When SD-WAN meets eBPF

Sofiane Messaoudi*, Franck Messaoudi*, Adlen Ksentini*, Christian Bonnet*
*Eurecom
*Sophia Antipolis, France
Email: *name.surname@eurecom.fr

***Abstract*—eBPF represents a major advancement for SD-WAN implementations, enabling high-performance, programmable, and secure network processing capabilities directly within the Linux kernel. Unlike traditional SD-WAN solutions that rely on user-space processing—which often introduces latency and performance bottlenecks—eBPF permits custom packet handling to operate at near-wire speed within the kernel. This kernel-level processing allows flexible traffic control and QoS management, enabling SD-WAN providers to implement custom traffic steering, load balancing, and QoS policies, optimizing resource allocation and dynamically prioritizing applications. In this paper, we introduce HELIOS, the first implementation of SD-WAN edge nodes powered by eBPF with centralized control facilitated by advanced applications on an ONOS controller. Extensive benchmarking showcases the efficiency and responsiveness of the proposed framework, highlighting eBPF's potential to deliver substantial performance improvements in SD-WAN contexts.**

I. Introduction

Cloud Edge Continuum (CEC) is undoubtedly the next evolution of cloud computing, where workload is executed over the continuum to gain the low-latency capabilities of edge and far-edge computing nodes, while centralized cloud resources handle high-load, and delay-tolerant tasks. The CEC will build on the advance and democratization of edge computing, with nodes positioned close to end users and an increasing array of end-user devices—such as Internet of Things (IoT) gateways, Unmanned Aerial Vehicles (UAVs), and smartphones—capable of running workloads. Meanwhile, the shift toward microservice-based applications will further drive CEC adoption, enabling application components to be distributed across the continuum to meet Service-Level Agreement (SLA) requirements. However, interconnecting cloud, edge, and far-edge nodes presents challenges, as multiple providers must collaborate to build a cohesive continuum and ensure seamless application deployment per SLA standards. This cooperation is essential, as edge nodes have limited computing capacity and require support from cloud and other edge providers to manage sudden computation demands.

Interconnecting CEC nodes that carry the Data-Plane (DP) of deployed microservices is essential, requiring (i) programmability to enable network orchestrators to specify Quality of Service (QoS) levels and resiliency for each service, (ii) QoS enforcement to ensure SLAs—such as low latency, high bandwidth, and minimal packet loss—and (iii) resiliency by enabling route selection flexibility for services. Two primary interconnection solutions are considered: relying on the underlying network (e.g., Segment Routing or Multi-Protocol Label Switching (MPLS) tunnels) or using overlay networks over existing Internet links (e.g., Software-defined Wide Area Network (SD-WAN)). While the first approach efficiently supports QoS, it assumes that all CEC nodes can control the underlying network to define QoS levels and paths for application microservices, a capability that cloud providers typically possess when connecting their own data centers with MPLS tunnels. However, this approach is limited since cloud and edge providers often rely on network links operated by third-tier Internet Service Providers (ISPs). Additionally, CEC infrastructures often include edge and far-edge computing resources connected via standard network connectivity, with no control over the underlying network.

On the other hand, SD-WAN is a key technology for interconnecting CEC nodes as it meets the three main requirements of programmability [1], QoS, and resiliency. Leveraging Software-Defined Networking (SDN) principles, SD-WAN simplifies network management by decoupling hardware from control programs and using software and open APIs to abstract infrastructure. It creates overlay networks on top of heterogeneous underlay networks, including those from different ISPs, while maintaining consistent addressing. SD-WAN supports multiple concurrent WAN connections and can interconnect sites in various topologies, such as mesh, to build a full overlay where SD-WAN edge nodes manage routing for SLA, resiliency, and scalability without relying on underlying network resources. However, most current SD-WAN solutions are proprietary, limiting innovation. This work addresses that gap by proposing an SD-WAN framework that leverages extended Berkeley Packet Filter (eBPF) in the Linux kernel to design SD-WAN edge nodes, enabling overlays between CEC nodes. These nodes are managed by ONOS controller to ensure QoS across links that connect microservices across the continuum.

