Wharf: Sharing Docker Images in a Distributed File System

Chao Zheng
University of Notre Dame
czheng2@nd.edu

Lukas Rupprecht
IBM Research–Almaden
lukas.rupprecht@ibm.com

Vasily Tarasov
IBM Research–Almaden
vtarasov@us.ibm.com

Douglas Thain
University of Notre Dame
dthain@nd.edu

Mohamed Mohamed
IBM Research–Almaden
mmohamed@us.ibm.com

Dimitrios Skourtis
IBM Research–Almaden
dimitrios.skourtis@ibm.com

Amit S. Warke
IBM Research–Almaden
aswarke@us.ibm.com

Dean Hildebrand
Google
hildebrand@google.com

ABSTRACT

Container management frameworks, such as Docker, package diverse applications and their complex dependencies in self-contained *images*, which facilitates application deployment, distribution, and sharing. Currently, Docker employs a shared-nothing storage architecture, i.e. every Docker-enabled *host* requires its own copy of an image on local storage to create and run containers. This greatly inflates storage utilization, network load, and job completion times in the cluster. In this paper, we investigate the option of storing container images in and serving them from a distributed file system. By sharing images in a distributed storage layer, storage utilization can be reduced and redundant image retrievals from a Docker registry become unnecessary. We introduce Wharf, a middleware to transparently add distributed storage support to Docker. Wharf partitions Docker's runtime state into *local* and *global* parts and efficiently synchronizes accesses to the global state. By exploiting the layered structure of Docker images, Wharf minimizes the synchronization overhead. Our experiments show that compared to Docker on local storage, Wharf can speed up image retrievals by up to 12×, has more stable performance, and introduces only a minor overhead when accessing data on distributed storage.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Information systems →** *Cloud based storage*;

KEYWORDS

Container, Docker, Distributed File System

ACM Reference Format:
Chao Zheng, Lukas Rupprecht, Vasily Tarasov, Douglas Thain, Mohamed Mohamed, Dimitrios Skourtis, Amit S. Warke, and Dean Hildebrand. 2018. Wharf: Sharing Docker Images in a Distributed File System. In *Proceedings of ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC'18)*, 12 pages.
https://doi.org/10.1145/3267809.3267836

1 INTRODUCTION

Operating system containers [28, 39] are rapidly becoming a popular solution for sharing and isolating resources in large-scale compute clusters. As an example, Google reports to run all of its applications in containers, resulting in more than two billion launched containers per week [19]. Furthermore, all major cloud vendors have added container services to their offerings [1, 3, 9, 11].

Containers are attractive as they provide more lightweight isolation compared to conventional virtual machines [27]. Additionally, they abstract the application from the infrastructure, which allows developers to focus on functionality rather than on deployment [18]. This facilitates a *micro-service* oriented compute model, in which individual computational tasks or functions are executed in their own, separate containers [38].

While the Linux kernel has included support for containers for over a decade, they have only recently received extensive attention. This is due to the rise of container management and orchestration frameworks such as Docker [6], CoreOS [4], and Kubernetes [9], which drastically simplify the creation, execution, and sharing of containers. Docker provides a set of interfaces, implemented by the Docker *daemon*, which allow users to conveniently package an application and all its dependencies in *images* and to start containers from these images. To facilitate image storing and sharing, Docker provides a *registry* service which acts as an image repository. Daemons can push/pull images to/from the registry and run them locally.

In Docker, each daemon process is *shared-nothing*, i.e. it pulls and stores images in local storage and starts containers from the local copies. This design introduces a tight coupling between the daemon and the node's local storage. Considering the rate at which containers start in cloud environments [19] and the low startup latencies required for micro services [38], this tight coupling leads to four major problems: (i) if containers start from the same image on different nodes, the image exists on all of the nodes. This wastes storage space; (ii) large-scale applications often consist of thousands of identical tasks, which are distributed across a large number of nodes. To run such an application, each node has to pull the same image from a Docker registry to its local storage. This wastes network bandwidth and increases the application startup time; (iii) while only 6.4% of the image data is read by containers on average [21], each node pulls the *complete* image from a registry. This further wastes storage space and network bandwidth; (iv) some clusters, e.g., in HPC environments, only offer shared storage while compute nodes themselves are diskless [20, 42]. This prohibits running containers in such environments.

To solve these problems and exploit the benefits of a highly scalable shared storage layer [30], we argue that the architecture of the container runtime should be able to provide a way of storing images in a shared file system, and each daemon should only maintain the minimum necessary private state. Previous work has dealt with this problem from the perspective of virtual machines [34, 35, 40], proposing to locally cache the bare minimum an image needs to boot and thereafter load data on demand. However, these approaches cannot readily be applied to containers as they do not consider the synchronization during pushing and pulling of images. More recent work has focused on speeding up container startup times by sharing storage across registry and daemons [21]. However, parallel image accesses at scale are not considered.

In this paper, we present Wharf, a system to efficiently serve container images from a shared storage layer such as NFS [31] or IBM Spectrum Scale [12]. Wharf enables distributed Docker daemons to *collaboratively* retrieve and store container images in shared storage and create containers from the shared images. Wharf significantly decreases network and storage overheads by holding only one copy of an image in central storage for all daemons. Designing Wharf requires careful attention to the semantics of operations on container images to ensure consistency, scalability, and performance.

Wharf exploits the structure of Docker images to reduce the synchronization overhead. Images consist of several *layers* which can be downloaded in parallel. Wharf implements a fine-grained *layer lock* to coordinate access to the shared image store. This allows daemons to pull different images and different layers of the same image in parallel and therefore increase network utilization and avoid excessive blocking. Wharf splits the data and metadata of each daemon into *global* and *local* state to minimize the necessary synchronization. Additionally, using the layer lock, Wharf ensures that only consistent layer state can be seen by each daemon and prevents the entire cluster from failing when single daemons fail.

Wharf is designed to be transparent to users and allows to reuse existing Docker images without any changes. Wharf is independent of the distributed storage layer and can be deployed on any storage system, which provides POSIX semantics. Wharf currently supports two common Docker copy-on-write storage drivers— `aufs` [2] and `overlay2` [14]. Furthermore, though we implemented Wharf for Docker, the presented design is applicable to other container management frameworks [13, 25].

After introducing the relevant background on Docker (§2), we make three contributions:

(1) We analyze the implications of porting Docker's storage model to distributed storage (§3);
(2) We design Wharf, a shared image store for Docker that enables storing images in a distributed file system (§4);
(3) We implement Wharf and apply fine-grained layer locking, a synchronization mechanism which exploits the layered structure of Docker images, to reduce synchronization overhead (§5).

We evaluate Wharf using a combination of static and runtime benchmarks on a 20-node Amazon EC2 cluster. We show that Wharf can reduce image pulling times by a factor of up to 12× compared to Docker on local storage, while only introducing a small overhead of 2.6% during container execution due to remote data accesses (§6). Additionally, we show that Wharf can significantly reduce networking and storage resource consumption. After discussing related work in §7 we conclude in §8.

2 DOCKER CONTAINERS

We start by discussing the basic concepts of Docker (§2.1) and how images are constructed and distributed (§2.2). We then explain container creation in more detail (§2.3).

