`eZtunnel`: Leveraging eBPF to Transparently Offload Service Mesh Data Plane Networking

Arthur J Simas, Fabricio E Rodriguez Cesen, Christian Esteve Rothenberg
Universidade Estadual de Campinas (UNICAMP)
Campinas, Brazil
a249927@dac.unicamp.br, f163682@dac.unicamp.br, chesteve@dca.fee.unicamp.br

***Abstract*—Cloud-native applications, characterized by scalability, resilience, and flexibility, adopt microservices architectures to decompose applications into smaller, independently manageable services. Although microservices offer significant benefits, this architectural approach introduces challenges in service-to-service communication, commonly relying on advanced orchestration and communication frameworks such as Kubernetes and Istio, respectively. However, the added complexity imposes substantial overhead by introducing longer packet processing paths. This paper discusses performance bottlenecks arising from service meshes and proposes eZtunnel to address some of the identified challenges. Leveraging *extended Berkeley Packet Filter* (eBPF) to transparently offload networking traffic, the elongated network path is bypassed, optimizing resource utilization and enhancing application performance. Experiments show that eZtunnel can reduce median latency over 20% and jitter to almost 10%.**

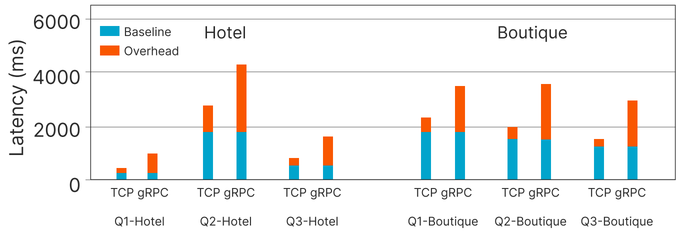
***Index Terms*—Service Mesh, Kubernetes, eBPF, Offloading**

I. Introduction

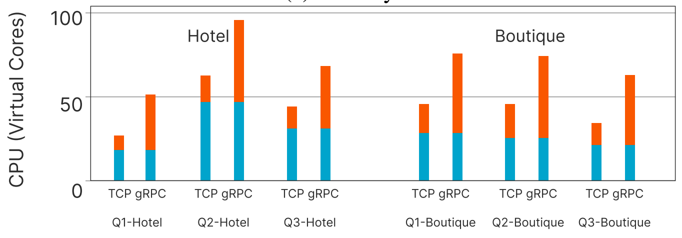
In the rapidly evolving landscape of cloud computing, the adoption of cloud-native technologies represents a fundamental shift in how applications are developed, deployed, and managed [1], [2]. These applications are commonly based on microservices architectures to make the most of cloud environments' scalability, resilience, and flexibility.

To manage a large number of microservices, Kubernetes and a service mesh are often used to, respectively, (*i.*) orchestrate containerized applications across diverse environments [3] and (*ii.*) provide advanced inter-service communication features, such as service discovery, load balancing, encryption, authorization, and observability [4].

Service meshes often implement the sidecar deployment mode, which injects a proxy beside each service. This mode dramatically degrades performance, particularly when the Kubernetes cluster orchestrates many services [5], [6]. To overcome this problem, some service mesh proposals introduce the concept of a per-node shared agent [7]. However, both deployment modes present increased overheads due to a longer communication path. As highlighted in Fig. 1, it may result in higher latency and CPU consumption, ultimately degrading the Quality of Service (QoS) [5].



(a) Latency



(b) CPU usage

Figure 1: Measurements of latency and CPU overheads caused by using Istio Service Mesh compared to the baseline scenario without a service mesh. The Hotel Reservation [8] and Online Boutique [9] applications were used as workload under three different queries. Adapted from [5].

The root cause of this issue is the excessive traversals of the kernel network stack caused by the use of middleware between services, which results in inefficient resource usage [6].