The rest of this paper is organized as follows: Section II presents the eBPF technologies. Section III details our solution. Section IV describes the methodology and testbed. Section V presents the performance evaluation and results. Finally, Section VI concludes the paper.

II. Background

A. (e)BPF

eBPF is a Linux-based Virtual Machine (VM) that runs sandboxed programs in a privileged mode to *safely* and *efficiently* extend the capabilities of the kernel with custom code that can be injected at run-time without requiring changes in the kernel source code or load kernel modules. The eBPF is an *event-driven* program triggered when the kernel or application

passes a certain *hook point*. eBPF has some predefined hook points that include system calls, every kernel function, kernel trace points, and network events, to name few. If a predefined hook does not exists for particular need, it is possible to create one at kernel probe or user probe, almost anywhere.

B. eXpress Data Path (XDP)

Is a network type of eBPF programs, designed for high-performance packet processing. Identified by its hook point within Linux kernel's networking stack, specifically in the reception chain of the *network device driver*, prior to Socket Kernel Buffer (SKB) allocation. The XDP program is triggered *immediately* on the *ingress* path in response to network events, typically upon packet reception. The hook point varies according to how the XDP program is attached. We distinguish the *generic*, *native*, and *offloaded* XDP. Generic XDP is loaded into the kernel as part of the regular network path, making it suitable for testing. Native XDP is loaded within the network card driver, providing better performance but requiring driver support. Offloaded XDP runs directly within (smart) embedded Network Interface Controllers (NICs) and requires specific device support. Network device drivers may not support XDP hooks, in which case the generic model is utilized. In Linux 4.18 and later, supported drivers include Veth, Virtio, Tap, Tun, Lxgbe, I40e, Mlx5, and MLX4, to name few. XDP has become a popular mechanism for accelerating and offloading packet processing from user-space applications for high-performance networking applications.

III. SD-WAN Featuring eBPF Proposal

A. Holistic Perspective

The High-Efficiency Layered Infrastructure with eBPF for Optimized SD-WAN (HELIOS) architecture depicted in Figure 1, was inspired by the Metro Ethernet Forum (MEF) organization [2]. It is designed to meet the need for scalable and flexible connectivity across distributed networks by decoupling the Control-Plane (CP) from the DP. The architecture comprises two layer.

a) Control Layer: This layer hosts the central attraction (i.e., *ONOS SDN controller*). It acts as a server and a client for, respectively, the above and below layers. As a server, it receives requests from the application logic via its North Bound Interfaces (NBIs), extracts the flow rules and distribute them among the infrastructure using the Google Remote Procedure Calls (gRPC) protocol. It aims to dynamically, seamlessly, and simultaneously create, update, and delete eBPF maps within Edge-Gateways. As a client, it monitors the network traffic and devices, gathering the necessary information from the infrastructure (below) layer to push modifications in real-time in the event of misbehavior, while also returning statistics and monitoring data to Verticals.

b) Infrastructure Layer: (Or DP) comprises the network elements such as end-user devices, Edge-Gateways, MPLS equipments, and Cloud resources. Our focus is on the Edge-Gateways, which serve as endpoints for SD-WAN tunnels, where we implement eBPF accelerations. Traditional SD-WAN transit traffic by user-space, causing overhead and reducing performance; even optimized solutions like Data Plane Development Kit (DPDK) dedicate hardware (Central Processing Unit (CPU) and NICs) exclusively to specific applications. In contrast, eBPF enables resource sharing, improving efficiency. Each Edge-Gateway hosts our Low-Latency Intelligent Network eXecution using eBPF (LINX) prototype (see Section III-B), which maintains eBPF maps that store the flow rules. Each LINX XDP Section Program attaches to one interface of the gateway. We distinguish three types namely; *Generic Routing Encapsulation (GRE)* interface(s), which facilitate(s) communication between sites over WAN links. *Non-GRE* interface(s) for Local Area Network (LAN) communication, for instance between end-user devices, and the Edge-Gateway. respectively. Beside that a control (*gRPC*) interface is used to listen for SDN controller requests.

