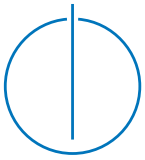
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Leveraging eBPF in Orchestrated Edge Infrastructures

Ben Julian Riegel



DEPARTMENT OF INFORMATICS

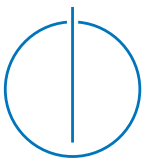
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Leveraging eBPF in Orchestrated Edge Infrastructures

Nutzung von eBPF in orchestrierten Edge-Infrastrukturen

Author:	Ben Julian Riegel
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Giovanni Bartolomeo, Dr. Nitinder Mohan
Submission Date:	30.09.2024



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 30.09.2024 Ben Julian Riegel

Acknowledgments

The journey—quite literally—that this thesis has taken me on has been an incredible experience, and I am deeply grateful to those who supported me along the way. Special thanks to my parents for their unwavering support, not only during this thesis but throughout my entire academic path.

Abstract

Due to the heterogeneous, resource-constrained, and distributed nature of edge deployments, there are several challenges associated with deploying virtual network functions (VNFs) such as firewalls, load balancers, or NATs on edge servers to enforce network policies.

Extended Berkeley Packet Filter (eBPF) is a Linux technology that allows code to be executed in the kernel. It combines the flexibility of user-space applications with the performance of kernel applications, making it particularly well-suited for writing programs that handle network packets with high efficiency. This thesis proposes a framework that enables the deployment of eBPF-based VNFs on edge servers within Oakestra, an edge orchestration framework.

As part of this thesis, an eBPF-based proxy was developed for Oakestra using the presented framework. The framework was evaluated through a case study, which demonstrated that the eBPF-based proxy significantly outperforms the original implementation in many network-relevant metrics. In a series of measurements, it was shown that throughput between two services deployed on a single Oakestra worker node increased by 8053.85%, while jitter and latency were reduced by 74.85% and 11.16%, respectively. Furthermore, when proxying 100 Mbit/s of TCP traffic between the two services, a 73.30% reduction in CPU utilization was observed.

Kurzfassung

Aufgrund der heterogenen, ressourcenbeschränkten und verteilten Natur von Edge-Deployments gibt es mehrere Herausforderungen beim Einsatz virtueller Netzwerkfunktionen (VNFs) wie beispielsweise Firewalls, Load Balancer oder NATs auf Edge-Servern zur Durchsetzung von Netzwerkrichtlinien.

Extended Berkeley Packet Filter (eBPF) ist eine Linux-Technologie, die es ermöglicht, Code direkt im Kernel auszuführen. Sie vereint die Flexibilität von User-Space-Anwendungen mit der Leistungsfähigkeit von Kernel-Anwendungen und eignet sich daher besonders gut für das Schreiben von Programmen, die Netzwerkpakete mit hoher Effizienz verarbeiten. Diese Arbeit präsentiert ein Framework, das die Bereitstellung von eBPF-basierten VNFs auf Edge-Servern innerhalb von Oakestra, einem Edge-Orchestrierungs-Framework, ermöglicht.

Im Rahmen dieser Arbeit wurde ein auf eBPF-basierter Proxy für Oakestra unter Verwendung des vorgestellten Frameworks entwickelt. Das Framework wurde durch eine Fallstudie evaluiert, die zeigte, dass der eBPF-basierte Proxy in vielen netzwerkrelevanten Metriken die ursprüngliche Implementierung deutlich übertrifft. In einer Messreihe wurde nachgewiesen, dass der Durchsatz zwischen zwei auf einer einzigen Oakestra-Worker-Node bereitgestellten Services um 8053,85% gesteigert wurde, während Jitter und Latenz um 74,85% bzw. 11,16% reduziert werden konnten. Darüber hinaus konnte bei der Übertragung von 100 Mbit/s TCP-Traffic zwischen den beiden Services eine Reduktion der CPU-Auslastung um 73,30% festgestellt werden.

Contents

Acronyms

CCA Congestion Control Algorithm

CDF Cumulative Distribution Function

CNI Container Network Interface

CP Control Plane

DP Data Plane

eBPF Extended Berkeley Packet Filter

EWMA Exponentially Weighted Moving Average

FD File Descriptor

MP Management Plane

NAT Network Address Translation

NFR Non-Functional Requirement

NIC Network Interface Card

QDISC Queuing Discipline

QoS Quality of Service

RTT Round-Trip Time

SDN Software-Defined Networking

SLA Service-Level Agreement

TC Traffic Control

VM Virtual Machine

VNF Virtual Network Function

XDP eXpress Data Path

1. Introduction

Edge computing is rapidly emerging as a critical component in the architecture of modern distributed systems, particularly as the proliferation of Internet of Things (IoT) devices, real-time applications, and data-intensive workloads continue to grow[13]. Unlike traditional cloud computing, where data processing and storage are centralized in large data centers, edge computing brings these capabilities closer to the data source. This shift is driven by the need to reduce latency, optimize bandwidth usage, enhance data privacy, and improve the overall reliability of distributed systems. In scenarios such as autonomous vehicles or smart cities[15], the ability to process data locally—at or near the point of generation—is crucial for achieving real-time responsiveness and ensuring operational efficiency.

Despite the advantages of edge computing, its deployment presents significant challenges. The decentralized nature of edge environments, with potentially thousands of heterogeneous devices spread across vast geographic areas, creates a complex management landscape. This complexity is exacerbated by the need to dynamically allocate resources, manage diverse workloads, and ensure seamless integration with centralized cloud systems. Without effective orchestration frameworks, managing these distributed edge resources becomes impractical, leading to inefficiencies, increased operational costs, and potential security vulnerabilities.

Oakestra[1] is an edge computing orchestration framework that addresses these challenges by providing a comprehensive solution for automating the deployment, management, and scaling of applications across distributed edge nodes. Oakestra is designed to optimize resource allocation, ensuring that computing power, storage, and networking capabilities are efficiently utilized across the edge.

1.1. Motivation

Deploying software in traditional cloud environments, using frameworks like Kubernetes[19], typically allows users to define network policies within service level agreements (SLAs), which are then enforced by the cloud provider's data centers. To fulfill these SLAs, modern data centers usually deploy Virtual Network Functions (VNFs) alongside services on cloud servers.

However, deploying VNFs in edge environments presents significant challenges. Unlike the powerful servers found in centralized data centers, edge servers typically have limited CPU power, memory, and network bandwidth. Although frameworks like DPDK[7] can be used to efficiently implement VNFs, they often require exclusive access to hardware, which is not feasible in resource-constrained edge environments.

Oakestra currently enforces network policies using Linux tools like *iptables*[38] and implements more complex network functions in user-space, such as its proxy. However, *iptables* evolves slowly and is challenging to customize, while user-space implementations, though flexible, suffer from considerable performance overhead due to data copying and context switching between kernel and user-space.

Extended Berkeley Packet Filter (eBPF) provides a promising solution by enabling the execution of small, user-space-like programs within the kernel without requiring kernel modifications. This capability positions eBPF as an ideal technology for deploying custom network functions within the kernel, blending the flexibility of user-space programs with the performance advantages of kernel-level execution. This thesis seeks to explore the integration and optimization of eBPF within Oakestra, particularly in edge computing contexts, addressing the following research questions:

1. Is it possible to leverage eBPF in Oakestra to effectively deploy VNFs in edge environments?
2. How can existing network functions, such as Oakestra's proxy, be optimized using eBPF to enhance performance?
3. What measurable performance advantages does the implementation of eBPF network functions provide for Oakestra in edge environments?

1.2. Contribution

This thesis introduces the eBPF Manager for Oakestra's worker nodes. The eBPF Manager operates on these worker nodes (edge servers), facilitating the deployment of custom VNFs written in eBPF. A key feature of this system is the ability to update or modify these VNFs at run-time, without requiring changes to Oakestra's underlying implementation. This approach significantly enhances Oakestra's flexibility, allowing for the efficient implementation of even highly specialized use cases in edge deployments.

Later in this thesis, it is demonstrated that reimplementing Oakestra's proxy using eBPF and the proposed eBPF Manager results in improvements across multiple network metrics, including a throughput increase of over 8000%. This substantial performance enhancement highlights the significant potential of eBPF in optimizing network functions within edge environments.

1.3. Outline

To systematically address the research questions posed in Section 1.1, this thesis begins by establishing the necessary theoretical foundation in Chapter 2. Since the proposed approach closely aligns with the philosophy of VNFs in Software-Defined Networking (SDN), the general structure of a VNF is covered in Section 2.1. Following this, Section 2.2 provides a comprehensive introduction to the eBPF technology. Then, Section 2.3 introduces the architecture of Oakestra, with a particular focus on Worker Nodes and the *NetManager* running on them. Finally, Section 2.4 further motivates the problem statement of the thesis and presents three related works relevant to this research.

In Chapter 3, the overall system design of the proposed eBPF framework is introduced. First, the Non-Functional Requirements (NFRs) for the system are established in Section 3.1, from which the final approach is subsequently derived in Section 3.2. Section 3.3 covers the critical design decisions made, while Section 3.4 provides detailed documentation of the API for the proposed eBPF framework. Section 3.5 showcases three examples of eBPF-based VNFs that can be deployed onto Oakestra worker nodes using the proposed framework, including an eBPF-based implementation of the Oakestra Proxy.

Chapter 4 evaluates the eBPF framework using the eBPF-based Proxy. The evaluation examines various metrics, including latency in Section 4.2, jitter in Section 4.3, throughput in Section 4.4, and resource consumption in Section 4.5.

In the concluding Chapter 5, Section 5.1 summarizes the thesis findings and addresses the research questions introduced in the beginning. Lastly, Section 5.2 explores potential future work and summarizes the key limitations of the implementations developed in this thesis.

2. Background & Related Work

2.1. Virtual Network Function

A network function refers to a specific operation or task performed within a network, typically related to managing, controlling, and forwarding network traffic. These functions include routing, firewalling, load balancing, Network Address Translation (NAT), and more.

In Software-Defined Networking (SDN)[18], network functions are abstracted from the underlying hardware and implemented in software, allowing for centralized control and management. This enhances flexibility because the network function is no longer connected to the network as a physical middlebox. Instead, it operates as a software component that can be easily updated or scaled and often runs in a virtual machine alongside the actual application. These virtualized network functions are called Virtual Network Functions (VNFs) and typically follow a structure where data plane, control plane and management plane are decoupled from each other[18]. Figure 2.1 depicts this structure, while Sections 2.1.2, 2.1.3, and 2.1.1 outline the responsibilities of each plane.

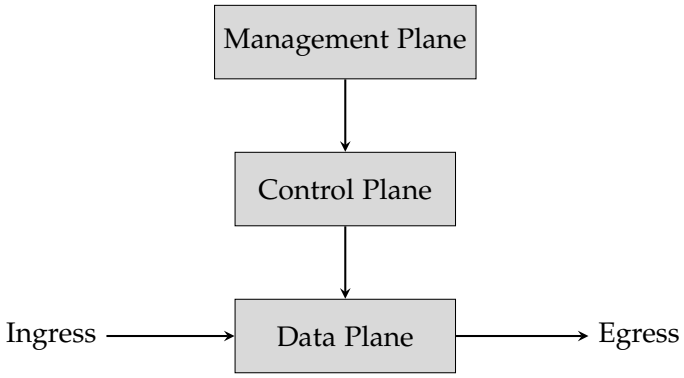


Figure 2.1.: Common Structure of a VNF in SDN

2.1.1. Management Plane

The Management Plane (MP) of a Virtual Network Function (VNF) is responsible for the overall lifecycle management and orchestration of the VNF. This includes tasks such as deployment, configuration, monitoring, scaling, and updating the VNF. The management plane ensures that the VNF operates efficiently within the network environment and adheres to the desired performance and security standards. It interacts with the control plane to implement policies and with the data plane to monitor traffic, enabling dynamic adjustments to the VNF based on real-time conditions and network demands.

2.1.2. Control Plane

The Control Plane (CP) of a VNF is responsible for managing and configuring the network function. It handles the decision-making processes, such as setting up routing tables, managing session states, and enforcing policies. The control plane communicates with other network components to ensure that the VNF operates according to the desired network policies and configurations. It also interacts with the data plane, which handles the actual forwarding of network traffic based on the rules and instructions set by the control plane.

2.1.3. Data Plane

The Data Plane (DP) of a VNF is responsible for the actual processing and forwarding of network traffic. It handles the movement of data packets through the network according to the rules and policies set by the control plane. This includes tasks like packet forwarding, filtering, encryption, and load balancing. The data plane operates at high speed to ensure efficient packet processing, making it crucial for maintaining the performance and reliability of the VNF in a network environment.

2.2. Extended Berkeley Packet Filter (eBPF)

Extended Berkeley Packet Filter (eBPF)[9][29] is a technology that allows for the execution of custom programs within the Linux kernel. Originally conceived as a tool for network packet filtering, eBPF has evolved into a framework used for a wide range of applications, including networking, security, performance monitoring, and more. The main advantages of eBPF are its ability to run safely and efficiently within the kernel, without requiring kernel modifications or restarts.

At its core, eBPF functions by injecting small programs into various kernel hooks and running them within an in-kernel virtual machine. These programs are typically written in a restricted subset of C and compiled into eBPF bytecode. Alternatively, they can be written in a language called eBPF assembly. eBPF programs can be Just-In-Time compiled by the kernel for extra performance[27].

For security reasons, before any eBPF code is injected into the kernel, it is verified to ensure secure execution. More details on the verifier[11] are provided in Section 2.2.1.

eBPF programs can call other eBPF programs during their execution through a mechanism known as tail calls[28], which will be explored in more detail in Section 2.2.2.

eBPF programs can be attached to a variety of kernel hooks, including network events, system calls, tracepoints, and more. When the associated event occurs, the eBPF program is triggered, allowing it to inspect and manipulate data, collect metrics, or take specific actions based on its logic.

eBPF programs are typically limited to a stack size of 512 bytes, which significantly restricts the amount of data that can be stored in variables. Additionally, eBPF programs cannot dynamically allocate arbitrary amounts of memory inside the kernel; instead, they must use designated data structures known as eBPF maps[10]. Further details about these data structures will be discussed in Section 2.2.3.

In the context of networking, the execution of an eBPF program is triggered upon the arrival of a packet. Within the networking stack, the eXpress Data Path (XDP) and the Traffic Control (TC) hooks are of particular importance for this thesis. Both provide mechanisms for processing network packets, but they differ in terms of where and how they operate within the Linux networking kernel stack. Consequently, Sections 2.2.4 and 2.2.5 will provide a detailed examination of these hooks. Figure 2.2 illustrates the position of these hook point within the kernel stack.

2.2.1. eBPF Verifier

One of the critical components of the eBPF ecosystem is the in-kernel eBPF verifier[11]. Before an eBPF program is loaded into the kernel, it must pass the verifier. Given the kernel's critical nature and the potential impact of unsafe code, the verifier is essential for maintaining system stability and security.

