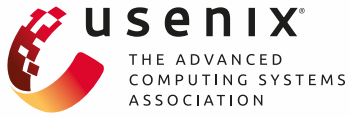


Slim: OS Kernel Support for a Low-Overhead Container Overlay Network

Danyang Zhuo and Kaiyuan Zhang, *University of Washington;*
Yibo Zhu, *Microsoft and Bytedance;* Hongqiang Harry Liu, *Alibaba;*
Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson, *University of Washington*

https://www.usenix.org/presentation/zhuo

This paper is included in the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19).

February 26–28, 2019 • Boston, MA, USA

ISBN 978-1-931971-49-2

Open access to the Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19) is sponsored by

NetApp®

Slim: OS Kernel Support for a Low-Overhead Container Overlay Network

Danyang Zhuo Kaiyuan Zhang Yibo Zhu[†*] Hongqiang Harry Liu[#]
Matthew Rockett Arvind Krishnamurthy Thomas Anderson

University of Washington [†] Microsoft Research [#] Alibaba

Abstract

Containers have become the de facto method for hosting large-scale distributed applications. Container overlay networks are essential to providing portability for containers, yet they impose significant overhead in terms of throughput, latency, and CPU utilization. The key problem is a reliance on packet transformation to implement network virtualization. As a result, each packet has to traverse the network stack twice in both the sender and the receiver's host OS kernel. We have designed and implemented Slim, a low-overhead container overlay network that implements network virtualization by manipulating connection-level metadata. Our solution maintains compatibility with today's containerized applications. Evaluation results show that Slim improves the throughput of an in-memory key-value store by 71% while reducing the latency by 42%. Slim reduces the CPU utilization of the in-memory key-value store by 56%. Slim also reduces the CPU utilization of a web server by 22%-24%, a database server by 22%, and a stream processing framework by 10%.

1 Introduction

Containers [6] have quickly become the de facto method to manage and deploy large-scale distributed applications, including in-memory key-value stores [32], web servers [36], databases [45], and data processing frameworks [1, 26]. Containers are attractive because they are lightweight and portable. A single physical machine can easily host more than ten times as many containers as standard virtual machines [30], resulting in substantial cost savings.

Container overlay networks—a key component in providing portability for distributed containerized applications—allow a set of containers to communicate using their own independent IP addresses and port numbers, no matter where they are assigned or which other containers reside on the same physical machines. The overlay network removes the burden of coordinating ports and IP addresses between application developers, and vastly simplifies migrating legacy enterprise applications to the cloud [14]. Today, container orchestrators, such as Docker Swarm [9], require the usage of overlay network for hosting containerized applications.

[*] Yibo now works at Bytedance.

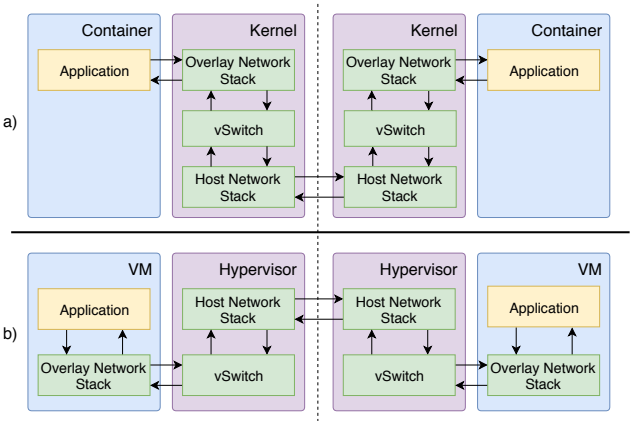


Figure 1: *Packet flow in: (a) today's container overlay networks, (b) overlay networks for virtual machines.*

However, container overlay networks impose significant overhead. Our benchmarks show that, compared to a host network connection, the throughput of an overlay network connection is 23-48% less, the packet-level latency is 34-85% higher, and the CPU utilization is 93% more. (See §2.2.) Known optimization techniques (e.g., packet steering [40] and hardware support for virtualization [22, 14]) only partly address these issues.

The key problem is that today's container overlay networks depend on multiple packet transformations within the OS for network virtualization (Figure 1a). This means each packet has to traverse network stack **twice** and also a virtual switch on both the sender and the receiver side. Take sending a packet as an example. A packet sent by a container application first traverses the overlay network stack on the virtual network interface. The packet then traverses a virtual switch for packet transformation (e.g, adding host network headers). Finally, the packet traverses the host network stack, and is sent out on the host network interface. On the receiving server, these layers are repeated in the opposite order.

This design largely resembles the overlay network for virtual machines (Figure 1b). Because a virtual machine has its own network stack, the hypervisor has to send/receive raw overlay packets without the context of network connections. However, for containers, the OS kernel has full knowledge of each network connection.

In this paper, we ask whether we can design and implement a container overlay network, where packets go through

the OS kernel's network stack **only once**. This requires us to remove packet transformation from the overlay network's data-plane. Instead, we implement network virtualization by manipulating connection-level metadata at connection setup time, saving CPU cycles and reducing packet latency.

Realizing such a container overlay network is challenging because: (1) network virtualization has to be compatible with today's unmodified containerized applications; (2) we need to support the same networking policies currently enforced by today's container overlay network on the data-plane; and (3) we need to enforce the same security model as in today's container overlay networks.

We design and implement Slim, a low-overhead container overlay network that provides network virtualization by manipulating connection-level metadata. Our evaluations show that Slim improves the throughput of an in-memory key-value store, Memcached [32], by 71% and reduces its latency by 42%, compared to a well-tuned container overlay network based on packet transformation. Slim reduces the CPU utilization of Memcached by 56%. Slim also reduces the CPU utilization of a web server, Nginx [36], by 22%-24%; a database server, PostgreSQL [45], by 22%; and a stream processing framework, Apache Kafka [1, 26], by 10%. However, Slim adds complexity to connection setup, resulting in 106% longer connection setup time. Other limitations of Slim: Slim supports quiescent container migration, but not container live migration; connection-based network policies but not packet-based network policies; and TCP, defaulting to standard processing for UDP sockets. (See §7.)

The paper makes the following contributions:

- Benchmarking of existing container overlay network with several data-plane optimizations. We identify per-packet processing costs (e.g., packet transformation, extra traversal of network stack) as the main bottleneck in today's container overlay network. (See §2.2, §2.3.)
- Design and implementation of Slim, a solution that manipulates connection-level metadata to achieve network virtualization. Slim is compatible with today's containerized applications and standard OS kernels. Slim supports various network policies and guarantees the same security model as that of today's container overlay network. (See §4.)
- Demonstration of the benefits of Slim for a wide range of popular containerized applications, including an in-memory key-value store, a web server, a database server, and a stream processing framework. (See §6.)

Fundamentally, Slim integrates efficient virtualization into the OS kernel's networking stack. A modern OS kernel already has efficient native support to virtualize file systems (using *mount* namespace) and other OS components (e.g., process id, user group). The network stack is the remaining performance gap for efficient container virtualization. Slim bridges this gap.

Mode	Applications use	Routing uses
Bridge	Container IP	–
Host	Host IP	Host IP
Macvlan	Container IP	Container IP
Overlay	Container IP	Host IP

Table 1: *Container networking mode comparison.*

2 Background

We first describe the architecture of traditional container overlay networks and why they are useful for containerized applications. We then quantify the overhead of today's container overlay network solutions in terms of throughput, latency, and CPU utilization. Finally, we show that the overhead is significant even after applying known overhead reduction techniques (e.g., packet steering [40]).

2.1 Container Overlay Network

Containers typically have four options for communication: bridge mode, host mode, macvlan mode, and overlay mode. Table 1 shows the comparison between different modes in terms of the IP addresses used by containerized applications and routing in the host network. Bridge mode is used exclusively for containers communicating on the same host. With bridge mode, each container has an independent IP address, and the OS kernel routes traffic between different containers.

How can we enable communication between containers on different hosts? With host mode, containers directly use the IP address of their host network interface. The network performance of host mode is close to the performance of any process that directly uses the host OS's network stack. However, host mode creates many management and deployment challenges. First, containers cannot be configured with their own IP addresses; they must use the IP address of the host network interface. This complicates porting: distributed applications must be re-written to discover and use the host IP addresses, and if containers can migrate (e.g., after a checkpoint), the application must be able to adapt to dynamic changes in their IP address. Worse, because all containers on the same host share the same host IP address, only one container can bind to a given port (e.g., port 80), resulting in complex coordination between different applications running on the same host. In fact, container orchestrators, such as Kubernetes, do not allow usage of host mode [27] due to these issues.

