecting methods

Before going on with the examination of a Kubernetes Network Policy, the ways to select objects must be introduced, as they apply to many of the subsequent fields and, more generally, to most objects in Kubernetes.

Labels

Like many resources, Pods can have labels: these are written in a `key: value` format and are used to identify them.

As an example, consider a simple application called `myapp` consisting of a pod running a database and one running a web server.

Since they both belong to the same application, a very clever thing to do is to make them share a `app: myapp` label.

Then, a `role: webserver` label may be assigned to the web server pod and `role: database` to the other one. The example below is visual representation of this.

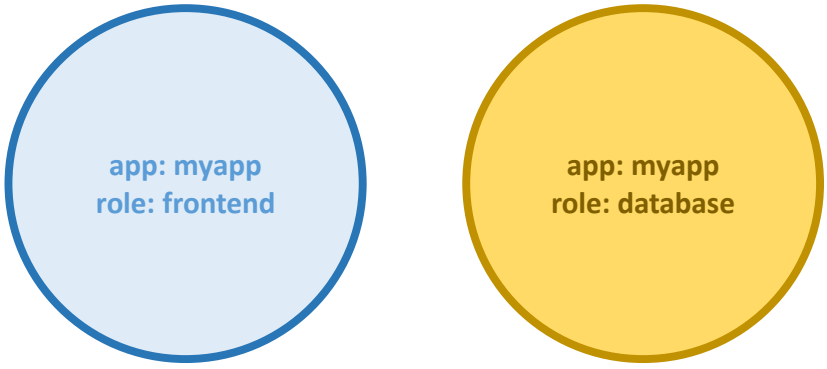

Figure 3.1. Two pods belonging to the same application but serving different purposes: their labels are used to reflect this situation.

Label selection

Many objects in Kubernetes can be selected by their labels: pods and namespaces, used in *Kubernetes Network Policies*, are among them.

Resources may contain as many labels as they need to, but they will be selected only if they match *all* the desired ones.

The image below shows which resources are going to be selected in case the following labels are provided:

```
app: myapp
role: frontend
```

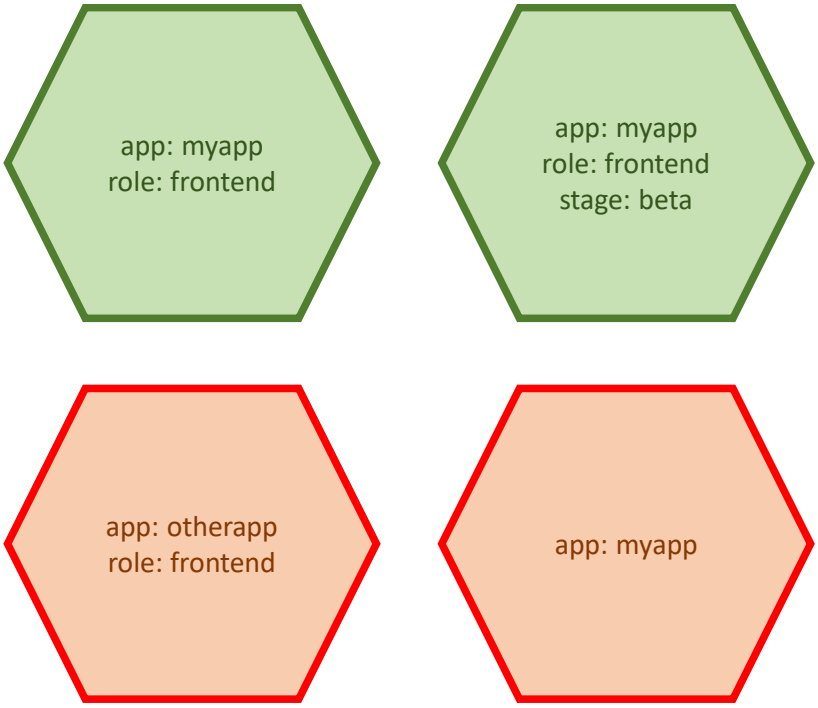

Figure 3.2. A visual representation of the labels selection process.

The objects in the image above are not pods or namespaces but general objects, and they are used to demonstrate the label selection algorithm: both the key and the value of the labels provided as needle should be present, regardless of all the other labels.

So, the resources on the first line are going to be selected correctly because they match *at least* the needed labels. Those on the second line are going to be ignored as the aforementioned rule is not satisfied.

Expressions

Expressions provide more flexible selection capabilities with operators like `In`, `NotIn`, `Exists`, and `DoesNotExist`.

Nonetheless, expressions are rarely used not only because label selection is more than enough for most clusters, but is not even supported by all resources. For this reason, in this project the label selection is the only selection method that is actually implemented.

All resources

Selecting all resources is done by using curly brackets `{}`.

Particular care must be taken if the namespace is not defined: in this case, the policy's namespace – defined in its metadata – will be used instead.

No resources

To select no resources, square brackets `[]` must be used.

In a *Network Policy*, since no resources are selected, this syntax is used to block all resources from a given namespace or the one defined in the policy's metadata, if not specified.

3.2.5 PodSelector

This operator selects the pods that need to be protected and it is included under the `spec` field. The selection methods defined thus far can be used, although at least one resource must be selected, so the `[]` syntax is not supported.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: protect-frontend
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: frontend
```

The policy above will be applied to all pods that contain the labels listed under `matchLabels` and that are in a namespace called `default`: as Figure 3.3 shows, the pod on the left satisfies both the label and the namespace condition, so the policy will be applied to it. The one the right, instead, is on a different namespace.

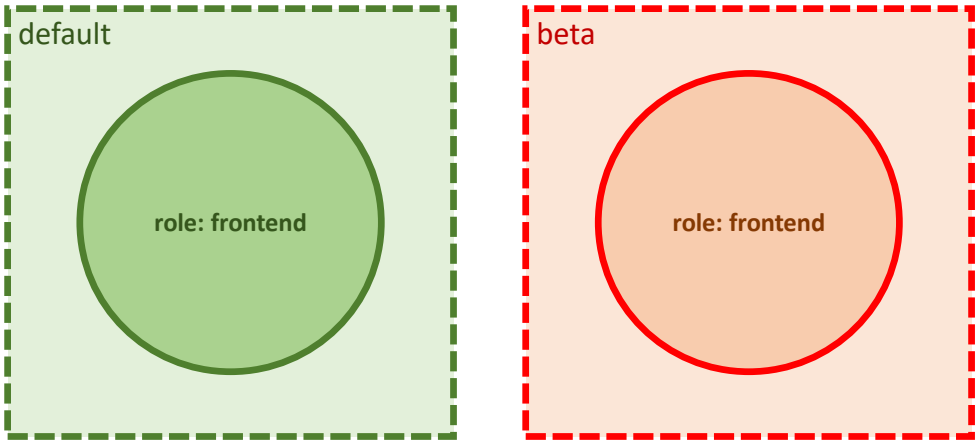

Figure 3.3. Two pods that have the needed labels but are on different namespaces.

3.2.6 Ingress and Egress peers

The `ingress` and `egress` fields start the restricting process: they include the peers that are allowed to communicate with the pod – or the other way around, respectively – and as such, they must be included under `spec` only once.

Depending on the direction, peers are defined under the `from` field for `ingress` or `to` for `egress`.

As a reminder, *Egress* should be added to the `PolicyTypes` field before using `egress`.

To make an example: the following policy restricts both `ingress` and `egress` connections. The parts that are not relevant have been cut, focusing on the policy types only.

```
spec:
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from: # Peer 1
      # Rules...
  - from: # Peer 2
      # Rules...
  egress:
  - to: # Peer 1
```

```
    # Rules...
  - to: # Peer 2
    # Rules...
```

3.2.7 Ports and Protocols

The transport protocol and the destination port can be defined by filling the `ports` field and include it in each peer selector. If empty, rules will match any protocol and any port.

Currently, Kubernetes Network Policies support `TCP` and `UDP`. `SCTP` protocol can also be included, but its rare usage and its *alpha* stage in Kubernetes led to the decision not to support it in Polycube.

As a result, policies that only contain `SCTP` will not be enforced: instead of leaving the cluster with potential vulnerabilities, such policies will translate in a "drop all traffic" action instead.

The following rule will match packets that are sent via `TCP` on port `8080` by the specified peer, not included for the sake of clarity.

```
ingress:
  - from: # Peer 1
    # Peer definition...
    ports:
    - protocol: TCP
      port: 8080
```

3.2.8 Selecting allowed peers

There are four ways to define the allowed peers: `IPBlock`, `PodSelector`, `NamespaceSelector` and a combination of the latter two, acting as a unique selector.

Instead of going into details for each them, they have been divided by their suggested usage or, better put, the peer's location they mostly mean to target: *The external world* and *The internal world*.

All the following examples relate to `ingress` connections and, thus, their inclusion under a `from` field is implied. Nonetheless, they also apply to `egress` by inserting them under a `to` field.

3.2.9 The external world

The definition means "anything that is not under direct control of Kubernetes": simply put, anything that is not a pod. For this purpose, the `IPBlock`

feature must be used.

IPBlock

This peer selector is the easiest one, as it allows a communication by simply writing the *IP range* the peers belong to, properly written in a *CIDR* notation.

Exceptions can also be defined: they are inserted under the `except` field and are written in the same fashion as the allowed IPs.

In order to allow all hosts inside the `100.100.100.0/24` subnet but excluding its first sixteen hosts, this peer selector can be taken as an example.

```
- ipBlock:
    cidr: 100.100.100.0/24
    except:
    - 100.100.100.0/28
```

Caveats

`IPBlock` can also be used to select pods inside the cluster, although this behavior is highly discouraged.

As a matter of fact, pods are a volatile – or ephemeral – entity: they can be instantiated and deleted on demand, and as such, the IP address that is assigned to them is not written in stone and will change accordingly.

As a consequence, using `IPBlock` to select a pod as an allowed peer means treating it as a fixed resource: it may work fine at the beginning, but after its first crash and re-deploy, no guarantee can be given as to what its new address will be. In fact, most probably it will be different from the one it had when the `IPBlock` field was defined.

This brings a non-negligible collateral effect: the automatic protection capability will be completely lost because no meaningful information about the pod to restrict was given to Polycube, and its IP address certainly cannot suffice.

For this reason, `IPBlock` must only be used when targeting something that is not a pod or is anyway located far from the Kubernetes cluster, i.e. private or proprietary machines: hosts from the university's campus, internal laboratories or proprietary servers.

3.2.10 The internal world

As opposed to the external world, the internal world peer selectors obviously focus on pods.

PodSelector

`PodSelector` simply selects groups of pods by using their labels as identifiers, following the same rules that have been presented earlier.

Used as a peer selector, both the `{}` and the `[]` operators can be used: for the former, all pods inside the policy's namespace will be selected; for the latter, no pods from it will be allowed.

Specific selection has already been discussed, so the next example focuses in the two operators just mentioned.

```
metadata:
  namespace: beta
spec:
    # ...
  ingress:
  - from:
    - podSelector: {}
```

The above policy will match packets from any pod inside the `beta` namespace, while the one below, instead, will reject them.

```
metadata:
  namespace: beta
spec:
    # ...
  ingress: []
```

NamespaceSelector

This selector does the same job as the `PodSelector` but applied to namespaces, with the additional functionality of selecting all pods inside them.

It is important to stress that `NamespaceSelector` does not care about the name of the namespace, but only about the *labels* of the namespace the peers are in, whereas the namespace name will only be considered for the pod to protect, i.e. the one defined in the policy's *spec.podSelector*.

```
metadata:
  namespace: default
spec:
  # ...
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          env: production
```

In the policy above, all connections coming from *any* pod inside namespaces that include the label `env: production` will be allowed.

Figure 3.4 better illustrates such behaviour: when specifying the pod to protect, its namespace is taken by name – `default`, defined in the policy's metadata – but the peers' namespace is taken by its labels, its name is ignored.

The name or the labels of the pods inside the namespaces is irrelevant, and the picture just focuses on the namespaces: the first line contains its name, while the second one contains its labels.

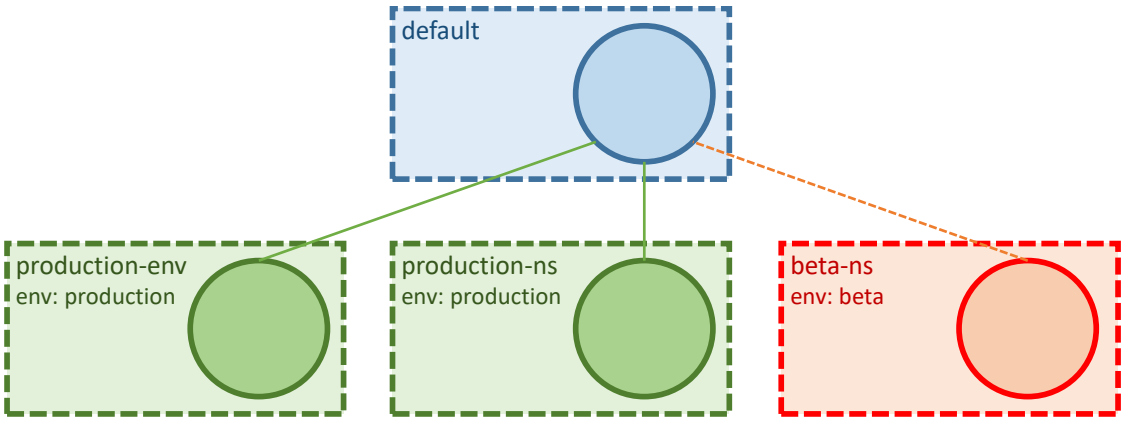

Figure 3.4. *namespaceSelector* works with the *labels* of a namespace: its name is ignored.

podSelector *and* namespaceSelector

If used together, `PodSelector` and `NamespaceSelector` allow for more fine-grained selections: only pods that have specific labels and are contained in namespaces that match the desired labels will be allowed.

The policy below is an example of the two selectors put together.

