Explaining and Evaluating the Use of RDMA in High Performance Containers

Honors Computer Science Bachelor's Thesis Proposal Document

Author: Emerson Ford

Advisor: Ryan Stutsman

Honors Faculty Advisor: Tom Henderson

Director of Undergraduate Studies: James de St. Germain

July 29, 2021

Abstract

Containers are an increasingly popular packaging framework for complex applications, offering benefits such as lightweight isolation, portability, and ease of deployment. These benefits have the potential to solve a myriad of issues that have long been present in the high performance computing world, presenting a compelling narrative for their use in these environments. Unfortunately, standard container runtimes rely on relatively slow, virtualized network stacks to achieve network isolation and portability, which are incompatible with the kernel bypass networking technology RDMA. This has limited the widespread adoption of containers in high performance computing environments where RDMA-reliant applications are quite common.

Fortunately, recent projects such as Microsoft's FreeFlow and Mellanox's RDMA hardware have created solutions for enabling RDMA inside of containers while maintaining varying degrees of the benefits of standard containers. However, despite the strong claims made by these solutions, many of their characteristics such as performance, isolation sacrifices, and scalability are either not well documented or simply not known; these are characteristics that are critical to assess for several high performance computing use cases. This thesis attempts to remedy this issue by identifying, measuring, and comparing these characteristics for the various solutions available today for enabling RDMA in containers; ultimately providing high performance computing end-users a more holistic perspective on which solutions may be most viable for their environment and use case.

1 Introduction

Containers offer a promising solution to a host of issues that have long plagued the high performance computing (HPC) world from dependency management to portable and reproducible environments to even lightweight sharing of hardware resources; all this is achieved through the use of statically built images and standardized container runtimes [**containershpc**]. For users, cluster administrators, and application developers, this would provide more freedom and ease in application development and deployment as it eliminates the burden of working around the hundreds of possible environments on which the application may be deployed. As an added benefit, this also opens the door for easy migration and utilization of new HPC platforms such as the cloud, Kubernetes, and SLATE that offer new compute and scheduling capabilities and treat containers as first-class objects [**slatepaper**].

However, despite these numerous benefits, the adoption of containers in the HPC world has been notably slow [**hpcappscontainers**]. Among other issues like the until-recent lack of rootless container runtimes, native containers do not support Remote Direct Memory Access (RDMA), a networking technology that provides extremely low latency and high throughput with minimal CPU overhead by offloading memory accesses from the CPU to NIC hardware (hence the term "kernel bypass networking") [**mellanoxrocerdmabenefits**]. This lack of support heavily restricts the use of native containers in HPC environments as trends like disaggregated storage, multi-machine workloads, and increased use of parallel libraries such as MPI have necessitated the use of high performance networking like RDMA to maintain performance. To make

the lack of RDMA support even worse, standard container network stacks by themselves have been shown to be both CPU intensive and significantly reduce container networking performance [**abbasi2019performance**, **xavier2013performance**, **lee2018impact**, **zhao2017performance**, **kapovcius2020performance**], further worsening the networking downsides of containers in HPC environments.

While it is possible to run containers without container networking by exposing the host's NIC directly to running containers (a mode called "host networking"), this comes with several notable disadvantages: (1) controlling fair sharing of the RDMA-capable NIC between multiple containers cannot be done programmatically, (2) running containers are not relocatable/portable across multiple hosts, (3) common container orchestration tools like Kubernetes cannot be used, and (4) network isolation and network routing policies for containers is disabled. Thus, while this mode may be a viable for some environments, it is not adequate for the vast majority of container environments — such as those that utilize Kubernetes heavily — and additional solutions for enabling RDMA in containers are still needed.

Unfortunately, enabling support for RDMA in containers (and fast networking as a whole) is a nontrivial task. Container runtimes rely on "virtualized" network stacks for creating flexible network overlays, enforcing isolation, and providing portability [**xavier2013performance**]. These network stacks have, until recently, only been expressible through software layers provided by the host kernel, adding significant software overhead to the processing of all network packets originating from and destined to containers. When provided by software alone, these network stacks are fundamentally incompatible with RDMA which requires bypassing the host kernel entirely for its

performance guarantees.

