High Performance Userspace Networking for Containerized Microservices

Xiaohui Luo, Fengyuan Ren(✉), and Tong Zhang

NNS Group, Tsinghua University, Beijing, China
{luo-xh17,zhangt14}@mails.tsinghua.edu.cn
renfy@tsinghua.edu.cn

Abstract. Containerized microservices have become popular for building systems using simple, lightweight, loosely coupled services. Due to replacing the monolithic application with multiple microservices, inner function calls become inter-microservice communications, which increases the network pressure. However, the networking of containerized microservice built on the kernel that is inefficient. In this paper, we propose DockNet, a high-performance userspace networking solution for containerized microservices. We (1) leverage DPDK and customized LwIP as the high-performance data plane and TCP/IP stack, respectively. (2) introduce a master-slave threading model to decouple execution and management. (3) adopt namespace mechanism to control the access of microservices to data planes and employ timer-based rate limiters to achieve performance isolation. (4) construct fast channels between partner microservices to improve network performance further. In our various experiments, DockNet shows over 4.2×, 4.3×, 5.5× of higher performance compared with existing networking solutions - kernel bridge, Open vSwitch and SR-IOV, respectively.

Keywords: High-performance network · Container · Microservice

1 Introduction

Recently, containerized microservices have redefined the software development landscape. Containerized microservices have been supported by leading cloud providers, such as Amazon EC2 Container Service [4], and Microsoft Azure Container Service [6]. Companies such as Netflix [3] and Uber [16], have replaced their large monolithic applications into multiple small containerized microservices that coordinate to provide required functionalities.

However, microservices increase the network pressure. Since a monolithic application is divided into multiple containerized microservices, inner function calls become inter-microservice communications (e.g., Remote Procedure Calls, and RESTful APIs), which imposes great pressure on the network. For example, over 99% of 5-billion API calls per day in Netflix are internal (but across microservices) [14]. In other words, the microservice architecture makes a trade-off between simplifying development and increasing network pressure.

C. Pahl et al. (Eds.): ICSOC 2018, LNCS 11236, pp. 57–72, 2018.
https://doi.org/10.1007/978-3-030-03596-9_4

Unfortunately, network in the container is inefficient. Containers are essentially processes running in separated and isolated runtime environments. The network of a container is based on the kernel's network stack. Recent researches have shown the performance limitations in the kernel's network stack [9,12,13], which means that these limitations also affect container's network performance. Among these researches, userspace high-performance network stacks have shown significant performance improvements [9,13].

In this paper, we motivate to introduce the userspace high-performance network stack to containerized microservices. We summarize the requirements as follows: *(i) High-Performance.* Although userspace network stacks can offer high-performance, they cannot be directly used by containerized microservices. Because the container's network interface is built upon the kernel, we must provide another specialized userspace interface for the container. Meanwhile, we also need to provide a userspace TCP/IP stack for these network services. *(ii) Easy Management.* Userspace network stacks bypass the kernel, we must implement management functionalities in the userspace network stack. *(iii) Isolation.* Multiple microservices running on the same host is common. We must provide isolation mechanisms for both data plane security and performance guarantee. *(iv) Fast Intra-host Communication.* In some scenarios, partner microservices (e.g., one for requests, the other for responses) locate in the same host. Without crossing the hosts, we should provide higher performance for intra-host communications.

We propose DockNet, a userspace high-performance networking stack for containerized microservices using commodity hardware. DockNet (1) leverages DPDK [5] and customized LwIP [10] as the high-performance data plane and TCP/IP stack, respectively. (2) introduces a master-slave threading model to decouple execution and management. In the master-slave threading model, the master thread is responsible for control and management. The slave threads (i.e., the container threads) are only responsible for processing packets. (3) uses the hardware-based flow director mechanism to virtualize a physical network interface card (NIC) into multiple lightweight network interfaces, which effectively supports multiple containers. For isolation, we adopt namespace mechanism to control the access of containers to data planes. We also employ timer-based rate limiters to achieve performance isolation. (4) constructs fast channels between partner microservices to improve network performance further.

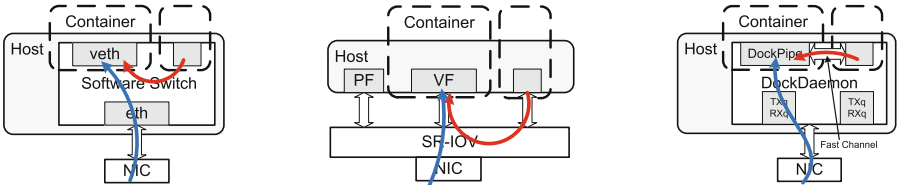
In our evaluations, DockNet achieves high throughput and low latency for both intra- and inter-host traffics, effectively supports bandwidth isolation and multiple containerized microservices, and outperforms dominant networking solutions by more than 4.2× in service-oriented cases. Finally, we build a web cluster based on DockNet to scale the web performance and demonstrate its almost linear scalability.

