FreeFlow: High Performance Container Networking

Tianlong Yu†, Shadi A. Noghabi♭, Shachar Raindel‡
Hongqiang Harry Liu‡, Jitu Padhye‡, Vyas Sekar†
†CMU, ‡Microsoft, ♭UIUC

Abstract— As the popularity of container technology grows, many applications are being developed, deployed and managed as groups of containers that communicate among each other to deliver the desired service to users. However, current container networking solutions have either poor performance or poor portability, which undermines the advantages of containerization. In this paper, we propose FreeFlow, a container networking solution which achieves both high performance and good portability. FreeFlow leverages two insights: first, in most container deployments a central entity (i.e. the orchestrator) exists that is fully aware of the location of each container. Second, strict isolation is unnecessary among containers belonging to the same application. Leveraging these two observations allows FreeFlow to use a variety of technologies such as shared memory and RDMA to improve network performance (higher throughput, lower latency, and less CPU overhead), while maintaining full portability – and do all this in a manner that is completely transparent to application developers.

1 Introduction

The history of all hitherto computer science is (often) the history of a struggle between isolation, portability and performance.
(With apologies to Karl Marx.)

At the dawn of computing, applications had access to (and had to manage) raw hardware. Applications were not portable as they were tailored for specific platforms. Isolation between applications was non-existent as well. Operating systems emerged and offered a modicum of isolation and portability. As users demanded more portability and better isolation across applications, OSes became more sophisticated, and deep layering became the norm; the modern TCP/IP stack is a classic example of layering. Layering improves the application's portability across systems and types of networks, but incurs well-known performance issues [37, 41]. Soon enough, solutions like DPDK [6] and RDMA [33] emerged that traded off some portability and isolation to provide better performance. The trend continued with virtualization, which offers even more isolation, and additional portability (e.g., you can pack up and move VMs at will, even live-migrate them). In return, performance – especially the network performance is further reduced [26]. Numerous technologies have been proposed to remedy the situation (e.g., [6, 22, 26, 37]) – again at the cost of isolation and portability.

The latest step in this trend is *containerization* [1, 2, 5]. By wrapping a process together a complete filesystem and namespace cell, a container has everything needed to run the process, including executables, libraries and system tools. A container has no external dependencies, which makes it highly portable. The namespace of the container is isolated from other containers, eliminating worries about naming and version conflicts. For example, it is possible to run two containers as two web servers without introducing port conflicts (port 80 or 443) on a bare-metal machine. Such portability and independence significantly simplifies the life cycle of a containerized application, from testing to high availability maintenance.

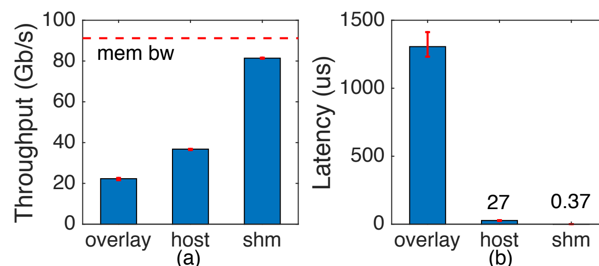

Figure 1: Performance comparison of two modes of container networking and shared memory IPC.

Unfortunately, containers too suffer the veritable curse of having to sacrifice one or more of performance, isolation, and portability. To understand these potential performance bottlenecks, we conducted a simple experiment. We set up two Docker containers on a single server[1]. We consider three ways for the containers to communicate with each other: (1) *Shared Memory:* This requires special setup[2] to bypass the namespace isolation, and offers the least isolation, and the least portability; (2) *Host mode:* in which a container binds an interface and a port on the host and uses the host's IP to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotNets-XV, November 09 - 10, 2016, Atlanta, GA, USA

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4661-0/16/11…$15.00

DOI: http://dx.doi.org/10.1145/3005745.3005756

[1] See Section 5 for HW and SW details

[2] We setup the shared memory data transfer through `shared memory object` in a shared IPC namespace and measure the time to pass the pointer and make one copy of the data.

