Evaluation of traditional and eBPF-based packet processing in Kubernetes for network slicing

Ákos Leiter[†], Döme Matusovits[‡†], and László Bokor[‡§]

***Abstract*—In recent years, the proliferation of cloud-native technology enablers, such as microservice deployment and management with Kubernetes, have presented new challenges for telecommunications service providers. Strict data transmission requirements have emerged in various areas, such as immediate interventions in intelligent transportation, video conferencing, etc. With the advent of 5G networks, this demand can also be fulfilled thanks to an innovative technology called Network Slicing. In terms of its operation, we can separate networks into individual segments to continuously satisfy the desired service requirements. However, packet processing on top of Kubernetes may need to be changed to support the emerging number of microservices during slicing. This is where the Extended Berkeley Packet Filter (eBPF) comes into the picture to boost the capacity of data centers and keep service guarantees. This paper presents how eBPF can support network slicing through its performance evaluation in a Kubernetes environment.**

***Index Terms*—network slicing, eBPF, Kubernetes, 5G/6G mobile cellular architectures, cloud-native applications**

I. Introduction

Implementing end-to-end (E2E) network slicing is still a heavily researched problem in the telco industry. It is impossible to properly fulfill E2E network slicing requirements if one segment or domain of the network does not deal with service guarantees. For example, if the core network part of the E2E network slice instance has proper resource assignment and implementation, other network parts have to act similarly. Radio, transport, and data center networking must also be prepared for network slicing requirements. The end-to-end network slicing concept is depicted in Figure 1. All the network function elements are considered to run in the cloud environment.

In this work, we focus on data center networking, especially Kubernetes-based packet processing solutions, and whether or not they can ensure a particular service level that requires low latency and a guaranteed throughput during packet traversal. In the fifth-generation mobile network standards, communication with these constraints is called Ultra-reliable Low-latency Communication service (URLLC) [1]. We assume that the number of microservices (Kubernetes Services) will be continuously increasing (due to autoscaling, edge deployments, and further radio cloudification and decoupling [2] [3] [4] [5]), so we examine how this will affect network Key Performance Indicators (KPI) considering throughput and latency. We evaluate this on two test environments: the traditional Kubernetes packet processing method kube-proxy-based on the top of iptables and the extended Berkley Packet Filter (eBPF) [6] using the Cilium Container Networking Interface (CNI) [7] in the context of network slicing.

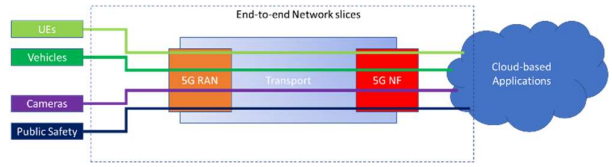

Figure 1 – End-to-end network slicing concept [8]

eBPF is powerful because it makes the Linux operating system programmable without modifying the kernel's source code or writing a new kernel module. Consequently, it also solves the complication of developing a monolithic Kernel, which saves much time when adding new features to the OS core. Furthermore, it provides an alternative to low-performed Netfilter-based packet processing. This is why Kubernetes has started to utilize eBPF in CNIs, which are responsible for Kubernetes' internal and external networking. This includes interface and IP address management and packet processing mechanisms. There are two CNIs publicly available that implement eBPF-based networking: Cilium and Calico [9]. Our test system relies on the Cilium-based solution.

This paper is organized as follows: Section II gives a technological introduction to iptables and eBPF. Then, Section III covers the most crucial technological background of Kubernetes and Cilium. The related work is presented in Section IV. The testbed details are explained in Section V, while measurement results are elaborated in Section VI. Conclusion and future work are drawn in Sections VII and VIII, respectively.

II. Background behind iptables and Extended Berkeley Packet Filter

Although in the Linux world, packet processing/filtering technologies are already available (such as nftables [10]), which can enhance the traditional Netfilter based approach, in Kubernetes, the iptables is still the most widely used option. It is developed under the Netfilter project [11], which consists of community-driven collaboratives. It is built up with

[†] Nokia Bell Labs Budapest, Bókay János u. 36-42, 1083 Hungary
[‡] Department of Networked Systems and Services, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, Műegyetem rkp. 3., H-1111 Budapest, Hungary
[§] HUN-REN-BME Cloud Applications Research Group, Magyar Tudósok Körútja 2, H-1117 Budapest, Hungary
(E-mail: akos.leiter@nokia-bell-labs.com; dome.matusovits@nokia.com; bokorl@hit.bme.hu)

DOI: 10.36244/ICJ.2025.1.7

kernel modules linked to the kernel at runtime, extending the monolithic kernel's functionality. The iptables framework communicates (Figure 2) with different predefined hookpoints of the kernel's protocol stack. These are where the Nework Address Translation (NAT), Network Address and Port Translation (NAPT), packet filtering, and other packet manipulation procedures take place:

- NF_IP_PRE_ROUTING: This is where the incoming traffic directly enters the kernel stack. There isn't any routing processing at this point.
- NF_IP_LOCAL_IN: The routing procedures have already been done, and the packet has been forwarded to the local host.
- NF_IP_FORWARD: This is the same as the previous case, except that the packet is destined for a remote host.
- NF_IP_LOCAL_OUT: The traffic is locally generated and destined to a remote host
- NF_IP_POST_ROUTING: Packet processing procedures after routing

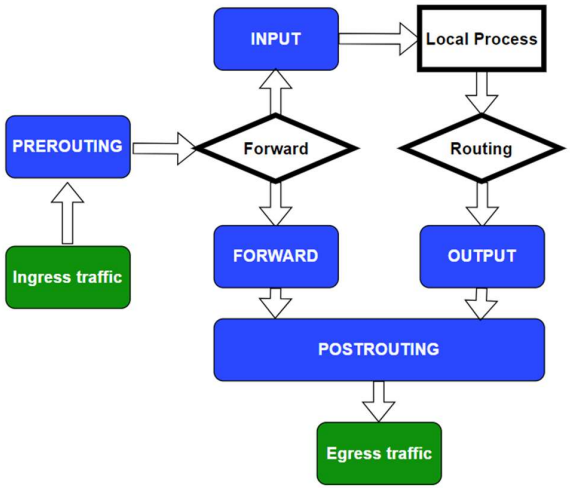

Figure 2 – The iptables packet processing in a nutshell

The iptables consists of rules, which contain targets. If a rule is evaluated, then the target is the action that needs to be executed (e.g., ACCEPT, DROP, RETURN, REJECT). Rules are part of chains that have two types: built-in (by the Linux OS) and custom (such as Kubernetes CNI-defined ones). The built-in chains are triggered by the abovementioned hookpoints respectively: PREROUTING (NF_IP_PRE_ROUTING), INPUT (NF_IP_LOCAL_IN), FORWARD (NF_IP_FORWARD), OUTPUT (NF_IP_LOCAL_OUT), POSTROUTING (NF_IP_POST_ROUTING).

