vSocket: Virtual Socket Interface for RDMA in Public Clouds

Dongyang Wang*
University of Science and Technology of China
China
Huawei Technologies Co., Ltd
China
dyw@mail.ustc.edu.cn

Binzhang Fu
Huawei Technologies Co., Ltd
China
fubinzhang@huawei.com

Gang Lu
Huawei Technologies Co., Ltd
China
lugang3@huawei.com

Kun Tan
Huawei Technologies Co., Ltd
China
kun.tan@huawei.com

Bei Hua
University of Science and Technology of China
China
bhua@ustc.edu.cn

Abstract

RDMA has been widely adopted as a promising solution for high performance networks, but is still unavailable for a large number of socket-based applications running in public clouds due to the following reasons. There is no available virtualization technique of RDMA that can meet the cloud's requirements. Moreover, it is cost prohibitive to rewrite the socket-based applications with the Verbs API. To address the above problems, we present *vSocket*, a software-based RDMA virtualization framework for socket-based applications in public clouds. *vSocket* takes into account the demands of clouds such as security rules and network isolation, so it can be deployed in the current public clouds. Furthermore, *vSocket* provides native socket API so that socket-based applications can use it without any modifications. Finally, to validate the performance gains, we implemented a prototype and compared it with current virtual network solutions against 1) basic network benchmarks and 2) the Redis, a typical I/O intensive application. Experimental results show that the latency of basic benchmarks can be reduced by 88% and the throughput of Redis is improved by 4 times.

*The work was performed during an internship at Huawei.

VEE '19, April 14, 2019, Providence, RI, USA

ACM ISBN 978-1-4503-6020-3/19/04…$15.00
https://doi.org/10.1145/3313808.3313813

CCS Concepts • **Software and its engineering → Virtual machines**.

Keywords Public Clouds, Virtualization, RDMA

ACM Reference Format:
Dongyang Wang, Binzhang Fu, Gang Lu, Kun Tan, and Bei Hua. 2019. vSocket: Virtual Socket Interface for RDMA in Public Clouds. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '19), April 14, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3313808.3313813

1 Introduction

With the emergence of data-intensive applications, such as the in-memory database and large-scale AI (Artificial Intelligence) systems, the efficiency of the network communication has become a major factor affecting overall system performance. Meanwhile, RDMA (Remote Direct Memory Access) offers higher throughput, lower latency and lower CPU overhead than the standard TCP/IP networking. Based on these observations, a lot of literature proposed to exploit RDMA to speed up their systems. For example, FaRM [13] and HERD [25] exploit RDMA to improve the performance of main memory distributed computing by an order of magnitude in both latency and throughput, and [38] tries to improve the performance of TensorFlow [11] by replacing the gRPC-based transport with a new proposed solution based on RDMA.

On the other hand, more and more applications are being migrated to clouds. Among them, socket-based applications account for the majority of the proportion. Meanwhile, cloud providers are constantly looking for higher performance network solutions. Unfortunately, although RDMA has been proved to be effective for abundant data-center applications, it is still challenging to apply RDMA for the

socket applications in the cloud, due to the following two reasons:

1) RDMA's virtualization solution does not meet the needs of the cloud environment: In the scenario of public clouds, there are requirements such as security rules (e.g. ACL [4]) and network isolation (e.g. VXLAN [27]). However, RDMA's current virtualization methods such as SR-IOV [10], HyV [30] do not meet these requirements. For example, when using SR-IOV, traffic from the VM is passed directly to the NIC (Network Interface Card) but the NIC hardware does not support VXLAN tunneling. AccelNet [16] uses an extra piece of hardware to solve the problems of SR-IOV such as VXLAN tunneling (by offloading the needs to the new hardware). However, it needs additional hardware.

2) Huge difference between Socket API and the Verbs API: It is cost prohibitive to rewrite socket applications with the Verbs API. To send and receive data through the RDMA network we need to use the Verbs [7] API, but most existing applications utilize the BSD socket API. The main difference between them is that they assume different programming semantics. For example, for the socket interfaces, applications could reuse the memory buffers as soon as the data is sent to the network. But for the Verbs API, applications can not reuse the buffers until the success notifications of *Work Requests* are explicitly received. Therefore, rewriting the applications with the Verbs API is not a straightforward job. Furthermore, there are many applications that are no longer actively maintained. Thus, a method to exploit RDMA to improve the performance of socket applications without modifying the source code is highly expected. To this end, VMA (Mellanox Messaging Accelerator) [28] has been proposed recently. It dynamically intercepts related calls to the socket API and translates them into Verbs operations. One major advantage of VMA is that the socket-based applications could use the RDMA network without any modifications to the source code. However, in order to ensure there is no packet loss, VMA embeds a TCP stack. This will result in high CPU processing overhead and low throughput. (Sec. 6.2.2, 6.2.3)

Therefore, the goal of *vSocket* is to provide an RDMA-based high-performance networking system that meets the public clouds' requirements for socket applications.

Particularly, to apply security rules to vSocket, we propose to reuse the kernel TCP connection instead of totally removing it. To this end, *vSocket* first establishes a TCP connection through the kernel stack as normal, after that a *vSocket* connection is created and mapped to an RDMA connection for the above TCP connection. Therefore, we could exploit this *vSocket* connection (actually the RDMA connection) to accelerate the data transmission in normal cases. Because the establishment process of kernel TCP connection obeys all existing security rules of public clouds, the *vSocket* connection can be created without violating any security rules. Furthermore, to provide the network isolation, *vSocket* connections are implemented based on the para-virtualized I/O scheme. Thus each packet can be tunneled in the backend. Other demands such as QoS are also easy to implement based on the para-virtualization I/O framework.

To provide an RDMA-based para-virtualized I/O framework for *vSocket* connections, a virtualized RDMA device should be realized. However, providing such a device will introduce extra maintenance work. Because upgrading physical RDMA devices or driver will inevitably lead to upgrades of the driver for the virtual RDMA device. Therefore, we recommend virtualizing RDMA connections instead of providing a virtualized RDMA device. Thus we only need a simple para-virtualized I/O device to exchange data between the frontend and the backend. In *vSocket*, an RDMA connection can be virtualized to multiple *vSocket* connections to avoid scalability issues in RDMA. Moreover, the RDMA device is driven through the standard Verbs API, so the upgrade of physical RDMA devices or driver is transparent to *vSocket*.

Finally, to provide a high performance *vSocket* connection. We eliminate the complicated TCP processing and the data copy between the frontend and the backend which is introduced by the para-virtualized I/O solution. Besides, other overheads such as system calls, VM exits and interrupts are also eliminated by adopting the user space driver and polling scheme.

To sum up, this paper tries to bring the benefits of RDMA networking to socket-based applications running in the cloud VMs. The contributions can be summarized as follows:

1. To the best of our knowledge, *vSocket* is the first software-based solution that provides a fully compatible interface over the RDMA network to socket-based applications in public clouds.

2. We propose to reuse kernel connections and virtualize RDMA connections to enable RDMA in the virtual environment, which is quite different from the conventional device virtualization.

3. We implemented the prototype and compared the performance with the state-of-the-art approaches. Evaluation results demonstrate that the connection virtualization technique is a promising solution to RDMA virtualization, especially for connection-oriented applications.

2 Overview

In this section, we will present the design principles of *vSocket* and show how it achieves the following goals: 1) zero modification to applications, 2) satisfying cloud requirements, and 3) high performance.

