KU LEUVEN

FACULTY OF SCIENCE

Conflict detection of network security policies across the Kubernetes stack with incremental approach

Jasper GORIS

Promotor: Prof. dr. ir. W. Joosen
DistriNet

Co-promotor: *Dr. E. Truyen*
DistriNet

Co-promotor: *Prof. dr. T. Mühlberg*
DistriNet

Mentor: *G. Budigiri*
DistriNet

Thesis submitted to the obtaining the degree of Master of Science in Applied Computer Science

Academic Year 2023-2024

Preface

Summary

Samenvatting

Abstract

List of Figures

List of Tables

Table of Contents

1. Introduction

1.1 Context

Application software is everywhere: ranging from social media apps to online web-shops, login systems, healthcare systems or even photoshop. The usage of these applications is irreversibly integrated with our way of living, powered by wireless internet connections, personal computers, mobile devices and many more technologies to access them whenever and wherever we want. It comes as no surprise that the demands of these systems has therefore increased over time which introduced entire new ranges of problems. One of these problems is how to cost-effectively scale the computational resources in order to handle a fluctuating amount of end-users. For example, a web-shop might have 20 times more clients during December due to the holiday season than any other month of the year, and as a result the servers are not being used to their capacity for 11 months a year.

One of the solutions to handle this increased and fluctuating demand is the trend towards cloud native computing: the practice of building and running dynamically scaling applications in cloud environments **?**. This is not unexpected, since cloud native computing offers advantages such as scalability, resource efficiency and security **?**. Although cloud native computing is not a new phenomenon it seems to only grow in popularity **?** and is rapidly replacing old legacy systems and in-house servers. One of the key components to enable this growth are container formats built upon the specifications of the Open Container Initiative OCI **?**, such as Containerd. These containers are a source of application portability by including the application code and all the dependencies required to run the code. This means they can easily be run in any environment by container engines like rkt **?**,CRI-O **?**, LXC **?** and Docker **?**. Benefits of these containers include the possibility to easily change geographic location to ensure faster user connections and having multiple deployments of the same container in order to serve more users by spreading workload.

However, new problems arise when demand increases and clusters of sometimes even thousands of containerised applications need to be created to meet those demands. This is where orchestration solutions such as Kubernetes (K8s) **?**, Mesos **?** and OpenStack **?** come into play. They allow automatic handling of deployment, load balancing of users, scaling of resources and containers, defining security practices and monitoring of the applications, all based on settings defined by the cluster manager.

K8s is currently the de facto standard container orchestration tool in the cloud community according to the 2022 Cloud Native Computing Foundation annual survey. An excerpt of this data is shown in Table 1.1 **?**. A Kubernetes cluster exists of many different components where the highest level components are called nodes. Nodes are physical or virtual machines which can be responsible for managing the cluster, running one or more pods, or both. Pods are the lowest level component in Kubernetes and specify how one or more containerised applications should be run. Pods give their containerised applications access to networking resources and shared storage and allow for duplication of these containerised applications to increase workload. To ensure the security of clusters Kubernetes has different solutions for different attack surfaces. One of these are the network policies that define rules for traffic flow to and from the IP address of a pod. These network policies work based on key-value labels to select the pods they are applied to and are intended to restrict free communication between pods **?**.

User Type	Using K8s in Production (%)	Using K8s in Piloting/Evaluating (%)
End Users	64	25
Non-End Users	49	20

Table 1.1: Kubernetes Usage Data **?**

"CNCF End Users are member companies that utilize cloud native technologies internally, refrain from selling any cloud native services externally, and do not fall under the categories of vendors, consultancies, training partners, or telecommunications companies. Individuals within these end user companies exhibit a fervor for problem-solving through cloud native architectures and strive to provide teams with self-service solutions, fostering a more inclusive and iterative process." **?**

1.2 Problem

Kubernetes talks about the 4 C's of cloud native security on their website as seen in Figure 1.1: The code, container, cluster and cloud layer **?**. The code and container layer can be secured with defensive programming guidelines **?** and specifications such as those of OCI **?**. However they cannot safeguard against misconfiguration of the cluster and cloud layer. Imagine a container on the cluster being compromised by a malicious attacker. In the worst case this scenario can lead to an entire cluster of infected pods and containers due to unrestricted communication. Luckily there are security principles available such as network policies in K8s and security groups in Openstack. These two examples are present in the cluster and cloud layer respectively. **? ?**.

Kubernetes network policies allow to define whether or not communication between containers are allowed. Communication between inter-dependant containers should be allowed while unrelated containers should be separated to prevent the spread of malicious attacks and to prevent network congestion. Keeping track of all the existing network policies and ensuring correct network connectivity can become a complicated and daunting task due to the dynamic nature of a K8s cluster with containers scaling up to the hundreds of thousands. On top of this conflicts between security rules in the cloud and cluster layer can arise due to misconfiguration and a conflict of interests. We will illustrate this with an example.

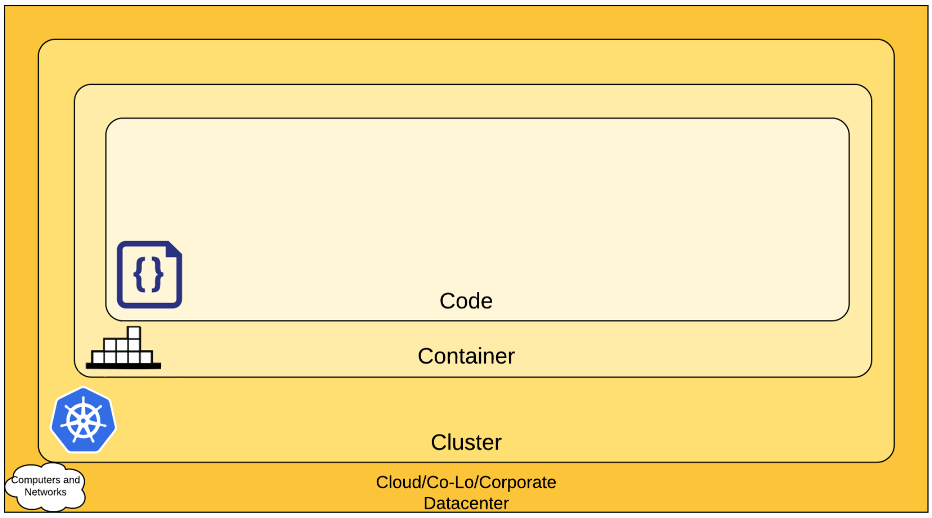


Figure 1.1: The 4C's of Cloud Native security (taken from source **?**)

Imagine a cluster with two deployed containers: a web application and a database upon which the web application depends for the data that will need to be displayed. The application manager responsible for the web application knows of this dependency on the database and defines the correct network policies in the cluster layer so that the containers can communicate. What the application manager does not know however is that the person responsible for the VMs in the cloud layer, who we will call the cloud manager, works according to the principle of least-privilege **?**. As a result the VMs on which the containers are deployed are prevented from communicating by their networking rules, since they had no reason to communicate before the deployments. We now have a conflict between the cluster and cloud layer which results in an unreachable database container and a non-functioning web application.

These types of conflict can easily be overlooked and need to be verified often due to the constantly changing state of a cluster. Additionally, keeping the principle of least privilege enforced while allowing necessary communication gets more complicated as the size of the cluster grows as well. At the moment of writing and to the best of our knowledge there is currently no solution available that detects conflicts in configuration between network rules in the cloud and cluster layer.

1.3 State-of-the-art

When looking for solutions for conflict detection of network security rules that include both the cloud and cluster layers we found that existing research solutions often came close,but were always missing at least one essential part. We will briefly describe some of these existing solutions that came closest to the desired properties of conflict detection

Grashopper ? aims to solve the same problem of misconfigurations between the cloud and cluster layer, but does not use conflict detection. Instead it will generate the security groups of the cloud layer based on the network policies in the cluster layer. conflicts are thus prevented instead of detected. However it starts with the assumption that network policies are always correct, called the base truth. We differ from this approach by not having a base truth since the two different layers are managed by separate people and/or instances and their priorities might not align. Since we do not assume a base truth we can not offer a resolution step in this thesis: we can not decide whether the the cloud layer is incorrect or the cluster layer, only whether or not they are misaligned.

Kano ? detects inconsistencies in network policies to ensure no redundancy or conflicts exist between them. To achieve this Kano generates a square matrix of size kxk where $k = amount\ of\ containers\ in\ the\ cluster$ where an 1 in position [i][j] means that the pod with index [i] can communicate towards the pod with index [j] respectively . With this matrix as a baseline it detects various possible misconfigurations due to network policies. However it does not look at the cluster layer for conflicts and is therefore not extensive enough in its approach. Still, the methods and practices described in the Kano paper, such as the generation of this matrix are a solid base on which we will build upon. We did however find a drawback to the generation method of the matrix: the kanomatrix needs to be fully regenerated after every change in cluster status that might affect it, such as adding or removing a pod/container or a network policy. With the changing nature of a cluster an incremental approach of updating the kanomatrix instead of fully regenerating it might be more beneficial and could possibly mean an increase in efficiency. For a more in-depth description of Kano we refer to section 2.4.

NFVGuard ? is the first solution we found that does multilevel security verification but applies this to the NFV stack. NFV stands for Network functions virtualization and is a way to replace proprietary hardware for network services with standard hardware with virtualized components. In the 4C's of Kubernetes security the NFV stack would be placed within the cloud layer, and thus does not provide conflict detection between cluster and cloud layer. However, some principles can be taken from the NFVGuard approach, such as the collection of relevant security data across the different layers of a stack before verifying their properties. Less interesting for us is how NFVGuard turns the collected data into First Order Logic (FOL) properties after which they use existing Constraint Satisfaction Problem (CSP) solvers. This approach makes sense in the very differing layers and data sources of the NFV stack, but would introduce too much overhead in our own solution. The different network security rules between the cluster and cloud layer are more straightforward in their correlation and promise to allow a more direct comparison without translation to FOL.

TenantGuard ? verifies network isolation between different tenant in the same cloud environment. Their solution promises to be scalable by using an incremental approach to keep the computation time low. However, it does not provide isolation between the cluster and cloud layer, but only within the latter of the two. Fortunately, Tenantguard does use some techniques that will be useful in our own implementation such as the use of tree-based data structures for quick data retrieval, and most of all an incremental approach. Tenantguard identifies all events on the cloud that influence isolation and will trigger an update of its conflict evaluation based on these events. It will only look at the parts of the isolation that are influenced by this specific event instead of recalculating the conflicts for the entire cluster, decreasing the computation time. In this thesis we will use a similar event-driven incremental approach.

Microsegmentation tools leverage the principle of least privilege **?** and zero trust **?** to prevent conflicts from existing, effectively making conflict detection redundant. Some noteable examples of microsegmentation tools are Paloalto's PrismaCloud **?**, Illumio core **?**, cisco ACI **?** and VMWare NSX **?**. The goal of these tools is to limit the possibility of a malicious attack spreading trough a cloud stack by hindering lateral movement. To do this it leverages network rules that limit connections to the minimum required for normal operation. Although this does prevent conflicts within these network rules an essential part is missing in all these microsegmentation solutions: they do not orchestrate these network rules across different layers in the cloud stack. Therefore conflicts can still exist between the cloud and cluster layer, even when these tools are correctly applied

1.4 Goals

As mentioned in the previous section there is a gap in current state-of-the-art technologies for detection of conflicts in network rules through the cloud and cluster layer. Our goal is to fill this gap with an algorithm that incrementally verifies network policies and pods in a container cluster, after which we detect conflicts between network policies and pod scheduling on the one hand and security groups and their security rules on the other hand. With an incrementally verifying algorithm we refer to an algorithm that, upon each event in the cluster that might induce new conflicts or resolve existing conflicts due to a chance in the cluster layer, triggers a new verification. Events in the cloud layer might affect conflicts as well but are out of scope for this thesis. Additionally, the algorithm should detect any existing conflicts in the cluster state upon execution to ascertain whether or not the current state is conflict-free.

1.5 Approach

The approach of this thesis is split up into the following steps:

- A literature study of the current technologies around container orchestration, cluster security and conflict detection.
- Defining a problem statement and scope for the thesis
- researching possible solution approaches to the problem statement and creating a first pseudo-code algorithm. After this we research capable data structures and coding practices to enable the creation of the algorithm.

- Implementation of the algorithm using an iterative approach.
- evaluating the final algorithm with the following research questions:
  - What is the difference in time cost between an incremental update of the Kano matrix compared to newly generating the Kano matrix for every event and how does this difference scale with pod/policy numbers
  - What is the difference in space cost between an incremental update of the Kano matrix compared to newly generating the Kano matrix for every event and how does this difference scale with pod/policy numbers
  - What is the relationship between pod/policy numbers and the time cost of conflict detection?
  - What is the relationship between pod/policy numbers and the space cost of conflict detection?

1.6 Text overview

We will now describe how the remainder of the chapters is divided. In chapter 2 we will give background information about technologies that will be used within this thesis. We continue with chapter 3 which will describe our solution for the conflict detection in depth with the help of pseudo-code algorithms of our implementation. chapter 4 is where we will evaluate our algorithm based on the four earlier mentioned research questions. Lastly chapter 5 will provide a conclusion to this thesis as well as some self-reflection.

2. Background

This chapter aims to give some background information about concepts and technologies related to the thesis. We will start talking about Containers and their advantages, after which we talk about how to run these containers using a container engine, specifically Docker Engine. The third section will talk about the container orchestration software Kubernetes and it's network policies. After this the focus shifts to Kano: a system to cover container network policy verification. Lastly we will talk about the cloud operating system Openstack and its network security solutions.

2.1 Containers

Deploying an application on a cloud computing infrastructure or platform is not an easy task: Even though the application might work perfectly when locally developed and tested it is not guaranteed to work in a cloud environment due to possible differences in Operating System (OS) or hardware settings. Luckily this problem can easily be solved by creating a virtual machine (VM) with the same OS and hardware settings as the local developing environment, which is called a hypervisor deployment **?**. However, such a hypervisor deployment also has its drawbacks: A VM often has more functionality than the application strictly requires, such as access control and user management. This in turn decreases scalability since the VM files increase the size of deployments, causing slower deployment, deletion and rebooting times.

A solution for these limitations is using containers instead. with containers, applications share an operating system with each other when deployed on the same host server, meaning a decrease in size compared to a fully fledged VM environment for each application. Additionally, containers allow quicker deployment, deletion and rebooting then the hypervisor alternative since they offer a limited number of services. Containers are therefore more flexible in scalability according to the elasticity of demands. these differences between hypervisor and container deployments are shown in Figure 2.1.