Despite this, multiple groups have recently developed solutions for enabling RDMA in containers, each with certain sacrifices in container and/or RDMA guarantees to overcome this incompatibility. The challenge is then to determine which solution (and subsequent sacrifices in guarantees) works best for a given environment. This is the core problem this thesis seeks to address. It is currently unnecessarily difficult to pick a solution due to a lack of exhaustive performance data and thorough explanations on exactly which guarantees are sacrificed for these various solutions. This information is critical in HPC environments as applications may heavily rely on RDMA's low latency and/or high throughput guarantees; similarly, container security and isolation guarantees may be a requirement for those HPC environments that are multi-tenant or are running untrusted applications.

Therefore, this thesis will analyze multiple properties of the several solutions available today for enabling RDMA in containers, such as SoftRoCE [**pandeysroce**], Mellanox's Shared HCA/SRIOV [**mellanoxdockerroce**], Microsoft's FreeFlow [**kim2019freeflow**], and MasQ [**he2020masq**]; these properties being grouped in two main categories:

Container Network Properties:

- *Network Isolation*: each container has its own interface, port space, and network policies (also known as a network namespace); further, a container's IP should not inherently depend on the underlying NIC's IP
- *Controllability*: enforcing admission control, routing policies, and traffic shaping on a container with an RDMA-capable NIC
- *Resource Utilization*: container networks should not be CPU or memory intensive

RDMA Properties:

- *Throughput*: how closely can a solution match host RDMA throughput
- *Latency*: what latency overhead is incurred per message for a given solution

and analyzed with varying levels of:

- number of containers present across the cluster and on a single host
- message size of network payloads
- messages sent per second

which should adequately test the scalability of these properties. Finally, comments will also be made on:

- *Proprietary*: can a solution be used across various RDMA-capable NIC vendors
- *Maturity*: what support exists for a solution, how well tested is a solution
- *Ease in Deployment*: are application changes necessary for a solution, how difficult is it to deploy the solution
- *Execution Privileges*: can the solution be used without elevated capabilities such as `CAP_SYS_ADMIN`
- *Network Pressure*: what additional pressure do these solutions put on the network

to assist in determining if a given environment can even support a given solution.

Additionally, these solutions can be generalized to three implementations: pure software, paravirtualization-like, and pure hardware; with each solution in an implementation likely having similar properties. Thus, this thesis should also provide general insight as to which implementations may be best suited for a given HPC environment.

2 Background

2.1 Networking Planes

The state and logic of networking operations can be generally abstracted into three planes: the control plane, the data plane, and the management plane. Understanding this abstraction is vital for understanding how these various RDMA in container solutions operate and why they exhibit their specific performance characteristics.

The *control plane* handles the rules of how and to where network packets are forwarded. Operations such as initiating connections, terminating connections, and connection resource management are considered as part of control plane operations and policies such as firewalls, routing policies and tables, traffic shaping, and interface properties (e.g. IP addresses) are considered as part of control plane policies. It is desirable to have the control plane be highly flexible and configurable to allow for a wide expression of network topologies.

The *data plane*, also known as the forwarding plane, handles the actual transmission of network data based on the configured logic of the control plane. Operations such as memory copies into network buffers and packet transmission over the wire are considered as part of data plane operations and policies such as the size of network buffers are considered as part of data plane policies. It is desirable to keep the data plane as low overhead as possible as overhead in the data plane has a significant impact on network performance due to every byte of a packet having to traverse the data plane. Notably, it is the data plane that handles the "execution" of control plane policies, thus the control plane is only as expressive as the data plane allows.

The *management plane* handles the configuration of the control plane and data plane policies. Its operations consists primarily of using software tools such as the iproute2 collection of tools, `iptables` / `nftables`, `DHCP`, and `ovs-vsctl` that take input to reconfigure the specified plane's policies. The management plane's policies primarily consist of security; specifically, defining who has access to use its tools, with what capabilities, and in what contexts.

In traditional host networking, the control plane and the data plane[1] exist as software in the kernel. This provides an extremely high degree of flexibility and configurability for the control plane and data plane, but it results in the kernel being interposed in every networking operation. As a result, this incurs performance penalties from the overhead of context switches, memory copies from user-space to kernel-space (or from RAM to CPU to the network device), and from simply being software (i.e. not being hardware accelerated or run on FPGAs). On top of this, use of standard network protocols like TCP may exacerbate the performance penalties due to higher code complexity, built in retries and throttling, and protocol header overhead.

