Understanding Container Network Interface Plugins: Design Considerations and Performance

Shixiong Qi[†], Sameer G Kulkarni[*†], K. K. Ramakrishnan[†]
[†]University of California, Riverside, [*]IIT Gandhinagar, India.

***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 (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 paper, 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, as a result of the network model it implements, interaction with the host network protocol stack and the network policies implemented in iptables rules. The choice of the CNI plugin may also be based on whether intra-host or inter-host communication dominates.**

***Index Terms*—Kubernetes, Container network interface, Performance**

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. A pod may comprise of 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.

The proliferation of microservices [2] and function-as-a-service [3] architectures for deploying cloud-based services make it necessary to support large numbers of containers, and to provide efficient communication between them. Thus, networking is a core foundation of a good Pod orchestration framework. Kubernetes adopts the Container Network Interface (CNI) specification [4]. Each Pod in a Kubernetes cluster is given a unique IP address for communication. There are a number of different CNI implementations, and these CNI 'plugins' perform the tasks for Pod networking in a Kubernetes cluster. The CNI plugin is responsible for maintaining and orchestrating the Pod network. With thousands of Pods running in a cluster, the network 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 network namespace with the host network namespace (*e.g.,* , 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 [5], Weave [6], Cilium [7], Calico [8], and Kube-router [9] are popular [10]. While each find their application in different contexts due to their unique and distinct networking characteristics, we believe there is inadequate understanding and a lack of a comprehensive characterization on both the qualitative and performance aspects of these different CNI plugins. While existing works [11]–[14] study the overall performance of different CNI plugins at a preliminary level, there is still lack of in-depth understanding on how the various design considerations affect performance.

In this paper, we provide an insight on the overall performance of Pod networking with different CNI plugins, by examining throughput limits, latency and fine-grained CPU measurements of the various components of the networking stack. First, we present our qualitative analysis on the popular CNI plugins, namely: Flannel, Weave, Cilium, Calico and Kube-router to provide a high-level operational view and feature support of different CNIs. We also consider the different variants of the Calico plugin. We omit the other less frequently used (and some outdated) CNIs such as Romana [15], Canal [16] and Contiv-vpp [17] from our study. Then, we provide a measurement-driven quantitative analysis of their performance, accounting for the overheads due to the involvement of different aspects of the networking stack, user space and kernel space operations and their impact on CPU utilization. To summarize, our contributions include:

- We provide a qualitative analysis for different CNI plugins in terms of the subset of network or datalink layer features they support (*e.g.,* IPv6, Encryption support).
- 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 tunnelling, BPF, etc.) to determine the critical function calls that contribute to overhead.
- Based on this qualitative analysis, we examine the root cause for the performance differences across the CNI plugins for packet transmission through the entire network protocol stack with a measurement-based quantitative evaluation.

978-1-7281-8154-7/20/$31.00 ©2020 IEEE

- We also study the time to set up the pod network and the component latency, which impact the Pod "cold-start" time.

II. Qualitative Comparison on CNI Plugins

CNI plugins can be classified based on the network model, *i.e.,* networking layer they operate at (Layer 2/3), encapsulation and routing model they use to support intra-host (pods on same host node) and inter-host (pods on different nodes) communication. We also study the use of underlying kernel network configurations, the associated overheads with the user-space/kernel-space context switch and iptable chains.

A CNI plugin consists of i) CNI daemon and ii) CNI binary. The CNI daemon, which runs in user space, is responsible for setting up the host network devices (*e.g.,* bridge, overlay tunnel endpoint), the tunneling options (*e.g.,* VXLAN, UDP, *etc.*) and iptables (*e.g.,* network policies). In addition, some CNIs such as Flannel and Weave incorporate the packet encapsulation/decapsulation function within the daemon, rather than using kernel driver functionality, which results in additional context switches during packet processing. The CNI binary configures the network of a Pod, *i.e.,* setting up virtual ethernet (veth) and Pod IP address. It is called by the container runtime only when setting the network for a newly created Pod.