2.1 Docker Architecture

Docker is a framework to simplify managing and deploying containers. It consists of three main parts: the Docker daemon, a Docker client, and the Docker registry.

The **Docker Daemon** is responsible for starting and stopping containers, creating images, and storing images in and retrieving images from the registry. The daemon runs on a Docker-enabled host machine and accepts commands from Docker clients.

The **Docker Client** is used to communicate with the daemon and manage its containers. Communication is implemented via a RESTful API and the interface includes commands such as `run` and `stop` to start/stop a container, `pull` to retrieve images from the registry, or `rm` to remove a container after it stops. Commonly, Docker client and daemon run on the same machine.

The **Docker Registry** is Docker's image repository. Users can create images locally and *push* them to a public registry such as Docker Hub [7] or to a private registry. Images are versioned in the registry and can be *pulled* to Docker hosts to start a corresponding container.

2.2 Images and Layers

Every Docker container is created based on a *container image*. An image contains all necessary files to run the container and consists of multiple read-only *layers*. A layer is a set of files which represents a part of the container file-system tree. Layers are stacked on top of each other and joined to expose a single mountpoint via a union file system such as AUFS [2] or OverlayFS [14]. An image typically consists of several layers with sizes ranging from several KB to hundreds of MB [16].

The layered image structure allows sharing layers across containers. Docker identifies layers via a hash over their contents. When multiple containers are created from images which contain identical layers, Docker does not duplicate those layers. Instead, the different containers read from the same copy of the layer. This saves storage space and improves the startup times of containers.

Images can be shared between Docker hosts via the registry. When an image is pushed to the registry, its corresponding layers are compressed and archived as tarballs and then uploaded to the registry. Additionally, the registry creates an image *manifest* that references the layers associated with the image so that clients can retrieve all required layers when pulling an image. While the registry offers a convenient, centralized way of sharing images, it can become a bottleneck when a large number of images is pulled at the same time.

2.3 Container Creation and Graph Drivers

When the Docker daemon receives a request for creating a new container, it executes the following steps. It first checks whether the required image is already available locally. If not, it retrieves it from the registry by first reading the manifest and then downloading and extracting the corresponding layers to local storage. If the daemon detects that one of the layers already exists locally, it will not download that specific layer. By default, the daemon downloads three layers in parallel.

Once the image is retrieved, the daemon will create the container. Therefor, it first unions the read-only layers at a single mountpoint and then creates a *writable* layer on top of it. Any changes made to the container's root file system are stored in the writable layer via a *copy-on-write* (COW) mechanism. That means that before modifying a file from a read-only layer, it is copied to the writable layer and all changes are made to the copy. Once the container exits, the writable layer is typically discarded.

The exact way of creating the union mount and performing COW is defined by the Docker *graph driver*. Docker supports several different graph drivers that vary considerably in performance, portability, and compatibility with kernels and system software [41]. The most widely used graph drivers can be divided into two broad categories: overlay drivers and specialized drivers.

Overlay drivers are based on overlay file systems, e.g., *AUFS* [2] or *OverlayFS* [14]. These file systems layer on top of existing file systems and intercept operations on files and directories. When used, multiple read-only file systems are merged into a single logical file system and any changes are written to a single writable file system via COW.

Specialized drivers apply the same COW principle but use the special, native capabilities of the file system or block device to implement the layer model and COW. These drivers either require special block devices or a block device that is formatted with a specialized file system. `Devicemapper` [5] and `btrfs` [37] are examples of specialized drivers.

3 DOCKER ON DISTRIBUTED STORAGE

In this section, we discuss how to run Docker on distributed storage. We first describe the naive approach, which is inherently supported by Docker, and discuss its shortcomings (§3.1). We then introduce the design goals for a native, efficient integration of Docker with distributed storage (§3.2).

3.1 Naive Solution

In its current deployment mode, Docker assumes that local storage is used exclusively by a single daemon. The daemon uses this storage to store image and layer data, and metadata on the state of the daemon and its running containers. Hence, two daemons cannot operate on the same storage as they would override each other's state. This does not only hold for multiple daemons accessing the same shared storage but also for multiple daemons on the same host, accessing the same local storage.

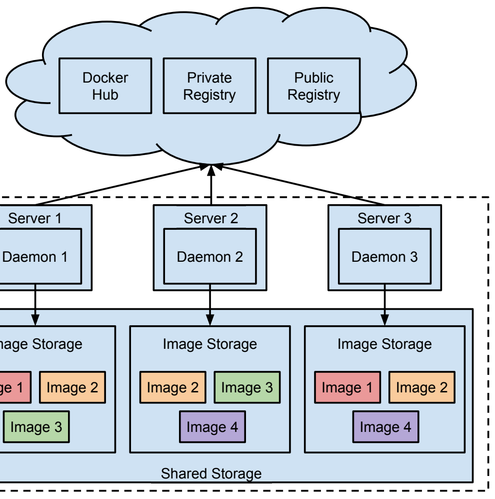


Figure 1: Docker on distributed storage, naive solution

The simplest way of enabling Docker to run on distributed storage is to partition the distributed storage. Each daemon receives its own partition where it can store all of its state and image data (see Figure 1). The partitioning can be physically done as part of the storage or simply by assigning each daemon a different directory. This approach is supported by Docker today without any additional implementation effort, as long as the underlying storage exposes a POSIX interface.

While this design is simple, it comes with several shortcomings: (i) it does not solve the problem of redundant pulls during a large-scale workload. Each daemon still has to pull an image separately and store it in its partition. This leads to both network and storage I/O overhead and increased container start-up latencies; (ii) it over-utilizes storage space because a copy of the image has to be stored for each daemon. As image garbage collection is still challenging, it is a common problem for Docker users to run out of disk space due to unused images/containers not being removed. This can quickly lead to high storage utilization; (iii) image pull latencies can be much higher—up to 4× longer in our experiments—compared to Docker on local storage, due to extra data transfers between the daemon and the distributed storage.

3.2 Design Goals

The above issues suggest that the naive approach is not sufficient to successfully integrate Docker with a distributed storage layer. Based on the shortcomings of the naive solution, we identify five main design goals for a native integration approach.

1) Avoid redundancy. To efficiently run on distributed storage, a native solution needs to avoid redundant data transfers. If an image is required by multiple daemons, it should only be retrieved and stored once. Additionally, all daemons should be aware of the existing layers. If an image requires layers that are already present in the distributed storage, only the missing layers should be pulled.

2) Collaboration. Docker daemons should work together collaboratively to ensure correct and efficient operation. For example, images can be pulled in parallel by multiple daemons to speed up pulling as more resources are available. Furthermore, coordination is necessary to prevent individual daemons from deleting images if they are still in use by other daemons.

3) Efficient synchronization. When multiple daemons access the same storage, locking and synchronization between the daemons is necessary to avoid race conditions. To minimize the impact on container startup times, synchronization should be lightweight and exploit Docker's layered image structure, i.e. accesses should be synchronized at layer rather than image granularity.

4) Avoid remote accesses. As data is now accessed remotely from the shared storage layer, additional latency is induced for read/write operations. This can impact both Docker client calls and the workload running inside a container. In the worst case, when connectivity drops due to network failures, the entire container can stall until the connection is restored. Hence, the amount of necessary remote accesses should be minimized.