The verifier checks the eBPF bytecode for any illegal instructions or unsafe operations. This includes ensuring that all instructions are valid eBPF instructions and that they adhere to the constraints imposed by the eBPF architecture.

Also, the verifier ensures that the eBPF program will terminate. It does this by analyzing loops and ensuring they have a bounded number of iterations.

The verifier checks all memory accesses to ensure they are within bounds. This

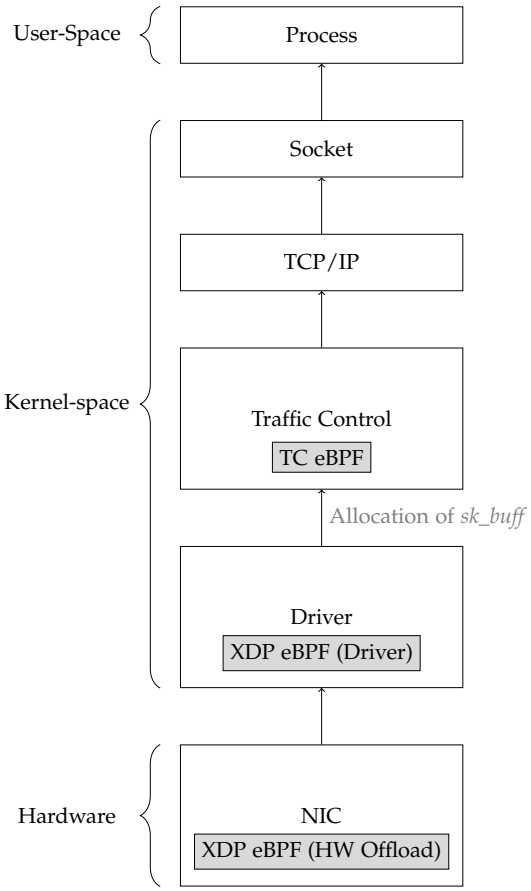


Figure 2.2.: XDP and TC hooks in the Linux networking stack

prevents the program from accessing invalid memory areas, which could lead to crashes or security vulnerabilities.

The verifier imposes limits on the resources that an eBPF program can use. This includes limits on stack usage (typically 512 bytes), the number of instructions, and the size of maps that the program can access. These limits prevent eBPF programs from consuming excessive resources and impacting system performance.

2.2.2. eBPF Tail Calls

eBPF tail calls[28] are a powerful mechanism that enables an eBPF program to transfer control directly to another eBPF program without returning to the original caller, allowing for the effective chaining of multiple programs.

This technique is particularly useful when there is a need to execute multiple eBPF

programs in sequence, such as applying a series of packet processing steps on a network interface. Tail calls help to overcome the limitations of eBPF program size and complexity by distributing the logic across multiple programs, each handling a specific task.

When a tail call is invoked, the current program is terminated, and control is transferred to the target program, with the stack being reset. This reset ensures that each eBPF program operates within its own context, but it also means that any local data or variables from the original program are lost.

There is a limit to the number of tail calls that can be made in a single execution path, typically up to 33, which is designed to prevent infinite loops and ensure the stability of the system.

2.2.3. eBPF Maps

eBPF maps[10] act as key-value data stores, allowing eBPF programs to persist data across program calls and also exchange data with user-space applications.

The following types of maps are available for eBPF programs on kernel version 5.8 or later:

- **Hash Maps:** Generic key-value stores that allow arbitrary key-value pairs.
- **Array Maps:** Sequential integer-indexed key-value stores.
- **Per-CPU Hash/Array Maps:** Per-CPU instances of hash or array maps to reduce contention.
- **LRU (Least Recently Used) Hash Maps:** Hash maps with automatic eviction of the least recently used entries.
- **Queue and Stack Maps:** Maps implementing queue (FIFO) and stack (LIFO) data structures.
- **BPF Program Array Maps:** Maps holding references to other eBPF programs for chaining or tail-calling.
- **Ring Buffer Maps:** High-performance ring buffer maps for data transfer between kernel and user-space.
- **Socket Maps:** Maps that allow redirection of packets to specific sockets, bypassing the normal kernel network stack processing.

2.2.4. eXpress Data Path (XDP) eBPF

XDP is an eBPF-based data path designed for high-performance packet processing as early in the Linux networking stack as possible. It can run eBPF programs in three different modes: Driver mode, Generic mode and Hardware Mode.

In Driver mode, the eBPF program is executed at the driver level immediately after a packet is copied from the Network Interface Card (NIC) into the driver. This mode requires driver support and offers the advantage that minimal processing has been done to the packet at this stage. The packet has not yet been copied into the kernel, and no *sk_buff* (the standard Linux data structure for network packets) has been initialized, allowing for efficient packet handling and analysis.

Generic mode functions independently of network driver support, executing the eBPF program higher in the networking stack after the *sk_buff* has already been allocated. This mode is typically only utilized for testing purposes and does not take advantage of the performance capabilities of XDP eBPF programs.

In Hardware mode, the kernel attempts to offload the eBPF program to the NIC. This process requires both the NIC and its driver to support such offloading. This mode may provide a reduced set of features, as the hardware may not support certain eBPF functionalities. Also, this mode is currently not supported by many devices. The main advantage of Hardware mode is that program execution requires no CPU cycles, offloading the processing entirely to the network hardware.

As of now, XDP can only be attached to the ingress path of a network interface. Furthermore, only one eBPF program can be attached to an interface at a time. If multiple eBPF programs need to be executed in sequence, tail calls must be used.

For an arriving packet, the following actions can be taken by an eBPF program:

- **XDP_PASS**: The packet continues traveling up the networking stack.
- **XDP_DROP**: Drops the packet.
- **XDP_ABORTED**: Drops the packet with a tracepoint exception.
- **XDP_TX**: Bounces the packet back to the same NIC it arrived on.
- **XDP_REDIRECT**: Redirects the packet to another NIC.

2.2.5. Traffic Control (TC) eBPF

In contrast to XDP (see Section 2.2.4) TC can function without eBPF. The TC subsystem[23] has been part of the Linux networking stack long before eBPF was introduced and provides a wide range of traffic shaping, scheduling, policing, and classifying

capabilities using traditional mechanisms like Queuing Discipline (QDISC), classes, and filters.

QDISCs are mechanisms that control how packets are enqueued and dequeued in the network interface queues. Users can specify QDISCs for both ingress and egress traffic, and they can be nested to create complex configurations. QDISCs are typically used to implement Quality of Service (QoS) policies and optimize traffic throughput within a network. When a packet is received or sent, the kernel first places it into the configured QDISC for the interface. The QDISC then determines the order and manner in which packets are dequeued, ensuring efficient and prioritized handling of network traffic.

The integration of eBPF into the TC subsystem allows for the development of custom classifiers and actions in eBPF, which can then be loaded and attached to ingress and egress QDISCs. This allows the user to implement their very specific use cases efficiently, as their classifier/action is not bloated with features they might not need, thus consuming unnecessary resources.[37]

Another advantage of the TC hook is that, unlike XDP, it allows multiple eBPF programs to be attached to a single interface. These eBPF programs are treated just like regular classifiers and filters within QDISCs, with their order of execution determined by the configuration and type of the QDISC. The simplest example would be a FIFO QDISC, where the first packet to enter the queue is also the first to exit.

TC eBPF programs are executed slightly further up the stack compared to XDP programs, as illustrated in Figure 2.2. As a result, they have access to the *sk_buff* allocated by the kernel and additional metadata. However, this comes at the cost of an additional copy of the packet, which typically makes TC programs slightly less performant than XDP programs.

For a packet, an eBPF program attached to TC can perform the following actions:

- **TC_ACT_OK**: The packet continues traveling up (or down) the networking stack.
- **TC_ACT_SHOT**: The packet is dropped.
- **TC_ACT_RECLASSIFY**: The packet is reclassified and processed again from the beginning of the classification chain.
- **TC_ACT_REDIRECT**: Redirects the packet to another network interface.
- **TC_ACT_PIPE**: The packet continues to the next action in the chain without altering the packet.
- **TC_ACT_STOLEN**: The packet is consumed by the eBPF program and will not be processed further by the kernel networking stack.

2.3. Oakestra

Oakestra[1] is a lightweight, hierarchical orchestration framework specifically designed for edge computing environments. Unlike traditional orchestration platforms like Kubernetes[19], which are optimized for homogeneous, high-bandwidth cloud environments, Oakestra aims to efficiently manage resources in heterogeneous and dynamic edge settings. This section presents an overview of Oakestra and its architecture, with a focus on the components relevant to this thesis.

2.3.1. Architecture

Oakestra is designed with a hierarchical structure to distribute the entire platform across multiple clusters, enabling these clusters to operate more independently from one another. This approach is particularly beneficial in edge environments, where connections between different clusters are generally less reliable than in cloud environments. This hierarchical architecture is composed of three main components: the root orchestrator, the cluster orchestrators, and the worker nodes. Figure 2.3 illustrates the hierarchical structure of Oakestra with N clusters and N worker nodes in each cluster.

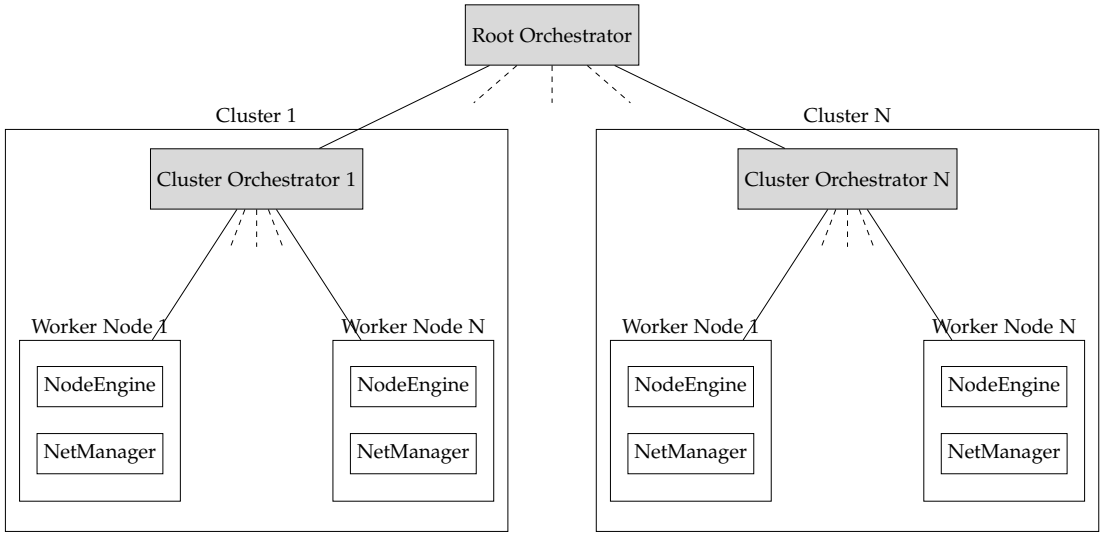


Figure 2.3.: Oakestra Hierarchical Architecture

Root Orchestrator

The root orchestrator functions as the centralized control plane, similar to the control plane in Kubernetes. It coordinates the underlying clusters by communicating with their respective cluster managers. The root orchestrator is responsible for scheduling deployment requests submitted by developers through Service-Level Agreements (SLAs).

Cluster Orchestrator

In Oakestra, a cluster refers to a group of nodes that are collectively managed by a single entity: the cluster orchestrator. The cluster orchestrator provides the root orchestrator with utilization statistics for its cluster and receives deployment requests that align with the available resources.

Worker Node

The worker nodes are the edge servers inside a cluster that actually deploy and run the services. They report their available resources (RAM, CPU, etc.) and status to the cluster orchestrator, which then assigns deployment requests to appropriate nodes based on SLA requirements.

Each worker node runs two native applications: the *NetManager* and the *NodeEngine*. Although a worker node can theoretically function without the *NetManager*, this would prevent deployed services on these nodes from communicating with other services, which drastically limits the benefits of an orchestration framework. Therefore, for the purposes of this thesis, we will assume that every worker node consistently operates both the *NodeEngine* and the *NetManager*.

The *NodeEngine* is tasked with setting up the execution environment and managing the deployed service within it. Currently, Oakestra's SLAs enable developers to run their software in containers, with the recent addition of support for unikernels[35].

Once the *NodeEngine* has set up the execution environment, it instructs the *NetManager* to connect the service to the Oakestra network. The *NetManager* ensures that the service can communicate with other services on the same node, as well as with services on other worker nodes. Additionally, if the firewall rules permit, each service can access IP addresses on the open internet.

How this inter-service addressing in Oakestra works from a functional perspective is explained in Section 2.3.2. The technical perspective detailing how the *NetManager* implements this addressing schema is discussed in Section 2.3.3.

2.3.2. Semantic Addressing

Oakestra[1] employs a semantic addressing scheme, which is thoroughly detailed in its networking documentation[26]. This chapter provides a brief summary of the information relevant to this thesis.

Oakestra assigns so-called service IP addresses to each service. Since multiple instances of a service can be deployed, a Service IP references a group of instances. There also exists a so-called instance IP, which directly represents one specific instance of a service. These virtual IP addresses are taken from the `10.30.0.0/16` IP range and allow for inter-service communication within Oakestra. Service IPs are similar to ClusterIPs in Kubernetes; however, a unique aspect of Oakestra is that each service is assigned as many Service IPs as the platform supports balancing policies. Each service IP represents one balancing policy.

In the case where one instance communicates to another service by using one of its service IPs, an instance is selected based on the balancing policy represented by the chosen service IP. It is important to note that once a session between two instances is established, these can be TCP or UDP, Oakestra keeps these two communication partners consistent and does not balance each packet to a new instance.

By targeting an instance IP directly instead of a service IP, one can reach that specific instance directly, bypassing any balancing policies.

Currently, Oakestra implements only round-robin as a balancing policy, which evenly distributes traffic among all instances of a service. However, the authors also envision implementing a "closest node method", where the geographically nearest instance is selected as the target.

2.3.3. NetManager

On each worker node, the *NetManager* creates a Linux network namespace[22] for every service on that node. Each service runs inside a network namespace to encapsulate it from the host machine.

The *NetManager* then connects all namespaces to a bridge using pairs of virtual Ethernet ports, creating a virtual network with all namespaces. Each namespace is assigned a unique IP address by the *NetManager*, which are referred to as namespace IPs. Additionally, a proxy is connected to the bridge via a TUN device, which can tunnel packets to other worker nodes through a UDP tunnel. This setup is graphically illustrated in Figure 2.4.

One might wonder why instance IPs are needed if namespace IPs already uniquely reference instances. Instance IPs are necessary because an instance might be re-deployed (to a different worker node) at any time, leading to a change in its namespace IP, while

the service and instance IPs remain consistent across redeployments.