But this is not the end of the story, since without an industry standard around containers they would not be deployable on any hosts, nor be portable and scalable. To support this effort of creating industry standards the Open Container Initiative (OCI) **?** was introduced by the Linux Foundation **?**. OCI dictates in 3 specifications how to standardize your application so that it can be deployed in a container **?**. The first step in achieving containerized deployment is packaging the application into an image format. if this image is made according to the OCI image format specifications it will define all necessary information for the launch of the application on a specified platform, such as environment variables and application arguments. To facility the sharing of these images public and private registries can be utilised, which follow the OCI distribution specifications.

Almost all mainstream players in the industry support and follow these OCI guidelines, effectively creating a strong standard in the industry with many benefits as result. For example, outsourcing cloud management to public providers enables lower cost due to economy of scale, where the location cost of the servers is divided between all the user that run their applications on these servers. If you want to use these public providers you often only need to create and share your image. Changing providers for your deployment is also fairly easy thanks to this industry standardization, making for a competitive market. Furthermore the portability of container images allow for quick deployments, compared to hypervisor deployments which require more manual configuration.

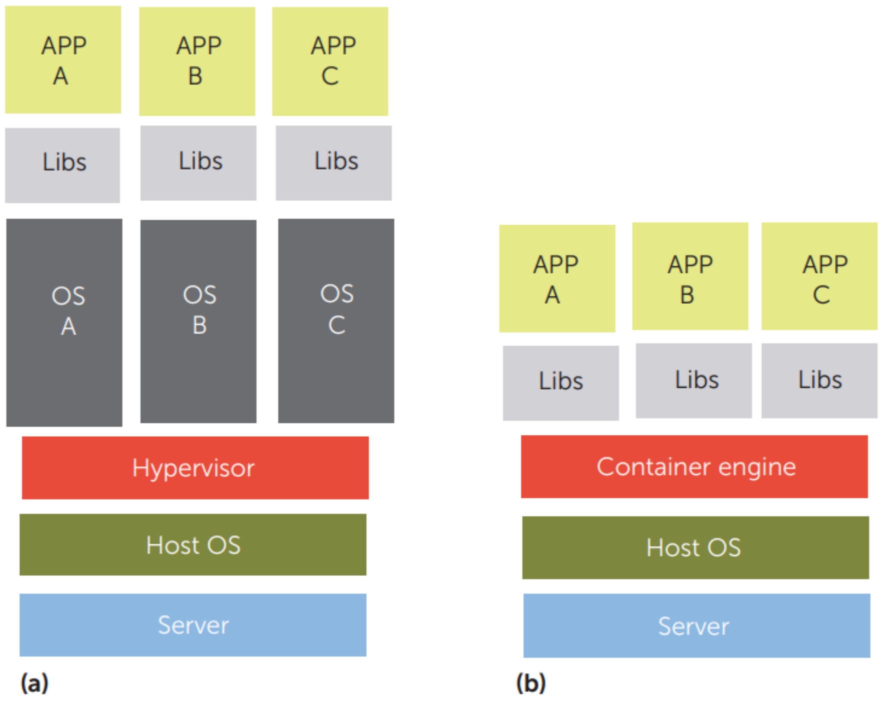


Figure 2.1: Comparison of (a) hypervisor and (b) container-based deployments (taken from source **?**)

2.2 Docker Engine

Once an OCI standard image is created for an application and is stored in a registry it still needs to be deployed as a container. We previously talked about cloud providers, but even they need a way to turn the image into a running container. This is where container engines such as Docker Engine come into play **?**. Container engines are the bridge between the end-user and the deployment of a container based of an image: they accept user requests and pull the images from the registry to finally run the container based on the metadata in the image. It also offers an API so the engine can be called by a higher abstraction layer such as Kubernetes (see section 2.3).

Container engines such as Docker engine not only run containers, but also manage dependencies and allow multiple containers to run on the same OS by using virtualization while keeping different applications separated for security reasons **?**. Figure 2.2 shows an example of the usage of Docker Engine for a deployment of 4 containers that together provide 2 different services (S1 and S2). The engine itself only installs the required binaries and libraries for the containers as described in the image, and shares these between all the containers requiring them. This saves on resources and improves booting speed since binaries and libraries only need to be loaded once.

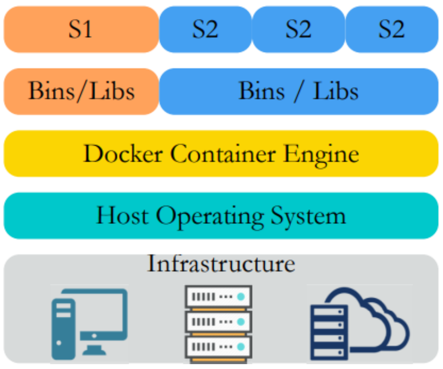


Figure 2.2: a container deployment using Docker Engine (taken from source **?**)

Docker Engine is owned by Docker **?** and is a pure container engine without any further bells and whistles. If more functionality is required such as a load balancer, integration with Docker Compose or image vulnerability scanning then Docker Desktop should be considered **?**. In this thesis we will not be using Docker Desktop, but an alternative container orchestration tool called Kubernetes (see section 2.3). Kubernetes also builds directly on Docker Engine, therefore we will now look at Docker Engine's inner working.

If we delve a bit deeper into the development of Docker Engine we find that it is an open source project supported by the Moby community **?**. Moby is a framework that consists of multiple plug-and play components that enable you to manage images, configuration and secrets, while providing networking and provisioning to your container, as seen in Figure 2.3. Docker engine thus offers some extra functionality for managing a container thanks to the Moby framework, but underneath the shell it in turn relies on a container runtime to actually run its containers.

Container runtimes manage the life-cycle of containers and serve as the bridge between the host system and the container **?**,**?**. when a container runtime is requested to run a container it will first pull the correct image from its image repository and will then ready the host system based on the metadata included in that image. This includes giving the container network attachments and storage if required. Afterwards it will communicate with the kernel to start the container. container runtimes adhere to OCI's runtime specifications which means they are interchangeable with many alternatives such as containerD

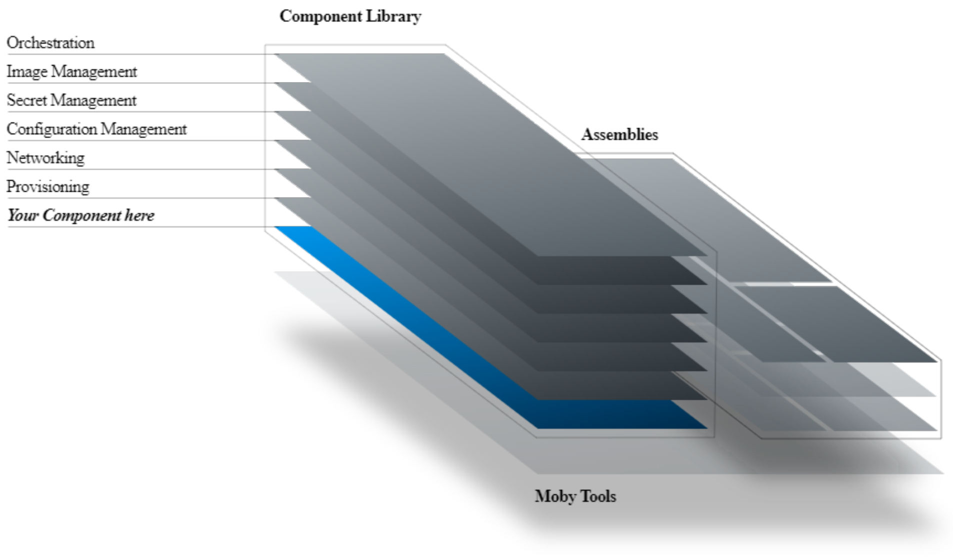


Figure 2.3: Moby framework (taken from source **?**)

?, CRI-O **?** and Mirantis **?** on the market. Our container engine, Docker Engine, is designed specifically with containerd in mind which makes it our obvious choice as container runtime.

2.3 Kubernetes

Even though Docker Engine can easily deploy multiple containers, managing all of them can become challenging very quickly, especially as the size of your container cluster scales up to possibly hundreds or thousands of containers. Issues such as container crashes, loss of network connectivity, running out of resources and many more can arise and need to be dealt with correctly. To help solve these challenges container orchestration tools were introduced such as Docker Swarm **?**, Apache Mesos **?**, and the current industry standard Kubernetes **? ?**. This thesis will research and utilise Kubernetes.

Kubernetes (K8s) is an open-source platform that automates container orchestration whether it is for one, hundreds, or even thousands of containers, and is supported by most major public cloud providers such as Amazon Web Services (AWS) **?**, Microsoft Azure **?** and Google Cloud Platform (GCP) **?**. Kubernetes automates functionalities such as scaling, deployment, storage, rollbacks, load balancing, secret management and many more, all of which can be configured using specifications created by a cluster manager in YAML or JSON format. To have a better understanding Kubernetes Figure 2.4 shows an overview of K8s components, where they are typically deployed and how they communicate with each other in a cluster. In the rest of this section we will describe the Kubernetes components and concepts in the figure that are important for this thesis.

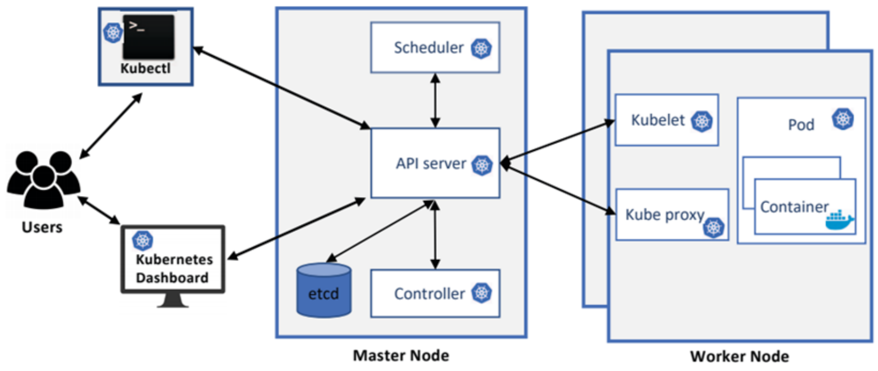


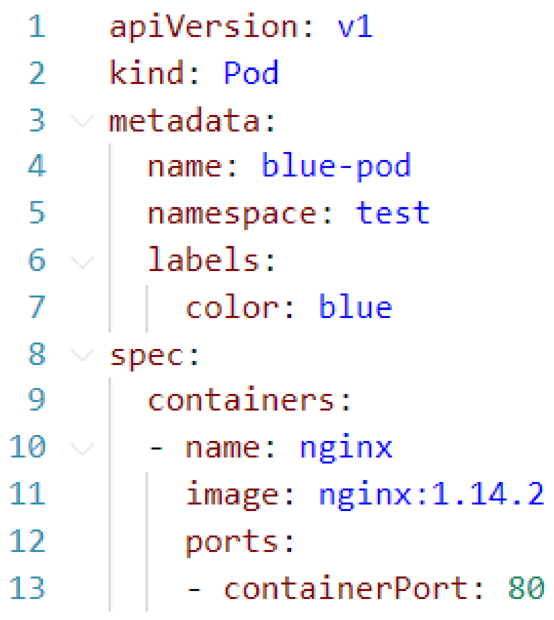
Figure 2.4: a brief overview of Kubernetes (taken from source **?**)

Node: Nodes are the biggest component in a K8s cluster and have two variations which are not mutually exclusive: worker and master nodes. They are machines, either virtual or physical, on which containerized applications run. Worker nodes host pod(s), a kubelet service and a kube-proxy service. Master nodes on the other hand are part of the control plane, and are responsible for managing the worker nodes. They host components such as the controller,scheduler, API server and etcd storage. These control plane components can be spread across multiple master nodes to provide fault-tolerance. Each cluster needs at least one node to operate correctly but can easily scale up to thousands. **?**

Pod: Pods are the smallest deployable computational components in a K8s cluster, run on a specific node, and house one or more containers that are tightly coupled. A pod remembers trough its specification file how to run the container(s) it is responsible for and shares storage and network resources between them. If containers are tightly coupled they can run on a single pod to have unrestricted communication and shared resources, but this introduces some security concerns such as the possibility of malware spreading between those containers. Duplicates of containers can be deployed on separate pods in order to increase an application's workload (horizontal scaling). To redirect requests to an application without any noteable differente for the end-user the load-balancer is introduced. It will automatically redirect requests to the application with the least current tasks to assure optimal spreading of end-users. Alternatively to horizontal scaling the available resources for a pod can be increased (vertical scaling) to scale workload capabilities. Figure 2.5 shows an example of a pod specification in the form of a YAML file. **?**

Kubelet: Each node in a cluster runs its own kubelet service component. The kubelet manages all the pods on the node it is deployed on by deploying, monitoring and deleting pods. It knows how to deploy these pods thanks to the PodSpec YAML or JSON file it receives from the API server in the control plane. By using the kubelet Kubernetes can ensure pod crashes get noticed and handled according to specifications. **?**

API server: The Kubernetes API Server offers REST operations trough which users can



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: blue-pod
  namespace: test
  labels:
    color: blue
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

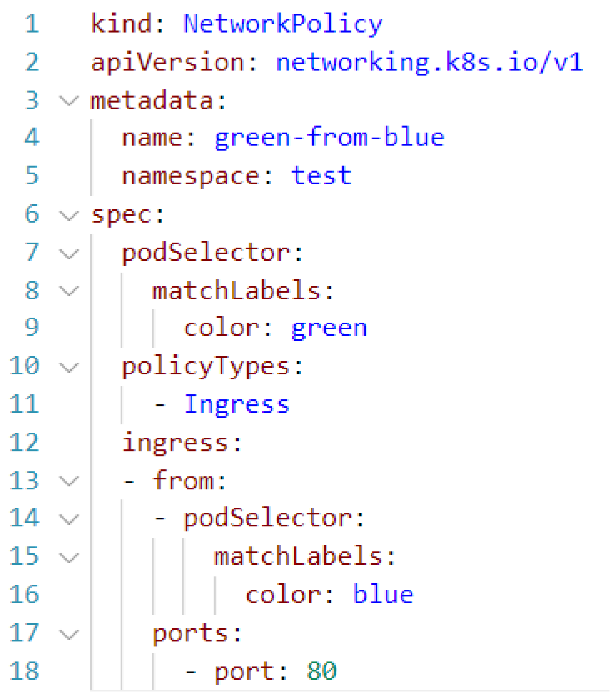
Figure 2.5: an example pod specification

interact with the cluster. It takes user inputs from the command line interface Kubectl or from a Kubernetes dashboard and communicates the necessary changes to the other components. It also validates request and configures the data that passes trough it. Libraries have been built on top of the REST operations to facilitate new ways to interact with the api server, and for this thesis we shall be using the python Kubernetes library as the main method of communication with the cluster **? ?**.