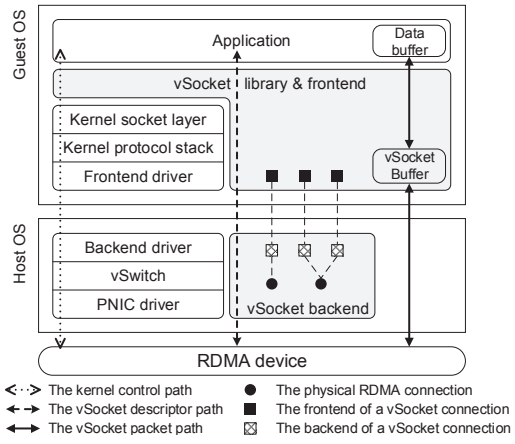


Figure 1. Architecture overview of *vSocket*.

The first goal of zero modification to applications indicates being compatible with the BSD Socket API. As shown in Figure 1, *vSocket* provides a user-level library running in the VM. The library is responsible for intercepting the invocations on the socket API[1]. More specifically, we leverage the LD_PRELOAD option to make the application load the *vSocket* library. Through that, socket related invocations are handled by the library, which is similar to VMA [28]. Note that some invocations to optimize performance, such as setting send buffer size, are ignored by *vSocket* since *vSocket* exploits a different transport protocol from TCP.

To obey the security rules (such as ACL [4], security groups [9]) in the cloud, we propose a novel hybrid way to boost the communication performance by reusing the kernel protocol stack to establish connections and only accelerating data transfer. As shown in Figure 1, after taking over the execution by intercepting *connect()*/*accept()* calls, *vSocket* will first establish a kernel connection, then establish a new *vSocket* connection for it. Since the establishment of the kernel connection obeys all existing security rules, the new *vSocket* connection can be established safely. Subsequently, data operations such as *send()*/*recv()* from the application will be accelerated through the *vSocket* connection.

Furthermore, in a cloud scenario, network isolation mechanisms (such as VXLAN [27]) are required to isolate traffic between different tenants. Since the RDMA NIC does not support VXLAN currently, the VXLAN encapsulation & decapsulation must be done in the software (or modify the hardware). Therefore, *vSocket* connections must be implemented based on para-virtualized I/O framework so that the encapsulation & decapsulation of data packets can be done in the host.

[1]Some file descriptor related operations such as *write()*, *fcntl()* and *epoll_wait()* are also intercepted.

Now, the question is how to provide a para-virtualized framework over RDMA. Traditional solutions, such as vRDMA [31] and HyV [30], try to provide a virtual RDMA device for VMs. But providing a fully-functional virtual RDMA device will inevitably increase maintenance costs. For example, HyV has to maintain a device/vendor dependent driver in both the frontend and the backend, making it difficult for the cloud provider to upgrade physical devices and driver. For *vSocket*, since the connection establishment and data transfer are separated, there is no need to provide a full-featured virtual device like HyV. So we proposed connection virtualization to solve the problem. As shown in Figure 1, *vSocket* maintains multiple *vSocket* connections and maps them to the physical RDMA connections. Meanwhile, since the *vSocket* backend drives RDMA via the Verbs API which is independent of the underlying devices and driver, the service provider could transparently upgrade their infrastructures.

Finally, providing high-performance *vSocket* connections is still a challenging job. To replace the kernel TCP connection, the basic functions of TCP should be maintained by the *vSocket* connection. TCP mainly provides two kinds of services, one is reliable transmission service and the other is congestion control. Since we use RDMA to transfer data and RDMA already provides congestion control [40], we no longer need to provide it. Although the RDMA NIC also provides reliable transmission, only transmissions over the physical network are guaranteed to be reliable by the RDMA NIC. A software virtual I/O device may also drop packets when its queue is full. A solution is to use another userspace TCP stack such as lwIP [14], but the stack processing in software will consume a lot of CPU and degrade the application's performance. Besides, the para-virtualization I/O will introduce extra overhead such as data copy, VM exit/entry. How to avoid these overheads is still challenging. We will address these challenges in Sec. 4.

3 Design

In this section, we will first introduce the definition of a *vSocket* connection. Then we will explain how to establish a *vSocket* connection. Finally, we will show how to transmit data through a *vSocket* connection. In these processes, we will see that the requirements of clouds are satisfied in the design of *vSocket*.

3.1 The *vSocket* Connection

As shown in Figure 1, there are three types of connections in *vSocket*. The first one is the traditional socket connection (TCP) which serves as a control path. The socket connection is established completely through the kernel as if we never intercept the invocations from the application. We call this connection **the kernel socket connection**. A kernel socket connection can be uniquely identified by a four-

tuple <*source IP*[2], *source port, destination IP, destination port*>. The second is **the *vSocket* connection** which is a virtual connection that serves as the fast data path. A *vSocket* connection consists of the frontend part (**the frontend connection**) in the VM and the backend part (**the backend connection**) in the host. The frontend connection is uniquely represented by the same four-tuple <*source IP, source port, destination IP, destination port*>, while the backend connection is represented by a five-tuple <*tenant ID*[3], *source IP, source port, destination IP, destination port*>. Because tenants may share the network address space, a tenant ID is necessary to distinguish connections belonging to different tenants. Between the frontend connection and the backend connection, data is exchanged through a para-virtualized I/O device. The third is **the physical RDMA connection** which is the real transmission path between two physical machines. In *vSocket*, a physical RDMA connection is built on a single RC (Reliable Connection) mode QP (Queue Pair).

It is notable that the physical RDMA connection is transparent to the VM. In the perspective of a VM, there is no RDMA connection at all, only one *vSocket* connection for each kernel socket connection. In the perspective of the host, multiple *vSocket* connections are mapped to one physical RDMA connection. That is to say, An RDMA connection is virtualized as *vSocket* connections. Therefore we can consider a *vSocket* connection as a *virtual connection* of a physical RDMA connection.

Usually, *vSocket* only maintains one RDMA connection between two physical machines to alleviate the scalability problem of RDMA. That means only the first *vSocket* connection establishment request needs to initiate the physical RDMA connection. For all subsequent connection establishment requests between the two machines, the job becomes to map the *vSocket* connection to the corresponding RDMA connection.

3.2 Establish a *vSocket* Connection

We will show the *vSocket* connection establishment process in this section. This process exhibits that the establishment of *vSocket* connection highly respects the security rules of clouds. We assume that the client and server are running in separate VMs on two physical machines. Therefore, to communicate with each other, the client should first establish a connection with the server. As shown in Figure 2 (Since the process on the other host is similar, we only draw one host to save space), the following steps are necessary to establish a *vSocket* connection.

[2]The source and destination IPs are those of the VMs. The IP of the host OS is transparent to VMs in clouds.

[3]A tenant ID can be the VXLAN VNI or some other identifications adopted by the cloud provider.

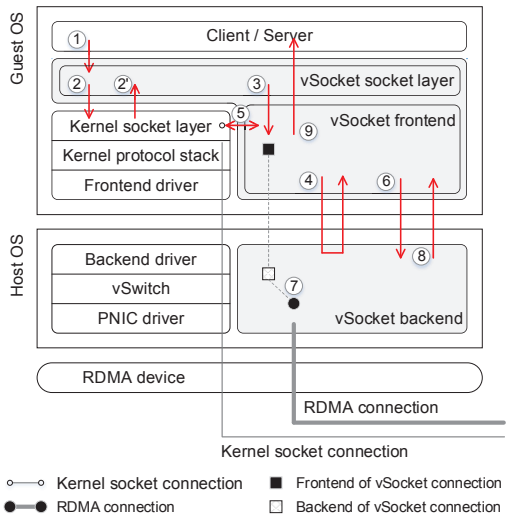


Figure 2. Steps to create a *vSocket* connection

Step 1: The client calls connect() and the server calls accept() both of which are intercepted by the *vSocket* library[4].

