An Analysis and Empirical Study of Container Networks

Kun Suo*, Yong Zhao*, Wei Chen† and Jia Rao*
*Department of Computer Science and Engineering, the University of Texas at Arlington
†Department of Computer Science, the University of Colorado, Colorado Springs
Email: kun.suo@uta.edu, yong.zhao@uta.edu, cwei@uccs.edu, jia.rao@uta.edu

***Abstract*—Containers, a form of lightweight virtualization, provide an alternative means to partition hardware resources among users and expedite application deployment. Compared to virtual machines (VMs), containers incur less overhead and allow a much higher consolidation ratio. Container networking, a vital component in container-based virtualization, is still not well understood. Many techniques have been developed to provide connectivity between containers on a single host or across multiple machines. However, there lacks an in-depth analysis of their respective advantages, limitations, and performance in a cloud environment.**

In this paper, we perform a comprehensive study of representative container networks. We first conduct a qualitative comparison of their applicable scenarios, levels of security isolation, and overhead. Then we quantitatively evaluate the throughput, latency, scalability, and startup cost of various container networks in a realistic cloud environment. We find that virtualized network in containers incurs non-negligible overhead compared to physical networks. Performance degradation varies depending on the type of network protocol and packet size. Our experiments show that there is no clear winner in performance and users need to select an appropriate container network based on the requirements and characteristics of their workloads.

I. Introduction

While an increasing number of applications are being deployed in the cloud, the overhead of virtualization remains a critical concern. Traditional virtual machines (VMs) operate on virtualized hardware and run a complete copy of operating system (OS). The large footprint of full-fledged OSes limits the number of VMs that can be consolidated on a single machine and the long OS startup time makes it expensive for VMs to host short-lived applications. Container-based virtualization addresses these issues by sharing OS libraries and the kernel among applications, each of which runs in an isolated namespace, a.k.a., container. Studies have shown that containers incur negligible overhead and achieve near-native performance [21], [28], and take much less space [20]. Containers can be launched within a second and are ideal for hosting event-driven microservices [19], [25]. As a result, a single machine is expected to host hundreds or thousands of frequently launched and terminated containers. For example, Google Search launches about 7,000 containers every second [2] and a survey of 8 million container usage shows that 27% of containers have a lifetime shorter than 5 minutes and 11% shorter than 1 minute [16].

Providing network connectivity to a large volume of short-lived containers in a dynamic cloud environment presents a great challenge. First, the flexibility of OS-level virtualization allows multiple ways to connect a container to the external world. For example, there are four networking modes to interconnect Docker [3] containers on the same host and various ways to connect containers across machines. It is difficult to determine the appropriate network for a particular container workload. Second, the high density of containers on a single server requires that the network incurs minimal network performance degradation to individual connections as well as scaling to a large number of connections. Third, containers are not available for use until inter-container connections are established. Although a container can be started with sub-second latency, the time to establish its network can add considerable delays to the overall container launch time. Such delays are harmful to on-demand, short-lived container applications, such as event-triggered serverless code [22], [34]. Last, containers provide less security isolation than VMs. Thus, most cloud providers, e.g., Amazon Web Services, run containers inside VMs. The interaction between container network and the underlying VM network is not well understood.

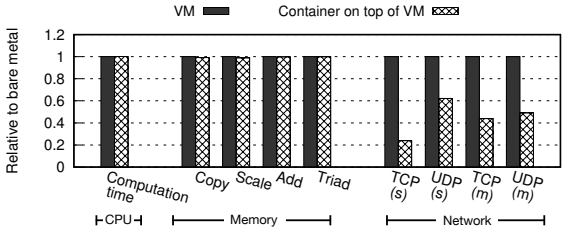

Fig. 1: CPU, memory, and network performance in VM and container on top of VM. Here (s) and (m) denote the performance on a single VM and multiple VMs, respectively.

Many studies revealed that containers incurred negligible overhead. To study the overhead of containerization in the cloud, we compared the performance of different types of applications in two settings. The baseline is to run applications directly in a VM. In contrast, containerization runs applications in a Docker container on top of a VM. we used a synthetic CPU benchmark that executed 100 million times calculation, *Stream* [15] and *Sockperf* [13] to measure the CPU, memory and end-to-end TCP and UDP throughput, respectively. Figure 1 shows that containerization does not degrade the performance of compute- and memory-intensive workloads in virtualized environments. However, network throughput drops dramatically when applications communicate through the container network compared to communicating directly via the VM network. Note that we used the default networking modes in Docker for the single VM and multiple VMs tests. In the baseline setting, both the client and server were separate processes and they communicated through the loopback net-

Cloud	Network on a single host	Network on multiple hosts
Amazon EC2	Bridge (Default) None Host	NAT (Default) Overlay Third party solutions
Docker Cloud	Bridge (Default) None Container Host	Overlay (Default)
Microsoft Azure	NAT (Default) Transparent Overlay L2Bridge L2Tunnel	NAT (Default) Transparent Overlay L2Bridge L2Tunnel

TABLE I: Available container networks in the public clouds.

work interface in the single VM case and through the VMs' IP addresses in the multiple VMs case. In contrast, when the client and server ran in separate containers on top of VMs, their communications occurred through an additional abstraction of container network, which caused significant network performance drop.

As networking performance is important to the user experience in containerized applications, it is critical to select the right network for containers. Unfortunately, this is not an easy task because many techniques have been developed to virtualize the network stack in the host OS to create isolated network for individual containers. Table I lists the available container networks in three public clouds. Broadly speaking, container networks fall into two categories: single-host network and multi-host network. Since communications within the same host are in fact carried out through shared memory, single-host network is mainly to provide a networking interface to containerized applications. Multi-host network centers on providing IP addressing services, such as network address translation (NAT), overlay network, or routing, to interconnect containers on different hosts.

