Docker Image Sharing in Distributed Fog Infrastructures

Arif Ahmed, Guillaume Pierre

▶ **To cite this version:**

Arif Ahmed, Guillaume Pierre. Docker Image Sharing in Distributed Fog Infrastructures. CloudCom 2019 - 11th IEEE International Conference on Cloud Computing Technology and Science, Dec 2019, Sydney, Australia. hal-02304285

HAL Id: hal-02304285
https://inria.hal.science/hal-02304285

Submitted on 3 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Docker Image Sharing in Distributed Fog Infrastructures

Arif Ahmed
Univ Rennes, Inria, CNRS, IRISA

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA

***Abstract*—Fog computing platforms offer virtualized resources located in the vicinity of their end users. Their broad geographical distribution force them to split physical resources in large numbers of relatively weak machines. The limited available disk space per fog node however creates problems for Docker-based systems which locally cache a copy of every container image they execute: first, caches may fill very quickly, whereas standard Docker never automatically evicts any image from its cache. This motivates us to implement automatic cache replacement in Docker. Second, splitting cache space in multiple disjoint partitions negatively impacts the hit rates. This motivates us to propose allowing multiple co-located Docker servers to share their caches. Our trace-based evaluations show that the proposed design achieves significant cache hit improvements, leading to reductions of average container deployment times between 37% and 78% depending on the scenarios.**

***Index Terms*—Fog computing, Docker, containers.**

I. Introduction

Fog computing aims to extend traditional cloud data centers with additional compute, storage and networking resources located at the edge of the Internet. By bringing the server-side applications in the immediate vicinity of their end users, fog computing promises to deliver very low end-to-end network latencies for highly-interactive applications such as augmented reality gaming, and to drastically reduce the usage of long-distance data transfers for applications such as IoT analytics where large volumes of transient data can be processed locally.

Fog computing architectures are fundamentally different from traditional clouds: to maintain proximity with a large number of users, fog resources must be dispersed across a large geographical area such as a city or an entire country. As a consequence, fog resources are often organized in a large number of Points-of-Presence (PoPs) dispersed across the covered area. Each PoP may not be composed of datacenter-grade servers but rather of a small number of weak machines such as single-board computers connected with each other and with the rest of the Internet using commodity networks [1], [2]. Fog platforms typically automate application deployment across the different PoPs using container orchestration systems such as Docker and Kubernetes [3], [4].

Fog users are usually mobile, which implies that the applications running in the fog may need to be frequently re-deployed in different PoPs to maintain proximity, low latency, and reduce long-distance traffic [5]. However, software deployment can be painfully slow when the fog node needs to download a full container image of the deployed application before starting the container itself [6]. Having their fog-hosted application freeze frequently while new containers get started would clearly be a source of frustration for most end users. Reducing the probability of such image cache misses, and the performance impact of their occurrence when they cannot be avoided, is therefore of crucial importance for providing the end users with a satisfactory quality of experience.

Docker was originally designed for powerful server machines [6]. It therefore keeps a copy of every container image in each server's local cache so it does not need to be downloaded again in case the same image is deployed in the future. Docker also never removes content from its caches unless explicitly requested by their user to do so [7]. Although this strategy makes perfect sense in powerful server machines where disk space is rarely an issue, it creates important storage capacity problems in an environment composed of many weak machines with limited storage space and where containers are frequently started and stopped. If the working set of frequently-deployed images is larger than the storage capacity of a fog computing node, then the same image may need to be repeatedly downloaded, utilized and deleted, creating unnecessary delays and network transfers when re-deploying a container after its image had to be removed from the local node. Another effect of keeping separate image caches in each node is that these caches are likely to contain highly redundant content due to the fact that the same popular images may have been deployed multiple times in different nodes.

We propose to transform these issues in opportunities by allowing multiple fog nodes within a PoP to share the content of their Docker image caches. Instead of using the fog nodes' storage capacity as a set of limited and isolated caches, we propose to aggregate the storage capacity of clusters of co-located fog nodes using a distributed file system. The end result is a single sizable Docker image cache per PoP where large numbers of images can be stored, thereby significantly reducing the probability that images need to be downloaded over a long-distance network upon container deployment.

Our contributions in this paper can be unfolded in *four* parts: (1) we analyze a large Docker registry workload and demonstrate the potential for deployment time improvements of Docker image sharing; (2) we survey distributed file systems (which were typically designed for high-performance computing environments) and discuss their suitability in fog computing environments; (3) we present the design of our Docker image sharing framework which supports image cache sharing

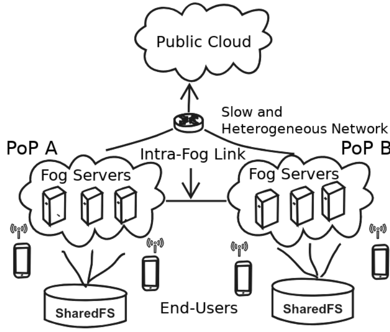


Fig. 1. Distributed fog architecture.

among multiple co-located fog nodes; and finally (4) our trace-based evaluations show that sharing caches among multiple fog nodes delivers significant cache hit rate improvements, and leads to reductions of the average container deployment times between 37% and 78% depending on the scenario.