The rest of this paper is organized as follows. In Sect. 2, we summarize the existing container networking solutions and discuss the kernel inefficiency. In Sect. 3, we present the overview of DockNet, describe its design and implemen-

tation in detail. We validate and evaluate the performance of DockNet in Sect. 4. Finally, the whole paper is concluded in Sect. 5.

2 Background

In this section, we first introduce existing networking approaches for containerized microservices. Then we discuss the performance limitations in the kernel.



(a) Software Switching (b) SR-IOV (c) DockNet

Fig. 1. Container networking architectures.

2.1 Existing Approaches

We survey the existing network approaches to build containerized microservices. Their fundamental techniques can be summarized as follows:

Software Switching: As Fig. 1(a) shows, the software switch is responsible for forwarding packets to the correct receivers. Considering that the packets have already arrived at the host, the extra forwarding operations are redundant. The extra operations waste CPU cycles and make the whole procedure heavyweight. Software switching is adopted by the mainstream container networking solutions, such as Linux bridge and Open vSwitch (OVS) [1].

SR-IOV: As shown in Fig. 1(b), with SR-IOV [2], a virtual function (VF) acts just like a real physical NIC. Packets can be directly be routed (by the hardware) to the correct receivers. SR-IOV eliminates the forwarding costs and provides a lightweight networking architecture. However, packets across different VFs needs to be switched in the NIC's embedded hardware switch. It means that in the intra-host traffic cases, packet forwarding requires both VFs and the NIC, which introduces two extra copies across the PCIe bus.

From the above, we can see that these existing networking solutions are mainly built upon the kernel's network stack that has performance limitations.

2.2 Kernel Inefficiency

Previous studies have exposed three main performance limitations of the kernel: inefficient packet processing, general resource sharing, overheads of context

switching. (1) Per-packet memory (de)allocation and heavy data structures (e.g., `sk_buff`) limit the efficiency [12,13,17]. (2) Resources are shared in many folds: a multi-thread application shares only one listening sockets [12,15], sockets share file descriptors with the regular files [12]. (3) Frequent system calls result in increasing the overheads of context switching [18]. To overcome these limitations introduced by the kernel, many userspace approaches have been proposed.

High-Performance Packet I/O Frameworks: High-performance packet I/O frameworks, such as DPDK [5], PSIO [7], PF_RING [8] and netmap [17], are closely correlated with network interface cards (NICs). These frameworks share similar techniques to accelerate the packet processing: (1) Using memory pools to store packets can reduce the (de)allocating costs during processing packets. (2) Packets are sent/received in batches instead of one by one to amortize costs of system calls. (3) Polling is used for receiving packets rather than triggering interrupts. (4) Bypassing the kernel, data are transmitted between the framework and applications directly.

Userspace Networking Stack: Systems such as mTCP [13] and IX's data plane [9] run their entire networking stack in the userspace. Userspace stacks have the following benefits: (1) Without adverse effects of the complex kernel, userspace networking stacks can optimize packet processing and eliminate expensive system calls. (2) These approaches use lock-free data structures to scale on multicore systems. (3) Flow-consistent hashing of incoming traffic is used for pinning a flow to the same core. (4) Batching is extensively used to amortize system call transition overheads and improve instruction cache locality.

3 System Design and Implementation

Figure 1(c) sketches the overall architecture of DockNet. DockNet provides DockPipes for containerized microservices for high-performance packet I/O. DockDaemon is responsible for functionalities related to management and control, such as isolating DockPipes, setting flow directors to the hardware queues for directly forwarding, and constructing fast channels between specific microservices.

3.1 DockPipe

A DockPipe is essentially a lightweight networking interface for a container. Here we describe a DockPipe's data structure, attaching, lightweight data path, isolation, and fast channels.

Data Structure: Figure 2 shows that a DockPipe only consists of three types of ring buffers: a receiving ring buffer, a transmitting ring buffer, and some fast channel (FC) ring buffers. The receiving buffer and the transmitting buffer are directly pinned to a pair of hardware queues. An FC ring buffer is used to construct a fast channel. By using a fast channel, a container can directly put packets into the receiver's peer FC ring buffer.

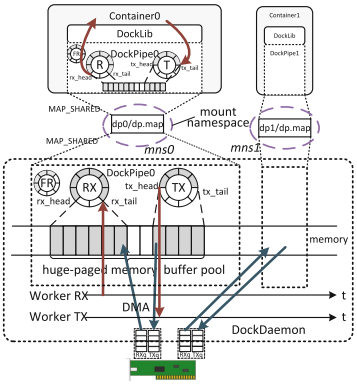


Fig. 2. Structure and isolation of a DockPipe.

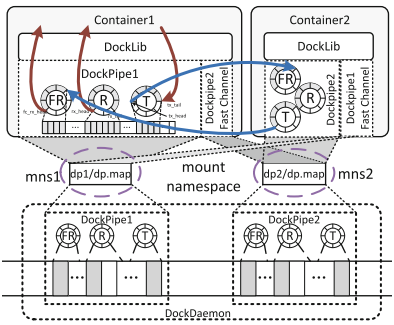


Fig. 3. Sharing of DockPipes and the structure of a fast channel.