Namespace: A namespace is a mechanism to separate groups of resources in a Kubernetes cluster. Within a namespace each resource must have a unique name, but equal names can exist in different namespaces. objects that are used cluster-wide, like a node for example, do not allow the specification of a namespace. An example usage for a namespace would be the separation of different tenants according to their subscription plan on the cluster's resources **? ?**.

Labels: Labels are key/value pairs which allow us to identify objects such as pods and network policies. Multiple labels can be applied to a single object, much like a Role Based Access Control (RBAC) system where roles are linked to users **?**. Important to note is that for each object there can be no duplicate keys. e.g. applying *role: database* and *role: application* to a single object is not allowed and will result in errors. In the previously mentioned Figure 2.5 we can see an example of a Kubernetes pod that has the label *color: blue* attached to it.

In order to retrieve all objects based on labels the use of label selectors are required. If multiple labekl requirements are specified in a label selector then the found objects will have to match with not one but all of these requirements. Furthermore there is a difference between equality-based requirements and set-based requirements. When using the first matching objects must satisfy the (non)-equality with a specific label, e.g. it must (not) have the label *role: db*. The set-based requirement on the other hand offers three choices to be used with a set of values: *in*, *notin* and *exists*. An example of a set-based requirement would be *role in (database, application)* where either *role: database* or *role: application* would fulfil the requirement. Figure 2.6 shows an example of equality based-requirements in a network policy. **?**



```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: green-from-blue
  namespace: test
spec:
  podSelector:
    matchLabels:
      color: green
  policyTypes:
    - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          color: blue
    ports:
      - port: 80
```

Figure 2.6: an example network policy specification

Network policy: A Network Policy (NP) is used to control network traffic on the IP address or port level. Network policies are applied only to pods, but can use different type of selectors: a namespace selector will just target all pods in the defined namespace, an ipBlock selector targets pods based on a match in IP, and lastly pod selectors target based on labels. This last option will only look at pods with matching labels in the same namespace in which the NP is deployed. In this thesis we only focus on pod selectors with labels and work in a single namespace.

To describe the general build of a network policy we will look back at Figure 2.6. We can see that the NP is in the namespace *test* and that it selects pods that have the label *color: green*. This NP will thus only apply to pods in the namespace test with the same label. There are two types of policies: Ingress policies that define from which pods communication is allowed and egress policies that define to which pods communication is allowed. Thus the example NP in Figure 2.6 specifies that all pods with the label *color: green* in the namespace test are allowed to accept communication from all pods with the label *color: green*.

Important to note is that this does not mean pods with the label *color: blue* are allowed to send messages to pods with the label *color: green* yet. For this we need a correctly defined egress rule as well. In practice at least two network policies are required for pods to communicate. **?**

2.4 Kano

Although network policies offer extra security for a Kubernetes cluster it also has its drawbacks. They need to be closely managed to ensure no contradictions are specified and that containers can connect to other container they rely upon. To combat these difficulties Kano was introduced by researchers at the Tsinghua University in Bejing.

Kano is a container network policy verification tool presented in 2020 that, to quote the authors, solves the following problem: "In a container network, do the network policies violates the network constraints?" **?**. It does so by creating a matrix representing the container connections based on the defined network policies. Once this matrix is created it is leveraged to find underlying issues such as redundant NPs or an isolated container and those issues are reported to the user. We will continue to explain Kano in more detail since the solution algorithm of this thesis builds directly on some of Kano's concepts.

Kano Reachabilitymatrix: Kano starts by modelling a container network as a bipartite graph where the vertices are containers with 2 sets of edges *E1* and *E2*: ingress and egress network policies respectively. Figure 2.7 depicts this, but in 2 bipartite graphs instead of one to easily separate between the ingress and egress sets. The intersection of these sets ($E1 \cap E2$) represents connections between connections allowed by both an ingress and an egress rule. In order to save storage space and to allow quicker computations Kano does not actually save the container network as a bipartite graph, but as a matrix with bit arrays as rows, which we will call the kanomatrix or reachability matrix from now on. The kanomatrix is a square matrix of size *kxk* where $k = amount\ of\ containers\ in\ the\ cluster$. A 1 in position [i][j] means that the container with index [i] can communicate towards the pod with index [j] respectively. Figure 2.8 shows the ingress and egress matrix that correspond to the bipartite graphs in Figure 2.7, as well as the final kanomatrix which is equal to the intersection of the two matrices, achieved by using the bitwise AND operation between the matrices' bit arrays.

Prefiltration algorithm: In a container cluster changes happen frequently and therefore the reachability matrix often needs updating. A naive solution to update the reachability matrix would be to iterate over all policies and match them with containers with corresponding labels. This solution is not scalable to execute for every change since the time complexity would be $O(mn)$ with $m = number\ of\ policies$ and $n = number\ of\ containers$. To counter this Kano introduced a prefiltration algorithm based on bit arrays. The algorithm start by creating a hashmap, where each key is a label that is applied to at least one container, and the values are bit arrays with a length equal to the amount of containers in the cluster. In such a bit array the position of a set bit represents that the container with its index equal to the position has the corresponding

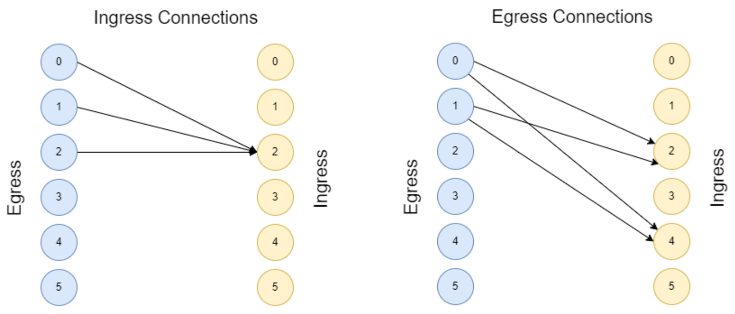
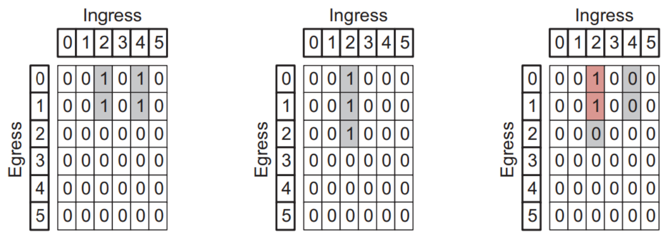


Figure 2.7: Bipartite graphs for Kano matrix generation



(a) Egress connections (b) Ingress connections (c) Final connections

Figure 2.8: Container network reachability matrix model (taken from source **?**)

label key applied. This is seen in Figure 2.9.

When searching to which containers a network policy applies we simply find the bit arrays corresponding to the selector labels of a policy in the hashmap. By applying a bitwise AND to these bit arrays we get a bit array with bits set for each container that matches all labels from the policy. This is illustrated in Figure 2.10. The time complexity of creating the hashmap based on the containers is $O(m)$, while the lookup for a policy is $O(1)$, which is a drastic improvement on the naive solution.

Violation check: Once the reachability is created with the prefiltration algorithm it can be used to verify some violations. The following descriptions are taken directly from the Kano paper **?**:

- *All reachable:* A container can be reached by all containers.
- *All isolated:* A container cannot be reached by any container.
- *User cross:* A container can reach other user's container in the container network.
- *Policy shadow:* The connections built by a policy are completely covered by another policy, then this policy may be redundant
- *Policy conflict:* The connections built by a policy are totally contradict the connections built by another.

Additionally, you can define your own constraints with the use of a declarative language, for example to guarantee two containers are able to communicate. Since the violation checker will not be used in this thesis we will not go into detail about its implementation

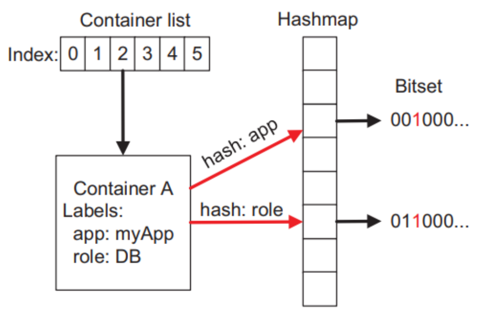


Figure 2.9: Prefiltration of container labels (taken from source **?**)

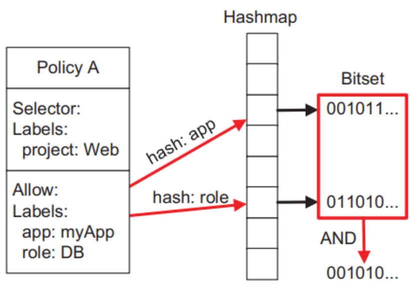


Figure 2.10: Prefiltration of policies (taken from source **?**)

2.5 Calico

Pods, and by extension containers, In a Kubernetes cluster are not able to communicate with each other by default. Instead you need to either explicitly create links between the pods or give each pod its own unique IP address within the cluster. The latter option of two is called the Kubernetes network model, and to achieve it a Container Network Interface (CNI) plugin is required. **? ?**. The CNI is a specification for writing plugins that should only be concerned with network connectivity of containers and deleting these as the container gets removed **?**. Many plugins based on the CNI project exist, such as Weave **?**, Cilium **?** and Project Calico **?**.

There are some rules that Kubernetes set to which CNI plugins must adhere before being allowed into the ecosystem. First and foremost pods must be able to communicate with each other, independently of the node on which they are deployed without the necessity of Network Address Translation (NAT). Secondly all Kubernetes components on a node such as the kubelet and kube-proxy must be able to communicate with all pods on that node. Together these 2 rules guarantee that CNI plugins do not hinder the working of the Kubernetes cluster, but still allow freedom for addition of functionalities.

Calico has been chosen for this thesis to ensure ready-to-go communication between the containers for testing purposes. Calico offers specific optimisations for Kubernetes by using eBPF, a Linux kernel feature that allows you to run a VM inside the kernel itself **?**. This means that Calico does not have to rely on the default iptables based Linux standard for network routing and was able to improve latency and performance with its own solutions. Calico also enforces the Kubernetes network policies and, although not used in this thesis, offers its own kind of network policies on top of the default K8s ones if required. The last reason we choose Calico instead of one of the alternatives is the default support for VM networking with Openstack, thanks to the Neutron ML2 plugin **?**.

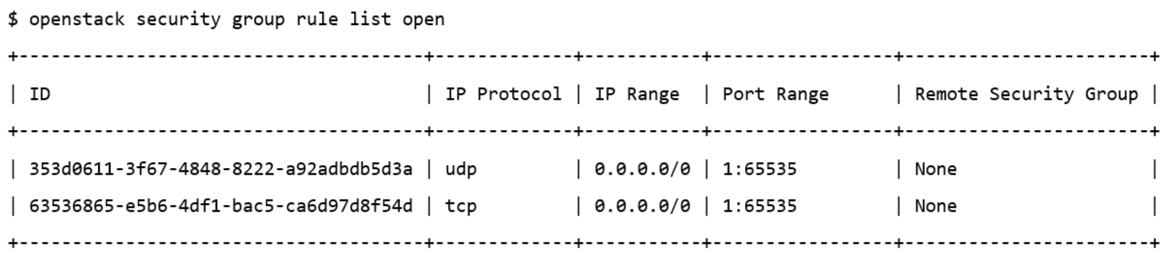
2.6 Openstack

In order to host our nodes in a Kubernetes cluster we have two options: Using physical devices or Virtual Machines. The second option not only reduces overhead, but requires no extra physical hardware if your existing hardware has enough resources, and can thus be scaled easily according to the needs of the cluster. Just like physical machines VMs need to be managed, and this is where virtual operating systems come into play, often synonymously called cloud operating systems due to cloud native computing becoming the standard **?**. Some examples of cloud operating systems are Google Chrome OS **?**, AWS **?** and Openstack, which is our choice for deploying our VMs in this thesis **?**.

Openstack automates the managing of computing and network resources for clusters of physical and virtual machines, which can be easily directed trough the dashboard application or directly via the API's. Openstack has great modifiability by allowing cluster administrators to select the components of their choice with example categories such as hardware life-cycle, storage, orchestration and application life-cycle. This way Openstack can be expanded to the requirements of the cluster. A VM deployed with OpenStack is called an instance, and in this thesis Openstack is used to deploy instances that serve as Kubernetes nodes. Openstack also has its own network security solutions to manage connection between instances, which we will more in depth in the following paragraphs.

Security Groups Security Groups (SG) are a way to group network security rules together to be easily applied to instances within the Openstack cluster. They are identified by their name, which must be unique within the cluster, and can carry a description to describe its intended usage. Every instance has the $default$ security group applied if no custom SG are linked. This default security group denies all incoming traffic and allows only outgoing traffic to your instance. A limit can be imposed on the maximum rules per security group and on the maximum defined security groups in the cluster by the security groups quota. **? ?**

Security Group Rules Security group rules are essentially IP filter rules to allow or block communication from certain IPs. These rules also specify a protocol for which they are applied, with the choices being TCP, UDP or ICMP. When wanting to target multiple protocols multiple security group rules are needed. The IP or IP range on which the rule applies must be specified in CIDR notation and a port range can be included as well. Alternatively to selecting IPs another security group can be targeted instead, with the advantage that instances can be added or removed from security groups without the need for revision of the security group rules. Figure 2.11 shows a security group wit the name *open* that holds two security group rules, one for UDP and one for TCP but both selecting all possible IPs and ports (65535 being the highest value of an unsigned 16-bit integer and thus the highest possible port value). **?**



```
$ openstack security group rule list open
+--------------------------------------+-------------+-----------+-----------------+-----------------------+
| ID                                   | IP Protocol | IP Range  | Port Range      | Remote Security Group |
+--------------------------------------+-------------+-----------+-----------------+-----------------------+
| 353d0611-3f67-4848-8222-a92adbdb5d3a | udp         | 0.0.0.0/0 | 1:65535         | None                  |
| 63536865-e5b6-4df1-bac5-ca6d97d8f54d | tcp         | 0.0.0.0/0 | 1:65535         | None                  |
+--------------------------------------+-------------+-----------+-----------------+-----------------------+
```

Figure 2.11: Security group rule example (taken from source **?**

3. Proposed Solution

In this thesis, we aim to create an algorithm that, once running on one of the control plane nodes of a Kubernetes cluster, continuously watches the state of that cluster and detects any conflicts between cloud and cluster layer. To achieve this We capture the six types of events in the cluster that can influence the connection between containers in the cluster layer as seen in Table 3.1. When one of these events is captured an automatic conflict detection should be triggered and the algorithm must report on all possible conflicts between the cloud and cluster layer. It should be noted that changes in the cloud layer can influence connections as well (e.g. security groups being created) but that this is outside the scope for this thesis.

pod	network policy
create	create
delete	delete
update	update

Table 3.1: watcher events

In order to work incrementally we want to store all necessary information about the current state of the cluster. With this information we can look directly at the relevant containers and network policies for each event and reduce the computation time per event. For example, there is no need to check containers that are not affected by a newly created network policy. We call this the incremental approach, since it incrementally updates the stored cluster state with each event. As a bonus we added a simple startup detection method that can detect conflicts upon startup of the algorithm. periodically running this startup detection could serve as an alternative in case a continuously running algorithm would prove unviable in any way.

