Challenges in Storing Docker Images

ALI ANWAR, LUKAS RUPPRECHT, DIMITRIS SKOURTIS, AND VASILY TARASOV

Ali Anwar is a research staff member at IBM Research–Almaden. He received his PhD in computer science from Virginia Tech. In his earlier years he worked as a tools developer (GNU GDB) at Mentor Graphics. Ali's research interests are in distributed computing systems, cloud storage management, file and storage systems, AI platforms, and the intersection of systems and machine learning. Ali.Anwar2@ibm.com

Lukas Rupprecht is a researcher in the Storage Systems Group at IBM Research–Almaden. His research interests are broadly related to distributed systems for data management, including scalability, performance, fault tolerance, and manageability aspects. He received his PhD from Imperial College London and holds MSc and BSc degrees from Technical University Munich. Lukas.Rupprecht@ibm.com

Dimitris Skourtis is a Researcher at IBM Research–Almaden. Prior to that he worked on resource management and scheduling for ESXi at VMware. He has a PhD in computer science from UC Santa Cruz and a masters in mathematics from the University of St Andrews. His interests include distributed systems, data management, and QoS for modern storage devices. Dimitrios.Skourtis@ibm.com

In this article, we describe the structure of Docker images, how they are managed by Docker clients, and how they are stored at Docker registries. We then present several weaknesses in the current design that can cause Docker images to consume excessive storage capacity, degrade container performance, and create contention on the network and the underlying storage infrastructure. We suggest several improvements to alleviate these problems.

At times it seems surprising that hardware virtualization, established virtual machines (VMs), rather than software containers took precedence in the technology evolution. Indeed, in so many practical use cases, one simply wants to run multiple isolated applications on top of a single kernel instead of emulating an entire operating system. This lightweight approach allows containers to start in a fraction of a second and, compared to VMs, consume much less memory and storage, save CPU cycles, and require only a single OS license.

A number of OS-level virtualization technologies appeared in the early 2000s (e.g., Solaris Zones, Linux-VServer, Virtuozzo), but it was only in 2013, with the advent of Docker, that containerization started its conquest of datacenters, clouds, and human minds. By 2013, the Linux kernel components required for containerization—cgroups and namespaces—were already sufficiently mature to provide reliable resource control and boundary separation. However, what was missing was a user-friendly, practical, and yet flexible way to create, deploy, and manage containers. Docker provided this technology. At its heart are Docker images, which form the basic abstraction for users to operate containers.

Container Images and Their Storage

Docker storage can be roughly split into two main parts: client-side storage of images and image distribution via a central registry. In the following, we describe both of these aspects.

Docker Images and Client-Side Storage

In the majority of today's systems, a running application expects its binaries, libraries, and configuration and data files to be stored and accessed through a file system. Hence, the file system tree is an integral part of an application runtime environment. A container *image*, at its core, can be viewed as a file system tree containing all files required by an application to operate. In a simple image implementation, one could copy the required file system tree to a directory and run a containerized application on top of it. However, when a new instance of the same application needs to be started, a new copy of the entire tree has to be created in order to keep any file changes local to each application instance. This slows down container startups significantly.

As a solution to this problem, Docker employs a copy-on-write (CoW) approach to speed up file system creation for containers. Specifically, similar to the "gold images" concept in VMs, Docker defines images as immutable entities. To create a fully functional—and, in particular, writable—root file system for a container, Docker makes use of technologies such as OverlayFS [2]. OverlayFS can create a logical file system on top of two different directories, also known as *layers*, one of which is designated as *writable* while the other one is *read-only*. When Docker creates a new container, the writable layer is initially empty while the read-only layer contains the file system tree of an immutable image.

