Slim and Fast: Low-Overhead Container Overlay Network With Fast Connection Setup

Fusheng Lin, Xin Zhang, Guo Chen, Li Chen, Kenli Li, *Senior Member, IEEE*, and Hongbo Jiang, *Senior Member, IEEE*

***Abstract*—Large-scale cloud applications today are often deployed using multiple containers, and a container overlay network is the de facto method to provide connectivity among these containers. However, the existing tunneling-based overlay network incurs significant performance overhead due to the need of transformation for every packet. Recent work *Slim*, through manipulating connection-level meta-data, allows containers to use host OS sockets directly thus they can achieve good performance without extra packet tunneling. Nevertheless, the connection setup is significantly slowed down, which requires an extra round-trip communication between both sides to pass the mapping information of the host OS socket and the container socket. This greatly hurts the performance of many cloud applications that must process short connections at high speed. We propose *SlimFast*, a low-overhead container overlay network which provides a fast connection setup. *SlimFast* directly uses the host OS socket for container communication as *Slim*. However, *SlimFast* needs no extra communication during connection setup. We reserve a dedicated host port for the container network and use socket mapping table to locally find the right container socket during connection setup. We implement *SlimFast* which is compatible with existing container applications. Experiments show that, *SlimFast* can improve the connection setup time by about 2.1x compared with *Slim*, meanwhile maintaining low-overhead during data transmission as *Slim*. This brings significant performance improvement to real applications. Particularly, testbed results show that *SlimFast* improves the throughput of Nginx proxy and Memcached by about 0.9x and 2.2x, respectively.**

***Index Terms*—Container, container overlay network, connection setup.**

I. Introduction

CONTAINERS, which offer lightweight isolation and portability, have now become the major way of managing, deploying and executing cloud applications [24], [25], [26], [27].

Manuscript received 15 September 2022; revised 20 April 2023; accepted 25 April 2023. Date of publication 2 June 2023; date of current version 8 March 2024. This work was supported in part by the National Natural Science Foundation of China under Grants 62222204 and 62172148, in part by the National Key Research and Development Program of China under Grant 2020YFB2104000, in part by the Natural Science Foundation of Hunan Province for Excellent Young Scholars under Grant 2021JJ20027, and in part by the Training Program for Excellent Young Innovators of Changsha. Recommended for acceptance by K. Gopalan. *(Fusheng Lin and Xin Zhang are co-first authors.) (Corresponding author: Guo Chen.)*

Fusheng Lin, Xin Zhang, Guo Chen, Kenli Li, and Hongbo Jiang are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410000, China (e-mail: linfusheng@hnu.edu.cn; zxzx2020@hnu.edu.cn; guochen@hnu.edu.cn; lkl@hnu.edu.cn; hongbojiang2004@gmail.com).

Li Chen is with Huawei, Longgang 518000, China (e-mail: crischenli@gmail.com).

Digital Object Identifier 10.1109/TCC.2023.3282238

For large-scaled applications, typically, they use a large number of containers to host various parts of their application logic, which collaborate together as a whole to serve certain tasks [20], [21]. This requires efficient and flexible network connectivity among containers possibly located on multiple machines in the data center.

Container overlay network [3], [17], [18], [19], is the de facto technique that provides such network connectivity between containers. In general, container overlay networks are based on tunneling approach (e.g., VxLAN) [6], that can provide a customized virtual network between containers on top of physical data center networks. Although offering flexible network connectivity regardless of existing physical infrastructure, such tunneling-based container overlay network incurs significant performance overhead, leading to much lower bandwidth, higher latency, and consuming more CPU resources compared to native host network, as observed by previous works [16], [22], [23].

Fundamentally, the performance overhead for tunneling-based container overlay network is due to the need of transformation for every packet, traversing the network stack twice (host stack and overlay stack) at both the sender and receiver sides. As such, recently *Slim* [23] tackles this problem by directly passing sockets in the host operating system (OS) to containers when container sockets are created. Such approach helps *Slim* to avoid extra packet transformation and keep the same performance as host sockets during data transmission. However, to maintain unchanged flexibility and application compatibility of original container overlay network, *Slim* has to manipulate connection-level metadata during connection setup. Particularly, in a flexible virtual network, upper-layer container sockets may bind to any virtual IP/port as they like. Therefore, before setup the connection, *Slim* requires an extra round-trip communication between the client and the server, so the client knows which under-layer host socket (i.e., which port[1]) on the server it should connect to thus to reach the target upper-layer container socket. This greatly lengthens the time for connection setup, which hurts the performance of many cloud applications that must process short connections at high speed [14], such as backend systems (e.g., memcached clusters) [13], middleboxes (e.g., SSL proxies [12] and redundancy elimination [11]) and serverless applications[2] [20], [21].

[1]In practice, the mapping of container IP to host IP is globally known [23]. Detailed background is introduced in Section II-B.

[2]Containers that host serverless functions come and go, so the connection lasting time are very short [15].

In this article, we ask whether we can *use host socket for container communication as Slim but need no extra communication during connection setup*, thus keeping low-overhead and fast in both data transmission and connection setup. This calls for an alternative way to find the mapping between container sockets and host sockets when the client tries to connect to the server. To address this problem, we present *SlimFast*. *SlimFast*'s key insight is that: The client does not try to find some certain target host socket. Instead, the client connects and establishes a connection to a dedicated and publicly known host socket (port) which is specifically reserved for container overlay network. Necessary information of the destined container is embedded when the client connects with the host socket. Since the server host knows all its local upper-layer container sockets (monitoring container socket `listen`), according to the embedded information, it can distribute the established host socket to the target container, and use it directly for later data transmission in the container. This saves an extra communication during connection setup.

Based on the above idea, we design and implement *SlimFast*, a low-overhead container network with fast connection setup. *SlimFast* is compatible with current Linux and Docker, which needs no modification to existing applications (Our implementation is open-sourced at [8]). Testbed experiments show that, *SlimFast* can improve the connection setup time by about 2.1x compared with *Slim*. Meanwhile, *SlimFast* also maintains low-overhead during data transmission as *Slim*, which can saturate 10 Gbps network for bulk data transfer with about 56% CPU utilization compared to the current tunneling-based overlay solution. Moreover, we have evaluated the performance of upper-layer applications using *SlimFast*. For intensive short-lived connection scenarios, *SlimFast* improves the throughput of Nginx proxy and Memcached by about 0.9x and 2.2x, respectively, compared with *Slim*. Meanwhile, for long-lived connection scenarios, *SlimFast* achieves about 22% and 92% higher throughput for Nginx proxy and Memcached, respectively, than the current tunneling-based overlay solution (which is also slightly higher than *Slim*). Furthermore, based on the idea of connection mapping, we extend the design of *SlimFast* to support connectionless protocol and packet-level security policy, which is a hard task in current low-overhead container overlay networks.

The major contributions of this article are summarized as follows:

- We observe the long connection setup time in the latest low-overhead container overlay network, and evaluate its impact on the performance of typical applications.
- We propose *SlimFast*, a low-overhead container network with fast connection setup, which uses reserved server port and local socket mapping table to remove extra round-trip communication during connection setup.
- *SlimFast* also avoids the overlay protocol stack encapsulation overhead of connectionless protocol and is compatible with existing container overlay network security policies, with the help of extended socket matching table.
- We thoroughly evaluate *SlimFast*'s performance using dedicated micro-benchmarks as well as real applications including Nginx and Memcached. We show that *SlimFast* is fast during connection setup time and keeps low-overhead during data transmission, which can greatly improve the performance of upper-layer applications.

II. Container Overlay Network and the Problem

In this section, we first describe how container network works and why current tunnel-based overlay network is inefficient. Next, we show how *Slim* improves the efficiency of container overlay network in data transmission, but sacrifices the efficiency of connection setup. Finally, we show that fast connection setup is important to container applications and how slow connection setup can affect the performance of 2 real-world applications.

