MasQ: RDMA for Virtual Private Cloud

Zhiqiang He*
University of Science and Technology of China
Huawei Technologies Co., Ltd
zhqhe@mail.ustc.edu.cn

Dongyang Wang*
wangdongyang7@huawei.com
Huawei Technologies Co., Ltd

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

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

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

Zhi-Li Zhang
zhzhang@cs.umn.edu
University of Minnesota

Kai Zheng
kai.zheng@huawei.com
Huawei Technologies Co., Ltd

ABSTRACT

RDMA communication in virtual private cloud (VPC) networks is still a challenging job due to the difficulty in fulfilling all virtualization requirements without sacrificing RDMA communication performance. To address this problem, this paper proposes a software-defined solution, namely, MasQ, which is short for "queue masquerade". The core insight of MasQ is that all RDMA communications should associate with at least one queue pair (QP). Thus, the requirements of virtualization, such as network isolation and the application of security rules, can be easily fulfilled if QP's behavior is properly defined. In particular, MasQ exploits the virtio-based paravirtualization technique to realize the control path. Moreover, to avoid performance overhead, MasQ leaves all data path operations, such as sending and receiving, to the hardware. We have implemented MasQ in the OpenFabrics Enterprise Distribution (OFED) framework and proved its scalability and performance efficiency by evaluating it against typical applications. The results demonstrate that MasQ achieves almost the same performance as bare-metal RDMA for data communication.

CCS CONCEPTS

• Networks → Cloud computing; **• Software and its engineering**;

KEYWORDS

RDMA, Network virtualization, Datacenter network

ACM Reference Format:
Zhiqiang He, Dongyang Wang, Binzhang Fu, Kun Tan, Bei Hua, Zhi-Li Zhang, and Kai Zheng. 2020. MasQ: RDMA for Virtual Private Cloud. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, NY, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3387514.3405849

1 INTRODUCTION

The remote direct memory access (RDMA) technique allows applications to transfer data directly to or from memories of two computers over a network. By enabling zero-copy networking, RDMA allows for high-performance data-intensive applications and has been widely used in high-performance computing (HPC) scenarios, supported by expensive InfiniBand (IB) networks [25]. Recently, many more economical RDMA technologies, such as RoCE/RoCEv2 [24, 26] and iWARP [27], have been developed, and the routable RoCEv2 [24, 26] has become the technology of choice due to its low complexity. In the remainder of this paper, we will therefore focus exclusively on RoCEv2 networks.

Currently, public clouds usually provide high-performance instances equipped with RDMA networks by building separate resource pools, which significantly increases the capital expense. Thus, enabling RDMA in the VPC network, where VPC is an on-demand resource pool allocated within a public pool, becomes one of the most important jobs for cloud vendors. Right now, virtual switches are exploited to create a logically isolated VPC network, but this solution does not work for RDMA, mainly because the RDMA network interface controller (RNIC) offloads the network protocol and then bypasses the virtual switches. To address this problem, there are two categories of solutions: 1) hardware solutions and 2) software solutions. However, neither of them can simultaneously achieve both high performance and high scalability.

The core idea of hardware solutions is to put network virtualization operations, such as encapsulating and decapsulating Virtual eXtensible LAN (VXLAN) [37] headers, in the middle of two single root input/output virtualization (SR-IOV)-enabled RNICs. New functions can be realized in SmartNICs [6], in top-of-rack switches, or in separate FPGA boards [22]. Hardware solutions, in general, are

[*] The first two authors contributed equally to this paper, and Binzhang Fu is the corresponding author. Zhiqiang He performed this work during an internship at Huawei.

SIGCOMM '20, August 10–14, 2020, Virtual Event, NY, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7955-7/20/08…$15.00
https://doi.org/10.1145/3387514.3405849

performance-optimized but lack scalability. For example, RNIC has to cache the contexts of virtual networks (such as the VXLAN tunnel table) to realize network virtualization, but the on-chip cache is usually limited. Therefore, if the VPC network is large, then communication performance is reduced since RNIC must frequently fetch contexts from DRAM. As reported in [17], the throughput of `stat` operations decreases by almost 50% when the number of clients increases from 40 to 120 due to the sharply increased possibility of cache misses.

Conversely, the core idea of software solutions is to redirect the RDMA control path or even the data path to a software component, such as a virtual switch, to apply network virtualization enforcement [33]. The software solution is flexible, but involving software in the critical path is not a good idea to obtain the best performance. For example, involving virtio [40] in the critical path of the "post_send" operation can slow down the performance by 101 times, as discussed in Sec. 3.1. This dramatic performance degradation prevents the solution from being widely adopted.

To address the above dilemma, we propose a new kind of RDMA virtualization, where software defines the "communication rules" (to keep the solution scalable), while hardware executes the "communication operations" (to minimize performance overhead). We name this solution software-defined virtualization and will present the design of MasQ following this principle in this paper. The core idea of MasQ is that each RDMA communication should associate with one QP, and the QP context (QPC) maintains all necessary information to send or receive a message [1]. Thus, if QPC is properly "virtualized", then the RDMA network is accordingly correctly virtualized. To this end, we first classify all Verbs into two categories based on whether QPC is manipulated. For example, Verbs manipulating QPC (such as "create_qp") are labeled control path Verbs, while others that only implicitly use QPC (such as "post_send") are classified as data path Verbs. Then, MasQ solely exploits the control path Verbs to realize RDMA virtualization to avoid performance overhead.

To achieve the above goals, two major challenges should be addressed. First, as RNIC bypasses the OS kernel and thereby virtual switches running in the kernel, a *virtual* L2 RDMA network cannot rely on VXLAN and virtual switches to support network virtualization. Second, as the security group and firewall as a service (FWaaS) are commonly implemented in virtual switches that are also bypassed by RNIC, it is unclear how to apply security rules to a virtual RDMA network. We note that instead of coming up with new "custom-made" security mechanisms (the security properties of which may be unproven) for virtual RDMA networks, we want to support the same security mechanisms, i.e., security group and FWaaS, that have been widely deployed and employed by existing public cloud providers, as their APIs are familiar to most application developers and network operators of cloud tenants [29].

In this paper, MasQ first proposes the *vBond* and *RConnrename* techniques, which together provide an abstract of virtual RoCE networks for each VPC. Note that there is an important difference between a virtual RoCE device and a virtual IB device, where the RoCE device provides both Ethernet and RDMA interfaces, and the IB device only provides RDMA interfaces. With MasQ, RDMA applications can communicate with each other by still using their virtual IPs. Furthermore, an RDMA version connection tracking module, namely, the *RConntrack*, is proposed to support both the security group and FWaaS to protect virtual RDMA networks from attack. Note that all of the above techniques reside on the control path, so MasQ's performance overhead is negligible. We summarize our main contributions as follows:

(1) To the best of our knowledge, MasQ is the first work that targets RDMA network virtualization for VPC. It employs three innovative mechanisms to tackle the challenges of providing virtual RoCE abstraction, tenant logical segregation and applying security rules.
(2) A prototype of MasQ is implemented and evaluated against big data and HPC applications. The results prove that RDMA can be deployed in virtualized data centers with negligible overhead.

2 RELATED WORK

RDMA I/O virtualization: I/O virtualization is the first step in enabling RDMA in a virtual machine (VM). Currently, there are two prominent techniques: 1) direct device assignment and 2) paravirtualization. Direct device assignment, such as SR-IOV, can provide near-native performance but is not flexible. In contrast, paravirtualization is more flexible but at the expense of lower performance. For example, VMware's vRDMA [39] adopts this solution. A paravirtualized network stack splits the device driver into a frontend driver and a backend driver. To complete a network I/O operation, the frontend driver forwards the command to the backend driver, and then, the backend driver delivers it to the physical device. Such separation provides greater flexibility but inevitably incurs additional overhead during data path operations. To mitigate this overhead, HyV [38] and virtio-RDMA [21] employ a *hybrid* virtualization technique that is tailored for RDMA. Taking advantage of RDMA's separation of control and data paths, HyV adopts paravirtualization for the control path but implements a zero-copy data path. Therefore, no performance overhead is introduced. Compared with the above counterparts, MasQ adopts a similar approach of I/O virtualization but focuses on the new fundamental challenges of enabling RDMA in VPC networks, such as network isolation and applying security rules.

RDMA network virtualization: for containerized clouds, FreeFlow [33] proposed a paravirtualization-based solution to virtualize the RDMA network. To this end, the FreeFlow router (FFR) is exploited to manipulate intra- and inter-host RDMA flows. Similar to vRDMA [39], FreeFlow also trades performance for manageability by forwarding data path operations between container and FFR. Microsoft Azure proposed AccelNet [22] to realize network virtualization in an FPGA-based SmartNIC sitting outside the RNIC. Basically, AccelNet can meet all our requirements, but it requires specialized hardware. Furthermore, since network virtualization is implemented in hardware, it also suffers from the scalability issue due to its limited on-chip resources.