5) Fault Tolerance. The system as a whole must stay operational even if one or several individual daemons fail, i.e. a failing daemon should not corrupt the global state and pending operations should be finished by the remaining daemons.

4 WHARF

We now introduce Wharf, an extension of Docker for distributed storage, which meets the above described design goals. Wharf allows Docker daemons based on the same distributed file system to collaboratively download and share container layers and thereby, reduce storage consumption and network overhead. We first discuss the overall architecture of Wharf (§4.1) which addresses design goal 1) and then explain fine-grained layer locking (§4.2) to implement design goals 2) and 3). We describe Wharf's local write optimization (§4.3) to address design goal 4) and finally discuss Wharf's approach to fault tolerance (design goal 5) in §4.4.

4.1 System Architecture

The core idea of Wharf is to split the graph driver contents into *global* and *local* state and synchronize accesses to the global state. Global state contains data that needs to be shared across daemons, i.e. image and layer data, and static metadata like layer hierarchies and image manifests. It also includes runtime state, e.g., the progress of currently transferred layers, relationships between running containers, and pulled images and layers. Global state is stored in the distributed file system to be accessible by every daemon.

Local state is related to the containers running under a single daemon, such as network configuration, information on attached volumes, and container plugins, such as the Nvidia GPU plugin, which simplifies the process of deploying GPU-aware containers. This data only needs to be accessed by its corresponding daemon and hence, is stored separately for each daemon. Local state can either be stored on a daemon's local storage or on a separate location in the distributed file system.

Splitting the graph driver content into global and local state ensures that all image-related information is stored once and not duplicated across different daemons. This addresses design goal 1) as no redundant information is retrieved or stored.

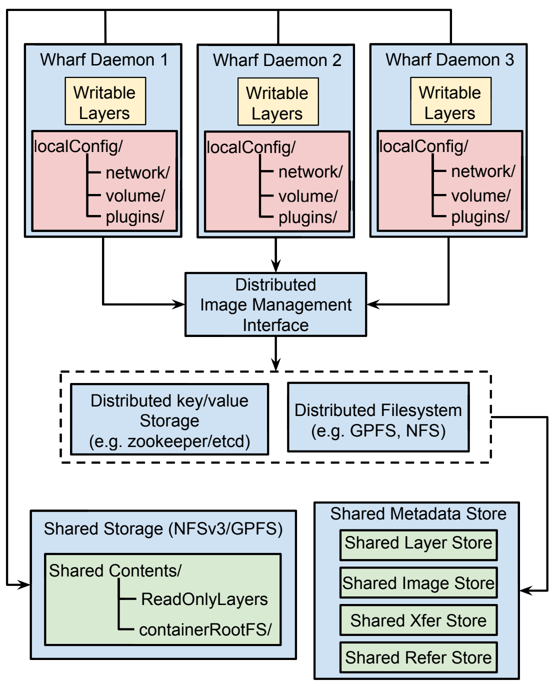


Figure 2: Wharf architecture

The architecture of Wharf is shown in Figure 2. It consists of three main components: Wharf daemons, an image management interface, and a shared store for data and metadata.

1) Wharf daemons. Wharf daemons run on the individual Docker hosts and manage their own local state for their local containers. When a Wharf daemon receives a request to create a container, it sets up the container root file system such that the writable layer is stored at a private, non-shared location and the read-only layers are read from the distributed file system. Wharf daemons exchange information via updating the global state on the distributed file system but do not communicate with each other directly.

2) Image management interface. The image management interface is the gateway through which Wharf daemons access the shared global state. It ensures that concurrent accesses are synchronized by locking the parts of the global state which are updated by a Wharf daemon. This keeps the global state consistent. The image management interface offers different ways of implementing a distributed lock, e.g., using zookeeper [23] or etcd [8], if these systems are available. If supported by the underlying file system, Wharf can also rely on file locking (`fcntl()` interface) for access synchronization. This approach is highly portable as it does not require any additional external services.

3) Shared store. The shared store hosts all global state and is split into two parts. The *Shared Content Store* contains the data of the readonly layers and the root file systems of the running containers. The *Shared Metadata Store* holds the metadata on which

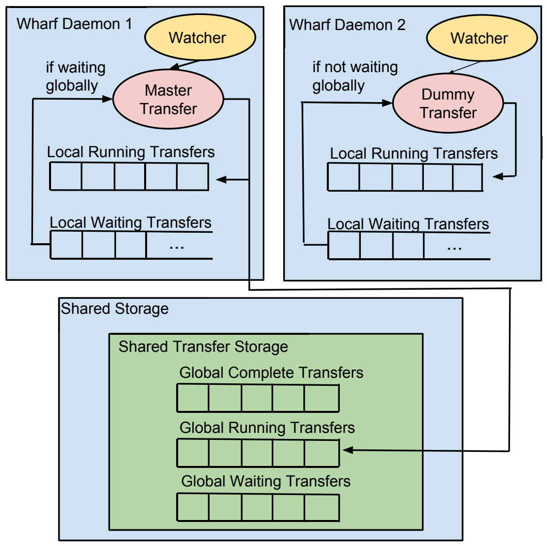


Figure 3: Implementation of concurrent layer pulling

layers and images exist (Shared Layer Store and Shared Image Store), are currently being pulled (Shared Transfer Store), and currently being referenced by containers (Shared Reference Store). Each of these metadata stores can be locked individually. The metadata is concurrently readable by multiple daemons, but can only be updated by one daemon at a time. Any attempt by a daemon to access a layer from the Shared Content Store requires it to first read the Shared Metadata Store and check the status of the layer, i.e. whether it already exists, is currently being pulled, or is referenced by a running container. As write access to the metadata is protected, daemons are guaranteed to have a consistent view of the Shared Content Store.

4.2 Layer-based Locking

Design goals 2) and 3) state that locking should be efficient and daemons should collaborate with each other when retrieving or deleting images. Wharf achieves this by exploiting the layered structure of Docker images and implements *layer-based locking* as part of the image management interface. Layer-based locking allows Wharf daemons to write to the shared store by adding single layers rather than an entire image. Read accesses do not have to be synchronized. This means that multiple Wharf daemons

(1) can collaboratively pull different layers of an image in parallel,
(2) only lock a small portion of the shared store such that unrelated, parallel operations are unaffected,
(3) do not pull or remove the same layer at the same time to avoid redundant work,
(4) and can read data from the same layer in parallel.

Figure 3 shows how an image is pulled in parallel under layer-based locking. In Docker, a daemon can start multiple transfers and each transfer is responsible for pulling one layer. Each transfer has two states: running and waiting. By default, each daemon has 3 threads for pulling layers in parallel.

When a daemon receives a layer pulling request, it will create a *Master Transfer* and register it in a local map data structure (*Local Running Transfers*). If all pulling threads are busy, the daemon will add the request to the *Local Waiting Transfers* queue for later processing, once a thread becomes available. If another client tries to pull a layer that is already being pulled by the daemon, the daemon will create a *Watcher* to report the pulling progress back to the client.

In Wharf, pulls not only have to be coordinated across multiple local clients but also across multiple remote daemons (and their clients). Therefore, Wharf uses three additional map data structures, *Global Running Transfers*, *Global Waiting Transfers*, and *Global Complete Transfers*, which are serialized and stored in the *Shared Transfer Store* on the shared storage. Write access to the Shared Transfer Store is protected by a lock.