Macvlan mode or similar hardware mechanisms (e.g., SR-IOV) allow containers to have their own IP addresses different from their hosts. Macvlan or SR-IOV allow the physical NIC to emulate multiple NICs each with a different MAC address and IP address. Macvlan [1] extends the host network into the containers by making the container IP routable on the host network. However, this approach fundamentally

[1]There are software approaches (e.g., Calico [3]) to extend the host network into containers. They have the same problem as macvlan.

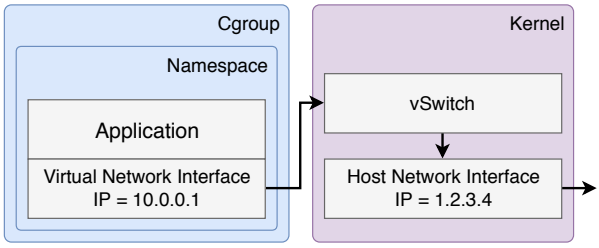


Figure 2: *Architecture of container overlay network.*

complicates data center network routing. Let's say a distributed application with IP addresses IP 1.2.3.[1-10] is not co-located on the same rack, or starts co-located but then some containers are migrated. Then the host IP addresses will not be contiguous, e.g., one might be on host 5.6.7.8 and another might be on host 9.10.11.12. Macvlan requires the cloud provider to change its core network routing to redirect traffic with destination IP 1.2.3.[1-10] to 5.6.7.8 and 9.10.11.12, potentially requiring a separate routing table entry for each of the millions of containers running in the data center. Another limitation is that containers must choose IP addresses that do not overlap with the IP addresses of any other container (or host). Because of these complications, today, most cloud providers block macvlan mode [29].

To avoid interference with routing on the host network, the popular choice is to use overlay mode. This is the analog of a virtual machine, but for a group of containers—each application has its own network namespace with no impact or visibility into the choices made by other containers or the host network. A virtual network interface (assigned an IP address chosen by the application) is created per-container. The virtual network interface is connected to the outside world via a virtual switch (e.g., Open vSwitch [42]) inside the OS kernel. Overlay packets are encapsulated with host network headers when routed on the host network. This lets the container overlay network have its own IP address space and network configuration that is disjoint from that of the host network; each can be managed completely independently. Many container overlay network solutions are available today—such as Weave [52], Flannel [15], and Docker Overlay [8]—all of which share similar internal architectures.

Figure 2 presents a high-level system diagram of a container overlay network that uses packet transformation to implement network virtualization. It shows an OS kernel and a container built with namespaces and cgroups. Namespace isolation prevents a containerized application from accessing the host network interface. Cgroups allow fine-grained control on the total amount of resources (e.g., CPU, memory, and network) that the application inside the container can consume.

The key component of a container overlay network is a virtual switch inside the kernel (Figure 2). The virtual switch has two main functionalities: (1) network bridging, allowing containers on the same host to communicate, and (2) network tunneling to enable overlay traffic to travel across the physical network. The virtual switch is typically configured using the Open vSwitch kernel module [42] with VXLAN as the tunneling protocol.

To enforce various network policies (e.g., access control, rate limiting, and quality of service), a network operator or a container orchestrator [27, 9, 18] issues policy updates to the virtual network interface or the virtual switch. For example, firewall rules are typically implemented via *iptables* [20], and rate limiting and quality of service (QoS) can also be configured inside the Open vSwitch kernel module. These rules are typically specified in terms of the application's virtual IP addresses, rather than the host's IP addresses which can change depending on where the container is assigned.

The hosts running a set of containers in an overlay network must maintain a consistent global network view (e.g., virtual to physical IP mappings) across hosts. They typically do this using an external, fault-tolerant distributed datastore [13] or gossiping protocols.

2.2 Overhead in Container Overlay Networks

The overhead of today's container overlay networks comes from per-packet processing (e.g., packet transformation, extra traversal of the network stack) inside the OS kernel.

2.2.1 Journey of an Overlay Network Packet

In our example (Figure 2), assume that a TCP connection has previously been established between 10.0.0.1 and 10.0.0.2. Now, the container sends a packet to 10.0.0.2 through this connection. The OS kernel's overlay network stack first writes the virtual destination IP address 10.0.0.2 and source IP address 10.0.0.1 on the packet header. The OS kernel also writes the Ethernet header of the packet to make the packet a proper Ethernet frame. The Ethernet frame traverses a virtual Ethernet link to the virtual switch's input buffer.

The virtual switch recognizes the IP address 10.0.0.2 inside the Ethernet frame as that of a container on a remote host. It adds a physical IP header to the Ethernet frame using host source and destination addresses from its routing table. The packet now has both a physical and a virtual header. On the host network, the packet is simply a UDP packet (assuming the tunneling protocol is VXLAN) and its UDP payload is the Ethernet frame. The OS kernel then delivers the encapsulated packet to the wire using the host network stack.

The receiving pipeline is the same except that the virtual switch removes the host network header instead of adding one. The receiving side receives the exact same Ethernet frame from the sending side's virtual network interface.

We can thus see why the overlay network is expensive: delivering a packet on the overlay network requires one extra traversal of the network stack and also packet encapsulation and decapsulation.

Setup	Throughput (Gbps)	RTT (µs)
Intra, Host	48.4 ± 0.7	5.9 ± 0.2
Intra, Overlay	37.4 ± 0.8 (23%)	7.9 ± 0.2 (34%)
Inter, Host	26.8 ± 0.1	11.3 ± 0.2
Inter, Overlay	14.0 ± 0.4 (48%)	20.9 ± 0.3 (85%)

Table 2: *Throughput and latency of a single TCP connection on a container overlay network, compared with that using host mode. Intra is a connection on the same physical machine; Inter is a connection between two different physical machines over a 40 Gbps link. The numbers followed by ± show the standard deviations. The numbers in parentheses show the relative slowdown compared with using host mode.*

2.2.2 Quantifying Overhead

We give a detailed breakdown of the overhead in one popular container overlay network implementation, Weave [52]. Our testbed consists of two machines with Intel Xeon E5-2680 (12 physical cores, 2.5 GHz). The machines use hyper-threading and therefore each has 24 virtual cores. Each machine runs Linux version 4.4 and has a 40 Gbps Intel XL710 NIC. The two machines are directly connected via a 40 Gbps link. The physical NIC is configured to use Receive Side Scaling (RSS). In all of our experiments, we do not change the configuration of the physical NICs.

We create an overlay network with Weave's fast data-plane mode (similar to the architecture in Figure 2). We use *iperf3* [19] to create a single TCP connection and study TCP throughput atop the container overlay network. We use *NPtcp* [39] to measure packet-level latency. For comparison, we also perform the same test using host mode container networking. In all of our experiments, we keep the CPU in maximum clock frequency (using Intel P-State driver [47]).

The overhead of the container overlay network is significant. We compare TCP flow throughput and packet-level latency under four different settings. Table 2 shows average TCP flow throughput with maximum ethernet frame size over a 10-second interval and the round trip latency for 32-byte TCP packets for 10 tests. For two containers on the same host, TCP throughput reduces by 23% and latency increases by 34%. For containers across physical machines, TCP throughput reduces by almost half (48%) and latency increases by 85%. Intra-host container overlay network has lower overheads because packet encapsulation is not needed.

To understand the source of the main bottleneck, we measure CPU utilization with a standard Linux kernel CPU profiling tool, *mpstat*. We specifically inspect the overlay network across two different physical machines. We set the speed of the TCP connection to 10 Gbps and then use *mpstat* to identify where CPU cycles are spent for 10 tests where each test lasts 10 seconds. Figure 3 shows the overall CPU utilization and the breakdown. Compared with using a direct host connection, in the default mode (Random IRQ load balancing), the overlay network increases CPU utilization (relatively) by 93%. RPS (receive packet steering) and RFS (receive flow steering) are two optimizations we have done to Weave. (See §2.3.)

