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

© Copyright by KU Leuven

Without written permission of the promoters and the authors it is forbidden to reproduce or adapt in any form or by any means any part of this publication. Requests for obtaining the right to reproduce or utilize parts of this publication should be addressed to KU Leuven, Faculteit Wetenschappen, Celestijnenlaan 200H - bus 2100, 3001 Leuven (Heverlee), telephone +32 16 32 14 01.

A written permission of the promoter is also required to use the methods, products, schematics and programs described in this work for industrial or commercial use, and for submitting this publication in scientific contests.

Preface

This thesis marks the end of a long academic journey that would not have been possible without the continuous support of all the people around me to whom I would like to dedicate this section.

First of all, I would like to thank my promotors and mentor for their support and guidance throughout this journey. Specifically, I would like to thank Professor E. Truyen for his understanding, patience and continuous aid in applying his expertise every step of the way. Naturally, I want to extend my gratitude to my mentor, G. Budigiri, as well who was an immense help in finding my way around the many pitfalls of this thesis and gladly answered all my questions no matter how big or small.

I am forever grateful to my parents, whose love and support created the foundations of my academic pursuits. They gave me the opportunity to express myself and pursue this academic career, and I can say with certainty that this thesis would not have come to fruition without them. I also owe a debt of gratitude towards all my friends and family, whose words of encouragement in both good and challenging times never failed to motivate me. Finally, I would like to thank my girlfriend for being my source of strength. In the moments when self-doubt crept in, she never ceased to stand by me with an unwavering belief in my capabilities, making her my guiding light through the rough patches of this journey.

Finally, I would like to express my heartfelt appreciation to each and every person who has contributed, in their own unique way, to the completion of this thesis. Their impact, however small, has left an unforgettable mark on this work.

Jasper Goris

Abstract

Cloud-native development is an approach to building and deploying applications within containers in cloud environments, and has become an industry standard thanks to its many advantages such as cost efficiency, scalability, and automation. The cloud-native technology stack can be divided into many different layers with their own responsabilities, such as the code, container and cloud layer. For example, the container layer handles the deployment and management of containerized applications, while the cloud layer manages everything regarding the virtual machines upon which the containerized applications run.

These seperate layers are usually managed by specialized tools that provide their own security features. An often recurring security feature is the network communication rules, which restrict communication between components in order to prevent the spread of malicious attacks throughout the layer. However, conflicts may still arise when the communication rules of different layers are not aligned with each other, resulting in unreachable components or an unexpected attack vector for malicious attackers. Additionally, the alignment of these rules is hindered by the dynamic nature of cloud deployments, since components can be added or removed for the purpose of automatic scaling.

This thesis presents a conflict detection algorithm to verify network security rules between the cloud and cluster layer, specifically Kubernetes network policies and OpenStack security group rules. To do this, it leverages the reachability matrix introduced in the research paper of Kano, while trying to increase time performance by including an incremental update approach for this reachability matrix. The conflict detection is triggered by any event that can influence the connections between containers in the cluster, found by continuously monitoring the cluster. When such an event is captured our incremental approach is used to find any connectivity changes which are then verified against a mocked cloud layer of security group rules to find any newly introduced conflicts. We evaluate the proposed algorithm and compare our incremental approach to an existing generative approach of updating the reachability matrix. The results show that our incremental update approach proves to be faster when the cluster has increased enough in size with the drawback of extra memory consumption. Our entire conflict detection solution has proven to only add a little overhead in time on top of the incremental update approach. With a maximum average of 474ms in the biggest cluster size of our experiments, it has proven to be faster than a cold-start pod deployment.

Samenvatting

cloud-native ontwikkeling is een methode om container-applicaties te implementeren en uit te rollen in cloud-omgevingen in plaats van op on-site infrastructuur, en is een standaard geworden in de industrie dankzij de vele voordelen zoals kost-efficiëntie, schaalbaarheid en automatisering. De cloud-native technologiestapel kan opgedeeld worden in verschillende lagen met hun eigen verantwoordelijkheden, zoals de code, container en cloud laag. Bijvoorbeeld, de container laag behandelt het uitrollen en onderhouden van container applicaties, terwijl de cloud laag alles opvolgt in betrekking tot de virtuele machines waarop de container applicaties draaien.

Deze verschillende lagen zijn vaak beheerd door gespecialiseerde tools met hun eigen veiligheidsfunctionaliteiten. Een vaak wederkerende functionaliteit is de netwerkcommunicatieregels dat de communicatie tussen verschillende onderdelen beperkt om zodoende de verspreiding van malafide aanvallen te voorkomen. Desondanks kunnen conflicten nog altijd ontstaan wanneer de netwerkcommunicatieregels van de verschillende lagen niet op elkaar zijn afgestemd, wat resulteert in een onbereikbaar component of een onverwachte opening voor cyber-aanvallen. Bijkomend wordt het afstemmen van deze regels bemoeilijkt door de dynamische natuur van cloud deployments doordat componenten toegevoegd of verwijderd kunnen worden door de automatische schaalveranderingen.

In deze masterproef presenteren we een conflictdetectie-algoritme om netwerkbeveiligingsregels tussen de cloud- en clusterlaag, met name Kubernetes en Openstack, te verifiëren. Om dit te doen maken we gebruik van de reachabilitymatrix, geïntroduceerd in de publicatie van Kano. Tegelijkertijd proberen we de tijdprestaties van Kano te verbeteren door een incrementele update methode voor deze reachabilitymatrix te implementeren. Conflict detectie word getriggerd door de cluster continu te monitoren en de events te filteren die de verbindingen tussen containers in de cluster kunnen beïnvloeden. Wanneer zo een event wordt gevonden wordt de incrementele update methode gebruikt om eventuele connectiviteitswijzigingen te vinden, die vervolgens worden vergeleken met een nagebootste cloudlaag van security group rules om nieuw geïntroduceerde conflicten te vinden. We evalueren het voorgestelde algoritme en vergelijken onze incrementele update methode met de bestaande Kano methode. De resultaten laten zien dat onze implementatie sneller blijkt te zijn dan de kano's zodra de cluster een bepaalde grootte heeft behaald, met als nadeel extra geheugenverbruik. Met een maximum gemiddelde van 474ms in de grootste cluster setup van ons experiment bewijst onze conflict detectie methode sneller te zijn dan de gemiddelde cold-start container opstart tijd, terwijl hetslechts een kleine extra tijdskost bovenop de incrementele update methode introduceert.

List of Abbreviations

ABAC Attribute-Based Access Control. 13

AWS Amazon Web Services. 11, 18

BNP Bidirectional Node Policy. 59

CNI Container Network Interface. 8, 17, 37

FOL First Order Logic. 5

GCP Google Cloud Platform. 11

K8s Kubernetes. 1–3, 5, 8–13, 15, 17, 18, 20–22, 24, 28, 35, 37, 41, 48, 50, 51, 58–60

KIC Kubernetes Information Cluster. 21, 24–26, 28, 30, 35, 39

NAT Network Address Translation. 17

NP Network Policy. 2–6, 8, 13–15, 17, 20, 21, 23, 24, 26, 28, 30, 31, 37, 39–43, 46, 48, 50–52, 55–57, 59

OCI Open Container Initiative. 1, 2, 8–10

OS Operating System. 8, 10

SGIC Security Group Information Cluster. 21, 24–26, 35, 39, 49

VM Virtual Machine. 2, 3, 8, 17, 18, 37

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 and even mobile banking apps. 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 have therefore increased over time which in turn introduced entire new ranges of problems. One of these problems is how to cost-effectively scale the computational resources 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 [9]. This is not unexpected, since cloud native computing offers advantages such as scalability, resource efficiency and security [10]. Although cloud native computing is not a new phenomenon it seems to only grow in popularity [8] and is rapidly replacing old legacy systems and in-house servers. One of the key components to enable this growth are container formats such as Containerd, which are built upon the specifications of the Open Container Initiative OCI [11], an open governing structure dedicated to creating industry standards. These OCI compliant containers are a source of application portability by including the application code and all the dependencies required to run the code. Thanks to this there are many choices in so-called container-engines such as rkt [12], CRI-O [13], LXC [14] and Docker [2], which not only run the container but also manages all the required dependencies. Benefits of containers include, but are not limited to, the possibility of easily changing geographic location to ensure faster user connections and the possibility to autimatically scale the amount of deployments in order to serve more users.

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) [2] and Mesos [15] 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 standard container orchestration tool in the cloud community according to the 2022 Cloud Native Computing Foundation annual survey, as seen in the data excerpt shown in Table 1.1 [8]. A K8s cluster consists of many different components such as nodes, which are the highest-level component that run on physical or virtual machines and are responsible for either managing the cluster, running one or more pods, or both. Pods are the lowest-level component in K8s 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 K8s offers different security features that can be utilized to defend workloads against specific attack surfaces. One of these solutions is the network policies (NPs) that define rules for traffic flow between pods. These NPs work based on key-value labels to select the pods they are applied to and are intended to restrict communication between pods to counter the spread of malicious attacks from compromised pods [16].

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 [8]

> *"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 are passionate about solving problems using cloud native architectures and providing teams with self-service solutions which create a more inclusive, iterative process." [8]*

In order to work, the K8s nodes need to be deployed on servers or virtual machines that offer correct networking interfaces and computational resources. OpenStack allows us to easily orchestrate these VMs while offering automation of these aforementioned functionalities along with the possibility to add functionalities according to the needs of the cluster. One of these functionalities is the security groups and security group rules, which select VM instances based on assigned security groups or IP addresses to restrict communication between instances [7] [17]. The goal of the security group rules is comparable to that of the K8s network policies, although they work on a lower level in the technology stack to achieve it. Finding the balance between the right amount of restrictions in communication while leaving enough communication open to allow for the expected behaviour of the virtual machines is a daunting task, made even more difficult by the dynamic natures of instances and the K8s nodes.

1.2 Problem

K8s 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 [1]. The code and container layer can be secured with defensive programming guidelines [18] and specifications such as those of OCI [11]. However, they cannot safeguard against misconfiguration of the cluster and cloud layer's network rules. 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 features available such as NPs in K8s and security groups in Openstack that can try to mitigate these risks. These two examples are present in the container and cloud layer respectively [16] [19].

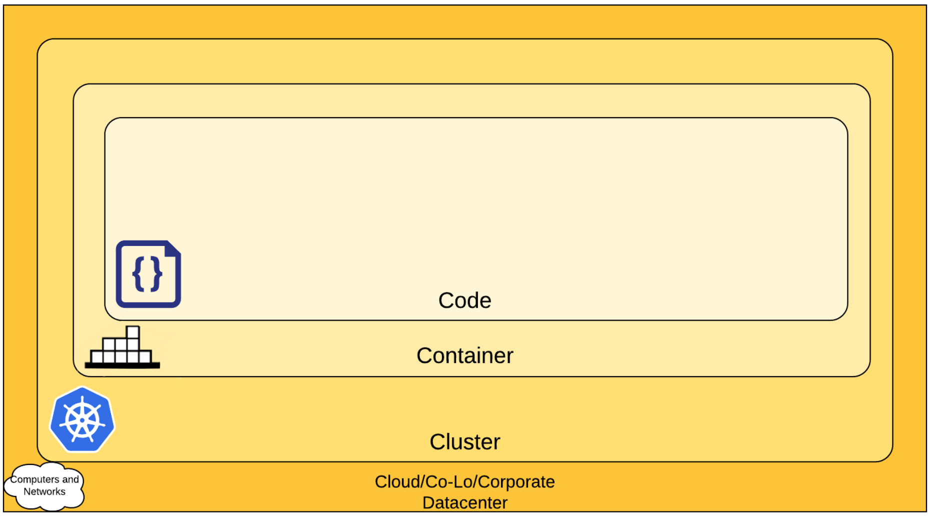


Figure 1.1: The 4C's of Cloud Native security (taken from source [1])

Although the configuration of the K8s NPs and their intended effect might remain consistent, their result might deviate from the expectations due to changes in the cluster state. K8s pods might be added, updated or removed and depending on the matches between its applied labels and existing NP label selectors this can result in unwanted or missing connections. Even when such unwanted network behaviour gets caught it might be hard to resolve due to the size of the cluster. The scale of this problem increases when we take into account that this same issue can happen when adding, updating or removing K8s NPs or OpenStack security groups and security group rules. Additionally, these different configurations in the separate layers of the 4C's security model can negatively impact each other. We illustrate this impact with an example.