To this end, we propose a transparent offloading technique to enable efficient communications in service meshes, independent of the proxy deployment mode (sidecar or per-node shared agent). Our proposal leverages the *extended Berkeley Packet Filter* (eBPF) as the key technology to address and mitigate the problem. This approach involves deploying eBPF programs to intercept and redirect packets at the kernel level, bypassing the kernel network stack. Our experiments show that, compared to the default setup of a well-known service mesh, Istio [4], we can reduce request latency by up to 21.35% and jitter by 9.78%. In summary, the following are the key contributions of this paper:

- We investigate the state-of-the-art service mesh offloading to find open gaps and improvement opportunities;
- We present a transparent offloading method based on eBPF to improve microservice communication in diverse service mesh environments;

- We conduct an experimental evaluation of our implementation on a testbed deployed with Kubernetes and Istio to assess our approach's potential to alleviate service mesh overheads.

The remainder of this manuscript presents the background and related works (Section II), the proposed offloading strategy design and implementation (Section III), and its respective experimental results (Section IV). Finally, we conclude and discuss the future work (Section V).

II. Background and Related Work

In this section, we introduce the background of service meshes, the ambient mesh alternative, and eBPF, then relate our work to the prior efforts found in the literature.

A. Service Mesh

The granularity and dynamism of microservices introduce challenges in service-to-service communication, security, and observability [10]. To address these complexities in cloud-native applications, service meshes have been introduced.

A service mesh, usually implemented with sidecars (*i.e.* proxies alongside each service instance), provides a dedicated software layer for handling advanced inter-service communication features without altering the application code. Fig. 2 presents a typical design of a service mesh.

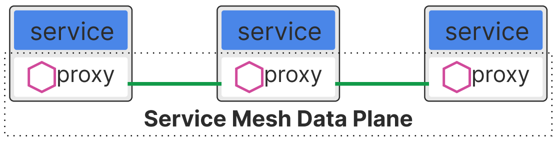


Figure 2: Sidecar-based service mesh architecture.

B. Ambient Mesh

The majority of service mesh data plane implementations use sidecars. Deploying a sidecar per-service results in underutilization of resources across the cluster as the number of services increase. The per-node shared agent design, called *ambient mesh* in Istio, aims to address this issue [7].

As shown in Fig. 3, a shared agent detaches the growing relation between service and sidecar count, allowing the Kubernetes cluster to schedule more services without incurring performance penalties due to an injected proxy per service.

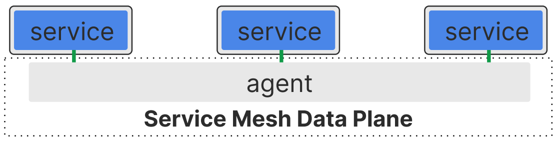


Figure 3: Per-node shared agent service mesh architecture.

C. eBPF - extended Berkeley Packet Filter

The eBPF technology enables a safe and efficient execution of custom programs within the Linux kernel, extending its capabilities without recompiling the kernel [11].

There are different eBPF program types available, each specific for a given aspect of the kernel. They can be attached to various pre-defined hooks, including syscalls, function entry or exit, tracepoints, network events, and custom hook points.

This flexibility makes diverse use cases possible, notably in networking. It can be used to perform packet filtering, manipulate network traffic, and monitoring, allowing real-time, low-latency processing of network packets without forwarding them to user-space processes.

In this regard, eBPF Maps allow programs to exchange data, which are key-value data objects stored in-kernel. User-space programs are also allowed to access these kernel structures.

D. Prior work

The landscape of network offloading is rapidly evolving, driven by the need to enhance the performance, security, and scalability of Kubernetes and other cloud environments.

Within this context, various approaches have been explored to optimize networking and reduce overheads. X-IO [12], a high-performance I/O interface, aims to eliminate kernel networking overheads and contention of microservices using shared memory processing. Although it offers a 2.8∼4.1× latency improvement, it requires changes to the application code and is unable to run alongside a service mesh, lacking many benefits provided by it.