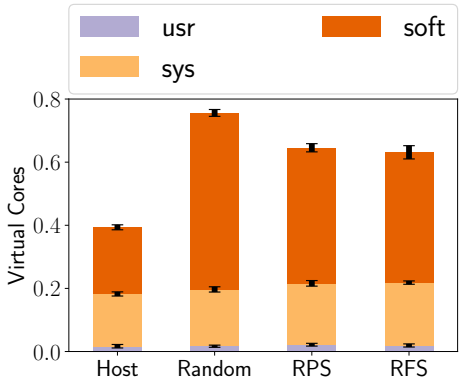


Figure 3: *CPU utilization under different overlay network setups measured by number of virtual cores used for a single 10 Gbps TCP connection. The CPU cycles are spent: in user-level application (usr), inside kernel but excluding interrupt handling (sys), and serving software interrupts (soft). Error bars denote standard deviations.*

The main CPU overhead of the overlay network comes from serving software interrupts; in the default overlay setting, it corresponds to 0.56 virtual cores. The reason why the extra CPU utilization is in the software interrupt handling category is that packet transformation and the traversal of the extra network stack is not directly associated with a system call. These tasks are offloaded to per-core dedicated *softirq* thread. For comparison, using the host mode, only 0.21 virtual cores are spent on serving software interrupts. This difference in CPU utilization captures the extra CPU cycles wasted on traversing the network stack one extra time and packet transformation. Note here we do not separate the CPU utilization due to the virtual switch and due to the extra network stack traversal. Our solution, Slim, removes both these two components from the container overlay network data-plane at the same time, so understanding how much CPU utilization these two components consume combined is sufficient.

In §2.3, we show that existing techniques (e.g., packet steering) can address some of the performance issues of a container overlay network. However, significant overhead still remains.

2.3 Fine-Tuning Data-plane

There are several known techniques to reduce the data-plane overhead. Packet steering creates multiple queues, each per CPU core, for a network interface and uses consistent hashing to map packets to different queues. In this way, packets in the same network connection are processed only on a single core. Different cores therefore do not have to access the same queue, removing the overhead due to multi-core

Setup	Throughput (Gbps)	RTT (µs)
Random LB	14.0 ± 0.4 (48%)	20.9 ± 0.3 (85%)
RPS	24.1 ± 0.8 (10%)	20.8 ± 0.1 (84%)
RFS	24.5 ± 0.3 (9%)	21.2 ± 0.2 (88%)
Host	26.8 ± 0.1	11.3 ± 0.2

Table 3: *TCP throughput and latency (round-trip time for 32-byte TCP packets) for different packet steering mechanisms atop a container overlay network across two physical hosts. The numbers followed by ± show the standard deviations. The numbers in parentheses show the relative slowdown compared with using the host mode.*

synchronization (e.g., cache-line conflicts, locking). Table 3 shows the changes to throughput and latency on a container overlay network using packet steering.

Packet steering improves TCP throughput to within 91% of using a host TCP connection, but it does not reduce packet-level latency. We experimented with two packet steering options, Receive Packet Steering (RPS) and Receive Flow Steering (RFS), for internal virtual network interfaces in the overlay network. RPS[2] ensures that packets in the same flow always hit the same core. RFS, an enhancement of RPS, ensures that software interrupt processing occurs on the same core as the application.

Although packet steering can imporve throughput, it has a more modest impact on CPU utilizaion than throughput and almost no change to latency. Packets still have to go through the same packet transformations and traverse the network stack twice. Our design, Slim, focuses directly on removing this per-packet processing overhead in container overlay networks.

3 Overview

Slim provides a low-overhead container overlay network in which packets in the overlay network traverse the network stack exactly once. Like other container overlay network implementations [52, 8, 15], Slim creates a virtual network with a configuration completely decoupled from the host network's. Containers have no visibility of host network interfaces, and they communicate only using virtual network interfaces that the OS kernel creates.

We require Slim to be (1) *readily deployable*, supporting unmodified application binaries; (2) *flexible*, supporting various network policies, such as access control, rate limiting, and quality of service (QoS), at both per-connection and per-container levels; and (3) *secure*, the container cannot learn information about the physical hosts, create connections directly on host network, or increase its traffic priority.

Figure 4 shows Slim's architecture. It has three main components: (1) a user-space shim layer, *SlimSocket*, that is dynamically linked with application binaries; (2) a user-space router, *SlimRouter*, running in the host namespace; and (3) a small optional kernel module, *SlimKernModule*, which augments the OS kernel with advanced Slim features (e.g., dynamically changing access control rules, enforcing security).

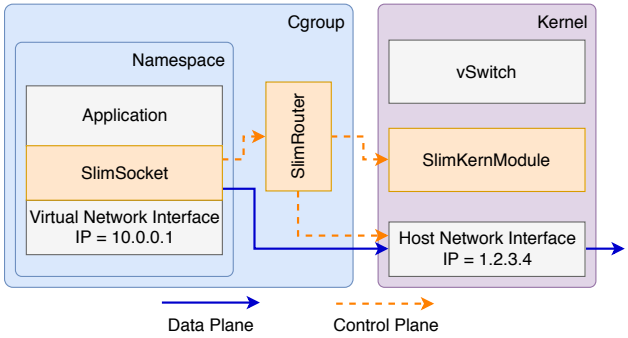


Figure 4: *Architecture of Slim.*

Slim virtualizes the network by manipulating connection-level metadata. *SlimSocket* exposes the POSIX socket interface to application binaries to intercept invocations of socket-related system calls. When *SlimSocket* detects an application is trying to set up a connection, it sends a request to *SlimRouter*. After *SlimRouter* sets up the network connection, it passes access to the connection as a file descriptor to the process inside the container. The application inside the container then uses the host namespace file descriptor to send/receive packets directly to/from the host network. Because *SlimSocket* has the exact same interface as the POSIX socket, and Slim dynamically links *SlimSocket* into the application, the application binary need not be modified.

In Slim, packets go directly to the host network, circumventing the virtual network interface and the virtual switch; hence, a separate mechanism is needed to support various flexible control-plane policies (e.g., access control) and data-plane policies (e.g., rate limiting, QoS). Control-plane policies isolate different components of containerized applications. Data-plane policies limit a container's network resource usage and allow prioritization of network traffic. In many current overlay network implementations, both types of policies are actually enforced inside the data-plane. For example, a typical network firewall inspects every packet to determine if it is blocked by an access control list.

SlimRouter stores control-plane policies and enforces them at connection setup time. This approach obviates the need to inspect every packet in the connection. Before creating a connection, *SlimRouter* checks whether the access control list permits the connection. When the policy changes, *SlimRouter* scans all existing connections and removes the file descriptors for any connection that violates the updated access control policy through *SlimKernModule*. Slim leverages existing kernel functionalities to enforce data-plane policies.

Sending a host namespace file descriptor directly to a ma-

[2]RSS requires hardware NIC support. RPS is a software implementation of RSS that can be used on virtual network interfaces inside the OS kernel.

licious container raises security concerns. For example, if a malicious container circumvents *SlimSocket* and invokes the *getpeername* call directly on the host namespace file descriptor, it would be able to learn the IP addresses of the host machines. A container could also call *connect* with a host network IP address to create a connection directly on the host network, circumventing the overlay network. Finally, a container could call *setsockopt* to increase its traffic priority.

To enforce the same security model as in today's container overlay network, Slim offers a secure mode. When secure mode is on, Slim leverages a kernel module, *SlimKernModule*, to restrict the power of host namespace file descriptors inside containers. *SlimKernModule* implements a lightweight capability system for file descriptors. *SlimKernModule* has three roles: (1) track file descriptors as they propagate inside the container, (2) revoke file descriptors upon request from *SlimRouter*, and (3) prohibit a list of unsafe system calls using these file descriptors (e.g., *getpeername*, *connect*, *setsockopt*). *SlimSocket* emulates these system calls for non-malicious applications.

4 Slim

We first describe how to implement network virtualization without needing packet transformations in the data-plane while maintaining compatibility with current containerized applications. We then describe how to support flexible network polices and enforce security for malicious containers.

Slim does not change how virtual to physical IP mappings are stored. They can still be either stored in external storage or obtained through gossiping. As with today's container overlay network, Slim relies on a consistent and current view of containers' locations in the host network.

4.1 Connection-based Network Virtualization

Slim provides a connection-based network virtualization for containers. When a container is initiated on the host, Slim dispatches an instance of *SlimRouter* in the host namespace. Slim links a user-level shim layer, *SlimSocket*, to the container. When the process inside the container creates a connection, instead of making standard socket calls, *SlimSocket* sends a request to *SlimRouter* with the destination IP address and port number. *SlimRouter* creates a connection on behalf of the container and returns a host namespace file descriptor back to the container. We first present an example that shows how Slim supports traditional blocking I/O. We then describe how to additionally make Slim support non-blocking I/O.

Support for blocking I/O. Figure 5 shows how a TCP connection is created between a web client and a web server on Slim. Consider the web server side. The container first creates a socket object with the *socket* function call. This call is intercepted by *SlimSocket* and forwarded to *SlimRouter*, which creates a socket object in the host network. When the container calls *bind* on the socket with virtual network interface IP address 10.0.0.1 and port 80, *SlimRouter* also calls *bind* on the host network interface IP address 1.2.3.5 and with some unused port 1234. The port translation is needed because a host can run multiple web servers binding on port 80, but the host network interface only has a single port 80. *SlimRouter* updates the port mapping. The web server then uses *accept* to wait for an incoming TCP connection. This function call is also forwarded to *SlimRouter*, which waits on the host socket.