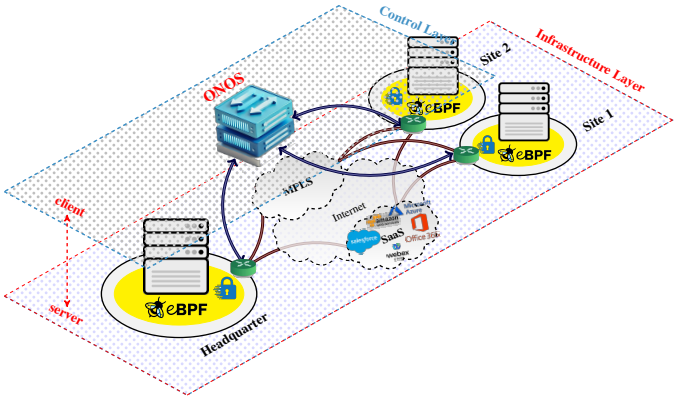


Fig. 1: HELIOS Architecture (global view)

B. Detailed View

Now, we focus on the eBPF-based DP architecture. Modern SD-WAN supports two deployment approaches: Datacenter-to-Datacenter (DC-to-DC) or End-User-to-Datacenter (End-User-to-DC). Our solution supports both, managing traffic from End-User and/or DCs. To meet these requirements, and in line with the SD-WAN purposes (including, efficiently forwarding and load-balancing traffic across DCs), we have designed the LINX architecture (Figure 2) as follows:

a) Management Layer: Operates at the user space. This layer functions as a core library responsible for configuring both user Data-Path and the edge itself. It includes the following components: (i) - *Yet Another Markup Language (YAML) Config Validator*, which applies configuration settings for specific Data-Path aspects, defining CP and DP interfaces, ports, and supporting eBPF acceleration. (ii) - *gRPC Server* enables interaction with the SDN controller via Remote Procedure Callss (RPCs), handling control messages on the default port 50051 for creating, updating, and deleting GRE overlays. (iii) - *GRE Overlay Manager* oversees GRE tunnels, it receives packet processing rules from the gRPC server, and coordinates tasks like prioritizing Packet Detection Rules (PDRs) and managing Forwarding Action Rules (FARs) and

Load Balancing Rules (LBRs). (iv) - *eBPF Program Manager* manages the lifecycle of eBPF programs within the device driver, enabling real-time configuration changes on the flow rules via on-the-fly generated *eBPF skeleton* APIs.

b) Data-Path Layer: Responsible for processing End-User/DC traffic, implementing a pipeline where decisions are made about the fate of each packet—whether it is passed, dropped, or redirected. The pipeline is divided into four key components as follow: (1) - *Traffic Parser* is considered as the entry-point to the Data-Path; it parses incoming traffic and matches fields in PDRs. (2) - *Traffic Classifier* categorizes traffic by uplink (e.g., GRE) and downlink (e.g., NON-GRE) flows, as well as other criteria such as Classes of Traffics (CoTs) and protocols, using PDRs and matching Packet Detection Information (PDI). (3) - *Traffic Forwarder*, following classification, it directs traffic to its destination: either to the LAN (i.e., End-User) by removing GRE headers or to the WAN (i.e., DC) with appropriate GRE encapsulation, in alignment with the FARs. Finally, (4) - *Traffic Load Balancer*, ensures traffic adheres to predefined LBRs updates, regarding GRE overlay selection, monitoring and dynamically adjusting flows to optimize network Key Performance Indicators (KPIs) and enhance resource efficiency.

C. Control Signaling

In this paper, we define the following key concepts and terms to better understand our data model in Schema 1:

- *Flow Rule*: A set of instructions dictating how traffic is handled within SDN architectures. They are crucial for efficient secure traffic management, as they define specific criteria and attributes (such as source and destination IP addresses, protocols, and ports) against which traffic is evaluated for redirection, modification, or dropping.
- *Match Field*: An attribute within the PDR that is used to identify and categorize incoming packets based on predefined criteria. It plays a crucial role in determining how packets are processed by the network, serving as basis element of a flow rule. In addition to the standard criteria, match field can also include a Virtual Local Area Network (VLAN) tag, an MPLS Label, or any other header field such as Type of Service (ToS).
- *Information Element (IE)*: A structured data field used to convey specific types of information within control gRPC signaling. Each IE encapsulates essential data required for the management and control of packet flows, represented as a tuple (type, length, value), where type denotes the information kind (e.g., IP address or protocol), length specifies the data size, and value contains the actual content.
- *PDR*: A set of rules or criteria used to identify and handle packets in a specific manner within the Data-Path. Each PDR defines conditions (i.e., Match Fields) and actions (ie., FAR and LBR) for packet processing.
- *FAR*: Defines the actions to be taken on packets matching a PDR, such as forwarding, buffering, or duplicating packets.
- *LBR*: A rule designed to distribute network traffic across multiple paths or endpoints efficiently, often for load balancing, redundancy, or failover purposes.