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 NPs 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 [20]. 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 briefly describe some of these existing solutions that came closest to the desired properties of conflict detection

Grashopper [21] 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 NPs in the cluster layer. Conflicts are thus prevented instead of detected. However, it starts with the assumption that NPs 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 or the cluster layer is incorrect, only whether or not they are misaligned.

Kano [6] detects inconsistencies in NPs 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 a 1 in the 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 NPs. However, it does not look at the cloud 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 upon which we build our thesis. 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 or a NP. 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 mean an increase in efficiency. For a more in-depth description of Kano, we refer to section 2.4.

NFVGuard [22] 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 virtualized components. In the 4C's of K8s 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 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 [23] verifies network isolation between different tenants 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 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 [20] and zero trust [24] to prevent conflicts from existing, effectively making conflict detection redundant. Some notable examples of microsegmentation tools are Paloalto's PrismaCloud [25], Illumio core [26], cisco ACI [27] and VMWare NSX [28]. The goal of these tools is to limit the possibility of a malicious attack spreading through 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 monitors a cluster for changes in NPs and pod configurations upon which it will start conflict detection between communication states of K8s pods and OpenStack instances. Our algorithm is incremental in the sense that it constantly monitors the cluster layer for events that might change communication between components after which it will update the stored cluster state with that specific event. This new cluster state will then be used to detect new conflicts. Events in the cloud layer might affect conflicts as well but are out of scope for this thesis. Additionally, the algorithm offers a startup detection that will find any existing conflicts upon execution of the algorithm to ascertain whether or not the starting cluster state is conflict-free.

With this algorithm, we want to guarantee that all misconfigurations between the cloud and cluster layer are detected and reported in a timely fashion. We hope to provide conflict detection that is faster than the time it takes for the changes of the triggering event to become effective in the cluster. Additionally, we will check any affected components for redundancy upon events that decrease the number of connections in the cluster, such as NPs that have become redundant because their only matching pod got removed. This ensures that no redundant components are left to be forgotten, which could otherwise open up new attack vectors.

The main application strategy for our algorithm is two-fold: It could be integrated into a scheduler to guarantee that proposed pod placements do not introduce conflicts, or it could be used as a verification tool that constantly monitors for conflicts or is periodically executed so that operators know at which components to look to find and resolve conflicts.

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. Afterwards, 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 conflict detection with the help of pseudo-code descriptions of our implementation. chapter 4 is where we will evaluate our algorithm based on the four earlier-mentioned research questions by executing two main experiments. 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 K8s and its network policies. The focus then shifts to Kano: a system to cover container NP verification and directly afterwards we look at Calico, the Container Network Interface that allows nodes in our K8s cluster to communicate. 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 [2]. 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 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 than the hypervisor alternative since they offer a limited number of services. Containers are therefore more flexible and can scale according to the elasticity of demands. these differences between hypervisor and container deployments are shown in Figure 2.1.

One of the main reasons that containers have become the industry standard that they are today is the portability, scalability and ease of deployment on virtually any public cloud provider. This is all possible thanks to the efforts of the Open Container Initiative (OCI) [11], which was introduced by the Linux Foundation [29] and offers industry guidelines. OCI dictates in 3 specifications how to standardize your application so that it can be deployed in a container [30]. 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 facilitate the sharing of these images public and private registries can be utilised, which follow the OCI distribution specifications.

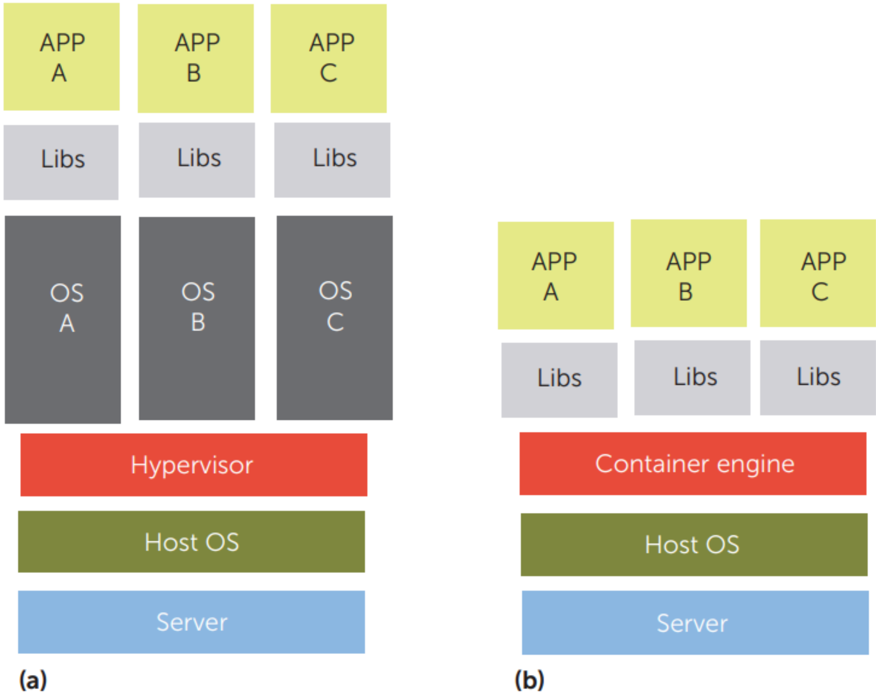


Figure 2.1: Comparison of (a) hypervisor and (b) container-based deployments (taken from source [2])

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 a result. For example, outsourcing cloud management to public providers enables lower costs due to economy of scale, where the location cost of the servers is divided between all the users that run their applications on them. 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 allows for quick deployments, compared to hypervisor deployments which require more manual configuration.

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 [31]. Container engines are the bridge between the end-user and the deployment of a container based on an image: they accept user requests and pull the images from the registry to 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 K8s (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 [32]. Figure 2.2 shows an example of the usage of Docker Engine for the 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.

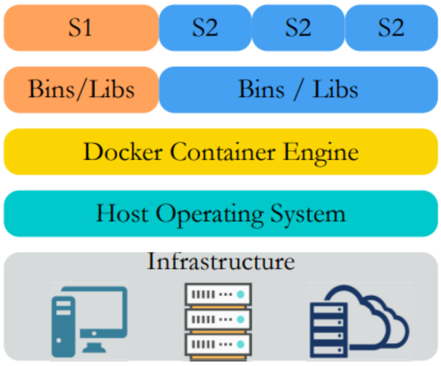


Figure 2.2: a container deployment using Docker Engine (taken from source [3])

When looking at the inner workings of Docker Engine we see that it is an open-source project build upon Moby[4]. 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 this does not yet explain how the image is run as a container, which brings us to container runtimes.

Container runtimes manage the life-cycle of containers and serve as the bridge between the host system and the container [33],[34]. 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 such as containerD [35], CRI-O [13] and Mirantis [36] adhere to OCI's runtime specifications which means they are interchangeable. Our container engine, Docker Engine, is designed specifically with containerd under the hood.

When more functionality is required than Docker Engine can offer such as load balancing, integration with Docker Compose or image vulnerability scanning, then we need to look at container orchestration tools. In this thesis, we will be using K8s, which will be

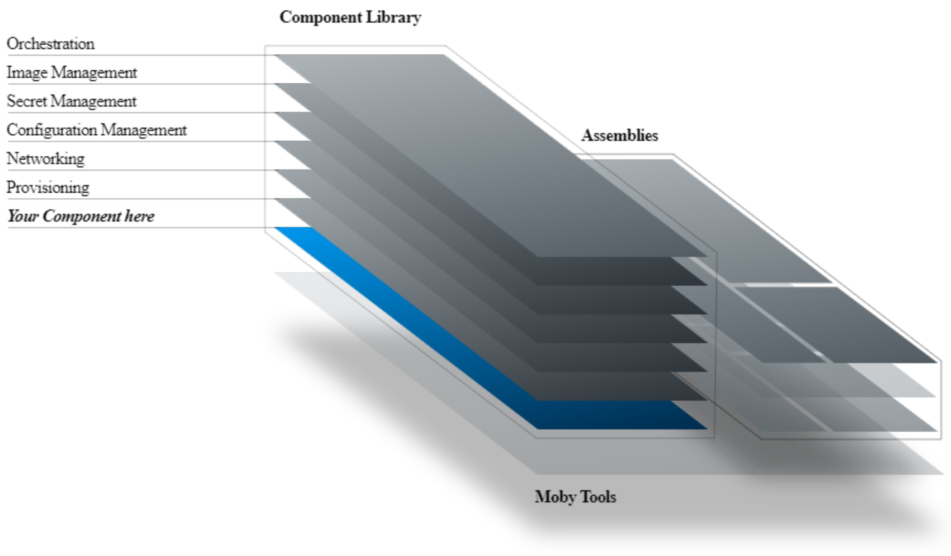


Figure 2.3: Moby framework (taken from source [4])

described in the next section.

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 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 [37], Apache Mesos [38], and the current industry standard K8s [3] [8]. This thesis will research and utilise K8s.

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) [39], Microsoft Azure [40] and Google Cloud Platform (GCP) [41]. K8s 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 of K8s 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 K8s components and concepts in the figure that are important for this thesis.