A. Current Tunnel-Based Overlay Network

For container communication, there are mainly 4 options: bridge mode, host mode, macvlan mode and overlay mode. Bridge mode creates a virtual bridge in the host. Containers connect to the virtual bridge via veth (Virtual Ethernet Device) [29]. When a container in bridge mode communicates with the container of other hosts, the container IP address is translated to host IP address according to NAT rules, and then forward to NIC through veth pairs; In Macvlan mode, hosts map container NIC to virtual NIC of a host using mapping relationships. When containers communicate with containers on other hosts, routing problems caused by a large number of containers need to be considered; In host mode, the container abandons isolation and uses the host network stack and NIC, that is, the host's IP address and unallocated port. The container does not have a separate IP address, which makes it difficult for the data center to manage the container, and the ports available to the container will be limited by all the containers on the host. Host mode and Macvlan mode enables containers in different hosts to communicate but they complicate management in datacenters. In today's data centers, container overlay network (i.e., overlay mode) [3], [17], [18], [19] has become the de facto method for cross-host container communication as it provides flexibility for containers.

The current overlay network provides a network isolation for containers through tunneling. Each container has their own IP address, virtual network interface, IP routing tables, firewall rules, etc.. Unlike the host mode or macvlan mode, in overlay mode, the container's IP is not dependent on the host's, allowing better flexibility for containers. In overlay network, containers connect to a virtual switch (e.g., Open vSwitch [28]) for communicating with outside world via veth. The virtual switch enables the overlay traffic to travel across the physical network by encapsulation of overlay traffic, i.e., wrapping packets inside of other packets (if the tunnel protocol is VxLAN, it wraps Ethernet frame inside of a UDP packet). Fig. 1(a) shows the packets flow in container overlay network. When a container application sends a packet, it first traverses the overlay network stack, adding network header (e.g., virtual IP header and virtual Ethernet header) and then is sent from the virtual network interface. The packet will be forwarded out by virtual switch according to its forwarding strategy. As Fig. 1(b) shows, the virtual switch will add a UDP header, physical IP header and physical Ethernet header to the Ethernet frame (assuming VxLAN is the tunneling protocol). The Ethernet frame is encapsulated as a UDP packet and delivered from the physical network interface. As discussed

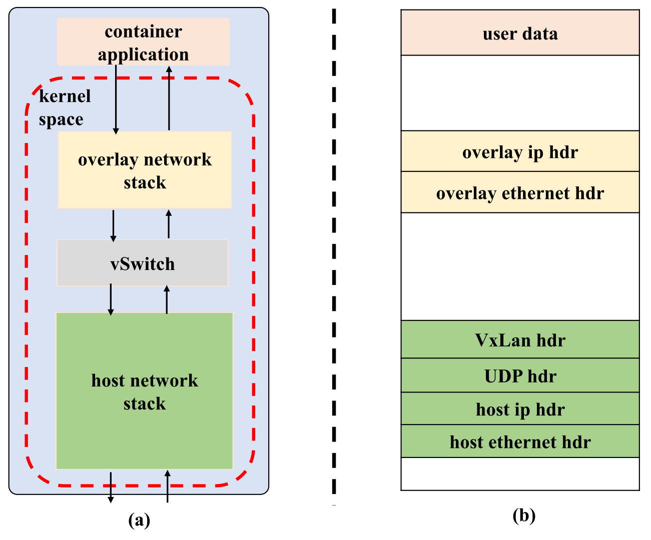


Fig. 1. (a) Packet processing path in tunnel-based container overlay network, (b) Packet structure in tunnel-based container overlay network (VxLan-based). User data that comes from container applications should first traverse the overlay network stack and then traverse the host network stack.

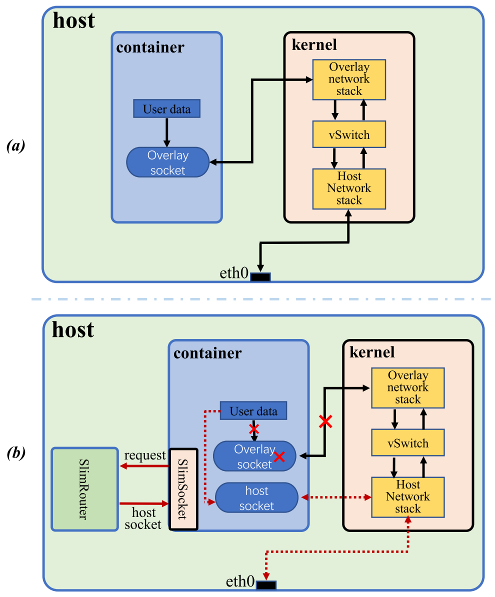


Fig. 2. Comparison of packet flow during data transmission between *(a)* tunnel-based overlay network and *(b) Slim (the dotted line)*.

above, the drawback of overlay mode is the high-overhead of the packet transformation.

B. Overlay Network's IP Address Management

The container overlay network provides network-level isolation to the container using IPAM (IP Address Management) that monitors and manages IP addresses. When a container starts, the IPAM automatically chooses a unique IP address from the available IP pools and assigns it for that container, and that address will be released when the container exits. When a container uses the IP address to communicate with another container, the container overlay network matches the corresponding host address through IPAM. Different solutions have different IPAM strategies. For example, in *Weave* [18], each host owns a certain IP address space and those hosts share the information of IP address space that belongs to themselves via gossip mechanism, thus hosts can learn about the IP mapping information between container and host. Other solution like *Flannel* [3] uses a KV store (*etcd*) to maintain a mapping between allocated subnets and real host IP addresses. For overlay network solutions, the mapping between containers' overlay virtual IP and hosts' physical IP are usually cached in each host, which can be fetched quickly when establishing a network connection.

C. Latest Low-Overhead Overlay Network: Slim

In Section II-A, the container overlay network needs to enter the overlay network stack and host network stack. The high overhead of packet conversion results in poor performance of the overlay network. It easily consumes about 2x CPU resources during data transmission compared with native host socket, as observed in previous works [16], [22], [23]. In order to relieve the burden of per-packet transformation (i.e., each packet traverses the network stack twice) during data transmission, *Slim* [23], a low-overhead container overlay network, directly passes the host namespace file descriptor to the container. By this way, packets sent by container applications can bypass the overlay network stack and vSwitch, reducing the overhead of per-packet transformation by only traversing the host network stack.

Fig. 2(a) shows the packet processing path in a current tunnel-based overlay network. In contrast, as shown in Fig. 2(b), *Slim* replaces the overlay socket with the host socket with the help of *SlimSocket* and *SlimRouter*. *SlimSocket* is a user-space shim layer dynamically linked with application libraries, intercepting invocations of container's socket-related system calls (syscall). *SlimRouter* is a user-space process running on the host that establishes and maintains a host socket for container communication. SlimSocket can transfer data with SlimRouter through inter-process communication. SlimSocket intercepts Socket system calls and forwards them to SlimRouter, completing the connection between containers and informing clients of the information needed for host connection. SlimRouter will establish connections between hosts according to host information. When the host connection is established, the host socket will be transferred to SlimSocket through inter-process communication. SlimSocket will replace the original container socket with the host socket. As such, containers can operate the host socket directly, bypassing the overlay network stack and the vSwitch forwarding.

D. Long Connection Setup Time in Slim

In this section, we will introduce Slim and the problems with it in more detail. Although *Slim* provides an efficient mechanism for data transmission in container overlay network, it sacrifices the connection setup time, which is much longer than the current tunnel-based overlay network solutions. Fig. 6(a) overviews the connection setup procedure in *Slim*. Specifically, when server container calls `bind` on the socket with its overlay IP (e.g., 10.0.0.1) and port (e.g., 80), it will send a request to *SlimRouter*, then, *SlimRouter* calls `bind` with host IP (e.g., 1.2.3.4) and an unused host port (e.g., port 1234), and return the host socket to the container. The server container stores the overlay socket and host socket at the same time, and listens for

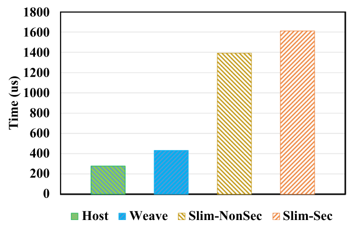


Fig. 3. 99_*th* percentile TCP connection setup time in native host network, *Weave* (tunnel-based overlay network) and *Slim*.

