FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds

Daehyeok Kim and Tianlong Yu, *Carnegie Mellon University;*
Hongqiang Harry Liu, *Alibaba;* Yibo Zhu, *Microsoft and Bytedance;*
Jitu Padhye and Shachar Raindel, *Microsoft;* Chuanxiong Guo, *Bytedance;*
Vyas Sekar and Srinivasan Seshan, *Carnegie Mellon University*

https://www.usenix.org/conference/nsdi19/presentation/kim

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

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

ISBN 978-1-931971-49-2

FreeFlow: Software-based Virtual RDMA Networking for Containerized Clouds

Daehyeok Kim[1], Tianlong Yu[1], Hongqiang Harry Liu[3], Yibo Zhu[2], Jitu Padhye[2]
Shachar Raindel[2], Chuanxiong Guo[4], Vyas Sekar[1], Srinivasan Seshan[1]
[1]*Carnegie Mellon University,* [2]*Microsoft,* [3]*Alibaba,* [4]*Bytedance*

Abstract

Many popular large-scale cloud applications are increasingly using containerization for high resource efficiency and lightweight isolation. In parallel, many data-intensive applications (*e.g.,* data analytics and deep learning frameworks) are adopting or looking to adopt RDMA for high networking performance. Industry trends suggest that these two approaches are on an inevitable collision course. In this paper, we present *FreeFlow*, a software-based RDMA virtualization framework designed for containerized clouds. *FreeFlow* realizes virtual RDMA networking purely with a software-based approach using commodity RDMA NICs. Unlike existing RDMA virtualization solutions, *FreeFlow* fully satisfies the requirements from cloud environments, such as isolation for multi-tenancy, portability for container migrations, and controllability for control and data plane policies. *FreeFlow* is also transparent to applications and provides networking performance close to bare-metal RDMA with low CPU overhead. In our evaluations with TensorFlow and Spark, *FreeFlow* provides almost the same application performance as bare-metal RDMA.

1 Introduction

Developers of large-scale cloud applications constantly seek better performance, lower management cost, and higher resource efficiency. This has lead to growing adoption of two technologies, namely, *Containerization* and *Remote Direct Memory Access (RDMA)* networking.

Containers [7, 11, 6] offer lightweight isolation and portability, which lowers the complexity (and hence cost) of deploying and managing cloud applications. Thus, containers are now the de facto way of managing and deploying large cloud applications.

RDMA networking offers significantly higher throughput, lower latency and lower CPU utilization than the standard TCP/IP based networking. Thus, many data-intensive applications, *e.g.,* deep learning and data analytics frameworks, are adopting RDMA [24, 5, 18, 17].

Unfortunately, the two trends are fundamentally at odds with each other in clouds. The core value of containerization is to provide an efficient and flexible management to applications. For this purpose, containerized clouds need containers to have three properties in networking:

- *Isolation*. Each container should have its dedicated network namespace (including port space, routing table, interfaces, etc.) to eliminate conflicts with other containers on the same host machine.
- *Portability*. A container should use virtual networks to communicate with other containers, and its virtual IP sticks with it regardless which host machine it is placed in or migrated to.
- *Controllability*. Orchestrators can easily enforce control plane policies (*e.g.,* admission control, routing) and data plane policies (*e.g.,* QoS, metering). This property is particularly required in (multi-tenant) cloud environments.

These properties are necessary for clouds to freely place and migrate containers and control the resources each container can use. To this end, in TCP/IP-based operations, networking is fully virtualized via a software (virtual) switch [15].

However, it is hard to fully virtualize RDMA-based networking. RDMA achieves high networking performance by offloading network processing to hardware NICs, bypassing kernel software stacks. It is difficult to modify the control plane states (*e.g.,* routes) in hardware in shared cloud environments, while it is also hard to control the data path since traffic directly goes between RAM and NIC via PCIe bus.

As a result, several data-intensive applications (*e.g.,* TensorFlow [24], CNTK [5], Spark [18], Hadoop [17]) that have adopted both these technologies, use RDMA only when running in dedicated bare-metal clusters; when they run in shared clouds, they have to fundamentally eschew the performance benefits afforded by RDMA. Naturally, using dedicated clusters to run an application is, however, not cost efficient both for providers or for customers.

Thus, our goal in this paper is simple: we want cloud-based, containerized applications to be able to use RDMA as

Property	Native	SR-IOV [21]	HyV [39]	SoftRoCE [36]
Isolation	✗	✓	✓	✓
Portability	✗	✗	✓	✓
Controllability	✗	✗	✗	✓
Performance	✓	✓	✓	✗

Table 1: RDMA networking solutions that can be potentially used for containers.

efficiently as they would in a dedicated bare-metal cluster; while at the same time achieving the isolation, portability and controllability requirements in containerized clouds. [1]

Currently, there is no mature RDMA virtualization solutions for containers.[2] Table 1 summarizes some important options that can potentially be extended to support containers, although they fail to achieve the key requirements or have to do so at a substantial performance cost.

For instance, hardware-based I/O virtualization techniques like SR-IOV [21] have fundamental portability limitations [39, 28], since they require reconfiguration of hardware NICs and switches to support migrations of containers. Control path virtualization solutions, such as HyV [39], only manipulate the control plane commands for isolation and portability, and they do not have the visibility or control of the data traffic. Because of this, they cannot flexibly support data plane policies needed by cloud providers. Software-emulated RDMA, *e.g.,* SoftRoCE [36], can easily achieve isolation, portability, and controllability by running RDMA on top of the UDP networking stack and use existing virtual IP networking solutions, but its performance will be limited by UDP.

In this paper, we present *FreeFlow*, a software-based virtual RDMA networking framework for containerized clouds, which simultaneously achieves isolation, portability and controllability and offers performance close to bare-metal RDMA. At the heart of *FreeFlow* is a software virtual switch running on each server to virtualize RDMA on commodity RDMA NICs. *FreeFlow* does not require any specialized hardware or hardware-based I/O virtualization. The software virtual switch has the full access to both control path (*e.g.,* address, routing) and data path (*e.g.,* data traffic) of the communications among containers. This design philosophy is similar to existing software virtual switches used for TCP/IP networking in the containerized cloud, *e.g.,* Open vSwitch (OvS) [15] although *FreeFlow*'s actual design is dramatically different from OvS due to RDMA's characteristics.

The design of *FreeFlow* addresses two key challenges. First, we want *FreeFlow* to be completely transparent to the application. This is challenging because RDMA requires a NIC to manipulate memory buffers and file descriptors, while applications inside containers do not directly interact with the NIC due to network virtualization. Our key insight to address this challenge is that containers are essentially processes, and they can easily share resources like memory and file descriptors with *FreeFlow*. If *FreeFlow* and a container share the same memory (§4.3) and file descriptor (§4.4), any operations on the underlying physical RDMA NIC will automatically take effect inside the container. A further problem is that sharing resources transparently to applications is not straightforward, given that applications do not cooperatively create resources that are shareable. We design methods to convert resource from non-shareable to shareable with no or minimal modifications on application code.

Second, *FreeFlow* must offer throughput and latency that is comparable to bare-metal RDMA. We identify the performance bottlenecks in throughput and latency as memory copy and inter-process communication respectively. We leverage a zero-copy design for throughput (§4.3), and a shared memory inter-process channel with CPU spinning for latency (§5.2). We also optimize *FreeFlow* for bounding CPU overhead.

We evaluate the performance of *FreeFlow* with standard microbenchmarking tools and real-world data-intensive applications, Spark and TensorFlow without any or with minimal modification on them. *FreeFlow* achieves the performance comparable to bare-metal RDMA without much CPU overhead. We also show that *FreeFlow* significantly boosts the performance of real-world applications by up to 14.6 times more in throughput and about 98% lower in latency over using conventional TCP/IP virtual networking. *FreeFlow* has drawn interests from multiple RDMA solution providers, and is open sourced at `https://github.com/Microsoft/Freeflow`.

2 Background

This section provides a brief background on container and RDMA networking, to motivate the need for software-based RDMA virtualization for containers.

Containers and container networking: Containers are becoming the de facto choice [30, 27, 25] to package and deploy data center applications. A container bundles an application's executables and dependencies in an independent namespace using mechanisms such as chroot [4]; thereby offering a lightweight isolation and portability solution.

Most containerized applications use microservices architecture, and are composed of multiple containers. For example, each mapper and reducer node in Spark [2] is an individual container; each parameter server node or worker node in TensorFlow [22] is also an individual container. The containers exchange data via a networking solution. The design of the networking solution affects the degree of isolation and portability.