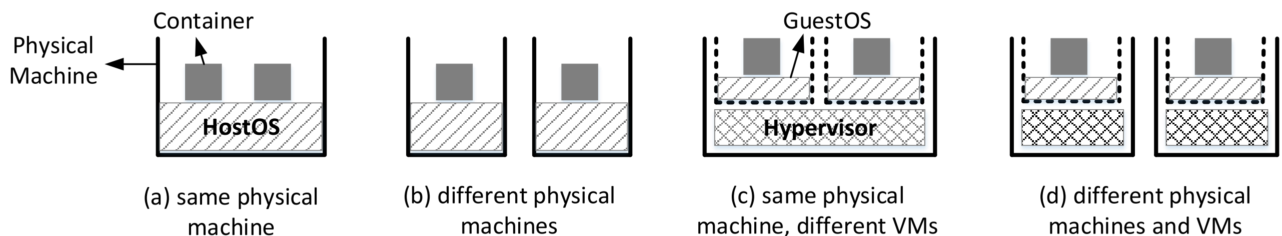

Figure 2: Representative running environments of containers.

communicate, like an ordinary process. Hence, containers are not truly isolated as they must share the port space; and (3) *Overlay mode:* in which the host runs a software router which connects all containers on the host via a bridge network. The software routers enable overlay routing across multiple hosts to provide maximum portability as each container can even have public IPs assigned.

Figure 1 is a telling demonstration of the fundamental tussle between portability, isolation, and performance. We make two observations from this figure. First, the throughput and latency of host and overlay modes of inter-container communication are significantly worse than the throughput and latency of shared memory based communication. The reason is obvious: both host and overlay modes require a "hairpin" path through the full TCP/IP stack.

Second, the performance of overlay networking is worse than host mode. The reason, again, is simple: in case of overlay networking, hairpinning happens twice, since the packets must traverse through the software router as well. This figure thus clearly illustrates the performance cost of isolation and portability.

As the popularity of container networking grows, this inefficiency must be addressed. On one hand, the low throughput and high latency directly impacts the overall performance of large scale distributed systems, such as big data analytics [8, 11, 12, 13], key-value stores [17], machine learning frameworks [28], etc. On the other hand, it forces the applications to reserve substantial CPU resources to merely perform traffic processing, which significantly raises the cost of running the applications.

One may argue that there is nothing new here: virtualization suffers from similar inefficiencies and we know how to address them using techniques like SR-IOV [22] and NetVM [26]. Unfortunately, these ideas cannot be directly applied to the container world. SR-IOV typically scales to tens of VMs per server. In typical deployments, there are hundreds of containers per server. The cost of supporting so many containers in the NIC hardware will be prohibitive. NetVM [26] cannot be applied to containers without destroying their portability, because it requires two VMs to be on the same server.

In this paper we outline a solution, called FreeFlow to address this issue. Our vision is to develop a container networking solution that provides high throughput, low latency and negligible CPU overhead and fully preserves container portability in a manner that is completely transparent to application developers.

To achieve these seemingly conflicting goals, we observe an opportunity to leverage two key aspects of typical container deployments: (1) they are typically managed by a central orchestrator (E.g., Mesos, YARN and Kubernetes [2, 24, 39]) and (2) they are typically deployed over managed network fabrics (e.g., a public cloud provider). Taking advantage of these easily available additional bits of information, we sketch a roadmap of an overlay-based solution that obtains the relevant deployment-specific information from the aforementioned container orchestrator and fabric manager and use this in conjunction with the "right" I/O mechanism (e.g., shared memory when containers are co-located, vs. RDMA when they are not).

While this sounds conceptually simple, there are several architectural and system design challenges in realizing this vision in practice. In the rest of the paper, we discuss these challenges and sketch a preliminary design. We will also present results from an early prototype.

2 Background

Container technology has gain tremendous popularity [15, 16, 27] since it enables a fast and easy way to package, distribute and deploy applications and services.