The paper is organized as follow. Section II presents the background and related work. Section III analyses the workload of large docker registries and demonstrates the potential of image sharing in fog infrastructures. Then, Section IV presents our cooperative Docker framework and Section V evaluates its performance. Finally Section VI concludes.

II. Background and related work

A. Background

1) Fog computing architectures: A typical distributed fog architecture is depicted in Figure 1. In fog computing context, the servers which belong to the same PoP may easily be connected to each other using a fast local-area network. They can therefore be configured to use a shared file system to aggregate their respective storage capacity. On the other hand, the heterogeneous and potentially slow nature of connections between PoPs means that sharing storage between PoPs is unlikely to deliver reasonable performance.

2) Docker: Docker is an open source tool to manage applications inside containers [8]. Applications are packaged in the form of an *image* which usually contains multiple *layers*. Every layer is a full file system with application packages, libraries, binaries, configuration files, etc. Layers are stacked over each other such that every new layer may add, subtract or modify files present in the lower layers. Layers remain physically separated on disk, but a virtual file system such as AUFS and OverlayFS exposes a single "merged" file system view to the container applications [9]. Docker encourages layer reusability so different images often share the same bottom-level layers and differ only by their top-level layers [10].

In Docker, container image layers are always read-only. To allow the running containers to modify their file system, a final read-write layer is dynamically created and stacked on top of the other layers at the starting time of a container. All file system modifications issued within the container are stored in this top-level layer using a copy-on-write (CoW) policy.

When it is instructed to deploy a container, if the requested image is not present in the cache, Docker first downloads *manifest* of the image from the central registry. This manifest contains the list of layers identified by a hash of their content. Docker then downloads the necessary layers which are not already present locally, and finally can start the container itself.

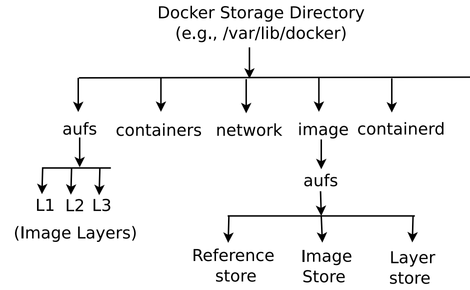


Fig. 2. Docker storage directory.

Figure 2 shows how Docker stores the layers as well as metadata about images and layers in persistent files inside *docker storage directory*. (e.g., `/var/lib/docker`). The metadata has three basic structures: (1) **the reference store** contains the manifests of all the images present in the local image cache; (2) **the layer store** contains the list of all locally-available layers, identified by their sha256 ID as well as metadata such as the layer's size, parent layer etc. and finally (3) **the image store** contains image configuration information such as the CPU architecture it relies on, the default exposed ports, attached volumes, and image history with a list of layers identified by their sha256 ID. As the metadata information are frequently accessed, Docker keeps a copy of the metadata files in memory to speed-up container operations.

The standard mechanism to share Docker images between servers is based on a centralized registry where all available images and layers are stored [11]. A registry supports two main operations: *docker push* and *docker pull*. A push operation uploads a new set of image layers and their manifest file to the registry server, whereas a pull downloads a container image from the registry to the docker daemon.

Although this mechanism provides image sharing, in a fog computing environment it presents three drawbacks. Firstly, downloading an image from the repository must therefore be considered as an expensive operation in terms of download time as well as quantity of data to be transferred long-distance. Secondly, the total size of all downloaded layers would quickly exceed the storage capacity of a resource-limited fog node. Finally, the content of different server caches is likely to be highly redundant with the same highly popular images.

We aim to resolve these issues by allowing multiple co-located fog nodes to *share* their Docker cache with each other. This provides every node with a larger cache than it could locally support, and thereby increases the probability upon container launch that the necessary image layers have been previously downloaded by one of the servers of the PoP.

B. Related work

Container-based virtualization has been widely adopted to handle application deployment in Fog computing infrastructures [3], [4]. These platforms are often built using single-

board computers such as Raspberry Pis which offer excellent performance/cost/energy ratios and are well-suited to scenarios where the device's physical size and energy consumption are important enablers for actual deployment [2], [12]. However, the performance of Docker in such environments is often poor, which justifies the need for improved solutions.

One option to improve the performance of container orchestration systems is to redesign the image registry. CoMICon proposes a distributed registry which distributes the image layers across multiple nodes to speed-up deployment time and increase availability [13]. Similarly, Nitro uses deduplication and network-aware data transfer strategies to reduce the transfer time of images over wide-area networks [14]. These systems aim to reduce the performance impact of a Docker cache miss, but they do not change the cache hit rate. They are therefore complementary to our approach which aims at improving the cache hit rate thanks to shared image caches.

