IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 13 October 2025, accepted 30 November 2025, date of publication 3 December 2025,
date of current version 15 December 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3639761

RESEARCH ARTICLE

Mixed-Runtime Pod Networking for Kubernetes-Based Edge Computing

TOM GOETHALS, (Member, IEEE), MIEL VERKERKEN, MERLIJN SEBRECHTS, (Member, IEEE), FILIP DE TURCK, (Fellow, IEEE), AND BRUNO VOLCKAERT, (Senior Member, IEEE)
Department of Information Technology, IDLab, IMEC, Ghent University, 9052 Gent, Belgium
Corresponding author: Tom Goethals (tom.goethals@UGent.be)

This work was supported in part by the Flanders Research Foundation (FWO) through Junior Postdoctoral Researcher under Grant 1245725N, and in part by the NATWORK Horizon Europe Project.

ABSTRACT In recent years, microservice-based applications have shifted towards cloud-edge computing, taking advantage of green energy, lower latencies, and local computational resources. Simultaneously, various alternatives to containers, such as unikernels and WebAssembly, have emerged and are gradually being adopted by container-based orchestration software such as Kubernetes. However, solutions that integrate such alternatives are not generic: they only integrate a single specific non-container option, and generally do not allow for mixed-workload pods with both container and non-container workloads. Importantly, no solution is currently capable of integrating mixed-workload pods with all the features of pod networking as standardized by Kubernetes. This article presents a generic, runtime-agnostic solution which provides mixed-workload pods with all the standard behavioral features of pod networking. The core functionality of the proposed solution is sub-pod networking, which sets up small independent subnets for each pod rather than providing them a single IP address, and which is based on eBPF programs for efficient and secure traffic processing. The architecture is implemented in Feather, a Kubernetes-compatible orchestration agent which supports both containerd and OSv unikernels, and evaluations show that the end-to-end latency overhead of the implementation for various types of network traffic ranges from just 22μs to 200μs, and scales predictably with the number of non-container workloads in a pod. Additionally, the implementation has no significant memory overhead, and CPU overhead is 2Gbps to 6.4Gbps per percentage of a single core, depending on runtime and evaluation scenario.

INDEX TERMS Edge computing, container networking, pod networking, eBPF, edge kubernetes.

I. INTRODUCTION

Recently, microservice-based applications have shifted from cloud-only clusters into the cloud-edge continuum, to take advantage of better response times, efficient use of local resources and green energy, and improved privacy [1], [2]. However, this requires extending orchestration control planes and container networking to edge devices. Although it is possible to run orchestration software such as Kubernetes directly on resource-constrained devices, this approach often takes up too much device memory to be feasible [3]. Alternatives such as KubeEdge and Azure IoT integrate edge devices with cloud infrastructure through a set of dedicated agents and components, but this approach is sensitive to cloud-based bottlenecks and does not generally allow for direct device-to-device communication [4].

In parallel with the shift to the cloud-edge continuum, various alternatives to containers have emerged, promising faster start times, better computational performance, and superior isolation from the host system. For example, micro Virtual Machines (microVMs) aim to provide smaller attack surfaces than containers, relatively low resource use, and optimal process isolation by merging existing programs with a microkernel into a VM image [5]. Solutions such as

The associate editor coordinating the review of this manuscript and approving it for publication was Hadi Tabatabaee Malazi.

Firecracker, OSv, and Nanos are at the forefront of this technology. Another example is WebAssembly (WASM), which provides a custom bytecode compilation target for many existing languages [6], with WebAssembly System Interface (WASI) providing the standards to securely execute WASM on various runtimes, offering process isolation, high performance and universal portability.

However, integrating the move to cloud-edge computing with heterogeneous workloads and runtimes is non-trivial, because the practices and standardization efforts of orchestration software have historically focused on containers [7], which are generally combined into pods consisting of multiple containers forming a logical unit of services. Containers in a single pod commonly share a network namespace, allowing for localhost communication, while network traffic between pods is handled by standardized pod networks [8]. There have been several initiatives to integrate alternative runtimes into container-based orchestration, although these generally enable the use of a single type of back-end or runtime [9], [10], and do not integrate with other types of workloads in terms of pods and pod networking. Some alternatives, such as SpinKube [11], solve this issue by implementing a containerd shim [12]. At a higher level, technologies such as the Virtual Kubelet [13] allow the implementation of a custom Kubernetes kubelet which may target any combination of generic back-ends. This approach was first created to allow Kubernetes clusters to easily interface with other cloud providers and orchestrators.

The expanding range of virtualization and runtime options, each with its relative advantages, is an excellent match for the evolution towards edge computing, which harbors great computational potential in a highly heterogeneous networking and computing environment. Specific runtimes may execute more efficiently on different devices, and even a mix of runtimes and workload types may be desirable within a single pod, depending on other requirements such as security and privacy of some workloads and high performance of others. For example, the primary workload could be run in a microVM for process isolation purposes, while sidecars are run as containers to lower resource use. Additionally, this approach can reduce configuration complexity compared to splitting workloads into different pods and forming service meshes, and reduce cross-chatter as traffic for sidecar services remains local by not publicly exposing sidecar endpoints. In practical use cases, this could be applied to IoT gateways and hubs, allowing vendors or trusted parties to run certain workloads as secure microVMs, with container sidecars in the same pods to localize traffic, while leaving third party software to run as either containers or microVMs depending on trust and isolation requirements. Similarly, this could be applied to frameworks such as Home Assistant, which runs components and third-party plugins as Docker containers regardless of source or trust level. To support this, there is a need for a mixed-runtime solution in service orchestration. Feather [14] is a multi-runtime, edge-oriented Kubernetes agent based on Virtual Kubelets, however it does not intrinsically support pod networking for non-container runtimes.

To the best of our knowledge, the issue of **mixed-runtime pod networking**, allowing workloads running in different runtimes, and attached to different network interfaces to communicate both intra- and interpod, has not been solved in the state of the art. Therefore, this article poses the following research questions:

- **RQ1:** How can a pod networking solution be devised that addresses containers and non-container workloads in the same pod, while using the same cluster-facing IP address?
- **RQ2:** How can this same solution enable 'localhost' traffic between mixed-runtime workloads in the same pod, as if they were all containers using the same network interface?
- **RQ3:** What is the performance overhead of such a solution in terms of latency, processing and throughput?

To address these research questions, this article proposes a mixed-runtime networking architecture, and an implementation in Feather. The implementation supports networking for both containers and OSv unikernels in the same pod, and is highly extensible for additional future runtimes.

The rest of this article is organized as follows. Section II presents existing research related to workload runtimes and container networking, while Section III introduces high level architecture aspects and important implementation choices. In Section IV, the evaluation setup, scenarios and methodology are detailed, while the results are presented in Section V and discussed in Section VI, which also discusses ideas for future work. Finally, Section VII draws high level conclusions from the article.

II. RELATED WORK

A. RUNTIMES

Several alternatives for containers have been developed in recent years, many of which have been integrated with Kubernetes to some degree. For example, Kata Containers [15] aim to provide additional security by encapsulating containers in a micro Virtual Machine (microVM). Unikernels [16] are a different type of microVM, which compile a target program, along with only the system libraries required for its operation, into a single kernel-space image. Furthermore, unikernels are divided between POSIX-compatible and custom System Interfaces, the former being an ideal alternative for containers. Examples include OSv [17] and Nanos [18], both of which support various programming languages and POSIX binaries without modification, offering varying CPU/memory tradeoffs compared to containers. WebAssembly (WASM) and WebAssembly System Interface [6] offer a solution in-between containers and microVMs by providing an alternate compilation target for existing programming languages, the resulting bytecode of which can be run by various runtimes (custom System Interfaces) ranging from servers to microcontrollers.

Several studies have compared the performance characteristics of these alternatives, such as gVisor and Kata Containers [19], and gVisor versus microVM alternatives [20].

Various plugins and frameworks exist to integrate such alternatives into clusters, for example OSv workloads [10] or WASI as a shim [9]. Feather [14] is a variation on this concept, implementing a custom Kubernetes agent based on Virtual Kubelets [13], with the aim of supporting mixed runtime workloads (e.g. containers, OSv unikernels) side-by-side in the same pod.

B. NETWORKING

Extended Berkeley Packet Filters (eBPF) enable the development of various (networking) functions which operate in kernel space [21]. eBPF functionality is used extensively to improve network security [22], [23], for service mesh network optimization [24], and for Kubernetes networking [25], notably by Cilium [26] which provides a wide array of eBPF programs. Other eBPF applications include composable, hardware accelerated NFV functions, for example Polycube [27], and the TCP Path Changer [28] framework to extend TCP path awareness for optimal traffic routing.

Container and pod networking are highly standardized within Kubernetes clusters, enabling the use of various interchangeable plugins [8] depending on requirements such as security, performance and flexibility. Plugin design varies, but is usually based on Layer 3 (IP) routing, and eBPF for performance, with several alternatives evaluated by Koukis et al. [29] in the context of edge computing. Non-container runtimes may be integrated using various methods; shim implementations such as CWASI [9] may be able to interface directly with a CNI plugin. Kata Containers [15], on the other hand, are implemented using a custom runtime which embeds the containers of an entire pod in a microVM, and the microVM network interface is attached to the pod network. Finally, microVM workloads may integrate with a CNI to some degree, however evaluations [10] (OSv, Firecracker, etc) and guides [30] (Nabla) never deploy more than a single workload per pod, nor do they explicitly discuss (pod) networking architecture. As a microVM uses a single IP address per machine, it is doubtful whether these solutions support more than a single workload per pod from a networking perspective. Furthermore, many microVM options are designed for Function-as-a-Service (FaaS) operation in the cloud (e.g. OSv with Firecracker), which by design does not require pod networking.

Cilium [26] is highly modular and often used for innovative networking solutions in Kubernetes; however, it is generally used in CNI plugins for high performance in complex service networking applications (e.g. Google Kubernetes Engine), and for its observability and Kubernetes-oriented secure networking features. Despite its similar area of application as the proposed solution, it does not support non-container networking by default, while this manuscript presents an extensible method of integrating various runtimes into a single pod network based on eBPF.

Multus [31] is a Kubernetes meta-CNI which is capable of adding multiple network interfaces to a single pod (i.e. multi-homing). While similar in topic, this mechanism is the inverse of the proposed solution in this article - Multus is deployed as a CNI and adds alternative network interfaces (and routing) to existing container pods, whereas the proposed solution merges the network interfaces from several workload instances in different runtimes into a single pod-wide network interface with no necessary modifications to the Kubernetes cluster or its existing CNI plugins.

Confidential Containers [32] is a Kubernetes Operator leveraging Kata Containers and Trusted Execution Environments (TEEs) to provide secure execution of cloud-native workloads. However, it requires extensive modification of existing Kubernetes cluster to support its functions, and does not provide the level and flexibility of cross-runtime networking the proposed solution aims to.

KubeEdge [4] is a Kubernetes-based platform extending across the cloud-edge continuum, however Kubernetes networking and pod practices are only used in the cloud; edge devices communicate through a bus architecture mediated by cloud components, and thus cannot communicate directly with each other. Warrens [33] leverages eBPF and a novel IP addressing scheme to form ad-hoc tunnels between nodes and networks, allowing efficient point-to-point container traffic across private networks in the edge without the need for cloud gateways or proxies.

In contrast to the alternatives discussed, the solution in this manuscript provides a universal pod networking solution without resorting to cloud components, potential cloud bottlenecks, or custom runtimes. Using eBPF and a fraction of the IP address pool of each node managed by the solution, it instead creates a sub-pod networking infrastructure that integrates any potential backend or workload runtime into a uniform container- and pod network, with public facing CNI compatibility.