Containers are essentially processes, but they use mechanisms as `chroot` [19] to provide namespace isolation. The dependencies of containerized applications are bundled together with the application, making them highly portable: they can be easily distributed and deployed [1]. Compared to VMs, containers are much more lightweight and can be initialized much faster [31, 34]. Containers can be deployed both within a VM or on bare metal and are supported by both Linux and Windows operating system. Docker [1] is perhaps the most popular container management system, although there are many others as well [2, 5].

Most containerized applications are usually composed of multiple containers. For example, each mapper and reducer node in Hadoop [4] is an individual container. A modern web service with multiple layers (load balancers, web servers, in-memory caches and backend databases) is deployed with multiple containers in each layer. These containers are usually deployed into a multi-host server cluster, and the deployment is often controlled by a cluster orchestrator, e.g. Mesos [24] or Kubernetes [2]. Such an architecture makes it easier to upgrade the nodes or mitigate failures, since a stopped container can be quickly replaced by a new one on the same or a different host. Working as a single applica-

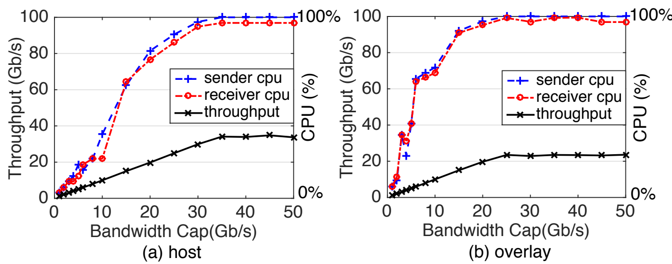

Figure 3: CPU limits overlay networking throughput.

tion, containers need to exchange data, and the network performance has a significant impact on the overall application performance [8, 11, 12, 13].

Depending on whether containers run on bare-metal hosts or VM hosts, there are four cases any container networking solution must handle. These cases are illustrated in Figure 2. For maximum portability, containers today often use overlay networks. A number of software solutions are available to provide overlay fabrics, such as Weave [7] and Calico [3]. In these solutions, the host (i.e., the server or the VM) runs the software router which connects to the NIC of the host and to the virtual interfaces of the containers on the host via a software bridge. The router also performs appropriate tunneling (encapsulation and decapsulation) to move traffic between the physical network and the overlay fabric. The router uses standard networking protocols like BGP to connect with software routers on other hosts. Containers send and receive IP packets via this overlay, and hence are agnostic to locations of other containers they are communicating with.

The CPU overhead of processing packets in the software bridge, as well as in the software router (for off-host communication) is the primary performance bottleneck in overlay networks, as illustrated in Figure 3. This figure shows throughput and CPU utilization of two containers, running iPerf3. For Figure 3(a), the containers were on the same server, communicating via host mode. For Figure 3(b), the containers were on different servers, connected via overlay routing using Weave [7]. We vary the iPerf traffic generation rate, and plot the achieved throughput and corresponding CPU utilization. We see that in host mode, the throughput tops off at 30Gbps, while in overlay mode it is just 20Gbps. In both cases, the CPU is the bottleneck – the sender and the receiver CPUs are fully utilized.

3 Overview

In this section, we discuss the overall architecture of FreeFlow, and discuss the key insights that enable FreeFlow to achieve a high network performance without sacrificing portability of containers.

3.1 High network performance without sacrificing portability

Container deployments opt for overlay-based networking since it is most portable: a container does not have to worry about where the other endpoint is. For example, in Figure 4(a), Container 1 and Container 3 cannot distinguish whether Container 2 is on Host 1 or Host 2, as long as Container 2 keeps its overlay IP address (2.2.2.2) and the overlay routers know how to route packets to this IP.

Existing overlay-based container networks sacrifice performance for good portability, because traffic needs to go through a deep software stack, as shown in Figure 4(a). The key to achieve high performance and low overhead overlay network for containers is to avoid, in the data-plane, any performance bottlenecks such as bridges, software routers and host OS kernel. Given that containers are essentially processes, the communication channels provided by host-based IPC and hardware offloaded network transports (RDMA) give us numerous options to build a better container network. For instance, containers within a single host, like Container 1 and Container 2 in Figure 4(a), can communicate via a shared-memory channel, and overlay routers in different hosts can talk via RDMA (or DPDK) to bypass the performance bottlenecks. Note that communication paradigms like shared-memory and RDMA partially sacrifice the isolation of containers. However, since in most cases containers that communicate with each other are part of the same larger, distributed application deployed by a single tenant, we believe that that trading off a little isolation for a large boost in performance is acceptable. We will discuss security implications of our design in more details later in the paper.