```
metadata:
  namespace: default
spec:
  # ...
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: api
      namespaceSelector:
        matchLabels:
          env: production
```

All pods with label `role: api` are allowed, but only if they are inside namespaces with label `purpose: production`.

Figure 3.5 elaborates more on this concept: pods that are on namespaces that do not match the labels included in *namespaceSelector* will never be considered. The pods that are indeed in the correct namespaces, instead, will be allowed only if their labels match the ones provided in the *podSelector*.

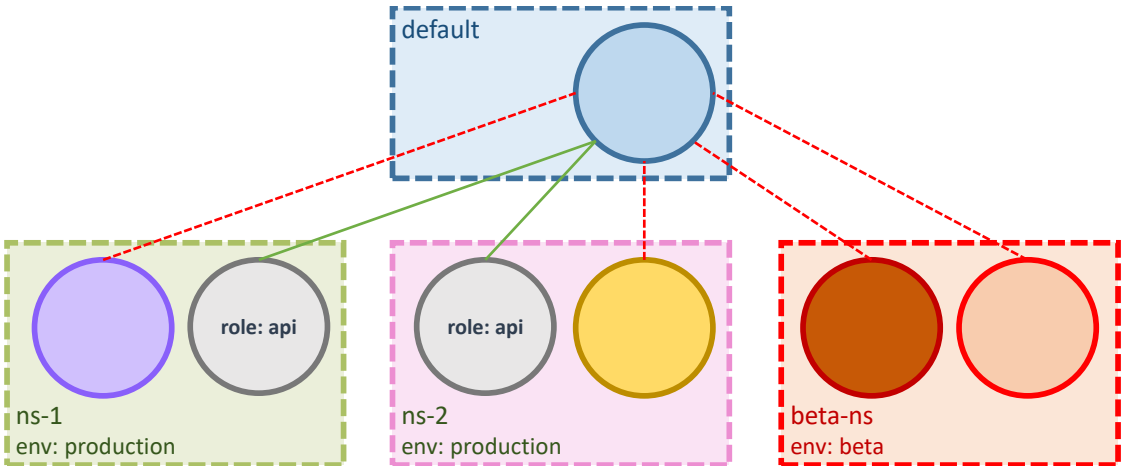

Figure 3.5. *namespaceSelector* and *podSelector* can be joined for a more sophisticated selection.

3.3 Combinations

This last part will focus on the results of combining the fields that have been described until now.

3.3.1 Combining Peers Selectors

When multiple peer selectors are included, the rules are obviously considered independently: this means that they are *OR-ed* with each other.

The following policy accepts connections from hosts belonging to the *100.100.100.0/24* subnet *OR* from pods that include `role: frontend` among its labels list.

```
- from:
  - IPBlock:
    cidr: 100.100.100.0/24
  - podSelector:
      matchLabels:
        role: frontend
```

3.3.2 Combining Protocol and Ports

Multiple protocols and ports can be defined, and the same method of the previous section is applied: protocols will be *OR-ed* in a non-exclusive fashion.

For example, to match packets that come with `TCP` on port `8080` *OR* with `UDP` on port `5000` the following selector can be used:

```
ports:
- protocol: TCP
  port: 8080
- protocol: UDP
  port: 5000
```

3.3.3 Putting them all together

Peers and protocols are *AND-ed* with each other: this means that they are valid at the same time.

```
- from:
  - IPBlock:
    cidr: 100.100.100.0/24
  - podSelector:
      matchLabels:
        role: frontend
  ports:
    - protocol: TCP
```

```
      port: 8080
    - protocol: UDP
      port: 5000
```

The policy above will match the following packets, each line is an allowed peer:

- source: `100.100.100.0/24` *AND* protocol `TCP` with port `8080`
- source: `100.100.100.0/24` *AND* protocol `UDP` with port `5000`
- source: `role: frontend` *AND* protocol `TCP` with port `8080`
- source: `role: frontend` *AND* protocol `UDP` with port `5000`

3.4 Deploying

To deploy a policy, the traditional `apply` command can be entered:

```
# Local policy
$ kubectl apply -f path/to/policy.yaml

# Remote policy
$ kubectl apply -f https://example.com/policy.yaml
```

The system will start a validation process, and in case errors are found, a comprehensive error message will be displayed. Otherwise, the usual successful message should appear and Polycube will start fetching the policy.

The same command serves for the updating process, but in this case both the name and the namespace of the policy should be left the same, otherwise the policy is interpreted as a new policy.

To get the list of the policies currently deployed, one can use the `kubectl get networkpolicies` command, though appending `-o wide` is suggested:

```
# Get list of policies on the default namespace
$ kubectl get networkpolicies -o wide
```

Finally, to cease the policy, two commands can be used:

```
# Delete by path
$ kubectl delete -f path/to/policy.yaml

# Delete a policy called "api-allow"
$ kubectl delete networkpolicy api-allow
```

As a final note, all previous commands work in case the policy is deployed in the `default` namespace: to refer to policies in a specific namespace, `-n namespace-name` should be appended:

```
# Get policies in the production namespace
$ kubectl get networkpolicies -n production

# Or...
$ kubectl get networkpolicies --namespace=production
```

3.5 Viewing the results

When policies are deployed, they are immediately processed by Polycube with the goal of translating them into *firewall rules*. The results of this process can be checked in Polycube's *CLI*, with the `polycubectl` command. The following command will show rules that have been inserted in the ingress *chain* of a firewall called "fw":

```
# Show the rules in the ingress chain
$ polycubectl firewall fw chain ingress rule show
```

A table containing all the details of the active rules will be displayed, but it can be tailored to be shown in a *JSON* or *YAML* format by appending `-json` or `-yaml`.

3.6 Applying security: a basic flow

A couple of examples will be presented to illustrate what was explained until now, along with the concept of priority and event reaction.

Before beginning, an introduction to chains is made.

3.6.1 Chains

As Figure 3.6 shows, firewalls work with a *chains* concept.

Incoming packets, which are the ones that are going to be restricted by an *Ingress* rule in a network policy, travel on the firewall's *egress* chain.

On the opposite side, outgoing packets, the ones matched by the policy's *Egress* rules and forwarded by the pod, are sent on the firewall's *ingress* chain.

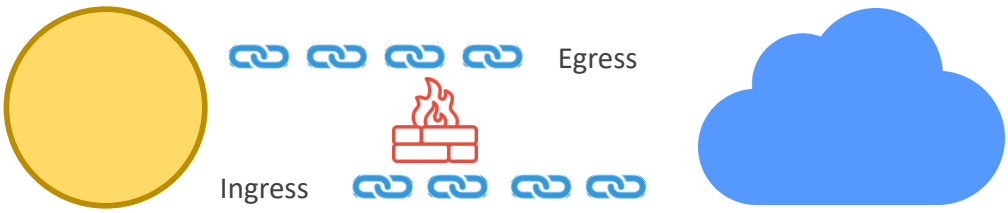

Figure 3.6. Traffic directed to the pod travels in the *egress* chain. Outgoing packets go through the *ingress* chain

As a result, in order to see rules and statistics about incoming packets, the *egress* chain must be consulted. The *ingress* chain must be selected for the opposite direction.

3.6.2 Preparing the environment

The first step is to start with two simple pods, one that acts as a database and has label `role: db` and one that acts as a front end server with a `role: frontend` label.

For demonstration purposes and to keep things simple, pods in this example actually run very simple applications and are all exposed on port 80. Their names are just high level abstractions.

A database

In the following examples, the database pod will be the one that is going to be protected and targeted by the other pods. In order to start it, the following command can be written:

```
# Start the pod, give it a name and a label, and expose it on port 80
$ kubectl run database --image=nginx --labels role=db --expose --port
 ↪  80
```

After a couple of minutes, a new pod is present on namespace `default`:

```
# Start the pod, give it a label, and expose it on port 80
$ kubectl get pods -o wide
```

A frontend

The front end pod, and all the other ones from now on, can be started with the following command:

```
# Start the front end pod and get inside it
$ kubectl run frontend --rm -i -t --image=alpine --labels
 ↪  role=frontend -- sh
```

The above command not only starts a new pod on namespace `default`, but also allows the user to get inside it and perform actions on its behalf by using its default *shell* application.

Preparing the node

The following command must be entered in the node that is running the `database` pod and will show the list of firewalls: they all have name in the format of `fw-<ip>`, where `<ip>` is the pod's IP.

```
# Show the list of firewalls in the current node
$ polycubectl firewall show ?
```

Supposing that the database has address `192.168.5.38`, the following commands can be used to see rules:

```
# Show rules and statistics
$ polycubectl firewall fw-192.168.5.38 chain <chain_name> show
```

Where `<chain_name>` is `ingress` or `egress`, as explained in Section 3.6.1.

3.6.3 First example: the internal world

This first example covers the following situation: the network administrator wants the database to be accessed only through the api server, any other entity should be prohibited from doing so.

In Kubernetes terms and for the environment that has just been created, this can translate to the following: pods with label `role: db` will only accept connections from pods with label `role: api` and that are inside the namespace called *default*.

Figure 3.7 shows such situation.

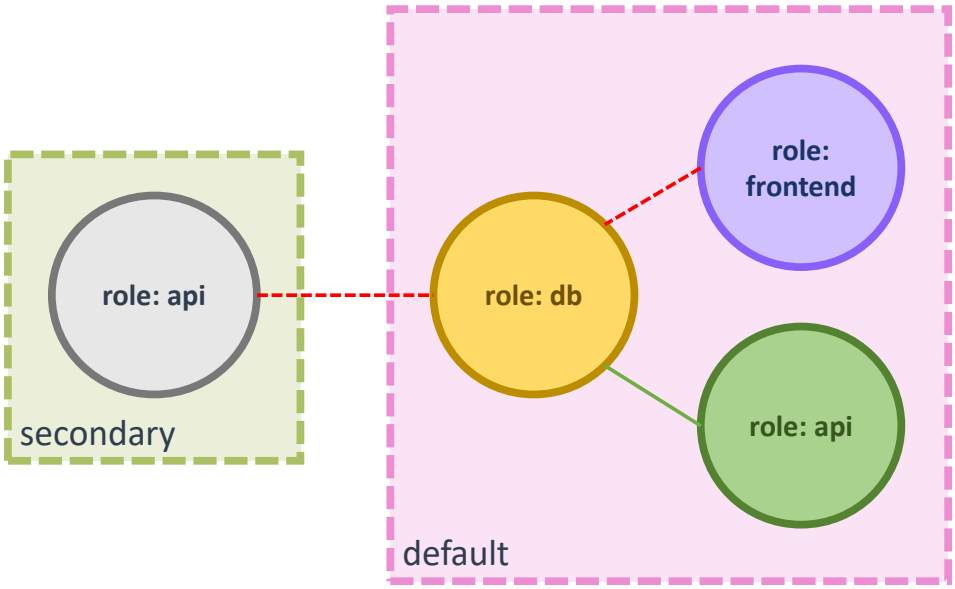

Figure 3.7. The example environment: databases can only be accessed by an api server located in the same namespace.

Rather than environments, this example treats namespaces actually as a group of micro services all working for the same *application*, though the names have been kept to `default` and `secondary` for simplicity.

Non-isolation

Issuing the following command from the shell of the pod acting as a front end will result in a message welcoming it to the database:

```
# Contact the database from the front end pod
$ wget -qO- --timeout=5 http://database
```

Since no policy has been deployed yet, the database is in a non-isolation mode and will accept all traffic.

This can be further verified by watching the rules regarding the `egress` chain of the database firewall: the default action is `forward` and so all connections are being accepted. The statistics will prove that a number of packets indeed travelled on the egress chain.

Protect the database

Of course this is not the desired situation, and it will be soon fixed by creating the following policy:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: api-allow
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
      - podSelector:
          matchLabels:
            role: api
```

Deploying this policy, as specified in Section 3.4, will make Polycube aware of the fact that has been a change in the *desired state* of the cluster: from now on, it will be in charge of making sure that the current state of the cluster always matches the desired one, and, more specifically, to protect the database from unauthorized access while granting it to all api servers that are part of the `default` namespace.

Verifying isolation mode

The situation of Section 3.6.3 is now reversed and the command will timeout after five seconds: the database just switched to isolation mode and will only accept connections allowed by the policy just deployed.

The statistics of the firewall's `egress` chain will now show a number of packets dropped and not allowed to further travel and reach the pod.

Non-isolation for outgoing packets

The policy did not include any policy type, and thus it reverted to the default situation, which is to only restrict *incoming* packets: the `ingress` chain of the firewall – refer to Figure 3.6 and Section 3.6.2 – will show that the default action is `forward`. This means that the database can start communications with anyone without restrictions.

Nonetheless, this does not represent a security hole for the database: as verified in the previous section, all incoming connections – no matter their status: `new`, `established` or `invalid` – will be dropped, unless they come from the api server.

So, the database can initiate communications, but won't be able to receive any reply, either because the other peer will not accept it or because the database itself will not accept its response.

The api server

The shell of the front end pod can be closed by entering `exit`.

In order to see if the policy is actually respected correctly, one needs to deploy an api server:

```
# Start the api pod and get inside it
$ kubectl run apiserver --rm -i -t --image=alpine --labels role=api
 ↪  -- sh
```

When the shell is ready, the database can be contacted again with the same method presented in Section 3.6.3.

The welcoming message will print on screen, confirming that the api server is indeed welcome to communicate with the database.

Firewall Reaction

What just happened is the reaction process in action.

In the previous section, the state of the cluster diverged twice from the desired state and the appropriate measures have been taken by both Kubernetes and Polycube to realign it, but in different contexts:

1. *A new pod is deployed*:
   - Kubernetes checks its specifications – i.e.: containers list, scaling needs, resources allocations etc. – and makes sure it is able to run correctly by scheduling it to the appropriate node.
   - Polycube enables it to perform network operations and instantiates a firewall to protect it. No policy applies to it yet, so it will be able to communicate with anyone.
2. *The new pod is an api server but can't contact the database yet*:

- The database firewall's default behavior for unknown peers is to reject their connections requests, and the firewall does not know who this new pod is yet: this is a clear violation of the policy and, thus, a difference from the desired state of the cluster.

