University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Measuring the performance of a new eBPF implementation of the Kubernetes ClusterIP Service

Bruno Campos e João Henri

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Daniel Macêdo Batista

São Paulo
2023

Acknowledgments

The first acknowledgment is reserved for Professor Daniel, who, in addition to being an exceptional supervisor, also taught the course MAC0470/MAC5856 - Free Software Development, which was both essential and a precursor to this work.

Moving forward, we extend our appreciation to the Kubernetes community, whose warm and supportive reception was invaluable, with special recognition for the maintainers of ’kpng’: Jay Vyas, Andrew Stoycos, Ricardo Katz, and Per Andersson.

Lastly, we wish to convey our deep gratitude to our friends and family. Their unwavering support, encouragement, and understanding have been pivotal throughout this journey.

A journey of a thousand miles begins with a single step.

— Lao Tzu

Resumo

Bruno Campos e João Henri. **Medindo o desempenho de uma nova implementação em eBPF do serviço ClusterIP do Kubernetes**. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

O Kubernetes é uma plataforma de orquestração de contêineres de código aberto amplamente usada para automatizar o gerenciamento, escalabilidade e implantação de aplicativos, principalmente em ambientes *cloud native*. O *kube-proxy* é um de seus componentes fundamentais, responsável por gerenciar as regras de rede e os endereços virtuais, ou seja, encaminhar o tráfego entre os *serviços* e *pods* dentro de um *cluster*. Tradicionalmente, o *kube-proxy* utilizou o *iptables* como *backend* principal para realizar essa tarefa. No entanto, com a evolução das tecnologias de rede e a busca por maior desempenho e flexibilidade, tem surgido uma alternativa promissora: a integração do eBPF (extended Berkeley Packet Filter) como mecanismo de encaminhamento de pacotes. O eBPF é uma tecnologia que permite a execução de código personalizado no kernel Linux, usado principalmente para processamento avançado de pacotes e monitoramento de recursos. Nesta monografia, é feita uma análise de desempenho da implementação de um *backend* usando eBPF, a partir de um comparativo de latência com o kube-proxy tradicional. Foi possível observar uma latência média da viagem de ida e volta, com um nível de confiança de 99%, para a implementação eBPF de 1,23 ± 0,00223 ms e para a implementação tradicional baseada em iptables de 1,28 ± 0,00463 ms, indicando um desempenho ligeiramente superior para a nova implementação.

Palavras-chave: Kubernetes. kube-proxy. eBPF. Linux. Roteamento de pacotes. Computação em Nuvem.

Abstract

Bruno Campos e João Henri. **Measuring the performance of a new eBPF implementation of the Kubernetes ClusterIP Service**. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Kubernetes is a widely used open-source container orchestration platform designed to automate application management, scalability, and deployment, particularly in cloud-native environments. One of its fundamental components is *kube-proxy*, responsible for managing network rules and virtual addresses, i.e., routing traffic between *services* and *pods* within a *cluster*. Traditionally, *kube-proxy* has utilized *iptables* as its primary backend to perform this task. However, with the evolution of network technologies and the pursuit of increased performance and flexibility, a promising alternative has emerged: the integration of eBPF (extended Berkeley Packet Filter) as a packet forwarding mechanism. eBPF is a technology that allows the execution of custom code in the Linux kernel, primarily used for advanced packet processing and resource monitoring. This thesis conducts a performance analysis of the eBPF-backend implementation, comparing latency with the traditional kube-proxy. It was possible to observe a round-trip average latency, with a 99% confidence level, for the eBPF implementation of 1.23 ± 0.00223 ms, and the traditional iptables implementation of 1.28 ± 0.00463 ms, indicating a slightly superior performance for the new implementation.

Keywords: Kubernetes. kube-proxy. eBPF. Linux. Packet routing. Cloud Computing.

Contents

Chapter 1

Introduction

Kubernetes is a widely used open-source container orchestration platform designed to automate the management, scalability, and deployment of applications, primarily in cloud-native environments. Kube-proxy is one of its fundamental components, responsible for managing network rules, virtual addresses, and routing traffic between Services and Pods within a cluster. Traditionally, kube-proxy used `iptables` as the primary backend for this task. Our capstone project evaluates a promising alternative to `iptables`: `eBPF` (extended Berkeley Packet Filter).

eBPF is an evolution of the original BSD packet filter (BPF) McCanne and Jacobson, 1993. BPF uses a register-based virtual machine to describe filtering actions. Support for compiling (restricted) C and Rust code into eBPF is included in the LLVM compiler infrastructure Lattner and Adve, 2004. eBPF allows the execution of custom code in the Linux kernel.

Kube-Proxy New Generation (KPNG) [1] is a project backed by the Cloud Native Computing Foundation (CNCF) [2] that intends to decouple access to the Service description objects from their network backend implementation. In this context, to introduce another networking mode for Kubernetes Services, one could create a backend that uses any networking tool. Our project evaluates an eBPF implementation of Kubernetes ClusterIP service deployed through KPNG.

In Chapter 2, basic concepts and tools are explained. Chapter 3 addresses the test environment and how the solutions were built. Chapter 4 shows the results and Chapter 5 the conclusions.

[1] The official KPNG repository is https://github.com/kubernetes-sigs/kpng

[2] More information about the CNCF can be found at https://cncf.io

Chapter 2

Basic Concepts

Container communication is essential for the proper operation of applications that embrace the microservices architecture. Understanding the concepts behind this architecture is crucial for comprehending the motivation behind solutions like Kubernetes and KPNG. Therefore, this chapter introduces some fundamental concepts such as Microservices Architecture, Containers, Kubernetes' resources, Kubernetes Services' inner workings, KPNG (Kube-proxy new generation), Linux networking, and eBPF.

2.1 Microservices Software Architecture

In the early days of software development, applications were mostly monoliths. That is, they ran in a single process on a single computer. Such applications had what is known as a monolithic architecture and were simpler to develop and deploy, making them more suitable for small programs. However, during the 2000s, with the advancement of network communications, computer programs grew larger, became harder to maintain, and demanded many more resources than before. It was in this context that the microservices architecture was created. This approach mandates breaking down an application into multiple program modules running in different processes that communicate with each other using inter-process communication mechanisms when they run on the same operating system or network communication mechanisms when they run on distinct computers or containers. The new architecture proved to be more reliable for more complex systems and has become the industry standard for these environments Raharjo *et al.*, 2022.

There are several advantages to using the microservices architecture over the monolith, including:

- Code Maintenance: It's easier to understand and maintain small, decoupled programs than a large monolithic one.
- Runtime Errors: A runtime error in a monolith impairs the entire application, whereas in a microservices-based system, it only affects the module in which it occurred.
- Deployment Independence: With a monolithic architecture, when scaling horizontally (adding copies of the process), you must run the entire application in each

replica. In a microservices-based application, you can scale individual modules that may require more resources than others without having to run the entire application again.

Considering that each microservices module runs in a different process, it is important to isolate these processes from one another to prevent one from interfering with another and causing unexpected runtime errors. A container provides a way to run these processes in complete isolation, even when they are running on the same machine with the same operating system (without using a virtual machine).

2.2 Containers

Virtual machines have been in existence since the era of mainframes. Due to the immense processing power of these machines, it was improbable that a single process would consistently utilize all the available resources. Consequently, users were allocated a fraction of the machine's capacity for their processes, enabling concurrent program execution. This execution model was termed "multi-programming" and significantly improved the efficiency of computer resource utilization Hansen, 1972. With the advent of personal computing, this concept transitioned to smaller computers, primarily designed to facilitate the simultaneous operation of two operating systems on the same hardware. Instances of operating systems coexisting on the same hardware are referred to as Virtual Machines Goldberg, 1974. Virtual Machines necessitated the installation of an entirely new operating system and the use of specialized software known as a hypervisor to mediate between the virtualized operations of the guest operating system and the host system. However, when the need arises to run an isolated program exclusively in user space, a Virtual Machine can introduce significant overhead. A more efficient approach involves employing lightweight tools like chroot [1], leading to the development of lightweight virtualization solutions, such as containers.

A container is an isolated runtime environment for a process, offering a streamlined alternative to running a complete operating system within a virtual machine. Containers share the kernel of their host system and can possess varying degrees of isolation. These degrees of isolation are determined by the host system's allocation of resources, typically including the file system, PID list, network connections, as well as resources like memory and CPU.

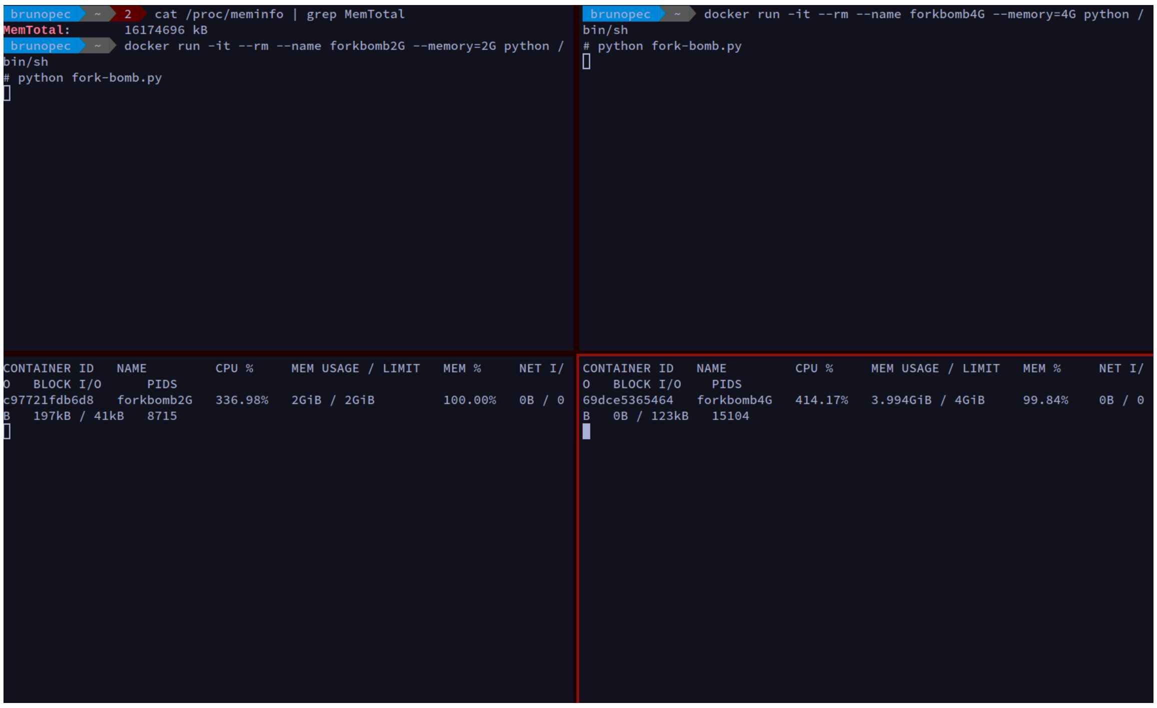
The primary advantages of using containers are as follows:

- Lightweight: Containers are exceptionally lightweight because they do not require booting, installing, and managing an entire operating system (OS), running a separate kernel, or executing various OS-related processes to host a single process. For example, in Zhang *et al.*, 2018, the authors conducted experiments in which they compared the boot-up time of Docker containers and VMs running the same application with an increasing number of instances. With 256 instances, the containers

[1] Chroot, short for 'change root,' is a Unix operating system feature that enables the modification of the apparent root directory for a process, limiting its access to specific file systems.

took 479 seconds to initialize, whereas the VMs took 24,295 seconds to initialize, which is 50.72 times longer.

- Portability: Containers are OS-agnostic, allowing the same container to run on diverse OSs without requiring extensive configuration changes. For example, on https://hub.docker.com/_/postgres, it is possible to obtain an image of the PostgreSQL DBMS that is compatible with any Linux distribution and Windows Subsystem for Linux, without the need for any specific configuration for these systems.
- Isolation: Containers are constrained to access only the specified resources, limiting their resource consumption and enhancing security. For example, Figure 2.1 is a screenshot of four terminals. The top left one shows the total amount of memory of the computer, which is 16 gigabytes. In the example, a fork bomb (an infinite loop that allocates memory in each iteration) was run inside two Python containers, one with a limit of 2 gigabytes (left terminals) and another with a 4-gigabyte limit (right terminals). The bottom terminals show the running resource stats of both containers, indicating that they are restricted to only 2 gigabytes and 4 gigabytes of memory, even though the computer on which it is running has 16 gigabytes of memory.
- Dependency Management: Containers can maintain isolated file systems, eliminating conflicts and simplifying dependency management when running complex applications. For instance, Figure 2.2 displays two terminals in which two containers were executed: one containing a PostgreSQL database, and the other containing the tools for running Python programs. When we inspect the Python dependencies in both containers, it becomes evident that the PostgreSQL container lacks many of the dependencies present in the Python container.



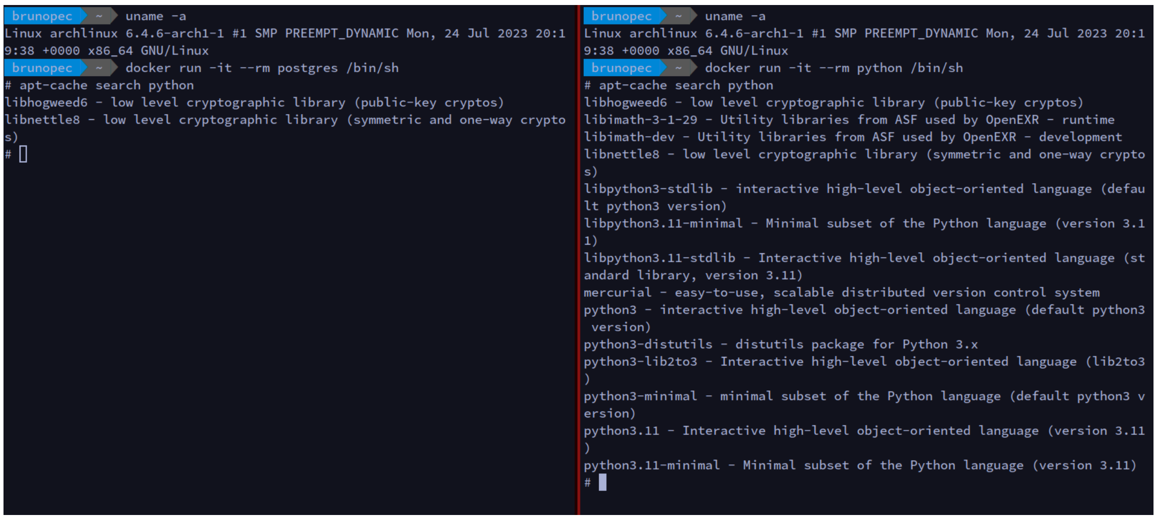
```
 brunopec  ~  2  cat /proc/meminfo | grep MemTotal
MemTotal:        16174696 kB
 brunopec  ~  docker run -it --rm --name forkbomb2G --memory=2G python /
bin/sh
# python fork-bomb.py
```

```
 brunopec  ~  docker run -it --rm --name forkbomb4G --memory=4G python /
bin/sh
# python fork-bomb.py
```

```
CONTAINER ID   NAME         CPU %     MEM USAGE / LIMIT   MEM %     NET I/
O   BLOCK I/O      PIDS
c97721fdb6d8   forkbomb2G   336.98%   2GiB / 2GiB         100.00%   0B / 0
B   197kB / 41kB   8715
```

```
CONTAINER ID   NAME         CPU %     MEM USAGE / LIMIT   MEM %     NET I/
O   BLOCK I/O    PIDS
69dce5365464   forkbomb4G   414.17%   3.994GiB / 4GiB     99.84%    0B / 0
B   0B / 123kB   15104
```

Figure 2.1: *Screenshot of an example of Docker's container resource constraint feature*



```
brunopec  ~  uname -a
Linux archlinux 6.4.6-arch1-1 #1 SMP PREEMPT_DYNAMIC Mon, 24 Jul 2023 20:1
9:38 +0000 x86_64 GNU/Linux
brunopec  ~  docker run -it --rm postgres /bin/sh
# apt-cache search python
libhogweed6 - low level cryptographic library (public-key cryptos)
libnettle8 - low level cryptographic library (symmetric and one-way crypto
s)
# 
```

```
brunopec  ~  uname -a
Linux archlinux 6.4.6-arch1-1 #1 SMP PREEMPT_DYNAMIC Mon, 24 Jul 2023 20:1
9:38 +0000 x86_64 GNU/Linux
brunopec  ~  docker run -it --rm python /bin/sh
# apt-cache search python
libhogweed6 - low level cryptographic library (public-key cryptos)
libimath-3-1-29 - Utility libraries from ASF used by OpenEXR - runtime
libimath-dev - Utility libraries from ASF used by OpenEXR - development
libnettle8 - low level cryptographic library (symmetric and one-way crypto
s)
libpython3-stdlib - interactive high-level object-oriented language (defau
lt python3 version)
libpython3.11-minimal - Minimal subset of the Python language (version 3.1
1)
libpython3.11-stdlib - Interactive high-level object-oriented language (st
andard library, version 3.11)
mercurial - easy-to-use, scalable distributed version control system
python3 - interactive high-level object-oriented language (default python3
 version)
python3-distutils - distutils package for Python 3.x
python3-lib2to3 - Interactive high-level object-oriented language (lib2to3
)
python3-minimal - minimal subset of the Python language (default python3 v
ersion)
python3.11 - Interactive high-level object-oriented language (version 3.11
)
python3.11-minimal - Minimal subset of the Python language (version 3.11)
# 
```

Figure 2.2: *Screenshot of an example of Docker's container package isolation feature*

Containers rely on Linux's isolation capabilities, including Namespaces (Process ID (PID), Networks, Interprocess Communication (IPC)), Cgroups (CPU, memory, disk space), and chroot (filesystems). To run a container, two key components are required: a container engine and a container runtime. A container engine is software that interprets user data or Representational State Transfer (REST) API commands into calls to the container runtime. Common examples of container engines include Docker [2] and Podman [3]. A container runtime, on the other hand, is a low-level software that leverages OS-level isolation tools to create and manage containers. Prominent container runtimes include ContainerD [4] and CRI-O [5].

Figure 2.3 provides a visual comparison between traditional VMs and containers, illustrating the role of the Hypervisor and Container Engine, as well as the VMs and Containers themselves. Additionally, Figure 2.4 offers an in-depth overview of Docker's internal structure, showcasing the various layers and components involved in Docker and container execution. For example, a user employs the Docker CLI program to create a container. Subsequently, the program sends a request to the Docker daemon API, which runs in the background. The Docker API performs various checks, including authentication, security, state, and network checks, to determine the validity of the request at that moment. If the checks are successful, it utilizes the container engine (containerD) to carry out operations in the OS layer with Runc (container engine module responsible for spawning and running containers), effectively creating the container.