We will only monitor a single namespace when executing the algorithm, which can be specified as an argument. The reasoning behind this decision is that the main purpose of namespaces is to isolate groups of resources within the cluster and conflicts between multiple namespaces should thus not exist when used correctly **?**. If required multiple instances of the algorithm can be run for different namespaces. We would also like to note that we will deploy only one container per pod in our algorithm, and that therefore the term pod and container might frequently be used as synonyms in the case of variables.

In the rest of this chapter We will describe our solution, implemented in Python, that meets all the requirements and solves the problem of conflict detection between cloud and cluster layers. The implementation is based on a Kubernetes cluster with nodes deployed on an Openstack installation. Figure 3.1 demonstrates how the algorithm is split up in separate parts, which correlates directly to different python files with the corresponding name and functionalities. We briefly summarize these functionalities for each component of the algorithm before describing them more in depth:

- *Watcher:* The watcher initializes the other components and leverages the Python Kubernetes library to continuously monitor for each of the six events described in Table 3.1, which it will forward to the analyzer.
- *Analyzer:* The analyzer orchestrates the handling of events passed by the watcher. First these events must be forwarded to the parser to be turned into usable objects. Afterwards the analyzer calls the KIC to update the reachability matrix and cluster state and the SGIC to look at this updated reachability matrix for conflicts.
- *Parser:* The parser turns the event data from the watcher into objects that can be used by the Analyzer. Since it's functionality is straight forward this component will not be described in depth in the rest of this chapter.
- *Kubernetes Information Cluster (KIC):* The KIC handles the incremental update of the reachability matrix and stores all information about the cluster state.
- *Security Group Information Cluster (SGIC):* The SGIC generates a randomised set of security groups and corresponding rules and binds them to nodes upon startup. Additionally it performs conflict detection between the cluster and cloud layer.
- *Model*: The model defines data structures to be used within the other components. These data structures are often created by the parser out of event data, such as Container, Policy and Reachabilitymatrix objects. Finally it leverages the Kano generation method to create a reachability matrix upon startup which offers the base matrix which we update incrementally with each captured event.
- *LabelTree*: A labelTree is a custom tree-like data structure used to store network policies by their selector labels, for quick retrieval by the KIC later on.

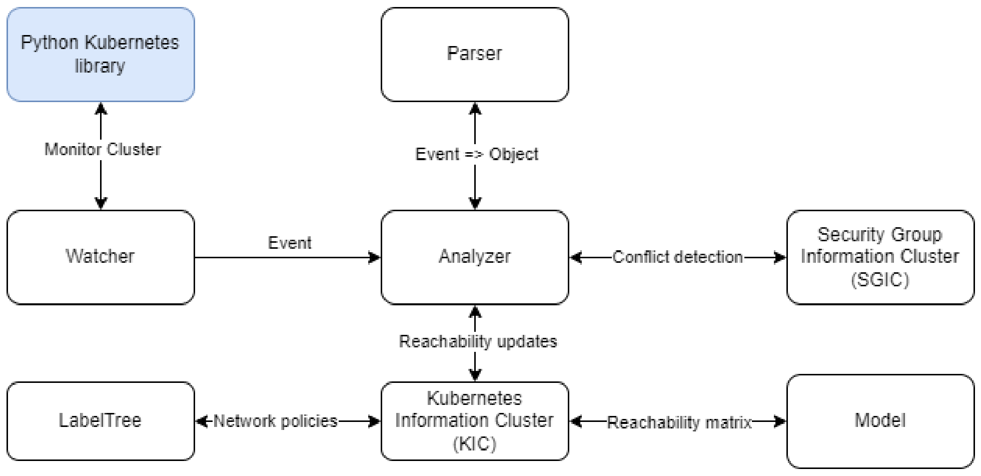


Figure 3.1: Algorithm structure

3.1 Model

There are many data structures used in the algorithm, most of which are stored in the model file. To ensure a good understanding of these underlying structures in the other components we will first describe the ones that occur most often: the Policy, Container, Store and ReachabilityMatrix.

Policy: The Policy structure stores multiple variables which often are specific data structures on their own. We will not describe each of these sub-structures in depth, but instead briefly explain the most important variables in a short manner:

- *name:* Stores the name of the policy. Although they are not enforced to be unique in the algorithm they will be in practice, since all objects in the same namespace must have unique names.
- *selector:* This data structure stores the labels which selects the grouping of pods to which the policy applies.
- *allow:* This data structures stores the labels that select the ingress sources or egress destinations.
- *direction:* The direction is a boolean that indicates weather it is an ingress (True) or egress (False) policy.
- *id:* The id is the identifier for the policy. The usage of the id field as the identifier is preferred above the usage of name since it can be chosen and changed within the algorithm. The name on the other hand is defined by Kubernetes and can only be changed if the policy is removed and added under the new name.

Furthermore the policy stores variables such as port and CIDR to provide further details that might be required for future work.

Container: The container has some overlap in variable names and usage with the Policy data structure, such as id and name. additionally it stores a list of labels applied to the container, a string variable called nodeName for the node on which the container is deployed, and a matrix_id. This last integer variable indicates which position in the kanomatrix this container corresponds to. Since the kanomatrix will change in size when containers get removed or added, this variable will often change throughout the algorithm's handling of events.

Store: The store is a custom structure that is based on a dict (the Python equivalent of a hash table). It takes two integers, which we will call the key-duo, which get combined as a tuple to serve as a key in the dict. The value related to this custom tuple key is a list that can store objects depending on the need. The data structure offers functions to retrieve the list coupled to a key-duo, add an item to the list coupled to a key-duo, remove a specific value within a specific key-duo's list, and to remove the entire entry for a specific key-duo out of the dict. This data structure is used within the ReachabilityMatrix data structure to store the policies responsible for a connection between two containers. An example of a Store can be found in the next paragraphs.

ReachabilityMatrix: The reachabilityMatrix data structure stores multiple variables, which all get filled when calling its most important function: *build_matrix*. This function takes a list of containers and policies and generates the corresponding reachability matrix according to Kano's algorithm **?**. the generated matrix will then be used as a base that we will incrementally update for further events. During the creation of this matrix many useful results get stored in variables for later usage which we will briefly describe:

- *dict_pods:* This dict stores the containers (which all get deployed in their own pod, hence the name) as values, with an incrementing list of numbers as keys. This variable ensures us that the order of these containers is set for later handling. This variable ensure that the pods will be set in the same order when incrementally updating the matrix, thus keeping the unaffected rows and columns in the same position.
- *dict_pols:* Similar to dict_pods but for network policies
- *label_map:* This is the hashmap of container labels as described in the Kano pre-filtration algorithm in section 2.4.
- *resp_policies:* The resp_policies is a Store object as described above and stores the responsible network policies for each container connection.
- *matrix:* The end-result of the *build_matrix* function is stored in this variable: the reachability matrix created with the given containers and policies, stored as a list of bit arrays.

Figure 3.2 shows an example of a reachabilitymatrix and some corresponding variables. This could be the result of running the *build_matrix* function given a list of 3 containers (a, b and c) and a list of 6 policies (u, v, w, x, y, z). We can see that three container connections are allowed by the given network policies, indicated as a 1 in the matrix. the resp_pols Store object thus stores 3 key-duos in it's dict, one for each connection. The connection between container 1 and 3 is enabled by two sets of network policies: (1, 5) and (1,4) where the first policy in these duos is an ingress rule and the second an egress rule. When leveraging the dict_pols and dict_pods variables we can thus give the following statement:

Container a can send messages to container c, and this connection is allowed both by the combination of ingress rule u and egress rule y and the combination of ingress rule u and egress rule x.

Note that this also means there are redundant policies. We can remove policy z since it is not used, and even policy x or policy y since one can replace the usage of the other.

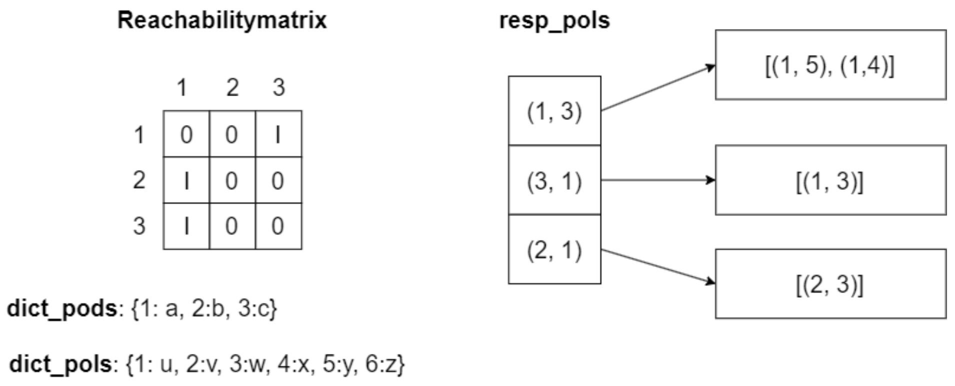


Figure 3.2: Example of a reachabilitymatrix and some corresponding variables

3.2 Watcher

Execution of the algorithm starts by calling the main method of the watcher, which will then initialise all the other required components. The following flags and arguments are available when calling the watcher file:

- **namespace:** The namespace in which the algorithm will look for conflicts (**required**).
- **verbose:** if the verbose flag is set, either by using -v or –verbose, the updated reachability matrix and corresponding container ids will be printed after each event (**optional**).
- **debug:** If the debug flag is set, either by using -d or –debug, all data structures will be printed out to provide more information about changes in stored cluster state (**optional**).
- **startup:** If the debug flag is set, either by using -s or –startup, then a conflict detection will be executed when the algorithm is executed. This offers a quick conflict check of the current cluster state upon startup (**optional**).

When executed the watcher will start by collecting all the currently existing containers and network policies on the cluster in the namespace that is defined in the arguments. To do this it will leverage the Python Kubernetes library to retrieve lists of pods and policies, and pass them to the Parser to be turned into usable objects. These objects are passed to the analyzer which in turn calls upon the Kubernetes Information Cluster to generate the base reachabilitymatrix using Kano's generative method. The analyzer also calls upon the Security Group Information Cluster to generate random security groups and security group rules and assign them to nodes. More information about the Analyzer, KIC and SGIC can be found in section 3.3, section 3.5 and section 3.6 respectively

After this initialisation stage the watcher is responsible for capturing all events on the cluster from Kubernetes by once again leveraging the Python Kubernetes library. It filters the resulting stream of data to find the six events defined in Table 3.1 and adds them to an event queue to be handled. Container and policy events are outputted by different

API endpoints of the Kubernetes cluster, so in order to watch these API's simultaneously multiple concurrent threads are required. To retrieve the events from the event queue and analyze them for changes in the cluster, all while maintaining the simultaneous monitoring of the API's, a third thread is introduced which we call the consumer. This thread will continuously take an event from the queue, send it to the analyzer for further handling, and await a response before going to the next event. This way the events are handled in the same order as their occurrence in the API's and only one event at a time. This prevents mistakes such as trying to analyze the deletion of a pod without it being present in the last saved cluster state, since the creation of that pod has not been handled yet.

3.3 Analyzer

The main task of the analyser is to handle all events received from the watcher. In order to do this it is closely coupled with the parser, Kubernetes Information Cluster and Security Group Information Cluster, all of which are initialized during the analyzer's initialization. The $analyseEvent(event)$ function is the main reason of existence for the Analyzer and is shown in a simplified version in Algorithm 1. We will now briefly describe how it works.

the raw event data is first parsed into a policy or container object in the parser. It then immediately continues with calling the KIC to update the reachabilitymatrix with the new object. Although it is shown here as a single function call for any of the six events, it is actually different functions for each one which will be described in more detail in section 3.5. The rest of the function's behaviour is dependant of the event type as well.

If the handled event is for a policy object then the size of the new reachability matrix has not changed: the amount of containers stays the same. We can thus create a deltamatrix by using the bitwise AND operation on the new reachabilitymatrix and the reachability matrix stored in the KIC that represents the previous cluster state. The result is a matrix of the same size as these reachability matrices but with a 1 on any position with a changed value and thus a changed container connection. By looking for these 1's in the deltamatrix we know where changes occur on which we can report and for which containers we must call upon the SGIC for conflict detection.

If a container event is being handled the size of the new reachability matrix will change due to the direct correlation between matrix size and the amount of containers in the cluster. The exception would be the container update event, which will use the deltamatrix in a similar fashion as the policy events in the previous paragraph. When handling a container delete or create event we start by looking at the matrix_id of the object: If the container haS the matrix_id i then we look at position $[i][j]$ and $[j][i]$ in the matrix with $j\ in\ range(number\ of\ containers)$. We must be careful whether we use the new or previous-state reachability matrix for this position lookup: a delete event will mean the object is not present in the new reachabilitymatrix and vice-versa for the creation event. Afterwards we report on all these matrix positions where the value is 1: either a connection is made (create event) or a connection existed and is now removed (delete event). Lastly we call upon the SGIC conflict detection, which is described in section 3.6.