III. SOLUTION ARCHITECTURE

This section describes our solution architecture, and its implementation as a networking option in Feather [14]. By default, Feather supports running container and OSv unikernel workloads side-by-side in a single pod, and the solution presented in this manuscript allows the full integration of such mixed-runtime pods with a Kubernetes pod network. While Kubernetes pods are normally defined as a logical set of containers which combine into a single self-sufficient service (e.g. a pod with a main service, a logger, and database sidecar), for the purposes of this article a pod is defined as a collection of **workloads**, each of which may use any runtime (i.e. containerd, KVM/qemu).

Fig. 1 shows the high-level component overview of the proposed architecture, integrated into Feather. Existing components are indicated by yellow, while novel components are green. Feather supports mixed runtime workloads (e.g. containers, OSv unikernels) side-by-side in the same pod, unifying aspects such as volume mounts, resource

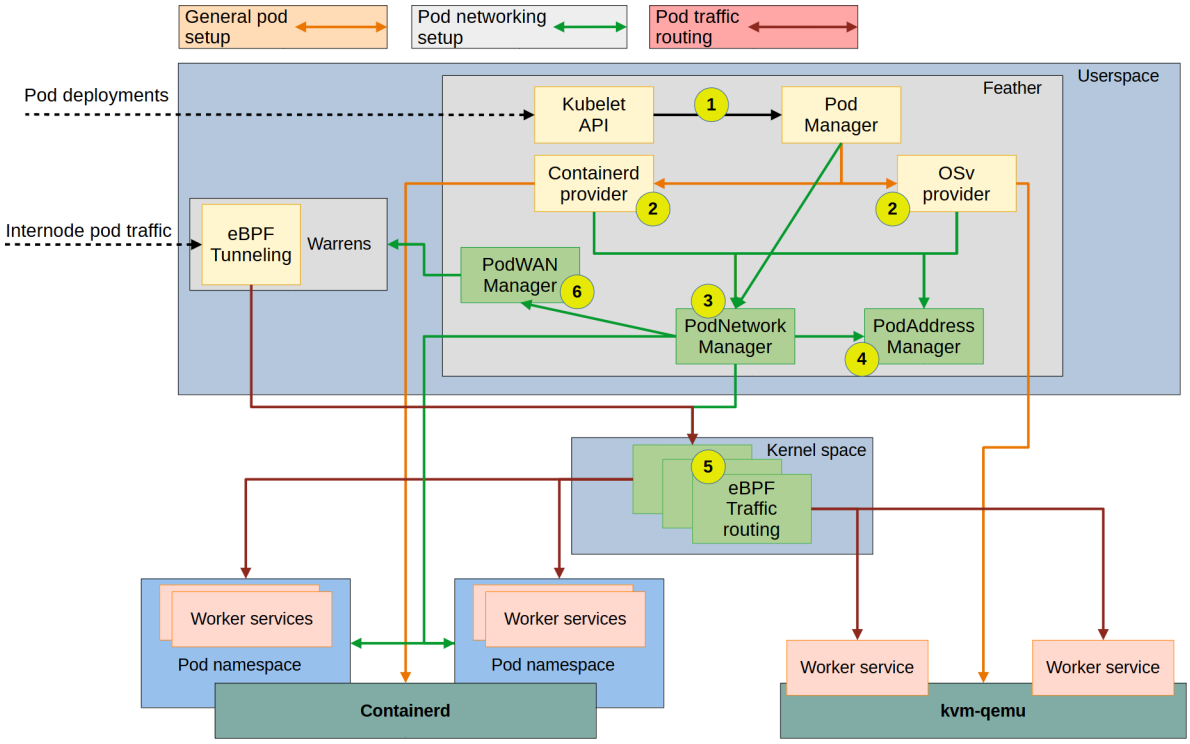


FIGURE 1. Architecture overview of mixed-runtime pod networking. The novel contributions of this article are highlighted by green components, integrating with existing Feather components through "Pod networking setup" interactions, which enable "Pod traffic routing" interactions.

restrictions, and workload image handling. The latter is achieved by packaging non-container formats (e.g. OSv unikernels) into an OCI image and adding metadata according to OCI standards; when Feather receives a pod deployment manifest from Kubernetes through the **Kubelet API**, the **Pod Manager** ① pulls the required workload images, examining the aforementioned metadata to determine the runtime providers that should execute them; these are indicated by the **Containerd provider** and **OSv provider** components ②. This approach avoids any changes to the Kubernetes cluster itself, or having to customize deployment manifests or the scheduler. Feather allows for innovative pod compositions, for example to improve the runtime security of certain workloads in a pod, or to improve performance of critical workloads. However, the providers only handle workload image management, resource restrictions (e.g. cgroups), and basic runtime parameters to call their runtimes (containerd and KVM-qemu, respectively). Importantly, Feather is aimed at low-resource edge devices, and by default provides its own container networking implementation using round-robin pod IP assignment rather than relying on complex APIs and CNI plugins, while blocking default CNI plugin deployment on nodes running Feather to avoid interference. As a result, it is a suitable host framework for a practical implementation of the solution presented in this article.

Mixed-runtime pod networking is enabled at the pod level by interaction of the **Pod Manager** with the **PodNetwork Manager** ③, which creates a dedicated bridge interface for each pod using the pod IP address, and a dedicated pod network namespace connected to the bridge through a veth (Virtual Ethernet Device) pair. Different implementations of this component exist, matching the **Network Modes** described in Section III-A. The pod network namespace supports any workload process that can directly access the container network interface inside it (e.g. "cni0" for containers), and can be optionally disabled for performance reasons if no workloads within a pod require it. Any pod workload that requires its own interface, e.g. a TAP [34] device for a qemu VM, is attached to the bridge instead of moving it into the pod network namespace. To support cross-runtime networking, providers must call either a **PodNetwork Manager** function which assigns a workload to the pod network namespace, or a function which creates a sub-pod ready TAP device for a workload to use; in Feather the **Containerd provider** adds containers to the pod network namespace, while the **OSv provider** requests TAP devices. Feather provides a straightforward interface to integrate additional runtimes as providers, making this approach highly extensible.

The **PodNetwork Manager** and providers interact with the **PodAddress Manager** ④, which generates IP addresses as described in Section III-B. As the **PodNetwork Manager** is workload agnostic, any network configuration required by the runtimes is instead passed to them by the providers. In addition to pod- and workload level network initialization, the **PodNetwork manager** is responsible for **sub-pod networking**, the core functionality of our solution which enables mixed-runtime networking by treating each pod

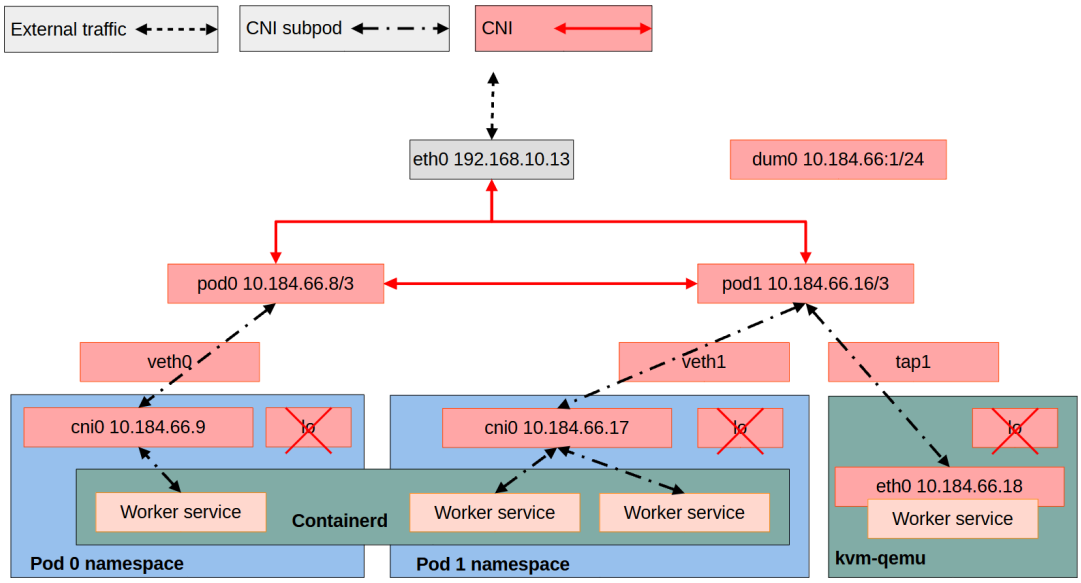


FIGURE 2. Overview of the main networking components involved in a concrete mixed-runtime pod networking scenario with two pods. While containers in the same pod still share the same network interface (e.g. a "cni0" container network adapter in their respective network namespaces), pod traffic can also be rerouted to dedicated interfaces attached to microVMs (e.g. tap1). The dum0 interface is a dummy interface used to enable pod "localhost" traffic.

as a miniature subnet under its assigned pod IP address. To that end, our solution uses custom eBPF traffic routing programs ⑤ to ensure optimal performance, and to disguise sub-pod networking from the point of view of other pods, as well as from other Kubernetes nodes. The lifecycles and routing tables of these programs are handled by the **PodNetwork manager** based on provider and **PodAddress manager** interaction. Finally, the **PodNetwork manager** also interacts with the **PodWAN Manager** ⑥ to determine how external pod traffic should be routed by eBPF programs to the **eBPF Tunneling** component. In the proposed architecture, Warrens [33] is used to set up tunnels between nodes, which uses its own eBPF functionality. However, configuration options also allow the **PodWAN Manager** to directly route traffic to a suitable network interface for compatibility with Kubernetes clusters expecting default CNI behavior.

A concrete example of the networking components created by this architecture is shown in Fig. 2, in which two pods are deployed on a Feather-managed node; **pod0** consists of a single container, while **pod1** consists of two containers and an OSv unikernel. Three types of network traffic must be managed in this example; "localhost", interpod and internode traffic. Sections III-C through III-E describe in detail how the proposed architecture enables each of these types of traffic.

A. OPTIONS

The solution offers a number of options for use in various situations, some of which are mutually incompatible.

Agent Mode: Although Feather is capable of operating within Kubernetes clusters, it can also run standalone. As such, pod networking must be capable of bootstrapping itself when no parameters are supplied by a control plane.

- Clustered mode connects to a Kubernetes cluster, receiving deployments and relevant node settings (e.g. Pod CIDR) from the control plane.
- Standalone mode runs the agent without connecting to a Kubernetes cluster. Pods (and workloads) may be loaded from disk or deployed through a REST API offering basic pod CRUD operations. In this mode, pod addresses are generated as illustrated by Fig. 3 and elaborated by Section III-B.

Network Mode: Mixed-runtime pod networking incurs a performance penalty, and may not be desirable in some use cases. Several network modes are devised to increasingly enable mixed-runtime networking features for specific types of edge computing scenarios.