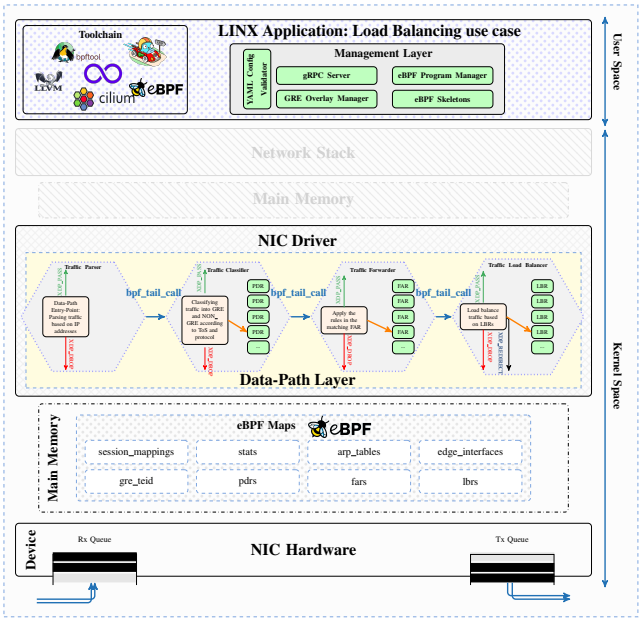


Fig. 2: LINX Architecture Details

Schema 1 Flow Rules' Data-Model/Grammar

```
request: operand rules
operand: create | update | delete
rules: rule [,rules]
rule: PDR | FAR | LBR | [IEs]
IEs: IE [,IEs]
IE: (type, length, value)
length: u8 | u16 | u32 | u64
value: CONST (decimal)
typ: IP address | MAC address | protocol |
ToS | action | interface | interface index
PDR: (IP address, u32, src_ip), (IP
address, u32, dst_ip), (protocol, u8,
TCP|UDP), (ToS, u8, CONST)
FAR: (action, u8, REDIRECT|PASS|DROP),
(interface, u32, GRE-WAN1|GRE-WAN2|..|
GRE-WANi|NON-GRE), (ifIndex,u32, 1|2|..|i+1)
LBR: FAR, (ToS, u8, CONST)
```

D. Data-Path Traffic Processing

The in-kernel packet processing journey begins at the NIC Receiver (Rx) queue where the NIC's Direct Memory Access (DMA) triggers a Hardware Interrupt Request (IRQ) (HardIRQ), invoking the NIC Driver's IRQ handler. This handler then initiates the New API (NAPI) subsystem via a Software IRQ (SoftIRQ), starting packet processing via the driver's registered poll function, which implements the XDP hook for eBPF XDP program. To ensure uninterrupted processing, the NIC driver disables further IRQs. The eBPF XDP program, executed by the XDP hook, marks the entry-

point for the created XDP pipline. Starting execution, the XDP program accesses a context object metadata, encapsulated within the optimized *xdp_md struct* [3]. Following data parsing, control may transfer to other XDP programs via *bpf_tail_call* function [4]. After parsing, metadata fields can be read from the context object, which also allows attachment of custom metadata [5]. In this regard, XDP programs access persistent data structures (*eBPF maps*) via functions such as *bpf_map_lookup_elem* or *bpf_map_update_elem* [6], [7]. A final verdict is issued, determining packet handling, such as dropping, re-transmission, kernel processing, or redirection. Once packet processing is completed, the NAPI subsystem is deactivated, and IRQs from the NIC device are re-enabled. This treatment is summarized in Algorithm 1

