Assessing Container Network Interface Plugins: Functionality, Performance, and Scalability

Shixiong Qi, Sameer G. Kulkarni, and K. K. Ramakrishnan, *Fellow, IEEE*

***Abstract*—Kubernetes, an open-source container orchestration platform, has been widely adopted by cloud service providers (CSPs) for its advantages in simplifying container deployment, scalability, and scheduling. Networking is one of the central components of Kubernetes, providing connectivity between different Pods (a group of containers) both within the same host and across hosts. To bootstrap Kubernetes networking, the Container Network Interface (CNI) provides a unified interface for the interaction between container runtimes. There are several CNI implementations, available as open-source 'CNI plugins'. While they differ in functionality and performance, it is a challenge for a cloud provider to differentiate and choose the appropriate plugin for their environment. In this article, we compare the various open-source CNI plugins available from the community, qualitatively, and through detailed quantitative measurements. With our experimental evaluation, we analyze the overheads and bottlenecks for each CNI plugin, especially because of the interaction with the datapath/iptables as well as the host network stack. Overlay tunnel offload support in the network interface card plays a significant role in achieving the good performance of CNIs that use overlay tunnels for inter-host Pod-to-Pod communication. We also study scalability with an increasing number of Pods, as well as with HTTP workloads, and briefly evaluate Pod startup latency. Our measurement results inform the outline of an ideal CNI environment for Kubernetes.**

***Index Terms*—Containers, container networking interface, Kubernetes, performance, scalability.**

I. Introduction

KUBERNETES is the leading container orchestration platform used by cloud service providers (CSPs) to improve the utilization of their cloud resources [1]. Kubernetes provides the flexibility to run a variety of containerized cloud applications, with the ability to deploy on both physical and virtual cloud resources. In Kubernetes, a "Pod" is the atomic unit of deployment, for scaling and management [2]. A Pod may comprise one or more containers that share the same resources including the networking context. Pods can be scaled to multiple instances to meet the workload characteristics and also to provide failure resiliency.[1]

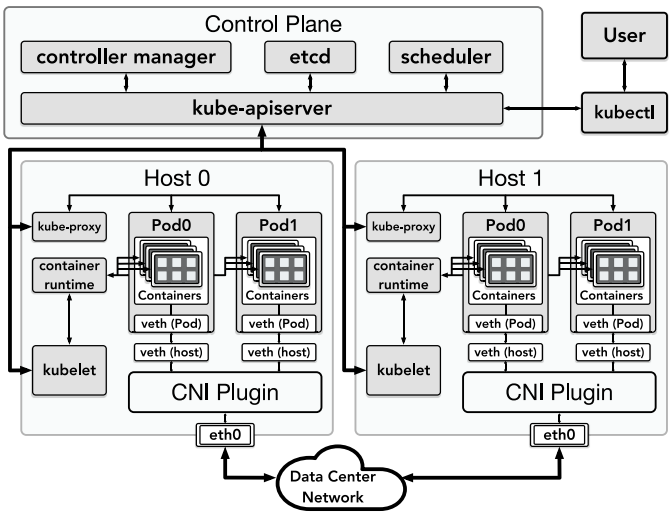


Fig. 1. Kubernetes Cluster and the Role of CNI.

The proliferation of microservices [4] and function-as-a-service [5] architectures for deploying cloud-based services make it necessary to support large numbers of containers, and to provide efficient communication between them. Hence, orchestration and networking are critical and need to be automated, scalable, and secure to benefit large production deployments. Kubernetes adopts the Container Network Interface (CNI) specification as its core networking foundation [6]. Each Pod in a Kubernetes cluster is given a unique IP address for communication. A general architecture overview of a Kubernetes cluster is shown in Fig. 1, with one control plane host and two worker hosts. The control plane host is in charge of maintaining the cluster state, and the worker host is responsible for running cloud workloads in the execution unit named Pod, while the CNI Plugin facilitates communication among these execution units [7]. There are a number of different CNI implementations, and these CNI 'plugins' perform the tasks for Pod networking in a Kubernetes cluster. With thousands of Pods running in a cluster, the network's status can change rapidly, with frequent creation and/or termination of Pods. When a new Pod is added, the CNI plugin coordinates with the container runtime and connects the container

Manuscript received June 16, 2020; revised November 2, 2020 and December 15, 2020; accepted December 15, 2020. Date of publication December 25, 2020; date of current version March 11, 2021. This work was supported by the U.S. NSF, through grant CNS-1763929. The associate editor coordinating the review of this article and approving it for publication was G. Schembra. *(Corresponding author: K. K. Ramakrishnan.)*

Shixiong Qi and K. K. Ramakrishnan are with the Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92508 USA (e-mail: kk@cs.ucr.edu).

Sameer G. Kulkarni is with the Department of Computer Science and Engineering, Indian Institute of Technology Gandhinagar, Gandhinagar 382355, India.

Digital Object Identifier 10.1109/TNSM.2020.3047545

[1] *This article provides significant additions on our previously published IEEE LANMAN 2020 [3] conference paper.*

network namespace with the host network namespace (e.g., by setting up the virtual ethernet (veth) pair), assigns a unique IP address to the new Pod, applies the desired network policies and distributes routing information to the rest of the cluster.

Several open-source CNI plugins are available for use in a Kubernetes environment. Amongst them, Flannel [8], Weave Net (or Weave) [9], Cilium [10], Calico [11], and Kube-router [12] are popular and have been adopted by many Kubernetes distributions [13]. While each finds their application in different contexts due to their unique and distinct networking characteristics, we believe there is an inadequate understanding and a lack of a comprehensive characterization of these different CNI plugins on both the qualitative and performance aspects. To better understand the operations of different CNI plugins, it is necessary to generalize the working of the different CNI plugins based on the underlying network model and identify the key implementation differences and their corresponding impact on the performance and scalability aspects. While existing works [14]–[19] study the overall performance of different CNI plugins at a preliminary level, there is still a lack of in-depth understanding on how the various design considerations affect performance. Besides, there is also a lack of examination on the CNI plugins' performance under large scale deployments. With the help of autoscaling feature, the number of Pods on a single host can be easily scaled up to hundreds [20], [21], which results in additional interference and impact on the performance of the CNI plugins. Hence an analysis of the impact of background communication for different CNIs is necessary to understand scalability, and associated performance impact.

In this article, we provide an insight into the overall performance of Pod networking with different CNI plugins, by examining throughput, latency, and fine-grained CPU measurements of the various components of the networking stack. First, we present a qualitative analysis of the popular CNI plugins, namely: Flannel, Weave, Cilium, Calico, and Kube-router, to provide a high-level operational view of the feature support of different CNIs. We also consider the different variants of the Calico to demonstrate the effect of tunneling and overheads with different encapsulation modes. We omit the other less frequently used (and some outdated) CNIs such as Romana [22], Canal [23], and Contiv-vpp [24] from our study. Next, we provide a measurement-driven quantitative analysis of their performance for various communication modes. To summarize, the contributions of our work include:

1) We provide qualitative analysis for different CNI plugins in terms of the subset of network or datalink layer features they support (e.g., IPv6, encryption support), and accordingly classify and generalize the CNI Plugin networking model into four different classes.
2) We analyze the interactions with the host networking stack including the network filter configurations (iptables rules) across different dimensions (e.g., iptables chains, packet forwarding, overlay tunneling, extended Berkeley Packet Filter (eBPF), *etc.*) to determine the critical function calls that contribute to overhead.

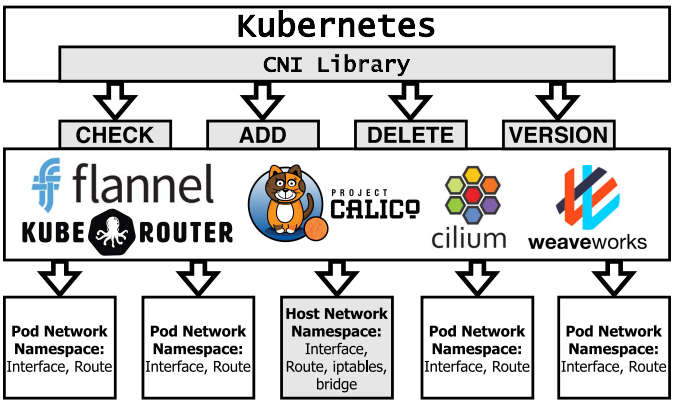


Fig. 2. CNI Plugin: Interfacing w/Kubernetes & Namespaces.

3) Based on this qualitative analysis, we examine the root cause for the performance differences across different CNI plugins for packet transmission throughout including the entire network protocol stack with a measurement-based quantitative evaluation.
4) We present extensive performance analysis by considering different real-world traffic patterns for varying scale of deployment Pods and concurrent clients to help understand the suitability of using different CNIs. We briefly examine Pod startup latency.

A technical report [25], an extended version of this article provides additional details, particularly on Pod startup time.

II. Background

In Kubernetes, networking plays a pivotal role in enabling the cluster-wide communication among the Pods. In this section, we briefly present the details of the Kubernetes networking and CNI plugin models.

A. Container Network Interface

The Container Network Interface (CNI) is a container networking specification proposed by CoreOS [6] and has been adopted by several open-source projects such as Cloud Foundry, Kubernetes, Mesos, etc. and has also been accepted by the Cloud Native Computing Foundation (CNCF) [26] as an industry standard for container networking.[2] Container Network Model (CNM) is an alternative container networking standard proposed by Docker [28]. Although, both CNI and CNM are modular and provide plugin-based interfaces for network drivers to create, configure, and manage networks, the CNM is designed to support only the Docker runtime, while CNI can be supported with any container runtime [29].

The CNI specification defines a simple set of interfaces (e.g., CHECK, ADD, DELETE) for adding and removing a container from the network. With the help of CNI plugin, network interface, route, iptables, *etc* can be efficiently set up in a Pod/host network namespace. A modular/driver-based approach allows the integration with several 3rd party

[2]CNCF is backed by a large number of companies that currently support CNI being the de facto standard for container networking [27].

TABLE I
QUALITATIVE COMPARISON ON THE FEATURES OF DIFFERENT CNI PLUGINS

CNI Solutions	*Network Model*	*Tunneling Options*	*Network Policies*		*Packet Encryption*	*Additional Features*	*Default MTU (Bytes)*
			Kubernetes / 3rd party	*Ingress / Egress*			
Flannel	Hybrid + Underlay/Overlay	VXLAN	No / No	No / No	No	–	1450
Weave Net	Hybrid + Overlay	VXLAN	Yes / No	Yes / Yes	Yes	Multicast	1376
Cilium	L3 + Underlay/Overlay	VXLAN/Geneve	Yes / Yes	Yes / Yes	Yes	IPv6	1500
Calico	L3 + Underlay/Overlay	IP-in-IP/VXLAN	Yes / Yes	Yes / Yes	No	IPv6	1440
Kube-router	Hybrid + Underlay/Overlay	IP-in-IP	Yes / No	Yes / No	No	IPVS/LVS, DSR	1500

implementations of the CNI specification, called the CNI plugins. A CNI plugin is implemented as an executable and the container runtime is responsible for invoking this plugin to set up and destroy the container network stack as shown in Fig. 2. The CNI plugin is then responsible for IP Address Management (IPAM), to connect the Pod network namespace with the host network, provide IP address allocation to the container network interface, manage the IP address allocation across different Pods in the cluster, configure the routes on both the host and Pod network namespaces, etc.