For instance, in the *host mode* networking, containers use their host's IP and port space, and communicate like an ordinary process in the host OS. This mode has poor isolation

[1]Indeed, our primary motivation to start this work is to enable a large-scale AI application at a leading cloud provider to be migrated from a dedicated cluster to clouds, and yet continue to use RDMA.

[2] There are some recent proposals from industry [35, 26] but these have limitations as we discuss in §9.

#Machines / #GPUs	Transport layer	Normalized speed
1 / 8	-	1.00×
2 / 16	TCP/IP (host)	0.45×
2 / 16	RDMA (host)	1.38×

Table 2: Speeds of a RNN job over TensorFlow on a single machine and multiple machines with TCP and RDMA networking. Speeds are normalized to the single machine case.

(*e.g.,* port conflicts) and portability (*e.g.,* must change IP addresses and ports after migrating to another host).

Thus, many applications use *virtual mode* networking. In this mode, the network namespaces of containers are fully isolated, and containers communicate via a virtual (overlay) network composed of software virtual switches on host machines. The virtual IPs of the containers are highly portable, given that the routes to the virtual IPs can be controlled in the software virtual switches. Since all data traffic must go through the virtual switches, they have access to the traffic, which provides the full controllability to the container networks. Such isolation and portability give orchestrators full flexibility in container placement and migrations, and such controllability offers cloud providers the power to enforce their policies on both control and data plane.

Indeed, orchestrators like Kubernetes [11] mandate the use of virtual networking mode [12]. A number of software solutions are available to provide virtual networking fabrics for containers, such as Weave [23], and Docker Overlay [7].

RDMA networking: Many modern applications (*e.g.,* deep learning and data analytics frameworks) have adopted RDMA networking [18, 17, 22, 5] to get higher throughput, and lower latency than the traditional TCP/IP stack. RDMA offers these gains by offloading most of the networking functionality to the NIC, effectively bypassing the OS kernel.

Table 2 shows measured performance improvements of using RDMA for a deep learning application – training a Recurrent Neural Network (RNN) speech recognition model. The application was first benchmarked on a single machine with 8 GPUs. When the application run on two machines with 16 GPUs, traditional TCP/IP networking becomes a bottleneck, and the performance degrades. With RDMA, however, the extra GPUs offer performance gains.

The reason is that this RNN training task consists of thousands of steps. In each step, all GPUs must shuffle the training model parameters, and the total traffic volume ranges from 100 MB to 10 GB. The time spent on communication is essentially wasting GPU's time, since GPUs are idle during shuffling. TCP performs badly in these frequent and bursty workloads, while RDMA can instantaneously climb to full bandwidth at the beginning of each shuffle.

Need for software-based RDMA virtualization: We have noted the benefits of *virtual mode networking* for containerized applications – namely, enhanced isolation, portability, and controllability. We have also noted that RDMA can offer significant performance boost to many applications that have a microservice architecture.

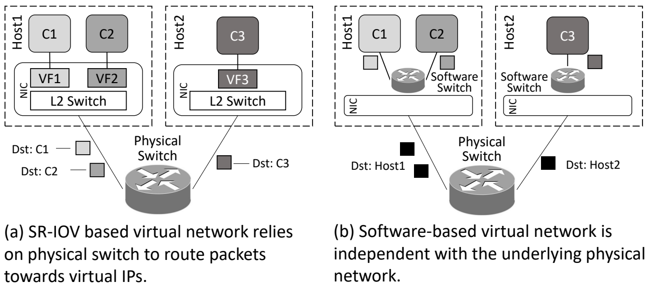

Host1
C1
C2
VF1
VF2
L2 Switch
NIC
Host2
C3
VF3
L2 Switch
NIC
Physical Switch
Dst: C1
Dst: C2
Dst: C3
(a) SR-IOV based virtual network relies on physical switch to route packets towards virtual IPs.
Host1
C1
C2
Software Switch
NIC
Host2
C3
Software Switch
NIC
Physical Switch
Dst: Host1
Dst: Host2
(b) Software-based virtual network is independent with the underlying physical network.

Figure 1: Comparison between hardware-based (SR-IOV) and software-based virtual networking solutions.

The question then is, how do we use RDMA networking with containerized applications that require virtual mode networking, especially in a cloud environment.

RDMA networking, as we saw earlier, relies on offloading most of the networking functionality to a NIC. One possible approach to "virtualize" RDMA networking is to use hardware-based solutions such as SR-IOV [21]. However, this would limit the portability offered by the virtual mode networking. As an example shown in Figure 1(a), with SR-IOV, the NIC runs a simple layer-2 switch that merely performs VLAN forwarding. Hence, all packets generated from and destined to a virtual network have to be directly routed in the underlying physical network. Thus, migrating container C1 to Host2 requires reconfiguring the physical switch to route C1's packets to Host2 rather than Host1. Also, in production, physical switches need to maintain a huge size of routing table to manage routes for all containers in virtual networks, which can be infeasible in a large-scale cloud environment.

Thus, we believe that the right approach to virtualizing RDMA network for containers is to use a software switch – just like it is done for virtualizing traditional TCP/IP networking. As shown in Figure 1(b), the physical network is only in charge of delivering packets targeting on different hosts, and virtual networking routing is completely realized in software switches inside each host, which is independent with the physical network. The software switch can control all addressing and routing, thereby providing good isolation and portability for control plane. It can also be used to implement network functions on data plane such as QoS and metering.

3 Overview

The goal of *FreeFlow* is to provide an virtual interface inside each container, and applications can use RDMA via a virtual network on top of the virtual interface in an unmodified way. Ideally, the performance of the virtual network should be close to bare-metal RDMA, and policies on both control and data path are flexible to be configured purely in software. In this section, we present the system architecture and key challenges in the design of *FreeFlow*.

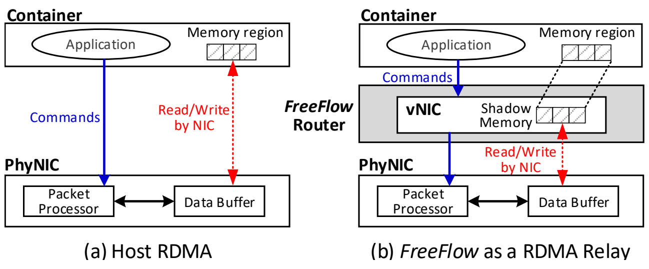


Figure 2: Design overview: *FreeFlow* router directly accesses NIC(s) and serves as a RDMA relay for containers. Blue and red lines are control and data path, respectively.

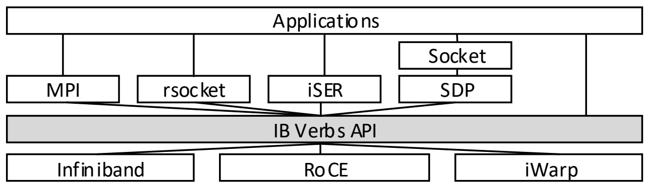


Figure 3: IB Verbs is the de facto "narrow waist" of various RDMA-based network offloading solutions.

3.1 Overall Design

In native RDMA, as shown in Figure 2(a), applications leverage RDMA APIs to directly send commands to the hardware NICs for both control and data path functions. *FreeFlow* intercepts the communication between applications and physical NICs, and performs control plane and data plane policies inside the software *FreeFlow* router which runs as another container on the host machine. In particular, for controlling the data path, *FreeFlow* router only allows the physical NIC to directly read and write from its own memory (the shadow memory in Figure 2(b)) and take the charge of copying data from and to the applications' memory. Note that the memory inside container and the shadow memory in the *FreeFlow* router can be the same piece of physical memory for zero-copy (§4.3).

3.2 Verbs: the "narrow waist" for RDMA

There are multiple ways to intercept the communications between applications and physical NICs, but we must choose an efficient one. A number of commercial technologies supporting RDMA are available today, including Infiniband [9], RoCE [8] and iWarp [19]. Applications may also use several different high-level APIs to access RDMA features, such as MPI and rsocket [20]. As shown in Figure 3, the de facto "narrow waist" of these APIs is the IB Verbs API (Verbs). Thus, we consciously choose to support Verbs in *FreeFlow* and by doing so we can naturally support all higher-level APIs.

Verbs uses a concept of "queue pairs" (QP) for data transfer. For every connection, each of two endpoints has a send queue (SQ) and a receive queue (RQ), together called QP. The send queue holds information about memory buffers to be sent, while the receive queue holds information about which buffers to receive the incoming data. Each endpoint also has a separate completion queue (CQ) that is used by the NIC to notify the endpoint about completion of send or receive requests. The Verbs library and associated drivers allow applications to read, write and monitor the three queues. Actual transfer of the data, including packetization and error recovery, is handled by the NIC.

To transparently support Verbs, *FreeFlow* creates virtual QPs and CQs in virtual NICs and relates the operations on them with operations on real QPs and CQs in the physical NICs.