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

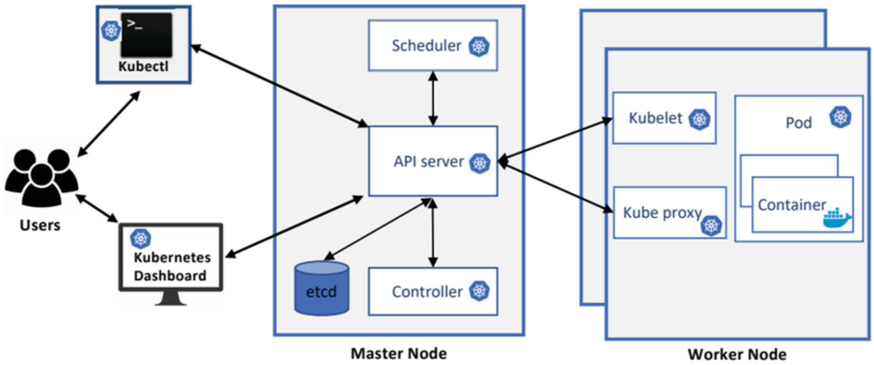


Figure 2.4: a brief overview of K8s (taken from source [5])

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. [42]

Pod: A pod is the smallest deployable computational component in a K8s cluster, runs on a node, and houses one or more containers that are tightly coupled. A pod remembers through 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 to increase an application's workload (horizontal scaling). To redirect requests to an application without any notable difference for the end-user the load-balancer is introduced. It will automatically redirect requests to the application with the least current tasks to ensure 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. [43]

Kubelet: Each node in a cluster runs its kubelet service component. The kubelet manages all the pods on the same node by deploying, monitoring and deleting them. The kubelet 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 K8s can ensure pod crashes get noticed and handled according to specifications. [44]

API server: The K8s API Server offers REST operations through which users can interact with the cluster. It takes user inputs from the command line interface Kubectl or a K8s dashboard and communicates the necessary changes to the other components.

```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

It also validates requests and configures the data that passes through 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 K8s library as the main method of communication with the cluster [45] [46].

Namespace: A namespace is a mechanism to separate groups of resources in a K8s cluster. Within a namespace each resource must have a unique name, although 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 [47] [48].

Labels: Labels are key/value pairs which allow NPs to select pods. Multiple labels can be applied to a single object, much like a Attribute-Based Access Control (ABAC) system where attributes are linked to object [49]. 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 K8s pod that has the label *color: blue* attached to it.

To retrieve all objects based on labels the use of label selectors is required. If multiple label 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. [50]

```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 NP specification

Network policy: A Network Policy (NP) is used to control network traffic on the IP address or port level. NPs are applied only to pods, but can use different types 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 NP 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 pod communication is allowed and egress policies that define to which pod 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: blue* in that same namespace.

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 NPs are required for pods to communicate. [51]

2.4 Kano

Although NPs offer extra security for a K8s cluster it also has its drawbacks. They need to be closely managed to ensure no contradictions are specified and that containers can connect to another container they rely upon. To combat these difficulties Kano was introduced by researchers at the Tsinghua University in Bejing.

Kano is a container NP verification tool presented in 2020 that, to quote the authors, solves the following problem: "In a container network, do the network policies violate the network constraints?" [6]. It does so by creating a matrix representing the container connections based on the existing 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 NPs 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 pods allowed by both an ingress and an egress rule. To save storage space and to allow quicker computations Kano does not 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 need to be generated often. A naive solution to generate 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 starts 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 number 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 that position has the corresponding label key applied. This is seen in Figure 2.9.

When searching the containers to which a NP applies we retrieve the bit arrays corresponding to the selector labels of a policy in the hashmap. By applying a bitwise AND opperation to these bit arrays we get a bit array with bits set for each container that

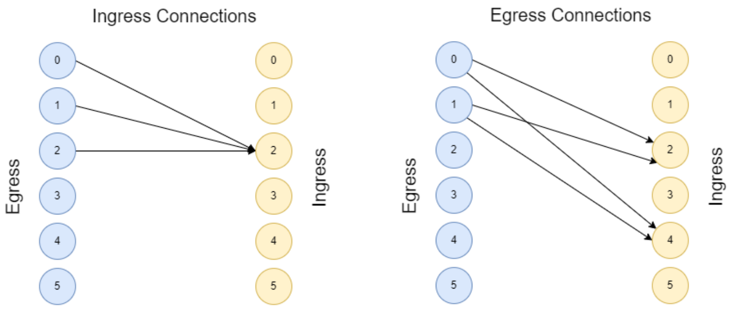
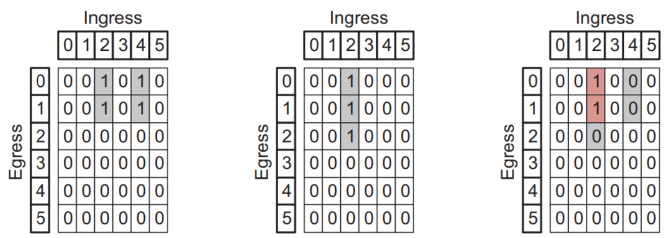


Figure 2.7: Bipartite graphs for Kano matrix generation



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

Figure 2.8: Container network reachability matrix model (taken from source [6])

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.

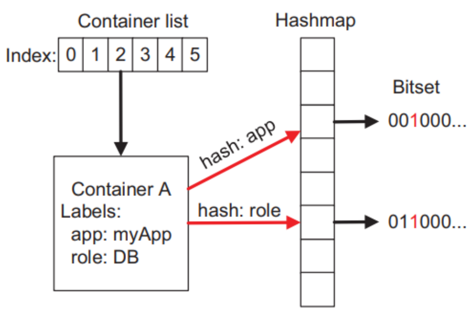


Figure 2.9: Prefiltration of container labels (taken from source [6])

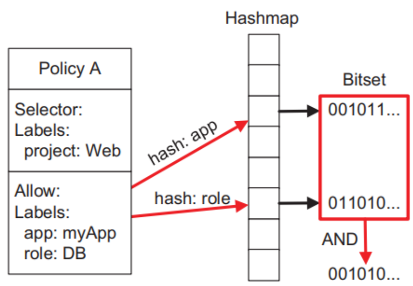


Figure 2.10: Prefiltration of policies (taken from source [6])

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 [6]:

- *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 another 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 contradict the connections built by another.

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

2.5 Calico

Pods, and by extension containers, in a K8s 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 unique IP address within the cluster. The latter option of two is called the K8s network model, and to achieve it a Container Network Interface (CNI) plugin is required. [52] [53]. 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 [54]. Many plugins based on the CNI project exist, such as Weave [55], Cilium [56] and Project Calico [57].

There are some rules that K8s 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 K8s 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 K8s cluster, but still allow freedom for the 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 K8s by using eBPF, a Linux kernel feature that allows you to run a VM inside the kernel itself [58]. This means that Calico does not have to rely on the default iptables-based Linux standard for network routing and can improve latency and performance with its solutions. Calico also enforces the K8s NPs and, although not used in this thesis, offers its own type of network security rules 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 [59].

2.6 Openstack

To host our nodes in a K8s 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 [8]. Some examples of cloud operating systems are Google Chrome OS [60], AWS [39] and OpenStack, which is our tool of choice for this thesis [61].

OpenStack automates the management of computing and network resources for clusters of physical and virtual machines, which can be easily directed through the dashboard application or directly via the API. OpenStack has great modifiability by allowing cluster administrators to extend their cluster with the components of their choice with example categories such as hardware life-cycle, storage, orchestration and application life-cycle. A VM deployed with OpenStack is called an instance, and in this thesis, OpenStack is used to deploy instances that host our K8s nodes. OpenStack also has network security solutions to manage connections between instances, which we describe more in-depth in the following paragraphs.

Security Groups Security Groups 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 their intended usage. Every instance has the $default$ security group applied if no custom 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 group's quota. [7] [62]

Security Group Rules Security group rules use IP filters 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 with 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). [17]

```
$ 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 [7]

3. Proposed Solution

In this thesis, we aim to create an algorithm that, once running on one of the control plane nodes of a K8s cluster, continuously watches the state of that cluster and detects any conflicts between the cloud and cluster layer. To achieve this we capture the six types of events 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. Additionally all pods and NPs affected by an event must get checked for redundancy. e.g. after the deletion of the last pod that matches a NP label selector it must report that that policy has become redundant. Lastly, it should be noted that changes in the cloud layer can influence connections as well (e.g. security groups being created) but this is outside the scope of this thesis.

pod	NP
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 NPs for each event and reduce the computation time. 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 the 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. Since the main purpose of namespaces is to isolate groups of resources within the cluster there should not be any communication between namespaces anyway which means conflicts can not exist [47]. 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 terms 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 K8s cluster with nodes deployed on an OpenStack installation. Figure 3.1 demonstrates how the algorithm is split up into separate parts, which correlates directly to different Python files with the corresponding names 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 K8s 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 its functionality is straightforward this component will not be described in depth in 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 will update incrementally with each captured event.
- *LabelTree*: A labelTree is a custom tree-like data structure used to store NPs by their selector labels, for quick retrieval by the KIC later on.

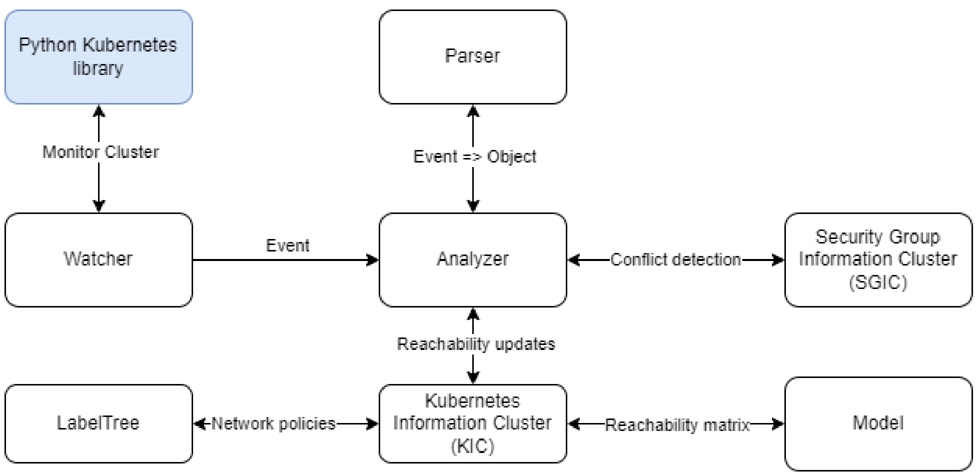


Figure 3.1: Algorithm structure

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:

- *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 select the grouping of pods to which the policy applies.
- *allow:* This data structure stores the labels that select the ingress sources or egress destinations.
- *direction:* The direction is a boolean that indicates whether 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 the name since it can be chosen and changed within the algorithm. The name on the other hand is defined by K8s 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, which is 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 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 section describing the reachablityMatrix structure.

ReachabilityMatrix: The reachabilityMatrix data structure stores multiple variables, which all get values assigned 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 [6]. 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 seperate pods, hence the name) as values, with an incrementing list of numbers as keys. This variable ensures 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 prefiltration algorithm in section 2.4.
- *resp_policies:* The resp_policies is a Store object that stores the responsible NPs for each container connection. The key-duo will represent the from- and to-container of the connection respectively, while the values in the corresponding list are a set of 2 values. The first value of this set is always the ingress rule, while the second is the corresponding egress rule.
- *matrix:* The 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 its dict, one for each connection. The connection between containers 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.

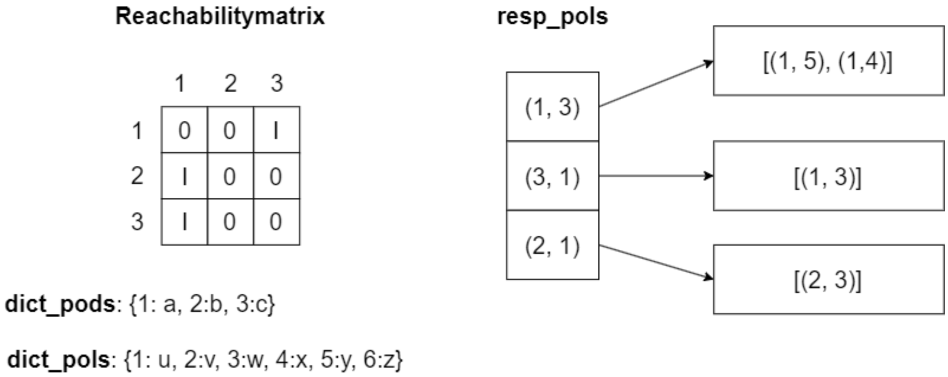


Figure 3.2: Example of a reachabilitymatrix and some corresponding variables

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 the 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 NPs on the cluster in the namespace that is defined in the arguments. To do this it will leverage the Python K8s 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 K8s events on the cluster by once again leveraging the Python K8s 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 K8s cluster, so in order to watch these APIs 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 APIs, 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 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. To achieve 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 for the existence of 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 in the figure as a single function call for any of the six events, it is actually a different function for each one, and these functions will be described in more detail in section 3.5. The rest of the analyzer's behaviour is dependent on 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 an 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 exists and is now removed (delete event). Lastly, we call upon the SGIC conflict detection, which is described in section 3.6.

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 NPs 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 NP 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 NPs 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 its constant data retrieval time due to the max tree depth of 3, and its intuitive representation.

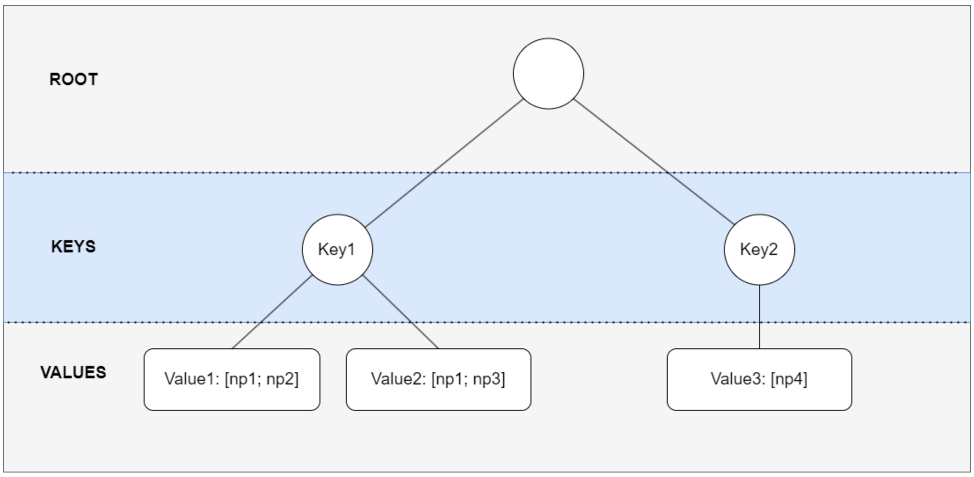


Figure 3.3: Structure of a Labeltree

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
```