A. An Overview of CNI Design Paradigms

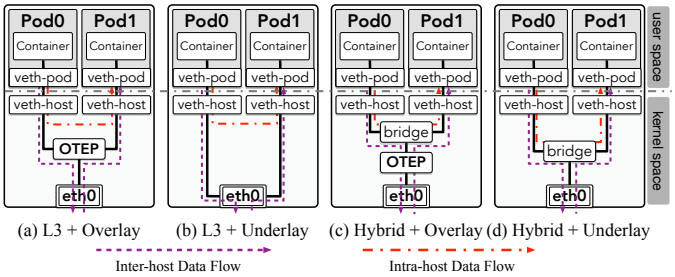


Fig. 1: An Overview of CNI Network Models

1) Network Models: The network model of the different CNI plugin designs can be classified into the following broad classes: i) Layer3(L3)+Overlay, ii) L3+Underlay, iii) Hybrid+Overlay and iv) Hybrid+Underlay. A hybrid approach is a combination of L2 and L3 operation. It typically utilizes a Linux bridge (L2) for intra-host packet forwarding, while using L3 inter-host packet forwarding.

L3+Overlay uses an overlay tunnel endpoint (OTEP) and multiple veth-pairs (Fig. 1(a)). The veth-pair enables data exchange between the Pod network namespace and the host network namespace. The OTEP is used to encapsulate/decapsulate the packets. An intra-host data exchange is handled by the host protocol stack in L3. For inter-host communication, the outgoing data packet is delivered to the OTEP via IP forwarding, where it get encapsulated and sent to its destination via the host's physical interface (eth0).

L3+Underlay comprises of veth-pairs (Fig. 1(b)). The intra-host data exchange using 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 a L2 and L3 design with overlays. It consists of a Linux bridge, an OTEP and multiple veth-pairs (Fig. 1(c)). A Linux bridge in the host network namespace connects with the Pods through veth-pairs facilitating intra-node data exchange using Layer 2 bridging. For inter-hosts data exchanges, the outgoing data packet first arrives at the bridge, 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. 1(d)). The intra-host data exchange in the hybrid approach works the same as in the Hybrid+Overlay approach. For inter-host communication, the outgoing data packet first arrives at the bridge, which is then handed over to the host protocol stack operating as a Layer 3 forwarder. With the host's IP forwarding turned on, the data packets will be sent through the host's physical interface (eth0) to the other node.

2) Iptables: This is a user space interface that interacts with CNI plugins and modifies the Netfilter rules based on the specified network policies. Netfilter inserts five hook points (callback function points) into the network stack to implement packet filtering and security related policies. The five hook points, namely: *PREROUTING*, *INPUT*, *OUTPUT*, *FORWARD* and *POSTROUTING*, are referred to as a 'chain' in iptables. Fig. 2 shows the iptables function chain processing. Generally, a packet will go through one of the following three paths when the conditions are satisfied:

- Path ①:Incoming packet's destination IP matches host's IP;
- Path ②:Incoming packet's dest. IP doesn't match host's IP;
- Path ③:Outgoing packet from the host.

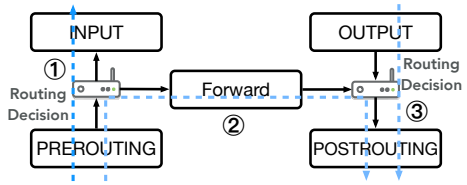


Fig. 2: Iptables chain processing and hook points

TABLE I: Qualitative Comparison on Features of CNI Plugins

CNI Solutions	*Network Models*	*Tunneling options*	*Network Policy*	*Additional Features*
Flannel	Hybrid+Underlay/Overlay	VXLAN/UDP	No	–
Weave	Hybrid+Overlay	VXLAN/UDP	Yes	Multicast, Encryption
Cilium	L3+Underlay/Overlay	VXLAN/Geneve	Yes	IPv6, Encryption
Calico	L3+Underlay/Overlay	IP-in-IP/VXLAN	Yes	IPv6
Kube-router	Hybrid+Underlay/Overlay	IP-in-IP/GRE	Yes	IPVS/LVS

B. Comparison of Different CNI Plugins

Table. I summaries the features of different CNI plugins. We describe the design for each of the CNI plugins based on the classification above.