"CNI plugins" comprise of two major components, namely the CNI daemon and CNI binary files. The CNI daemon is mainly used to do network management jobs, such as updating the routing information of the hosts,[3] maintaining network policies,[4] renewing the subnet leasing,[5] Border Gateway Protocol (BGP) updating,[6] and maintaining some self-defined resources (e.g., IP-Pool in Calico).[7] The CNI binary files are mainly used to create the network devices (e.g., Linux bridge) and allocate IP address to Pods. Although it is possible to concurrently support multiple CNIs for a Pod (*i.e.*, setup a Pod with multiple network interfaces, which are managed by different CNIs), we focus on cases having only a single CNI.

B. Linux Network Namespace and Kubernetes Namespace

Before we delve into the Kubernetes networking model, it is necessary to understand the namespace model and distinguish between the Linux network namespace and Kubernetes namespace and the associated impact on setting up of the CNI.

Linux network namespace is designed for network isolation. Each Pod has its own network namespace, which is isolated from the host network namespace and the network namespace of other Pods. This enables the Pod to operate its own network stack and interfaces without interference and collision with other Pods. When a new Pod is created, the container runtime interface (CRI) creates the network namespace for the new Pod. Thereafter it invokes the CNI plugin, which allocates the IP address for the Pod, attaches the virtual Ethernet pair (veth-pair) to link up the Pod's network namespace to the host network namespace, and add the corresponding routing and network policy rules.

On the other hand, the Kubernetes namespace is used to divide the physical cluster into multiple virtual clusters. This enables sharing the physical cluster resources among different groups of users and makes the management of the cluster more flexible. This also means that the Pods in different Kubernetes namespaces are not strictly isolated. Kubernetes starts with several system namespaces typically prefixed with 'kube-' (e.g., kube-system, kube-pubilc, kube-node-lease, and default). Users are also allowed to create their own Kubernetes namespaces to isolate their workloads from other users. When creating the user namespaces, users can specify the resource quota for the created namespace, such as the maximum number of running Pods, CPU, and memory limits to avoid the threat of exorbitant resources requests. The Kubernetes namespace can be used as a selector in the network policy, to apply a specific network policy to a group of Pods in that namespace, which decouples the Pods from their static IP addresses and improves the resource management efficiency in a large scale cluster.

C. Kubernetes Networking Model

The Kubernetes networking model is proposed for dealing with four different kinds of communication: i) intra-Pod or Container-to-Container communication within a Pod, ii) inter-Pod or Pod-to-Pod communication, iii) Service-to-Pod communication and iv) External-to-Service communication [30]. To achieve these four communication services, Kubernetes only provides the specification of the network model, while the actual implementation is handed over to the CNI plugins. The key requirements of the Kubernetes network model include i) Pods are IP addressable and must be able to communicate with all other Pods (on the same or different host) without the need for network address translation (NAT), and ii) all the agents on a host (e.g., Kubelet) are able to communicate with all the Pods on that host. CNI plugins may differ in their architecture but meet the above network rules. Thus, there is a range of CNI plugins that adopt different approaches. Popular CNI plugins are Flannel, Weave, Calico, Cilium, and Kube-router [30].

D. Kubernetes Network Policy

Kubernetes Network Policy is the means to enforce rules indicating which network traffic is allowed and which Pods can communicate with each other. The policies applied to Pod network traffic can be based on their applicability to ingress traffic (entering the Pod) and egress traffic (outgoing traffic). The control strategies include "allow" and "deny". By default, a Pod is in a non-isolated state. Once a network policy is applied to a Pod, all traffic that is not explicitly allowed will

[3]https://docs.projectcalico.org/reference/architecture/data-path

[4]https://kubernetes.io/docs/concepts/services-networking/network-policies

[5]https://github.com/coreos/flannel/blob/master/Documentation/reservations.md

[6]https://docs.projectcalico.org/networking/bgp

[7]https://docs.projectcalico.org/reference/resources/ippool

be rejected by the network policy. However, other Pods that do not have network policies applied to them are not affected. CNI plugins in Kubernetes can implement elaborate traffic control and isolation mechanisms.

III. Qualitative Comparison on CNI Plugins

The different open source CNIs vary in their design and approach towards facilitating intra-host and inter-host Pod communication, and their support for Pod network policies. We provide a careful qualitative analysis of different CNIs based on the layer of operation, packet forwarding and routing approach for Pods within the same host or across hosts. CNI performance and scalability are influenced by the overheads in the network protocol stack (typically in the kernel). We also consider several other factors such as support for encryption, IPv6, and multicast functionalities.

Network model: As Pods are uniquely identified by their IP addresses, CNIs primarily operate at Layer-3 (Network) to facilitate inter-Pod communication. However, CNIs can also operate at Layer-2 (Link) for intra-host Pod communication, e.g., using the 'Bridge' or 'MacVLAN' capabilities. Layer-2 CNIs take advantage of a software Linux bridge. This greatly simplifies the configuration and management of Pod networking, especially for the intra-host Pod-to-Pod communication. However, they suffer from scale limitations and also exhibit significant time to adapt to the network changes because of MAC learning and forwarding updates. Layer-3 CNIs, based on IP routing and forwarding, are better for scalability and to support cluster-wide communication. They often depend on Border Gateway Protocol (BGP) support to cross autonomous system (AS) boundaries, which could be a potential security concern in some cloud sites [31]. BGP suffers from its security vulnerabilities, such as BGP hijack [32], route leaks [33], etc. Alternatively, the 'hybrid Layer-2/Layer-3 solutions' (*i.e.*, Layer-2 for intra-host and Layer-3 for inter-host) may facilitate efficient intra-host, and scalable inter-host, communication.

Packet Forwarding and Routing: Another aspect to consider is the process for packet forwarding between hosts, which has a significant impact on the performance (*i.e.*, latency and throughput). Based on how the packets are forwarded across hosts, the CNI can be classified into two categories:

1) Overlay networking is a virtual network that is built on top of an underlying physical infrastructure, which not only provides new isolation or security benefits but also gets around the dependency (e.g., IP address, routing, etc) on the underlying infrastructure support. All the hosts in an overlay network communicate with each other via the virtual links, which are also called overlay tunnels. When packets go through the overlay tunnel, they are encapsulated with an outer header based on the adopted overlay protocol, such as Virtual Extensible LAN (VxLAN), Generic Routing Encapsulation (GRE), etc. Although using overlay technologies remove the dependency on the underlying infrastructure, it increases the difficulty to trace data packets when errors occur in addition to the performance reduction.

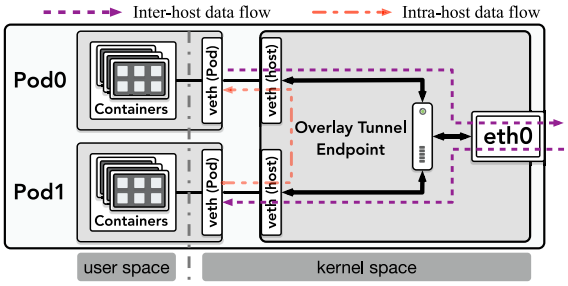


Fig. 3. Network model for CNIs operating at Layer-3 in overlay mode (e.g., Calico, Cilium).

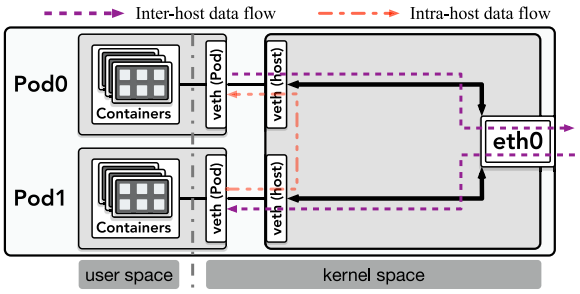


Fig. 4. Network model for CNIs operating in Layer-3 in underlay mode (e.g., Calico, Cilium).

2) Underlay networking is just the datalink layer or Layer-3 (IP) infrastructure for packet forwarding. The underlay network provides the native routing connectivity between different hosts in the cluster. This typically requires BGP, where the hosts act as BGP peers and share the routing information among them to support inter-host routing without the need for any encapsulation.

Based on the existing implementations of different CNI Plugins, we classify the networking models and datapath design into the following four broad classes: i) Layer-3 + Overlay; ii) Layer-3 + Underlay; iii) Hybrid + Overlay; iv) Hybrid + Underlay.

Layer-3 + Overlay uses an overlay tunnel endpoint (OTEP) and multiple veth-pairs (Fig. 3). The OTEP is used to encapsulate/decapsulate the packets. The veth-pair enables data exchange between the Pod network namespace and the host network namespace. The intra-host data exchange is handled by the host protocol stack in Layer-3. For inter-host case, the outgoing data packet is delivered to the OTEP via IP forwarding, where the packets get encapsulated at the endpoint and sent to its destination via the host's physical interface.

Layer-3 + Underlay comprises of veth-pairs (Fig. 4). The intra-host data exchange with IP routing works the same as in the overlay-based design. The inter-host communication is also processed in the host at Layer-3.

Hybrid + Overlay combines the hybrid Layer-2/Layer-3 design with overlays. It consists of a Linux bridge, an OTEP, and multiple veth-pairs (Fig. 5). A Linux bridge in the host network namespace connects with the Pods through veth-pairs facilitating intra-host data exchange. For inter-host data exchanges, the outgoing data packet first arrives at the bridge

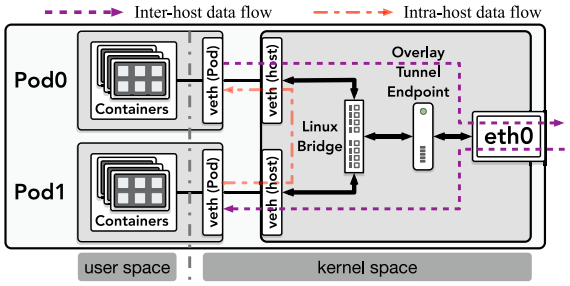


Fig. 5. Network model for CNIs operating in both Layer-2 & Layer-3 in overlay mode (e.g., Flannel, Weave, Kube-router).

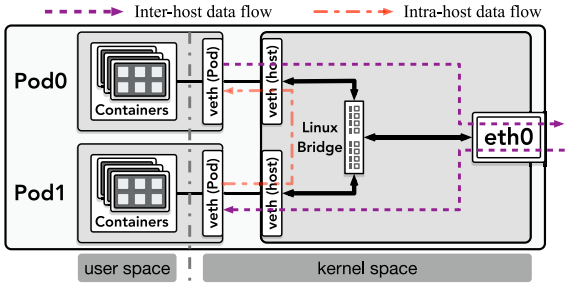


Fig. 6. Network model for CNIs operating in both Layer-2 & Layer-3 in underlay mode (e.g., Flannel, Kube-router).

and is then handed over to the host protocol stack operating at Layer-3. The host protocol stack forwards the packet to OTEP via IP forwarding. At the OTEP, the packet is encapsulated (based on the overlay encapsulation type) and sent to the destination via host eth0.

Hybrid + Underlay comprises multiple veth-pairs and a Linux bridge (Fig. 6). The intra-host data exchange here works the same as in the Hybrid+Overlay approach. For the inter-host communication, the outgoing data packet first arrives at the bridge, which is then handed over to the host protocol stack operated in Layer-3. With the host's IP forwarding turned on, the data packets will be sent through the host's physical interface to the other hosts.