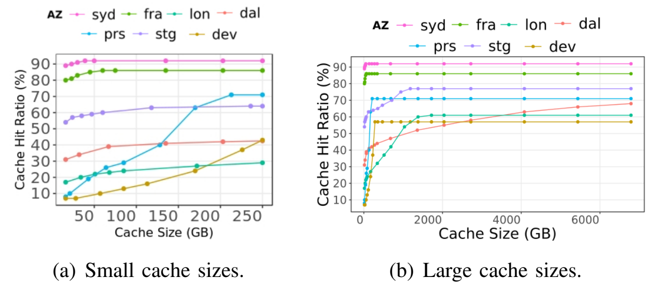
Docker-pi reorganizes the download and extraction of image layers in the Docker server to better exploit hardware-level parallelism [6]. It reduces the performance impact of a Docker cache miss, but it does not influence the frequency of occurrence of these cache misses. In contrast, we rather focus on the management of the Docker cache and its sharing between multiple Docker servers.

We are not the first ones to propose sharing the Docker images across multiple servers. Slacker proposes to store the Docker images in a shared NFS file server [10]. With the use of a specialized storage driver in every Docker server, only the relevant parts of each image need to be downloaded from the NFS server to the Docker server upon every container start operation. However, this assumes that all relevant images are already stored in the NFS file server. This organization would be extremely difficult to manage in a fog computing server as it would require sufficient capacity in every PoP to store the entire set of images that may potentially need to be deployed there one day. Also, Slacker requires every image to be flattened into a single layer, which deviates from the Docker philosophy of image layer reusability.

The closest work to ours is Wharf, which proposes to share the caches of multiple Docker servers in a distributed file system to reduce storage utilization and the number of redundant image retrievals [15]. This work differs from ours in a number of ways. First, it focuses on powerful server clusters where network bandwidth and data storage are cheap. In consequence it mostly discusses container startup times and does not address the issues related to limited cache size in fog computing servers. It also relies on the ability of Docker servers to download multiple image layers simultaneously, which was shown to perform poorly in the context of single-board fog servers [6]. On the other hand, our study focuses on sharing Docker images in Fog infrastructures that are made of large numbers of strongly resource-constrained nodes.

III. Potential benefits of cache sharing

To evaluate the potential benefits of sharing Docker caches among fog servers, and in the absence of publicly-available fog computing workloads, we analyze a container deployment workload in a cloud computing context. As previously discussed, we expect fog computing platforms to experience more frequent re-deployments of the same images than in a normal cloud platform. The results presented here therefore represent a worst-case analysis in terms of cache hit rates.



(a) Small cache sizes. (b) Large cache sizes.

Fig. 3. Cache hit ratios of different AZs vs. shared cache size.

A. Simulation setup

We simulate the behavior of Docker image caches under a registry workload composed of a collection of HTTP-logs generated from 36 IBM Docker registry servers [16]. These registries are classified in 7 Availability Zones (AZ) based on their geographical location and the type of workload they serve. Four AZs (*fra, syd, dal* and *lon*) are dedicated to serve production workloads: *fra* and *syd* are relatively new and have fairly small workloads (86GB and 92GB respectively) whereas *dal* and *lon* serve a much larger working set of images (1718 and 6789 GB respectively). Two AZs (*prs* and *stg*) are used for staging (pre-production) purpose, both are having sizable workload (213GB and 1181GB respectively) and finally *dev* is dedicated for development purpose (283GB).

Each entry in this trace contains the signature of an HTTP request made by a Docker server to the registry for operations such as *docker pull* and *docker push*. The signature provides information about the request such as name of the image, the type of request, and timestamp. However, the trace does not contain sufficient information to reliably detect *container deployments* which resulted in a *cache hit* in the Docker servers. Due to the fact that Docker never deletes cached image layers unless explicitly requested to do so [7], we can see this trace as the residual cache miss traffic from a large set of isolated, infinite-sized image caches. Any temporal locality found in this trace therefore highlights an opportunity for cache sharing between multiple servers.

The simulator replays the container deployment logs and reproduces the behavior of a Docker image cache: when deploying a new container, the server checks if the image is available in the cache storage and, if found, immediately starts the container. Otherwise, it downloads the missing layer(s) in the image cache before starting the container.

B. Cache hit ratio analysis

Figure 3 depicts the cache hit ratio that a shared cache for each AZ would have with different storage capacity. In our

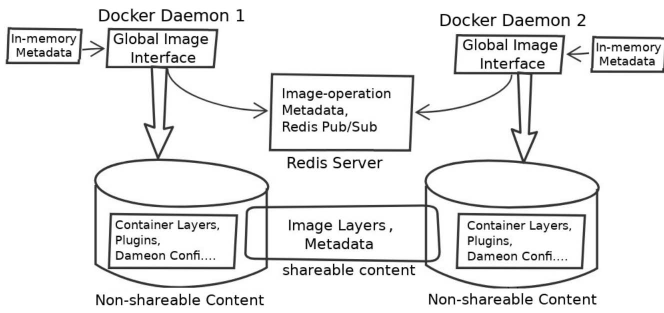


Fig. 4. Docker shared image cache architecture.

simulation we use the well-known Least-Recently Used (LRU) policy to decide which layer should be removed when the cache does not have sufficient available capacity to store a newly-requested image.