Containers can be effectively employed in Microservices architecture to combine the strengths of both paradigms. However, achieving scalability, automation, and compatibility across diverse infrastructure environments necessitates more than just using a container engine. To address these requirements, specialized software known as container orches-

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

[3] https://podman.io/

[4] https://containerd.io/

[5] https://cri-o.io/

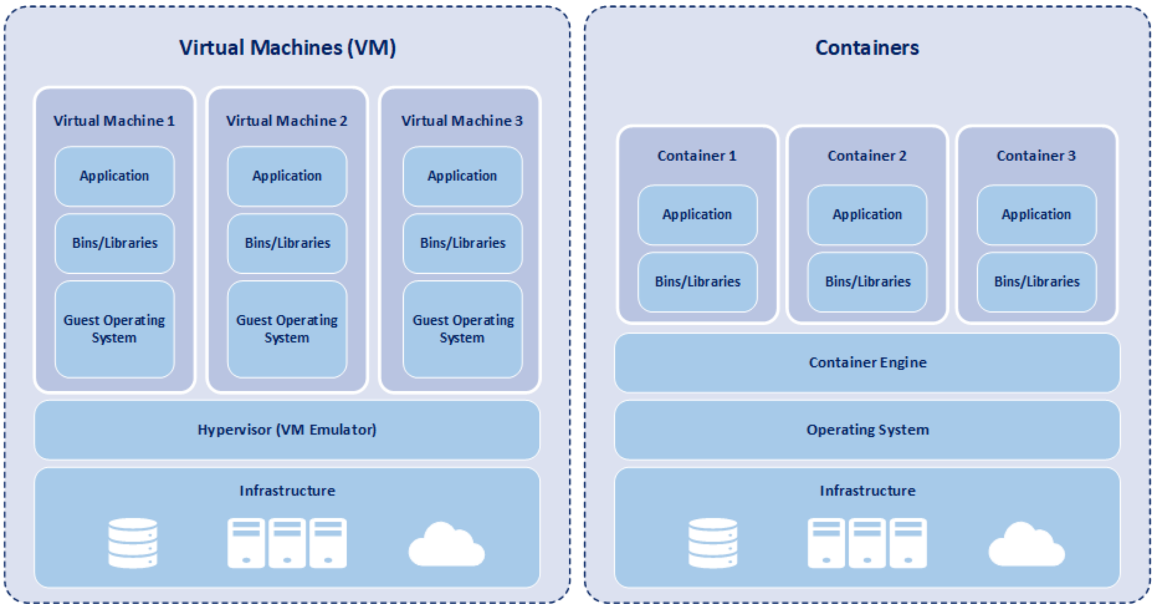


Figure 2.3: *Comparison between Containers and Virtual Machines. Source: https://bi-insider.com/posts/virtual-machines-vs-containers/*

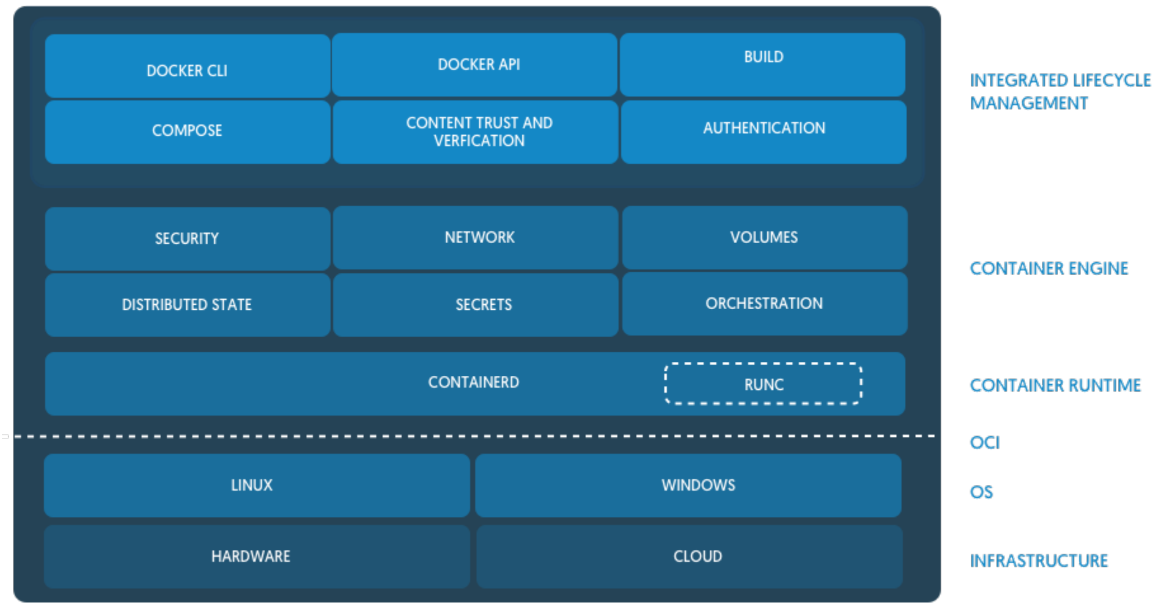


Figure 2.4: *Breakdown of Docker internal structure. Source: https://tansanrao.com/docker-explained/*

trator is recommended. Container orchestrators enable Microservices systems to operate their components within containers, offering scalability, reliability, redundancy, security, and centralized control over the application.

2.3 Kubernetes

Kubernetes [6], also referred to as *K8s* by its community, is a container orchestration tool that has seen significant growth since its inception in 2015. A container orchestration tool's purpose is to manage the entire environment in which containerized applications run. It generally holds the following responsibilities:

- Automation of container deployment: creating and deleting containers across multiple computing instances.
- Auto-scaling: managing the container fleet based on predetermined factors like service demand or time period.
- Service discovery: classifying containers according to their service type.
- Load balancing: distributing request loads across multiple containers of a service.
- Rolling updates: managing container versions within the container fleet.
- Self-healing: overseeing the container's life cycle to ensure high service availability.
- Resource management: handling the computer cluster's resources and distributing containers across them.
- Security and compliance: ensuring security throughout the entire infrastructure.

Kubernetes delivers these primary functionalities and additional auxiliary functions in a declarative manner. In other words, the desired end state is described without detailing the step-by-step process of achieving it, and Kubernetes ensures the necessary changes are made to achieve it. Moreover, it is entirely infrastructure-agnostic, meaning it has no requirements for the machines it runs on, making it an excellent choice for cloud-based environments where machines can be quickly and easily exchanged and provisioned.

Furthermore, according to a 2022 report by Red Hat *The State of Enterprise Open Source: A Red Hat report* 2022, 70% of IT leaders worldwide work for companies that use K8s. The Kubernetes project is an open-source initiative that has garnered 98,400 stars and 3,432 contributors on its GitHub repository since its inception. These figures, combined with its relatively short lifespan (9 years), highlight the rapid importance that the software has gained in both enterprise and open-source contexts.

Kubernetes Resources

Kubernetes boasts a highly sophisticated structure and divides its features into a Resource-Based System. This system instantiates internal functionalities into entities called Resources. Each resource has its own role in running the user-demanded microservice

[6] https://kubernetes.io/

application. Resource creation and access can be automated with the help of other higher-level resources.

The fundamental Kubernetes resource is the Pod. The Pod's responsibility is to encapsulate a container or a set of containers that represent an instance of a micro-service app. All containers can communicate with each other via localhost, and they can be accessed by other pods in the cluster using a single IP address. Additionally, the containers share the network namespace and can share the same volumes. Finally, Pods can be categorized using Labels. Pods with the same label can, for example, represent multiple instances of a micro-service application. Since the Pod is the resource that encapsulates a container, we're going to use both terms interchangeably to describe the entity that effectively runs a microservice process.

A Kubernetes cluster operates in a distributed manner within a computer cluster. A fundamental resource in the Kubernetes architecture for identifying and abstracting these computers is the 'Node.' There are two primary types of Nodes: the 'control plane' Nodes, dedicated to managing and controlling Kubernetes functions, and the 'worker' Nodes, which host the Pods responsible for running applications. Each Node possesses its own unique IP address and a set of Pods that execute within its environment. Application Pods exclusively operate within the worker Nodes, while system-control Pods, which will be elaborated upon later, are confined to the control-plane Nodes.

Another significant Kubernetes resource is the Service. The service represents a gateway through which a micro-service can be reached. Using labels, a service can select a fleet of Pods running the same application (Pod replication can be automated by other resources). Services enable accessing a service through a single IP address, rather than having to reach each individual IP address of every Pod. Kubernetes Services are the primary focus of this work.

2.3.1 Kubernetes Services

The microservices architecture allows applications to decouple their internal parts and run them in a distributed manner. For example, an e-commerce microservice application could have a service for item storage and retrieval, one for managing user operations such as authentication and order history, and another for processing payments. In a Kubernetes Cluster, all these services run in multiple distinct Pods, and it would be too difficult to find which Pods provide a service and distribute the requests among those Pods. In this context, the Service resource is responsible for creating a central point of access for a certain service and distributing requests for this specified service to their corresponding container fleet. That way, users don't need to know the IP of every Pod to reach a certain service. Furthermore, the Service resource also provides load balancing between the Pods that provide a certain service, making it an essential part of a Kubernetes cluster architecture.

Aside from those fundamental Service features, there are multiple types of Service, that provide different usages and enhance its functionality with additional features. The ones that are the focus of this work are:

- ClusterIP: The ClusterIP service is the basic type of service, and as such, it only provides the functionalities previously described. Its definition requires a label selector to select the Pod replicas that are in the same category, the port at which the process is running inside the Pod allocated to receive the service's requests, and which external parties can use to access that Service resource. These ports can be distinct; for instance, the Pod can run an HTTP server on port 80, and the Service can receive requests for that service on port 443. It is important to mention that a ClusterIP service can only be accessed by Pods within the Kubernetes Cluster and not by external entities.

- NodePort: The NodePort service is built on top of the ClusterIP Service. It provides endpoints for clients outside of the cluster to connect to a Service. To achieve this, it allocates a port on each of the Cluster's Nodes that redirects requests to an internal ClusterIP service. In addition to the ClusterIP parameters, the NodePort requires the port to be allocated on each Node to establish outbound connections. Figure 2.5 provides a graphical overview of NodePort and, consequently, ClusterIP Services. Within the Cluster, there are two working Nodes, and a NodePort Service allocates port 30080 on each of them to receive external requests from the cluster. These requests are received and then redirected to the internal ClusterIP service, which distributes them to the Pods that are part of this Service on port 80.

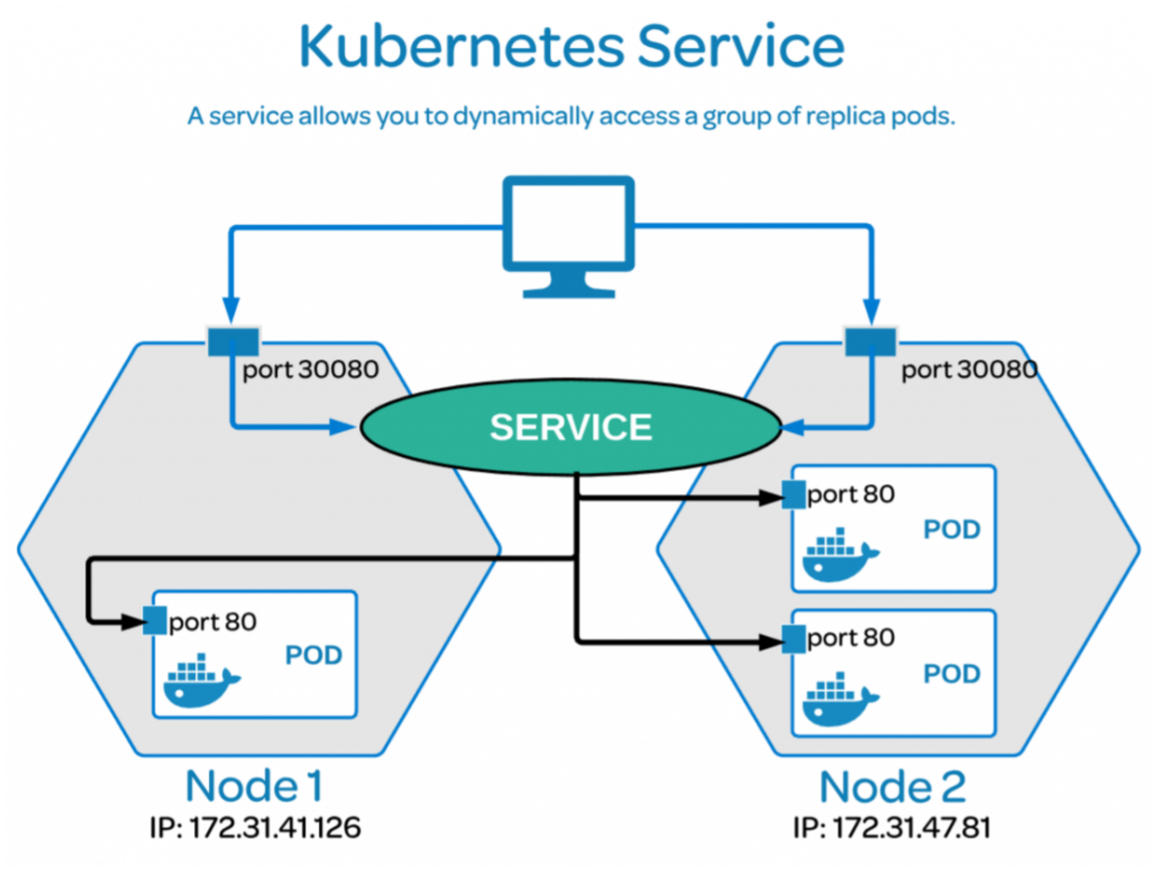


Figure 2.5: *Kubernetes's Service Resource overview. Source: https://medium.com/avmconsulting-blog/single-and-multi-port-service-in-kubernetes-k8s-8b08529d9ba6*

Services Internal Structure

Internally, Kubernetes Services work by creating another resource called an "Endpoint" for each new pod that integrates with the Service's fleet. The Endpoint resource contains the IP address and port at which the pod is running the server process.

Furthermore, an important aspect of Services is that they do not require a server to be running in order to route packages to the corresponding Pods. All the forwarding is achieved through Network Address Translation (NAT) techniques. There are various methods to achieve this, and the internal component of a Kubernetes Cluster responsible for this task is known as "Kube-Proxy." In practice, "Kube-Proxy" is a fleet of Pods distributed across the Cluster Nodes (with at least one for each Node to ensure functionality throughout the cluster). These Pods use various Linux features and software to create equivalent NAT rules for Services. Figure 2.6 illustrates an example of how a packet transmission to a ClusterIP Service is handled. First, Pod A tries to establish a connection to Service 1, which is a ClusterIP service. Consequently, the packet has a source IP address of Pod A and a destination IP address of Service 1. Then, Kube-Proxy applies NAT to change the packet's destination address to Pod B. If there were multiple Pods, it would perform load balancing to determine which Pod to forward the packet to. After Pod B receives the packet, it sends a response back to Pod A through Kube-Proxy. Finally, Kube-Proxy changes its source address to the ClusterIP address, and the packet is delivered to Pod A.

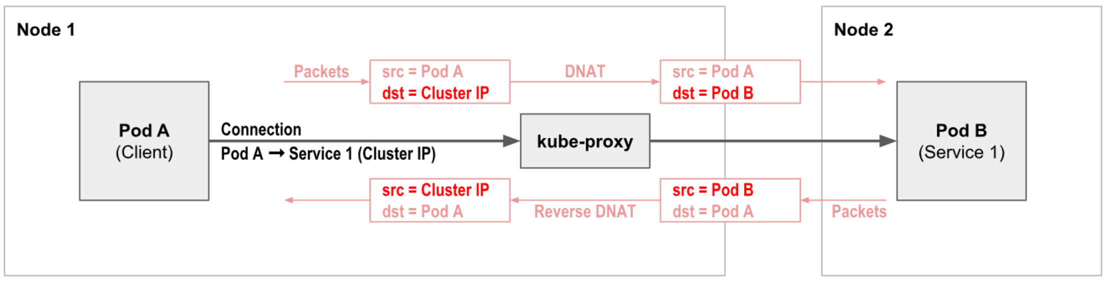


Figure 2.6: *Kubernetes Kube-Proxy functionality for ClusterIP Services. Source: https://docs.tigera.io/calico-cloud/tutorials/training/about-kubernetes-services*

The default method used by Kube-Proxy to implement NAT is through Linux's iptables, a software tool responsible for creating rules for handling incoming and outgoing network packets. In addition to iptables, out of the box, there's also the possibility to utilize IP Virtual Server (IPVS) as the NAT provider. IPVS is a utility that implements transport-layer load balancing on Linux.

Due to its internal code structure, Kube-Proxy is tightly coupled with these two solutions. Therefore, employing another NAT program to handle packet forwarding would necessitate the recreation of the entire component. In this context, there is a project known as Kube-Proxy New Generation (KPNG) that aims to redesign the Kube-Proxy component in a way that decouples it from its NAT solution. KPNG's Kube-Proxy component is agnostic regarding its NAT backend, making it a suitable option for testing new NAT software for Kubernetes (K8s).

2.3.2 Kubernetes Workloads

A workload within Kubernetes refers to an application being executed. Whether it's a single part or multiple components working in tandem, when running on Kubernetes, it operates within a set of pods. In Kubernetes, a pod is a representation of actively running containers within the cluster.

Kubernetes pods follow a predefined lifecycle. For instance, if a pod is running in the cluster and there's a critical issue with the node it's on, all pods on that node fail. In this scenario, Kubernetes considers the failure irreversible, requiring the creation of a new pod for recovery, even if the node later becomes stable.

To simplify operations, there's no need to directly manage each pod. Instead, workload resources are used, controlling a set of pods. These resources configure controllers that ensure the right number of the correct type of pods are running to match the specified state.

DaemonSet

A DaemonSet ensures that all (or some) Nodes run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are removed from the cluster, those Pods are garbage collected. Deleting a DaemonSet will clean up the Pods it created.

ReplicaSet

The primary objective of a ReplicaSet is to ensure a consistent set of replica Pods is continuously running. This functionality is commonly employed to ensure a specific number of identical Pods remain available.

To define a ReplicaSet, various fields are specified, including a selector for identifying Pods it can acquire, the desired number of replicas to maintain, and a pod template detailing the configuration of new Pods required to meet the replica count. The ReplicaSet achieves its purpose by dynamically creating and deleting Pods as necessary to achieve the specified number. When new Pods need creation, the ReplicaSet employs its Pod template.

The association between a ReplicaSet and its Pods is established through the `metadata.ownerReferences` field in the Pods. This field indicates the resource that currently owns the object. All Pods acquired by a ReplicaSet contain information about their owning ReplicaSet within their ownerReferences field. This linkage allows the ReplicaSet to monitor the state of the maintained Pods and make informed decisions accordingly.