Kubernetes Network Policy: Before implementing network policies in a Kubernetes cluster, a network policy controller needs to be installed. This is provided by the CNI plugin. The CNI plugins which support the network policy include Weave, Calico, Cilium, and Kube-router. The CNI plugin daemon works as a Network Policy Controller (a daemon ensures that some or all hosts in a Kubernetes cluster run a copy of a Pod). Users can provide the Network Policy annotations to the daemon to enable various filtering rules.

A. Description of Each CNI Plugin

1) Flannel:

i) *Layer of Operation:* Flannel uses a combination of Layer-2 and Layer-3 operation. Pods in the same host can use the Linux bridge to communicate (Layer-2), while Pods on different hosts use an overlay tunnel endpoint to encapsulate their traffic and use Layer-3 routing and forwarding.

ii) *Packet Forwarding across Hosts:* With its default settings, Flannel configures the Kubernetes cluster with an overlay network. Flannel has several different types of overlay backends (e.g., VxLAN, UDP) that can be used for encapsulation and routing. The default and recommended method is to use VxLAN because of its better performance. The VxLAN backend is in kernel space while the UDP backend works in the user space, which has more context switches (up to 3) compared to the VxLAN mode. Besides the overlay mode, Flannel also has an optional underlay mode, which transfers packets using IP routing (Layer-3). However, this mode requires direct Layer-2 connectivity between communicating containers from the underlying network infrastructure. In both the VxLAN overlay and underlay modes, only one context switch (userspace to kernel space) happens when the packets are delivered from the Pods into the host network stack. However, when using the UDP overlay mode, the packets need to be encapsulated with a UDP header via the Flannel daemon in the user space. After the packets are delivered from the Pod into the host network stack, it will be delivered back to the userspace to be processed by the Flannel daemon. Then they will enter into the host network stack again to be sent to its destination. Hence, in total three context switches happen under the UDP mode resulting in more CPU overhead and poor performance.

iii) *Network Policy Support:* Flannel does not implement the network policy controller of its own hence lacks the support to realize any network policies.

iv) *Miscellaneous:* In the overlay network, each host has its own subnet (a fixed Classless Inter-Domain Routing), which is used to allocate IP addresses internally. When spinning up a new Pod, the Flannel daemon on each host will assign an address to each new Pod from that address pool.

Note: We configure Flannel in VxLAN mode.

2) Weave:

i) *Layer of Operation:* Likewise to Flannel, Weave also operates at both Layer-2 and Layer-3. Intra-host communication is performed at Layer-2, where the packets are forwarded to its destination via the Linux bridge and Layer-3 for the inter-host communication.

ii) *Packet Forwarding across Hosts:* Weave only uses overlay links (VxLAN or UDP) for communication between hosts. The VxLAN mode is running based on the kernel's native Open vSwitch datapath module, while the UDP mode relies on the Weave CNI daemon to implement encapsulation. Likewise to Flannel, Weave incurs one context switch when running in VxLAN mode, and three context switches when running in UDP encapsulation mode.

iii) *Network Policy Support:* Weave provides network policy support for Kubernetes clusters. When setting up Weave, a network policy controller is automatically installed and configured. Weave can only configure the standard Kubernetes Network Policy, which implements network policies on Layer-3 (Network) and Layer-4 (Transport)

attributes. Weave implements its network policy based on iptables. Weave uses state extension in iptables, which is a subset of the connection tracking extension (conntrack). Weave uses state extension to speed up the processing of iptables. For an established connection, only the first packet needs to be matched with the iptables, the remaining packets will be allowed to pass directly. Moreover, Weave uses 'ipset' to speedup iptables processing. Ipset uses a hash table to map a rule to a set of IP addresses, and a hash table lookup for a packet to find the target rule.

iv) *Miscellaneous:* A feature unique to Weave, which is not available in most of the existing CNI plugins, is simple encryption of the traffic based on NaCl (a networking and cryptography library),[8] with a user API to simplify the implementation of encryption in the networking system.[9] Multicast support is provided in Weave to improve throughput and save bandwidth for applications such as streaming video or the exchange of lots of data across multiple containers.[10]

Note: We configure Weave in VxLAN mode.

3) Calico:

i) *Layer of Operation:* Unlike Flannel and Weave, Calico operates at Layer-3 for both intra-/inter-host communication.

ii) *Packet Forwarding across Hosts:* Calico allows for both underlay/overlay packet forwarding across hosts. The underlay mode uses native routing based on BGP. Calico uses BGP to distribute and update routing information providing better scalability and performance.[11] Calico's overlay mode uses IP-in-IP or VxLAN encapsulation. Calico only incurs one context switch for both the overlay modes using IP-in-IP or VxLAN and underlay mode using BGP.

iii) *Network Policy Support:* Calico has very good features for Kubernetes network policy customization. Users can enable both the Kubernetes network policy as well as Calico's own network policy, covering the policy from Layers 3 to 7. Calico can also be integrated with the service mesh, *Istio*, to implement strategies for workloads within the cluster at the service mesh layer.[12] This means that users can configure iptables rules that describe how Pods should send and receive traffic, thus enhancing the security of the network environment. Calico implements its network policy based on iptables. It inserts user-defined chains on top of the system default chain. Calico uses conntrack to optimize its iptables chains just like Weave. Once a connection has been established, the following packets will be allowed to pass directly instead of being examined by the iptables. In addition, Calico also uses ipset to provide better scale and performance than default iptables.[13]

iv) *Miscellaneous:* Calico provides IPv6 support. However, it is limited to underlay mode only and the overlay (with IP in IP and VxLAN) can support only IPv4 addresses. With IPv6 enabled, the larger address space allows for scalability in the number of endpoints in the Kubernetes cluster.

Note: Calico can be run in two intra-host modes: Calico-wp (native routing with network policy) and Calico-np (native routing without network policy). In 'Calico-wp', we consider the ordering/priority, allow/deny rules, *etc* rules to be enabled. For the inter-host scenario, Calico has two network models ('Layer-3 + overlay' & 'Layer-3 + underlay'). This in conjunction with network policy support provides four different modes namely Calico-wp-ipip, Calico-wp-xsub, Calico-np-ipip, and Calico-np-xsub. "ipip" means the IP-in-IP overlay mode and "xsub" means IP based underlay. Accordingly, we evaluate each of these distinct configuration modes.

4) Cilium:

i) *Layer of Operation:* Cilium is also a Layer 3-only solution. For intra-host communication, Cilium relies on eBPF programs attached at the veth-pairs to redirect packets to their destination. Cilium builds its datapath based on a set of eBPF hooks that run eBPF programs. The eBPF hooks used in Cilium include XDP (eXpress Data Path), Traffic Control ingress/egress (TC), Socket operations, and Socket send/recv. TCs, attached to the veth, are utilized to forward packets through eBPF function calls, e.g., *bpf_lxc*, *bpf_netdev*.[14] Arriving packets at the veth cause eBPF programs to be executed and route traffic.

ii) *Packet Forwarding across Hosts:* Cilium allows for both an underlay (IP) packet forwarding and an overlay solution with VxLAN or Geneve as the encapsulation options. Like Calico, Cilium's underlay mode uses native routing based on BGP. If Cilium uses the overlay solution, *bpf_overlay* will be executed to direct the packet from veth to the OTEP. Like Calico, Cilium also incurs one context switch for both overlay and underlay mode of operation.

iii) *Network Policy Support:* Cilium supports both the standard Kubernetes network policy and its own network policy customization based on the iptables. These network policies can work in Layers 3-7. In addition, Cilium can use eBPF hooks (e.g., XDP, TC) to define packet filters. Since this filtering occurs earlier than the network protocol stack, it can achieve better performance than iptables.

iv) *Miscellaneous:* Cilium supports packet encryption in Layer-3, *i.e.*, it provides encryption using the IPSec tunnels. Cilium also provides support for IPv6.

Note: We configure Cilium in VxLAN mode.

5) Kube-Router:

i) *Layer of Operation:* Kube-router operates in both Layer-2 and Layer-3. Kube-router uses Linux bridge to forward packets intra-hosts (Layer-2). It uses Layer-3 operation to forward packets inter-hosts.

[8]http://nacl.cr.yp.to/

[9]https://www.weave.works/docs/net/latest/concepts/encryption/

[10]https://www.weave.works/use-cases/multicast-networking/

[11]https://www.projectcalico.org/why-bgp/

[12]https://www.projectcalico.org/category/istio/

[13]https://docs.projectcalico.org/about/about-network-policy

[14]https://docs.cilium.io/en/v1.7/architecture/

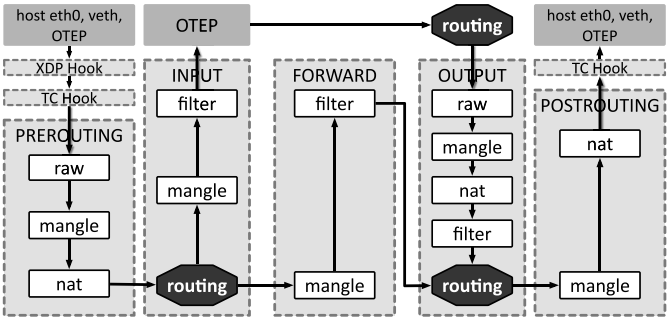


Fig. 7. Iptables chain processing and hook points [34].

ii) *Packet Forwarding across Hosts:* Kube-router allows for both an underlay (IP) packet forwarding across the hosts based on BGP and an overlay solution using IP-in-IP encapsulation. As with Calico and Cilium, packet forwarding in Kube-router incurs just one context switch for both overlay and underlay modes.

iii) *Network Policy Support:* Kube-router only supports the standard Kubernetes Network Policy APIs to apply at Layer-3 and Layer-4. Kube-router implements its network policy based on iptables. Kube-router uses conntrack to speed up the processing of iptables. Kube-router also uses ipset to get around the overhead of a large iptables.

iv) *Miscellaneous:* An important feature of Kube-router is the usage of IPVS/LVS kernel features to improve service load balancing performance.[15] Kube-router applies a Direct Server Return (DSR) feature to implement a high-efficient ingress for load balancing, which is a unique feature compared to other CNIs.[16]

Note: We configure Kube-router in IP-based underlay mode.

B. Iptables Comparison

In order to implement specific packet forwarding, routing, and networking policies, the CNI Plugins leverage 'iptables' - a userspace interface to setup, maintain and inspect the tables of IP packet filter (Netfilter) rules in the Linux kernel (we use the terms iptables and Netfilter interchangeably in this article). Netfilter registers five hook points (callback function points) into the network stack to implement packet filtering, NAT, and security-related policies. The five hook points are: *PREROUTING*, *INPUT*, *OUTPUT*, *FORWARD* and *POSTROUTING*. These are 'chains' of tables of iptable-related processing of a packet, for the purpose of filtering and execution of security policies. Fig. 7 shows the iptable chain processing. Each iptables chain comprises a number of tables, each with a different purpose. There are four kinds of tables: *raw*, *mangle*, *nat*, and *filter*. The *raw* table is used to split the traffic without a need for the connection to be tracked. The *mangle* table is used to change the QoS settings of packets. The *nat* table is for network address translation and the *filter* table is used for packet filtering.