In this paper, we present a detailed analysis of available container networks in Docker and perform an empirical study of their performance. We first investigate container networks on a single host and analyze their differences and use cases (§ II). Then we study container networks on multiple hosts, and analyze their techniques, respective advantages and disadvantages (§ III). We use representative benchmarks to evaluate the performance, scalability, and overhead of container networks (§ IV), and review the related work (§ V). Finally, we summarize this paper with insights and conclusion (§ VI).

To summarize, this paper has the following findings:

- Performance and security isolation present a difficult tradeoff in container networks on a single host. Good performance can be attained by sharing the same network namespace while security is enforced by using isolated namespaces.
- Multi-host networks present a different tradeoff. Overlay networks incur significant overhead due to packet encapsulation and decapsulation but allows more flexibility and security in network management. NAT and host mode networking achieves good performance but undermines security. Routing network shows good performance but requires additional support.
- Container network alone impose certain overhead. However, its deployment in VMs incurs additional throughput loss and latency increase. This is due to complex interactions between container networks and the virtualized network stack in VMs.
- Container networks have an order of magnitude difference in startup times. Short-lived and latency-sensitive workloads should take the startup time into consideration when selecting a container network.

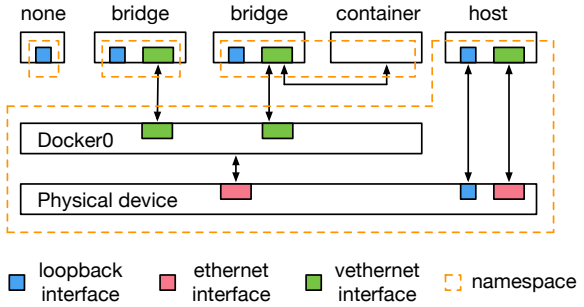

Fig. 2: Four container networking modes on a single host.

II. Container Network on a Single Host

In this section, we introduce four container networking modes on a single host, compare their differences, and discuss the use cases. Some of these networking modes are the basis for container networks on multiple hosts.

None mode The none network sets the container to a closed network stack. Such containers only have the loopback interface and cannot connect to containers on the same host or external networks. However, it achieves a *high* degree of isolation and security, and is suitable for services that do not require network access, such as offline data computation, batch processing, backup jobs, etc.

Bridge mode The bridge networking is the default network setting of Docker containers on a single host. As shown in Figure 2, Docker creates a bridge named `docker0` in the host OS once the Docker daemon `dockerd` is launched. When a new container is started, a pair of *veth* ports are created to connect the container to `docker0`. All containers connecting to *Docker0* belong to one virtual subnet and can communicate with each other using private IP addresses. Bridge mode alone does not connect containers to external networks and relies on other services, such as NAT and overlay, for inter-host communication. Bridge mode allows each container to own an isolated network namespace and an IP address, and all inter-container communications need to go through the `docker0` bridge. Packet transmission on the same host between containers needs to invoke the packet send and receive paths in isolated network namespace, thereby leading to high communication overhead. On the other hand, separating containers in independent network namespaces provides a *moderate* degree of security.

Container mode As shown in Figure 2, the container mode involves multiple containers, which share one network namespace. Within a group, one container is designated as a proxy and configured with the bridge mode. Other containers of the group connect to the external network through the proxy's virtual Ethernet (`veth`) interface. Since all containers in a group share one network namespace, only one IP address is assigned to the group and individual containers are identified

Network	Intra-machine communication	Inter-machine communication	Access to external networks	Namespace	Security
None	/	/	/	Independent, isolated	High
Bridge	`docker0` bridge	/	Bind host port, NAT	Independent, isolated	Moderate
Container	Inter-process communication	/	Bind host port, NAT	Shared with group leader	Medium
Host	Host network stack	Host network stack	Host network stack	Shared with the host OS	Low

TABLE II: Summary of four container networking modes on a single host.

by the group IP plus a port number. The container mode is widely adopted in many container management frameworks. For example, a *pod* in Kubernetes is a group of containers that share the same network namespace and an IP address. Containers with the container mode network can communicate with each other using standard inter-process communications (IPC). The container mode incurs less communication overhead than the bridge mode and achieves a *medium* level of security. While containers that belong to different groups are isolated, those in the same group do not have any protection from each other. Therefore, the container mode is most suitable for containers belonging to the same user. However, compared to the bridge mode, it does not allocate an IP address for each container. When a large-scale container deployment spans multiple machines, the user bears the burden to implement a hybrid communication scheme with both IPC and IPs.

Host mode The host networking allows all containers on the same host to share the network namespace of the host OS. As such, all containers are visible to each other and inter-container communications are based on IPC. The host mode provides the *lowest* level of security among the four networking modes [18] as all users share the same IP address as well as the namespace of the host machine.

[Summary] Table II summarizes the characteristics of the four networking modes on a single host. From the top to the bottom, the network becomes more efficient, whereas the degree of security isolation declines.

III. Container Network on Multiple Hosts

Host mode As discussed in Section II, containers in the host mode share the network stack and namespace of the host OS. Thus, two host mode containers on different machines can easily communicate with each other like two processes using the IP addresses of the machines. Although host mode network is straightforward to configure, only two host mode containers can communicate. For example, a bridge mode container can send packets to a host mode container on a different host using the destination host's IP address, but not vice versa. Furthermore, the host mode does not enforce security isolation between containers on the same host.

Network address translation (NAT) is the most commonly used technique in multi-host container network prior to Docker 1.9. NAT maps a container's private IP address to its port number in the NAT table. Communications must use host machines' public IP addresses plus a port number to identify a particular container. When a packet is sent by a container, the host machine at the source remaps the container's private IP address to the host's public IP address and changes the packet header. The destination host machine uses the port number in a received packet to map to the destination container's private IP address. Both the sender and receiver side translations are performed at bridge `docker0`.