Deployment

A Deployment provides declarative updates for Pods and ReplicaSets.

You describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

2.4 Kube-Proxy New Generation

Kube-Proxy New Generation [7] (KPNG) is a project backed by the Cloud Native Computing Foundation (CNCF) that intends to decouple access to the Service description objects from their Network Backend implementation. This makes it simpler to use different options for packet forwarding for Services. The primary advantage of KPNG, in comparison to the out-of-the-box Kube-Proxy, is that the default one is tied to the two Network Backend options that come with it: iptables and IPvs. Therefore, KPNG provides more flexibility and agility for using newer and superior technologies to handle networking for Services. For instance, iptables, the default option for K8s cluster services, is now deprecated and faces scalability issues, which also apply to the ipvs backend. A new technology that can be used in spite of the default ones is eBPF, which will be discussed later.

To achieve its objectives, KPNG's structure consists of a client-server architecture. The server provides actual data about the Services, such as their corresponding Endpoint resources, types, and ports. The client consumes this information through an API. Finally, the Backend utilizes the client infrastructure to program the networking rules required to implement the desired actions on the actual Nodes. In this context, to introduce another networking mode for Kubernetes Services, one could create a Backend that uses any networking tool with the data provided by the client to establish the necessary rules for it to function. Figure 2.7 provides an in-depth overview of KPNG's internal structure. The KPNG client retrieves the state from the API endpoint and delivers it to the Backend. Inside the blue area, there are the interfaces that the Backend should implement. These include the Sink interface, which defines how the Backend receives resource information from the KPNG client, the FilterReset Syncer, which specifies the incremental update of information, and the custom logic responsible for the proper installation and management of the Backend to create networking rules.

To ensure that the network rules are applied throughout the entire cluster, KPNG deploys one working pod for each of the Cluster's Nodes. Each of these pods is responsible for correctly deploying the backend program and managing it to reflect the current state in terms of Services and Pods. Figure 2.8 shows the KPNG working pods, which are highlighted by the red squares and named kpng-XXXX. From the "Node" column, it is evident that there is exactly one of these pods for each Node. There are two working Nodes and one Control-plane Node.

An interesting technology that has been used for networking is Berkeley Packet Filter (BPF). BPF, or eBPF [8] (Extended Berkeley Packet Filter), is a Linux kernel feature that enables the creation of hooks in kernel functions to execute verified, safe code in the kernel space. With BPF, it is possible to hook functions to analyze packets and modify them in the kernel space, across multiple layers of Linux's network stack. Since it operates in the kernel space, it has the potential to be faster than traditional filtering applications such as iptables and IPvs for performing operations like NAT. KPNG enables developers to create solutions for Service networking with BPF programs. This project aims to test the performance of an existing BPF-based KPNG Backend solution.

[7] https://github.com/kubernetes-sigs/kpng

[8] https://ebpf.io/

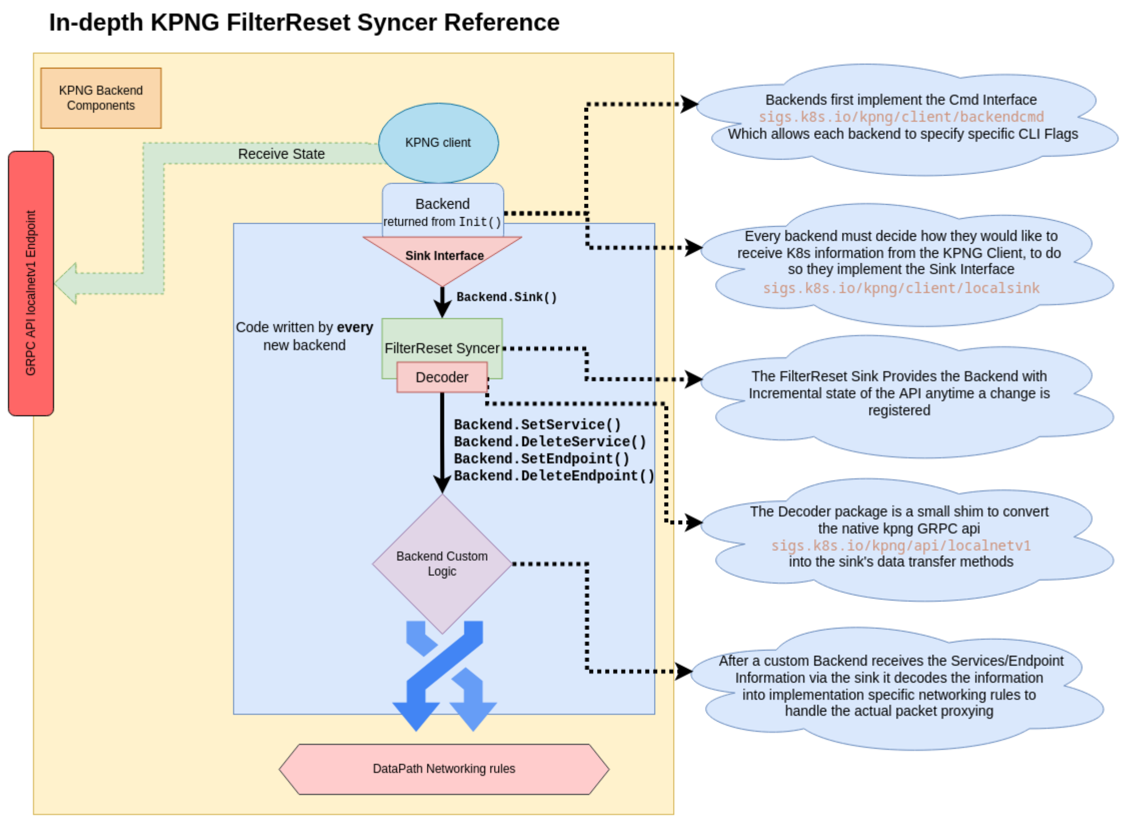

In-depth KPNG FilterReset Syncer Reference
KPNG Backend Components
GRPC API localnetv1 Endpoint
Receive State
KPNG client
Backend
returned from Init()
Sink Interface
Backend.Sink()
Code written by every new backend
FilterReset Syncer
Decoder
Backend.SetService()
Backend.DeleteService()
Backend.SetEndpoint()
Backend.DeleteEndpoint()
Backend Custom Logic
DataPath Networking rules
Backends first implement the Cmd Interface sigs.k8s.io/kpng/client/backendcmd Which allows each backend to specify specific CLI Flags
Every backend must decide how they would like to receive K8s information from the KPNG Client, to do so they implement the Sink Interface sigs.k8s.io/kpng/client/localsink
The FilterReset Sink Provides the Backend with Incremental state of the API anytime a change is registered
The Decoder package is a small shim to convert the native kpng GRPC api sigs.k8s.io/kpng/api/localnetv1 into the sink's data transfer methods
After a custom Backend receives the Services/Endpoint Information via the sink it decodes the information into implementation specific networking rules to handle the actual packet proxying

Figure 2.7: *Overview of KPNG's internal components. Source: https://github.com/kubernetes/enhancements/blob/599a2ef14df7a7172c1933c3d989c1cfbfe8c11c/keps/sig-network/2104-reworking-kube-proxy-architecture/README.md*

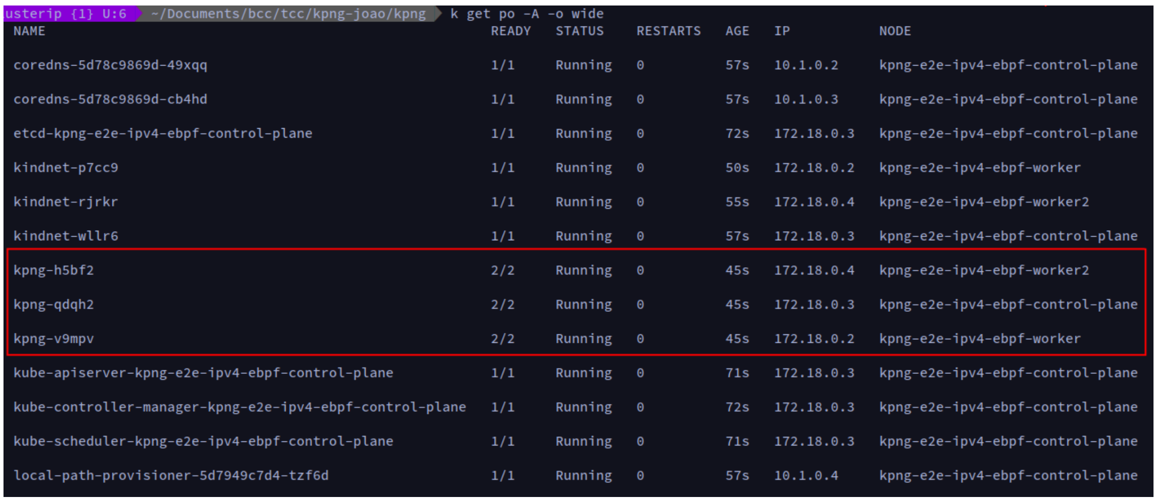

```
usterip {1} U:6  ~/Documents/bcc/tcc/kpng-joao/kpng  k get po -A -o wide
 NAME                                                  READY   STATUS    RESTARTS   AGE   IP           NODE
 coredns-5d78c9869d-49xqq                              1/1     Running   0          57s   10.1.0.2     kpng-e2e-ipv4-ebpf-control-plane
 coredns-5d78c9869d-cb4hd                              1/1     Running   0          57s   10.1.0.3     kpng-e2e-ipv4-ebpf-control-plane
 etcd-kpng-e2e-ipv4-ebpf-control-plane                 1/1     Running   0          72s   172.18.0.3   kpng-e2e-ipv4-ebpf-control-plane
 kindnet-p7cc9                                         1/1     Running   0          50s   172.18.0.2   kpng-e2e-ipv4-ebpf-worker
 kindnet-rjrkr                                         1/1     Running   0          55s   172.18.0.4   kpng-e2e-ipv4-ebpf-worker2
 kindnet-wllr6                                         1/1     Running   0          57s   172.18.0.3   kpng-e2e-ipv4-ebpf-control-plane
 kpng-h5bf2                                            2/2     Running   0          45s   172.18.0.4   kpng-e2e-ipv4-ebpf-worker2
 kpng-qdqh2                                            2/2     Running   0          45s   172.18.0.3   kpng-e2e-ipv4-ebpf-control-plane
 kpng-v9mpv                                            2/2     Running   0          45s   172.18.0.2   kpng-e2e-ipv4-ebpf-worker
 kube-apiserver-kpng-e2e-ipv4-ebpf-control-plane       1/1     Running   0          71s   172.18.0.3   kpng-e2e-ipv4-ebpf-control-plane
 kube-controller-manager-kpng-e2e-ipv4-ebpf-control-plane  1/1 Running   0          72s   172.18.0.3   kpng-e2e-ipv4-ebpf-control-plane
 kube-scheduler-kpng-e2e-ipv4-ebpf-control-plane       1/1     Running   0          71s   172.18.0.3   kpng-e2e-ipv4-ebpf-control-plane
 local-path-provisioner-5d7949c7d4-tzf6d               1/1     Running   0          57s   10.1.0.4     kpng-e2e-ipv4-ebpf-control-plane
```

Figure 2.8: *Kubernetes Kube-Proxy functionality for ClusterIP Services.*

2.5 eBPF

2.5.1 The eBPF Virtual Machine

The eBPF, extended Berkeley Packet Filter, is an evolution of the original BSD packet filter (BPF) McCanne and Jacobson, 1993 which has seen extensive use in various packet-filtering applications over the last decades. BPF uses a register-based virtual machine to describe filtering actions. Support for compiling (restricted) C and Rust code into eBPF is included in the LLVM compiler infrastructure Lattner and Adve, 2004.

eBPF evolved to have a broader instruction set, including arithmetic and logic instructions and, mainly, a call instruction for function calls. It adopts the C language conventions used on the architectures supported by the kernel. eBPF also increased register width to 64-bit, enabling one-to-one mapping to hardware registers on the 64-bit architectures supported by the kernel, making efficient just-in-time (JIT) compilation into native machine possible. Even further, it is achievable to map a BPF call instruction to a single native call instruction, enabling function calls with close to zero additional overhead, both in eBPF helper functions and calls within the same program.

It is important to notice that, although most eBPF programs are written in C, a very important component of its architecture is the verifier, which places limitations on the programs loaded into the kernel to ensure that the user-supplied programs cannot harm the running kernel. The verifier is further approached in Subsection 2.5.3.

Due to the security provided by the verifier, it is safe to execute the code directly in the kernel address space, which makes eBPF useful for a wide variety of tasks in the Linux kernel. As all programs can share the same set of maps, they are able to react to arbitrary events in many distinct parts of the kernel.

The Figure 2.9 shows the general interaction between user space and kernel space, which occurs through the `bpf` syscall.

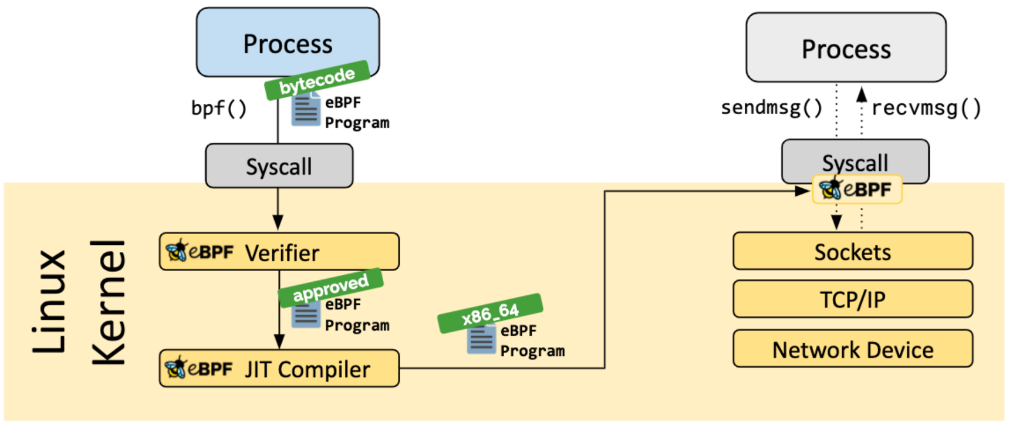


Figure 2.9: *The interaction begins when a user space process executes the first in the series of bpf syscalls used to load an eBPF program into the kernel. The kernel then runs the verifier, which enforces constraints that ensure the eBPF program is valid (more on that later). If the verifier approves the program, the verifier will finalize the process of loading it into the kernel, and it will run when it is triggered.*

2.5.2 eBPF Maps

eBPF programs are executed in response to an event in the kernel (a packet arrival, file creation, a specific system call, etc.). Each time they are executed they start in the same initial state, and they do not have access to persistent memory storage in their program context. Instead, the kernel exposes helper functions giving programs access to BPF maps.

Maps are key/value stores that are defined upon loading an eBPF program and can be referred to from within the eBPF code. They can exist in both global and per-CPU variants, and can be shared, both between different eBPF programs running at various places in the kernel, as well as between eBPF and userspace. In packet-filter applications, it is common for a map to be fully operated by a userspace program, meanwhile, a responsible eBPF program just accesses the map to make the routing decisions.

They can have different types, e.g. generic hash maps, arrays, and radix trees, as well as specialized types containing pointers to eBPF programs (used for tail calls between eBPF programs).

2.5.3 eBPF Verifier

Since eBPF code runs directly in the kernel address space, it can directly access, and potentially corrupt, arbitrary kernel memory.

To prevent this from happening, the kernel enforces a single entry point for loading all eBPF programs (through the `bpf()` system call). When loading an eBPF program it is first analysed by the in-kernel eBPF verifier. The verifier performs a static analysis of the program byte code to ensure that the program performs no actions that are unsafe (such as accessing arbitrary memory), and that the program will terminate. The latter is ensured by disallowing loops and limiting the maximum program size.

The verifier operates through a two-stage process. Initially, it constructs a directed acyclic graph (DAG) representing the control flow of the program. In the first stage, a depth-first search is conducted on this DAG to confirm its acyclic nature, meaning it should not contain loops, and to ensure that there are no unsupported or unreachable instructions. The aim here is to validate the fundamental structure of the program.

In the second stage, the verifier meticulously examines all possible paths within the DAG. This pass is designed to verify that the program exclusively conducts safe memory accesses and that any invoked helper functions receive the correct argument types. The verifier enforces safety by rejecting programs that contain load or call instructions with invalid arguments. To ascertain the validity of these arguments, the verifier carefully tracks the state of all registers and stack variables throughout the program's execution. This comprehensive analysis during both stages ensures that only secure and well-structured eBPF programs are allowed to execute within the kernel.

The goal of this register state tracking mechanism is to make sure the program doesn't access memory in places it shouldn't, even when we don't know the exact boundaries beforehand. We can't know the boundaries because programs have to handle data packets of various sizes, and the content of maps isn't known in advance, so we can't be sure if a particular lookup will succeed. To address this, the verifier ensures that the program itself checks for boundaries before using data from packets and verifies that map lookups don't involve NULL values before using them. This approach gives control to the program writer for how these checks are implemented The Linux Kernel documentation, 2023.

The verifier's main role is to protect the kernel from potentially harmful or buggy eBPF programs rather than optimizing their performance. Loading programs requires administrative privileges, and it's up to eBPF programmers to avoid such issues.

2.5.4 XDP: the eXpress Data Path

XDP, or eXpress Data Path, is a powerful packet processing framework tightly integrated into the Linux kernel. It operates just above the network device driver, offering the capability to handle packets with remarkable speed and efficiency. XDP is designed for low-level packet processing, making it well-suited for scenarios where rapid packet analysis and manipulation are essential. It accomplishes this by leveraging the Berkeley Packet Filter (BPF) technology, allowing for the execution of user-defined programs in the kernel to filter, modify, or drop packets in real-time.

In terms of performance, XDP is renowned for its exceptional capabilities. It excels in environments that demand low-latency, high-throughput packet processing, such as network function virtualization (NFV), data centers, and cloud infrastructures. This framework can work seamlessly alongside other Linux networking technologies like Traffic Control (TC) and routing, enabling complex network policy enforcement and packet steering based on application requirements. Furthermore, XDP is an open-source solution, meaning it's accessible to a broad community of users and developers, fostering collaboration and innovation in the realm of high-performance networking Høiland-Jørgensen *et al.*, 2018.

In Figure 2.10, there's a diagram showing how XDP integrates with the Linux network stack.

The XDP system is composed of four major components, they being:

1. The **eBPF virtual machine** executes the byte code of the XDP program and just-in-time-compiles it for increased performance.
2. **BPF maps** are key/value stores that serve as the primary communication channel to the rest of the system.
3. The **eBPF verifier** statically verifies programs before they are loaded to make sure they do not crash or corrupt the running kernel.
4. The **XDP driver hook** is the main entry point for an XDP program and is executed when a packet is received from the hardware.

