JOURNAL
COMPUTER SCIENCES INSTITUTE

JCSI 35 (2025) 150–158
Received: 19 January 2025
Accepted: 05 March 2025

The impact of using eBPF technology on the performance of networking solutions in a Kubernetes cluster

Wpływ wykorzystania technologii eBPF na wydajność rozwiązań sieciowych w klastrze Kubernetes

Konrad Miziński*, Sławomir Przyłucki

Department of Computer Science, Lublin University of Technology, Nadbystrzycka 36B, 20-618 Lublin, Poland

Abstract
The aim of this study was to investigate the impact of eBPF technology on the performance of network solutions in Kubernetes clusters. Two configurations were compared: a traditional iptables-based setup and eBPF based solution via the Cilium networking plugin. Performance tests were conducted, measuring throughput, latency, CPU usage, and memory consumption under unloaded and loaded conditions. The results indicate that the traditional configuration achieved higher throughput and lower latency in unloaded scenarios. However, under load, the eBPF-enabled cluster demonstrated advantages, including reduced CPU and memory usage and slightly improved latency. This study highlights the potential of eBPF as an efficient technology for Kubernetes environments, particularly in scenarios demanding high performance and resource efficiency.

Keywords: Kubernetes; Ebpf; Cilium; Cni; performance

Streszczenie
Celem badania było określenie wpływu technologii eBPF na wydajność rozwiązań sieciowych w klastrach Kubernetes. Porównano dwie konfiguracje: tradycyjną, opartą na iptables, oraz wykorzystującą technologię eBPF za pośrednictwem wtyczki sieciowej Cilium. Przeprowadzono testy wydajności, mierząc przepływność sieci, opóźnienia, zużycie procesora i pamięci w warunkach niskiego oraz wysokiego obciążenia. Wyniki wskazują, że w warunkach bez obciążenia tradycyjna konfiguracja osiągała wyższą przeływność i mniejsze opóźnienia. Jednak w środowisku obciążonym klaster wykorzystujący eBPF wykazał istotne korzyści, takie jak mniejsze zużycie procesora i pamięci oraz nieco niższe opóźnienia. Badanie to podkreśla potencjał eBPF jako efektywnej technologii dla środowisk Kubernetes, szczególnie w scenariuszach wymagających wysokiej wydajności i efektywnego wykorzystania zasobów.

Słowa kluczowe: Kubernetes; Ebpf; Cilium; Cni; wydajność

*Corresponding author
Email address: konrad.mizinski@pollub.edu.pl (K. Miziński)

1. Introduction

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. Kubernetes, over the years, has had a significant influence on the software industry, redefining how applications are developed, deployed and managed. Kubernetes quickly gained popularity because it simplified complex tasks like scaling and load balancing, allowing organizations to focus on developing new features and less on infrastructure management.

Networking in Kubernetes is critical in ensuring Pods (a group of one or more containers) can communicate with each other, with services within the cluster, and with external resources. This in-cluster communication is most of the time managed by a Container Network Interface (CNI) plugin, which provides routing across the cluster nodes.

The primary aim of this study is to investigate the impact of the Extended Berkeley Packet Filter (eBPF) technology on the network performance within Kubernetes clusters using Cilium as example CNI. This study seeks to quantify the performance improvements that eBPF can bring to Kubernetes networking, particularly in terms of network throughput, latency, CPU and RAM usage.

Hypotheses proposed in this work:

H1. The use of eBPF technology in a Kubernetes cluster significantly improves network performance compared to traditional network traffic management methods.

H2. Implementing eBPF in Kubernetes reduces system resource load (CPU, memory) due to more efficient packet processing.

2. Extended Berkley Packet Filtering (eBPF)

eBPF is a technology that enables execution of user-defined programs at various kernel hooks. These programs can dynamically extend kernel functionality without requiring changes to kernel source code, loading additional modules or any system reboots. eBPF programs are written in a restricted subset of C, compiled into bytecode, and then verified for safety by eBPF verifier built into the kernel. Once verified, they are attached to specific kernel events, such as system calls, network events or tracepoints and executed in response to those events. This

allows developers to gather insights and take actions directly within the kernel with minimal performance overhead. Originally designed for network packet filtering, eBPF has evolved into a versatile framework allowing performance monitoring, security enforcement, and network observability.

3. Related works

Several studies have focused on comparing the network performance of eBPF with traditional iptables-based approach. Articles [1-4] explores the foundational aspects of eBPF, highlighting its potential to replace traditional iptables with more efficient packet processing capabilities. The studies emphasize the flexibility and performance benefits that eBPF offers over conventional methods.