Step 2 and 2': *vSocket* forwards the invocation to the kernel to establish a kernel socket connection. After that, the control returns to the *vSocket* library. (If the kernel socket connection establishes successfully, it means that the connection request obeys the security rules of the cloud and we can establish a *vSocket* connection for it; otherwise *vSocket* will return a failure to the application. After connected, *vSocket* will keep the kernel socket connection alive by sending heartbeat packets regularly.)

Step 3: *vSocket* registers the frontend connection.

(After that the underlying RDMA connection between the corresponding two hosts should be established. In order to establish the RDMA connection, *vSocket* needs to know the peer host's IP address. However, the VM is only aware of the peer VM's IP address. The following three steps will show how the proposed *vSocket* solves this problem.)

Step 4: *vSocket* queries the IP address of the local host from the backend. (Note that in order to hide the host IP from the VM, the backend will encrypt the IP address.)

Step 5: *vSocket* exchanges the encrypted host IP address with the remote library through the kernel socket connection established in step 2.

Step 6: The following three types of information are forwarded to the backend, including 1) the four-tuple of the frontend connection, which is <*source IP, source port, destination IP, destination port*>, 2) the encrypted IP address of the remote host and 3) the role, which is either *client* or *server*.

[4]We assume that the client and server have finished the preparation phase such as *socket()* and *bind()* through the kernel.

Step 7: The backend decrypts the peer host's IP address. In the client side, the backend will initialize an RDMA connection with the peer's backend if the RDMA connection is not established yet. After that, the backend connection identified by the five-tuple (i.e., *<tenant ID, source IP, source port, destination IP, destination port>*) is mapped to the physical RDMA connection.

Step 8 and 9: Return to the application.

3.3 Data Transfer Through the *vSocket* Connection

The *vSocket* connection is used as the fast data path of the inter-VM communication. In this section, we will discuss how to transmit packets through the *vSocket* connection without breaking the network isolation.

As shown in Figure 3, there are three kinds of threads that collaborate to transfer packets: the application (Client/Server) thread, the *vSocket* frontend polling thread and the *vSocket* backend polling thread. The details of the threading model will be discussed in Sec. 4.3 while this section focuses on how they work together to send and receive data. For all the queues in *vSocket* which store the descriptors of the data packets, we suggest lock-free FIFOs. Assuming that a client need to send a message to the server, the following steps are required.[5]

Step 1A, 1'A: The *vSocket* library takes over the execution by intercepting *send()* in the client and *recv()* in the server.

Step 2A, 2'A: The data is copied from the application buffer to the *vSocket* buffer, if there is available space in the sending window. The discussion of the sending window is in Sec. 4.2. On the other side, the server checks if there is data arrived.

Step 3A, 3'A: A *vSocket* header defined in Figure 4 is composed and prepended the payload in the *vSocket* buffer. After that, the *vSocket* library constructs and pushes a descriptor into the socket FIFO, where the descriptor contains the address of the packet. Note that *vSocket* maintains a socket FIFO for each *vSocket* connection. On the other side, the server finds that no data arrives and then blocks on the corresponding eventfd, waiting for data to arrive. (For the kernel socket, *recv()* will return success immediately if there is data in the receive buffer; else it will block waiting for data (blocking mode) or return failed with *errno=EAGAIN* (nonblocking mode). We assume the *recv()* is in blocking mode here.)

Step 4A and 5A: A send-command is issued to the CMD queue. The command contains the information about the descriptor to be send. Then the execution returns to the application. (Note that the *send()* will return immediately once the data is copied to send buffer in the kernel socket, no matter the *send()* is blocked or not. So *vSocket*'s *send()* operation should also return at this point).

Step 6F and 7F: Once the send-command is polled out from the CMD FIFO, the frontend polling thread will fetch the descriptor from the socket FIFO indicated by the send-command, then convert the virtual address in the descriptor into corresponding guest physical address and finally forward it to the virtio FIFO.

(Step 8-14 will show how *vSocket* forwards packets in the backend. Meanwhile, the network isolation is guaranteed by adding a tenant ID to each packet.)

Step 8B, 9B, 10B, and 11B: The backend polling thread polls out the descriptor from the virtio FIFO and exploits cuckoo hash [15] to look up the backend connection where the hash key consists of the four-tuple carried in the *vSocket* header and the tenant ID of the VM. After that, the corresponding physical RDMA connection can be determined by the backend connection. Then the packet, its descriptor and tenant ID are sent out through the RDMA connection using standard RDMA verbs API. (*vSocket* uses *RDMA WRITE* to transfer data. The address in the descriptor is already replaced by the remote host virtual address, the details on how to determine the remote memory address will be discussed in Sec. 4.1.)

Step 12B, 13B, and 14B: The newly arrived descriptor is polled out and forwarded to the backend connection determined by the *vSocket* header and tenant ID. Then the address in the descriptor is converted to the corresponding guest physical address. Finally, the descriptor of the packet is forwarded to the corresponding virtio FIFO.

Step 15F, 16F, and 17F: The descriptor is polled out from the virtio FIFO by the frontend polling thread, and the guest physical address is converted to guest virtual address. Then the descriptor is switched to the corresponding socket FIFO based on the 4-tuple in the packet header. After that, the corresponding eventfd is written to notify the application thread.

Step 18A, 19A, and 20A: Once waking up by the eventfd, the application thread will get the descriptor from its socket FIFO and copy the data from the *vSocket* buffer to the application buffer.

Step 21A, 22A, and 23A: A CREDIT is created and pushed into the socket FIFO, then a CREDIT-command is issued to the CMD FIFO. Finally, *recv()* returns to the application. Note that the CREDIT is used to manage the receive memories and the sending window which will be discussed in Sec. 4.

Step 24 and later: The procedure of sending the CREDIT is similar to sending a packet as discussed above, therefore we omit the detailed discussions here.

To sum up, *vSocket* has two distinct advantages. On the one hand, the TCP-related processing is removed from the critical path. On the other hand, in both send and receive path, *vSocket* avoids all unnecessary data copies except the

[5]Note that the uppercase alphabets *A*, *F*, and *B* behind the sequence number respectively denote the execution contexts of *the application thread (A)*, *the frontend polling thread (F)* and *the backend polling thread (B)*.