2.5.5 cgroup/connect4

The current eBPF implementation uses a hook at the cgroup/connect4 kernel function. Cgroups (short for control groups) are a Linux Kernel feature that allows grouping

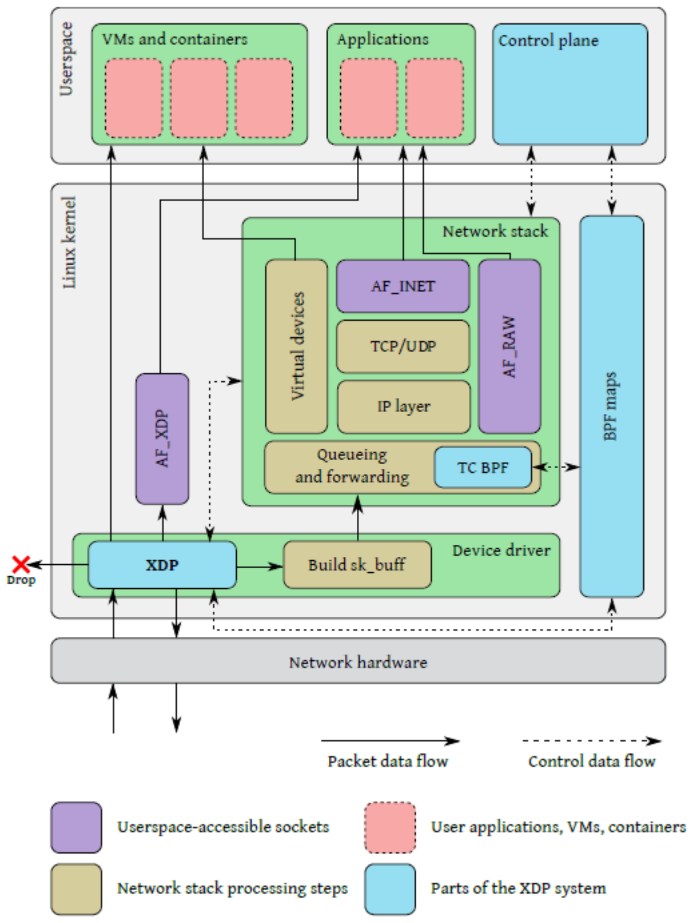


Figure 2.10: *When a packet arrives, a special program runs in the device driver before handling the packet data. This program can choose to do various things like dropping packets, sending them back through the same interface, redirecting them to other interfaces (including virtual machines), or passing them to userspace via special AF_XDP sockets. It can also allow packets to continue into the regular network stack for further processing using a separate TC BPF hook. These programs can communicate with each other and with userspace through BPF maps. The source of this Figure is the original XDP paper* Høiland-Jørgensen *et al., 2018.*

processes and managing, allocating, and restricting their access to system resources. The connect4 hook is executed after every IPv4 packet is received in the kernel. The eBPF function has a socket buffer, a fundamental data structure in the Linux networking code that contains basic information about a packet at the network and transport layer levels, which can be modified during its execution. In the implementation, a modification of source and destination IP addresses and ports takes place to forward the packet to the correct service backend.

2.6 Observability Tooling

This section will go over the technologies used to create metrics, monitor, and compare the eBPF backend to the `iptables` one.

2.6.1 Prometheus

Prometheus is a powerful open-source monitoring and alerting toolkit designed for achieving comprehensive observability in modern, dynamic environments. It is a Cloud Native Computing Foundation (CNCF) project that has gained widespread adoption in the realm of DevOps and system administration. At its core, Prometheus is designed to collect and store time-series data, making it particularly well-suited for monitoring the performance and health of various components within a distributed system.

One key feature of Prometheus is its ability to scrape and collect metrics from diverse sources, such as applications, services, and infrastructure components. It utilizes a pull-based model, where Prometheus servers periodically query designated endpoints, known as exporters, to retrieve metrics data. This flexibility allows Prometheus to adapt to dynamic and cloud-native environments, where components may be constantly scaling up or down.

In addition to its data collection capabilities, Prometheus provides a powerful query language called PromQL, enabling users to perform sophisticated analysis on the collected metrics. This includes aggregations, filtering, and transformation of data to derive meaningful insights into system behavior. Furthermore, Prometheus supports alerting rules, allowing users to define conditions based on the collected metrics and receive notifications when certain thresholds are reached. This combination of data collection, querying, and alerting makes Prometheus a crucial component in the observability stack, empowering organizations to proactively monitor and troubleshoot issues in their applications and infrastructure.

In the Prometheus monitoring and alerting toolkit, a Golang library is available with a set of functions that allow applications to export custom metrics. For instance, `prometheus.NewSummaryVec` and `prometheus.NewHistogramVec` are functions used to create vectorized versions of summary and histogram metrics, respectively. These vectorized versions allow you to define and work with multiple instances of the same metric, each distinguished by a set of labeled dimensions. This capability is particularly useful in scenarios where you want to differentiate metrics based on specific characteristics or attributes of the observed entities.

NewSummaryVec

NewSummaryVec is used to create a vectorized version of the summary metric type in Prometheus. A summary metric is designed for tracking the distribution of observed values over time, providing insights into the quantiles (percentiles) of a dataset. The vectorized version enables you to create multiple summaries, each identified by a unique set of labels. For example, you might use this to track the response time percentiles for different API endpoints, and the sender and receiver Kubernetes nodes.

The code below shows how the summary metric for an HTTP request latency is declared. It has only a name, a description, and a list of strings. The list of strings is the parameters/dimensions of the given metrics. In this example, a user would be able to filter the latencies of a specific endpoint or a specific server node.

```
summaryVec := prometheus.NewSummaryVec(
    prometheus.SummaryOpts{
        Name: "http_request_duration_seconds",
        Help: "Summary of HTTP request durations",
    },
    []string{"endpoint", "clientNodeName", "serverNodeName"},
)
```

Then, the code below shows how the client code can register a new latency after executing an HTTP request. Note that `endpoint`, `clientNodeName`, and `serverNodeName` are string variables.

```
start = time.Now()
// Perform HTTP request ...
end := time.Since(start)

summaryVec.
    WithLabelValues(endpoint, clientNodeName, serverNodeName).
    Observe(float64(end.Milliseconds()))
```

NewHistogramVec

NewHistogramVec is used to create a vectorized version of the histogram metric type in Prometheus. A histogram metric is ideal for observing the distribution of samples (such as request durations or response sizes) and calculating quantiles. The vectorized version allows you to create multiple histograms with different labels to distinguish between various dimensions or characteristics of the observed data.

The code below is very similar to `summaryVec` previously. The main difference is that `HistogramVecs` allows users to define the buckets into which observations are counted. Each element in the slice is the upper inclusive bound of a bucket. The values must be sorted in strictly increasing order.

```
histogramVec := prometheus.NewHistogramVec(
    prometheus.HistogramOpts{
        Name:    "http_request_duration_seconds",
        Help:    "Histogram of HTTP request durations",
        Buckets: []float64{1, 2, 4, 8, 16, 32, 64, 128, 256, 512},
    },
    []string{"endpoint", "clientNodeName", "serverNodeName"},
)
```

The API to register a new observation into an `HistogramVec` is the same as the `SummaryVec`:

```
start = time.Now()
// Perform HTTP request ...
end := time.Since(start)

histogramVec.
    WithLabelValues(endpoint, clientNodeName, serverNodeName).
    Observe(float64(end.Milliseconds()))
```

2.6.2 Grafana

Grafana is a widely adopted open-source platform designed for observability and visualization of time-series data. Originally released in 2014, Grafana has become a central component in many monitoring and analytics stacks, empowering users to create insightful dashboards, alerts, and explore their data in real-time. It plays a crucial role in enhancing the observability of complex systems by providing a unified and visually intuitive interface to analyze and understand metrics, logs, and traces.

One of Grafana's key strengths is its ability to integrate with various data sources, making it a versatile solution for different monitoring needs. It supports popular time-series databases such as Prometheus, InfluxDB, Graphite, and Elasticsearch, as well as relational databases like MySQL and PostgreSQL. This flexibility enables users to consolidate data from diverse sources into a single, cohesive dashboard for holistic insights into their infrastructure, applications, and services. Figure 2.11 has an example similar to what happens in the architecture used in this project, with Prometheus acting as the metrics repository.

Grafana's dashboard creation is both user-friendly and feature-rich. Users can design dashboards through an intuitive web interface, arranging panels that visualize different metrics or logs. The platform offers a wide range of visualization options, including graphs, tables, heatmaps, and more. Additionally, Grafana supports templating and annotation features, allowing users to dynamically explore and analyze data over time. A sample of Grafana's capabilities is shown in Figure 2.12.

The alerting capabilities of Grafana further contribute to its role in observability. Users can define alert rules based on the metrics displayed on their dashboards and receive notifications when predefined thresholds are crossed. Integration with various notification channels, including email, Slack, and others, ensures that critical information reaches the right people at the right time.

Grafana's vibrant community and extensive plugin ecosystem contribute to its con-

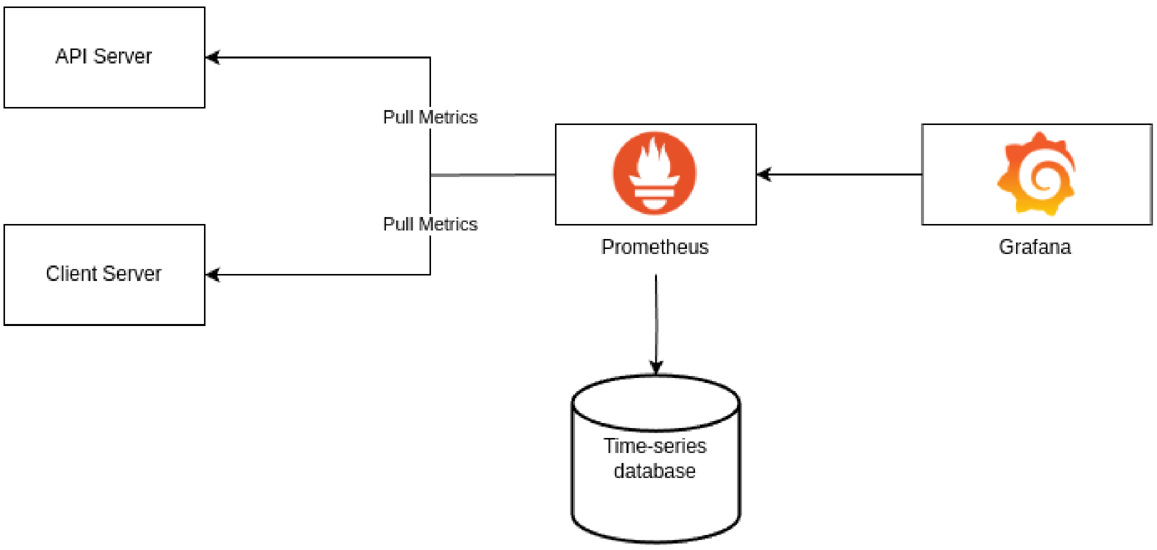


Figure 2.11: *Observability stack with Grafana querying from Prometheus, who is responsible for scraping metrics from API and client servers.*

tinuous evolution and adaptation to emerging technologies. As organizations embrace cloud-native architectures and microservices, Grafana remains a go-to solution for visualizing and understanding the dynamic and complex nature of modern systems. Overall, Grafana's combination of flexibility, user-friendliness, and powerful features makes it an indispensable tool for teams seeking to enhance their observability and monitoring capabilities.

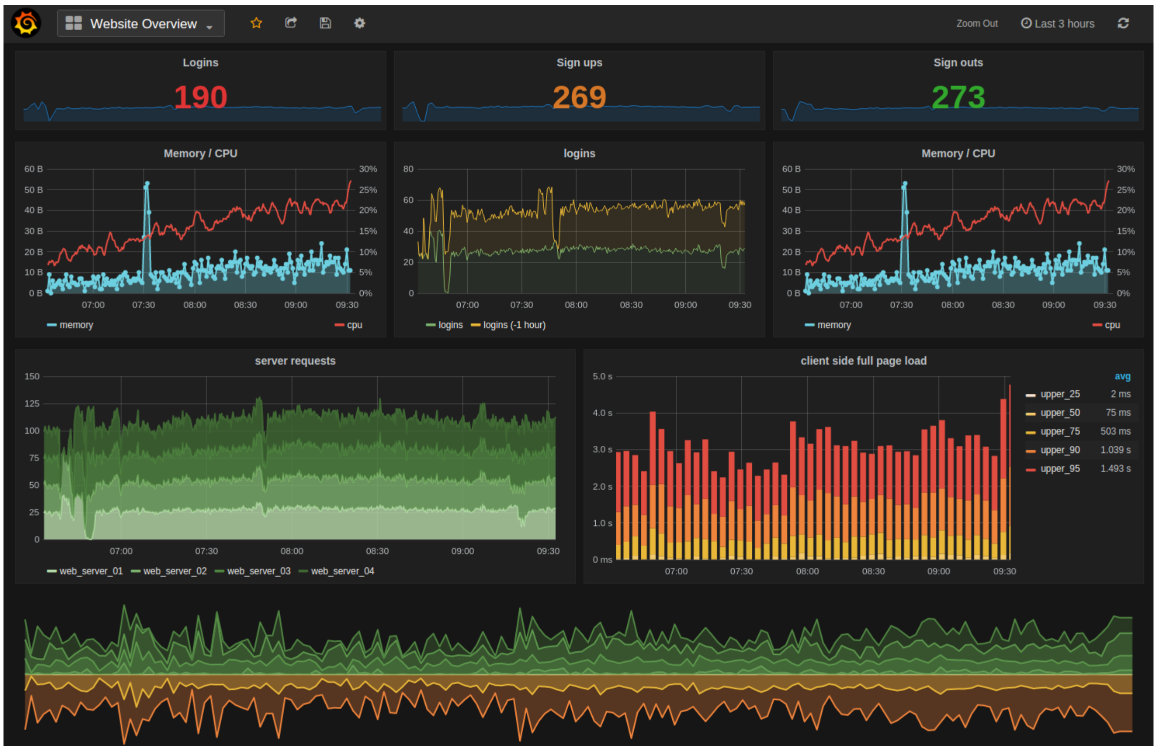


Figure 2.12: *Grafana example dashboard page. Source: Linux Screenshots, (2016, January 25th). https://www.flickr.com/photos/xmodulo/24311604930/. Accessed on January 21st, 2024.*

Chapter 3

Solution

Our objective is to create a reliable environment for testing the existing eBPF implementation of Kubernetes ClusterIP Service and then compare it to the current iptables kube-proxy. For that, a solution that constantly measures the latency between all of the nodes in the cluster and all of the Pods in the Service's fleet was created. This tool is called **k8s-node-latency** and will be further addressed in Subchapter 3.2.

Before getting into the details of the latency measuring solution, it is important to describe how the eBPF implementation works in conjunction with KPNG.

3.1 KPNG Implementation

Every service backend in KPNG implements the sink interface to retrieve the current status of the services. To implement this interface, it is necessary to set up callback functions for service state changes: SetService(), DeleteService(), SetEndpoint(), DeleteEndpoint(). These callback functions call routines from the underlying tool used to properly create NAT rules for the packets. An overview of this process can be seen in Figure 3.1.

The eBPF code of this implementation makes use of two eBPF features: *eBPF maps* and a *cgroup/connect4* hook. eBPF maps are used to copy Kubernetes services' state data to the BPF environment, allowing the hook to use this data for correct address translations. To create and update these maps every time a change is made to the services, the callback functions call *makeebpfmaps()*, which correctly parses the services' data and uses system calls to generate/update the maps, as shown in Figure 3.2.

The structure of the maps is designed in a way that when a new packet arrives at the *cgroup/connect4* hook, it uses the IP destination of the packet to look in the services. If it finds that the packet is directed at a running service, it then randomly chooses a backend (Pod) and retrieves its address through the eBPF maps, as described in Figure 3.3.

Once the hook function has the proper address of the backend pod that will receive the request, it can alter the destination IP and port of the packet and send it up through the Linux kernel network stack.