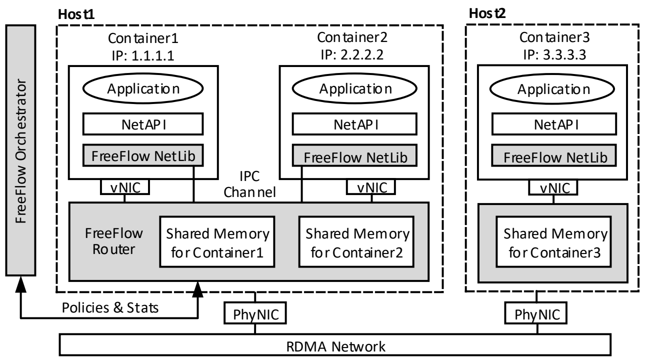


Figure 4: *FreeFlow* architecture.

3.3 *FreeFlow* Architecture

The architecture of *FreeFlow* is shown in Figure 4. The three components of container networking stack that we modify or introduce are shown in gray: (i) the *FreeFlow* network library (*FFL*), (ii) the *FreeFlow* software router (*FFR*), and (iii) the *FreeFlow* network orchestrator (*FFO*).

FFL, located inside the container, is the key to making *FreeFlow* transparent to applications. From application's perspective, it is indistinguishable from the standard RDMA Verbs library [16]. All applications and middleware built atop the Verbs API can run with no (or negligible) modification. *FFL* coordinates with *FFR*.

FFR runs a single instance on each host and works with all containers on the same host to provide virtual networking. In the data plane, *FFR* shares memory buffers with containers on the same host and isolates the shared memory buffers for different containers. *FFR* sends and receives data in the shared memory through the NIC, relying on *FFL* to sync data between application's private data buffers and the shared memory buffers. *FFR* implements the data-plane resource policies, *e.g.,* QoS, by controlling the shared-memory channel between containers and *FFR*. It also works with *FFO* to handle bookkeeping tasks such as IP address assignment.

FFO makes control-plane decisions for all containers in its cluster based on user-defined configurations and real-time monitoring of the cluster. It also maintains centralized memory maps, as we shall discuss in §4.3.

3.4 Challenges

In designing *FreeFlow*, we need to address two key challenges. First, *FreeFlow* should provide an RDMA interface which *transparently* supports all types of existing RDMA operations. There are various types of RDMA operations including one- and two-sided operations for data transfer, poll- and event-based mechanisms for work completion notification, and TCP/IP and RDMA-CM for the connection establishment. We observe that it is not straightforward to support them transparently due to the complexity of RDMA operations. Second, *FreeFlow* should provide near bare-metal RDMA performance while minimizing CPU and memory overhead. Since *FFR* intercepts the Verbs calls from applications via *FFL*, we need to carefully design the communication channel between *FFR* and *FFL*.

We will present our approach for each challenge in §4 and §5, respectively.

4 Transparent Support for RDMA Operations

Verbs supports multiple types of operations and mechanisms. With one-sided operations such as WRITE and READ, a writer (reader) can write (read) data to (from) a specific memory address in the remote side, without the latter aware of this operation. With two-sided operations such as SEND and RECV, the receiver must first get ready to receive before a sender sends out the data. Also, applications can use either poll-based or event-based mechanisms to get work completion notifications. Different applications use different operation types as their needs, and we see all of them used in popular applications [32, 18, 17, 22].

FreeFlow completely and transparently supports such different types of RDMA operations. The primary challenge is to support one-sided operations and event-based completion notifications, in which RDMA NIC can modify memory or file descriptors in *FFR* silently. *FFR* cannot know about the modifications immediately unless it keeps busily polling the status of the memory or file descriptor, so that it is hard to convert the operations from physical NICs to virtual NICs inside containers as soon as possible. We solve this challenge taking advantage of the fact that containers are essentially processes, so that *FFL* and *FFR* can share memory and file descriptors, and physical NIC's modifications can automatically be passed into containers. Sharing memory between *FFL* and *FFR* is also not straightforward for application transparency, because applications inside containers do not allocate memory in IPC shared memory space, and we need to convert the memory to shared memory transparently.

4.1 Connection Establishment

Two RDMA communication endpoints need to first establish a connection. They create a QP in each one's NIC, registering a buffer of memory to the QP and pairing local QP with remote QP. After a connection is established, the application can ask the NIC to send the content in the registered memory to the remote end or put received data into the local buffer.

Steps 1–7 in Figure 5 show the typical process of connection establishment using Verbs. The left column shows the sequence of Verbs calls made by the application. The two

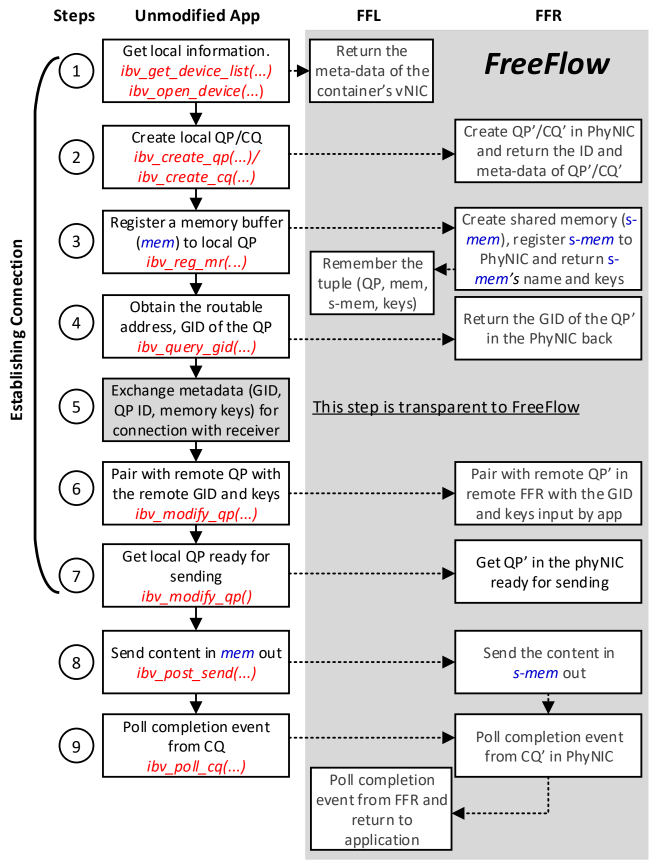

Steps
Unmodified App
FFL
FFR
FreeFlow
Establishing Connection
1
Get local information. ibv_get_device_list(...) ibv_open_device(...)
Return the meta-data of the container's vNIC
2
Create local QP/CQ ibv_create_qp(...)/ ibv_create_cq(...)
Create QP'/CQ' in PhyNIC and return the ID and meta-data of QP'/CQ'
3
Register a memory buffer (mem) to local QP ibv_reg_mr(...)
Create shared memory (s-mem), register s-mem to PhyNIC and return s-mem's name and keys
Remember the tuple (QP, mem, s-mem, keys)
4
Obtain the routable address, GID of the QP ibv_query_gid(...)
Return the GID of the QP' in the PhyNIC back
5
Exchange metadata (GID, QP ID, memory keys) for connection with receiver
This step is transparent to FreeFlow
6
Pair with remote QP with the remote GID and keys ibv_modify_qp(...)
Pair with remote QP' in remote FFR with the GID and keys input by app
7
Get local QP ready for sending ibv_modify_qp()
Get QP' in the phyNIC ready for sending
8
Send content in mem out ibv_post_send(...)
Send the content in s-mem out
9
Poll completion event from CQ ibv_poll_cq(...)
Poll completion event from CQ' in PhyNIC
Poll completion event from FFR and return to application

Figure 5: The workflow of a RDMA SEND operation.

columns in the blue/shaded area shows how *FreeFlow* traps the Verbs calls from the application, and to establish a connection between the sender's *FFR* and the receiver's *FFR*.

Step 1: The application queries for the list of NICs whose drivers support Verbs. *FFL* intercepts the call and returns the context data object of the virtual NIC of the container.

Step 2: The application creates a QP and a CQ on its virtual NIC, while *FFR* creates the corresponding queues (QP$'$ and CQ$'$) on the physical NIC. The QP-IDs and other metadata information of the queues will be forwarded to the application by *FFL* after *FFR* finishes the creations of the queues.

Step 3: The application registers a block of memory (*mem*) to the QP. *FFR* allocates a corresponding block memory (*s-mem*) in its shared memory inter-process communication (IPC) space with the same size as *mem*, registers *s-mem* to QP$'$. *FFR* returns the ID (a host-wide unique name of the IPC memory) it used to create *s-mem*. With this ID, *FFL* can map *s-mem* into its own virtual memory space.