UDP Tunnel

All proxies on the worker nodes can exchange packets destined for instances on other worker nodes. These packets are wrapped in UDP packets and tunneled to the corresponding worker node, as the virtual IP addresses assigned by the *NetManager* are not routable on the open internet.

The tunnel is effectively transparent to the proxy, making it seem as though all deployed instances are running on a single worker node connected to a large network via a bridge.

Proxy

The Proxy is responsible for implementing the addressing scheme described in Section 2.3.2. When the *NetManager* is initialized, it inserts *iptables*[38] rules that forward all inter-service traffic (destination IPs from the `10.30.0.0/16` range) through the proxy.

When the proxy receives a packet whose destination IP is a service IP, a specific instance is selected to which the packet should be sent. There are two scenarios to consider: If the packet is the first in a session, such as the TCP-SYN packet of the three-way handshake, one of the available instances is selected using the balancing policy determined by the service IP. Newly established sessions are then cached by the proxy for subsequent packets. If the packet is part of an ongoing session, the same target instance is selected again using the cache entry.

Once the proxy has selected an instance, the destination IP address is translated into the current namespace IP of that instance. Additionally, the source IP is translated into the instance IP of the sender, allowing the receiver to identify which instance it is communicating with. The proxy then passes the translated packet to the tunnel component. At this point, two scenarios are possible: if the target instance is deployed on the same worker node, the tunnel immediately returns the packet to the proxy; if the target instance is on a different worker node, the packet is tunneled to the appropriate worker node and handed over to the proxy there. This is what was previously described as the transparency of the tunnel with respect to the proxy. The proxy simply receives packets from the tunnel component without knowing whether they originate from its own worker node or have been tunneled through the internet.

The packets handed over by the tunnel are now forwarded to the bridge, as it is ensured that the namespace in which the target service is running must be deployed on the current worker node.

Lookup Table

The proxy maintains a lookup table component that allows it to translate between service, instance, and namespace IPs. The proxy can obtain information about deployed services and their instances from its Cluster Manager through so-called Table Queries. Further implementation details are beyond the scope of this thesis, and it will be assumed that this lookup table is available and always holds the requested information.

Example

Figure 2.4 illustrates the setup described above using the example of a worker node with two deployed instances: Instance 1 of Service A (`A.1`) and Instance 1 of Service B (`B.1`). The proxy will be explained in more detail at the example of a packet that is to be sent from `A.1` to Service B. The proxy's cache is still empty, meaning there are no existing sessions between any instances.

In step 1↑, `A.1` first sends a packet to the round-robin Service IP of Service B. This packet is forwarded by the bridge in step 2↑, as the destination IP address falls within the `10.30.0.0/16` range.

Next, the proxy performs a Table Query in step ←3 to look up the possible translations for the Service IP. Since `B.1` is the only deployed instance of Service B, the response in step 4→ contains only this instance. Using the round-robin method, the proxy selects a target instance. In this trivial case, it selects the only deployed instance, `B.1`.

The proxy then replaces the packet's destination address with the namespace IP of `B.1`. It also replaces the source address with the instance IP of `A.1`, in case `A.1` has been redeployed and received a new namespace IP by the time a potential response arrives. Additionally, the newly established session between `A.1` and `B.1` is added to the proxy's cache.

If `B.1` would be located on a different worker node, the translated packet would be tunneled to the corresponding node in steps 4.1→ and ←4.2. In steps ↓5 and ↓6, the packet is forwarded to the target namespace with `B.1` running inside.

2.4. Related Work

Building on the background knowledge provided in the previous sections, this section will highlight the issues with Oakestra's current Proxy implementation, which served as the initial motivation for this thesis. It will then review related works and existing solutions to identify how the approach proposed in this thesis can benefit from them.

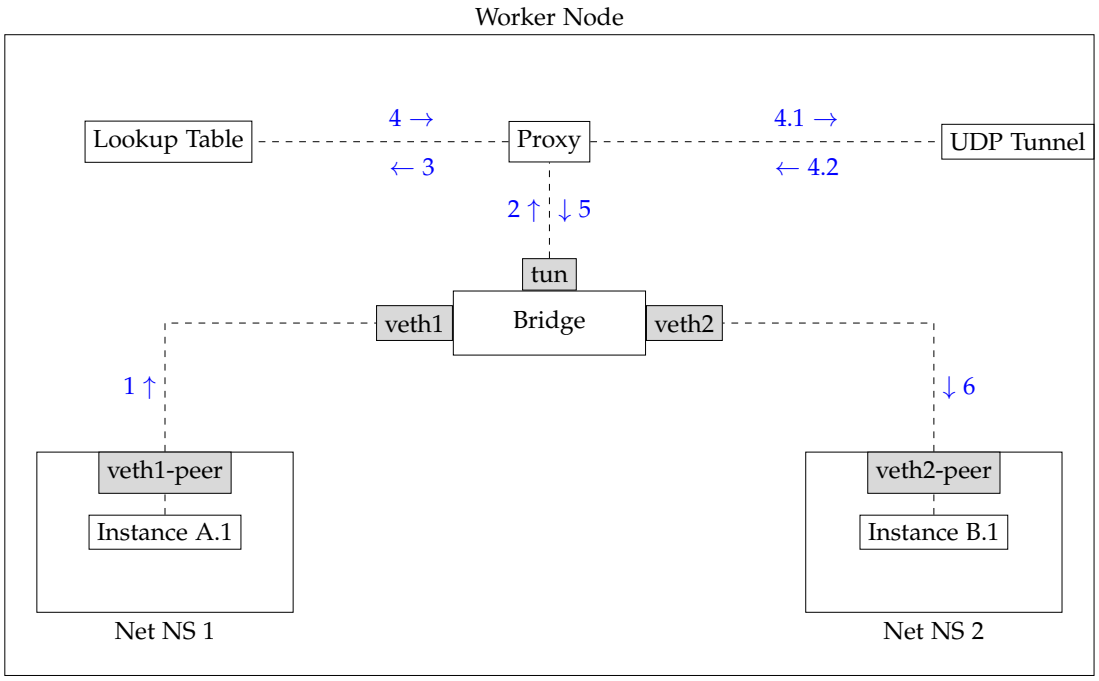


Figure 2.4.: Oakestra NetManager: Example of Proxy-Based Service IP Translation

To fully grasp the core motivation behind this thesis, it is crucial to emphasize once again that the *NetManager* operates as a user-space application, meaning both the proxy and tunnel also run in user-space. As a result, every packet passing through them must be copied to user-space for processing and then copied back to kernel-space for forwarding. This process consumes CPU cycles, which adversely affects the proxy's overall resource consumption, latency, and throughput, as demonstrated later in the thesis.

The counterpart to Oakestra's proxy in Kubernetes is called kube-proxy[8], which supports multiple modes, including a user-space mode and an *iptables* mode. The user-space mode works similarly to the Oakestra proxy and shares the same disadvantages mentioned above, which is why the *iptables* mode has become the default for Linux. In this mode, the proxying entirely happens in the kernel by inserting rules into *iptables*, making kube-proxy more of an *iptables* configurator than the actual proxy.

A similar approach could be taken for Oakestra, but the challenge lies in the fact that Oakestra's edge-specific requirements demand a high degree of flexibility from the proxy. For example, implementing a "closest node" load balancing policy based exclusively on *iptables* rules is difficult.

This is exactly where eBPF can show its full potential. Since eBPF programs can

be developed and deployed almost like user-space programs, they offer a promising foundation for implementing the DP of the proxy within the kernel while still retaining the flexibility needed to meet the requirements of Oakestra and edge environments.

This idea was further extended to not just rewrite the Oakestra proxy or tunnel in eBPF, but to develop an eBPF framework that enables the deployment of arbitrary eBPF-based VNFs in Oakestra. This would allow existing functions such as the proxy, tunnel, or firewall to be replaced with eBPF-based equivalents or even entirely new VNFs to be developed and deployed.

2.4.1. Cilium

Cilium[4] is an open-source project recognized for introducing eBPF-based networking solutions to cloud-native environments, particularly Kubernetes. While Cilium can operate outside of Kubernetes and even without an orchestration framework in some cases, its feature set and the community's interest are closely aligned with Kubernetes. Cilium can be integrated into Kubernetes' networking layer as a Container Network Interface (CNI)[6] plugin, allowing it to either extend specific networking functionalities or fully replace them.

Cilium was developed in response to the rise of microservices architectures, which have greatly increased the complexity and scale of modern networks. As microservices proliferate, the demands on networking infrastructure grow, necessitating more efficient and flexible tools. Traditional networking tools like *iptables*[38], though reliable, have limitations due to their slow evolution and the challenges of adapting them to rapidly changing network environments. eBPF emerged as a solution to these challenges because it allows for the execution of custom code directly in the Linux kernel, providing the advantage of developing and updating networking solutions with the flexibility of a user-space application. Cilium, therefore, was driven by motivations similar to those of this thesis, though within the cloud context rather than the edge.

Cilium's primary use cases encompass networking, observability, and security, each of which is addressed through individual eBPF-based solutions. For example, Cilium offers an L4 load balancer implementation and a replacement for Kubernetes' kube-proxy, demonstrating the feasibility of implementing the Oakestra Proxy using eBPF. Additionally, Cilium enhances network logging and enables traffic encryption between Kubernetes nodes.

Although Cilium's specific implementations are not directly applicable to this thesis—primarily due to their cloud-centric design and strong focus on Kubernetes—it is one of the largest projects implementing eBPF-based networking, making Cilium a valuable resource for this thesis. Cilium has released a comprehensive *BPF and XDP Reference Guide*[29] as well as a Go package[3] that simplifies tasks such as loading,

compiling, and debugging eBPF programs with minimal external dependencies. Since the Oakestra *Netmanager* is also written in Go, this package forms the foundation for the proposed eBPF framework.

2.4.2. Polycube

While Cilium can be seen as a collection of separate implementations, each serving a specific use case, Polycube[24][25] is more of a general framework aimed at deploying arbitrary eBPF-based VNFs in cloud environments. It provides implementations for VNFs like bridges, routers, NAT, load balancers, firewalls, DDoS mitigators, and more. Polycube also offers an interface for the user to create their own VNFs and integrate them into the Polycube ecosystem. These so-called "cubes" can be arbitrarily chained and interconnected to form complex network topologies. Each cube functions as an independent plugin that can be dynamically installed and loaded at run-time. Users have the flexibility to combine different cubes to create the precise network functionality they require.

A cube is composed of a DP implemented in eBPF that operates in kernel-space, and a CP that runs in user-space. A cube can also have a slow path, where packets can be sent to the CP if the processing requires user-space knowledge or if a specific feature cannot be implemented in eBPF, such as when an unbounded loop is needed for the processing logic.

Cubes are categorized into two main types: standard cubes and transparent cubes. A standard cube defines multiple ports through which it can receive or send traffic, making forwarding decisions between these ports based on its control plane logic. The port of a standard cube can either be connected to a Linux port (e.g. virtual Ethernet ports) or to another standard cube port, allowing traffic to flow from one cube to the next.

In contrast, transparent cubes are designed for network functions that do not require dedicated forwarding decisions between multiple ports. These cubes do not define their own ports but are instead attached and chained to existing ports. Typical examples of such VNFs include firewalls or DDoS mitigation functions, which only need to decide whether to drop a packet. Additionally, transparent cubes can be used for traffic analysis purposes, reducing the overhead associated with standard cubes.

Figure 2.5 shows an example topology to give an idea of what can be built with Polycube. The gray nodes represent standard cubes, while the white nodes represent transparent cubes that have been attached to the ports of the standard cubes. In orchestration frameworks, *eth0* may serve as the interface connecting a worker node to the internet, while the virtual Ethernet ports interface with containers running the orchestrated services within the worker nodes.

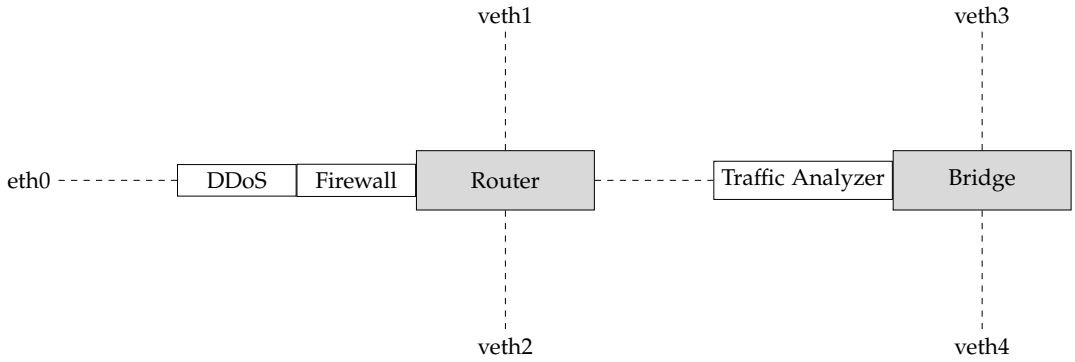


Figure 2.5.: Polycube Example Topology with Standard and Transparent Cubes

One of Polycube's most interesting features is its ability to abstract the hook points where eBPF programs are attached in the kernel. A cube's DP logic can be written independently of the specific hook point, allowing users to decide at run-time where in the network stack the cube should be attached. This is accomplished by offering hook point-independent eBPF helper functions and wrapping the eBPF programs in logic that abstracts the context before execution and restores it afterward.

The core logic of Polycube runs in a daemon that can be controlled via a REST API. Polycube comes with its own CLI that translates console commands, such as creating or connecting cubes, into API calls to the daemon.

2.4.3. NetEdit

Recently, Meta published a paper about their experiences developing NetEdit[2], an orchestration framework for eBPF-based VNFs that was running in production for the past five years. Although this paper was published after the main development phase of this thesis, it offers valuable insights into how eBPF can be utilized on a large scale, as demonstrated by Meta.

In summary, they were able to significantly enhance their network performance while measuring a very low resource footprint of the eBPF programs in the kernel. By comparing environments with and without *NetEdit* enabled, they demonstrated, among other things, that their eBPF framework was responsible for a 4.5X reduction in retransmissions and a 10% increase in throughput. As with all performance measurements in this context, these results are highly dependent on the hardware and specific circumstances in which eBPF was used, but they nonetheless reflect a trend similar to the findings of this thesis.

3. System Design

This chapter aims to design the architecture for a framework[34] that enables Virtual Network Functions (VNFs) written in Extended Berkeley Packet Filter (eBPF) to be dynamically deployed onto Oakestra worker nodes.

To achieve this, Section 3.1 first analyzes the Non-Functional Requirements (NFRs) for this framework, from which a general approach is derived in Section 3.2. Following that, the key design decisions required to implement this approach are discussed in Section 3.3. After a description of the framework's API in Section 3.4, the final Section 3.5 presents three concrete implementation examples of VNFs for the framework: a packet counter[31], a firewall[30], and a proxy[32].