DockPipe Attaching: The attaching of a DockPipe is controlled and managed by DockDaemon. As Fig. 2 shows, When creating a DockPipe, DockDaemon generates a mapping file which records the memory layout and offset of the DockPipe. The container attaches to the DockPipe by loading and remapping (using `mmap()`) the memory mapping file. After attaching, the container can directly send or receive packets through the DockPipe. Through a DockPipe, a containerized microservice is only allowed (1) to put packets into the buffer. (2) to read packets from the buffer. The DockDaemon is responsible for real packet moving, such as transmitting packets to the NIC and receiving packets from the NIC.

High-Performance Data Path: A DockPipe uses DPDK as the high-performance userspace packet I/O data plane. As shown in Fig. 2, a DockPipe is directly pinned to a pair of hardware queues. DockDaemon sets hardware-based filters (e.g., destination IP address) to the NIC. When packets arrive at the NIC, the pre-configured filters (e.g. destination IP address) will guide the NIC and redirect the packets to a specified hardware queue. Then, the packets are directly put into the corresponding DockPipe's receiving ring buffer through DMA.

DockPipe Isolation: When deploying multiple containerized microservices on a single host, the security concern is critical: A DockPipe should be protected and provides an isolated data plane for the granted container. Here we describe how to isolate multiple DockPipes.

Since a DockPipe is remapped to a container by `mmap()` based on memory sharing, we can conveniently isolate DockPipes by controlling their scopes of shared memory. In container-related techniques, the ***mount namespace*** offers this capability. The mount namespace of a process is essentially the set of mounted filesystems in its perspective. Conventionally, mounting or unmounting a filesystem will change all processes' scope because a global mount namespace can be seen by all processes. In the container context, we can utilize the

per-process mount namespace. Then a process can have its exclusive scope of mounted filesystems. Based on the per-process mount namespace, DockNet provides a lightweight isolation for DockPipes.

Take Fig. 2 for example, DockDaemon (1) generates *DockPipe0's* memory mapping file as *dp0/dp.map*. (2) mounts file dictionary *dp0* (in mount namespace *mns0*) to *container0* by leveraging the mount namespace. In this way, *DockPipe0* can only be accessed by *container0* and is invisible to *container1*.

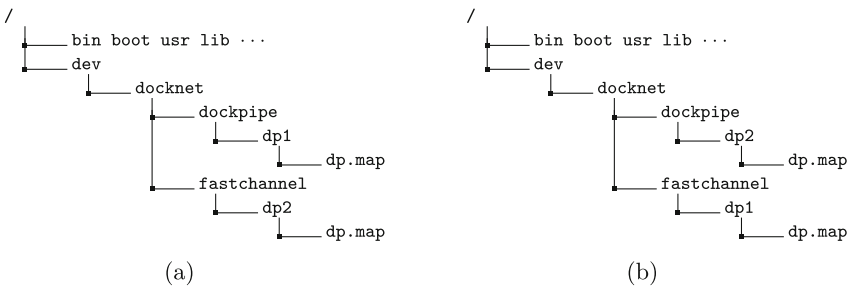


Fig. 4. Mounted filesystems (a) perspective of *container1*. (b) perspective of *container2*.

Fast Channel Between DockPipes: In some particular scenarios (such as private clouds) strict isolation is unnecessary. For example, partner containerized microservices belong to the same application and locate on the same host. In such scenarios, DockNet can offer higher throughput and lower latency for containerized microservices by constructing a fast channel.

We have explained how to use the mount namespace for controlling the scopes of shared memory. By default, a DockPipe's memory mapping file is mounted to a single container, namely a DockPipe is monopolized by a container. However, a DockPipe can also be remapped into two containers. In this way, a fast channel (FC) between two containers can be constructed.

As illustrated in Figs. 3 and 4, a DockPipe can be remapped as either a normal DockPipe or an FC, respectively. When a DockPipe is remapped as an FC, only its FC receiving ring buffer is used. When a packet is ready to be sent, the container first checks whether it can be sent through an FC. If so, the container sets the packet's pointer to the tail of the FC's receiving ring buffer. Otherwise, the container sends the packet to its DockPipe's transmitting ring buffer as usual. For instance, in Fig. 4, *DockPipe1* is remapped to *container1* as a normal DockPipe, and remapped to *container2* as an FC. When *container2* sends packets to *container1*, the packets' pointers are directly put into the tail of *DockPipe1*'s FC receiving ring buffer. Then, *container1* can directly consume the packets from the head of *DockPipe1*'s FC receiving ring buffer. For *DockPipe2*, the receiving process is the same except for exchanging *container1* and *container2* in sender/receiver locations.

3.2 DockDaemon

DockDaemon offers management and control functionalities. The management functionalities include assigning hardware queues to DockPipes, setting flow directors for directly forwarding, isolate DockPipes, and (de)constructing fast channels. Their technical essences have been described in Sect. 3.1.

In this section, we focus on the how to control DockPipes in DockDaemon.