2.2 RDMA Networking

Remote Direct Memory Access (RDMA) is an aptly named network protocol designed for the direct reading and writing of the memory of remote hosts with extremely low latency, high bandwidth, and minimal CPU overhead. Support for it has been added to a number of parallel applications and libraries, such as TensorFlow and OpenMPI, and has seen wide spread adoption in the high performance computing world where it is run on

[1] barring the actual transmission on the wire which the NIC handles

top of either Infiniband or Ethernet [**kim2019freeflow**]. At its core, RDMA is a form of kernel-bypass networking in that it shifts almost all of the data plane and control plane from the kernel to a combination of user-space and the network card. While this eliminates most of the control plane and data plane configurability provided by the kernel, it provides substantial improvements in network latency and bandwidth. For example, current top-of-the-line RDMA enabled NICs are capable of achieving sub-1µs latency with >100Gb/s flows, performance not possible with even the most aggressive tuning of the Linux networking stack [**mellanoxcx6doc**, **cloudflarelowlatency**, **intel800nic**].

Understanding the data paths, architecture, and implementation of network planes for RDMA itself is critical for understanding how and why these various RDMA in containers solutions work and will therefore be explained.

2.2.1 Data Plane

While traditional host networking relies on kernel syscalls and interrupts as a communication channel for signalling packet sends and receives, RDMA instead utilizes in-memory queues that live in regions of memory explicitly shared between the user-space process and the NIC[2] as its data plane communication channel. A user-space process will write a *work request* (WR) to the designated TX Queue to initiate a message[3] send. The NIC will spin-loop on these shared-memory queues to detect when a work request has been posted. Inside of this work request will be a fat pointer[4] to another region of shared memory — this being the actual message payload. The NIC will consume the

[2] the NIC using IOMMU to directly access memory without CPU interposition

[3] in contrast to TCP and UDP where the user-space process handles chunking a message into packets, RDMA operations post full messages and the NIC handles chunking the message into packets

[4] fat pointers being a pointer with extent information

posted work request, read the payload directly from memory (a process known as Direct Memory Access or DMA), then transmit the payload over the wire, all without any kernel or CPU interposition. The user-space process is then notified of whether that message send succeeded or not by a *completion queue event* (CQE) that is posted by the NIC to the designated Completion Queue.

Following this, a user-space process can receive messages by posting a *receive request* (RR) to the designated RX Queue, with these requests containing fat pointers to writable buffers in shared memory. Only after a receive request with a memory buffer large enough has been posted will a NIC accept an incoming message. When an incoming message is accepted, the NIC will similarly write the incoming payload to the specified shared-memory buffer found in the receive request [**rdmaawareprogramming**].

It is worth noting that what is described above is technically considered *two-sided* RDMA operations (these use the verbiage *send* and *receive*). RDMA is also capable of *one-sided* operations that consist of directly reading and writing the memory of the remote host (these use the verbiage *write* and *read*) without the remote host being involved in or aware of such operations, i.e. the remote user-space process does not need to post receive requests. These one-sided operations follow a nearly identical data path to two-sided operations, minus the need for the RX Queue.

2.2.2 Control Plane

With the data plane split between shared-memory and the NIC, it then follows that the majority of the RDMA control plane must also live between shared-memory and the NIC. This, in conjunction with RDMA having a different implementation for each network