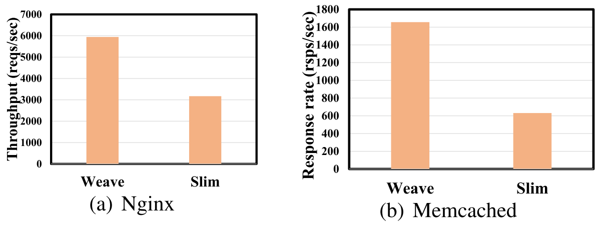


Fig. 4. Performance of applications using *Weave* (tunnel-based overlay network) and *Slim*, respectively. (a) Throughput of the backend web server with 1 K concurrent connections. (b) Response rate of Memcached server, creating 30 K connections in total.

the overlay socket and host socket. When *SlimSocket* on the client side detects that the client container wants to set up a connection with the server container that binds on IP 10.0.0.1 and port 80, it first establishes a network connection to the server container using a standard tunnel-based overlay network (the mapping between container IP and host IP). When the client container is successfully connected to the server container, the server container sends the host IP 1.2.3.4 and port 1234 to the client container through `send`. Then, the client container gets the target host IP address and the port which are actually used by the target server container, through this tunnel-based overlay network connection. Until *SlimSocket* learns about the information, it can send a request carrying the target host IP address 1.2.3.4 and port 1234 to *SlimRouter*. Then *SlimRouter* on the client side starts to establish a network connection with the *SlimRouter* process that on the server side, using the host IP 1.2.3.4 and port 1234. When the host network connection has established, it returns the host socket back to the container. Then the containers on both sides can directly use the host socket for communication, requiring no extra packet transformation (e.g., VxLAN encapsulation/decapsulation).

To evaluate the connection setup time, we conduct a testbed experiment on a 10 Gbps network (detailed testbed settings described in Section V). Fig. 3 shows the 99_*th* percentile time consumed by TCP connection (we test 10 K times in total). *Slim* takes 3.4x and 4.8x longer connection setup time on average compared to tunnel-based overlay network (Weave) and native host network, respectively. The reason why *Slim* takes such long time is because of the complex connection establishment mechanism, which consists of a number of steps: e.g., establishing overlay network connection, sending mapping information between client and server via the overlay TCP connection, the interprocess communication, etc., as discribed above. The performance is even worse when the secure mode is on. This significantly hurts the application performance.

E. Fast Connection Setup is Important to Applications

Short-lived connections are not uncommon for container applications in datacenters [38], [39], [40]. Actually, many applications (e.g., PHP applications [41]) tend to establish/close connections on demand instead of maintaining long-lived persistent connections, because of the resource limitation, difficulty of debugging, and the risk of causing deadlock/error in the server side, etc.. For those applications, the connection setup time is crucial to their overall performance. We use two real-world applications, Nginx [30] and Memcached [44] as examples to illustrate this point.

Nginx Load Balancer: Nginx [30] load balancer is a widely used container application in datacenters [7]. In practice, container applications that run Nginx load balancer need to process many concurrent short-lived connections with high-performance, under the following typical scenarios:

1) For the sake of performance, Nginx load balancer maintains a number of idle persistent connections (adjustable by setting *keepalive* parameter in the config file) with backend web servers, avoiding the overhead of creating a connection for each request. However, the number of idle persistent connections is recommended to be set small enough considering resource consumption [34]. Also, those persistent connections are not always alive because they will be closed periodically to free per-connection memory allocations [33]. Therefore, the Nginx load balancer usually should quickly create new connections with the backend server upon request burst when the pre-created persistent connections are not enough. For example, if there are only 100 persistent idle connections alive, but there come 10,000 concurrent client connections, the Nginx load balancer needs to quickly create 9,900 new TCP connections with backend servers.

2) HTTP/1.0, which doesn't support persistent connections is still in use today. We observe that there is still a lot of HTTP/1.0 traffic in the Internet according to real measurements [42], [43]. For HTTP/1.0 users, the Nginx load balancer should create a new connection for each coming request. Although it is possible to configure to use a persistent connection to serve HTTP/1.0 requests in the recent Nginx version, maintaining a large number of alive persistent connections between Nginx load balancer and the backend web servers to deal with the burst requests is unpractical. One reason is to avoid wasting resources, since a large number of connections will consume a lot of resources, but the burst traffic may only last for a short period of time. Another reason is that the persistent connections with the backend web server have their own survival time (configurable in the config file with default survival time to be 65 s). Those persistent connections will be closed automatically.

The long network setup time may greatly hurt the performance of the Nginx load balancer. To evaluate its impact, we run an Nginx load balancer in a container on one machine and a backend web server (also using Nginx) in one container on another machine, and use *Apache Benchmarking tool* (*ab*) [4] to measure the throughput of the backend web server (detailed testbed setup is described in Section V). We use *ab* to send 10 K requests using 1 K concurrent connections in total (using a short-lived connection between the client and the Nginx load balancer), to emulate the client request burst. As shown in Fig. 4(a), in *Slim*, the performance of Nginx backend server decreases heavily because of the long network setup time,

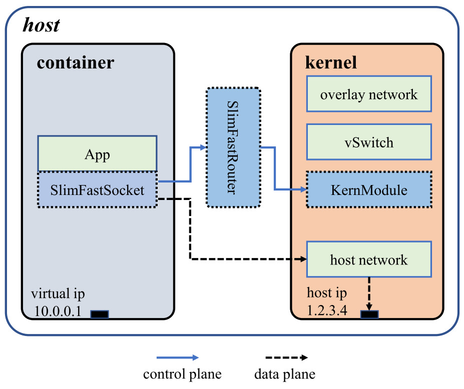


Fig. 5. Architecture of *SlimFast*.

achieving only about 53% throughput of tunnel-based overlay network *Weave*.

Memcached: Memcached [44] is a high-performance in-memory key-value store that widely used in datacenters [35], [36], [37]. Real measurements in datacenters [39] report that there are a lot of flows in *Cache* client/server whose size are less than 10 KB (e.g., about 99% of flows in intra-rack and 80% of flows in intra-datacenter) and last for about only 1 ms (e.g., about 20% of flows in intra-rack). The long connection setup time is unfriendly to those short-lived connections in Memcached, which may decrease the overall performance greatly. We run a Memcached server in a container on one machine and use mcperf [1] to issue requests to the Memcached server in a container on another machine (detailed testbed setup is described in Section V). We use mcperf to create 30 K connections in total and each connection is created after previous connection is closed. As shown in Fig. 4(b), the Memcached response rate of *Slim* is only about 37% compared with tunnel-based overlay network *Weave*, which is affected by the long connection setup time.

III. Design

A. Overview

SlimFast provides a low-overhead container overlay network which directly uses the host OS socket for communication, and provides a fast connection setup without extra communication to get the mapping between the host socket and overlay socket. Note that the current *SlimFast* design focuses on connection-oriented sockets (e.g., TCP) for a container overlay network. We will discuss connectionless sockets (e.g., UDP) in Section III-C. Without explicitly specifying, in this article, sockets refer to connection-oriented sockets (e.g., TCP).

SlimFast follows the same architecture as *Slim*, as shown in Fig. 5. *SlimFast* contains three main components, 1) SlimFastSocket, 2) SlimFastRouter and 3) KernModule. SlimFastSocket is a shim layer library dynamically linked with container application binaries, intercepting applications' socket-related system calls such as `socket()`, `bind()`, `listen()`, `connect()`, etc.. SlimFastRouter is a user-space process that runs in the host namespace, establishing connections in the host stack and passing host sockets to containers used as overlay sockets. *SlimSocket* connects to the *SlimRouter* through inter-process communication (IPC) socket (e.g., the implementation is to use *Unix Domain Socket*) and forwards system calls to the *SlimRouter*. KernModule is an optional module that can be loaded into the OS kernel. When secure mode is on, KernModule can track and revoke the host sockets used inside containers, and prohibit containers' unsafe system calls using these host sockets.