Step 4: The application queries the address (so-called GID in RDMA) of the local QP. This address information will be shared with the other side for pairing the local QP and remote QP together. At the end of this step, *FFR* returns the *actual* GID of QP$'$.

Step 5: The application exchanges GID and QP-ID with the remote end. Applications can exchange this information via any channels such as TCP/IP or RDMA-CM.[3]

Step 6: The application pairs its local QP with the remote container's QP using the receiver's GID. *FFL* forwards this GID to *FFR*. *FFR* pairs QP′ with this GID.

Step 7: The application modifies the state of local QP to Ready to Send/Receive state, while *FFR* modifies the state of QP′ accordingly.

After Step 7, from the application's point of view, it is ready to send or receive data – it has created a QP and a CQ, registered *mem* to the QP, paired with the remote QP and established a connection with the remote QP.

From *FreeFlow*'s point of view, it has created QP′ and CQ′ which are associated with the QP and CQ in the application, registered *s-mem* as the shadow memory of *mem*, and paired with the QP′ in the remote *FFR*. It is also ready to get and forward Verbs calls from the application.

FreeFlow may increase the latency for connection establishment due to the additional interactions between *FFR* and *FFL*. However, it does not much affect the overall latency of *FreeFlow* since it is a one-time cost; many RDMA applications re-use pre-established connections for communications.

4.2 Two-sided Operations

Each sender or receiver needs to go through two steps to perform a data transfer. The first step is to use QP to start sending or receiving data, and the second step is to use CQ to get completion notifications. Steps 8–9 in Figure 5 shows this process.

Step 8: The application invokes the SEND call, and supplies pointer to *mem*. *FFL* first copies data from *mem* to *s-mem*, and *FFR* then invokes its own SEND call to send *s-mem* to the remote *FFR*. We avoid the memory copies from *mem* and *s-mem* by applying our zero-copying mechanism described in §4.3. Note that the remote router would have posted a corresponding RECV call by this time.

Step 9: The application either polls the CQ or waits for a notification that indicates the completion of the send. *FFR* also polls/waits-on CQ′ associated with QP′ and forwards it to *FFL*.

For subsequent SEND operations on the same QP, the application only needs to invoke Step 8 and 9 repeatedly. The workflow of a RECV operation is similar, except that at Step 9, *FFL* will copy data from *s-mem* to *mem* after the QP′ finishes receiving data, which is the opposite of Step 8 in SEND operation.

The presence of *FFL* and *FFR* is completely transparent to the application. To the application, it appears that it is performing normal verbs operations on its vNIC. The steps in Figure 5 are standard way of writing Verbs programs. The *FreeFlow* behavior illustrated here is sufficient to fully support SEND and RECV operations.

[3]*FreeFlow* also has an extension to support RDMA-CM with similar a design to support IB Verbs, while we omit the details due to space limit.

4.3 One-sided Operations

In one-sided operations, a client needs not only the GID of a server, but also the address of the remote memory buffer, and the security key for accessing the memory. This information is exchanged in Step 5 in Figure 5 and becomes available to *FreeFlow* in Step 8 (where WRITE or READ can be called).

Compared to two-sided operations, it is more challenging to transparently support one-sided operations. There are two problems to support one-sided operations in *FreeFlow*.

First, the target memory address *mem* is in the virtual memory of the remote container. However, the local *FFR* does not know the corresponding *s-mem* on the other side. For example, in Figure 6(a), when the sender tries to write data in *mem-1* to remote memory *mem-2*, it fails at stage 3) because the target memory address *mem-2* is not accessible for *FFR* on the receiver side.

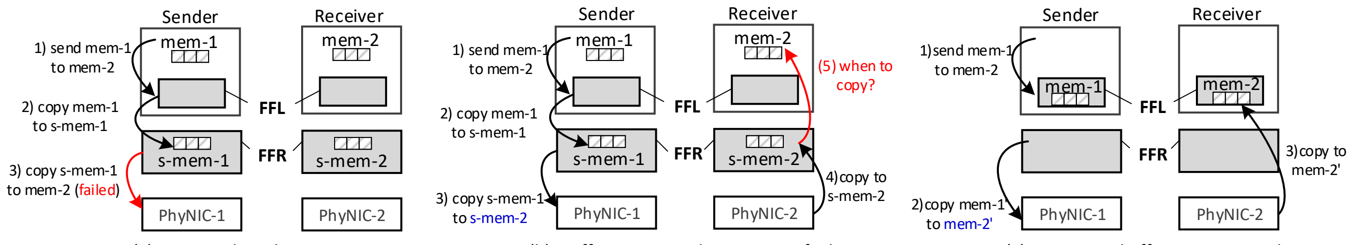
To solve this problem, *FreeFlow* builds a central key-value store in *FFO* for all *FFR*s to learn the mapping between *mem*'s pointer in application's virtual memory space and the corresponding *s-mem*'s pointer in *FFR*'s virtual memory space. Updating this table adds latency to Step 3 in Figure 5, when applications register memory to their virtual NIC. However, data plane performance is not impacted because *FFR* can cache the mappings locally.

Second, even if we know the memory mapping on the remote side, WRITE and READ can remotely modify or copy data without notifying the remote side's CPU, so that *FFR* does not know when to copy to or from application's memory. For instance, in Figure 6(b), the sender finds the correct address of *s-mem-2* and send the data to it. However, after the data is available in *s-mem-2*, there is no notification for the *FFR* in the receiver side to know when to copy *s-mem-2* to *mem-2*. One way to solve this is to continuously synchronize *s-mem-2* and *mem-2*. This would consume a lot of CPU and memory bus bandwidth.

To address this, in *FreeFlow*, we design a *zero-copy based mechanism* to efficiently support one-side operations. The high-level idea is to make *mem* and *s-mem* the same physical memory, so that *FFR* does not need to do any copy, and the data will be naturally presented to the application. Figure 6(c) illustrates this design. By getting rid of memory copies, we can also improve *FreeFlow* performance.

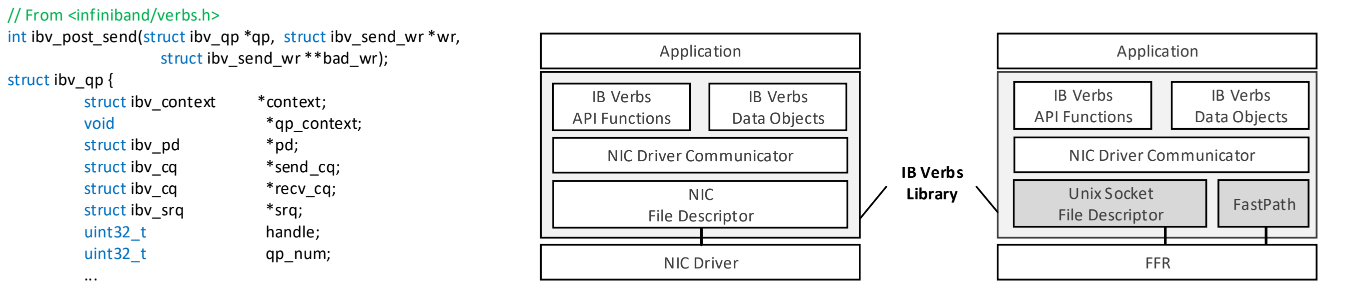
The key here is to make applications directly allocate and use shared memory with *FFR* for data transfers. For this, *FreeFlow* provides two options:

Option 1—Allocating shared buffers with new APIs: We create two new Verbs functions, `ibv_malloc` and `ibv_free`, to let applications delegate the memory creation and deletion to *FreeFlow*. This allows *FFL* to directly allocate these buffers in the shared memory region (shared with *FFR*), and thus avoid the copy. The drawback of this option is the need



(a) Write Failure due to incorrect destination memory address
(b) Inefficient Write due to unnotified memory update
(c) Correct and efficient Write with Zero-Copy design

Figure 6: Zero-copy design enables *FreeFlow* to address the challenges to support one-sided operations efficiently.



```
// From <infiniband/verbs.h>
int ibv_post_send(struct ibv_qp *qp,  struct ibv_send_wr *wr,
                  struct ibv_send_wr **bad_wr);
struct ibv_qp {
        struct ibv_context     *context;
        void                    *qp_context;
        struct ibv_pd           *pd;
        struct ibv_cq           *send_cq;
        struct ibv_cq           *recv_cq;
        struct ibv_srq          *srq;
        uint32_t                handle;
        uint32_t                qp_num;
        ...
};
```

(a) A typical function and data structure definition in IB Verbs
(b) The structure of Verbs Library
(c) FreeFlow "hijacks" the communication between Verbs and NIC driver

Figure 7: The structure of Verbs API and library. *FFR* intercepts the calls between Verbs library and NIC drivers.

to modify application code, despite the modification should be only several lines on the data buffer creation.