3.1. Non-functional Requirements

Based on the overall goal and the environment Oakestra was developed for, the following NFRs can be derived for the eBPF framework:

- **Compatibility**: An edge environment is typically heterogeneous, with various devices from different manufacturers, each with different resources and architectures. It is important to consider that some nodes may not be fully or partially compatible with eBPF (e.g., due to different kernel versions). Therefore, the framework must operate alongside the existing implementation.
- **Modularity**: Someone who wishes to inject eBPF into their Oakestra nodes must not necessarily be an Oakestra maintainer. Therefore, deploying new VNFs should be possible without changing the underlying *NetManager* or any other Oakestra component. The eBPF framework should function like a plugin system, allowing eBPF code to be injected at run-time.
- **Efficiency**: In an edge environment, resource conservation is typically more crucial than in a cloud environment. While a performance boost that requires increased resource usage may be advantageous, it is not optimal. The ideal scenario is to achieve improved performance while maintaining or reducing resource consumption, thereby increasing efficiency.

- **Scalability**: The framework must scale both with the number of instances on a worker node and with the overall number of worker nodes in an Oakestra environment.

3.2. Approach

Based on the NFRs (Section 3.1), we can now derive the final approach for our eBPF framework[34]. The core concept involves introducing a new component to the *NetManager*, referred to as the eBPF Manager (see Section 3.2.2). This component is responsible for managing the lifecycles of the eBPF Plugins, referred to as eBPF Modules (see Section 3.2.1).

Each eBPF module comes with a Control Plane (CP) and a Data Plane (DP) written in eBPF. When multiple eBPF Modules are deployed on a single worker node, their DPs are chained up at each service. The eBPF Manager and the deployed eBPF Modules can be configured via a REST API.

Figure 3.1 revisits the *NetManager's* architecture discussed in Section 2.3.3 and extends it by incorporating the proposed approach, which will be explained in detail over the next subsections.

3.2.1. eBPF Module

eBPF Modules are plugins that can be developed by third parties to extend Oakestra's worker nodes with custom VNFs. An eBPF Module includes the code for the DP compiled into eBPF bytecode, the CP compiled into a shared object, and some glue scripts to build the eBPF Module.

Data Plane

The DP of an eBPF module is implemented in eBPF and is attached to both the ingress and egress paths of the virtual Ethernet ports that are connected to the bridge (*veth1* and *veth2* in Figure 3.1). Therefore, the DP must provide two functions: one for managing ingress traffic and another for managing egress traffic.

A critical detail that arises from this design is the distinction between how traffic is classified depending on its direction. Traffic forwarded from the service to the bridge is categorized as egress traffic from the service's perspective but is treated as ingress traffic by the ports on the bridge and vice versa. Consequently, traffic forwarded from a service toward the bridge passes through the ingress function of a DP, while traffic forwarded from the bridge toward a service passes through the egress function of a DP.

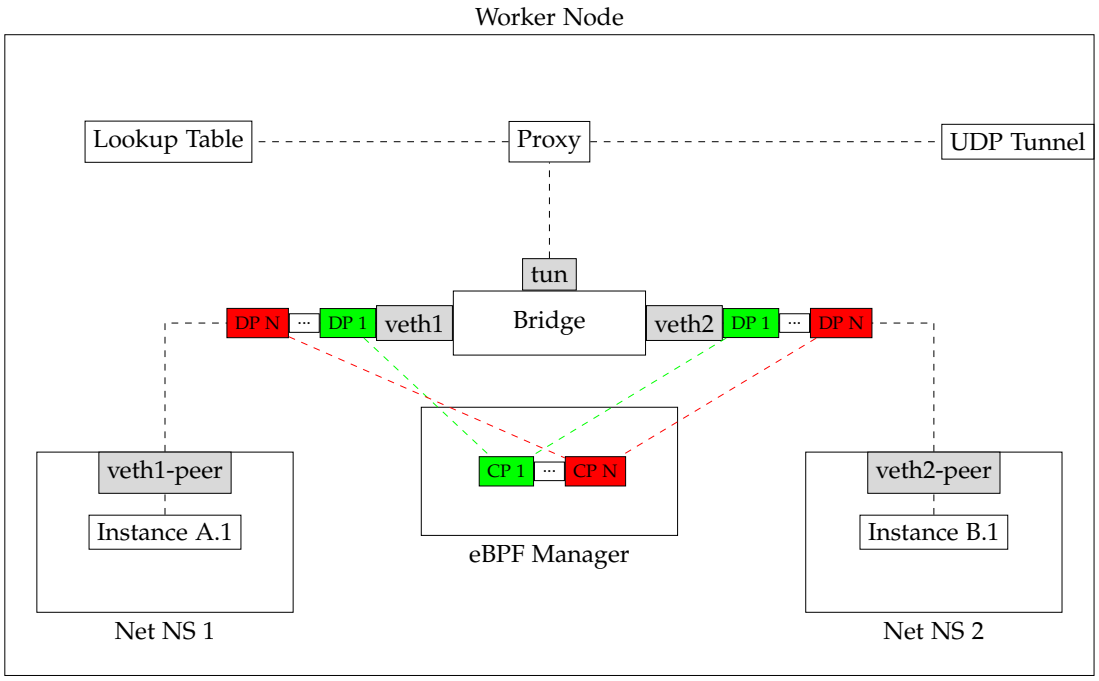


Figure 3.1.: Illustration of Chained Data Plane Instances Between Services and the Bridge

An eBPF module attaches its DP to every service, resulting in one DP instance operating on the traffic of each service. When multiple eBPF modules are deployed on the same worker node, their DP instances are chained together at each service in the order they were deployed (as shown in Figure 3.1). The output of one DP serves as the input for the next. As a result, packets on the egress or ingress path pass through every DP one by one. It is important to note that the order of these instances has a significant impact on the behavior of the chained eBPF programs. For example, if a packet is dropped by a firewall at the beginning of the chain, it will never reach the other DP instances.

If an entire eBPF module is removed, the eBPF Manager removes the corresponding DP instances from all chains while maintaining the order of the remaining DP instances.

With this position in the worker node's network, the DP instances have direct access to the traffic generated by the services, intercepting it before it reaches any other components of the *NetManager*. This enables the DPs to filter, analyze, or manipulate packets at an early stage, remaining effectively transparent to the rest of the network components. As a result, the eBPF framework can operate seamlessly alongside the original implementation, with other components unable to detect whether packets have

been altered or dropped by an eBPF DP before.

For the remainder of this thesis, the following terminology will be used:

- **Data Plane** refers to the eBPF code.
- **Data Plane instance** refers to a deployed instance of the data plane, which is typically part of a chain of DP instances from other eBPF Modules.
- **An eBPF Module's data plane** refers to all data plane instances associated with one eBPF Module and managed by a single CP (represented by the colors in Figure 3.1).

Control Plane

The CP is a program that is compiled into a shared object and loaded into the *NetManager's* process at run-time. The CP must implement an interface expected by the eBPF Manager, enabling the eBPF Manager to instantiate and interact with the CP.

The primary role of the CP is to supply its DP instances with the necessary information to perform their tasks. Communication between the CP and DP occurs through eBPF maps.

Upon initialization, the CP receives the File Descriptors (FDs) of its corresponding DP instances from the eBPF Manager. Additionally, it can register subroutes within the eBPF Manager's API, enabling external configuration of the CP.

3.2.2. eBPF Manager

The eBPF Manager fulfills a role similar to that of the Management Plane (MP) within the context of VNFs. It manages the lifecycle of the deployed eBPF Modules and is controlled via an API, which should not be exposed to the end-user (developer). Instead, it is designed to be used by components like the *NodeEngine* or the *Cluster Manager*, to enforce network policies on eBPF-compatible worker nodes.

Managing the life cycle of eBPF Modules involves the following three core responsibilities:

- **Initialization**: Upon initializing of a new eBPF Module, the module's CP code is loaded as a shared object. Subsequently, the DP (eBPF code) is attached to the virtual Ethernet ports (see Figure 3.1). This is followed by the initialization of the CP, where the FDs of the eBPF programs and their maps are passed to the CP.
- **Maintainance**: At run-time, the eBPF Manager communicates relevant events to the CPs of the deployed eBPF Modules and monitors their health.

- **Termination**: Notifying the module of its termination, followed by detaching all eBPF programs and maps from the kernel, which were previously attached by the eBPF Manager for this particular eBPF Module.

Deployment/Undeployment

Up to this point, we have described a worker node as having a static set of deployed services. However, in reality, this is not the case. Services can be undeployed or new ones can be deployed at any time.

When a service is undeployed, the eBPF Manager removes all DP instances from the chains by detaching the associated eBPF programs and maps from the kernel and then informs the CPs.

For each new service, the eBPF Manager attaches the DP of every deployed eBPF module to a new chain at that service and then notifies the running CPs, providing them with the necessary FDs for the eBPF programs and their maps.

3.2.3. UML Class Diagram

The system design of the eBPF Manager is illustrated as a UML diagram in Figure 3.2. This diagram is not intended to serve as a blueprint for implementing the approach but rather to support the understanding of the proposed system. Certain aspects have been simplified or omitted to keep the focus on the overall architectural design. The highlighted functions are exposed through the API and represent typical CRUD functionality for eBPF modules.

The `EbpfManager` class holds a set of references to the deployed eBPF Modules, the `nextPriority` in the DP chains and a `router` representing the `/ebpf` route in the *NetManagers* API. Since the `EbpfManager` class cannot know the implementation details of an eBPF Module, as these are loaded at run-time, the `Module` interface guarantees the presence of a `base` attribute and two functions: `OnEvent()` and `DestroyModule()`. The `base` attribute of type `ModuleBase` abstracts those attributes that all eBPF Modules have in common and that are therefore known at compile-time.

When an eBPF Module is initialized via the API, its shared object is loaded, and the eBPF Manager instantiates an object of type `ModuleBase`, storing it in the module's `base` attribute. This object contains the module's `id`, `name`, `priority` in the DP chains, `config` that was passed through the API, and a `router` representing the `/ebpf/<module ID>` sub-route in the eBPF Manager's API.

Every time a new service is deployed or undeployed, an `AttachEvent` or `DetachEvent` is communicated to the eBPF Module through the `OnEvent()` method. The `AttachEvent` carries the corresponding FDs to the module's eBPF DP.

When an eBPF Module is deleted, the eBPF Manager deallocates the eBPF resources in the kernel and calls `DestroyModule()` to notify the module.

3.2.4. Modifications to the NetManager

The proposed approach required several modifications to the *NetManager*, which are addressed in this section to provide a clearer understanding of the changes.

Since the eBPF Manager needs to be aware of when services are deployed or undeployed, the existing *EventManager* was extended with callback-based events. The eBPF Manager can now register callbacks for the `ServiceCreated` and `ServiceRemoved` events and act accordingly as described in the previous sections. Currently, these events are only emitted for deployments using containers as the virtualization method, not for unikernels.

The `EnvironmentManager` interface was extended with the `GetDeployedServices` function, which provides the eBPF Manager with access to all currently deployed services, particularly the names of their virtual Ethernet pairs. This information is crucial for the eBPF Manager to attach the eBPF code to the appropriate interfaces.

3.3. Design Decisions

This section outlines the two most critical design choices made for the eBPF Manager. The goal is to enhance the transparency of these decisions and to document the reasoning and outcomes for future development and reference.

3.3.1. Underlying eBPF Hook

When deciding which underlying eBPF hookpoint to use for attaching the eBPF code of the eBPF Modules, several trade-offs must be considered.

eXpress Data Path (XDP) is generally regarded as the most performant hook because it operates at the lowest level of the Linux network stack, even before the *sk_buff* is allocated. However, as the presentation [20] from the eBPF Summit 2023 suggests, XDP achieves its maximum performance benefits in hardware mode, while in driver mode, it only slightly outperforms the Traffic Control (TC) hook. Since all interfaces relevant to the proposed approach are virtual, the performance advantage of hardware mode cannot be leveraged in this use case. Therefore, the TC hook presents minimal performance drawbacks while offering the advantage of allowing eBPF programs to access both ingress and egress traffic, as well as the *sk_buff* and its associated metadata.

Another advantage of the TC hook is that Queuing Disciplines (QDISCs) natively support attaching multiple programs and provide a built-in mechanism for chaining

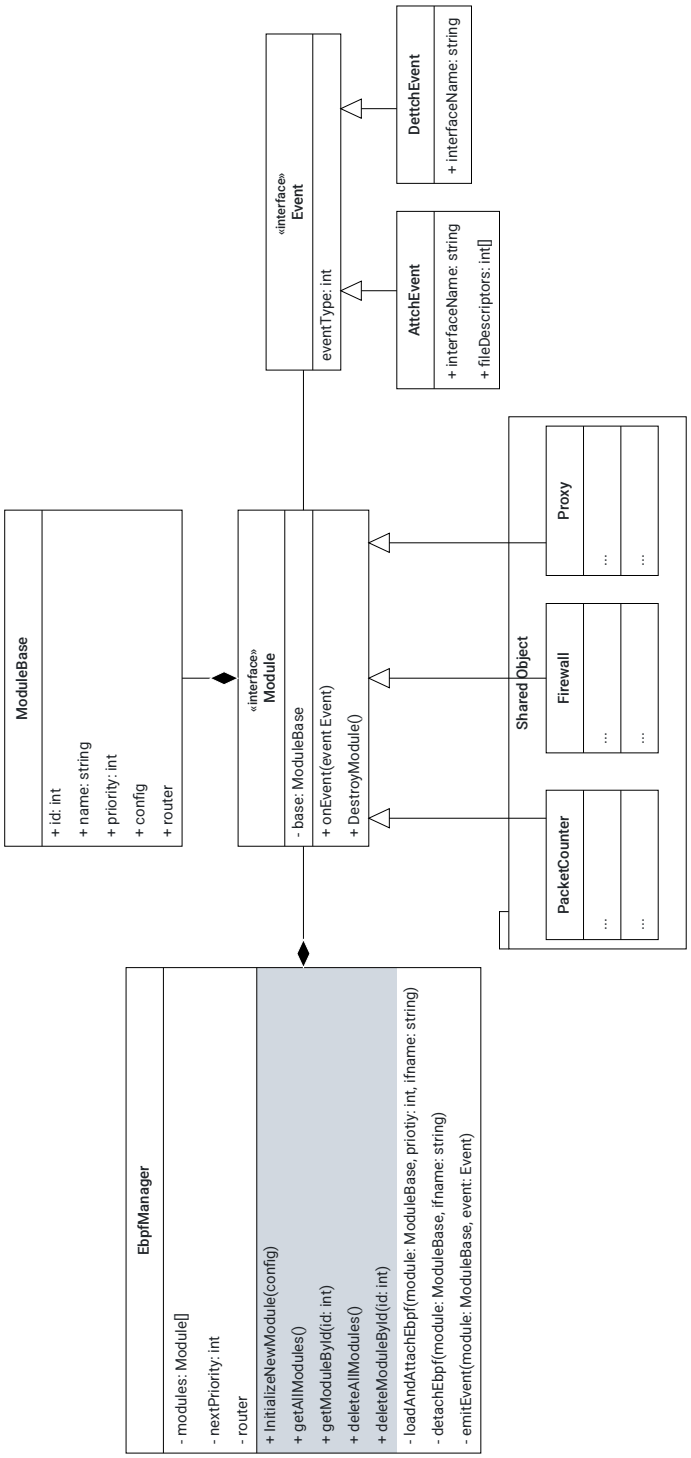


Figure 3.2.: UML Class Diagram of the eBPF Manager

these together. By assigning a priority within a QDISC, it's possible to determine the order in which the eBPF programs are executed. In the case of XDP, a custom chaining mechanism would need to be built using eBPF tail calls, as done by Polycube[24].