We move next to the web client side. The client performs similar steps to create the socket object. When the client side connects the overlay socket to the server side at IP address 10.0.0.1 port 80, *SlimRouter* looks up the virtual IP address 10.0.0.1 and finds its corresponding host IP address 1.2.3.5. *SlimRouter* then contacts the SlimRouter for the destination container on 1.2.3.5 to locate the corresponding host port, 1234. *SlimRouter* sets up a direct connection to port 1234 on 1.2.3.5. After the TCP handshake is complete, *accept*/*connect* returns a file descriptor in which socket send/recv is enabled. *SlimRouter* passes the file descriptor back to the container, and *SlimSocket* replaces the overlay connection file descriptor with the host namespace file descriptor using system call *dup2*. From this point on, the application directly uses the host namespace file descriptor to send or receive packets.

To ensure compatibility with current containerized applications, *SlimSocket* exposes the same POSIX socket interface. Besides forwarding most socket-related system calls (e.g., *socket*, *bind*, *accept*, *connect*) to *SlimRouter*, *SlimSocket* also carefully maintains the expected POSIX socket semantics. For example, when a containerized application calls *getpeername* to get an IP address on the other side of the connection, *SlimSocket* returns the overlay IP address rather than the host IP address, even when the file descriptor for the overlay connection has already been replaced with the host namespace file descriptor.

Support for non-blocking I/O. Most of today's applications [32, 36] use a non-blocking I/O API (e.g., *select*, *epoll*) to achieve high I/O performance. Slim must also intercept these calls because they interact with the socket interface. For example, *epoll* creates a meta file descriptor that denotes a set of file descriptors. An application uses *epoll_wait* to wait any event in the set, eliminating the need to create a separate thread to wait on an event in each file descriptor. On connection setup, we must change the corresponding file descriptor inside the *epoll*'s file descriptor set. *SlimSocket* keeps track of the mapping between the *epoll* file descriptor and epoll's set of file descriptors by intercepting *epoll_ctl*. For an *accept* or *connect* on a file descriptor that is inside an *epoll* file descriptor set, *SlimSocket* removes the original overlay network file descriptor from the *epoll* file descriptor set and adds host namespace file descriptor into the set.

Service discovery. Our example in Figure 5 assumes that the *SlimRouter* on the client side knows the server side has bound to physical IP 1.2.3.4 and port 1234. To automatically

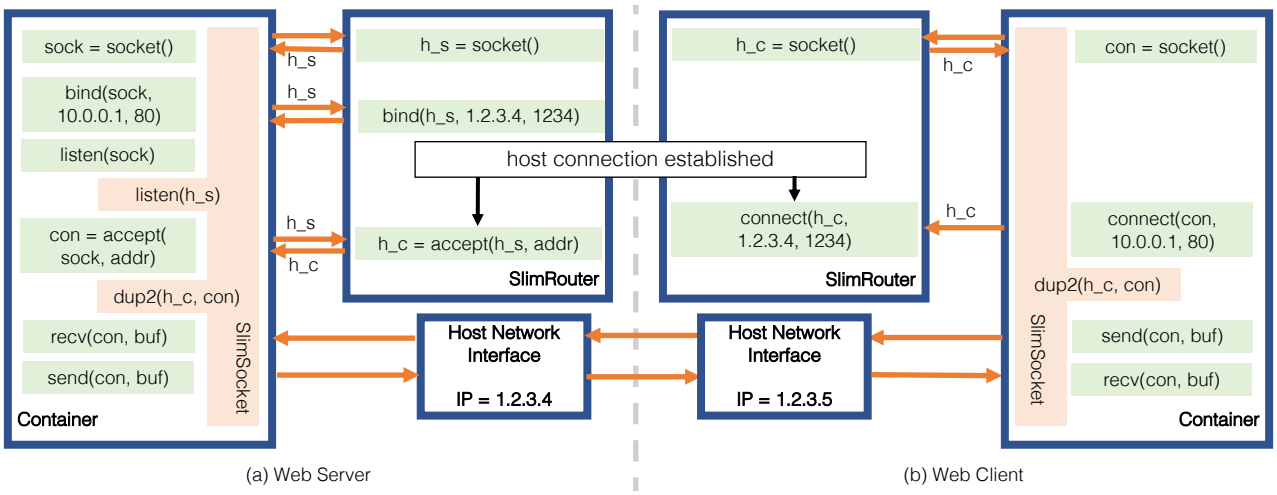


Figure 5: *TCP connection setup between a web client and a web server atop Slim.*

discover the server's physical IP address and port, we could store a mapping from virtual IP/port to physical IP/port on every node in the virtual network. Unfortunately, this mapping has to change whenever a new connection is listened.

Instead, Slim uses a distributed mechanism for service discovery. Slim keeps a standard container overlay network running in the background. When the client calls *connect*, it actually creates an overlay network connection on the standard container overlay network. When the server receives an incoming connection on the standard overlay network, *SlimSocket* queries *SlimRouter* for the physical IP address and port and sends them to the client side inside the overlay connection. In secure mode (§4.3), the result queried from *SlimRouter* is encrypted. *SlimSocket* on the client side sends the physical IP address and port (encrypted if in secure mode) to its *SlimRouter* and the *SlimRouter* establishes the host connection. This means connection setup time is longer in Slim than that on container overlay networks based on packet transformation. (See §6.1.)

4.2 Supporting Flexible Network Policies

This section describes Slim's support for both control- and data-plane policies.

Control-plane policies. Slim supports standard access control over overlay packet header fields, such as the source/destination IP addresses and ports. Access control can also filter specific types of traffic (e.g., SSH, FTP) or traffic from specific IP prefixes.

In the normal case where policies are static, Slim enforces access control at connection creation. *SlimRouter* maintains a copy of current access control policies from the container orchestrator or network operator. When a connection is created by *accept* or *connect*, *SlimRouter* checks whether the created connection violates any existing access control policy. If so, *SlimRouter* rejects the connection by returning -1 to *connect* or by ignoring the connection in *accept*.

Access control policies can change dynamically, and any connection in violation of the updated access control policy must be aborted. *SlimRouter* keeps per-connection state, including source and destination IP addresses, ports, and the corresponding host namespace file descriptors. When access control policies change, *SlimRouter* iterates through all current connections to find connections that are forbidden in the updated policies. *SlimRouter* aborts those connections by removing the corresponding file descriptors from the container. Removing a file descriptor from a running process is not an existing feature in commodity operating systems such as Linux. We build this functionality in *SlimKernModule*. (See §4.3 for more details.)

Data-plane policies. Slim supports two types of data-plane policies: rate limiting and quality of service (QoS). Rate limiting limits the amount of resources that a container can use. QoS ensures that the performance of certain applications is favored over other applications.

Slim reuses an OS kernel's existing features to support data-plane policies. A modern OS kernel has support for rate limiting and QoS for a single connection or a set of connections. Slim simply sets up the correct identifier to let the OS kernel recognize the container that generates the traffic.

In Slim, rate limits are enforced both at the per-connection and per-container level. Per-connection rate limits are set in a similar way as in today's overlay network using Linux's traffic control program, *tc*. For per-container rate limits, Slim first configures the *net_cls* cgroups to include the *SlimRouter* process. The *net_cls* cgroup tags traffic from the container or the corresponding *SlimRouter* with a unique identifier. *SlimRouter* then sets the rate limit for traffic with this identifier using *tc* on the host network interface. In this way, the network usage by *SlimRouter* is also restricted by the rate limit. Correct accounting of network usage is the fundamental reason why each container requires a separate *SlimRouter*.

Quality of service (QoS) also uses *tc*. *SlimRouter* uses socket options to set up the type of service (ToS) field (via

setsockopt). In this way, switches/routers on the physical network are notified of the priority of the container's traffic.

Compatibility with existing IT tools. In general, IT tools[3] need to be modified to interact with *SlimRouter* in order to function with Slim. IT tools usually use some user-kernel interface (e.g., iptables) to inject firewall and rate limits rules. When working with Slim, they should instead inject these rules to *SlimRouter*. Because Slim is fundamentally a connection-based virtualization approach, a limitation of our approach is that it cannot support packet-based network policy (e.g., drop an overlay packet if the hash of the packet matches a signature). (See §7.) If packet-based policies are needed, the standard Linux overlay should be used instead.