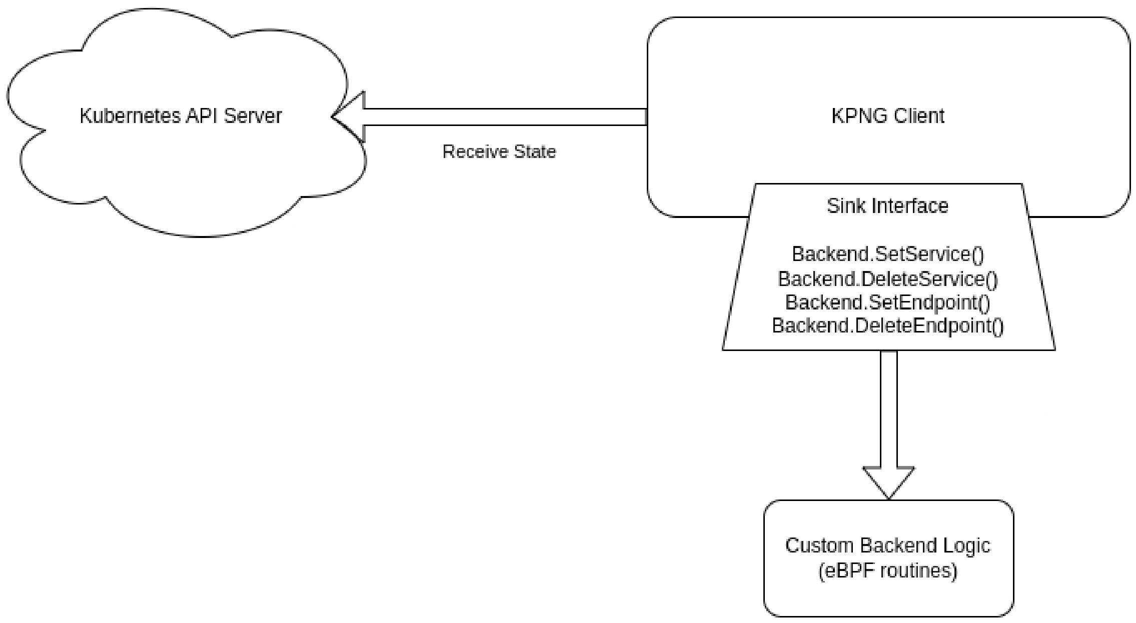


Figure 3.1: *Overview of the data consumption model of the KPNG backend.*

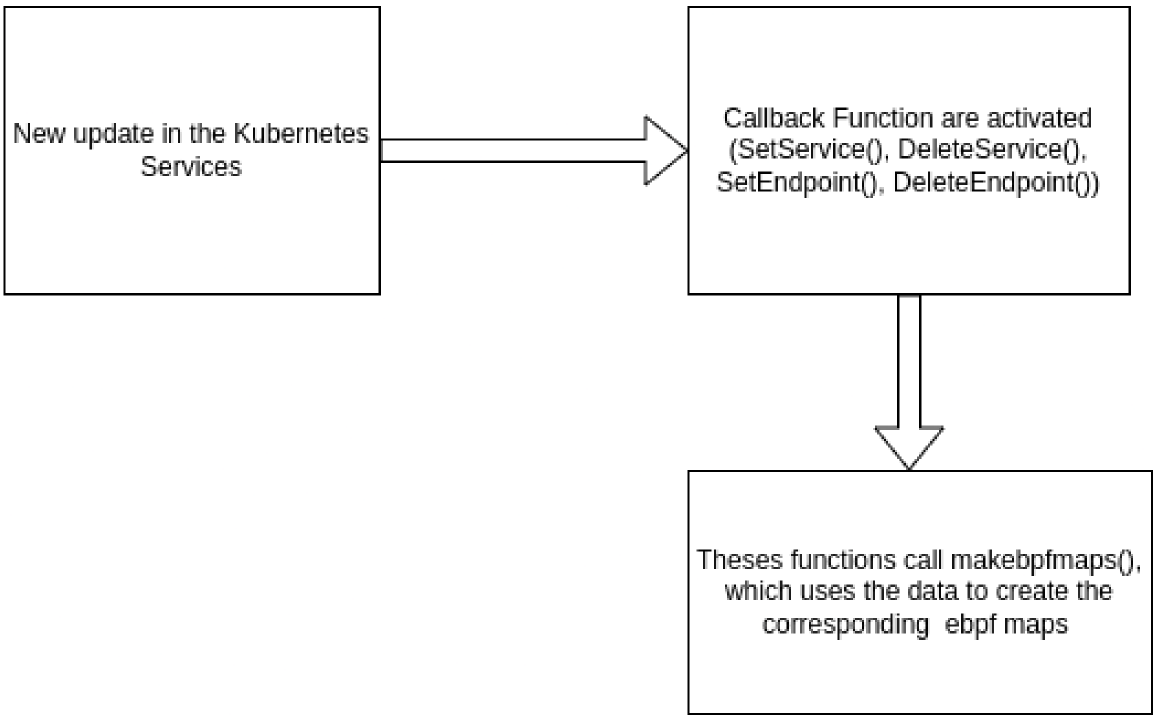

New update in the Kubernetes Services
Callback Function are activated (SetService(), DeleteService(), SetEndpoint(), DeleteEndpoint())
Theses functions call makebpfmaps(), which uses the data to create the corresponding ebpf maps

Figure 3.2: *Triggering of callback functions by the KPNG backend.*

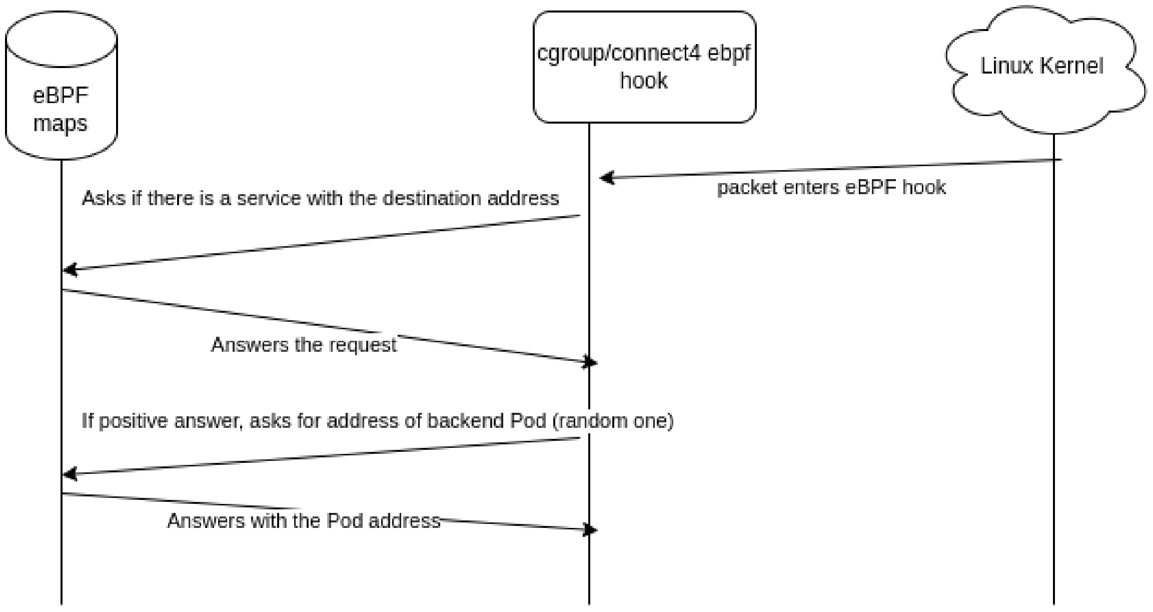

eBPF maps
cgroup/connect4 ebpf hook
Linux Kernel
packet enters eBPF hook
Asks if there is a service with the destination address
Answers the request
If positive answer, asks for address of backend Pod (random one)
Answers with the Pod address

Figure 3.3: *Sequence diagram illustrating the interaction between the eBPF function, eBPF maps, and the Linux kernel.*

3.2 k8s-node-latency

The main implementation done in this capstone project is **k8s-node-latency**. It is available as free and open-source software in https://github.com/jaehnri/k8s-node-latency. k8s-node-latency was developed in Golang and is deployed as a client-server architecture, where the client pods are deployed once per node, and an arbitrary number of server pods are distributed between all Kubernetes nodes. Note that the clients are deployed as a DaemonSet and the servers are deployed as a Deployment.

The objective of k8s-node-latency is to measure the latency between two Kubernetes nodes (or even the latency inside a node itself). Modern Kubernetes clusters are usually composed of tens of nodes that may be distributed across different regions. Thus, it is very important to monitor and consider the average request time from each node to each other. This is also a very good way of health-checking the nodes. One scenario of k8s-node-latency being deployed in a cluster is shown in Figure 3.4.

A few times per second [1], the client calls the server's ClusterIP and receives an OK from a random server pod. For every request, the client registers a few different metrics:

- **http_connection_request_duration_ms**: time to complete the connection's dial, i.e., to establish the connection.
- **http_first_byte_request_duration_ms**: time to receive the first byte of the response headers.
- **http_total_ping_request_duration_ms**: total time to send the request and receive the response. This is what we will call *round-trip latency* from now on.

[1] The exact parameters used in the tests will be detailed in the next chapter.

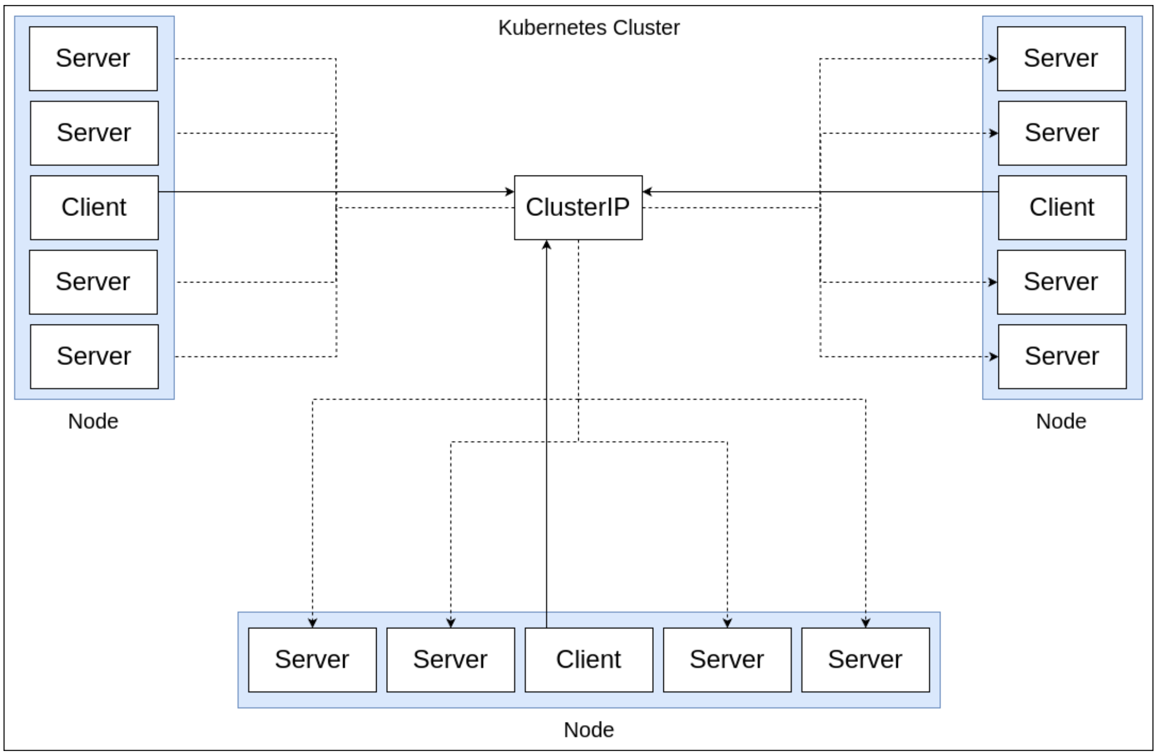


Figure 3.4: *Overview of the architecture of k8s-node-latency. In this example, there's a Kubernetes cluster with 3 nodes, represented by the blue panels. In the image, we can see there are 3 clients and 12 server pods equally distributed across all nodes. Clients access the servers through the ClusterIP, represented by the thick lines. The ClusterIP then does Layer 4 NAT to one of the servers, represented by the dashed lines.*

The server response is also composed of three important pieces of information that are parsed by the client and incorporated into its metrics:

```
type NodeLatencyResponse struct {
        OneTripTime    time.Time ‘json:"oneTripTime"‘
        ServerNodeName string    ‘json:"serverNodeName"‘
        ServerPodName  string    ‘json:"serverPodName"‘
}
```

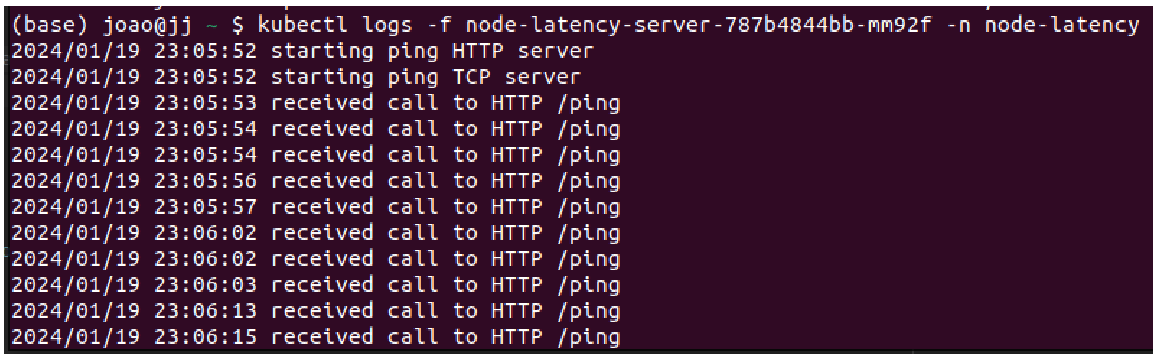
`OneTripTime` represents the time when the request was received by the server. This metric is important because, essentially, the client-to-server path is where the ClusterIP NAT happens, so it is possible to compare this time to the request start time to measure what we call **one-way-trip latency**. However, it is important to notice that if the client and the server aren't in the same node or physical computer, the times may not be synchronized, so the one-way-trip latency must be used carefully.

The other two pieces of information, `ServerNodeName` and `ServerPodName` respond to the names of the server node and pod, respectively. As mentioned in Subchapter 2.6.1, metrics are usually time-series dimensional data. The dimensions of each metric shown above are `ClientPodName`, `ClientNodeName`, `ServerPodName`, and `ServerNodeName`. This way, it is possible to generate charts that compare latencies from one node to another, for instance, *average latency from node X to node Y*.

3.2.1 Server

The server exposes a TCP server on port 3000 and an HTTP server on port 8080, serving an API that has two endpoints: `/ping` and `/metrics`.

Whenever a request comes in, the server increments the number of requests it already received and logs it. The logs exemplify its work as in Figure 3.5.



```
(base) joao@jj ~ $ kubectl logs -f node-latency-server-787b4844bb-mm92f -n node-latency
2024/01/19 23:05:52 starting ping HTTP server
2024/01/19 23:05:52 starting ping TCP server
2024/01/19 23:05:53 received call to HTTP /ping
2024/01/19 23:05:54 received call to HTTP /ping
2024/01/19 23:05:54 received call to HTTP /ping
2024/01/19 23:05:56 received call to HTTP /ping
2024/01/19 23:05:57 received call to HTTP /ping
2024/01/19 23:06:02 received call to HTTP /ping
2024/01/19 23:06:02 received call to HTTP /ping
2024/01/19 23:06:03 received call to HTTP /ping
2024/01/19 23:06:13 received call to HTTP /ping
2024/01/19 23:06:15 received call to HTTP /ping
```

Figure 3.5: *Sample of the server logs.*

The `/metrics` endpoint exposes many boilerplate Prometheus metrics regarding CPU, memory, heap, cache, and garbage collector usage. Figure 3.6 shows that.

3.2.2 Client

The client initializes by picking up the ClusterIP from the Kubernetes API Server, as well as picking up its `nodeName` and `podName` from environment vars. This process is shown in Figure 3.7. The client also has an HTTP `/metrics` endpoint in port 8081.

Another important tool used by the client code is Golang's `net/http/httptrace` [2] library. This library allows one to put certain hooks in specific points during the HTTP request, enabling the dial and first byte metrics detailed previously. The code in 3.8 evidences it.

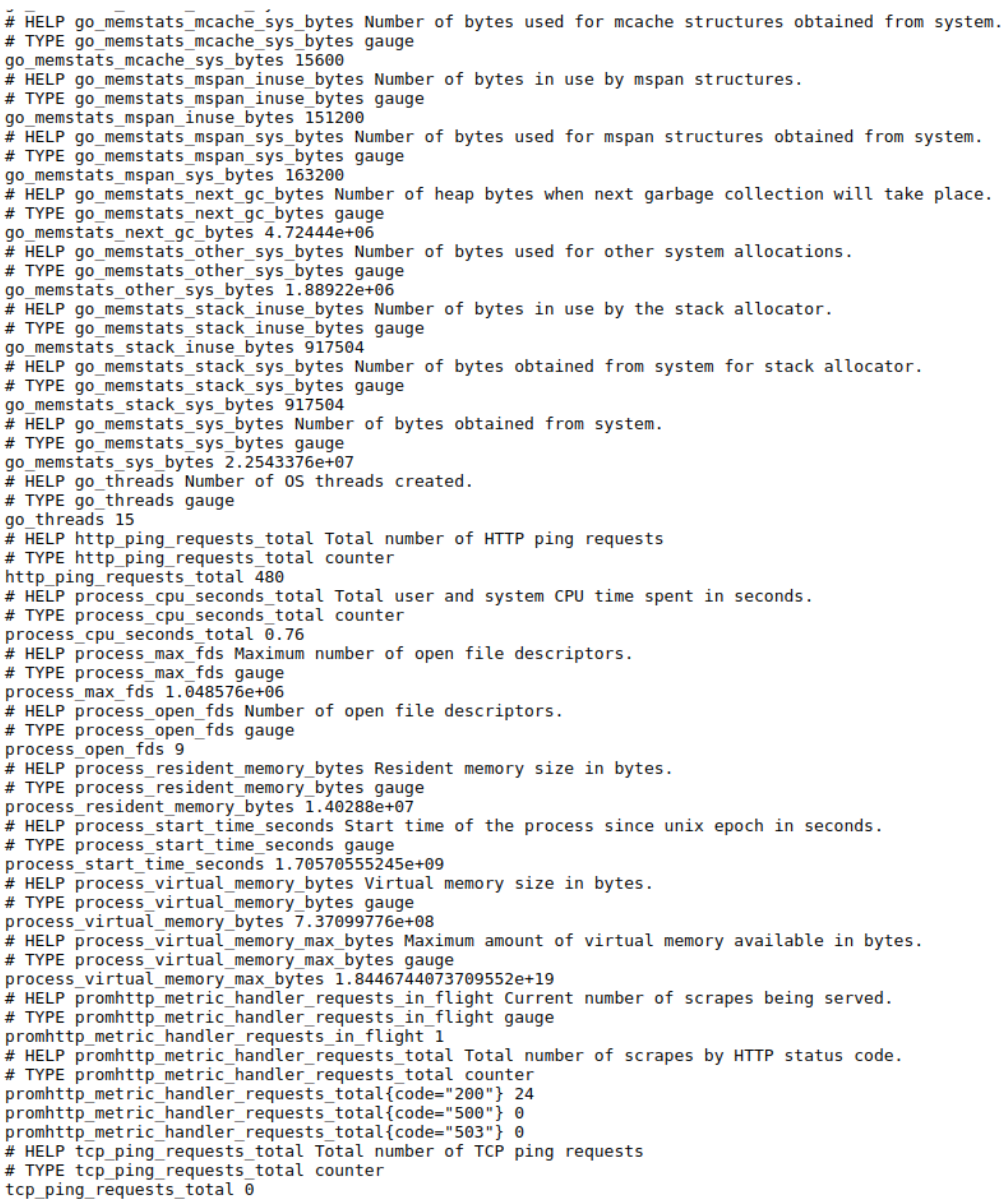
A few times per second, the client sends a ping request to collect more samples. One thing to notice is that the HTTP requests have the `keep-alive` set to false, otherwise, the client would always send requests to the same server, not going through the ClusterIP NAT being tested. A few logs in Figure 3.9 help to understand the client.

3.2.3 Monitoring

As mentioned before, k8s-node-latency generates metrics that can be tracked in real time. Therefore, this project is meant to be used alongside *Kube Prometheus Stack*, i.e., Prometheus and Grafana. A template dashboard for this project is available in the GitHub repository.