  Polycube realizes this and makes changes in all the appropriate firewalls to fix this.

Changes in the egress chain

A further proof of the desired state being re-enforced can be seen by looking at the database firewall's `egress` chain: new rules, stating that packets sent by the new api server pod must be accepted, have been inserted.

The statistics confirms that some packets have transited successfully.

Namespace isolation

As stated in Section 2.1.3, pods belonging to different namespaces are only logically separated: now that a policy is deployed, isolation for incoming packets is enforced.

To test this, a new namespace can be created like so:

```
# Create a namespace called secondary
$ kubectl create namespace secondary
```

Now, Section 3.6.2 and Section 3.6.3 can be followed once again with a slight difference in the command, as `-namespace=secondary` should be appended in order to deploy them to the `secondary` namespace:

```
# Run a front end pod in the secondary namespace
$ kubectl run frontend --rm -i -t --image=alpine --labels
 ↪  role=frontend --namespace=secondary -- sh
```

The result will be the same for both situations: both the front end and the api server won't be able to access the database because they don't belong to the same namespace of the database.

As a reminder, isolation is only for incoming packets: the database can still communicate with pods in the secondary namespace but won't receive any reply, and, specifically for this case, not even from api servers. Refer to Section 3.6.3 for more details.

3.6.4 A Second Example: the external world

Finally, a second policy may be deployed, one that targets hosts outside of the Kubernetes cluster.

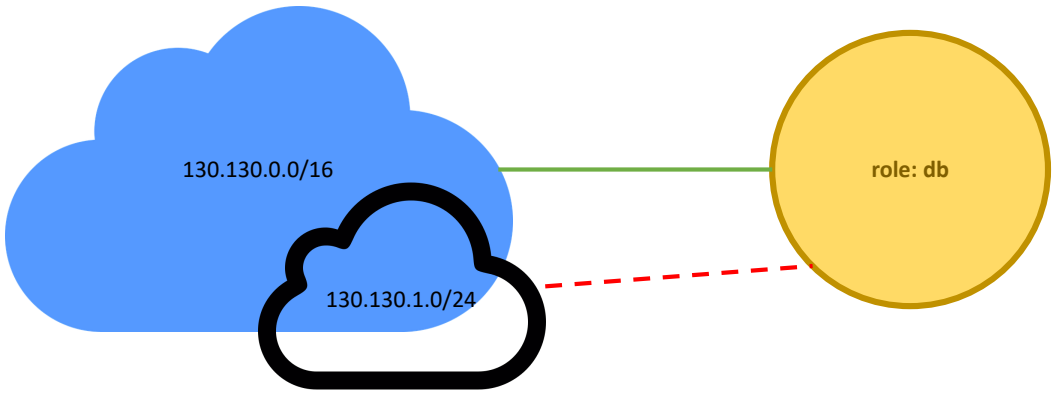

Figure 3.8. The database should be accessible by some external hosts.

Externally the database should be accessed by hosts from `130.130.0.0/16`, but not those that are from `130.130.1.0/24`.

Deploy the policy

To achieve this, the following policy may be deployed:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-external
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - ipBlock:
        cidr: 130.130.0.0/16
        except:
        - 130.130.1.0/24
```

Seeing the results

The `egress` chain of the firewall had an interesting change: the rules generated by this policy have been inserted *before* the ones generated by the one deployed earlier.

Priorities

This is the concept of policy priority: more recent policies take precedence against the older ones and thus the rules generated by the former will be checked first.

The examples performed thus far may not expose their purpose very clearly, but the following situation will do it. Suppose that two policies are deployed in sequence: first, one that rejects all traffic regardless of the peer and protocol and, later, one that selectively allows peers, like the one on Section 3.6.3.

Without a concept of priorities the second policy would be completely irrelevant: the first rule, generated by the policy that rejects all traffic, will match first and no other rule will be checked.

Clean up

To clean everything up, all open shells – the ones from an api server or front end – should be closed with `exit`.

Then, the following commands should be entered:

```
# Remove the database
$ kubectl delete deployment database
$ kubectl delete service database

# Delete the policies
$ kubectl delete networkpolicy api-allow
$ kubectl delete networkpolicy allow-external

# Delete the secondary namespace
$ kubectl delete namespace secondary
```

Chapter 4

Existing Solutions

This chapter will analyze the way the three main network plugins handle the security functionality.

4.1 Calico

Calico is integrated with all major cloud providers: from *Kubernetes* to *OpenStack*, *Amazon Web Services* to *Google Compute Engine* and currently uses the standard Linux kernel data plane, Windows Host Networking Service (HNS), and some capabilities of Extended Berkeley Packet Filter (eBPF).

It creates and manages a flat Layer 3 network, assigning each workload a fully routable IP address. Workloads can communicate without IP encapsulation or network address translation to increase performance, easier troubleshooting, and better interoperability. In environments that require an overlay, Calico uses IP-in-IP tunneling or can work with other overlay networking such as flannel.

Additionally, its data plane supports both IPv4 and IPv6, performs packet filtering at layer 3/4 via Linux kernel iptables with ipsets and supports configurable MTU.

4.1.1 Network Policies

Features

Calico supports both the Kubernetes Network Policies and its own take on security with Calico Network Policies, which provide a richer set of policy

capabilities, including explicit policy ordering/priority, different rule actions, IPv6, and support for multiple transport protocols.

Calico network policies can also apply to multiple types of endpoints, such as pods, VMs, and host interfaces and, when used with Istio service mesh, support securing applications on layers 5-7.

Other functions include `HTTP` match, ports range, negative field matching – i.e. the ability to allow all protocols except the one defined in the `notProtocol`.

Calico Network Policies, like the Kubernetes ones, are namespace-scoped: in order to be applied to pods in multiple namespaces, the *Global Network Policies* can be used.

Syntax

The syntax is very similar to a Kubernetes Network Policy with focus on operands that can be found in a programming language, like the "==" operator, and functions:

```
apiVersion: projectcalico.org/v3
kind: NetworkPolicy
metadata:
  name: allow-tcp-6379
  namespace: production
spec:
  selector: color == 'red'
  ingress:
  - action: Allow
    protocol: TCP
    source:
      selector: color == 'blue'
    destination:
      ports:
        - 6379
```

The policy above instructs pods that have a label with key `color` and value `red` in the `default` namespace to accept connections from pods with labels `color: blue` if they come with `TCP` and directed to port `6379`.

4.2 Cilium

At the foundation of Cilium is the BPF technology.

By leveraging Linux BPF, it retains the ability to transparently insert security visibility and enforcement, but does so in a way that is based on service/pod/container identity – in contrast to IP address identification in traditional systems – and can filter on application-layer, i.e. `HTTP` other than on the transport and network ones.

Networking is based on a simple and flat Layer 3 and also supports overlay and native routing using the regular routing table of the Linux host.

4.2.1 Network Policies

Features

Cilium's custom Network Policies are supported along with those from Kubernetes, and both are very similar in terms of functions.

The additional features provided include application-layer filtering for `HTTP`, `Kafka` and `DNS`, and support for services without selectors.

Syntax

Its syntax very closely resembles that of Kubernetes, focusing on labels, the use of the `{}` and `[]` operands, and very similar semantics. By deploying the following policy, pods with label `role: backend` on the `default` namespace will accept connections coming with `TCP` on port `80` if they come from pods with labels `role: frontend` on its same namespace.

```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "l4-rule"
spec:
  endpointSelector:
    matchLabels:
      role: backend
  ingress:
  - fromEndpoints:
    - matchLabels:
        role: frontend
    toPorts:
    - ports:
      - port: "80"
        protocol: TCP
```

4.3 Istio

Istio provides load balancing, service-to-service authentication, monitoring, and other functions by installing it as a "sidecar" proxy in the environment to intercept all network communication between microservices.

Its features include automatic load balancing for `HTTP`, `gRPC`, `WebSocket`, and `TCP` traffic; fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection; automatic metrics logs, and traces for all traffic within a cluster; secure service-to-service communication in a cluster with identity-based authentication and authorization.

4.3.1 Istio Policies

Istio's take on policies is very different from the solutions just mentioned as they are application-layer based and, so, policies can be based on virtual host, URL, or other HTTP headers. In order to use network policies, another network provider should be used, like the ones specified above.

The Istio's proxy is based on *Envoy*, which is implemented as a user space daemon in the data plane that interacts with the network layer using standard sockets. This gives it a large amount of flexibility in processing, and allows it to be distributed in a container.

Network Policies data plane is typically implemented in kernel space (e.g. using iptables, eBPF filters, or even custom kernel modules). Being in kernel space allows policies to be extremely fast, but not as flexible as the Envoy proxy.

Chapter 5

Polycube Network Policies

In order to provide additional functionality and leverage on the full power of Polycube's firewall component, a brand new set of Network Policies have been created called *Polycube Network Policies*.

5.1 Features

Polycube Network Policies are meant to include almost all the features that the Kubernetes ones already do and be compliant with their "philosophy", ease up the operator's work when securing the cluster, and add some additional features.

Obviously, Polycube policies have a different `kind` and `apiVersion`:

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
```

They share the same philosophy of isolation mode with Kubernetes policies: when a new Polycube policy is deployed, pods will stop accepting all traffic and will only follow what's stated there.

5.1.1 Human Readable Policies

Kubernetes policies can sometimes be challenging to read and, thus, use. One of the goals of this project is to also encourage operators to write more effective policies, and the very first thing that can help this process is a friendly and understandable syntax.

In Chapter 3 the `labelSelector` was introduced, but it was used both when selecting the pod to protect and when selecting the peer to be protected from.

Along with that, the `[]` and `{}` operators were mentioned: in almost all programming languages, they both mean "empty", but in Kubernetes, instead, their purpose is to select everything but with a different approach when it comes to the action: this may naturally lead to confusion and can increase the learning curve considerably, other than seriously harm the security of the cluster as a byproduct.

The following Kubernetes policy blocks all traffic:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-all
spec:
  podSelector:
    matchLabels:
      role: db
  ingress: []
```

Polycube policies perform the same job with this format:

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: deny-all
applyTo:
    target: pod
    withLabels:
      role: db
spec:
  ingressRules:
    dropAll: true
```

As shown, Polycube policies follow the natural language that humans speak – humans with a basic understanding of the English language, that is. As a matter of fact, when selecting the pods that the policy must be applied to, the `applyTo` field must be used, and the `target` field clearly specifies the need to protect a `pod` that has labels `role: db`, as finally stated in the `withLabels` field.

In order to drop all traffic, the `dropAll` can simply be set to `true`, leaving no room for confusion whatsoever.

The same thing applies when wanting to accept all traffic: the `allowAll: true` must be specified in this case.

```
# ...
  ingressRules:
    allowAll: true
```

Lastly, in order to apply a policy to all pods in the policy's namespace, the following `applyTo` field can be written:

```
applyTo:
    target: pod
    any: true
```

5.1.2 Automatic type detection

Section 3.2.3 explained that the `policyTypes` must be specified when wanting to target `Egress` traffic as well, and wrong policies may be generated if forgetting do so.

This behaviour can be found really tedious and annoying. As a further effort to bring more "friendliness", Polycube policies get rid of this by smartly recognizing the policy type accordingly.

The following policy does not mention incoming traffic in any kind, so it is going to be recognized as an `Egress` policy correctly.

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: pod-allow-all
applyTo:
    target: pod
    withLabels:
      role: db
spec:
  egressRules:
    allowAll: true
```

So, pods with labels `role: db` will be allowed to establish new connections with anyone.

5.1.3 Explicit Priority

Priority can explicitly be declared by properly writing it in the policy, where a lower number indicates its importance in rules insertion.

It can be done by setting the `priority` field:

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: pod-drop-all
priority: 1
# ...
```

This will tremendously help in situations where the removal of a policy cannot be done or when not wanting to do so.

Just to provide a very simple and trivial example, if wanting to temporarily block all accesses, a policy dropping all traffic – i.e. the one specified above – can be deployed with `priority: 1`. Supposing all other policies were deployed with a priority number greater than that, i.e with `2`, the rules generated by this policy will be inserted *before* all the other ones and, thus, checked before them.

Policies that have the same priority will be treated with the same rules as the Kubernetes ones: the most recent one will take precedence.

5.1.4 Strong distinction between the internal and external

The rules that can be specified are divided by what is internal to the cluster and what is outside.

This is done to prevent the clear bad behaviour of using `IPBlock` to target pods, as already mentioned in Section 3.2.9. Peers are divided in two groups: `pod`, and `world`.

The internal world can be specified like so:

```
ingressRules:
  rules:
    - action: allow
      from:
        peer: pod
        withLabels:
          role: api
```

```
      onNamespace:
        withNames:
          - beta
          - production
```

Once again, the syntax closely resembles a natural spoken language: the above policy `allows pods withLabels role: api` that are `onNamespace` either named `beta` or `production` to communicate with the target pod, here removed for shortness and clarity.

In Kubernetes Network Policies, namespaces can be targeted only by the labels they have: basically, when wanting to target them, the operator is forced to assign labels to namespaces even if they just need to target very few of them. As the policy above shows, Polycube Policies provide a way to select namespaces by their names as well, while also providing the ability to do so by their labels.

The external world, instead, can be restricted by writing `world` in the `peer` field. The example of Section 3.2.9 can be written in a Polycube policy in different way:

```
ingressRules:
  rules:
    - action: drop
      from:
        peer: world
        withIP:
          -  100.100.100.0/28
    - action: allow
      from:
        peer: world
        withIP:
          -  100.100.100.0/24
```

So, there is no need to write *exceptions*, because Polycube policies also have a clear distinction between actions, as written in the following section.