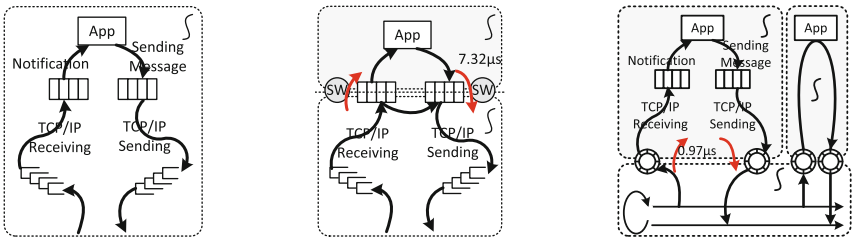


Fig. 5. (a) shows the routine of the run-to-completion execution model. (b) shows that of the multi-stage wheels execution model. (c) shows DockNet's master-slave execution model.

Threading Model: Threading model greatly affects the system performance. Here we summarize the two existing models and propose the master-slave model.

(i) *Run-to-completion model:* As shown in Fig. 5(a), this model integrates packet receiving/transmitting, protocol processing, and application logic into a single executing routine. Due to eliminating unnecessary intermediate stages (e.g. context switches), this model can achieve low latency. However, this model requires privileged access to the physical NIC, which may result in security risks and increase the managing and controlling difficulty. IX [9] adopts this threading model, and its data plane optimizes for both bandwidth and latency.

(ii) *Multi-stage wheels model:* As Fig. 5(b) shows, in this model, each pair of threads are pinned to the same CPU core. One thread is responsible for processing packets in the application. The other thread is devoted to receiving and sending packets. The decoupling can protect the physical NIC from being directly accessed by the containers. However, the decoupling introduces costs of thread context switches (the red arrow in Fig. 5(b)) and incurs higher latency. This model is adopted by mTCP [13], which is specialized for handling short messages on multicore systems. Besides that, the similar threading model is also applied to the kernel's stack.

(iii) *Master-slave model:* Inspired by the two models above, we design a master-slave model for containerized microservice networking. As shown in Fig. 5(c), in this model, there is one master thread and many slave threads.

The slave thread acts like the run-to-completion model and integrates all related processing into a single executing routine. The master thread is devoted to really moving packets, including sending packets to the physical NIC, receiving packets from the physical NIC, and copying packets between containers. The master and slave threads decouple the whole packet routine into two sub-stages. Because all the packets moving is done by the master thread, it simplifies the control and management, such as limiting the bandwidth. The decoupling also introduces extra overheads. To avoid high costs caused by thread switching, we let the master thread monopolize a core. The slave threads use the other cores located on the same CPU socket. In Fig. 5(c), the two red arrows mean the synchronization and mutual exclusion that based on atomic operations.

By using the master-slave model, a DockPipe is only allowed to put packets into or get packets from its ring buffer. DockDaemon controls all the packets sending and receiving. The master-slave model can both achieve low latency and simplify the controlling.

Timer-Based Rate Limiter: To achieve bandwidth isolation, we design a simple but efficient mechanism based on timer-based rate limiter in DockDaemon. Two timers are introduced. One is fine-grained T_{fine}, and the other is coarse-grained T_{coarse}. They are triggered at different intervals. The fine-grained timer produces tokens for each DockPipe, and the number of tokens constrains how many bytes a DockPipe can send or receive. However, since a container may keep silent for an extended period, it likely accumulates much more tokens than others. Then it may consume too much bandwidth when it becomes active. To solve this problem, the coarse-grained timer periodically clears all the tokens and sets an initial number of tokens intermittently.

3.3 APIs

DockNet provides two types of APIs: DockDaemon commands and DockLib. DockDaemon commands are received by DockDaemon through a kernel-based listening socket. These commands are used for managing the DockDaemon, such as creating or terminating a DockPipe, assigning a DockPipe to a containerized microservice, limiting a DockPipe's bandwidth. DockLib is used for containerized microservices to access the DockPipe and develop applications. DockLib contains a packet engine that interacts with DockPipe and a lightweight userspace TCP/IP stack based on LwIP [10]. Figure 7 presents the skeleton of using DockNet to write a TCP server containerized microservice.

Packet Engine: DockNet uses DPDK as the high-performance packet I/O data plane. After remapping, a DockPipe offers its transmitting and receiving queue for a container. Packet engine mainly includes two components: the sending component and the receiving component. In the sending component, when packets from the stack are sent to DockPipe, the engine first checks their destination IP address. (1) If a packet's destination IP matches a fast channel, the packet is