If static connection-based access control is the only network policy needed, then existing IT tools need not be modified. If an IT tool blocks a connection on a standard container overlay network, it also blocks the metadata for service discovery for that connection on Slim, thus it blocks the host connection from being created on Slim.

4.3 Addressing Security Concerns

Slim includes an optional kernel module, *SlimKernModule*, to ensure that Slim maintains the same security model as today's container overlay networks. The issue concerns potentially malicious containers that want to circumvent *SlimSocket*. Slim exposes host namespace file descriptors to containers and therefore needs an extra mechanism inside the OS kernel to track and manage access.

SlimKernModule implements a lightweight and general capability system based on file descriptors. *SlimKernModule* tracks tagged file descriptors in a similar way as taint-tracking tools [12] and filters unsafe system calls on these file descriptors. We envision this kernel module could also be used by other systems to track and control file descriptors. For example, a file server might want to revoke access from a suspicious process if it triggers an alert. Slim cannot use existing kernel features like seccomp [50] because seccomp cannot track tagged file descriptors.

SlimKernModule monitors how host namespace file descriptors propagate inside containers. It lets *SlimRouter* or other privileged processes tag a file descriptor. It then interposes on system calls that may copy or remove tagged file descriptors, such as *dup*, *fork* and *close*—to track their propagation. If the container passes the file descriptor to other processes inside the container, the tag is also copied.

Tagged file descriptors have limited powers within a container. *SlimKernModule* disallows invocation of certain unsafe system calls using these file descriptors. For example, in the case of Slim, a tagged file descriptor cannot be used with the following system calls: *connect*, *bind*, *getsockname*, *getpeername*, *setsockopt*, etc. This prevents containers from learning their host IP addresses or increasing their traffic priority. It also prevents containers from directly creating a host network connection. For a non-malicious container, *SlimSocket* and *SlimRouter* emulate the functionalities of these forbidden system calls.

SlimKernModule revokes tagged file descriptors upon request. To do so, it needs a process identifier (pid) and a file descriptor index. *SlimRouter* uses this functionality to implement dynamic access control. When the access control list changes for existing connections, *SlimRouter* removes the file descriptors through *SlimKernModule*. *SlimKernModule* revokes all the copies of the file descriptors.

Secure versus Non-secure mode. Whether to use Slim in secure mode (with *SlimKernModule*) or not depends on the use case. When containers and the physical infrastructure are under the same entity's control, such as for a cloud provider's own use [28], non-secure mode is sufficient. Non-secure mode is easier to deploy because it does not need kernel modification. When containers are potentially malicious to the physical infrastructure or containers of other entities, secure mode is required. Secure mode has slightly (∼25%) longer connection setup time, making the overall connection setup time 106% longer than that of a traditional container overlay network. (See §6.1.)

5 Implementation

Our implementation of Slim is based on Linux and Docker. Our prototype includes all features described in §4. *SlimSocket*, *SlimRouter*, and *SlimKernModule* are implemented in 1184 lines of C, 1196 lines of C++ (excluding standard libraries), and 1438 lines of C, respectively.

Our prototype relies on a standard overlay network, Weave [52], for service discovery and packets that require data-plane handling (e.g., ICMP, UDP).

SlimSocket uses LD_PRELOAD to dynamically link to the application binary. Communication between *SlimSocket* and *SlimRouter* is via a Unix Domain Socket. In non-secure mode, file descriptors are passed between *SlimRouter* and *SlimSocket* by *sendmsg*. For secure mode, file descriptors are passed with *SlimKernModule*'s cross-process file descriptor duplication method.

SlimRouter allows an network operator to express the access control as a list of entries based on source/destination IP address prefixes and ports in a JSON file. *SlimRouter* has a command-line interface for network operators to issue changes in the access control list via reloading the JSON file. Slim rejects any connection matched in the list. *SlimRouter* uses *htb* qdisc to implement rate limits and *prio* qdisc for QoS with *tc*.

SlimRouter and *SlimKernModule* communicate via a dummy file in *procfs* [46] created by *SlimKernModule*. *SlimKernModule* treats *write*s to this file as requests. Accessing the dummy file requires host root privilege.

[3] We only consider IT tools that run on the host to manage containers but not those run inside containers. IT tools usually require root privilege to the kernel (e.g., iptables) and are thus disabled inside containers.

SlimKernModule interposes on system calls by replacing function pointers in the system call table. *SlimKernModule* stores tagged file descriptors in a hash table and a list of unsafe system calls. *SlimKernModule* rejects unsafe system calls on tagged file descriptors.

SlimKernModule also interposes on system calls such as *dup*, *dup2* and *close* to ensure that file descriptor tags are appropriately propagated. For process fork (e.g., *fork*, *vfork*, *clone* in Linux kernel), *SlimKernModule* uses the *sched_process_fork* as a callback function. Slim does not change the behavior of process forking. A forked process still has *SlimSocket* dynamically linked.

6 Evaluation

We first microbenchmark Slim's performance and CPU utilization in both secure and non-secure mode and then with four popular containerized applications: an in-memory key-value store, Memcached [32]; a web server, Nginx [36]; a database, PostgreSQL [45]; and a stream processing framework, Apache Kafka [1, 26]. Finally, we show performance results for container migration. Our testbed setup is the same as that for our measurement study (§2.2). In all the experiments, we compare Slim with Weave [52] with its fast data-plane enabled and with RFS enabled by default. We use Docker [6] to create containers.

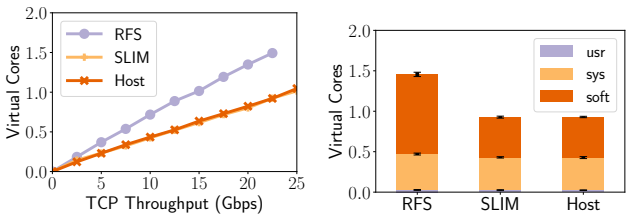
6.1 Microbenchmarks

Similar to the performance tests in §2.2, we use *iperf3* [19] and *NPtcp* [39] to measure performance of a TCP flow. We use *mpstat* to measure CPU utilization.

A single TCP flow running on our 40 Gbps testbed reaches 26.8 Gbps with 11.4 µs latency in both secure and non-secure modes. Slim's throughput is the same as the throughput on the host network and is 9% faster than Weave with RFS. Slim's latency is also the same as using the host network, and it is 86% faster than Weave with RFS.

Using Slim, the creation of a TCP connection takes longer because of the need to invoke the user-space router. On our testbed, in a container with Weave, creating a TCP connection takes 270 µs. With the non-secure mode of Slim, it takes 444 µs. With the secure mode, it takes 556 µs. As a reference, creation of a TCP connection on the host network takes 58 µs. This means that Slim is not always better, e.g., if an application has many short-lived TCP connections. We did not observe this effect in the four applications studied because they support persistent connections [35, 37], a common design paradigm.

For long-lived connections, Slim reduces CPU utilization. We measure the CPU utilization using *mpstat* for Slim in secure mode and Weave with RFS when varying TCP throughput from 0 to 25 Gbps. RFS cannot reach 25 Gbps, so we omit that data point. Figure 6a shows the total CPU utilization in terms of number of virtual cores consumed. Compared to RFS, CPU overhead declines by 22-41% for Slim;



(a) *Total CPU Utilization* (b) *Breakdown*

Figure 6: *CPU utilization and breakdown for a TCP connection. In Figure 6a, the Slim and the host lines overlap. In Figure 6b, the usr bar is at the bottom and negligible. Error bars denote standard deviations.*

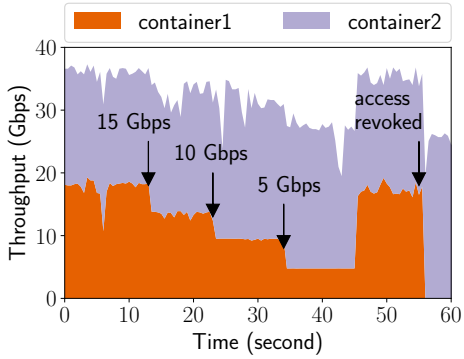
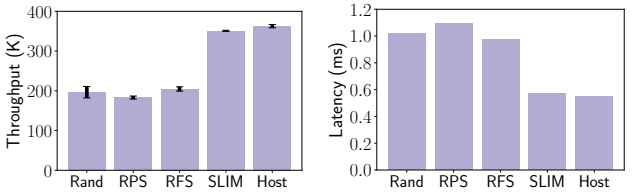


Figure 7: *A bar graph of the combined throughput of two Slim containers, with rate limit and access control policy updates to one of the containers.*

Slim's CPU costs are the same as using the host network directly. To determine the source of this reduction, we break down different components using *mpstat* when TCP throughput is 22.5 Gbps. Figure 6b shows the result. As expected, the largest reduction in CPU costs comes from serving software interrupts. These decline 49%: Using Slim, a packet no longer needs data-plane transformations and traverses the host OS kernel's network stack only once.