Articles [5-9] examine the impact of different Container Network Interface (CNI) plugins on network performance and scalability within Kubernetes cluster. By measuring different performance metrics such as bandwidth, latency, CPU usage, they identify which CNI extenstions are most suited to performance oriented applications. These articles collectively highlight the importance of selection appropriate CNI plugins based on specific workload requirements. They underscore that efficient CNI choices are crucial for maintaining network reliability and responsiveness in high-demand scenarios like edge computing and data-intensive applications.

Artciles [10-12] highlight eBPF's role in advancing network observability and monitoring capabilities in Kubernetes. eBPF-based approaches provide detailed, protocol-independent network insights with minimal disruption to application performance. Researchers in this area emphasize the adaptability of eBPF-based monitoring solutions, which can accommodate diverse Kubernetes configurations while maintaining low resource consumption. These studies suppors eBPF as a robust tool for achieving high-fidelity network observability, which is especially valuable in complex cloud-native environments like those required by 5G and emerging telecommunications networks

Articles [13-15] emphasize eBPF's flexibility for enhancing and extending Kubernetes' existing networking infrastructure. Collectively, these studies showcase how eBPF can incrementally add network functionality such as security filtering, and socket acceleration without requiring extensive infrastructure overhauls. Together, these studies position eBPF as a strategic technology for evolving Kubernetes networking, especially in environments that demand high security, low latency, and high customizability, such as cloud-native 5G networks.

4. Research methods

The main subject matter of this research are two Kubernetes clusters having the Cilium CNI installed. The selection of Cilium was primarily driven by several key factors, with the most important one being the maturity of its eBPF technology implementation, as well as the wide range of configuration options available. In terms of these criteria, Cilium is widely known as a leading solution, offering the highest degree of flexibility and functionality compared to other available CNIs.

The first of the two clusters has Cilium configured in such a way that it does not make use of the eBPF technology. Instead, it relies on more traditional approach for handling network traffic, such as routing and packet filtering. These traditional methods are based on the native mechanisms that are built into the Linux kernel. Specifically, the first cluster utilizes Linux's Netfilter framework in combination with iptables, a widely used frontend for packet filtering and firewall management. Iptables is directly managed by a Kubernetes component called kube-proxy. Kube-proxy runs on every Kubernetes node, and its role is to ensure communication between nodes within the cluster.

In the second cluster, Cilium has been configured to maximize the utilization of eBPF technology. This configuration is designed to exploit the capabilities of eBPF for managing routing and packet filtering processes between the various objects or pods within the Kubernetes cluster. Moreover, by utilizing eBPF technology, Cilium is able to eliminate the need for kube-proxy, which reduces the number of components in the network path.

It should be noted that both clusters are configured to be identical in every respect, except in terms how Cilium is implemented. This setup allows for a direct comparison of the two approaches, providing valuable insights into the benefits and trade-offs of using eBPF for network management in Kubernetes environments.

To investigate the impact of eBPF technology on the performance of network solutions in a Kubernetes cluster, an experimental approach was used based on several key steps:

- Literature Review: The goal of this stage was to identify studies, best practices, and potential research gaps in the existing scientific literature dedicated to the topic of eBPF technology and its use in Kubernetes clusters.
- Test Environment Configuration: Two Kubernetes clusters were created. One cluster based on the standard networking approach, one fully utilizing eBPF technology. Each cluster consisted of three nodes one master node and two worker nodes.
- Performance Testing: Experiments with different scenarios were conducted under controlled conditions, allowing precise performance measurements.
- Empirical Data Analysis: Data on network performance such as throughput, latency, CPU, memory usage were collected. This data came from system monitoring and network analysis tools.

5. Testbed components

In order to conduct performance tests of the Cilium network extension, it was decided to utilize several tools, with each tool being selected based on the specific metrics that needed to be gathered. The tools that were chosen for this purpose include the following:

- Iperf3 [16] – Command Line Interface (CLI) tools that serves the purpose of measuring the speed of data transfer between Pods that are located on different nodes within a Kubernetes cluster. This tool helps in evaluating network throughput and performance by generating traffic between the nodes.
- Prometheus [17] – software application designed for the collection of metrics, which includes, among others, data on CPU usage and memory (RAM) consumption by the nodes in a Kubernetes cluster. Prometheus acts as a monitoring solution that scrapes and stores these metrics, allowing for detailed analysis over time.
- Prometheus node exporter [18] – open-source tool that helps monitor system metrics and make them available for collection by Prometheus
- Grafana [19] – visualization tool that is used for creating graphical representations, such as charts and graphs, based on the metrics collected by Prometheus. Grafana allows to build dashboards that provide real-time insights into the system's performance and resource usage.
- Sockperf [20] – network benchmaring tool used for testing latency between Kubernetes Pods
- Postman [21] – used for generating requests loading clusters

These tools collectively form the basis for the performance testing and monitoring process in this study, each playing a crucial role in the analysis and visualization of network and system metrics.