Vasily Tarasov is a Researcher at IBM Research–Almaden. His most recent studies focus on new approaches for providing storage as a service in containerized environments. His broad interests include system design, implementation, and performance analysis. vtarasov@us.ibm.com

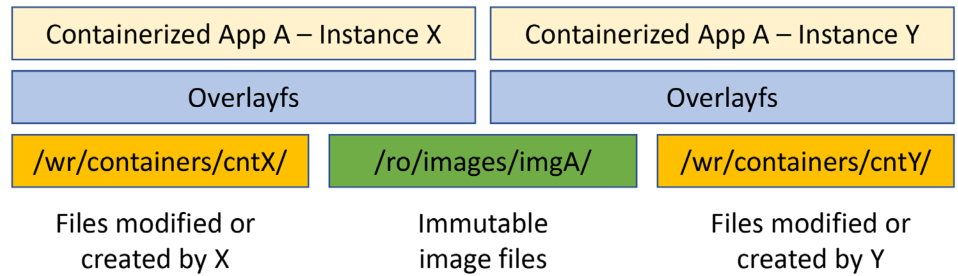

Figure 1: Two containers X and Y running the application A from the same image

A file in OverlayFS serves as a proxy to either a file in the image (read-only layer) or in the writable layer. For example, reading of a file in OverlayFS is initially redirected to the corresponding file in the read-only layer. However, when an application tries to update a file, OverlayFS seamlessly copies it to the writable layer and updates the file there. After that, all I/O operations to the file go to the copy in the writable layer. In such a design, starting a container is a breeze, as it only requires the creation of an empty writable layer and mounting the OverlayFS. Data copying is performed later and only on demand (copy-on-write). Figure 1 schematically illustrates this setup.

So far, we assumed that immutable container images already exist. But how are they created initially? The capability to easily build images is an important property that makes Docker so attractive. It relies on the ability to convert writable layers to read-only layers and assemble an immutable image from a collection of read-only layers. Figure 2 depicts this organization. In Docker, an image is treated as a stack of read-only layers, where each layer contains the changes, at file granularity, compared to the lower layer. The lowest layer in a stack contains the changes compared to an empty file system. Therefore, every layer can be thought of as a collection of files and directories, and layers belonging to the same image comprise its entire file system tree. OverlayFS is capable of assembling a collection of read-only layers and one writable layer into a single logical file system for a running container. Besides OverlayFS, there are other approaches, which can support the above described storage model of Docker images, e.g., AUFS, device-mapper, or Btrfs. Support for each of these storage back ends is implemented through a *graph driver*.

To create a container image, one can start from an empty container, copy files to its writable layer, and then use the *docker commit* command to convert the writable layer to a read-only layer. As this is tedious, Docker provides the concept of a *Dockerfile* and the *docker build* command for convenience. In this case, Docker creates a temporary build container, updates its root file system using the instructions in the Dockerfile, and commits the writable layer (i.e., converts it to read-only) after every instruction. Images can also be created from previously built images (e.g., an OS distribution). This results in different images sharing layers (see Figure 2).

Registry-Side Storage

For ease of distribution, Docker images are kept in an online store called a *registry*. A registry, such as Docker Hub [1], acts as a storage and content delivery system, holding named Docker images, available in different tagged versions. Figure 3 shows the basic structure of a typical registry and how users interact with it. Users create *repositories*, holding images for a particular application (e.g., Redis or WordPress) or a basic operating system (e.g., Ubuntu or CentOS). Images in a repository can have different versions, identified by *tags*. The combination of a repository name (which typically also includes a user name) and a tag uniquely defines the name of an image.

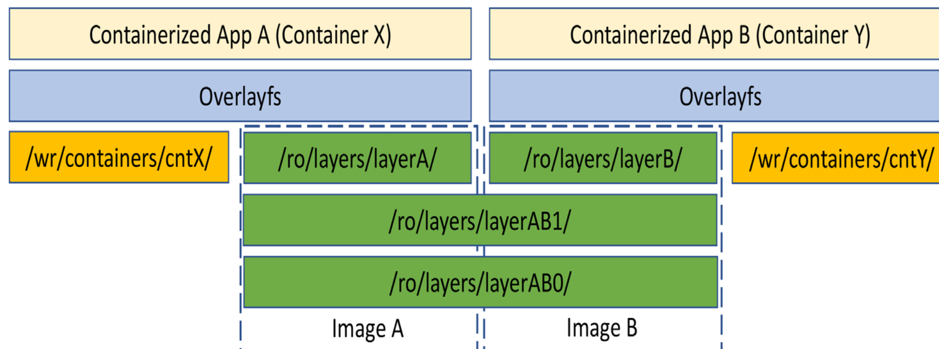

Figure 2: Two applications A and B running in two containers X and Y from two images that share two layers AB0 and AB1 between them

Users can add new images or update existing ones by *pushing* to the registry and retrieve images by *pulling* from the registry. The information about which layers constitute a particular image is kept in a metadata file called a *manifest*. The manifest includes additional image settings such as target hardware architecture, executable to start in a container, and environment variables.