Each time a Wharf daemon tries to pull a layer, it will first check the Global Running Transfers map for whether the requested layer is already being pulled by another daemon. If not, it will create the Master Transfer and add it to both the Local and Global Running Transfers. Waiting transfers are also registered in both the local and global data structures. Daemons are responsible for removing waiting transfers from the global map once they start retrieving the corresponding layer.

If a Wharf daemon finds that a layer is already being pulled by another daemon, it will create a *Dummy Transfer* and add it to the list of running transfers. The Dummy Transfer is a placeholder to transparently signal the client that the layer is being pulled without actually occupying a thread. Each daemon runs one background thread to periodically check, whether a Master Transfer for a corresponding Dummy Transfer has finished. Therefor, it polls the Global Complete Transfers map where finished Master Transfers are registered.

By using layer-based locking, Wharf accelerates image transfers for large images, which consist of multiple layers and are required by many daemons. It also avoids redundant layer pulling and increases the layer usage as it can now be shared by containers across different daemons. To avoid potential conflicts caused by deleting layers that are currently in use by other daemons, Wharf uses a global reference counter for each layer. A layer can only be deleted from the Global Layer Store if its reference counter is 0.

4.3 Local Ephemeral Writes

In a typical Docker deployment the root file system of a container is *ephemeral*, i.e. the changes written to the file system while the container is running are discarded after the container stops. This translates to the removal of the writable layer by the underlying storage driver. User applications frequently create substantial amounts of intermediate data in the root file system which inflates the writable layer. Furthermore, overlay storage drivers, like `overlay2` and `aufs`, copy the whole file to the writable layer even when only a small portion of the file is modified. This leads to significant write amplification at the file system-level compared to the user writes.

Docker typically stores writable and readable layers in the *same* file system. In case of a distributed file system, this translates to a large quantity of ephemeral container data being transferred across the storage network, adding network and remote storage server overhead. Unlike Docker, Wharf stores writable and readable layers in two *separate* locations. Specifically, Wharf puts the writable layers of running containers on locally attached storage while read-only layers are stored in the shared storage so that any daemon can access it. This design addresses design goal 4) by decreasing the amount of writes to the distributed file system.

4.4 Consistency and Fault Tolerance

To achieve design goal 5), Wharf requires a *strong consistency model* for its global state and needs to be able to deal with daemon crashes.

Consistency. As mentioned in §4.2, a fine-grained layer lock is used to synchronize the global image and layer state between daemons. While the global state can be read by multiple daemons simultaneously, it can only be written by one daemon at a time. Additionally, global metadata cannot be cached locally at daemons to prevent them from operating on stale data. Exclusive writes combined with no caching of state provides the necessary strong consistency of the global state across daemons and makes sure that daemons always operate on the same metadata view.

When a daemon performs an operation that requires updating image data, e.g., pulling a new image or layer, it proceeds in three steps: i) It goes through a *metadata phase* in which it locks the necessary part of the global state and registers its actions, e.g., pull layer l_1, if no other daemon is already performing the same action. It then releases the lock; ii) it then continues with a *data phase* during which the actual data is retrieved; iii) finally, if the operation was successful, it again acquires the necessary metadata lock and updates the metadata. As the data phase is always preceded by a metadata phase, no two daemon can perform the same action twice.

Fault Tolerance. Wharf needs to deal with two types of failures: First, Wharf needs to handle the case of a daemon crash while that daemon is still holding a lock. In such a case, the entire system can be stalled if the lock is not released correctly. To avoid such a situation, Wharf uses lock timeouts after which any lock will be released automatically. As daemons only need to acquire a lock to access metadata, the periods during which a lock is required are short and hence, timeouts can be set to low values (e.g., 1 s).

Second, Wharf needs to handle daemon crashes during data phases. If a daemon crashes while pulling a layer, the corresponding image (and all other images that depend on that layer), will never finish pulling. To continue downloading an image after a daemon crashes, Wharf daemons use heartbeats. Heartbeats are periodically sent and the last heartbeat timestamp is stored with the transfer in the Global Running Transfers map. This allows other daemons to check, whether a daemon is still pulling its layer or has crashed. In case of a crash, a new daemon can continue the transfer.

5 WHARF IMPLEMENTATION

Next, we describe our implementation of Wharf in Docker. We first describe how Wharf is able to share global state between the individual daemons (§5.1) and then explain how images can be pulled collaboratively in parallel (§5.2). The described implementation adds approximately 2,800 lines of code to the Docker code base. To run Wharf, the user only has to run the Docker daemon with the `shared-root` parameter set. All other commands can be used without any further changes for the user.

5.1 Sharing State

To efficiently share the global state between daemons, Wharf has to deal with two main problems: distributed synchronization for access to the global state and in-memory caching of state in individual daemons.

Docker uses three main `structs` which store the global state: 1) `LayerStore` for information on available layers (a list of readonly and writable layers); 2) `ImageStore` for information on local images (storage backend and image metadata); and 3) `ReferenceStore`, which includes the references to all available images. Each daemon keeps an in-memory copy of those data structures and can generate them by reloading the directory that holds the persistent state of the daemon. To accommodate multiple concurrent clients, a Docker daemon protects its in-memory state via a mutex.

Wharf extends the design of Docker by serializing the above data structures and to make them available to all daemons, it stores them in the shared storage. It uses a distributed locking mechanism to synchronize accesses. Read accesses can happen concurrently and only require a read-only lock whereas write accesses require an exclusive lock on the part of the state that should be updated.

By default, Wharf uses the `fcntl` system call to implement the locking. As `fcntl` is supported by most distributed file systems, this approach can be used out-of-the-box without extra software and library dependencies other than what Docker requires. Via Wharf's Image Management Interface, it also allows users to replace the default file-based locking with, e.g., an in-memory key/value store.

As multiple daemons can now update the global state, the in-memory state of an individual daemon can become invalid. Hence, before reading from their in-memory state, daemons have to check whether the global state has been updated. To ensure daemons always have the latest version of metadata before processing any operation, Wharf applies a lazy update mechanism, which only updates the cache of an individual daemon if the operation requires metadata access. Wharf updates its in-memory data structures by deserializing the binary files from the distributed storage and overwrite its in-memory data with the retrieved data.

5.2 Concurrent Image Retrieval

When pulling an image concurrently, Wharf daemons need to collaborate such that no redundant data is pulled. To enable multiple daemons to pull layers concurrently, Wharf extends the layer transfer model of Docker by adding two new components: the `SharedTransferManager` and the `SharedTransferStore`, to communicate between daemons. The `SharedTransferManager` is the distributed version of Docker's `TransferManager` struct while the `SharedTransferStore` is part of the global state and also serialized to the shared storage. Figure 4 describes in detail how an image is pulled in parallel by multiple Wharf daemons if they require the same image at the same time.

Each daemon starts by fetching the image manifest and extracting the layer information. The daemons will then dispatch a configurable number of threads to pull the image layers in parallel. A daemon will first check if the layer already exists. If not, it will check whether it is already being pulled by one of its local threads.

In case another thread is already pulling the layer, the daemon will generate a watcher to monitor the progress of the transfer.