- Legacy mode assigns incremental IP addresses to workloads, but does not enable traffic between non-container workloads, nor does it organize workloads behind a single pod IP address. While performant, this mode is only suitable for edge devices with workloads that contact cloud services and nothing else.
- Mixed Runtime mode organizes workloads behind a single pod IP address, allowing logical addressing from the point of view of other pods. While this mode is technically compatible with Kubernetes, it does not allow for sidecar functionality and is most suitable for running a collection of related, non-interacting workloads in the same pod.
- Mixed Runtime Localhost mode additionally allows for localhost traffic between workloads in the same pod, and

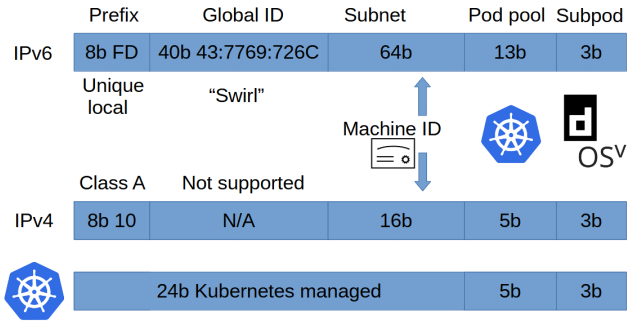


FIGURE 3. Structure of Kubernetes IPv4 addresses, and standalone IPv6 and IPv4 addresses when used for mixed-runtime IP addressing. In all cases, the last 3 bits are reserved for sub-pod addressing.

is fully compatible with container networking from a Kubernetes point of view, i.e. Clustered mode.

IP Version: As Fig. 3 shows, mixed-pod networking supports both IPv4 and IPv6 addressing; IPv4 is generally used for compatibility with Kubernetes CNI plugins, while IPv6 offers a significantly larger address pool and potential performance benefits.

- IPv4 addressing: the solution is run with IPv4 addresses, as presented in the rest of this article. This addressing mode is compatible with both Clustered and Standalone mode.
- IPv6 addressing allows for both more computational nodes in a cluster, and more pods per node. This is the default addressing mode used with Standalone mode. Additionally, IPv6 offers potentially higher performance as it does not have IP header checksums to be recalculated in eBPF programs as with IPv4.

B. SUB-POD ADDRESSING

Mixed-runtime pod networking attaches workloads in various runtimes to the same bridge, which is assigned the pod IP address. However, each attached interface needs its own sub-pod IP address to properly route pod traffic to the correct workloads. Therefore, a suitable sub-pod addressing scheme is required. In Kubernetes clusters, each node is generally assigned a pod CIDR by the CNI plugin. Most plugins use IPv4 by default, allocating a /24 subnet to each node. To support sub-pod addressing, the assigned pod CIDR on any node must be divided into relatively small subnets. Fig. 3 shows the address structures used for both IPv4 and IPv6, based on previous work in Warrens [33]. The structure is similar for both IPv6 and IPv4; the prefix indicates either a unique local (IPv6) or Class A (IPv4) address. Furthermore, the IPv6 addresses reserve 40 bits for the global ID "Swirl", which is omitted from IPv4 due to size constraints. The next field contains a hash of the machine ID, truncated to 16 bits for IPv4, or just 65536 physical devices. The remaining bits are divided among pod addresses and subpod addresses; pods are limited to 8 workloads (3 bits), while the address structure allows for 32 (IPv4) or 8192 (IPv4) pods per device. Note that Feather is aimed at specifically edge devices, and supports both in-cluster (Kubernetes managed pod CIDRs) and standalone operation.

When operating in Kubernetes clusters, only IPv4 is supported for basic CNI plugin compatibility. The resulting limited space is balanced by reserving 5 bits for a maximum of 32 pod addresses, and 3 bits for sub-pod addresses, for a maximum of 7 active interfaces within a pod. However, because any workloads that can share a network namespace only require a single address, this does not necessarily limit the number of active workloads to 7. For standalone operation with IPv4, the same ratio for pods to sub-pod addresses is used, although this is mainly for evaluation purposes; standalone mode uses an IPv6 scheme by default, which allows for larger node pools and up to 8.192 pods per node, along with far lower chances for address space collision as detailed by [33]. The construction of this addressing scheme answers **RQ1** by providing room for subpod IP addresses within container addressing.

C. LOCALHOST TRAFFIC

In Kubernetes, containerized processes in the same pod run in the same network namespace, using the same network interfaces. As such, the (web)services they provide are mutually reachable at **localhost** (or generally 127.0.0.1). However, due to the use of different network interfaces attached to a bridge in mixed-runtime networking, this "feature" is no longer available by default. Fig. 4 shows how the proposed framework enables localhost traffic even when some workloads may not be able to share the same network interfaces.

In this example, **pod1** consists of two containers running in the same network namespace, and a microVM workload running under kvm-qemu, attached through a TAP device.

Importantly, to process localhost traffic outside its own environment it must not only be interceptable, but "incoming" and "outgoing" traffic must be distinguishable. The default **lo** loopback interface produces frames with all zero MAC addresses, containing packets with identical source and destination IP addresses, thereby violating these requirements. Therefore, any default **lo** loopback devices must be circumvented, and a single concession to networking standards and practices must be made: localhost must be defined in the hosts file of any workload and network namespace to be a dummy address, in this case 10.244.66.1 assigned to **dum0**. Workloads should not generally be affected by this change; using "localhost" rather than hardcoded IP addresses is not only IP value, but also IP version agnostic.

When a container in **pod1** communicates with localhost, traffic is ① sent out through the network namespace interface to the pod bridge. It is intercepted by eBPF TC programs ② attached to all veth/TAP interfaces, designed specifically to handle sub-pod localhost traffic. When the eBPF program detects the target IP address is the dummy IP address, it duplicates received packets for each interface (i.e. sub-pod IP address) attached to the pod bridge **pod1**, including

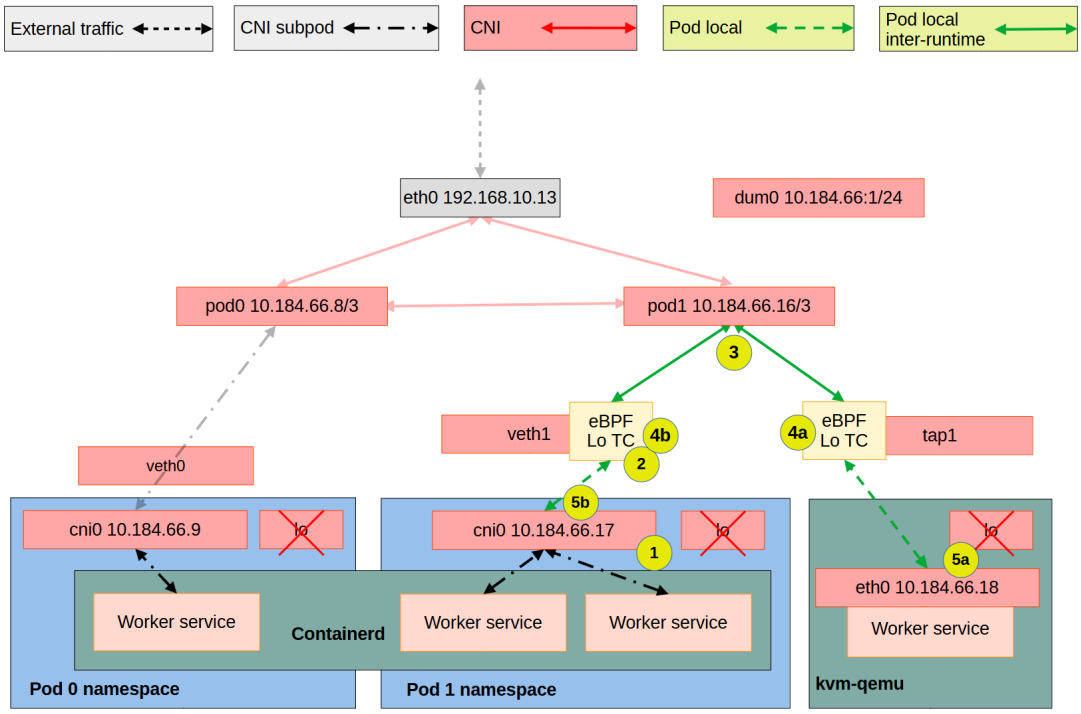


FIGURE 4. Network flow for localhost traffic going from a container in the pod1 network namespace to all workloads in the pod, including a microVM running under kvm-qemu. Importantly, the dummy0 interface must exist to subvert the default localhost IP address in workloads and process localhost traffic on the host network (namespace).

the origin, as the target may very well be another container. This step replaces the source address with the pod IP address in addition to changing the target address to workload IP addresses. The original packet is dropped, and all duplicate packets are redirected by the program from the bridge egress ③ to their respective interfaces ④a④b, bypassing some standard network stack processing. The eBPF programs at each veth/TAP interface again examine the packets, letting them pass unmodified. The packets are received by workloads, and responses are generated ⑤a⑤b. Responses are handled slightly differently, as the eBPF program now determines the destination to be the pod IP address, and re-duplicates responses, this time using the dummy IP address as the source. This two-address approach allows the programs to distinguish "incoming" and "outgoing" traffic, avoiding duplication of request/response packets ad infinitum. Note: any traffic that does not have the pod or dummy IP address as destination is immediately passed for performance reasons. This method has an overhead of $O(n)$ in terms of network traffic and processing, where n is the number of non-container workloads, and $O(n^2)$ for some ICMP messages such as ping. However, the full impact relative to actual workload processing is limited and is evaluated in Section IV. Finally, some types of packets, such as TCP responses with the RST flag set, are blocked by the eBPF program as these would effectively disable any TCP traffic both inside and between mixed-runtime pods by prematurely ending TCP connections meant for other workloads.

D. INTERPOD TRAFFIC

In standard pod networking, interpod traffic is trivial as only a single network interface is involved at both ends. However, in mixed-runtime pod networking routing interpod traffic is more complex; not only are there potentially multiple network interfaces at both ends (i.e. container + microVMs), but the bridge devices at both ends must also impersonate the pod IP address. Therefore, the proposed architecture handles interpod traffic with eBPF programs as it does localhost traffic. While the flow for network traffic between (mixed-runtime) pods is similar to that of mixed-runtime localhost traffic, there are important differences that merit attention. Fig. 5 shows the complete flow from request to response between two pods **pod0** and **pod1**. The container in **pod0** sends a request towards a service in **pod1** ①, the packets of which are picked up by an eBPF TC Pod2External (P2E) program attached to the bridge interface ②. This program replaces the source IP address with the pod IP address, and determines whether packets are meant for the local node or should be routed through the public interface **eth0** to another node. In this case, the request packets are routed to the **pod1** bridge, where they are picked up by an eBPF External2Pod (E2P) program ③, which duplicates them for each interface attached to the bridge, dropping the original packet. These duplicates are received and processed by their respective interfaces/workloads ④a④b, and the workload hosting the targeted service produces a response which is emitted back towards the pod1 P2E ⑤. This program modifies the source