Option 2—Re-mapping applications' virtual memory address to shared memory: When an application registers a private memory piece with virtual memory address *va* as a data buffer (*e.g.,* Step 3 in Figure 5), *FFL* releases the physical memory piece behind *va* and assign a shared physical memory piece from *FFR* to *va*. In Linux, this operation is only valid when *va* is an address at the start of a memory page. To force the application to allocate memory always at the start of a page, *FFL* intercepts the calls like `malloc` in C language and makes it always return page aligned memory addresses. While this option can achieve zero memory copy without modifying application code, it forces all memory allocations in the application to be page aligned, which can result in lower memory efficiency on the host.

In practice, we recommend the first option since it is cleaner and efficient. However, since many RDMA applications already make their data buffer page aligned for better performance (*e.g.,* RDMA-Spark [18]), we can directly use the Option-2 without intercepting `malloc`, so the side-effect is limited. Note that if a developer chooses to modify an application using the option 1 or an application originally supports page-aligned buffers, in either case, *FreeFlow* will not incur any overhead in actual memory usage.

4.4 Event-based Operations

There are two options to get notified from CQs (Completion Queue). The first option is to let application poll the CQs periodically to check whether there are any completed operations. The second option is event-based, which means the application creates an event channel and add CQs into the channel. The channel contains a file descriptor which can trigger events when operations are completed.

In *FreeFlow*, since the raw file descriptor is created from physical NIC, *FFR* needs to pass the file descriptor to *FFL*, otherwise the latter cannot detect any events associated with the file descriptor. We take advantage of the fact that *FFL* and *FFR* are essentially two processes sharing the same OS kernel, and leverage the same methodology to pass file descriptors between processes [41] to pass event channels from *FFR* to *FFL*.

5 Communication Channel between *FFL* and *FFR*

Since *FreeFlow* intercepts every Verbs calls via *FFL*, translates, and forwards them to physical NICs via *FFR*, it is crucial to have an efficient channel between *FFL* and *FFR* that provides high RDMA performance while minimizing system resource consumption. In this section, we present two designs of such communication channels, which allows trade RDMA performance for resource consumption and vice versa depending on the requirements of applications.

5.1 Verbs Forwarding via File Descriptor

A straightforward way to pass Verbs calls between *FFL* and *FFR* is to use RPC: *FFL* passes API name and parameters to *FFR*, and *FFR* modifies the parameters properly, executes the API and returns the result of the API call back to *FFL*. Nevertheless, this simple RPC approach does not work well in *FreeFlow* because of the complexity of input data structures of the Verbs calls. As shown in Figure 7(a), a typical function call in Verbs, *e.g.,* `ibv_post_send`, has inputs (*qp*, *wr*) and outputs *bad_wr* that are pointers to complex data structures. Since *FFL* and *FFR* are in two different processes, the pointers of *FFL* will be invalid in *FFR*.

One may advocate "deep copy" which traces down the complex input/output data structures and transfer the data objects under all pointers between *FFL* and *FFR*. However, this approach has two severe drawbacks. First, data structures in Verbs are quite deep (*i.e.,* multiple levels of pointers and nesting) and such deep copies can hurt the performance. Second, there are customized data structures that are defined by user code whose deep copy methods cannot be predefined by *FreeFlow*.

To address this issue, we take advantage of the structure of the current Verbs library. As shown in Figure 7(b), the Verbs library consists of three layers. The top layer is the most complicated one and hard to be handled as described above. However, when it comes down to the middle layer that communicates with the NIC file descriptor, Verbs library must prepare a simple enough (no pointers) data structure that the NIC hardware can digest.

Therefore, instead of forwarding the original function calls of Verbs, we forward the requests to be made for the NIC file descriptor. We replace the NIC file descriptor in the container with a Unix socket file descriptor whose the other end is *FFR*, as shown in Figure 7(c). By doing this, *FFR* can learn the command sent by the application and the supplied parameters. *FFR* will map the operations to virtual queues in the container to the same operations to the actual queues in the physical NIC. It then converts the replies from the physical NIC to replies from the virtual NIC for the virtual queues, and returns the new reply to *FFL* via the Unix socket. The NIC driver communication layer in *FFL* will process the reply normally without knowing about the operations behind the Unix socket file descriptor.

While this Unix socket based approach consumes little CPU, it can incur additional latency due to the inherent delay from communicating via the socket. Our measurement shows that the round trip time over Unix socket (and shared-memory with semaphore) can easily be $\geq 5\ \mu$s in a commodity server. Because of this, the Unix socket communication channel in Figure 7(c) can become a performance bottleneck for latency sensitive applications that expects ultra low latency (*e.g.,* $<5\ \mu$s).

For applications requiring low latency communication, we will describe the design of *Fastpath*, which optimizes the communication delay by trading CPU resources, in the next section.

5.2 Fastpath between *FFL* and *FFR*

To accelerate the communication between *FFR* and *FFL*, we design a *Fastpath* in parallel with the Unix socket based channel between them. As shown in Figure 8, *FFL* and *FFR* co-own a dedicated piece of shared memory. With Fastpath, *FFR* spins on a CPU core and keeps checking whether there is a new request from *FFL* got written into the shared memory piece. Once a request is detected, *FFR* will immediately executes it, while *FFL* starts to spin on a CPU core to check whether the response is ready. After reading the response, *FFL* will stop the CPU spinning on its side.

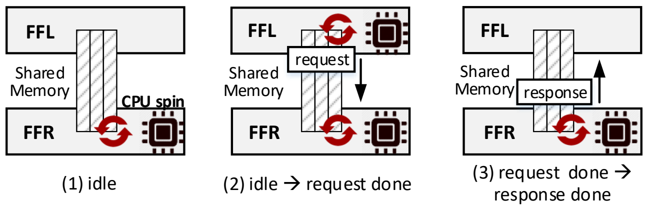


Figure 8: Fastpath channel between *FFR* and *FFL*.

As we will see in § 8.1.2, Fastpath can significantly reduce the latency. However, the price is the CPU cycles spent on spinning for reading requests and responses. To limit the CPU overhead brought by Fastpath, we make two design decisions: (1) *FFR* only spins on one CPU core for all Fastpath channels with *FFL* on the same host; (2) Fastpath is only used for functions which are on data path and are non-blocking, so that the CPU spinning time on *FFL* to wait for a response will be short (few microseconds). Overall, Fastpath only consumes one CPU core per host on average to significantly shorten the latency of message passing (§8.1.2). In addition, if *FFO* knows there is no latency sensitive application on a host machine (according to running container images), it can disable Fastpath and the CPU spinning.

6 Implementation

We implement *FFL* by modifying `libibverbs` (v1.2.1), `libmlx4` (v1.2.1) and `librdmacm` (v1.1.0).[4] We add about 4000 lines of C code to implement *FreeFlow*'s logic. We have implemented *FFR* from scratch in about 2000 lines of C++ code. For *FFO*, we use ZooKeeper to store the user defined information; *e.g.,* IP assignment, access control, resource sharing policies, and memory mapping information for one-sided operations. Due to space limits, we only show three representative implementation details next.

Control & data plane policies: Since *FreeFlow* can control both control and data plane operations requested by containers, it can support common control and data plane policies including bandwidth enforcement, flow prioritization, and resource usage enforcement.

As an example of control plane policy, in our prototype, *FreeFlow* enforces a quota for the number of QPs each container can create, since large number of QPs is a major reason of the performance degradation of RDMA NICs [32]. This control plane policy prevents a container from creating too many QPs which can impact other containers on the same host machine.

Also, as an example of data plane policy, *FreeFlow* enables per-flow rate limiting with little overhead. We imple-

[4] `libibverbs` and `librdmacm` are libraries that allow userspace processes to use InfiniBand/RDMA Verbs and RDMA communication manager interfaces, respectively. `libmlx4` is a userspace driver for `libibverbs` that allows userspace processes to use Mellanox hardware.

ment a simple token-bucket data structure in *FFR*. When an application creates a new QP, we check the policies that are stored in *FFO*, and associate a token-bucket with pre-set rate limit to the QP. Upon every application's send request, the router checks whether the QP has enough tokens to send out the requested message size. If so, the send request is forwarded to the real NIC immediately. Otherwise, *FFR* will notify *FFL* and delay it until there are enough tokens. Note that it is only an example of implementing QoS policies. *FreeFlow* provides flexible APIs for implementing sophisticated QoS algorithms in *FFR*, while we omit the details due to space limit.