The chains are located in tables. They are separated according to their appropriate functionality. For this reason, we can differentiate between Filter, NAT, Mangle, Raw, and Security tables. In the Filter table, the decision is made on whether the packet should enter or leave the network. The NAT rules can be found in the NAT table, as its name implies. The Mangle table contains packet manipulation rules. For configuration exemptions, where you do not want certain traffic to be tracked, you use the Raw table. It is designed to set a mark (NOTRACK) on a packet that has not wished to be tracked. For stricter access management, some Linux distributions include the security table. In order to achieve this, a mandatory access control (MAC) has been implemented.

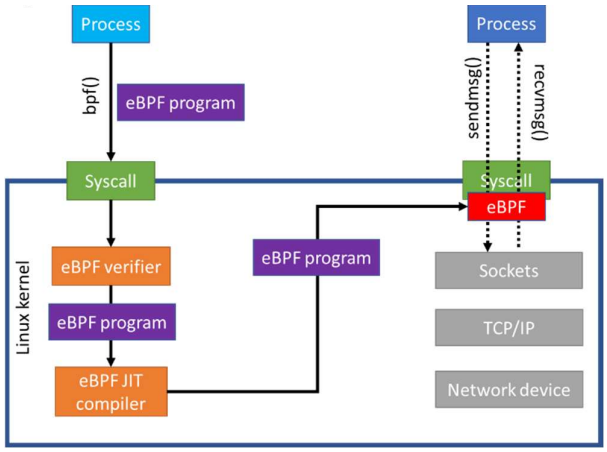

Figure 3 – eBPF execution flow

When the packet processing takes place, in the background the appropriate chain is selected within the table. Also, the desired rule should be applied in that chain. The problem occurs during the lookup phase. The table elements are not indexed; hence, the selection mechanism is sequential. At a small number of entries, it won't cause any problems. However, this number could be a significantly larger value in a production environment. In this case, we will experience substantial performance degradation in the packet processing. It could seriously harm SLA attributes, such as throughput and latency. That's why it is essential to develop a better solution that can enhance processing performance and help meet the requirements of URLLC communication.

Two well-known alternatives can boost packet processing: Vector Packet Processor (VPP) [12] and extended Berkeley Packet Filter (eBPF). The former solution implements a network stack, bypassing the Linux kernel. The essence here is that a new approach is being introduced to handling incoming traffic. The traditional Linux kernel-based solution is scalar processing, which typically processes one packet at a time. In contrast, at VPP, multiple packets are processed by their own network stack. These groups of packets are called vectors. In Kubernetes, the Calico CNI [9] is an example of its implementation. We do not go beyond this solution further, as our scope only focuses on eBPF.

In eBPF, we use eBPF programs to be loaded into the kernel from userspace (Figure 3). They are written in C language, but multiple development libraries can provide higher abstraction language levels, like bcc [13], libbpf [14], and eBPF Go[15]. Depending on the type of library, it uses a clang or Low Level Virtual Machine (LLVM) compiler to produce the so-called bytecodes from the source code. These are CPU-independent instruction sets translated by a Just-in- Time (JIT) compiler into machine-specific instruction sets. This way, we can optimize

the execution speed of the program using a natively compiled kernel code or a kernel module codebase. At a higher level abstraction, we can say that a virtual machine is practically embedded into the Linux kernel, where these eBPF programs run. They can be injected and executed on any level of the protocol stack. These are actually hookpoints, where certain events trigger the program execution. There can be many hookpoints, like kernel functions (kprobes) or user functions (uprobes) execution, system calls, or any tracepoints at the kernel. Also, eBPF programs can be attached to network interfaces or sockets as well (the latter two examples will be important in eBPF-based networking at Cilium). That's the key because, with this approach, we can extend the kernel's functionality without kernel source code modification or any usage of kernel modules. Additional functionalities, such as verification (depicted in Figure 3). improve and secure the execution compared to pure kernel modules, where there is no built-in and easy protection against, e.g., kernel panic. Before we get into the details of how eBPF can enhance packet processing in a Kubernetes environment, we introduce the related works that describe the most important eBPF use cases.

III. Relevant Parts of Kubernetes and Extended Berkeley Packet Filter

One of the most essential parts of our proposed testing environment is the Kubernetes integration of Cilium CNI, where our measurement results were gathered, depicted in Figure 4. It can leverage both iptables and eBPF-based networking to the whole cluster, where the Master and Worker nodes are located. The Master node is the control plane of Kubernetes. That is the component that involves the resource database (etcd) and the reconciliation loop mechanism (kube-controller manager), which controls the entire operation of the cluster. The API endpoint (apiserver) can also be found here, providing the cluster with reachability via HTTPS. What's more, the scheduling of workload resources (kube-scheduler) is also specified here. The Worker node provides the cluster's data plane. There, we could find the workloads that accomplish the desired services to be up and running.

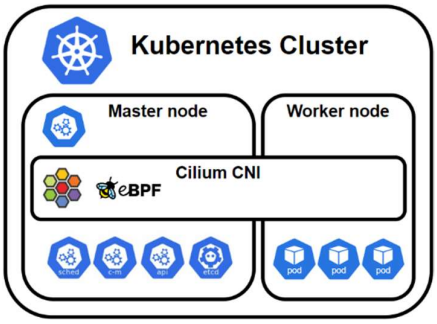

Figure 4 – High level architecture of Kubernetes with Cilium CNI

So far, all the cluster functionalities we mentioned have been implemented in Kubernetes' smallest unit, called the pod. Most of the time, a pod realizes a single container, but there can be a case when more than one container is embedded in a pod (e.g., a sidecar container, which receives the traffic, and there is another database container for information storage).