Drop or Allow

The flexibility of the Polycube firewall has been fully ported to the Polycube Policies, and actions can be specified for each rule: `drop` or `forward`:

```
ingressRules:
  rules:
```

```
    - from:
        peer: pod
        withLabels:
          role: api
      action: forward
```

In order to allow a connection, the actions that can be written are `allow`, `pass`, `forward` and `permit`.

The same applies when blocking connections, and the following words can be used: `block`, `drop`, `prohibit` and `forbid`.

The presence of multiple words to define a single action has been done to aid the definition of a policy, allowing for a more flexible semantic that is easier to remember.

Black list

Providing both `Drop` and `Forward` actions brings the possibility of creating black lists: everything is allowed, except for some connections that are specifically banned.

The following policy is deployed first:

```
priority: 2
spec:
  ingressRules:
    rules:
      - from:
          peer: pod
          any: true
        action: forward
        protocols:
          - protocol: tcp
            ports:
              source: 8546
              destination: 8080
```

This policy will accept all pods – provided they are in the same namespace as the policy's one – to contact the target pod on port 8080 from 8546 with `TCP`. The priority is `2` because later a "ban" policy is deployed:

```
priority: 1
spec:
  ingressRules:
    rules:
```

```
    - action: block
      from:
        peer: pod
        withLabels:
          status: beta
      protocols:
        - protocol: tcp
          ports:
            source: 8546
            destination: 8080
```

Now, all pods will be allowed to communicate with the pod *except* for those that have label `status: beta`. This policy, having a priority number lower than the one deployed earlier, will insert its rules before it: they will be checked before any other.

Finally, this was a clear example of the flexibility of this solution, but one must take very careful steps when creating a black list kind of policy: although this could introduce some benefits in some cases, like lighter firewalls, it could also add some subtle inconsistencies and errors, like wrongly allowing pods to start connections.

5.1.5 Service aware policies

Consider the following Polycube policy:

```
apiVersion: polycube.network/v1beta
kind: PolycubeNetworkPolicy
metadata:
  name: service-allow-api
applyTo:
    target: service
    withName: database
spec:
  ingressRules:
    rules:
      - from:
          peer: pod
          withLabels:
            role: api
        action: allow
```

By writing `service` as a `target`, Polycube will be aware of the fact that

pods have a service applied to them and will make all the necessary steps to protect the pods according to it.

Supposing that service named `database` has `80` and `443` as *targetPorts* with protocol `TCP`, all the pods that apply such service will accept connections from pods that have label `role: api`, but only on the aforementioned ports and protocol.

This serves both as a convenient method for targeting pods without specifiying the labels – `withName: database` can be seen as a clear shortcut in this case – and without specifying the ports as well.

Being *service-aware* means that firewalls will react to *Service* events, too: if, later, the cluster's needs change and only the more secure `443` port is decided to be supported, the service can be updated to reflect this change and the solution will react as well by removing the behaviour it used to apply for port `8080`.

The service-aware functionality is made for those particular use cases when a pod does not need a more advanced rule filtering, like allowing a pod on a certain port and allowing others on another one: as already mentioned, this is a convenient method for specifying all ports at once, and if such scenario is needed, it must be done by specifying `pod` as the peer instead of using `service`.

As a final note, only services with selectors are supported: services without selectors need to be selected by writing `world` as the peer.

Chapter 6

Polycube Security in K8s: Architecture

6.1 Overview

The integration of Polycube in Kubernetes brings up its flexibility and components to enable security functions and let pods communicate with each other, be it on the same node or other nodes in the cluster, or with the external world.

At the very core, a switch component is placed on the node and pods are attached to it. For security reasons, on each of such ports, a dedicated *transparent* firewall is also connected.

Pods are an ephemeral entity: they can be deployed, updated and destroyed on demand whenever the need to do so presents itself. Scalability is the simplest example that can come to mind, but even creating a pod to be used as a safe sandbox to execute unsafe commands can be a perfect example of this.

As a result, the security solution must be totally asynchronous and able to detect and react to such events in a proper way.

The *Data Plane* is in charge of enforcing and ceasing policies, as well configuring the low level firewalls to reflect the events occurred in the cluster, ensuring that policies are always respected or, in Kubernetes terms, that the current state of the cluster always matches the desired one.

The *Control Plane* knows when something occurred to policies or to the pods in the cluster by leveraging on the Kubernetes API, translating the policies in a format that the data plane can easily understand.

6.2 The Data Plane

The data plane performs many tasks: the most important ones are to inject the rules in to the low level firewalls, change their default action, set up the appropriate order of rules based on policy deployment and know how to properly react to cluster events.

It has to deal with lots of events coming from the cluster: pods may rise and die, policies may be deployed and integrate with pre-existing ones. For this reason, particular effort has been made to divide its operations in three core functions:

- *Enforce*: Given a new policy or an update of an existing one, the firewall components need to be updated by applying the rules specified in it.
- *Cease*: A policy has been removed from the cluster and its rules must be removed.
- *React*: Something happened in the cluster and the pods need to be protected in the appropriate way. This means restrict access or remove stale rules.

Such three core components will be covered later, while the following sections will describe some minor aspects that are nonetheless important to provide security functions.

6.2.1 Creating the Firewall

The creation process is not trivial: different factors must be taken into account and the use case should be analyzed thoroughly in order to choose the approach that best fits the cluster's needs.

The problem

When pods are created by the user or any other entity, they cannot yet communicate with other existing pods, as a network interface is not yet assigned to it. As a result, the pod cannot be reached by other pods or by the external world, nor can it start such communications: creating a firewall cannot be performed at this point as it makes no sense.

In this scenario the *CNI* plugin is responsible for inserting a network interface into the pod's container network namespace and making all necessary

changes on the host. Additionally it will assign an IP address to it or delegate assignment to a separate *IPAM* plugin.

When network capabilities are ready, the pod's network interface is attached to the node's switch, allowing it to communicate with any entity.

The situation cannot stay like this because it is obviously unsafe and a firewall instance is created to fix it. The flexibility that is built on Polycube allows for different approaches to be chosen for such an operation: the *Fully Managed Security* approach and the *Unmanaged Security* are both very valid solutions but cover very different scenarios and, thus, may not apply to every cluster usage.

The following sections focus on each of these approaches and provide comprehensible *whys* and *why-nots*.

6.2.2 Fully Managed Security

This approach fully entrusts the security of the cluster – including event reaction and configuration – to Polycube, and it's perfect for most scenarios, when policies that need to be used can be considered pretty standard, i.e. only restricting packets based on their fields, and no other rule elaboration needs to be performed.

Firewall creation is totally handled by Polycube and presents some non-mutually exclusive creation possibilities that depend on where the firewall must be placed:

- *Closer to the pod*
- *Closer to the node*

These approaches present their benefits and disadvantages and can be adopted to satisfy a specific use case or integrate one another, although, in the latter case, particular care must be taken when doing so, as the number of traversed network functions can potentially increase very quickly, further burdening the control and data planes and introduce non-negligible performance penalties.

Picking the solution that prevents a single firewall cube from getting too large can be considered a good rule of thumb, as deciding for a solution that does not involve creating firewalls where not needed is as well.

In this project, firewalls are placed as close as possible to the pod they are meant to protect, as this solution is the one that applies to most clusters and fits most use cases.

Firewall closer to the pod

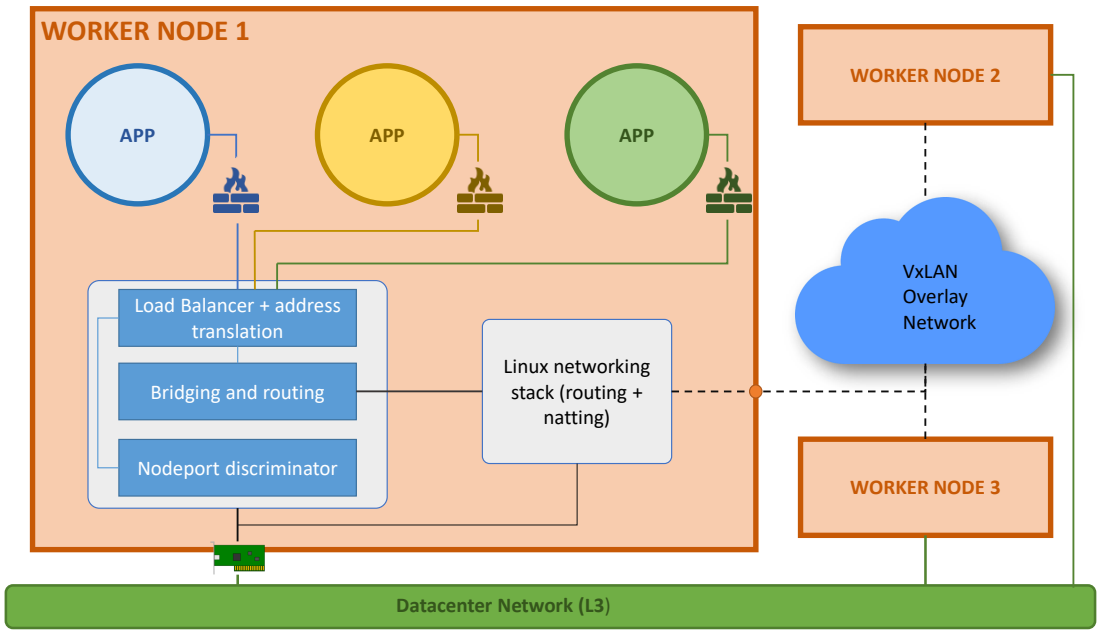

Figure 6.1. Firewalls are placed close to the pod they need to protect

As Figure 6.1 shows, in this approach each pod in the node is protected by a dedicated firewall.

Firewall closer to the pod: benefits

This approach can be labeled as a *one-firewall-per-pod* model for its specialized nature: only traffic that pertains to the pod to protect is filtered, be it traffic destined to it or generated by it.

Firewalls are not "cluttered" by something that does not relate to the pod and, as a consequence, it is very uncommon for the rules list to get too large in average-to-big clusters. Nonetheless, this occurrence may still happen, particularly in situations where policies are not created in a clever way, but this model ensures that delay is not caused because of unnecessary match lookup.

Firewall closer to the pod: a variant

This approach can be further evolved by passing from a *one-firewall-per-pod* model to a *one-firewall-per-application*: such model is presented in Figure 6.2.

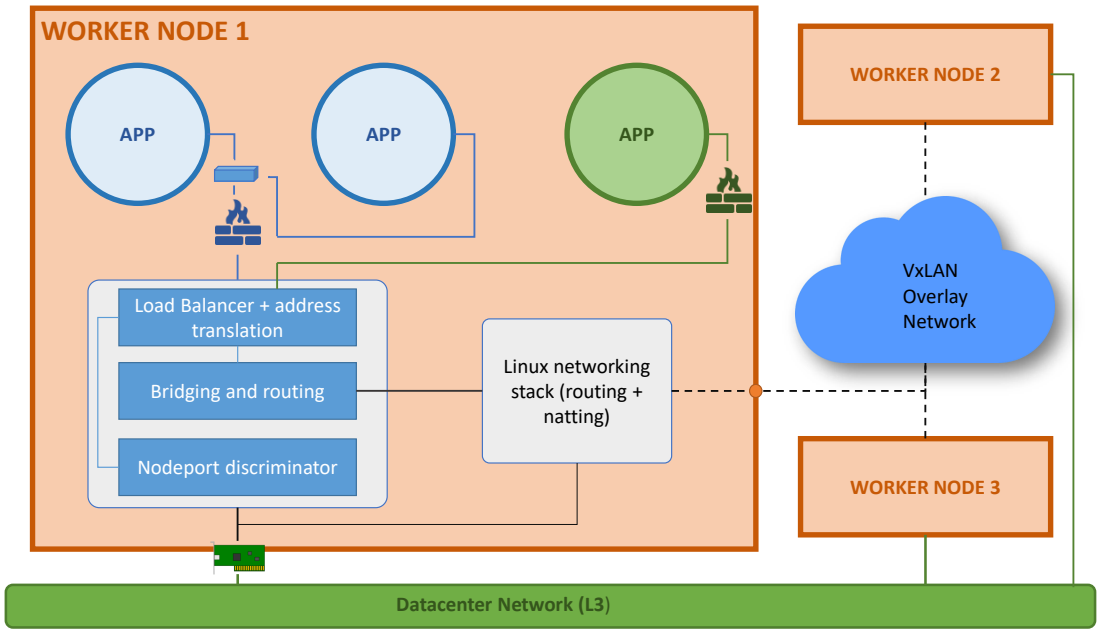

Figure 6.2. Firewalls can be created to protect instances of the same application.

As the figure shows, when multiple instances of the same applications are recognized, they are all connected to the same switch, exclusively dedicated to them, which is in turn attached to a port in the node's main switch, where the firewall is also connected.

Nothing prevents this model from working fine, but it was not the one that was decided to be implemented in this project, as it brings more complexity to the architecture and its creation process, along with additional delays in creating and traversing the appropriate network functions.

Firewall closer to the pod: caveats

Pushing firewalls close to the pods brings redundancy issues when it comes to scalability or the deployment of multiple instances of the same application in general.

As a matter of fact, one can note the presence of multiple equal firewalls as applications scale up, and this obviously leads to question oneself if they can be grouped in some way.

The increased presence of firewalls brings the additional minor problem of having small rules list, which is a direct consequence of the way policies are used: if the they are not mindfully defined and deployed, there would be a

situation where firewalls act as just simple packet forwarders and don't do any meaningful filtering.

So, a better understanding of network policies usage and, especially, the security needs of the applications, is encouraged.

The abstracted variant

Certainly, one way to solve the firewall redundancy issue can be to adopt the variant introduced earlier, but, due to its already outlined caveats, this was not the model that was actually implemented.

So, instead of solving it that way, a high level approach has been developed to mitigate this problem: the *one-firewall-per-application* model is abstracted by creating a high-level component dedicated for this purpose.

This high level component, that can be called "Firewall Manager", manages and configures all firewalls protecting instances of the same application: pods that run the same application are all "linked" to the same firewall manager.