Each layer is stored as a compressed tarball in the registry and has a content-addressable identifier called a *digest*, which uniquely identifies a layer. The digest is a collision-resistant hash of the layer's data (SHA-256 by default). The identifier allows the user to efficiently check whether two layers are identical, share identical layers across different images, and transfer only the missing layers of an image between registries and clients.

Clients communicate with the registry using a RESTful HTTP API. To pull an image from the registry, a Docker client first fetches the image manifest by issuing a `GET` request. Then the client uses the manifest to identify individual layers unavailable in local storage. Finally, the client `GET`s and extracts the missing layers. Pushing works in reverse order compared to pulling.

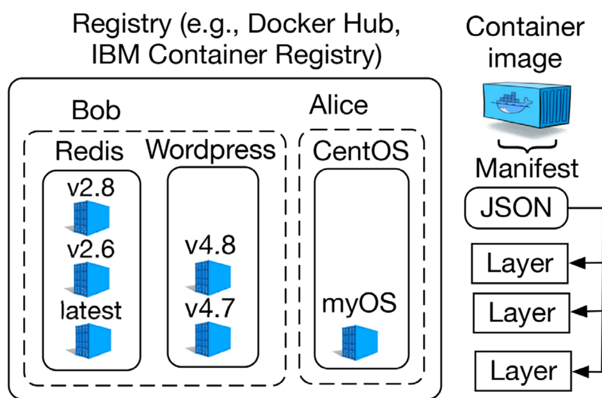

Figure 3: On the left: relationship between registry, users (Bob and Alice), repositories (Redis, WordPress, CentOS), and tagged images (v2.8, latest, v4.8, myOS, etc.). On the right: Docker image structure.

After creating the manifest locally, the client first `PUT`s all the new layers that are not yet stored in the registry, and then `PUT`s the manifest itself.

The existing Docker registry server is a single-node application. To concurrently serve a high-request load, production deployments typically use a load balancer in front of several independent registry instances. All instances store and retrieve images from a shared backend storage. Currently, the Docker registry supports multiple storage back ends such as in-memory for reference and testing purposes, file system for storing layers in a local directory tree, and object storage for storing layers as objects in popular object stores such as Amazon S3.

Challenges of Scale

The increasing popularity of containers and the shift in application development towards cloud-native applications pose several challenges for Docker storage on the client and registry sides.

High Redundancy

As of March 2019, Docker Hub contains more than 2 million public images. Grossly underestimating, we found that those images would utilize more than 1 PB of storage in raw format. The utilization is likely several times higher as we have not considered all images, e.g., we omitted the ones stored in private repositories. Additionally, every day more than 1,500 new images are added. This puts pressure on the storage infrastructure, and it is important to understand the challenges in storing Docker images in order to keep registries and client-side storage scalable.

As described above, Docker employs two mechanisms to reduce image storage utilization: layering of images and compression. However, even with these space optimizations, the storage utilization is still significant. Looking at the individual contributions of each mechanism on the 10,000 most popular images in Docker Hub, we found that layering provides a reduction of 1.48×, and compression decreases the data set by an additional 2.38×. Combined, this results in a total reduction of 3.54×. While this would reduce the estimated 1 PB to approximately 290 TB, storing all

images still requires a significant infrastructure budget. Using AWS S3 standard storage, the resulting annual cost for storage alone would be between $75,000 and $130,000 (depending on the specific AWS region) plus any additional networking costs. For companies that provide registries as a service, e.g., Docker Hub, Jfrog, Artifactory, or Quay, this is particularly problematic. However, even companies maintaining their own registries are sensitive to the high cost of the required storage infrastructure.

To reduce storage utilization of Docker images, the primary goal is to remove any existing redundancy in the stored data, as is intended by the layering of images. However, we found that this is ineffective in its current form [9]. In our sample data set of the 10,000 most popular Docker Hub images, 67,047 unique layers still contain almost 80% duplicate files.

