Wharf: Sharing Docker Images across Hosts from a Distributed Filesystem

Chao Zheng†, Lukas Rupprecht*, Vasily Tarasov*, Mohamed Mohamed*
Dimitrios Skourtis*, Amit S. Warke*, Dean Hildebrand*, Douglas Thain†
† University of Notre Dame, * IBM Almaden Research Center
czheng2@nd.edu,lukas.rupprecht@ibm.com,vtarasov@us.ibm.com,mmohamed@us.ibm.com,
dimitrios.skourtis@ibm.com,aswarke@us.ibm.com,dhildeb@us.ibm.com,dthain@nd.edu

ABSTRACT

Due to their portability and less overhead compared to traditional virtual machines, containers are becoming an attractive solution for running HPC workloads. Docker is a popular toolset which enables convenient provisioning and management of containers and their corresponding images. However, Docker does not natively support running on shared storage, a crucial requirement in large-scale HPC clusters which are often diskless or access data via a shared burst buffer layer. This lack of distributed storage support can lead to overhead when running containerized HPC applications. In this work, we explore how Docker images can be served efficiently from a shared distributed storage layer. We implement a distributed layer on top of Docker that allows multiple Docker daemons to access container images from a shared file system such as IBM Spectrum Scale or NFS. Our design is independent of the underlying storage layer and minimizes the synchronization overhead between different Docker daemons.

KEYWORDS

Linux Container, Docker, Shared Storage, HPC

1 INTRODUCTION

Due to the inherent performance overhead of traditional virtual machines (VMs), the HPC community has avoided using virtualization technology. With the rise of lightweight container technology, it has now become possible for HPC clusters to deliver an isolated environment with low overhead [2, 7]. Compared to VMs, the container runtime eliminates overhead by sharing the host system kernel across containers. Even though the basic technology for enabling containers has been existent in Linux distributions for years, it has only recently been standardized and adopted in the form of tools. Docker [1] is one of the most popular container runtimes and used by many companies and organizations from both industry and academia.

Docker consists of three components: (i) a registry service; (ii) a daemon running on a Docker host; and (iii) a client to interact with the daemon. The Docker registry is used to store and distribute Docker images across daemons. An image consists of multiple *layers*, each of which is a set of files that will be included in a running container. Layers can be shared across different images.The Docker daemon is the persistent process that runs on a host machine and manages containers and images while the Docker client is used to send user requests to the Docker daemon such as starting or stopping a container from a specific image.

When the daemon receives a request for creating a new container, it will use Linux cgroups to isolate compute resources and generate a private namespace based on Linux namespaces. It will then union the image's read only layers and create a writable layer to provision the file system for the container. By default, the Docker daemon stores its configuration about images, layers, and containers in a directory called *graph driver*.

2 A SHARED IMAGE STORE

The above described architecture is designed for a setup where the daemon is tightly coupled with the local storage of a Docker host but not for a shared storage cluster. However, HPC clusters decouple compute from storage and often lack local disks [3]. Additionally, clusters use *burst buffers* to absorb bursty I/O during scientific workloads. Burst buffers are deployed remotely as a fast storage layer and also accessed via a shared storage layer by multiple nodes [6]. This way of accessing data requires Docker to serve images from shared storage to enable containerized HPC workloads.

Figure 1 shows a naive architecture to enable sharing images in Docker. As shown in the figure, all daemons access the same shared storage, but each daemon has its own private graph driver on the shared storage. Such a design leads to three main problems: (i) large scientific workloads can spawn hundreds of containers simultaneously from the same image to perform parallel computation. This leads to large network overhead due to redundant fetches of the same image; (ii) storage space is wasted because multiple copies of the same image are stored; and (iii) maintaining a single graph driver per daemon is difficult as it quickly becomes unclear which image can/should be garbage collected to free space.

To solve the above problems, we propose to share the graph driver across daemons from a distributed storage layer such as IBM Spectrum Scale or NFS. This prevents redundant fetches, as an image only has to be fetched once, and saves storage space by avoiding redundant copies of an image. In addition, shared images increase the layer reuse rate and open the possibility for lightweight container migration.