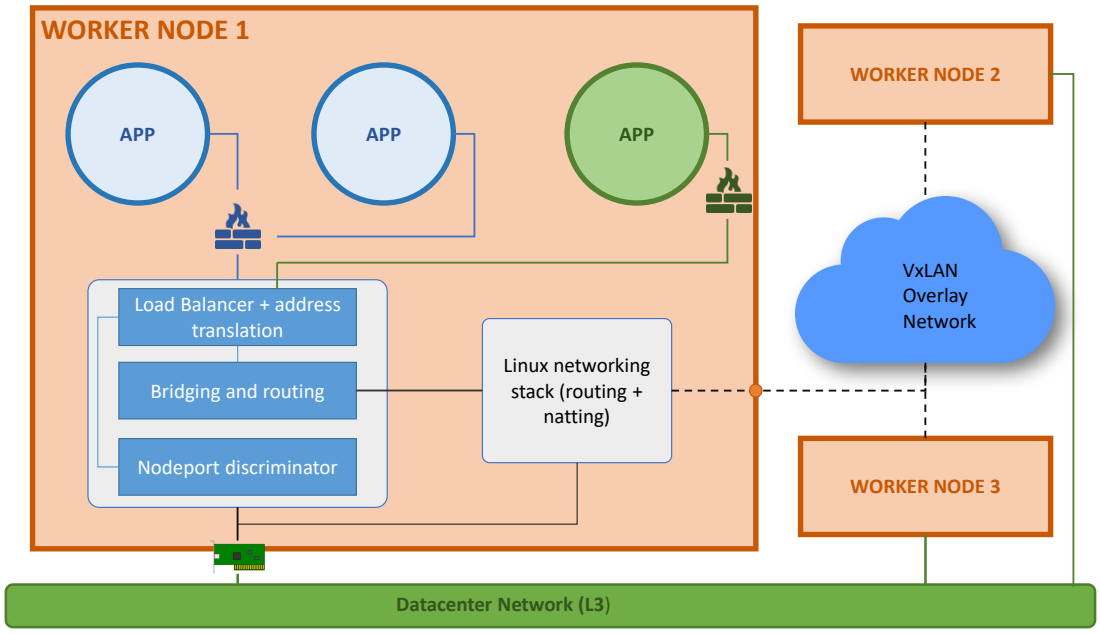

Figure 6.3. A high level component manages security for pods that are similar: simply put, pods that are instances of the same application.

As Figure 6.3 shows, at high level this *looks like* having a firewall with multiple ports that protects instances of the same application, but the low

level reality is still the one of Figure 6.1.

This model also helps for resiliency and scalability issues, which will be covered later.

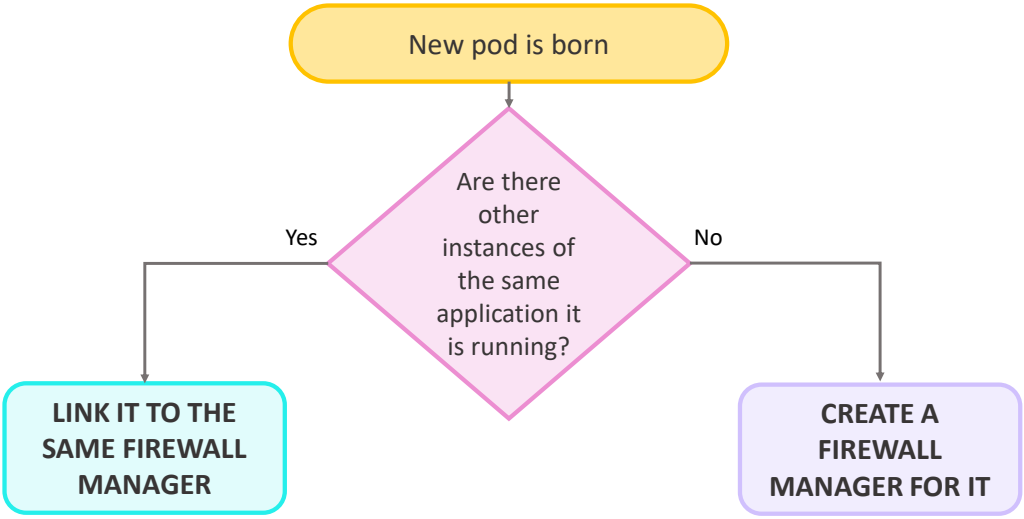

Figure 6.4. Flow chart of the firewall linking mechanism.

Finally, Figure 6.4 shows a very basic flow chart of what was just said.

6.2.3 Unmanaged Security

There are certain situations in which rules elaboration must take some "unorthodox" steps or otherwise needs to go through some additional operations with proprietary tools.

These scenarios are available by adopting the *Sidecar* pattern.

The Sidecar Pattern

This design pattern is used in micro-services architectures and consists of running a supporting application in the same container as the main one, or even in the same machine when not using containers. The purpose of the sidecar pattern is to to be able to perform operations based on certain conditions without changing/complicating the main application, or when doing so is not feasible.

Some variants exist, like the *Adapter* and the *Ambassador*, but are all based on the principle just explained.

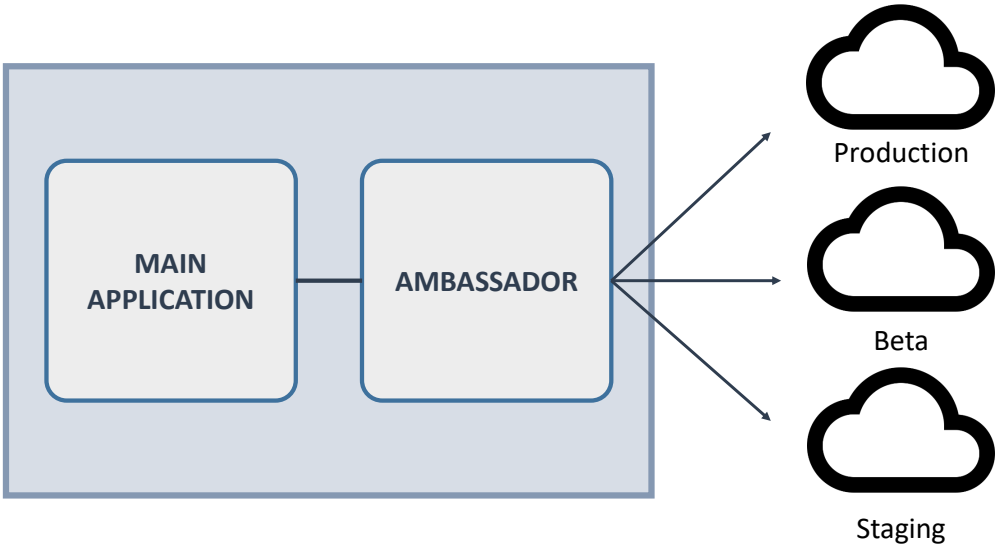

Figure 6.5. The ambassador pattern.

Figure 6.5 shows a typical example of a sidecar pattern called *Ambassador*: the main application keeps working as usual, and the sidecar application, named the *ambassador application*, is installed in the container with the purpose of re-forwarding the traffic generated by the main one to different environments based on some initial configuration set by the administrator.

Polycube can be installed as a sidecar inside the pods to take the role of the "Security Ambassador".

Polycube as the Security Ambassador

Installing Polycube in sidecar mode leaves the end-user, or any third party entity, totally in charge of the pod's security.

This method consists in adding Polycube in a Deployment's – or even a single Pod's – containers list. It can then be contacted through its REST API.

Security functions, i.e. a firewall, can be created by an agent installed inside the cluster or even remotely: they "live" inside the pod and act as totally transparent services.

As per the firewalls instances, they can still be managed internally, but this is not the purpose of this approach: the whole point of it is to be able to update and handle them remotely in a direct way. So event reaction and rules generation are the user's responsibility and policies can even be totally

ignored by using static rules.

Particular care must be taken in this case, as a non-mindful utilization of the firewalls can potentially isolate the pod and only a force-removal will solve the problem.

Nonetheless, this approach can potentially satisfy many security needs and can seriously be considered as a valid production-ready security solution: from firewalls to ddos mitigators and other security functions, it allows for the most complex, versatile and flexible solutions to secure a service.

As a final note, it is worth mentioning that although the provided method works for pods in the Kubernetes orchestrator, this approach can be extended and made to work for any existing orchestrator or any solution that involves the use of containers in general.

Figure 6.6 shows an example of such approach: firewalls protect the pods from the inside.

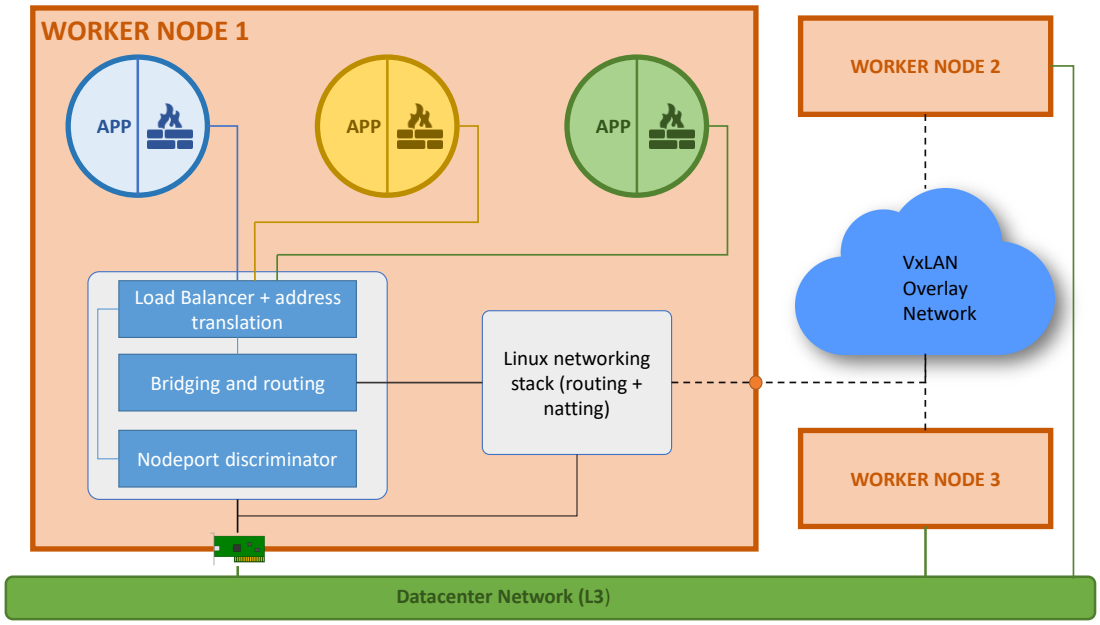

Figure 6.6. Polycube as a sidecar.

6.2.4 Compliance with Kubernetes

Polycube firewalls can work on any kind of situation, but on this project they were adapted to work as intended by Kubernetes guidelines and documentation.

Non-isolation Mode

By default, pods are non-isolated and can accept connections coming from anyone as well as establish them. This situation persists as long as no Network Policy selects them: when this occurs, the network plugin will have to make the appropriate changes to isolate the pod from unauthorized access.

Isolation mode is covered in Section 6.2.5.

Connection-oriented Filtering

Both the standard Kubernetes policies and those created by Polycube work on connections: whenever a connection is established, the peers should be able to receive packets and reply on that same connection.

A better coverage of this concept is examined in section 6.2.6.

Policy Priority

Multiple policies can be active for a given pod.

Priority in Polycube is detected during the enforcing and reacting process, and rules are built according to it.

Refer to Section 3.6.4 for an example of the priority concept and Section 6.2.6.

6.2.5 Dealing with unknown traffic

The firewall's default action affects its behavior in case the examined packet matches no rule, i.e. when the combination of peer and/or protocol is not present among its list of rules. These packets can be thought of as *unknown* traffic.

The following sections should serve as a better explanation of the non-isolation mode that was witnessed in Section 3.6.3.

At least one policy is enforced

As mentioned in Chapter 3, when there is at least one policy selecting a pod, the non-isolation mode should be ceased and the pod must become isolated.

In Polycube this situation is handled during the *Policy Enforcement* process. Isolation is applied depending on the policy type that it is intended to be enforced, but ultimately, it is done by changing the default action to *Drop*, and the firewalls will only accept what it is specifically stated in the policy rules.

No policy enforced

This situation arises when a pod starts running in a namespace with no policies or none of them selects the newly born pod, as well as when all policies have been ceased for a given pod.

Depending on the direction, non-isolation is verified: if there are no *ingress type* policies enforced, unknown traffic is forwarded to the pod; if there are no *egress type* policies, unknown traffic generated by the pod itself will be able to traverse the firewall.

Basic flow chart

Figure 6.7 is an over-simplified flow chart of what was discussed in the two previous sections, but should serve as a good summary.

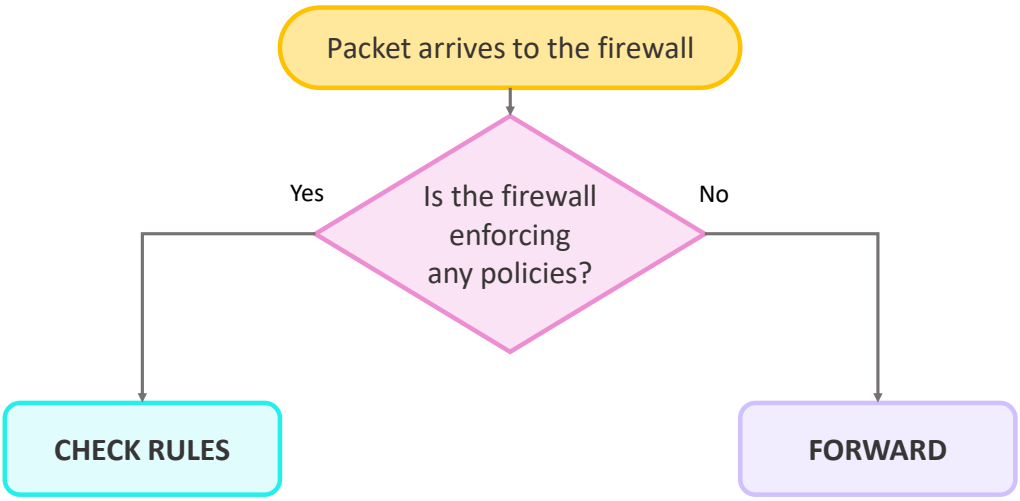

Figure 6.7. A basic flow chart of the isolation mode.