Algorithm 1 Event Analysis

```
function ANALYSEEVENT(event)
    obj ← parser.create_object_from_event(event)
    new_reach ← kic.update_kano_matrix(obj)
    if obj is a policy then
        deltamatrix ← kic.kano_reach AND new_reach
        if deltamatrix not all zeroes then
            for i, j where deltamatrix[i][j] == 1 do
                report changes
                sgic.check_sg_connectivity(i.node, j.node, connection_wanted)
            end for
        end if

    else if obj is a container then
        if event['custom'] == "create" then
            for i in new_reach.containers do
                if new_reach[obj][i] == 1 then
                    report changes
                    sgic.check_sg_connectivity(obj.node, i.node, True)
                end if
                if new_reach[i][obj] == 1 then
                    report changes
                    sgic.check_sg_connectivity(i.node, obj.node, True)
                end if
            end for
        else if event['custom'] == "delete" then
            for i in kic.containers do
                if kic.kano_reach[obj][i] == 1 then
                    report changes
                    sgic.check_sg_connectivity(obj.node, i.node, False)
                end if
                if kic.kano_reach[i][obj] == 1 then
                    report changes
                    sgic.check_sg_connectivity(i.node, obj.node, False)
                end if
            end for
        end if
    end if

    update cluster state
end function
```

3.4 Labeltree

Before continuing to the Kubernetes Information Cluster we must briefly talk about the Labeltree. This custom data structure is used in the KIC to store all network policies of the current cluster state based on their selector labels (see section 2.3). It is based directly on tree structures, with some changes in the update, delete and get methods to account for the key-value labels. Figure 3.3 shows the structure of a labeltree: it always has a max depth of three, with depth one being the root node, depth two the keys of labels and depth three the values corresponding to the parent key.

Because a network policy might have multiple selector labels the policy might be present in multiple leaf nodes. This is required so that, given a container with multiple labels, we can find all network policies that might apply to a single label. However, before assuming a policy applies to a container it must always be verified that all the selector labels of that policy are present in the container. The main reason to use this custom structure is it's constant date retrieval time due to the max tree depth of 3, and it's intuitive representation.

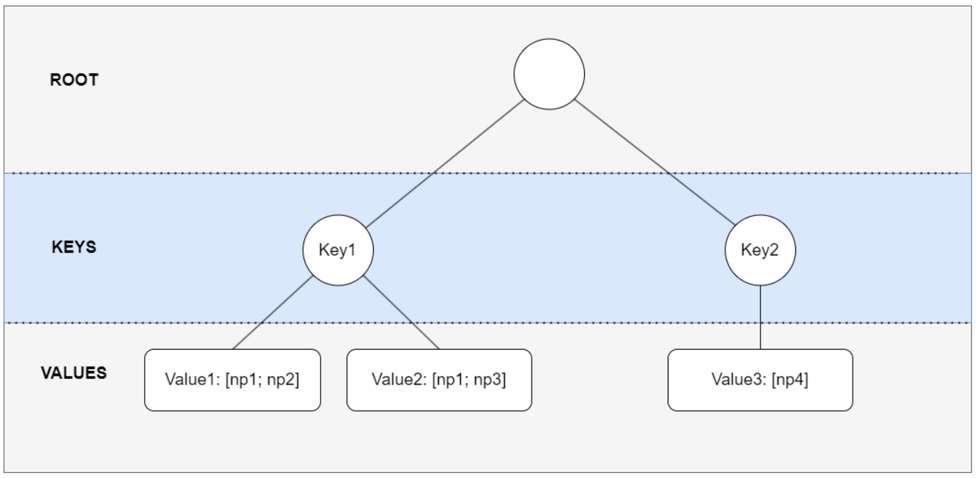


Figure 3.3: Structure of a Labeltree

3.5 Kubernetes Information Cluster

The Kubernetes Information Cluster has two main functionalities: store all information about the current cluster state, and update the cluster state given an event. In order to achieve this first functionality the KIC stores the following variables:

- *egressTree:* The egress tree is a Labeltree used to store all the egress policies in the current cluster state.
- *ingressTree:* The ingress tree is a Labeltree used to store all the ingress policies in the current cluster state.
- *reachabilitymatrix:* This is a ReachabilityMatrix data structure that stores the current state in the format of a Kanomatrix and corresponding variables, as described in section 3.1.
- *pods:* A list of current containers
- *pols:* A list of current network policies

Additionally the KIC also offers functions to update these variables.

For the second functionality of updating the current cluster state and reachabilitymatrix when given an event different functions are required for different events. But before we can describe these functions we need to look at two smaller portions of the algorithm that reoccur often: getting all the containers that match all the label selectors in the allow set and similarly for the select set.

Both these algorithms use the same principle: for each label in a set of label selectors we retrieve the bitarray of containers that have this label applied according to the label_map (see Figure 2.4). We then do a bitwise AND operation to find the containers that have all these labels. When looking for the containers of the allow section of a network policy we must take into account that multiple label selector sets can exist. Therefore we retrieve the containers for each set separately and use a bitwise OR operation to find the final containers. This is described in Algorithm 2 and Algorithm 3.

Algorithm 2 Find containers matching the select labelselectors of a given policy

```
Input: a network policy
Output: bitarray of containers matching the select labelselector

select_containers_final ← bitarray(0 * amount of containers)
first ← True
for select_label in policy.selector do
    containers ← new_reach.label_map.get(select_label)
    if containers is not empty then
        if first then
            first ← False
            select_containers ← containers
        else
            select_containers AND containers
        end if
    else:
        select_containers ← bitarray(0 * amount of containers)
        break
    end if
end for
```

Algorithm 3 Find containers matching the allow labelselectors of a given policy

```
Input: a network policy
Output: bitarray of containers matching the allow labelselector

allow_containers_final ← bitarray(0 * amount of containers)
for allow in policy.allow do
    allow_containers ← bitarray(0 * amount of containers)
    first ← True
    for allow_label in allow do
        containers ← new_reach.label_map.get(allow_label)
        if containers is not empty then
            if first then
                first ← False
                allow_containers ← containers
            else
                allow_containers AND containers
            end if
        else:
            allow_containers ← bitarray(0 * amount of containers)
            break
        end if
    end for
    allow_containers_final OR allow_containers
end for
```

For the final part of this subsection we will describe the functions in the KIC that are responsible for handling the events described in Table 3.1 except the update events. Since each update has differences regarding the values that changed within the object the function handling these events would have to take into account many variables. Simply calling the delete and create method shortly after one another will get the same result instead. The functions that we created for the other four events all take an object as parameter which is either a container or policy object as seen in section 3.1. We will now briefly describe each function and show their implementation in pseudo-code.

- **Delete Network Policy (Algorithm 4):** To delete a network policy we first copy the existing reachabilitymatrix to the new reachabilitymatrix. We then use Algorithm 2 and Algorithm 3 to get bit arrays of all the containers that respectively match the select and allow label selectors. We use the labels of each container in the allow_containers bitarray to find policies in the opposite direction with matching selector labels (i.e. if the deleted policy is ingress we look for an egress policy). In the following step we only need to check if these opposite policies match the select_containers of the deleted policy with their allow label selectors. If this is the case we have a match and need to remove the responsible policies for these 2 containers, as well as update the matrix to include a 0 at the correct position if no other responsible policies between these containers exists.
- **add Network Policy (Algorithm 5):** To add a network policy we follow the same steps as in the delete policy algorithm: we find the select container, the allow containers and the opposite policies, and if they all match up we got a match. However we now add a 1 to the correct position in the reachabilitymatrix and add to the resp_policies instead of removing from it.
- **Delete Container (Algorithm 6):** To delete a container we first create a new reachabilitymatrix with one less row and column than the existing current state reachabilitymatrix. We then iterate over all the existing containers, get their corresponding row in the existing reachabilitymatrix, and remove the bit that corresponds to the deleted container before adding the edited row to the new reachabilitymatrix. We update the matrix_ids by decrementing each one that is higher than the removed container's matrix_id . Updating the label_map of the containers is done similarly as to how the matrix has been updated: going over each bit array, removing the bit corresponding to the removed container and moving each bit behind the removed bit up by 1 position. Lastly the empty bit arrays get removed from the label_map to remove unused labels.
- **Add Container (Algorithm 7):** To add a container we first copy the existing reachabilitymatrix into the new_reach variable and give a new matrix_id to the container. By assigning a matrix_id higher than that of all other containers we can guarantee that the new container is added at the end of the matrix for better visualization. Next we add a 0 on the end of each existing bit array and append a row of zeroes to the end of the reachabilitymatrix. following this we find all rules that are applied to this new container and store them in the rules set which will be traversed to find the containers that match the allow selectors of these rules

using Algorithm 3. Now that we have a container that is selected by a policy that selects our new container, we must only find another network policy in the opposite direction. If such a NP exists a 1 is added on the correct position of the reachabilitymatrix and all the related variables are updated.

Algorithm 4 Delete policy from reachability matrix

```
function REACHABILITYDELETENP(policy)
    new_reachability ← kic.reachabilitymatrix
    select_containers ← Algorithm 2
    allow_containers ← Algorithm 3
    opposite_policies ← {}
    for allow_container in allow_containers where bit == 1 do
        for label in allow_container.labels do
            if policy is ingress then
                treenode = egressTree.find(label)
            else
                treenode = ingressTree.find(label)
            end if
            for policy2 in treenode do
                if all labels in policy2.selector are in allow_container.labels then
                    opposite_policies.add((policy2, allow_container))
                end if
            end for
        end for
    end for
    for (policy2, allow_container) in opposite_policies do
        for select_container in select_containers where bit == 1 do
            for allow in policy2.allow do
                if all labels from allow in select_container.labels then
                    remove policy and policy2 from resp_policies for the containers
                    if no responsible policies between the containers exist then
                        set bit in reachabilitymatrix to 0
                    end if
                end if
            end for
        end for
    end for
    update variables
    return (new_reachability)
end function
```

Algorithm 5 add policy to reachability matrix

```
function REACHABILITYADDNP(policy)
    new_reachability ← kic.reachabilitymatrix
    select_containers ← Algorithm 2
    allow_containers ← Algorithm 3
    opposite_policies ← {}
    for allow_container in allow_containers where bit == 1 do
        for label in allow_container.labels do
            if policy is ingress then
                treenode = egressTree.find(label)
            else
                treenode = ingressTree.find(label)
            end if
            for policy2 in treenode do
                if all labels in policy2.selector are in allow_container.labels then
                    opposite_policies.add((policy2, allow_container))
                end if
            end for
        end for
    end for
    for (policy2, allow_container) in opposite_policies do
        for select_container in select_containers where bit == 1 do
            for allow in policy2.allow do
                if all labels from allow in select_container.labels then
                    add policy and policy2 to resp_policies for the containers
                    set bit in reachabilitymatrix to 1
                end if
            end for
        end for
    end for
    update variables
    return (new_reachability)
end function
```

Algorithm 6 Delete container from reachability matrix

```
function REACHABILITYDELETECONTAINER(container)
    new_reachability ← $nXn$ matrix of bitarrays of 0's, with n = # of containers
                                                  ▷ Updating the reachability matrix
    for i, cont in enumerate(containers) do
        row ← reachabilitymatrix[cont.matrix_id]
        row.pop(cont.matrix_id)                   ▷ bit for the removed container
        if container.matrix_id > container.id then
            container.matrix_id -= 1
        end if
        store row in new_reachability
    end for
                                                  ▷ Updating the label_map matrix
    new_label_map ← {}
    for label, old_arr in reachabilitymatrix.label_map.items() do
        for k in range(len(old_arr)) do
            if k > container.matrix_id then
                new_label_map[label][k - 1] ← old_arr[k]
            else if k < container.matrix_id then
                new_label_map[label][k] ← old_arr[k]
            end if
        end for
    end for
                                        ▷ Removing empty bitarrays from label_map
    new_label_map_v2 ← {}
    for label, arr in new_label_map.items() do
        if not any bits set in arr then
            delete new_label_map_v2[label]
        end if
    end for
    new_reachability.label_map ← new_label_map_v2
    for pod in pods do
        if pod.matrix_id > container.matrix_id then
            pod.matrix_id -= 1
        end if
    end for
    return (new_reachability)
end function
```

Algorithm 7 Add container to reachability matrix

```
function REACHABILITYADDCONTAINER(container)
    new_reach ← kic.reachabilitymatrix
    container.matrix_id ← len(containers)
    matrixId_to_Container[container.matrix_id] = container
    for label, array in label_map do
        array.append(False)
    end for
    for row in new_reach do
        row.append(0)
    end for
    new_reach.append(bitarray(0 * amount of containers))
    rules = Set()
    for label in container.labels do:
        for policy in eggressTrie.find(label) do
            if all policy.select.label in container.labels then
                rules.add
            end if
        end for
        for policy in ingressTrie.find(label) do
            if all policy.select.label in container.labels then
                rules.add
            end if
        end for
    end for
    for rule in rules do
        allow_containers ← Algorithm 3
        for secondcontainer in allow_containers do secondRules = Set()
            for secondlabel in secondcontainer.labels do:
                for policy in eggressTrie.find(label) do
                    if all policy.select.label in container.labels then
                        secondRules.add
                    end if
                end for
                for policy in ingressTrie.find(label) do
                    if all policy.select.label in container.labels then
                        secondRules.add
                    end if
                end for
            end for
            for secondrule in secondRules do
                if all labels of secondrule.selector in secondcontainer then
                    for secondallow in secondrule.allow do
                        if all labels of secondallow in container.labels then
                            update the matrix for new connection
                        end if
                    end for
                end if
            end for
        end for
    end for
    return new_reachability
end function
```

3.6 Security Group Information Cluster

Since handling cloud layer events is out of scope for this thesis we also have no need to actively monitor the cloud layer. Therefore the Security Group Information Cluster mimics the security groups and security group rules one might find in the cloud layer by generating them with randomised variables when the Watcher is initialized. Security group objects are based upon the Openstack Security Group definitions, which means they contain a list of rules and have a unique name **?**. The amount of rules in each security group is randomised as well, and each rule contains variables such as port, direction, protocol and ethertype filled with randomised values. Additionally a security group rule targets other nodes by either a single remote IP, an IP with a subnetmask to act as a range of IPs, or the name of another existing security group which targets all nodes part of that security group.

Once the security groups and their rules are created each node in the Kubernetes cluster gets linked to one or multiple security groups. We then use directed graphs to store allowed connections, where nodes are the vertices and an edge between two nodes indicates that connection is allowed in that direction. Each edge includes the security group and SG rule number responsible for the connection. Thereafter both an ingress and egress graph get created and filled by going trough all security groups, finding their nodes and adding the corresponding security group rules to these nodes in the graphs. We then use these 2 directed graphs to create a VMmatrix of size kxk where $k = amount\ of\ nodes\ in\ the\ cluster$, similarly to the Kano reachabilitymatrix. When an edge is present in both graphs the connection is allowed in both directions and thus results in a 1 in the VMmatrix. Since we created the directed graphs we can easily retrieve responsible security groups and SG rules for a specific node connection. The VMmatrix on the other hand quickly tells us whether or not the connection is possible in the first place.