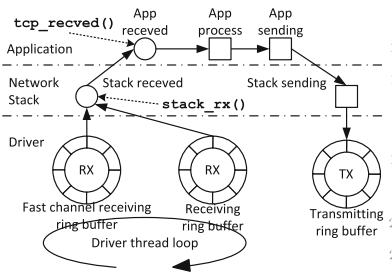


Fig. 6. Packets processing routine. Circle means a callback function that needs to be registered by the upper level code. Box means a normal function call.

```
err_t app_process(struct tcp_pcb *tcb, struct mbuf *m)
{
  // the processing details
  tcp_sent(tcb, on_sent);
  tcp_write("OK!")
}

err_t on_accecpted(struct tcp_pcb *tcb)
{
  tcp_recv(tcb, app_process);
}

int main(int argc, char **argv)
{
  struct dockpipe *dp = dockpipe_attach("/dev/docknet/dockpipe/dp0",
        ethernet_input);
  tcp_pcb tcb = tcp_new();
  tcb = tcp_listen(tcb);
  err_t err = tcp_bind(tcb, IP_ADDR_ANY, 80);
  tcp_accept(tcb, on_accecpted);
  sys_thread_new("dockpipe_driver_thread",
    dockpipe_driver_thread, NULL,
    DEFAULT_THREAD_STACKSIZE, DEFAULT_THREAD_PRIO);
  pause();
  return 0;
}
```

Fig. 7. Skeleton of a TCP example.

directly appended to the remote peer's FC receiving ring buffer. (2) If a packet's destination IP matches no fast channels, the packet is appended to the DockPipe's transmitting ring buffer. In the receiving component, when receiving packets, the engine first tries to receive packets from its FC receiving ring buffer, then goes to receive packets from its receiving ring buffer.

As shown in Fig. 6, the packet engine exposes a function pointer `stack_rx()`, through which the received packets can be sent to the upper network stack. The packet engine thread repeatedly checks the two receiving ring buffers. After receiving a packet, the engine delivers it to the stack through the callback function.

Userspace TCP/IP Stack Library: Since DockPipe's ring buffers are all located in userspace, DockNet needs to provide a userspace TCP/IP stack for containerized microservices. In DockNet, we port the stack from LwIP. We mainly modify its data structures to achieve zero-copy and be adapted to DockNet. Here we briefly describe the key procedures of using DockNet to write a TCP server code in a container. When the server starts, it first executes initializations. Then the server registers `stack_rx()` callback function, which is a function pointer inside DockPipe. Next, the server registers `tcp_recved()` callback function, which is a function pointer in DockLib. Finally, the server launches a new packet engine thread and runs it, then blocks the main thread. Both `stack_rx()` and `tcp_recved()` will be called in the engine's thread context, which avoids introducing context switching overhead.

4 Evaluation

In this section, we evaluate DockNet with the following goals:

(i) Demonstrate that DockNet can provide high throughput and low latency in both inter-host and intra-host communications.
(ii) Validate DockNet's ability to guarantee performance isolation among microservices.
(iii) Show that DockNet can support multiple containerized microservices.
(iv) Construct a web cluster based on containerized microservices to evaluate the integrated performance of our DockNet.

Testbed: Our testbed consists of two machines, and each has two 6-core CPUs (Intel Xeon E5-2620 v3 @ 2.4 GHz), 64 GB RAM, and an Intel NIC with two 10GbE ports. The NICs of the two machines are connected directly. By default, we only use one 6-core CPU and one NIC port in the following experiments. Our DockNet runs on Linux 3.10.0. TSO is turned on in kernel-based evaluations. Each container is pinned to one single core.

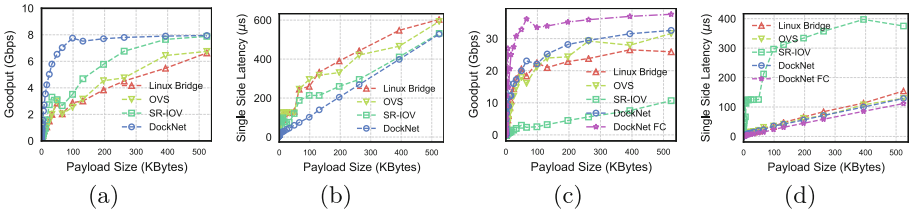


Fig. 8. The basic performance benchmark of DockNet. In (a) and (b), two microservices are deployed in two hosts. In (c) and (d), two microservices are deployed in the same host. (a) and (c) show the goodput. (b) and (d) show the singe side latency.

4.1 Throughput and Latency

High throughput and low latency are critical requirements for containerized microservices.

We construct a pair of TCP microservices, one being the server and the other being the client, and deploy each microservice in a container. The two applications directly exchange a fixed-size message back and forth. Two primary metrics – goodput and one-way latency are measured. The reference networking solutions include Linux bridge, OVS, SR-IOV, and DockNet. Figure 8 shows the two primary performance metrics of four networking solutions under different message sizes.

In Fig. 8(a) and (b), the two microservices (server and client) are deployed in two different physical hosts. In this case, DockNet shows both the highest goodput and the lowest one-way latency. Obviously, when the message size grows to

100 KB, the link utilization of DockNet reaches 8 Gbps (i.e. line rate), while the other three are still below 8 Gbps even when the message size equals 400 KB. From the perspective of latency, DockNet also has lower one-way latency than the other three solutions under all message sizes. It is because kernel bridge, OVS, and SR-IOV provide kernel-based interfaces, which inevitably introduces overheads, such as context switching and data copying between kernel and userspace. Besides, kernel bridge and OVS also introduce the software switching cost. Instead, DockNet leverages a user-level DockPipe to exchange data, thus eliminates both data copying and context switching.