Changing the default action

The situations that will trigger a change in the default action are those that should interrupt the *non-isolation* mode defined by Kubernetes or dictate its return, and are the ones specified in the two previous sections:

- *A new policy is being enforced*: isolation will be enforced for the direction the policy is targeting, but only if this policy is the first one targeting that direction.

From now on, unknown traffic in that direction will be blocked, unless – and uselessly – the policy specifically says to accept all traffic.

- *A policy is being removed*: non isolation mode is returned for the direction the policy was targeting, but only if there are no policies targeting that direction anymore.

6.2.6 Enforce

Enforcing a policy is the act of injecting the rules in the firewall instances and apply them, so that the pod can be properly protected from any entity that is not allowed to communicate with it or, as a derived advantage, protect the rest of the cluster by preventing the pod from performing illegal/malicious requests.

The enforcing process is composed of some smaller tasks.

Activating policy actions

A policy must not be enforced just once, but every time there's a change in the current state of the cluster: security must always be brought at the desired state, and the way to do that is to know the action that must be taken when a certain condition applies.

This means evolving the enforcing process to be a bit smarter by making it aware of the fact that events can happen.

When a request to enforce a policy is made, *Policy Actions* are activated: they can be thought of as *hooks* built by the control plane, and Figure 6.8 can provide a good example, reiterating what was shown in Section 3.6: when an event, related to the *Event Actor*, occurs, the proper *Event Action* is applied.

In order not to be overwhelmed with events, only pods that actually matter, i.e. the ones specified by the policy, are listed as event actors.

Figure 6.8. A visual representations of policy actions.

The purpose is to make the enforcing and reaction components collaborate to realign the current state of the security for a given pod to the desired state.

A direct consequence and benefit of this is that events may appear while the enforcing process is still ongoing: after setting the actions, no events are going to be missed even if the process is not finished yet, as they are going to be queued and handled when it will be over.

This further increases rules consistency and updates up to the latest events.

Priority detection

Polycube policies have an explicit concept of priority, so their insertion is really trivial, as the lower their priority number is the more important they are. But in case it is not specified or for policies with the same priority number, their behavior is the same of Kubernetes Network Policies, which is explained here.

Policies belong to a sort of "time slot": most recent policies are not inserted on top by default, but according to their deployed time – or creation time – and the rules generated by them will follow the same pattern.

This is done because a policy may be updated, and the update could change the selectors and thus target pods that it wasn't targeting before: this is like deploying a "new-old" policy.

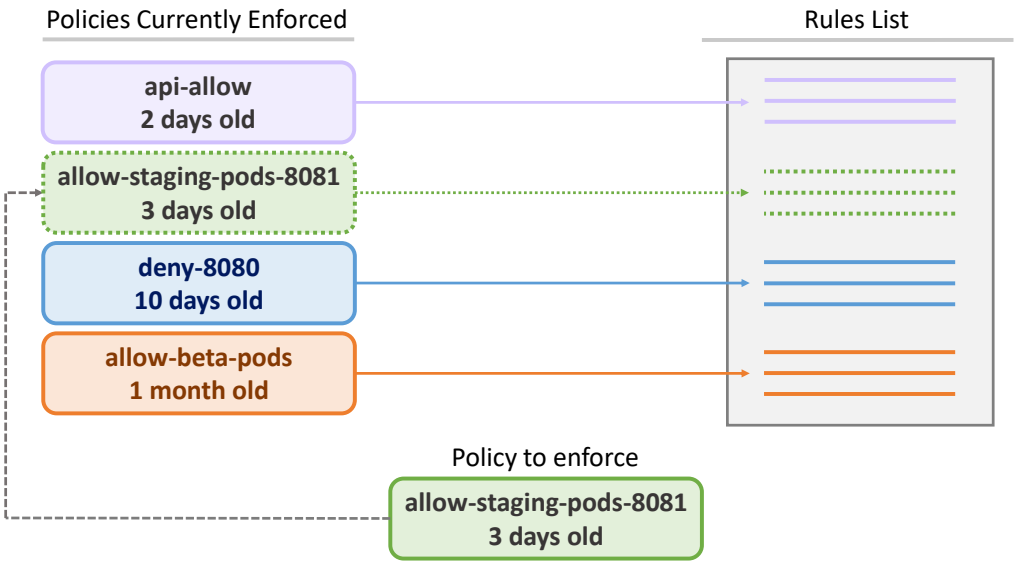

Figure 6.9. The priority detection mechanism.

As Figure 6.9 shows, policies are ordered based on their "age" – the youngest taking first positions – and the policy to enforce, *allow-staging-pods-8081*, should be placed on the second position. Its rules should behave in a similar way: they are inserted right after the last rule generated by the policy that precedes this one, all the other rules will shift downwards.

So, in the rule matching process, they will be checked right after those of the "fresher" policies, in case none of these ones match.

Rules Insertion

The last step in the enforcing process is to finally insert rules.

The firewall manager will loop through all linked firewalls and will push the rules via their API.

In order to speed up insertion, *ingress* and *egress* rules are inserted concurrently and applied independently.

Results

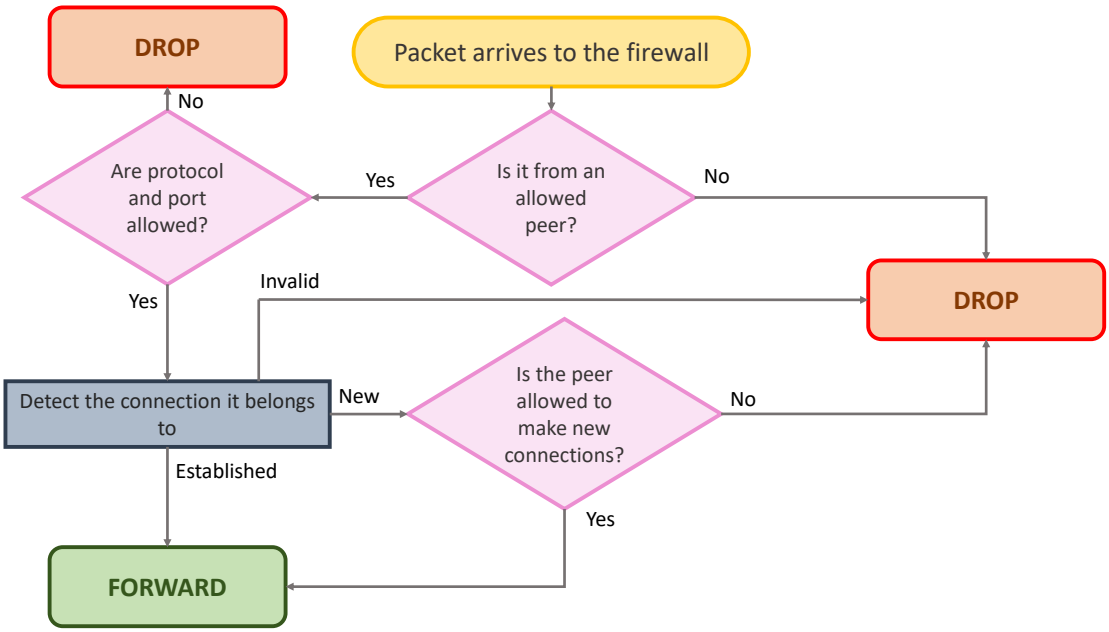

Figure 6.10. Flow chart of the behavior of a firewall when packets arrive to it after a policy has been enforced.

After the enforcing process is finished, the default action is changed – if needed – and the results can be immediately acknowledged.

Packets will now be forwarded only if they meet specific criteria defined by the policy just enforced.

For example, the Figure 6.10 contains a flow chart that can be followed for packets arriving to a generic firewall after a policy has been enforced: this is a very simplified rule matching algorithm performed by the firewall.

As it is shown, packets belonging to established connections will always be forwarded, as it implies that the connection has been accepted and follows the principle that peers should be able to communicate on the same connection. So, if the pod is allowed to accept a connection from an allowed peer, it should also be allowed to get its replies and other packets on that same connection.

No guarantees can be given as to if new connections can be made on the opposite direction – i.e. if a policy allows the pod to accept connections from a certain peer but not to instantiate one with it – but replies to accepted connections, instead, must be accepted.

Finally, connections that are explicitly prohibited do not go through the previous flow chart, as they are obviously dropped as soon as the packet matches the according rule.

6.2.7 Cease

Ceasing means removing all the rules that were generated by the policy to be removed. This does not necessary mean reverting to a situation where peers and connections that were first forbidden will now be allowed, as different operations are involved.

Remove the Policy Actions

Everything that belonged to the policy to be ceased must now be removed, including the actions that were generated during the enforcing process.

Removing the policy actions will also affect the behavior of any event waiting to be processed: as a matter of fact, they will search for the correspondent policy actions only to find no match and stop instantly.

Detect the Policy Type

At this point, the policy to be removed is examined in order to know the connections direction it was restricting.

This is done to check if non-isolation should be re-introduced or not for that direction.

Removing rules

All the rules generated by the policy are selected and removed from all the appropriate firewalls.

Results

After removing a policy, different scenarios may arise, all depending on the number of policies still actively enforced: isolation mode could be ceased or kept.

Refer to Section 6.2.5 for the details of the first situation, and Figure 6.10 for the latter.

6.2.8 On removing all policies

Ceasing policies may leave the pod unprotected from the external world or from illegal access by entities inside the cluster.

Although there is no particular issue in leaving pods with no policies, this is not a viable option for clusters that take security as a critical feature.

A very common practice, also recommended by Kubernetes, is to deploy a policy that denies all traffic – or at least the one coming from the external world or from others namespaces – as the first one for any given pod and later deploy all other ones.

This will leave no security holes as the policies are ceased, because the very first one, the one that drops all packets, will still continue to work.

6.2.9 Reacting

A Kubernetes cluster is a dynamic entity, composed by resources that may vary with time and go through different stages: the current state of the cluster, and specifically its security part in this case, will diverge numerous times from the desired state throughout its life time.

Performing the action

The reaction component can actually be seen as an extension of the enforcing or ceasing components.

The policy actions that were defined during the enforcing process will be respected: rules may be deleted in case of pods that were detected to be

dead, or, on the opposite side, they may be inserted for newly born – and allowed – pods.

On removing rules

As for dead pods, the rules that targeted them are deleted to prevent stale rules: if they were kept, some pods may later be deployed and take the IP address that earlier belonged to the one that just died.

As a consequence, the new pod may be prevented from contacting the pod, or may even be wrongly allowed to, because the firewall would wrongly think it is still the pod that died some time prior.

6.2.10 Keeping Consistency

As illustrated in Section 6.2.2, the model used in this project consists in logically "grouping" firewalls that protect instances of the same application.

A benefit of this approach is that policies need to be parsed and examined only once: they're going to be applied to all the proper firewalls and they will all have the same rules, updated up to the latest events, according to the details explained until now.

This is very convenient for new pods as well: when they're born to a node that is running other instances of the same application, all the latest rules – already present in the other instances' firewalls – will be instantly injected to the new pods' firewalls without further ado. This process is completely transparent to the control plane.

Slow Path and Fast Path

What was described thus far can be summarized in the flow chart of Figure 6.11.

The part on the left involves computation and queries to both the Kubernetes API and cache, by both the data plane and the control plane: it can be considered a "*Slow Path*", because it may introduce non-negligible delays due to the tasks it has to perform.

The part on the right is a "*Fast Path*" because another pod, running the same application as the new pod, is already present on that node: someone already did the hard work before, so the new pod will benefit from this by having everything already prepared and ready.

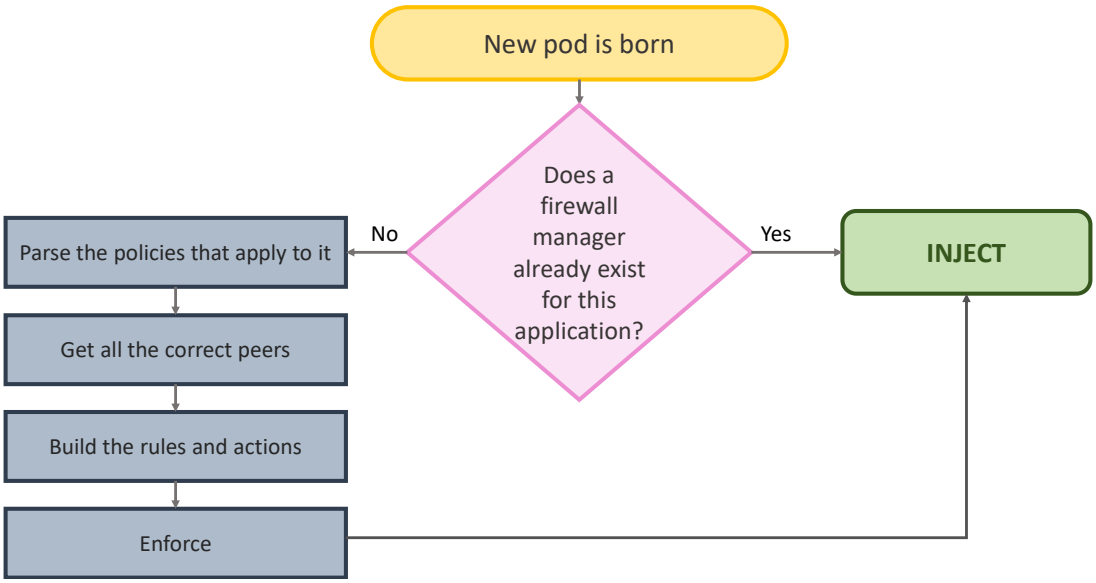

Figure 6.11. Flow chart of the protection of a pod.