Flannel uses the Hybrid+Overlay as its default network model. Flannel provides VXLAN (default) and UDP overlay choices. Flannel also has an optional underlay mode, which transfers packets using IP routing (Layer 3), but this mode needs direct Layer 2 connectivity between the hosts running Flannel. When operating in VXLAN mode, packets cross the boundary only once from veth at pod's side to veth at host's side (Fig. 1). But, in UDP mode, packets cross the boundary

three times as the UDP encapsulation needs to be done by the CNI daemon running in the user space. Packets to be transmitted will be sent back from the kernel to the CNI daemon in user space by the Linux bridge. After CNI daemon performs UDP encapsulation, the packets are again forwarded back to the kernel network stack with the encapsulation.

Weave uses Hybrid+Overlay approach with VXLAN tunnel as the default. In particular, Weave uses the Open vSwitch datapath module in the Linux kernel to implement its datapath. It adds an extra veth-pair between the OTEP and the Linux bridge. Thus, unlike the inter-hosts data exchange in a typical Hybrid+Overlay approach, Weave forwards the outgoing data packets directly to the OTEP via the bridge, without relying on host protocol stack. In Weave, packet processing incurs one context switch when operating in VXLAN mode. It has three context switches when operating in UDP mode, the same as Flannel.

Cilium applies L3+Overlay approach with VXLAN/GENEVE tunnelling. Cilium can also be changed to use underlay mode across hosts when the underlying infrastructure is able to route across hosts using the IP addresses of the Pods. The intra-host routing is implemented based on the extended Berkeley Packet Filter (eBPF) instead of the Linux IP forwarding function. For different network models and tunneling options, Cilium incurs just one context switch.

Calico provides support for both L3+Overlay and L3+Underlay. IP-in-IP overlay tunnel is the default choice, and offers a VXLAN overlay tunnel as a alternative. With the Border Gateway Protocol (BGP) enabled, Calico can fully operate at Layer 3 using IP routing. Calico also incurs just one context switch for all the working modes.

Kube-router can work in either a Hybrid+Overlay mode or use the Hybrid+Underlay approach. In the Hybrid+Overlay mode, Kube-router provides IP-in-IP or Generic Routing Encapsulation (GRE) as encapsulation options. When the underlying infrastructure can support BGP routing, Kube-router operates in the Hybrid+Underlay mode. Kube-router also has one context switch for all the working modes.

III. Quantitative Evaluation and Analysis

A. Experimental Setup

All the CNI plugins are evaluated on Cloudlab testbed [18]. We build the Kubernetes cluster on two nodes. Each node has a Ten-core Intel E5-2640v4 at 2.4 GHz, 64GB memory and 2 Dual-port Mellanox ConnectX-4 25 GB NIC. The nodes are connected via a 10Gbps Dell switches. 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 the current, latest version. We use Netperf for throughput and round-trip latency measurement. We use Netperf's TCP stream mode with default settings to perform the throughput measurements. For round-trip latency measurement, we use Netperf's request-response (RR) mode. For UDP, we use a message payload size of 1 byte. For all the experiments, we run each test for 10 seconds, and repeat the test 50 times.

B. Intra-Host Performance

We study intra-host performance (communication within a single server node) when communicating among number of containers deployed within the node. This enables us to better understand and distinguish the communication overheads that arise due to the usage of bridge, iptables rules, eBPF and the interaction with the host network stack.

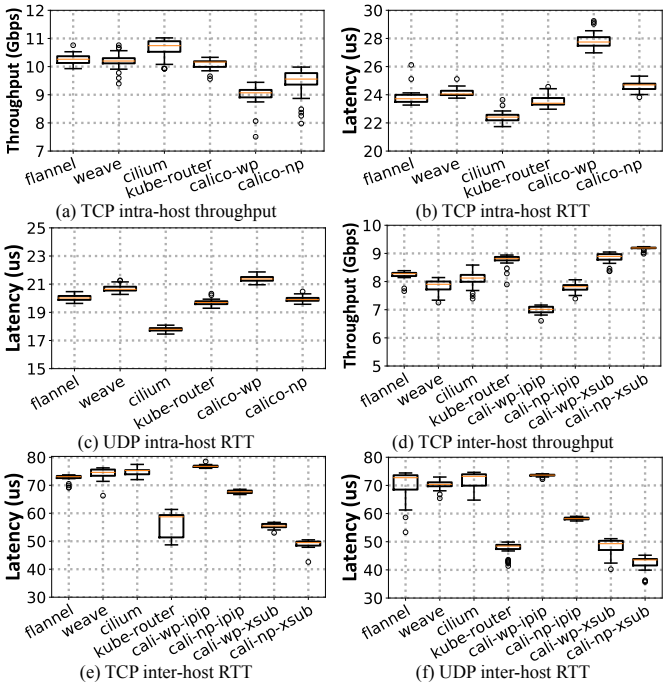


Fig. 3: Intra-Host & Inter-host Performance - (50 repetitions).