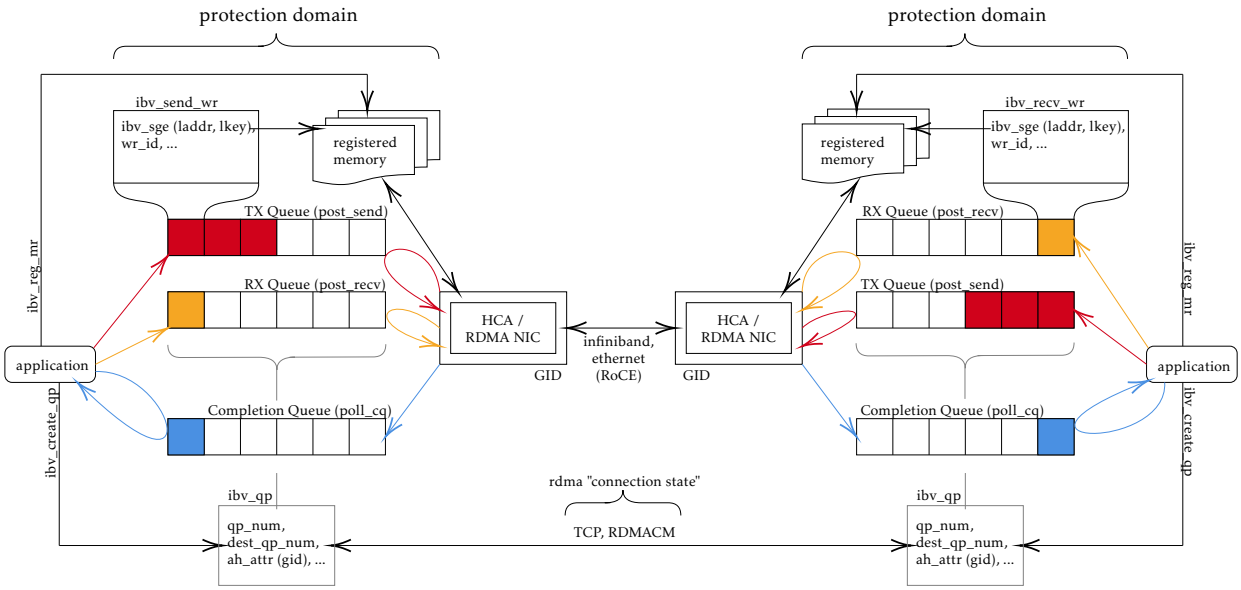

Figure 1: Diagram of RDMA's data paths

fabric, results in the RDMA control plane having a radically different paradigm than that of traditional host networking. There are four key pieces that make up the RDMA control plane: the RDMA API library and device drivers, connection state, routing information, and interface management (i.e. multiplexing).

First, user-space processes utilize RDMA through a library known as `libibverbs`, which provides a central API for RDMA use regardless of network fabric [**rdmacoredocumentation**]. This library handles both data plane operations, such as creating work requests or polling the completion queue, and control plane operations such as device metadata queries, queue creation and instantiation, or registration of shared memory with the NIC. However, while data plane operations use the shared-memory regions as a communication channel, most control plane operations cannot do the same. For example, these operations may involve the actual creation or manipulation of the shared memory region, or may be invoked when a shared memory region does not exist. Thus, device drivers must be used as a communication channel for these operations. This is, critically, one of two