Generally, one container should decide how to communicate with another according to the latter's location, using the optimal transport for high networking performance. There are two issues to realize this key idea: (1) How to discover the real-time locations of containers; (2) How to enable containers to use different mechanisms to communicate with different peers.

One way is to solve these two issues is to depend on containerized applications themselves: the applications can discover and exchange location information and agree on a communication mechanism to use. This method requires applications to do extra work (and the code can become quite complicated, as the programmer deals with different communication methods), and hence is undesirable.

Instead, we take an alternative approach: using a (conceptually) centralized orchestrator to decide how containers should communicate, and keeping the container locations and the actual communication mechanisms transparent to containerized applications. Our key insight is that since currently most of the container clusters are managed by a centralized cluster orchestrator (e.g., Mesos, Kubernetes, and Docker Swarm)[3], the information about the location of the other endpoints can be easily obtained by querying the orchestrator. By leveraging this information, we can choose the right communication paradigm for the specific scenario. Furthermore, all of the complexity of communication mech-

[3]They can easily be deployed on private bare-metal clusters or cloud VM clusters without any special supports from cloud providers.

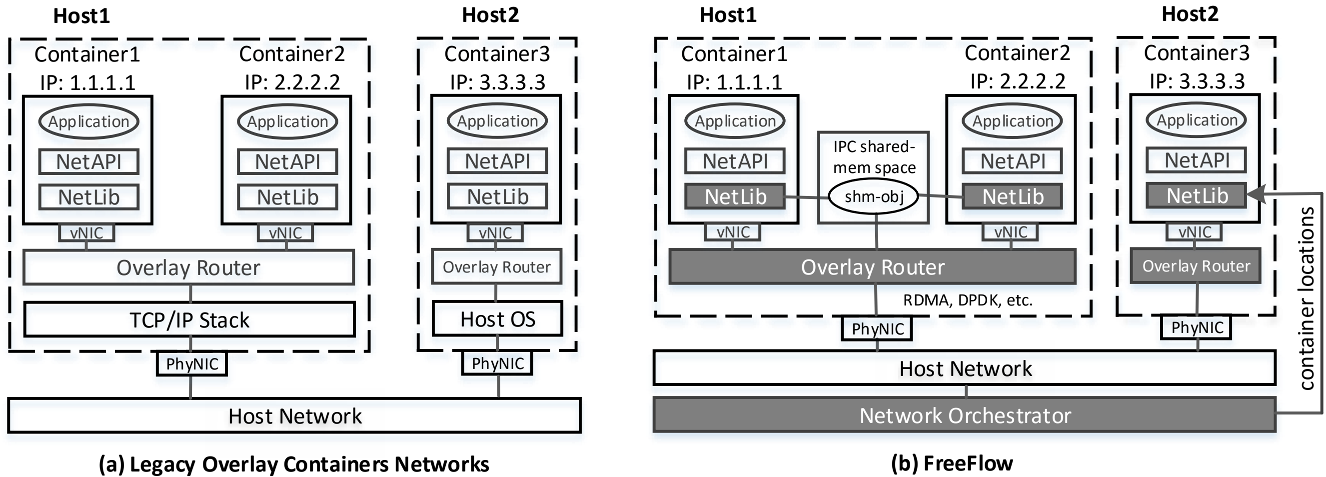

Figure 4: The overall system architecture of existing overlay network and FreeFlow. Gray boxes are building blocks of FreeFlow.

anism selection and execution can be hidden from the application by bundling it into a customized network library supporting standard network APIs. Next, we sketch the architecture of our solution.