5.1. Research stand

In order to minimize the discrepancies in the results obtained, both clusters were established based on KVM/QEMU virtual machines on three Proxmox hypervisors running version 8.2.2.

5.1.1.Physical servers

Each of the hypervisors is a physical HP ProLiant BL460c Gen9 server with the following specifications:

- Two Intel(R) Xeon(R) CPU E5-2680 v4 processors operating at a clock speed of 2.40 GHz, providing substantial computing power for processing tasks and managing virtual environments.
- An HP FlexFabric 10Gb 2-port 536FLB Adapter network card, configured with Link Aggregation Control Protocol (LACP) to enhance bandwidth and provide redundancy, ensuring stable and efficient network performance.
- A Smart Array P244br Controller managing storage, equipped with two SAMSUNG SSD PM893 drives, each with a capacity of 1920 GB and utilizing the SATA 3.2 interface, configured in RAID 1 mode to provide data redundancy and improved fault tolerance.
- A total of 256 GB of RAM, comprised of 16 HP DIMM DDR4 2133 MHz modules, each with a capacity of 16 GB and featuring ECC (Error-Correcting Code) for enhanced reliability in memory operations.

5.1.2.Virtual machines

Each Kubernetes cluster consists of three virtual machines. One of these machines serves as the master node, responsible for ensuring the fundamental mechanisms that constitute the Kubernetes cluster, while the other two function as worker nodes, where performance test pods are deployed. Depending on the intended purpose of the virtual machine, the allocation of resources has been appropriately adjusted. Regardless of the role of each virtual machine within the cluster, the following configuration settings are consistent across all virtual machines:

- Utilization of KVM/QEMU virtualization technology, which provides efficient and flexible virtualization capabilities.
- Operating system: Ubuntu 22.04 LTS, chosen for its stability and robust support for enterprise applications.
- Processor virtualization configured to Host Model, allowing optimal performance by presenting the virtual machine with an accurate representation of the host CPU features.
- BIOS type set to SeaBIOS, a widely used open-source BIOS implementation that facilitates virtual machine boot processes.
- Virtualized chipset type: i440fx, ensuring compatibility and efficient emulation of the hardware environment.
- Virtual machine disk controller: VirtIO SCSI Single, which enhances disk performance by providing a paravirtualized interface for the virtual disks.
- Network virtualization: VirtIO, with a configured bandwidth limit of 1250 MBps (equivalent to 10 Gbps), an MTU (Maximum Transmission Unit) size of 1500 bytes, firewall configuration disabled on the master node side to ensure streamlined communication, and frame tagging enabled for better network management

For the machines designated as master nodes, the following resource limits have been established:

- 4 CPU cores, enabling the master node to efficiently manage and orchestrate tasks within the cluster.
- 8 GB of RAM, providing sufficient memory for the master node to handle control plane operations and resource management.
- 30 GB of disk space allocated for the file system, ensuring adequate storage for system-related files and configurations.

In the context of the worker nodes, the allocated resource limits are outlined below, ensuring that each node has the necessary resources to perform optimally and handle the demands of the workloads efficiently:

- 10 CPU cores, allowing the worker nodes to perform intensive computational tasks and run multiple pods simultaneously.
- 20 GB of RAM, ensuring that each worker node has sufficient memory to execute performance tests and manage workloads effectively.

- 50 GB of disk space allocated for the file system, providing ample storage for the applications and data processed by the worker nodes.

5.1.3.Kubernetes clusters

The Kubernetes clusters have been created based on the RKE2 distribution, version v1.26.15+rke2r1 with Cilium CNI in version 1.15.1.

Listing 1: Configuration of RKE2 cluster that does not employ eBPF technology

```
server: https://172.19.12.23:9345
token: DH1buBx9rGSHm7VuSUiDx5k9rSn5wgcAzZkE2NJ0JsDPxtdSrBH4VjxdQp0pYYtL
data-dir: /var/lib/rancher/rke2
cni: cilium
tls-san:
  - cluster.local
  - 172.19.12.23
  - kubcilium.vrrp.k8s.dc
disable: ['rke2-canal', 'rke2-ingress-nginx', 'rke2-snapshot-controller', 'rke2-snapshot-controller-crd', 'rke2-snapshot-validation-webhook']
snapshotter: overlayfs
node-name: kubciliumn01
node-ip: 172.19.12.94
kubelet-arg:
  - "container-log-max-files=3"
  - "container-log-max-size=1G"
  - "image-gc-high-threshold=80"
  - "image-gc-low-threshold=60"
```

The configuration options for RKE2 cluster that does not employ eBPF technology are presented on Listing 1.