Taking the discussed trade-offs into account, TC was chosen as the underlying eBPF hook. However, future work could explore abstracting the hookpoint, similar to the approach in Polycube[24], allowing the most appropriate hookpoint to be selected for each application individually.

3.3.2. Plugin Architecture

As outlined in Section 3.1, an eBPF Module is intended to be dynamically loaded at runtime, similar to a plugin. Since the DP must run in the kernel, this discussion focuses on the CP, which operates in user-space. Three common architectural approaches for pluggable software were considered, each offering different levels of encapsulation. The CP of an eBPF module could be:

1. Executed in their own virtualized environment (Virtual Machine (VM)/Container).
2. Executed in a child process.
3. Loaded as a shared object into the main process of the *NetManager*.

Using a full VM is ruled out for several reasons. First, it would be far too resource-intensive, and second, applications running inside the VM would not have access to the host kernel, where the CP resides.

Lightweight virtualization, such as containers, typically shares the host kernel, which could, in theory, be a viable option. However, the main motivation for running a plugin in a container is to isolate it from the host system. Since we are already loading external eBPF code directly into the kernel, the most critical component of a machine, the additional isolation provided by containers becomes questionable, considering the significant resource overhead this approach entails.

Running the CP in its own child process offers a lower level of encapsulation, but by restricting the process's permissions, you can still mitigate some major risk factors. The main process and the child process could communicate via a socket or another inter-process communication method. Compared to the third approach, this option requires more resources and maintenance since the main process needs to manage all child processes. However, it is a solid approach that could be implemented in the future.

As should now be clear, the third approach was chosen, primarily due to its simplicity and performance. It is quick to implement while still fulfilling all major requirements.

3.4. API Description

This section aims to document the REST API exposed by the eBPF Manager (see Section 3.2.2). The API is exposed as part of the *NetManager*'s API (default port 6000) and is located under the `/ebpf` route.

Get all eBPF Modules

```
GET /ebpf
Content-Type: None
```

This request returns a list of all eBPF Modules currently managed by the eBPF Manager in JSON format.

Get a specific eBPF Module by ID

```
GET /ebpf/<id>
Content-Type: None
```

This request returns the eBPF Module corresponding to the specified ID in JSON format. If no module with this ID exists, a `404 NOT FOUND` HTTP status code is returned.

Create a new Instance of an eBPF Module

```
POST /ebpf
Content-Type: application/json
Body:
{
    "name": "<module name>",
    "config": {...}
}
```

This requests lets the eBPF Manager initialise the eBPF Module under: `ebpfManager/ebpf/<module name>`.

The specified path must include the DP compiled into bytecode and the CP compiled into a shared object. If either or both files are missing from this path, or if any other error occurs, a `500 INTERNAL` HTTP status code is returned.

The `config` attribute is a JSON object passed to the eBPF Module during initialization. It can contain arbitrary configuration parameters specified by the eBPF Module.

Delete all eBPF Modules

```
DELETE /ebpf
Content-Type: None
```

This request deletes all eBPF Modules currently managed by the eBPF Manager.

Delete an eBPF Module by ID

```
DELETE /ebpf/<id>
Content-Type: None
```

This request deletes the eBPF Module corresponding to the specified ID. If the modules were successfully deleted or no module with this ID exists, a `200 OK` HTTP status code is returned.

eBPF Module Subroutes

As mentioned in Section 3.2.1, a CP can register a subroute within the eBPF Manager's API during its initialization. These subroutes are registered under:

`/ebpf/<eBPF Module ID>`

A typical flow begins with initializing an eBPF module via a `POST` request to `/ebpf`. The response will include an ID that can be extracted to access the API of the eBPF Module.

3.5. eBPF Modules

As part of this thesis, three eBPF Modules were developed, each exemplifying a typical type of network function: traffic analysis (Packet Counter), traffic filtering (Firewall),

and traffic manipulation (Proxy). This section provides an overview of the architecture and structure of these modules.

3.5.1. Packet Counter

For each of its DP instances, the Packet Counter[31] tracks the number of packets that have passed through its ingress and egress paths since initialization. These counters are stored in an array map with two slots, one for ingress and one for egress. They are read from user-space upon request via the API. Therefore, the Packet Counter registers the following request:

```
GET /ebpf/<module ID>/counts
Content-Type: None
```

The returned JSON object contains the ingress and egress counters for each DP instance, identified by the name of the interface to which it is attached:

```
[
    "<interface name>": {
        "ingress": <count>,
        "egress": <count>
    },
    ...
]
```

3.5.2. Firewall

The firewall[30] implements a lookup table for its firewall rules using an eBPF map of the Hashmap type. The keys in this map are 5-tuples consisting of the source IP, destination IP, protocol, source port, and destination port. If a packet matches a key, it is allowed through; if no matching key is found, the packet is dropped. To configure the lookup table, the firewall registers the following request:

```
POST /ebpf/<module ID>/rule
Content-Type: application/json
Body:
{
    "srcIp": "<IP>",
    "dstIp": "<IP>",
    "proto": "<PROTOCOL>",
    "scrPort": <Port>,
    "dstPort": <Port>
}
```

Limitations of the Firewall

Traffic can only be allowed between specific IP addresses, not entire IP ranges. Additionally, the firewall only supports IPv4 and operates on UDP, TCP, or ICMP protocols, while other protocol stacks are ignored.

3.5.3. Proxy

The proxy[32] is the most complex of the three eBPF Modules and is designed to replace the Oakestra Proxy.

Each proxy DP instance manages three eBPF maps:

- `translations`**:** Used to translate a service IPs, instance IPs and namespace IPs.
- `open_sessions`**:** Caches open sessions with other services.
- `ip_updates`**:** When the DP receives a packet directed to an IP that does not yet have an entry in the `translations` map, it instructs the CP to perform the necessary Table Query in user-space using this map.

Outgoing Proxy

For each outgoing packet from the perspective of a service (ingress path on the DP), the DP first checks whether the destination IP belongs to the `10.30.0.0/16` range. If the packet's destination IP does not fall within this range, the packet is simply forwarded without any further processing by the proxy.

However, if the destination IP is from that IP range, the DP performs a lookup in the `open_sessions` map to determine if there is already an existing session between the source instance and the destination. If such a session exists, the destination IP is replaced with the corresponding namespace IP from the proxy cache.

If no active session is found, the destination IP is translated into a namespace IP using the `translations` map. An instance is selected based on the appropriate balancing policy. A new entry is then created in the `open_sessions` map to cache this newly established session.

But what happens if the `translations` map does not yet contain a translation? In that case, the entire packet is copied to the `ip_updates` map, triggering a perf event[21] that notifies the user-space CP of the missing translation. Perf events are a Linux mechanism that enables user-space applications to be notified of kernel events without requiring constant polling.

The CP can then initiate a Table Query in Oakestra for the corresponding IP to retrieve the translation. If no translation is found, the packet is dropped. If a translation is found, the `translations` map is updated accordingly, and the packet is re-injected into the interface. The packet will then go through the outgoing proxy procedure again, but this time, the necessary translation will be present in the `translations` map.

Because modifying the destination IP within the IP header requires recalculating the L3 and L4 checksums, this step is performed as the final action before the packet is forwarded.

The entire outgoing proxy procedure is illustrated as a flowchart in Figure 3.3. The gray-shaded nodes represent processes and decisions occurring in user-space, while the others take place in kernel-space.

Incoming Proxy

Since the outgoing proxy modifies the destination IP address of packets after they leave the sender, there must be a corresponding mechanism to translate the source address of the responses from the target service, which will now be a namespace IP, back to the original IP. This translation is necessary because, without it, the host would be unable to correctly associate the response with the original request and would most likely drop the response.

For each incoming packet from the perspective of the services (egress path on the DP), the DP must check the `open_sessions` map to verify whether a session was previously established. If no session is found, the packet is irrelevant to the proxy and is not processed further.

If a session is found, the source address in the packet is replaced with the original namespace IP that was used when the session was initiated.

Modifying the source IP within the IP header requires recalculating the L3 and L4 checksums, and this operation is performed as the final step before the packet is forwarded.

Implementation Outlook and Limitations

The previous Section 3.5.3 outlined the system design of a proxy developed throughout this thesis. However, the actual implementation deviates in some respects from the design described above. This is because the proxy and its architectural ideas were developed gradually, with the initial goal of creating a proof of concept rather than a production-ready proxy.

Currently, sessions between two instances are not stored by their instance IPs in `open_sessions`, but rather by their namespace IPs. This approach is functional as long as no service redeployments occur, which could result in changes to the namespace IPs. However, since no unexpected redeployments occurred in a controlled test environment, this limitation was considered acceptable for the purposes of this thesis. In a production-ready proxy, sessions would need to be established between instance IPs rather than namespace IPs to ensure functionality across multiple worker nodes. As a result of this simplification, the proxy currently does not handle instance IPs properly and operates exclusively on service IPs.

Another significant deviation from the proposed system design is that in the current implementation, the entire user-space flow of the proxy (represented by the gray nodes in Figure 3.3) has been considerably simplified due to an issue with the eBPF verifier. Ideally, `ip_updates` would be implemented as a ring buffer map, allowing memory allocations for complete network packets to be copied into user-space, as previously described. However, the verifier repeatedly flagged this memory reservation as "out of bounds." As a workaround, only the destination IP address, rather than the entire packet, is copied to user-space, and the packet is dropped. User-space then triggers a table query for the respective IP and updates the `translations` map accordingly. This approach functions because the packet is typically retransmitted shortly afterward, by which time the appropriate translation is likely available in the `translations` map. However, this work-around leads to an increased retransmission rate.

The verifier requires that the size of an eBPF map be known at compile time. As a result, the maximum number of sessions that can be stored must be predetermined at compile time. This constraint is inherent to the design of eBPF, but it can largely be mitigated by allocating sufficiently large values, assuming there is enough RAM to accommodate these eBPF maps.

Currently, the proxy only supports the round-robin load balancing policy and is limited to IPv4 traffic, ignoring other protocols. Additionally, the proxy does not currently support IPv4 options; the header must be exactly 20 bytes long.

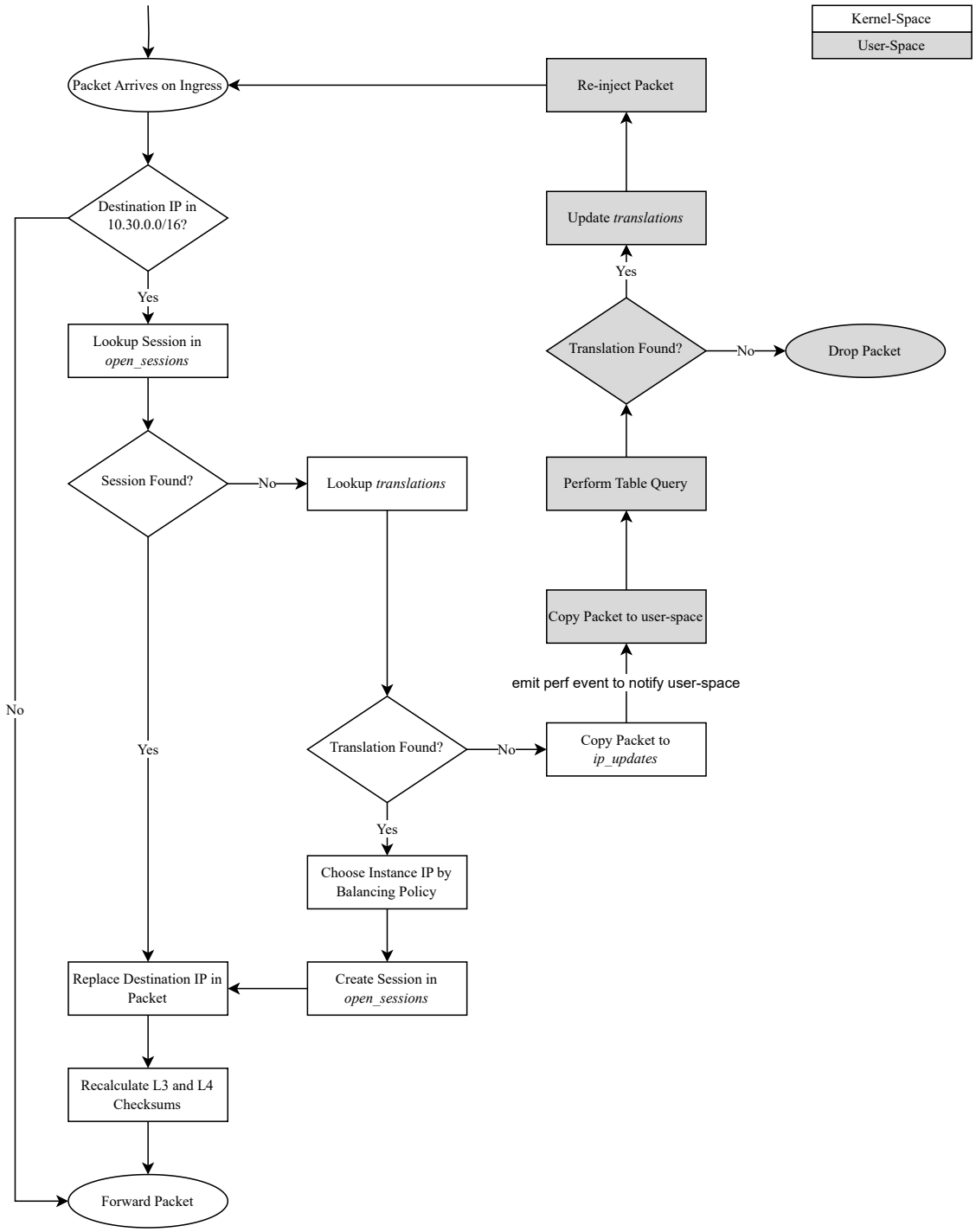


Figure 3.3.: Outgoing eBPF Proxy Flowchart

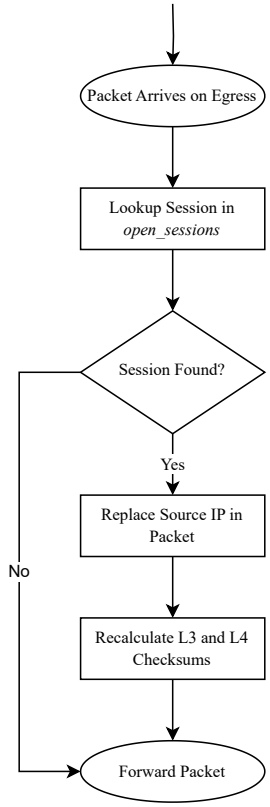


Figure 3.4.: Incoming eBPF Proxy Flowchart

4. Evaluation

This chapter focuses on evaluating the eBPF Manager, whose design was introduced in Chapter 3 and implemented as part of this thesis. The primary objective of this evaluation is to assess whether the utilization of eBPF-based Virtual Network Functions (VNFs) within Oakestra can deliver the anticipated performance improvements in practical scenarios.