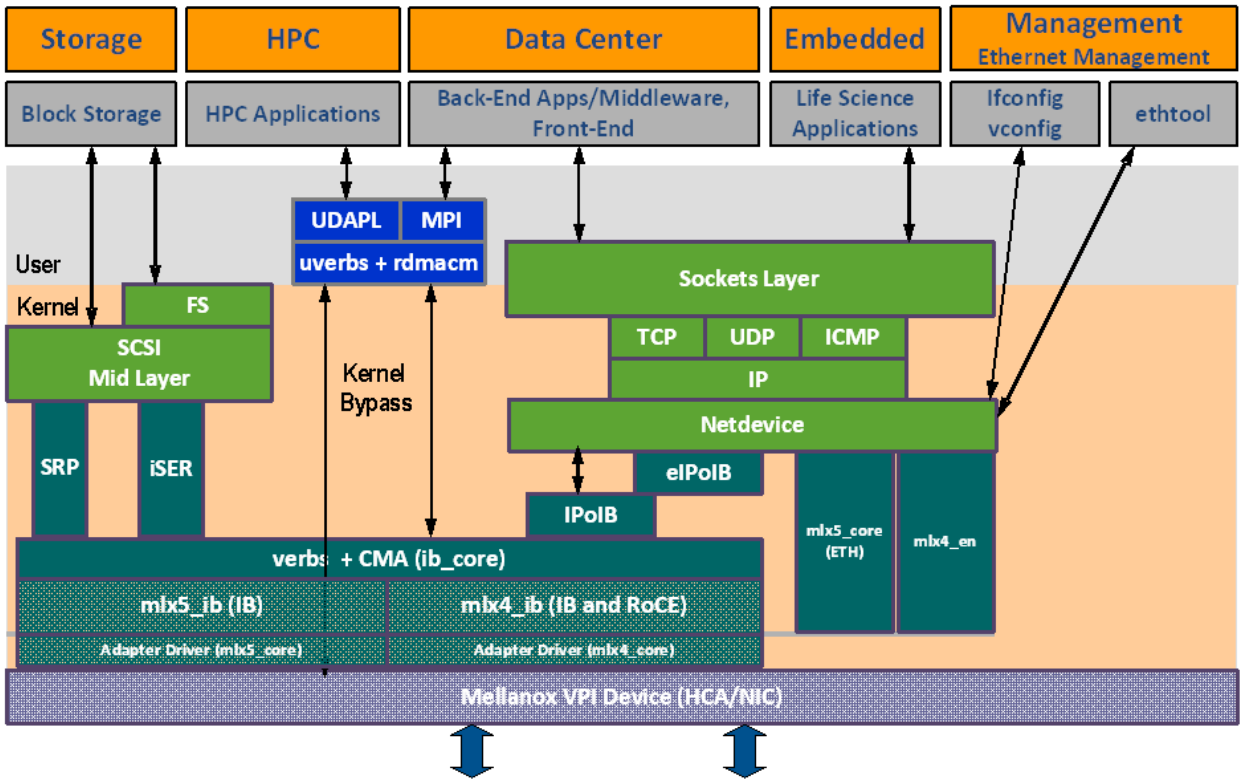

Figure 2: `libibverbs` stack
Source: [**mlnxofedmanual**]

times the kernel is involved in RDMA operations[5]. Alongside `libibverbs` are drivers for the NIC itself; these drivers expose character device files (usually in `/dev/infiniband` and/or `/sys/class/infiniband`) that user-space processes can interact with[6] as a control plane communication channel to the NIC [**linuxkernellibibverbs**]. Thus, all RDMA operations are initiated with `libibverbs`, which then feed into either shared memory for data plane operations or device drivers using character devices for control plane operations.

Second, RDMA maintains a notion of connection state, similar to that of TCP, stored in a structure known as `ibv_qp`. This structure contains fields such as `qp_num`, `ibv_device`,

[5] these operations tend to be so infrequent they incur almost no network performance penalty
[6] usually with `read`, `write`, or `ioctl` syscalls, which is what results in the kernel / driver involvement

`dest_qp_num`, and `dlid` / `dgid`[7]. These are, in respective order, analogous to local port, local IP, remote port, and remote IP in a TCP connection, and tell the NIC to which machine and to which receive queue on that machine all work requests in a specific TX queue should go to. Further, this structure also contains fields such as `ibv_pd` which specify which regions of shared memory a remote host can access using a set of generated cryptographic keys.

However, there is the issue of how some of the state in this structure — specifically `dest_qp_num`, `dlid` / `dgid`, pointers to remote memory, and the cryptographic keys for accessing those regions of remote memory — is populated. The RDMA protocol itself does not specify how to share this state and is thus deferred to standard networking facilities, this being the second time the kernel is involved in RDMA operations. The user-space process can either manually populate and share this state over a standard protocol like TCP, or use the `rdmacm` facilities (having its own library and character devices) which handles the full creation of `ibv_qp` using the most appropriate network protocol. Once this state is populated, this structure is then registered with the NIC [**rdmacoredocumentation**, **rdmaawareprogramming**].

Third, RDMA-capable NICs are, first and foremost, standard NICs and thus have at least minimal support for routing capabilities. On Ethernet fabrics, this means that at the hardware layer (ergo being exposed to the RDMA control plane) the RDMA-capable NIC has a MAC address and, in many cases, support for VLAN tagging. These properties are used by the now outdated RDMA protocol RoCE v1, which allowed for RDMA over a single Ethernet broadcast domain. Now, a MAC address alone does not suffice for