Other works using RDMA in clouds: recently, many efforts have been made to pave the way for the application of RDMA in clouds. The first group of efforts focuses on exploiting RDMA to

[1] This paper mainly focuses on the connection-oriented RDMA communications. Connectionless transports, such as unreliable datagram, will be shortly discussed in Sec. 3.3.4.

improve the performance of key applications, such as latency sensitive socket applications [43], RDMA-based HDFS [28], FaRM [20], FaSST [32] and HERD [30]. Although the above works are not directly related to MasQ, they motivate our work by proving that RDMA is the key technology for achieving high-performance applications. Based on this observation, another group of efforts [18, 23, 45] focuses on enabling the large-scale deployment of RDMA in data centers. Although they are also orthogonal to MasQ, the success of deploying RDMA in a large-scale system gives us enough confidence to deploy RDMA for VPC.

In summary, there is still no software solution that can simultaneously realize both RDMA I/O and network virtualization without significant performance degradation. Thus, MasQ is proposed to fill this gap.

3 PROPOSED MASQ

In this section, we will first discuss the rationale behind MasQ and prove that RDMA can be efficiently virtualized in a software-defined manner. Then, we will discuss how the software-defined MasQ addresses the challenges of RDMA for VPC in detail.

3.1 Rationale

To obtain the optimized performance in data communication, RDMA is designed in a software-defined manner by default, where the control path clearly separates from the data path. As shown in Fig. 1, to communicate with each other, there are three phases in both the client and server. The first phase is the setup phase, where both the client and server prepare for communication by calling the Verbs shown in red (or in italics), including creating resources, exchanging communication information, and setting up the QP states. One important feature of the first phase is that all involved Verbs are one-time operations for an application at most times. For example, once a QP is created, it can always be used to send and receive messages, unless it is destroyed explicitly. During the second phase, the client and server can exchange data with each other. Note that the second phase should be repeated until all data have been successfully exchanged. Once all communications finish, the final phase will release all resources accordingly.

We find that the Verbs used in the first and third phases only manipulate resources/QPC and are not directly involved in real data communication in the second phase; then, these Verbs are named control path Verbs. Conversely, Verbs in the second phase are named data path Verbs. Furthermore, control path Verbs are one-time operations, so they are not performance sensitive. This provides us with the chance that if virtualization can be realized only on control path Verbs with acceptable overhead, then the proposed solution will be feasible.

To determine whether the overhead to virtualize the control path Verbs is acceptable. We first evaluate the raw performance of each Verbs and then estimate its virtualized performance by adding the overhead of virtio. As shown in Table 1, the "Host-RDMA" column represents the raw performance of Verbs, and the "w/ virtio" column represents the results that further consider the latency introduced by virtio (according to our evaluation, the average latency of a roundtrip communication using virtio between VM and host is 20 μs). As shown in the "Slow down" column, we can find that the maximal performance degradation for an individual Verbs is 130%. However, the following three facts prove that the overhead is acceptable. First, if we take the first and third phases as a whole, then the total performance overhead of the control path is only 9% (2.62 ms vs 2.86 ms). Second, most RDMA applications maintain long-lived connections, so that overhead is a one-time cost for each application. Third, most RDMA applications run for tens of minutes or even several days, and the overhead, which is smaller than 0.3 ms, is negligible.

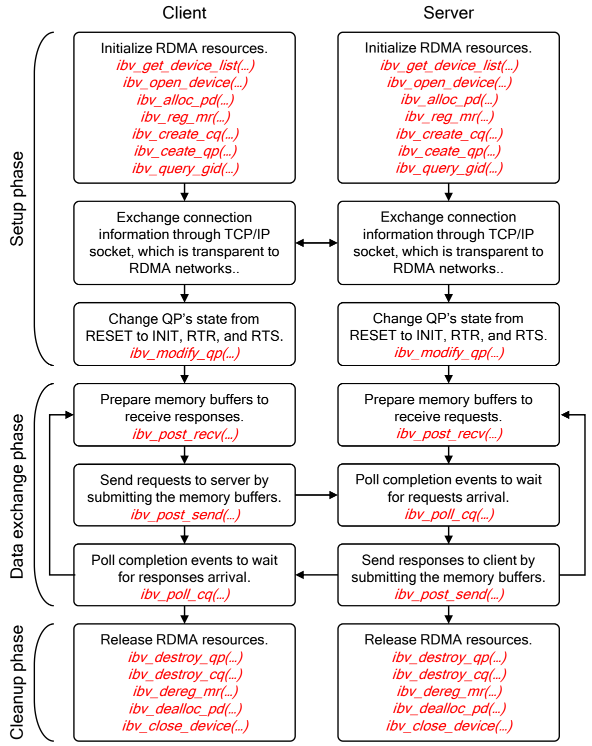


Figure 1: An illustrative client-server RDMA application. Related *Verbs* are shown in red.

3.2 I/O virtualization

In general, MasQ adopts a hybrid I/O virtualization approach, as shown in Fig. 2, where the control path is virtualized based on virtio, and the data path is directly memory mapped. Thus, the data path-related resources can be directly accessed by the RNIC and applications in the VM in both directions. For example, there are two types of resources in RDMA's data path. One is the hardware registers, such as Doorbell. Generally, hardware registers are mapped to the VM's physical address space so that applications in the VM can access them through memory-mapped I/O (MMIO). The other is the user memories in the VM, such as QPs and user-registered memory regions (MRs). The above memories are visible in the host's physical address space. Therefore, by mapping the guest virtual address (GVA) to the host physical address (HPA), RNIC can directly access user memories in the VM through DMA. In general, the above solution is not a new idea and was also adopted by HyV [38] and virtio-RDMA [21]. Therefore, we omit the details here and provide

Table 1: Performance comparison between nonvirtualized and virtualized Verbs.

Class	Step	Verbs API	Call time (μs)		Slowdown
			Host-RDMA	w/ virtio	
Control path Verbs for resources setup	1	`ibv_get_device_list(...)`	396	416	1.1
	2	`ibv_open_device(...)`	1115	1135	1.0
	3	`ibv_alloc_pd(...)`	3	-*	1.0
	4	`ibv_reg_mr(buf_size=1KB, ...)`	78	98	1.3
	5	`ibv_create_cq(cqe=200, ...)`	266	286	1.1
	6	`ibv_create_qp(max_send/recv_wr=100, max_send/recv_sge=1, ...)`	76	96	1.3
	7	`ibv_query_gid(...)`	22	-*	1.0
	8	`ibv_modify_qp(INIT, ...)`	231	251	1.1
	9	`ibv_modify_qp(RTR, ...)`	62	82	1.3
	10	`ibv_modify_qp(RTS, ...)`	73	93	1.3
Data path Verbs for data exchanging	11	`ibv_post_send/recv(...)`	0.2	20	101.0
	12	`ibv_poll_cq(...)`	0.03	20	667.7
Control path Verbs for resources cleanup	13	`ibv_destroy_qp(...)`	170	190	1.1
	14	`ibv_destroy_cq(...)`	79	99	1.3
	15	`ibv_dereg_mr(...)`	35	55	1.6
	16	`ibv_dealloc_pd(...)`	2	-*	1.0
	17	`ibv_close_device(...)`	16	36	2.3

∗ These Verbs are implemented in pure software and not forwarded to RNIC, so it is unnecessary to virtualize them.

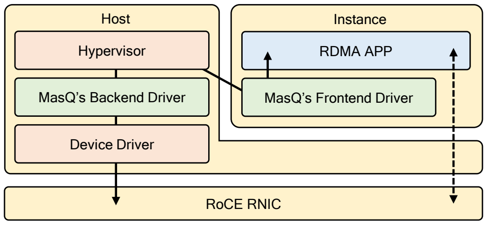


Figure 2: Architecture of MasQ's I/O virtualization, where solid and dashed arrows represent control and data path, respectively.

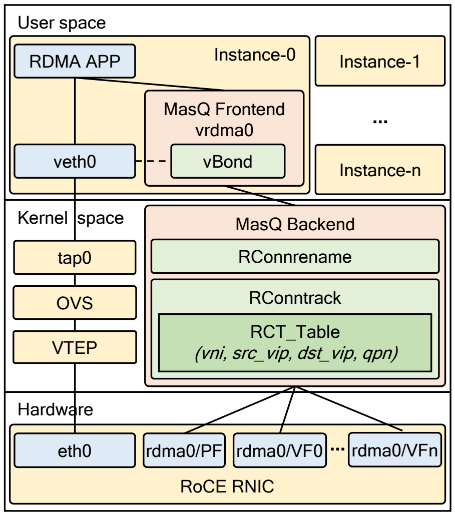


Figure 3: Architecture overview of the control path of proposed MasQ, where green components are proposed to address the problem of network virtualization.

a short introduction in Appendix A. Interested readers can also refer to [38] or [21] for more details.

3.3 Network virtualization

As shown in Fig. 3, as in the case of physical RoCE RNIC, a *virtual* RoCE RNIC is also represented with two interfaces, i.e., a virtual Ethernet interface and a virtual RDMA interface. More specifically, MasQ reuses `vhost_net` to realize the virtual Ethernet interface and exploits the above I/O virtualization technique discussed in Sec. 3.2 to virtualize the RDMA interface. The rest of this section will discuss how MasQ realizes network virtualization on the virtual RDMA interface.