In Fig. 8(c) and (d), the two microservices are deployed in the same physical host. SR-IOV shows inferior performance. Goodput is always below 8 Gbps, and one-way latency reaches up to 400 µs, which is far worse than other solutions. The reason is that SR-IOV has to switch packets through the physical NIC, and introduces the transmission cost over PCIe compared with other solutions. The other four networking solutions all achieve superior goodput and latency, and our DockNet and DockNet with FC achieve further better performance than kernel bridge and OVS. We can see that, as the message size increases to 200 KB, the goodput, of all solutions except for SR-IOV exceeds 22 Gbps, our DockNet reaches over 27 Gbps, and DockNet FC achieves more than 35 Gbps. Both Kernel bridge and OVS forward packets in the kernel, hence introduce data copying and context switching overheads. However, DockNet based on the forwarding table eliminates the context switching, and DockNet with FC removes both context switching and data copying, therefore outperforms than other networking.

In summary, SR-IOV performs poorly in one-host scenarios but performs well in two-host scenarios. Kernel bridge and OVS have good performance in the one-host condition but still perform poorly in two-host tests. DockNet shows excellent performance in both one-host and two-host tests.

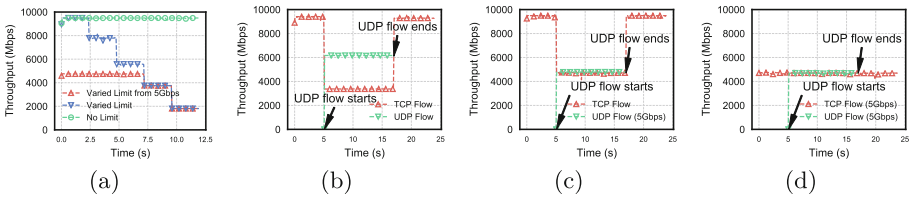


Fig. 9. Rate limiter tests. (a) shows the ability to dynamically change the rate limiter. (b), (c) and (d) shows different policies on two DockPipes.

4.2 Bandwidth Isolation

As stated above, bandwidth isolation is necessary for containerized microservices to avoid aggressive microservices consuming too much bandwidth. Next, we demonstrate DockNet's ability to limit microservices' maximal bandwidth. We construct a pair of TCP client/server microservices as well as a pair of UDP

client/server microservices. The four microservices are deployed in four different containers, respectively. Two server and clients are arranged in one host, respectively. During the test, two servers keep sending packets to their corresponding receivers by a TCP flow and a UDP flow separately. Figure 9 depicts the bandwidth usage under different rate limitations.

Figure 9(a) presents the results of two tests: (1) Without any rate limiters, the throughput of the only TCP flow is evaluated as the baseline. The TCP throughput is depicted as the blue (delta dotted) line. (2) After starting containers, periodically change the rate of the limiter according to the value list {8000, 6000, 4000, 2000} (Mbps). The TCP throughput is described as the orange (square dotted) line. Clearly, with the rate limiter, the flow throughput rapidly changes in line with the rate limits, which validates the effectiveness of the rate limiter.

In the latter three subgraphs in Fig. 9, we first run the TCP flow, then start the UDP flow at the 5th second and terminate it at the 17th second. Figure 9(b) manifests the results without rate limiters, indicating that the UDP flow consumes the majority of bandwidth and nearly beats down the TCP flow's throughput. In Fig. 9(c), only the UDP flow's rate limiter is set to be 5 Gbps, in which case the TCP flow can still occupy half of the whole bandwidth. In Fig. 9(d), we set all DockPipes' rate limiters to be 5 Gbps, and the result demonstrates that the bandwidths of flows are appropriately isolated.

From above experiments, we show that DockNet can indeed limit the maximum bandwidth of a microservice and further enable the bandwidth isolation, in both single-flow and multi-flow scenarios.

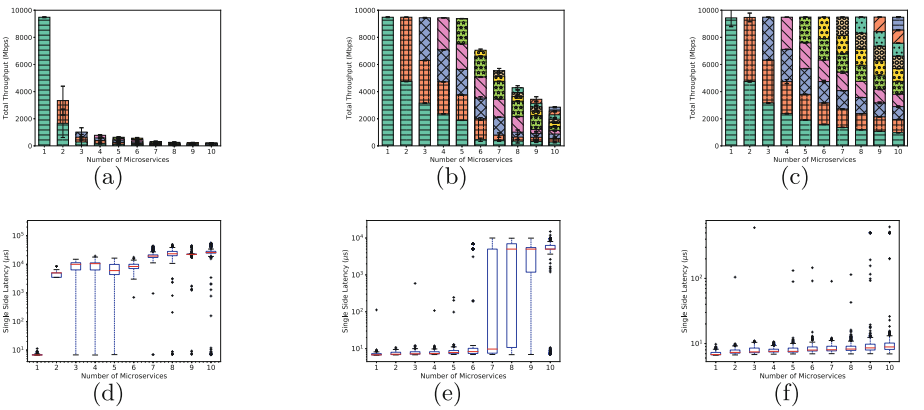


Fig. 10. Multiple microservices tests. (a) and (d) describes the total throughput and latency results when microservices runs on one slave core. (b) and (e) describes the results when microservices runs on one socket (5 slave cores). (c) and (f) describes the results when microservices runs on two sockets (10 slave cores).