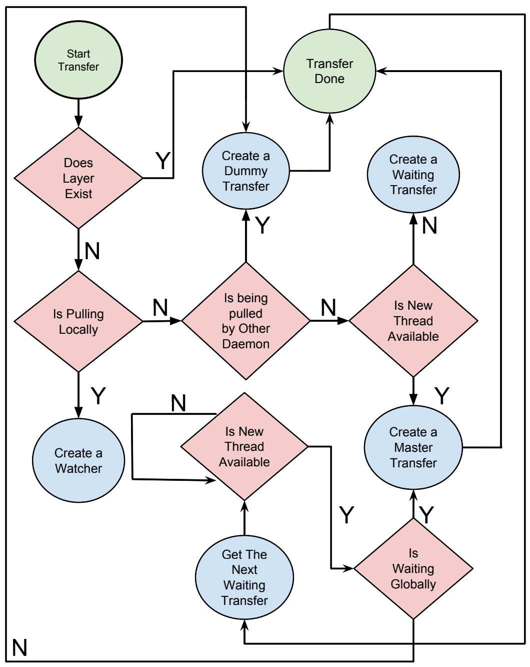


Figure 4: Layer pull procedure in Wharf

Otherwise, the daemon will check if the layer is currently being pulled by another daemon on a different host. This information can be obtained via checking the `SharedTransferStore`. If another daemon is found, a *dummy transfer* is generated to monitor the *master transfer*.

If no other daemon is pulling the layer and not all of the daemon's local threads are busy, the daemon will generate the master transfer and dispatch a thread to start downloading the layer. Once the master transfer is complete, the daemon will update the `SharedTransferStore`. The `SharedTransferStore` is accessible by all daemons, thus other daemons with allocated dummy transfers can learn about the completion of the matching master transfer.

In case all local threads are busy pulling other layers, the daemon creates a waiting transfer and pushes it to a local and global waiting transfer queue. Once a local transfer finishes, the daemon will take the next waiting transfer from the local queue and if it is still on the global queue, start the download. Otherwise, that layer is already being pulled by another daemon.

5.3 Graph Drivers and File Systems

As Wharf does not have direct access to the underlying block storage, its applicability is limited to the category of overlay drivers (see §2.3). Currently, Docker supports two overlay drivers: `aufs` and `overlay2`[1]. Conceptually, Wharf is compatible with both graph drivers and any POSIX distributed file system. However, running `overlay2` drivers with a distributed file system is less common and, therefore, not well tested. We found that some combinations are currently not operational. We tried two different file systems with Wharf: NFS and IBM Spectrum Scale [12].

NFS. Wharf can work with NFS and both the `aufs` and `overlay2` graph drivers. However, we observed a problem when using NFSv4 and `overlay2` due to the `system.nfs4_acl` extended attribute, which is set on NFS files. OverlayFS is not able to copy the extended attribute to the *upper file system*, i.e. the file system which stores the changed files (ext4 in our case). We believe this is due to incompatibilities between NFSv4 ACLs and the ACLs of the upper file system. While it is possible to mount NFSv4 with a `noacl` option, we found that this is not supported in our Linux distribution.

Spectrum Scale. We also were able to run Wharf on top of IBM's Spectrum Scale parallel file system. While the `aufs` driver was working correctly, we again experienced problems using `overlay2`. We observed that Docker tries to create the upper file system for OverlayFS on Spectrum Scale, even though, local writes to ext4 are configured. This leads to an error ("`filesystem on '/path' not supported as upperdir`"). We currently do not know the exact reason for this behavior, especially because we were able to manually create an OverlayFS mountpoint on Spectrum Scale, but are planning to investigate the problem in the future.

As OverlayFS is part of the mainline Linux kernel and hence, offers better portability, we used the combination of *overlay2* and NFSv3 for our experiments with Wharf.

6 EVALUATION

To evaluate Wharf, we compare it to two other setups: (i) **DockerLocal**, which uses local disks to store the container images, the corresponding layers, and the writable layers; and (ii) **DockerNFS**, which uses NFS as a storage backend but separate directories to store the data for each daemon.

We run our experiments on an Amazon EC2 cluster using 5 to 20 t2.medium instances. Each instance has 2 vCPUs, 4 GB RAM, and 32 GB EBS disks and runs Ubuntu Linux 16.04 with kernel version 4.4.0-1048-aws. Wharf is based on Docker Community Edition 17.05 and we use this version in all of our experiments. To avoid network speed variations between a public Docker registry and the daemons, we set up a private registry on one t2.medium instance.

6.1 Pull Latency and Network Overhead

We start by running a set of microbenchmarks to measure the impact of Wharf on image pull latencies. We consider five dimensions: (i) the number of layers; (ii) the size of each layer; (iii) the number of files per layer; (iv) the network bandwidth between the registry and the daemons; and (v) the number of daemons. We set up five experiments, and in each, vary one of the above dimensions while fixing the others. We use the default number of 3 concurrent pulling threads for each daemon.

6.1.1 Varying Number of Layers. To investigate how the granularity of images influences the pulling latencies for the three different setups, we pull an image that has a varying number of layers ranging from 2 to 40. Each layer contains one file. We use 10 Docker hosts and fix the image size to 2 GB. We measure the average pull latencies of daemons. The first row of figure 5 presents the results.

Wharf shows the shortest average pull latencies (73 s) and lowest overheads for both external (to the registry) and internal (to NFS)

[1]While there is an older driver based on OverlayFS, Docker recommends to use the new `overlay2` driver.