NAT is a simple means to realize container connectivity across different hosts. It does not need complex configurations or the support of third-party software. In addition, as NAT allows containers to be addressed using the IP address of their host machine, it does not need a large number of public IP addresses for large-scale container deployment. However, network address translation at every packet sending and receiving incurs overhead and leads to some performance loss. Such overhead may not be neglected when a high density of containers rely on NAT [21] or the containerized applications are sensitive to performance degradation. Another limitation of NAT is that containers' public IP addresses are bound to their host IPs, making it difficult to implement dynamic container networks. In a dynamic network with short-lived containers, avoiding port conflicts in NAT is also a challenge.

Overlay network An overlay network runs on top of another network to build customized virtual links between nodes. Common forms of overlay network include IPIP, virtual extensible LAN (VXLAN), virtual private network (VPN), etc. There exist many overlay networks for Docker containers. Although they differ in implementation, the key idea is similar. Containers save the mapping between their private IP addresses and their host IP in a key-value (KV) store, which is accessible from all hosts. Containers use private IP addresses in a virtual subnet to communicate with each other. The overlay inserts an additional layer in the host network stack. When a packet is sent by a container, the overlay layer looks up the destination host IP address in the KV store using the private IP address of the destination container in the original packet. It then creates a new packet with the destination host IP address and uses the original packet sent by the container as the new packet's payload. This process is called packet encapsulation. Once the encapsulated packet arrives at the destination host, the host network stack decapsulates the wrapped packet to recover the original packet and delivers it to the destination container using the container's private IP address.

Compared to NAT, overlay network provides isolated address spaces and allows containers to communicate using private IP addresses. It is also easier to manage and resilient to changes in network topology. However, overlay network has two drawbacks. First, packet encapsulation and decapsulation are expensive operations and prolong the critical path of the network stack. Second, packet encapsulation changes the original packet size. When the underlying network is limited by a maximum transmission unit (MTU), the space overhead of packet encapsulation can increase the number of packet to be transmitted because the new packet may not fit in the MTU.

(a) The Docker's native overlay network is available since Docker 1.9 and has been the default network solution across multiple machines. As shown in Table III, Docker's overlay network adopts a VXLAN bridge to connect containers on the same host. The overlay network is built on *libkv* [8]

	Docker overlay	Weave	Flannel	Calico (IPIP)
Container	Container interface eth0	Container interface ethwe	Container interface eth0	Container interface cali0
Bridge	VXLAN bridge br0	Weave bridge vethwe-bridge	Docker bridge docker0	Docker bridge docker0
Overlay network device	VXLAN backend vxlan1	Weave router	VXLAN backend flannel0	VXLAN backend tunl0
Physical device	NIC interface eth0	NIC interface eth0	NIC interface eth0	NIC interface eth0

TABLE III: The network hierarchy in container overlay networks.

and *libnetwork* [9] and uses VXLAN to implement container connectivity across multiple hosts. The overlay network is the default network of *Docker Swarm* [4] and Docker provides extensive API support to configure the overlay. Before the native overlay support in Docker, many third-party solutions have been proposed. Here we discuss some most widely adopted solutions.

(b) *Weave* [17] is a virtual network solution developed by Weavework. Weave deploys a weave router container on each Docker host and the weave network is composed of these connected weave routers. The communications among weave routers can be encrypted using the NaCl crypto libraries [10] to enhance the data security. Weave creates a network bridge, `vethwe-bridge`, on each host to connect containers and the weave router. On the sender side, when a packet leaves a container, the weave router in the host captures this packet in promiscuous mode using *pcap*. It excludes the local traffic and forwards the packet to the weave routers on a remote host. On the receiver side, the weave router also uses *pcap* to transfer packets to the bridge and then forwards packets to destination containers. The weave router is implemented as a user-level service and handles packet encapsulation and decapsulation. As such, the overhead of building an overlay using weave also includes the cost to copy packets between kernel space and user space.

(c) *Flannel* [7] is a virtual network developed by CoreOS. In Docker native overlay, the Docker daemons on individual hosts allocate private IP addresses to containers independently. IP conflicts could occur when multiple containers on different machines are placed into the same overlay. Therefore, in Docker the overlay network is first created and containers are later added to the overlay to avoid IP conflicts. One major drawback of this approach is that the overlay cannot be dynamically created for already running containers as there is no way to address the potential IP conflicts. Flannel addresses this issue by ensuring that all containers on different hosts have different IP addresses. To achieve this, flannel restricts all containers on one host to the same subnet and allocates different subnets for different hosts. As such, overlay networks can be dynamically reconfigured as no IP conflicts would occur. Flannel maintains a distributed KV store, *etcd* [6], to store information about overlays and their address mappings. It inserts a virtual network interface, `flannel0`, between bridge `docker0` and the physical Ethernet device. Packet encapsulation and decapsulation are performed at `flannel0`. Flannel supports encrypted communication through Transport Layer Security (TLS) to secure the traffic.

(d) *Calico* [1] is a scalable and efficient virtual network developed by Tigera. It provides two options for container connectivity across multiple hosts: the IPIP overlay and Border Gateway Protocol (BGP) routing. The latter will be discussed in the following routing networks. Different from the stated overlay networks, which use either UDP or VXLAN packets for packet encapsulation, IPIP encapsulates packets at the network (IP) layer. The original IP packet is wrapped in another IP packet at the source of a network tunnel (i.e., `tunl0`) and decapsulated at the destination side of the tunnel. While Calico share the same drawback of expensive encapsulation and decapsulation, it imposes one additional limitation. It also requires the underlying infrastructure to support the IPIP protocol. However, many cloud providers, e.g., Microsoft Azure, do not support IPIP yet.

Routing Overlay networks offer a logical network connection between containers through creating a virtualized network layer on top of another network. Besides high overhead, overlay networks also make packet monitoring difficult as the real packet is encapsulated. To addresses these issues, Calico provides an alternative approach for inter-machine communication. It implements a virtual router in the host kernel and uses BGP for packet routing. As a network layer solution, Calico does not incur as much overhead compared to NAT and overlay network. Nevertheless, it has several limitations. First, Calico only supports a limited number of network protocols, such as TCP, UDP, ICMP, which limits its applicability. Second, BGP is not yet widely supported in data center networks. For instance, BGP cannot cross the zone boundaries in public clouds, Third, the size of the routing table limits the scale of a container network. It is also expensive to update routing information in BGP for highly dynamic networks with short-lived containers. Calico supports TLS to encrypt the network traffic between the etcd cluster and Calico components.

