Container Orchestration by Kubernetes for RDMA Networking

Coleman Link*
Rochester Institute of Technology
cal8384@rit.edu

Jesse Sarran*
Rochester Institute of Technology
jes4313@rit.edu

Garegin Grigoryan
Rochester Institute of Technology
gg5996@rit.edu

Minseok Kwon
Rochester Institute of Technology
mxkvcs@rit.edu

M. Mustafa Rafique
Rochester Institute of Technology
mrafique@cs.rit.edu

Warren R. Carithers
Rochester Institute of Technology
wrc@cs.rit.edu

***Abstract*—With the widespread usage of containerized virtualization in data centers and clouds, it is important to enabling high-throughput and zero-copy data transfer between those containers. Remote Direct Memory Access (RDMA) allows bypassing the kernel for packet processing by offloading it to specific RDMA-enabled NICs. The existing solutions enabling RDMA with containers are either based on custom container orchestrators (e.g., FreeFlow) or lack the ability for the control plane to manage the underlying RDMA traffic (e.g., Kubernetes RDMA plug-in via SR-IOV). The work in this paper builds off of previous work in Kubernetes to make an architecture that allows control over bandwidth requirements of RDMA within a Kubernetes cluster.**

I. Introduction

For data center networking, RDMA has recently gained tremendous attention for its capability to provide extremely high throughput, low latency, and low CPU utilization. RDMA achieves all these by zero-copy networking in which network adapters directly transfer data with no CPU involvement bypassing the OS kernel. Containers (or containerized applications) are another technology widely used for data center networking for its isolation, lightweight nature, and portability. Containers are also popular as they provide reliable and persistent service at low cost. In a data center, it is often the case that hundreds of containers are deployed concurrently, and this necessitates the use of container orchestrators, e.g., Kubernetes [1], to automate the management of containers, their load balancing, and fast replication in case of system or network failures.

The isolation provided with containers, however, impedes their access to a host's RDMA interfaces. Additional support is needed to configure and connect these RDMA interfaces to running containers, as well as to ensure that containers are deployed to hosts within a cluster that can provide the necessary RDMA resources. FreeFlow [2] attempted to solve this problem as a container orchestrator for use of RDMA with an overlay router. FreeFlow, however, requires containers to be modified with its own library, and does not provide an interface comparable to that of Kubernetes, rapidly-developing open-source software with a large community of contributors for container orchestration. A Kubernetes plugin named *k8s-rdma-sriov*, designed by Mellanox [3], creates an overlay network among pods of Kubernetes, and virtualizes RDMA NICs for containers in pods using SR-IOV hardware-implemented technology for virtualizing PCI express device. Unfortunately, the current solution for RDMA still lacks in several properties:

- Bandwidth limiting: The ability to customize virtualized interfaces, i.e., Virtual Function (VF), to meet the need of each container.
- Heterogeneous nodes: Handling nodes with diverse numbers and capabilities of VFs.
- Multiple interfaces: Handling a request for a pod that requires multiple VFs.
- Per-container versus per-pod VF allocation: Containers can request a VF, while interfaces are shared across a pod in Kubernetes.

This is our first attempt to extend Kubernetes with the above capabilities. We design a new RDMA architecture for Kubernetes with the following components: (a) *RDMA Daemon Set* that runs on each node for initializing RDMA resources, (b) *Kubernetes Scheduler Extender* that finds the most appropriate node for deploying a pod with RDMA requirements, (c) a modified *Container Network Interface (CNI) plugin* that virtualizes an RDMA NIC for multiple containers. Our contributions are as follows:

- We provide RDMA bandwidth limiting and bandwidth reservation on a per-interface basis as part of a Kubernetes pod definition. With this, containers can have the amount of RDMA bandwidth they use limited, or have bandwidth set aside for their exclusive usage.
- We significantly improve support for heterogeneous clusters containing nodes with varying quantities of physical RDMA interfaces. Additionally, our scheduler considers bandwidth in use when finding a target node for a pod.
- We provide support for the allocation of multiple RDMA interfaces per pod as well as pods that require no RDMA interfaces.

*Both authors contributed equally to this research.