[7] these are an abstracted form of layer-2 and layer-3 addresses respectively for use on all network fabrics

standard networking, thus the host networking stack adds an IP layer on top of this for layer-3 routing. Normally, this IP layer exists solely in the host's software, but drivers for the RDMA-capable Ethernet NICs have added functionality to expose IP addresses and VLAN tags to the NIC, which are then stored in the NIC's GID table. These are then accessible to the RDMA control plane and are used by the RDMA protocol RoCE v2 that encapsulates RDMA packets inside of UDP packets for IP routable RDMA. On Infiniband fabrics, the story is more straight forward. In contrast to Ethernet NICs, most of the Infiniband networking logic lives on the NIC hardware (these Infiniband NICs being formally called HCAs), thus much of the Infiniband interface information — such as its Local Identifier (LID), a layer-2 address, and Global Identifier (GID), a layer-3 address — is stored on the NIC and is accessible to the RDMA control plane [**mlnxofedmanual**]. However, for both Ethernet and Infiniband, routing information — specifically subnet mask, gateway address, and which interface to send out of — is not exposed by the kernel or drivers to the NIC. Thus, the user-space process shares these remaining pieces of required routing information using the `ibv_ah_attr` and `ibv_global_route` structures, which are nested inside of `ibv_qp` [**rdmaawareprogramming**].

Support beyond these minimal routing capabilities for the RDMA control plane, such as firewalls at the host level and complex routing rules, is dependent on the features the NIC itself provides as these would have to be executed in the NIC's hardware. This results in widely varying capabilities of the RDMA control plane depending on the NIC used — a point that will be pertinent when discussing the various RDMA in container solutions. Regardless, even if the NIC does not support some RDMA control plane functionality, an option may be to enforce it at the network level if the network fabric supports it.

Finally, RDMA-capable NICs tend to support the creation of multiple interfaces on top of a single NIC using both software and hardware based multiplexing. The exact capabilities and functionality of this multiplexing is, again, dependent on what the NIC or NIC driver provides; however, in general, software based multiplexing is managed by the kernel and integrates with the NIC using the aforementioned sharing and storage of IP addresses and VLAN tags to the NIC's GID table. When a new interface or IP address assignment is created in the kernel (using commands like `ip addr add`), this creates a new entry in the NIC's GID table, which can then be used for both RDMA connection state sharing (using the kernel's interface abstraction), and for RDMA sends and receives (by specifying this new GID in the `ibv_device` structure).

Hardware-based multiplexing, in general, uses Single Root I/O Virtualization (SR-IOV) which allows a single NIC (called the Physical Function or PF) to expose itself as multiple PCIe devices (each called a Virtual Function or VF). These PCIe devices will appear as unique hardware network devices (as opposed to a software abstraction) to the kernel or hypervisor that can be handed out to individual virtual machines or isolated in namespaces. On Ethernet, these VFs will appear as unique interfaces to the network, each with its own MAC address and, on some NICs, can have VLAN restrictions set on them by the host VM that cannot be overridden by the guest VM. On Infiniband, the network is not made aware of each VF (formally called a vHCA), and the PF handles switching from the network fabric to each individual VF. SR-IOV provides another form of virtualization, often useful in the context of virtual machines, but does suffer from hardware constraints as it relies on available hardware resources. For example, the number of possible VFs is usually capped to under 150; use of VFs on Ethernet NICs can result in IP address

exhaustion; and VFs will usually share the PF's GID table, which has a fixed size, meaning GID table entry exhaustion — which would result in the disabling of RDMA on some VFs — is a real possiblity [**mlnxofedmanual**].

2.2.3 Management Plane

The management plane for RDMA, which is integrated in the management plane for the NICs themselves, is then relatively straight forward. Similar to control plane operations, character devices and device drivers are used as the communication channel for management plane operations, with libraries like `libibumad` being used to define the communication data types. User tools like `ibstat` or `ibdiag`, as well as network managers like Infiniband's Subnet Manager, will use these libraries to reconfigure NIC policies. These character devices can also be hidden from user-space processes as a method to prohibit NIC policy configuration.

Recently, RDMA-capable NIC device drivers have begun adding functionality to existing kernel systems to allow for minimal NIC and RDMA policy configuration with native OS tools like `devlink` and iproute2. This has allowed for some native compatibility with existing kernel systems, like using capabilities to control IP assignment on RDMA-capable NICs [**mlnxofedmanual**]. However, it still remains that the majority of the kernel's data plane and control plane systems, such as `iptables`/`nftables`/netfilter, are not exposed to the RDMA-capable NIC and are thus invalid in the context of RDMA operations.