This will further help scale the solution, as no further modules are going to be needed for the new instances, and no additional computation is going to be performed, since all the "heavy" load needs to be carried only once.

6.2.11 Ensuring Resilience

A direct consequence of the system just discussed is the resilience question.

When pods die and are configured to be re-instantiated, they are going to be re-deployed by Kubernetes on that same node, or another one.

This situation can be considered the same as the one in the previous Section (6.2.10): the pod that will take its place would be considered a new instance and will benefit from all the features presented thus far.

As a final note, the Firewall Manager assigned to a certain application will continue to react to events and parse policies even in case such pods all die.

This is done to wait for the pods to be re-deployed or re-scheduled to that node: if this does happen, firewalls will once again have all rules ready for them as if they always were in that node; but if it does not happen, the module will automatically die after a configurable timeout is expired, thus freeing machine resources for other tasks.

Chapter 7

The Control Plane

7.1 Overview

The control plane leverages on the Kubernetes API and structures to make the automatic security solution aware of its environment and the changes that occurs around it.

Other features include policy parsing and firewall rules generation.

7.1.1 Controllers

Controllers are a key component in Kubernetes. As their name suggests, they control the current state of the cluster and can detect when it diverges from the desired state.

In practice, this means running a non-terminating loop to continuously monitor changes. As an example, the *Replicaset Controller*, created by Kubernetes, is in charge of matching the current number of replicas of a Pod to the desired one specified in its definition, or the *Node Controller*, which monitors that status of the nodes and performs the appropriate actions when a server goes down or is added to the cluster.

Figure 7.1 provides a high level view of the main features of controllers: they specialize on a single resource type, i.e. pods, and list resources based on provided criteria or "watch" events that occurred to them. Finally, they supply such information to all modules that are interested to it.

In order to provide automatic security capabilities, a couple of controllers have been created for this project, all adopting a *Publisher/Subscriber* model.

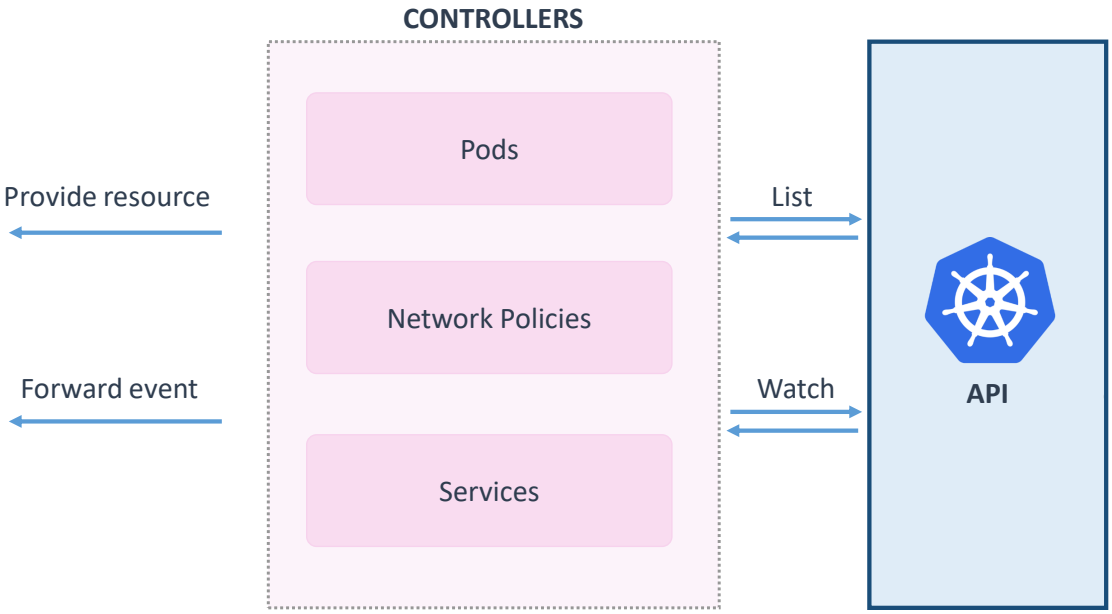

Figure 7.1. Controllers monitor the state of the cluster through Kubernetes API.

A basic flow

Figure 7.2 defines a basic flow of the way Controllers do their job: blue parts are handled by Kubernetes API and structures, red ones are specific to the single controller.

The detected event can be of three types: *Added*, *Updated* or *Deleted*. Events are processed and then inserted in a local storage, and can later be accessed by using an index assigned to them.

The first part is managed by an *Informer*, a key component of controllers, which lists resources using the Kubernetes API and caches, and watches for changes in them.

Events pass through callbacks that handle them: they can process the event immediately, or send it to a queue and let someone else do it.

Finally, the controller, through one or multiple *Workers*, will perform the needed actions with the event.

This particular flow is called *ListAndWatch*.

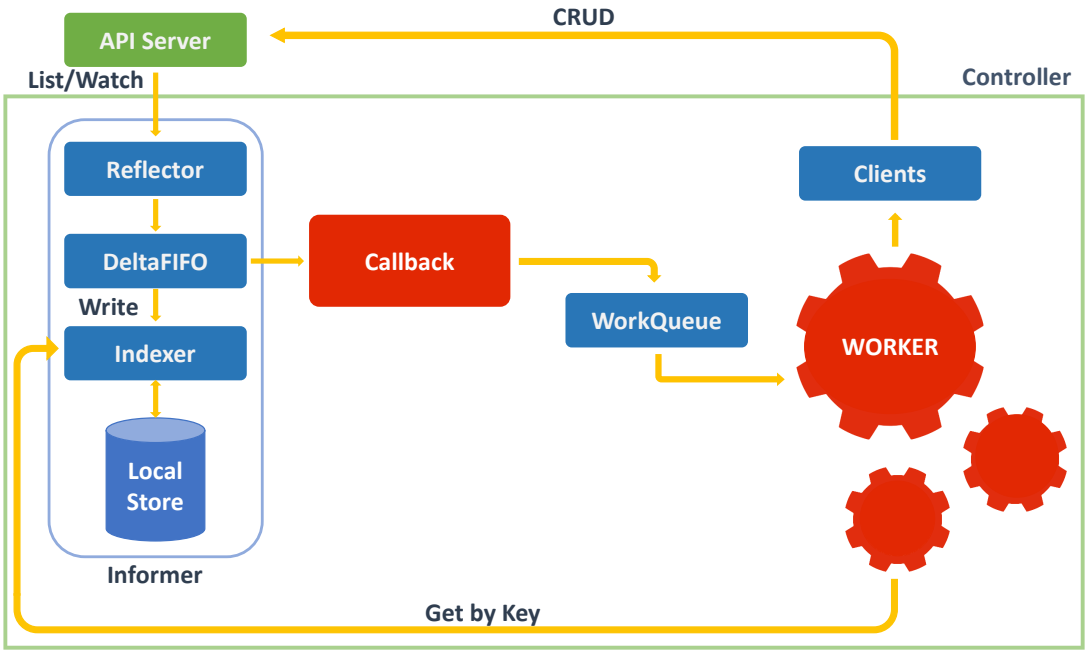

Figure 7.2. A basic flow of the functionality of controllers

7.1.2 Subscribe

Creating a controller is a particularly tedious task and brings a lot of repetitive and boilerplate code. In order to solve the problem of duplicate use of controllers and in an effort to bring more simplicity to the previous flow, during the course of this thesis a *Publisher/Subscriber* pattern has been adopted for the controllers.

In this model, entities interested in watching events about specific resources are called *Subscribers*, and they are notified about occurred events by the Controllers, which act as the *Publishers* of such events.

This model considerably simplifies the way a module can listen for changes, as it strips the complexity of creating and starting a new controller entirely and reduces overhead by making use of just one controller per resource kind.

On Figure 7.3 the subscription flow is shown: to start monitoring changes about a certain resource kind, it is enough to "subscribe" to the appropriate resource controller and specify the event type – and, in some situations, some specifics about the resource to be notified of, i.e. only if the resource is "running" – and the action to perform when such event occurs.

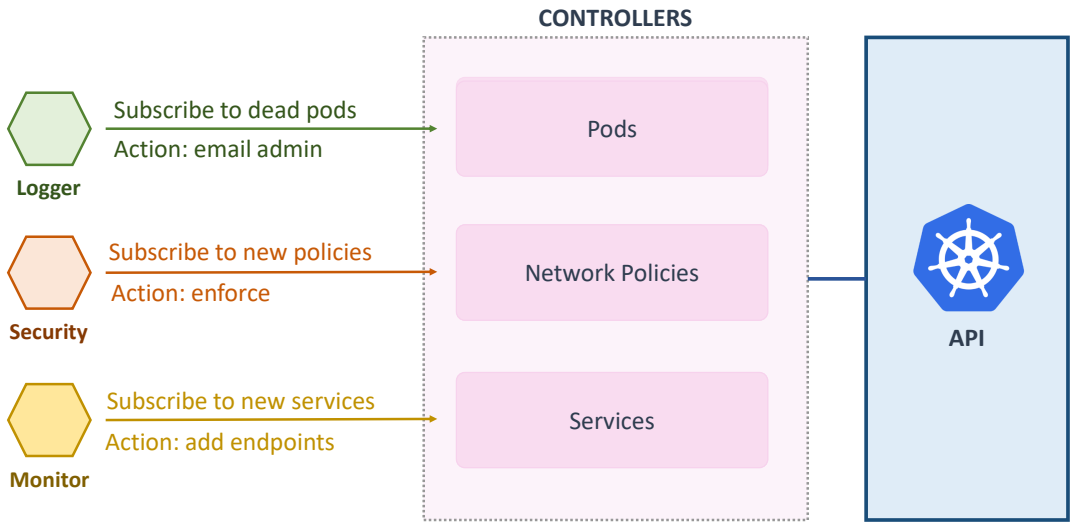

Figure 7.3. An example of the subscription model.

From that point on, every time that event is detected by the controller, the provided action is going to performed.

Caveats

The Subscription model makes watching for changes a lot easier, but, depending on the scenario, also brings with it some caveats to be aware of. Most of them are due to the very unpredictable nature of events and resources and are common to all events detection mechanisms, even those that just rely on the one provided by Kubernetes, i.e. the informer introduced in Section 7.1.1.

As the number of resources monitored by a controller grows larger, an increase in events and, consequently, of threads started by the controller can also be noticed: actions to perform should be designed to perform as little computation as possible if such situations are predicted to happen. Although the issue can be mitigated by using multiple workers in the controller, it is still something to take into consideration.

As a final note, the previous situation may also happen in case of unstable resources: as a matter of fact, such resources continually crash and redeploy, thus launching several events that are going to be uselessly processed.

7.1.3 Queries

Controllers can watch and list resources of a particular kind. In order to provide a way for modules to get resources which only satisfy certain criteria, a very simple querying system has been developed, further stripping down the complexity of learning and knowing how to use the resource's specific structures to do so.

Figure 7.4 shows an example of the querying mechanism: a logger that wants to get all pods that are on the `default` namespace and have at least the label `role: api`.

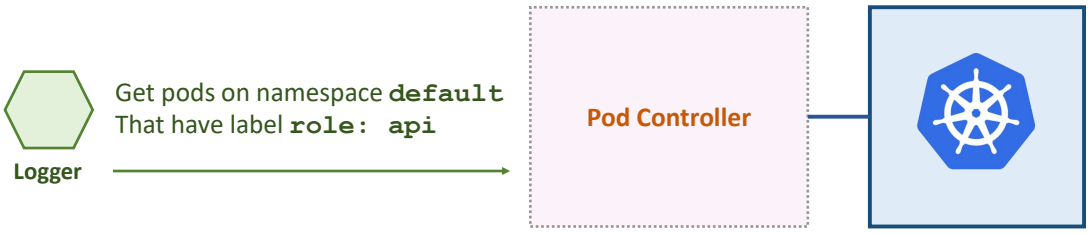

Figure 7.4. An example of the query system.

7.2 From Policy to Firewall

The control plane detects when changes to the state of the security in the cluster occurred and activates all the necessary procedures in order to maintain the desired state, as specified by the cluster administrator.

Events concerning pods are more relevant to the data plane, so that it knows if a firewall should be created, or existing ones should be updated to reflect the event. Refer to Chapter 6 for such details.

Changes related to policies are more interesting to the control plane: proper configuration and rules must be generated in order for the data plane to know what to do.

7.2.1 Policy events

Naturally, the most important event is the deployment of a new policy, as it – more than the other – is a clear and "radical" change of the security needs of the cluster.

This particular event involves lots of operations that are briefly covered below.

The pod and the namespace

The first, and arguably most important, step, is to know the "target" of the policy, i.e. the kind of pods the policy is intended for.

In order to get it, one needs to get the namespace of the policy, which also specifies where the target pod must be found and its labels.

Once done, the pods could be found easily by querying the controllers, and the protection process could begin, although this is not the mechanism that was developed in this project: as a matter of fact, this step may require some serious computation needs and involve many calls to the Kubernetes API to get the needed resources.

Another approach, slightly different, has been adopted: as mentioned in Section 6.2.2, there is a high-level component in charge of protecting instances of the same application and assigned exclusively to them.