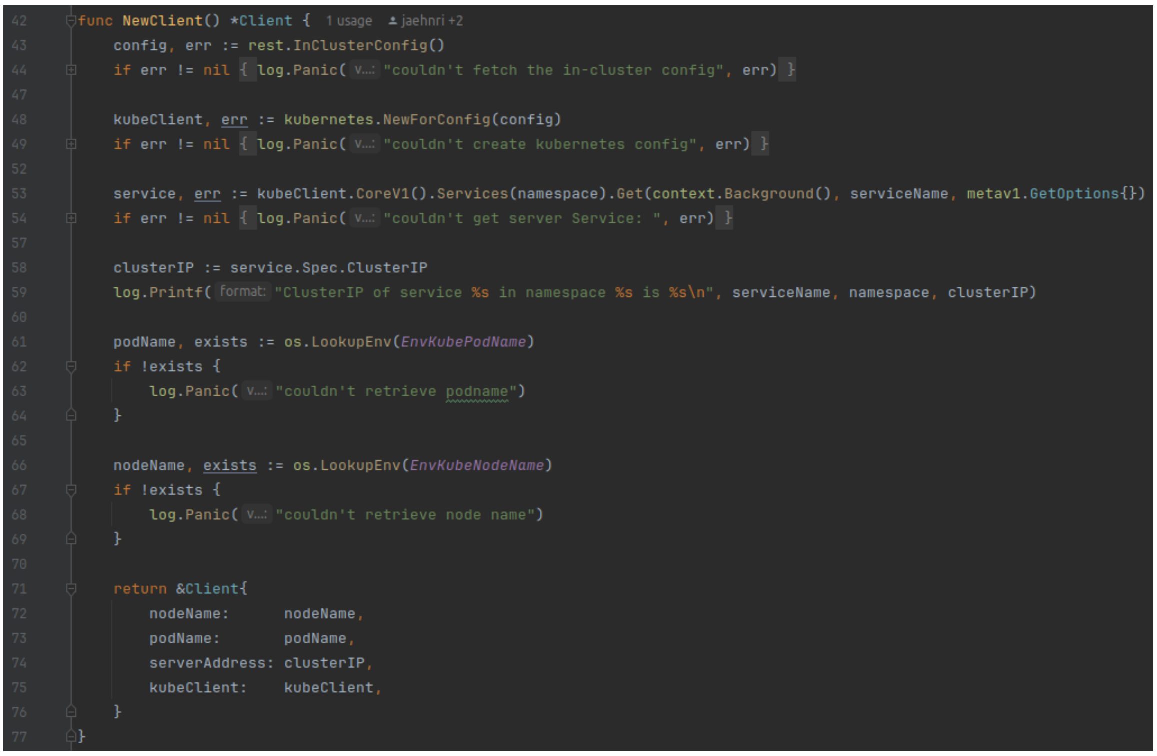
In figures 3.10 and 3.11, there are the template samples of real-time statistics and charts plotted by k8s-node-latency. Note that, by using the metrics detailed in the subsections

[2] httptrace is available at https://pkg.go.dev/net/http/httptrace



```
# HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures obtained from system.
# TYPE go_memstats_mcache_sys_bytes gauge
go_memstats_mcache_sys_bytes 15600
# HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.
# TYPE go_memstats_mspan_inuse_bytes gauge
go_memstats_mspan_inuse_bytes 151200
# HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures obtained from system.
# TYPE go_memstats_mspan_sys_bytes gauge
go_memstats_mspan_sys_bytes 163200
# HELP go_memstats_next_gc_bytes Number of heap bytes when next garbage collection will take place.
# TYPE go_memstats_next_gc_bytes gauge
go_memstats_next_gc_bytes 4.72444e+06
# HELP go_memstats_other_sys_bytes Number of bytes used for other system allocations.
# TYPE go_memstats_other_sys_bytes gauge
go_memstats_other_sys_bytes 1.88922e+06
# HELP go_memstats_stack_inuse_bytes Number of bytes in use by the stack allocator.
# TYPE go_memstats_stack_inuse_bytes gauge
go_memstats_stack_inuse_bytes 917504
# HELP go_memstats_stack_sys_bytes Number of bytes obtained from system for stack allocator.
# TYPE go_memstats_stack_sys_bytes gauge
go_memstats_stack_sys_bytes 917504
# HELP go_memstats_sys_bytes Number of bytes obtained from system.
# TYPE go_memstats_sys_bytes gauge
go_memstats_sys_bytes 2.2543376e+07
# HELP go_threads Number of OS threads created.
# TYPE go_threads gauge
go_threads 15
# HELP http_ping_requests_total Total number of HTTP ping requests
# TYPE http_ping_requests_total counter
http_ping_requests_total 480
# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.76
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.048576e+06
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 9
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 1.40288e+07
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.70570555245e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 7.37099776e+08
# HELP process_virtual_memory_max_bytes Maximum amount of virtual memory available in bytes.
# TYPE process_virtual_memory_max_bytes gauge
process_virtual_memory_max_bytes 1.8446744073709552e+19
# HELP promhttp_metric_handler_requests_in_flight Current number of scrapes being served.
# TYPE promhttp_metric_handler_requests_in_flight gauge
promhttp_metric_handler_requests_in_flight 1
# HELP promhttp_metric_handler_requests_total Total number of scrapes by HTTP status code.
# TYPE promhttp_metric_handler_requests_total counter
promhttp_metric_handler_requests_total{code="200"} 24
promhttp_metric_handler_requests_total{code="500"} 0
promhttp_metric_handler_requests_total{code="503"} 0
# HELP tcp_ping_requests_total Total number of TCP ping requests
# TYPE tcp_ping_requests_total counter
tcp_ping_requests_total 0
```

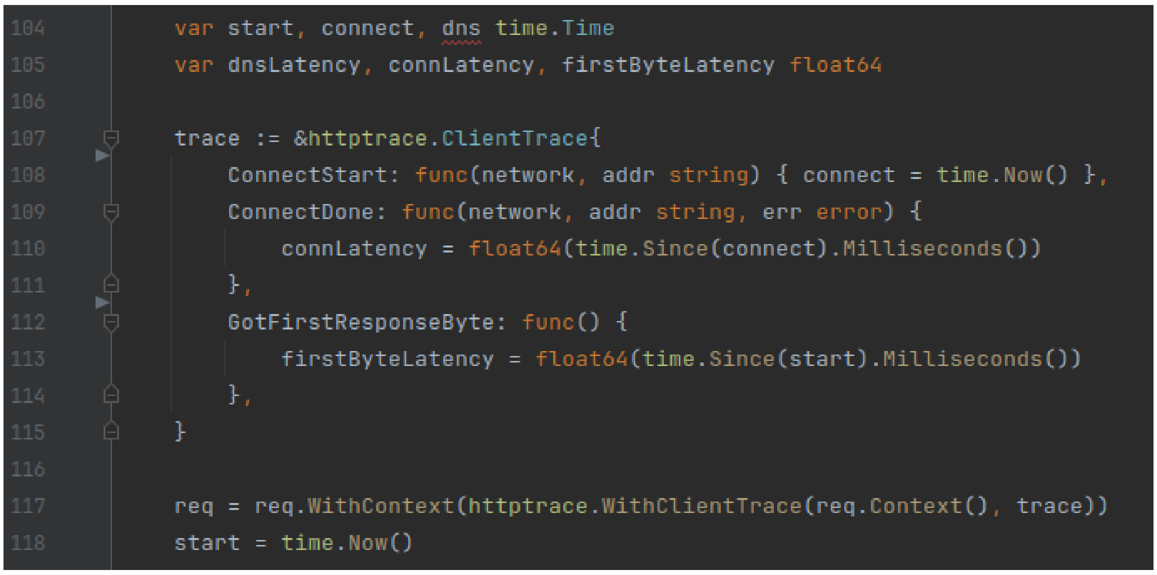
Figure 3.6: *Sample of the server metrics.*



```go
func NewClient() *Client {  1 usage  jaehnri +2
    config, err := rest.InClusterConfig()
    if err != nil { log.Panic( v...: "couldn't fetch the in-cluster config", err) }

    kubeClient, err := kubernetes.NewForConfig(config)
    if err != nil { log.Panic( v...: "couldn't create kubernetes config", err) }

    service, err := kubeClient.CoreV1().Services(namespace).Get(context.Background(), serviceName, metav1.GetOptions{})
    if err != nil { log.Panic( v...: "couldn't get server Service: ", err) }

    clusterIP := service.Spec.ClusterIP
    log.Printf( format: "ClusterIP of service %s in namespace %s is %s\n", serviceName, namespace, clusterIP)

    podName, exists := os.LookupEnv(EnvKubePodName)
    if !exists {
        log.Panic( v...: "couldn't retrieve podname")
    }

    nodeName, exists := os.LookupEnv(EnvKubeNodeName)
    if !exists {
        log.Panic( v...: "couldn't retrieve node name")
    }

    return &Client{
        nodeName:      nodeName,
        podName:       podName,
        serverAddress: clusterIP,
        kubeClient:    kubeClient,
    }
}
```

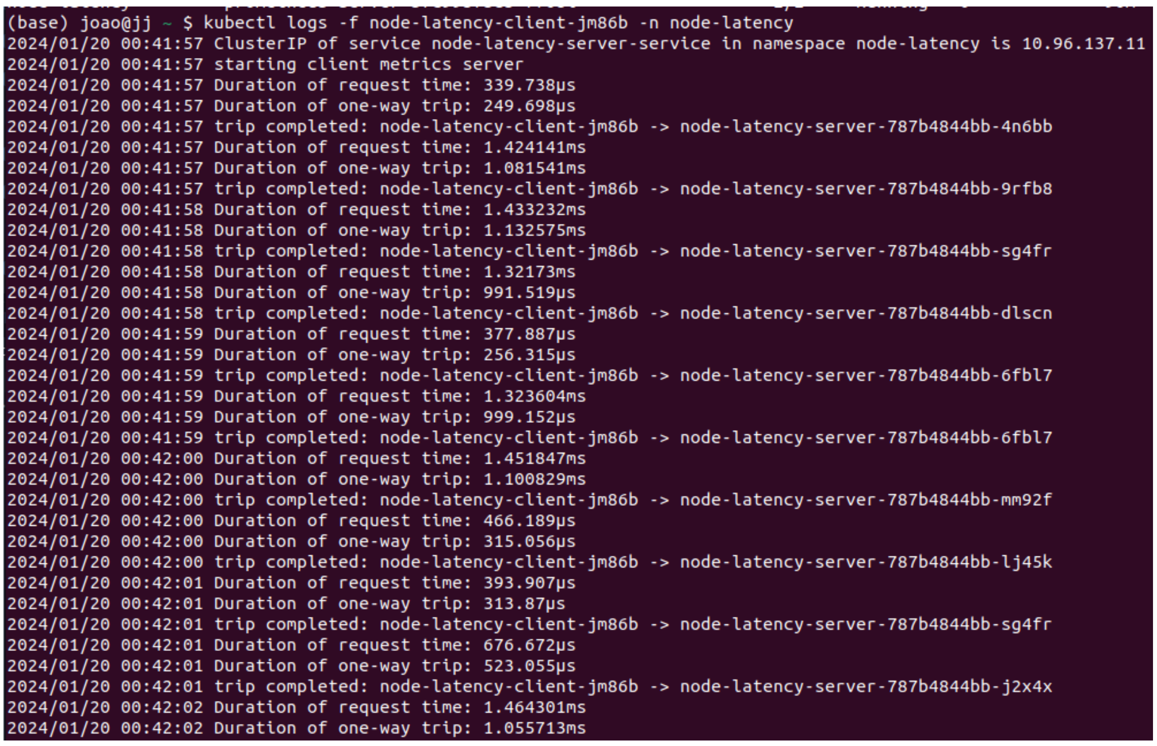
Figure 3.7: *Client initialization code. It is interesting to see how it uses the Kubernetes client-go* [a] *library to communicate with the API Server.*

[a] client-go library is available at https://github.com/kubernetes/client-go



```go
    var start, connect, dns time.Time
    var dnsLatency, connLatency, firstByteLatency float64

    trace := &httptrace.ClientTrace{
        ConnectStart: func(network, addr string) { connect = time.Now() },
        ConnectDone: func(network, addr string, err error) {
            connLatency = float64(time.Since(connect).Milliseconds())
        },
        GotFirstResponseByte: func() {
            firstByteLatency = float64(time.Since(start).Milliseconds())
        },
    }

    req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
    start = time.Now()
```

Figure 3.8: *The* `httptrace.ClientTrace` *object holds which hooks were chosen and what they should execute. In this case, the hooks measure how long it takes for the connection to be established and measure when the first response byte is available. This trace object then decorates the HTTP request context.* `start` *is the object that holds the UNIX time of when the request is sent.*



```
(base) joao@jj ~ $ kubectl logs -f node-latency-client-jm86b -n node-latency
2024/01/20 00:41:57 ClusterIP of service node-latency-server-service in namespace node-latency is 10.96.137.11
2024/01/20 00:41:57 starting client metrics server
2024/01/20 00:41:57 Duration of request time: 339.738µs
2024/01/20 00:41:57 Duration of one-way trip: 249.698µs
2024/01/20 00:41:57 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-4n6bb
2024/01/20 00:41:57 Duration of request time: 1.424141ms
2024/01/20 00:41:57 Duration of one-way trip: 1.081541ms
2024/01/20 00:41:57 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-9rfb8
2024/01/20 00:41:58 Duration of request time: 1.433232ms
2024/01/20 00:41:58 Duration of one-way trip: 1.132575ms
2024/01/20 00:41:58 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-sg4fr
2024/01/20 00:41:58 Duration of request time: 1.32173ms
2024/01/20 00:41:58 Duration of one-way trip: 991.519µs
2024/01/20 00:41:58 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-dlscn
2024/01/20 00:41:59 Duration of request time: 377.887µs
2024/01/20 00:41:59 Duration of one-way trip: 256.315µs
2024/01/20 00:41:59 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-6fbl7
2024/01/20 00:41:59 Duration of request time: 1.323604ms
2024/01/20 00:41:59 Duration of one-way trip: 999.152µs
2024/01/20 00:41:59 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-6fbl7
2024/01/20 00:42:00 Duration of request time: 1.451847ms
2024/01/20 00:42:00 Duration of one-way trip: 1.100829ms
2024/01/20 00:42:00 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-mm92f
2024/01/20 00:42:00 Duration of request time: 466.189µs
2024/01/20 00:42:00 Duration of one-way trip: 315.056µs
2024/01/20 00:42:00 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-lj45k
2024/01/20 00:42:01 Duration of request time: 393.907µs
2024/01/20 00:42:01 Duration of one-way trip: 313.87µs
2024/01/20 00:42:01 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-sg4fr
2024/01/20 00:42:01 Duration of request time: 676.672µs
2024/01/20 00:42:01 Duration of one-way trip: 523.055µs
2024/01/20 00:42:01 trip completed: node-latency-client-jm86b -> node-latency-server-787b4844bb-j2x4x
2024/01/20 00:42:02 Duration of request time: 1.464301ms
2024/01/20 00:42:02 Duration of one-way trip: 1.055713ms
```

Figure 3.9: *Sample of the client logs.*

above, one can extend and create custom dashboards to use other types of charts or even monitor specific nodes.

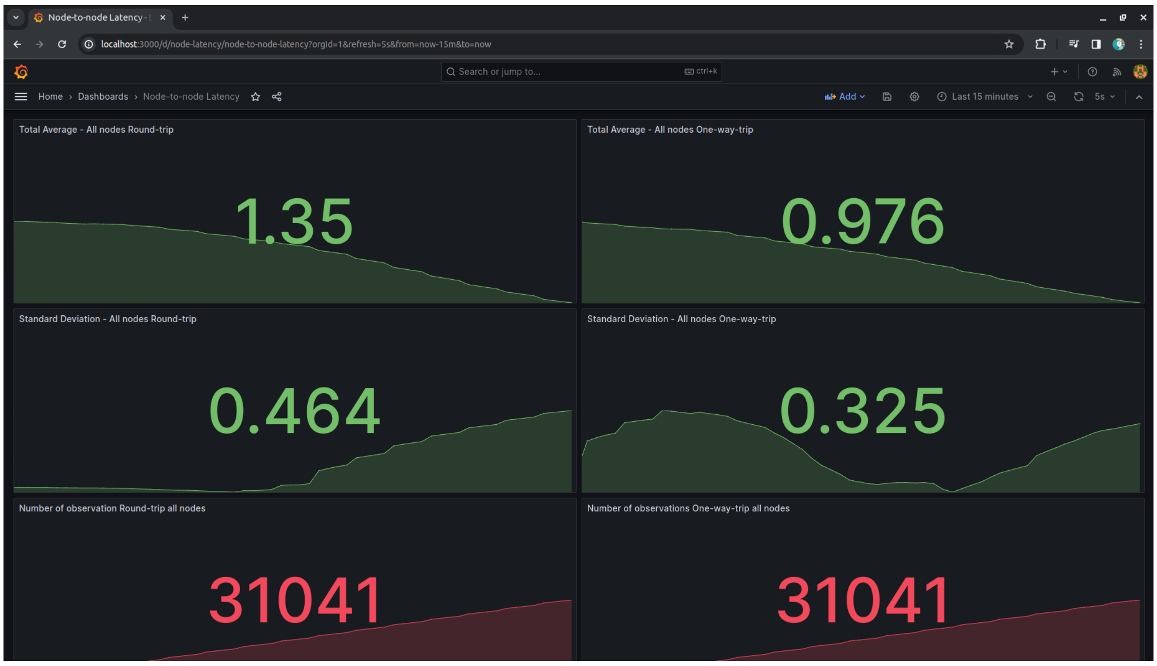
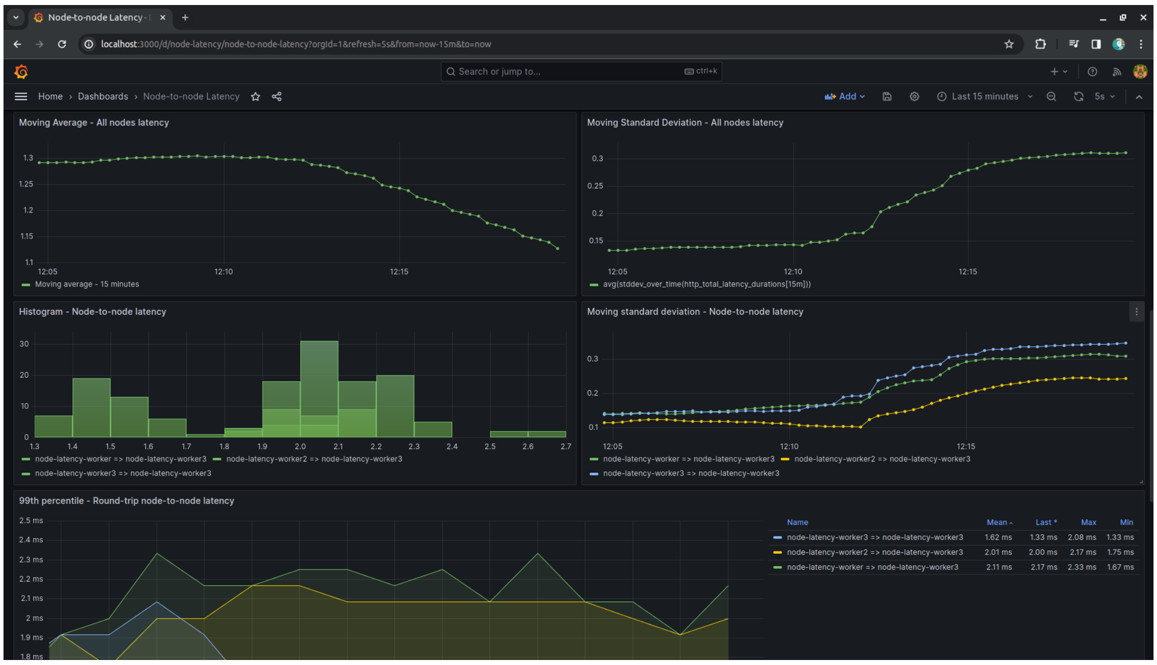