3.5 Kubernetes Information Cluster

The K8s Information Cluster has two main functionalities: store all information about the current cluster state and update the cluster state given an event. 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 forn of a reachabilityMatrix 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 bit array of containers that have this label applied according to the label_map (see section 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 NP 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 with the execption of 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 methods 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 a 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 NP (Algorithm 4):** To delete a NP 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 bit array 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 exist.
- **add NP (Algorithm 5):** To add a NP 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 align we have 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 NP 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 do not 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 [19]. 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 K8s 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 security group rule number responsible for the connection. Thereafter both an ingress and egress graph get created and filled by going through 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 security group 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 with 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 can 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

```
function CHECK_SG_CONNECTIVITY(node1, node2, connection_wanted)
    print security groups for node1
    print security groups for node2
    if vmMatrix[node1, node2] == 1 then
        if connection_wanted == True then
            print security group rules responsible for connection
        else
            report conflict with responsible security group rules
        end if
    end if
    if vmMatrix[node2, node1] == 1 then
        if connection_wanted == True then
            print security group rules responsible for connection
        else
            report conflict with responsible security group rules
        end if
    end if
end function
```

4. Evaluation

This chapter aims to evaluate our algorithm described in chapter 3. This evaluation will be based on 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 describe in the first section. The second and third sections will talk about the first and second experiment respectively and include subsections about the experiments' approach, setup and results. We end with an overview of conclusions drawn from the experiments

4.1 Evaluation setup

The K8s 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 its 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 its OS. There are no Security Groups applied to the instances except for the default. Calico is used as CNI to provide the connection between the K8s nodes, which all run on the K8s Git version v1.22.17. K8s runs on default settings and specifies the maximum amount of pods per worker node at around 100 pods.

Each pod we deploy is created with 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 the 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 NPs according to its labels. However, we only measure four out of the six events that we can capture, since updating NPs 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 terms 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 NPs based on the reachabilitymatrix it generates when provided with a list of pods and a list of 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 the 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 NPs 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 adapted algorithm for the incremental approach now allows for a direct comparison with Kano's generative approach. For both approaches, 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 on the parameters used in the evaluation of Kano [6]. However, our maximum number of pods and policies is 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 number 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 number of NPs that will be generated and deployed on the cluster.
- *key limit*: This variable describes the number of distinct keys out of which one will randomly be selected each time a key-value label is generated. The lower this number, the higher the chance of matches between containers and NP label selectors.
- *Value limit*: This constant describes the number 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 allowed fields in a network policy. each NP will thus have between 1 and this value's amount of allowed fields.
- *Pol allow label limit*: This constant describes the maximum amount of allow selectors in an 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 in 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 then 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 K8s 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 NPs 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 NPs 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 NPs get deployed in the namespace defined in the *namespace* argument with the help of the Python K8s API. We use extra tests and timeouts to guarantee all objects are successfully deployed and ready before continuing.

STEP 3: We start the watcher with the flags for debug mode, verbose mode and start the checkup all set to False. With the use of threading events, we guarantee that the threads that watch the APIs for pods and NPs are successfully running, 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 NPs 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 is the case the sub-experiment is cancelled and debug information is 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 it 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

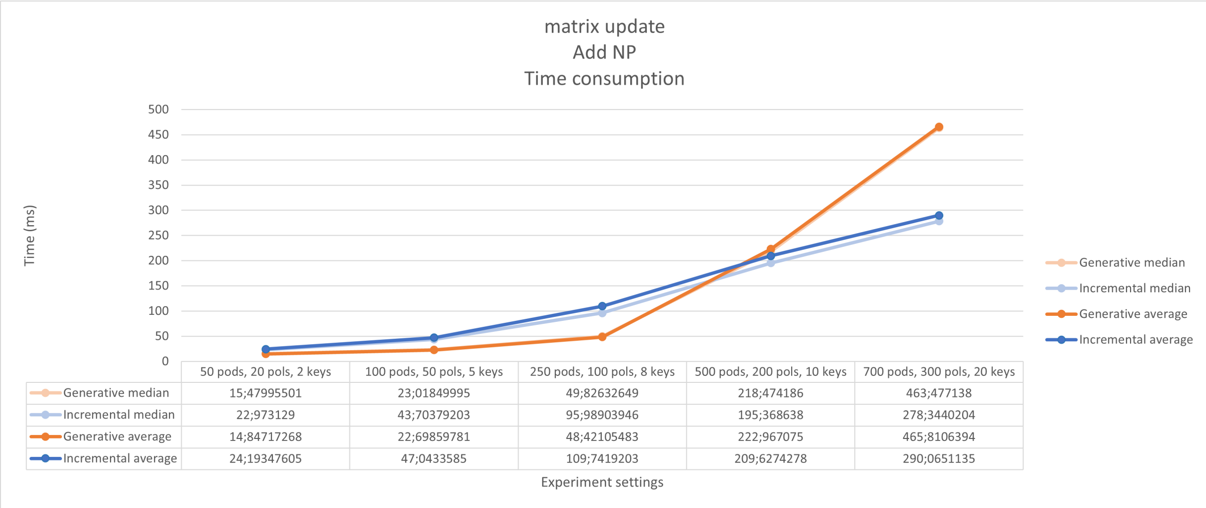
quickly deduce that our incremental approach starts more time expensive but becomes faster as the scale of the cluster increases. This trend seems to continue as the cluster grows and is in line with our hypothesis for Q1 described in section 4.2. We will now take a deeper look into the data and describe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

- Thanks to the included median values we see that the deletion events have fewer 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 time.

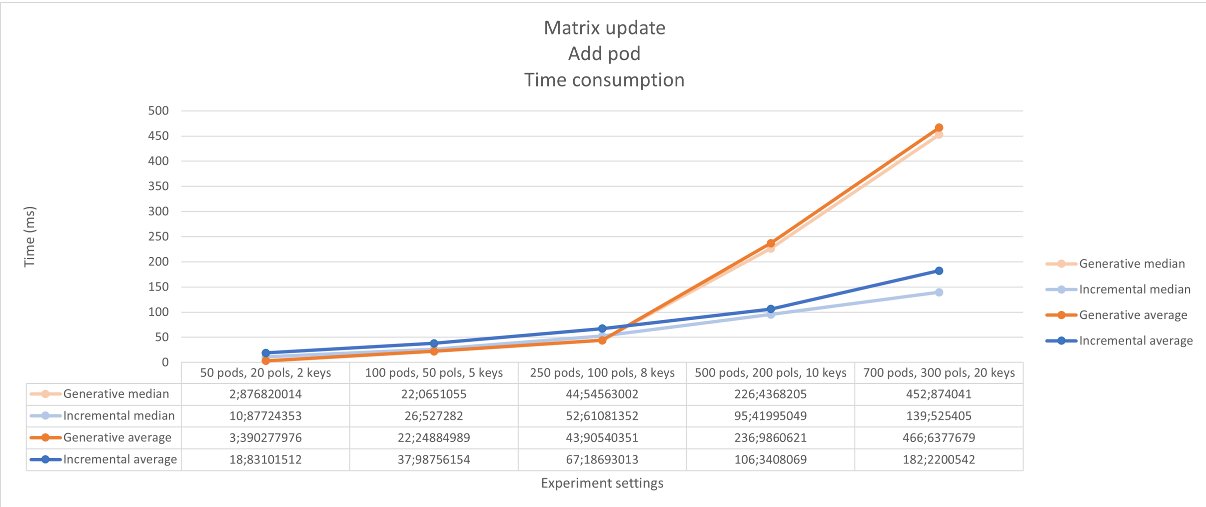
- Although the generative approach from Kano stays consequent throughout the four events our solution is more dependent on the type of event being handled. The incremental approach stays in the similar range of values for the first sub-experiments, but as the cluster size increases the pod events generally 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 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 draw 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 container