Four series of measurements were conducted, described in Sections 4.2, 4.3, 4.4, and 4.5, each focusing on network characteristics that are critical for the performance of edge deployments and networks in general.

4.1. Test Setup

All measurements were conducted on CloudLab[12] to ensure reproducibility. CloudLab is a research platform designed to create customizable network environments, enabling the testing of various network architectures and configurations. A specific environment can be defined within a profile, allowing tests to be repeatedly executed in identical conditions.

The measurements were conducted on an `xl170`[5] node, with the Root Orchestrator, Cluster Orchestrator, *NodeEngine*, and *NetManager* all deployed on the same node. Below is a summary of the key specifications under which the measurements were conducted. For further details, please refer to the hardware specifications[5] and the provided CloudLab profile in Appendix A.1:

- **Architecture:** x86_64
- **RAM:** 8 x 8GB = 64GB
- **CPU Cores:** 10
- **Clock Speed:** 2400Hz
- **OS:** Ubuntu 22.04

In all four measurements, the original implementation of the Oakestra proxy is used as the baseline for comparison against the eBPF proxy implementation described in

Section 3.5.3. Before each measurement is conducted, any services from previous executions are cleaned up if present. Afterward, a client and a server are deployed as two services in Oakestra, and the measurements are executed between them.

Currently, only the proxy has been implemented in eBPF, while the tunnel has not. If the measurements were conducted across multiple worker nodes, the tunnel would counteract the expected performance benefits of the eBPF-based proxy. Therefore, a single-node setup was selected for the measurements.

To access the raw data or to reproduce the results from this chapter, detailed instructions can be found in Appendix A.

4.2. Latency

Latency, particularly Round-Trip Time (RTT), is a fundamental metric for any network topology as it represents the minimum time required to receive a response to a request. To compare the RTT behavior of the original and eBPF proxy, 30 measurements were performed for each implementation. During each measurement, the client sent 15 TCP pings using hping3[36], resulting in a total of 450 ping samples per implementation.

During a TCP ping, the client initiates a TCP three-way handshake by sending the SYN packet. The time it takes to receive the SYN-ACK response from the server is then recorded. ICMP pings were not used because ICMP traffic is not proxied within Oakestra.

For each implementation, the ping samples were plotted as a Cumulative Distribution Function (CDF) in Figure 4.1 to allow for estimation and comparison of their distributions, with the blue line representing the Original Proxy and the orange line representing the eBPF Proxy.

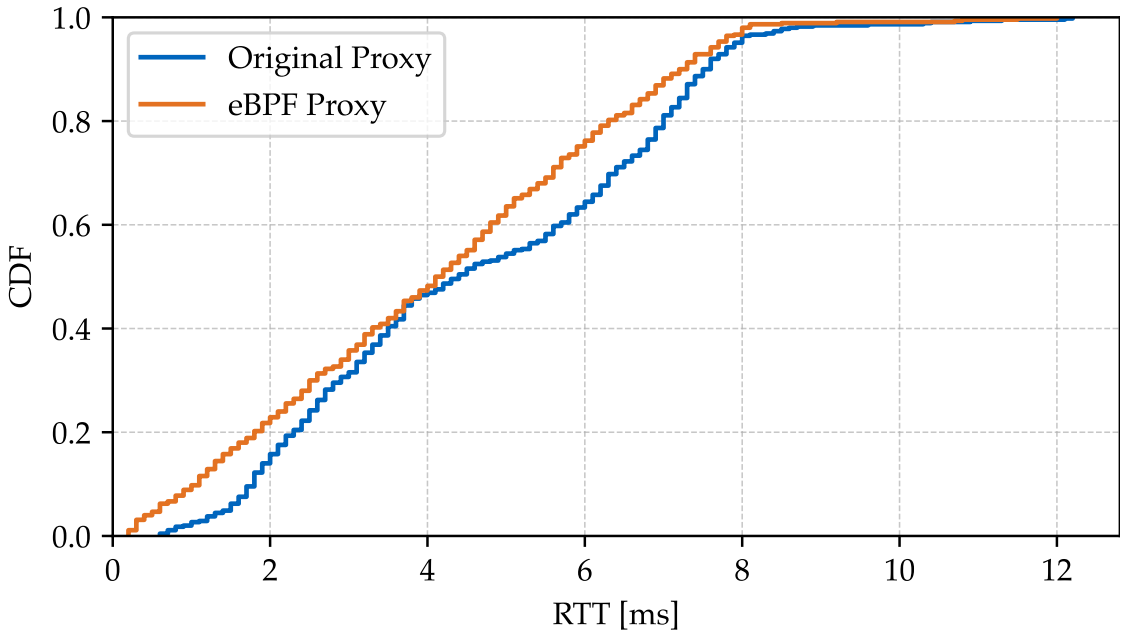


Figure 4.1.: Latency Comparison between Original and eBPF Proxy

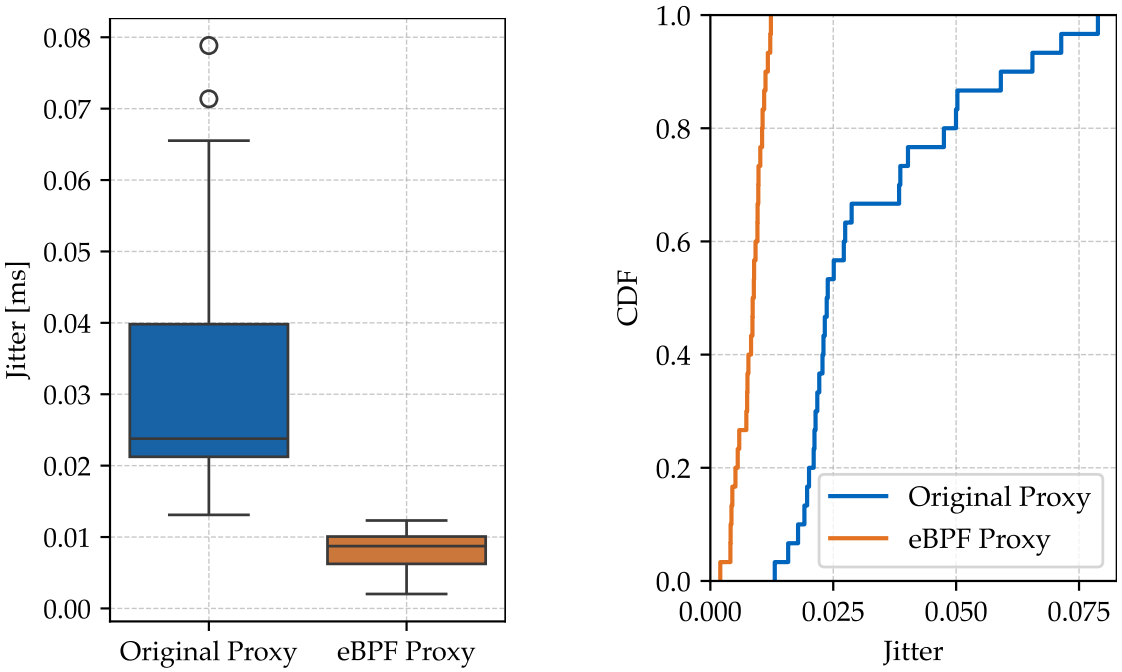
The results indicate that the eBPF implementation performs better than the original implementation, with a higher proportion of RTT samples showing a lower latency. However, the overall shapes of the curves are not drastically different, especially when looking at the tail latencies. The tail of the Original Proxy curve extends slightly further, indicating that some packets experienced higher latencies, reflecting less consistent performance. However, for both implementations, less than 4% of the samples have an RTT higher than 8ms. On average, the eBPF implementation achieves a latency of 4.14ms, compared to 4.66ms for the original proxy, which is an improvement of 11.16%.

4.3. Jitter

As mentioned in the introduction, edge deployments are often advantageous for real-time applications because they bring data processing closer to the source of the data. However, real-time applications typically require a consistent data stream to maintain quality. Jitter, which is the variation in packet arrival times, can disrupt this consistency, leading to gaps or overlaps in the data stream.

Since eBPF allows for packet processing directly in the kernel without the need to copy packets to user-space or wait for context switches, it is expected that jitter can be reduced, improving the stability of packet arrival times.

To investigate this, 30 measurements were conducted where a 1 Mbit/s UDP data stream was transmitted for 20 seconds between client and service using *iPerf3*[17]. The results of these measurements are depicted in Figures 4.2a and 4.2b, displayed as a boxplot and a CDF, respectively.



(a) Boxplot (b) CDF

Figure 4.2.: Jitter Comparison between Original and eBPF Proxy

The eBPF Proxy, plotted in orange, demonstrates considerably lower jitter values compared to the Original Proxy, which is plotted in blue. The CDF curve for the eBPF Proxy quickly reaches a value of 1 at around 0.02 seconds of jitter. This suggests that all jitter values for the eBPF Proxy are below this threshold, indicating a highly consistent and low-jitter performance.

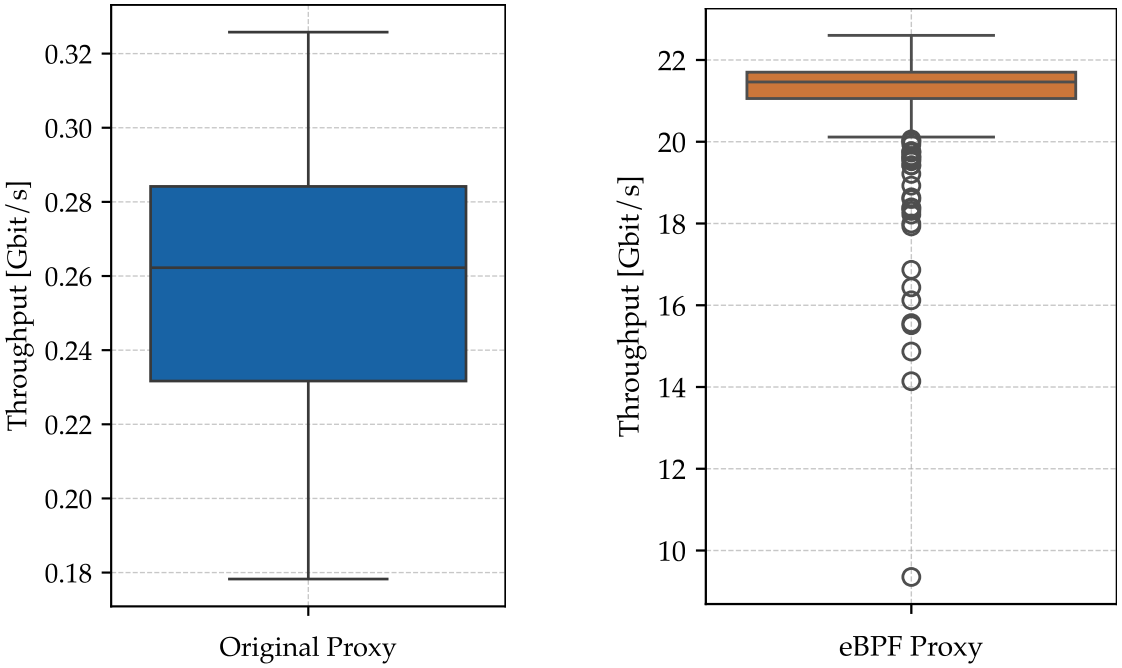
In contrast, the Original Proxy shows a broader range of jitter values, with its CDF curve only reaching 1 at around 0.08 seconds of jitter. This broader range indicates that the Original Proxy experiences more variability in jitter, leading to less consistent performance. The less steep slope of the Original Proxy's CDF curve further supports this observation, as it indicates that jitter values are more widely dispersed, contributing to greater instability in network performance.

On average, the jitter is 0.0326 ms for the Original Proxy and 0.0082 ms for the eBPF

Proxy, representing a 74.85% improvement with the eBPF Proxy. In summary, the eBPF Proxy significantly outperforms the Original Proxy in terms of jitter.

4.4. Throughput

Another critical metric for services deployed at the edge is the achievable throughput between them. To evaluate this, a TCP data stream was established between the service and the client without any bandwidth limitations, causing *iPerf3*[17] to push as much data as possible from client to server. Since this is a TCP connection, the underlying Congestion Control Algorithm (CCA) determines the maximum sending rate for the client, which in these measurements was CUBIC[16].



(a) Throughput of Original Proxy

(b) Throughput of eBPF Proxy

Figure 4.3.: Throughput Comparison between Original and eBPF Proxy

Figures 4.3a and 4.3b illustrate that the eBPF implementation achieves an average throughput of 21.2Gbit/s, which is over 100 times greater than the original implementation's average throughput of 0.26Gbit/s. Given the latency measurements in Section 4.2, we can conclude that this increased throughput is not mainly due to reduced per-packet latency, as there is too little difference between the two implementations in that regard.

This suggests that the eBPF implementation can handle more packets simultaneously, leveraging its strong parallelism in packet processing to its advantage.

4.4.1. Bottleneck Estimation

The previous measurement does not indicate whether the original proxy is indeed a bottleneck in this setup or if the low throughput results shown in Figure 4.3a are due to a particularly conservative estimation by the underlying CCA. While this is unlikely, the CCA operates as a black box in this scenario, making it difficult to directly assess its behavior.

Therefore, a new measurement was conducted by establishing a UDP data stream between the client and server with increasing target throughputs. Since UDP's sending rate is not controlled by a CCA, *iPerf* can increase the sending rate until the network buffers at the bottleneck become fully congested. This congestion should be measurable on the server side in the form of increased one-way latency as the bottleneck network buffer starts filling up. If the sending rate is increased to the point where the bottleneck buffer overflows, rising packet loss should also be measurable at the server. If these indicators of a bottleneck appear at around the previously measured 0.26 Gbit/s, we can conclude that the proxy must be a bottleneck in this scenario since a higher throughput of over 20 Gbit/s was already demonstrated to be possible when replacing the original implementation with the eBPF-based version.