[15]https://cloudnativelabs.github.io/post/2017-05-10-kube-network-service-proxy/

[16]https://www.kube-router.io/docs/user-guide/

Based on the user-defined network policies for different Pods (input as a *Yet Another Markup Language* file), the CNI's network policy controller configures the iptables on all the Kubernetes worker hosts. When packets arrive at an iptables chain, a *nf_hook_slow*() function call is executed to traverse the list of tables in that chain and process the matching iptable rules. This rule processing can be a major source of overhead in iptables [34].

Moreover, when a host starts up, the kube-proxy installs a set of default iptables rules on the worker hosts. These default iptables rules contribute to the Netfilter processing overhead for packet transmission. These rules are used to redirect the traffic from the default iptables chains to the Kubernetes' user-defined iptables chains (e.g., kube-services, kube-forward, etc). Kubernetes relies on these user-defined chains along with the internal rules to support different kinds of communication. For example, the kube-proxy installs NAT rules to support 'External-to-Service' communication.

Iptables' Routing Management: Based on the network model, different paths will be exercised through the iptable chains. For intra-host communication (Layer-3 IP routing and Layer-2 bridge forwarding), we have the packet processed through the *PREROUTING*, *FORWARD*, and *POSTROUTING* chain. When a source Pod sends out a packet, the packet will be checked first by the *PREROUTING* chain. It will look up three tables (*raw*, *mangle*, *nat*) in the *PREROUTING* chain and match with the rules in these tables. So, if there are any matched rules, the corresponding actions specified will be taken on the packet (e.g., NAT, change ToS (Type of Service) field, etc). The packet will then be checked by *FORWARD* and *POSTROUTING* chain before it arrives at the veth interface of destination Pod. The intra-host routing works for both bridge forwarding and IP forwarding network models, following the same processing. The exception is for the eBPF approach, as the eBPF integrates the packet filtering functions in the eBPF program and bypasses the iptables processing by leveraging the XDP or TC hooks.

In the case of inter-host communication, the traffic can be classified into two different cases: ingress traffic (*i.e.*, inbound traffic originating from an external network and destined towards the internal host network) and egress traffic (outbound traffic originating in the internal network). For ingress traffic in overlay solution, first the packets follow the *PREROUTING* $\rightarrow$ *INPUT* path, to be routed from the host Ethernet interface to the OTEP. Then, after the packet is decapsulated by the OTEP, the packets follow the *PREROUTING* $\rightarrow$ *FORWARD* $\rightarrow$ *POSTROUTING* path, to be routed from the OTEP to the destination Pod. For egress traffic in overlay solution, the packets follow the *PREROUTING* $\rightarrow$ *FORWARD* $\rightarrow$ *POSTROUTING* chain first, when routed from the Source Pod to the OTEP. Next, the packets follow the *OUTPUT* $\rightarrow$ *POSTOUTING* route, to be routed from the OTEP to the host Ethernet interface. The configuration for the underlay solution has the *PREROUTING* $\rightarrow$ *FORWARD* $\rightarrow$ *POSTOUTING* route applied for both egress/ingress traffic.

C. Summary of Qualitative Comparison

Overall, the CNIs are plug-and-play components in Kubernetes. They provide an off-the-shelf network paradigm for the users to easily set up the networking environment. Based on Table I, we can observe that none of the existing CNIs cover all the qualitative features, e.g., Flannel's primary drawback in lack of network policy support. Each CNI has its unique feature with the comparison to others. Users can choose the best-suited CNI in regard to their qualitative needs, e.g., network model, tunneling option, etc.

As shown in Table I, Calico and Cilium use a Layer-3 based network model. Both of them support either an overlay or underlay solution. For the encapsulation options in overlay mode, Calico offers IP-in-IP and VxLAN while Cilium offers VxLAN and Geneve. Flannel, Weave, and Kube-router use a hybrid based datapath. Flannel and Kube-router support an overlay or underlay solution. Weave only supports an overlay. Flannel provides a simple network model, and users can easily set up an environment, while Weave and Cilium provide encryption support and also provide rich support for network policy and in addition Cilium and Calico are also able to support $3^{rd}party$ network policies and enable IPV6 support. And, Kube-router offers a unique DSR feature to enhance the load balancing in the Service-to-Pod mode of communication.

D. Additional Feature Considerations

Multi-interface CNIs Multi-interface CNIs are required in a number of cloud scenarios (e.g., Virtual Private Network (VPN) connectivity, multi-tenant networks), where network isolation is needed [35]. By enabling multiple network devices and multiple subnets for a single Pod, Multi-interface CNI is able to provide a better separation of the control and user planes. A typical Multi-interface CNI is Multus [35]. Multus works as an orchestrator instead of configuring the Pod networking itself. It calls the Single-interface CNIs (e.g., Flannel, Weave, *etc.*) to configure the underlying Pod networks, including routes, iptables, etc. With Multus, multiple Single-interface CNIs can coexist on the same host, and each Single-interface CNI has its own subnet, which is separated from each other. With the orchestration from Multus, we can choose the right subnet to use for each Pod based on its networking needs.

Hardware Acceleration: Tunnel Offload Most modern network interface cards (NICs) support the tunnel offload for a number of tunneling protocols, e.g., IP-in-IP, GRE, VxLAN, GENEVE, *etc.* Tunnel offload includes the following components: TCP segmentation offload (TSO, which is also performed for normal IP), checksum processing, Receive Side Scaling (RSS) selection (for multiplexing/demultiplexing), *etc.* This can accelerate packet processing and reduce the load on the CPU. However, some NICs may only support a limited set of tunneling protocols to offload. This can severely impact the performance of the CNI plugin. For example, Flannel can be configured to use IP-in-IP or VxLAN tunneling for inter-host communication. Hence, it is necessary to make the appropriate choice of the CNIs overlay networking based on the hardware supported tunneling options. We demonstrate the performance benefits achieved by the CNIs utilizing the tunnel offload capabilities in Section IV-C. We find that it is necessary for the individual container framework to leverage the NICs offload capabilities and match it with the appropriate CNI to leverage that hardware acceleration to maximize performance.

IV. Quantitative Evaluation and Analysis

In this section, we first compare the performance of different CNI plugins with a single connection between the Pods. Based on the performance results on the single connection, we breakdown the packet transmission overhead into different components and study the principal factors that influence the performance. Subsequently, we evaluate the performance with the increasing number of connections across Pods. Then, we examine the performance of different CNIs by applying a typical HTTP workload. We also assess the impact of netfilter rules and iptables chain configurations.

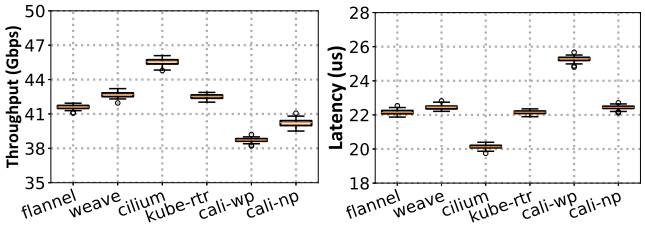
A. Experimental Setup

All the CNI plugins are evaluated on the Cloudlab testbed [36]. We build the Kubernetes cluster on two physical hosts. Each host machine has a Ten-core Intel E5-2640v4 at 2.4 GHz and 64GB memory, and 2 Dual-port Mellanox ConnectX-4 25 Gb NIC. We use Ubuntu 18.04 with kernel version 4.15.0-88-generic. Kubernetes is directly running on the physical machine, so there is no extra virtualization overhead introduced. All the Kubernetes related packages are installed with their current, latest version. We primarily use Netperf for throughput and latency measurement, with each test lasting 30 seconds and being repeated 50 times. To fully utilize the bandwidth in both intra-host and inter-host communication, the application data size[17] in Netperf is set to 4MB for the TCP throughput test, which is the TCP socket buffer limit in the OS. For latency measurement, we use Netperf's request-response (RR) mode to get the round-trip time (RTT) for TCP traffic. The packet size used in the latency measurement is 1 Byte as a default. To examine the effect of Tunnel offload support, we choose another host machine with Tunnel offload supported on its NIC. This kind of host has two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz and 192GB Memory, and one Dual-port Intel X520-DA2 10Gb NIC.

B. Intra-Host Performance

We study intra-host performance (communication within a single server host) when communicating among a number of Pods deployed within the host. This enables us to better understand and distinguish the communication overheads that arise due to the usage of the Linux bridge, iptables rules, eBPF, and the interaction with the host network stack. We evaluate Flannel, Weave, Cilium, Kube-router, and Calico variants (Calico-wp and Calico-np). 'Calico-np' helps to isolate the overheads of network policies (additional Netfilter rules).

[17]Application data size in Netperf is the "Send Message Size", the total data bytes to be transferred in a given connection, regardless of the socket buffer size, MSS or MTU settings.



(a) TCP Throughput (b) TCP Round-trip latency

Fig. 8. Intra-Host Throughput and Latency.

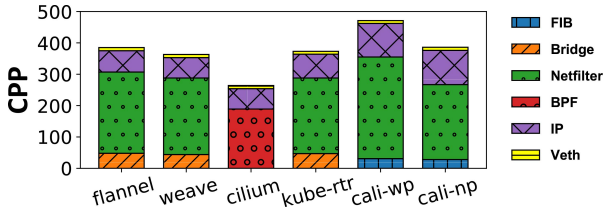


Fig. 9. Overhead Breakdown for Intra-host scenario.

1) Overall Performance: The overall throughput comparison across CNIs is shown in Fig. 8(a). For TCP throughput, Cilium with its native solution based on eBPF outperforms the other alternatives. Layer-3 routing based solutions (Calico-wp and Calico-np) perform worse than the Layer-2 based solutions. Accordingly, we also observe that Cilium achieves the lowest latency and Calico-wp has the worst round-trip latency as shown in Fig. 8(b). This is primarily due to the overheads involved in processing the Netfilter rules and Layer-3 routing, which is avoided with the eBPF based CNIs.

In order to understand how the datapath and iptables affect overall performance, we further break down the packet processing time into different components of the network stack and measure the CPU cycles as a packet goes through each component. We identify the following distinct components of the network stack: Forwarding Information Base (FIB), eBPF, Netfilter, Veth, and IP forwarding. By analyzing CPU cycles spent per packet in each component of the network stack, we establish the relationship between the achieved performance and the specific network activity involved in routing a packet with that specific CNI.

Methodology: In order to measure the *CPU cycles per packet* (*CPP*) spent in each network stack component, we first use the Linux *perf* tool [37] to count the total CPU cycles consumed in a 60-second packet transmission ($Cycle_{total}$), which is repeated 5 times. We also use *perf* to trace the function calls and measure the percentage of the overall CPU cycles spent in the corresponding function ($Cycle_{percentage}$). With the total number of packets sent in a 60-second packet transmission (N_{packet}), we can calculate the *CPP* of a specific function call as follows:

$$CPP = Cycle_{total}/N_{packet} \times Cycle_{percentage}. \quad (1)$$

2) Overhead Breakdown for Intra-Host Comm: The total *CPP* with the corresponding break down for intra-host communication is shown in Fig. 9. For the overall overhead of the complete network stack, Calico-wp (with Netfilter being the major contributor) has the highest *CPP* and Cilium the lowest.

Bridge: Flannel, Weave, and Kube-router use bridge-based solutions to forward packets in this intra-host scenario. When packets pass through the Linux bridge, the bridge-related function calls (e.g., *br_forward*()) are executed. Fig. 9 shows that the bridge overhead of Flannel, Weave, and Kube-router to be similar, at $\sim$ 45 *CPP*.