Memory management in Fastpath: In Fastpath implementation, we use assembly codes to explicitly force the cache lines of requests and responses written by *FFL* and *FFR* to be flushed into main memory immediately. This is necessary because otherwise, the CPU will keep the newly written lines in cache for a while to wait more written lines, slowing down the message exchanging speed on Fastpath.

Supporting parallelism: Since applications can create multiple QPs and use multiple threads to transfer data in parallel, each Unix domain socket between the *FFL* and *FFR* needs a lock. To improve performance, we create multiple Unix domain sockets between the *FFL* and *FFR*. We avoid "head of the line blocking" by dedicating more of these sockets to data plane operations and event notifications and only a few of sockets to creation, setups and delete operations. On *FFR*, we use a dedicated thread for each incoming Unix domain socket connection. We also create a dedicated data structures for each container and a dedicated shared memory region for each registered memory buffer to keep the data path lock free.

7 Discussion

In this section, we discuss about some primary concerns and potential extensions in the current design of *FreeFlow*.

CPU overhead: Similar to software-based TCP/IP virtual networking solutions, *FreeFlow* incurs CPU overhead. In particular, *FreeFlow* uses a CPU core for polling control messages between *FFL* and *FFR* to support low latency IPC channel (§5.2). We admit that this is a cost for network virtualization on top of current commodity hardwares. One possible approach to address this is to utilize hardwares that support offloading CPU tasks, such as FPGA, ARM coprocessor, or RDMA NICs [1]. We leave it as a future work to eliminate the CPU overhead in Fastpath.

Security: One concern is that since *FFR* shares its memory with containers, whether one container can read the communications of other containers on the same host by scanning the IPC space. This is not a concern for *FreeFlow* because *FFR* creates a dedicated shared memory buffer for each individual QP. Only those shared memory buffers that belong to a container will be mapped into the container's virtual memory space. Another concern is the security of the memory keys. If one can see the keys by wiretapping, subsequent communications can be compromised. This problem is inherent in the way one-sided operations in raw RDMA work, and is not made worse by *FreeFlow*.

Working with external legacy peers: Containers in *FreeFlow* can naturally communicate with external RDMA peers, since each *FFR* works independently. *FFR* does not distinguish whether the remote peer is another *FFR* or an external RDMA peer.

Container migration: *FreeFlow* supports offline migrations naturally. If a container is captured, shutdown, moved and rebooted in another host machine, its IP address is not changed, so that its peers re-establish RDMA connections with it as if it is just got rebooted. Nowadays, offline migrations are commonly used in container clusters for resource packing or fail-over. *FreeFlow* does not support live migration, since RDMA has poor mobility nowadays [39].

VM host: Our prototype (and evaluation) is based on containers running on bare-metal host machines. But *FreeFlow* can be directly used on containers deployed inside VMs if the VMs use SR-IOV to access the physical NIC.

Congestion control: RDMA NICs already have congestion control mechanisms, and *FreeFlow* relies on them.

8 Evaluation

We evaluate the performance and overhead of *FreeFlow*. We start from microbenchmarks (§8.1) and then the performance of real-world applications on *FreeFlow* (§8.2).

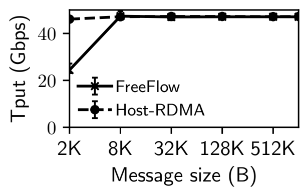
8.1 Microbenchmarks

Setup: We run microbenchmarks on two testbeds. One testbed runs InfiniBand, which is a traditional RDMA-dedicated fabric. The servers are equipped with two Intel Xeon E5-2620 2.10GHz 8-core CPU, 64GB RAM, and 56Gbps Mellanox FDR CX3 NIC. The OS is Ubuntu 14.04 with the kernel version 3.13.0-129-generic.

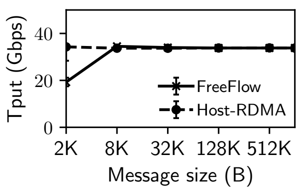
The other testbed runs RoCE (RDMA over Converged Ethernet). As the name indicates, RoCE only requires conventional Ethernet switches (in our case, Arista 7050QX as the ToR switch). The servers in this testbed cluster have Intel Xeon E5-2609 2.40GHz 4-core CPU, 64GB RAM, 40Gbps Mellanox CX3 NIC and Ubuntu 14.04 with the kernel version 4.4.0-31-generic.

We run containers using Docker (v1.13.0) [7] and set up a basic TCP/IP virtual network using Weave (v1.8.0) [23] with Open vSwitch kernel module enabled. Unless otherwise specified, we run Fastpath (§5.2) enabled *FreeFlow*.

We mainly compare *FreeFlow* with bare-metal RDMA, which is a stand-in for the "optimal" performance. We will show that *FreeFlow* enables virtual RDMA networking for containers with minimal performance penalty. In §8.1.4, we will also demonstrate the performance of translating TCP socket calls into RDMA on top of *FreeFlow*, so that conventional TCP applications can also benefit from *FreeFlow*. There we also compare *FreeFlow* with bare-metal TCP and Weave which supports virtual TCP/IP virtual networks for containers.

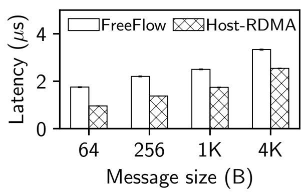


(a) 56Gbps InfiniBand

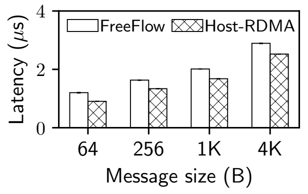


(b) 40Gbps RoCE

Figure 9: RDMA SEND throughput between a pair of containers on different hosts. *FreeFlow* enables container virtual networks with minimal performance penalty.



(a) SEND on InfiniBand



(b) WRITE on InfiniBand

Figure 10: RDMA latency between a pair of containers on different hosts. SEND is a typical two-sided operation, while WRITE is one-sided.

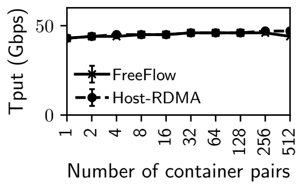


Figure 11: Aggregate throughput when scaling up the number of container pairs.

Host RDMA	Fastpath	LowCPU
1.8μs	2.4μs	17.0μs

Table 3: 2-byte message latency of two *FreeFlow* modes.

8.1.1 Throughput and Latency

We focus on two basic performance metrics, throughput and latency. We use the benchmark tools provided by Mellanox perftest [13]: `ib_send_lat` and `ib_send_bw` to measure latency and throughput of two-sided operation (SEND), `ib_write_lat` and `ib_write_bw` for one-sided operation (WRITE). These tools can run on *FreeFlow without any modification*, as explained in §4.3. In general, *FreeFlow* does not differentiate the inter-host setting (sender and receiver run on different hosts) and the intra-host setting. Here we just show inter-host performance values.

Throughput: We measure the single thread RDMA SEND/WRITE throughput on two testbeds, and show the RDMA SEND results in Figure 9. Each run transmits 1GB data with different sizes of messages ranging from 2KB to 1MB. *FreeFlow* RDMA WRITE results are in fact slightly better than SEND, and omitted for brevity. We see that with message size equal or larger than 8KB, *FreeFlow* gets full throughput as bare-metal RDMA (46.9Gbps on InfiniBand and 34.5Gbps on RoCE). In addition, when we increase the number of concurrent container pairs (flows) to up to 512, the aggregated throughput of all flows is still close to optimal (Figure 11). We also verify that the bandwidth is fairly distributed among different flows by calculating Jain's fairness index [31] (0.97 on average).

In general, the bandwidth-hungry applications tend to use larger message sizes than a few KB. For example, in one of our internal storage clusters that uses RDMA, typical message sizes are 1MB or more. *FreeFlow* will have no throughput penalty in this case (see §8.1.2 for CPU overhead).

Even when the message sizes are small, like 2KB, *FreeFlow* still achieves more than half of the full throughput. We verified that, in this case, the throughput is bounded by the single *FFR* Fastpath thread (§5.2). This bottleneck can be easily removed by assigning one more CPU core to the *FFR* and balancing RDMA request loads across the two cores. While we leave this option open, developers usually do not expect to saturate the full bandwidth with small messages. Instead, for small messages, developers usually care about latencies.

Latency: We measure the latency of sending a 64B, 256B, 1KB, and 4KB message, respectively. Like the throughput benchmark, the two containers run on different hosts connected via the same ToR switch. For each message size, we measure the latency 1000 times. We plot the median, 10- and 99th-percentile latency values.

Figure 10 shows the one-way latency reported by the perftest tools. We can see that one-sided WRITE operation have lower latency than two-sided SEND operation, and also smaller gap between *FreeFlow* and bare-metal RDMA. However, even with the two-sided operation, *FreeFlow* causes less than 1.5 μs extra delay. The extra delay is mainly due to the IPC between the *FFL* and *FFR*. One-sided operation will trigger IPC only one time, while two-sided operations will trigger two times and one time memory copy. This explains the larger latency gap of two-sided operations.