In *Slim*, the *SlimSocket* function is to establish a connection between containers and transmit information about the server host to client container via send. *SlimRouter* is accountable for establishing the connection between hosts and returning the host socket to *SlimSocket* through inter-process communication. The socket is replaced by *SlimSocket*. In *SlimFast*, SlimFastSocket intercepts socket system call forwarded to SlimFastRouter and then listens for inter-process communication socket. SlimFastRouter establishes the socket mapping table according to the server container information and establishes the connection between hosts. We gave up the process of container network in SlimSocket, we no longer listened to the container network socket, but listened to the inter-process communicating socket with SlimFastRouter. SlimFastRouter establishes a connection between hosts. According to the socket mapping table, the host socket is returned to the server container of the corresponding inter-process communication socket. SlimFastSocket will overwrite the original container socket with the host socket.

Slim requires an additional container connection to connect a host. In *SlimFast*, we reserve a dedicated and pre-known host listening port. When the client container needs to establish a connection, the client can directly establish a host connection with the server host IP provided by IPAM and the listening port. *SlimFast* relies on existing container overlay IPAM schemes to manage the mapping between host IP and container overlay IP (introduced in Section II-A). The IP mapping information can be updated in the background, which is not on the critical path of the container connection setup and data transmission.

We design the socket mapping table, which consists of the server container's IP address, port, and IPC socket. SlimFastRouter utilizes this table to forward the host socket to the server container. When the server container needs to listen, SlimFastSocket sends its IP address and port to SlimFastRouter through the IPC socket. SlimFastRouter binds the IPC socket of SlimFastSocket to the IP address and port of the server container and adds it to the mapping table. When a client connects to the server, it sends the server container's IP address and port to the server's SlimFastRouter. SlimFastRouter then utilizes the mapping table to transfer the host socket to SlimFastSocket via the IPC socket. The mapping table matches the server container only to the IPC socket within the host. As a result, neither the client nor the server container needs to be concerned with additional network management.

B. Fast Connection Setup

In this section, we will talk about the working mechanism of *SlimFast* and then supplement it with a detailed example. Different from *Slim*, *SlimFast* can establish a connection without extra communication between both sides. Fig. 6(b) overviews the connection setup procedure in *SlimFast*. Instead of getting port mapping information from the server before establishing a connection, *SlimFast* reserves a dedicated and pre-known host port for the container overlay network. When the server

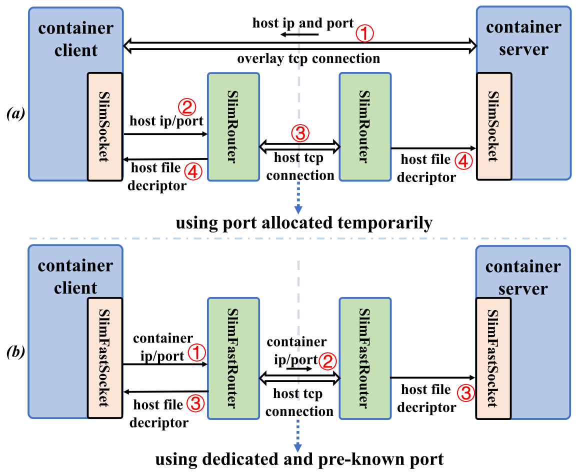


Fig. 6. Connection setup overview. *(a) Slim. (b) SlimFast.*

container creates a socket, it will store the IP address and port of the container into the socket mapping table through SlimFastRouter, and then enter the listening state. When a client container connects to a server container, SlimFastSocket intercepts and forwards the syscall to SlimFastRouter, which obtains the server container's host address using IPAM. SlimFastRouter establishes a host connection to the server's dedicated host port and includes the target overlay IP/port information while doing so. After accepting the host connection and getting the embedded overlay IP/port information, the server SlimFastRouter passes the host socket to the corresponding container application working as its overlay socket.

The server SlimFastRouter finds the correct container overlay socket by maintaining a local *socket mapping table*. Specifically, once the server SlimFastSocket detects the container application starts a server socket (e.g., monitor the system call *listen()*), it establishes a local connection with the SlimFastRouter via a local inter-process communication socket. Meanwhile, it tells the SlimFastRouter that which overlay IP/port the container server socket is listening on. Then the SlimFastRouter inserts an entry in its *socket mapping table* recording the mapping information between the overlay IP/port and the IPC socket to the SlimFastSocket.Next, whenever receiving a new connection, by looking up the *socket mapping table*, the SlimFastRouter can pass the host connection socket to the corresponding IPC socket to the SlimFastSocket, and return it to the container application as its overlay socket.

A Detailed Example of How Connection Setup Works in SlimFast. On the client side, assume that the container TCP client is going to make a connection with the container TCP server. As Fig. 7 shows, it first calls the *socket()* to create a socket, and then, calls the *connect()* to start the three-way handshake procedure. Instead of executing the original syscall, the *connect()* call will be intercepted by SlimFastSocket and forwarded to SlimFastRouter. When the container application calls *connect()*, SlimFastSocket will send a request to SlimFastRouter carrying the IP address and port number (i.e., IP 10.0.0.1 and port 80) of the overlay server socket that the client is going to connect. Once the server SlimFastRouter receives the connection setup request, it parses the request message, learning that the destination server container IP is 10.0.0.1. Through overlay IPAM, the SlimFastRouter already knows the IP address of the host (e.g., 1.2.3.4 in Fig. 7) where the container with IP 10.0.0.1 runs on, so it can immediately set up a host network connection with the destination SlimFastRouter using the pre-known dedicated port (e.g., 1,234).

After the host connection has been established, SlimFastRouter sends a message that carries the server container IP/port information (i.e., IP = 10.0.0.1 and port = 80) to the server side via the host network connection and passes the host file descriptor to the local container working as its overlay socket.

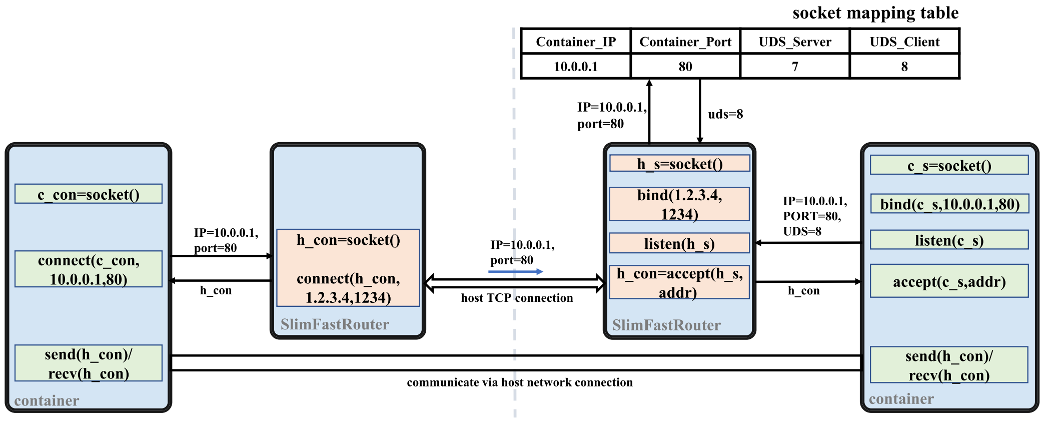
On the server side, SlimFastRouter always listens on a dedicated and public port (e.g., 1,234). SlimFastSocket will monitor (e.g., monitor the function *listen()*) whether a server socket starts in a container. If a server socket starts, SlimFastSocket first establishes a connection with SlimFastRouter via a local IPC socket such as Unix Domain Socket (UDS). Then it sends a register message that contains IP/port (i.e., 10.0.0.1/80) that the container socket is listening on to SlimFastRouter via the IPC connection. When SlimFastRouter receives a register message from SlimFastSocket, it adds an entry to the *socket mapping table*. SlimFastRouter uses the mapping table to distinguish container overlay sockets. After SlimFastRouter accepts a host connection from the client, it will also receive an embedded message with the target container overlay IP/port (i.e., 10.0.0.1/80). SlimFastRouter looks up the corresponding IPC socket in the mapping table, and via the corresponding IPC socket (e.g., UDS 8), it passes the host socket file descriptor to the target container as its overlay socket. After that, the connection setup is done and the containers on both sides can communicate directly via host sockets without extra packet transformation.