In both nodes, an entity should redirect control/data traffic to the desired endpoint (i.e., pod). That is where the kube- proxy comes into the picture. In most Kubernetes CNI solutions, iptables is used for packet processing. The kube-proxy's task is handling the appropriate chains, rules, and targets for traffic routing and manipulation. We aim to enhance packet processing performance by replacing iptables and hence, the kube-proxy.

eBPF also facilitates kernel programmability in Kubernetes [16]. Since there is only one kernel on a host, any application running in a container within a pod (in Kubernetes) must use the kernel whenever it requests access to hardware, manages files, or receives network messages. Regardless of the number of pods deployed on a machine, the kernel is always involved, whether we are talking about Bare Metal or a virtual machine. Containers do not have their own kernel; they use the existing kernel on the host machine. Thus, with proper eBPF instrumentation in the kernel, an agent can monitor all activities in the user space across all applications or cloud-native functions (micro-services). This enables complex eBPF tools to gain comprehensive observability across the entire node, providing deep insights into the cluster.

The two data paths that are associated with our experiment are shown in Figure 5. As a CNI, Cilium can deal with incoming traffic from the network interface of a Kubernetes Worker node or another Pod. Furthermore, the traffic destination can also be a Pod or the network interface of the Kubernetes Worker node. All the traffic goes through various iptables chains. The orange chains represent the default iptables chains; the blue ones are the Kubernetes-added ones. Cilium defines its own chains, depicted in purple.
The PREROUTING chain in Figure 5 is responsible for classifying whether traffic is local or must be forwarded. KUBE-SERVICES chains manage Kubernetes Services.

As shown in Figure 5, the many iptables chains on the data path can cause processing overhead and increased latency. This is where eBPF comes into the picture to circumvent issues with multiple iptables chains. An eBPF program can be loaded into the kernel to intercept traffic before the iptables-based processing starts. The hookpoint where the eBPF program is attached is called Traffic Control (TC). For incoming traffic, it is located before the PREROUTING, and for outgoing traffic, it can be found after the POSTROUTING chain. All of these mean that the eBPF- based solution intends to replace kube-proxy in Worker nodes that utilize iptables.

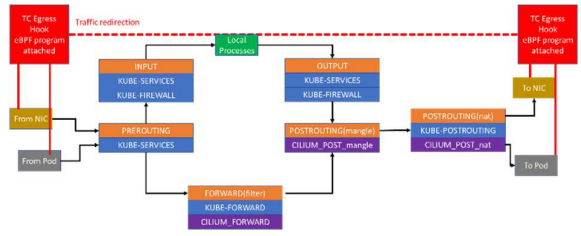

Figure 5 – iptables and eBPF-based Cilium data path [7]

IV. RELATED WORKS

eBPF can be used in many application fields related to security, observability, or performance enhancement scenarios. A summarization of the related papers is depicted in Figure 6.

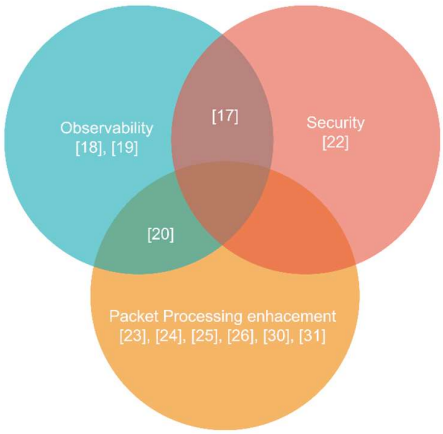

Figure 6 – eBPF use cases by summarized literatures

Regarding observability, eBPF can be used to monitor certain events. Attaching the written eBPF code to the appropriate hookpoints of the Linux kernel can trigger these eBPF programs to collect analytical traffic stream data. *David Soldani et al*. [17] used this approach to estimate cloud-native functions' energy consumption and derive performance counters and gauges for transport networks, 5G applications, and non-access stratum protocols. Furthermore, *Abderaouf Khichane et al*. [18] [19] have found a more profound way of measuring the behavior of a network function or protocol. Also, they could identify potential bottlenecks and SLA violations more accurately. *Carmine Scarpitta et al*. [20] describe a high-performance solution for end-to-end delay monitoring for SRv6-based networks. It leverages the Simple Two-way Active Measurement Protocol (STAMP) [21] to monitor the delay between two nodes called STAMP Session-Sender and Session-Reflector. The monitoring is implemented with eBPF programs.

Packet filtering mechanisms could be achieved more efficiently, like in the abovementioned paper by *David Soldani et al*. [17], where they detected and responded to unauthorized access to cloud-native resources in real time using eBPF. *Dominik Scholz et al*. [22] give a brief overview of analyzing the performance of eXpress Data Path (XDP), the lowest level before the network stack. They used for installing application-specific packet filtering configurations acting on the socket level. It is implemented with eBPF programs that are attached to the XDP hookpoint. It is well applicable for DoS prevention. Their case studies focus on performance aspects. Their packet filtering approach with eBPF doesn't have as much engineering cost. The performance losses are below 20%, while security is improved through better isolation between applications.

Attaching eBPF programs to the Linux kernel's protocol stack could also enhance the packet processing performance. It could be more efficient than the traditional netfiler [8] approach. That's the key point, as our goal was to evaluate performance using eBPF technology. *Matteo Bertrone et al*. [23] describe how the acceleration of packet processing can be achieved by emulating the iptables filtering semantic with eBPF, using Traffic Control (TC) or XDP. Depending on their use cases, such as delivering local traffic directly to the output port or connection tracking, they configured the data path respectively. This firewall solution is called bpf-iptables.

The paper by *Sebastiano Miano et al*. [24] extends the above scope by diving deep into the overall architecture of bpf-iptables, mentioning additional enhancements that make this technology perform better. Nftables [11] is also considered a relevant firewall alternative in these measurement scenarios. These are similar measurements in that they consider the TC hookpoint as we did. However, they only measure throughput, and the testbed is not in a cloud- native environment. We will also measure the latency and evaluate performance using a virtualized network infrastructure. Besides the observability aspect, in this previously mentioned approach by *Carmine Scarpitta et al*. [20], they managed to build the monitoring system where the eBPF implementation outperforms their examined solutions with negligible impact on the forwarding capability of the router. It uses XDP hookpoint, which differs from our scenario. Also, they only considered the throughput, similarly to paper [24].