978-1-7281-2700-2/19/$31.00 2019 © IEEE

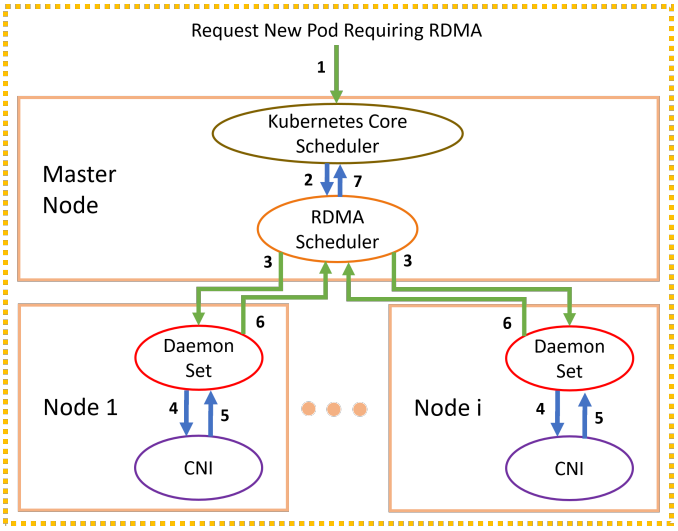

Fig. 1. System Architecture

II. System Architecture

The system architecture is shown in Figure 1. Two important goals in the system design and implementation are 1) not to modify any Kubernetes core code, and 2) to keep in line with the Kubernetes built-in plugins and extension concept. As shown in the figure, the solution contains three components, specifically the RDMA Hardware Daemon Set, the Scheduler Extender, and the Container Networking Interface. Note that the existing RDMA Device Plugin [4] provided by Mellanox is removed in its entirety. The components we created are highly extensible allowing for new features to be added easily in the future.

A. RDMA Hardware Daemon Set

The RDMA Hardware Daemon Set (or a *daemon set* in short) provides two critical functions: VF creation and maintenance. The daemon set is a pod that runs across all nodes in the cluster and has two containers that are associated with it. The first container is an *init* container, which is run before any other containers in the pod. Its purpose is to initialize all the VF's that exist on any node. The second container that runs after the *init* container is a *server*. It provides a RESTful endpoint that displays metadata on the VF's and RDMA SR-IOV enabled interfaces, also known as physical functions (PFs), on a node. When the container starts up, the server container scans the node to find all the PFs. It then sets up a RESTful endpoint and upon an HTTP GET request, it will return the metadata about each PF and their associated VFs in JSON format.

B. Scheduler Extender

The Scheduler Extender is a process that runs on the master node of a Kubernetes cluster. It decides whether each node has the necessary requirements for a pod deployment based on the number of VFs required and the minimum bandwidth requirements of each of those VFs. For example, in the case where a pod has a requirement of two VFs with a bandwidth of 100Gb/s each, it will only be placed on a node that has either a single interface that has at least 200Gb/s of unused bandwidth or has two interfaces that each has at least 100Gb/s of unused bandwidth.

C. Container Networking Interface (CNI) plugin

CNI is responsible for moving VFs from the node's namespace to the pod's namespace when the pod is starting up. When the pod is shutting down, CNI moves VFs back to the node's namespace. The CNI's original capabilities had come from Mellanox CNI [4], as it is only capable of adding a single VF to a pod's network namespace regardless of the pod's requirements. In our design, CNI is able to read the pod's configuration and determine how many VFs has been requested. The CNI then performs the same operation as the scheduler extender where it decides which interfaces to be used. The CNI then moves all the requested VFs to the pod's network namespace from the node's network namespace. Finally, the CNI sets an IP address for each VF, and configures the bandwidth requirements described in the pod's configuration.

III. Testing And Verification

In order to verify the functionality of our solution, we deployed containers with different bandwidth reservations and limitations, then ran connectivity and throughput tests between them. Our testing environment consisted of a master node with the Kubernetes orchestrator and two slave nodes, each with a pair of 100Gb/s RDMA NICs. To ensure the bandwidth management was working correctly, we ran a test that involved multiple pods on both slave nodes in the Kubernetes cluster. Finally, we tested pods that requested various numbers of RDMA interfaces, up to a couple of pods that required 60 interfaces each. This served as a stress test of the CNI plugin, which worked correctly when assigning interface names and IP addresses.

IV. Conclusion

In this work, we bridge the gap between Kubernetes container orchestrator and control over its underlying RDMA traffic. Unlike previous approaches, our solution moves away from a per-container basis for requesting RDMA resource to a per-pod basis. As a result, operators of a containerized data center can manage the RDMA traffic used by the applications in a pod by simply editing that pod's configuration file before its deployment.

References

[1] Kubernetes, "Kubernetes.io," 2018. [Online]. Available: https://kubernetes.io/

[2] D. Kim, T. Yu, H. H. Liu, Y. Zhu, J. Padhye, S. Raindel, C. Guo, V. Sekar, and S. Seshan, "Freeflow: Software-based virtual rdma networking for containerized clouds," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[3] Mellanox, "Kubernetes IPoIB/Ethernet RDMA SR-IOV Networking with ConnectX4/ConnectX5," 2019. [Online]. Available: https://community.mellanox.com/s/article/kubernetes-ipoib-ethernet- rdma-sr-iov-networking-with-connectx4-connectx5

[4] Mellanox, "Kubernetes Rdma SRIOV device plugin," 2019. [Online]. Available: https://github.com/Mellanox/k8s-rdma-sriov-dev-plugin