Another proposal, SPRIGHT [13], a serverless framework, makes use of shared memory and eBPF to improve the scalability of the data plane. It exhibits competitive performance results of 53× latency reduction and 27× CPU usage savings compared to Knative. Despite that, it also suffers from the lack of service mesh features.

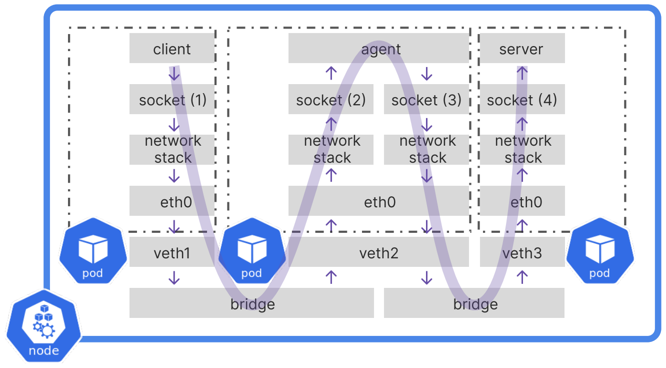
As an alternative, Cilium [14], a sidecar-free service mesh, heavily uses eBPF to implement its features. The downside is that it requires the usage of its own Container Network Interface (CNI). In this sense, clusters deployed with other CNIs cannot use Cilium.

Finally, [6] presents a network optimization based on eBPF to bypass the kernel network processing. The work improves request latency by up to 21% for 90% of requests. Still, it only works for Istio in sidecar mode.

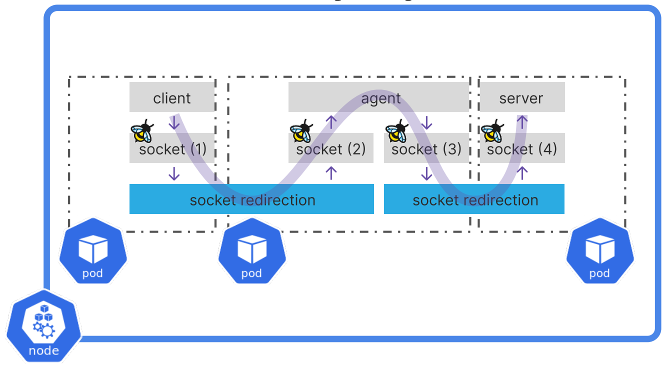
Compared to related work, our eZtunnel proposal offers a better approach to optimize service mesh data plane networking. It works with the newest service mesh proposals, like ambient mesh, offloads the network transparently, and does not require redeployment of the cluster.

III. eZtunnel: Design and Implementation

This section presents eZtunnel, a transparent service mesh data plane networking offload using eBPF. We introduce the architecture overview, the design, and the implementation.



(a) Default packet path



(b) Packet path with our proposed solution

Figure 4: Packet path of a per-node shared agent service mesh with and without our proposed solution.

A. Architecture Overview

The core problem we aim to address is the excessive traversal of the kernel network stack. Fig. 4a depicts the packet's path from a client to a server process in a shared agent-based service mesh environment.

Initially, a message is written to socket (1). It traverses the network stack down to the network interface, where a virtual bridge forwards the packet to the respective interface of the agent's pod. It then traverses again the network stack, and socket (2) delivers the message to the process. The same process is repeated from socket (3) to socket (4) to send the response back from the server to the client.

Our design to shorten the packet's path involves a socket redirection mechanism based on eBPF, as illustrated in Fig. 4b. Instead of traversing the network stack, network interfaces, and virtual bridges, as the packet is written to the socket, it is directly redirected to the other socket's end and delivered to the process, entirely bypassing the intermediary kernel network processing. This mechanism works regardless of the service mesh deployment mode (*e.g.*, sidecar-based or per-node agent).