3.2 The architecture of FreeFlow

Figure 4(a) shows the architecture of existing overlay networking solutions for containers. Each container has a virtual NIC that is attached to the overlay router of the host via a software bridge. Different overlay routers exchange routing information and build routing tables via standard routing protocols, such as BGP. The fabric built by virtual NICs, bridges, overlay routers, physical NICs and the host network is the actual data-plane for packets traversing the overlay from container to another one. Inside each container, applications use standard network APIs to access the network. The API calls are implemented in network libraries, such as `glibc` for Socket API, and `libibverbs` for RDMA Verbs API.

FreeFlow reuses many control-plane features like IP allocation and routing implemented by existing solutions such as *Weave*. However, FreeFlow modifies multiple existing modules in the networking stack to achieve a smarter and more efficient data-plane.

Figure 4(b) shows the overall architecture of FreeFlow. The gray boxes in Figure 4(b) represent the three key building blocks of FreeFlow: customized network library, customized overlay router and customized orchestrator.

FreeFlow's network library is the core component which decides which communication paradigm to use. It supports standard network programming APIs, e.g. Socket for TCP/IP, MPI and Verbs for RDMA, etc. It queries the network orchestrator for the location of the container it wishes to communicate with. Whenever possible, it uses shared memory to communicate with the other container, bypassing overlay router. FreeFlow's overlay routers are based on existing overlay routers. We add two new features: (1) the traffic between routers and its local containers goes through shared-memory instead of software bridge; and (2) the traffic between different routers is delivered via kernel bypassing techniques, e.g. RDMA or DPDK, if the hardware on the hosts is capable. Network orchestrator keeps track of the realtime locations of each container in the cluster. Our solution extends existing network orchestration solutions, and allows FreeFlow's network library to query for the physical deployment location of each container.

The architecture enables the possibility to make traffic among containers flow through an efficient data-plane: shared-memory for intra-host cases and shared-memory plus kernel bypassing networking for inter-host cases.

However, we have two challenges to achieve FreeFlow's simple vision. First, the network library of FreeFlow should naturally support multiple standard network APIs for transparency and backward compatibility. Second, for inter-host cases, overlay routers should connect the shared-memory channel with local containers and the kernel bypassing channel between physical NICs to avoid overhead caused by memory copying. In next section, we discuss how FreeFlow addresses these challenges.

4 Design

This section presents the designs of FreeFlow's major components.

4.1 The network library of FreeFlow

The network library of FreeFlow is the key component which makes the actual communication paradigm transparent to applications in the containers. It has two goals: (1) supporting most common network APIs, such as Socket (TCP/IP), Verbs (RDMA), MPI (parallel computing) and so on; and (2) selecting the most efficient communication paradigm no matter which network API is used.

One straightforward way to build the network library is to develop several independent libraries each of which deals

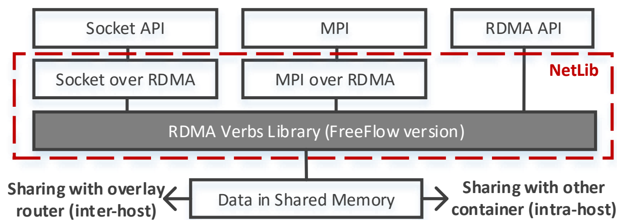

Figure 5: The internal structure of FreeFlow's network library. The gray box is built by FreeFlow.

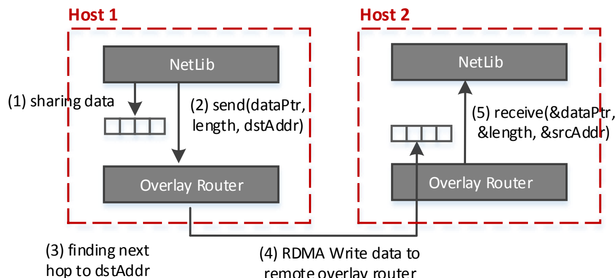

Figure 6: The working flow of sending data from one host to another via overlay routers in FreeFlow.

with a specific network API. For instance, we extend the socket implementation of `glibc` for supporting Socket API and design a new `libibverbs` for RDMA Verbs API. However, writing different libraries is clearly suboptimal. Instead, as shown in Figure 5, we merely develop a new library for RDMA API, and use existing "translation" libraries such as [9, 18, 23, 30] to support socket and MPI APIs atop the RDMA API. Note that we could have made the choice the other way as well: e.g. support socket API natively and use translation libraries to support other APIs atop it. We chose RDMA API as our primary interface, since it provides a message-oriented interface that maps naturally to the communication patterns of many containerized applications. The TCP/IP sockets interface can be easily implemented over RDMA [21].