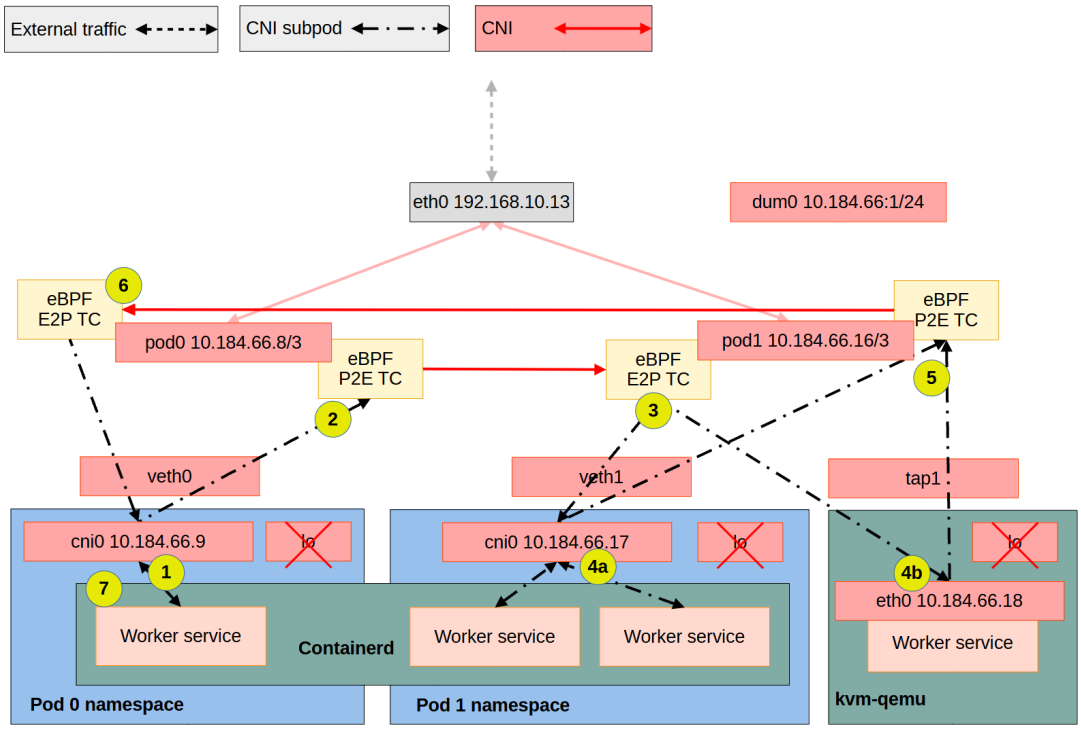


FIGURE 5. Network flow for traffic between workloads in different pods. Packets coming from the pod0 namespace are handled by eBPF programs to look as if sent by the pod 0 IP address, and to target individual sub-pod IP addresses of pod1 rather than the pod 1 interface itself.

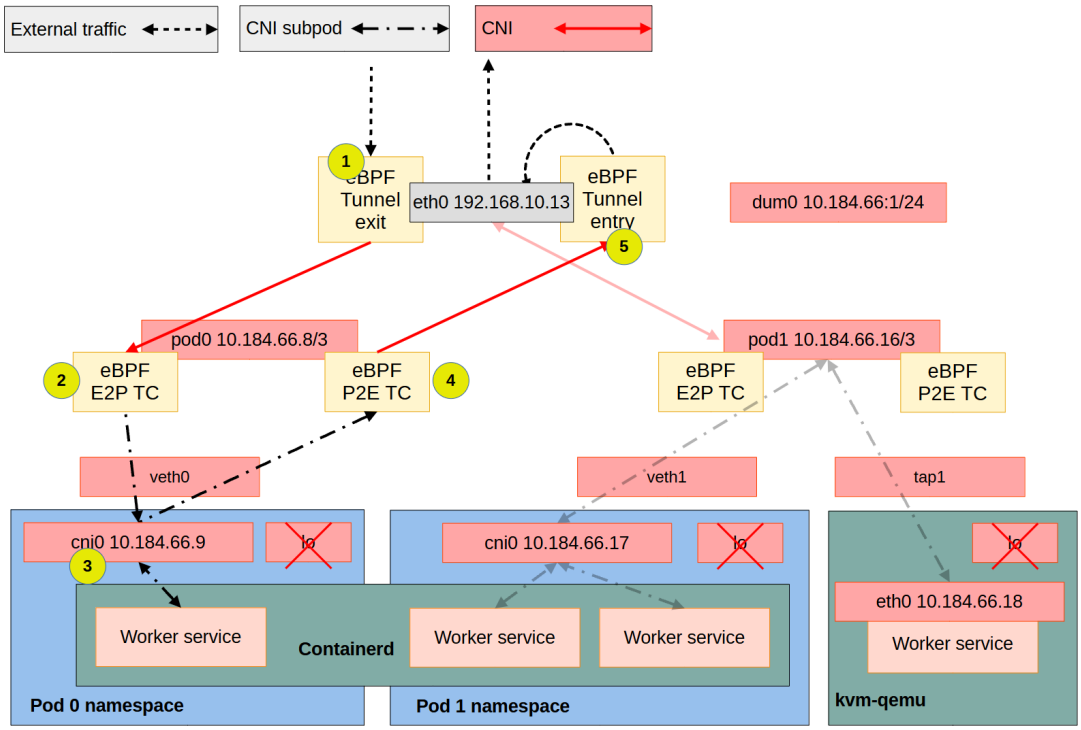


FIGURE 6. Network flow for traffic to and from workloads on other nodes. The same eBPF programs for interpod traffic are used, but for remote addresses they default to routing traffic through the public interface ("eth0"), which may be combined with Warrens (eBPF Tunnel programs) for edge traffic tunneling.

IP address to the pod IP address, and redirects the response packet(s) back to the **pod0** E2P ⑥, which duplicates the packet towards its component workloads; in this case, the single container that sent the original request ⑦.

Technically, the E2P and P2E programs are part of a single eBPF program using tail calls; different eBPF program configurations were attempted, including XDP implementations, but these did not function as required. Tail calls introduce some extra overhead from checking which subprogram to call, but also allow immediate exits for packets that should not be processed rather than calling several eBPF programs with their own packet checks.

E. INTERNODE TRAFFIC

Internode traffic is mainly handled by the E2P and P2E programs as shown in Fig. 6, but instead of rerouting to a local pod bridge, traffic is routed through the device's external interface. In the presented situation, mixed-runtime networking is combined with Warrens [33] for internode communication. When another node or service sends a request to **pod0**, packets enter through the **eth0** physical interface, where they are intercepted by an eBPF tunnel decapsulation program ①. The program determines the correct pod bridge to redirect to, in this case **pod0**, and emits the packets from the **eth0** egress. At the pod bridge, E2P ② and P2E ④ perform the same job as they do in Fig. 5, although in this case P2E sends intercepted packets back to **eth0**, where they are received by the eBPF tunnel encapsulation program ⑤ and sent to the relevant remote node. Note that without Warrens, the flow is essentially the same, with standard IP routes routing packets to other nodes in ① and ②. The latter method is applicable when all nodes are mutually reachable without any sort of cross-network tunneling.

This workflow, along with those for interpod traffic and localhost traffic, provides a solution for **RQ2** by providing a complete solution for inter- and intrapod traffic for workloads in different runtimes, while publicly exposing only pod IP addresses.

F. SECURITY CONSIDERATIONS

The proposed solution enables localhost traffic between workloads in the same pod, independent of their runtimes. As a result, security improvements are almost exclusively the result of runtime choice, e.g. OSv unikernels will offer better process isolation and a far smaller kernel attack surface by design compared to containers. However, there are potential security improvements compared to container-only pods; the solution will only forward outgoing traffic to other workloads in a pod if the traffic is meant for localhost, except for containers which are all connected to the same pod network interface. In container-only pods (i.e. default Kubernetes), containers are all attached to the same interface regardless, and may intercept non-localhost traffic at will. As a result, multi-runtime networking allows, for example, one workload to run as an OSv unikernel attached through a TAP device, making it impossible to sniff non-localhost traffic even inside the same pod. Unlike container-only pods, localhost traffic may leave the pod network namespace, posing a potential risk. However, eBPF programs intercept at the lowest possible level, and the flow presented in Fig. 4 both intercepts and emits network packets before they can be intercepted by packet sniffing programs in the host (root) network namespace.

IV. EVALUATION

In order to gauge its impact on system resources, network throughput and network latency, the proposed architecture is evaluated for multiple networking scenarios. For the purposes of the evaluation, only the Containerd and OSv runtimes are used; these are already implemented in Feather. From Section III the conclusion can be drawn that although additional runtimes are highly desirable, they can be attached through existing methods (e.g. Kata Containers using KVM/Qemu and TAP, WASM by running it in the container network namespace), such that a comparison would not reveal anything new about the proposed solution, but only compare individual runtime behaviors. This section describes the evaluation setup, evaluation scenarios, methodology and any tools used. The code used for the eBPF programs and Feather, and the evaluation scenarios, is made available on GitHub.[1]

A. EVALUATION SETUP

Evaluations are performed on a Ubuntu 24.04 desktop machine with an Intel Core Ultra 7 155H processor, 32GiB dual channel LPDDR5, and a 1TB M.2 NVMe drive. However, the proposed solution is technically compatible with any device with a Linux kernel version over 5.15, up to and including Raspberry Pi 3 models; the version of Feather in which the solution is implemented has been shown [14] to require only around 60MiB disk space and, when optimized, around 55MiB of free memory. The runtimes used for the evaluation are containerd v1.7.27 for Docker containers, and OSv v0.57.0 images on KVM-qemu v8.2.2. Despite ARM compatibility for Feather and eBPF, evaluations are not run on ARM due to OSv compatibility concerns, e.g. missing syscall implementations on some devices and issues with reliable image building. However, recent evaluation results of a similar eBPF-based solution for internode point-to-point tunnels for container traffic [33] provide some indication of ARM performance compared to x86.

B. EVALUATION METRICS

Four aspects are considered for the evaluations, three of which are separate scenarios and one is an in-depth analysis of the solution architecture versus evaluation results:

- Latency Overhead: this single-metric scenario measures the latency overhead introduced by eBPF processing, for both inter- and intrapod traffic. Latency is measured for individual ICMP echo request/response packets by running the ping utility for 1000 seconds.

[1] https://github.com/togoetha/feather-multiruntimenetwork

- Throughput: the CPU load and throughput of the architecture are evaluated using iperf3 to fully saturate network traffic between pods for a period of at least 60 seconds.
- Video Streaming: to simulate the impact of mixed-pod networking on the network throughput and latency of a straightforward but real-life application, this scenario uses a minimal video streaming service hosting chunks of a 40 second movie trailer. It also illustrates the low overhead of the solution for an application with high concurrency from users, yet a relatively low processing overhead per request, as well as evaluating how overhead scales with variable throughput throughout the scenario. Each chunk varies in size from ≈620KiB to ≈4.2MiB, containing a constant 4 seconds of video data at high resolution. Apache JMeter acts as a client, building up a load of 500 clients over a period of 20 seconds, each watching the video 3 times before stopping. As a result, the default runtime without processing delays is close to 140 seconds. This scenario is repeated 10 times to map the full range of performance and identify statistical outliers.
- Efficiency/scalability: detailed data from the Video Streaming scenario is offset against architectural and runtime details to determine their relative efficiency and scalability in terms of CPU load.

C. METHODOLOGY

Packet data is gathered using Wireshark, as are relative time differences as packets manipulated by eBPF programs and routing pass through various network interfaces, for example when tracking ICMP packets in the first scenario. While this method does not guarantee nanosecond precision, the most important steps are in the μs range or higher. CPU load for containers is measured by monitoring contained processes (e.g. iperf3) directly using the top utility, while for OSv unikernels qemu-system-x64 processes are monitored. eBPF CPU utilization is extracted from system statistics using **bpf_stats_enabled**, which yields detailed data for each program showing CPU use up to nanosecond precision. No other programs were run during the evaluations, except those required for monitoring and data processing, and the CPU governor was set to performance mode to avoid throttling effects. For aspects other than iperf3 evaluation, network throughput is measured by parsing interface statistics from **/proc/net/dev**. CPU load, network statistics, and eBPF statistics are gathered automatically once per second through a script that runs for the duration of each evaluation run.

For each of the scenarios, the relevant metrics are measured for traffic between pods using: only a network namespace (container-only pods), only a VM TAP (OSv-only), or a combination of both (container + OSv pods). Server programs (iperf3 server, video streaming service) may run in either a container or an OSv unikernel. However, for practical purposes, client programs are always run from a network namespace; no VM-to-VM evaluations are performed.