Listing 2: Configuration of Cilium that does not employ eBPF technology

```
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-cilium
  namespace: kube-system
spec:
  valuesContent: |-
    operator:
      replicas: 1
      prometheus:
        enabled: true
      nodeSelector:
        kubernetes.io/hostname: kubciliumn01
    ipam:
      mode: "cluster-pool"
      operator:
        clusterPoolIPv4PodCIDRList:
          - 10.42.0.0/16
    ipv4NativeRoutingCIDR: "10.42.0.0/16"
    routingMode: "native"
    tunnelProtocol: ""
    autoDirectNodeRoutes: true
    prometheus:
      enabled: true
    hubble:
      enabled: true
      metrics:
        enabled:
        - dns
        - drop
        - tcp
        - flow
        - port-distribution
        - icmp
        - httpV2
      relay:
        enabled: false
```

In the case of the configuration for Cilium, it is presented on Listing 2.

Listing 3: Configuration of RKE2 cluster that does employ eBPF technology

```
server: https://172.19.12.24:9345
token: tKR6B2tHuXjvBEf5jAOhNHGlv09WWmWrYGgFyEAp5DedQcfbIyGaPeW5LmaULDzy
data-dir: /var/lib/rancher/rke2
cni: cilium
tls-san:
  - cluster.local
  - 172.19.12.24
  - kubcilium-ebpf.vrrp.k8s.dc
disable: ['rke2-canal', 'rke2-ingress-nginx', 'rke2-snapshot-controller', 'rke2-snapshot-controller-crd', 'rke2-snapshot-validation-webhook']
disable-kube-proxy: true
snapshotter: overlayfs
node-name: kubcilium-ebpfn01
node-ip: 172.19.12.156
kubelet-arg:
  - "container-log-max-files=3"
  - "container-log-max-size=1G"
  - "image-gc-high-threshold=80"
  - "image-gc-low-threshold=60"
```

In contrast, for the cluster utilizing eBPF technology, the configuration of RKE2 cluster is displayed in Listing 3.

Listing 4: Configuration of Cilium that does employ eBPF technology

```
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-cilium
  namespace: kube-system
spec:
  valuesContent: |-
    k8sServiceHost: "172.19.12.24"
    k8sServicePort: "6443"
    operator:
      replicas: 1
      prometheus:
        enabled: true
      nodeSelector:
        kubernetes.io/hostname: kubcilium-ebpfn01
    ipam:
      mode: "cluster-pool"
      operator:
        clusterPoolIPv4PodCIDRList:
          - 10.42.0.0/16
    ipv4NativeRoutingCIDR: "10.42.0.0/16"
    socketLB:
      hostNamespaceOnly: true
    kubeProxyReplacement: "true"
    routingMode: "native"
    tunnelProtocol: ""
    autoDirectNodeRoutes: true
    loadBalancer:
      mode: dsr
    bpf:
      tproxy: true
      masquerade: true
    ipMasqAgent:
      enabled: true
      config:
        nonMasqueradeCIDRs: ["10.42.0.0/16"]
    loadbalancer:
      acceleration: true
    hubble:
      enabled: true
      metrics:
        enabled:
        - dns:query;ignoreAAAA
        - drop
        - tcp
        - flow
        - icmp
        - http
      relay:
        enabled: false
```

When it comes to the configuration of Cilium, the details are expressed on Listing 4.

The primary distinction between the configurations of the clusters lies in the fact that the cluster utilizing eBPF does not employ kube-proxy. Instead, Cilium leverages eBPF technology to handle packet routing. This configuration is achieved by setting the option "disable-kube-proxy: true" in the RKE2 configuration, along with specifying "k8sServiceHost" and "k8sServicePort" in the Cilium configuration. Furthermore, the "kubeProxyReplacement: true" option in Cilium facilitates the replacement of kube-proxy functionality, enabling Cilium to take over the responsibilities of service routing and load balancing directly through eBPF. Several additional options further optimize the performance of eBPF within the cluster. The "loadBalancer.mode: dsr" (Direct Server Return) option optimizes traffic handling by allowing responses from backend pods to be sent directly to clients, bypassing the load balancer. This approach minimizes latency and improves throughput. The "bpf.tproxy" option enables transparent proxying, allowing for more efficient packet handling without modifications to the original packet flow. Lastly, the "loadbalancer.acceleration: true" option boosts performance by allowing Cilium to utilize optimized load balancing mechanisms such as attaching eBPF programs directly to paravirtualized Network Interface Cards (NICs), reducing overhead and improving the overall responsiveness of the cluster.