[Summary] Table IV summarizes available container networks on multiple hosts. Host mode networking is simple and fast, but sacrifices security and isolation. NAT is easy to use but containers' IP addresses are bound to host IPs, which poses challenges in port management and limits flexibility in network management. Table III summarizes the structures of various overlay networks. Overlay networks provide good isolation and security, but incur various degrees of overhead for per-packet processing. Further, all discussed overlays except weave rely on a cluster-wide KV store for private-public address mapping. This not only becomes a single point of failure but also imposes delays to container launch time. Routing is more efficient than overlay networks, but it only supports a subset of network protocols and requires the underlying infrastructure to support BGP.

IV. The Performance of Container Networks

This section presents a performance evaluation of the discussed container networks. We first evaluate the networking modes on a VM and multiple VMs using different protocols and various packet sizes, and then we study the impact of virtualization on container network performance. Last, we evaluate how

Network		How it works	Protocol	KV store	Security
Host mode		Sharing host network stack and namespace	ALL	No	No
NAT		Host network port binding and mapping	ALL	No	No
Overlay	Docker overlay	VXLAN	ALL	Yes	No
	Weave	VXLAN or UDP	ALL	No	Encrypted
	Flannel	VXLAN or UDP	ALL	Yes	Encrypted
	Calico (IPIP)	IP in IP	TCP/UDP/ICMP/ICMPv6/SCTP/UDPlite	Yes	Encrypted
Routing	Calico (BGP)	Border Gateway Protocol	TCP/UDP/ICMP/ICMPv6/SCTP/UDPlite	Yes	Encrypted

TABLE IV: Summary of container networks on multiple hosts.

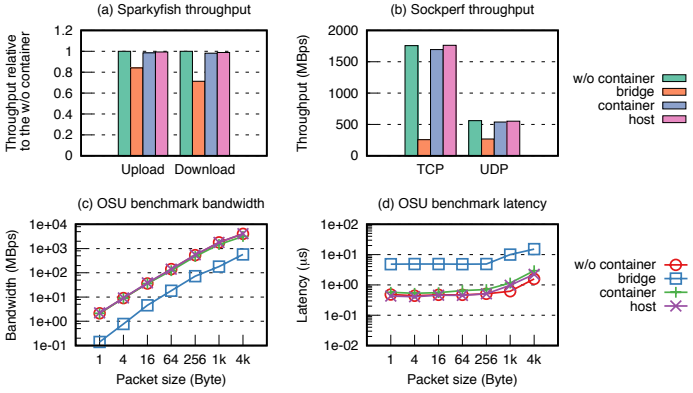

Fig. 3: Container network performance in a single VM.

different container networks respond to interference, their overhead in terms of CPU consumption, and the scalability. If not otherwise stated, we run containers in VMs to emulate a container hosting environment in public clouds.

A. Experiment settings

Hardware Our experiments were performed on two DELL PowerEdge T430 servers, connected by Gigabit Ethernet. Each server is equipped with a dual ten-core Intel Xeon E5-2640 2.6GHz processor, 64GB memory, and a 2TB 7200RPM SATA hard disk. Simultaneous multithreading (SMT) was disabled to reduce variability across multiple experiment runs.

Software We used Ubuntu 16.10 and Linux kernel 4.9.5 as the host and guest OS. The hypervisor was *KVM* 2.6.1 and the *Docker* version was Community Edition 1.12. The VMs are assigned with default rtl8139 NIC drivers. The overlay network software we used were *etcd* 2.2.5, *weave* 1.9.3, *flannel* 0.5.5 and *calico* 2.1.

Benchmarks We selected the following benchmarks to measure the performance of container networks.

- *Netperf* [11] is a network benchmark which provides unidirectional throughput and end-to-end latency measurement. The version we used was 2.7.0.
- *Sockperf* [13] is a network benchmarking utility over socket API that is designed for testing performance (latency and throughput) of high-performance systems. The version we used was 2.8.
- *Sparkyfish* [14] is an open-source network bandwidth and latency tester. Sparkyfish uses TCP streams to test the performance of bulk download and upload. The version we used was 1.2.
- *OSU benchmarks* [12] is a suite of benchmarks that measure the performance of Message Passing Interface (MPI) applications. The version we used was 5.3.2.

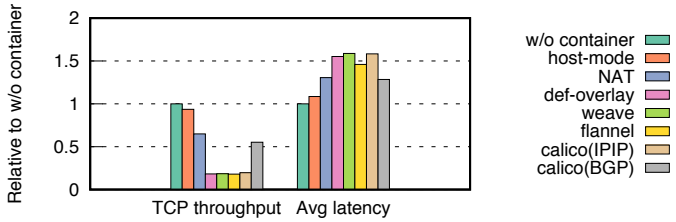

Fig. 4: Container network performance in multiple VMs.

B. Experiment results

Containers in a single VM We first compare point-to-point network performance in a single VM. The VM was configured with 4 vCPUs and 4GB memory. The benchmarks we used include a client and a server. As a comparison baseline, we ran the client and server as separate processes in the same VM (denoted as *w/o container*). Figure 3 (a) shows the performance of bulk network upload and download bandwidth using Sparkyfish. All container networks except the bridge mode achieved close performance to the baseline. In contrast, the bridge mode network incurred 18% and 30% throughput loss in upload and download, respectively. Figure 3 (b) shows TCP and UDP throughput of Sockperf in various modes. The packet size was set to 1024 bytes. Similarly, the bridge mode caused even more significant performance loss for TCP and UDP while other networking modes did not incur much overhead. Next, we used *osu_bw* and *osu_latency* to test network throughput and latency using various packet sizes. Likewise, the container mode and host mode achieved close performance to the baseline across all packet sizes. However, the bridge mode caused about 10-fold throughput loss and latency hike in all tests.