Jung-Bok Lee et al. [25] implement an eBPF-based load-balancer. They also compare the performance of their eBPF-based solution to normal iptables, as we did in this paper. They developed a containerized high-performance load balancer that uses eBPF with the Linux kernel to distribute traffic, which can be easily managed via Kubernetes. They conducted tests simulating real-world traffic patterns using Internet Mix (IMIX) traffic streams. Their experimental results show that the proposed load balancer significantly outperforms the Destination Network Address Translation-based iptables solution, with the performance gap widening as packet size decreases. The measurements were conducted in a cloud environment, but their scope was only throughput performance scenarios, as in the previous papers. Also, they used XDP, instead of TC.

Federico Parola et al. show [26] a case study for Multi-access Edge Computing (MEC) technology, which is relevant in implementing the User Plane Function (UPF) deployed near the Radio Access Network (RAN), enabling telcos to provide services at close proximity to mobile users. In this scenario, high-performance data plane technologies, such as Data Plane Development Kit (DPDK) [27], may not be appropriate because they require dedicated resources like CPU cores and network interfaces. Furthermore, its proprietary drivers make it challenging to maintain and integrate DPDK. For this reason, they came up with a new idea to implement some of the functionalities of Mobile Gateway with eBPF/XDP, such as GPRS Tunneling Protocol Handling, QoS Management, Traffic Classifying, and Routing. They evaluated this approach with different Mobile Gateway data plane technologies like BESS [28], OpenvSwitch-DPDK (OvS-DPDK), and OvS-kernel [29].

The results show that eBPF competes with traditional kernel-bypass technologies. Although some performance degradation can be seen in some cases, it is still worth it because of higher integration with the kernel and more flexible resource usage. They used XDP hook as opposed to our case. Likewise, latency wasn't taken into account in these performance evaluations.

Dushyant Behl et al. [30] present a paper about the feasibility of eBPF for efficient implementation of network functions. They propose an eBPF-based framework to make the usage of eBPF CNI-agnostic. Their approach allows for replacing existing network functions with independent, eBPF-based modules. They were using multiple hookpoints: TC, XDP, and socket. We used only the TC hookpoint in our testbed. Also, they focused on enhancing the packet processing on the socket level by examining the throughput, where they could achieve a consistent 50% increase per scenario. There weren't any other attributes considered in their approach.

Code reusability is also an issue in the field of eBPF. *Federico Parola et al.* [31] address this problem by using PolyCube [32] [33]. PolyCube facilitates the development of efficient, modular, and dynamically reconfigurable network functions that run within the Linux kernel. This solution significantly improves Pod-to-Pod, Pod-to-Service, and Internet-to-Service throughput even in multi-node clusters compared to Flannel [21], Calico, and Cilium. This is the closest approach to our measurement use cases: it is based on Kubernetes, the traffic flow path is similar (Pod-to-Service scenario at least), and the eBPF hookpoint is the same (TC). They were even replacing the kube-proxy control plane element with eBPF programs as we did (they also achieved that with Cilium CNI in one of their test cases). However, they were scaling the associated pods to the Kubernetes Services not the number of Services itself. This is because they were curious about the load-balancing performance attributes when using eBPF. Also, as we can see in the previous papers, they only examined the throughput as a KPI.

V. The Implemented Test Environments

Based on Section III, we have created two test environments (Figure 8 and Figure 9) to study the performance of both solutions. The testbeds are built on OpenStack; the high-level design can be seen in Figure 7.

A. General principles of the test environments

The blue line represents the incoming, and the purple line shows the outgoing direction of the traffic (Figure 8, Figure 9). All the traffic originates from the *ITGSend* module of the Distributed Internet Traffic Generator (D-ITG) [34] on the client. The traffic is received by *ITGRecv* module, which is embedded in a Kubernetes Pod. There is a dedicated signaling port for connection establishment. To expose our D-ITG pod outside of the cluster, we need Kubernetes services (actually, ClusterIP is an exception because it makes the pod accessible only within the cluster). We can choose between ClusterIP, NodePort, LoadBalancer, and ExternalName. The latter option isn't remarkable for us since it only applies to mapping a service to a DNS name. Since the ITGSend module remains in the client network, the pod will be accessible through the Worker Node's interface with a private IP address. That means the NodePort service will be enough as it opens a port on the Worker node's interface and redirects the traffic to the pod. Note that LoadBalancer is preferred in production. We can use more protocols that can flow through it. Moreover, since it exposes the pod by acquiring an IP address for the desired service, we can make it accessible on the Internet (with a public IP address). However, for simplicity, we used NodePorts instead. Furthermore, the allocation of IPv4/v6 addresses for many services would harden the building of the testbed. The data ports were randomized. The maximum number of NodePorts is 2767, so when all of the ports were reserved, the remaining services were replaced with the type of ClusterIP during the service number increase, detailed in Section VI. To preserve the client's source IP, we use an annotation in the service definition file called *externalTrafficPolicy=Local*. With this annotation, the kube-proxy/eBPF program only proxies requests to local endpoints, which means we can avoid SNAT translation to node IP during any considered traffic flow.

B. High-level design

The client and the router VMs were placed in the client network created by OpenStack. The Kubernetes cluster – including the master and the worker node – was in the data center network, which was also created by OpenStack. All the elements in the test system (Client, Router, Master, and Worker Node) are OpenStack-instantiated virtual machines. We installed Ubuntu OS with version 20.04 (5.4.0 Linux kernel version) for the VMs. Also, we reserved 20Gb virtual memory with 1VCPU (1 core) and 2Gb RAM for the Client and the Router. Regarding the Master and Worker node instances, the setup was 40Gb memory with 2VCPUs (2 cores) and 4Gb RAM. The CPU clock rate was configured with 1500 MHz for each setup. There is also a management node to examine the system behavior without affecting the measurements, represented by orange lines. The black lines show the actual traffic path to be measured.

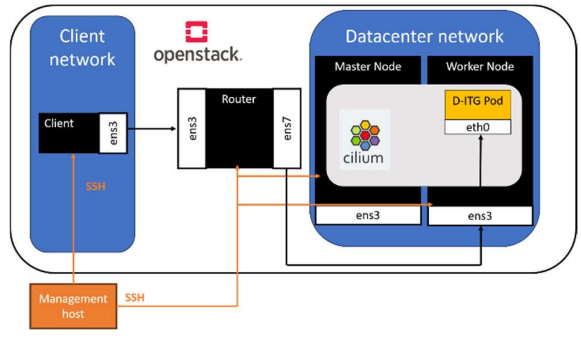

Figure 7 – The high-level testbed design with implementation details

C. Test environment for kube-proxy (iptables)

The kube-proxy-based test environment is shown in Figure 8. The red rectangles represent the relevant iptables chains through which the traffic goes.