2.3 Containers

Containers are, as mentioned previously, a Linux-based packaging and deployment framework for complex applications. Reminiscent of a unikernel, containers provide portable runtime environments, strong isolation, and fine grain access control to kernel resources for user-space processes. These are accomplished with a combination of several Linux kernel systems.

First, for portable runtime environments. Containers are bundled as *container images*, which contains a file system[8] that holds all runtime dependencies and binaries for the application, much like a static binary. The kernel will then isolate the application process to this environment using `chroot`, or something similar like `pivot_root`, such that the application can view only the provided file system and will be restricted from viewing the host's file system [**dockerunderthehood**].

Second, for strong isolation. The Linux kernel provides *namespace* functionality that allows for isolating a given process's view of kernel resources. For example, if a process is placed in its own PID namespace, the kernel provided `procfs` will return, to that process, information such that the process appears to be running as PID 1 and will see only it and its children in the process tree[9]; therefore preventing the PID-namespaced process from viewing any other processes that may be running on the host. Similarly, other namespaces like the network namespace will return to the process an isolated view of available network resources. In total, the Linux kernel provides namespaces for PIDs, IPCs, Cgroups, IPCs, mounts, users, time, and hostnames (through the UTS

[8] usually a union file system to allow for deduplication across multiple container images

[9] internally, the kernel maintains a map of PIDs in process namespaces to PIDs in the root namespace for resolving the "true" PID of a process for operations like permission checks

namespace) [**juliaevanscontainers**, **namespacesman**].

Third, for fine grain access control. In conjunction with namespaces that can act as a rudimentary form of access control to kernel resources, the kernel provides *cgroups* for resource usage limiting and monitoring; as well as *capabilities* and *seccomp* for restricting kernel operations and syscalls. Cgroups are configurable controllers in the kernel that can monitor and limit per process usage of CPU time, memory, and IO usage. Capabilities are a form of permission control on groups of kernel operations for processes. For example, removing the `CAP_NET_ADMIN` capability from a process will restrict it from all network configuration operations and `CAP_IPC_LOCK` will restrict all memory locking operations. Seccomp is like capabilities, but addresses individual syscalls as opposed to groups of operations, thus providing more fine grain control [**juliaevanscontainers**, **cgroupsman**].

As these are all systems within the kernel, this provides an extremely low overhead and fast to reconfigure form of isolation, as compared to say virtual machines. Hundreds of containers can feasibly run on a single system with no more overhead than that of running hundreds of processes, and these containers can be deployed at speeds similar to that of launching a new process on Linux [CITE NEEDED].

2.3.1 Container Networking

Up to this point, the systems that have been described provide intra-machine isolation in that they can isolate multiple containers running on a single host. If a container utilizes networking, there is then the question of intra-machine isolation and access control, such as how to restrict to what hosts a container can initiate network connections with. These are accomplished with Linux's netfilter system (for which `iptables` or `nftables` are the

front end) and general network stack (for which the iproute2 collection of tools is the front-end).

First, netfilter provides a framework for defining network rules based on packet metadata. When a packet arrives or leaves an interface, the kernel will iterate over the defined netfilter rules to determine how to manipulate or restrict that packet — for example, to act as a host-level firewall or as a NAT. The kernel also provides hooks between netfilter and cgroups to allow netfilter to tag all packets originating from a container, where these tags can then be used in netfilter rules. Second, the Linux network stack has several highly expressive systems for interface management, routing tables, and traffic control. These tools can be used to create virtual interfaces, virtual bridges, advanced routing rules (often in conjunction with netfilter rules), and traffic shaping policies on specific network interfaces [**iproute2wiki**, **netfilterhome**].

Together, netfilter and Linux's expressive network stack are used to create controllable, highly expressive, and isolated virtual networks for containers. Virtual interfaces will typically be created for individual containers that are then added to that container's network namespace. Outside of the container, that interface will often be attached to virtual bridge interfaces, as well as have netfilter, traffic control, and routing rules put in place (e.g. NAT on outgoing traffic), to control where to route and how to shape network traffic to or from the container. This allows for a wide expression of network topologies, has the benefit of being fast to configure with these all being software systems, and allows containers to be relocatable in that they can be moved to other hosts without any disruptions in service availability. This also ensures that the container can be restricted from viewing the host's actual network, and vice versa where the container can be hidden

from the host's network.