C. Connectionless Protocol

We further extend the connectionless protocol with a pre-defined listening port and hash tables. Using host sockets directly with connectionless protocols presents a problem in which the client container cannot determine the host port used by the server container and, therefore, cannot send data directly. In connectionless protocols, it is unnecessary to waste at least one round-trip matching time by informing the server container of the port through connection establishment. Instead, by inserting a hash value into the UDP packet, SlimFast can directly transfer the packet to the specified server container using a pre-defined listening port. This container matching process is completed in the first round-trip time, without wasting any bandwidth.

1) Hash Table Structure: In connectionless protocol, *SlimFast* will directly create host sockets. Both the client and server maintain their own hash tables and use their own hash computation methods. Fig. 8(b) and (c) display the hash table structures maintained by the server and client, respectively. The server hash table stores the hash value of the server container, as well as the IP address and port of the server host. This allows the client to translate container information to host information. Similarly, the client hash table stores the hash value of the client container, IP address, and port of the client host.

2) How Connectionless Protocol Works in SlimFast: The client container calls the socket(), it will directly create the host socket. When sending the packet, SlimFastSocket intercepts the system call, calculates the hash value of the server container's



(a) TCP client (b) TCP server

Fig. 7. Example of the detailed procedure in creating a container TCP connection using *SlimFast*.

IP address and port, and places it into the packet. SlimFast then checks the server hash table to see if the hash value exists. If it does not, SlimFast directly queries the IPAM to obtain the corresponding IP address of the server host. Finally, the packet is sent to the listening port of the server host.

On the server side, SlimFastRouter parses the hash value and searches the socket mapping table to determine the host IP address and port used by the server container. It then inserts the hash value of the client.s information and the host IP/port into the client hash table. The hash value is included in the packet and sent through inter-process communication to the server container. The server container parses the hash value and finds the corresponding client host IP/port in the client hash table. When the server container needs to reply to data, it sends it directly to the corresponding client host. Additionally, the server container includes the hash value of the overlay IP/port in the packet when sending it.

After receiving the packet, the client host updates the host port corresponding to the hash value in the server hash table. When the client needs to send data to the server container again, it queries the server hash table for the corresponding host IP address and port.

During the first round-trip time, the client must repeat the above process since it does not initially know the host port associated with the server container. After receiving the first data packet from the server container, the client can query the server hash table for the host information corresponding to the server container, and then directly send data to the host IP address and port.

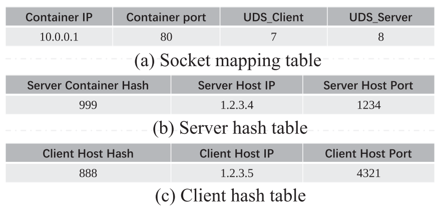
3) End Host Logic: The server container creates the host socket and waits for incoming packets, which can be received directly from clients or through SlimFastRouter. When the server receives packets from SlimFastRouter, it parses the hash value to determine the corresponding client host IP/port, which is then used as the sending address. When sending data, the server container calculates its IP address and port as the hash value and adds it to the packet. The client can locate the server container using the hash value. If the host socket directly receives data, the hash value will be resolved and discarded.

SlimFastRouter is responsible for searching the server container and providing it with information about the client host IP address. When SlimFastRouter receives packets, it parses the hash value and finds the IP address and port of the corresponding server host in the server hash table. SlimFastRouter then calculates the IP address and port of the client host from the data packet as a hash value, adds it to the client hash table, and forwards the data packet with the hash value to the server container.

The client maintains a server hash table that is built from both IPAM information and received packet host information. The hash value of the server overlay IP/port is added to packets, and the server hash table is queried. If the hash value is not found, IPAM is used to search for the host IP address. The packet is then transmitted directly to the host IP address with the pre-defined listening port. The client writes the server host port into the server hash table after receiving data packets and parsing the hash value. When the hash value is queried, it can be directly sent to the host IP address and port corresponding to the server container.

D. Security Policy

SlimFast directly uses the host socket for container communication as *Slim*, thus keeping low-overhead during data transmission. Also, similar as *Slim*, *SlimFast* can configure flexible control-plane (e.g., access control over overlay packet header fields) or data-plane policies (e.g., rate limiting and quality of service, etc.) as original tunnel-based overlay solution, by inserting rules in the SlimFastRouter. Moreover, through KernModule, *SlimFast* can maintain the same security model as today's tunnel-based container overlay networks. In *SlimFast*, we extend the socket mapping table maintained in TCP and UDP service, which can complete the packet-level flow control policy. After we use the host socket, the packet will be processed in the host. *Slim*'s security policy will not be able to perform packet-level detection on containers that use host sockets, because the host address used by the restricted container is not known. After the security mode is enabled, *SlimFast* maintains the socket mapping table on the server side and adds the corresponding host IP address and port into it. When the specified container IP address and port need to be hashed, we can find the host IP address and port through the socket mapping table to replace the container information, thus completing the packet-level filtering.



Container IP	Container port	UDS_Client	UDS_Server
10.0.0.1	80	7	8

(a) Socket mapping table

Server Container Hash	Server Host IP	Server Host Port
999	1.2.3.4	1234

(b) Server hash table

Client Host Hash	Client Host IP	Client Host Port
888	1.2.3.5	4321

(c) Client hash table

Fig. 8. (a) Socket mapping table. (b) Server hash table. (c) Client hash table.

IV. Implementation

A. Overview

We implement *SlimFast* on Linux OS and Docker with about 3.5 K lines of C++ code, on top of the code-base of *Slim*. We use an overlay network solution, *Weave*, to manage IP addresses for containers. Unix domain socket is used as the IPC socket for communication between the SlimFastSocket and the SlimFastRouter. The structure of *SlimFast*'s socket mapping table in our implementation is shown in Fig. 8.

B. Reuse Component

SlimFastSocket uses LD_PRELOAD to dynamically link to the application binary. SlimFastSocket implements a set of socket APIs such as *socket(), bind(), listen(),* etc.. SlimFastRouter uses a system call *send_msg()* to send a host socket to SlimFastSocket in both non-secure and secure mode. We reused Slim's security module. In secure mode, SlimFastSocket sends the socket index on container application to SlimFastRouter after it receives the host socket. Then KernModule stores the socket index and the PID of container application in the list. In SlimFastSocket, maintain a mapping table of file descriptors. When using non-blocking IO to listen for a container network socket, we intercept the existing epoll call and replace it with an IPC socket. When using *connect*, we will replace it directly with the host socket, no longer requiring the container socket.

C. Address Translation

Weave's IPAM does not provide an interface for upper-layer calls to retrieve the host IP address that corresponds to a container IP address. To retrieve this information, we can query Weave's DNS (Domain Name System) using a command. In Weave's DNS, the IP addresses of containers and hosts are mapped. SlimFast extracts the IP addresses of containers and hosts for mapping and saves them in entries. When translating overlay IP, SlimFast can directly look up the table. The whole process can be updated in the background using a separate process, which has no effect on the container connection. Similarly, other container overlay networks also have interfaces that obtain the mapping between the overlay IP addresses and the host IP addresses. During initialization, SlimFast write the host IP address corresponding to the subnet in advance.

D. Socket Mapping Table

The socket mapping table is used within the server to map the server container and UDS. During connection establishment, the SlimFastSocket uses UDS to connect to SlimFastRouter. SlimFastSocket passes the ip and port of the container to SlimFastRouter via `send_msg` when `bind` is used on the server container. SlimFastRouter inserts the ip and port of the server container and the corresponding UDS into the socket mapping table. When the host connection is established and the server container ip address and port are received, SlimFastRouter queries the socket matching table and uses epoll notification to replace the host socket to the specified container.

E. Optimized Connection Establishment

To avoid the high overhead of frequently creating and terminating threads, SlimFastRouter creates a thread pool, enabling it to process routines in parallel. The thread pool consists of two main parts: Processing inter-process requests and establishing connections. After SlimFastRouter starts up, it uses multithreading to process the requests of SlimFastSocket, while interprocess connections and requests are processed in parallel. SlimFastRouter optimizes connection requests by using multiple pre-reserved ports and multiple threads. To further improve processing efficiency, SlimFast uses non-blocking I/O to listen on interprocess sockets and network sockets. SlimFastSocket uses epoll to monitor inter-process communication and waits for SlimFastRouter to wake up after completing the connection establishment. Similarly, SlimFastRouter uses epoll to listen on multiple pre-reserved ports until the client requests a wake-up call.