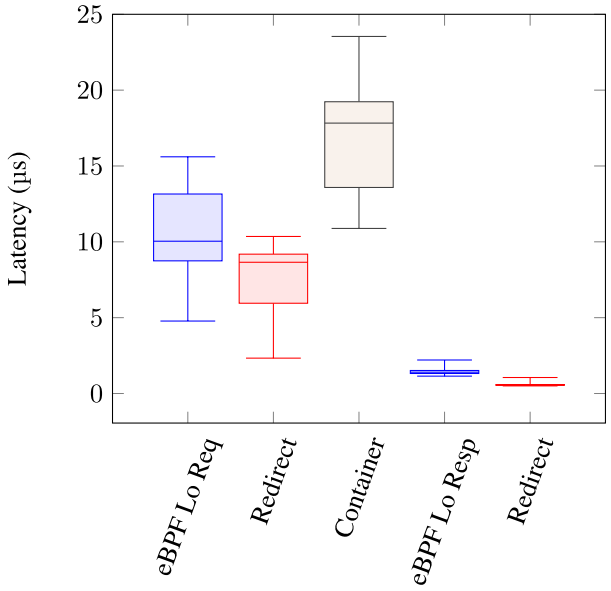


FIGURE 7. Components of basic container localhost ping latency. Colors are used to identify logically similar steps in the process.

Some issues were encountered when running iperf3 in an OSv image;[2] despite attempting different approaches to build a working image from scratch, the **/dev/urandom** device was not present in the images. Changing the iperf3 source code to use **/dev/random** instead created a working image. While the effect on throughput and efficiency is unknown, comparison of the iperf3 and video streaming scenarios in Section V indicates no measurable impact.

V. RESULTS

This section presents the results of the evaluations per aspect, examining the relevant metrics as mentioned in Section IV-C. For the video streaming scenario, time series charts show representative full runs rather than averaging all 10 runs, specifically to illustrate possible instabilities and time-dependent phenomena. Additionally, time series charts indicate the range for statistical outliers from -2 to $+2$ standard deviations calculated over all 10 runs where it does not interfere with readability.

A. LATENCY

Fig. 7 shows the results for localhost ping latency in a pod with a single container; ICMP requests and responses are thus redirected to the host namespace for eBPF processing, but not duplicated. This minimal operation consists of five steps, logically grouped by colors in the figure: blue for eBPF processing, red for Linux cross-namespace processing, and gray for latency caused by the pod network namespace and ICMP protocol itself. In median cases, the processing latency caused by eBPF programs is around 10% higher than cross-namespace latency, and 56% of pod namespace latency. However, every step other than "Container" is technically

[2]https://github.com/cloudius-systems/osv/issues/1333

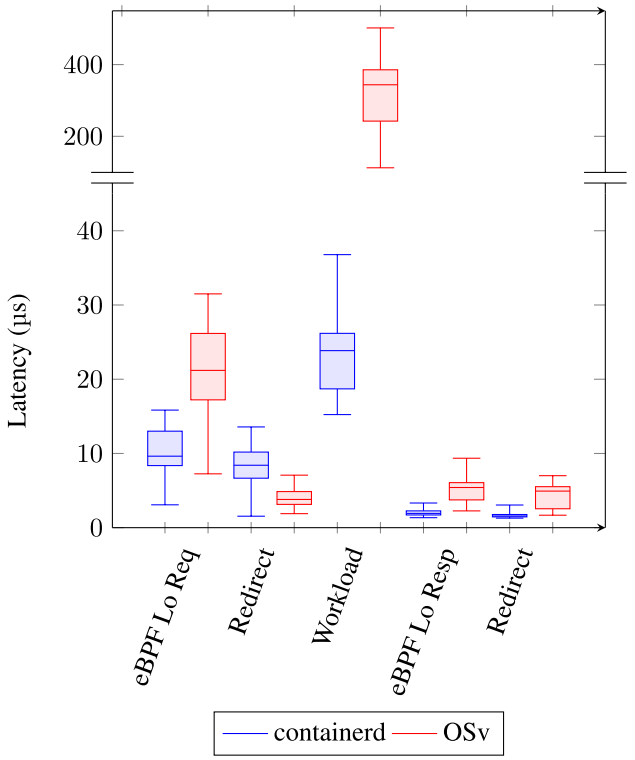


FIGURE 8. Components of localhost ping latency for a multi-runtime pod running both a container and an OSv unikernel. Note that a localhost ping results in responses from both components of the pod.

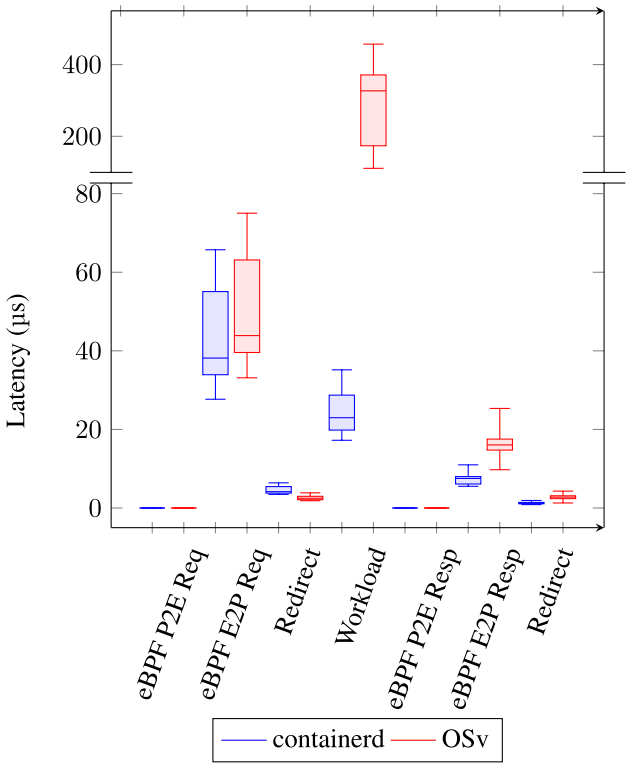


FIGURE 9. Components of cross-pod ping latency from a pod with one container to a multi-runtime pod running both a container and an OSv unikernel.

overhead caused by the mixed-pod networking solution, resulting in a 55% median overhead compared to standard container networking, and ranging from 40% to 58.8% in outlier cases. In absolute values, this amounts to 9μ to 27.6μs of added latency.

The results of localhost ping latency for a pod with both a container and a unikernel are shown in Fig. 8. eBPF processing of the ICMP request logically takes about twice as long for the unikernel as it does for the container; this is merely a consequence of duplicated packets sent out in a loop. However, redirecting ICMP requests to a workload through a TAP device (unikernel) is around 50% faster than through a veth pair (container namespace). Curiously, processing inside the namespace itself takes 30% longer than in the previous case, and median in-unikernel latency is 14 times higher than for a container. The ICMP response path (eBPF Lo Resp + Redirect) is independent of pod size, although unikernel response processing and redirecting is around 3 times slower. As a result, the first step is shown to scale with pod size, and while the redirect and response step latencies vary between containers and unikernels, the total net overhead of mixed-pod networking is similar, at 22.2μs (8.7μ to 33.2μs) for containers and 24.2μs (12.7μ to 36.1μs) for unikernels, the latter excluding the overhead of the first "eBPF Lo Req" loop iteration.

Fig. 9 shows the results for ping latency from a pod with a single container to a mixed pod with a container and a unikernel. This process involves some extra steps

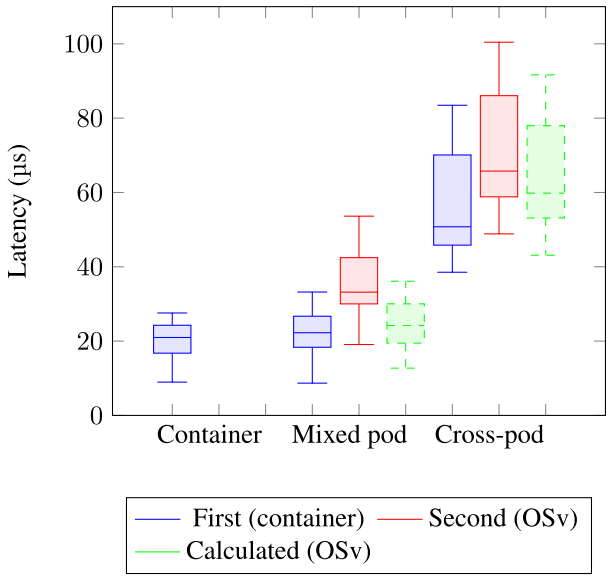


FIGURE 10. Latency of ping requests in different situations and for varying runtimes in a single pod. The first workload in a pod is always run as a container, while the second is always an OSv unikernel. "Calculated" indicates an estimate for a unikernel running as first workload.

as shown in Fig. 5. Notably, the eBPF P2E programs are reported to run in zero time, or in the context of the evaluation, faster than the time measurement error. While this is likely to be a Wireshark measurement error, the actual overhead is inconsequential compared to the more complex

E2P program. Scalability of E2P is more complex than the localhost program, with the results indicating that the packet duplication loop has a relatively small impact; the second loop iteration (OSv unikernel) takes only 6μs (or 16% of the runtime). The rest of the performance measurements are virtually identical to Fig. 8, with the exception of the response path of the E2P program, which is 3 to 4 times slower than the eBPF Lo program due to additional packet validation and tail calls. Considering total overhead, this situation has a 50.8μs (38.5μs to 83.5μs) overhead for containers and a net overhead of 59.8μs (43.1μs to 91.6μs) for OSv unikernels, the latter estimated from eBPF P2E scalability.

Fig. 10 summarizes the total latencies discussed for all situations, showing the overhead of eBPF processing plus redirecting, both including and excluding the first loop iteration overhead in the case of OSv.

B. THROUGHPUT

Fig. 11 shows the TCP throughput achieved by an uncapped iperf3 instance, along with various factors contributing to CPU load. Although iperf3 is single-threaded and essentially limited to a single CPU core, the OSv instance uses two CPU cores total, indicating a significant overhead outside of iperf3 itself. Despite this, median throughput is around 65.9Gbps for the container, but significantly lower for the OSv instance at 42.2Gbps. Curiously, iperf3 client CPU use is slightly lower relative to throughput in the OSv case, despite running in a different pod (and container) and thus not in direct contact with any OSv networking. This is likely due to the container version of the evaluation being client-side CPU limited, while the unikernel version is server-side CPU limited. The statistical outlier ranges for both versions confirm this; despite a reasonably variable throughput in all cases, the CPU limited end dictates throughput while the CPU load of the other end scales with throughput. eBPF performance is slightly better in the OSv case, using only 9.2% of a single CPU core vs 17.8% for the container, or in relative terms 4.5Gbps/core% for OSv vs 4.06Gbps/core% for the container. Both numbers indicate a low overhead for mixed-pod networking, technically saturating a 10Gbps server connection at 2-2.5% of a CPU core in high traffic scenarios. In both versions of the scenario, the full eBPF performance range is fairly stable, varying by around 5% for containers and 10% for unikernels.

C. VIDEO STREAMING

The throughput for runtime of the video streaming evaluation is shown in Fig. 12. For 500 clients and the size of video chunks used, throughput is expected to fall between 2Gbps and 2.5Gbps throughout the test; this is the case for all (mixed) runtime alternatives, although throughput destabilizes for OSv unikernels after around 80 seconds. In the case where the unikernel is the only workload in the pod, throughput wobbles excessively but keeps up with client demand on average; in the case of a mixed pod, throughput drops sharply and continues at 10% of the expected amount after the planned evaluation runtime, until all clients have finished.