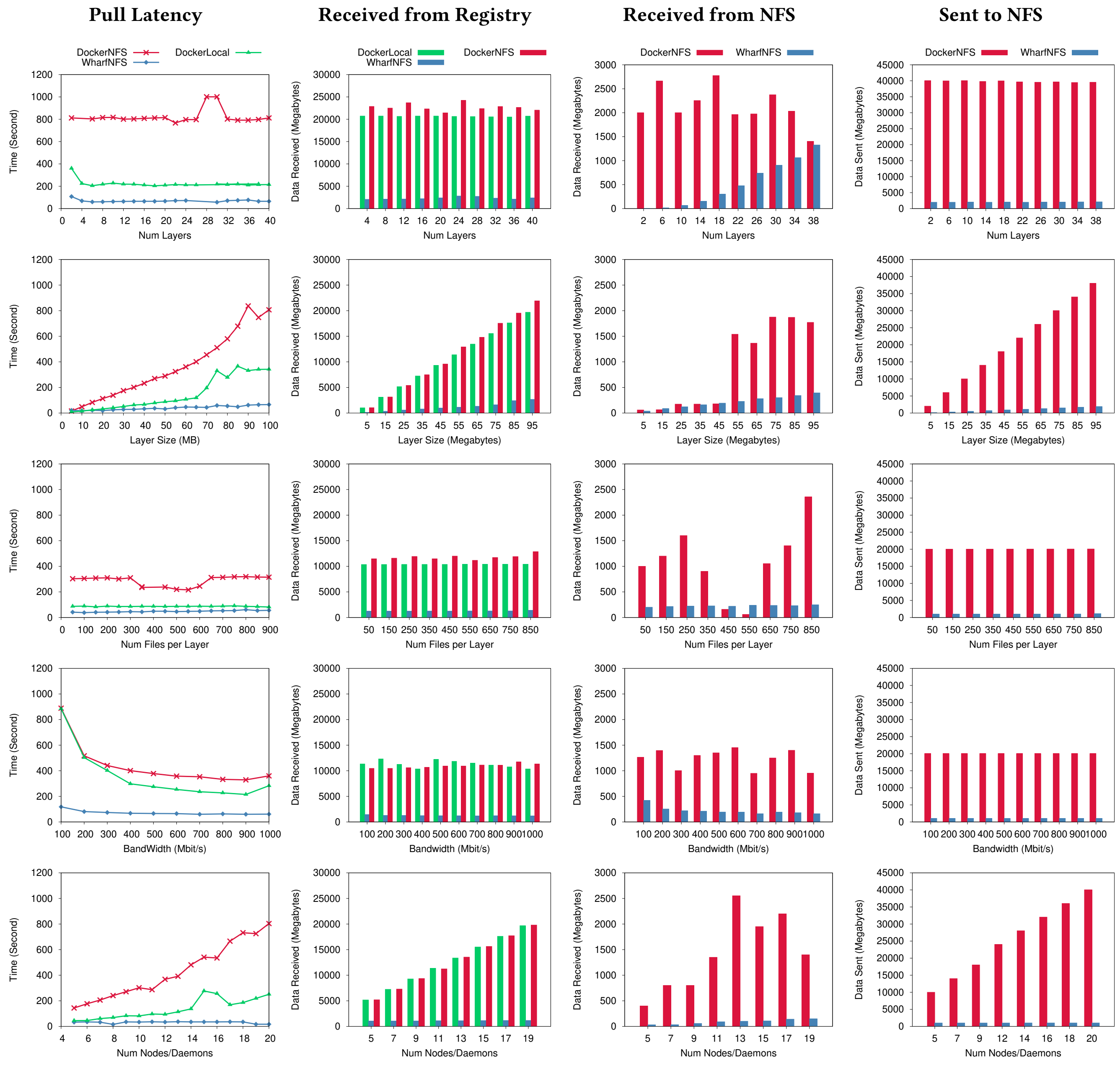
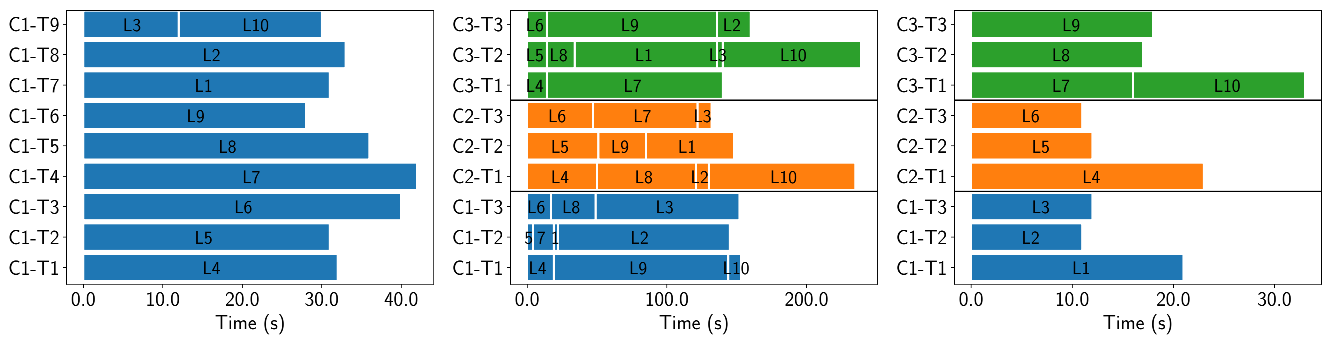


Figure 5: Pull latencies and network performance

The figures show image pull latencies and network utilization for different configurations. From the first row to the last row: (i) 10 daemons pull 20 images each, each image size is 2 GB and layers per image number vary from 2 to 40; (ii) 10 daemons pull 20 images with each image containing 20 layers and the size of each layer varies from 5 MB to 100 MB; (iii) 10 daemons pull 18 images and each image has 20 layers with 50 MB per layer and 50–900 files per layer; (iv) 10 daemons pull 20 images from a local registry with the egress network bandwidth varying from 100 Mbps to 1000 Mbps; (v) 20 images are pulled by a cluster with a varying number of nodes, ranging from 5 to 20.

network traffic. During each pull operation, all Wharf daemons combined receive on average 2128 MB from the registry, 532 MB from the NFS server, and sent 2049 MB to the NFS server. For a small number of layers, we can see a slight increase in pulling times as in those cases Wharf cannot exploit the available parallelism.

The performance of DockerNFS is limited due to the data transfer between each daemon and the NFS server. It has the longest average



(a) Single Docker daemon with 9 pulling threads

(b) Three Docker daemons with 3 pulling threads each

(c) Three Wharf daemons with 3 pulling threads each

Figure 6: Time diagram of layer pulls per client per thread. Every rectangle represents a pull of a single layer. C stands for *client* and T for *thread*. The layer identifiers L are placed on top of their corresponding rectangles.

pull latencies (953 s) and highest external and internal network traffic, receiving/sending an order of magnitude more data from the registry/to the NFS server compared to Wharf.

DockerLocal generally performs worse than Wharf, due to the fact that when all daemons are redundantly pulling the same image, the network link to the registry becomes a bottleneck. On the other hand, DockerLocal performs significantly better compared to DockerNFS as the retrieved image data does not have to be sent over the network again. Similar to Wharf, we also observe a slight increase in pulling times for DockerLocal when the number of layers is low.

6.1.2 Varying Layer Size. Next, we vary the size of each layer from 5 to 100 MB to analyze the impact of different image sizes. We fix the number of layers for our test image at 20 and again use 10 Docker hosts to pull the image in parallel. The results are shown in the second row of Figure 5.

For all three setups, we observe an increase in pull latency for larger layer sizes. This is expected as more data needs to be pulled as the image size increases. Because Wharf only has to pull the image once, its pull latency only increases by a factor of 1.1× overall while DockerNFS and DockerLocal increase by 16× and 7×, respectively. Due to the same reason, the network traffic for DockerNFS and DockerLocal grows significantly faster compared to Wharf.

When looking at small image sizes, we observe that Wharf adds a small overhead compared to DockerLocal and DockerNFS. For a layer size of 5 MB, Wharf takes 21 s to pull the image while DockerLocal takes 9 s and DockerNFS takes 16 s. This is due to the remote accesses and the additional synchronization in Wharf. When moving to a layer size of 10 MB, the overhead disappears and Wharf performs similarly to DockerLocal (18 s and 15 s respectively) while outperforming DockerNFS (49 s).

6.1.3 Varying Number of Files. In early tests with NFS, we observed large pulling latencies due to unpacking the compressed layer tarballs. This is because NFS was using *synchronous* communication by default, i.e. the NFS server replies to clients only after the data has been written to the stable storage. To mitigate this effect, we configured NFS to use *asynchronous* communication. To see if the number of files in the tarball can affect pulling times even in asynchronous mode, we vary the number of files per layer from 50 to 900 while fixing the number of layers at 20 and the total image size at 2 GB.

As shown in the third row of Figure 5, the pull latencies of DockerLocal and Wharf are close and are unaffected by the file size. Consistent with previous results, DockerNFS performs worse than the two but also does not show any significant variation due to the number of files per layer.