We can see in Figure 3(a) that, even with very small shared cache sizes, several AZs exhibit significant cache hit ratios. Most of the AZs (except *prs* and *dev* AZs) exhibit in-between 31% and 89% hit rates with a shared cache size of 32GB. This indicates that a small number of highly popular container images is repeatedly deployed in different servers from the same zone. For these scenarios, sharing even an extremely small cache delivers significant performance improvement compared with no sharing. On the other hand, *prs* and *dev* observe almost no cache hits with a very small cache size. This is probably due to the fact that, during development and pre-staging phases, each container image is deployed only a small number of times before either being replaced with an updated version (in case a bug was detected) or being moved to staging or production.

Figure 3(b) extends these curves until cache sizes of several TBs. When looking at slightly larger shared cache sizes, we observe that shared cache sizes in the order of a few hundred GBs are usually sufficient to exploit the temporal locality and reach hit rates of 50-95%. We can observe that such cache sizes do not deliver additional benefits compared to much smaller sizes. The only exception is *dal* here every cache size increase (up to the size of its total working set) delivers performance improvement.

We conclude the cache sharing between multiple Docker servers would clearly deliver significant benefits in almost all considered scenarios, even with limited size for the shared caches. This is a good news for us considering that in a fog computing platform each PoP would probably have limited storage capacity. The only exceptions where the benefits of sharing caches are limited derive from scenarios where each image is deployed only a couple of times (e.g., during the development phase) or the overall image working set if extremely large.

IV. System design

The general framework for image cache sharing among a group of Docker servers is depicted in Figure 4. In this design, multiple Docker servers use a shared file system to store the (immutable) image metadata and layer files. On the other hand, each Docker server keeps in their local storage the containers' read-write layers, plugins, configuration files etc. However, realizing this design forces us to address a number of difficult challenges which are addressed in the following sub-sections.

A. Choice of distributed file system

A distributed file system (DFS) is defined as any file system that allows access to files from multiple hosts sharing via a computer network [18]. Distributed file systems were mostly designed for high-performance computing environments where the servers are computationally powerful and connected by a high-speed network. In contrast, we aim to use them in severely resource-constrained fog computing environments. Our choice of DFS is therefore largely guided by an analysis of the resource requirements of different DFS.

The design of the Metadata Server (MDS) is an important differentiating factor between the many available distributed file systems. Depending on the DFS implementations the DFS may be centralized in a single machine or decentralized. In the centralized case, the machine which holds the MDS may incur a significant extra load and potentially become a performance bottleneck. A distributed MDS would share this load among the available servers and arguably exhibit better scalability and fault-tolerance properties.

Distributed file systems also differ in the storage medium used to keep the metadata during its operation. Some file systems load the metadata in memory (which promises fast metadata access) while others keep them on disk. In a resource-limited environment such as a fog computing PoP, memory must be considered as a scarce resource. Although keeping metadata in memory may remain affordable if the number of shared files was small, any container image would contain a large number of (usually small) files, and therefore require significant memory resources to maintain their metadata.

Table I presents a comparison of six popular file systems: HDFS [19], CephFS [20], MooseFS [21], GlusterFS [22], iRods [23] and Lustre [24] based on information found on the respective file systems' web sites as well as a survey on distributed file systems [25]. CephFS and GlusterFS stand out because they rely on a distributed metadata server.

CephFS is a fully scalable distributed file system. A CephFS cluster must include one monitor (which maintains a master copy of the cluster map), one manager daemon (in charge of monitoring the file system cluster), at least three Object Storage Daemons (OSD) and at least one MetaData Server (MDS). The monitor and manager are lightweight processes which can easily run in a specific node from a fog computing PoP. The OSDs are in charge of storing all objects from the file system. Finally, the metadata servers share the metadata workload with one another. Since CephFS stores metadata in disk instead of main memory, it can easily be deployed in resource-constrained compute nodes.

GlusterFS is a fully decentralized file system which does make use of any metadata server. Instead, it uses an Elastic Hash Algorithm to deterministically choose in which location each file must be stored [26]. Similar to CephFS, GlusterFS

TABLE I
COMPARISON OF POPULAR DISTRIBUTED FILE SYSTEMS.

	HDFS	CephFS	MooseFS	GlusterFS	iRods	Lustre
Metadata Server	Centralized	**Distributed**	Centralized	**Decentralized**	Centralized	Centralized
Metadata storage	Memory	**Disk**	Memory	**Algorithm**	Disk	Disk
Placement policy	Auto	**User controlled [17]**	No	Random	Admin	No
Striping	Yes	Yes	Yes	Yes	No	Yes
Replication	Yes	Yes	Yes	Yes	Yes	No
Suitable For files	Big files	**Big and small files**	Big files	**Big and small files**	Big files	Big files
Caching	Yes	Yes	Yes	No	Yes	Yes
Memory usage (GB)	8	**1**	20	**1**	2	1
API access	FUSE	**FUSE, ceph, librados**	FUSE	**FUSE**	FUSE	FUSE
POSIX compliance	No	**Yes**	Yes	**Yes**	No	Yes

is sufficiently lightweight (especially thanks to its absence of metadata servers) to be deployed in a fog computing PoP.

CephFS and GlusterFS are the best two contenders for being used in a fog computing scenario. We experimentally compare their performance in Section V-A.