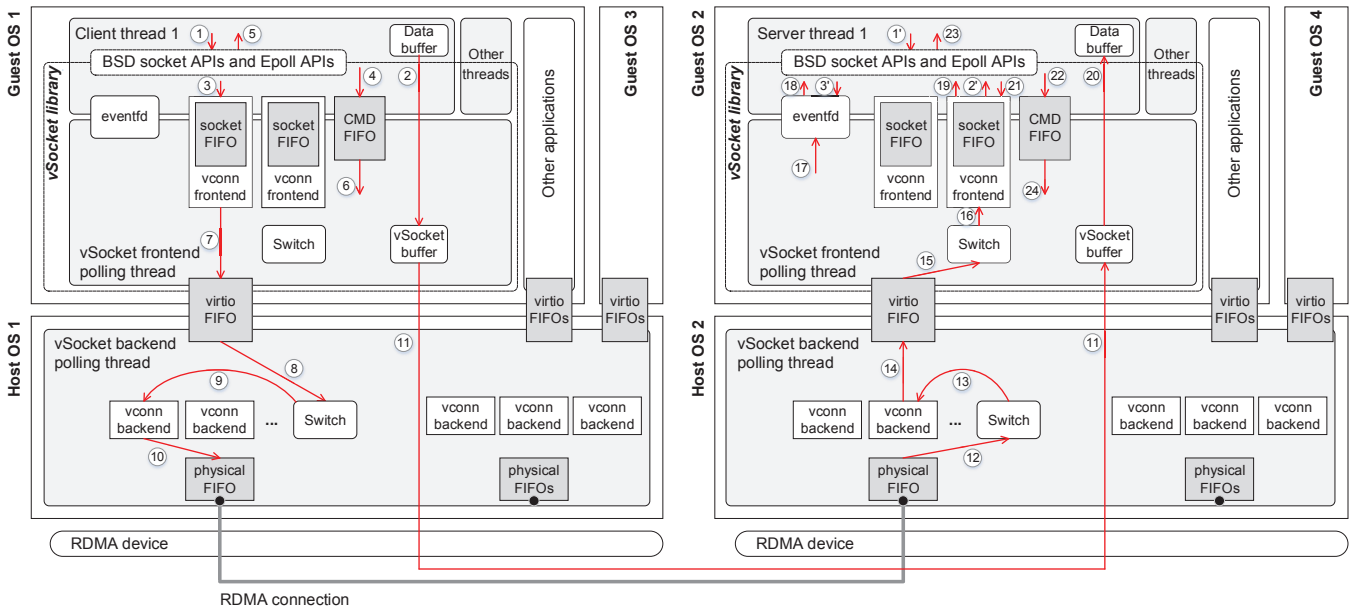


Figure 3. Steps to transfer data through virtual connections

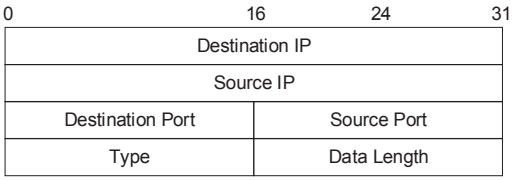


Figure 4. vSocket header structure.

one between the application buffer and the *vSocket* buffer to keep compatible with the BSD Socket API.

4 Performance Challenges

There are a few challenges to maximize the performance of the data path. In the section, we will identify three major challenges and discuss how to manage them.

4.1 One Copy

One major cost for transferring data is the data copy. Generally, zero-copy is expected to minimize the overhead. However, the socket programming model allows applications to reuse their buffers as soon as the data is successfully sent to the socket buffer. Therefore, data copies between the application and *vSocket* buffers are hard to avoid. Thus, the first challenge is reduced to achieving zero-copy between the local and remote *vSocket* buffers.

Since both the local and remote *vSocket* buffers reside in VMs, a strawman solution to transfer data requires three phases: copy data from the local *vSocket* buffer to a backend buffer, forward the data from the local backend buffer to the remote backend buffer through RDMA, and finally copy data from the remote backend buffer to the remote *vSocket* buffer. As a result, two copies are required between the *vSocket* buffer and backend buffer which may significantly degrade the performance of latency-sensitive applications especially for transmitting large messages [17, 34, 35]. Therefore, we are going to remove the above two copies.

In fact, the first copy in the sender side is easy to eliminate. We just need to ensure that the backend can access the *vSocket* buffer within the VM. It means that *vSocket* needs to provide a mapping between GVA (Guest Virtual Address), GPA (Guest Physical address) and HVA (host virtual address). To this end, we exploit huge pages to accommodate the *vSocket* buffer and map them into the address space of the backend process. The RDMA driver just needs the HVA to DMA data (there are page tables in the RDMA NIC), thus maintaining the mapping between HVA and HPA (host physical address) is no longer needed. However, providing such a mapping is not sufficient to eliminate the copy in the receiver side. Because the RDMA NIC does not parse the self-defined *vSocket* header and thus does not know which *vSocket* connections the incoming packets belong to. Therefore, a backend buffer is needed to temporarily store the packets. To address this problem, we ask the backend of the sender side to manage the receiver's *vSocket* buffers. Specifically, the memory for receiving packets are first created in the receiver side during the connection establishment phase. Then the receiver sends all the necessary information about the receiving memories to the sender side, including the base addresses and the sizes. The receiving memory can only be written (*RDMA*

WRITE) by the sender and read by the receiver. Therefore, the sender side could determine where to WRITE packets by itself. Furthermore, after a packet in the receiving memory is copied to the receiver's application buffer, a CREDIT will be created and sent back to notify the sender that the packet has been received and the corresponding receiving memory can be reused.

4.2 No Loss and HoL Blocking

To replace TCP, providing reliable transport service is necessary for *vSocket*. As shown in Figure 3, the RDMA data path between the local and remote *vSocket* buffers is reliable because *vSocket* exploits the reliable connection (RC) transport model. However, the *vSocket* descriptor path, i.e., the end-to-end path between the local and remote socket FIFOs, is not naturally reliable. Although RC transport model is adopted to guarantee the reliability between the local and remote physical FIFOs, the path between a socket FIFO and a physical FIFO is not reliable. For example, if a virtio FIFO is full, descriptors sent from socket FIFOs will be dropped. Similarly, if a physical FIFO is full, descriptors sent from virtio FIFOs will be dropped too.

Traditionally, a lightweight TCP-like protocol is utilized to provide end-to-end reliable services. For example, to this end, VMA embeds the lwIP [14] protocol stack. However, introducing an additional protocol stack will degrade the performance as discussed in 6. Since the physical connection is reliable, another strawman solution is to exploit lossless flow control in the software. For example, once a downstream FIFO is full, it will stop the transmission of all upstream senders. However, since a downstream FIFO is shared by multiple upstream FIFOs, the lossless flow control technique will lead to significant Head-of-Line (HoL) blocking problem [33].

To address the above problem, we design a flow control strategy to prevent the FIFOs from overflowing. For each *vSocket* connection, we add a sending window. The initial window size W is not in bytes, but in packets (since it is only used to avoid FIFO overflow). Whenever a packet is sent out, the window size of the connection will be decremented by one. When the window size is reduced to zero, sending process will be paused. In other words, the send operation on the socket will be blocked (in blocking mode) or failed with *errno=EAGAIN* (in non-blocking mode). When the receiver gets a packet, it will create and send back a CREDIT to the sender (The CREDIT is also used for memory management as described in Sec. 4.1). The CREDIT contains the connection information and the memory information of the packet just received. Upon receiving the CREDIT, the sender will increase the connection's window size by one. This leads to a simpler flow control design. Unlike TCP we do not need timers, retransmissions, etc.. According to the design, the total amount of in-flight packets/CREIDTs of each connection never exceeds the initial window size W. As a result, as long as the virtio FIFO and the physical FIFO is large enough ($W * C$, where C is the total number of connections linked to the FIFO), no packet/CREDIT will be lost in *vSocket*.

Unfortunately, the setting can consume a lot of memory. Supposing there are 1 million connections between two hosts and the maximum window size of each connection is set to 32, then the virtio/physical FIFO size should be 32M. However, considering the realistic bandwidth (B) and delay (D), the size of FIFO can be set smaller. To fulfill the bandwidth, we need $B * D$ memory in the host according to *Bandwidth-Delay Product* [23]. Let us denote the minimum packet size as $minPS$ (RDMA header + *vSocket* Header), so the physical FIFO size should be $B * D/minPS$. If $B = 100Gb/s$, $D = 10\mu s$ and $minPS = 100Bytes$, then the size of virtio FIFO and physical FIFO should be $B * D/minPS = 1.25K$, which is much less than 32 M. Furthermore, considering the CPU frequency limit, 100 Gb/s cannot be reached when small packets are transmitted. According to our test, when transmitting small packets through RDMA, the packet rate R is about 6 Mpps. So the FIFO size should be $B * D/PktSize = R * PktSize * D/PktSize = 60$. That is, when the delay is less than 10us, the FIFO size 60 can guarantee no packet loss. To be on the safe side, we set the size of virtio FIFO and physical FIFO to be 32768, which can tolerate about 5ms delay in the network. We believe that state-of-the-art congestion control algorithms [12, 19, 40] can easily control the network delay within this budget. Furthermore, to prevent packet loss due to high tail latency, we also added a backup system. The backup region will not overflow due to the existence of flow control. Since it has a maximum size of $C * W$.