The Security Group Information Cluster is also responsible for conflict detection between the cloud and cluster layer. Once the KIC finishes creating the updated reachabilitymatrix the Analyzer will call the check_sg_connectivity function in the SGIC which we will now explain with the help of Algorithm 8. The function takes three parameters: 2 node names and a boolean indicating whether or not a connection between these 2 nodes is wanted. E.g. a newly created container can communicate to another container on a different node: then the function gets called with the boolean connection_wanted set to True. The function will then print out whether or not the nodes are able to communicate by looking at the VMmatrix and print out the responsible security groups and security group rules by retrieving them from the directed graphs. If the connection is contradictory to the boolean we report a conflict.

Algorithm 8 Conflict detection

```
1: function CHECK_SG_CONNECTIVITY(node1, node2, connection_wanted)
2:     print security groups for node1
3:     print security groups for node2
4:     if vmMatrix[node1, node2] == 1 then
5:         if connection_wanted == True then
6:             print security group rules responsible for connection
7:         else
8:             report conflict with responsible security group rules
9:         end if
10:    end if
11:    if vmMatrix[node2, node1] == 1 then
12:        if connection_wanted == True then
13:            print security group rules responsible for connection
14:        else
15:            report conflict with responsible security group rules
16:        end if
17:    end if
18: end function
```

4. Evaluation

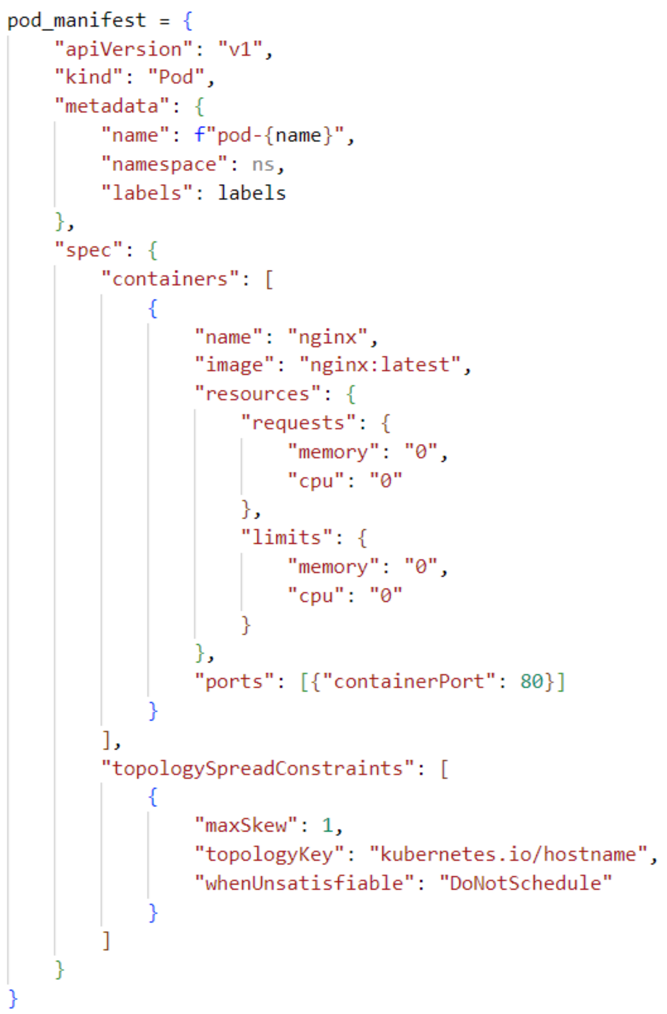
This chapter aims to evaluate our algorithm described in chapter 3. This evaluation will be based of 4 research questions, split up into 2 experiments. The first experiment will be a comparison between Kano and the comparable part of our algorithm that is responsible for incrementally updating the matrix. The second experiment is a general test of our overall algorithm in terms of time and memory consumption. Both of the evaluation experiments are run on the same cluster setup and share configuration specifics, which we will be described in the first section. The second and third section will talk about the first and second experiment respectively and include subsections about the experiments' approach, setup and results.

4.1 evaluation setup

The Kubernetes cluster used for the experiments consists of 8 nodes: 7 workers nodes and a single control-plane node that does not run any containers. Each of these nodes are located on their own VM, which are deployed as instances on the Openstack installation of the Department of Computer Science at KULeuven. Each instance has 2 VCPUs, 4GB RAM and 20GB memory, and runs Ubuntu 22.04 jammy for it's OS. There are no Security Groups applied to the instances except for the default. Calico is used as CNI to provide connection between the kubernetes nodes, which all run on the Kubernetes Gitversion v1.22.17. Kubernetes runs on default settings and specifies the maximum amount of pods per worker nodes at around 100 pods.

Each pod we deploy is created wit the latest nginx image, which is version 1.25.3 at the time of the experiments and writing. Figure 4.1 shows the pod manifest structure for each deployed pod, with *name* and *labels* randomly generated and *ns* depending on the execution arguments which we will describe in the next sections. As shown in the picture the pods are specified to not use any resources when deployed. The experiment algorithms are executed on the control-plane node and will execute without need for interaction, on the condition that the specified namespace used for the experiment already exists on the cluster.

Each experiment will be run once for each of the events described in Table 3.1, since they can differ in space and time cost. For example, removing a container might be faster than adding a new container, since the first mainly just removes data while the latter includes a search for matching network policies according to it's labels. However, we only measure four out of the six events that we can capture, since updating network policies and containers equals directly to first executing a deletion event, directly followed by a creation event. The update events can thus be calculated from the creation and deletion events saving time when executing the experiments.



```
pod_manifest = {
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "name": f"pod-{name}",
        "namespace": ns,
        "labels": labels
    },
    "spec": {
        "containers": [
            {
                "name": "nginx",
                "image": "nginx:latest",
                "resources": {
                    "requests": {
                        "memory": "0",
                        "cpu": "0"
                    },
                    "limits": {
                        "memory": "0",
                        "cpu": "0"
                    }
                },
                "ports": [{"containerPort": 80}]
            }
        ],
        "topologySpreadConstraints": [
            {
                "maxSkew": 1,
                "topologyKey": "kubernetes.io/hostname",
                "whenUnsatisfiable": "DoNotSchedule"
            }
        ]
    }
}
```

Figure 4.1: Template for pod creation

We want to know how each experiment behaves when the cluster size increases in term of pods and network policies. For this reason we define 5 cluster setups that define variables such as number of pods and number of network policies. The combinations of 5 cluster setups and 4 events gives us a total of 20 smaller experiments for each experiment, which we will call sub-experiments. Each of these 20 sub-experiments is run a hundred times giving us a total of 2000 runs and their respective data per experiment.

4.2 Experiment 1

Kano is a research solution that verifies network policies based on the reachabilitymatrix it generates when provided with a list of pods and a list policies. For some background about Kano we refer to section 2.4. Kano only has a generative algorithm to create this reachabilitymatrix, while our solution provides an incremental approach that updates the reachabilitymatrix instead of regenerating it. Naturally a comparison between each approach is required and will be described in this section. This comparison will be used to answer the following two research questions:

- *Q1:* What is the difference in time cost between an incremental update of the Kano matrix compared to newly generating the Kano matrix for every event and how does this difference scale with pod/policy numbers
- *Q2:* What is the difference in space cost between an incremental update of the Kano matrix compared to newly generating the Kano matrix for every event and how does this difference scale with pod/policy numbers

Before we delve into the experiment approach, setup and results we declare our hypotheses for these questions:

- *Hypothesis Q1:* We predict that the time cost of our incremental approach will be higher in small clusters due to writing times of variables, but will prove better than the time cost of the generative approach as the cluster increases, since the incremental approach does not have to verify all network policies and containers.
- *Hypothesis Q2:* We predict that the space cost of the incremental approach will be higher than that of the generative approach due to the memory required for storing the current cluster state. We predict this difference will only increase as the cluster increases in size.

4.2.1 approach

Since our algorithm does much more than just updating the reachabilitymatrix we first have to remove the irrelevant parts from our codebase. The Security Group Information Cluster is removed from the algorithm altogether, while the Analyzer gets modified so that it only calls upon the Kubernetes Information Cluster to update the reachability-matrix. As a result the analyzer does not use the updated matrix to compare it with the previous cluster state nor does it print out any information anymore. This adapated algorithm for the incremental approach now allows for a direct comparison with Kano's generative approach. For both approahces we want to collect the time (ms), memory at the start of the method (mb), the highest memory peak during execution of the method (mb) and the difference between the peak and start memory (mb)

Each event type will be tested in 5 different setups which will be identical for each event and are described in Table 4.1. These parameter values are based of the parameters used in the evaluation of Kano **?**. However, our maximum number of pods and policies are lower than Kano's evaluation due to the limits of our experiment cluster. We will now summarize the meaning of each parameter:

- *Pod num*: This variable describes the amount of pods that will be generated and deployed on the cluster. Each pod has a single container thus this variable directly annotates the amount of containers as well.
- *Pol num*: This variable describes the amount of network policies that will be generated and deployed on the cluster.
- *key limit*: This variable describes the amount of distinct keys out of which one will randomly be selected for each time a key-value label is generated. The lower this number is the higher the chance of matches between containers and network policy label selectors.
- *Value limit*: This constant describes the amount of distinct values out of which one will randomly be selected each time a key-value label is generated.
- *Pol select limit*: This constant describes the amount of selector fields in a network policy.
- *Pol select label limit*: This constant describes the maximum amount of label selectors in a selector field of a network policy. each selector field will thus have between 1 and this value's amount of randomised label selectors.
- *Pol allow limit*: This constant describes the maximum amount of allow fields in a network policy. each network policy will thus have between 1 and this value's amount of allow fields.
- *Pol allow label limit*: This constant describes the maximum amount of allow selectors in a allow field of a network policy. each allow field will thus have between 1 and this value's amount of randomised label selectors.
- *Pod label limit*: This constant describes the maximum amount of labels a container. Each container will thus have between 1 and this value's amount of randomised key-value labels.

Name	setup 1	setup 2	setup 3	setup 4	setup 5
Pod num	50	100	250	500	750
Pol num	20	50	100	200	300
Key limit	2	5	8	10	20
Value limit	10	10	10	10	10
Pol select limit	1	1	1	1	1
Pol select label limit	3	3	3	3	3
Pol allow limit	3	3	3	3	3
Pol allow label limit	3	3	3	3	3
Pod label limit	5	5	5	5	5

Table 4.1: Experiment 1 parameter values

4.2.2 Execution

We will now describe the execution of a single sub-experiment step by step.

STEP 0: Call the python file experiment1.py with the following arguments: *number_of_runs*, *number_of_pods*, *number_of_policies*, *namespace*, *key_limit* and *event_type* according to the sub-experiment settings. The algorithm will them execute STEP 1 to STEP 7 as many times as defined in argument *number_of_runs*.

STEP 1: We fully reset the namespace defined in the *namespace* argument with the help of the Python Kubernetes API. This is coded in a separate delete.py file and extended with some extra tests and timeouts to guarantee that all objects are successfully removed before continuing.

STEP 2: We deploy as many pods and network policies as defined in arguments *number_of_pods* and *number_of_policies*. For this functionality a separate deploy.py python file has been created that will generate and deploy the network policies and pods, when called upon with the variables *namespace* and *key_limit* as parameters. The other relevant constants in Table 4.1 are hard coded since they don't change between sub-experiments. The deploy file is equipped with a list of distinct keys and values to leverage when creating the randomised pods and NPs. However, the key list is first shortened until it has a length equal to the *key_limit* parameter before being utilised. The randomised pods and network policies get deployed in the namespace defined in the *namespace* argument with the help of the Python Kubernetes API. We use extra tests and timeouts to guarantee all object are successfully deployed and ready before continuing.

STEP 3: We start the watcher with the flags for debug mode, verbose mode and start checkup all set to False. With the use of threading events we guarantee that the threads that watch the API's for pods and NPs are successfully running, in order to prevent the next step from executing too early, therefore missing the event and being unable to handle it.

STEP 4: We execute one event, depending on the *event_type* argument. If it is a delete event we call the delete.py file again and let it randomly remove one of the existing objects that correspond to the event type. If it is a deploy event we call the deploy.py file to generate one more randomly generated object according to the correct parameters.

STEP 5: We once again leverage threading events to get notified when the consumer thread has received and handled its first event. Since the watcher was initialized after all pods and network policies were fully ready it will always be the event from step 4 that will be caught. When the consumer catches the event it will start the timer right before calling the analyzer for event handling. Once the analyzer returns the function call the timer immediately gets stopped to get a final execution time. Similarly we measure the memory usage at the start of the algorithm, and the highest peak of memory usage during the event handling. With this we can calculate the difference and thus how much memory was used (in bytes). These memory and time measurements get stored in variables in the watcher.

STEP 6: Once we get the message that the event is handled we retrieve the measurements from the watcher and store it locally. We can then stop the watcher.

STEP 7: We now call retrieve all existing pods and NPs in the cluster and put them in lists. Once we have started a new timer and started tracing memory we give these lists to the original matrix generation method from Kano. Upon return of the reachabilitymatrix the timer and memory measurement are stopped and stored in variables. We also include a comparison between the incremental and generative reachabilitymatrices to ensure the results are correct and no bugs or edge-cases were discovered. If the latter would be the case the sub-experiment is cancelled and debug information printed out.

STEP 8: Once the hundred runs of the sub-experiment have finished we store all the data in a CSV file and store it on the control-plane node that ran the experiment. We can then retrieve this file and combine them with other sub-experiment files for evaluation.