One commonality among the baseline, the container mode and the host mode is that they all share one network namespace. The baseline and the host mode share the network stack in the host OS while containers in the container mode share the proxy container's namespace. As such, inter-container communications in these modes were performed at the loopback interface and in fact were carried out through IPC. In comparison, bridge mode containers had separate network namespaces and all communications needed to go through bridge `docker0`. The additional layer in the network stack introduces two sources of overhead. First, it prolongs the critical path of packet processing. Packets need to be transmitted through `docker0`, leading to a traversal of the send and receive paths in the network stack as well as invoking an additional softirq for the bottom half of the receive interrupt handler. Second, it consumes more CPU resources and could impose queuing delays if CPU becomes the bottleneck.

Containers in multiple VMs To evaluate container network

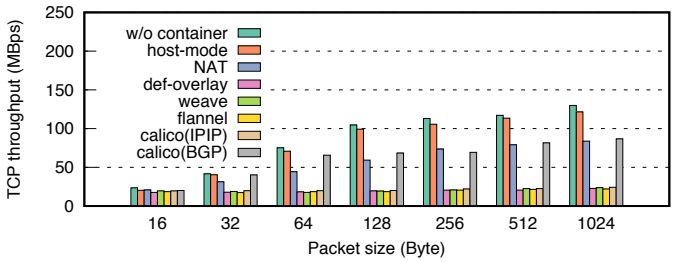

Fig. 5: TCP throughput in different packet sizes.

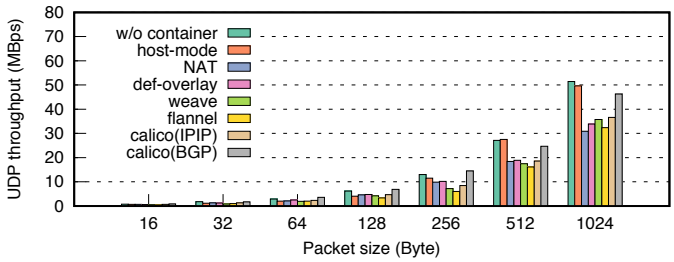

Fig. 6: UDP throughput in different packet sizes.

across multiple hosts, we used two VMs on the same PM. This setting allows for a much higher inter-VM link speed than the physical 1Gbps Ethernet link. Each VM was configured with 4 vCPUs and 4GB memory. We pinned vCPUs of the two VMs to separate cores to avoid any interference between the two. *Sockperf* was the benchmark and the packet size was set to 256 bytes for the TCP throughput test and 16 bytes for the TCP latency test. As shown in Figure 4, compared to the baseline (i.e., *w/o container*), the host mode network achieves almost the same throughput and average latency. Both NAT and Calico (BGP mode) incurred considerable performance degradation. This was due to the overhead of address translation and packet routing. However, all overlay networks caused unexpectedly high performance loss. For example, Docker native overlay inflicted 82.8% throughput drop and 55% latency increase compared to the baseline. Overall, packet encapsulation and decapsulation are more expensive than NAT and routing. As previously discussed, the bridge mode introduces additional network processing and all overlay networks require the containers to connect to the network bridge. Moreover, packet processing needs to traverse the network stack back and forth in overlay networks. For instance, packet decapsulation is invoked when the host OS processes an overlay packet at the transport layer (layer 4) and finds an enclosed packet. The recovered packet is then transmitted to the Ethernet interface and packet processing starts over from the receive interrupt handler. Later, the packet has to go through the protocol stack again when transmitted to the container via network bridge.

Packet size Packet size plays an important role on the network performance [24], [32]. Given a sustained data rate, a large packet size results in fewer packets being transmitted; given a fixed packet rate, a large packet size leads to a higher data rate. We used *Sockperf* to measure TCP and UDP throughput across two VMs on the same machine. *Sockperf* employs an open loop control flow at the sender side and sends a fixed number of packets. Thus, increasing the packet size increases the data rate. Figure 5 and Figure 6 show that both the baseline *w/o container* and the host mode scaled well with the data rate under TCP and UDP. However, NAT and Calico (BGP mode) incurred some overhead compared to the baseline but did scale the overall throughput. In contrast, all overlay networks suffered significant throughput loss. Under TCP, overlays were unable to scale as the packet size increases. Under UDP, the throughput of overlays scaled but with considerable loss. As previously discussed, packet encapsulation and decapsulation require more processing resources for each packet and a single packet of MTU size can hold more packets with smaller size, thereby leading to higher CPU consumption for larger packet size. We measured the CPU consumption of the VM at the receiver side, the single-stream TCP and UDP tests overloaded one vCPU in the 4-vCPU VM. However, the SMP VM was unable to parallel the packet processing using multiple vCPUs. An overlay packet requires multi-step processing, including packet decapsulation, interrupt handling for decapsulated packets, and packet forwarding on bridge `docker0`. However, the network stack is unaware of the amount of computation until overlay packets are examined at a particular protocol stack. This prevents effective load balancing on packet processing in the VM. In addition, virtualizing network IO at the VM level incurs increasing overhead as data rate increases. Thus, the aggregate CPU demand of virtualized I/O and overlay packet processing in the VM can exceed the capacity of one CPU, causing queuing delays on overlay packets and throughput loss. This also explains the overlay throughput loss in the UDP test.