To put these latency values into perspective, one hop in network, *i.e.*, a hardware switch, has 0.55μs latency [3]. Thus, *FreeFlow* latency overhead is comparable to an extra switch hop in the network. In comparison, host TCP stack latency is at least 10μs (§8.1.4) and then TCP/IP virtual network latency is even larger (more than 40μs in our test). This means *FreeFlow* preserves the latency advantage of RDMA while enabling virtual network for containers.

8.1.2 CPU Overhead and Trade-off

FreeFlow achieves good performance with low CPU overhead. *FreeFlow* has two modes: Fastpath and non-Fastpath (or LowCPU, in §5.1). By default, Fastpath is enabled and provides the best performance in terms of latency. In this mode, *FFR* spins on one CPU core and serves Verbs requests as soon as possible. One CPU core is capable of serving all the containers on one host, thanks to the fact that *FFR* only handles message-level events, instead of at packet-level like in Open vSwitch. On a commodity server with many CPU cores, this is acceptable.

In addition, users may choose the LowCPU mode, which uses a Unix socket as the signal mechanism instead of core

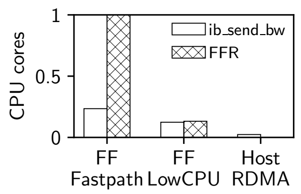


Figure 12: CPU usage of ib_send_bw and *FFR* when measuring the throughput with 1MB messages. 100% CPU means one fully utilized CPU core.

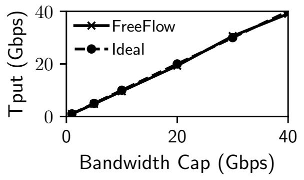
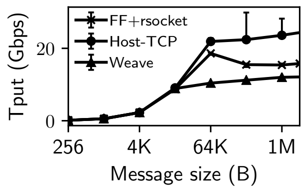
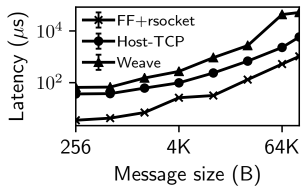


Figure 13: *FreeFlow* can accurately control the rate of traffic flows from containers.



(a) iperf throughput on IB



(b) NPtcp latency on IB

Figure 14: TCP throughput and latency between a pair of containers on different hosts. We compare native TCP with *FreeFlow* + rsocket (socket-to-Verbs translation).

spinning. This hurts latency performance (increase from 2.4μs to 17.0μs), as shown in Table 3. In Figure 12, we record the per-process CPU utilization when measuring inter-host throughput. The throughput of all three cases in the figure are the same (full bandwidth). It shows the CPU benefit of LowCPU mode, especially on the *FFR*. In LowCPU mode, *FFR* CPU overhead scales with the actual load.

We recommend choosing the mode according to the workload requirement. Latency-sensitive or non-CPU heavy (*e.g.,* GPU-heavy) applications should be run with Fastpath mode while the rest can be run with LowCPU mode. However, even with Fastpath, *FFR* consumes at most one CPU core, and the extra overhead due to *FFL* is less than 30% for full bandwidth throughput.

8.1.3 Rate Limiter and Performance Isolation

We demonstrate the performance of rate limiter mentioned in §6. In Figure 13, we start a single flow between two containers on different hosts, on Infiniband testbed. We limit the flow rate and set different bandwidth caps from 1Gbps to 40Gbps. We see that the controlled bandwidth (y-axis) is close to the bandwidth cap we set (x-axis). *FreeFlow* achieves this with only 6% CPU overhead.

FreeFlow can isolate performance (i.e., throughput) for different containers using the rate limiter. To demonstrate this, we ran 10 concurrent flows between container pairs and applied the different rate limits to each flows (from 1 to 10Gbps). We verified that the throughput of each flow is accurately capped.

8.1.4 TCP Socket over RDMA

Enabling virtual RDMA can also benefit the performance of socket-based applications. Below we show that *FreeFlow* provides better performance than conventional TCP/IP virtual networks with the help of `rsocket`, an existing socket-to-Verbs translation layer.

We run the experiments on both InfiniBand and RoCE clusters. By dynamically linking with `rsocket` during runtime,[5] application socket calls are transparently translated into RDMA Verbs calls. We run `iperf` [10] for measuring TCP throughput, and `NPtcp` [14] for TCP latency *without any modifications on these tools*. We compare against the same tools running on the virtual and host mode network.

As Figure 14 shows, *FreeFlow* always outperforms Weave. Especially for small message latency, *FreeFlow* is consistently lower than even host TCP/IP, by up to 98%. For throughput, *FreeFlow* is sometimes worse than host TCP and cannot achieve full throughput like raw RDMA, due to the overhead of socket-to-Verbs translation. However, it is still 6.8 to 13.4 times larger than Weave with large messages.

The are two reasons for *FreeFlow*'s good performance. First, the RDMA stack and *FreeFlow* architecture works only in the userspace and avoids the context switching in kernel TCP stack. This advantage is not unique; customized userspace network stacks can also achieve this. The second reason *FreeFlow* outperforms Weave is fundamental. The existing TCP/IP virtual networking solutions perform packet-by-packet address translation from virtual network to host network. However, *FreeFlow* performs message-based translation from virtual connection to physical connection. Thus, *FreeFlow* always outperforms Weave, though rsocket introduces some socket-to-Verbs translation overhead.

8.2 Real-world Applications

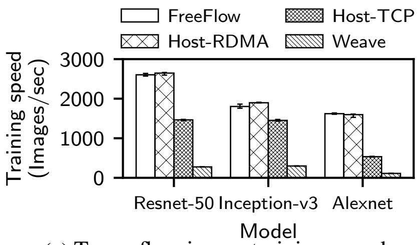
In this section, we show the performance of TensorFlow and Spark, a representative machine learning and data analytics framework, running in containers. We compare the application performance on *FreeFlow* against Host-RDMA, Host-TCP, and Weave.

Since TensorFlow requires GPUs that our RoCE cluster does not have, we run all the experiments on our InfiniBand cluster. Based on the microbenchmarks, we believe RoCE clusters will have similar trends if equipped with GPU.

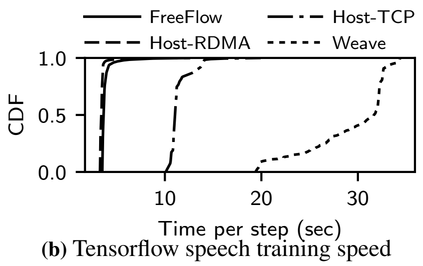
8.2.1 Tensorflow

We run RDMA-enabled Tensorflow (v1.3.0) on three servers in the InfiniBand cluster. We modified a single line of the source code of Tensorflow to replace the original memory allocation function with our custom memory allocator (§4.3). Each server has eight NVIDIA GTX 1080 Ti GPUs. One of the servers is a master node and also a parameter server, while the other two servers are workers. We run two main types of training workloads for deep learning, namely, image recognition based on Convolutional Neural Network (CNN),

[5]This can be easily configured by setting an environment variable called LD_PRELOAD in Linux.



(a) Tensorflow image training speed



(b) Tensorflow speech training speed

Figure 15: TensorFlow performance on *FreeFlow*.

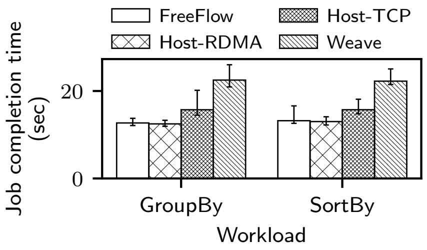


Figure 16: Spark performance on *FreeFlow*.

and speech recognition based on Recurrent Neural Network (RNN).

For image recognition, we run three specific models, ResNet-50 [29], Inception-v3 [42] and AlexNet [33]. We use synthetic ImageNet data as training data. Figure 15(a) shows the median training speed per second with 10-percentile and 99-percentile values. From the results of all three different models, we conclude, first, the network performance is indeed a bottleneck in the distributed training. Comparing host RDMA with host TCP, host RDMA performs 1.8 to 3.0 times better in terms of the training speed. The gap between *FreeFlow* and Weave on container overlay is even wider. For example, *FreeFlow* runs 14.6 times faster on AlexNet. Second, *FreeFlow* performance is very close to host RDMA. The difference is less than 4.9%, and *FreeFlow* is sometimes even faster. We speculate that this is due to measurement noise.