Figure 4.4 presents the results of this measurement. Each data point represents the average latency and relative packet loss for a UDP data stream transmitted between the client and server over a 60-second period at a given target throughput rate. The target throughput rate in *iPerf* was increased in 50 Mbit/s increments.

The results indicate that for target throughputs up to 150 Mbit/s, latency remains largely constant and packet loss is nearly zero, suggesting no signs of congestion up to this point. However, as we approach the previously calculated bottleneck of approximately 260 Mbit/s, the average latency begins to rise, which can be attributed to the network buffers at the proxy filling up faster than they can be processed. From around the 250 Mbit/s mark, significant packet loss is observed, indicating that beyond this data rate, the network buffer reaches its full capacity, resulting in packets being dropped.

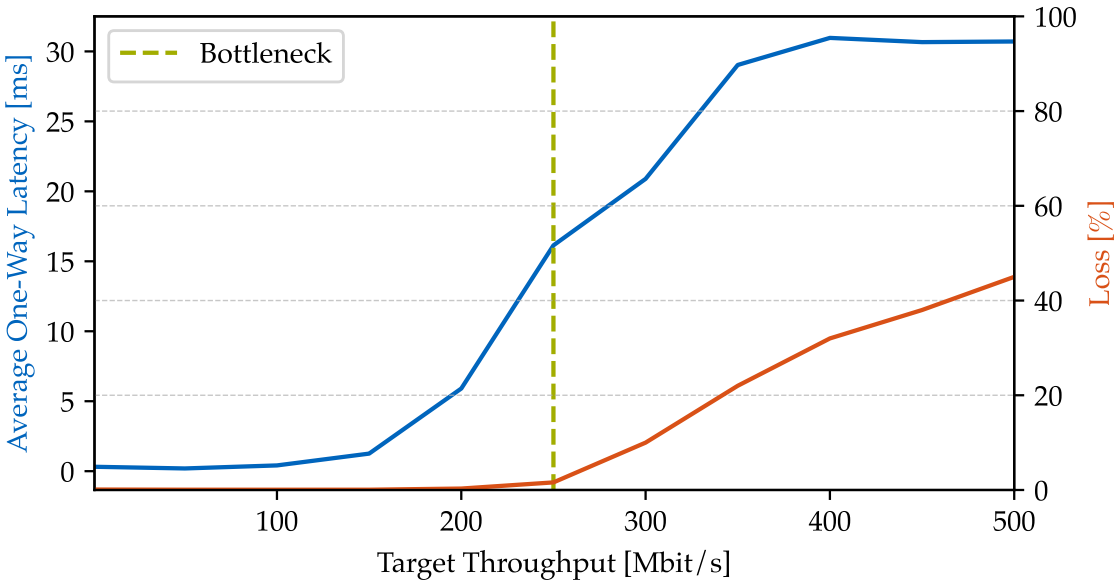


Figure 4.4.: Bottleneck Measurement of the Original Proxy

Since CUBIC is a loss-based CCA, it is now clear why the previous measurement yielded an average throughput of approximately 260 Mbit/s. This corresponds to the throughput where the network buffer at the proxy begins to experience significant congestion and packet loss occurs. In summary, it can be concluded that the original proxy is indeed a bottleneck in Oakestra, which can be drastically optimized by employing an eBPF-based alternative, leading to an 8053.85% improvement in average throughput in the presented measurements.

4.4.2. Limitations

Unfortunately, it was challenging to generate a UDP throughput of multiple Gbit/s using *iPerf*. Although the selected packet size and UDP buffer size could influence the resulting throughput, no higher throughput than 5 Gbit/s was ever achieved on the testing hardware. The exact reason for this could not be determined, but therefore, it was impossible to repeat the bottleneck measurement for the eBPF proxy and confirm that the 21.2 Gbit/s approximated by CUBIC is indeed the absolute maximum throughput that can be achieved. Although unlikely, this could also be a conservative underestimation of the maximum throughput.

4.5. Resource Consumption

As highlighted multiple times throughout this thesis, edge servers generally have fewer resources than centralized cloud servers, making it essential for an edge orchestration framework to minimize its resource consumption, thereby maximizing the availability of resources for the actual applications.

30 measurements were conducted, each lasting 45 seconds. During the entire test duration, RAM and CPU samples were taken every 0.5 seconds. The samples were categorized into three phases: Idle, Load, and Cleanup, each occupying one-third of the total test duration.

- **Idle:** During the Idle phase, only the core components of Oakestra were running. This phase establishes the baseline and provides a reference point for the subsequent phases.
- **Load:** At the start of the Load phase, two services were deployed on the worker node, with a 100 Mbit/s TCP data stream established between them using *iPerf3*[17]. A data rate of 100 Mbit/s was chosen based on the findings from Section 4.4. This rate is expected to be sufficient to induce noticeable changes in CPU load while preventing congestion or system overload.
- **Cleanup:** In the third phase, the two services were undeployed to observe how close the system can return to the Idle state and how quickly it does so.

CPU

Figures 4.5, 4.6 illustrate the CPU usage of throughout the three phases. The gray lines in the background represent the actual CPU samples, while the black line shows the Exponentially Weighted Moving Average (EWMA) with $\alpha = 0.2$, applied to smooth out the results. The three phases are highlighted in green, red, and blue.

Figures 4.5 and 4.6 clearly illustrate that the eBPF proxy results in significantly lower CPU usage during the load phase compared to the original proxy. Specifically, the eBPF implementation maintains an average CPU usage of 5.39% during this phase, whereas the traditional proxy consumes 20.19% of CPU resources. Following the load phase, both implementations exhibit the expected decrease in CPU usage, returning to idle phase levels during the cleanup phase.

RAM

Unfortunately, the RAM samples showed no significant changes in consumption on the test hardware.

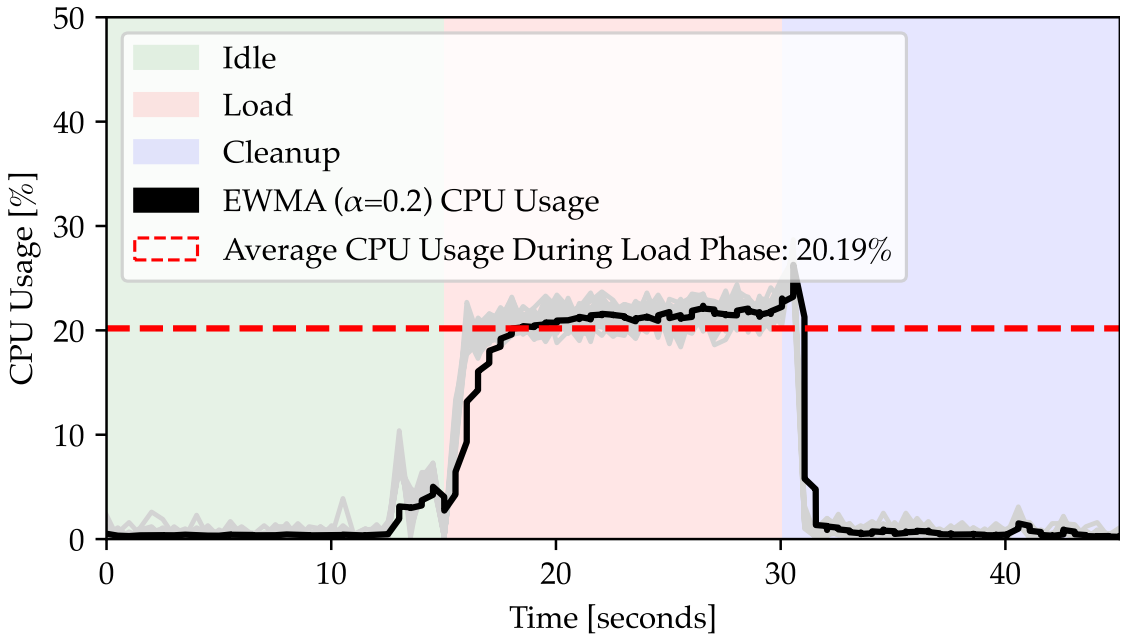


Figure 4.5.: Original Proxy: CPU Usage Over Time

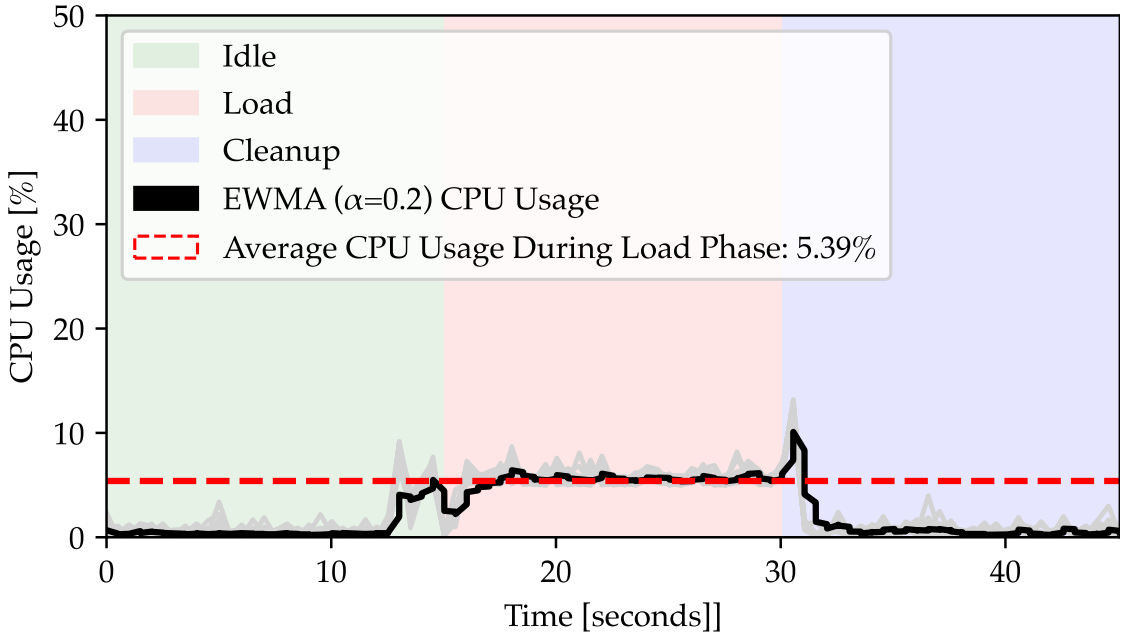


Figure 4.6.: eBPF Proxy: CPU Usage Over Time

5. Conclusion

The primary goal of this thesis was to answer the research questions introduced in Chapter 1. Therefore, Section 5.1 revisits the research questions and discusses whether and how successfully they were addressed. Additionally, Section 5.2 outlines potential directions for future research.

5.1. Findings

Is it possible to leverage Extended Berkeley Packet Filter (eBPF) in Oakestra to effectively deploy Virtual Network Functions (VNFs) in edge environments?

A framework for loading so-called eBPF modules into Oakestra worker nodes was successfully designed and implemented. Each eBPF module includes all the essential components typically found in a VNF, such as a Data Plane (DP) and a Control Plane (CP). The Management Plane (MP) resides in the proposed eBPF Manager, which handles the lifecycle management of the loaded modules.

With this setup, eBPF modules can analyze, filter, and manipulate all traffic between services deployed on eBPF-compatible worker nodes.

The following Non-Functional Requirements (NFRs) were established for the eBPF Manager and successfully implemented in the proposed design:

- **Compatibility**: The proposed approach allows for both eBPF-compatible and non-eBPF-compatible worker nodes. For each worker node, it can be individually determined whether and which eBPF modules should be loaded onto it. Additionally, within a single worker node, it can be dynamically decided whether to utilize eBPF, without impacting the existing Oakestra components.
- **Modularity**: Developing and loading an eBPF module onto a worker node does not require modifying Oakestra code, nor does it require restarting the *NetManager* or any other Oakestra component. eBPF modules can be loaded, configured, and removed at run-time.
- **Efficiency**: While the efficiency of an eBPF module ultimately depends on its implementation and usage, the proposed eBPF proxy module clearly demonstrated

that eBPF modules in Oakestra can significantly optimize the efficiency on worker nodes.

- **Scalability**: Scalability in terms of the number of worker nodes is ensured by Oakestra's design, while scalability with respect to the number of instances on a single worker node is achieved by the fact that each instance operates with its own DP instance chain running in front of it.

Additionally to the eBPF framework, the thesis has also successfully introduced three eBPF Modules: a packet counter, a firewall, and a proxy.

How can existing network functions, such as Oakestra's proxy, be optimized using eBPF to enhance performance?

Oakestra's proxy enables inter-service communication between services by implementing Oakestra's semantic addressing scheme. It is particularly relevant in the context of eBPF and is therefore explicitly mentioned in the research question, as the proxy runs as a user-space application. Implementing VNFs entirely in user-space presents several drawbacks, which were discussed in this thesis. The hypothesis was that implementing the proxy in the kernel using eBPF could optimize its performance.

This hypothesis was validated, as the proposed eBPF-based implementation of the proxy demonstrated (significantly) better performance across all examined metrics. This clearly indicates that Oakestra can achieve a substantial performance enhancement by leveraging eBPF for the proxy.

What measurable performance advantages does the implementation of eBPF network functions provide for Oakestra in edge environments?

Utilizing the newly proposed eBPF framework, its potential in terms of network latency, jitter, throughput, and resource consumption was analyzed using the proxy as a case study. The key findings are summarized in Table 5.1. The proposed eBPF-based implementation of the proxy demonstrated significant improvements across all examined metrics.

The average Round-Trip Time (RTT) between two services in Oakestra was reduced by 11.16% with the eBPF implementation, lowering the average RTT from 4.66 ms to 4.14 ms.

Jitter was reduced from an average of 0.0326 ms to 0.0082 ms, representing a 74.85% improvement.

The most notable results were seen in the throughput experiments, where throughput increased by 8053.85%. While the original proxy implementation hit a bottleneck at around 260 Mbit/s, the eBPF implementation exceeded 21.2 Gbit/s.

In terms of resource consumption, it was demonstrated that while performing the same task, the eBPF proxy used, on average, 73.30% less CPU than the original proxy.

Measurement	Original Proxy	eBPF Proxy	Improvement
Latency	4.66 ms	4.14 ms	11.16%
Jitter	0.0326 ms	0.0082 ms	74.85%
Throughput	0.26 Gbit/s	21.20 Gbit/s	8053.85%
CPU Usage	20.19%	5.39%	73.30%

Table 5.1.: Summary of Evaluation Results

It is important to note that these results are heavily dependent on the hardware used and may not be generalizable under all circumstances. However, the trend and potential of eBPF at the edge have been clearly demonstrated.