3.3.1 Tenant isolation. MasQ proposes a new per-connection instead of a traditional per-packet virtualization technique, namely, the ***RConnrename***. The core idea is that tenants (applications in VMs) and the cloud provider (MasQ's backend driver) can refer to the same connection by different names, where tenants use virtual addresses, and cloud provider uses the corresponding physical addresses. Therefore, once the connection is established, packets can be encapsulated by RNIC with the physical addresses directly, without any per-packet overhead. Actually, the connection-based virtualization technique has been used in the TCP/IP networks proposed by Slim [46]. However, when this idea is applied to RDMA networks, there are several new challenges to be addressed.

A VM may have multiple virtual RNICs, but an application may only specify a virtual destination IP address to start a communication. Therefore, MasQ must be able to determine which local virtual RNIC should be used. For a physical RoCE network, this is not a problem since both Ethernet and RDMA interfaces are abstracted from the same PCIe device. For example, drivers can first obtain the Ethernet interface by looking up the local routing table and then obtain the associated RDMA interface belonging to the same PCIe device. However, virtual interfaces are abstracted from separate

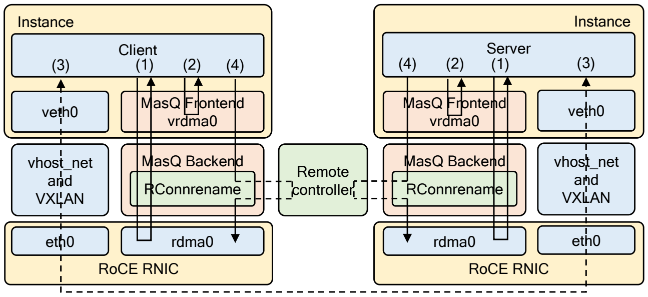


Figure 4: Key steps of MasQ (*RConnrename*) to establish an RDMA connection.

virtio devices. Thus, an artificial bond between two virtual interfaces is necessary. The ***vBond*** is developed to tackle this challenge. Since the presentation of *RConnrename* depends on *vBond*, we will discuss *vBond* first.

vBond is designed to allow for the application running in a VM to use two virtual interfaces by specifying only one single IP address, as in the case when an application is accessing a physical RoCE device. To address this problem, *vBond* is developed to dynamically bind the virtual Ethernet and virtual RDMA interfaces. For this, *vBond* first obtains the (virtual) MAC address of the virtual Ethernet interface, to which the virtual RDMA interface should bind (note: tenants are not allowed to modify virtual MAC addresses). This is done by querying the backend driver during the initialization. If the virtual Ethernet interface has been assigned a valid IP address, then *vBond* will immediately initialize the global identifier (GID) accordingly and binds the virtual Ethernet interface with the corresponding virtual RDMA interface. Thereafter, *vBond* registers a callback function to the notification chain of `inetaddr` events in the OS. Whenever the IP address of a virtual Ethernet interface changes, OS will notify *vBond* to check and update the GID accordingly. Note that GID is used to identify the RDMA interface in an RDMA network.

RConnrename is proposed to guarantee that correct network addresses are used to encapsulate RDMA packets. To prevent RNIC from using the remote virtual (IP/MAC) addresses as the destination addresses [2], the configuration commands issued by the application must be properly managed. Fortunately, as shown in Fig. 3, all commands issued by the application are first handled by MasQ's backend driver before being sent to the real device driver. Therefore, MasQ can exploit this opportunity to present two different views of the same QPC to the application and RNIC. In other words, the application will see the QPC as configured with virtual network addresses, while RNIC will view it as configured with physical addresses. We call this solution the *RDMA Connection Rename* or *RConnrename* in short. A major benefit of *RConnrename* is its low cost since network addresses are configured only once for each connection.

We now use an example to illustrate how *RConnrename* works in detail. As shown in Fig. 4, to establish an RDMA connection, both the client and server need to create a QP and register their memory regions following the control path (1). The "create_qp" command and the address mapping (GVA, GPA) of the QP are forwarded to the host. Upon receiving the command, the backend driver will further map the QP's GVA to HPA and then create the QPC by calling the device driver. Note that QPC is maintained on the host. Once the local resources are ready, the application can query its local GID. Note that the *vBond* module maintains the virtual GID as discussed above. Thus, *vBond* can directly reply to the query request as the control path (2) shows. After obtaining local connection information such as QP number (QPN) and virtual GID (vGID), the application needs to exchange these information with the peer. This is usually done over a pre-established TCP connection, as shown in step (3).

With the peer's connection information, the application needs to configure its QPC with the other side's vGID as the destination address. As discussed above, *RConnrename* will intercept the command and replace the peer's virtual address in the command by the peer's physical address. Now, the challenge is how to obtain the corresponding physical address. Recall that the vGID, which is dynamically synchronized with the IP address of the virtual Ethernet interface, has no relationship with the physical GID. Therefore, an additional mechanism to maintain the mapping between virtual and physical GIDs is necessary.

Furthermore, since the public cloud provides tenants with independent IP address spaces, different tenants' virtual IP addresses may be the same. This means that there may be multiple identical virtual GIDs in the cloud, so we need other information to identify the physical GID. To this end, we use the tenant ID and vGID as the key to find the corresponding physical GID in a mapping table. In practice, the tenant's VXLAN network identifier (VNI) can be used as the tenant ID. We propose the utilization of a logically centralized controller to maintain such a mapping table. Once a vGID is created or updated, *vBond* should immediately notify the controller to update its mapping table. *RConnrename* can then query the controller to obtain the physical GID corresponding to the vGID. To reduce the performance overhead, we further employ a local cache of the mapping table. Specifically, the mapping record returned by *RConnrename*'s first query will be inserted into the local cache. Then, the queries hitting the local cache will be completed in a few microseconds, which are negligible compared with RDMA's connection setup time. In addition, in a common case, a mapping record will not be updated after insertion into the local cache. Therefore, the cache hit will always be maintained. To further avoid the overhead of cache misses, the controller can be configured to push down the mappings in advance. Generally, at least 35 bytes of memory are required to hold a mapping record, including virtual GID (16 B), VNI (3 B), and physical GID (16 B). Therefore, the local cache will take up ~0.33 MB of memory to support ten thousand VM peers, which can be easily satisfied in DRAM. We should note that controller performance is a key issue for all software-defined networking solutions, including the proposed MasQ. Fortunately, SDN has been widely studied and deployed in public clouds, and many practical solutions have been proposed, such as DevoFlow [19] and Onix [34]. In our opinion, improving the performance of an SDN controller is very important but out of the scope of this paper.

Therefore, to configure the QPC, the application will issue the command following the control path (4). Upon receiving the command, *RConnrename* will query the remote controller to retrieve the physical GID for communicating with the peer. As the local cache

[2] Note that source addresses are configured by host device driver using the physical IP/MAC addresses of the RNIC.

Table 2: Behavior of the application and RNIC when QP state is modified to ERROR.

Application	Post receive request	Allowed
	Post send request	Allowed
	Poll completion queue	Allowed but get an error CQE
RNIC	Receive request processing	Flushed with error
	Send request processing	Flushed with error
	Incoming packets	Dropped
	Outgoing packets	None

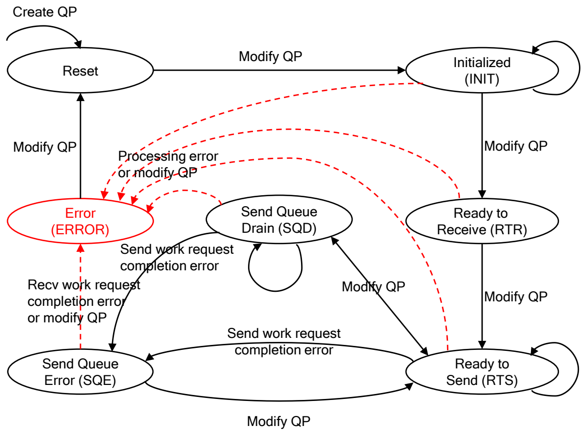


Figure 5: The state machine of QP. Dashed red lines indicate that an QP can switch to ERROR state from any other state.

should have maintained the mapping information, the query can be completed within a few microseconds. Once QPCs on both sides are configured, the connection is thereby successfully established. We see that both the client's and the server's QPCs are configured with peer's physical GIDs. This ensures that all subsequent data packets will be encapsulated with the correct physical network addresses so that they can be routed correctly to their destinations through the underlay network.

3.3.2 Security isolation. RDMA security in VPC includes both network security and user memory security. Specifically, any traffic violating the network security rules should not be injected into the network. In addition, a user's memory, such as QPs and MRs, should not be accessed by any unauthorized user. In the following, we will present MasQ's solution to address the above two challenges.

Network security: to protect virtual RDMA networks in public clouds, we would like to support the same two-level security mechanisms, *FWaaS* at the network level and *security group* at the VM level, that have been deployed by public clouds and are familiar to most tenants. Generally, security rules follow a similar structure – INPUT rules, OUTPUT rules, and FORWARD rules – and each of them is organized as a chain. Upon receiving a packet, it is checked against the rules of each chain, one by one in descending order of priority, and if a rule matches, then the corresponding action is taken. If none of the rules apply, then by default, the packet is denied and thus dropped. These security rules are commonly implemented in virtual bridges or virtual switches in the hypervisor.