6.1.4 Varying Network Bandwidth. The default bandwidth between our private registry and daemons is between 1 Gbps and 10 Gbps (as per AWS specification). However, in practice, when connecting to a public registry via a wide area link, bandwidths can vary and throughput can drop significantly. To explore how network bandwidth affects the system, we use the Linux tool `tc` to vary the egress bandwidth of the registry node from 100 Mbps to 1 Gbps and measure the pull latencies. The pulled image consists of 20 layers of 50 MB each and we use 10 daemons to retrieve the image in parallel. We do not vary the bandwidth to the NFS server to simulate a realistic scenario in which distributed storage is cluster-local and available via a fast interconnect. The fourth row of Figure 5 shows the results.

For the lowest bandwidth of 100 Mbps, the average pull latencies are 118 s, 885 s, and 889 s for Wharf, DockerLocal and DockerNFS, respectively. As bandwidth increases, the pull latencies of DockerNFS and DockerLocal drop quickly, while pull latency of Wharf stays almost constant. As Wharf minimizes the network traffic to the registry, it offers stable performance and is independent of bandwidth variations on the registry connection.

6.1.5 Varying Number of Daemons. To analyze how the cluster size affects system performance, we vary the number of nodes, and hence the number of daemons, from 5 to 20. We use the same 20-layer image as in the above experiment for pulling and again measure the average pull latencies of daemons and accumulated network traffic. The results are presented in the fifth row of Figure 5.

As the number of nodes increases, the image pull latencies of DockerNFS grow significantly from 129 s with 5 nodes to 826 s with

Container Runtime	Total Exec Time	Avg Exec Time (s)	Min Exec Time (s)	Max Exec Time (s)	Data Received (MB)	Data Sent (MB)
Docker	7 m 26 s	158	31	252	3227	50
Wharf	7 m 47 s	154	46	263	354	768

Table 1: Runtime performance summary

20 nodes. While the growth for DockerLocal is less, it is visible and spans from 40 s with 5 nodes to 215 s for 20. In contrast, Wharf shows stable pull latencies regardless of the number of nodes.

The reason is that DockerNFS and DockerLocal pull the image to each node, i.e. as the number of nodes increases, the load increases. Wharf only pulls an image once and is hence independent of the number of nodes. This is also reflected in the network traffic which stays constant for Wharf but increases for the other two setups.

6.1.6 Varying Pull Parallelism. To take a closer look at an image pull operation, we zoom into the pull operation at the layer level and launch three microbenchmarks: (i) one Docker daemon with 9 concurrent pulling threads; (ii) three Docker daemons with 3 pulling threads each; and (iii) three Wharf daemons, with 3 pulling threads each. The image pulled consists of 10 layer with each layer ranging from 100 MB to 120 MB.

As shown in Figure 6, the single Docker daemon with 9 threads completes the task in 40 s. This is similar to Wharf, which completes the task in 33 s. In both cases, the image is only pulled once with 9 parallel threads but as Wharf combines the resources from three machines to retrieve the image, it can improve the pull latency.

The three Docker daemons take 151 s to 256 s to complete the task. As each daemon pulls the image separately, every layer needs to be pulled 3 times which causes network contention and slows down the individual daemons.

6.2 Runtime Performance

While sharing images from distributed storage can reduce storage and network overhead, containers have to now access data remotely. This could add additional latency to individual tasks in a workload.

To analyze any potential runtime performance degradation in Wharf we run a real-world workload and compare its performance to DockerLocal. Our workload comes from the HPC domain and computes the Burroughs-Wheeler Alignment (BWA), a highly computational intensive job, which usually spawns thousands of parallel tasks. The dataset used for this benchmark is generated by the CCL Makeflow Examples respository [22]. We use the Makeflow workflow system [15] with the WorkQueue execution engine [17] to implement the BWA and run it on 10 nodes with 1,000 parallel tasks with DockerLocal and Wharf.

Table 1 presents the summary of execution times. We can see that on average, Wharf only adds an absolute overhead of 4 s which is equal to 2.6%. Looking at the total execution time, the 5 workflow executions run 21 s slower in Wharf compared to DockerLocal, translating to an overhead of 4.7%. This shows that the remote accesses incurred by Wharf only add a small overhead. As we showed in our microbenchmarks, the benefit of Wharf also increases

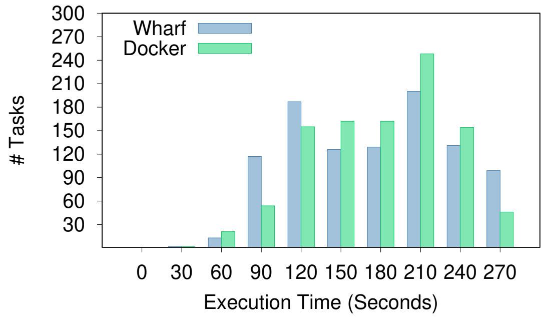


Figure 7: Effect of Wharf on task execution time

with larger cluster sizes and hence, we expect Wharf to perform better compared to DockerLocal at scale.

Additionally, Wharf requires significantly less network resources, retrieving 9.1× less data from the external network. While Wharf introduces more sent network traffic (15× larger compared to Docker) due to NFS traffic, the interconnect to the distributed storage is usually faster compared to an external, wide area network and hence, we do not expect it to become a bottleneck.

Figure 7 shows a histogram of the task execution times for one of the runs of the workload. In general, we see that Wharf produces more short running tasks compared to Docker which is due to the shorter image pull latencies. In general, the two system setups have a similar distribution of task execution time and we thus conclude that using Wharf does not introduce significant runtime performance degradation.

7 RELATED WORK

Previous work has studied how to better adapt container technology to cloud environments. The studies can be classified into several categories, targeting different use cases.

Container Runtimes. Several container runtime systems have emerged to meet specific user requirements. Charliecloud [33] enables container workflows without requiring privileged access to data center resources through user namespaces. Singularity [25] is an alternative container solution that aims to provide reproducibility and portable environments. It prevents privileged escalation in the runtime environment, which improves cluster security, when containers can run arbitrary code. Compared to them, Wharf focuses on improving the efficiency of image distribution and reducing network and storage overheads.

Container registry. Existing work has looked into how to optimize the registry to reduce the cost of pushing and pulling images. VMware Harbor [10] is an optimized registry server designed for enterprise-level clusters. CoMICon [29] introduces a decentralized, collaborative registry design to enable daemons to share images between each other. Anwar et al. [16] analyzed registry traces and designed improvements to speed up image pulls. However, in all the above systems, images are still local to individual daemons as the improvements are only made at the registry side.

Image Distribution. Slacker [21] proposes to lazily load image content from a shared storage backend to reduce the amount of transferred unused image data. While this is similar to Wharf, Slacker does not discuss the implications of sharing images between daemons. Shifter [20] supports sharing images from a distributed file system. However, it implements its own flat image format to serve images and 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. FID [24] uses a P2P protocol to speed up the process of image distribution but still keeps duplicate copies for each daemon.