V. Evaluation

Experimental Setup: Our experimental setup consists of 2 physical machines: *machine A* (Linux version 5.4.0-137-generic) and *machine B* (Linux version 4.15.0-142-generic). The *machine A* has 2 CPUs (Intel Xeon CPU E5-2650 0 @ 2.00 GHz) and each CPU has 8 physical cores. The *machine B* also has 2 CPUs (Intel Xeon CPU E5-2620 v3 @ 2.40 GHz) and each CPU has 6 physical cores. We set the CPU frequency governor as "performance" in all of the experiment. As for the network link, we directly connect 2 hosts with 10 Gbps NIC.

A. Microbenchmarks

In this section, we measure the connection setup time of TCP applications with a host network, overlay network, *Slim* and *SlimFast*, respectively. Then, we use iperf [5] to evaluate the throughput, also monitor the CPU utilization both in client and server side. Finally, we test the round-trip time(RTT) delay under connectionless protocol.

Connection Setup Time Benchmarking: We run a TCP client process in one container and a TCP server process in another container. Those 2 containers are started in *machine A* and *machine B*, respectively. The TCP client process creates one connection, recording the connection setup time and then exit immediately. The system time at the beginning of connection construction is recorded, and the system time after connection construction is recorded. According to the difference between the two, we can get the time required for the whole process of connection construction. To avoid accidental errors, We test the above process 10 K times in total.

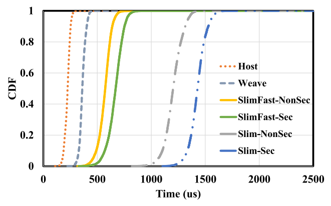


Fig. 9. TCP connection setup time.

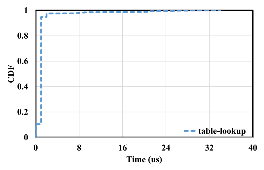


Fig. 10. Time cost of table-lookup at scale (1K entries).

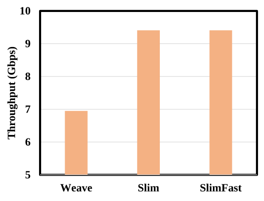


Fig. 11. Single TCP flow throughput with *Weave*, *Slim* and *SlimFast*.

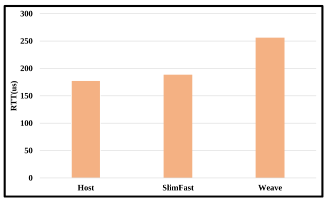


Fig. 12. Connectionless protocol communication delay.

Results: As shown in Fig. 9, the 99.5_*th* percentile connection setup time with a host network, *Weave*, *SlimFast-NonSec*, *SlimFast-Sec*, *Slim-NonSec* and *Slim-Sec* are 284 μs, 440 μs, 720 μs, 842 μs, 1343 μs and 1557 μs, respectively. *Slim* takes much longer time than other ways because it needs extra communication to learn the mapping information from the remote side. *SlimFast* achieves a great deal better performance than *Slim* because it removes the extra communication but lookups the mapping information locally. The secure mode requires more time cost than non-secure mode is because it needs to do some other operations (e.g., communicating with the KernModule). In *SlimFast*, the interprocess communication between SlimFastSocket and SlimFastRouter requires additional time consumption, causing a little of performance loss comparing with *Weave*. The long packet processing path (i.e., each packet traverses the network stack twice) of *Weave* brings a degree of performance loss comparing with the host network.

Mapping Table Lookup Time: SlimFast removes the extra communication while it lookups the mapping information in a local mapping table. *SlimFast* adds a new entry into the mapping table while a new TCP server starts listening in container. A number of containers may be deployed in one host, which leads to a large number of entries in the mapping table. We use a hash table to store the mapping entries as it provides fast lookup operations.

In *SlimFast*, lookuping an entry in the mapping table requires minimal overhead. In this experiment, we run 1 K TCP servers inside one container (on *machine B*). Accordingly, *SlimFast* adds 1,000 entries into the mapping table. We then run a TCP client in another container (on *machine A*). This TCP client randomly selects one server out of one thousand and then makes a connection with that server. We test 10 K times in total. On each connection setup time stage, *SlimFast* lookups a corresponding entry in the mapping table and we record the time cost that table lookup consumes.

Results: As shown in Fig. 10, the 95_*th* percentile time consumption is 2 μs. It shows that *SlimFast* can easily adapt to large-scale environment.

Throughput and CPU Utilization: We run an iperf client and a server on *machine A* and *machine B* respectively. The iperf client creates a TCP flow with server, measuring the maximum achievable bandwidth on this link. We also use *mpstat* tool [2] to monitor the CPU utilization of the iperf client and server.

Results: Fig. 11 shows that *Weave*, *Slim* and *SlimFast* achieve the throughput of 6.95 Gbps, 9.41 Gbps and 9.41 Gbps, respectively. *Weave* hurts the performance because it requires extra tunneling encapsulation. As shown in Fig. 13, in client side, *SlimFast*, *Slim* and *Weave* use 0.33, 0.33 and 1.23 CPU core, respectively, while 0.47, 0.46 and 1.49 CPU core in server side. In *Weave*, the CPU utilization in *softirq* increases greatly since the tasks of extra packet transformation are offloaded to per-core dedicated *softirq* thread. In *Slim* and *SlimFast*, the CPU utilization of the server is higher than that of the client, which mainly comes from the communication transmission between processes. They both need to convert the container information into the host information through inter-process communication.

Connectionless Protocol Performance: The iperf approach is not used in testing connectionless protocol performance, because the iperf tool does a pre-build connection when testing connectionless protocol performance. We choose to record the client's `sendto` and `recvfrom` completion time and record the completion time to compare the host, SlimFast, and Weave network. The communication method we use is that after the client sends the packet, the server receives the data and re-sends it to the client. To avoid accidental errors, 1 k rounds were continuously tested, with the client changing sockets every round and the server running all the time. The packet size of each experiment is 1 KB, and each round will cycle our communication mode 1 K times.

Results: Fig. 12 shows that the communication latency under the host communication, SlimFast, and Weave networks is 177.64us, 188.57us, and 256.2us. The performance of SlimFast is similar to that of the host. This is because the host operating system socket is directly used for communication, but not completely consistent because the hash value calculation and address replacement are added. Weave's performance is down because it uses container network sockets and is wrapped twice.

B. Applications

We apply *SlimFast* to 2 real-world applications: Nginx and Memcached. There exits several similar container overlay network solutions, e.g., *Weave* [18], *Calico* [17], *Flannel* [3], etc.. In

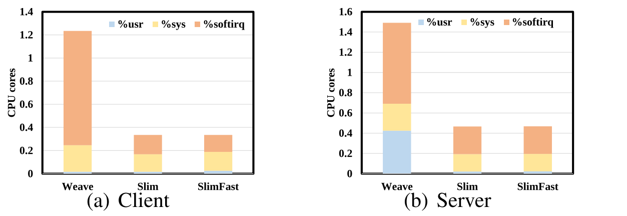


Fig. 13. CPU utilization with *Weave*, *Slim* and *SlimFast*. (a) iperf client, (b) iperf server.

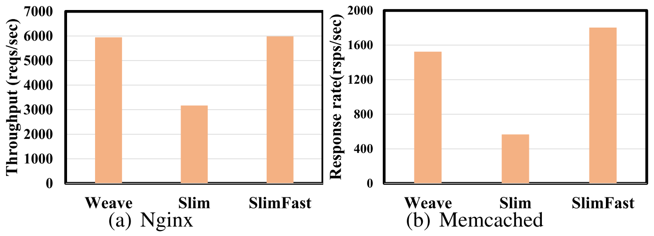


Fig. 14. (a) Throughput of Nginx server with 10 K concurrent visits, (b) response rate of Memcached server while using short-lived connection.

this article, we choose *Weave* as the benchmark. Our evaluation uses *SlimFast* and *Slim* running in non-secure mode.