However, since the data path of a virtual RDMA network bypasses the hypervisor, it is impossible to fully reuse the traditional solution. Fortunately, we can exploit a feature of security rules to circumvent this problem. Clearly, scanning the policy chains is time-consuming. Hence, a feature called "connection tracking" is utilized, through which the state of each connection is tracked. Packets belonging to established (thus permitted) connections are allowed without the need to scan policy chains. Using this observation, we can divide the problem of applying security rules to virtual RDMA networks into the following three subproblems. First, an RDMA connection cannot be established unless it is explicitly allowed by a security rule. Second, no RDMA packet is allowed unless it belongs to an established RDMA connection. Third, once rules are updated, connections that are no longer allowed should be blocked as soon as possible.

The first two subproblems are relatively easy to address since MasQ's backend can deny all requests violating security rules and RNIC never sends an RDMA message until the connection is established. In general, we can enforce security rules during the connection establishment phase, as the connections through the virtual Ethernet interface do not bypass the hypervisor and are protected by security groups and firewalls. For example, if a VM attempts to establish an RDMA connection with another VM that is not allowed, then the installed security group or firewall security rules will drop the packets carrying the corresponding connection request information. Without such information, the RDMA connection will not be established. Therefore, no data will be transferred through the virtual RDMA interface (and thus the physical RNIC to which it is mapped).

To address the third subproblem, MasQ proposes the ***RConntrack*** to perform connection tracking for all RDMA flows. Thus, *RConntrack* can identify and disable a virtual RDMA data path as soon as a previous rule allowing this connection has been deleted or updated to deny it. When a violating connection is found, no packets should be transmitted over that connection. In the TCP/IP network, the firewall only needs to drop the packets that violate security rules. However, since the RDMA data path bypasses the host, *RConntrack* cannot drop packets like TCP. Therefore, we need other ways to abort the transmission.

Since the control path is para-virtualized in MasQ, the states of QPs can be controlled by the host. As shown in Fig. 5, an QP has several states such as initialized (INIT), ready to receive (RTR), ready to send (RTS) and error (ERROR), and the QP behaves differently in different states. For example, a QP should be in the RTS state when transmitting data. Therefore, when an RDMA connection violates security rules, we need to switch the QP's state to another state that cannot send data. Moreover, we need to notify the application that the connection has been disconnected. Table 2 shows that if a QP switches to the ERROR state, then RNIC will immediately stop data processing and generate an error completion queue element (CQE) to notify the application. In addition, any state can change to the ERROR state by modifying QP, which means that *RConntrack* can actively switch that QP state to the ERROR state at any time.

Now we exploit an illustrative example to present how *RConntrack* works. As shown in Fig. 6, we assume that a tenant has two subnets with masks of 192.168.1.0/24 and 192.168.2.0/24. First, security rules allow virtual machines in different subnets to establish RDMA connections. For example, suppose that VM A issues an

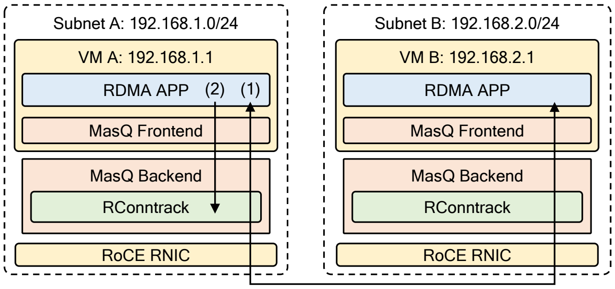


Figure 6: An illustrative example of *Rconntrack.*

RDMA connection request (1) to VM B in the other subnet. The request (i.e., "modify_qp to RTR") will be intercepted by the backend, and then, the *RConntrack* finds that this is a request from 192.168.1.1 (VM A) to 192.168.2.1 (VM B); this request is allowed by security rules. Then, the request will be forwarded to *RConnrename* to complete the remaining work. After successfully establishing the connection, *RConntrack* will record the following tuple (<Tenant ID: 192.168.1.1, 192.168.2.1>) in the connection table.

Next, suppose the user updates the security rules to prevent the two subnets from communicating through the RDMA network. Once the rule is updated, the new cross-subnet connection request (2) will be detected as illegal when the request is processed by the *RConntrack*. Thus, request (2) will be refused. Moreover, *RConntrack* will also check the connection table to find established connections that violate the rules. In this example, connection (1) will be found and disconnected by modifying the QP to the ERROR state.

User memory security: MasQ relies on RDMA's security mechanisms to protect user memory. First, RDMA resources, such as QPs, MRs, and protection domains (PDs), are created by the backend driver of MasQ; then, one VM cannot manipulate resources belonging to other VMs. Second, to communicate with a remote QP, a connection should be established in reliable connection mode, or a Q-Key will be required in unreliable datagram mode. Thus, illegal requests can be easily identified and denied in this phase. Third, to correctly access a remote MR, a memory key is required, and the remote QP and MR belong to the same PD. Furthermore, any memory operation should be associated with a QP in RDMA. Therefore, the above three "preconditions" hold, regardless of the relative position of QPs. Finally, RNIC checks the boundaries of the MR targeted by each RDMA operation. Thus, it is impossible to access a memory location outside the legal region of a legal MR; in summary, there is no additional overhead introduced by MasQ to protect user memory.

Actually, it was recently found that a side-channel attack is possible in the RDMA network [41]. Pythia [41] depends on the accurate detection of small changes in network latency to steal information. However, considering that network latency always changes in a large network due to traffic bursts and network congestion, the possibility of applying the above attack is relatively low. Generally, hardware architecture techniques, such as new cache architectures [36, 44], are adopted to address side-channel attacks. Therefore, the security problem caused by a side-channel attack is orthogonal to MasQ.

3.3.3 Quality of service. In modern data centers, quality of service (QoS) is also an important requirement, especially for high-end users. To minimize the performance overhead introduced by QoS operations, such as rate limiting, MasQ leverages hardware-based rate limiter to do this job. Specifically, MasQ provides QP-level QoS by mapping QPs to different rate limiters, where each is configured with a predefined QoS policy. In addition, to achieve better scalability given limited hardware resources, MasQ proposes QP grouping, with which QPs can be grouped following a certain rule, and then mapped to one rate limiter. For example, MasQ's default policy is first grouping QPs by tenant and then performing the mapping. Thus, this approach can provide as many tenants as the number of rate limiters, while the QoS of each tenant is guaranteed.

Currently, MasQ exploits SR-IOV VF to implement QoS. The reason is that SR-IOV is well supported by commodity RNICs, and the way to configure QoS policies per VF is also well studied [5]. The main difference is that we do not pass VF directly to VMs; instead, we ask the MasQ's backend driver to determine which VF is used to serve the requests (e.g., "create_qp") from different tenants or different VMs or different applications as shown in Fig. 3. Note that the mapping policy between QPs and VFs is left for future work. However, the solution leveraging SR-IOV VF as the hardware-based rate limiter is a little "overkill". We suggest that the future RNIC support more lightweight and finer-grained rate limiters, such as providing QoS guarantees to each QP group.

3.3.4 Connectionless transport. This paper mainly focuses on connection-based transport, such as the reliable connection (RC) mode. However, it is well known that RC-mode RDMA faces the challenge of scalability issues [20, 42]. Therefore, supporting datagram-based RDMA is also very important. We should note that extending the proposed MasQ to support datagram-based RDMA is straightforward. Since network information will be carried by the work queue element (WQE) of each RDMA datagram message, we can ask the user space library to forward all datagram WQEs through the control path instead of the zero-copy data path. Therefore, the *RConnrename* can easily replace the virtual network information by the physical one before forwarding the WQE to the real device driver. Once receiving the WQE, the RNIC can directly write the message data to the application's user space through DMA. MasQ handles the datagram WQEs in a similar way as vRDMA [39] and FreeFlow [33]. Therefore, we omit the detailed discussions. Furthermore, some software, such as Mellanox's VMA [7], may depend on the raw Ethernet mode. Note that raw Ethernet can also be supported by MasQ in exactly the same way as supporting the datagram transport.

4 EVALUATION

In this section, we will answer the following two questions by comparing MasQ with state-of-the-art solutions – 1) whether MasQ can provide competitive performance and 2) whether all virtualization requirements of VPC are fulfilled – and we summarize the main conclusions here:

(1) MasQ achieves almost the same performance as Host-RDMA among all test cases against both benchmarks and typical applications. MasQ's overhead mainly resides on the control path, which will result in a slightly longer time to setup

Table 3: Experiment settings.

Parameter		Setting
Server	CPU	Two Intel Xeon E5-2690 v4 2.60 GHz 14 core CPUs
	Memory	DDR4 96 GB
	OS	Ubuntu 14.04.1 (3.13.0-32-generic)
VM	CPU	14 cores
	Memory	32 GB
	OS	Ubuntu 14.04.1 (3.13.0-32-generic)
Container	CPU	14 cores
	Memory	32 GB
	Image	Ubuntu 14.04
Topology		Direct
RNIC		Mellanox CX-3 Pro 40 Gbps RoCE
RDMA driver		Mellanox OFED-4.0-2.0.0.1 for Ubuntu 14.04
Hypervisor		QEMU-2.1.5, Docker-17.03.0-ce
Virtual TCP/IP network		Open vSwitch-2.7.0 & VLXAN, Weave-2.5.2 & VXLAN

connections. However, for most applications, connection establishment is not on the critical path and thus has little effect on the overall performance.