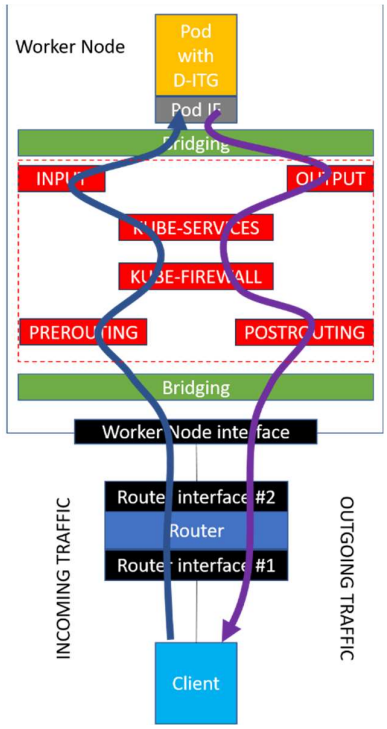

Figure 8 – Test environment #1: iptables-based forwarding

D. Test environment for eBPF

The eBPF-based test environment is depicted in Figure 9. The green rectangle shows the hook points where the eBPF program is attached. This means that after the packet arrives at the node interface, the eBPF program is triggered, and the packet processing and forwarding continue without iptables interaction.

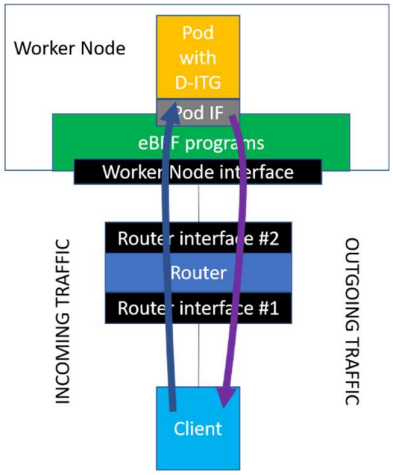

Figure 9 – Test environment #2: eBPF-based forwarding

VI. MEASUREMENTS DESCRIPTIONS AND RESULTS

We have examined several aspects of packet processing for our evaluation purposes. As we mentioned earlier, every measurement uses Kubernetes Services with NodePorts. A port number is associated with the node's IP address and will be translated to the Pod's IP and port number where you can reach the server. This Kubernetes object is responsible for routing traffic from the worker node's interface to the Pod handled by the kube-proxy/eBPF program. The traffic distribution between NodePorts is random. D-ITG is used for traffic generation. The test environments introduced in Section IV are applied.

We have continuously increased the number of Kubernetes Services to conclude the related bottlenecks of Kubernetes. Meanwhile, we evaluated two packet processing methods: normal kube-proxy-based (iptables) and eBPF- based.

E. TCPthroughputmeasurements

We used a relative scale as it is tough to determine maximum throughput in a virtualized environment. All the virtual links have been limited to 500 Mbps. One hundred measurements have been executed in every scenario – with 30-second-long TCP streams – where the number of Kubernetes Services is increased by 1000 (except from 1 to 1000). Altogether, 1100 measurements were evaluated overall.

Goal: Concluding the difference between kube-proxy (iptables) and eBPF-based packet processing in the case of IPv4 and IPv6 and within the context of throughput behavior.

Measurement results: From the data point of view, we highlight the standard deviation (Table 1) as there is no significant difference between the minimum, maximum, average, and median values. These values are also represented in Figure 10 and Figure 12, showcasing a different perspective on the measurement data.

TABLE I
eBPF-BASED THROUGHPUT STANDARD DEVIATION RATIOS COMPARED TO IPTABLES-BASED IN THE CASE OF IPv4 AND IPv6

Number of Kubernetes Services	Standard deviation ratio (IPv4)	Standard deviation ratio (IPv6)
1	0.20	0.85
1000	1.01	0.78
2000	1.07	0.95
3000	1.41	1.22
4000	0.93	0.56
5000	3.05	1.33
6000	0.92	1.20
7000	0.68	0.74
8000	0.91	0.56
9000	0.67	1.07
10000	0.67	0.88

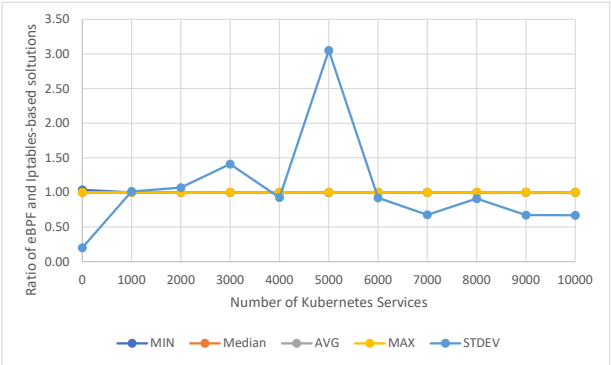

Figure 10 – Summarized diagram of throughput measurements between iptables and eBPF with IPv4

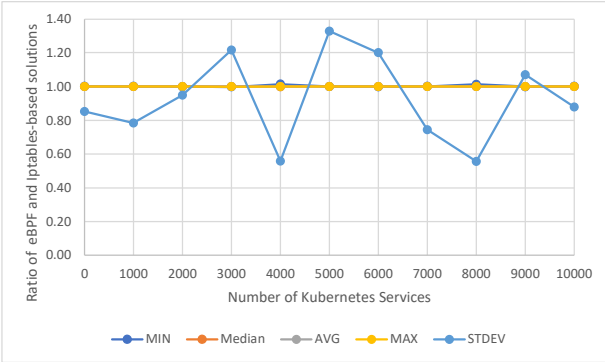

Figure 11 – Summarized diagram of throughput measurements between iptables and eBPF with IPv6

Conclusion: In the case of a high number of Kubernetes Services, the standard deviation of throughput is lower when eBPF is used in most cases. IPv6-based throughput values are more stable compared to IPv4 in both approaches, as the standard deviation of the eBPF to iptables ratio is lower.

F. Latency measurements

We also use a relative scale, just as we did in the case of throughput measurements. UDP traffic originated 100 times in every scenario with a 30-second-long flow, with the same service scaling as in the throughput measurements. This means 1100 measurements were in summary.