4.3 Multi-container

It is common that multiple microservices are deployed on one host, in which case the underlying networking has to support multiple containers. In this subsection, we investigate DockNet's performance when supporting multiple containerized microservices.

In Fig. 10, we deploy multiple microservices in two physical hosts: one as the server side and the other as the client side. Each microservice is assigned a DockPipe, and all DockPipes are limited to the same rate by rate limiters according to the number of containers. DockDaemon dedicates one master core, while microservices are deployed on slave cores. When measuring the throughput, each client keeps sending 1024B message to a server. In one-way latency tests, each pair of server and client keeps exchanging a 1024B message back and forth. In our experiment, microservices are respectively deployed on one slave core, one socket (5 slave cores) and two sockets (10 slave cores) to observe DockNet performance with the different number of cores. In the one-core and one-socket cases, DockDaemon uses one core as the master core, while in the two-socket case, two cores are used as master cores (each NUMA-node owns a master core).

Figure 10(a) and (d) show the throughput and one-way latency when microservices are deployed on one core, respectively. Clearly, the overall throughput sharply decreases as the number running microservices grows. Similarly, the one-way latency drastically increases from 6.4 µs to 7 ms as the number of containers grows from 1 to 2. Such performance degradation is mainly caused by the container scheduling. Worse still, both the server and client will produce such scheduling costs. Because the time slice is ranging from 3 ms to 24 ms in our testbeds, such scheduling overheads are considerable and cause drastic performance loss.

In Fig. 10(b) and (e), microservices are deployed on one socket (5 slave cores). When the number of microservices is under 5, the total throughput always keeps at about 9.4 Gbps, and each microservice's latency maintains at about 6.4 µs. However, when the number of microservices is larger than 5, the scheduling cost is introduced. Take the 6-microservice case for example. The slave cores are $core_i$ ($i = 1...5$), two microservices are deployed on $core_1$, and the other four microservices are deployed on the rest four slave cores, respectively. From their bandwidth usage, we can see that the scheduling cost only occurs on $core_1$, because there exists scheduling between two microservices on $core_1$.

In Fig. 10(c) and (f), microservices are deployed on two sockets (10 slave cores). DockDaemon employs two master cores located in two NUMA nodes. It is shown that in the condition of no scheduling, the total throughput is always 9.4 Gbps, and all the microservices achieve low latency (about 8.7 µs).

As our benchmark microservices are just receivers and echo servers, their cost of processing a packet is at most 54.83 ns (only including consume the packet in the application logic, excluding the TCP/IP processing). Nonetheless, real workloads can spend more CPU cycles on processing received messages, which relieves the I/O pressure. In these experiments, DockNet can properly work in the actively stressful condition where containers almost do not spend any cycles

on data processing. The experimental results in this subsection confirm that DockNet is lightweight and hardly becomes the bottleneck in real workloads.

4.4 Application

In this subsection, we study the performance of DockNet used in real containerized microservices. As shown in Fig. 11(a), we assume that the web server's processing power is the bottleneck, so we can scale the web cluster's processing power by simply adding more web servers. The DockNet holds five microservices. One acts as a NAT-based load balancer. The other four microservices all construct fast channels with the NAT container.

Network Address Translation (NAT): The NAT-based load balancer provides the ability that there is no need to change the client's behaviors because it can alter the destination IP into inner IPs. It introduces overheads because of modifying packets and searching lookup tables. To measure the overheads, we deploy a NAT microservice and a TCP server microservice on the server host. The client runs on another host. The server and the client behave similarly to those tests in Sect. 4.1. In the NAT-enabled test, the client sends packets with 1-Byte payload to the NAT, then the NAT transfers their destination IP addresses and forwards them to the server. When the server sends packets to the NAT, the NAT transfers the source IP addresses and sends them to the client. The results shown in Fig. 11(b) indicate that NAT microservice only add about 1.05 µs to the single side latency.

Simple Web Server: We port `httpd` to DockNet as the simple web server microservice and measure its performance. To avoid disturbance from disk access, we use `makefsdata` to transfer HTML files to C arrays. The HTML-content C codes can be directly compiled to the server. In this way, after processing HTTP request, the server can directly find the content of files located in the C arrays. One `httpd` microservice is deployed on the server host. On the client host, we use `wrk` [11] to saturate the `httpd` server. The results presented in Fig. 11(c) show that `httpd` running on DockNet can achieve the perfect performance. DockNet improves 5.2×, 6.3×, 5.5× performance gain compared with kernel bridge, OVS, and SR-IOV, respectively.