(2) MasQ's flexibility makes it very effective to realize all requirements of VPC, such as supporting a large number of instances and providing isolations for traffic, performance, and security.

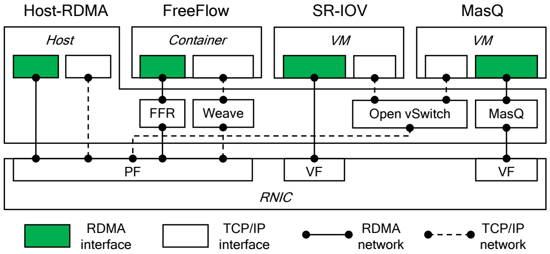
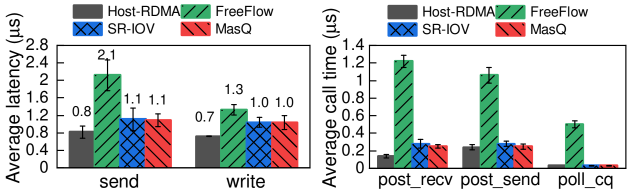


Figure 7: Architecture overview of four candidates under test.

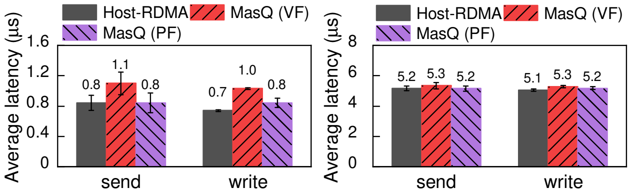
4.1 Methodology

As shown in Fig. 7, four candidates are evaluated in this section, including Host-RDMA to identify the upper-bound performance of virtualization technologies. As shown in Table 3, all candidates are evaluated in the same testbed consisting of two servers directly connected by Mellanox CX-3 Pro 40 Gbps RoCE RNICs. Each server, which runs Ubuntu 14.04.1 with kernel version 3.13.0-32-generic, is equipped with two Intel Xeon E5-2690 2.60 GHz 14 core CPUs and 96 GB RAM. We run virtual machines using QEMU (v2.1.5) [16] and set up a VXLAN-based virtual TCP/IP network using Open vSwitch (v2.7.0) [10]. To prevent VM resources from being the bottleneck, we provide 14 cores and 32 GB memory for each VM. Since FreeFlow is only available in containers, we also use Docker (v17.03.0-ce) [1] to run containers and set up a VXLAN-based virtual TCP/IP network using Weave (v2.5.2) [15]. We use Docker's runtime options [14] to limit the CPU and memory resources of each container to the same settings as the VM. Unless otherwise specified, all abovementioned hardware resources are on the same NUMA node.



(a) 2 B latency **(b) Verbs overhead**

Figure 8: RDMA latency between a pair of VMs on different hosts and the overhead of Verbs.



(a) 2 B latency **(b) 16 KB latency**

Figure 9: MasQ performs better with PF than VF.

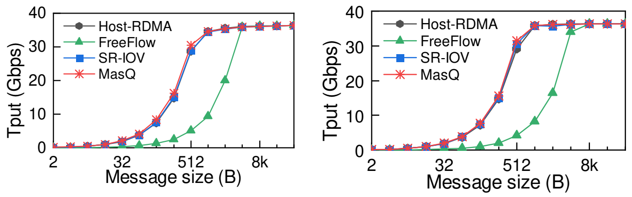
4.2 RDMA performance and overhead

In this subsection, we will evaluate the basic communication performance of MasQ through both the perftest and MPI benchmarks. Furthermore, we will determine MasQ's overhead by profiling the process of RDMA's connection establishment.

4.2.1 *Basic RDMA performance.*

Since high-performance data communication is one of the most important reasons why applications choose RDMA, we first evaluate MasQ's latency, throughput, and scalability to show that the proposed MasQ can provide competitive performance for applications running in the VMs of public clouds. RDMA supports both two-sided (`send`) and one-sided (`write`/`read`) operations. We use `ib_send_lat` and `ib_send_bw` to test two-sided performance, and `ib_write_lat` and `ib_write_bw` to test one-sided performance. All these tools come from the perftest suite (v3.0) [12].

Latency: we measure the latency of `send`/`write` by sending/writing a 2-byte message 1000 times. Furthermore, to determine how much overhead is introduced by the software of the data path, we also measure the average call time of relevant Verbs with the standard system API for time acquisition, i.e., `gettimeofday`. From Fig. 8a, we can see that MasQ has the same performance as that of SR-IOV. We also observe that the latency of VF-based virtual networks (MasQ/SR-IOV) is a slightly longer than that of PF-based (Host-RDMA) virtual networks. We suspect more complex communication and resource management on the RNIC when dealing with VF to account for lower performance. However, the difference is negligible considering that the network queueing delay is usually higher than 10 μs [45]. In addition, if the best-effort service model is adopted, MasQ can map VMs to PF instead of VF so that applications can achieve almost the same RDMA latency as that of Host-RDMA, as shown in Fig. 9. `send` latency on FreeFlow is



(a) send throughput

(b) `write` throughput

Figure 10: RDMA throughput between a pair of VMs on different hosts.

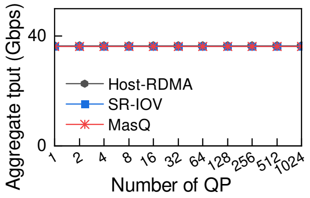


Figure 11: Aggregate throughput of multiple QP connections.

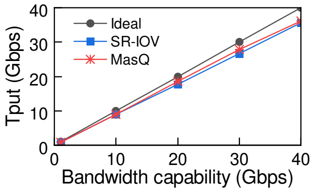


Figure 12: MasQ can effectively control the rate of flows.

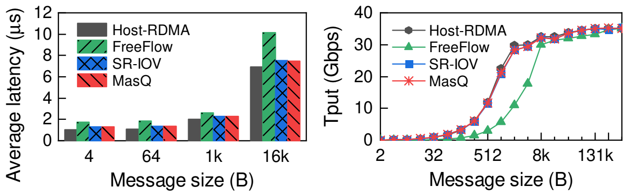
approximately 2.6 and 2 times higher than that on Host-RDMA and MasQ, respectively, because FreeFlow needs to redirect each data path operation to its software-based backend, i.e., FFR, which introduces extra overhead to the corresponding Verbs and negatively affects end-to-end latency. As shown in Fig. 8b, the time to perform FreeFlow's data path Verbs is at least 5 times higher than that for Host-RDMA, while MasQ and SR-IOV remain the same. Since MasQ is built on hardware and directly maps the data path resources, e.g., QPs and MRs, to the VM, there is no more overhead on the data path. This fact guarantees that MasQ can obtain almost the same RDMA latency as that of SR-IOV or even Host-RDMA, and the experimental results confirm this expectation.

Throughput: we measure throughput by sending and writing different sizes of messages ranging from 2 B to 32 KB through one QP connection. The result in Fig. 10 shows that MasQ achieves the same throughput for all sizes of messages compared to Host-RDMA and SR-IOV. FreeFlow, however, has lower throughput when the message size is not large enough (~8 KB). This is because FreeFlow consumes more CPU cycles in the data path than does MasQ when sending one message. This negatively affects throughput when the FFR CPU is a bottleneck. This finding again confirms that MasQ introduces no overhead into the data path.

Scalability: to prove that MasQ has the same scalability as Host-RDMA and SR-IOV, we evaluate the aggregate throughput of MasQ over multiple QP connections. We use the tool `ib_write_bw` and set the message size to 65536 bytes. As shown in Fig. 11, when we increase the number of QPs from 1 to 1024, the throughputs of MasQ and SR-IOV remain the same. This proves that MasQ introduces negligible overheads in the critical path and has the same scalability as Host-RDMA and SR-IOV.

4.2.2 *MPI benchmark performance.*

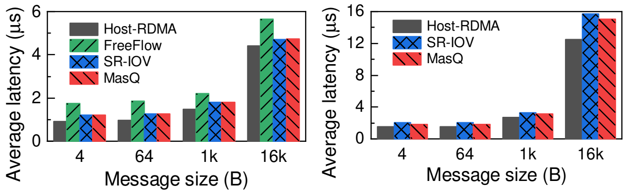
MPI is the standard communication paradigm adopted by most HPC applications. Therefore, its performance is one of the most important metrics. We exploit the MVAPICH2 (v2.3.2) [9] and built-in OSU micro-benchmarks to evaluate the performance of different MPI operations. The tests run with two processes distributed on two VMs/hosts/containers in our testbed. The results are shown in Fig. 13. For both the communication latency and bandwidth, MasQ has the same performance as that of SR-IOV. Furthermore, we compare the performance of collective MPI operations that are widely used by HPC applications. Typically, there is a set of collective operations, such as broadcast, scatter/gather, and allreduce. In this paper, we choose the broadcast and allreduce operations as the representative. It should be noted that all reduce-related operations failed to run on FreeFlow due to memory corruption. Therefore, we omit FreeFlow in some tests. Again, as shown in Fig. 14, we can find that the proposed MasQ can obtain the same or even better performance than that of SR-IOV. It should be noted that both SR-IOV and MasQ perform slightly worse than Host-RDMA in all tests. However, MasQ can achieve better performance by mapping VMs to PF.