We evaluate Flannel, Weave, Cilium, Kube-router, Calico-np and Calico-wp CNIs. In order to isolate the overheads of network policies (Netfilter rules), we try Calico without the network policies, named as 'Calico-np'. We name the default Calico with network policies (*e.g.,* ordering/priority, allow/deny rules, *etc.*) fully installed as 'calico-wp'. In the intra-host scenario, Calico-np and Calico-wp forward packets using host IP protocol stack. Cilium builds a direct connection between the endpoints in the same host and directly forwards the packets by the eBPF programs attached to the veth. Kube-router, Weave and Flannel use Linux bridge to forward packets between different endpoints in the same host.

*1) **Overall performance**:* The overall throughput performance comparison is shown in Fig. 3 (a)-(c). For the TCP throughput (Fig. 3 (a)), Cilium with the native solution based on eBPF outperforms the other alternatives. Layer 3 routing based solutions (Calico-wp and Calico-np) are worse than the Layer 2 based solutions. Fig. 3 (b) and (c) show the TCP/UDP round-trip latency for intra-host packet forwarding. The relative ordering of TCP and UDP round-trip latency between different CNIs is only slightly different. Further, the TCP round-trip latency of each CNI is slightly higher than for UDP, primarily due to the protocol differences. Accordingly, we also observe that Cilium achieves the lowest round-trip latency and Calico-wp has the worst round-trip latency. 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 network models 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 that a packet needs to go 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 a specific CNI.

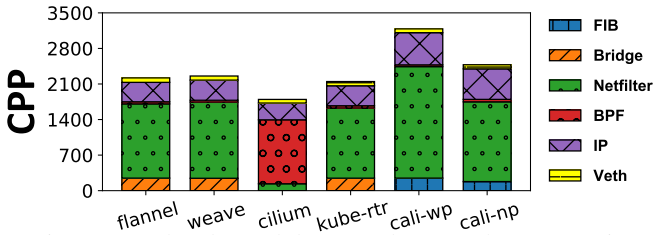


Fig. 4: Overhead Breakdown under Intra-host scenario.

Methodology: In order to measure the *CPU cycles per packets* (*CPP*) spent in each network stack component, we first use the Linux *perf* tool [19] to count the total CPU cycles consumed in a 10-second packet transmission ($Cycle_{total}$). 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 10-second packet transmission (N_{packet}), we can calculate the *CPP* of a specific function call as follows:

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

*2) **Overhead breakdown**:* The total *CPP* with the corresponding break down for intra-host communication is shown in Fig. 4. 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 adopt bridge-based solutions to forward packets in the intra-host scenario. When packets pass through the Linux bridge, the bridge-related function calls (*e.g., br_forward()*) are executed. Fig. 4 shows the bridge overhead of Flannel, Weave and Kube-router to be similar ~ 250 *CPP*.

FIB & IP forwarding: We put 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 look up 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 is around 200 *CPP*, which is slightly lower than the overhead using the Linux bridge (~ 250 *CPP*). But, the IP forwarding processing in Calico consumes an extra 240 *CPP*. This overhead of both FIB and IP forwarding is nearly 1.7× higher than the overhead of bridging approaches.

eBPF: Cilium relies heavily on eBPF. Instead of bridge/IP forwarding and Linux Netfilter [20], it utilizes a set of eBPF hooks in the network stack to run eBPF programs to support intra-host packet forwarding and filtering functions. Cilium attaches the eBPF programs at each veth resulting in each packet forwarding operation to incur a eBPF processing overhead. Fig. 4 shows Cilium has a somewhat high eBPF overhead of ~ 1200 *CPP*.

Veth: All the CNI plugins spend almost the same ~ 80 *CPP* (for send and receive) on veth, a small percentage of the overall overhead, with little impact on the performance differences.