B. Sharing Docker images

Docker uses a single local directory (e.g., `/var/lib/docker`) to keep all cached data such as image manifests, layer metadata, and the layers themselves(see Figure 2). To implement image sharing between multiple docker servers it is important to distinguish the cached content which should be shared from the one which should not.

Shareable content consists of image layers data and the metadata files. To allow multiple Docker servers to access the same layers we mount the distributed file system over the directories which contain these files.

Non-shareable content consists of other content such as container read-write layers, server-specific configurations and plugins.Although Docker stores container read-write layers in the same directory as the read-only layers, we configured Docker to create the read-write layers in a separate directory out of the mounted distributed file system.

Sharing the image metadata and layer files across multiple Docker servers is necessary for our approach, it is by no means sufficient. Docker keeps a copy of the cache metadata in memory, and it does not systematically check the consistency of the in-memory data with the persistent ones before using them. We therefore need to design additional mechanisms to maintain these data consistent, as we discuss next.

C. Consistency maintenance of in-memory metadata

Sharing Docker images through a distributed file system is not sufficient to guarantee the in-memory metadata of the image cache remains consistent with the shared content over time. For instance, when a Docker server executes an image operation such as adding an image in the shared image cache, the updates in the image cache are reflected in the shared file system and the in-memory metadata present in the concerned machine itself, but they are not propagated to the other Docker servers. As a result, in case another Docker server wants to deploy the same image, it will not find it and download it unnecessarily.

To maintain the consistency of the in-memory metadata across all servers within a PoP, we use the popular Redis system and create a publish-subscribe channel to disseminate any update to the in-memory metadata [27]. When one server incurs an update in its in-memory metadata after adding or removing an image, it sends a message in this dissemination channel. When a server receives this message from the above channel, it discards its in-memory metadata and re-reads the image metadata from the shared file system. With this simple mechanism, the in-memory metadata remain consistent across all the servers of the PoP.

D. Preventing concurrent deployments of the same image

In a Docker cluster it is frequent to start multiple instances of the same image simultaneously, for instance to aggregate the processing capacity of multiple servers. In our case, if multiple servers from the same PoP attempted to concurrently deploy the same image, they may all notice that the image is not present in cache and redundantly download the same image. We must therefore allow multiple servers to coordinate with each other and download each image layer only once.

We propose to let a single Docker server downloads all the required layers. Other servers simply block when they discover the image they want to deploy is being downloaded, and resume the normal deployment process after the image download has completed. To allow each Docker server to reliably detect if an image is already being downloaded by another server we store locks in the Redis key-value store: each image is controlled by a separate lock identified by the sha256 ID of the image.

Figure 5 illustrates the updated workflow of the *docker pull* operation. When pulling an image, the Docker server first checks in the Redis database whether a lock for the same image has been created by another server. If the image is not already being downloaded, the server creates a lock in the Redis database under the ID of the image, then pulls the image normally. When the download is completed, it removes the lock and sends a notification to a Redis channel with the same ID. The lock test and set operations are executed within a transaction [28] to ensure atomicity and avoid race conditions.

If a server discovers that the image it needs to pull is already being downloaded by another server, it simply subscribes to the Redis channel (again within a transaction) and waits for a notification that the image download has completed. It can then

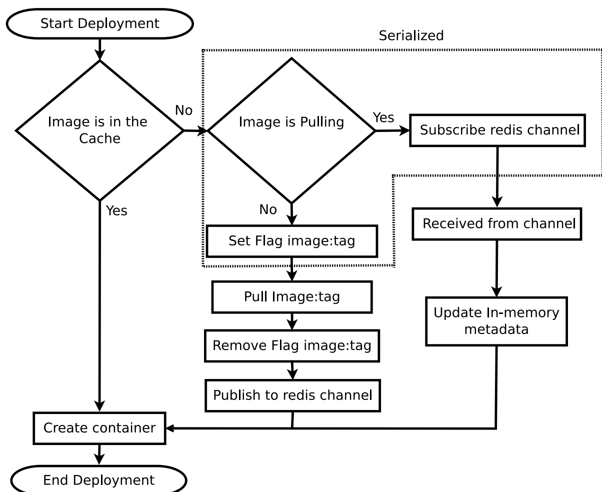


Fig. 5. Flowchart of the updated *docker pull* deployment command.

Algorithm 1: Image replacement algorithm.

Input:
S = size of the new image,
available_space = available capacity of the shared cache,
{I} = list of unused images,
i = image to be deleted

```
1 while (available_space < S) do {
2     i = least_recently_used(I)
3     docker rmi i
4     Redis_Publish(Update_Metadata)
5     {I} = {I} - i
6 }
```

update its in-memory metadata as discussed in the previous section and terminate the deployment normally.

E. Cache replacement

In a fog computing platform where we expect a large variety of applications to be deployed over time, it is important to ensure that only the most relevant container images are kept in cache, and that the less frequently-deployed ones get discarded to save space. This significantly deviates from the standard Docker policy of never removing any image automatically and of rather relying on human administrators to remove unnecessary images manually [7].