(a) Latency

(b) Bandwidth

Figure 13: MPI point-to-point performance.



(a) Broadcast

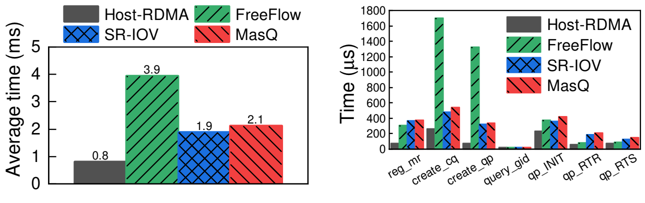
(b) Allreduce

Figure 14: MPI collective performance.

4.2.3 *Control path overhead.*

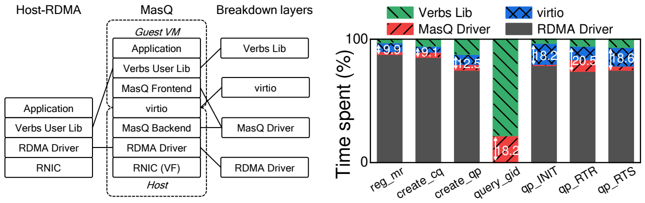
The overhead introduced by MasQ mainly resides on the control path and slightly increases the delay of connection establishment. To clarify these effects, a simple program following the procedure of Fig. 1 is used. The program establishes a specific number of connections at a time. In the end, it reports the average delay to establish one RDMA connection and the average call time of each Verbs. For MasQ, we also use ftrace [2] to measure the execution time of all critical functions in each Verbs' kernel routine. Then, we can determine the cost of each software layer illustrated in Fig. 16a.

Currently, we do not consider the overhead of the remote controller for the following three reasons. 1) Although the round trip time for querying the controller usually takes approximately 100 μs, it is not necessary at most times with the help of a local cache. 2) For latency-sensitive applications, the controller can push down the mapping information in advance. As discussed in Sec 3.3.1, the overhead to maintain such a mapping table is negligible. 3) In



(a) Average delay **(b) Breakdown**

Figure 15: Performance of RDMA connection establishment.



(a) Software layers. **(b) Breakdown.**

Figure 16: Overhead breakdown of MasQ.

addition, both FreeFlow and SR-IOV solutions depend on remote controllers, so it is fair to omit the controller factor here.

As shown in Fig. 15a, MasQ takes approximately 2.1 ms to establish an RDMA connection, which is 0.2 ms longer than does SR-IOV. The overhead is mainly caused by virtio-based frontend/backend communication, which introduces a delay of approximately 25 μs for each Verbs. Since most RDMA-based applications, such as HPC and distributed machine learning applications, maintain long-lived RDMA connections, the connection overhead has little effect on the application's overall performance regardless of the message size. However, for short connections, it takes slightly longer (~11% in our test cast) time to establish an RDMA connection over MasQ than that over SR-IOV based solutions. We also observe that both MasQ and SR-IOV take much longer to establish a connection than does Host-RDMA. This gap is caused by VF, which increases the processing delay of each control path Verbs on the RNIC. FreeFlow takes the slowest time to establish an RDMA connection, which is 1.8 and 4.8 times longer than MasQ and Host-RDMA, respectively. This is because FreeFlow must virtualize data path resources on the control path, which requires additional memory allocation and mapping operations. As shown in Fig. 15b, these operations introduce large overhead for Verbs such as "reg_mr", "create_cq" and "create_qp".

For all control path Verbs on MasQ, Fig. 16b shows their detailed cost on each software layer. Note that the cost of MasQ (labeled "MasQ Driver") is obtained by adding up the delay introduced by both frontend and backend components of MasQ. We can see that more than 80% of the overhead actually comes from the RDMA kernel driver and user space library, while less than 20% comes from MasQ. This proves that the implementation of MasQ is very efficient.

4.3 Feature validation

In this subsection, we will validate whether all claimed features of MasQ have been effectively achieved.

Table 4: Cost of security-related operations.

Caller	Basic op. function	Time cost (μs)
update_rules	`insert_rule()`	1.5
	`reset_conn()`	518
modify_qp_RTR	`valid_conn()`	2.5
	`insert_conn()`	1.5
destroy_qp	`delete_conn()`	1.5

Table 5: Maximum number of VMs.

RDMA Virtualization	Max #VM	Limitation factor
SR-IOV	8	Non-ARI PCIe
MasQ	160	Host memory

4.3.1 QoS and performance isolation. The proposed MasQ implements QoS, i.e., rate limiting, by exploiting SR-IOV VFs. We first show that when the bandwidth of a VF is well limited, the aggregate bandwidth of all corresponding MasQ RNICs can be limited accordingly. We start a single flow using `ib_write_bw` between two VMs on different hosts. We limit the flow rate of the VF and set the maximum bandwidth from 1 Gbps to 40 Gbps. From Fig. 12, we can find that the controlled bandwidth is close to the bandwidth we set. It should be noted that MasQ achieves this without any CPU overhead.

MasQ can isolate performance between two VMs, which means that the throughput of one VM can be accurately regulated without affecting the other. We demonstrate this by running two concurrent flows between VM pairs and report their average throughput over each second. As shown in Fig. 17, two VMs first obtain similar bandwidth of approximately 18.9 Gbps in the absence of rate limiting. Then, the bandwidth of VM 0 is limited to 10 Gbps and then to 5 Gbps, and we find that VM 1 can quickly consume all spare bandwidths since there is no limitation on it.

4.3.2 Security. In the rest, we will determine whether RDMA connections can be successfully torn down if the corresponding rules request it. As shown in Fig. 17, VM 0's bandwidth successfully drops to 0 once the security rule kicks off. Now, let us analyze the cost of the above mechanism. Generally, *RConntrack* exposes two types of operations, one for maintaining security rules and the other for tracking RDMA connections. We use ftrace to measure performance. Table 4 shows that the delay to reset a connection is approximately 518 μs, while other operations can finish in a few microseconds. Furthermore, as shown in Fig. 18, we find that the cost of resetting an RDMA connection mainly comes from the RNIC and varies with different traffic loads. Generally, connection reset is faster on PF than VF. In addition, a longer delay is expected with increased traffic load. Note that connection reset is only triggered when the IT facility updates security rules and never introduces overhead into normal RDMA communication. Although we believe that the overhead is acceptable, we strongly suggest that future RNICs provide more efficient ways to stop a QP.

4.3.3 Scalability. As discussed in section 3, MasQ enables the flexibility to compose virtual devices for VMs at a finer granularity, i.e., the QP-level. To demonstrate the benefit, we launch as many VMs as possible on the host, where each VM's vCPU is set to 1, and

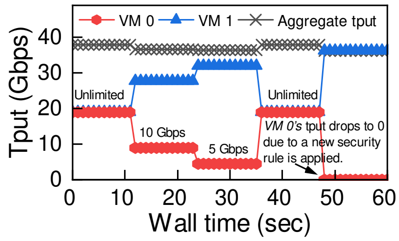


Figure 17: Applying rate limiting and security rules to *VM 0*.

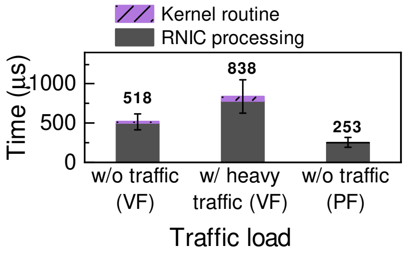


Figure 18: Cost breakdown to reset an RDMA connection.

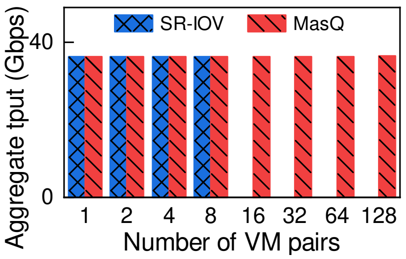


Figure 19: Aggregate throughput of VMs.

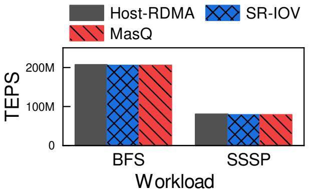


Figure 20: Graph500 performance.

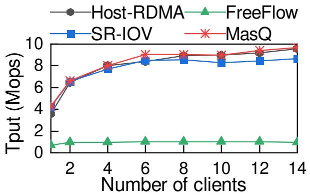


Figure 21: KVS performance.

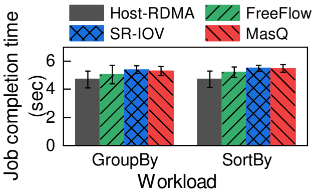


Figure 22: Spark performance.