B. Design

To route messages between sockets and skip the Linux network stack, sockets must first be captured, stored, and monitored for messages.

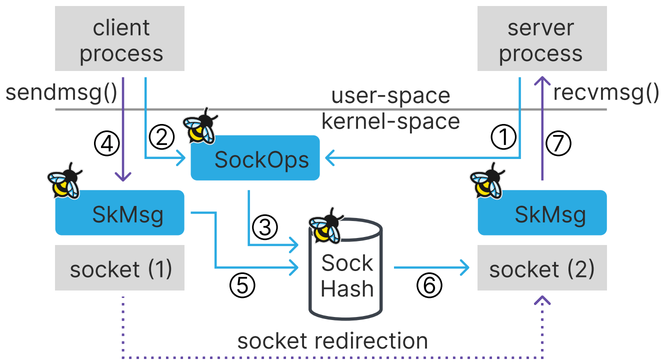


Figure 5: Socket redirection workflow using eBPF.

A *SockOps* program is attached to *cgroups*, which responds to socket events (*e.g.*, socket established). It allows us to change socket parameters and opportunistically **store** them in an eBPF Map [15]. eBPF provides a variety of Maps, including *SockHash*, which we use to store sockets in a hash table with a user-defined key.

A *SkMsg* program is attached to *SockHash* to handle messages sent through one of the stored sockets, *i.e.* when '`sendmsg`' and '`sendfile`' syscalls are executed on sockets that are part of the Map the *SkMsg* program is attached to.

Fig. 5 exemplifies the workflow. ① When the server socket is created, it is captured by *SockOps* program and ③ stored in *SockHash* Map. ② The client socket is also captured and ③ stored in the Map. ④ When a message is written to the socket, ⑤ the *SkMsg* program attached to the Map is triggered and ⑥ redirects it to the corresponding socket, ⑦ delivering it to the server process.

C. Implementation

The implementation is based on the *aya* library [16], an eBPF library built entirely in Rust. It offers a compile-once, run-anywhere solution independent from the Linux distribution or kernel version. The compilation process uses *cargo*, and the crates *bpf-linker* and *bindgen-cli*. When the user-space code is executed, the compiled eBPF program is loaded into the kernel without requiring additional tooling.

IV. Experimentation Evaluation

In this section, we present the experimental settings, metrics of interest, and results obtained from our implementation.

A. Testbed setup

a) Server Specifications: The server used to run the experiments has the following settings:

- CPU: AMD Ryzen 7 4700U, base clock 2 GHz, boost clock up to 4.1 GHz, 8 cores, 8 threads;
- RAM: 2x 8 GB, DDR4, 2667 MHz;
- Swap Memory: 22 GB;
- Storage: SSD NVMe 256 GB;
- Operational System: Fedora 36;
- Kernel: Linux 6.2.15, cgroup2 enabled.

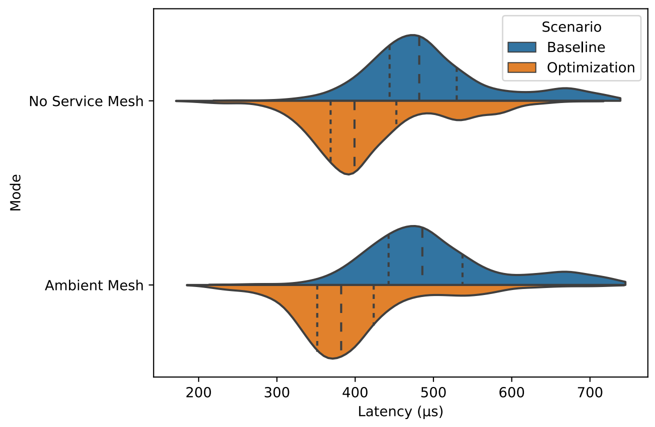


Figure 6: Latency distribution plot.