To automatically handle the removal of unused container images, we implemented a cache replacement mechanism within Docker [29]. This mechanism get triggered when the available shared disk space is not sufficient to store a new image which is being downloaded. While deploying an image, if the storage capacity is insufficient then Docker pauses the deployment process and calls the *Image Replacement Interface* to remove one or more unused images.

It would obviously be incorrect to remove an image from the cache while it is being used by any of the PoP's Docker servers. To inform other servers about the images they are currently using, Docker servers register the image name and current number of instances in the Redis database.

TABLE II
DISTRIBUTED FILE SYSTEM CONFIGURATIONS.

Scenario	File system	Client	Number of nodes	Replication degree
1	CephFS	Kernel	3	1
2	CephFS	FUSE	3	1
3	GlusterFS	FUSE	3	1

Algorithm 1 depicts the replacement mechanism. When the available storage capacity is too small to store a new image, the Docker server builds a list of the currently unused images and the date they were last accessed. We use the popular Least Recently Used policy which evicts the image that was unused for the longest period of time. The in-memory metadata of all Docker servers is then updated using the same mechanism as discussed in Section IV-C. This removal process is repeated until the system has sufficient storage to store the new image.

V. EVALUATION

We evaluate our system using micro- and macro-benchmarks. Micro-benchmarks show the performance of a single container deployment using various file system configurations, whereas macro-benchmarks show the system's performance under an actual scenario with multiple deployments.

We use a set of 10 VMs representing the nodes of a PoP. These VMs are created using KVM on a Dell PowerEdge R430 server with two Intel Xeon E5-2620 v4 processors running at 2.10GHz, with 8 hyperthreaded cores each, and 64 GB of RAM. Each VM has 2 vCPUs, 1 GB RAM and 32 GB disk and runs Ubuntu 18.04 server with Linux kernel 4.15.0-47-generic. We based ourselves on Docker-pi 18.04, which already contains a number of optimizations designed for Fog computing infrastructures [6]. To avoid interferences from the long-distance network or the Docker hub server, we deployed a private Docker registry within the testbed.

A. Micro-benchmarks

We first evaluate the performance of our system with different distributed file system configurations. Table II depicts the three experimental scenarios used in this study. Scenarios 1 and 2 rely on CephFS (Ceph version 13.2.4) with either its user-level FUSE client [30] of the kernel-based one. Conversely, Scenario 3 relies on Gluster (Glusterfs 4.0.2) with its native FUSE client (GlusterFS does not provide a kernel-level client). All configurations are using three nodes and a replication degree of 1, in order to maximize file system write performance while downloading a new image.

1) Deployment time: We deploy the popular *Ubuntu:latest* image using either regular Docker with no shared image cache, or one of the three shared cache scenarios listed in Table II. All machines are kept otherwise idle while deploying the image.

Table III compares the container's deployment time with shared and non-shared storage, measured from the time the *docker run* command is issued to the moment the container has started. In the case of a cache miss we observe that the configuration with no cache sharing requires 5.2 s to deploy the image, whereas in the distributed file system cases deployment

TABLE III
DEPLOYMENT TIMES OF AN *ubuntu:latest* CONTAINER.

	File system configuration			
	No cache sharing	**Ceph kernel**	**Ceph FUSE**	**Gluster FUSE**
Deployment time (Cache miss)	5.2 s	7.01 s	27.01 s	32.1 s
Deployment time (Cache hit)	0.99 s	1.23 s	2.43 s	2.54 s

times range from 7.01 s to 32.1 s. This may be as writing the image on a distributed file system creates additional tasks for the Docker server compared to simply writing it on the local drive. We however notice large performance variations depending on the client being used to access the distributed file system: although the kernel-based Ceph driver delivers similar performance to a native local drive, the FUSE-based clients suffer from considerable overhead.

In the case of a cache hit, results are similar although the difference between kernel-based and FUSE clients is less important. This is due to the fact that it is not necessary to read the entire image to start a container so the impact of file system performance is lower compared to the other operations that must be conveyed upon container creation.

2) Resource utilization: We instrumented the testbed machines to trace the overall deployment time as well as the node's resource consumption, especially network throughput which is measured using the *nethogs* utility [31].

Figures 6(a) and 6(b) depict the download and upload bandwidth of the Docker server machine while the container image is being deployed upon a cache miss. For obvious reasons the native Docker server does not upload any significant amount of data during the image pull operation, whereas the distributed file systems scenarios see both download (from the image registry to the Docker server) and upload (from the Docker server to the other nodes which participate in the distributed file system). We can also see that the Ceph+kernel configuration can upload data to the distributed file server at a similar rate as the image is being fetched from the registry, whereas the FUSE-based Ceph and Gluster configurations achieve a much lower transfer rate. This is probably due to the fact that FUSE works in user space, which generate large numbers of context switches upon any I/O operation.

Figure 6(c) and 6(d) respectively show the download and upload bandwidth of the Docker server while a container is being deployed from an already cached image. The upload speed is negligible because no content needs to be written to disk. We however observe some download traffic which corresponds to the read operation from the distributed file system. We observe the same phenomenon as in the cache miss scenario, where Ceph+kernel is the only configuration capable of reaching significant throughput in this operation.