A partial explanation is provided by Fig. 13, which shows CPU use for the video streaming service and eBPF programs. While CPU use for the service is around 5-7% of a single core when running in a container, the OSv version uses nearly 8 to 10 times more. Furthermore, around 70 seconds unikernel CPU use climbs to a little over 100%, saturating the resource limit for the VM. In the case of a pod with only a unikernel, CPU use eventually recovers, but it remains unstable and the run concludes with a few seconds of delay. In a mixed-runtime pod, CPU use stays at or above the VM limit and client latencies rise steadily, resulting in a growing backlog of requests while the number of requests handled per second drops. This run eventually concludes several minutes after the designed end time. It is not known at this time why the unikernel version exhibits this behavior, as it is contrary to previously observed performance [14], [20], which indicated request handling and memory consumption advantages for unikernels in specific scenarios (i.e. REST service, Minecraft server). Unikernel-hosted services seem to experience a momentary backlog of requests at some point during this scenario, which snowballs into CPU saturation. However, further investigation falls outside the scope of this article, as this is neither a networking issue, nor related to the implemented architecture. Note that increasing the number of CPUs assigned to the unikernel does not mitigate the issue, as OSv has been observed to scale negatively with an increasing number of cores [20], which was confirmed during the evaluation runs.

Apart from OSv behavior itself, mixed-runtime networking overhead is minimal; total eBPF CPU use hovers around 1% of a single core throughout most runs, with the exception of a video streaming container in a mixed-runtime pod, which uses around 1.5%. The latter is in line with earlier observations, since traffic is duplicated to both the streaming container and an idle unikernel. As such, there seems to be roughly a 55% overhead for each additional runtime interface added to a pod bridge in this scenario. Although no meaningful conclusions can be drawn from the OSv results after 70 seconds, eBPF performance is slightly better when a streaming unikernel is the only workload in the pod, rather than a container, which is also in line with Fig. 11. Curiously, eBPF CPU use only rises around 20% when an idle sidecar container is added to the pod containing the streaming unikernel, making it only 10% to 15% more resource intensive than running the streaming service in a container by itself.

D. EFFICIENCY AND SCALABILITY

Figs. 14.a through 14.d provide an overview of the factors contributing to eBPF CPU use in the video streaming scenario, when using either a container or OSv unikernel to host the streaming service. Despite the irregularities of OSv performance, comparing individual eBPF program performance with implementation details yields interesting

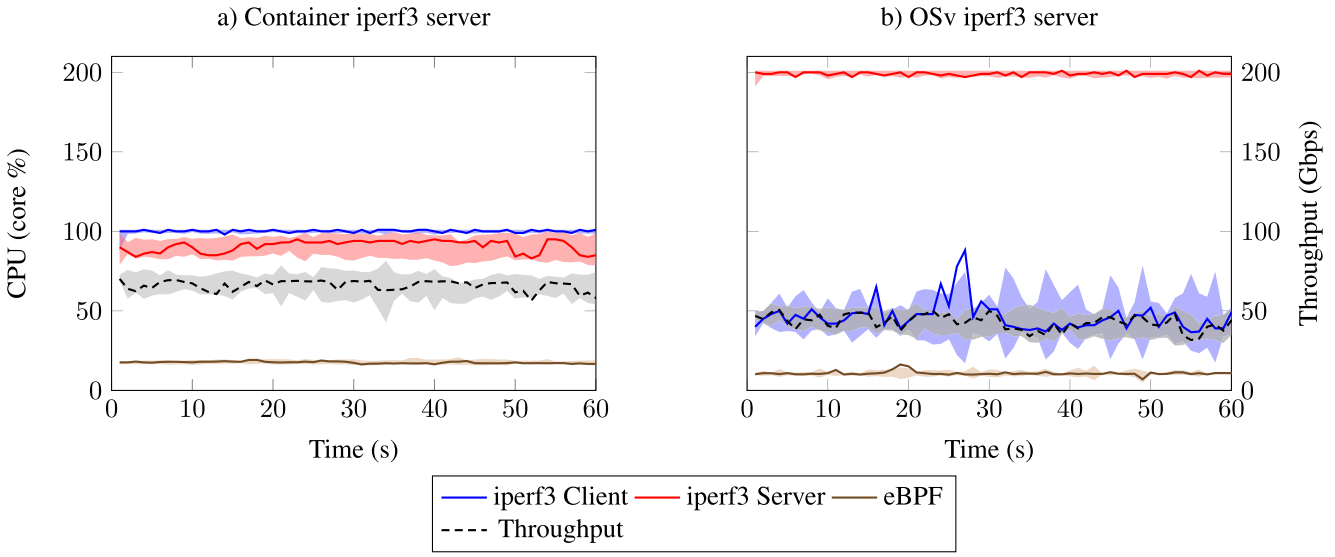


FIGURE 11. Mixed-runtime, cross-pod networking iperf3 performance from a client in a container to a server in container and OSv workloads. While mixed-runtime eBPF performance is similar in both cases, the containerized iperf3 server is significantly more CPU efficient than its OSv counterpart.

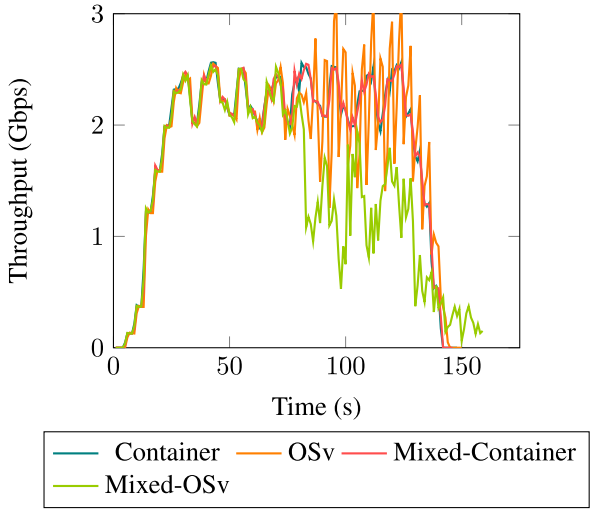


FIGURE 12. Video streaming throughput for various runtimes and pod compositions. 'Mixed-' indicates the server is running in a pod with a sidecar using a different runtime. While initially on par with containers, OSv-based video streaming services exhibit erratic behavior after 70 seconds.

results. Fig. 12 shows throughput to be independent of the choice of streaming service runtime until around 70 seconds; subfigure 14.a shows the same behavior for the E2P/P2E program of the video streaming client (i.e. JMeter), until the OSv program starts to break down due to hypothesized request queuing problems. Given this period of stable performance, subfigure 14.b indicates a 20% lower CPU load for eBPF programs using a TAP interface to communicate with a microVM (i.e. unikernel), compared to communicating with a network namespace over a veth pair. This behavior is affirmed by subfigures 14.b and 14.d: the localhost program on the client side is agnostic w.r.t. choice of server runtime, while the program handling the server pod is 20% more efficient when using an OSv unikernel. However, considering the CPU overhead shown in Fig. 13, the container version of the video streaming server is preferable. Overall, for interpod traffic, the localhost program causes a relatively limited 7% overhead as it is not technically required; it inspects packets and immediately lets them pass as they are not targeted at the localhost address. The ranges for statistical outliers indicate stable eBPF performance across runs, with minimum and maximum CPU use at any point being within 10% of each other. Furthermore, some of this variance can be attributed to small timing errors across runs, sometimes leading or trailing other runs by a fraction of a second. Finally, the outlier ranges also show that once an OSv unikernel destabilizes, its performance becomes entirely unpredictable and eBPF performance (and thus throughput) varies by a factor of 50% to 100% between observed runs.

To gauge scalability, Fig. 15 compares eBPF program CPU use between a container-only client pod and mixed-workload server pods. Results for the localhost program are not shown, as it is unaffected by mixed-workload scaling in the context of cross-pod traffic and causes the same overhead as in Figs. 14.c and 14.d. Performance scaling of the E2P/P2E program is straightforward when the main workload is hosted in a container: Fig. 15.a shows the client pod eBPF program is unaffected by any changes in the composition of the streaming service pod, while the server pod eBPF program scales slightly better than linear when a sidecar workload is added through a (TAP) interface. Over the runtime of the scenario, the container-hosted mixed-workload streaming service pod uses 77% more CPU on average than a container-only pod. Additionally, eBPF CPU use for the client pod shows a significant increase. When the main workload is hosted in an OSv unikernel, eBPF CPU use also scales, but

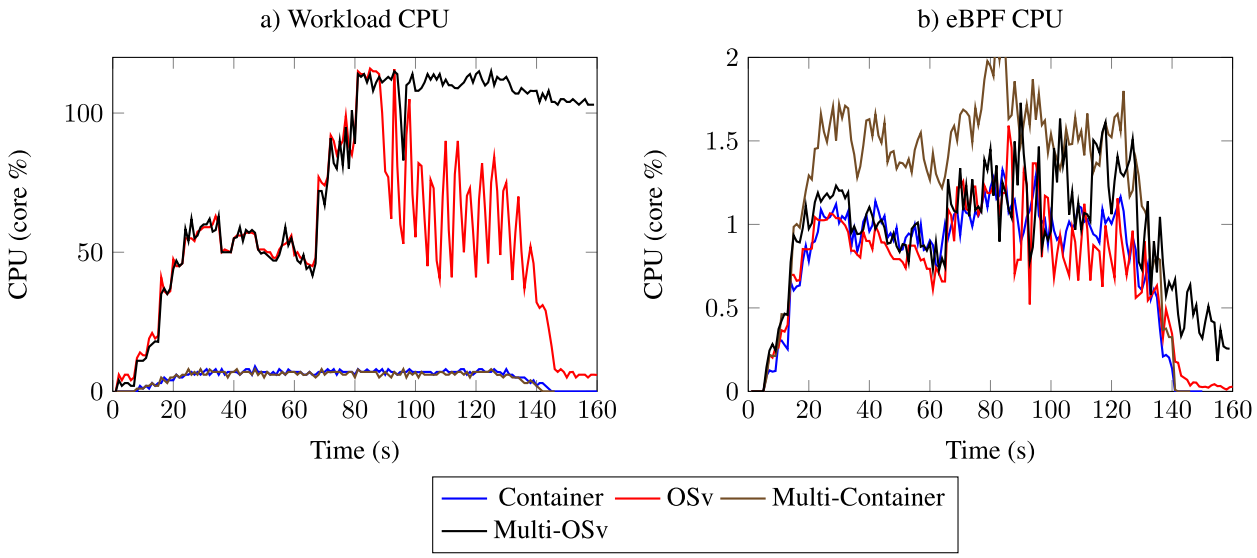


FIGURE 13. Various factors contributing to CPU use in the video streaming scenario, for different variations of pod composition. OSv-based streaming services use an order of magnitude more CPU than containerized ones. CPU use rising to over 100% causes a backlog of requests, exacerbating erratic service behavior and resulting in a longer runtime.

with a lower factor; the server E2P/P2E program CPU use increases by only 40% when adding sidecar containers to an OSv-based pod, compared to the 77% of adding OSv sidecars to a container-based pod. Additionally, this option incurs no scaling penalty on the client-side E2P/P2E program. To summarize, mixed-pod networking CPU use scales with $O(n+m)$, where n is the number of interfaces attached to the client pod bridge, and m is the number of interfaces attached to the server pod bridge. However, the factor added for each additional workload depends on the relative amounts of traffic handled by either containers or OSv unikernels, with the latter enabling more efficient eBPF use.