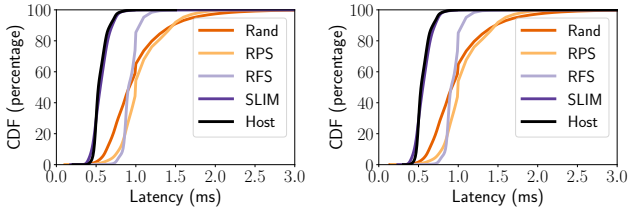
Network policies. Slim supports access control, rate limiting and QoS policies, including when applied to existing connections. We examine a set of example scenarios when rate limits and access control are used. We run two parallel containers, each with a TCP connection. The other end of those connections is a container on a different machine. We use *iperf* to report average throughput per half second. Figure 7 shows the result.

Without network policy, each container gets around 18-18.5 Gbps. We first set a rate limit of 15 Gbps on one container. The container's throughput immediately drops to around 14 Gbps, and the other container's throughput increases. A slight mismatch occurs between the rate limit we set and our observed throughput, which we suspect is due to *tc* being imperfect. We subsequently set the rate limit to 10 Gbps and 5 Gbps. As expected, the container's throughput declines to 10 and 5 Gbps, respectively, while the other con-



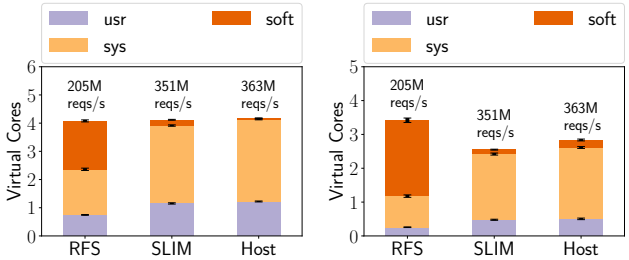
(a) *Throughput* (b) *Latency*

Figure 8: *Throughput and latency of Memcached with Weave (in various configurations) and with Slim. Error bars in Figure 8a shows the standard deviation of completed Memcached operations per-second.*



(a) *GET* (b) *SET*

Figure 9: *Distribution of latency for Memcached SET and GET operations, illustrating tail latency effects. The Slim and Host lines overlap.*



(a) *Client* (b) *Server*

Figure 10: *CPU utilization of Memcached client and server. Error bars denote standard deviations.*

tainer's throughput increases. Finally, Slim stops rate limiting and sets an ACL to bar the container from communicating with the destination. The affected connection is destroyed, and the connection from the other container speeds up to standard connection speed.

6.2 Applications

We evaluate Slim with four real world applications: Memcached, Nginx, PostgreSQL, and Apache Kafka. From this point on, our evaluation uses Slim running in secure mode.

6.2.1 Memcached

We measure the performance of Memcached [32] on Slim. We create one container on each of the two physical machines; one runs the Memcached (v1.5.6) server, and the other container runs a standard Memcached benchmark tool, *memtier_benchmark* [33] developed by redislab [49]. The benchmark tool spawns 4 threads. Each thread creates 50 TCP connections to the Memcached server and reports the average number of responses per second, the average latency to respond to a memcache command, and the distribution of response latency (SET:GET ratio = 1:10).

Slim improves Memcached throughput (relative to Weave). Figure 8a shows the number of total Memcached operations per-second completed on Slim and Weave with different configurations. Receive Flow Steering (RFS) is our best-tuned configuration, yet Slim still outperforms it by 71%. With the default overlay network setting (random IRQ load balancing), Slim outperforms Weave by 79%. Slim's throughput is within 3% of host mode.

Slim also reduces Memcached latency. Figure 8b shows the average latency to complete a memcache operation. The average latency reduces by 42% using Slim compared to RFS. The latency of the default setting (random IRQ load balancing), RPS, and RFS are not significantly different (within 5%). Slim's latency is exactly the same as host mode.

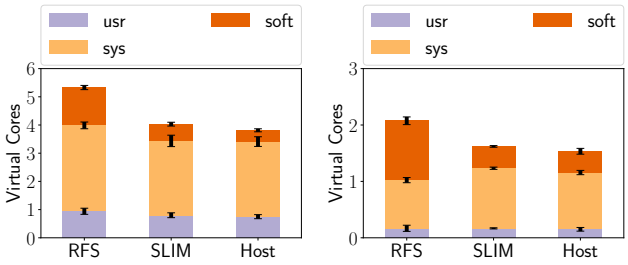
Slim also reduces Memcached tail latency. Figure 9 shows the CDF of latency for SET and GET operations. The default configuration (i.e., IRQ load balancing) has the worst tail latency behavior. Synchronization overhead depends on temporal kernel state and thus makes latency less predictable. RPS and RFS partially remove the synchronization overhead, improving predictability. Compared to the best configuration, RFS, Slim reduces the 99.9% tail latency by 41%.

Slim reduces the CPU utilization per operation. We measure average CPU utilization on both the client and the Memcached server when *memtier_benchmark* is running. Figure 10 shows the result. The total CPU utilization is similar on the client side, while the utilization is 25% lower with Slim on the server compared to RFS. Remember that Slim performs 71% more operations/second. As expected, the amount of CPU utilization in serving software interrupts declines in Slim. We also compare CPU utilization when the throughput is constrained to be identical. Slim reduces CPU utilization by 41% on the Memcached client and 56% on the Memcached server, relative to Weave.

6.2.2 Nginx

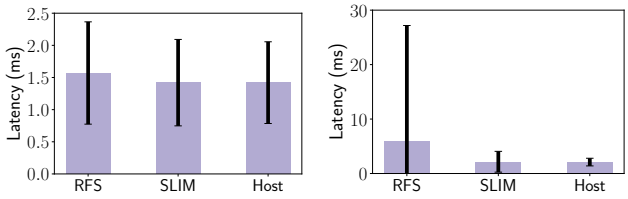
We run one Nginx (v1.10.3) server in one container and a standard web server benchmarking tool, *wrk2* [53], in another container. Both containers are on two different physical machines. The tool, *wrk2*, spawns 2 threads to create a total of 100 connections to the Nginx server to request an HTML file. This tool lets us set throughput (requests/sec), and it outputs latency. We set up two HTML files (1KB, 1MB) on the Nginx server.

Nginx server's CPU utilization is significantly reduced with Slim. We use *mpstat* to break down the CPU utilization of the Nginx server for scenarios when RFS, Slim, and host



(a) *1KB, 60K reqs/s* (b) *1MB, 3K reqs/s*

Figure 11: *CPU utilization breakdown of Nginx. Error bars denote standard deviations.*



(a) *1KB, 60K reqs/s* (b) *1MB, 3K reqs/s*

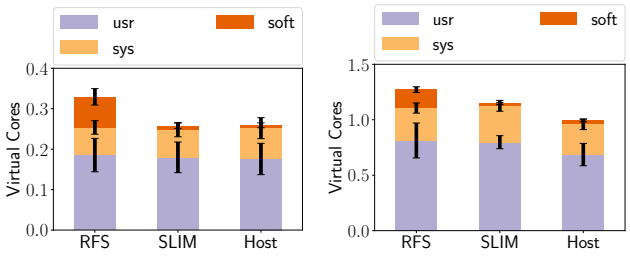
Figure 12: *Latency of Nginx server. Error bars denote standard deviations.*

can serve the throughput. Figure 11 shows the CPU utilization breakdown when the file size is 1KB and the throughput is 60K reqs/second, and also when the file size is 1MB and the throughput is 3K reqs/second. (We choose 60K reqs/second and 3K reqs/second because they are close to the limit of what RFS can handle.). For the 1 KB file, the CPU utilization reduction is 24% compared with RFS. For the 1 MB file, the CPU utilization reduction is 22% compared with RFS. Note that much of the CPU utilization reduction comes from reduced CPU cycles spent in serving software interrupts in the kernel. The CPU utilization still has a 5%-6% gap between Slim and host. We expect this gap is from the longer connection setup time in Slim. Unlike our Memcached benchmark, where connections are pre-established, we observe that *wrk2* creates TCP connections on the fly to send HTTP requests.