We believe that this is due to two main reasons. First, Docker images must be self-contained, contrary to earlier approaches for software packaging (e.g., RPM or DEB). As a result, completely unrelated images may rely on common components like binaries or shared libraries. In our 10,000-image data set, we found that libraries such as libslang, libstdc++, or libc are present in over 1,000 images. Second, developers create their images independently without exhaustively considering existing layers. This leads to many “almost equal” layers, i.e., layers that share a large number of, but not all, files with existing layers and as a result are not identical and so must store separate copies. That is not to blame developers; examining existing layers is not a task to be performed manually, and further, one needs to have the required incentives to even consider doing so.

On top of the registry storage redundancy, network traffic and client-side storage are also affected. Suboptimal layering means that duplicate data is unnecessarily transferred over the network, potentially increasing expensive outbound network traffic in a typical public cloud offering. Additional network traffic can increase startup times, whereas “almost equal“ layers can increase storage space utilization on a single client unnecessarily.

We proposed one approach for solving the redundancy problem through layer restructuring that considers both storage and network utilization [6]. The approach takes the existing layers in a registry and constructs new layers out of the set of all files, such that storage space and network redundancy are minimized. Preliminary results on a small, 100-image data set show that we can achieve storage space savings of up to 2.3×. In the same paper, we discussed the redundancy problem in more detail and explain why file-level deduplication on the registry-side is insufficient.

Low Performance

While containers are, in most cases, much more performant in terms of startup times compared to virtual machines, new use cases such as serverless computing are demanding even lower latencies. Those requirements put pressure on the storage infrastructure, both at the registry and the client side.

As previous work has found, pulling can contribute as much as 76% to the overall container startup time [4]. Hence, the registry is a critical component in the container infrastructure and needs to be designed to minimize pull latencies and serve images as fast as possible. One direction for improving registry performance is to exploit workload characteristics and integrate workload-aware optimizations in a registry’s design or configuration. We performed an in-depth analysis of production traces from the IBM Cloud Container Registry to study common registry workloads and drive potential optimizations [3, 5]. The analysis revealed several important characteristics. First, there are often hotspot layers, which are accessed more frequently than others, leading to a skewed workload. For example, at one of the registry sites, 59% of requests only went to 1% of the layers. Second, most layers are small, with 65% being smaller than 1 MB while 80% are smaller than 10 MB. Third, requests are correlated, i.e., if a client requests an image manifest from a repository and the repository has recently seen new layers being pushed, then these new layers are likely to be pulled.

These observations encourage the use of layer caching and prefetching optimizations to reduce registry load and pull latencies. Using these lessons, we proposed a new registry design [3]. The design employs a two-tier registry cache and exploits the correlation of push and manifest pull requests to preload layers that are likely to be pulled into the cache. Each time a client requests a manifest for an image in a repository that has seen an update in the recent past (defined by a threshold parameter), the layers from the manifest are prefetched into the cache. Our evaluation revealed that having such an optimized backend storage system for the registry can reduce the latency from 100 ms to 10 ms for layers smaller than 1 MB.

Besides the registry, client-side storage can also affect container startup and runtime performance. This is particularly problematic in large-scale setups, where either many containers run on a single host or the same image needs to be pulled by a large number of nodes to run a parallel workload.

In the first case of many containers being started simultaneously on one host, the choice of storage driver can significantly impact how fast containers start and complete [7]. The most important property is the granularity at which the driver performs copy-on-write, i.e., at file- or block-granularity. For example, we found that for the OverlayFS driver, startup latencies can reach hundreds of seconds for write-heavy workloads, which trigger large copies of data due to copy-on-write. As a result, the completion of those containers is also delayed significantly. In contrast, drivers, which perform copy-on-write at block granularity (e.g., Btrfs or ZFS), did not significantly affect startup latencies.

However, other workloads draw a different picture. For example, when running an Ubuntu `dist-upgrade` in 10 containers in parallel, file-based drivers (both OverlayFS and AUFS) outperformed block-based drivers significantly. This could be due to the fact that block-based drivers are often based on native file systems and, hence, benefit less from the Linux page cache, which could slow down containers with mixed read/write workloads. However, we do not know the exact reason at this point.