Web Cluster: To scalable the performance of processing HTTP requests, we use NAT as the load balancer to distribute requests to multiple `httpd` containers. We deploy a NAT microservice and multiple `httpd` microservices on the server host, and add some computing code to mimic processing cost (about 10 µs). `wrk` is deployed on the client host to saturate the `httpd` servers. The performance of the web cluster employing DockNet is shown in Fig. 11(d). It can be seen that the total requests per second scale when deploying more `httpd` containers. The performance scales 1.92×, 2.61×, 3.46× when deploying 2, 3, 4 `httpd` microservices comparing to a single `httpd` microservices. It validates that DockNet can be easily adopted in the real containerized microservices.

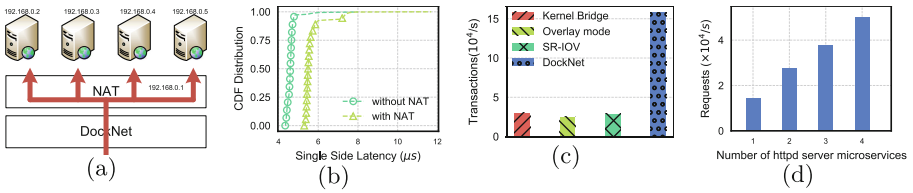


Fig. 11. Real applications tests. (a) shows the configuration of the tests. (b) shows the latency introduced by NAT. (c) shows the `httpd`'s transaction performance. (d) shows the DockNet can scale the performance of the web-cluster.

5 Conclusion

DockNet proposes a novel high-performance userspace networking solution for containerized microservices. DockNet splits the receiving/transmitting routine into two sub-stages and proposes a master-slave execution model to achieve both high performance and easy management, and provides a light-weight namespace-based access control mechanism to isolate microservices. Meanwhile, microservices mutually trusted can construct fast channels to achieve higher performance. In experiments, we verify that DockNet can provide high throughput and low latency in both intra-host and inter-host traffics, effectively support bandwidth isolation and multiple microservices. In transactions testing, DockNet shows over 4.2−52.4× of higher performance compared with existing conventional networking solutions. Finally, we build a DockNet-based web cluster and demonstrate its almost linear scalability.

Acknowledgments. The authors gratefully acknowledge the anonymous reviewers for their constructive comments. This work is supported in part by Suzhou-Tsinghua Special Project for Leading Innovation.

References

1. Open vSwitch (2009). http://openvswitch.org/. Accessed 2 June 2018
2. An Introduction to SR-IOV Technology (2011). http://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html. Accessed 2 June 2018
3. Netflix: A Microscope on Microservices (2015). http://techblog.netflix.com/2015/02/a-microscope-on-microservices.html. Accessed 2 June 2018
4. Amazon EC2 Container Service (2018). https://aws.amazon.com/ecs/. Accessed 2 June 2018
5. Intel DPDK: Data Plane Development Kit (2018). http://dpdk.org/. Accessed 2 June 2018
6. Microsoft Azure Container Service (2018). https://azure.microsoft.com/en-us/services/container-service/. Accessed 2 June 2018
7. Packet i/o engine (2018). http://shader.kaist.edu/packetshader/io_engine. Accessed 2 June 2018

8. PF_RING (2018). http://www.ntop.org/products/packet-capture/pf_ring/. Accessed 2 June 2018
9. Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., Bugnion, E.: Ix: a protected dataplane operating system for high throughput and low latency. In: OSDI 2014, No. EPFL-CONF-201671 (2014)
10. Dunkels, A.: lwIP - A Lightweight TCP/IP stack (2002). https://github.com/vadimsu/ipaugenblick. Accessed 2 June 2018
11. Glozer, W.: wrk- A Modern HTTP Benchmarking Tool (2012). https://github.com/wg/wrk. Accessed 2 June 2018
12. Han, S., Marshall, S., Chun, B.G., Ratnasamy, S.: Megapipe: a new programming interface for scalable network i/o. In: Presented as Part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12), pp. 135–148. Hollywood, CA (2012)
13. Jeong, E., et al.: mTCP: a highly scalable user-level TCP stack for multicore systems. In: NSDI 2014, pp. 489–502 (2014)
14. Musser, J.: KPIs for APIs (2014). http://www.slideshare.net/jmusser/kpis-for-apis. Accessed 2 June 2018
15. Pesterev, A., Strauss, J., Zeldovich, N., Morris, R.T.: Improving network connection locality on multicore systems. In: Proceedings of the 7th ACM European Conference on Computer Systems. EuroSys 2012, pp. 337–350 (2012)
16. Reinhold, E.: Rewriting Uber engineering: the opportunities microservices provide (2016). https://eng.uber.com/building-tincup/. Accessed 2 June 2018
17. Rizzo, L.: Netmap: a novel framework for fast packet i/o. In: 2012 USENIX Annual Technical Conference (USENIX ATC 12), pp. 101–112. Boston, MA, June 2012
18. Soares, L., Stumm, M.: Flexsc: Flexible system call scheduling with exception-less system calls. In: Proceedings of the 9th USENIX Conference On Operating Systems Design and Implementation, pp. 33–46. USENIX Association (2010)