4.3 Threading Model

In order to eliminate overheads such as interrupts, VM exits/entries, and system calls, we use polling threads for transmitting packet descriptors between the frontend and the backend as discussed in Sec. 3.3. In the host, the backend continues polling virtio FIFOs and physical FIFOs for new packet descriptors. Each VM usually has a virtio device, which contains multiple virtio FIFOs. A process in the VM usually requests one virtio FIFO, and a polling thread is created by the *vSocket* library to serve the application threads. Thus avoids polling from each application thread. The polling thread continues polling descriptors from the virtio FIFO and the CMD FIFOs. Because there may be a lot of socket FIFOs and many of them may be idle, polling empty socket FIFOs will waste CPU. To address this problem, we add a command FIFO between each application thread and the frontend polling thread. The command FIFOs and application threads are one-to-one mapped to avoid locks.

Basically, there are two types of commands in the command FIFO, *send descriptor* and *send CREDIT*. When a

application thread sends or receives a packet, it generates a descriptor/CREDIT and pushes it into the socket FIFO, then pushes a send descriptor/CREDIT command into the command FIFO. For each connection, we create an eventfd which is a file descriptor that can be used to wait for events. When the sending window size is decreased to zero (full) or increased to the maximum size (empty), the *send()/recv()* operation over this connection will be blocked on the corresponding eventfd (for blocking fd) or failed with *errno=EAGAIN* (for non-blocking fd). When the polling thread receives packet descriptors or CREDITs from the virtio FIFO, it will forward them to the socket FIFO. If the fd is blocked or the fd is waiting by an epoll instance, it will simply *write()* to the eventfd to wake up the blocked application thread. For the I/O multiplexing interfaces like *epoll_wait()*, they are also intercepted by *vSocket* and the kernel *epoll_wait()* is called to wait on the corresponding eventfd instead of the socket fd. After the kernel's *epoll_wait()* returns, *vSocket* library will convert the return value from eventfd to socket fd and then return it to the application. So the Socket API remains unchanged and the application can exploit *vSocket* without modification.

5 Discussions

Live migration: To support live migration, we propose a solution called connection switching, which means *vSocket* will switch to kernel connection for data transmission when migration happens. In AccelNet [16], Azure uses a similar approach to migrate RDMA connections. However, it requires applications to actively switch RDMA connections to kernel TCP connections. The main difference with AccelNet is that our connection switching is done by the *vSocket* library, which is transparent to the application. The design idea behind migration is simple. When migration occurs, *vSocket* will close the *vSocket* connection and fall back to the kernel connection to transfer data. After the migration completes, *vSocket* will create a new *vSocket* connection.

Communication between co-located VMs: For two VMs running on the same machine, the best communication performance can be achieved by shared memory. For example, The NetVM [21] facilitates data sharing among VMs by providing an emulated PCI device of huge pages to them. However, sharing memory by multiple VMs may cause security issues because a VM can access packets that do not belong to it. Thus *vSocket* does not adopt shared memory solution for co-located VMs. To exchange data, the *vSocket* backend directly copies the packets to the *vSocket* buffer and forwards the packet descriptors to the virtio FIFO of the destination VM.

Scalability: Since FIFOs are lockless and not shared by processes or VMs, the possible scalability problem, if exists, resides in shared polling threads. 1) When the number of physical machines is large, the number of physical FIFOs will increase, indicating more time is needed to poll all of the physical FIFOs. To prevent this problem, we can use *WRITE-IMM* instead of *WRITE* to write the descriptor through RDMA NIC. By using *WRITE-IMM*, the backend will be notified if new packets arrive. Moreover, destination physical FIFO can be directly determined by the immediate number taken with the RDMA request. 2) When the number of VMs inside a physical machine is large, the number of the virtio FIFOs will increase, indicating the backend poling thread needs to poll more virtio FIFOs. We can ease the bottleneck by adopting more polling threads in the backend. 3) When the number of processes inside in a VM is large, the number of polling threads in the VM will increase, indicating more CPU cores are occupied to poll the socket FIFOs. In fact, we set up a per-process polling thread with the purpose of obtaining the best performance, which is also widely adopted in DPDK [22] related projects. If we prefer to avoid the potential wastes of the CPU cycles in the scenario of abundant processes, a polling process instead of the per-process polling thread can be adopted. Similar to the polling thread, the interaction between the application process and the polling process can be achieved through eventfd and the CMD FIFO can be implemented with shared memory.

CPU polling: When the workload is low, it is wasteful to use a polling core, we are considering to use interrupt instead of polling in this scenario.

Other requirements in public clouds: There are many other requirements such as traffic shaping [37] and traffic statistics in public clouds. Since *vSocket* connection uses para-virtualization I/O, these requirements are easy to implement in the *vSocket* backend. For example, traffic shaping can be implemented by adding a token bucket [36] to each VM in the backend.

6 Evaluation

As shown in Figure 5, there are four typical approaches that are mostly adopted for inter-VM communication by socket-based applications: a) split device driver with vhost-net and kernel-OVS in host(*VM/Kernel-OVS*); b) split device driver with DPDK vhost-user and DPDK OVS in host(*VM/DPDK-OVS*); c) passthrough of RDMA NIC with RDMA libraries and VMA library(*SRIOV-VMA*); and d) passthrough of non-RDMA ethernet with TCP/IP software stacks(*SRIOV-Kernel*) [6] [7].

In this section, we answer the following questions by comparing *vSocket* with the above four approaches:

1. Latency: Does vSocket provide lower latency for socket connections? Sec. 6.1.1 shows that *vSocket* outperforms all

[6]Note that the SRIOV based approaches are not currently applied in public clouds.

[7]We only select approaches that support the BSD Socket API.

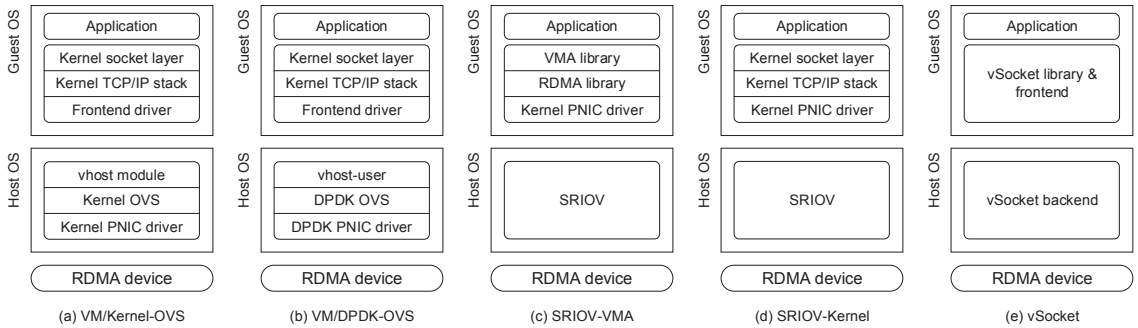


Figure 5. The software layout of *vSocket* and four other approaches in terms of inter-VM communication.

non-RDMA based approaches, *VM/Kernel-OVS*, *VM/DPDK-OVS*, and *SRIOV-Kernel*.

2. Bandwidth: Does vSocket provide higher bandwidth for socket connections? Sec. 6.1.2 shows that *vSocket* has the highest bandwidth. Even compared to *SRIOV-VMA*, the single flow bandwidth increases by nearly 2X.