Node-to-node Latency
localhost:3000/d/node-latency/node-to-node-latency?orgId=1&refresh=5s&from=now-15m&to=now
Search or jump to...
Home › Dashboards › Node-to-node Latency
Add
Last 15 minutes
5s
Total Average - All nodes Round-trip
1.35
Total Average - All nodes One-way-trip
0.976
Standard Deviation - All nodes Round-trip
0.464
Standard Deviation - All nodes One-way-trip
0.325
Number of observation Round-trip all nodes
31041
Number of observations One-way-trip all nodes
31041

Figure 3.10: *k8s-node-latency stats dashboards. By default, k8s-node-latency collects the average latency and standard deviation of the last 15 minutes. However, these pieces of data are dimensional and can be extended/customized to build charts more appropriate to the user.*


Node-to-node Latency
localhost:3000/d/node-latency/node-to-node-latency?orgId=1&refresh=5s&from=now-15m&to=now
Search or jump to...
Home › Dashboards › Node-to-node Latency
Add
Last 15 minutes
5s
Moving Average - All nodes latency
Moving average - 15 minutes
Moving Standard Deviation - All nodes latency
avg(stddev_over_time(http_total_latency_durations[15m]))
Histogram - Node-to-node latency
node-latency-worker => node-latency-worker3
node-latency-worker2 => node-latency-worker3
node-latency-worker3 => node-latency-worker3
Moving standard deviation - Node-to-node latency
99th percentile - Round-trip node-to-node latency

	Name	Mean	Last *	Max	Min
🟦	node-latency-worker3 => node-latency-worker3	1.62 ms	1.33 ms	2.08 ms	1.33 ms
🟨	node-latency-worker2 => node-latency-worker3	2.01 ms	2.00 ms	2.17 ms	1.75 ms
🟩	node-latency-worker => node-latency-worker3	2.11 ms	2.17 ms	2.33 ms	1.67 ms

Figure 3.11: *k8s-node-latency node-to-node dashboards. By default, k8s-node-latency creates line charts that monitor the overall average latency and standard deviation. Moreover, it creates node-to-node histograms and line charts that monitor the latency between two nodes.*

Chapter 4

Results and Analysis

This chapter will focus on the test environment and show the results of the measurements performed. Later, a comparison between the iptables and the eBPF backends is done.

4.1 Test Environment

A Linux computer was used to test the backends. As these tests are mostly to measure the latencies of the NAT mechanism, using a single physical computer and simulating nodes with `kind` [1] was more appropriate. kind is a tool for running local Kubernetes clusters using *Docker container nodes*. kind was primarily designed for testing Kubernetes itself but may be used for local development or CI.

It is valid to point out that running these containerized nodes is somewhat expensive in terms of memory and therefore, this work did not focus on doing load tests or scalability tests. The main objective is to test the ClusterIP NAT translation under normal conditions. Table 4.1 has information on the test environment and test parameters.

Operating System	Ubuntu 22.04.3 LTS
Kernel	Linux 6.2.0-37-generic
CPU	Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
Motherboard	Z370M AORUS Gaming-CF
RAM	16 GiB DDR4 Synchronous Buffer
GPU	GP106 [GeForce GTX 1060 6GB]
Number of nodes	2
Number of clients	3 (1 per node)
Number of servers	12 (4 per node)
Client pings per second	2

Table 4.1: *Hardware specifications and parameters used in the iptables and eBPF backend tests.*

[1] kind is available at https://kind.sigs.k8s.io

4.2 iptables

Figures 4.1 to 4.8 present the results of the iptables backend performance test.

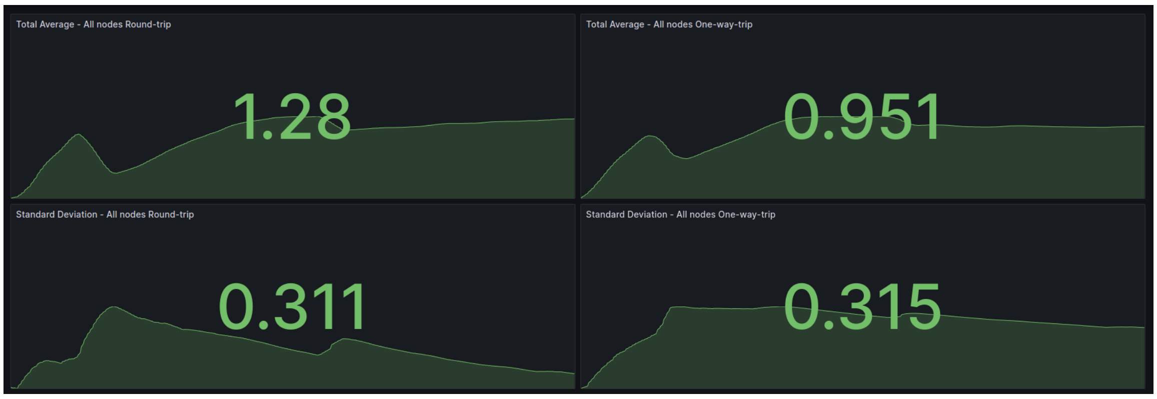

Total Average - All nodes Round-trip
1.28
Total Average - All nodes One-way-trip
0.951
Standard Deviation - All nodes Round-trip
0.311
Standard Deviation - All nodes One-way-trip
0.315

Figure 4.1: *iptables: Average and standard deviation gathered from all data points for both Round-trip and One-way-trip.*

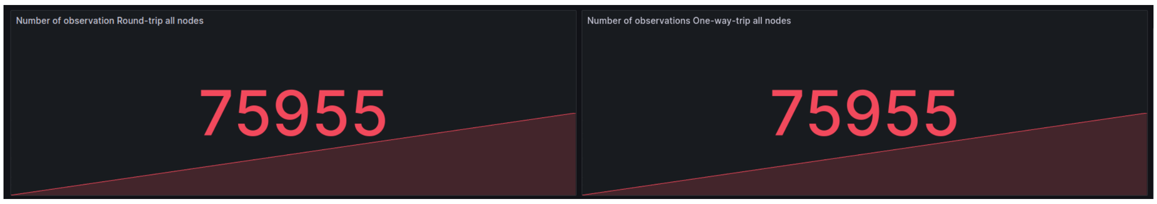

Number of observation Round-trip all nodes
75955
Number of observations One-way-trip all nodes
75955

Figure 4.2: *iptables: Total amount of data points for Round-trip and One-way-trip.*

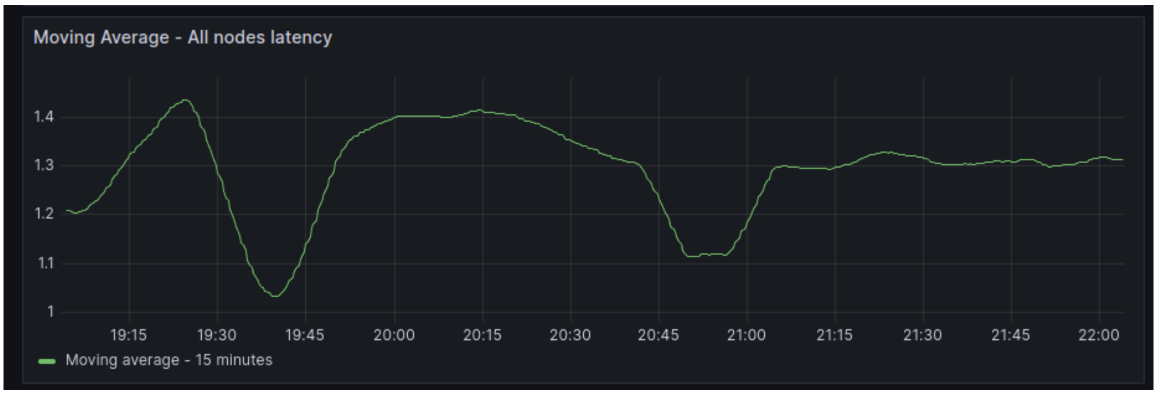


Figure 4.3: *iptables: Moving average of the Round trip over time.*

4.3 eBPF

Figures 4.9 to 4.16 present the results of the eBPF backend performance test.

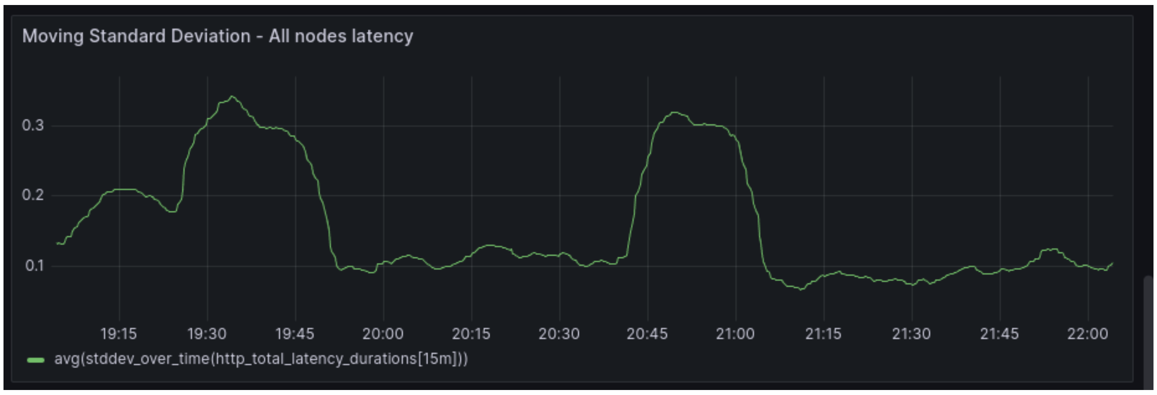


Figure 4.4: *iptables: Moving standard deviation of the Round-trip over time.*

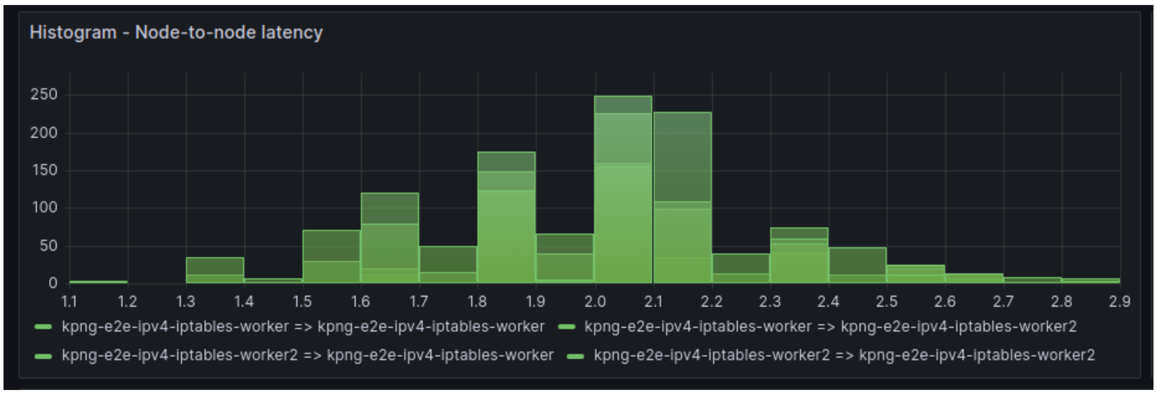
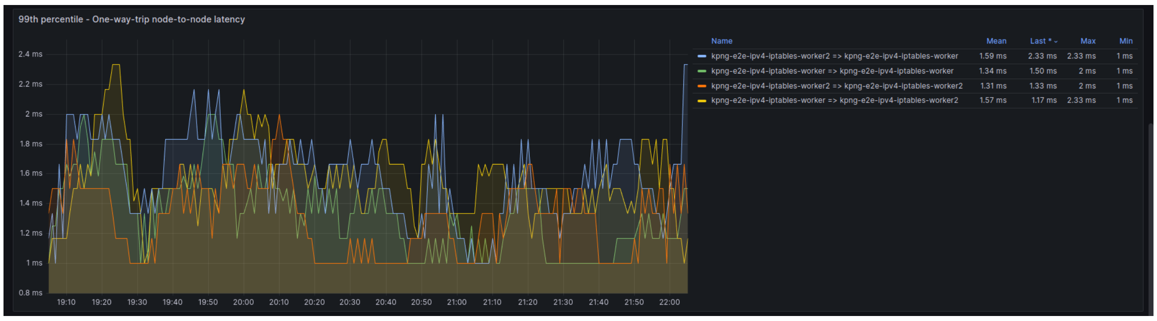
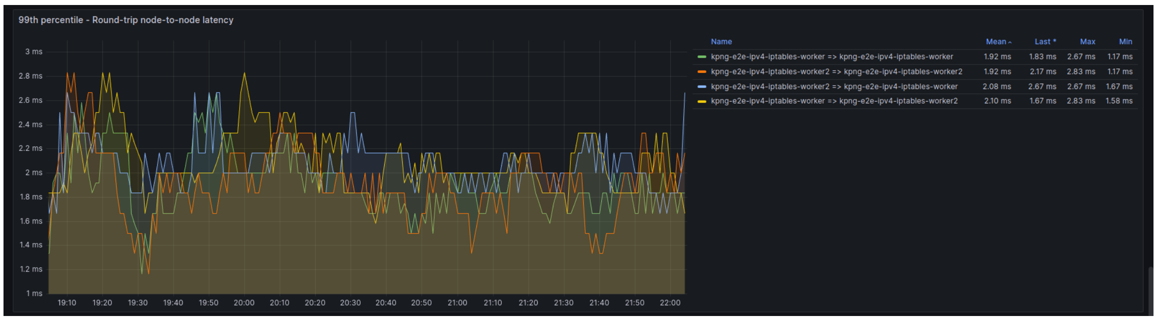


Figure 4.5: *iptables: Histogram of the distribution of Round-trip observations.*



Name	Mean	Last *	Max	Min
kpng-e2e-ipv4-iptables-worker2 => kpng-e2e-ipv4-iptables-worker	1.59 ms	2.33 ms	2.33 ms	1 ms
kpng-e2e-ipv4-iptables-worker => kpng-e2e-ipv4-iptables-worker	1.34 ms	1.50 ms	2 ms	1 ms
kpng-e2e-ipv4-iptables-worker2 => kpng-e2e-ipv4-iptables-worker2	1.31 ms	1.33 ms	2 ms	1 ms
kpng-e2e-ipv4-iptables-worker => kpng-e2e-ipv4-iptables-worker2	1.57 ms	1.17 ms	2.33 ms	1 ms

Figure 4.6: *iptables: Node to node One-way-trip data points over time.*



Name	Mean	Last *	Max	Min
kpng-e2e-ipv4-iptables-worker => kpng-e2e-ipv4-iptables-worker	1.92 ms	1.83 ms	2.67 ms	1.17 ms
kpng-e2e-ipv4-iptables-worker2 => kpng-e2e-ipv4-iptables-worker2	1.92 ms	2.17 ms	2.83 ms	1.17 ms
kpng-e2e-ipv4-iptables-worker2 => kpng-e2e-ipv4-iptables-worker	2.08 ms	2.67 ms	2.67 ms	1.67 ms
kpng-e2e-ipv4-iptables-worker => kpng-e2e-ipv4-iptables-worker2	2.10 ms	1.67 ms	2.83 ms	1.58 ms

Figure 4.7: *iptables: Node to node Round-trip data points over time.*

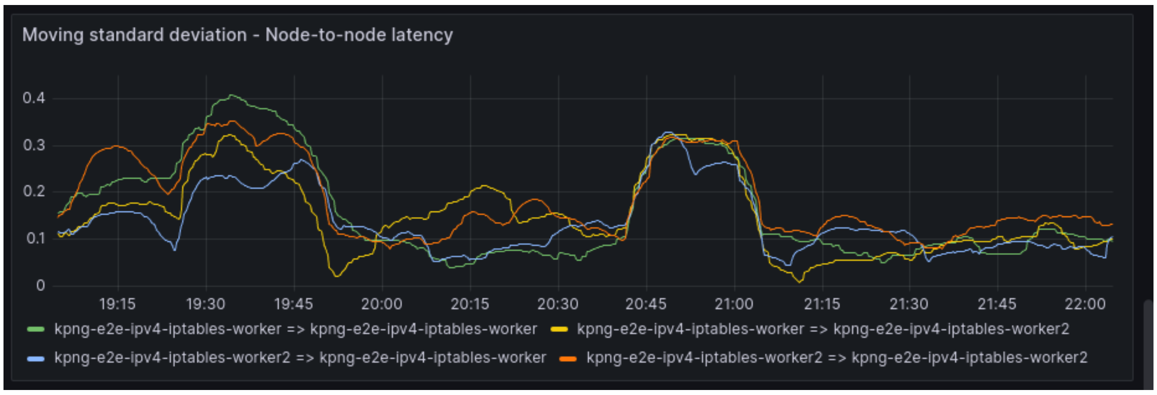


Figure 4.8: *iptables: Node to node standard deviation over time.*

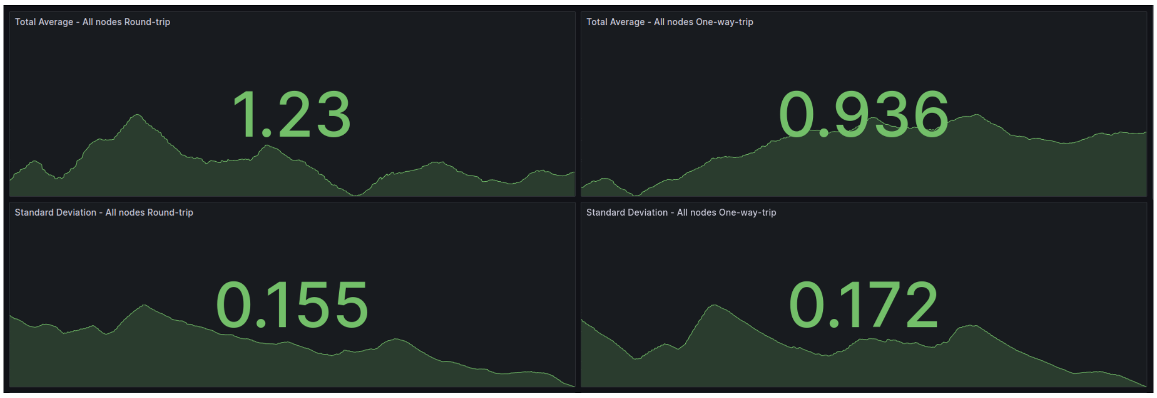

Total Average - All nodes Round-trip
1.23
Total Average - All nodes One-way-trip
0.936
Standard Deviation - All nodes Round-trip
0.155
Standard Deviation - All nodes One-way-trip
0.172