After the network library receives calls to send data to another container, it first checks (from the network orchestrator) the receiver container's location. If the container is on the same host, it will put the data into a shared memory block and send the pointer of the memory block to the receiver's network library module. The latter will use correct API semantics to notify the application on the receiver container that the data is ready to read. Otherwise, if the receiver container is on a different host, the network library will share the data with the overlay router on the same host, and tell the overlay router to send the data to the receiver container's IP. Then it relies on the overlay routers to deliver the data to the destination.

4.2 The overlay router of FreeFlow

Overlay router has functionalities in both control-plane and data-plane. In control-plane, it allocates IP addresses for new containers according to default or manually configured policies. It also exchanges routing information and compute routes to all containers in the cluster. FreeFlow inherits the control-plane behaviors from existing overlay network solutions.

In the data-plane, FreeFlow has its own implementation to make the data transfer more efficient. Figure 6 shows an example of how overlay routers deliver data from a sender container to a receiver container. As described in §4.1, the network library in the sender container will share the data with its local overlay router (Step 1) if the former finds that the receiver is on another host. After that, the network library will tell the overlay router to send the data to the receiver's IP address (Step 2). The overlay router will check its routing table to find the next hop router towards the receiver (Step 3) and (Step 4) it writes the data to the next hop overlay router via RDMA (or DPDK, if available). If RDMA or DPDK are not available, normal IP communication is used. If the next hop router finds the receiver is on its host, it will share the received data with the network library on the same host and notify the latter that the data is ready to fetch (Step 5).

Note that in this design, there is only one time data copy from one host to another host, which is unavoidable. The communications between network library and overlay router on the same host are all performed via shared-memory.

4.3 The network orchestrator

The main functionality of the network orchestrator is to maintain the real-time information of container locations and VM locations (if needed). Since containers are typically started and managed by a cluster orchestrator (e.g. Mesos), the container to host mapping can be easily accessed from the cluster orchestrator. FreeFlow only adds a small module into existing clusters orchestrator to allow the network library modules to query the container-to-host mapping. Note that the orchestrator can either push the mappings to network libraries, or the libraries can pull it. The two choices have different scalability implications. We are investigating this tradeoff in the further.

5 Preliminary Implementation

We have implemented a prototype of FreeFlow on a testbed of clustered bare-metal machines (Intel Xeon E5-2609 2.40GHz 4-cores CPU, 67 GB RAM, 40Gbps Mellanox CX3 NIC, CentOS 7). The prototype selects the most efficient data-plane mechanism based on the location of two containers: if the two containers are intra-host, shared-memory mechanism will be selected for data transfer, and if the two containers are inter-host, RDMA will be selected. We implemented the shared-memory via multiplexing `IPC namespace`, and enabled containers to use RDMA by using the `host mode`. We evaluated our prototype and compare it with state-of-the-art container overlay network - `Weave`, which is using socket communication. The results are shown in Figure 7. We see that the FreeFlow prototype achieves higher throughput and lower latency. The CPU utilization per bit/second is also significantly lower for FreeFlow compared to that of

Weave. This is because our prototype always selects the most efficient communication mechanism in different settings – for intra-host, our prototype chose shared memory; and for inter-host, our prototype chose RDMA.