Algorithm 1 Data-Path Packet Processing

```
Receive Packet on Rx queue
DMA Triggers HardIRQ
Invoke NIC Driver's IRQ Handler
Initiate NAPI Subsystem (SoftIRQ)
Start Packet Processing via Driver Poll
Trigger XDP pipeline
Packet enters XDP_Parser
bfp_tail_call XDP_Classifier
bfp_tail_call XDP_Forwarder
bfp_tail_call XDP_Load_Balancer
Redirect packet to TX queue
```

IV. METHODOLOGY AND TESTBED

A. Experimental Setup

We are interested into subjecting our solution to rigorous stress testing to measure its resilience, using RFC 2544-like tests [8]. These tests are designed to evaluate the throughput, latency, and overall performance of the solution under various conditions. The System Under Test (SUT) consists of a *loopback* between two devices: one hosting a traffic generator such as TRex, and the other, the Device Under Test (DUT), hosting the solution to test, in our case LINX. TRex [9] is an open-source realistic traffic generator powered by DPDK, used to measure the maximum sustainable throughput of a DUT.

B. Experimental Platform

To evaluate the performance of the HELIOS framework we utilized two identical hosts for the testing environment. One host was configured with TRex traffic generator, while the LINX solution was deployed on the second host. On both devices, we have installed an Ubuntu 22.04 operating system, with kernel version 5.15 to ensure optimal compatibility with eBPF and DPDK. Each device is powered with the 12$^{\text{th}}$ Generation of the Intel i7 embedding 12 Cores, clocking at 4.9 GHz, and using 25 MiB of L3 cache memory. Regarding the network interfaces, both hosts were outfitted with Dual-port Intel Ethernet X550T adapters, supporting 10 Gbps on each port, ensuring reliable high-speed data transmission.

C. Test Cases

We scripted Python test cases to mimic data traffic on both uplink and downlink scenarios, leveraging TRex APIs [10].

- *Uplink*: We generate UDP/TCP traffic in TRex, simulating a client sending data via one of the dual-port X550T adapters to the DUT. The LINX gateway, acting as the edge node, processes the traffic, *encapsulates* it in a GRE header, and forwards it to the appropriate WAN interface, as determined by the load balancer based on SDN controller rules. The traffic is looped back over the GRE overlay to TRex on the second port, simulating a remote-region node.
- *Downlink*: In the downlink scenario, the simulated remote-region node, represented by TRex, generates UDP/TCP traffic that is encapsulated in a GRE tunnel and sent to the DUT (i.e., LINX gateway) via the WAN interface. The DUT *decapsulates* the traffic and forwards the resulting packets to the end-user (i.e., TRex) over the LAN through the other port on the Intel X550T adapter, completing the End-to-End (E2E) delivery process.
- *Control Signaling*: The ONOS SDN Controller communicates with LINX gateways to dynamically manage and distribute the flow rules. We simulate situations where the SDN needs to update the flow rules within the remote regions, and estimate the delay.

V. PERFORMANCE

Now, we present the results of our measurement campaign regarding throughput, CPU usage, Packet Loss Rate (PLR), and control signaling latency. Across all tests, we analyze the effect of packet injection rate (f_i), packet size (s_p) ($\{100, \ldots, 1400\} \cup \{50, 1450\}$), and the number of Rx queues (c_j), where $(i, j, p) \in \{1, 2, \ldots, 10\} \times \{1, 2, \ldots, 12\} \times \{1, 2, \ldots, 16\}$. We used the 'irqbalance' service to distribute incoming traffic evenly across the active Rx Queues, with each Rx Queue pinned to a single CPU core to optimize performance. Each test configuration was repeated $50\times$ to ensure accuracy and consistency.

a) Throughput: Figure 3 shows the *peak* throughput (in MPackets/s) obtained for downlink (Figure 3a) and uplink (Figure 3b) scenarios (see Section IV-C) function of s_p and c_j. All curves exhibit a similar pattern, decreasing smoothly from 10 MPackets/s and converging toward 1 MPackets/s. This trend resembles an exponential decay function, with throughput gradually declining and approaching an asymptote. The similarity between the uplink and downlink curves suggests that direction has minimal impact on throughput behavior; however, packet size s_p and, to a lesser extent, the number of Rx Queues c_j, appear to influence the results. Indeed, the more s_p increases, the more throughput decreases, which is intuitive given that throughput is measured in MPackets/s. With a fixed 10 Gbps bandwidth, the maximum achievable throughput is approximately 10 MPackets/s (according to TRex performance) for 50-Byte packets; as packet size grows, the packet rate declines, even though the bandwidth remains fully utilized.