Netfilter: Calico-wp, with calico policy fully installed, consumes 2200 *CPP* on Netfilter, which is 1.5× higher than the others. Although Calico-wp suffers 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 builds its FORWARD chain based on a tree-based structure with multiple levels, which incurs more *ipt_do_table()* calls. Note: Cilium does not have any Netfilter overhead as it uses eBPF instead of iptables chain.

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. Besides, a fine-grained iptables chain (per veth) as in Calico-wp unfortunately hurts packet transmission performance. As shown in Fig. 3 (a)-(c), Calico-wp has lower throughput and higher latency than the others for both TCP and UDP traffics 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 nodes. Flannel, Weave and Cilium use the VXLAN overlay to forward packets between different hosts. Kube-router uses native IP routing, while Calico (wp or np options) can support either native IP routing or IP-in-IP overlay. Accordingly for Calico, we choose the Cali-np-ipip, Cali-wp-ipip, Cali-np-xsub and Cali-wp-xsub modes.

*1) **Overall performance**:* The results for inter-host packet forwarding is shown in Fig. 3 (d)-(f). We see the native routing solutions (Kube-router, Cali-wp-xsub and Cali-np-xsub) perform better than the overlay solutions (Flannel, Weave, Cilium, Cali-wp-ipip and Cali-np-ipip) in TCP throughput and TCP/UDP round-trip latency. The comparison between the TCP and UDP round-trip latency (Fig. 3 (e) and (f)) keeps the same pattern as we discussed in the intra-host scenario. Moreover, the CNIs with simple (system default) iptables perform much better than the complex (with a number of user-defined iptables chains and rules) ones. For overhead analysis, we additionally include the VXLAN tunnel, IP-in-IP tunnel, OVS-datapath components.

*2) **Overhead breakdown for Inter-host**:* Again, we use the same methodology to calculate the *CPP* of each function call

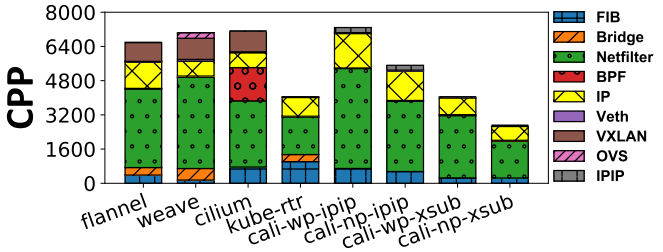


Fig. 5: Overhead Breakdown for Inter-host scenario.

on the receiver node. The total *CPP*, along with the breakdown is shown in Fig. 5. The native routing solutions (Kube-router, Cali-wp-xsub and Cali-np-xsub) have lower *CPP* compared to the overlay solutions (Flannel, Weave, Cilium, Cali-wp-ipip and Cali-np-ipip). Also, the solutions with simple iptables have lower *CPP* than the complex iptables chains.

Bridge: When transmitting a packet, Flannel and Kube-router use the host IP protocol stack to forward packets from the veth to the OTEP, bypassing the bridge forwarding overhead (Fig. 1 (c)). But with Weave, it uses an extra veth-pair to connect the Linux-bridge with the OTEP, and therefore incurs 2× more *CPP* when transmitting a packet from the container. Weave does not use the host IP protocol stack and instead relies on the bridge-related function calls (i.e., *br_forward()*). But, when receiving packets from the wire, all three (Flannel, Kube-router and Weave) CNIs use Layer 2 bridge forwarding to send packets from the overlay tunnel to the container.

FIB & IP forwarding: Weave, Cilium, Kube-router and Cali-*-xsub traverse the host IP stack once per packet transmission and have similar (700 ∼ 880 *CPP*) overhead. For Weave and Cilium, the IP protocol stack operations are performed between the VXLAN tunnel and host Ethernet interface; For Kube-router, between the Linux bridge and host Ethernet interface; For Calico-*-xsub, between the veth and host Ethernet interface. However, with Flannel and Cali-*-ipip, the host IP stack operations are performed twice per packet transmission and correspondingly incur about 1300 ∼ 1500 *CPP*. For Flannel, the IP protocol stack operation occurs between Linux bridge and VXLAN tunnel and then again between the VXLAN tunnel and host Ethernet interface; For the Cali-*-ipip, it first occurs between veth and IP-in-IP tunnel and then again between the IP-in-IP tunnel and host Ethernet interface.