While a shared graph driver comes with several benefits, it also poses challenges: (i) as with any shared architecture, synchronization between different components is required to prevent conflicts. For example, multiple daemons may try to pull the same image at the same time or a daemon may try to delete an image that is

SC'17, November 2017, Denver, CO USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 123-4567-24-567/08/06.
https://doi.org/10.475/123_4

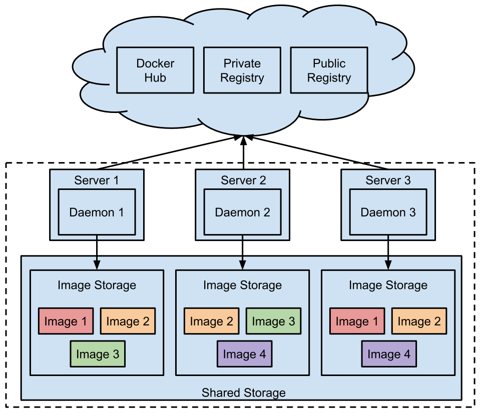

Figure 1: Architecture of a Docker enabled cluster

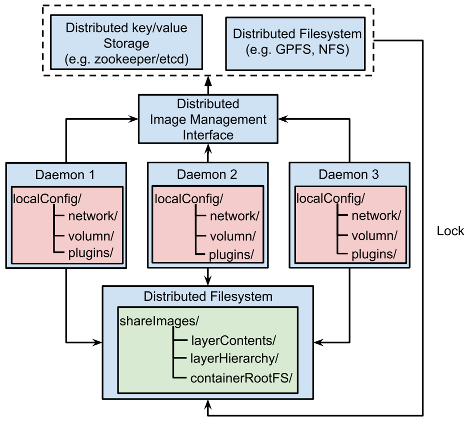

Figure 2: Wharf Architecture

still being used by other daemons; (ii) a shared graph driver can affect the overall performance of the cluster if image data has to be streamed over the network to the daemons; (iii) popular images can result in hot spots and create contention between multiple concurrent read accesses of different daemons.

3 WHARF ARCHITECTURE

To resolve the challenges, we introduce Wharf, a distributed image store for Docker. The architecture of Wharf is shown in Figure 2. Wharf splits the graph driver contents into *global* and *local* state. Global state comprises contents that need to be shared across daemons, i.e. image layers and metadata about images and containers. Wharf stores global state on a distributed file system. Local state is related to the containers running on a single daemon, such as networking or information about attached volumes. This data is stored separately for each daemon, either in its local storage or on a separate location in the distribute file system.

The Docker daemon uses in-memory metadata to cache the global state. To share the graph driver, Wharf synchronizes the in-memory metadata across all daemons via a shared file. Every time a daemon updates its in-memory metadata, it locks this file and flushes the changes. Wharf implements a distributed image management interface to allow to plug in different distributed locking mechanisms. By default, Wharf provides a read/write locking mechanism via the `fcntl` system call, which is part of the POSIX standard and supported by most distributed file systems.

Currently, our prototype of Wharf is based on the above described coarse-grained image locking. The coarse-grained lock allows concurrent reads and exclusive writes to the image store. This prevents conflicts by processing *all* write accesses sequentially. As this is expensive, because only a single write is allowed at a time to the image store, we are planning to implement a fine-grained *layer-based* lock which only blocks writes to the same layer.

4 RELATED WORK

Previous work has studied sharing Docker images. Slacker [4] proposes to lazily load image content from a shared storage backend to reduce the amount of transferred unused image data. However, Slacker does not consider a large-scale environment in which many daemons can compete for accessing few images. Shifter [3] implements its own flat image format to serve images from a distributed file system. It also requires several external dependencies, including MongoDB, Redis, and Celery. In contrast, Wharf keeps the Docker image structure and can run without additional software dependencies. CoMICon [5] introduces a decentralized, collaborative registry design to enable daemons to share images between each other. However, images are still local to individual daemons.

5 CONCLUSION AND FUTURE WORK

In this paper, we explored how to efficiently share Docker images through a shared storage layer to enable running Docker on large-scale HPC clusters. We proposed Wharf, a shared Docker image store that loads images into a distributed file system to prevent redundant fetches and ease image management. We expect highly parallel or high throughput HPC workloads to benefit from it.

A prototype of Wharf is currently under development and we are planning to implement a fine-grained layer locking mechanism. Additionally, we will investigate automatic layer replication mechanisms to mitigate possible read contention on popular images and enable container live migration on the distributed file system. Finally, we will use Wharf with container schedulers such as Kubernetes to evaluate its performance with large HPC applications.

REFERENCES

[1] [n. d.]. Docker. ([n. d.]). https://www.docker.com/.
[2] Theodora Adufu, Jieun Choi, and Yoonhee Kim. 2015. Is Container-based Technology a Winner for High Performance Scientific Applications?. In *APNOMS*.
[3] Richard Shane Canon and Doug Jacobsen. 2016. Shifter: Containers for HPC. In *Cray User Group*. https://cug.org/proceedings/cug2016_proceedings/includes/files/pap103.pdf.
[4] Tyler Harter, Brandon Salmon, Rose Liu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Container. In *FAST*.
[5] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. 2017. CoMICon: A Cooperative Management System for Docker Container Images. In *IC2E*.
[6] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An Ephemeral Burst-buffer File System for Scientific Applications. In *SC*.
[7] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. 2013. Performance Evaluation of Container-based Virtualization for High Performance Computing Environments. In *PDP*.