The impact of c_j appears minimal with the current SUT, but we anticipate that with a more powerful SUT, this impact will become evident. Transmitting hundreds of millions of packets would more clearly demonstrate the importance of multiple Rx Queues in handling high packet volumes efficiently.

b) CPU Load: Table I displays the average CPU usage in percentage as a function of f_i and c_j. The results reveal a trend where, the higher f_i, the greater CPU load will be across all configurations. At higher packet rates, particularly beyond $f4$ (4 MPackets/s), CPU usage nears saturation with a single Rx Queue (i.e., c_1), indicating inefficiency under high packet rates. In contrast, increasing the c_j value *significantly* improves CPU efficiency. For example, in configuration (f_4, c_1), CPU load is 99.23%, reaching 100% in (f_7, c_1) and higher. However, this load drops to 19.19% with only two Rx Queues. With $c_j \geq 6$, CPU load remains below 30% and increases more slowly with further packet injections, showing better efficiency and reduced risk of saturation. Configurations c_8 to c_{12} exhibit the lowest CPU usage, with c_8 at 20.06% and c_{12} at 7.84% for 10 MPackets/s, highlighting the effectiveness of multiple Rx Queues in reducing CPU load even at high packet rates.

TABLE I: CPU Load Function of Packet Rates and Rx Queues

Injections [MPackets/s]	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
c_1	7.92	10.79	31.41	99.23	99.33	99.73	100	100	100	100
c_2	2.88	5.56	9.35	19.19	65.15	68.34	98.41	98.87	98.95	99.98
c_3	1.86	3.52	5.74	10.63	15.21	23.22	68.38	76.85	93.11	93.29
c_4	1.82	2.75	5.67	7.47	9.39	12.29	23.45	30.76	83.85	84.49
c_5	1.46	2.48	4.98	4.39	7.78	15.67	24.30	27.51	45.09	45.67
c_6	1.19	2.36	3.44	3.62	6.34	6.35	17.51	18.32	29.56	29.86
c_7	1.12	1.48	2.84	4.68	4.69	5.65	16.14	17.10	24.88	25.17
c_8	1.01	1.84	3.18	3.46	5.38	5.80	15.43	16.88	19.78	20.06
c_9	1.09	1.46	2.45	3.18	3.59	4.07	9.48	10.01	15.98	16.69
c_{10}	1.01	1.28	2.45	3.35	3.64	4.28	8.32	9.62	11.34	11.69
c_{11}	1.20	1.24	2.33	3.16	3.23	5.34	5.98	6.57	8.7	8.9
c_{12}	1.10	1.18	2.12	2.91	3.11	3.27	3.89	4.48	7.28	7.84

c) Control Signaling Latency: Figure 4 depicts the Cumulative Distribution Function (CDF) obtained for the control signaling scenario, derived from a dataset of $1,000$ values representing the delay ratio achieved in less than $x\,ms$. The results show a low-latency distribution characterized by minimal variation with the following statistical measures: a mean of $0.8155\,ms$, variance of $0.3539\,ms^2$, standard deviation of $0.5949\,ms$, and a coefficient of variation of 0.7295. We find that approximately 50% of CP operations complete within $0.7\,ms$, illustrating that half of all operations complete under $1\,ms$, while the 10^{th} percentile is as low as $0.146\,ms$, indicating that a significant portion of operations are nearly instantaneous. The curve progresses smoothly, with 90% of operations achieving latency below $1.6\,ms$. The upper tail of the CDF reaches $2.6\,ms$, indicating the maximum observed latency, which likely represents rare worst-case delays. Overall, these results indicate reliable low-latency performance ideal for real-time applications requiring immediate rerouting or policy adjustments.