3. Connection establishment overhead: How long does it take to establish a *vSocket* connection? Sec. 6.1.3 shows that the establishment time of a *vSocket* connection is only 488 μs longer than that of a BSD socket connection when the backend RDMA connections are already created. However, the extra overhead merely affects the overall application performance, especially for long-term connections.

4. Application performance: Does *vSocket* benefit real applications under different workloads? Sec. 6.2 shows that using *vSocket*, the average latency of Redis can be reduced by 85% and the throughput can be increased by nearly 4X.

Experiment setup: We deploy two VMs respectively on two homogeneous machines. Each machine is with one Mellanox ConnectX-4 adaptor with Virtual Protocol Interconnect (VPI), supporting 100 Gb/s InfiniBand and 100 Gb/s Ethernet connectivity [2], and two NUMA nodes each of which consists of one processor (Intel®Xeon®E5-2690 v4 @ 2.60 GHz) and 32 GB of DRAM (DDR4 @ 2400 MHz). The two RDMA NICs are directly connected by a single cable. We disabled Intel®Hyper-Threading and Intel®Turbo Boost to keep the experiment environments simplified and easy to analyze. For the two VMs, each machine is with 4 virtual CPU and 8 GB of memory. The virtual CPUs are pinned to the physical CPUs within a single NUMA node where the allocated memory resides and the RDMA NICs connects.

For the software, we deploy the modules and libraries according to Figure 5. The host OS and guest OS are using the same Linux, Ubuntu 14.04.5 with 4.4.35 kernel. We use the native KVM module and QEMU 2.10.0 for virtualization. The Mellanox NIC driver is MLNX_OFED_LINUX-4.1-1.0.2.0 and runs in RoCEv2 mode. For the benchmarks, we run clients and servers respectively on two VMs. We use VMA 8.4.3 in the SRIOV-VMA approach. OpenvSwitch 2.8.4 and DPDK 17.05 are deployed for the OVS approaches.

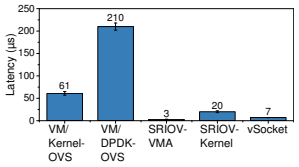


Figure 6. The RTT of *vSocket* and other approaches.

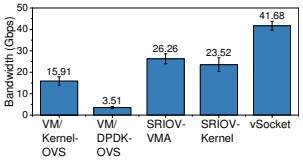


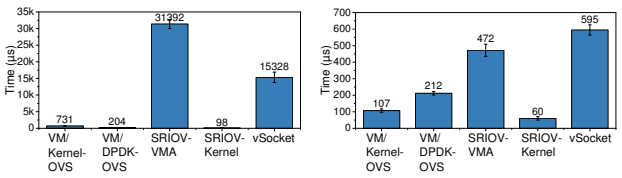
Figure 7. The bandwith of *vSocket* and other approaches.

6.1 Micro Benchmarks

In this section, we demonstrate the benefits of RDMA in terms of latency and bandwidth. We compare *vSocket* with the others using netperf [3], a popular micro-benchmark that can be used to measure the end-to-end latency and throughput of networking. Furthermore, we exploit a micro-benchmark to evaluate the overhead of connection establishment of different approaches. All the tests are repeated for five times and we show the average numbers in the figures.

6.1.1 Latency

To measure the best performance of the end-to-end latency, we set the message size to be only one byte and run netperf in the TCP_RR mode for 10 seconds. Figure 6 shows the RTT (round trip time) of all candidates. From the numbers, we can observe that it only takes 7 μs for *vSocket* to finish a round trip. And it is 61 μs for the *VM/Kernel-OVS* approach. That is to say, the RDMA based *vSocket* reduces the latency by more than 88% in this scenario. As discussed before, the time of *VM/Kernel-OVS* mainly spent on the processing of system calls, TCP processing, interrupts handling, VM exit/entry, etc. All these operations in the vSocket are eliminated. However, the *VM/DPDK-OVS* approach takes much more time than our expectation, we did some experiments around it and guessed that this was caused by the batching delay in the DPDK driver for the Mellanox ConnectX-4 VPI device. So we repeated the test with some background traffic to eliminate the delay. The



(a) The first connection (b) The subsequent connections

Figure 8. The establishment overhead of the connections between the client and the server which respectively resides in two VMs. The process includes the establishment of the backend RDMA connection if necessary.

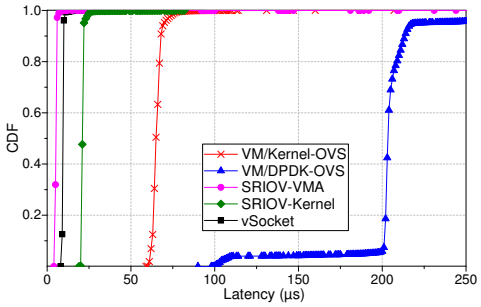


Figure 9. The CDF of response time over one connection using the native benchmark of Redis.

experimental results show that the real RTT of *VM/DPDK-OVS* is 43 μs, which is smaller than that of *VM/Kernel-OVS*. This is mainly because the backend interrupts are eliminated in the *VM/DPDK-OVS*. The *SRIOV-VMA* has lower RTT than *vSocket*. This is mainly because it uses the SRIOV virtualization framework while *vSocket* uses a para-virtualized framework to satisfy the requirements of public clouds. The *SRIOV-Kernel*'s RTT is 20 μs, mainly benefits from passthrough of the NIC.

6.1.2 Bandwidth

For the bandwidth, we run netperf for 1 minute in TCP_STREAM mode over one connection. Figure 7 shows the bandwidth of the five approaches. *vSocket* can achieve more than 40 Gb/s of bandwidth, which is respectively 1.6X, 1.8X, 2.6X, and 11.9X of *SRIOV-VMA*, *SRIOV-Kernel*, *VM/Kernel-OVS*, and *VM/DPDK-OVS*. We guess that it is still the batching delay within the DPDK driver that cause the worse performance of *VM/DPDK-OVS*.

6.1.3 Connection Establishment Overhead

Sec. 3.2 shows the process of establishing a full-fledged *vSocket* connection is longer than that of a conventional socket connection. For *vSocket*, a virtual connection needs to be created for every conventional socket connection. An intuitive question is how much extra overhead will be introduced. To answer this question, we wrote a micro-benchmark of which the server side keeps accepting external connections from the client side. The server and the client run in two different VMs. The client-side iteratively initiates 100 connections without any disturbance.

Note that to establish the first virtual connection, the backend connections have to be initialized first. Once the backend connections are created, the establishment of the subsequent *vSocket* connections can reuse these backend connections. Figure 8(a) and Figure 8(b) respectively shows the establishment overheads of the first and subsequent connections. First, we can observe that the establishment overhead of the first *vSocket* connection is 15.33 ms, while the establishment of the subsequent connections takes only 595 μs. Second, both *vSocket* and VMA need more time than *vSocket* to establish a full-fledged connection. The main reason is that VMA has to initialize resources such as route table for the first time. Both for *vSocket* and VMA, the overheads are unnecessary for the establishment of subsequent connections. Therefore the extra overhead merely affects the overall application performance, especially for long-term connections. Third, since the establishment of a *vSocket* connection consists of the establishment of a virtual connection and a kernel-based TCP connection which costs 107 μs, establishing the virtual connection of a subsequent vSocket connection takes about 488 μs.

6.2 Application Performance

In this section, we choose a real-world application, Redis [8], to measure the performance gains brought by *vSocket*. Redis is a widely used in-memory key-value store, commonly used as a database, cache, and message broker. In such scenarios, the response latency of individual requests is critical to the overall performance. We use the official benchmark *redis-benchmark* which simulates running commands done by N clients simultaneously sending M total queries [5]. The performance results reported include the response time of each request and the system throughput. We modified the *redis-benchmark* to increase the accuracy of the test response time to 1 μs.