Figure 4.9: *eBPF: Average and standard deviation gathered from all data points for both Round-trip and One-way-trip.*

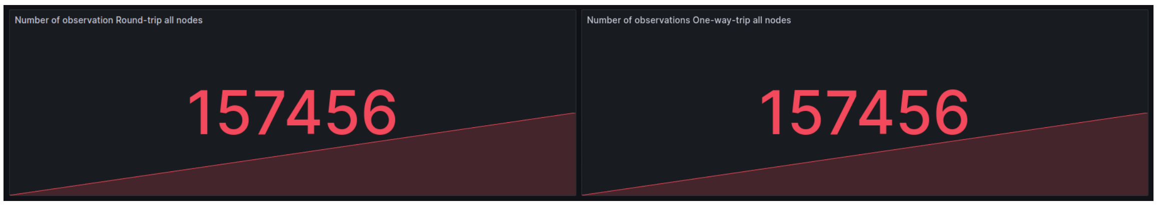

Number of observation Round-trip all nodes
157456
Number of observations One-way-trip all nodes
157456

Figure 4.10: *eBPF: Total amount of data points for Round-trip and One-way-trip.*

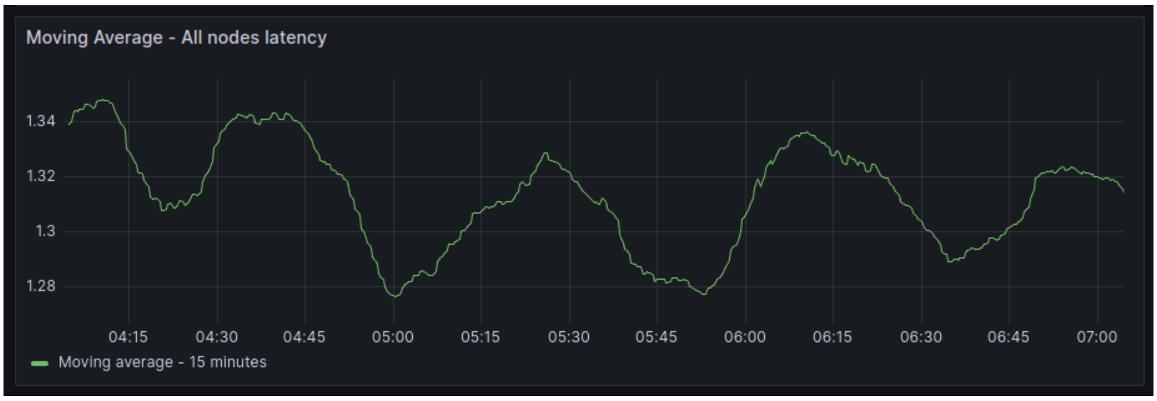


Figure 4.11: *eBPF: Moving average of the Round trip over time.*

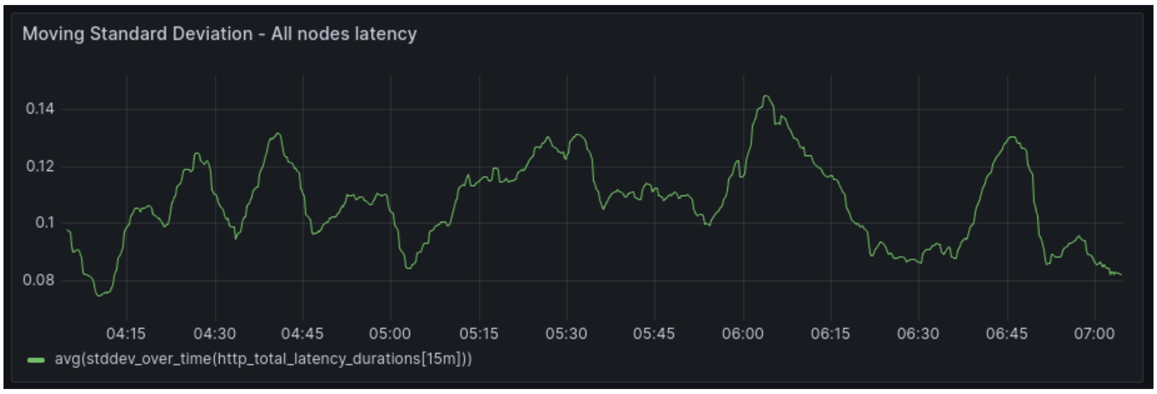


Figure 4.12: *eBPF: Moving standard deviation of the Round-trip over time.*

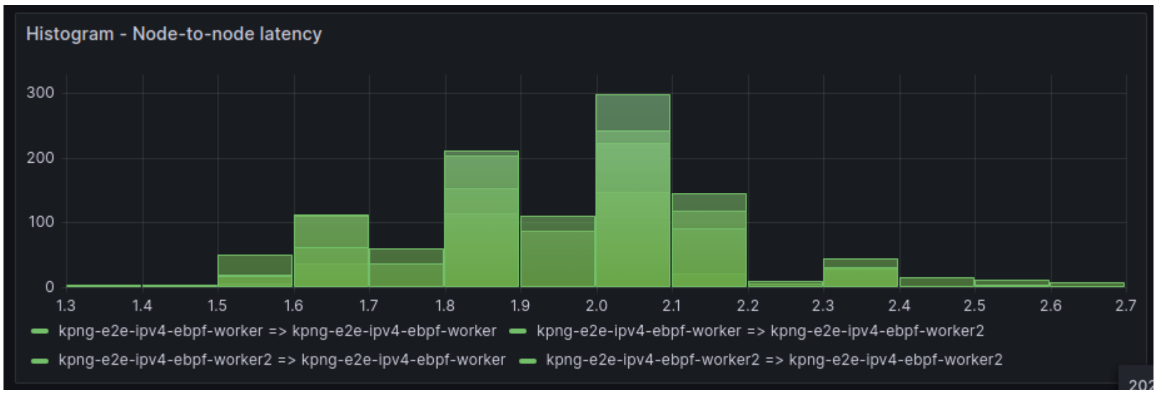
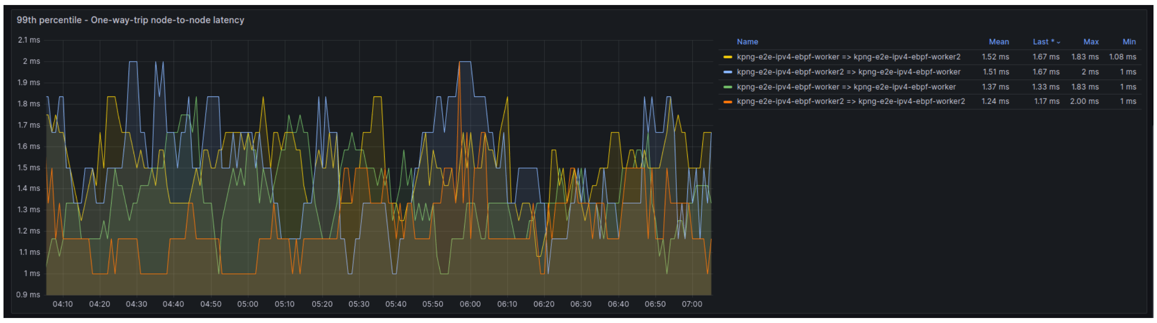
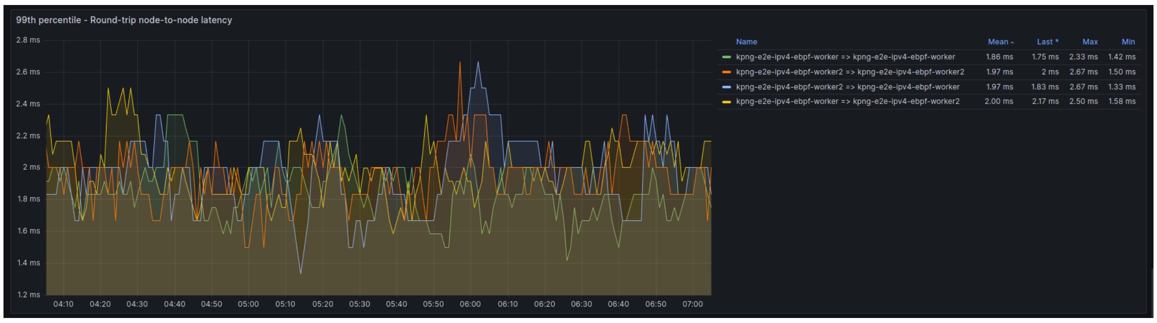


Figure 4.13: *eBPF: Histogram of the distribution of Round-trip observations.*



Name	Mean	Last *	Max	Min
kpng-e2e-ipv4-ebpf-worker => kpng-e2e-ipv4-ebpf-worker2	1.52 ms	1.67 ms	1.83 ms	1.08 ms
kpng-e2e-ipv4-ebpf-worker2 => kpng-e2e-ipv4-ebpf-worker	1.51 ms	1.67 ms	2 ms	1 ms
kpng-e2e-ipv4-ebpf-worker => kpng-e2e-ipv4-ebpf-worker	1.37 ms	1.33 ms	1.83 ms	1 ms
kpng-e2e-ipv4-ebpf-worker2 => kpng-e2e-ipv4-ebpf-worker2	1.24 ms	1.17 ms	2.00 ms	1 ms

Figure 4.14: *eBPF: Node to node One-way-trip data points over time.*



Name	Mean	Last *	Max	Min
kpng-e2e-ipv4-ebpf-worker => kpng-e2e-ipv4-ebpf-worker	1.86 ms	1.75 ms	2.33 ms	1.42 ms
kpng-e2e-ipv4-ebpf-worker2 => kpng-e2e-ipv4-ebpf-worker2	1.97 ms	2 ms	2.67 ms	1.50 ms
kpng-e2e-ipv4-ebpf-worker2 => kpng-e2e-ipv4-ebpf-worker	1.97 ms	1.83 ms	2.67 ms	1.33 ms
kpng-e2e-ipv4-ebpf-worker => kpng-e2e-ipv4-ebpf-worker2	2.00 ms	2.17 ms	2.50 ms	1.58 ms

Figure 4.15: *eBPF: Node to node Round-trip data points over time.*

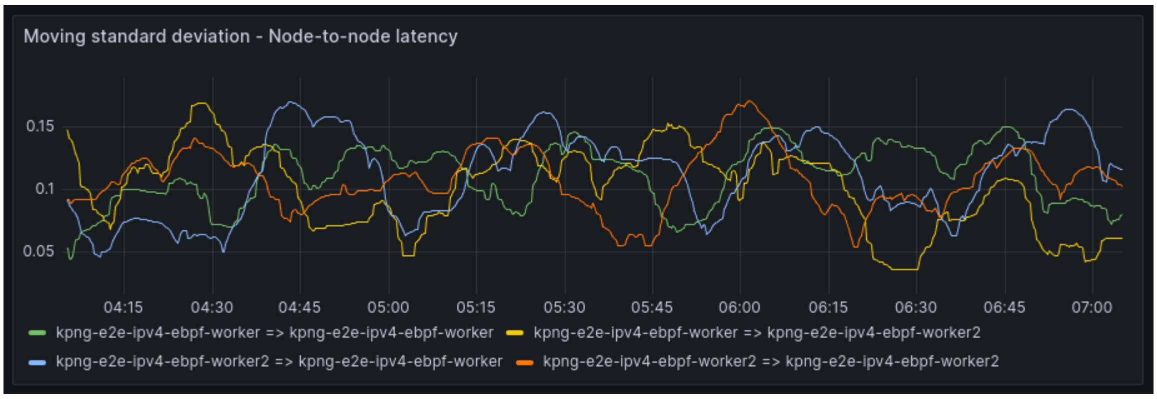


Figure 4.16: *eBPF: Node to node standard deviation over time.*

4.4 Analysis

By analyzing the graphs, both similarities and differences in the behavior of the backends can be observed. First, examining the similarities, it can be noted that both implementations achieved similar averages. The average total trip latency at a 99% confidence level for iptables is 1.28 ± 0.00463 ms, and for eBPF, it is 1.23 ± 0.00223 ms, indicating a modest 2.4% reduction in total latency with the eBPF solution, which is relatively small. The moving averages for both implementations exhibit similar patterns, with frequent ups and downs throughout the test duration.

The most notable difference between the two graphs is the higher dispersion of the iptables's graphs compared to the eBPF' graphs. The iptables total standard deviation was 0.311 for Round-trip and 0.315 for One-way-tip, while the corresponding values for the eBPF implementation were 0.155 and 0.172. These metrics align with the patterns observed in other graphs, revealing a larger gap between maximums and minimums and the average for iptables. This suggests that the eBPF solution might be more stable. However, it is important to point out that iptables has fewer data points than eBPF, which could potentially interfere with this metric.

Chapter 5

Conclusions and Future Work

The eBPF proof-of-concept is already able to match the battle-hardened iptables backend performance. However, it has a lot of room for improvement. For instance, the current implementation is hooked on `cgroup/connect4`. In theory, lower-level hooks such as XDP or TC would allow the system to make the routing decision much earlier in the network stack, potentially decreasing overall latency even further. Another argument is that XDP enables a technique called *offloading*, where XDP programs are loaded directly into the network interface card (NIC), executing without CPU.

The initial proposal of this project was to effectively implement XDP support into the KPNG eBPF backend. This preliminary and unfinished work can even be checked in a few branches listed below:

1. https://github.com/jaehnri/kpng/tree/xdp-clusterip: Exploring substitution of the `cgroup/connect4` hook with XDP in the ClusterIP implementation.
2. https://github.com/jaehnri/kpng/tree/xdp-nodeport: Initial implementation of the NodePort Service type in `XDP`.

However, this work was blocked due to a crucial XDP limitation: the missing support for outgoing packets. Given how the ClusterIP NAT is done by kube-proxy and KPNG, circumventing it was impossible. According to David Ahern, in the *XDP Newbies* Linux mailing list Ahern, 2020b, there is an ongoing effort to add TX path support into XDP, which would ultimately enable XDP to modify packets in the egress flow and further enable this work. However, to the date of writing of this project, this work was not available in official Kernel releases. This work can be tracked in the archives of the Linux Kernel Netdev community Ahern, 2020a.

Another alternative to the KPNG eBPF development would be to explore TC hooks. This has been somewhat studied, as seen on this TC NodePort proof-of-concept done primarily by Sanjeev Rampal in Rampal, 2022. It is important to note that this POC simulates Kubernetes and is not integrated into kube-proxy or KPNG, for instance. Implementing this in KPNG was thought out in this work, however, KPNG relies on Cilium's eBPF [1]

[1] Cilium's Golang eBPF library is available at https://github.com/cilium/ebpf

library to read, modify, and load eBPF programs and attach them to various hooks in the Linux kernel and, `cilium/ebpf` does not support and does not intend to support TC hooks as they are significantly more complex in both its design and its implementation Li, 2022. The effort to switch libraries would be out of the scope of the project and probably too complex.

A final opportunity for future work would be to do more exhaustive and load tests to the current eBPF implementation. This project was not funded by any organization, and thus, the authors have no resources to perform more scalability and/or load tests. It might be the case that eBPF outperforms iptables in scenarios with heavier loads, more NAT rules, or even more Pods behind a ClusterIP entry.

References

[Ahern 2020a] David Ahern. *[PATCH RFC v4 bpf-next 00/11] Add support for XDP in egress path.* 2020. url: https://lkml.kernel.org/netdev/20200227032013.12385-1-dsahern@kernel.org/ (visited on 10/23/2023) (cit. on p. 41).

[Ahern 2020b] David Ahern. *Re: Using AF_XDP To Modify Outgoing Packets.* 2020. url: https://www.spinics.net/lists/xdp-newbies/msg01629.html (visited on 10/23/2023) (cit. on p. 41).

[Goldberg 1974] Robert P. Goldberg. "Survey of virtual machine research". *Computer* 7.6 (1974), pp. 34–45. doi: 10.1109/MC.1974.6323581 (cit. on p. 4).

[Hansen 1972] P. B. Hansen. "Structured multiprogramming". *Commun. ACM* 15 (1972), pp. 574–578. doi: 10.1145/361454.361473 (cit. on p. 4).

[Høiland-Jørgensen *et al.* 2018] Toke Høiland-Jørgensen *et al.* "The express data path: fast programmable packet processing in the operating system kernel". In: *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies.* CoNEXT '18. Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66. isbn: 9781450360807. doi: 10.1145/3281411.3281443. url: https://doi.org/10.1145/3281411.3281443 (cit. on pp. 17, 18).

[Lattner and Adve 2004] Chris Lattner and Vikram Adve. "Llvm: a compilation framework for lifelong program analysis transformation." In: *International symposium on Code generation and optimization.* IEEE Computer Society, 2004 (cit. on pp. 1, 15).

[Li 2022] Vincent Li. *How can ebpf programs be attached to TC classes/qdiscs?* 2022. url: https://github.com/cilium/ebpf/discussions/769 (visited on 10/23/2023) (cit. on p. 42).

[McCanne and Jacobson 1993] Steven McCanne and Van Jacobson. "The bsd packet filter: a new architecture for user-level packet capture". In: *USENIX Winter, Vol. 93.* 1993 (cit. on pp. 1, 15).

[Raharjo *et al.* 2022] A. B. Raharjo *et al.* “Reliability evaluation of microservices and monolithic architectures”. *2022 International Conference on Computer Engineering, Network, and Intelligent Multimedia (CENIM)* (2022), pp. 1–7. doi: 10.1109/CENIM56801.2022.10037281 (cit. on p. 3).

[Rampal 2022] Sanjeev Rampal. *POC of K8s Nodeport service using BPF.* 2022. url: https://github.com/ebpf-networking/tc-nodeport (visited on 05/30/2023) (cit. on p. 41).

[The Linux Kernel documentation 2023] The Linux Kernel documentation. *eBPF Verifier.* 2023. url: https://www.kernel.org/doc/html/v6.1/bpf/verifier.html (visited on 10/20/2023) (cit. on p. 17).

[*The State of Enterprise Open Source: A Red Hat report* 2022] *The State of Enterprise Open Source: A Red Hat report.* Online. Red Hat, 2022. url: https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022 (cit. on p. 8).

[Zhang *et al.* 2018] Qi Zhang *et al.* “A comparative study of containers and virtual machines in big data environment”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* IEEE. 2018, pp. 178–185 (cit. on p. 4).