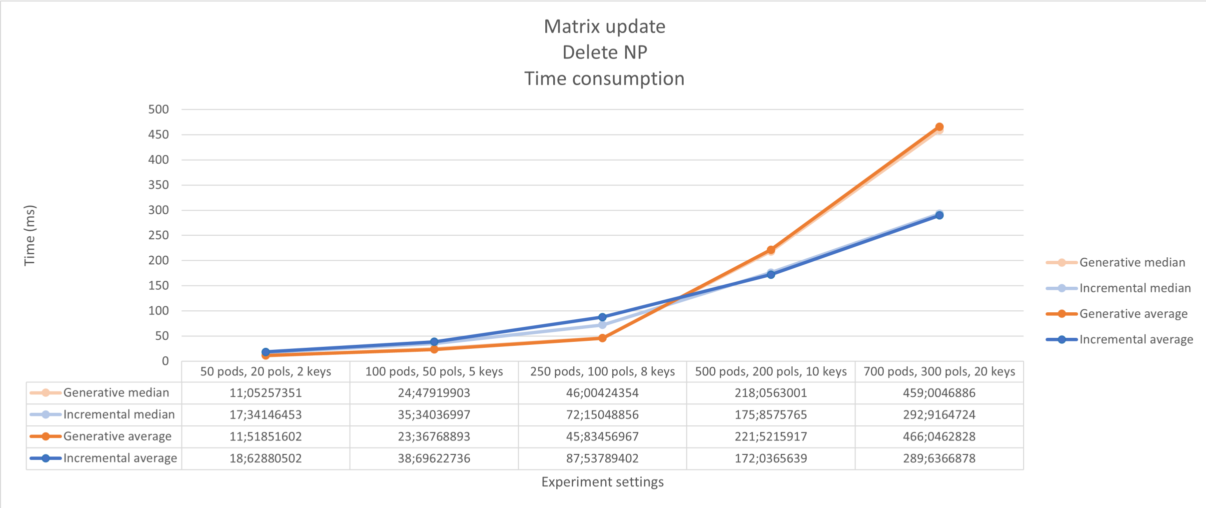


Figure 4.4: Time consumption of deleting a network policy

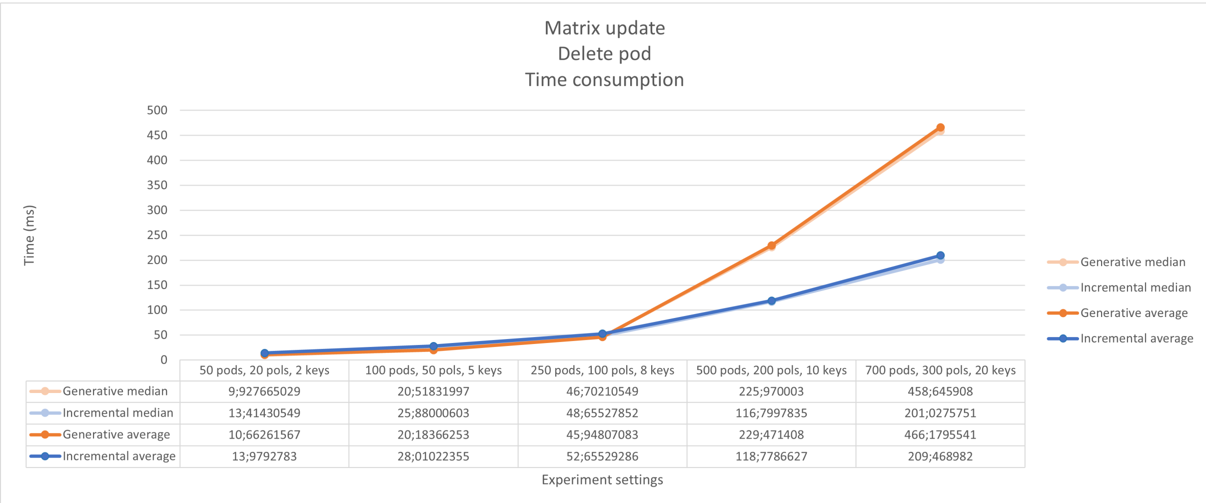


Figure 4.5: Time consumption of deleting a container

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. However, this remains an educated guess at best.

- We can see that the ratio in which the memory increases for the incremental method decreases slightly between the fourth and fifth sub-experiments. We have no direct explanation for this, and further research would be necessary to deduct more from this.

- Although the incremental approach has higher memory consumption it should be noted that the generative approach can not be used in its current form to execute conflict detection and would need adaptations to store more information. We can therefore not conclude that our incremental approach is strictly worse in terms of memory consumption than the generative approach.

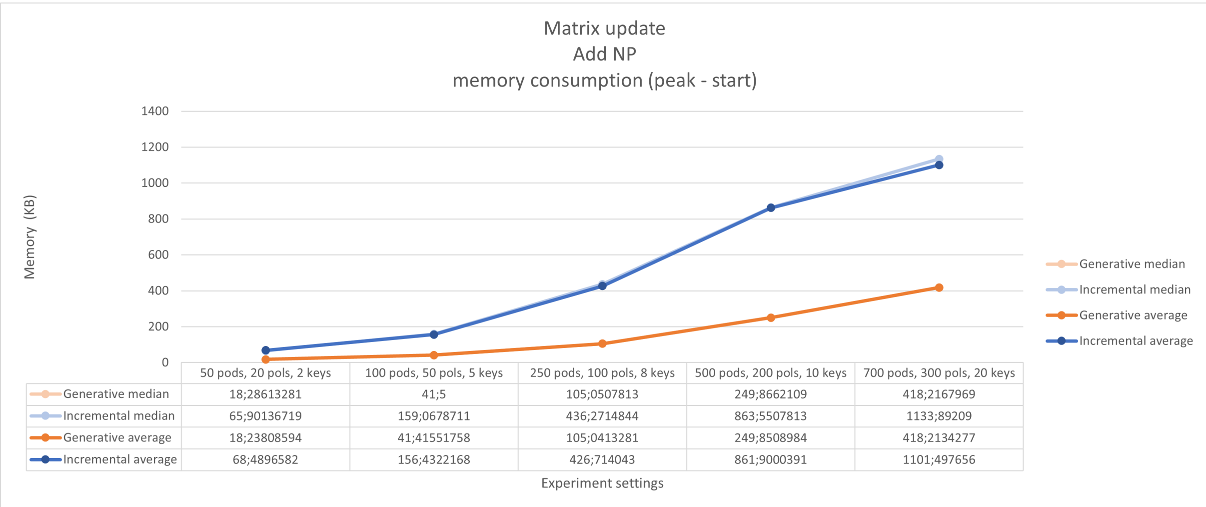
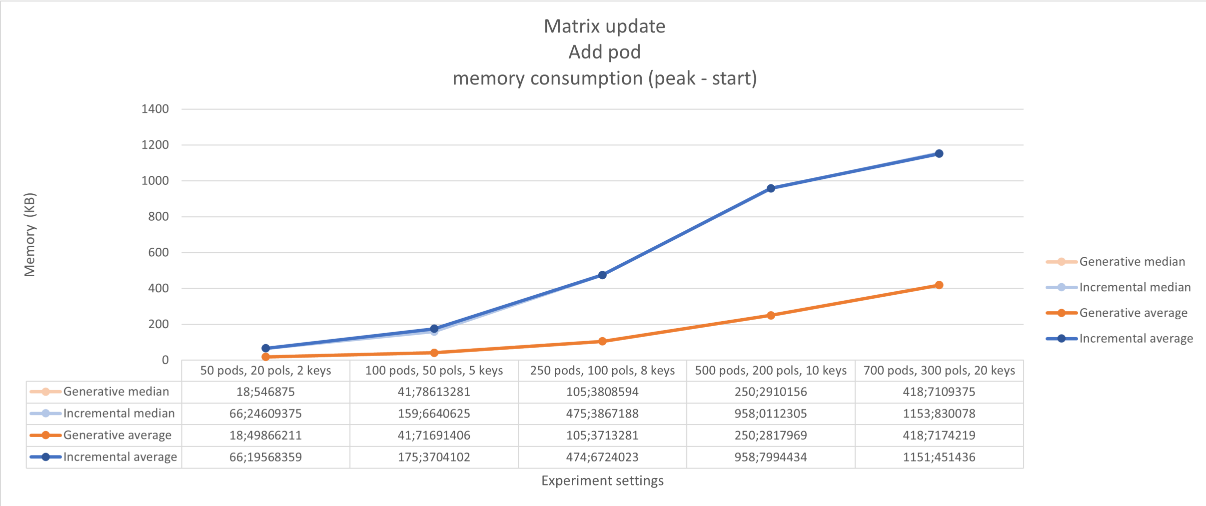
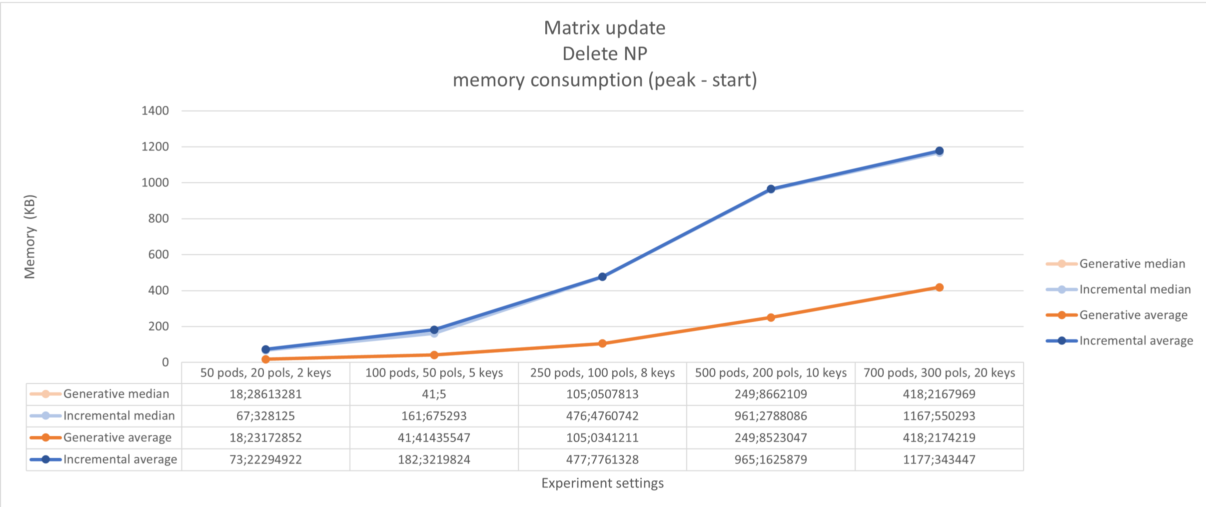


Figure 4.6: Memory consumption of adding a network policy



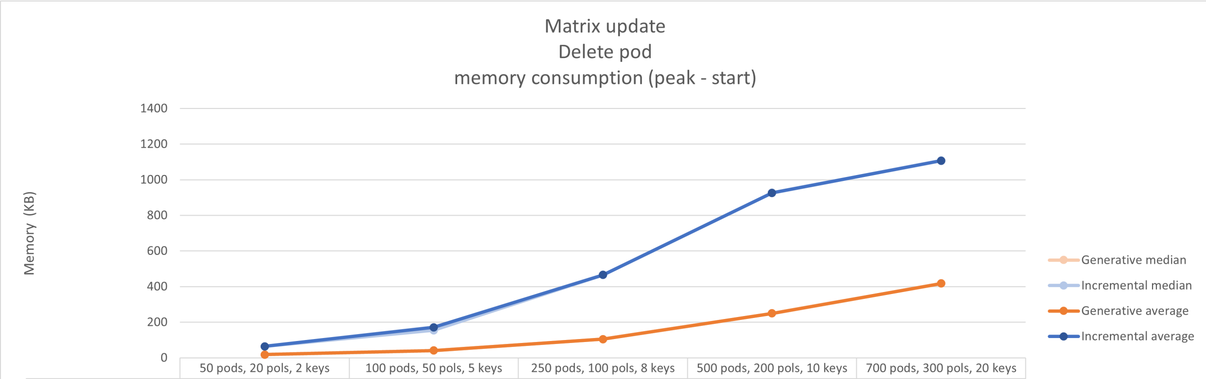
	50 pods, 20 pols, 2 keys	100 pods, 50 pols, 5 keys	250 pods, 100 pols, 8 keys	500 pods, 200 pols, 10 keys	700 pods, 300 pols, 20 keys
Generative median	18;546875	41;78613281	105;3808594	250;2910156	418;7109375
Incremental median	66;24609375	159;6640625	475;3867188	958;0112305	1153;830078
Generative average	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 container



	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 container

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 that 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 NPs in the cluster increases as well.