1) Nginx-Based Load Balancer: Setup. We deploy a Nginx proxy on *machine A* and a Nginx web server as the upstream server on *machine B*. We choose *ApacheBench (ab)* as the benchmarking tool and choose throughput (requests/second) as the evaluating indicator. We set up a HTML file with size of 1 KB on the Nginx backend server. In Nginx proxy, we set the configuration parameters *keepalive=100* (i.e., the maximum number of idle keepalive connections with upstream servers) and *keepalive_requests=100* (i.e., maximum number of requests that can be served through one keepalive connection). We set the number of Nginx worker process to 1 both in Nginx proxy and upstream server.

Performance Under High Concurrency. We use *ab* tool to simulate 1 K of concurrent users, requesting the HTML file with size of 1 KB. Each user sends one request using short-lived connection, and the *ab* tool generates 10 K requests in total. In such a scenario, the Nginx proxy needs to create a lot of new TCP connections with the upstream server because the number of pre-created persistent connections between Nginx proxy and upstream web server is not enough.

Results. As shown in Fig. 14(a), *SlimFast* achieves the throughput of about 5,987 requests/second while *Slim* achieves about 3,171 requests/second. *SlimFast* improves the throughput of Nginx-based-proxy by about 0.9x. The main reason of poor performance while using *Slim* as it spends too much time on connection setup. *SlimFast* resolves this problem, thus it can attain better performance.

Performance Under Low Concurrency. In this experiment, we use *ab* tool to simulate a single user, requesting the HTML file 10 K times in total. The frequency of making TCP connection between proxy and upstream server decreases greatly in this scene.

Results. As shown in Fig. 15(a), the throughput of Nginx server is 2,250, 2,752, and 2,753 requests/second while using *Weave*, *Slim* and *SlimFast*, repectively. The high-overhead of

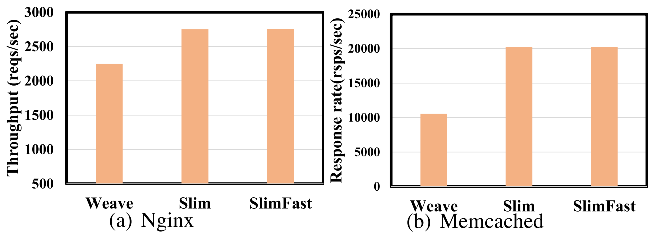


Fig. 15. (a) Throughput of Nginx server with 1 connection and proccessing10 K requests in total, (b) response rate of Memcached server while using persistent connection.

packet encapsulation in *Weave* leads to the decline of performance. Slim and SlimFast both avoid additional encapsulation.

2) Memcached: Setup: We deploy a Memcached server on *machine B*. Then, we run a Memcached benchmarking tool, mcperf [1], to evaluate the performance of Memcached server. In this case we choose the response rate as the evaluating indicator.

Short-Lived Connections. In this experiment, we use *mcperf* to create 30 K TCP connections in total. After the connection has established, it sends one request the item sizes derived from a uniform distribution in the interval of [1, 1,024) bytes to the Memcached server. Note that in this scene, each connection is created after previous connection is closed.

Results. As shown in Fig. 14(b), *SlimFast* improves the response rate Memcached by about 2.2x, as *SlimFast* achieves the response rate of 1803 responses/second while *Slim* archieves 566 responses/second. The reason why *SlimFast* gets better performance than *Slim* is because that the time *SlimFast* saved can be used to complete more requests. *SlimFast* is 1,803 responses/second and *Weave* is 1,523 responses/second. *SlimFast* has a higher throughput than *Weave*. This is because in memcached, query requests are carried out between containers and the connection length is short. Also, the connection is established only after the previous connection has been closed, so it is more accurate to see the effect of connection speed and stack encapsulation time.

Persistent Connections. We use persistent connections between *mcperf* and the Memcached server in this experiment. The *mcperf* creates one persistent connection, sending 30 K requests to server in total and the item size derives from a uniform distribution in the interval of [1, 1024) bytes.

Results. As shown in Fig. 15(b), the response rates of Memcached server are 10,558, 20,217 and 20,227 responses/second while using *Weave*, *Slim* and *SlimFast*, respectively. *SlimFast* improve the response rate of Memcached server by about 92% comparing whith *Weave*. It is resonable that *SlimFast* just 22% (Nginx) / 92% (Memcached) better than regular container-overlays for long-lived connections. For Nginx, the topology setup is: *ab* $\leftrightarrow$ *Nginx-based-proxy* $\leftrightarrow$ *Nginx*. Http requests/responses are first processed by *ab/Nginx-based-proxy/Nginx* processes, then traverse the OS network stack.

We also evaluate the Nginx server performance in the case of container to container. We start a container running ab tool in machine A and a container running Nginx server in machine B. Similar to previous Nginx-based proxy experiment, the *ab* tool will create a single persistent connection, requesting the HTML file 10 K times in total. The throughput is 3,880, 5,918 and 5,929 requests/s while using *Weave*, *Slim* and *SlimFast*, respectively. *SlimFast* is 53% better than *Weave*.

VI. Discussion and Limitation

General Mechanisms Behind. SlimFast's mechanisms are useful for other systems design beyond the specific problem it addresses. The method of locally differentiating container sockets on the server side can be applied by other overlay-network systems (using a similar way), and the data structure and mechanism used in *SlimFast* to guarantee good performance can also benefit other connection/packet dispatch systems.

NICs. Some commercial NICs (e.g., ConnectX-4) offer Vxlan offload functionality, which segments packets carrying the Vxlan header after encapsulation. However, this offload function requires the inclusion of the container IP header, Ethernet header, and VxLan header when encapsulating packets. This limitation reduces the number of containers that can be created and maintained due to on-chip space constraints. Furthermore, traffic management issues arise when managing container-orchestrated networks. While customized hardware can potentially solve these issues, this requires a design that strongly couples with the container orchestration network. Additionally, offloading Vxlan encapsulation is not essential for the container overlay network, and the network adapter's limited traffic management functions prevent the customization of traffic management policies.

VII. Related Work

Optimizations of Virtual Networking: There is a large body of work targeting at optimizing the virtual networking. Nam et al. proposed BASTION [48], a network stack for container network, which can enhance the security of container and improve the performance of container communications to a certain extent. BrFusion [50], a method that allows containers inside a VM to directly use the host layer networking stack. Although BASTION and BrFusion can improve the performance of container network, but neither BASTION nor BrFusion supports container overlay network. FreeFlow [49] provides a virtual RDMA networking solution for containers. Socket-outsourcing [51] provides fast networking for hosted virtual machines(VMs).

Optimizations of Packet Processing: With the improvement of network device performance, one CPU core may be inadequate, especially in container overlay network because of its high-overhead per-packet encapsulation. Receive Side Scaling (RSS) [52], a technique designed for multi-processor systems that aims to increase parallelism and improve performance of packet processing. With RSS enabled, the NIC will distribute different packets to different CPU cores, which improves performance of packet processing. Relatived to RSS, Receive Packet Steering (RPS) [53] can provide the same functionality as RSS but it is implemented in software manner. As observed by [22], RSS or RPS doesn't work effectively in container overlay network.

VIII. Conclusion

Container overlay network has become the de facto networking technique for distributed containers communication in cloud, yet it imposes significant overhead. We design and implement *SlimFast*, a low-overhead container overlay network with fast connection setup. *SlimFast* directly passes host socket to containers and uses reserved server port and local socket mapping table to remove extra round-trip communication during connection setup. Thus it keeps low-overhead and fast in both data transmission and connection setup. *SlimFast* significantly improves the throughput of Nginx proxy and Memcached server and also reduces the CPU utilization. SlimFast's source code is publicly available at https://github.com/zxzx9898/SlimFast.