Goal: Concluding the difference between kube-proxy (iptables) and eBPF-based packet processing in the case of IPv4 and IPv6 within the context of latency behavior.

Measurement results: From the data point of view, we highlight the maximum and standard deviation (Table 2, Table 3) as there is no significant difference between minimum, average, and median values. The data is also represented in the graphs of Figure 12 and Figure 13.

TABLE II
eBPF-BASED LATENCY MAXIMUM AND STANDARD DEVIATION RATIO COMPARED TO IPTABLES-BASED IN THE CASE OF IPv4

Number of Kubernetes Services	Maximum	Standard deviation ratio
1	1.10	1.82
1000	1.21	1.91
2000	1.17	1.15
3000	1.18	1.49
4000	0.86	0.62
5000	0.90	0.47
6000	1.13	1.22
7000	0.80	0.50
8000	1.04	1.08
9000	0.91	0.54
10000	0.84	0.40

TABLE III
eBPF-BASED LATENCY MAXIMUM AND STANDARD DEVIATION RATIO COMPARED TO IPTABLES-BASED IN THE CASE OF IPv6

Number of Kubernetes Services	Maximum	Standard deviation ratio
1	1.08	1.01
1000	0.92	0.93
2000	0.95	0.93
3000	0.91	0.94
4000	0.86	0.87
5000	1.16	1.57
6000	1.05	1.00
7000	1.18	0.98
8000	0.65	0.74
9000	1.09	1.26
10000	0.87	0.82

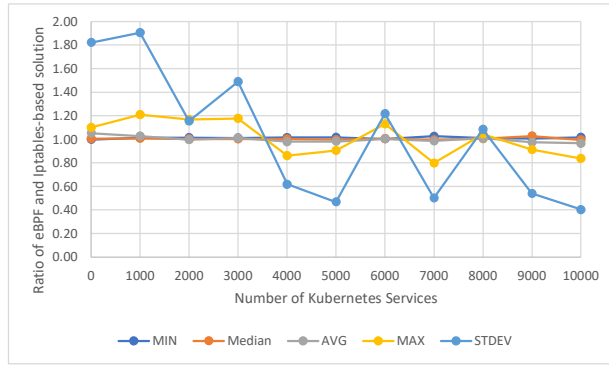

Figure 12 – Summarized diagram of delay measurements between iptables and eBPF with IPv4

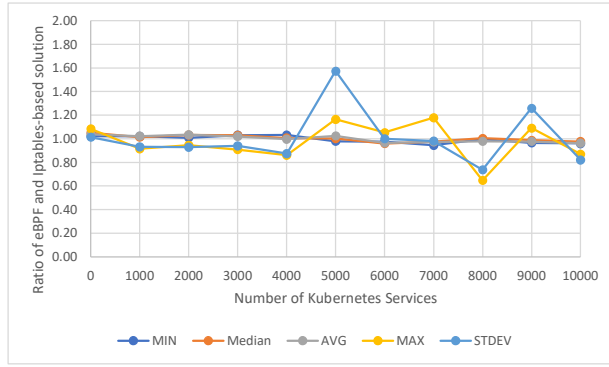

Figure 13 – Summarized diagram of delay measurements between iptables and eBPF with IPv6

Conclusion: The maximum latency values are higher for fewer Kubernetes services when using eBPF over IPv4. However, for IPv6 traffic, eBPF performs better in the case of fewer Kubernetes Services.

The standard deviation of latency for eBFP over IPv4 is lower in the case of 5 out of 11 scenarios. However, the greater the number of Kubernetes services used, the lower the standard deviation trend-wise in the case of eBPF. For IPv6, the standard deviation tends to be lower for eBPF with fewer Kubernetes services. However, this is not so significant compared to the IPv4 cases. Overall, IPv6 latency is more stable than IPv4 from the standard deviation point of view as the fluctuation of the values is lower.

G. Measurements conclusion

Generally, we can say that the eBPF-based solutions are more "stable" as the standard deviation is lower.

Generally, we can say that, based on the number of Kubernetes services, it is worth considering which type of Kubernetes packet processing is used because appropriate solutions can be suggested for different cases.

This experience can be utilized in URLLC network slices. Slicing SLAs always specify how reliably a particular parameter has to be kept (e.g., 99.999% of the time). These SLA requirements may be maintained better with lower latency fluctuation in certain eBPF cases. This can also contribute to telecommunication systems' overall software availability, as Varga et al. detailed in [35] [36].

Voice over IP services can also consider the results as lower jitter can be reached concerning the number of Kubernetes services in the telco cloud hardware.

H. Lessons learned

There were several difficulties during the creation of the test environment. Firstly, it is essential to differentiate the architecture of the measurement tools. As for the iPerf [37], there is a client and a server entity, and the connection establishment happens at the same port through which the data traffic flows. This means that the observed traffic is influenced by the signaling messages. In the case of D-ITG, there are dedicated ports for signaling and traffic generation, respectively. The desired data port we want to use is sent over the control plane as a plain-text message. Therefore, the NodePorts to pod's port translation won't happen. This means we cannot send traffic to the pod. Our solution was to choose the same port for the pod's port and the NodePort, so we had to define the NodePort to achieve that manually. Furthermore, there were some cases when the server was shut down randomly. So, we had to handle this and consider it inside the automatized shell script, which we used for measurements. Moreover, D-ITG components are more separated by their functionalities than iPerf. There are several entities present: ITGSend (at the client, it establishes the connection and generates the traffic), ITGRecv (at the server, it receives the traffic), ITGDec (at the client, it decodes the measurement result saved in a config file). Beyond the scope of our testbed, other entities can still be used for different scenarios, like ITGLog and ITGManager. So, we can see that the overall architecture of D-ITG is more complex than that of iPerf, which uses a simple point-to-point client-server model. Even though it is hard to implement D-ITG measurement in a cloud-native environment, this is still a valid solution as it is very flexible and has accurate traffic generation [38] [39]. It is also important to mention that the most unstable test scenarios were the IPv6-based traffic generations, where we used eBPF programs for packet processing. There were some cases where the traffic generator crashed. Not to mention that the higher the throughput was, the more time it took for ITGDec to decode the config file. Unfortunately, it is a limitation of the D-ITG software, which caused a massive impact on the measurement time. Delay measurement test cases took about 8-9 hours, and for the throughput analysis, it was 16-18 hours. All in all, generalizing the applied scripts required continuous development and spared much time to be usable. In a cloud environment – especially in public clouds – it is impossible to fully isolate a particular workload. Background traffic and other workloads may affect the measurement system and the performance of network functions. This might add an additional deviation in the results.