While Slim improves the CPU utilization, the improvements to latency are lost in the noise of the natural variance in latency for Nginx. The benchmark tool, *wrk2*, reports the average and the standard deviation of Nginx's latency. Figure 12 shows the result. The standard deviation is much larger than the difference of the average latencies.

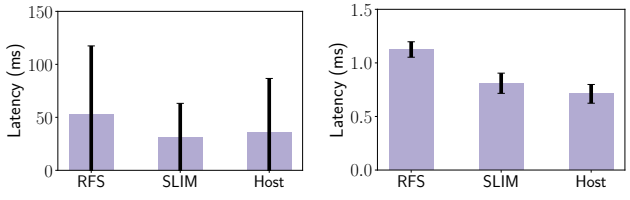
6.2.3 PostgreSQL

We deploy a relational database, PostgreSQL [45] (version 9.5), in a container and then use its default benchmark tool, *pgbench* [43], to benchmark its performance in another container. The tool, *pgbench*, implements the standard TPC-B benchmark. It creates a database with 1 million banking accounts and executes transactions with 4 threads and a total of



(a) *PostgreSQL* (b) *Kafka*

Figure 13: *CPU utilization of PostgreSQL and Kafka. Error bars denote standard deviations.*



(a) *PostgreSQL* (b) *Kafka*

Figure 14: *Latency of PostgreSQL and Kafka. Error bars denote standard deviations.*

100 connections.

Slim reduces the CPU utilization of PostgreSQL server. We set up *pgbench* to generate 300 transactions per second. (We choose 300 transactions per second because it is close to what RFS can handle.) Figure 13a shows the CPU utilization breakdown of the PostgreSQL server. Compared with RFS, Slim reduces the CPU utilization by 22% and the CPU utilization is exactly the same as using host mode networking. Note here, the reduction in CPU utilization is much less than in Memcached and Nginx. The reason is that the PostgreSQL server spends a larger fraction of its CPU cycles in user space, processing SQL queries. Thus, the fraction of CPU cycles consumed by the overlay network is less.

Similar to Nginx, the latency of PostgreSQL naturally has a high variance because of the involvement of disk operations, and it is difficult to conclude any latency benefit of Slim. The benchmark tool, *pgbench*, reports the average and the standard deviation of PostgreSQL's latency. Figure 14a shows the results. The standard deviation is much larger than the difference of the mean latencies.

6.2.4 Apache Kafka

We now evaluate a popular data streaming framework, Apache Kafka [1, 26]. It is used to build real-time data processing applications. We run Kafka (version 2.0.0) inside one container, and Kafka's default benchmarking tool, *kafka-producer-perf-test*, in another container on a different physical machine.

Slim reduces the CPU utilization of both the Kafka server and the Kafka client. We use *kafka-producer-perf-test* to set

	Weave	Slim
Stop running container	0.75 ± 0.02	0.75 ± 0.02
Extract fs into image	0.43 ± 0.01	0.43 ± 0.01
Transfer container image	0.44 ± 0.01	0.44 ± 0.01
Restore fs	0.82 ± 0.09	1.20 ± 0.10
Start *SlimRouter*	-	0.003 ± 0.001
Start container	0.90 ± 0.09	0.94 ± 0.17
Total	3.34 ± 0.12 s	3.76 ± 0.20 s

Table 4: *Time to migrate a Memcached container. The numbers followed by* $\pm$ *show the standard deviations.*

throughput to be 500K messages per second. (We choose 500K messages per second because it is close to what RFS can handle.) Each message is 100 bytes and the batch size is 8192. The tool spawns 10 threads that generate messages in parallel. Figure 13b shows the breakdown of CPU utilization. The total CPU utilization of the Kafka server reduces by 10% with Slim. The CPU utilization reduction is even smaller than PostgreSQL because Kafka spends more time in user space processing.

Slim reduces message latencies in Kafka. The benchmark tool, *kafka-producer-perf-test*, reports the latency of Kafka. Figure 14b shows the results. Kafka's latency reduces by 0.28 ms (28%), compared with RFS. There is still a 0.09 ms latency gap between using Slim and the host mode.

6.3 Container Migration

Slim supports container migration. On our testbed, we migrate a Memcached container from one physical server to another physical server on the 40 Gbps network. We test migration 20 times with/without Slim. The container's IP address is kept the same across the migration. Likewise, we do not change the host network's routing table. The container image extracted from the file system is 58 Mbytes.

Using Slim marginally slows down container migration. Table 4 is the breakdown of the average container migration time on Weave and on Slim. In total, Slim slows down container migration from 3.34 s to 3.76 s. Slim does not change most of the migration process. The extra overhead is introduced mainly in restoring the file system. With Slim, a container has an additional disk volume containing *SlimSocket* and also a dummy file to support communication over UNIX domain socket between *SlimSocket* and *SlimRouter*. We suspect that the additional disk volume slows down the file system restoration process. Further, starting a container with Slim adds a small amount of additional overhead.

7 Discussion

Connection Setup Time. One drawback of Slim is that connection setup time is significantly longer (§6.1). This can penalize applications with many short connections. Slim allows indiviual applications in a container to opt out by detaching *SlimSocket*. In the future, we want the choice of opting out to be at a per-connection level. We can either (1) allow developers to specify which connection to opt out, or (2) automatically opt out based on predicted flow sizes [10].

Container Live Migration. Although Slim does support quiescent container migration, it does not currently support container live migration. All the TCP connections are disconnected during the migration process, and memory states are not migrated. However, in live migration, live application state has to be fully restored, including state such as application threads waiting on events inside the OS kernel. Docker is currently experimenting with live checkpointing and restoration with *criu* [4], but it is focused on the simpler case of a single host [7]. Provided a practical live container migration system could be built, Slim would make that more difficult because: (1) the state of the container now includes host namespace file descriptors and (2) data-plane policies (e.g., rate limits) are enforced on host connection identifiers (i.e., five tuples) that would need to be properly translated when migrated.

UDP. The focus of this paper has been on improving the container communication performance of connection-oriented protocols, such as TCP, by moving operations from the data-plane to connection setup. This poses a challenge for connectionless protocols such as UDP. Slim potentially could support UDP using similar mechanisms as for TCP, by intercepting *socket*, *bind*, *connect*, *sendto*, and *recvfrom*. However, we chose not to do this in our prototype because of two reasons. First, we do not have a good mechanism to support flexible network policy for UDP. In UDP, a file descriptor does not describe a single network pair, but rather an open port to which every node in the virtual network can send packets. Second, the most common use case for UDP in data centers is to avoid the overhead of connection setup; since Slim makes connection setup more expensive, it would subvert some of those benefits. Instead, to work with unmodified applications that may use a mixture of TCP and UDP packets, our prototype simply directs UDP traffic to Weave.

Packet-based Network Policy. A limitation of Slim is that it supports connection-based network policy and not packet-based network policy. For example, a virtual network can be set up to prevent access to a backend database, except from certain containers; Slim supports this kind of access control. Packet-based filters allow the system drop packets if the hash of the overlay packet matches a signature. On Slim, the virtual overlay packet is never constructed and so checking against a signature would be prohibitively expensive. If packet-based network policy is needed, a standard overlay network should be used instead.

LD_PRELOAD. Our prototype uses LD_PRELOAD to dynamically link *SlimSocket* into unmodified application binaries. Some systems assume statically linked application binaries (e.g., applications written in Go). These can benefit from Slim by patching the application binaries to use *SlimSocket* instead of POSIX sockets; we do not implement this

support in our prototype.

Error Code. Our current prototype implementation is not transparent in one significant way. When an access control list changes, requiring Slim to revoke a file descriptor, the application receives a different error code when it used that file descriptor, relative to Weave. In Slim, a send on a revoked file descriptor returns a bad file descriptor error code, while in Weave the packet would be silently discarded. We believe it is possible to address this but it was not needed for our benchmark applications.

SmartNICs. A recent research trend has been to explore moving common case network data-plane operations to hardware. Catapult [48], for example, moves packet encapsulation required for virtual machine emulation to hardware. Catapult runs as a bump on the wire, however, so in order to offload overlay network processing, Linux would need to be modified to accept virtual network packets on its physical network interface. SR-IOV is commodity hardware, but it suffers from the same problem as macvlan mode. (See §2.1.) FlexNIC [22] has proposed a flexible model that can incorporate application, guest OS, and virtual machine packet management, but to date it is only experimental hardware. While new hardware support is likely to become increasingly available, what we have shown is that such hardware support is not necessary for efficient virtual overlay networks for containers; the container OS has all the information it needs to perform virtualization at connection setup.