In the case of large-scale parallel workloads, which require users to pull the same image on many different nodes, additional problems arise. Most importantly, pulling the same image several times (potentially hundreds or thousands of times depending on the scale of the workload) wastes network bandwidth during the pull and storage capacity on the individual Docker clients. Therefore, it is desirable to enable individual clients to collaborate when pulling an image, i.e., let different clients pull different layers of the image and only store a single copy of the image on shared storage such as an NFS file system. In environments where no local storage is available, such as an HPC cluster, sharing images is a necessity to enable containerized workloads.

To enable collaborative pulling and sharing of images, Docker clients need to be synchronized. With Wharf, we have built such a system [8]. As we assume the existence of a shared storage system for the container images, we can use this shared storage to store the global state for all clients, e.g., which images have been pulled already, which images are currently pulled, and who is pulling which layer. Wharf uses additional optimizations such as minimizing lock contention by exploiting the layered structure of Docker images and writing image changes to local storage, if available, to reduce overhead during pulling and running an image. For large images pulled in parallel to an NFS share, Wharf can improve pull latencies by up to 12× compared to a naïve solution, in which each client pulls its images to a separate location on the NFS share.

Conclusion

Containers are expected to form the backbone of prospective computing platforms. However, even though individual containers are lightweight, providing and operating infrastructure for millions of containers is a hard challenge. In this article, we described how Docker stores container images and presented the challenges that we discovered when operating large-scale container deployments: high data redundancy across images, inefficiencies in graph drivers, low-performing registries, the inability to effectively use images on shared storage, and others. We referenced some of the possible solutions and hope that this article will nourish the discussion on this important topic.

Acknowledgments

We would like to thank our collaborators and co-authors from academia, IBM Research, and other organizations: Ali Butt, Yue Chang, Hannan Fayyaz, Zeshan Fayyaz, Dean Hildebrand, Wenji Li, Michael Littley, Heiko Ludwig, Nagapramod Mandagere, Nimrod Megiddo, Mohamed Mohamed, Raju Rangaswami, Douglas Thain, Amit Warke, Ming Zhao, Nannan Zhao, and Chao Zheng.

References

[1] Docker Hub: https://hub.docker.com/.

[2] OverlayFS: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[3] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hilde-brand, and A. R. Butt, "Improving Docker Registry Design Based on Production Workload Analysis," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, pp. 265–278.

[4] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast Distribution with Lazy Docker Containers," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pp. 181–195.

[5] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt, "Bolt: Towards a Scalable Docker Registry via Hyperconvergence," in *IEEE International Conference on Cloud Computing (IEEE CLOUD 2019)*.

[6] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo, "Carving Perfect Layers Out of Docker Images," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '19)*, USENIX Association, 2019.

[7] V. Tarasov, L. Rupprecht, D. Skourtis, W. Li, R. Rangaswami, and M. Zhao, "Evaluating Docker Storage Performance: From Workloads to Graph Drivers," *Cluster Computing*, Online First, 2019.

[8] C. Zheng, L. Rupprecht, V. Tarasov, D. Thain, M. Mohamed, D. Skourtis, A. S Warke, and D. Hildebrand, "Wharf: Sharing Docker Images in a Distributed File System," in *Proceedings of the 9th ACM Symposium on Cloud Computing (SoCC '18)*, pp. 174–185.

[9] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt, "Large-Scale Analysis of the Docker Hub Dataset," in *IEEE International Conference on Cluster Computing (IEEE Cluster 2019)*.

SAVE THE DATES!

SRECON® EUROPE MIDDLE EAST AFRICA

OCTOBER 2–4, 2019 • DUBLIN, IRELAND
www.usenix.org/srecon19emea

SRECON® AMERICAS WEST

MARCH 24–26, 2020 • SANTA CLARA, CA, USA
www.usenix.org/srecon20americaswest

SRECON® ASIA PACIFIC

JUNE 15–17, 2020 • SYDNEY, AUSTRALIA
www.usenix.org/srecon20apac

SREcon is a gathering of engineers who care deeply about site reliability, systems engineering, and working with complex distributed systems at scale. SREcon challenges both those new to the profession as well as those who have been involved in SRE or related endeavors for years. The conference culture is based upon respectful collaboration amongst all participants in the community through critical thought, deep technical insights, continuous improvement, and innovation.

Follow us at @SREcon