References

[1] twemperf, 2014. [Online]. Available: https://github.com/twitter-archive/twemperf
[2] mpstat, 2022. [Online]. Available: https://man7.org/linux/man-pages/man1/mpstat.1.html
[3] Flannel, 2022. [Online]. Available: https://github.com/flannel-io/flannel/
[4] ab, 2012. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html
[5] iperf, 2014. [Online]. Available: https://iperf.fr/
[6] VxLan, 2014. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7348
[7] nginx - Docker Hub, 2021. [Online]. Available: https://hub.docker.com/_/nginx
[8] SlimFast code base, 2023. [Online]. Available: https://github.com/zxzx9898/SlimFast
[9] Containers in Google cloud, 2021. [Online]. Available: https://cloud.google.com/containers
[10] twemproxy, 2022. [Online]. Available: https://github.com/twitter/twemproxy
[11] S. Ihm and V. S. Pai, "Towards understanding modern web traffic," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf.*, 2011, pp. 295–312.
[12] K. Jang, S. Han, S. Han, S. B. Moon, and K. Park, "SSLShader: Cheap SSL acceleration with commodity processors," in *Proc. 8th USENIX Conf. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.
[13] R. Nishtala et al., "Scaling memcache at Facebook," in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 385–398.
[14] E. Jeong et al., "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. 11th USENIX Symp. Netw. Syst. Des. Implementation*, 2014, pp. 489–502.
[15] S. Thomas, L. Ao, G. M. Voelker, and G. Porter, "Particle: Ephemeral endpoints for serverless networking," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 16–29.
[16] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *Proc. IEEE Conf. Comput. Commun.*, 2018, pp. 189–197.
[17] Calico, 2022. [Online]. Available: https://www.projectcalico.org/
[18] Weave, 2022. [Online]. Available: https://www.weave.works/
[19] Docker overlay network, 2021. [Online]. Available: https://docs.docker.com/network/network-tutorial-overlay/#use-the-default-overlay-network
[20] S. Fouladi et al., "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads," in *Proc. 14th USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 363–376.
[21] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 263–274.
[22] J. Lei, K. Suo, H. Lu, and J. Rao, "Tackling parallelization challenges of kernel network stack for container overlay networks," in *Proc. 11th USENIX Workshop Hot Topics Cloud Comput.*, 2019, Art. no. 9.
[23] D. Zhuo et al., "Slim: OS kernel support for a low-overhead container overlay network," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 331–344.
[24] HPE EZMERAL CONTAINER PLATFORM, 2021. [Online]. Available: https://www.hpe.com/us/en/solutions/container-platform.html
[25] Titus, "The Netflix container management platform, is now open source," 2021. [Online]. Available: https://netflixtechblog.com/titus-the-netflix-container-management-platform-is-now-open-source-f868c9fb5436
[26] How yelp runs millions of tests every day, 2021. [Online]. Available: https://engineeringblog.yelp.com/2017/04/how-yelp-runs-millions-of-tests-every-day.html
[27] Containers at Google, 2021. [Online]. Available: https://cloud.google.com/containers

[28] Open vSwitch, 2021. [Online]. Available: https://www.openvswitch.org/
[29] Virtual ethernet device (veth), 2021. [Online]. Available: https://man7.org/linux/man-pages/man4/veth.4.html
[30] Nginx, 2021. [Online]. Available: https://www.nginx.com/
[31] R. Li, Y. Li, and W. Li, "An integrated load-balancing scheduling algorithm for Nginx-based web application clusters," *J. Phys.: Conf. Ser.*, vol. 1060, no. 1, 2018, Art. no. 012078.
[32] Z. Wen, G. Li, and G. Yang, "Research and realization of Nginx-based dynamic feedback load balancing algorithm," in *Proc. IEEE 3rd Adv. Inf. Technol. Electron. Autom. Control Conf.*, 2018, pp. 2541–2546.
[33] 2021. [Online]. Available: http://nginx.org/en/docs/http/ngx_http_core_module.html
[34] 2021. [Online]. Available: http://nginx.org/en/docs/http/ngx_http_upstream_module.html
[35] D. Cotroneo, R. Natella, and S. Rosiello, "Dependability evaluation of middleware technology for large-scale distributed caching," in *Proc. IEEE 31st Int. Symp. Softw. Rel. Eng.*, 2020, pp. 218–228.
[36] J. Yang, Y. Yue, and K. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 191–208.
[37] W. Zhang, J. Hwang, T. Wood, K. Ramakrishnan, and H. Huang, "Load balancing of heterogeneous workloads in memcached clusters," in *Proc. 9th Int. Workshop Feedback Comput.*, 2014.
[38] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 2010, pp. 267–280.
[39] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, "Inside the social network's (datacenter) network," in *Proc. ACM Conf. Special Int. Group Data Commun.*, 2015, pp. 123–137.
[40] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "AccelTCP: Accelerating network applications with stateful TCP offloading," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 77–92.
[41] Why persistent connections are bad, 2021. [Online]. Available: https://meta.wikimedia.org/wiki/Why_persistent_connections_are_bad#Why_persistent_connections_are_bad
[42] Shodan, 2021. [Online]. Available: https://www.shodan.io/
[43] Censys, 2021. [Online]. Available: https://censys.io/
[44] Memcached, 2021. [Online]. Available: https://memcached.org/
[45] Docker, 2021. [Online]. Available: https://www.docker.com/
[46] Network address translation (NAT), 2021. [Online]. Available: https://tools.ietf.org/html/rfc2663
[47] Linux network namespace, 2021. [Online]. Available: https://man7.org/linux/man-pages/man7/network_namespaces.7.html
[48] J. Nam, S. Lee, H. Seo, P. Porras, V. Yegneswaran, and S. Shin, "BASTION: A security enforcement network stack for container networks," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 81–95.
[49] D. Kim et al., "FreeFlow: Software-based virtual RDMA networking for containerized clouds," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 113–126.
[50] M. Bacou, G. Todeschi, D. Hagimont, and A. Tchana, "Nested virtualization without the nest," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
[51] H. Eiraku, Y. Shinjo, C. Pu, Y. Koh, and K. Kato, "Fast networking with socket-outsourcing in hosted virtual machine environments," in *Proc. ACM Symp. Appl. Comput.*, 2009, pp. 310–317.
[52] Receive Side Scaling (RSS), 2021. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt
[53] Receive packet steering (RPS), 2021. [Online]. Available: https://lwn.net/Articles/362339/
[54] Slim kernel module, 2021. [Online]. Available: https://github.com/danyangz/Slim/tree/master/kern_module

Fusheng Lin received the master's degree in computer science from Hunan University, in 2022. He is currently working with Tencent. His research interests in computer networking and networked system.

Xin Zhang received the bachelor's degree in network engineering from the Hunan Institute of Engineering, in 2020. He is currently working toward the master's degree with Hunan University. His research interests include docker, congestion control, and In-network processing.

Guo Chen received the PhD degree in computer science from Tsinghua University, in 2016. Before joining Hunan University, he was an associate researcher with Microsoft Research Asia from 2016 to 2018. He is currently a professor with Hunan University. His current research interests lie broadly in networked systems and with a special focus on data center networking.

Li Chen received the BE degree (Hons.) in electronic and computer engineering with a minor in mathematics and the MPhil degree from The Hong Kong University of Science and Technology (HKUST), in 2011 and 2013, respectively. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, HKUST. His currently works on topics in systems, networking and cybersecurity research with Zhongguancun Laboratory.

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2003. He has published more than 200 research articles in international conferences and journals, such as *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, *IEEE Transactions on Computers (TC)*, *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, and International Conference on Parallel Processing (ICPP). His research interests include parallel computing, high-performance computing, and grid and cloud computing. He currently serves on the editorial board for the *IEEE Transactions on Computers (TC)*.

Hongbo Jiang (Senior Member, IEEE) received the PhD degree in computer science from Case Western Reserve University, Cleveland, Ohio, in 2008. He is currently a full professor with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, China. Previously, he was a professor with the Huazhong University of Science and Technology. His research concerns computer networking, especially algorithms and protocols for wireless and mobile networks. He is the editor for the *IEEE/ACM Transactions on Networking*, and the associate editor for the *IEEE Transactions on Mobile Computing* and *IEEE Internet of Things Journal*. He is an elected fellow of Institution of Engineering and Technology (IET) and fellow of British Computer Society (BCS).