We conclude that the Ceph+kernel configuration is the only one which can deliver deployment performance similar to that of the native Docker, both in the *cache hit* and *cache miss* scenarios. Based on these findings, in the next sections we focus on the CephFS+kernel configuration only.

B. Macro-benchmarks

The purpose of sharing Docker image caches is to allow multiple resource-limited PoP servers to increase their cache hit rate by gaining access to a large image cache with a good probability that an image is already present at the time it must be deployed. We therefore evaluate the respective performance of non-shared and shared caches under the same container deployment workload as discussed in Section III.

We created a PoP composed of 5 machines with the same configuration as in the previous sections. When using the shared cache configuration, every machine from the PoP dedicates 10 GB of its disk space to the Ceph distributed file system, and keeps the rest for its local usage. Ceph reserves 1.5 GB space of each disk to store the underlying file system journal, so the total shared storage capacity is 43 GB.

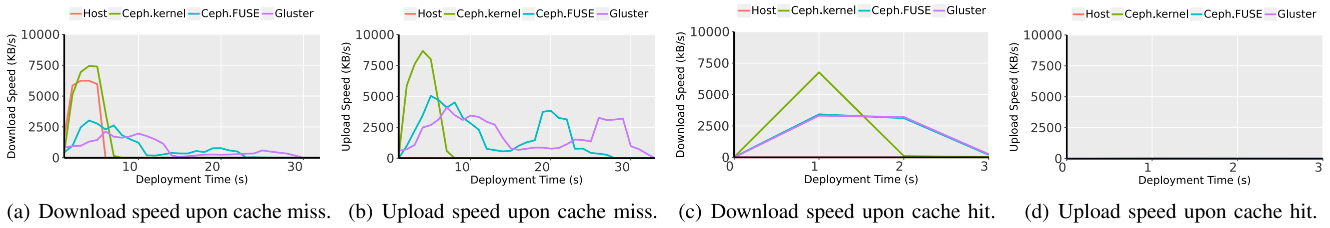
We replayed two traces of container deployments:

- The *fra* availability zone has a total working set of 31 GB. It is too large to fit in a single local cache but it can entirely fit in the shared cache whose aggregate capacity is 43 GB.
- The *dal* availability zone has a total working set of 54 GB. In this case, even the shared image cache is too small to store all downloaded images. It therefore applies the cache replacement mechanisms described in Section IV-E to keep only the most recently used images. As discussed in Section III-B, this workload has limited temporal locality properties and is expected to deliver modest cache hit rates.

Image deployments are issued to the different nodes of the PoP following a round-robin policy. Since we are only interested in the container deployment times, we stop every container immediately after the end of its deployment.

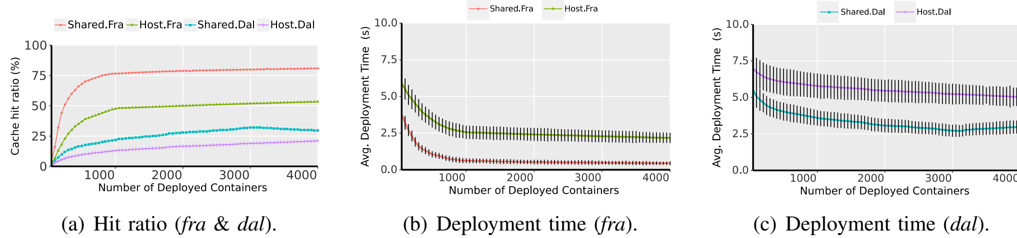
Figure 7(a) depicts the evolution of the cache hit ratio during the execution of the two traces, with and without shared cache, binned by groups of 50 consecutive deployments. For both workloads, the shared cache delivers a much greater hit rate than the non-shared caches. More precisely, during the first few hundred deployments, the shared cache's hit rate grows much faster than the non-shared caches. This is explained by the fact that the most popular images must be downloaded only once in the case of a shared cache whereas in the non-shared case the same image must be downloaded separately by multiple fog nodes. In the end of the curve we see the effect of increasing the cache size available to any Docker server: the cache hit rate of the *fra* zone stabilizes around 82% using the shared cache whereas the non-shared caches deliver only 52% hit rate. In the case of the *dal* zone the cache hit rates are more modest (as expected from the study in Section III-B) but there as well the shared cache delivers a significant cache hit rate improvement compared to non-shared caches.

Figures 7(b) and 7(c) compare the average and standard deviation of deployment times of *Fra* and *Dal* AZs during the same experiment. After deploying the first dozen deployments, in the *fra* trace the average shared-cache deployment time is approximately 4 s whereas the non-shared cache observes deployment times close to 6 s. This difference persists through the entire workload: the mean deployment time of shared caches stabilizes to a value 78% lower than non-shared caches.



(a) Download speed upon cache miss. (b) Upload speed upon cache miss. (c) Download speed upon cache hit. (d) Upload speed upon cache hit.

Fig. 6. Resource utilization upon a cache hit & cache miss.



(a) Hit ratio (*fra* & *dal*). (b) Deployment time (*fra*). (c) Deployment time (*dal*).

Fig. 7. Hit ratio and deployment time of shared vs. non-shared image caches under a workload of 4000 container deployments.