For speech recognition, we run one private speech RNN model consisting of a bi-directional encoder and a fully-connected decoder layers, with a hidden layer dimensionality of 1024 and a vocabulary size of 100k. The dataset is 4GB large including 18.6 millions samples. In each training step, GPUs "learn" from a small piece and communicate with each other for synchronization. Figure 15(b) shows the CDF of the time spent for each training step, including the GPU time and networking time. Again, *FreeFlow* is very close to host RDMA. The median training time is around 8.7 times faster than Weave.

8.2.2 Spark

We run Spark (v2.1.0) on two servers. One of the server runs a master container that schedules jobs on slave containers. Both of the servers run a slave container. The RDMA extension for Spark [18] is implemented by is closed source. We download the binary from their official website and did not make any modification.

We demonstrate the basic benchmarks shipped with the Spark distribution – GroupBy and SortBy. Each benchmark run on 262,144 key-value pairs with 2 KB value size. We set the number of Spark mappers and reducers to 8 and each of them is a single thread. Figure 16 illustrates the result. We conclude similar observations as running TensorFlow. The performance of network does impact the application end-to-end performance significantly. When running with *FreeFlow*, the performance is very close to running on host RDMA, better than host TCP, and up to 1.8 times better than running containers with Weave.

9 Related Work

RDMA virtualization for containers: There is an ongoing effort from Mellanox to extend network namespace and cgroup in Linux kernel to accommodate RDMA for networking isolation [34, 35]. It uses MACVLAN to split a physical interface to multiple virtual interfaces, inserts one or multiple interfaces to each container, and relies on VLAN routing to deliver traffic to the correct virtual interface. Apparently, it has portability issues for cloud environments, since moving an IP means updating VLAN routing in hardware. Also, it does not offer a flexible controllability, because it allows containers to directly access physical NICs.

Another approach is using programmable hardware to handle the RDMA virtualization for containers, such as smart NICs [26] or FPGA [38]. *FreeFlow*'s advantages compared with such hardware-based solutions are its lower cost by using commodity hardware and better flexibility to customize network features.

RDMA virtualization for VM: HyV [39] is the closest solution to *FreeFlow*. It also intercepts the communication between applications and NIC driver and provides address translation, QP/CQ mapping, and memory mapping. The key difference between HyV and *FreeFlow* is that HyV does not control data path to provide bare-metal performance in private clusters, while *FreeFlow* does for fitting in cloud environments. This creates more challenges to *FreeFlow*, such as making the performance still close to bare-metal quality while maintaining transparency to applications in data path. VMM-bypass I/O [37] has a similar design and issues as HyV. VMware has been working on para-virtualizing RDMA devices called vRDMA [40]. vRDMA is designed for VMware's hypervisor and VMs, so it does not inherently work for containers.

10 Conclusion

In this paper, we presented *FreeFlow*, a virtual RDMA networking solution that provides the isolation, portability and controllability needed in containerized clouds. *FreeFlow* is transparent to applications and achieves close-to bare-metal RDMA performance with acceptable overhead. Evaluations with real-world applications and microbenchmarks show that *FreeFlow* can support performance comparable to bare-metal RDMA and much better than the existing TCP/IP virtual networking solution. We open source the prototype of *FreeFlow*.

11 Acknowledgments

We would like to thank the anonymous NSDI reviewers and our shepherd, Hakim Weatherspoon for their helpful comments. This work was funded in part by NSF awards 1700521 and 1513764.

12 Availability

FreeFlow is open sourced at https://github.com/Microsoft/Freeflow.

References

[1] Mellanox coredirect. http://www.mellanox.com/page/products_dyn?product_family=61&mtag=connectx_2_vpi/, 2010.

[2] Apache spark. https://spark.apache.org/, 2018. Accessed on 2018-01-25.

[3] Arista 7050x & 7050x2 switch architecture. https://www.arista.com/assets/data/pdf/Whitepapers/Arista_7050X_Switch_Architecture.pdf, 2018. Accessed on 2018-01-25.

[4] chroot(2) - Linux man page. https://linux.die.net/man/2/chroot, 2018. Accessed on 2018-01-25.

[5] CNTK. https://github.com/Microsoft/CNTK/wiki, 2018. Accessed on 2018-01-25.

[6] CoreOS. https://coreos.com/, 2018. Accessed on 2018-01-25.

[7] Docker. http://www.docker.com/, 2018. Accessed on 2018-01-25.

[8] Infiniband architecture specification release 1.2.1 annex a16: Roce. https://cw.infinibandta.org/document/dl/7148, 2018. Accessed on 2018-01-25.

[9] Introduction to infiniband. https://en.wikipedia.org/wiki/InfiniBand, 2018. Accessed on 2018-01-25.

[10] Iperf - the TCP/UDP bandwidth measurement tool. http://iperf.fr, 2018. Accessed on 2018-01-25.

[11] Kubernetes. http://kubernetes.io/, 2018. Accessed on 2018-01-25.

[12] Kubernetes networking. https://kubernetes.io/docs/concepts/cluster-administration/networking/, 2018. Accessed on 2018-01-25.

[13] Mellanox perftest package. https://community.mellanox.com/docs/DOC-2802, 2018. Accessed on 2018-01-25.

[14] netpipe(1) - linux man page. https://linux.die.net/man/1/netpipe, 2018. Accessed on 2018-01-25.

[15] Open vswitch. http://openvswitch.org/, 2018. Accessed on 2018-01-31.

[16] Openfabrics, libibverbs release. https://www.openfabrics.org/downloads/libibverbs/, 2018. Accessed on 2018-01-25.

[17] Rdma-based apache hadoop. http://hibd.cse.ohio-state.edu/, 2018. Accessed on 2018-01-25.

[18] Rdma-based apache spark. http://hibd.cse.ohio-state.edu/, 2018. Accessed on 2018-01-25.

[19] Rdma-iwarp. http://www.chelsio.com/nic/rdma-iwarp/, 2018. Accessed on 2018-01-25.

[20] rsocket(7) - linux man page. https://linux.die.net/man/7/rsocket, 2018. Accessed on 2018-01-25.

[21] Single root I/O virtualization. http://pcisig.com/specifications/iov/single_root/, 2018. Accessed on 2018-01-25.

[22] Tensorflow. https://www.tensorflow.org/, 2018. Accessed on 2018-01-25.

[23] Weave Net. https://www.weave.works/, 2018. Accessed on 2018-01-25.

[24] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *USENIX OSDI* (2016).

[25] DATADOG. 8 suprising facts about real Docker adoption. https://www.datadoghq.com/docker-adoption/, 2016.

[26] DEIERLING, K. Ensuring both high performance and security for containers. In *Flash Memory Summit* (2017).

[27] DOCKER. Docker community passes two billion pulls. https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/, 2016.

[28] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., ET AL. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI* (2018).

[29] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *IEEE CVPR* (2016).

[30] IRON.IO. Docker in production – what we've learned launching over 300 million containers. https://www.iron.io/docker-in-production-what-weve-learned/, 2014.

[31] JAIN, R., CHIU, D. M., AND HAWE, W. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *DEC Technical Report*.

[32] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review* (2014), vol. 44, ACM, pp. 295–306.

[33] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *NIPS* (2012).

[34] LISS, L. Containing RDMA and high performance computing. In *ContainerCon* (2015).

[35] LISS, L. RDMA container support. In *International OpenFabrics Software Developer's Workshop* (2015).

[36] LISS, L. The Linux SoftRoce Driver. In *OpenFabrics Annual Workshop* (2017).

[37] LIU, J., HUANG, W., ABALI, B., AND PANDA, D. K. High Performance VMM-Bypass I/O in Virtual Machines. In *USENIX ATC* (2006).

[38] MOUZAKITIS, A., PINTO, C., NIKOLAEV, N., RIGO, A., RAHO, D., ARONIS, B., AND MARAZAKIS, M. Lightweight and Generic RDMA Engine Para-Virtualization for the KVM Hypervisor. In *High Performance Computing & Simulation (HPCS), 2017 International Conference on* (2017), IEEE, pp. 737–744.

[39] PFEFFERLE, J., STUEDI, P., TRIVEDI, A., METZLER, B., KOLTSIDAS, I., AND GROSS, T. R. A Hybrid I/O Virtualization Framework for RDMA-capable Network Interfaces. In *ACM VEE* (2015).

[40] RANADIVE, A., AND DAVDA, B. Toward a paravirtual vRDMA device for VMware ESXi guests. *VMware Technical Journal, Winter 2012 1*, 2 (2012).

[41] STEVENS, W. R., AND RAGO, S. A. *Advanced programming in the UNIX environment*. Addison-Wesley, 2013.

[42] SZEGEDY, C., VANHOUCKE, V., IOFFE, S., SHLENS, J., AND WOJNA, Z. Rethinking the inception architecture for computer vision. In *IEEE CVPR* (2016).