VI. DISCUSSION AND FUTURE WORK

The results show that the eBPF approach towards mixed-runtime pod networking adds an absolute latency ranging from only 22μs to around 65μs for the evaluated pod compositions, and is highly predictable based on the composition of both the sending and receiving pods. Although this is a significant relative overhead in terms of ping requests on a local machine, it is neglegible for more complex (non-localhost) traffic. As a rough estimate, latency for each additional interface attached to the bridge of the receiving pod increases by 11μs for local traffic, ranging from 22μs for just a container namespace to 88μs for a container namespace and 6 additional interfaces. Similarly, for cross-pod traffic latency increases by 15μs per interface, ranging from 51μs to 141μs. However, considering the composition of the sending pod or the order in which responses are received (e.g. the 'sender' is the last interface in the sending pod), the total latency can amount to up to 150μs for localhost traffic, and 200μs for cross-pod traffic. Compared to the latency overheads of edge networking and microservices, which generally range from several milliseconds upwards, this overhead can be deemed acceptable considering the freedom of choice for workload runtimes it enables.

The iperf3 evaluation illustrates CPU overhead relative to actual workloads in high-throughput scenarios, showing that the eBPF programs that enable mixed-runtime networking use 7 to 30 times less processing power than the client and server programs that process the actual traffic in an application designed specifically to evaluate bulk throughput. The actual ratio varies greatly with the choice of runtime for iperf3, and the scenario indicates that despite lower performance for OSv unikernels, eBPF efficiency is higher when using TAP devices rather than veth pairs. Finally, the scenario also suggests that the proposed implementation can saturate a 10Gbps connection with just 1.5-2.5% of a single CPU core.

The video streaming evaluation affirms the relatively low CPU use, although in this case it is around 1% for 2Gbps to 2.5Gbps sustained throughput for 500 clients. The effect of mixed-pod scaling is illustrated, showing that as with latency, server pod eBPF CPU use scales with the number of interfaces attached to a pod bridge. However, there are significant differences in the scaling factor depending on whether unikernel or container workloads contain the main workload; adding container sidecars to a unikernel-based pod causes a 20% overhead, while adding a unikernel to a container-based pod increases eBPF overhead by 55%. On the other hand, containers can be hosted in the same network interface, whereas each unikernel requires its own additional interface at a 20% CPU increase.

Finally, the contribution of each eBPF program towards total eBPF CPU use is analyzed for cross-pod traffic.

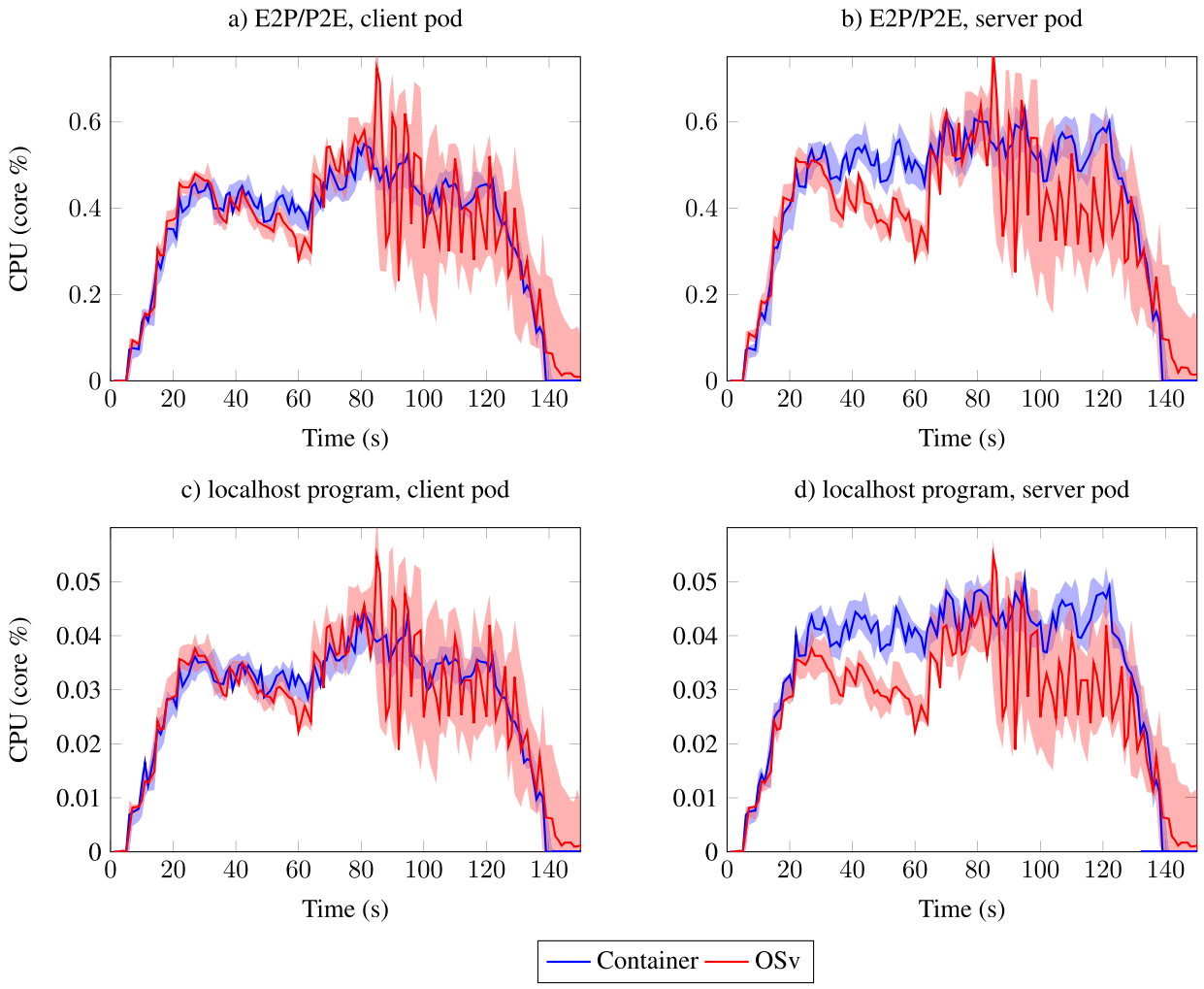


FIGURE 14. CPU use analysis of eBPF programs involved in mixed-workload processing for container-only vs OSv unikernel-only server pod in the video streaming scenario.

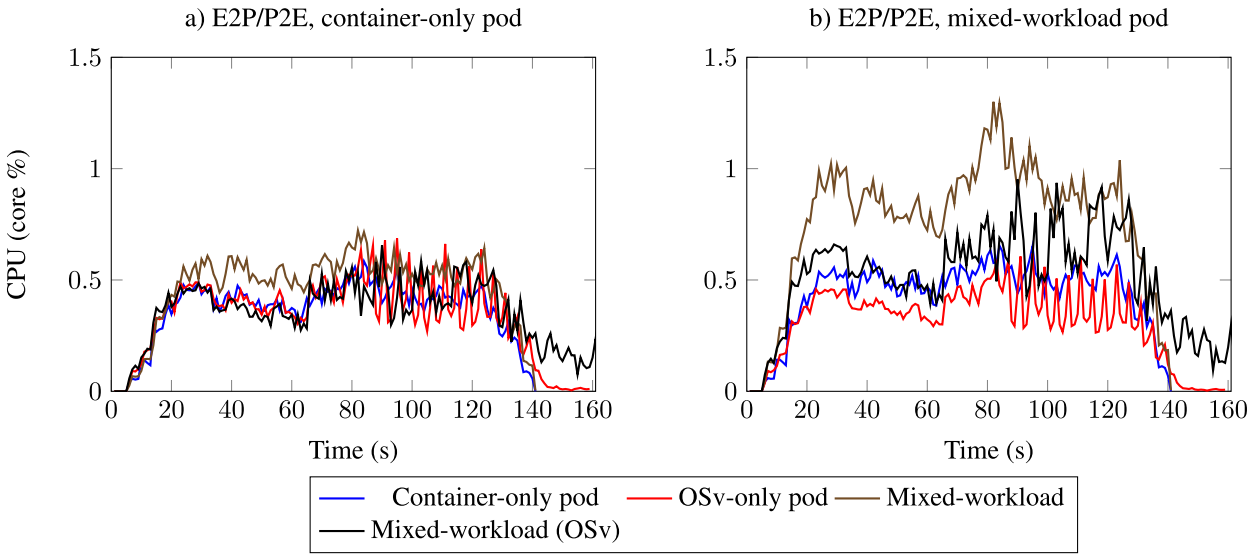


FIGURE 15. eBPF E2P/P2E program CPU use for traffic from a container-only client pod to various compositions of video streaming server pods.

The first case evaluates video streaming traffic between a container-only client pod and a container- or unikernel-only server pod, showing that eBPF programs interacting mostly with TAP interfaces (i.e. unikernels) execute significantly more efficiently than those that mostly interact with veth pairs (i.e. containers). The second case considers pods with mixed runtimes, and affirms that OSv unikernels i.c.w. TAP interfaces enable more efficient traffic processing in eBPF programs; baseline server pod E2P/P2E performance is better than that of a container-only pod, and adding container sidecars causes an 40% CPU use increase as compared to 77% when hosting the main workload in a container, the latter also incurring a performance penalty on the client pod eBPF program. However, containers can be hosted in the same network namespace, whereas each unikernel requires its own additional interface at a 40% CPU increase. As a result, unikernel workloads are optimal in use cases in which unikernels perform better than native (containerized) processes, and all sidecars are run as containers handling minimal amounts of network traffic.

While Section III answers **RQ1** and **RQ2**, the evaluation results answer **RQ3** by showing that the performance overhead is sufficiently low to be used in realistic mixed-runtime pod scenarios, and by providing extrapolations for more complex pod compositions.

While this article presents a fully operational solution for mixed-runtime pod networking, some aspects could be added or improved in future work. For example, the approach of adding workloads through a TAP device or directly into a container namespace can be further finetuned to universally support additional runtimes, especially in cases where both would be possible e.g. running WASM/WASI workloads through a shim, or in a microVM. Concretely, Kata Containers could be supported through KVM/Qemu, which is already implemented for OSv unikernels, and WASM could be supported through WasmTime, which allows IP/port binding and can be attached to the existing container network namespace. Furthermore, eBPF performance for both these options is expected to be in line with OSv and containers, respectively, as no changes to the framework itself are needed. Similarly, Firecracker could be added as a separate backend much like KVM/Qemu, using TAP devices if supported.

Additionally, although the presented solution is specifically designed for orchestration agents that support mixed-runtime pods (i.e. Feather), it could benefit from optimization according to container standards, and development into a full Container Network Interface (CNI) plugin which would improve compatibility with other CNI plugins or networking solutions.

Furthermore, the localhost and P2E programs have limited filtering to ensure TCP connections are not interrupted by TCP RST packets from workloads that do not host a specific service. For container-only pods, this is not an issue as packets do not need duplication and are handled by a single endpoint network interface. It is conceivable that other types of filtering or packet fixing are required for different types of protocol-level traffic, which would impact eBPF performance to some degree.