VII. CONCLUSION

In this paper, we have shown that there is room for eBPF to improve network performance in several use cases of Kubernetes-based telco cloud infrastructures. With the help of our results, operators can choose the packet processing methods that are the most suitable for their usage. With a significant service number, the throughput and latency values are more stable with IPv4 and eBPF. In IPv6-based measurements, the use of eBPF gives more stable results in most of the cases.

eBPF is not just about performance improvements; it is a complete framework supporting more straightforward and secure software development. Telecommunication networks can also benefit from better observability of network functions, which supports a variety of fields to be measured, such as energy consumption, SLA violation, logging, protocol analysis, security, etc. We believe this holistic approach will affect the whole telecommunication landscape. Even though there are some cases where eBPF does not outperform iptables, the application fields and feature sets mentioned above are worth the cost.

eBPF can support network slicing itself due to the increased observability, which leads to more control over particular data paths. Thus, it is easier to fulfill slice availability and performance requirements.

VIII. FUTURE WORK

The scalability of Kubernetes is crucial, and it is not just with Kubernetes services. It is worth examining how many Worker Nodes, Ingress controllers, etc., can be safely used by telecommunication applications or even in a regular IT cloud. As an example, Calico Typha deals also with scalability [40]. This pertains to, e.g., regulatory requirements where advanced logging is needed, which must also be scalable. In a Kubernetes-based telco environment, it is worth examining how eBPF can solve issues related to networking itself, such as assigning multiple interfaces to a pod or eliminating Network Address Translation (NAT) by Kubernetes services [41]. Furthermore, additional measurement points can be added to identify which packet processing section can cause increased volatility.

ACKNOWLEDGMENT

The authors thank Nandor Galambosi and Jozsef Varga for their constructive criticism and valuable comments while preparing the manuscript.

REFERENCES

[1] *IMT-2020 requirements*. Accessed: Aug. 21, 2024. [Online]. Available: https://www.3gpp.org/technologies/3gpp-meets-imt-2020

[2] G. Blinowski, A. Ojdowska, and A. Przybyłek, 'Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation', *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022, DOI: 10.1109/ACCESS.2022.3152803.

[3] O. Al-Debagy and P. Martinek, 'A Comparative Review of Microservices and Monolithic Architectures', in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 000 149–000 154. DOI: 10.1109/CINTI.2018.8928192.

[4] H. T. Nguyen, T. Van Do, and C. Rotter, 'Scaling UPF Instances in 5G/6G Core With Deep Reinforcement Learning', *IEEE Access*, vol. 9, pp. 165 892–165 906, 2021, DOI: 10.1109/ACCESS.2021.3135315.

[5] P. Mach and Z. Becvar, 'Mobile Edge Computing: A Survey on Architecture and Computation Offloading', *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017, DOI: 10.1109/COMST.2017.2682318.

[6] 'Extended Berkeley Packet Filter (eBPF)'. [Online]. Available: https://ebpf.io/

[7] 'Cilium Kubernetes CNI'. Accessed: Nov. 05, 2023. [Online]. Available: https://cilium.io/

[8] 'Nokia official documentation of Network Service Platform'. Accessed: Jun. 26, 2024. [Online]. Available: https://documentation.nokia.com/nsp/24-4/Transport_Slice_Controller/Overview.html

[9] 'Calico Kubernetes CNI'. Accessed: Nov. 05, 2023. [Online]. Available: https://docs.projectcalico.org/getting-started/kubernetes/

[10] 'Nftables'. Accessed: Jun. 28, 2024. [Online]. Available: https://netfilter.org/projects/nftables/

[11] *Netfilter project*. Accessed: Jul. 10, 2024. [Online]. Available: https://www.netfilter.org/

[12] 'Vector Packet Processing'. Accessed: Feb. 04, 2025. [Online]. Available: https://fdio-vpp.readthedocs.io/en/latest/overview/whatisvpp/what-is-vector-packet-processing.html

[13] 'BCC – Toolkit and library for efficient BPF-based kernel tracing'. Accessed: May 10, 2024. [Online]. Available: https://ebpf.io/applications/

[14] 'libbpf – C-based library'. Accessed: Mar. 22, 2024. [Online]. Available: https://docs.kernel.org/bpf/libbpf/libbpf_overview.html

[15] 'The eBPF Library for Go'. Accessed: Mar. 14, 2024. [Online]. Available: https://ebpf-go.dev/

[16] L. Rise, *Learning eBPF: Programming the Linux Kernel for Enhanced Observability Networking and Security, pp. 218, 2023*. [Online]. Available: https://github.com/lizrice/learning-ebpf

[17] D. Soldani *et al.*, 'eBPF: A New Approach to Cloud-Native Observability, Networking and Security for Current (5G) and Future Mobile Networks (6G and Beyond)', *IEEE Access*, vol. 11, pp. 57 174– 57 202, 2023, DOI: 10.1109/ACCESS.2023.3281480.

[18] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, '5GC-Observer: a Non-intrusive Observability Framework for Cloud Native 5G System', in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–10. DOI: 10.1109/NOMS56928.2023.10154433.

[19] A. Khichane, I. Fajjari, N. Aitsaadi, and M. Gueroui, '5GC-Observer Demonstrator: a Non-intrusive Observability Prototype for Cloud Native 5G System', in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–3. DOI: 10.1109/NOMS56928.2023.10154369.

[20] C. Scarpitta, G. Sidoretti, A. Mayer, S. Salsano, A. Abdelsalam, and C. Filsfils, 'High Performance Delay Monitoring for SRv6-Based SD-WANs', *IEEE Transactions on Network and Service Management*, vol. 21, no. 1, pp. 1067–1081, 2024, DOI: 10.1109/TNSM.2023.3300151.

[21] G. Mirsky, G. Jun, H. Nydell, and R. Foote, 'Simple Two-Way Active Measurement Protocol'. in Internet Request for Comments, no. 8762. RFC Editor, Fremont, CA, USA, Mar. 2020. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8762.txt

[22] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle, 'Performance Implications of Packet Filtering with Linux eBPF', in *2018 30th International Teletraffic Congress (ITC 30)*, 2018, pp. 209–217. DOI: 10.1109/ITC30.2018.00039.