Network Protocol Existing studies [26], [31] have demonstrated that it is important to select the appropriate network protocol to achieve desired quality-of-service (QoS). Figure 5 and 6 show that container networks performed differently under TCP and UDP. Overall, TCP achieved higher throughput than UDP did in all networks, including the baseline. TCP implements two mechanisms to efficiently utilize the physical link. First, the sliding window protocol avoids network congestion by dynamically adjusting the data sending rate. Second, TCP employs Nagles algorithm to combine small packets into bigger ones to improve link efficiency. Instead of transmitting a packet immediately after receiving it from user space, packets will be stored in a sending buffer in the kernel and sent out as one packet once the buffer is full. It is often advised that TCP is more desirable than UDP due to reliable transmission and optimized throughput. However, container networks in VMs present a new challenge in selecting a network protocol. As shown in Figure 5 and Figure 6, all inter-host container networks except the host mode incurred much higher throughput loss to TCP than that to UDP. For a representative 256-byte packet size [27], NAT and Calico (BGP) suffered 36% and 39% TCP throughput loss compared to the baseline, respectively, while overlay networks averaged an 82% TCP throughput loss. In comparison, NAT caused 24% loss and Calico (BGP) even outperformed the baseline under UDP. The average loss due to overlays was also much lower, averaging at 38%. As the acknowledgements in TCP involve more operations and CPU consumptions, the inter-host container networks incur more overhead to TCP than to UDP.

Impact of virtualization Containers can be deployed in VMs as well as on physical machines (PMs). There are four types of container deployment: (1) containers on the same PM; (2) containers in different VMs on the same PM; (3) containers on different PMs; (4) containers on different VMs on different PMs. So far, we have studied case (1)

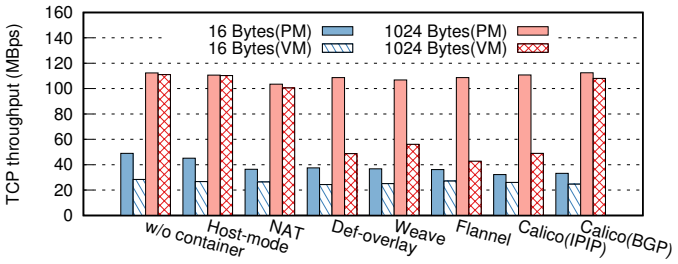

Fig. 7: Container network performance in PMs and VMs.

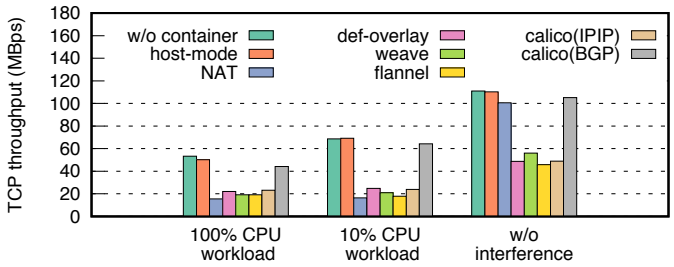

Fig. 8: Container network performance under interferences.

and (2). In this evaluation, we deploy container networks in case (3) and (4). We compare the performance of container networks using *Sockperf* in PMs and VMs, and show the results in Figure 7. For small packets which did not saturate the bandwidth, both hardware virtualization [29] and container network caused some throughput loss. For example, compared to the performance on physical machine, the TCP throughput of Sockperf executed directly in VMs and with Docker overlay networks in VMs degraded by 42% and 50.3%, respectively. The colocation of container networks and the virtualized network stack in VMs exacerbated the performance drop. This phenomenon is much more obvious for large packets. As shown in Figure 7, all the networks saturated the physical bandwidth when exeucting on the bare-metal using 1024-byte packets and deploying container networks on PMs caused marginal throughput loss. Similar with the small size packets, the multiple layer network virtualization introduced by VMs and container networks incurred drastic performance drop, especially in overlay networks.

Interference Due to low overhead, containers are often deployed in an environment with a high consolidation ratio. As container networking is a critical service to the container infrastructure, it is interesting to study how it responds to interference. We created a controlled environment to emulate interference from co-located containers in the same VM. We simulated two types of interference in the 4-vCPU, 4GB server VM: a) five containers, each executing a busy loop, to emulate workloads with persistent demand; b) five containers, each running a program periodically demanding 10% CPU, to emulate frequent container launch and termination. Figure 8 shows the impact of interferences on Sockperf TCP throughput. Compared to throughput without interferences, all cases, including the baseline, suffered TCP throughput loss under interference. This is because the server process in the benchmark was affected by the contending containers and received less CPU allocation. Likewise, container networks usually include a user space daemon for managing network flows and an in-kernel component for routing or packet encapsulation. The user space daemon is susceptible to slowdown due to interference while the kernel component is resilient to the influence as in-kernel processing has a strictly higher priority than any user-level computation. For example, both the host mode, which processes packets using the in-kernel network stack in the host OS, and Calico (BGP), which implements a in-kernel virtual router, were able to achieve performance close to the baseline as the container networking service was not affected by interference. In contrast, third-party overlays with user space daemons suffered most from interference. The experiments suggest that in a highly consolidated environment it is desirable to isolate container network services from other user container workloads or implement such services in the host OS kernel.

CPU overhead To evaluate the CPU overhead of various container networks, we created two one-vCPU, 4GB memory VMs on two PMs. We used *Netperf UDP_RR* to measure CPU utilization. The client and server side containers were configured as the same type of networking. Table V shows the CPU utilization and service demand of the client and server containers in various multi-host networks. Due to the reason that the transaction rate cannot be measured by Netperf for NAT, its results were not included in the table. The CPU utilization includes the time spent in user space, kernel space and waiting for I/O. It is measured in percentage. The service demand is the wall-clock time to complete one transaction in *UDP_RR*. Among the networks, the host mode consumed a similar level of CPU compared to the w/o container case. Calico (BGP mode) incurred more CPU overhead as it involves additional computation in the virtual router. All overlay networks consumed much more CPU due to packet encapsulation and decapsulation. The service demands show the cost of container networks at the request level. Similar to the trend in CPU consumption, networks that consumed more CPU, took a longer time to complete one transaction.