FIB & IP forwarding: We put the FIB overhead and IP forwarding overhead together as 'IP forwarding' related function calls (e.g., *ip_forward*()) are coupled with FIB function calls (e.g., *fib_table_lookup*()). When using the host IP protocol stack to forward packets, first the FIB table lookup determines the next hop. Then, the packet forwarding operation is performed. As Calico relies on the host IP protocol stack to forward packets, it incurs both FIB and IP forwarding overheads. The FIB overhead of Calico ($\sim$ 30 *CPP*) is slightly lower than the overhead using the Linux bridge ($\sim$ 45 *CPP*). But, the IP forwarding processing in Calico consumes an extra 108 *CPP*. This overhead of both FIB and IP forwarding is higher than the overhead of the bridging approaches.

eBPF: Cilium relies heavily on eBPF. Instead of bridge/IP forwarding and Linux Netfilter, it utilizes a set of eBPF hooks in the network stack to run eBPF programs to support the intra-host packet forwarding and filtering functions. Cilium attaches the eBPF programs at each veth resulting in each packet forwarding operation to incur the eBPF processing overhead. Fig. 9 shows that Cilium has the eBPF overhead of $\sim$ 189 *CPP*.

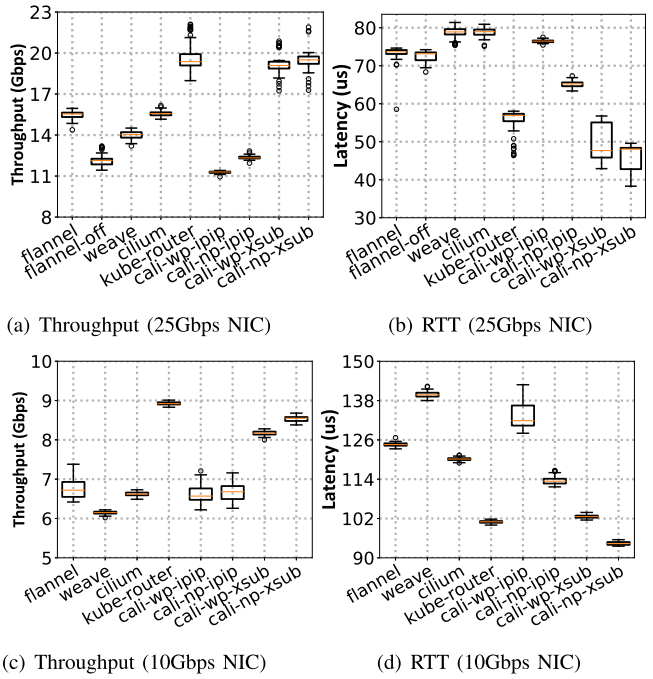
Veth: All CNI plugins spend almost the same $\sim$ 10 *CPP* (for send, receive) on veth, a small percentage of the overall overhead, with little impact on the CNIs' performance differences.

Netfilter: Calico-wp, with calico policy fully installed, consumes 324 *CPP* on Netfilter, which is 1.35 $\times$ higher than the others. Although Calico-wp suffers a significant performance penalty due to the overhead from Netfilter, it allows better network policy customization and packet filtering due to fine-grained iptables chains. Calico-wp adds user-defined chains to construct its iptables, which introduces more iptables rules as well as more iptables overhead. Note: Cilium does not have any Netfilter overhead as it uses eBPF instead of iptables.

Summary: For intra-host communication, a native routing datapath based on eBPF is much cheaper than a bridge-based datapath or native routing datapath based on IP forwarding. eBPF combines packet forwarding and filtering together, which reduces the packet forwarding overhead. Thus, Cilium achieves the highest throughput and lowest latency. Further, a fine-grained iptables chain as in Calico-wp unfortunately hurts packet transmission performance. As shown in Fig. 8, Calico-wp has lower throughput and higher latency than the others, because of the penalty from the iptables chain processing.

C. Inter-Host Performance

We use the same set of CNI plugins to communicate between two Pods on different hosts. For all inter-host experiments, Flannel, Weave, and Cilium use the VxLAN overlay



(a) Throughput (25Gbps NIC) (b) RTT (25Gbps NIC)

(c) Throughput (10Gbps NIC) (d) RTT (10Gbps NIC)

Fig. 10. Inter-Host Throughput and Latency. (a) and (b) are with Mellanox ConnectX-4 25Gb NIC. (c) and (d) are with Intel X520-DA2 10Gb NIC.

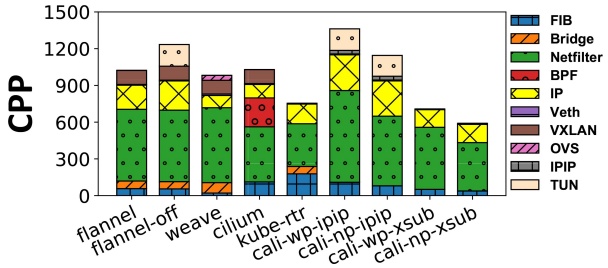


Fig. 11. Overhead Breakdown for Inter-host Scenario: CPU overhead from both sender and receiver hosts.

mode. Kube-router uses native IP routing (underlay), while Calico (wp or np options) can support either native IP routing (underlay) or IP-in-IP overlay. Accordingly for Calico, we study all four modes: Calico-np-ipip, Calico-wp-ipip, Calico-np-xsub and Calico-wp-xsub. We noted that the offloading of IP-in-IP tunnel is not supported by the Mellanox ConnectX-4 25 Gbps NIC used in our testbed. Hence, to study the impact of tunnel offload in the NIC, we additionally experimented with Flannel by explicitly disabling the tunnel offload feature for VxLAN tunneling ('flannel-off' mode). We also study the inter-host communication performance on another Intel X520-DA2 10Gb NIC where IP-in-IP tunnel offload is supported.

1) Overall Performance: TCP throughput and latency results are shown in Fig. 10. The native routing solutions (Kube-router, Calico-wp-xsub and Calico-np-xsub) perform better than the overlay solutions (Flannel, Flannel-off, Weave, Cilium, Calico-wp-ipip and Calico-np-ipip). However, without the tunnel offload in the Mellanox ConnectX-4 25Gb NIC, Flannel-off and Calico in IP-in-IP overlays perform poorly, with much lower throughput than Cilium and Calico with native routing (xsub) options. Moreover, for the same datapath, solutions with network policy disabled (Calico-np-xsub and Calico-np-ipip) perform $0.5 \sim 1.5$ *Gbps* better than when network policy enabled (Calico-wp-xsub and Calico-wp-ipip). Also, TCP round-trip latencies show a corresponding increase for those CNIs that see lower throughput. For comparison purposes, we also show the results with Pods on two hosts communicating across a 10 Gbps link, using Intel X520-DA2 10Gb NICs that support both VxLAN and IP-in-IP tunnel offload. We see that the performance of Flannel and Calico IP-in-IP CNIs are comparable. In fact, because the CPU utilization is reduced due to the tunnel offload, the Calico CNI performance 'with policies' and 'no policy' see comparable throughput (however the latency is higher with policies). To summarize, i) Tunnel offload is necessary to improve the CNI performance, and where the CNIs can support multiple overlay solutions (UDP, VxLAN, IP-in-IP), it is desirable to choose the overlay mode that can be supported via tunnel offload. ii) the underlay-based CNIs (Calico xsub and Kube-router) perform better than the overlay options, despite the NIC's offloading of the tunnel-related operations.

2) Overhead Breakdown for Inter-Host Comm.: For the overhead analysis, we include the VxLAN tunnel, IP-in-IP tunnel, OVS-datapath, and in-kernel tunnel offload processing components. Again, we use the same methodology to calculate the *CPP* of each function call on both the sender and receiver host. The total *CPP*, along with the breakdown is shown in Fig. 11. The native routing solutions (Kube-router, Calico-wp-xsub and Calico-np-xsub) have lower *CPP* compared to the overlay solutions (Flannel, Flannel-off, Weave, Cilium, Calico-wp-ipip and Calico-np-ipip). Also, the solutions with simple iptables have lower *CPP* than the complex iptables chains.

Layer-2 Bridging: Flannel, Weave, and Kube-router use the Linux bridge to forward packets between different virtual ethernet interfaces (veths) via Layer-2 bridging and incur similar overheads as in the intra-host scenario. However, as Weave uses an extra veth-pair to connect the Linux-bridge with overlay tunnel, it incurs 2 $\times$ more *CPP* than Flannel and Kube-router. With Weave, when the packets are forwarded from the veth to the overlay tunnel (sender side), they do not go through the host IP protocol stack but instead rely on the bridge-related function calls (i.e., *br_forward*()). These bridge-related function calls are invoked again when packets are forwarded from the overlay tunnel to the veth (receiver side). In contrast, Flannel and Kube-router use the host IP protocol stack to forward packets from the veth to the overlay tunnel, thus avoiding any bridge forwarding overhead on the sender side. On the receiving side, when forwarding packets from the overlay tunnel to the veth, they rely on Layer-2 bridge forwarding. Thus, the total bridge overhead of Flannel and Kube-router is half of that of Weave.

IP forwarding: Weave and Cilium traverse the host IP stack once per packet transmission and have similar ($100 \sim 110$ *CPP*) overhead. The IP protocol stack operations

of Weave and Cilium are performed between the VxLAN tunnel and host Ethernet interface. However, with Flannel, the host IP stack operations are performed twice per packet transmission. First between Linux bridge and VxLAN tunnel, and then again between the VxLAN tunnel and host Ethernet interface. Thus, we correspondingly incur about 197 *CPP*. Calico-*-ipip (Note: "*" means both Calico-wp-ipip and Calico-np-ipip) also performs host IP stack operations twice per packet transmission. First between veth and IP-in-IP tunnel, and then again between the IP-in-IP tunnel and host Ethernet interface. However, in this case, additional IP protocol stack processing overhead in the CPU occurs due to the lack of the IP-in-IP tunnel offload support in the NIC we used. This in-kernel tunnel processing (e.g., *ip_send_check*() function call) expends more CPU cycles. Calico-*-ipip consumes $\sim$ 290 *CPP* in the IP protocol stack. The underlay solutions (*i.e.*, Kube-router and Calico-*-xsub) consume $\sim$ 150 *CPP* in IP protocol stack.

Netfilter: CNI's Network Policy is a useful feature that leverages Netfilter rules to enhance network security. However, Netfilter is a major source of overhead as shown in Fig. 11, requiring a larger amount of *CPP* than the other components. Large and complex iptables chains incur higher processing overheads. Fig. 11 shows that Cali-wp-ipip has the highest Netfilter overhead compared to the other solutions due to its large iptables size, while the Kube-router and Cali-np-xsub have the lowest. Cilium has the least Netfilter overhead compared to the other overlay-based solutions, as it bypasses the *PREROUTING* $\rightarrow$ *FORWARD* $\rightarrow$ *POSTROUTING* route of Fig. 7 and uses eBPF instead.

eBPF: In the case of Cilium, we observe that the eBPF overhead for the inter-host packet forwarding is relatively higher (236 *CPP*) than the intra-host packet forwarding (189 *CPP*). This is due to an additional hook point at the host to support overlay mode and process the VxLAN tunneling.