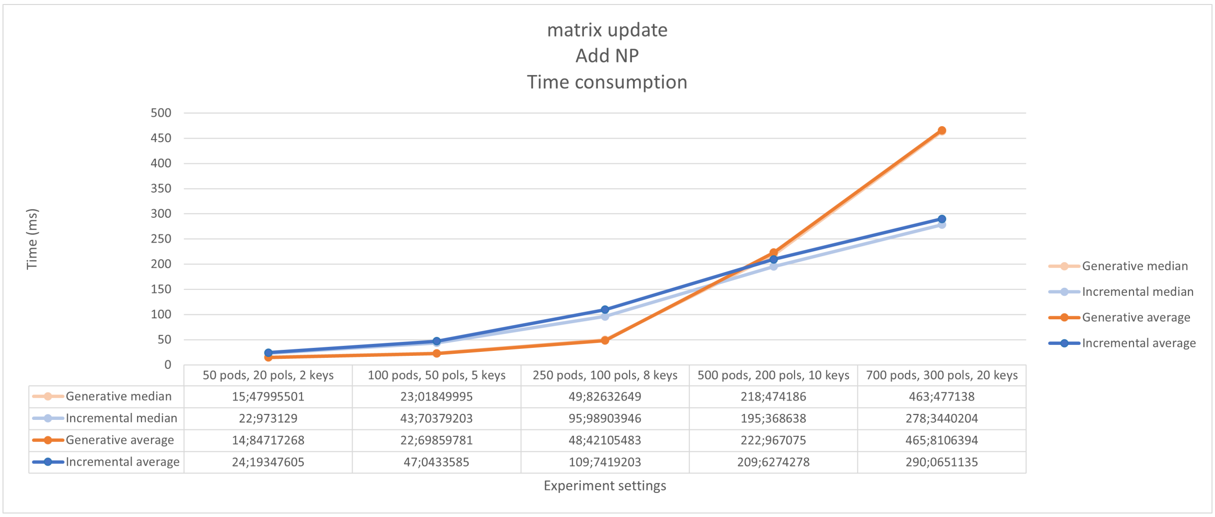
4.2.3 experiment results

The results of the experiment will be divided into eight different graphs: for each event type we created a graph displaying average and median time as well as a graph that shows average and median memory consumption. All the graphs use the same values and increments on their axis to allow a direct comparison. We will start by looking at the time graphs after which continue with the memory results.

Time consumption
When looking at the graphs in Figure 4.2, Figure 4.3, Figure 4.4 and Figure 4.5 we can

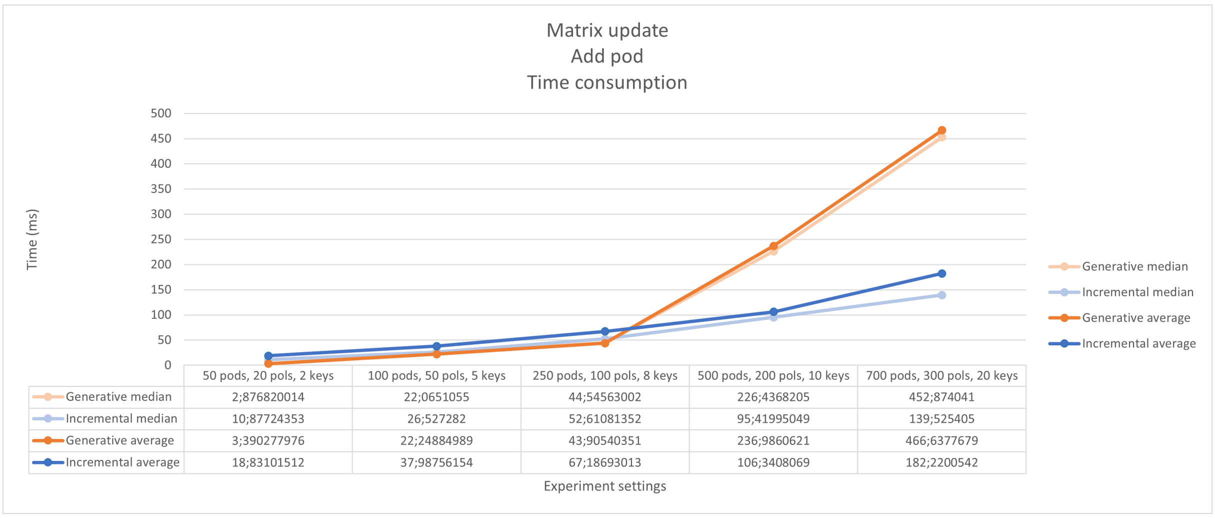
quickly deduce that our incremental approach starts out more time expensive but becomes faster as the scale of the cluster increases. This trend seems to continue as the cluster grows and is in line with our hypothesis for Q1 described in section 4.2. We will now take a deeper look into the data and describe some additional observations:

- If we look at the intersection point of the generative and incremental averages we see a difference between NP and pod events. For the network policy events the intersection always appears between the third and fourth sub-experiment, while for the pod events this is closer the the third sub-experiment. This can be explained due to the different approaches: when a NP is added or deleted all the pods with the corresponding labels must be retrieved, while a pod addition or deletion needs to find the corresponding network policies. Since our experiment settings always have more pods than network policies the search time for network policy events naturally becomes higher.
- Thanks to the included median values we see that the deletion events have less outlying values compared to the creation events. This can be explained once again by the approach to handle these events: when creating a pod or NP the time will be influenced by the number of matching label selectors between pods and network policies, which is randomised for each run. When deleting an object we do not need to search for a match between selectors, instead we simply remove the information from the cluster state, thereby saving on time.
- Although the generative approach from Kano stays consequent throughout the four events our solution is more dependant of the type of event being handled. The incremental approach stays in the similar range of values for the first sub-experiments, but as the cluster size increases the pod events generally take less time to update the reachabilitymatrix compared to the similar NP event.
- Although update events for containers and NPs are not measured for reasons stated earlier in this chapter we can try to make some assumptions about them. Since the method for handling update events equals to executing the deletion and then creation of the object in question, we can add these separate values together for an estimate. With the fifth sub-experiment setting the incremental approach already outperforms the generative for updating pods: the average time for adding a pod ($\sim$182 ms) combined with the average time for deleting a pod ($\sim$209 ms) is still lower than the time it takes for the generative approach ($\sim$466 ms). Additional testing in future work would be required to deduct more conclusions about this, but it is reasonable to assume that the trend continues and that there will be a cluster size for which the incremental approach intersects with the generative approach for update events, after which the incremental approach outperforms the generative for larger cluster sizes.



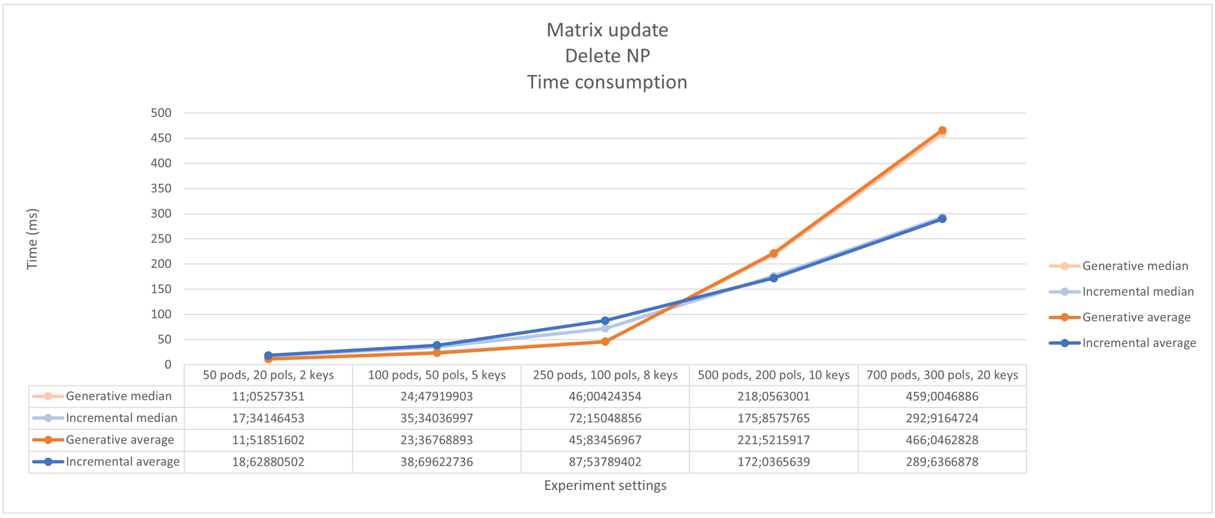
	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	15;47995501	23;01849995	49;82632649	218;474186	463;477138
Incremental median	22;973129	43;70379203	95;98903946	195;368638	278;3440204
Generative average	14;84717268	22;69859781	48;42105483	222;967075	465;8106394
Incremental average	24;19347605	47;0433585	109;7419203	209;6274278	290;0651135

Figure 4.2: Time consumption of adding a network policy



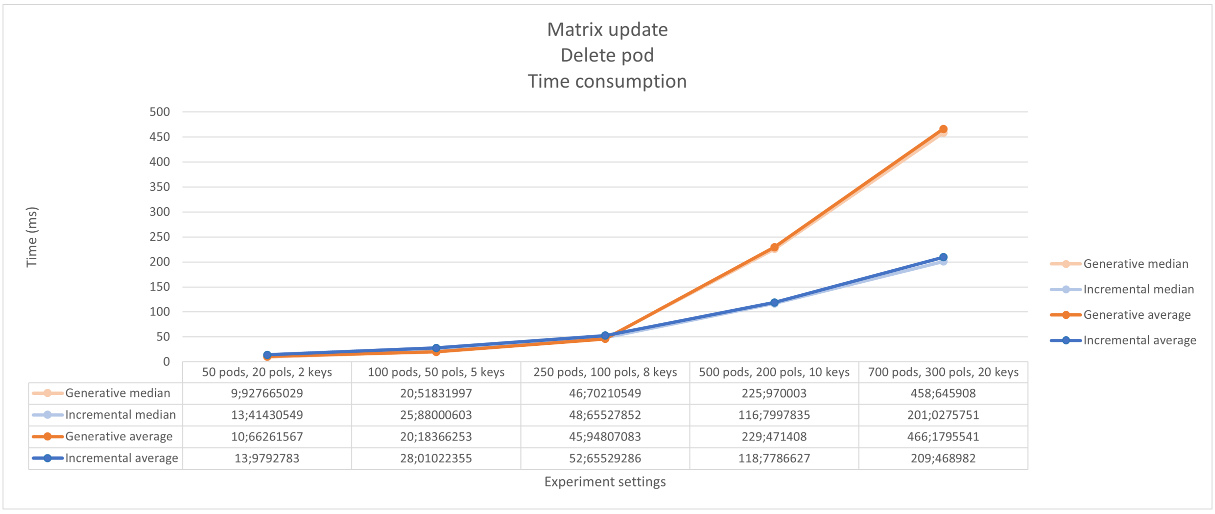
	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	2;876820014	22;0651055	44;54563002	226;4368205	452;874041
Incremental median	10;87724353	26;527282	52;61081352	95;41995049	139;525405
Generative average	3;390277976	22;24884989	43;90540351	236;9860621	466;6377679
Incremental average	18;83101512	37;98756154	67;18693013	106;3408069	182;2200542

Figure 4.3: Time consumption of adding a pod



	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	11;05257351	24;47919903	46;00424354	218;0563001	459;0046886
Incremental median	17;34146453	35;34036997	72;15048856	175;8575765	292;9164724
Generative average	11;51851602	23;36768893	45;83456967	221;5215917	466;0462828
Incremental average	18;62880502	38;69622736	87;53789402	172;0365639	289;6366878

Figure 4.4: Time consumption of deleting a network policy



	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	9;927665029	20;51831997	46;70210549	225;970003	458;645908
Incremental median	13;41430549	25;88000603	48;65527852	116;7997835	201;0275751
Generative average	10;66261567	20;18366253	45;94807083	229;471408	466;1795541
Incremental average	13;9792783	28;01022355	52;65529286	118;7786627	209;468982

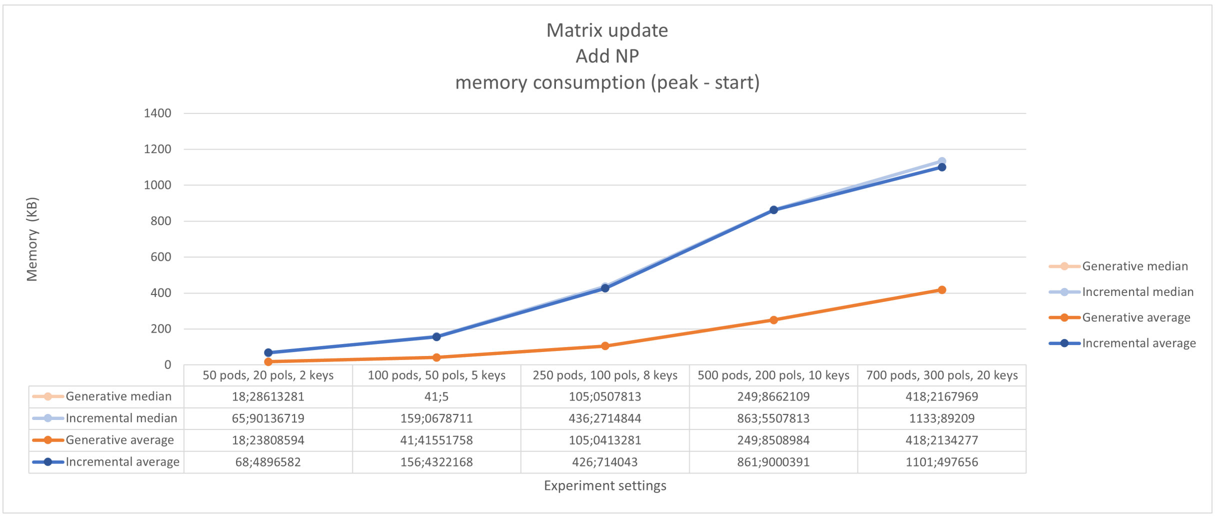
Figure 4.5: Time consumption of deleting a pod

Memory consumption

When looking at the graphs in Figure 4.6, Figure 4.7, Figure 4.8 and Figure 4.9 we can quickly deduce that our incremental approach is more expensive in terms of memory usage than the generative approach, with the difference only increasing as the cluster grows. This is in line with our hypothesis for Q2 described in section 4.2. Next we present some additional observations we can deduce from these graphs:

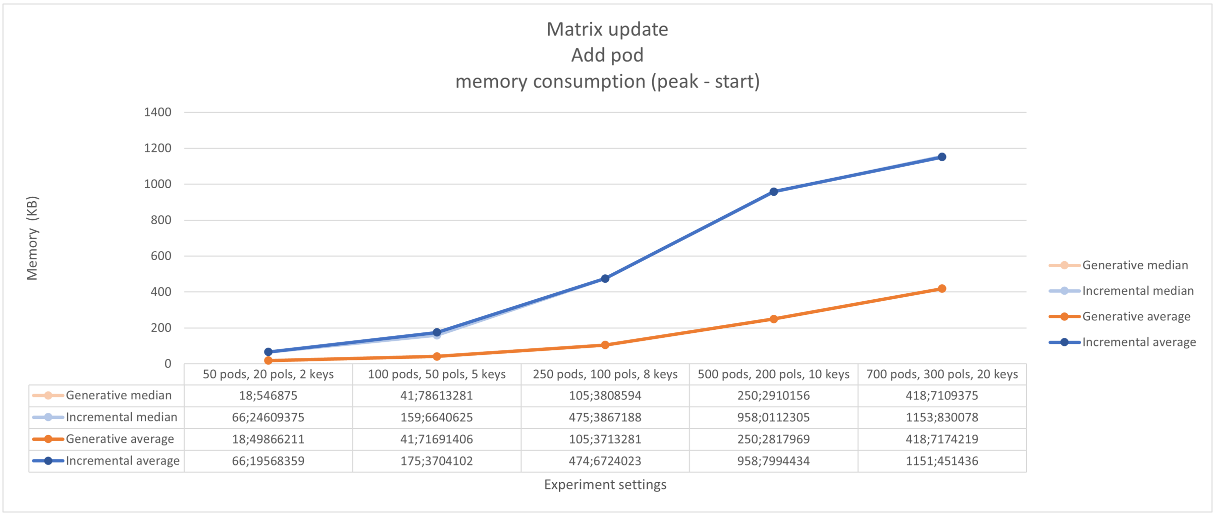
- With a small exception for the fifth sub-experiment in the creation event of a network policy, the median values are almost identical to the average values, meaning little skewing of data between the runs occurs. This makes sense since the cluster state that we store always has the same amount of cluster objects to keep track of within each run. The exception mentioned before might be due to a high occurrence of matching labels between objects which in turn increases the size of data structures such as the Store keeping track of NPs responsible for connections between pods. This remains an educated guess at best however.