In the case of the *dal* trace we observe a similar behavior. The lower cache hit rates of this difficult workload imply that the average deployment times remain above 2 s. However, here as well the shared cache delivers significantly lower deployment times than the non-shared scenario (37% reduction of the stabilized mean response time). We also observe smaller standard deviations of the deployment times in the shared-cache scenarios, which indicates that deployment times are more predictable than in the case of non-shared caches.

These improvements in container deployment times may significantly impact the perception that fog applications' end users have about the performance of the overall system.

VI. Conclusion

Docker was implemented with the assumption that every server's local cache would be large enough to store all the relevant container images after they are first downloaded. This assumption is however not true in fog computing environment where the compute resources are split between a large number of relatively weak machines. In such environments we extended Docker with a cache replacement policy which evicts unused images and maintains an acceptable image cache size. Splitting the available cache size also negatively impacts the cache hit rates because the same popular images must be downloaded and stored separately in multiple disjoint caches. We therefore proposed sharing caches among multiple co-located fog nodes. Our trace-based evaluations show that the proposed design achieves significant cache hit improvements, leading to reductions of average container deployment times between 37% and 78% depending on the scenarios.

References

[1] P. Basford *et al.*, "Performance analysis of single board computer clusters," *Future Generation Computer Systems*, 2019.
[2] A. van Kempen *et al.*, "MEC-ConPaaS: An experimental single-board based mobile edge cloud," in *Proc. IEEE Mobile Cloud*, 2017.
[3] S. Hoque *et al.*, "Towards container orchestration in fog computing infrastructures," in *Proc. IEEE COMPSAC*, 2017.
[4] M. Nardelli *et al.*, "Multi-level elastic deployment of containerized applications in geo-distributed environments," in *Proc. FiCloud*, 2018.
[5] L. Bittencourt *et al.*, "Mobility-aware application scheduling in Fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, 2017.
[6] A. Ahmed and G. Pierre, "Docker container deployment in fog computing infrastructures," in *Proc. IEEE EDGE*, 2018.
[7] Docker Inc., "Prune unused Docker objects," https://docs.docker.com/config/pruning/, 2019.
[8] ——, "Docker: Build, Ship, and Run Any App, Anywhere," https://www.docker.com/, 2019.
[9] Wikipedia, "aufs," https://en.wikipedia.org/wiki/aufs, 2019.
[10] T. Harter *et al.*, "Slacker: Fast distribution with lazy Docker containers," in *Proc. Usenix FAST*, 2016.
[11] Docker Inc., "Docker hub," https://hub.docker.com/, 2019.
[12] W. Hajji and F. P. Tso, "Understanding the performance of low power Raspberry Pi cloud for big data," *Electronics*, vol. 5, no. 2, 2016.
[13] S. Nathan *et al.*, "CoMICon: A co-operative management system for Docker container images," in *Proc. IEEE IC2E*, 2017.
[14] J. Darrous *et al.*, "Nitro: Network-aware virtual machine image management in geo-distributed clouds," in *Proc. IEEE/ACM CCGrid*, 2018.
[15] C. Zheng *et al.*, "Wharf: Sharing docker images in a distributed file system," in *Proc. ACM SOCC*, 2018.
[16] A. Anwar *et al.*, "Improving Docker registry design based on production workload analysis," in *Proc. Usenix FAST*, 2018.
[17] R. H. Inc., "Crush maps - ceph documentation," http://docs.ceph.com/docs/jewel/rados/operations/crush-map/, 2019.
[18] Wikipedia, "Comparison of Distributed File Systems," https://en.wikipedia.org/wiki/Comparison_of_distributed_file_systems, 2019.
[19] Apache Software Foundation, "Apache Hadoop," http://hadoop.apache.org/, 2019.
[20] Red Hat Inc., "Ceph," https://ceph.com/, 2019.
[21] Core Technology, "MooseFS distributed file system," https://moosefs.com/, 2019.
[22] Red Hat Inc., "Gluster | Storage for your Cloud," https://www.gluster.org/, 2019.
[23] iRODS Consortium, "iRODS," https://irods.org/, 2019.
[24] OpenSFS, "Lustre," http://lustre.org/, 2019.
[25] B. Depardon *et al.*, "Analysis of six distributed file systems," Research report, 2013, https://hal.inria.fr/hal-00789086.
[26] Gluster, Inc., "Cloud storage for the modern data center – an introduction to gluster architecture," 2011, http://moo.nac.uci.edu/~hjm/fs/An_Introduction_To_Gluster_ArchitectureV7_110708.pdf.
[27] Redis Labs, "Redis pub/sub," 2019, https://redis.io/topics/pubsub.
[28] ——, "Redis transactions," 2019, https://redis.io/topics/transactions.
[29] Wikipedia, "Cache replacement policies," https://en.wikipedia.org/wiki/Cache_replacement_policies, 2019.
[30] Michael Kerrisk, "fuse(4) - linux manual page," 2019, http://man7.org/linux/man-pages/man4/fuse.4.html.
[31] A. Engelen, "raboof/nethogs: Linux 'net top' tool," https://github.com/raboof/nethogs, 2017.