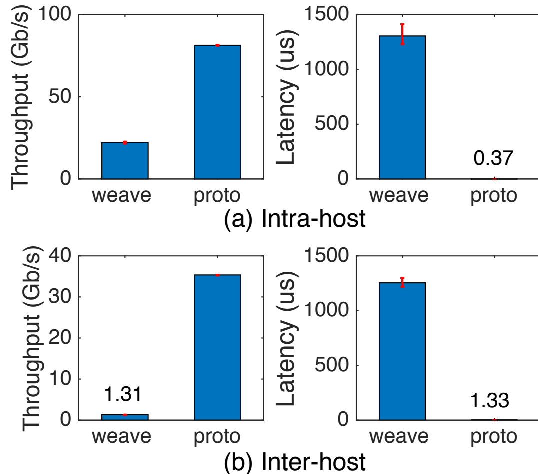

Figure 7: Compare FreeFlow prototype with weave.

6 Related Work

Inter-VM Communication: The tussle between isolation and performance is not unique to containers. The issue has also been studied in the context of Inter-VM Communication. For example, NetVM [26] provides a shared-memory framework that exploits the DPDK library to provide zero-copy delivery between VMs. Netmap [37] and VALE [38] (which is also used by ClickOS [32]) are sharing buffers and metadata between kernel and userspace to eliminate memory copies. However, these systems cannot be directly used in containerized setting. For example, the NetVM work is applicable only to intra-host setting, constrained by the possibility of shared memory. It does not handle inter-host communication. Similarly, the Netmap and VALE solutions are sub-optimal when the VMs/containers are located on the same physical machine: shared memory provides a much more efficient communication mechanism.

RDMA and HPC: RDMA originated from the HPC world, in the form of InfiniBand. The HPC community proposed RDMA enablement solutions for virtualization [36] and containerization [29] technologies. These solutions are addressing the challenges in exposing RDMA interfaces to virtualized/containerized applications, treating each VM/container as if it resides on a different node.

The HPC community has also been using shared-memory based communication [20, 25, 35] for intra-node communication. These solutions are targeting MPI processes residing on a shared non-containerized, non-virtualized machine. They do not attempt to pierce the virtualization/containerization for additional performance.

The same concepts described for FreeFlow can also be applicable for MPI run-time libraries. This can be achieved either by layering the MPI implementation on top of FreeFlow, or by implementing a similar solution in the MPI run-time library.

General improvements: A significant amount of work has been spent attempting to improve the performance of the networking stack [10, 18, 40] in various scenarios. However, none of them were aiming at optimizing the performance of networking communications between co-residing containers. While the performance of intra-node communication for containers was identified as relevant before [14], to our knowledge there was no attempt at addressing this challenge.

7 Conclusion and Discussion

In this paper, we discussed how to build a network solution for containers, named FreeFlow. It offers high performance, good portability and acceptable isolation. We sketched the design of FreeFlow which enables containers to communicate with the most efficient way according to their locations and hardware capabilities and keeps the decisions transparent to applications in containers. We are currently building FreeFlow, and we list a few important considerations below.

Live migration: FreeFlow could be a key enabler for containers to achieve both high-performance and capability for live migration. It will require the network library to interact with the orchestrator more frequently, and may require maintaining additional per-connection state within the library. We are currently investigating this further.

Security and middle-box: One valid concern for FreeFlow is how legacy middle-boxes will work for communication via shared-memory or RDMA, and whether security will be broken by using shared-memory or RDMA. We do not yet have complete answer to this issue. We envision that for security, FreeFlow would only allow shared-memory among trusted containers, for example, container belongs to the same vendor (e.g., running spark or storm). We are investigating how best to support existing middle-boxes (e.g. IDS/IPS) under FreeFlow.

VM environment: So far our evaluation and prototype is based on containers running on bare-metal. But our design easily generalizes to containers deployed inside VMs. Some issues, such as efficient inter-VM communication (perhaps using NetVM [26]) need to be addressed, but we believe that it can be easily done within the context of FreeFlow design.

Scalability of FreeFlow: Scalability of FreeFlow is a major focus of our ongoing work. The design proposed in §4 has a few potential scalability challenges. For example, overlay routers in FreeFlow may end up having to maintain per-flow state if they communicate with co-located containers using shared memory and with remote routers using RDMA – if done näively, the router may end up maintaining one queue pair for each pair of communicating containers. To solve this problem, we need to to multiplex a single queue pair on overlay routers for multiple container sessions.