- We can see that the ratio in which the memory increases for the incremental method decreases slightly between the fourth and fifth sub-experiment. We have no direct explanation for this, and further research would be necessary to deduct more out of this. Even with this decrease we can safely predict that our incremental solution will never be cheaper in terms of memory compared to the generative approach due to the fact that we have more data structures storing more expressive data values than the generative method.



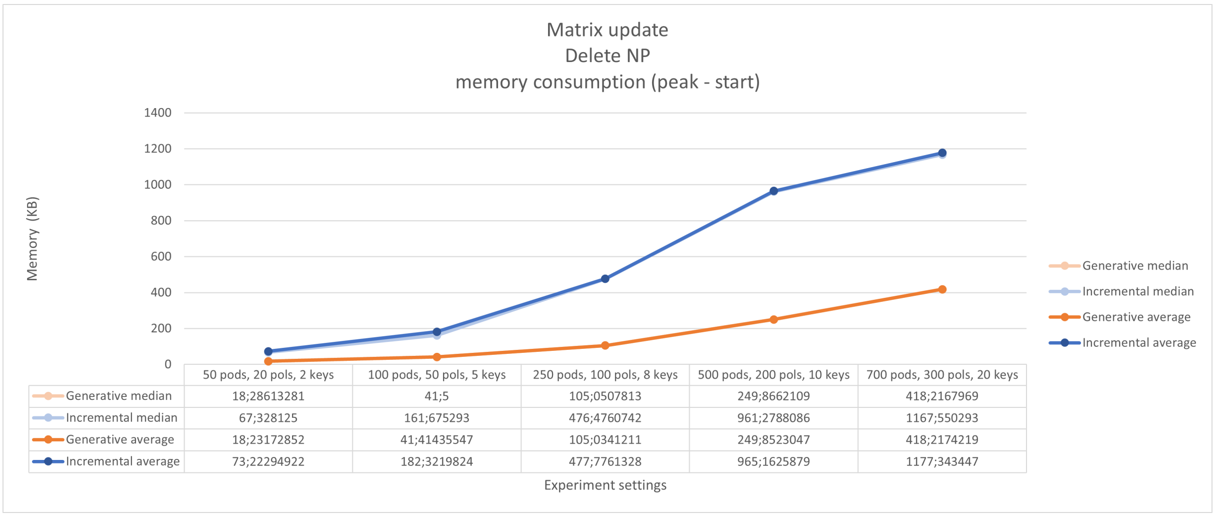
	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	18;28613281	41;5	105;0507813	249;8662109	418;2167969
Incremental median	65;90136719	159;0678711	436;2714844	863;5507813	1133;89209
Generative average	18;23808594	41;41551758	105;0413281	249;8508984	418;2134277
Incremental average	68;4896582	156;4322168	426;714043	861;9000391	1101;497656

Figure 4.6: Memory consumption of adding a network policy



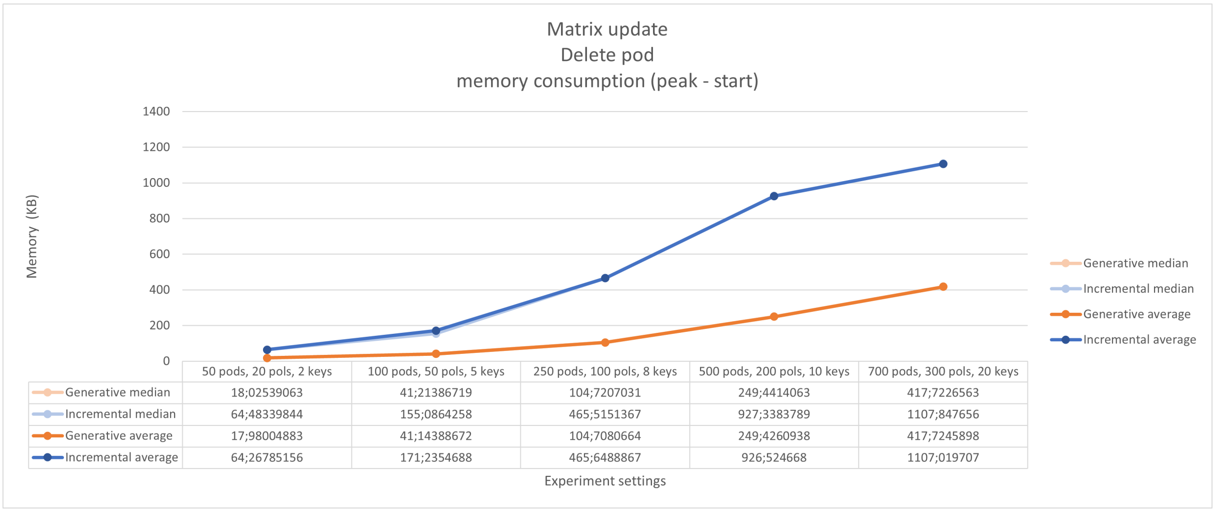
	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	18;546875	41;78613281	105;3808594	250;2910156	418;7109375
Incremental median	66;24609375	159;6640625	475;3867188	958;0112305	1153;830078
Generative average	18;49866211	41;71691406	105;3713281	250;2817969	418;7174219
Incremental average	66;19568359	175;3704102	474;6724023	958;7994434	1151;451436

Figure 4.7: Memory consumption of adding a pod



	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	18;28613281	41;5	105;0507813	249;8662109	418;2167969
Incremental median	67;328125	161;675293	476;4760742	961;2788086	1167;550293
Generative average	18;23172852	41;41435547	105;0341211	249;8523047	418;2174219
Incremental average	73;22294922	182;3219824	477;7761328	965;1625879	1177;343447

Figure 4.8: Memory consumption of deleting a network policy



	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	18;02539063	41;21386719	104;7207031	249;4414063	417;7226563
Incremental median	64;48339844	155;0864258	465;5151367	927;3383789	1107;847656
Generative average	17;98004883	41;14388672	104;7080664	249;4260938	417;7245898
Incremental average	64;26785156	171;2354688	465;6488867	926;524668	1107;019707

Figure 4.9: Memory consumption of deleting a pod

4.3 Experiment 2

The second experiment aims to give more insight into the overhead created by running our complete conflict detection algorithm on a cluster and will answer the following research questions:

- *Q3:* What is the relationship between pod/policy numbers and the time cost of conflict detection?
- *Q4:* What is the relationship between pod/policy numbers and the space cost of conflict detection?

Before we delve into the experiment approach, setup and results we declare our hypotheses for these questions:

- *Hypothesis Q3:* We predict that the time that the algorithm takes will be lower than the average deployment time of a pod, but might prove to be slower than the deployment time of a network policy since this has a very low initialisation period. The time cost of our algorithm will increase as the cluster size increases.
- *Hypothesis Q4:* We predict that the space cost of our algorithm will increase as the number of pods and network policies in the cluster increases as well.

4.3.1 approach

For this experiment we must use the entire algorithm instead of using a smaller part like we did in experiment 1. We can not compare our conflict detection however, since there is no other research solution that provides the same functionality to the best of our knowledge. Instead we measure generaral information about time and memory consumption when handling events. We once again use the same experiment setups as used in experiment 1 but must define additional parameters for the generation of the securiy groups and security group rules in the SGIC component. We will now describe the meaning of these additional parameters, while their values for this experiment can be seen in Table 4.2.

- *Nr of SG*: This constant describes the minimum and maximum amount of security groups that will be generated when initializing the SGIC. The exact amount of security groups is thus randomly selected between these values.
- *Nr of SG rules* : This constant describes the minimum and maximum amount of security groups rules that will be generated for each security group when initializing the SGIC. The exact amount of security group rules per SG is thus randomly selected between these values.
- *Nr of SG linked to node*: This constant describes the minimum and maximum amount of security groups that will be linked to a node. The exact amount of linked security groups is thus randomly selected between these values and is different for each node.

Lastly, we would like to mention that the variables within the security groups and security group rules are randomised as well. There is a 3/5 chance that a SG rule applies to a security group and a 2/5 change that it will specify an IP address. These IP addresses are randomised between 10 different values out of which 7 are the nodes on the cluster to increase the chance of matching rules. For more information we would like to refer to the SGIC implementation itself.

Name	setup 1	setup 2	setup 3	setup 4	setup 5
Pod num	50	100	250	500	750
Pol num	20	50	100	200	300
Key limit	2	5	8	10	20
Value limit	10	10	10	10	10
Pol select limit	1	1	1	1	1
Pol select label limit	3	3	3	3	3
Pol allow limit	3	3	3	3	3
Pol allow label limit	3	3	3	3	3
Pod label limit	5	5	5	5	5
Nr of SG	6-16	6-16	6-16	6-16	6-16
Nr of SG rules	3-5	3-5	3-5	3-5	3-5
Nr of SG linked to node	3-5	3-5	3-5	3-5	3-5

Table 4.2: Experiment 2 parameter values

The measurements retrieved from this experiment can be split in two parts: we analyze how long it takes for the watcher to get fully initialiazed and ready for capturing events, while also measuring the time and memory cost of our entire conflict detection algorithm in specific scenarios. For the first part of the experiment we only retrieve the time between calling the watcher and it returning the ready status (ms). The second part is more extensive as we collect the time between applying an event and detection by the watcher (ms), the time it takes for conflict detection (ms) and lastly the total time which is the combination of these last two (ms).

4.3.2 Execution

The approach for this experiment is very similar to experiment 1 but with small but very important changes included. We will now describe the execution of a single sub-experiment step by step where we will highlight the chances with experiment 1.

STEP 0: Call the python file experiment2.py with the following arguments: *number_of_runs*, *number_of_pods*, *number_of_policies*, *namespace*, *key_limit* and *event_type* according to the sub-experiment settings. The algorithm will them execute STEP 1 to STEP 7 as many times as defined in argument *number_of_runs*.

STEP 1: We fully reset the namespace defined in the *namespace* argument with the help of the Python Kubernetes API. This is coded in a separate delete.py file and extended with some extra tests and timeouts to guarantee that all objects are successfully removed before continuing.

STEP 2: We deploy as many pods and network policies as defined in arguments *number_of_pods* and *number_of_policies*. For this functionality a separate deploy.py python file has been created that will generate and deploy the network policies and pods, when called upon with the variables *namespace* and *key_limit* as parameters. The other relevant constants in Table 4.2 are hard coded since they don't change between sub-experiments. The deploy file is equipped with a list of distinct keys and values to leverage when creating the randomised pods and NPs. However, the key list is first shortened until it has a length equal to the *key_limit* parameter before being utilised. The randomised pods and network policies get deployed in the namespace defined in the *namespace* argument with the help of the Python Kubernetes API. We use extra tests and timeouts to guarantee all object are successfully deployed and ready before continuing. **We retrieve and store the current time and start the memory tracer**.

STEP 3: We start the watcher with the flags for debug mode, verbose mode and start checkup all set to False. With the use of threading events we guarantee that the threads that watch the API's for pods and NPs are successfully running, in order to prevent the next step from executing too early, therefore missing the event and being unable to handle it. **When the watcher returns that it is ready and monitoring the cluster we save the time and stop the memory tracer in order to store these for later evaluation**.

STEP 4: We execute one event, depending on the *event_type* argument. If it is a delete event we call the delete.py file again and let it randomly remove one of the existing objects that correspond to the event type. If it is a deploy event we call the deploy.py file to generate one more randomly generated object according to the correct parameters. **We save the time at which we send the API request to the Kubernetes API-server**.

STEP 5: We once again leverage threading events to get notified when the consumer thread has received and handled its first event. Since the watcher was initialized after all pods and network policies were fully ready it will always be the event from step 4 that will be caught. When the consumer catches the event it will start the timer right before calling the analyzer for event handling. Once the analyzer returns the function call the timer immediately gets stopped to get a final execution time. Similarly we measure the memory usage at the start of the algorithm, and the highest peak of memory usage during the event handling. With this we can calculate the difference and thus how much memory was used (in bytes). These memory and time measurements get stored in variables in the watcher.

STEP 6: Once we get the message that the event is handled we retrieve the measurements from the watcher and store it locally. We can then stop the watcher. **This includes the time at which the event was detected by the watcher. With all the time measurements that we collected we calculate some usable values such as the time between the Kubernetes API call and event detection, and the total time between the Kubernetes API call and completion of conflict detection**.

STEP 7: Once the hundred runs of the sub-experiment have finished we store all the data in a CSV file and store it on the control-plane node that ran the experiment. We can then retrieve this file and combine them with other sub-experiment files for evaluation. **Since there is no comparison in experiment 2 step 7 equals to step 8 of experiment 1**.

4.3.3 experiment results

5. Conclusion

Notes