6.2.1 Latency

We evaluate the performance in both scenarios of low and high workloads. In the case of low workload, we run 100 thousands of *SET* requests within a single connection. For the high workload, the requests are issued through ten concurrent connections. According to our experiments, 10 concurrent connections are large enough to stress the application. Besides, the payload size is set to 4 bytes.

Figure 9 and Figure 10 respectively show the CDFs (Cumulative Distribution Function) of the response time in the cases of one and ten connections. Figure 9 shows that the response time is slightly higher than the RTT reported

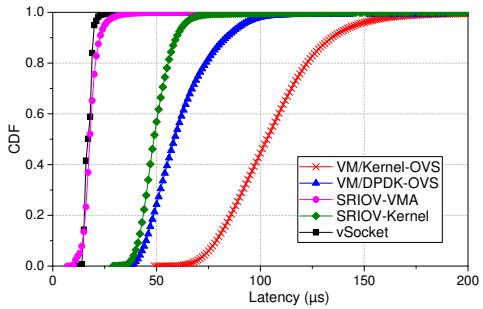


Figure 10. The CDF of response time over ten connections using the native benchmark of Redis.

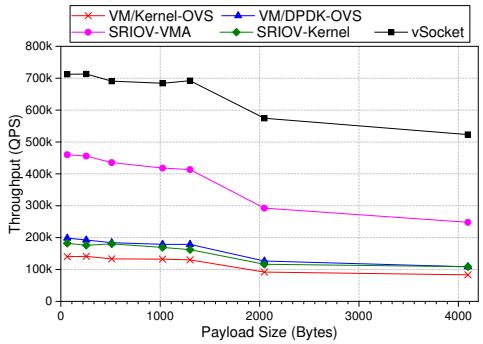


Figure 11. The throughput of Redis over eighty connections using the native benchmark.

in Sec. 6.1.1. The margin mainly comes from request handling in Redis server. Similarly, *vSocket* performs a little worse than *SRIOV-VMA* in the case of a single connection. However, when using multiple concurrent connections, *vSocket* becomes as fast as VMA as shown in Figure 10. This is because that the overall response time is not only determined by the delay on the network path, but also influenced by the processing speed of the application when the load is high.

6.2.2 Throughput

Generally, a single-threaded Redis server instance is not expected to be able to take advantage of multiple CPU cores. Therefore, in most situations, the bottleneck is the CPU core other than the network since a single Redis instance usually can not saturate a 100 Gb/s NIC according to our experimental results. Similarly, a single *redis-benchmark* instance may not saturate the Redis server too. Therefore, to obtain the maximum throughput, we simultaneously run four *redis-benchmark* instances each of which runs 20 concurrent connections.

Figure 11 shows the throughput of Redis against different sizes of payload. We can see that *vSocket*'s throughput is 3.6, 3.5, 4.8, 1.6 times of the throughput of the *VM/Kernel-OVS*, *VM/DPDK-OVS*, *SRIOV-Kernel*, and

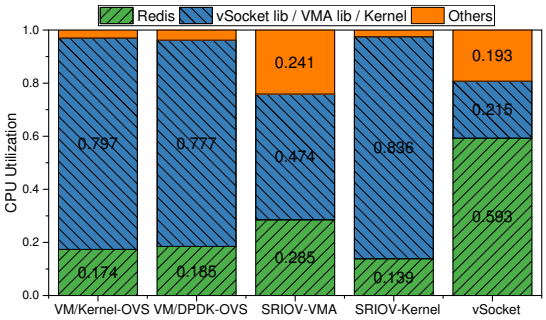


Figure 12. The breakdown of the server-side CPU usage when running the native benchmark with eighty connections.

SRIOV-VMA, respectively. Sec. 6.2.3 will explain the high throughput of *vSocket*.

6.2.3 CPU Breakdown

The throughput of Redis is determined by many factors. When the number of connection is small, Redis will mainly wait for I/O, thus the throughput in this case is mainly determined by the latency. The lower latency, the higher throughput. However, when the number of connection is large, Redis will rarely wait for I/O. Requests are almost always ready to be processed. So the throughput is mainly determined by the CPU cycles occupied by Redis.

In this section, we profile the CPU utilization of the CPU core running Redis-server by using *Perf* [6] tool. Two main components, Redis-server binary, *vSocket* library or VMA library or the kernel (including TCP/IP stack, Packet I/O, etc.) are analyzed. The Redis-server runs with 80 concurrent connections issued by four *redis-benchmark* instances. Figure 12 shows their proportions of CPU time. The CPU utilization breakdown shows that *vSocket* only consumed 21% of CPU cycles and 59% of CPU cycles are left to Redis-server binary. That means *vSocket* library introduces lower CPU overhead than VMA library or the kernel stack. Therefore, more CPU cycles are left for the application itself to process requests. However, it is worth noting that only 14% to 18% of CPU cycles are consumed by the Redis binary for *VM/DPDK-OVS*, *VM/Kernel-OVS*, and *SRIOV-Kernel* and the kernel consumes about 80% of the CPU time. This is consistent with the results in mtcp [24]. Moreover, the VMA library consumes 47% of the CPU time while it is only 21% for the *vSocket* library. We should owe the less overhead in *vSocket* to the elimination of the TCP/IP stack.

However, *vSocket* use another polling thread for packet processing. Fortunately, the polling thread can serve multiple application threads. So we conduct another experiment to profile the polling core. The results show that the polling thread only spends 20% of the CPU time on handling the packets and 80% of the CPU time on polling

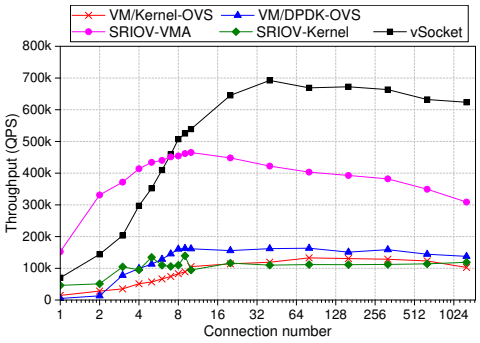


Figure 13. The throughput of Redis with the increasing number of connections.

empty queues. That is to say, the polling thread can serve five application threads concurrently. Thus, the polling core's overhead can be amortized by multiple application threads.

6.2.4 Scalability

Scalability of a library is important for today's large-scale applications. To demonstrate the good scalability of *vSocket*, we evaluate the throughput of Redis with different numbers of concurrent connections. The connection number increases from 1 to 1300 and the request is SET operation with 64 bytes of payload. Figure 13 shows that as the number of connections increases, *vSocket* can maintain high throughput. In Figure 13, it is notable that *vSocket* achieves lower throughput than VMA when the number of connections is small (less than 7 connections) but the result is reversed when the number of connection is large. The main reason is that when the number of connection is small, the throughput is mainly determined by the latency; however, when the number of connection is large, the throughput is mainly determined by the library overhead.

7 Related Work

We classify the related work into three categories: RDMA for socket applications, virtualization of RDMA and current solutions of public clouds.

The socket-based applications can use RDMA by using libraries to convert socket APIs to Verbs APIs. VMA [28] and Rsocket [20] provide the ability of converting Verbs API to Socket API. VMA exploits RDMA's RAW_PACKET mode to transmit packets, which is not reliable. So it embeds a TCP protocol stack in it to provide reliability. However, the protocol stack introduces processing overheads. As for Rsocket, it does not handle compatibility issues very well. For example, it does not support epoll. Moreover, they are not initially designed for virtual environments.