Scalability Many applications are composed of multiple components and are ideal use cases of containerization. For example, web services often consist of in-memory caches, load balancers, and backend databases. Big data frameworks, such as YARN, launch a large number of containers to parallelize a single job. Therefore, it is worth investigating the performance of concurrent communications between multiple containers. The all-to-all communication in the MapReduce shuffle phase is such an example. We used the `osu_alltoall` program in the *OSU benchmarks* to test all-to-all communication latency as the container network scales. The packet size was set to 1024 bytes and two VMs, each with 8 vCPUs and 4GB memory, were used to host containers. Figure 9 (a) shows the scalability of container networks in a single VM. We only tested the default bridge mode as it is the default as well as the basis for multi-host networks. While the baseline was to run the MPI program directly in the VM, the *w/ container* case ran MPI processes in separate containers. As shown in Figure 9 (a), the latency in the bridge mode network sharply increased as the number of containers increased. In comparison, the baseline scaled well with an increasing number of processes. It clearly shows that the centralized bridge `docker0` became the bottleneck when multiple containers connected to the bridge.

Figure 9 (b) and 9 (c) show the latency of container networks in multiple VMs. We evaluated two container distributions in two VMs. In Figure 9 (b), we fixed one container

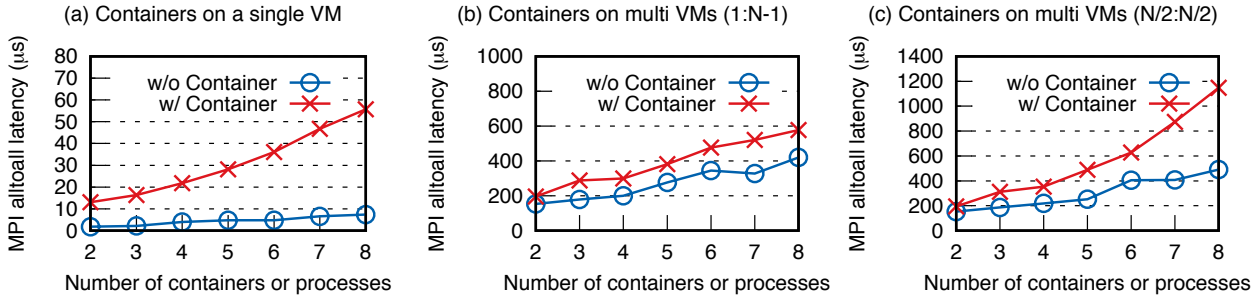

Fig. 9: Scalability of containers in single or multiple hosts. Here N denotes the number of containers.

Network	CPU (c)	CPU (s)	S.dem (c)	S.dem (s)
w/o container	33.37	18.41	75.466	41.626
Host mode	34.19	22.04	82.403	53.127
Def Overlay	38.75	42.91	95.867	106.145
Weave	54.92	47.56	150.678	130.478
Flannel	42.96	40.44	127.118	119.659
Calico(IPIP)	38.53	40.53	107.854	113.465
Calico(BGP)	37.72	36.92	99.665	95.035

TABLE V: CPU utilization and service demands (CPU time consumed per unit of work) of different container networks. Here (c) and (s) denote the client and server side.

Single host	Launch time	Multiple hosts	Launch time
None	539.6 ms	Host	497.4 ms
Bridge	663.1 ms	NAT	674.5 ms
Container	239.4 ms	Docker Overlay	10,979.8 ms
Host	497.4 ms	Weave	2,365.2 ms
/	/	Flannel	3,970.3 ms
/	/	Calico(IPIP)	11,373.1 ms
/	/	Calico(BGP)	11,335.2 ms

TABLE VI: The launch time of different networks

or process to a VM and placed all the other containers or processes onto the other VM. In Figure 9 (c), we placed an equal number of containers or processes on each VM. When the number of containers or processes are not divisible by 2, we rounded up to the nearest integer. Containers residing on the same VM communicated using the bridge mode and those on different VMs went through the overlay. We used the Docker native overlay for inter-VM communication. As Figure 9 depicts, we had several observations. First, the all-to-all latency across VMs was 10-fold higher than that in a single VM, suggesting that cross-VM latency was the culprit. The latency in the container network was higher than that in *w/o container*. As shown in Figure 9 (c), when more cross-VM communications were carried out through the overlay, the performance gap between *w/ container* and *w/o container* became wider as the number of containers increased. It suggests that the cross-VM overlay is the major bottleneck for scaling. The experiments provide insights on container placement. It is more desirable to pack communicating containers on the same host to avoid high communication cost in the overlay network.

Network launch time The time to establish a container network is critical to the short-lived or latency-sensitive workloads. We use the startup time of a *none* mode container as the baseline. Table VI lists the launch time of containers configured with various networks. The launch time was measured as the time between a container create command was issued and the container is responsive to network a ping request. The measurements were taken in a warmed-up system and were the average of ten runs. On a single host, all networks can be started with a latency similar to a docker image startup time except that the container mode unexpectedly accelerated the boot time. This is due to the fact that the proxy container has already initiated a network namespace and connecting an additional container to the proxy becomes cheaper than starting a new container.

On multiple hosts, the startup time of the host mode and NAT were comparable to that in a single host. However, launching an overlay or initializing the BGP routing tables took 4.5X to 23X longer than the original docker image startup time. For example, starting a Docker overlay took more than 10 seconds and most of the time was spent in registering the container with the KV store. Calico (BGP) was also slow to start as it took several seconds to propagate the routing table. Users should take the diverse startup latency of different networks into consideration to select the appropriate network to meet applications' latency requirements.

V. Related Work

Container network standard As containers are widely adopted in many fields and the scale of containerized applications continues to grow, there is a need for a standardized container network specification. Currently, there exist two standards for the container network interface. Container Network Model (CNM) is an official standard proposed by Docker, which is composed of modules such as sandbox, network and endpoints. CNM has been adopted by VMWare, Weave, etc. Container Network Interface (CNI) is a community standard proposed by Google and CoreOS for universal container network. Different from CNM, the design of CNI is more concise and flexible to use. CNI is supported by Apache Mesos, Kubernetes, etc. Many open source projects, such as Calico, support both two standards at the same time.