Finally, the scalability of both inter- and intrapod communication can be improved by implementing an adaptive port scan within the eBPF programs, each examining its own pod. For example, the P2E program could pass IP address and TCP port pairs derived from RST packets to the E2P program of the same pod by BPF map; initially this will increase processing overhead, but with enough negative responses from workloads the programs will be able to build an inventory of ports hosted by workloads in the same pod, and the loop used to split packets to each workload in a pod can be skipped for known ports, eliminating the performance overhead shown in Fig. 15.b. Additionally, this approach would eliminate the ability of non-container workloads in a pod to sniff incoming traffic of other workloads. The same result could be achieved by passing container ports defined in Kubernetes pod deployments to the E2P program BPF map, although these are not mandatory and thus cannot be relied on.

VII. CONCLUSION

Orchestration software such as Kubernetes is increasingly used to deploy non-container workloads, although such extensions generally enable the use of pods which run entirely on a single runtime. This article states a number of research questions on the topic of mixed-runtime pod networking, and presents a pod networking architecture implemented in Feather to answer these questions. The implementation allows Feather, which supports mixed-runtime pods by default, to allow workloads in the same pod but executing in different runtimes to share the same apparent pod IP address, while communicating over localhost as if they were containers.

Specifically, this work shows the proposed architecture enables pod localhost, interpod, and internode traffic through the use of a novel sub-pod IP addressing mechanism, which reserves part of the pod IP address space of a node to use as workload IP addresses below the pod level. This article also presents an implementation in which all interfaces required for sub-pod traffic are handled by two types of sub-pod network attachment: either by placing workloads in a dedicated network namespace per pod, or by attaching them to a bridge device through a TAP interface. Traffic between these sub-pod interfaces, and between pods, is enabled by a limited number of eBPF TC programs. Furthermore, this approach allows upwards of 7 simultaneously active mixed-runtime workloads in a single pod, depending on pod composition.

The evaluation considers the latency, throughput, processing overhead and scalability of the proposed architecture in a number of scenarios, showing that although latency overhead ranges from 20μs to potentially 200μs, this is negligible in terms of total internode latency at the application level. Similarly, the evaluation shows throughput to be in the order of Gpbs per % of a single CPU core, orders of

magnitude below the processing power required by even basic applications such as iperf3 or a video streaming service. Finally, while the number of workloads in a pod affects both latency and CPU use, careful selection of runtimes for the main workload may enable better performance than container-only pods.

In addition, some topics for future work are discussed, such as options to easily integrate additional runtimes and eliminate the most important factor contributing to scaling overhead of the proposed solution.

REFERENCES

[1] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE Access*, vol. 8, pp. 85714–85728, 2020.

[2] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu, "Data security and privacy-preserving in edge computing paradigm: Survey and open issues," *IEEE Access*, vol. 6, pp. 18209–18237, 2018.

[3] T. Goethals, F. De Turck, and B. Volckaert, "Extending kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Trans. Cloud Comput.*, vol. 10, no. 4, pp. 2623–2636, Oct. 2022.

[4] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," in *Proc. IEEE/ACM Symp. Edge Comput. (SEC)*, Oct. 2018, pp. 373–377.

[5] K. Lee and B. Tak, "MicroVM on edge: Is it ready for prime time?" in *Proc. 31st Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, Oct. 2023, pp. 1–8.

[6] P. P. Ray, "An overview of WebAssembly for IoT: Background, tools, state-of-the-art, challenges, and future directions," *Future Internet*, vol. 15, no. 8, p. 275, Aug. 2023.

[7] (Oct. 2024). *Open Container Initiative—An Open Governance Structure for the Express Purpose of Creating Open Industry Standards Around Container Formats and Runtimes*. [Online]. Available: https://opencontainers.org/

[8] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Understanding container network interface plugins: Design considerations and performance," in *Proc. IEEE Int. Symp. Local Metrop. Area Netw. (LANMAN*, Jul. 2020, pp. 1–6.

[9] C. Marcelino and S. Nastic, "CWASI: A WebAssembly runtime shim for inter-function communication in the serverless edge-cloud continuum," in *Proc. 8th ACM/IEEE Symp. Edge Comput.*, Dec. 2023, pp. 158–170.

[10] I. Mavridis and H. Karatza, "Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud," *Concurrency Computation: Pract. Exper.*, vol. 35, no. 11, p. 6365, May 2023.

[11] (Nov. 2024). *Hyper-Efficient Serverless on Kubernetes, Powered By Webassembly*. [Online]. Available: https://www.spinkube.dev/

[12] (Nov. 2024). *Docker–Alternative Container Runtimes*. [Online]. Available: https://docs.docker.com/engine/daemon/alternative-runtimes/

[13] (Sep. 2024). *Virtual Kubelet—An Open Source Kubernetes Kubelet Implementation Connecting Kubernetes To Other APIs*. [Online]. Available: https://github.com/virtual-kubelet/virtual-kubelet

[14] T. Goethals, M. De Clercq, M. Sebrechts, F. De Turck, and B. Volckaert, "Feather: Lightweight container alternatives for deploying workloads in the edge," in *Proc. 14th Int. Conf. Cloud Comput. Services Sci.*, 2024, pp. 27–37.

[15] A. Randazzo and I. Tinnirello, "Kata containers: An emerging architecture for enabling MEC services in fast and secure way," in *Proc. 6th Int. Conf. Internet Things: Syst., Manage. Secur. (IOTSMS)*, Oct. 2019, pp. 209–214.

[16] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 461–472, 2013.

[17] T. Goethals, M. Sebrechts, A. Atrey, B. Volckaert, and F. De Turck, "Unikernels vs containers: An in-depth benchmarking study in the context of microservice applications," in *Proc. IEEE 8th Int. Symp. Cloud Service Comput. (SC2)*, Nov. 2018, pp. 1–8.

[18] F. Moebius, T. Pfandzelter, and D. Bermbach, "Are unikernels ready for serverless on the edge?" 2024, *arXiv:2403.00515*.

[19] X. Wang, J. Du, and H. Liu, "Performance and isolation analysis of RunC, gVisor and kata containers runtimes," *Cluster Comput.*, vol. 25, no. 2, pp. 1497–1513, Jan. 2022.

[20] T. Goethals, M. Sebrechts, M. Al-Naday, B. Volckaert, and F. De Turck, "A functional and performance benchmark of lightweight virtualization platforms for edge computing," in *Proc. IEEE Int. Conf. Edge Comput. Commun. (EDGE)*, Jul. 2022, pp. 60–68.

[21] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Junior, and L. F. M. Vieira, "Fast packet processing with eBPF and XDP: Concepts, code, challenges, and applications," *ACM Comput. Surveys*, vol. 53, no. 1, pp. 1–36, Feb. 2020, doi: 10.1145/3371038.

[22] J. Nam, S. Lee, P. Porras, V. Yegneswaran, and S. Shin, "Secure inter-container communications using XDP/eBPF," *IEEE/ACM Trans. Netw.*, vol. 31, no. 2, pp. 934–947, Apr. 2023.

[23] S.-Y. Wang and J.-C. Chang, "Design and implementation of an intrusion detection system by using extended BPF in the Linux kernel," *J. Netw. Comput. Appl.*, vol. 198, Feb. 2022, Art. no. 103283. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804521002769

[24] W. Yang, P. Chen, G. Yu, H. Zhang, and H. Zhang, "Network shortcut in data plane of service mesh with eBPF," *J. Netw. Comput. Appl.*, vol. 222, Feb. 2024, Art. no. 103805. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804523002242

[25] F. Parola, L. D. Giovanna, G. Ognibene, and F. Risso, "Creating disaggregated network services with eBPF: The kubernetes network provider use case," in *Proc. IEEE 8th Int. Conf. Netw. Softwarization (NetSoft)*, Jun. 2022, pp. 254–258.

[26] (Sep. 2023). *Cilium.io–What is Cilium?*. [Online]. Available: https://cilium.io/get-started/

[27] 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 Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 133–151, Mar. 2021.

[28] M. Jadin, Q. De Coninck, L. Navarre, M. Schapira, and O. Bonaventure, "Leveraging eBPF to make TCP path-aware," *IEEE Trans. Netw. Service Manage.*, vol. 19, no. 3, pp. 2827–2838, Sep. 2022.

[29] G. Koukis, S. Skaperas, I. A. Kapetanidou, L. Mamatas, and V. Tsaoussidis, "Performance evaluation of kubernetes networking approaches across constraint edge environments," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jun. 2024, pp. 1–6.

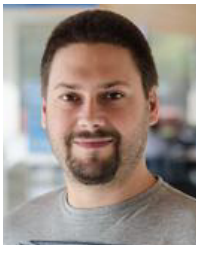
[30] (Nov. 2018). *Nabla on Kubernetes!*. [Online]. Available: https://nabla-containers.github.io/2018/11/05/nabla-k8s/

[31] (Sep. 2025). *Multus-CNI—Multi-Homed Pods*. [Online]. Available: https://github.com/k8snetworkplumbingwg/multus-cni

[32] (Oct. 2025). *Confidential Containers–Deploy Cloud Native Applications Inside Confidential Enclaves*. [Online]. Available: https://confidentialcontainers.org/

[33] T. Goethals, M. Al-Naday, B. Volckaert, and F. De Turck, "Warrens: Decentralized connectionless tunnels for edge container networks," *IEEE Trans. Netw. Service Manage.*, vol. 21, no. 4, pp. 4282–4296, Aug. 2024.

[34] (Nov. 2024). *Universal Tun/tap Device Driver*. [Online]. Available: https://docs.kernel.org/networking/tuntap.html



TOM GOETHALS (Member, IEEE) is currently a Senior Researcher with Ghent University and IMEC'S IDLaboratory Group, teaching distributed data processing, distributed systems management, and computer security. He has worked on several national and international research projects and is author or co-author of 20 peer-reviewed papers published in international journals and conference proceedings. His research deals with intelligent, decentralized cloud-edge discovery and orchestration, and optimization of orchestration components for low-resource edge devices. He received four awards.

MIEL VERKERKEN received the M.Sc. degree in information engineering technology from Ghent University, in 2018, and the Ph.D. degree from the Internet and Data Science Laboratory (IDLab-IMEC), Ghent University, in 2025. In 2024, he was a Visiting Researcher with the University of Liechtenstein. He co-authored 15 peer-reviewed publications in international journals and conference proceedings. His research lies at the intersection of networking, security, and machine learning, with a focus on network intrusion detection systems.

MERLIJN SEBRECHTS (Member, IEEE) is currently a Senior Researcher with IMEC and teaches with Ghent University, Belgium. He is the Ubuntu Community Council and is standardizing WebAssembly System Interfaces for the IoT devices as part of the W3C and the Bytecode Alliance. He teaches topics, such as distributed systems design, open source ecosystems, and computer security. His work has been published in over 20 scientific publications. He leads a number of research tracks focused on software deployment and trust in the cloud and on devices. He received four awards.

FILIP DE TURCK (Fellow, IEEE) leads with the Network and Service Management Research Group, Ghent University, Belgium, and IMEC. He co-authored over 750 peer reviewed papers. He is involved in several research projects with industry and academia. His research interests include design of efficient softwarized network and cloud systems. He serves as a Steering Committee Member for the IM, NOMS, CNSM, and NetSoft conferences. He received the IEEE ComSoc Dan Stokesberry Award, in 2021. He served as the Chair for IEEE Technical Committee on Network Operations and Management (CNOM). He served as an Editor-in-Chief for IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT.

BRUNO VOLCKAERT (Senior Member, IEEE) is currently a Professor of advanced software engineering and secure distributed systems with Ghent University and IMEC'S IDLaboratory Group. He has worked on over 80 national and international research projects and is author or co-author of more than 250 peer-reviewed papers published in international journals and conference proceedings. His research deals with reliable and high-performance distributed software for a.o. scalable data ingestion and processing, secure software architectures, and autonomous optimization of cloud/edge-based applications.