Veth: As with the intra-host case, the veth processing overhead is the least compared to the other components and is similar for most of the CNIs ($\sim$ 6 *CPP*). As Weave has 4 veth-pairs on the inter-host datapath as opposed to 2 for the other CNIs, it incurs twice the overhead ($\sim$ 12 *CPP*).

Overlay: Using an overlay incurs packet encapsulation and decapsulation overheads. Flannel, Weave, and Cilium use VxLAN overlay and Calico-*-ipip use an IP-in-IP overlay. Overhead from the VxLAN overlay is $\sim$ 114 *CPP*, while IP-in-IP incurs somewhat lower overhead ($\sim$ 30 *CPP*). Weave uses the OVS-datapath to implement the overlay processing and incurs some extra overhead ($\sim$ 40 *CPP*) on OVS-related function calls (Fig. 11). Thus, the overlay accounts for 3% $\sim$ 11% of the total overhead, depending on the packet encapsulation, potentially having a significant performance impact.

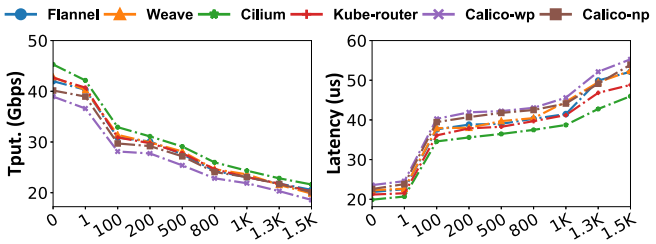
Tunnel Offload: The Mellanox ConnectX-4 NIC in our testbed machines support TSO and VxLAN offload, but not IP-in-IP tunneling.[18] Thus, all the IP-in-IP tunnel offload processing needs to be performed in the kernel, thus increasing the CPU burden. The in-kernel tunnel processing costs $\sim$ 170 *CPP* ("TUN" in Fig. 11) for Flannel-off and Calico-*-ipip. As shown in Fig. 10, the Calico-wp-ipip and Calico-np-ipip only achieve 11.1 *Gbps* and 12.5 *Gbps* TCP throughput for inter-host Pod communication respectively, which is much slower than the VxLAN overlay (14 $\sim$ 15.5 *Gbps*) and Layer-3 routing (18 $\sim$ 19.7 *Gbps*). Flannel-off also shows the same significant performance reduction when tunnel offload is disabled on the NIC, confirming the extra processing overhead introduced in the kernel stack when the VxLAN processing is done in the CPU. In Netperf's request-response mode, the size of the payload is only one byte. The tunnel offload processing becomes important only when the size of the packet (including the tunnel header length) exceeds the Maximum Segment Size (MSS). So, with the small payload size, we see equivalent packet forwarding performance with/without hardware tunnel offload support. As shown in Fig. 10 (b), the Calico-wp-ipip has a similar RTT latency compared to some of the VxLAN overlay CNIs (e.g., Weave, Cilium), while Calico-np-ipip achieves lower latency than the VxLAN overlay CNIs. Moreover, the Flannel and Flannel-off have similar RTT latency, indicating that the effect of hardware tunnel offload support for small packets is not significant.

Summary: Connecting the overlay tunnel and bridge via an extra veth-pair (as in Weave) may reduce the FIB and IP forwarding overhead, but increases the bridge and veth overhead. A powerful network policy mechanism can provide fine-grained packet filtering, allowing for improved security for packet transmission. However, more Netfilter calls result in lower packet forwarding performance. Users should carefully consider their needs for additional packet filtering rules and seek to manage the growth of iptables size as much as possible while meeting security requirements. Generally, a native routing datapath is cheaper than an overlay-based datapath. Removing unnecessary iptables chains and rules can help reduce Netfilter overhead. When used with the underlay network model, the Netfilter overhead of calico-wp-xsub can be kept around 507 *CPP*, which is lower than with the other overlay network models. Enabling a CNI's network policy support with native IP routing can achieve a good balance between the overhead and network security support. A NIC's offload capability is another important aspect that needs to be seriously considered when choosing CNI. Having the kernel and CPU perform the tunneling tasks can have a measurable impact on performance.

D. Performance for Larger-Scale Configurations

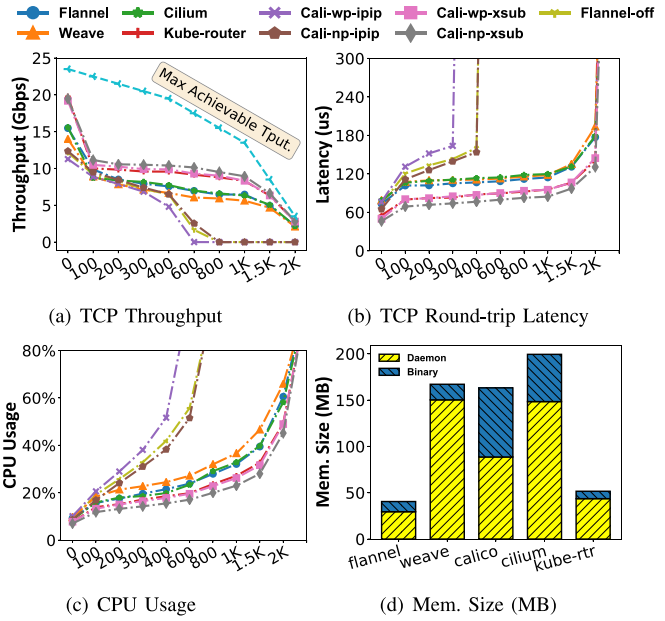
1) Experimental Setup: As the amount of communications between Pods increases, the individual conversation throughput will reduce, not only because of sharing resources among contending connections, but also interference and increasing overhead due to contention. To evaluate the performance variation with the scaling of the number of concurrent connections across Pods, we set up an increasing number of TCP connections between Pods as background traffic. The bandwidth of each of the background TCP connections was limited to

[18]We verified using "ethtool" for the supported offload tunnel functions on the NIC. Also, refer to: (https://community.mellanox.com/s/article/mellanox-adapters—comparison-table).



(a) TCP Throughput (b) TCP Round-trip Latency

Fig. 12. Intra-Host Performance with increasing # connections.



(a) TCP Throughput (b) TCP Round-trip Latency

(c) CPU Usage (d) Mem. Size (MB)

Fig. 13. Inter-Host Performance with increasing # background connections; CPU and memory overheads.

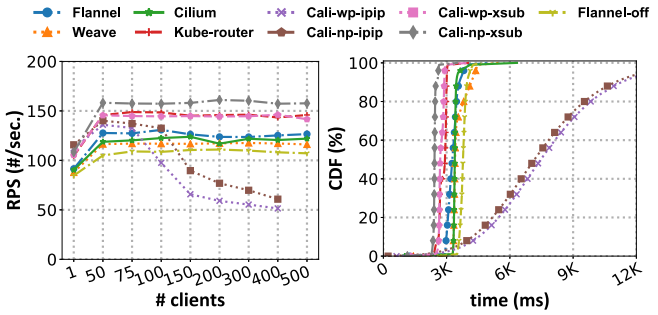
10Mbps for the inter-host case. As the intra-host has much higher achievable bandwidth, we set the bandwidth of each of the background TCP connections as 50Mbps for the intra-host case. We use *iperf3* to generate the background TCP connection traffic [38]. The test connection that we monitor generates traffic using Netperf in a distinct Pod communicating with a peer. Kubernetes has a maximum of 100 Pods per host [21]. We have different levels on the number of background TCP connections in our experiments. For the intra-host case, we deploy 49 *iperf3* Pods each as the server and the client end on a single host. For the inter-host traffic, we deploy 99 *iperf3* Pods as servers on one host and deploy 99 *iperf3* Pods as clients on another host, each with 1 TCP connection. Two Pods are deployed for Netperf test for both scenarios. To generate background connections more than the number of *iperf3* Pods, we leverage the "simultaneous connections" option in *iperf3*.

2) Intra-Host Performance With Background Traffic: Fig. 12 shows the intra-host performance comparison with different amounts of background traffic. The total background traffic with all 1.5K TCP connections (generating 50 Mbps each) is less than 75 Gbps. Cilium outperforms the other CNI plugins because of its low overhead in the datapath and in Netfilter. Calico-wp has the worst performance throughout, due to its large overhead from Netfilter rules. The TCP round-trip time is also better for Cilium. Further, we observe that even with just one active background connection, the test connection suffers about a $\sim$ 2 *Gbps* throughput reduction. This drop is consistent across all the CNIs and happens as soon as two processes are involved in sending and receiving the packets on the same host. We speculate it to be likely a result of resource contention especially the locks.

3) Inter-Host Performance With Background Traffic: Fig. 13 shows the inter-host communication performance (throughput and latency for a test connection) of different CNI plugins with varying amounts of background traffic, generated by an increasing number of background connections. The background connections not only consume bandwidth of the NIC and link, but also use up the host's CPU. However, the throughput reduction is more in line with the amount of added traffic from the background connections, except for the cases when the tunnel offloading is disabled, at higher loads. The difference among CNIs is noticeable in that the native routing solutions (Kube-router and Calico-xsub) outperform the overlay based solutions, because of less overhead on the datapath and Netfilter. In contrast, Calico in IP-in-IP mode performs worse than the others, because of the additional overhead since the NIC does not offload this function. This degraded performance is also observed with Flannel, when the tunnel-offload is turned off (Flannel-off). All of these see a precipitous drop in throughput beyond 400 background connections (each generating 10 Mbps) because the CPU is overloaded and the latency is correspondingly higher.

4) CPU Utilization and Memory Footprint: We also analyze the CPU utilization and memory footprint with different CNIs corresponding to these inter-host communication traffic experiments. Fig. 13(c), shows the CPU usage with the increasing workload. Native routing ('Calico-*-xsub', 'Kube-router') incurs relatively low CPU overhead, while the overlay mode CNIs (e.g., Flannel, Weave, etc.) have a much higher CPU load. Further, CNIs with tunnel offload disabled are overloaded with fewer background connections, due to the extra offload processing in the kernel, which inevitably increases the CPU overhead. We also assess the memory footprint incurred by different CNIs in Fig 13(d). We observe that Flannel and Kube-router have a low memory footprint (40 $\sim$ 50 *MB*), while Weave, Cilium, and Calico have a very high memory footprint (160 $\sim$ 200 *MB*). We profiled the memory usage of CNIs 'daemonset' and 'binaries' from the kernel's pseudo-filesystem (procfs). We find the memory usage is independent of the number of Pods/connections. The 'daemonset' is a running process with fixed memory size. And the CNI 'binary' file is an executable file, also with a fixed size.

Summary: In general, increasing the amount of background connections impacts performance as they consume both the CPU resource and bandwidth. CNIs using native routing can achieve better performance compared to those using an overlay. Moreover, overlay CNIs without tunnel offload support



(a) RPS, increasing concurrency (b) CDF Latency for concurrency 400

Fig. 14. HTTP Performance for inter-host communication.