8 Related Work

Network namespace. Mapping resources from a host into a container is not a new idea. In Plan9 [44], resources, such as directories in the file system or process identifiers, are directly mapped between namespaces. Our work revisits Plan9's idea in the networking context, but with performance as a goal, rather than portability. Today's Linux network namespace works at a per-device level, and so is not strong enough for supporting connection-based network virtualization. Slim uses the Linux networking namespace to isolate the container from using the host network interface.

Host support for efficient virtual networking. Host support for efficient virtual networking is an old topic, mostly in the context of VMs. Menon et al. co-design the driver of the virtual network interface and the hypervisor for efficient virtual network interface emulation [34]. Socket-outsourcing [11], VMCI socket [51], and Slipstream [5] improve intra-host networking. FreeFlow [23] redirects RDMA library calls to create a fast container RDMA network. To the best of our knowledge, Slim is first work that provides network virtualization at TCP connection setup time for unmodified containerized applications.

Redirecting system calls. Redirecting system calls is a useful technique for many purposes, such as taint tracking [12], building user-level file systems [16] and performing other advanced OS kernel features (e.g., sandboxing [24], record and replay [17], and intrusion detection [25]). In a networking context, mTCP [21] redirects socket calls to construct a user-level networking stack. NetKernel [38] redirects socket calls to decouple networking stack from virtual machine images.

Separation of control- and data-plane. The performance gain of Slim comes from moving network virtualization logic from the data- to the control-plane. Separation of the control- and the data-plane is a well-known technique to improve system performance in building fast data-plane operating systems [41, 2] and routing in flexible networks [31].

9 Conclusion

Containers have become the de facto method for hosting distributed applications. The key component for providing portability, the container overlay network, imposes significant overhead in terms of throughput, latency, and CPU utilizations, because it adds a layer to the data-plane. We propose Slim, a low-overhead container overlay network that implements network virtualization by manipulating connection-level metadata. Slim transparently supports unmodified, potentially malicious, applications. Slim improves throughput of an in-memory key-value store by 71% and reduces latency by 42%. Slim reduces CPU utilization of the in-memory key-value store by 56%, a web server by 22%-24%, a database server by 22%, and a stream processing framework by 10%. Slim's source code is publicly available at https://github.com/danyangz/slim.

Acknowledgments

We thank Antoine Kaufmann, Jialin Li, Ming Liu, Jitu Padhye, and Xi Wang for their feedback on earlier versions of the paper. We thank our shepherd Jon Howell and the anonymous reviewers for their helpful feedback on the paper. This work was partially supported by the National Science Foundation (CNS-1518702 and CNS-1616774) and by gifts from Google, Facebook, and Huawei.

References

[1] Apache Kafka. https://kafka.apache.org/.

[2] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *OSDI* (2014).

[3] Calico. https://www.projectcalico.org/.

[4] CRIU. https://criu.org/Main_Page.

[5] Dietz, W., Cranmer, J., Dautenhahn, N., and Adve, V. Slipstream: Automatic Interprocess Communication Optimization. In *USENIX ATC* (2015).

[6] Docker. http://www.docker.com.

[7] Docker Checkpoint and Restore. https://github.com/docker/cli/blob/master/experimental/checkpoint-restore.md.

[8] Docker container networking. https://docs.docker.com/engine/userguide/networking/.

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

[10] DUKIC, V., JYOTHI, S. A., KARLAS, B., OWAIDA, M., ZHANG, C., AND SINGLA, A. Is advance knowledge of flow sizes a plausible assumption? In *NSDI* (2019).

[11] EIRAKU, H., SHINJO, Y., PU, C., KOH, Y., AND KATO, K. Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments. In *ACM Symposium on Applied Computing* (2009).

[12] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *OSDI* (2010).

[13] etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/.

[14] FIRESTONE, D., PUTNAM, A., ANGEPAT, H., CHIOU, D., CAULFIELD, A., CHUNG, E., HUMPHREY, M., OVTCHAROV, K., PADHYE, J., BURGER, D., MALTZ, D., GREENBERG, A., MUNDKUR, S., DABAGH, A., ANDREWARTHA, M., BHANU, V., CHANDRAPPA, H. K., CHATURMOHTA, S., LAVIER, J., LAM, N., LIU, F., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., VAID, K., AND MALTZ, D. A. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *NSDI* (2018).

[15] Flannel. https://github.com/coreos/flannel.

[16] Filesystem in userspace. https://github.com/libfuse/libfuse.

[17] GUO, Z., WANG, X., TANG, J., LIU, X., XU, Z., WU, M., KAASHOEK, M. F., AND ZHANG, Z. R2: An Application-level Kernel for Record and Replay. In *OSDI* (2008).

[18] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *NSDI* (2011).

[19] iperf. https://iperf.fr/.

[20] iptables. https://linux.die.net/man/8/iptables.

[21] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *NSDI* (2014).

[22] KAUFMANN, A., PETER, S., SHARMA, N. K., ANDERSON, T., AND KRISHNAMURTHY, A. High Performance Packet Processing with FlexNIC. In *ASPLOS* (2016).

[23] KIM, D., YU, T., LIU, H. H., ZHU, Y., PADHYE, J., RAINDEL, S., GUO, C., SEKAR, V., AND SESHAN, S. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI* (2019).

[24] KIM, T., AND ZELDOVICH, N. Practical and effective sandboxing for non-root users. In *USENIX ATC* (2013).

[25] KING, S. T., AND CHEN, P. M. Backtracking Intrusions. In *SOSP* (2003).

[26] KREPS, J., NARKHEDE, N., AND RAO, J. Kafka: a Distributed Messaging System for Log Processing. In *NetDB* (2016).

[27] Kubernetes: Cluster Networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/.

[28] LIU, H. H., ZHU, Y., PADHYE, J., CAO, J., TALLAPRAGADA, S., LOPES, N. P., RYBALCHENKO, A., LU, G., AND YUAN, L. CrystalNet: Faithfully Emulating Large Production Networks. In *SOSP* (2017).

[29] Networking using a macvlan network. https://docs.docker.com/network/network-tutorial-macvlan/.

[30] MANCO, F., LUPU, C., SCHMIDT, F., MENDES, J., KUENZER, S., SATI, S., YASUKATA, K., RAICIU, C., AND HUICI, F. My VM is Lighter (and Safer) Than Your Container. In *SOSP* (2017).

[31] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR* (2008).

[32] Memcached. https://memcached.org/.

[33] memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[34] MENON, A., COX, A. L., AND ZWAENEPOEL, W. Optimizing Network Virtualization in Xen. In *USENIX ATC* (2006).

[35] MOGUL, J. C. The Case for Persistent-connection HTTP. In *SIGCOMM* (1995).

[36] Nginx. https://nginx.org/.

[37] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *NSDI* (2013).

[38] NIU, Z., XU, H., HAN, D., CHENG, P., XIONG, Y., CHEN, G., AND WINSTEIN, K. Network stack as a service in the cloud. In *HotNets* (2017).

[39] netpipe(1) - Linux man page. https://linux.die.net/man/1/netpipe.

[40] Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[41] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *OSDI* (2014).

[42] PFAFF, B., PETTIT, J., KOPONEN, T., JACKSON, E., ZHOU, A., RAJAHALME, J., GROSS, J., WANG, A., STRINGER, J., SHELAR, P., AMIDON, K., AND CASADO, M. The Design and Implementation of Open vSwitch. In *NSDI* (2015).

[43] pgbench. https://www.postgresql.org/docs/9.5/static/pgbench.html.

[44] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. The Use of Name Spaces in Plan 9. *SIGOPS OSR* (1993).

[45] PostgreSQL. https://www.postgresql.org/.

[46] proc - process information pseudo-filesystem. http://man7.org/linux/man-pages/man5/proc.5.html.

[47] Intel P-State driver. https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt.

[48] PUTNAM, A., CAULFIELD, A. M., CHUNG, E. S., CHIOU, D., CONSTANTINIDES, K., DEMME, J., ESMAEILZADEH, H., FOWERS, J., GOPAL, G. P., GRAY, J., HASELMAN, M., HAUCK, S., HEIL, S., HORMATI, A., KIM, J.-Y., LANKA, S., LARUS, J., PETERSON, E., POPE, S., SMITH, A., THONG, J., XIAO, P. Y., AND BURGER, D. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *ISCA* (2014).

[49] redislab. https://redislabs.com/.

[50] SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[51] VMCI Socket Performance. https://www.vmware.com/techpapers/2009/vmci-socket-performance-10075.html.

[52] Weave. https://www.weave.works/.

[53] wrk2. https://github.com/giltene/wrk2.