6. Tests Scenarios

There developed were four scenarios to compare various aspects of network performance in a Kubernetes cluster using eBPF versus the standard iptables-based network approach:

- Network Throughput Comparison: Load tests were conducted to measure network throughput between Pods in a Kubernetes cluster with and without eBPF.
- Network Latency Analysis: Delays in data transmission were examined in the Kubernetes cluster with and without eBPF.
- Resource Usage Impact: The effect of eBPF on system resource usage, such as CPU and memory, was asessed. Resource usage was compared during normal cluster operation and intensive network operations.

All results from the tests conducted according to the above scenarios were recorded and subsequently analyzed in the following stages to draw conclusions and determine the impact of eBPF technology on the performance of network solutions in Kubernetes clusters. To effectively evaluate the performance of both Kubernetes clusters, it is essential to monitor several key metrics while executing test scenarios. These metrics provide insights into the operational efficiency, responsiveness, and resource utilization of the clusters. Depending on the test scenarios appropriate metrics were used for data collection.

6.1. Metrics

The primary metrics considered in this paper include:

- throughput,
- latency,
- CPU utilization,
- RAM consumption.

6.1.1.Throughput

The base metric used for throughput evaluation was *node_network_receive_bytes_total* provided by Prometheus node exporter. Using this metric a new formula was created.

$$irate(node\_network\_receive\_bytes\_total\{ instance = \sim 172|\,|.19|\,|.12|\,|.95{:}9100, device = \sim "(?i)\hat{}(ens|eth).+\$"\}[1m]) \quad (1)$$

Formula **(1)** by using **irate** function calculates average number of received bytes per network interface.

6.1.2.Latency

Latency was measured using sockperf utility in ping-pong mode.

6.1.3.CPU Utilization

In order to measure CPU Utilization *node_cpus_seconds_total* was chosen.

$$100 - 100 * (avg(irate(node\_cpu\_seconds\_total\{mode="idle"\}[1m])) by (instance)) \quad (2)$$

CPU utilization was calculated using (2) formula. By calculating average rate of $node\_cpu\_seconds\_total$ metric in idle mode and then subtracting it from 100 average total CPU usage was estimated.

6.1.4.RAM consumption

RAM consumption is based on two complementary metrics *node_memory_memTotal_bytes* and *node_memory_memAvailable_bytes.*

$$(node\_memory\_MemTotal\_bytes\{instance = \sim "172\backslash\backslash.19\backslash\backslash.12\backslash\backslash.95{:}9100" - node\_memory\_MemAvailable\_bytes\{instance"172\backslash\backslash.19\backslash\backslash.12\backslash\backslash.95{:}9100"\}) / 1024 / 1024 \quad (3)$$

By subtracting $node\_memory\_MemTotal\_bytes$ from $node\_memory\_MemAvailable\_bytes$ and diving by 1024 two times in formula (3), RAM consumption in megabytes was assessed.

6.2. Experiments

The goal of the experiment is to evaluate the performance metrics of Kubernetes clusters by measuring the baseline throughput, latency, CPU usage, and RAM consumption under both loaded and unloaded conditions. This will

provide a comprehensive understanding of how these metrics behave in a controlled environment, enabling to draw meaningful conclusions about the performance and efficiency of the clusters.

6.2.1. Throughput measurment

To measure throughput, tests will be initiated using iperf3 running inside Pods on two worker nodes within the cluster. The experiment will be structured as follows:

- Test Initialization: The iperf3 tool will be deployed across two separate worker nodes inside Pods in the cluster.
- Repetition for Reliability: Each test will be executed five times for five minutes to mitigate the risk of local anomalies that could skew results. This repetition is crucial for obtaining consistent and reliable data, accounting for any transient network fluctuations or irregularities during the tests.
- Data Collection and Averaging: After conducting five tests, the throughput data generated will be collected. This data will then be averaged to produce a single representative value of the TCP throughput. The average will offer insight into the performance of the Kubernetes cluster.

6.2.2. Latency measurment

To measure latency, tests will be initiated using sockperf utility running inside Pods on two worker nodes within the cluster:

- Test Initialization: The same two Pods will be utilized to run latency tests, ensuring that the testing environment remains consistent and controlled.
- Repetition for Reliability: Latency measurements will be conducted five times, similar to the throughput tests. Each iteration will measure the time taken for packets to travel between the two Pods running on two different nodes, capturing the inherent latency in connections.
- Data Collection and Averaging: After completing all tests, the recorded latency data will be averaged to provide a latency metric.

6.2.3.CPU usage measurment

The third research scenario focuses on measuring CPU usage:

- Test Initialization: A series of tests will be executed, akin to those for throughput and latency, this time focusing on CPU consumption metrics. Prometheus node exporter in conjunction with Prometheus and Grafana will be employed to gather detailed data on CPU usage.
- Repetition for Reliability: Each test will again be repeated five times for five minutes to ensure data reliability.
- Average Calculation: After all tests are completed, the CPU usage data will be aggregated and averaged to determine the baseline CPU consumption of the k8s cluster during idle conditions.