This should highlight why container networking is not runtime performant. Every packet to or from a container must traverse the Linux network stack and netfilter system at least once for every interface, virtual or not, along the container's network path. For example, if a container has a virtual interface, that is then connected to a virtual bridge, which is then connected to the interface for the NIC; that container's network traffic must traverse the Linux network stack a minimum of three times. With the network stack being primarily implemented in software, meaning both the control plane and data plane are in software, the performance costs explained in Section 2.1 can add up.

Putting it together though, this, alongside the intra-machine isolation described in Section 2.3, provides an immense amount of power for container deployment. These systems are extremely quick to configure, highly expressive, and are relatively low overhead[10]. Further, container engines are typically used for the actual deployment of containers, such as Docker or podman, that handle the full configuration and monitoring of the aforementioned kernel systems, and use standards for both container images and schemas to describe container deployment. This makes container deployments simple, consistent across machines, and scalable, which explains why containers have become so widely adopted. As a final note, it is critical to once again emphasize that these are software systems that run inside of the kernel, and thus can only influence operations whose data path cross the kernel.

[10]for most users, other factors like processing time or multiple router hops overshadow the performance cost of container networks, thus container networking is usually considered low overhead

2.4 Related Work

2.4.1 RDMA Namespaces and Cgroups

With namespaces and cgroups being the standard systems for container isolation in the Linux kernel, there have been developments to add RDMA support for these to the Linux kernel itself.

[**rdmanamespaceman**, **rdmacgroups**, **mellanoxcontainersupdate2018**].

2.4.2 Sharing RDMA NICs between Virtual Machines

There's a lot of similarities between sharing RDMA NICs between containers and virtual machines. There are some differences though:

In particular, containers are unique in several key ways:

- Containers are treated as ephemeral -> need fast reconfigurability
- A single machine could run tens to potentially hundreds of containers
- Containers are relocatable
- Container networking is nearly all software-defined

2.4.3 Programmable NICs

Microsoft's FPGAs in Azure and Brent Stephen's work. This work is started to show up in RDMA capable NICs (e.g. ConnectX-6).

2.4.4 eBPF Container Networking

Projects like Cilium and Calico are using eBPF to eliminate context switch overheads in virtual networks. This adds some speed ups, but still suffers from the other overheads mentioned in Section 2.1.

2.4.5 DPDK in Containers

Another form of kernel bypass networking.

3 Overview

3.1 Paravirtualized Solutions: Microsoft Freeflow and MasQ

Keep the data plane in tact, but high jack the control plane. Requires keeping up to date with the libibverbs package.

In general, these sections should go:

- Table/Paragraph lists with Container Network Properties
- Graphs and commentary on RDMA performance
- Table/Paragraph lists on other comments like proprietary-ness

3.2 Hardware Solutions: Mellanox Shared HCAs and SRIOV

Implement more control plane flexibility in the NIC hardware.

Shared HCA and SRIOV overhead is already negligible, but is there any performance hits to using ASAP Offload?

3.3 Software Solutions: SoftRoCE

Just sucks for performance, really isn't meaningful to measure. Should only be used if compatibility with existing RDMA networks is required, but performance is not.

4 Discussion

4.1 Security

The RDMA protocol itself doesn't support encryption, required by the NIC for any meaningful performance.

4.2 Looking Forward

Microsoft Freeflow and MasQ high jack the `libibverbs` libraries, why not high jack the character devices themselves? Why not high jack just rdmacm itself? Actually, I think Mellanox might be going this direction with shared HCAs. They are kind of acting like "proxy" HCAs.

Likely moving more toward hardware implementations -> more proprietary.

Likely merging more RDMA NIC configurations into kernel drivers to get container support for free, iow interacting with RDMA NICs similarly to standard Ethernet NICs.

5 Conclusion

If you can afford it, NICs with hardware support for container topologies seem to be the best option for flexibility, support, and performance. You are at the mercy of Mellanox

though.

The paravirtualized solutions seem like a good alternative, but they require a lot of maintenance to keep up to date, especially with how complex and industry-driven the RDMA code base is.

References

Appendix