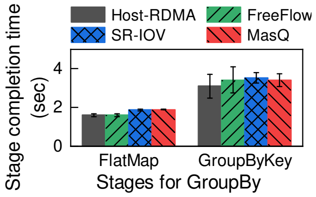


Figure 23: Breakdown of Spark Groupby.

memory is set to 512 MB. As shown in Table 5, MasQ's maximum number of VM on a single host is approximately 20 times that of SR-IOV. Note that the number of VMs supported by MasQ in this experiment is limited by the capacity of the main memory. Therefore, we can increase the number of VMs simply by either adding more DRAM or reducing the memory size of each VM. Furthermore, We randomly select a certain number of VMs and run one flow using `ib_write_bw` for each. Fig. 19 shows the aggregate throughput of all flows and proves that MasQ can obtain scalability without sacrificing communication performance.

4.4 Application performance

In this section, we show the performance of Graph500, KVS, and Spark. Graph500 [3] is an MPI benchmark for data-intensive HPC workloads. KVS is an implementation of HERD [31], which is the state-of-the-art solution for RDMA-based key-value stores. Spark is a large-scale data analytic engine. We compare the application performance on MasQ with that on platforms using Host-RDMA, FreeFlow, and SR-IOV.

4.4.1 Graph500. There are three kernels in Graph500, kernel 1 is used to construct graphs according to the input parameters, kernel 2 is used to perform the breadth-first search (BFS) on the graph constructed by kernel 1, and kernel 3 performs single source shortest path (SSSP) computation on the same graph. All results are validated by the validation procedure provided by the benchmark.

We run graph500 (v3.0.0) BFS and SSSP tests with parameters "scale=26" and "edge_factor=16", which consumes approximately 26 GB and 38 GB memory, respectively. The tests are run with 16 MPI processes that distribute on two VMs/hosts in a round-robin fashion. Each test is run five times, and the average results are reported. The performance metric TEPS means "traversed edges per second". Fig. 20 shows the performance of the two tests on different platforms (we omit FreeFlow here because we were unable to run this benchmark on FreeFlow due to memory corruption). Compared with Host-RDMA and SR-IOV, MasQ has almost no performance degradation.

4.4.2 KVS. Key-value stores are widely used in large-scale web systems. We use them to compare the performance of different virtual networks driven by a large number of small messages. Both the program and the benchmark are derived from `rdma_bench` [13], and we revise its RPC to use RC only. To prevent the CPU from becoming a bottleneck, we run a KVS server with 14 workers. Each worker owns a key space partition populated with 8 million key-value pairs, each of which maps a 16-byte key to a 32-byte value. Then, we use a variable number of client threads on a separate machine to issue requests. The workload consists of 95% GET and 5% PUT operations, with keys chosen uniformly at random from the inserted keys.

Fig. 21 shows the aggregate throughput of the KVS server. The maximum throughput on MasQ and Host-RDMA is 9.7 Mops. At this point, the RNIC is the bottleneck and we cannot further improve the throughput by adding more clients or workers. The highest throughput of SR-IOV is approximately 1 Mops lower than that of MasQ due to the cost of IOMMU (e.g., Intel VT-d). In principle, MasQ does not require IOMMU to perform DMA address translation, thus avoiding such overhead. FreeFlow has the worst performance, with throughput only up to approximately 1 Mops. This is because FreeFlow requires its software-based backend, i.e., FFR, to forward each data path operation. Thus, FFR quickly becomes a bottleneck. The only way to eliminate this bottleneck is to assign more forwarding threads to the FFR but at the cost of consuming more CPU cores.

4.4.3 Spark. Apache Spark is a popular platform for big data applications. The RDMA extension for Spark is published in binary from OSU's high-performance big data project [4]. We run RDMA-Spark (v0.9.5) and the basic benchmarks (v0.9.2) [11], GroupBy and SortBy, on two nodes. Each of them runs one worker and one executor. Moreover, we restrict the worker cores to 4 and memory to 32 GB on each node. Each benchmark is performed with 8 mappers and 8 reducers so that the job is running with a full subscription on 8 cores of the two nodes. The dataset has 131072 key-value pairs with a 1 KB value size. The experiment runs ten times, and the average job execution time is shown in Fig. 22. According to the results, we find that MasQ can obtain similar performance as that of SR-IOV.

We also observe that the performance of MasQ and SR-IOV is slightly worse than that of Host-RDMA and FreeFlow. We believe that it is mainly caused by the overhead of the VM. To reveal VM's effects, we do a breakdown of GroupBy job by DAG stages. This can be done with the help of Spark's application monitoring tool [8]. The GroupBy job is divided into two stages, FlatMap and GroupByKey, and executed by the Spark job scheduler sequentially. The first stage (FlatMap) has no network communication, but the second stage (GroupByKey) generates much network traffic due to data shuffling. As shown in Fig. 23, FlatMap consumes more time on VM (MasQ/SR-IOV) than on the host (Host-RDMA) and container (FreeFlow). However, since FreeFlow introduces overhead in network communication, MasQ and FreeFlow end up with almost the same completion time in the second stage. It should be noted that MasQ achieves this with no CPU involvement, while FreeFlow consumes at least one CPU core.

5 DISCUSSION

To achieve good performance, RoCEv2 requires a lossless network that is achieved by enabling priority-based flow control (PFC) within the network. The PFC pauses all related upstream sending queues once it detects a risk of packet loss. Although PFC helps to reduce the packet loss and the retransmission overhead, PFC storms may occur and punish victim flows. To address this problem, advanced congestion control algorithms are expected either to minimize the possibility of PFC storms or to enable PFC-free deployment of RDMA networks, such as DCQCN [45], NDP [23] and HPCC [35]. Since MasQ is orthogonal to them, any advanced algorithm can be used, and all MasQ's good features still hold.

Live migration for RDMA-capable VM is a difficult job for both hardware- and software-based virtualization solutions. The main reason is that RDMA bypasses the kernel as well as the hypervisor, so it is difficult to mark dirty pages during migration. Furthermore, one-sided RDMA operations bypass the software on the receiver side, so the software never knows which pages are modified by the remote peer. Recently, AccelNet [22] proposed a live migration solution with the help of applications. To migrate a VM, the application actively disconnects all RDMA connections, falls back to TCP/IP, and then starts migration. After migration, all RDMA connections should be re-established explicitly. We believe that this solution also applies to MasQ.

Modern data centers may use packet headers to perform network diagnosis or achieve other functionalities. Most of the features only rely on the information of the underlay network, but some may require tenants' information (e.g., virtual IPs) in the overlay network. Generally, MasQ has the ability to provide such information, which can be achieved by maintaining a mapping table between the (physical IP, QPN) and the virtual IP. Compared with tunnel-based solutions, this method introduces the overhead of table maintenance. However, one of the advantages is that it requires no additional header so that MasQ can carry more payload given a fixed MTU.

6 CONCLUSIONS

RDMA has become increasingly more important for improving the performance of large-scale applications. However, RDMA is still unavailable in VMs of public clouds due to the lack of a practical RDMA network virtualization solution. To fill this gap, we propose a software-defined RDMA virtualization solution, namely, MasQ, to achieve this goal with negligible overhead. In particular, MasQ proposes a low-cost solution, namely, *vBond*, to realize a virtual RoCE device by dynamically binding the virtual Ethernet and RDMA interfaces. Furthermore, to isolate tenants' RDMA traffic without degrading communication efficiency, a new per-connection technique, instead of the traditional per-packet technique, namely, *RConnrename*, is proposed. Then, we thoroughly discuss the requirements of applying security rules to virtual RDMA networks and propose a Neutron-compatible solution, namely, *RConntrack*, to guarantee that all RDMA connections are properly protected.

This work does not raise any ethical issues.

ACKNOWLEDGMENTS

We would like to thank our shepherd and the anonymous SIGCOMM reviewers for their valuable comments. We would also like to thank our former colleague, JinZhao Su and Ming Zhang for their contributions to this work. This work was partially supported by the NSFC under Grant No.: 61672499.

REFERENCES

[1] 2019. Docker. https://www.docker.com/. (2019).
[2] 2019. Ftrace. https://www.kernel.org/doc/Documentation/trace/ftrace.txt. (2019).
[3] 2019. Graph 500. https://graph500.org/. (2019).
[4] 2019. High-Performance Big Data. http://hibd.cse.ohio-state.edu/#spark. (2019).
[5] 2019. HowTo Configure QoS over SR-IOV. https://community.mellanox.com/s/article/howto-configure-qos-over-sr-iov. (2019).
[6] 2019. Mellanox ConnectX-6 Dx. https://www.mellanox.com/products/ethernet-adapter-ic/connectx-6-dx-ic. (2019).
[7] 2019. Mellanox VMA. https://github.com/Mellanox/libvma. (2019).
[8] 2019. Monitoring Spark applications. https://spark.apache.org/docs/latest/monitoring.html. (2019).
[9] 2019. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. http://mvapich.cse.ohio-state.edu/. (2019).
[10] 2019. Open vSwitch. https:// www.openvswitch.org/. (2019).
[11] 2019. OSU HiBD Benchmarks. http://hibd.cse.ohio-state.edu/#microbenchmarks. (2019).
[12] 2019. Perftest package. https://community.mellanox.com/docs/DOC-2802. (2019).
[13] 2019. RDMA-bench. https://github.com/efficient/rdma_bench. (2019).
[14] 2019. Runtime options with Memory, CPUs, and GPUs. https://docs.docker.com/config/containers/resource_constraints/. (2019).
[15] 2019. Weave Net. https://www.weave.works/. (2019).
[16] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41.
[17] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 14 pages. https://doi.org/10.1145/3302424.3303968
[18] 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

[19] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. 2011. DevoFlow: Scaling flow management for high-performance networks. In *Proceedings of the ACM SIGCOMM 2011 conference*. 254–265.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 401–414.