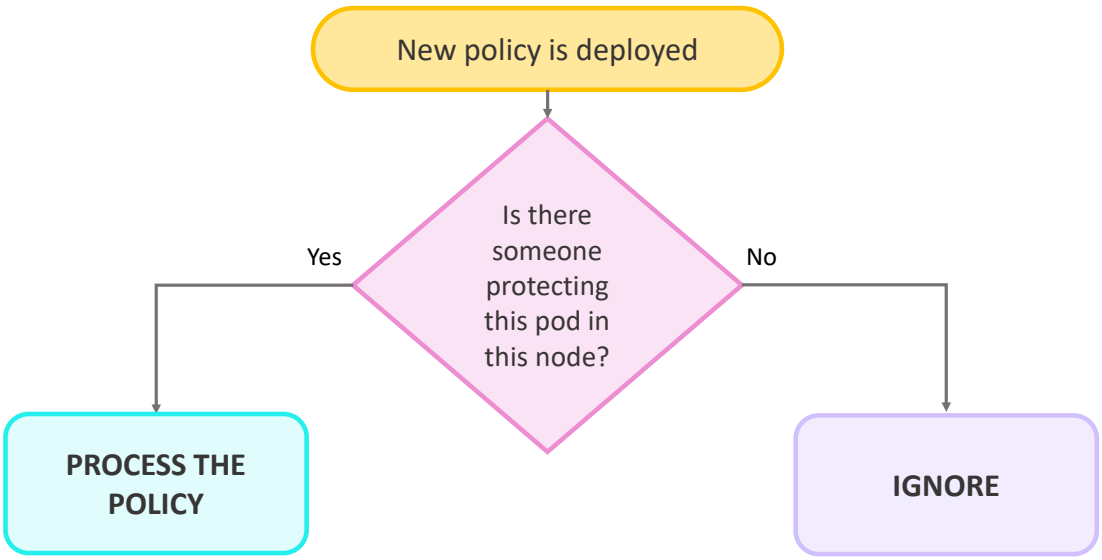

Figure 7.5. Policies are processed only if there is someone they apply to on the *local node*.

As Figure 7.5 shows, the control plane knows the list of such components that are currently active in its node, and rather than starting the process

of protecting the appropriate pods, it checks if this component exists: obviously, protecting something that is not there does not make any sense, and preemptively doing so is a waste of resources because there is no way to know if such pods will actually ever be scheduled to the node.

So, in case the aforementioned condition is not verified, the policy will simply be ignored and the job is "postponed" to when the pod will actually be scheduled on the node. The situation of Figure 6.11 will occur in this case.

As it is going to be explained later, getting a list of pods is a potentially expensive operation, and since this method is entirely local and does not involve searching for pods, it makes this solution more scalable.

Policy type detection

The policy type is closely examined, so that the data plane could later correctly configure isolation mode for the correct direction.

For a more detailed coverage of isolation mode, refer to Section 6.2.5.

Protocols parsing

Protocols and ports are parsed before any other field, i.e. peers selectors.

The reason for this is mainly to detect if the protocol is effectively supported by the firewall or not: knowing this information *a priori* will prevent the control plane to waste time by searching for pods if the protocol is not supported. Specifically, as stated in Section 3.2.7, Kubernetes Network Policies also support `SCTP`, which, for the reasons also specified there, is not recognized by the Polycube firewall.

Parsing and loading the allowed peers is an expensive operation, and it must be avoided if the protocols only involve unsupported ones.

Just to make an example with Kubernetes Network Policies, the following policy will only generate rules for `TCP`, ignoring `SCTP` entirely: packets that travel with `SCTP` will be dropped. In case only `SCTP` is specified, no rule will be generated at all.

```
ports:
- protocol: TCP
  port: 6379
- protocol: SCTP
  port: 6379
```

Peers detection

The other important part is detecting the peers that are allowed to communicate with the target, or the other way around.

This involves querying the API to get potentially lots of pods: in case of large clusters and not very specific selectors, a great number of pods are going to be searched – and, consequently, returned – by the controllers, and for this reason, the Kubernetes cache is used as much as possible.

Rules are generated for each peer found without any information about the ports and protocols: when the process is over, they are going to be merged together, and new rules are generated for each port and protocol.

As a final note, as a consequence of the peer searching, `Ingress` and `Egress` directions are parsed concurrently in case a pod targets them both. The picture below can be thought of as a summary of what has been presented until now: outline has been given to the fact that the pod controller is used to get the peers.

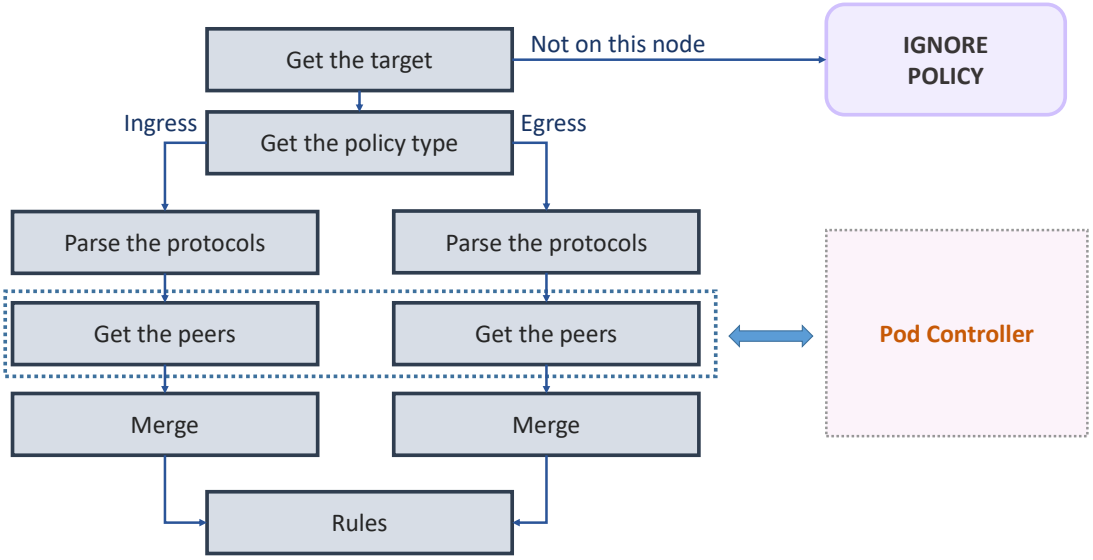

Figure 7.6. The policy processing flow chart.

Policy actions

Lastly, *templates*, or policy actions, are generated based on the rules that have been defined until now: these are going to be used by the data plane when events occur, so the peer can be correctly filtered when it occurs.

To make a very simple example, consider the following rule:

```
- from:
  - podSelector:
      matchLabels:
        role: api
  ports:
  - protocol: TCP
    port: 8080
```

This will generate rule templates with the provided protocols and ports only: this is the *action*. The *actor*, which is the pod that generated the event and that must trigger this action, is called `default|role: api`, which is a combination of namespace name and pod labels.

The data plane will incorporate such information in its enforcing process, as specified in Section 6.2.6.

Chapter 8

Evaluation

This last chapter will analyze the network throughput of *Polycube* with the automatic security solution activated and will compare the results to those of *Calico* and *Cilium*.

The tests are performed by setting the three plugins in the same environment and progressively increment the number of policies they have to enforce. The environment consists of two nodes, on which three pods, running the *iperf* tool, are deployed: one will act as a server, and the other two, which will send requests to the server one, will act as clients.

The following sections will examine the network performance between two pods and all the results are gathered considering the same packet size and same transport layer protocol.

8.1 On the same node

As Figure 8.1 shows, as the number of policies and, consequently, number of rules increase, all plugins have consistent performance when pods that are on the same node communicate with each other, with *Polycube* and *Cilium* falling closely behind *Calico*.

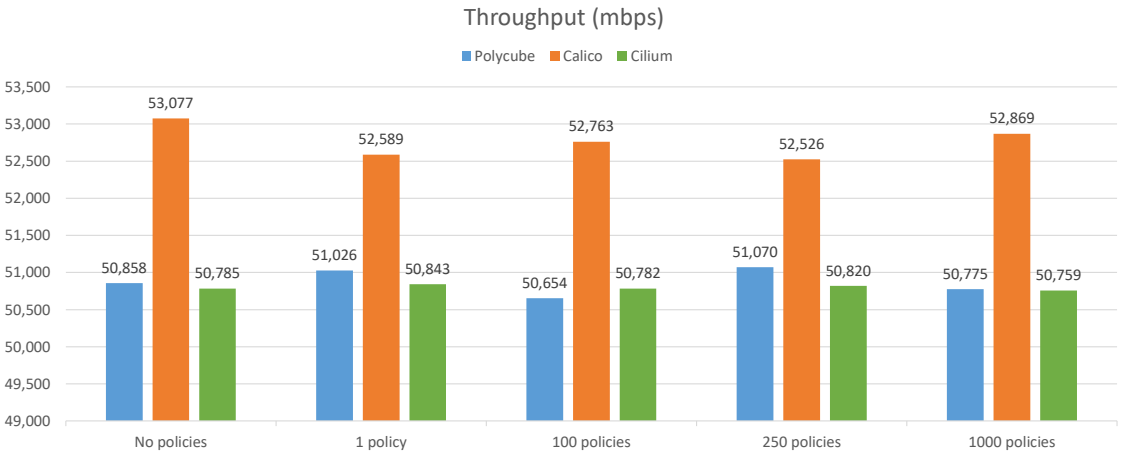

Figure 8.1. Pod to pod communication performance on the same node.

8.2 On different nodes

The test involving pods running on different nodes proves that *eBPF*-based solutions, such as *Cilium* and *Polycube*, to perform better than those utilizing *iptables*, such as *Calico*. The former two show, once again, very similar performance.

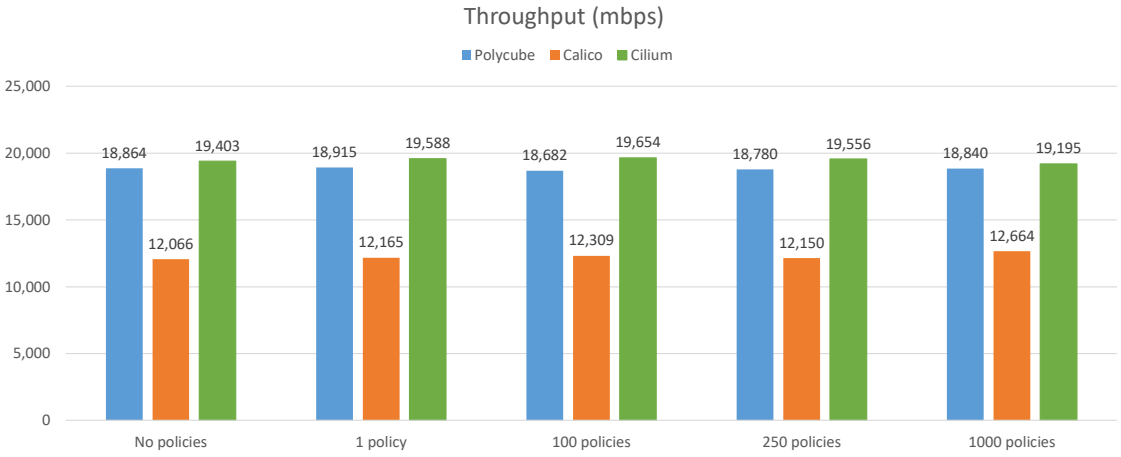

Figure 8.2. Pod to pod communication performance on different nodes.

Bibliography

[1] The Kubernetes Documentation,
https://kubernetes.io/docs/home/

[2] Polycube,
https://github.com/polycube-network/polycube

[3] Kubernetes 101: Pods, Nodes, Containers, and Clusters,
https://medium.com/google-cloud/kubernetes-101-pods-nodes-containers-and-clusters-c1509e409e16

[4] The New Stack,
https://thenewstack.io/kubernetes-an-overview/

[5] Kubernetes Network Policies Recipes,
https://github.com/ahmetb/kubernetes-network-policy-recipes

[6] An Introduction to Network Policies for Security People,
https://medium.com/@reuvenharrison/an-introduction-to-kubernetes-network-policies-for-security-people-ba92dd4c809d

[7] Securing Cluster Networking with Network Policies (KubeCon Talk),
https://www.youtube.com/watch?v=3gGpMmYeEO8

[8] A Deep Dive into Kubernetes Controllers,
https://engineering.bitnami.com/articles/a-deep-dive-into-kubernetes-controllers.html

[9] Understanding Kubernetes Networking Model,
https://sookocheff.com/post/kubernetes/understanding-kubernetes-networking-model/

[10] Writing Kubernetes Custom Controllers,
https://medium.com/@cloudark/kubernetes-custom-controllers-b6c7d0668fdf

[11] Multi Container Pod Design Patterns,
https://matthewpalmer.net/kubernetes-app-developer/articles/multi-container-pod-design-patterns.html

[12] Wireshark,

https://www.wireshark.org/
[13] Resilience,
https://en.wikipedia.org/wiki/Resilience_(network)
[14] Golang,
https://golang.org/doc/
[15] Docker,
https://docs.docker.com/
[16] Cilium,
https://cilium.readthedocs.io/en/stable/policy/
[17] ISTIO,
https://istio.io/docs/
[18] Calico,
https://docs.projectcalico.org/v3.8/security/calico-network-policy

Acknowledgements

It was an incredible experience. An amazing journey. I learned a lot, more than I could ever think and hope for.

First and foremost, I'd like to thank all my beloved family for their endless love, support and advice. I'm pretty sure you won't understand a thing of what I've written here and I know I'll have to translate everything for you, but I also know that all of you are worth the effort. I know it sounds cliché because everyone writes this in their thesis, but I really don't know how I could have done this without you.

Another big thank you to my professor Fulvio Risso for giving me the chance to work on this. When I started, I knew like five percent of what I was going to do. Now I don't think I have pushed it much further than seventy, but that's another thing I have to thank you for: the opportunity to keep learning and, consequently, fall in love with this field.

Another round of thanks to Mauricio and Matteo for the help they gave during this thesis. By giving me obvious replies to my endless nonsensical questions, you soon realized how unbearable it was to work with me. Thanks for your help and patience.

Finally, I want to thank all those who were with me throughout the years: cousins, friends, the ones that got away and those that, instead, decided to stay. Your contribution to this is zero, but you provided me with distractions and allowed me to unwind a bit, and I have to thank you for this, even if it was just for a short time.

Thank you,

Elis