[23] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, 'Accelerating Linux Security with eBPF iptables', in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, in SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 108–110. DOI: 10.1145/3234200.3234228.

[24] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi, 'Securing Linux with a faster and scalable iptables', *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 3, pp. 2–17, Nov. 2019, DOI: 10.1145/3371927.3371929.

[25] J.-B. Lee, T.-H. Yoo, E.-H. Lee, B.-H. Hwang, S.-W. Ahn, and C.-H. Cho, 'High-Performance Software Load Balancer for Cloud-Native Architecture', *IEEE Access*, vol. 9, pp. 123704–123716, 2021, DOI: 10.1109/ACCESS.2021.3108801.

[26] F. Parola, F. Risso, and S. Miano, 'Providing Telco-oriented Network Services with eBPF: the Case for a 5G Mobile Gateway', in *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, 2021, pp. 221–225. DOI: 10.1109/NetSoft51509.2021.9492571.

[27] 'Data Plane Development Kit (DPDK)'. Accessed: Jul. 01, 2024. [Online]. Available: https://www.dpdk.org/

[28] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, 'SoftNIC: A software NIC to augment hardware', *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155*, 2015.

[29] B. Pfaff *et al.*, 'The design and implementation of open {vSwitch}', in *12th USENIX symposium on networked systems design and implementation (NSDI 15)*, 2015, pp. 117–130.

[30] D. Behl, H. Huang, P. Kodeswaran, and S. Sen, 'On eBPF extensions to Kubernetes CNI datapath', in *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2023, pp. 207–209. DOI: 10.1109/COMSNETS56262.2023.10041357.

[31] F. Parola, L. D. Giovanna, G. Ognibene, and F. Risso, 'Creating Disaggregated Network Services with eBPF: the Kubernetes Network Provider Use Case', in *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, 2022, pp. 254–258. DOI: 10.1109/NetSoft54395.2022.9844062.

[32] S. Miano, F. Risso, M. V. Bernal, M. Bertrone, and Y. Lu, 'A Framework for eBPF-Based Network Functions in an Era of Microservices', *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 133–151, 2021, DOI: 10.1109/TNSM.2021.3055676.

[33] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, 'Creating Complex Network Services with eBPF: Experience and Lessons Learned', in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–8. DOI: 10.1109/HPSR.2018.8850758.

[34] S. Avallone, S. Guadagno, D. Emma, A. Pescape, and G. Ventre, 'D-ITG distributed Internet traffic generator', in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.*, 2004, pp. 316–317. DOI: 10.1109/QEST.2004.1348045.

[35] J. Varga, A. Hilt, J. Bíró, C. Rotter, and G. Jaro, 'Reducing operational costs of ultra-reliable low latency services in 5G', *Infocommunications Journal*, vol. X, pp. 37–45, 2018, DOI: 10.36244/ICJ.2018.4.6.

[36] J. Varga, A. Hilt, C. Rotter, and G. Járó, 'Providing Ultra-Reliable Low Latency Services for 5G with Unattended Datacenters', in *2018 11th International Symposium on Communication Systems, Networks Digital Signal Processing (CSNDSP)*, 2018, pp. 1–4. DOI: 10.1109/CSNDSP.2018.8471756.

[37] 'Iperf3 – traffic generator'. Accessed: Nov. 05, 2023. [Online]. Available: https://iperf.fr/iperf-download.php

[38] G. Aceto, C. Guida, A. Montieri, V. Persico, and A. Pescapè, 'A First Look at Accurate Network Traffic Generation in Virtual Environments', in *2022 IEEE Symposium on Computers and Communications (ISCC)*, 2022, pp. 1–6. DOI: 10.1109/ISCC55528.2022.9913058.

[39] D. Perepelkin and M. Ivanchikova, 'Problem of Network Traffic Classification in Multiprovider Cloud Infrastructures Based on Machine Learning Methods', in *2021 10th Mediterranean Conference on Embedded Computing (MECO)*, 2021, pp. 1–5. DOI: 10.1109/MECO52532.2021.9460171.

[40] 'Calico Typha'. [Online]. Available: https://docs.tigera.io/calico/latest/reference/typha/

[41] Ákos Leiter *et al*., 'Cloud-Native IP-Based Mobility Management: A MIPv6 Home Agent Standalone Microservice Design', presented at the CSNDSP 2022

Ákos Leiter graduated as a Computer Engineer MSc at the Department of Networked Systems and Services (HIT), Budapest University of Technology and Economics (BME) in 2015, specializing in Computer Networks. His thesis was about proposing an operator-centric, dynamic flow mobility protocol with IP in the Evolved Packet Core. He is a PhD candidate at HIT's Multimedia Networks and Services Laboratory (MEDIANETS) and a research engineer at Nokia Bell Labs. His main research field is Network Function Virtualization and Software Defined Networking, including Orchestration and Network Automation. His work-in-progress PhD thesis is about the cloudification of the Mobile IPv6 protocol family on top of Kubernetes.

Döme Matusovits graduated from the Budapest University of Technology with a BSc in Electrical Engineering in 2024. He is currently pursuing his MSc degree at the Department of Networked Systems and Services (HIT), specializing in computer and mobile networks. His ongoing MSc thesis focuses on the Orchestration of Network Slices and Inter-Slice Handover. His main research fields include 5G, Network Slicing, and Software Defined Networking within the scope of Network Automation. The research is conducted in close collaboration with Nokia Bell Labs and HIT. Since 2023, he has been working at Nokia in the Cloud and Network Services, developing 5G products.

László Bokor received his Ph.D. degree in computer engineering from Budapest University of Technology and Economics (BME) in 2014. He is currently an associate professor at the Department of Networked Systems and Services (HIT), where he leads the Vehicular Communications Research Group founded within strong industry-academic cooperation. He is a member of the HTE (Scientific Association for Infocommunications Hungary), the Hungarian Standards Institution's Technical Committee for Intelligent Transport Systems (MSZT/MB 911), the TPEGoverC-ITS Task Force within the TPEG Application Working Group of TISA, the ITS Hungary Association (the Hungarian organization of ERTICO's Network of National ITS Associations), and the BME's Multimedia Networks and Services Laboratory, where he participates in different R&D projects. His research interests include IPv6 mobility, SDN/NFV-based mobile networks, network simulation, mobile healthcare infrastructures, and V2X communication in cooperative intelligent transportation systems.