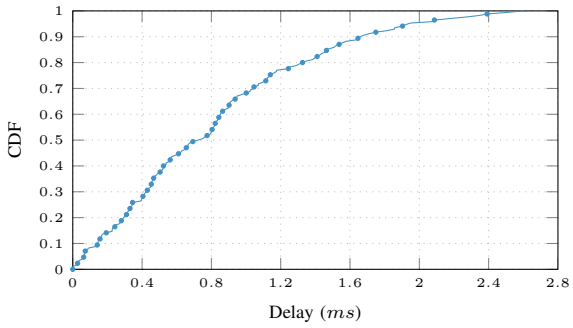


Fig. 4: Control Signaling Latency

d) Packet Loss Rate: Figure 5 presents the PLR (in percentage) plotted against packet size (s_p) and Rx Queue count (c_j) for the uplink scenario, where we fully utilize the system's available bandwidth (i.e., 10 Gbps). For each s_p, we have represented 12 PLR average values corresponding to the values of c_j, we thus obtained s_p-dependent c_j-dependent global measures. We have made several observations from the graph: *(i)* - The observed PLR trend resembles an exponential decay function, where the rate of decrease is rapid for smaller packet sizes and gradually levels off for larger packet sizes, approaching an asymptote rather than continuing to decrease linearly. *(ii)* - The c_j-dependent PLR values generally converge except for c_1 and c_2. In these cases, the high traffic volume- approximately 10 and 8 MPackets/s, for packet sizes of 50 and 100 Bytes, respectively- overwhelms a single Rx Queue, resulting in increased PLR. *(iii)* - With a sufficient number of Rx queues ($c_j \geq 3$), the PLR remains consistently below 0.15% and eventually stabilizes around 0.03%. This value is negligible in comparison to the total number of transmitted packets, indicating highly efficient packet delivery. *(iv)* - The PLR is impacted by the packet volume, not size. For instance, with 50-Byte packets, we can generate *in practice* around $10,$ MPackets/s (based on TRex experience). In contrast, larger packets like 1450 Bytes result in fewer packets per second, reducing the load on each RX queue, since this later can only handle a certain packet rate before overflow.

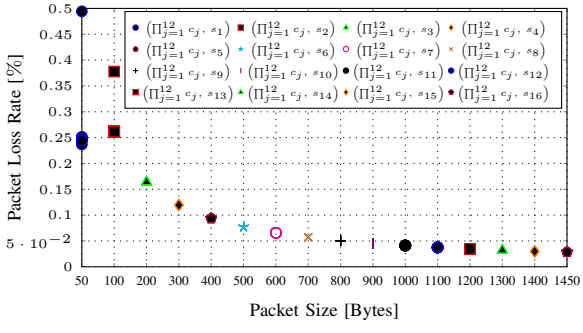
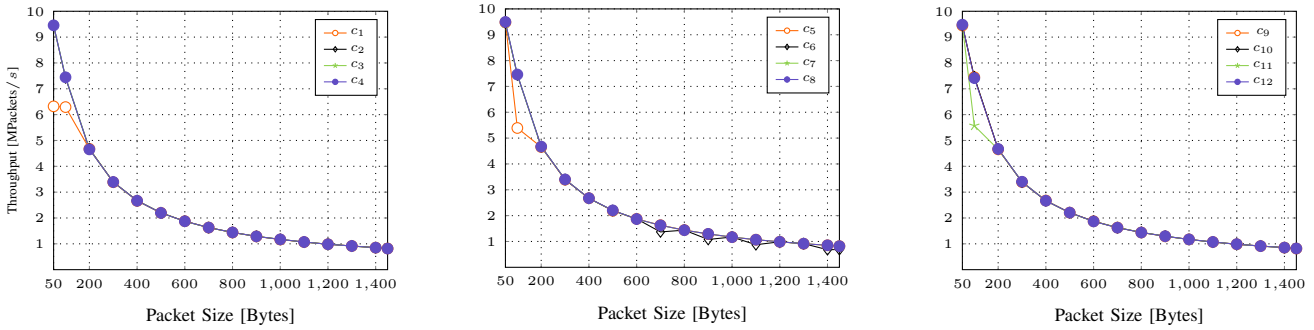
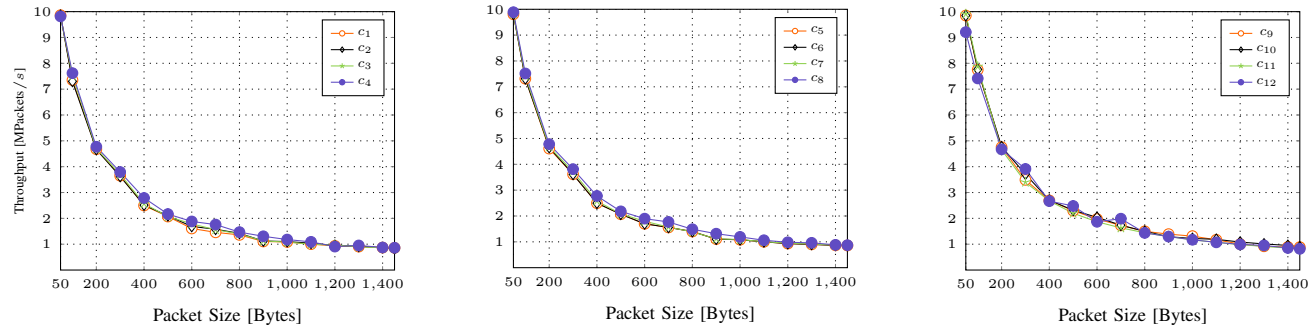