Container network optimization With the increasing popularity of containers, a growing number of studies start to focus on addressing inefficient networking in containers. Hu et al. [23] characterized typical NFV workloads in containers and observed that the shared network stack in the host OS was the main bottleneck. They proposed NetContainer, a software framework that achieved fine-grained hardware resource management for containerized NFV platform. To address poor performance in container network, Yu et al. [33] proposed FreeFlow, which used shared memory and RDMA to realize high throughput, low latency network among containers. There also exist many explorations in the industry, such as the

integration of Docker with SR-IOV or DPDK [5], to accelerate container networks. Those studies are orthogonal to our work.

VI. Insights and Conclusion

Insights To determine the appropriate network for a container workload is challenging. It requires the consideration of many factors. If users run containers on a single host, they need to balance the tradeoffs between performance, security, and isolation. If security and isolation are paramount, the bridge mode is the best option. If containers need to frequently communicate with each other and some containers need to access other containers' namespace for monitoring and management, the container mode should be the choice. The host mode delivers near-native performance, though not providing any isolation. If users build containers network across multiple hosts, Calico (BGP mode) is the best option from the performance perspective. NAT is the secondary choice if BGP is not supported in the architecture. All overlay networks perform similarly and introduce considerable overhead. Overlay network is best for workloads that do not require high network performance but have a frequently changing topology.

Besides the performance of container networks, the characteristics of the workloads also play an important role in choosing the appropriate network. For workloads communicating using small packets, e.g., messaging service, all the multiple hosts solutions perform similarly. For workloads with bulk transfer, Calico (BGP mode) achieves the better performance. In addition, container networks incur much larger overhead to TCP than to UDP workloads. Virtualization also introduces additional network overhead for containers, especially for overlay networks.

In order to improve the performance of container networks, the respective bottleneck in different container networks should be addressed. The shared network stack in the host OS, the centralized bridge `docker0`, software routing in Weave and Calico, and packet encapsulation and decapsulation in overlay networks could become the bottleneck for a particular workload. To improve container network on a single host, communications should be performed through shared memory when possible and avoid packet copying between user space and kernel space as much as possible. To improve container networks across multiple hosts, the expensive packet encapsulation and decapsulation operations can be accelerated through hardware offloading. As discussed in Section IV, there exist many factors influencing the network performance and it is challenging to select the right network. Techniques such as machine learning [30] can be used for the automate container network selection.

Conclusion In this paper, we present a detailed analysis of container networks on a single host and on multiple hosts. We perform a comprehensive empirical study of various container networks in a virtualized environment. To the best of our knowledge, this paper is the first to explore the many aspects of container networks. We have important findings that could help user select the appropriate network for their workloads and guide the optimization of existing container networks.

Acknowledgement

This work was supported in part by U.S. NSF grants CNS-1649502 and IIS-1633753.

References

[1] *Calico*. https://github.com/projectcalico/calico-containers.

[2] *Containers: The Future of Virtualization & SDDC*. https://goo.gl/Mb3yFq.

[3] *Docker*. https://www.docker.com/.

[4] *Docker Swarm*. https://docs.docker.com/engine/swarm/.

[5] *DPDK*. http://dpdk.org/.

[6] *etcd*. https://github.com/coreos/etcd.

[7] *Flannel*. https://github.com/coreos/flannel/.

[8] *Libkv*. https://github.com/docker/libkv.

[9] *Libnetwork*. https://github.com/docker/libnetwork.

[10] *NaCl: Networking and Cryptography library*. http://nacl.cr.yp.to/ec2/.

[11] *Netperf*. http://www.netperf.org/.

[12] *OSU benchmarks*. http://mvapich.cse.ohio-state.edu/benchmarks/.

[13] *Sockperf*. https://github.com/Mellanox/sockperf.

[14] *Sparkyfish*. https://github.com/chrissnell/sparkyfish.

[15] *Stream*. https://www.cs.virginia.edu/stream/.

[16] *The Truth about Docker Container Lifecycles*. https://goo.gl/Wcj894.

[17] *Weave*. https://github.com/weaveworks/weave.

[18] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *Proceedings of USENIX OSDI*, 2016.

[19] B. Burns and D. Oppenheimer. Design patterns for container-based distributed systems. In *Proceedings of USENIX HotCloud*, 2016.

[20] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *Proceedings of IEEE IC2E*, 2014.

[21] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *Proceedings of IEEE ISPASS*, 2015.

[22] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings USENIX HotCloud*, 2016.

[23] Y. Hu, M. Song, and T. Li. Towards full containerization in containerized network function virtualization. In *Proceedings of ASPLOS*, 2017.

[24] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of USENIX NSDI*, 2014.

[25] A. Panda, M. Sagiv, and S. Shenker. Verification in the age of microservices. In *Proceedings of USENIX HotOS*, 2017.

[26] K. K. Ram, I. C. Fedeli, A. L. Cox, and S. Rixner. Explaining the impact of network transport protocols on sip proxy performance. In *Proceedings of IEEE ISPASS*, 2008.

[27] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proceedings of ACM SIGCOMM*, 2015.

[28] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of ACM Middleware*, 2016.

[29] R. Shea and J. Liu. Network interface virtualization: challenges and solutions. *IEEE Network*, 2012.

[30] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of ACM SIGMOD*, 2017.

[31] G. Xylomenos and G. C. Polyzos. Tcp and udp performance over a wireless lan. In *Proceedings of IEEE INFOCOM*, 1999.

[32] M. Yanga, Y. Huanga, J. Kimb, M. Leec, T. Sudaa, and M. Daisuked. An end-to-end qos framework with on-demand bandwidth reconfiguration. In *Proceedings of IEEE INFOCOM*, 2004.

[33] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar. Freeflow: High performance container networking. In *Proceedings of ACM HotNet*, 2016.

[34] L. Zhang, J. Litton, F. Cangialosi, T. Benson, D. Levin, and A. Mislove. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In *Proceedings of ACM EuroSys*, 2016.