5.2. Future Work

The work presented in this thesis has laid a solid foundation for the use of eBPF within edge environments and the Oakestra framework. However, there are several paths for future research and development that could further enhance the system's capabilities and address some of the limitations encountered during this project. This section outlines potential future work that could build upon the achievements of this thesis.

5.2.1. Integration with Oakestra

So far, the eBPF Manager only exposes an API that has not yet been utilized by any other component within Oakestra. In the future, components such as the Cluster Manager or the *NodeEngine* will need to use this API to enforce network policies on eBPF-compatible worker nodes.

For instance, firewall rules defined in an Service-Level Agreement (SLA) could be implemented using an eBPF-based firewall on the worker node instead of *iptables*, provided the worker node supports eBPF.

For entirely new eBPF Modules that do not extend or replace existing VNFs, an SLA would need to include a new configuration option. For example, SLAs could have an optional `ebpfModules` attribute, listing desired eBPF Modules and their configurations. If this field is set, services defined by the SLA can only be deployed on eBPF-compatible worker nodes that support the specified eBPF Modules.

To illustrate this concept, Figure 5.1 shows a hypothetical example of what such a future SLA might look like. In this example, a Docker container `my_backend:1.1` is

deployed behind an eBPF-based Network Address Translation (NAT). Consequently, the deployment must be placed on a node with the `ebpfNAT` module installed. The `config` object is then passed to the eBPF module during initialization.

```
{
    ...,
    "code": "docker.io/user/my_backend:1.1",
    "ebpfModules": [
        {
            "name": "ebpfNAT",
            "config": {
                "type": "snat",
                "to_ip": "10.180.3.2",
                ...
            }
        },
        ...
    ],
    ...
}
```

Figure 5.1.: Example of a future SLA design for the deployment of a service alongside eBPF Modules

5.2.2. Executing Control Planes in Separate Processes

As described in Section 3.3.2, it was decided to load the CP of eBPF Modules as a shared object into the process of the eBPF Manager at run-time. While this approach is both efficient and quick to implement, it also carries certain pitfalls.

Go provides the *go plugin*[14] package, which facilitates the dynamic loading of other programs, simplifying the process. However, this approach currently restricts the DP to being written in Go and requires it to be compiled with the same compiler versions as the main executable.

Another issue is that if the dynamically loaded code fails, it causes the entire *NetManager* to crash. As mentioned in Section 3.3.2, an alternative approach would be to load the CP of each eBPF Module as a separate process. This would enable the eBPF Manager to continue running even if the loaded code crashes and also allow for the restriction of the process's permissions.

5.2.3. Testing

Currently, there are no tests in place for either the eBPF Manager or the eBPF Modules. In future work, it is essential to develop unit tests and, ideally, integration tests for the eBPF Manager. Testing the eBPF Modules presents a greater challenge, as eBPF requires a Linux kernel to run. To establish a testing environment for these modules, tools like bpftime[39], which offer an eBPF runtime in user-space, could be considered.

5.2.4. Heterogeneous Data Plane Chains

Currently, when an eBPF Module is loaded, the eBPF Manager adds the corresponding DP to the chain at each service. This constraint may be relaxed in the future, allowing for different numbers and types of DPs to be chained together at each service. However, particularly during the development of the proxy (see Section 3.5.3), this constraint was initially beneficial. It was clear that the proxy needed to be chained in front of all services anyway; therefore, this constraint significantly simplified the eBPF resource management.

5.2.5. Other

This section briefly summarizes three smaller suggestions for future work.

- Similar to the Polycube framework[24] described in Section 2.4, the actual eBPF hook in the kernel could be abstracted, allowing the eBPF Module to decide at runtime which hook point to utilize. This abstraction would also help prevent faulty eBPF modules from unintentionally disrupting the DP chains by ensuring that only appropriate actions, such as `TC_ACT_PIPE` or `TC_ACT_SHOT`, can be returned.
- Supporting shared maps between multiple DP instances could help conserve memory, as not every instance would need to maintain its own map when it contains shared information. For example, in the case of the proxy, maps like `translations` or `open_sessions` could be shared among all instances on one node.
- Implementing the Oakestra tunnel as an eBPF Module and integrating it with the proxy could yield interesting results when repeating the experiments from Chapter 4, particularly when deploying the client and server across different nodes.
- As outlined in Section 3.5.3, the eBPF Proxy should establish sessions based on instance IPs rather than namespace IPs.

- As mentioned in Section 3.5.3, when the eBPF proxy does not know the translation for an incoming packet, it should copy the packet to user-space. However, this has not been implemented due to issues with the verifier. The optimal solution would likely involve using ring buffers, a similar approach to the one taken by Polycube[24].
- As highlighted in Section 3.2.4, the proposed approach has so far only been implemented for the container runtime in Oakerstra and does not work with unikernels.
- As discussed in Section 3.2.4, the `ServiceCreated` and `ServiceRemoved` events are currently not emitted when unikernels are used as the virtualization method.
- As pointed out in Section 3.5.3, due to an issue with the verifier, the eBPF proxy is unable to handle IPv4 header options, which has been particularly problematic in scenarios involving fragmentation.
- So far, all eBPF DPs only support IPv4.

A. Reproducibility

All data from Chapter 4 was generated using a Python framework located in the `cloudlab-tests` directory of the complementary Git repository[33] for this thesis. This framework enables the automation and reproducibility of the measurements. The subdirectory `cloudlab-tests/raw_data` contains all the raw data from the measurements conducted for this thesis. If there is interest in not only reviewing the raw data but also reproducing it, the following instructions provide guidance on how to do so.

1. Instantiate a new experiment in CloudLab[12] using the profile provided in Figure A.1.
2. When the experiment is ready, SSH into the node that was created and set up Oakestra on that node. Make sure to use the *NetManager* version that supports the changes from this thesis.
3. Install the Python dependencies from `cloudlab-tests/requirements.txt`.

```
$ pip install -r requirements.txt
```

4. Execute the main Python script.

```
$ python3 run.py
```

5. After the measurements are finished, the raw data will be stored as CSV files in `cloudlab-tests/raw_data`.
6. To also recreate the plots, the provided plotting script can be used. The generated plots will be stored as PDF files in the `cloudlab-tests/plots` subdirectory.

```
$ python3 plot.py
```

Each individual test case consists of a server and a client, each defined in an SLA in the `cloudlab-tests/SLAs` directory. For each execution of a test case, the server is deployed first, followed by the client which automatically performs the measurement with the server. The results are then copied from the client container, parsed and stored as human-readable CSV files. Between deployments, the old containers are removed.

```
<rspec xmlns="http://www.geni.net/resources/rspec/3" xmlns:emulab="http://www.
    protogeni.net/resources/rspec/ext/emulab/1" xmlns:tour="http://www.
    protogeni.net/resources/rspec/ext/apt-tour/1" xmlns:jacks="http://www.
    protogeni.net/resources/rspec/ext/jacks/1" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://www.geni.net/resources/
    rspec/3 http://www.geni.net/resources/rspec/3/request.xsd" type="request">
    <node xmlns="http://www.geni.net/resources/rspec/3" client_id="node-1">
        <icon xmlns="http://www.protogeni.net/resources/rspec/ext/jacks/1" url="
            https://www.emulab.net/protogeni/jacks-stable/images/server.svg" />
        <routable_control_ip xmlns="http://www.protogeni.net/resources/rspec/ext
            /emulab/1" />
        <sliver_type xmlns="http://www.geni.net/resources/rspec/3" name="raw-pc
            ">
            <disk_image name="urn:publicid:IDN+utah.cloudlab.us+image+oakestra-
                PG0:1node" />
        </sliver_type>
        <services xmlns="http://www.geni.net/resources/rspec/3" />
        <hardware_type xmlns="http://www.geni.net/resources/rspec/3" name="xl170
            " />
    </node>
    <rspec_tour xmlns="http://www.protogeni.net/resources/rspec/ext/apt-tour/1">
        <description xmlns="" type="markdown">
            This profile was used to evaluate the proxy implementation of the
                Master Thesis "Leveraging eBPF in Orchestrated Edge
            Infrastructures".
        </description>
        <instructions xmlns="" type="markdown">
            - Setup Oakestra on the node. Make sure the NetManager supports eBPF.

            - Clone 'https://github.com/MrSarius/ma-complementary-resources.git'.

            - Run 'cloudlab-tests/run.py'. After successful completion, can be
                found in 'cloudlab-tests/raw_data'.
            - Run 'cloudlab-tests/Plot.py' to plot the results. The plots can be
                 found in 'cloudlab-tests/plots'.
        </instructions>
    </rspec_tour>
</rspec>
```

Figure A.1.: CloudLab Profile Used for the Measurements in This Thesis

List of Figures

List of Tables

Bibliography

[1] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing". In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 215–231. ISBN: 978-1-939133-35-9.

[2] T. A. Benson, P. Kannan, P. Gupta, B. Madhavan, K. S. Arora, J. Meng, M. Lau, A. Dhamija, R. Krishnamurthy, S. Sundaresan, N. Spring, and Y. Zhang. "NetEdit: An Orchestration Platform for eBPF Network Functions at Scale". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. ACM SIGCOMM '24. Sydney, NSW, Australia: Association for Computing Machinery, 2024, pp. 721–734. ISBN: 9798400706141. DOI: 10.1145/3651890.3672227.

[3] Cilium. *Cilium eBPF Go Package*. https://pkg.go.dev/github.com/cilium/ebpf. Accessed: 2024-08-31.

[4] *Cilium: eBPF-based Networking, Observability, and Security*. https://cilium.io/. Accessed: 2024-08-27. 2024.

[5] CloudLab. *CloudLab Node Type: xl170*. https://www.utah.cloudlab.us/apt/show-nodetype.php?type=xl170. Accessed: 2024-08-22. 2024.

[6] *Container Networking Interface (CNI)*. https://github.com/containernetworking/cni. Accessed: 2024-08-27.

[7] I. Corporation. *Data Plane Development Kit (DPDK)*. https://www.dpdk.org. Accessed: 2024-08-15. 2024.

[8] K. Documentation. *kube-proxy*. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. Accessed: 2024-08-28. 2024.

[9] L. K. Documentation. *eBPF - extended Berkeley Packet Filter*. https://www.kernel.org/doc/html/latest/bpf/index.html. Accessed: 2024-08-15.

[10] L. K. Documentation. *eBPF Maps*. https://www.kernel.org/doc/html/latest/bpf/maps.html. Accessed: 2024-08-15. 2024.

[11] L. K. Documentation. *eBPF Verifier*. https://www.kernel.org/doc/html/latest/bpf/verifier.html. Accessed: 2024-08-15. 2024.

[12] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14.

[13] B. Gill and D. Smith. *The Edge Completes the Cloud: A Gartner Trend Insight Report*. `https://www.gartner.com/`. Accessed: 2024-08-15. Sept. 2018.

[14] *Go Plugin Package*. `https://pkg.go.dev/plugin`. Accessed: 2024-08-12. 2024.

[15] M. Goudarzi, S. Ilager, and R. Buyya. "Cloud Computing and Internet of Things: Recent Trends and Directions". In: *New Frontiers in Cloud Computing and Internet of Things*. Ed. by R. Buyya, L. Garg, G. Fortino, and S. Misra. Cham: Springer International Publishing, 2022, pp. 3–29. ISBN: 978-3-031-05528-7. DOI: `10.1007/978-3-031-05528-7_1`.

[16] S. Ha, I. Rhee, and L. Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: `10.1145/1400097.1400105`.

[17] *iPerf: The TCP, UDP and SCTP network bandwidth measurement tool*. `https://iperf.fr/`. Accessed: 2024-08-23.

[18] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (2015), pp. 14–76. DOI: `10.1109/JPROC.2014.2371999`.

[19] *Kubernetes*. `https://kubernetes.io/`. Accessed: 2024-08-03. 2024.

[20] V. Kulkarni. *Performance Analysis of XDP-native, XDP-generic, and TC eBPF hooks*. `https://www.youtube.com/watch?v=nuXN5qSvDCw`. Accessed: 2024-08-31. 2023.

[21] Linux Kernel Organization. *Linux Perf Wiki*. `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 2024-08-22. 2024.

[22] *Linux Namespaces*. `https://man7.org/linux/man-pages/man7/namespaces.7.html`. Accessed: 2024-08-06. 2024.

[23] *Linux Traffic Control*. `https://man7.org/linux/man-pages/man8/tc.8.html`. Accessed: 2024-08-06. 2024.

[24] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu. "A Framework for eBPF-Based Network Functions in an Era of Microservices". In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 133–151. DOI: `10.1109/TNSM.2021.3055676`.

[25] P. Network. *Polycube Documentation*. https://polycube-network.readthedocs.io/en/latest/. Accessed: 2024-08-27. 2024.

[26] Oakestra. *Semantic Addressing*. https://www.oakestra.io/docs/networking/semantic-addressing. Accessed: 2024-08-21. 2024.

[27] C. Project. *eBPF Architecture - Just-In-Time Compilation*. https://docs.cilium.io/en/stable/bpf/architecture/#jit. Accessed: 2024-08-15. 2024.

[28] C. Project. *eBPF Architecture - Tail Calls*. https://docs.cilium.io/en/stable/bpf/architecture/#tail-calls. Accessed: 2024-08-15. 2024.

[29] C. Project. *eBPF Overview in Cilium*. https://docs.cilium.io/en/stable/bpf/. Accessed: 2024-08-15. 2024.

[30] B. Riegel. *eBPF-based Firewall implementation*. https://github.com/MrSarius/oakestra-net/tree/thesis-final-version/node-net-manager/ebpfManager/ebpf/firewall.

[31] B. Riegel. *eBPF-based Packet Counter implementation*. https://github.com/MrSarius/oakestra-net/tree/thesis-final-version/node-net-manager/ebpfManager/ebpf/packetCounter.

[32] B. Riegel. *eBPF-based Proxy implementation*. https://github.com/MrSarius/oakestra-net/tree/thesis-final-version/node-net-manager/ebpfManager/ebpf/proxy.

[33] B. Riegel. *ma-complementary-resources*. https://github.com/MrSarius/ma-complementary-resources.

[34] B. Riegel. *NetManager with eBPF Manager*. https://github.com/MrSarius/oakestra-net/tree/thesis-final-version/node-net-manager.

[35] P. Sabanic. *Supporting Unikernel-based Microservices at the Edge*. https://www.nitindermohan.com/documents/student-thesis/PatrickSabanicMT.pdf. Master's thesis, Technical University of Munich. 2022.

[36] Salvatore Sanfilippo. *hping3*. https://github.com/antirez/hping. Accessed: 2024-08-23. 2005.

[37] *TC eBPF*. https://man7.org/linux/man-pages/man8/tc-bpf.8.html. Accessed: 2024-08-06. 2024.

[38] *The Netfilter Project: iptables*. Accessed: 2024-08-25. netfilter.org. 2024.

[39] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, and A. Quinn. *bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions*. 2023. arXiv: 2311.07923 [cs.OS].