6.2.4. RAM Usage measurment

The fourth research scenario focuses on measuring RAM usage:

- Test Initialization: The RAM usage tests will be structured similarly to the CPU usage tests, involving the same repetition strategy. Prometheus node exporter, Prometheus and Grafana will be utilized to gather comprehensive data on memory consumption.
- Repetition for Reliability: Each RAM usage test will also be performed five times to ensure that the data collected is reliable and representative.
- Average Calculation: Once all tests are complete, the collected data will be averaged to derive a RAM usage figure.

7. Test results

The research scenarios were conducted in two separate phases to ensure comprehensive analysis and robust data collection. During the initial phase, the clusters operated under unloaded conditions, allowing for a baseline assessment of their performanc. In contrast, the second phase introduced a controlled workload on the cluster. The Guestbook application [22] was deployed on the cluster and subjected to load testing using the Postman application, which generated requests for both writing and reading data. Postman utilized the Collection Runner to execute these requests. A Performance-type Collection Runner was used, simulating 100 virtual users. The primary objective of these actions was to create a realistic simulation of a "production" cluster under heavy load. By systematically comparing the performance of the unencumbered cluster to that of the load-simulated environment, a valuable insight into the operational dynamics and efficiency of the Cilium network extension was gained.

7.1. First Phase - Idle clusters

The test results for the cluster in an idle state not utilizing eBPF technology are presented in Table 1.

Table 1. Test results for an unloaded cluster without eBPF

Test number	Throughput (Mbps)	Latency (µs)	CPU Usage (%)	RAM Usage (MB)
1	7660.57	488.859	19.21	1256.90
2	7757.71	486.943	19.30	1259.47
3	7867.42	486.655	19.24	1261.90
4	7864.38	481.298	19.24	1264.80
5	8104.38	485.396	19.41	1258.00
Avg	7850.89	485.830	19.28	1260.21
Std Dev	165.54	2.821	0.08	3.17

The test results for the cluster in an idle state utilizing eBPF technology are presented in Table 2.

Table 2. Test results for an unloaded cluster with eBPF

Test number	Through put (Mbps)	Latency (µs)	CPU Usage (%)	RAM Usage (MB)
1	6840.76	504.884	19.28	1248.38
2	6901.81	500.168	19.58	1248.72
3	7176.38	501.531	19.14	1251.90
4	7186.28	502.710	19.07	1257.14
5	7252.80	505.814	19.20	1263.90
Avg	7071.61	503.021	19.25	1254.01
Std Dev	186.47	2.33	0.20	6.55

In the unloaded state, where cluster resources were not heavily utilized, notable differences were observed between the two configurations. The traditional setup without eBPF demonstrated higher average throughput, achieving approximately 7850.89 (Table 1.) Mbps compared to 7071.61 (Table 2.) Mbps in the eBPF-enabled configuration. This indicates that, in an unloaded environment, the traditional iptables-based setup may have a slight edge in terms of raw throughput. However, this difference might be attributed to the additional processing overhead introduced by eBPF, as it executes directly within the kernel to manage packet processing more dynamically.

Latency results, reveal that the traditional setup had a slight advantage, with an average latency of 485.830 µs, while the eBPF-enabled setup exhibited a slightly higher latency of approximately 503.021 µs. Although the difference in latency is minimal, it suggests that the traditional approach might provide slightly faster response times under low-load conditions.

In terms of resource usage, the CPU and RAM metrics show contrasting results. The eBPF-enabled configuration displayed marginally lower CPU usage at 19.25% on average compared to 19.28% in the traditional configuration, likely due to less context switching between user and kernel space. eBPF also proved to be more efficient in terms of memory usage, with an average RAM consumption of 1254.01 MB compared to 1260.21 MB for the traditional setup. This suggests that while eBPF reduces network performance, it offers a more CPU and memory-efficient solution.

7.2. Second Phase - Loaded clusters

The test results for the cluster in a loaded state not utilizing eBPF technology are presented in Table 3.

Table 3. Test results for a loaded cluster without eBPF

Test number	Through put (Mbps)	Latency (µs)	CPU Usage (%)	RAM Usage (MB)
1	6688.00	546.410	47.62	1991.70
2	6797.09	514.680	46.20	1996.63
3	6801.90	548.652	45.10	1971.85
4	6986.90	544.042	45.47	1980.22
5	6660.19	552.555	44.75	1970.47
Avg	6786.82	541.268	45.83	1982.17
Std Dev	128.61	15.190	1.14	11.70

The test results for the cluster in a loaded state utilizing eBPF technology are presented in Table 3.