All the CNI plugins include FIB processing to forward the packets. We count the number of FIB events after transferring 100,000 packets, as shown in Fig. 6. Kube-router has the most FIB events. We attribute this additional overhead in Kube-router to the additional FIB lookup operation involved to support the Direct Server Return (DSR) which is implemented using a custom routing table.

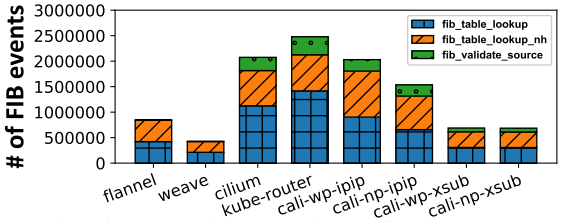


Fig. 6: Comparison on the # of FIB events.

Netfilter: Similar to intra-host routing, the Netfilter component is dominant, requiring much larger *CPP* than the others. Large and complex iptables chains incur higher processing overheads. Fig. 5 shows that Cali-wp-ipip has the highest Netfilter overhead, due to its large iptables size. Kube-router and Cali-np-xsub have the the lowest.

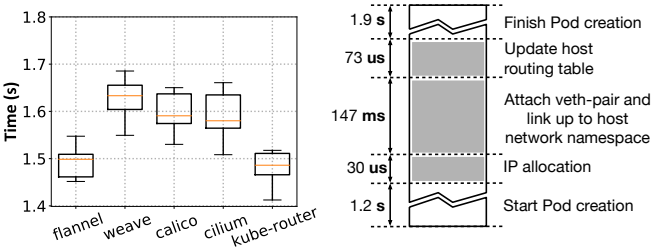
eBPF: Cilium uses eBPF to forward packets from veth to the overlay tunnel and vice versa. Due to the two additional eBPF hook points to attach with the overlay [20], the eBPF overhead for the inter-host routing is higher (1550 *CPP*) compared to the intra-host routing (1200 *CPP*), reducing Cilium's performance.

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 (∼ 45 *CPP*). As Weave has 4 veth-pairs on its inter-hosts datapath as opposed to 2 for others, it incurs double the overhead (∼ 90 *CPP*).

Overlay: Packet encapsulation and decapsulation are overlay overheads. Flannel, Weave and Cilium use VXLAN overlay and Cali-*-ipip use an IP-in-IP overlay. Overhead from the VXLAN overlay is ∼ 900 *CPP*, while IP-in-IP incurs much lower overhead (∼ 250 *CPP*). Weave uses the OVS-datapath to implement the overlay processing and incurs an extra overhead (∼ 250 *CPP*) on OVS-related function calls (Fig. 5).

Summary: Connecting the overlay tunnel and bridge via an extra veth-pair (as in Weave) can reduce the FIB and IP forwarding overhead, but increases the bridge and veth overhead. Although Weave has the least overhead for packet forwarding (including FIB, IP forwarding, Bridge and veth) it has a significant overhead for the Netfilter component, thus resulting in its somewhat lower overall performance for inter-host communication. A powerful network policy mechanism can provide fine-grained packet filtering, which allows for improved security for packet transmission. However, more Netfilter calls results in lower packet forwarding performance. Users should carefully consider their needs for the packet filtering and seek to manage the size of iptables 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.

D. Pod Launch Time Analysis



(a) Pod Launch Time in secs. (b) Breakdown of Pod launch time

Fig. 7: Pod Launch Time with different CNIs.

Starting a pod from scratch can take considerable time, and significantly impacts cloud-based microservices and Function as a Service offerings. The creation of a Pod is composed of multiple steps [21]. Once the "Kube-scheduler" schedules a

new Pod on a worker node, the Kubelet on the worker node invokes container runtime to create the namespace and the network interface for the Pod. The container runtime calls the CNI plugin to create network for the Pod. It will check the network configuration and allocate an IP address to the Pod. After the Pod gets its IP address, the CNI plugin attaches the veth-pair to the Pod and links it to the host network namespace. After the network initialization, the Pod network information will be persisted into etcd. Then the etcd will send an acknowledgement to the API server and the API server in turn generates an acknowledgment to the kubelet, which indicates the successful creation of a new Pod.