* In the research paper of Grasshopper they found that t during their experiments the deployment time of an application can be up to 4 seconds in case of a cold start [21]. Since there is no default startup time for a container in a K8s cluster we use this as a baseline.

4.3.1 Approach

For this experiment, we must use the entire algorithm instead of using a smaller part as 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 general 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 security 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 are thus randomly selected between these values.
- *Nr of SG rules* : This constant describes the minimum and maximum amount of security group rules that will be generated for each security group when initializing the SGIC. The exact amount of security group rules per security group 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 security group rule applies to a security group and a 2/5 chance that it will specify an IP address. These IP addresses are randomised between 10 different values out of which 7 are the IPS of the nodes on the cluster to increase the chance of matching rules.

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 into two parts: we analyze how long it takes for the watcher to get fully initialised 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 yet very important changes. We will now describe the execution of a single sub-experiment step by step where we will highlight the changes made in comparison 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 then 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 K8s 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 NPs 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 NPs 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 NPs get deployed in the namespace defined in the *namespace* argument with the help of the Python K8s API. We use extra tests and timeouts to guarantee all objects 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 the checkup all set to False. With the use of threading events, we guarantee that the threads that watch the APIs 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 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 K8s 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 NPs 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 K8s API call and event detection, and the total time between the K8s 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 it 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

Similarly to experiment 1 we created two graphs for each of the four captured events: one for the average and median time and the other for the average and median memory consumption. When comparing the graphs with experiment 1 we can see that there are only two data lines instead of four since there is only a single update method of the reachabilitymatrix used. Additionally, the graphs show data for the entire conflict detection solution instead of solely the update of the reachabilitymatrix. Aside from these 8 event graphs we also have two graphs regarding the time and memory consumption of the startup process of the watcher which is defined as the period between execution of the script up until the point at which the algorithm is readily monitoring the cluster. This startup data thus includes initialization of variables, reachabilitymatrix generation, cloud layer generation and setting up multithreaded API monitors. In these startup graphs, we do not separate the data per event since the event has not occurred yet when the algorithm is initialized and thus has no influence. Please note that the conflict detection graphs all use the same values and increments on their axis to allow a direct comparison, but the startup graph does not follow this promise: it shows the time in seconds instead of milliseconds. We will start by looking at the startup graphs after which we continue with the time and memory graphs for conflict detection.

Watcher Startup

The time and memory consumption for initialising the watcher can be seen in Figure 4.10 and Figure 4.11 respectively. The goal of these graphs is to give a general sense of resource consumption for the algorithm's startup phase, but we have no hard time limit imposed since the algorithm should in theory only be started once, and usually in a non-time-sensitive environment. We can deduct two interesting conclusions from the graphs:

- The median time deviates from the average time, which we attribute once again to the idea that the amount of connections in the cluster setup directly influences the speed of our solution. This difference increases with cluster size since more NPs and containers directly translate to a higher chance of container connection.
- The memory consumption shows no surprises nor does it show any significant difference between the average and mean values, indicating that our algorithm remains very stable. Interestingly enough this indicates that variables whose size depends on the number of allowed connections, such as resp_policies in the reachability structure (see section 3.1) do not have much influence on the total memory consumption.

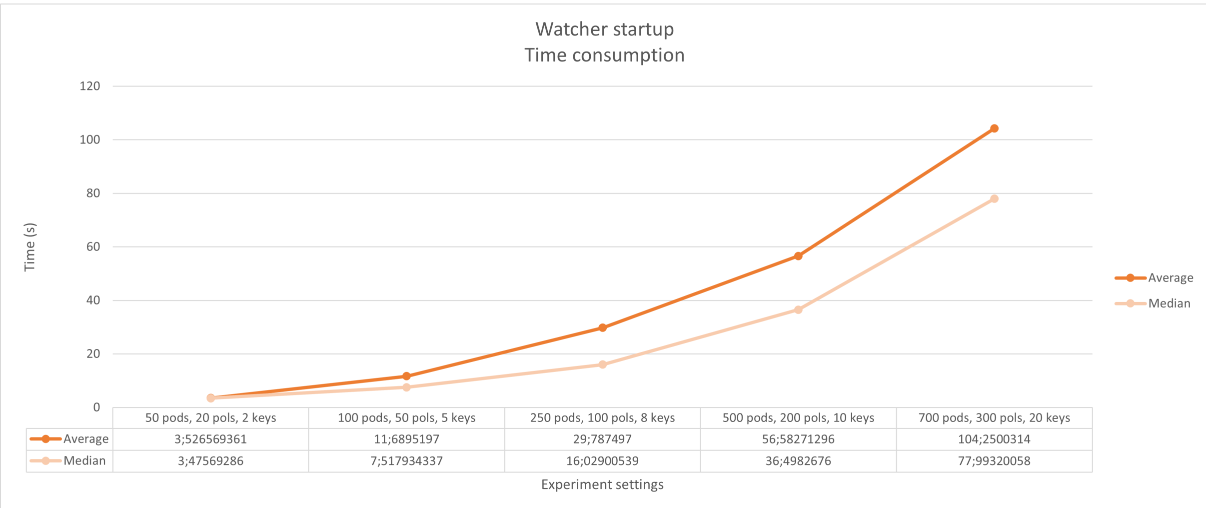


Figure 4.10: Time consumption of starting up the algorithm

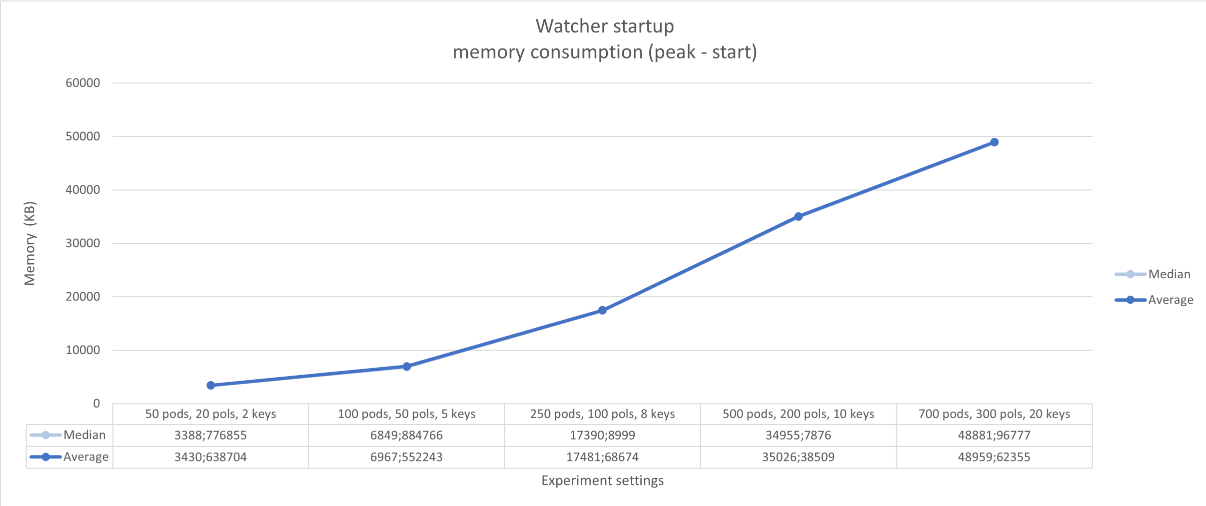


Figure 4.11: Memory consumption of starting up the algorithm

Time consumption

The time consumption for the full conflict detection solution for each of the four events are shown in Figure 4.12, Figure 4.13, Figure 4.14 and Figure 4.15. We can see that the conflict detection time remains safely below the 4-second pod deployment threshold stated in the hypothesis for Q3 with the highest average being 474ms for the deletion of a network policy. Furthermore, we can deduct some interesting information from the graphs:

- For each of the events and cluster setup combinations the median value lies lower than the average value, indicating the presence of high values that pull the average up. We once again assume this is related to the amount of container connections.

- When we compare the graphs for experiment 2 to the incremental approach of the corresponding event graphs in experiment 1 we see that the incremental approach for updating the reachabilitymatrix is the main contributor to time consumption, while the conflict detection only adds a slight overhead.

Figure 4.12: conflict detection time consumption - add network policy

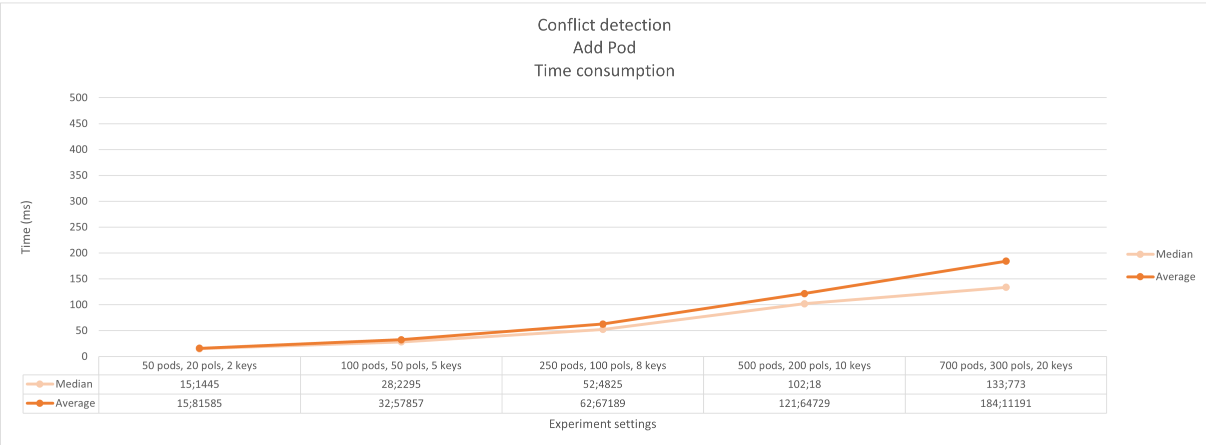


Figure 4.13: conflict detection time consumption - add container

Figure 4.14: conflict detection time consumption - delete network policy

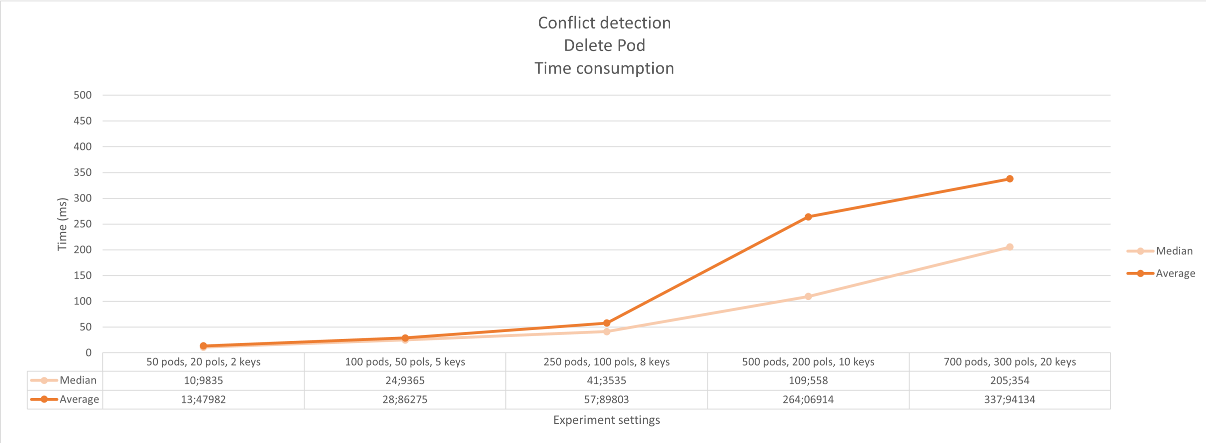


Figure 4.15: conflict detection time consumption - delete container

Memory consumption

The memory consumption for the full conflict detection solution for each of the four events are shown in Figure 4.16, Figure 4.17, Figure 4.18 and Figure 4.19. The average values are usually in line with the median values meaning little to no skewing of data occurs due to outliers. One exception is the addition of a NP where the median lies higher than the average, which was also visible in the corresponding graph in experiment 1. We, therefore, assume that this is due to the incremental update of the cluster state and not because of the conflict detection itself, although an exact reason could not be found and would require extra experimentation. We conclude that some small differences between events can be found but there are no specific outliers that catch the attention and that the memory increases as the cluster grows in size which is in line with our hypothesis for Q4.