Table 4. Rest results for a loaded cluster with eBPF

Test number	Through put (Mbps)	Latency (µs)	CPU Usage (%)	RAM Usage (MB)
1	6412.19	535.215	41.22	1768.00
2	6344.00	488.831	40.19	1772.63
3	6556.36	511.912	41.21	1775.63
4	6631.27	505.682	40.81	1761.90
5	6352.72	516.650	41.25	1771.54
Avg	6459.31	511.658	40.94	1769.94
Std Dev	128.32	16.858	0.45	5.26

In the loaded state, the differences between the traditional setup and the eBPF-enabled configuration became more pronounced. The results highlighted the trade-offs between throughput, latency, and resource usage under high load.

The traditional cluster configuration without eBPF demonstrated higher average throughput, achieving approximately 6786.82 Mbps (Table 3.), compared to 6459.31 (Table 4.) Mbps for the eBPF-enabled setup. This suggests that under load, the traditional iptables-based networking may still maintain slightly better raw data transfer rates, possibly due to the overhead introduced by eBPF's dynamic kernel processing.

Latency measurements showed a mixed outcome. While the eBPF-enabled cluster had a lower average latency of 511.658 µs, compared to 541.268 µs for the traditional setup, the reduction in latency was relatively small. This implies that eBPF offers some improvements in responsiveness during high-load scenarios, which could be beneficial for latency-sensitive applications.

Resource usage exhibited clear advantages for the eBPF-enabled setup. CPU usage was notably lower in the eBPF-enabled cluster, averaging 40.94%, compared to 45.83% in the traditional cluster. This highlights the efficiency of eBPF in minimizing user-to-kernel space transitions and reducing the computational load during network processing. Additionally, RAM consumption in the eBPF setup was significantly lower, averaging 1769.94 MB compared to 1982.17 MB in the traditional setup. This reduction in memory usage emphasizes eBPF's efficiency in handling high-load conditions without exhausting system resources.

Overall, the results demonstrate that while eBPF may slightly reduce throughput under load, it provides improvements in latency and resource utilization, making it a compelling choice for environments where efficiency and responsiveness are critical.

7.3. Discussion of the Results

The results of the experiments, conducted under both unloaded and loaded states, highlight the comparative performance of Kubernetes clusters with and without eBPF technology. In the unloaded state, the traditional cluster achieved slightly higher throughput and lower latency.

These differences suggest that the traditional setup provides marginally better raw performance in low-load conditions. However, the eBPF cluster showed advantages in resource efficiency, with lower average CPU and memory usage, suggesting a reduced system load when idle.

In the loaded state, the traditional cluster continued to exhibit higher throughput compared to eBPF cluster. Latency results were slightly reversed, with the eBPF-enabled configuration demonstrating lower latencies. Resource usage differences became more pronounced under load, as the eBPF cluster consumed significantly less CPU and memory compared to traditional setup. These findings indicate that while the traditional approach maintains an edge in throughput, eBPF offers improved latency under load and significantly better resource utilization, making it particularly effective in high-demand environments.

8. Conclusions

The aim of this study was to examine the impact of using eBPF technology on network performance within Kubernetes clusters, specifically assessing whether eBPF can enhance networking efficiency and resource usage. To address this objective, two main hypotheses were formulated H1 and H2.

The results of the experiments partially support these hypotheses. In the unloaded cluster, the traditional iptables-based configuration demonstrated slightly higher throughput and lower latency compared to the eBPF-enabled cluster. This suggests that traditional methods may retain an advantage in raw performance under minimal load conditions, potentially due to the additional kernel-level processing introduced by eBPF. However, under loaded conditions, the advantages of eBPF technology became more evident. The eBPF-enabled cluster exhibited improved resource efficiency, with significantly lower CPU and memory usage, and showed slightly better latency compared to the traditional configuration, despite a marginal reduction in throughput. These findings indicate that while traditional setups excel in low-demand scenarios, eBPF provides substantial benefits under high-demand conditions by efficiently managing system resources and maintaining competitive network performance.

These results are consistent with findings in prior studies. For instance, article [1] highlighted eBPF's superior efficiency in packet handling and its ability to maintain stable performance under load, which aligns with this study's observations of reduced resource usage in eBPF-enabled clusters. Similarly, [3] demonstrated latency improvements with eBPF in service mesh environments, supporting this study's findings of reduced latency under load in eBPF-enabled Kubernetes clusters. Additionally, [4] confirmed eBPF's resource-saving potential in packet filtering tasks, further corroborating the reduced CPU and memory usage observed in this study.

In conclusion, this study reinforces the potential of eBPF technology to enhance networking efficiency and resource usage in Kubernetes clusters, particularly under high-load scenarios. While traditional methods may outperform eBPF in unloaded environments, the resource efficiency and latency improvements offered by eBPF make it a compelling choice for resource-intensive and latency-sensitive applications. These findings align with and extend existing research, providing a strong case for adopting eBPF in Kubernetes networking solutions to meet the demands of modern distributed systems.