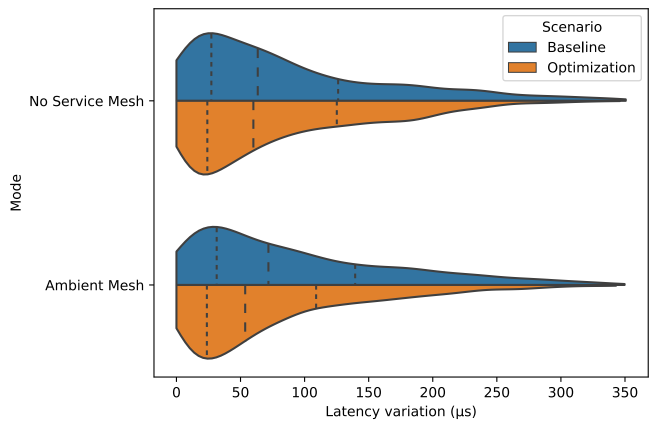


Figure 7: Latency variation distribution plot.

Table I: Latency quartiles values summary

		Latency		
Mode	**Scenario**	p25 (µs)	p50 (µs)	p75 (µs)
No Service Mesh	Baseline	443.98	481.83	529.71
	Optimization	368.61 (−16.98%)	399.15 (−17.16%)	452.57 (−14.56%)
Ambient Mesh	Baseline	442.69	485.82	537.12
	Optimization	351.51 (−20.60%)	382.10 (−21.35%)	423.66 (−21.12%)

Table II: Jitter values summary

Mode	**Scenario**	**Jitter (µs)**
No Service Mesh	Baseline	83.25
	Optimization	80.86 (−2.87%)
Ambient Mesh	Baseline	88.44
	Optimization	79.79 (−9.78%)

b) Kubernetes Cluster: The cluster is composed by a single node, running Kubernetes v1.29.2.

c) Service Mesh: We setup the Kubernetes Cluster with the Istio Service Mesh v1.21.2. Istio supports both sidecar (default) and per-node shared agent (ambient mesh) deployment models.

B. QoS Metrics

Our experiment observed three QoS metrics: latency, latency variation, and jitter:

a) Latency: To capture network delay, we measure the difference between the time the packet was sent and the time a response was received. This metric is also called 'Round-Trip Time latency' (RTT latency).

b) Latency variation: It captures the absolute difference between subsequent latency measurements.

c) Jitter: This metric is derived from network latency, defined by the average deviation from network mean latency.

C. Results

The benchmark results were produced using a workload consisting of a client and a server pod, sending requests in intervals of 100 milliseconds. We ran the workloads without the service mesh and with ambient mesh deployment mode.

Observing the latency plotted in Fig. 6, we can find a significant improvement when enabling the zTunnel optimization. The p75 latency in the optimized scenario is slightly near or below the p25 latency in the baseline. This represents a latency reduction of up to 21.35%. The values summary is presented in Table I. This reduction indicates that, even under higher load scenarios, three-quarters of all requests are now processed as quickly as the fastest one-quarter of requests were handled before the optimization.

We also analyze jitter to assess the impact of our optimization on latency stability. As observed in Fig. 7, latency variation quartiles are consistently lower than the baseline, and jitter decreased by as low as 9.78%, as in Table II. This reduction in jitter underscores our solution's ability to enhance network performance predictability and reliability without introducing instability.

The experimental outcomes presented in this section assert the viability of eZtunnel to significantly enhance networking performance in service meshes, reducing latency, jitter, and maintaining system stability.

V. Conclusions and Future Work

In this paper, we propose eZtunnel, a transparent offloading solution for shortening the communication path in cloud-native environments running a service mesh deployment. Using eBPF allows for substantially improving communication QoS and application performance. The results of our experimental evaluation in an Istio deployment show promising outcomes. Using our offloading solution, we achieved a significant 21.35% latency improvement. Furthermore, eZtunnel reduced jitter by up to 9.78%.