Fig. 5: PLR per Packet Size and Number of Rx Queues

To sum up our findings in this paper; we first found that eBPF XDP-based solutions are primarily influenced by the



(a) Downlink (GRE Headers Decapsulation)



(b) Uplink (GRE Headers Encapsulation)

Fig. 3: Throughput Obtained on the Downlink and Uplink per Packet Size and Number of Rx Queues

volume of injected traffic and the number of Rx Queues, while packet size affects throughput only at the injection phase.

VI. Conclusion

In this paper, we introduced the HELIOS architecture, an eBPF-based SD-WAN solution tailored for high-performance applications within CEC networks. We detailed the design of the LINX framework, which implements essential functionalities such as load balancing and firewalling on Edge Gateways. This solution effectively enables load balancing, protocol/ToS-aware traffic redirection across multiple GRE overlays, and early-stage filtering and dropping of packets. Our experimental results emphasize the need to reconsider both SD-WAN solutions and other DP components, like the 5G User Plane Function (UPF). The findings show significant improvements in throughput, PLR, and resource utilization, underscoring the potential of our solution to enhance network efficiency and performance.

Acknowledgment

This work was partially supported by the European Union's Horizon Europe Research and Innovation program AC^3 project under grant agreement No 101093129.

References

[1] A. Mokhtari and A. Ksentini, "Sd-wan for cloud edge computing continuum interconnection," in *Proceedings of IEEE Globecom 2024, Cap Town, South Africa*, 2024.

[2] H. S. D. Y. E. Standardization and W. D. S.-W. M. Growth, "Mef leads sd-wan service standardization & certification."

[3] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. da Silva Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and XDP: concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 16:1–16:36, 2021.

[4] eBPF, "bpf_tail_call," [online] available at: https://docs.ebpf.io/linux/helper-function/bpf_tail_call/ (last checked: Oct. 2024).

[5] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The express data path: fast programmable packet processing in the operating system kernel," in *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, X. A. Dimitropoulos, A. Dainotti, L. Vanbever, and T. Benson, Eds. ACM, 2018, pp. 54–66.

[6] eBPF, "bpf_map_lookup_elem," [online] available at: https://docs.ebpf.io/linux/helper-function/bpf_map_lookup_elem/ (last checked: Oct. 2024).

[7] ——, "bpf_map_update_elem," [online] available at: https://docs.ebpf.io/linux/helper-function/bpf_map_update_elem/ (last checked: Oct. 2024).

[8] S. Bradner and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices," RFC 7560, 1999.

[9] Cisco, "TRex Stateless API Reference," 2024, [online] available at: https://trex-tgn.cisco.com/ (last checked: Mai 2024).

[10] ——, "TRex Stateless API Reference," 2015, [online] available at: https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/api/index.html (last checked: Mai 2024).