Fig. 7(a) compares the Pod startup time for different CNIs. We launch a new Pod, repeating the measurement 30 times. On average, Flannel and Kube-router are better. Weave has the highest launch latency. To see how much the network startup time influences the total Pod startup time, we breakdown the time spent on Pod network initialization for Weave. Fig. 7(b) shows that the time for Pod network initialization accounts for only 4.5% of total Pod startup time. Most of the network initialization time is spent on attaching the veth-pair to the Pod and linking it to the host network namespace. The major Pod startup time overhead is from the interaction between different Kubernetes components, such as those between API server and kubelet, etcd, and the scheduler. The CNI plugins themselves contribute little to the overall Pod startup time.

IV. Related Work

Several works [10]–[14] have compared and evaluated the performance of different CNI plugins. Suo et. al. [11] 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 they attribute performance differences to the chosen CNI, they do not identify root causes, as we have done here. The primary source of overhead is from how the CNI plugins interact with the network stack.

Kapocius [12] evaluates the performance of Kubernetes CNI plugins on both the virtual machines and bare metal. However, the work lacks adequate analysis on the performance differences observed for different CNIs. Bankston et. al. [13] compares the performance of CNI on different public cloud environments (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 popular open-source CNIs. Park et. al. [14] specifically compare the performance of Flannel network, OVS-based network and native-VLAN network, but only at a high level. Ducastel [10] evaluates the most popular CNI plugins using several benchmarks. They also provides a qualitative comparison on security and resource consumption, but are limited to the inter-host case, providing a high-level, throughput-only comparison. In general, the existing works fail to provide a kernel-level analysis and comparison of CNI plugins as we do in this work. Besides, we provide key insights on performance impact and working of different network models, iptables configuration and interaction with the host network stack.

V. Conclusion

Through qualitative analysis and a careful measurement-driven evaluation, we provide an in-depth understanding of the different CNI plugins and identify their key design considerations and performance. The evaluation results show the interactions between the different network models/iptables organizations and the host network stack and their contribution to the overall performance. While there is no single universally 'best' CNI plugin, there is a clear choice depending on intra-host or inter-host container communication. For the intra-host case, Cilium appears best with BPF 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 the overlay counterpart. Although Netfilter rules incur overhead, their rich, fine-grained network policy and customization can enhance cluster security – highly desirable for CSPs.

Acknowledgement: This work was supported by US NSF grants CRI-1823270 and CNS-1763929.

References

[1] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
[2] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: experiments and lessons learned," in *2018 IEEE 11th international conference on cloud computing (CLOUD)*. IEEE, 2018, pp. 970–973.
[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2658–2659.
[4] "Container network interface - networking for linux containers," https://github.com/containernetworking/cni.
[5] Flannel, https://github.com/coreos/flannel/.
[6] Weave, https://github.com/weaveworks/weave.
[7] Cilium, https://cilium.io/.
[8] Calico, https://github.com/projectcalico/calico-containers.
[9] Kube-router, https://www.kube-router.io/.
[10] A. Ducastel, "Benchmark results of kubernetes network plugins (cni) over 10gbit/s network," https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4.
[11] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.
[12] N. Kapočius, "Overview of kubernetes cni plugins performance," *Mokslas–Lietuvos ateitis/Science–Future of Lithuania*, vol. 12, 2020.
[13] R. Bankston and J. Guo, "Performance of container network technologies in cloud environments," in *2018 IEEE International Conference on Electro/Information Technology (EIT)*. IEEE, 2018, pp. 0277–0283.
[14] Y. Park, H. Yang, and Y. Kim, "Performance analysis of cni (container networking interface) based container network," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2018, pp. 248–250.
[15] "Romana," https://github.com/romana/romana.
[16] "Canal," https://github.com/projectcalico/canal.
[17] "Contiv-vpp," https://github.com/contiv/vpp.
[18] R. Ricci, E. Eide, and C. Team, "Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.
[19] Perf, https://github.com/torvalds/linux/tree/master/tools/perf.
[20] "Architecture of cilium," https://docs.cilium.io/en/v1.7/architecture/\#datapath.
[21] "Kubernetes: Lifecycle of a pod," https://dzone.com/articles/kubernetes-lifecycle-of-a-pod.