[21] Shiqing Fan, Fang Chen, Holm Rauchfuss, Nadav Har’El, Uwe Schilling, and Nico Struckmann. 2017. Towards a Lightweight RDMA Para-Virtualization for HPC. In *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*.

[22] 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*.

[23] 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

[24] InfiniBand Trade Association 2010. *InfiniBand Architecture Specification Release 1.2.1 Annex A16: RoCE*. InfiniBand Trade Association. Rev.1.2.1.

[25] InfiniBand Trade Association 2014. *InfiniBand Architecture Specification Release 1.2.1*. InfiniBand Trade Association. Rev.1.2.1.

[26] InfiniBand Trade Association 2014. *InfiniBand Architecture Specification Release 1.2.1 Annex A17: RoCEv2*. InfiniBand Trade Association. Rev.1.2.1.

[27] Internet Engineering Task Force 2007. *A Remote Direct Memory Access Protocol Specification*. Internet Engineering Task Force. RFC5040.

[28] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. 2012. High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC ’12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 35, 35 pages.

[29] Cheng Jin, Abhinav Srivastava, and Zhi-Li Zhang. 2016. Understanding security group usage in a public IaaS cloud. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9.

[30] 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

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 437–450.

[32] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201.

[33] Daehyeok Kim, Tianlong Yu, Hongqiang Harry Liu, Yibo Zhu, Jitu Padhye, Shachar Raindel, Chuanxiong Guo, Vyas Sekar, and Srinivasan Seshan. 2019. FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds. In *NSDI*. 113–126.

[34] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. 2010. Onix: A distributed control platform for large-scale production networks.

[35] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, et al. 2019. HPCC: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*. 44–58.

[36] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, USA, 203–215. https://doi.org/10.1109/MICRO.2014.28

[37] M. Mahalingam, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. 2014. 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. (August 2014).

[38] Jonas Pfefferle, Patrick Stuedi, Animesh Trivedi, Bernard Metzler, Ionnis Koltsidas, and Thomas R Gross. 2015. A hybrid I/O virtualization framework for RDMA-capable network interfaces. *ACM SIGPLAN Notices* 50, 7 (2015), 17–30.

[39] Adit Ranadive and Bhavesh Davda. 2012. Toward a paravirtual vRDMA device for VMware ESXi guests. *VMware Technical Journal, Winter 2012* 1, 2 (2012).

[40] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Syst. Review (OSR* (2008), 103.

[41] Shin-Yeh Tsai, Mathias Payer, and Yiying Zhang. 2019. Pythia: Remote Oracles for the Masses. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC ’19)*. USENIX Association, USA, 693–710.

[42] Shin-Yeh Tsai and Yiying Zhang. 2017. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 306–324.

[43] 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 2019)*. Association for Computing Machinery, New York, NY, USA, 179 C192. https://doi.org/10.1145/3313808.3313813

[44] Zhenghong Wang and Ruby B. Lee. 2007. New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 494–505. https://doi.org/10.1145/1273440.1250723

[45] 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

[46] Danyang Zhuo, Kaiyuan Zhang, Yibo Zhu, Hongqiang Harry Liu, Matthew Rockett, Arvind Krishnamurthy, and Thomas Anderson. 2019. Slim:OS kernel support for a low-overhead container overlay network. In *16th USENIX Symposium on Networked Systems Design and Implementation NSDI 19*. 331–344.

APPENDIX

Appendices are supporting material that has not been peer-reviewed.

A SOFTWARE-BASED I/O VIRTUALIZATION

In this section, we will introduce two software-based I/O virtualization methods for RDMA: paravirtualization and hybrid I/O virtualization.

A.1 I/O para-virtualization

Paravirtualization splits the I/O device’s driver into a frontend and a backend driver, which runs in virtual machine (VM) and host, respectively. To complete an I/O operation, the frontend driver first forwards the command to the backend driver, which further delivers it to the physical device. Currently, the virtio framework is the de facto way to implement such a frontend and backend driver. The core of virtio is that it provides a mechanism, namely, *virtqueue*, to exchange the I/O command between the frontend and backend driver. Specifically, to deliver an I/O command, there are three steps. 1) The frontend driver in the VM first enqueues the I/O command into a pre-allocated *virtqueue* and then calls the `kick` API. The `kick` call will VM-exit into the host kernel and notify the corresponding backend driver that the arrival of a new I/O command. 2) After waking up, the backend driver first dequeues the I/O command from the *virtqueue* and then forwards it to the physical device for further processing. Once it obtains the result, the backend driver enqueues it into the *virtqueue* and injects an interrupt into the VM. 3) The corresponding interrupt handler will execute the callback function registered by the frontend driver to complete the I/O request’s subsequent processing.

A.2 Hybrid I/O virtualization

Hybrid I/O virtualization is proposed by HyV tailored for RDMA. It adopts paravirtualization to virtualize the control path but implements a zero-copy data path. Therefore, only control operations, such as creating a QP and registering a memory region, are virtualized in a “paravirtual” manner. Data path operations, such as posting a WQE and transmitting user data, are performed directly in a zero-copy manner. For example, to create a QP, a system call into the frontend driver is issued by an application running in a VM. Upon receiving the call, the frontend driver forwards the "create_-qp" command to the backend driver through the virtio *virtqueue*. The backend driver then delivers the command to the unmodified host device driver to create a QP in RNIC. In addition, the frontend driver maps the newly created QP into the address space of the guest application. Thereafter, I/O requests by the application, such as "post_send", are posted directly to the QP through the unmodified user space device driver in a zero-copy manner. similarly, as the application memory is registered similarly as in the QP creation, RNIC can also directly access the application data. In the next Appendix, we will present the way in which to establish the direct memory mapping between the guest application and the RNIC to enable the above zero-copy data operations.

B MEMORY MAPPING

In this section, we will present how memories are correctly mapped between VM and hardware in both directions, including both from VM to hardware and from hardware to VM.

B.1 From device to VM

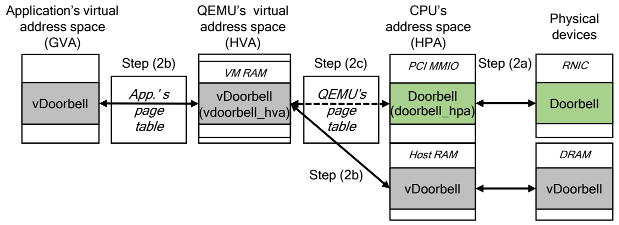


Figure 24: Steps to map the RNIC's registers (e.g., Doorbell) to the application in the VM.

Generally, the application accesses RNIC hardware registers through MMIO. To allow for this work in VM, we should introduce an additional mapping process to map the RNIC's registers, e.g., Doorbell, to the application's virtual address space in the VM. We illustrate this through an example of allocating a Doorbell during device initialization. (1) To initialize a device context, an application should call Verbs "open_device". (2) The corresponding library routine splits the handling process into three parts, as shown in Fig. 24. (2a) The backend driver is requested to allocate a Doorbell on the RNIC. Then, the backend driver obtains the Doorbell's address in the CPU's address space, i.e., HPA space. We record the address as `doorbell_hpa`. (2b) The frontend driver is requested to allocate a piece of memory with the same size as the allocated Doorbell and then maps it to the application's virtual address space. We record the memory (as well as the backing physical memory) as vDoorbell and its address in QEMU's virtual address space (i.e., HVA space) as `vdoorbell_hva`. (2c) The backend driver is requested to establish a mapping between the `vdoorbell_hva` and `doorbell_hpa` by adapting QEMU's page table so that access to the vDoorbell will be routed to the real Doorbell on the RNIC. To this point, a Doorbell on the RNIC can be accessed by the application in the VM through MMIO.

B.2 From VM to device

To allow for RNIC to access the user memories, e.g., QPs and MRs, in the VM through DMA, mappings between the virtual address and physical address of these memories should be configured into the RNIC's memory translation table (MTT). For applications running in VM, this can be achieved with extra memory pinning and translation. We use the example of creating a QP to illustrate this process. (1) To create a QP, an application should call the Verbs "create_qp". (2) The corresponding library routine first allocates a piece of memory for the QP. We record the virtual address of the memory as `qp_gva`. Then, it forwards the request with the memory information to the frontend driver. (3) The frontend driver first pins the memory and translates `qp_gva` to `qp_gpa` by walking the application's page table and then forwards the request with the memory information to the backend driver, including the address mapping (`qp_gva`, `qp_gpa`). (4) Upon receiving the request, the backend driver first translates `qp_gpa` to `qp_hva`. Then, it again pins the memory and translates `qp_hva` to `qp_hpa` by walking QEMU's page table. Futhermore, the backend driver has both `qp_gva` and `qp_hpa` of the QP memory, and the only thing left is to write them into the RNIC's MTT. To this point, a QP is successfully created for the application in the VM and can also be accessed by the RNIC through DMA. For MR, the creation process is the same as that illustrated above.