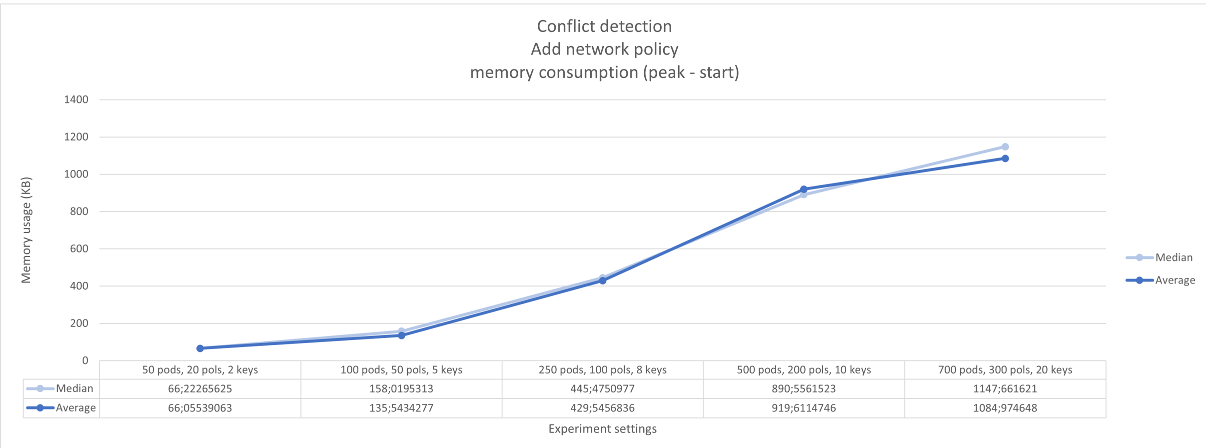


Figure 4.16: conflict detection memory consumption - add network policy

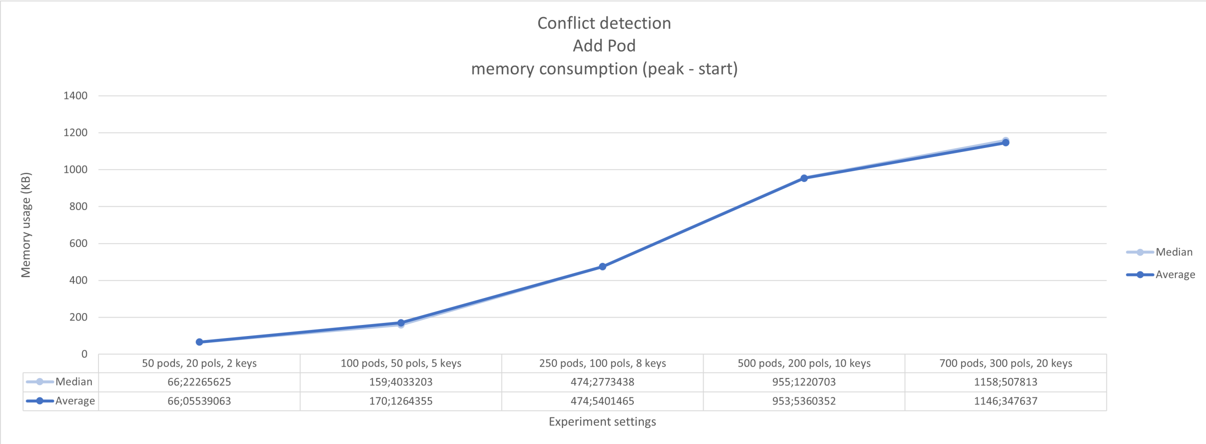


Figure 4.17: conflict detection memory consumption - add container

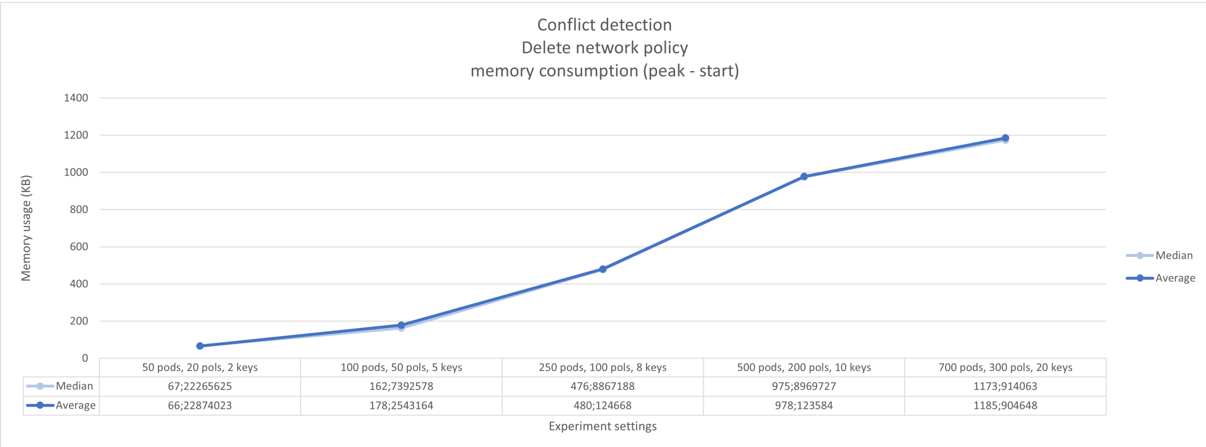


Figure 4.18: conflict detection memory consumption - delete network policy

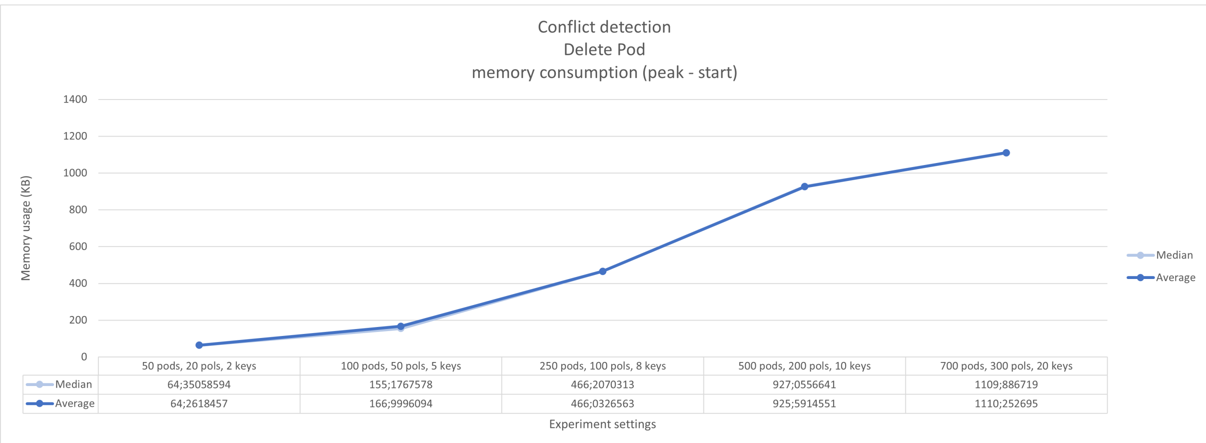


Figure 4.19: conflict detection memory consumption - delete container

4.4 Evaluation summary

We now combine the most important conclusions from these evaluation experiments into an overview:

- Our incremental approach for updating the cluster state becomes faster than Kano's generative approach when we pass a certain cluster size threshold. This threshold changes depending on the cluster setup and the number of matches between label selectors in NPs and labels in pods.
- The biggest drawback of the incremental update approach is the higher memory consumption due to the storage of the last cluster state compared to the generative approach. We must note however that the generative approach should be adapted to be usable for conflict detection in the algorithm and would require some extra variables which could lead to more memory usage as well.
- Updating the reachability matrix is the biggest contributor to time consumption whereas the conflict detection itself only adds slight overhead.
- Both memory and time consumption grow as the number of containers and NPs in the cluster increases. The biggest contributor to time consumption is the number of connections between containers that are defined by network policies.

5. Conclusion

5.1 Overview

In this thesis, we researched the K8s technology stack regarding network security and defined the possibility of conflicts arising due to misconfigurations between different layers of the stack. After we defined our problem statement we researched existing state-of-the-art solutions regarding these conflicts and identified a knowledge gap within the current solutions as well as a possibility for improvement for Kano, an existing research solution. We continued by leveraging Kano's ideas to implement our algorithm that detects and reports on these conflicts. The implementation uses an incremental approach of updating the last locally saved cluster state, triggered by the monitoring and capturing of events that influence the connections between containers.

In our evaluation we compared the incremental update functionality of our implementation against Kano's generative approach and concluded that the generative approach is faster than our incremental solution for smaller clusters, whereas our incremental approach outperforms Kano's when the cluster becomes sufficiently large. A drawback of our incremental approach is the increased memory usage due to the storage of cluster state information as well as extra data required for the conflict detection. In our second experiment, we evaluated the general time and memory consumption for our entire conflict detection algorithm which showed us that the time consumption average never crossed 474ms for the highest average thus making our algorithm faster than the average cold-start pod deployment.

5.2 Future work

We briefly mentioned a startup-verification method to detect already existing conflicts within the cluster, which is implemented but remains untested due to time constraints for this thesis. When the memory consumption of the implemented algorithm proves to be too excessive the generative approach combined with the conflict detection could be executed periodically instead in order to detect existing conflict within the cluster. Some adaptations should be made such as the API monitoring and the removal of variables that were only required for the incremental update approach.

Our solution only provides conflict detection and does not offer a conflict solution nor an automatic conflict resolution step due to the multiple-truth problem of the different layers. This research problem would prove for an interesting study.