TABLE II
HTTP PERFORMANCE OF CNIS FOR 1 & 400 CLIENTS

CNI Metrics	RPS		Avg.Latency (msec.)		Tput (Gbps)	
	c = 1	*c = 400*	*c = 1*	*c = 400*	*c = 1*	*c = 400*
Flannel	91.50	125.23	10.930	3168.911	3.57	4.90
Flannel-off	84.66	108.15	11.812	3521.380	3.3	4.22
Weave	87.81	117.08	11.388	3362.784	3.43	4.57
Cilium	90.35	120.57	11.069	3217.960	3.52	4.70
Kube-rtr	104.50	143.38	9.569	2829.294	4.08	5.60
Cali-wp-ipip	112.20	51.46	7.987	7481.709	4.38	2.01
Cali-np-ipip	115.74	60.80	7.367	6629.865	4.52	2.37
Cali-wp-xsub	105.36	146.08	9.491	2738.195	4.11	5.70
Cali-np-xsub	108.31	157.28	9.233	2434.811	4.23	6.14

in the NIC are impacted more, due to the increased in-kernel processing overhead.

E. Impact of CNI on Typical HTTP Workload

1) Experimental Setup: We use the Apache HTTP server benchmarking tool (*ab* [39]) to generate the HTTP workload. We emulate multiple concurrent clients and choose the *nginx* [40] as the HTTP server. We primarily study the CNI behavior in the inter-host scenario, deploying an HTTP server Pod and an '*ab*' client Pod on two different worker hosts. To get statistically reasonable results, we generate a total of 500K requests for each HTTP test. The size of the response payload for each request is fixed, at 5 MBytes. For each CNI, we also vary the level of concurrency (*i.e.*, number of clients sending HTTP requests simultaneously from 1, 50, 75, up to 500) to generate increasing amounts of HTTP traffic.

2) HTTP Performance Results: Table II shows the performance with different CNIs. We can observe that Calico (both overlay and underlay modes) outperform the rest of the CNIs in a *single* connection case. However, with increasing numbers of connections ($c = 400$), Calico-*-xsub (underlay mode) works the best, while the Calico-*-ipip (overlay mode) turns to be the worst. This degradation in HTTP throughput is primarily due to the lack of tunnel offload support at the NIC, which results in high CPU overhead and thus adversely impacts the HTTP throughput and latency. Fig. 14 (a) shows the impact on RPS with different CNIs for the increasing number of concurrent connections and Fig. 14 (b) shows the latency profile with 400 concurrent connections. We can clearly observe that with the increasing number of concurrent connections (from 100 to 400) the performance of Calico-wp-ipip and Calico-np-ipip starts to degrade. We observed this to be due to additional CPU overhead generated by the IP-in-IP tunnel processing in the kernel. In fact, we noticed that the HTTP server often times out on the Calico-wp-ipip and Calico-np-ipip at a concurrency level of 500 (hence the numbers are not reported). The results indicate the overload behavior may be the cause of the poor performance for the Calico overlay CNIs [41]. From Table II and Fig. 14(b), we also observe that Calico-wp-ipip and Calico-np-ipip exhibit much higher average and tail latency than the other CNIs, for the case of 400 clients, suggesting much higher resource utilization on the server Pods. Apart from Calico-*-xsub, we also observe that the underlay mode (Kube-router) performs somewhat better across all the cases when compared to the overlay modes (Flannel, Weave, and Cilium).

Summary: Our realistic HTTP workload tests indicate there is a measurable impact of the choice of the CNI. In general, a 'Layer-3 + Underlay' CNI (e.g., Calico, native routing) appears better suited for most HTTP traffic, especially at large scale (higher concurrency) traffic patterns.

TABLE III
NUMBER OF IPTABLES CHAINS AND RULES WITH DEFAULT CNI CONFIGURATIONS. INTER-HOST CASE COLLECTS FROM BOTH THE SOURCE HOST AND DESTINATION HOST

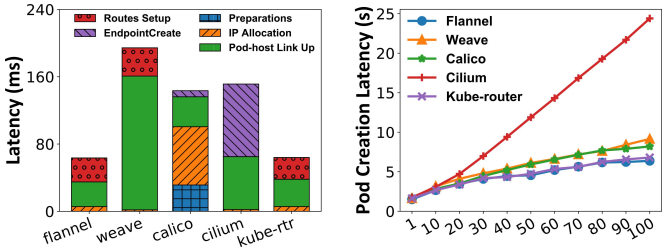
CNI Solutions	Intra-Host		Inter-Host	
	# of chains	*# of Rules*	*# of Chains*	*# of Rules*
Flannel	3	5	10	21
Weave	3	5	10	21
Cilium	0	0	4	20
Kube-router	3	5	6	10
Calico-wp-ipip	3	17	10	46
Calico-np-ipip	3	5	10	22
Calico-wp-xsub	3	17	6	28
Calico-np-xsub	3	5	6	12

F. Iptables Evaluation

In order to study the impact of iptable rules (e.g., number of rules, etc) by different CNIs, we use the 'iptables-save' command to profile the iptable configurations set up by the CNIs. We track the chains and rules that a packet will traverse going from the source to destination Pod. Table III presents the number of iptables chains and rules for different CNIs for the intra-/inter-host communication patterns respectively.

In the intra-host case, Cilium use eBPF for packet forwarding, which bypasses the iptables processing. Flannel, Weave, Kube-router, and Calico-np (both 'Calico-np-ipip' and 'Calico-np-xsub' in the intra-host case, as they are equivalent) have the same number of iptables chains and rules, and they all exhibit similar netfiler overhead ($\sim$ 245 *CPP*) for the intra-host case. Calico-wp (both 'Calico-wp-ipip' and 'Calico-wp-xsub') is configured with 17 iptables rules by default. This adds to a higher netfilter overhead (324 *CPP*) compared to the others.

In the inter-host case, Calico with network policy enabled (e.g., 'Calico-wp-ipip') has more iptables rules configured resulting in higher overhead (749 *CPP*). Moreover, with a similar number of rules, a CNI with fewer iptables chains applied (e.g., Cilium) has much less overhead (450 *CPP*) than the CNIs with more iptables chains applied (e.g., Flannel, Weave).



(a) Network Startup Latency Breakdown

(b) Pod Creation Latency: single host; increasing # Pods

Fig. 15. Pod Creation Latency with different CNIs.

We observed that CNIs using the underlay model have fewer iptables rules than the CNIs using the overlay model, as the overlay requires additional mangle/NAT rules to perform the necessary encapsulation/decapsulation. We also observed that the number of iptables chains and rules per packet does not increase with more background connections enabled.

Summary: Based on the iptables evaluation, we conclude that CNIs with fewer iptables chains and rules will have relatively less Netfilter overhead. However, to assure Pod network security, users needs to use the CNI's Network Policy API to install iptables rules, which could increase Netfilter overhead and lead to performance loss.

G. Pod Creation Time Analysis

Starting a Pod from scratch can take considerable time, and significantly impacts cloud-based microservices and Function as a Service offerings. The Pod creation latency is the time for starting a Pod from scratch until all the containers in the Pod have been created, comprising multiple steps [42], including network startup. We launch a single Pod each time and measure the network startup latency breakdown with different CNI plugins. We also measure the latency as we deploy an increasing number of Pods on a single host, with 10 repetitions. The container image used in our experiment is the 'pause container' [43], which just sleeps after being successfully started. The size of the pause container is 600KB.

The individual Pod creation latency is in the range of $1.48 \sim 1.63s$ and the networking component contributes about $60 \sim 195$ *ms* (Fig. 15 (a)). Please refer to [25] for more details on Pod startup, and a definition and breakdown of the individual steps. Flannel and Kube-router have a smaller network startup latency (~ 60 *ms*) compared to the other alternatives. Weave consumes about 165*ms* in the *Pod-host Link Up* step, due to the work of appending multicast rule in iptables. Calico spends ~ 80 *ms* in the *IP Allocation* step, which is primarily due to the interaction with the etcd store. The time spent by Cilium in the *Endpoint Creation* step accounts for ~ 90 *ms*. During this step, Cilium generates the eBPF code and links it into the kernel, which contributes to this high latency. Fig. 15 (b) shows the Pod creation latency for simultaneously starting a number of distinct Pods on the same host. Flannel and Kube-router have the smallest amount of increase in the creation latency as more Pods deployed together, while the latency with Cilium increases much more rapidly, which shows poor scalability.

V. Characteristics for an Ideal CNI

Based on our qualitative and quantitative analysis, we outline a design for an ideal CNI plugin and its characteristics. Our design choice for ideal CNI is motivated by the need to have very low overhead, low latency, and high throughput intra-host and inter-host container communication while also able to facilitate rich security and network policy support.

1) Based on our evaluation results in Section IV-B, we propose that an ideal CNI should seek to utilize the eBPF approach for *intra-host* communication. This is primarily because it generates the least amount of CPU overhead compared to the other solutions. We attach eBPF programs (with packet forwarding functionality) at the veth of Pods, so the intra-host packet forwarding can achieve better performance.
2) For packet forwarding across hosts, we propose the ideal CNI use native (IP) routing (based on the results in Section IV-C). This helps avoid packet encapsulation/decapsulation overheads, avoids unnecessary fragmentation due to large MTUs, and results in fewer iptable-chains to process. This can achieve the highest packet forwarding performance when crossing the host boundary. The daemon of the ideal CNI needs to be able to configure BGP between nodes to distribute routing information, which is necessary to support native routing across hosts. Moreover, the host's physical interface needs to be attached with an eBPF program, so that we can leverage the benefits brought by eBPF, just as in the intra-host case. Multicast support can be built by leveraging eBPF's TC hooks, which can be a good match with the ideal CNI's intra-/inter-host datapath.
3) When the underlying networking infrastructure does not provide support for native routing (such as BGP) or the users have a strong demand for network isolation, the ideal CNI should be able to offer sufficient overlay tunneling options to users (e.g., IP-in-IP, VXLAN, GRE, *etc.*). Support for several overlay modes is desirable, especially if different NICs in the cluster lack the support for offloading certain specific overlay tunneling modes. A practical design should be able to auto-configure and suggest the right overlay option for the user, i.e., the CNI daemon should be able to detect the tunnel offload support information provided by NIC for all the nodes in the Kubernetes cluster (interact with 'ethtool'), and help users to make the right decision on choosing the overlay tunnel that can exploit the offloading capabilities, so as to achieve maximum performance.
4) The ideal CNI should support the network policies that can be applied across all layers, *i.e.*, Layer 3 – Layer 7. This will provide a rich set of network policy attributes to provide needed network security. It is also desirable to have an eBPF-based iptables implementation [34], which could cooperate with the eBPF-based datapath design and achieve better packet filtering/forwarding performance.

VI. Related Work

There are a number of blog articles, as well as notes on Github repositories, that provide a good description of different CNIs [13], [44]–[46]. Moreover, several works [14]–[19] have compared and evaluated the performance of different CNI plugins. Suo *et al.* [14] study different container network models and evaluate them across different aspects, such as the TCP/UDP throughput, latency, scalability, virtualization overhead, CPU utilization, and launch time of container networks. While the work attributes the performance differences observed across different CNIs to their different datapaths, it fails to identify the root causes behind these differences, as we have done here. As we observe, the main overhead is from how the CNI plugins interact with the network stack, which heretofore has not been adequately examined.