Acknowledgments

This work was supported in part by NSF award number CNS-1440056.

8 References

[1] Docker. http://www.docker.com/.
[2] Kubernetes. http://kubernetes.io/.
[3] Project calico. https://www.projectcalico.org/.
[4] Apache Hadoop. http://hadoop.apache.org/, accessed 2016.
[5] CoreOS. https://coreos.com/, accessed 2016.
[6] Data plane development kit (DPDK). http://dpdk.org/, accessed 2016.
[7] Weave Net. https://www.weave.works/, accessed 2016.
[8] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *USENIX OSDI*, 2010.
[9] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda. Sockets direct protocol over infiniband in clusters: is it beneficial? In *IEEE ISPASS*, 2004.
[10] J. Brandeburg. Reducing network latency in linux. In *Linux Plumbers Conference*, 2012.
[11] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM*, 2015.
[12] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM*, 2011.
[13] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM*, 2014.
[14] J. Claassen, R. Koning, and P. Grosso. Linux containers networking: Performance and scalability of kernel modules. In *IEEE/IFIP NOMS*, 2016.
[15] Datadog. 8 suprising facts about real Docker adoption. https://www.datadoghq.com/docker-adoption/, 2016.
[16] Docker. Docker community passes two billion pulls. https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/, 2016.
[17] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: fast remote memory. In *USENIX NSDI*, 2014.
[18] M. Fox, C. Kassimis, and J. Stevens. IBM's Shared Memory Communications over RDMA (SMC-R) Protocol. RFC 7609 (Informational), 2015.
[19] FreeBDS. chroot – FreeBDS Man Pages. http://www.freebsd.org/cgi/man.cgi, FreeBDS 10.3 Rel.
[20] B. Goglin and S. Moreaud. Knem: A generic and scalable kernel-assisted intra-node MPI communication framework. *Journal of Parallel and Distributed Computing*, 73(2), 2013.
[21] D. Goldenberg, M. Kagan, R. Ravid, and M. S. Tsirkin. Zero copy sockets direct protocol over infiniband-preliminary implementation and performance analysis. In *13th Symposium on High Performance Interconnects (HOTI'05)*, pages 128–137. IEEE, 2005.
[22] P. S. I. Group. Single root I/O virtualization. http://pcisig.com/specifications/iov/single_root/, accessed 2016.
[23] S. Hefty. Rsockets. In *OpenFabris International Workshop*, 2012.
[24] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI*, 2011.
[25] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. D. Gropp, V. Kale, and R. Thakur. Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95, 2013.
[26] J. Hwang, K. Ramakrishnan, and T. Wood. Netvm: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1), 2015.
[27] 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.
[28] M. Li, D. G. Andersen, A. J. Smola, and K. Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, 2014.
[29] L. Liss. Containing RDMA and high performance computing. In *ContainerCon*, 2015.
[30] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over infiniband. *International Journal of Parallel Programming*, 32(3), 2004.
[31] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, et al. Jitsu: Just-in-time summoning of unikernels. In *USENIX NSDI*, 2015.
[32] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *USENIX NSDI*, 2014.
[33] Mellanox. RDMA aware networks programming user manual. http://www.mellanox.com/, Rev 1.7.
[34] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), 2014.
[35] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009.
[36] A. Ranadive and B. Davda. Toward a paravirtual vRDMA device for VMware ESXi guests. *VMware Technical Journal, Winter 2012*, 1(2), 2012.
[37] L. Rizzo. Netmap: a novel framework for fast packet i/o. In *USENIX Security Symposium*, 2012.
[38] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *ACM CoNEXT*, 2012.
[39] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, et al. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing*, 2013.
[40] J. Wang, K.-L. Wright, and K. Gopalan. Xenloop: A transparent high performance inter-vm network loopback. In *ACM HPDC*, 2008.
[41] Y. Zhu, H. Eran, D. Firestone, C. Guo, et al. Congestion control for large-scale RDMA deployments. In *ACM SIGCOMM*, volume 45, 2015.