Sharing VM Images. Previous work has studied how to accelerate the distribution of VM images and improve VM startup times. Razavi et al. [35] focus on simultaneous VM startups of the same VMI (Virtual Machine Image) and find that as the number of VM startups increase, networking becomes the bottleneck. They propose that only a small part of the VMI is loaded at first. In Squirrel [34], the authors note that capacity wise it is possible to keep a significant amount of VMI caches on all client nodes, therefore decreasing boot time further. FVD [40] retrieves data from images on demand, so a VM starts with the minimum data required. Snowflock [26] uses a parent VM to fork multiple child VMs (on arbitrary nodes within a network) by only shipping the critical RAM state required by the child VM to start. Compared to these approaches, Wharf focuses on containers and exploits Docker's layered image structure to retrieve and share images efficiently.

Other work has looked at image sharing via P2P [36] while VDN [32] presents a distribution network for VM images. However, this work does not consider sharing images from different hosts.

8 CONCLUSION

In this paper, we explored how to efficiently share Docker images through a shared storage layer to reduce network and storage overheads when running Docker in large-scale data processing environments. We discussed the drawbacks of a naive solution and the design goals for a native and more efficient approach. We proposed Wharf, a shared Docker image store, which fulfills the design goals and allows Docker daemons to collaboratively retrieve, store, and run images. Wharf uses layer-based locking to support efficient concurrent image retrieval and allows to write data to local storage to reduce remote I/Os. Our evaluation shows that Wharf can reduce image pull times by a factor of up to 12× and network traffic by up to an order of magnitude for large images while only introducing a small overhead of less than 3% when running container workloads.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers whose feedback and advice has helped improve this work. This work was supported in part by the NSF under grant 1642409.

REFERENCES

[1] 2018. Amazon Elastic Container Service. (2018). https://aws.amazon.com/ecs/.
[2] 2018. AUFS - Another Union Filesystem. (2018). http://aufs.sourceforge.net. Accessed March 2018.
[3] 2018. Azure Container Service. (2018). https://azure.microsoft.com/en-us/services/container-service/.
[4] 2018. CoreOS. (2018). https://coreos.com/.
[5] 2018. Device-mapper snapshot support. (2018). https://www.kernel.org/doc/Documentation/device-mapper/snapshot.txt. Accessed March 2018.
[6] 2018. Docker. (2018). https://www.docker.com/.
[7] 2018. Docker Hub. (2018). https://hub.docker.com/.
[8] 2018. Etcd: A distributed, reliable key-value store for the most critical data of a distributed system. (2018). https://coreos.com/etcd/.
[9] 2018. Google Kubernetes Engine. (2018). https://cloud.google.com/kubernetes-engine/.
[10] 2018. Harbor: An enterprise-class container registry server based on Docker Distribution. (2018). http://vmware.github.io/harbor/.
[11] 2018. IBM Cloud Container Service. (2018). https://www.ibm.com/cloud/container-service.
[12] 2018. IBM Spectrum Scale. (2018). https://www.ibm.com/us-en/marketplace/scale-out-file-and-object-storage.
[13] 2018. OpenVZ. (2018). https://openvz.org. Accessed March 2018.
[14] 2018. Overlay Filesystem. (2018). https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt. Accessed March 2018.
[15] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. Makeflow: A Portable Abstraction for Data Intensive Computing on Clusters, Clouds, and Grids. In *Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET)*.
[16] Ali Anwar, Mohamed Mohamed, Vasily Tarasov, Michael Littley, Lukas Rupprecht, Yue Cheng, Nannan Zhao, Dimitrios Skourtis, Amit S Warke, Heiko Ludwig, and others. 2018. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*.
[17] Peter Bui, Dinesh Rajan, Badi Abdul-Wahid, Jesus Izaguirre, and Douglas Thain. 2011. Work Queue + Python: A Framework for Scalable Scientific Ensemble Applications. In *Workshop on Python for High Performance and Scientific Computing (PyHPC)*.
[18] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *ACM Queue* 14, 1 (2016).
[19] Jack Clark. 2014. Google: 'EVERYTHING at Google runs in a container'. (2014). https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.
[20] Lisa Gerhardt, Wahid Bhimji, Shane Canon, Markus Fasel, Doug Jacobsen, Mustafa Mustafa, Jeff Porter, and Vakho Tsulaia. 2017. Shifter: Containers for HPC. *Journal of Physics: Conference Series* 898, 8 (2017).
[21] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*.
[22] Nick Hazekamp and Douglas Thain. 2017. Makeflow Examples Repository. (2017). http://github.com/cooperative-computing-lab/makeflow-examples.
[23] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference (ATC)*.
[24] Wang Kangjin, Yang Yong, Li Ying, Luo Hanmei, and Ma Lin. 2017. FID: A Faster Image Distribution System for Docker Platform. In *2nd IEEE International Workshop on Foundations and Applications of Self* Systems (FAS* W)*.
[25] Gregory M Kurtzer, Vanessa Sochat, and Michael W Bauer. 2017. Singularity: Scientific Containers for Mobility of Compute. *PloS One* 12, 5 (2017).
[26] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. 2009. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*.
[27] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.
[28] Paul Menage. 2007. Adding Generic Process Containers to the Linux Kernel. In *Linux Symposium*.
[29] Senthil Nathan, Rahul Ghosh, Tridib Mukherjee, and Krishnaprasad Narayanan. 2017. CoMICon: A Co-operative Management System for Docker Container Images. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*.
[30] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. 2011. BlobSeer: Next-generation Data Management for Large Scale Infrastructures. *J. Parallel and Distrib. Comput.* 71, 2 (2011).
[31] Bill Nowicki. 1989. *Nfs: Network file system protocol specification.* Technical Report.

[32] Chunyi Peng, Minkyong Kim, Zhe Zhang, and Hui Lei. 2012. VDN: Virtual machine image distribution network for cloud data centers. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*.

[33] Reid Priedhorsky and Tim Randles. 2017. Charliecloud: Unprivileged Containers for User-defined Software Stacks in HPC. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[34] Kaveh Razavi, Ana Ion, and Thilo Kielmann. 2014. Squirrel: Scatter Hoarding VM Image Contents on IaaS Compute Nodes. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*.

[35] Kaveh Razavi and Thilo Kielmann. 2013. Scalable Virtual Machine Deployment using VM Image Caches. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*.

[36] Joshua Reich, Oren Laadan, Eli Brosh, Alex Sherman, Vishal Misra, Jason Nieh, and Dan Rubenstein. 2012. VMTorrent: Scalable P2P Virtual Machine Streaming. In *Proceedings of the 8th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[37] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013).

[38] Neil Savage. 2018. Going Serverless. *Commun. ACM* 61, 2 (2018).

[39] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. *ACM SIGOPS Operating Systems Review* 41, 3 (2007).

[40] Chunqiang Tang. 2011. FVD: A High-Performance Virtual Machine Image Format for Cloud. In *USENIX Annual Technical Conference (ATC)*.

[41] Vasily Tarasov, Lukas Rupprecht, Dimitris Skourtis, Amit Warke, Dean Hildebrand, Mohamed Mohamed, Nagapramod Mandagere, Wenji Li, Raju Rangaswami, and Ming Zhao. 2017. In Search of the Ideal Storage Configuration for Docker Containers. In *2nd IEEE International Workshop on Foundations and Applications of Self* Systems (FAS* W)*.

[42] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. 2016. An Ephemeral Burst-buffer File System for Scientific Applications. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.