References

[1] N. de Bruijn, eBPF Based Networking, Master thesis, University of Amsterdam, Amsterdam, 2017.

[2] M. Tran, Y. Kim, Network Performance Benchmarking for Containerized Infrastructure in NFV environment, In 2022 IEEE 8th International Conference on Network Softwarization (NetSoft) (2022) 115–120 https://doi.org/10.1109/NetSoft54395.2022.9844100.

[3] W. Yang, Pe. Chen, G. Yu, Ha. Zhang, Hu. Zhang, Network shortcut in data plane of service mesh with eBPF, Journal of Network and Computer Applications 222 (2024) 103805, https://doi.org/10.1016/j.jnca.2023.103805.

[4] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, G. Carle, Performance Implications of Packet Filtering with Linux eBPF, In 2018 30th International Teletraffic Congress (ITC 30) (2018) 209–217, https://doi.org/10.1109/ITC30.2018.00039.

[5] T. P. Nagendra, R. Hemavathy, Unlocking Kubernetes Networking Efficiency: Exploring Data Processing Units for Offloading and Enhancing Container Network Interfaces, In 2023 4th IEEE Global Conference for Advancement in Technology (GCAT) (2023) 1–7, https://doi.org/10.1109/GCAT59970.2023.10353542.

[6] A. Gokhale, Z. Kang, K. An, P. Pazandak, A Comprehensive Performance Evaluation of Different Kubernetes CNI Plugins for Edge-based and Containerized Publish/Subscribe Applications, In 2021 IEEE International Conference on Cloud Engineering (IC2E) (2021) 31–41, https://doi.org/10.1109/IC2E52221.2021.00017.

[7] S. Sekigawa, C. Sasaki, A. Tagami, Toward a Cloud-Native Telecom Infrastructure: Analysis and Evaluations of Kubernetes Networking, In 2022 IEEE Globecom Workshops (GC Wkshps) (2023) 838–843, https://doi.org/10.1109/GCWkshps56602.2022.10008579.

[8] S. Qi, S. G. Kulkarni, K. K. Ramakrishnan, Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability, In IEEE Transactions on Network and Service Management (2020) 656–671, https://doi.org/10.1109/TNSM.2020.3047545.

[9] Y. Park, H. Yang, Y. Kim, Performance Analysis of CNI (Container Networking Interface) based Container Network, In 2018 International Conference on Information and Communication Technology Convergence (ICTC) (2018) 248–250, https://doi.org/10.1109/ICTC.2018.8539382.

[10] T. D. Zavarella, A methodology for using eBPF to efficiently monitor network behavior in Linux Kubernetes clusters, partial fulfillment of the requirements for the degree of Master, Massachusetts Institute of Technology, Massachusetts, 2021.

[11] C. Liu, Z. Cai, B. Wang, Z. Tang, J. Liu, A protocol-independent container network observability analysis system based on eBPF, In 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS) (2021) 697–702, https://doi.org/10.1109/ICPADS51040.2020.00099.

[12] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. F. Soliman, L. Di Giovanna, F. Monaco, G. Ognibene, F. Risso, eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond), IEEE Access (11) (2023) 57174–57202, https://doi.org/10.1109/ACCESS.2023.3281480.

[13] H. Sharaf, I. Ahmad, T. Dimitriou, Extended Berkeley Packet Filter: An Application Perspective, IEEE Access (10) (2022) 126370–126393, https://doi.org/10.1109/ACCESS.2022.3226269.

[14] D. Behl, H. Huang, P. Kodeswaran, S. Sen, On eBPF extensions to Kubernetes CNI datapath, In 2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS) (2023) 207–209, https://doi.org/10.1109/COMSNETS56262.2023.10041357.

[15] V. Duong, Y. Kim, A Design of Service Mesh Based 5G Core Network Using Cilium, In 2023 International Conference on Information Networking (ICOIN) IEEE (2023) 25–28, https://doi.org/10.1109/ICOIN56518.2023.10049044.

[16] A TCP, UDP, and SCTP network bandwidth measurement tool, https://github.com/esnet/iperf, [27.12.2024].

[17] A systems and service monitoring system, https://github.com/prometheus/prometheus, [27.12.2024].

[18] Exporter for hardware and OS metrics exposed, https://github.com/prometheus/node_exporter, [27.12.2024].

[19] A platform for monitoring and observability, https://github.com/grafana/grafana, [27.12.2024].

[20] A network benchmarking utility, https://github.com/Mellanox/sockperf, [27.12.2024].

[21] A platform for testing and developing APIs, https://www.postman.com, [27.12.2024].