Future work seeks to expand the testing benchmark to multi-node cluster configurations, monitor additional metrics, and add more workloads, such as those from 5G core and user plane functions [17]. Additionally, we will demonstrate the support for other service meshes and assess the security implications of bypassing the kernel network stack in complex environments.

Supplementary Material

Code repository and benchmark results of this research are available at: `github.com/smartness2030/eztunnel`

Acknowledgment

This work was supported by *Ericsson Telecomunicações LTDA*, and by the *São Paulo Research Foundation (FAPESP)* FAPESP, grant `2021/00199-8`, CPE SMARTNESS. This study was partially funded by *CAPES*, Brazil - Finance Code `001`.

References

[1] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native applications," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 16–21, 2017. DOI: 10.1109/MCC.2017.4250939.

[2] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-native workload orchestration at the edge: A deployment review and future directions," *Sensors*, vol. 23, no. 4, 2023, ISSN: 1424-8220. DOI: 10.3390/s23042215.

[3] "Kubernetes," Kubernetes. (), [Online]. Available: https://kubernetes.io.

[4] "Istio," Istio Authors. (), [Online]. Available: https://istio.io/.

[5] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan, "Dissecting overheads of service mesh sidecars," in *Proceedings of the 2023 ACM Symposium on Cloud Computing*, ser. SoCC '23, Santa Cruz, CA, USA: Association for Computing Machinery, 2023, pp. 142–157. DOI: 10.1145/3620678.3624652.

[6] W. Yang, P. Chen, G. Yu, H. Zhang, and H. Zhang, "Network shortcut in data plane of service mesh with ebpf," *Journal of Network and Computer Applications*, vol. 222, Feb. 2024, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2023.103805.

[7] J. Howard, E. J. Jackson, Y. Kohavi, I. Levine, J. Pettit, and L. Sun. "Introducing ambient mesh: A new dataplane mode for istio without sidecars," Istio Authors. (Sep. 2022), [Online]. Available: https://istio.io/latest/blog/2022/introducing-ambient-mesh/.

[8] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19, Providence, RI, USA: Association for Computing Machinery, 2019, pp. 3–18, ISBN: 9781450362405. DOI: 10.1145/3297858.3304013.

[9] "Microservices-demo: Sample cloud-first application with 10 microservices showcasing kubernetes, istio, and grpc," Google. (), [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo.

[10] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2019, pp. 122–1225. DOI: 10.1109/SOSE.2019.00026.

[11] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. 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, Feb. 2020, ISSN: 0360-0300. DOI: 10.1145/3371038.

[12] S. Qi, H.-S. Tsai, Y.-S. Liu, K. K. Ramakrishnan, and J.-C. Chen, "X-io: A high-performance unified i/o interface using lock-free shared memory processing," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 107–115. DOI: 10.1109/NetSoft57336.2023.10175428.

[13] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan, "Spright: Extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing," in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22, Amsterdam, Netherlands: Association for Computing Machinery, 2022, pp. 780–794, ISBN: 9781450394208. DOI: 10.1145/3544216.3544259. [Online]. Available: https://doi.org/10.1145/3544216.3544259.

[14] T. Graf. "Cilium service mesh – everything you need to know," Isovalent. (Jul. 2022), [Online]. Available: https://isovalent.com/blog/post/cilium-service-mesh/.

[15] *eBPF-Docs*, eBPF-Docs Authors. [Online]. Available: https://ebpf-docs.dylanreimerink.nl/linux.

[16] *Aya – eBPF library for the Rust programming language*, The Aya Contributors. [Online]. Available: https://aya-rs.dev/.

[17] T. A. Navarro do Amaral, R. V. Rosa, D. F. C. Moura, and C. Esteve Rothenberg, "Run-time adaptive in-kernel bpf/xdp solution for 5g upf," *Electronics*, vol. 11, no. 7, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11071022. [Online]. Available: https://www.mdpi.com/2079-9292/11/7/1022.