Kapočius [15] evaluates the performance of Kubernetes CNI plugins on both the virtual machines and bare metal. Kapočius [18] also evaluates several popular CNIs with different factors considered, e.g., MTU, the number of aggregated network interfaces, and NIC offloading conditions. Their results present the performance variation (at a high level) with different aggregated interfaces and NIC offloading configurations. However, both the papers do not analyze the performance differences, or provide an in-depth analysis of the kernel and namespace overheads observed for different CNIs. Bankston and Guo [16] compare the performance of CNI provided by different public cloud providers (e.g., AWS, Azure, and GCP) with different instances. They also evaluate the impact of encryption and MTU on performance. Their work provides limited insight into the different open-source CNIs. Park *et al.* [17] specifically compare the performance of Flannel network, OVS-based network, and native-VLAN network, but again, only at a high level. Ducastel [13] evaluates the most popular CNI plugins using several benchmarks. It also provides a qualitative comparison on security and resource consumption, but is limited to the inter-host case, providing a high-level, throughput-only comparison. Zeng *et al.* [19] study three container network solutions (Calico, Flannel, and Docker Swarm Overlay) and compare their performance based on TCP/UDP throughput and ping delay. However, the performance difference between different solutions are not explained in detail.

Generally, all these existing works fail to provide a kernel-level analysis and comparison for CNI plugins. This article offers an objective comparison across all of the CNIs, from both a qualitative and quantitative perspective, which is important for guiding users as well as the future development of a scalable, high-performance CNI.

VII. Conclusion

Through qualitative analysis and a careful measurement-driven evaluation, we provide an in-depth understanding of the different CNI plugins, identify their key design considerations and associated performance. Our evaluation results show the interactions between the different datapath (organization of iptables), usage of the host network stack contribute to the overall performance.

While there is no single universally 'best' CNI plugin, there is a clear choice depending on the need for intra-host or inter-host Pod-to-Pod communication. For the intra-host case, Cilium appears best, with eBPF optimized for routing within a host. For the inter-host case, Kube-router and Calico are better due to the lighter-weight IP routing mode compared to their overlay counterparts. Although Netfilter rules incur overhead, their rich, fine-grained network policy and customization can enhance cluster security. Tunnel offload is another aspect to be considered, which can help to achieve the maximum performance when working with a CNI's overlay mode. This may be very desirable for Cloud Service Providers.

Our work sheds light on the benefits and overheads of the different aspects of CNIs, thus informing us of the design of an ideal CNI for Kubernetes cluster environments. The ideal CNI supports several desirable features including the eBPF-based intra-host datapath, native routing for inter-host packet forwarding, support for sufficient overlay tunneling options, automatic tunnel offload support detection, feature-rich network policy support coupled with an eBPF-based iptables implementation.

Acknowledgment

The authors thank all the anonymous reviewers for their valuable feedback.

References

[1] D. Bernstein, "Containers and cloud: From LXC to docker to kubernetes," *IEEE Cloud Comput.*, vol. 1, no. 3, pp. 81–84, Sep. 2014.
[2] *PODS*. Accessed: Oct. 20, 2020. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/
[3] S. Qi, S. G. Kulkarni, and K. K. Ramakrishnan, "Understanding container network interface plugins: Design considerations and performance," in *Proc. IEEE Int. Symp. Local Metropolitan Area Netw. (LANMAN)*, Orlando, FL, USA, 2020, pp. 1–6.
[4] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, San Francisco, CA, USA, 2018, pp. 970–973.
[5] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Atlanta, GA, USA, 2017, pp. 2658–2659.
[6] *CNI—The Container Network Interface*. Accessed: Jun. 1, 2020. [Online]. Available: https://github.com/containernetworking/cni
[7] (2020). *Kubernetes Components*. Accessed: Oct. 20, 2020. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/
[8] *Flannel*. Accessed: Oct. 20, 2020. [Online]. Available: https://github.com/coreos/flannel/
[9] *Weave*. Accessed: Oct. 20, 2020. [Online]. Available: https://github.com/weaveworks/weave
[10] *Cilium*. Accessed: Oct. 20, 2020. [Online]. Available: https://cilium.io/
[11] *Calico*. Accessed: Oct. 20, 2020. [Online]. Available: https://github.com/projectcalico/calico
[12] *Kube-Router*. Accessed: Oct. 20, 2020. [Online]. Available: https://www.kube-router.io/
[13] A. Ducastel. (2019). *Benchmark Results of Kubernetes Network Plugins (CNI) Over 10Gbit/s Network*. Accessed: Oct. 20, 2020. [Online]. Available: https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4
[14] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, Honolulu, HI, USA, 2018, pp. 189–197.

[15] N. Kapočius, "Overview of kubernetes CNI plugins performance," *Mokslas Lietuvos Ateitis Sci. Future Lithuania*, vol. 12, Feb. 2020. [Online]. Available: https://doi.org/10.3846/mla.2020.11454
[16] R. Bankston and J. Guo, "Performance of container network technologies in cloud environments," in *Proc. IEEE Int. Conf. Electro/Inf. Technol. (EIT)*, Rochester, MI, USA, 2018, pp. 0277–0283.
[17] Y. Park, H. Yang, and Y. Kim, "Performance analysis of CNI (container networking interface) based container network," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Jeju, South Korea, 2018, pp. 248–250.
[18] N. Kapočius, "Performance studies of kubernetes network solutions," in *Proc. IEEE Open Conf. Elect. Electron. Inf. Sci. (eStream)*, Vilnius, Lithuania, 2020, pp. 1–6.
[19] H. Zeng, B. Wang, W. Deng, and W. Zhang, "Measurement and evaluation for docker container networking," in *Proc. Int. Conf. Cyber-Enabled Distrib. Comput. Knowl. Discov. (CyberC)*, Nanjing, China, 2017, pp. 105–108.
[20] (2016). *Autoscaling in Kubernetes*. Accessed: May 20, 2020. [Online]. Available: https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/
[21] (2020). *Building Large Clusters*. Accessed: May 20, 2020. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/
[22] *Romana*. Accessed: May 20, 2020. [Online]. Available: https://github.com/romana/romana
[23] *Canal*. Accessed: May 20, 2020. [Online]. Available: https://github.com/projectcalico/canal
[24] *Contiv-VPP*. Accessed: May 20, 2020. [Online]. Available: https://github.com/contiv/vpp
[25] S. Qi, S. G. Kulkarni, and K. Ramakrishnan, "Assessing container network interface plugins: Functionality, performance, and scalability," Dept. Comput. Sci. Eng., UC Riverside, Riverside, CA, USA, UCR CSE Networking Group Rep. Net-2020-1221, 2020. [Online]. Available: https://www.cs.ucr.edu/ sqi009/Net-2020-1221.pdf
[26] (2020). *Building Sustainable Ecosystems for Cloud Native Software*. Accessed: May 20, 2020. [Online]. Available: https://www.cncf.io/
[27] H. Sahni. (2017). *The Container Networking Landscape: CNI From CoreOS and CNM from Docker*. Accessed: Jun. 4, 2020. [Online]. Available: https://www.nuagenetworks.net/blog/container-networking-standards/
[28] *The Container Network Model*. Accessed: Jun. 4, 2020. [Online]. Available: https://github.com/moby/libnetwork/
[29] L. Calcote. (2016). *The Container Networking Landscape: CNI From CoreOS and CNM from Docker*. Accessed: Oct. 20, 2020. [Online]. Available: https://thenewstack.io/container-networking-landscape-cni-coreos-cnm-docker/
[30] *Cluster Networking*. Accessed: May 29, 2020. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/
[31] K. Butler, T. R. Farley, P. McDaniel, and J. Rexford, "A survey of bgp security issues and solutions," *Proc. IEEE*, vol. 98, no. 1, pp. 100–122, Jan. 2010.
[32] P. Sermpezis *et al.*, "ARTEMIS: Neutralizing BGP hijacking within a minute," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, pp. 2471–2486, Dec. 2018.
[33] K. Sriram, D. Montgomery, B. Dickson, K. Patel, and A. Robachevsky, "Methods for detection and mitigation of BGP route leaks, draft-ietf-idr-route-leak-detection-mitigation-06," IETF, Internet-Draft, 2017.
[34] M. Bertrone, S. Miano, F. Risso, and M. Tumolo, "Accelerating linux security with eBPF iptables," in *Proc. ACM SIGCOMM Conf. Posters Demos*, 2018, pp. 108–110.
[35] *multus-CNI*, Intel, Santa Clara, CA, USA. Accessed: Oct. 20, 2020. [Online]. Available: https://github.com/intel/multus-cni/
[36] R. Ricci, E. Eide, and C. Team, "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications," *Login Mag. USENIX SAGE*, vol. 39, no. 6, pp. 36–38, 2014.
[37] A. C. De Melo, "The new linux 'perf' tools," in *Proc. Slides Linux Kongress*, vol. 18, 2010, pp. 1–42.
[38] S. Qi. *iperf3 Container*. Accessed: May 26, 2020. [Online]. Available: https://hub.docker.com/repository/docker/shixiongqi/iperf3
[39] *AB—Apache HTTP Server Benchmarking Tool*. Accessed: May 29, 2020. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html
[40] *Nginx*. Accessed: May 29, 2020. [Online]. Available: https://nginx.org/
[41] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 217–252, 1997.
[42] H. Shah. (2017). *Kubernetes: Lifecycle of a Pod*. Accessed: May 29, 2020. [Online]. Available: https://dzone.com/articles/kubernetes-lifecycle-of-a-pod
[43] *Pause Container*, Google Inc., Mountain View, CA, USA. Accessed: Oct. 20, 2020. [Online]. Available: https://hub.docker.com/r/google/pause/
[44] J. Ellingwood. (2019). *Comparing Kubernetes CNI Providers: Flannel, Calico, Canal, and Weav*. Accessed: May 29, 2020. [Online]. Available: https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/
[45] G. Tobais. (2018). *How Kubernetes Networking Works—Under the Hood*. Accessed: Jun. 1, 2020. [Online]. Available: https://neuvector.com/network-security/advanced-kubernetes-networking/
[46] (2017). *Choosing a CNI Network Provider for Kubernetes*. Accessed: Jun. 5, 2020. [Online]. Available: https://chrislovecnm.com/kubernetes/cni/choosing-a-cni-provider/

Shixiong Qi received the B.Sc. degree in electronic and information engineering from the Nanjing University of Posts and Telecommunications, China, in 2015, and the M.Sc. degree in communication and information systems from Xidian University, China, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of California at Riverside, Riverside. His current research interests focus on cloud computing and network function virtualization.

Sameer G. Kulkarni received the Ph.D. degree from the University of Göttingen, Germany. He worked as a Postdoctoral Researcher with the University of California at Riverside, Riverside. He is an Assistant Professor with the Department of Computer Science and Engineering, Indian Institute of Technology Gandhinagar, Gandhinagar. His current research interests include parallel and distributed computing, software defined networks, network function virtualization, and cloud computing. His Ph.D. thesis received the IEEE Technical Committee on Scalable Computing Outstanding Dissertation Award in 2019.

K. K. Ramakrishnan (Fellow, IEEE) received the M.Tech. degree from the Indian Institute of Science in 1978, and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, MD, USA, in 1981 and 1983, respectively. He was a Distinguished Member of Technical Staff with AT&T Labs-Research. He is a Professor of Computer Science and Engineering with the University of California at Riverside, Riverside. Prior to 1994, he was a Technical Director and a Consulting Engineer of Networking with Digital Equipment Corporation. From 2000 to 2002, he was with TeraOptic Networks, Inc., as a Founder and a Vice President. He has published nearly 300 papers and has 183 patents issued in his name. He is a Fellow of ACM and AT&T, recognized for his fundamental contributions on communication networks, including his work on congestion control, traffic management, and VPN services.