The solution algorithm does not monitor cloud-level changes to the cluster but instead mimics it by generating randomised OpenStack security groups and their rules. Future work could include the continuous monitoring for events that influence inter-VM connections similarly to the implemented cluster-layer watcher.

5.3 Kano V2

During the creation of this thesis, the researchers responsible for Kano released a second paper which partially overlaps with our solution. Since a complete rework of the thesis was infeasible at the moment of the paper's release we decided to continue our heading. Additionally, we would not take into account their new ideas in order to reach our deadlines. As compensation, we introduce this section in which we will talk about the implications this new research paper might have on this thesis and its future work. To avoid confusion we will adapt the paper's annotation of nodes for objects that were called containers or pods in the previous parts of the thesis. [63]

Within the new Kano paper, which we will Kano V2, they improve their initial solution by introducing a more general ABAC-formed cloud-native NP verification method that replaces their previous more K8s network policy-specific approach. Additionally, they added incremental verification which only looks at affected parts of the network instead of regenerating the reachabilitymatrix. It is striking how similar our idea of the incremental update approach is to Kano V2's improvement even though these ideas were found independently. Both incremental approaches define the addition and deletion of NPs and nodes as the core events that necessitate the verification of the cluster state, although our solution also captures update events due to its Kubernetes-specific implementation.

A big addition to the Kano algorithm is the Bidirectional Node Policy (BNP) map which saves the matches between nodes and policies and allows for quick retrieval when given a specific policy or node to search for. This data structure leverages bit vectors to efficiently store these connections to improve space and time complexity. In the implementation of this thesis, we used a more rudimentary approach where we stored NPs in the labelTree structure and containers in the LabelMap. Although this enables us to quickly retrieve a NP for a given selector we still need to do extra verification to make sure all labelselectors of policies are present within a pod. As a direct result, our algorithm might become slower when the amount of labels within single objects increases. The BNP map allows for direct retrieval and could be an improvement if implemented in our solution, but only if updating the data structure can be done efficiently.

Although Kano V2 introduced a lot of new concepts and ideas our conflict detection method still proves an extension of the paper as it takes into account the cloud layer of the K8s stack as well, while their verification approach remains within a single layer. It would be interesting to try converting the cloud layer networking rules, such as OpenStack's security groups and rules, into ABAC style labels [7] [17]. This way conflict detection could still be performed while utilising the benefits of Kano V2. Additionally, the more general cloud-native approach of Kano V2 can be a guide in translating conflict detection to a more general solution that applies to different technologies in the stack instead of focusing on OpenStack and K8s. Unfortunately, we must leave these ideas and thoughts for future work.

Bibliography

[1] Overview of Cloud Native Security — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/security/overview/

[2] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, sep 2014.

[3] Y. Mao, Y. Fu, S. Gu, S. Vhaduri, L. Cheng, and Q. Liu, "Resource Management Schemes for Cloud-Native Platforms with Computing Containers of Docker and Kubernetes," oct 2020. [Online]. Available: https://arxiv.org/abs/2010.10350v1

[4] Moby. [Online]. Available: https://mobyproject.org

[5] M. S. Islam Shamim, F. Ahamed Bhuiyan, and A. Rahman, "XI Commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices," *Proceedings - 2020 IEEE Secure Development, SecDev 2020*, pp. 58–64, sep 2020.

[6] Y. Li, C. Jia, X. Hu, and J. Li, "Kano: Efficient container network policy verification," in *Proceedings - Symposium on the High Performance Interconnects, Hot Interconnects*, vol. 2020-Augus. Institute of Electrical and Electronics Engineers Inc., aug 2020, pp. 63–70.

[7] OpenStack Docs: Manage project security. [Online]. Available: https://docs.openstack.org/nova/queens/admin/security-groups.html

[8] CNCF Annual Survey 2022 — Cloud Native Computing Foundation. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/

[9] CNCF Cloud Native Definition v1.0. [Online]. Available: https://github.com/cncf/toc/blob/main/DEFINITION.md

[10] G. Gil, D. Corujo, and P. Pedreiras, "Cloud Native Computing for Industry 4.0: Challenges and Opportunities," *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, vol. 2021-Septe, 2021.

[11] Open Container Initiative Charter v1.3. [Online]. Available: https://github.com/opencontainers/tob/blob/main/CHARTER.md

[12] rkt - the pod-native container engine. [Online]. Available: https://github.com/rkt/rkt

[13] cri-o. [Online]. Available: https://cri-o.io/

[14] Linux Containers. [Online]. Available: https://linuxcontainers.org/

[15] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," *Symposium on Networked Systems Design and Implementation*, 2011. [Online]. Available: https://api.semanticscholar.org/CorpusID:892222

[16] Network Policies — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/

[17] OpenStack Docs: security group rule. [Online]. Available: https://docs.openstack.org/python-openstackclient/train/cli/command-objects/security-group-rule.html

[18] J. K. Teto, R. Bearden, and D. C. T. Lo, "The impact of defensive programming on I/O cybersecurity attacks," *Proceedings of the SouthEast Conference, ACMSE 2017*, pp. 102–111, apr 2017. [Online]. Available: http://dx.doi.org/10.1145/3077286.3077571

[19] OpenStack Docs: Manage project security. [Online]. Available: https://docs.openstack.org/nova/queens/admin/security-groups.html

[20] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[21] G. Budigiri, C. Baumann, E. Truyen, J. T. Muhlberg, and W. Joosen, "Zero-Cost In-Depth Enforcement of Network Policies for Low-Latency Cloud-Native Systems," *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2023-July, pp. 249–261, 2023.

[22] A. Oqaily, L. T. Sudershan, Y. Jarraya, S. Majumdar, M. Zhang, M. Pourzandi, L. Wang, and M. Debbabi, "NFVGuard: Verifying the Security of Multilevel Network Functions Virtualization (NFV) Stack," *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 2020-Decem, pp. 33–40, dec 2020.

[23] Y. Wang, T. Madi, S. Majumdar, Y. Jarraya, A. Alimohammadifar, M. Pourzandi, L. Wang, and M. Debbabi, "TenantGuard: Scalable Runtime Verification of Cloud-Wide VM-Level Network Isolation," may 2017.

[24] S. P. Marsh, "Formalising trust as a computational concept," 1994. [Online]. Available: http://hdl.handle.net/1893/2010

[25] Prisma Cloud. [Online]. Available: https://docs.prismacloud.io/en/classic/cspm-admin-guide/get-started-with-prisma-cloud/prisma-cloud

[26] Illumio Core for Kubernetes and OpenShift. [Online]. Available: https://docs.illumio.com/core/21.1/Content/LandingPages/Guides/kubernetes-and-openshift.htm

[27] Cisco ACI and Kubernetes Integration New and Changed Information 2. [Online]. Available: https://www.cisco.com/c/dam/en/us/td/docs/Website/datacenter/aci/virtualization/matrix/virtmatrix.html

[28] (2017) NSX Container Plugin for Kubernetes and Tanzu Application Service-Installation and Administration Guide NSX Container Plugin for Kubernetes and Tanzu Application Service-Installation and Administration Guide VMware, Inc. 2. [Online]. Available: https://docs.vmware.com/en/VMware-NSX-Container-Plugin/4.1/ncp-kubernetes/GUID-FB641321-319D-41DC-9D16-37D6BA0BC0DE.html

[29] Linux Foundation - Decentralized innovation, built with trust. [Online]. Available: https://www.linuxfoundation.org/

[30] About the Open Container Initiative - Open Container Initiative. [Online]. Available: https://opencontainers.org/about/overview/

[31] Docker Engine overview — Docker Docs. [Online]. Available: https://docs.docker.com/engine/

[32] Container Engines: How They Work and Top 7 Options. [Online]. Available: https://www.aquasec.com/cloud-native-academy/container-platforms/container-engines/{#}Docker

[33] A Practical Introduction to Container Terminology — Red Hat Developer. [Online]. Available: https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction{#}container{_}runtime

[34] What are container runtimes? — Opensource.com. [Online]. Available: https://opensource.com/article/21/9/container-runtimes

[35] containerd/containerd: An open and reliable container runtime. [Online]. Available: https://github.com/containerd/containerd

[36] Product Overview - Mirantis Container Runtime. [Online]. Available: https://docs.mirantis.com/mcr/20.10/overview.html

[37] Swarm mode overview — Docker Docs. [Online]. Available: https://docs.docker.com/engine/swarm/

[38] Apache Mesos. [Online]. Available: https://mesos.apache.org/

[39] Cloud Computing Services - Amazon Web Services (AWS). [Online]. Available: https://aws.amazon.com/

[40] Cloud Computing Services — Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us

[41] Cloud Computing Services — Google Cloud. [Online]. Available: https://cloud.google.com/?hl=en

[42] Nodes — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/nodes/

[43] Pods — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/

[44] kubelet — Kubernetes. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/

[45] kubernetes-client/python: Official Python client library for kubernetes. [Online]. Available: https://github.com/kubernetes-client/python

[46] kube-apiserver — Kubernetes. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

[47] Namespaces — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

[48] E. Truyen, A. Jacobs, S. Verreydt, E. H. Beni, B. Lagaisse, and W. Joosen, "Feasibility of container orchestration for adaptive performance isolation in multi-tenant SaaS applications," *Proceedings of the ACM Symposium on Applied Computing*, pp. 162–169, mar 2020. [Online]. Available: https://lirias.kuleuven.be/retrieve/566299

[49] E. Yuan and J. Tong, "Attributed Based Access Control (ABAC) for web services," *Proceedings - 2005 IEEE International Conference on Web Services, ICWS 2005*, vol. 2005, pp. 561–569, 2005.

[50] Labels and Selectors — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

[51] Network Policies — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/network-policies/

[52] Network Plugins — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

[53] Services, Load Balancing, and Networking — Kubernetes. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/{#}the-kubernetes-network-model

[54] containernetworking/cni: Container Network Interface - networking for Linux containers. [Online]. Available: https://github.com/containernetworking/cni

[55] weaveworks/weave: Simple, resilient multi-host containers networking and more. [Online]. Available: https://github.com/weaveworks/weave

[56] cilium/cilium: eBPF-based Networking, Security, and Observability. [Online]. Available: https://github.com/cilium/cilium

[57] projectcalico/calico: Cloud native networking and network security. [Online]. Available: https://github.com/projectcalico/calico

[58] eBPF use cases — Calico Documentation. [Online]. Available: https://docs.tigera.io/calico/latest/operations/ebpf/use-cases-ebpf

[59] calico/networking-calico at master · projectcalico/calico. [Online]. Available: https://github.com/projectcalico/calico/tree/master/networking-calico

[60] ChromeOS (Operating System) - Chrome Enterprise. [Online]. Available: https://chromeenterprise.google/os/

[61] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "OpenStack: Toward an Open-source Solution for Cloud Computing," *International Journal of Computer Applications*, vol. 55, no. 3, pp. 38–42, oct 2012.

[62] OpenStack Docs: Manage quotas. [Online]. Available: https://docs.openstack.org/nova/queens/admin/quotas2.html{#}manage-quotas

[63] Y. Li, X. Hu, C. Jia, K. Wang, and J. Li, "Kano: Efficient Cloud Native Network Policy Verification," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 3747–3764, sep 2023.

DEPARTMENT OF COMPUTER SCIE
Celestijnenlaan 200 A box 2
3000 LEUVEN, BEL
tel. + 32 16 32 7
fax + 32 16 32 7
wms.cs.kuleuven.b