Some other works enable RDMA for virtual machines. With the SR-IOV [10], a VM can achieve close to the bare-metal performance by directly accessing the hardware. But in public clouds, it still needs to tackle problems such as security, network isolation and flexibility. HyV [30] presents a hybrid virtualization architecture which separates the RDMA control path and the RDMA data path. Thereby it provides greater flexibility in resource management compared to SR-IOV without sacrificing the performance of the data path. But problems like security, network isolation are still difficult to solve. vRDMA [31] uses the para-virtualized framework to virtualize RDMA for VMWare's hypervisor, but it provides the Verbs API which is not friendly to socket-based applications. AccelNet [16] proposes to solve the problems of SR-IOV by offloading the security rules, network tunneling to a new piece of hardware. But it needs to modify hardware and provides the Verbs API. FreeFlow [39] enables RDMA for the container-based cloud. But the method of FreeFlow only works for the container.

In public clouds, the para-virtualization technology is the default solution. In the host, a virtual switch such as OpenvSwitch [29] is required to switch packets, and a virtual I/O driver such as virtio [32] and vhost [18] are needed to transfer packets between the guest and the host. To process all types of traffic, the virtual switch needs to be designed very complex, which will result in additional delays. In the VM, a TCP/IP stack needs to be provided for socket-based applications. The kernel stack is the default solution but has poor performance. A number of researches adopt the user space approach, such as, seastar [1], mtcp [24], and libuinet [26]. Nevertheless, their APIs are not fully compatible with the BSD Socket API. Porting existing applications is also cost prohibitive for most tenants. Moreover, the TCP stack is still processed by CPU, which will result in high library overheads.

8 Conclusion

In this paper, we propose a new inter-VM communication technique *vSocket*, which not only provides high performance but also meets the requirements of public clouds such as security rules and network isolation. Moreover, *vSocket* is compatible with BSD socket so that socket-based applications can use it without any modification. *vSocket* proposes to reuse the kernel stack and virtualize RDMA connections to enable RDMA networking in public clouds. Evaluations with Redis and microbenchmarks show that *vSocket* can achieve much better performance than current solutions for public clouds.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and suggestions, and our shepherd, Anil Madhavapeddy, for his editorial assistance.

References

[1] 2018. Cloudius Systems. Seastar. http://seastar.io/ [Online; accessed 8-December-2018].

[2] 2018. Mellanox ConnectX-4 adaptor. http://www.mellanox.com/page/products_dyn?product_family=201&mtag=connectx_4_vpi_card. [Online; accessed 19-September-2018].

[3] 2018. Netperf benchmark. https://hewlettpackard.github.io/netperf/ [Online; accessed 19-September-2018].

[4] 2018. Network ACLs. https://docs.aws.amazon.com/vpc/latest/userguide/vpc-network-acls.html. [Online; accessed 26-February-2019].

[5] 2018. The official benchmark of Redis. https://redis.io/topics/benchmarks [Online; accessed 19-September-2018].

[6] 2018. Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page. [Online; accessed 19-September-2018].

[7] 2018. The RDMA Verbs Specification. http://www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf. [Online; accessed 19-September-2018].

[8] 2018. Redis, an open source (BSD licensed), in-memory data structure store. https://redis.io/ [Online; accessed 19-September-2018].

[9] 2018. Security Groups for Your VPC. https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html. [Online; accessed 8-December-2018].

[10] 2018. Single root I/O virtualization. http://pcisig.com/specifications/iov/single_root/ [Online; accessed 8-December-2018].

[11] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[12] Inho Cho, Keon Jang, and Dongsu Han. 2017. Credit-Scheduled Delay-Bounded Congestion Control for Datacenters. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 239–252. https://doi.org/10.1145/3098822.3098840

[13] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, Berkeley, CA, USA, 401–414. http://dl.acm.org/citation.cfm?id=2616448.2616486

[14] Adam Dunkels. 2001. Design and Implementation of the lwIP TCP/IP Stack. In *Swedish Institute of Computer Science*.

[15] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. 2011. Cuckoo directory: A scalable directory for many-core systems. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. 169–180. https://doi.org/10.1109/HPCA.2011.5749726

[16] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA*.

[17] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin. 2005. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In *13th Symposium on High Performance Interconnects (HOTI'05)*. 128–137. https://doi.org/10.1109/CONECT.2005.35

[18] Stefan Hajnoczi. 2011. QEMU Internals: vhost architecture. http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html. [Online].

[19] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 29–42. https://doi.org/10.1145/3098822.3098825

[20] Sean Hefty. 2012. Rsockets. In *2012 OpenFabris International Workshop, Monterey, CA, USA*.

[21] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. 2015. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management* 12, 1 (2015), 34–47.

[22] DPDK Intel. 2018. Data plane development kit.

[23] Manish Jain, Ravi S Prasad, and Constantinos Dovrolis. 2003. *The TCP bandwidth-delay product revisited: network buffering, cross traffic, and socket buffer auto-sizing*. Technical Report. Georgia Institute of Technology.

[24] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.. In *NSDI*, Vol. 14. 489–502.

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. ACM, New York, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[26] Patrick Kelsey. [n. d.]. Libuinet. https://github.com/pkelsey/libuinet [Online].

[27] M. Mahalingam, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. 2018. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://www.rfc-editor.org/info/rfc7348 [Online].

[28] Mellanox. [n. d.]. Mellanox Messaging Accelerator. http://www.mellanox.com/page/software_vma [Online].

[29] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The Design and Implementation of Open vSwitch.. In *NSDI*, Vol. 15. 117–130.

[30] Jonas Pfefferle et al. 2015. A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces. In *Proceedings of the VEE (VEE '15)*. 17–30. https://doi.org/10.1145/2731186.2731200

[31] Adit Ranadive et al. 2012. *Toward a Paravirtual vRDMA Device for VMware ESXi Guests*. VMware Technical Journal. Georgia Institute of Technology, VMWare Inc.

[32] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[33] Michael Scharf and Sebastian Kiesel. 2006. Head-of-line Blocking in TCP and SCTP: Analysis and Measurements.. In *GLOBECOM*, Vol. 6. 1–5.

[34] P. Shivam, P. Wyckoff, and D. Panda. 2001. EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. 49–49. https://doi.org/10.1145/582034.582091

[35] Jia Song and Jim Alves-Foss. 2012. Performance review of zero copy techniques. *International Journal of Computer Science and Security (IJCSS)* 6, 4 (2012), 256.

[36] Wikipedia. 2018. Token bucket. https://en.wikipedia.org/wiki/Token_bucket [Online; accessed 8-December-2018].

[37] Wikipedia. 2018. Traffic shaping. https://en.wikipedia.org/wiki/Traffic_shaping [Online; accessed 8-December-2018].

[38] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. 2017. Towards Zero Copy Dataflows Using RDMA. In *Proceedings of the SIGCOMM Posters and Demos (SIGCOMM Posters and Demos '17)*. ACM, New York, NY, USA, 28–30. https://doi.org/10.1145/3123878.3131975

[39] Tianlong Yu, Shadi Abdollahian Noghabi, Shachar Raindel, Hongqiang Liu, Jitu Padhye, and Vyas Sekar. 2016. FreeFlow: High Performance Container Networking. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks (HotNets '16)*. ACM, New York, NY, USA, 43–49. https://doi.org/10.1145/3005745.3005756

[40] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. 2015. Congestion Control for Large-Scale RDMA Deployments. *SIGCOMM Comput. Commun. Rev.* 45, 4 (Aug. 2015), 523–536. https://doi.org/10.1145/2829988.2787484