Making containers lazy with Docker and CernVM-FS

N Hardi, J Blomer, G Ganis and R Popescu

CERN

E-mail: `nhardi@cern.ch`

Abstract. Containerization technology is becoming more and more popular because it provides an efficient way to improve deployment flexibility by packaging up code into software micro-environments. Yet, containerization has limitations and one of the main ones is the fact that entire container images need to be transferred before they can be used. Container images can be seen as software stacks and High Energy Physics has long solved the distribution problem for large software stacks with CernVM-FS. CernVM-FS provides a global, shared software area, where clients only load the small subset of binaries that are accessed for any given compute job.

In this paper, we propose a solution to the problem of efficient image distribution using CernVM-FS for storage and transport of container images. We chose to implement our solution for the Docker platform, due to its popularity and widespread use. To minimize the impact on existing workflows our implementation comes as a Docker plugin, meaning that users will continue to pull, run, modify, and store Docker images using standard Docker tools.

We introduce the concept of a thin image, whose contents are served on demand from CernVM-FS repositories. Such behavior closely reassembles the lazy evaluation strategy in programming language theory. Our measurements confirm that the time before a task starts executing depends only on the size of the files actually used, minimizing the cold start-up time in all cases.

1. Introduction

Linux process isolation with containers became very popular with the advent of convenient container management tools such as Docker [1], rkt [2] or Singularity [3].

The Docker system, in particular, has had a large influence in the Linux container community because it introduced the de facto container image format, layered images and git-like workflow appreciated by container developers. Compared to full VM virtualization, containers benefit from close-to-zero performance overhead for both compute intensive and I/O intensive workloads.

However, despite the fact of being composed of shared read-only layers to save space and bandwidth, the price for obtaining images can be very high. This is because image handling is coarse grained, and the full image has to be transferred in order to be used. For example, to start a cluster of 1000 nodes using an image of 1 GB the network has to transfer 1 TB until all containers start executing the payload.

In this paper, we present a solution to the problem of efficient image distribution based on CernVM-FS [4] for storage and transport of container images.

The solution exploits the fact that, on average, only 6% of the image is required [5]. We developed a Docker plugin [6] to refine data reuse granularity from layers to files. Additionally, the download is delayed until the file is accessed for the first time. Using this approach only a fraction of the image is initially transferred and containers can start almost instantly.

We implemented our solution for the Docker platform, but the similar approach is applicable to other containerization technologies as well.

2. Docker system

The main components of a Docker ecosystem are the Docker daemon and Docker command line utility for controlling the daemon on every node, together with a central Docker registry which serves a repository for distributing and sharing the Docker images.

2.1. The Docker images

Docker images are described by image manifests [7] and they are composed of one or more read-only layers. The image manifest and layers are immutable. A layer is a tarball containing a part of the image's file system. For instance, the base layer could contain an Ubuntu system, the second layer adds additional software such as OwnCloud and third layer contains user's configuration and tweaks. Manifest and layers are content addressable, i.e., identified by a cryptographic hash that is calculated from their contents. Such an organization is convenient for data reuse. Changes in containers can be committed and saved but this just results in a new read-only layer and a new image manifest. It is not possible to physically remove files from a Docker image but only to hide them. Removing of a file or directory is realized by masking the original file system entry through a so called whiteout file in the scratch area.

2.2. The Docker workflow

The Docker command line tool provides git style subcommands for creating, running, managing and sharing containers. A common usage pattern emerged which involves obtaining a Docker image, creating and running a container, modifying and sharing it.

Conventionally the whole Docker image needs to be obtained before containers based on it can start. Multiple containers can reuse the same image and get a dedicated ephemeral scratch area. When a container stops, the corresponding scratch area is destroyed. In order to permanently keep changes, a new image needs to be created by "burning in" the scratch area as the topmost layer onto the stack of the read-only layers. Newly created Docker images can be uploaded to a Docker registry to be shared. Additionally, new images can be created with Dockerfiles [8] by the build command. It is possible to specify an existing image as a base image.

2.3. The Docker graphdriver

The graphdriver [9] is the component of the Docker daemon that stores new layers and combines them together to images using a union file system (e.g., AUFS [10], OverlayFS [11]). Graphdrivers operate with random layer IDs instead of hash values of layers, which is the very reason why we later introduce thin images. Graphdrivers implement a very slim interface and have limited access to other information about the rest of the system.

2.4. Docker plugin system

Docker v2 plugin interface is substantially reengineered and is available since Docker v1.12. As of Docker v1.13 (January 2017), there is support for graphdriver plugins in addition to the already existing plugin interfaces. Docker plugins are only loosely coupled to the daemon and communicate with it through a TCP or Unix socket using HTTP. The v2 plugins are also called "managed plugins". It is possible to obtain them from a Docker registry, and install and configure them using the Docker command line tool.

3. Using CernVM-FS for the image transport

Image handling in the Docker system can be improved in such a way that only a small fragment of the images is transferred before containers can start. On average only 6% of the image contents is actually read during the lifetime of a container. This comes partially from the way that container images are created. Usually users pick a base image that comes with a set of standard libraries and utilities. For instance, the Ubuntu base image brings almost 150 MB but other software layers that are added on top do not use all the software that comes with such a base image. This presents an opportunity for improvements that we addressed with our solution. It is also important to notice that this effect has much less impact with smaller base images such as Atomic Host [12] and Alpine Linux [13] or even micro CernVM [14]. The overhead coming from transferring whole images that are not carefully built becomes much more important with huge images that can go even up to several tens of gigabytes.

This effect becomes even stronger in the cluster environment where each cluster node tries to obtain such a big image from the central repository. When all nodes download the image at once, the network links can get saturated quickly. With modern software development techniques and continuous integration where software is updated very often, images are rebuilt on daily basis or more often.

In CernVM-FS the unit of transfer and reuse is a file. Big files are chunked into smaller pieces. File and chunk level deduplication is provided transparently. Files are transferred on demand but only after they are requested for the first time. The goal is to store extracted image layers into a CernVM-FS repository and later download only the files that are required for executing the given workload. This approach implicitly brings all the other well known benefits of using CernVM-FS for distributing software, including leveraging HTTP proxy caches, content addressable storage, limited cache size and others.

4. Integration of CernVM-FS and Docker

Our implementation consists of 3 main components.

(i) The Docker plugin based on the existing Docker graphdrivers, using either AUFS or OverlayFS as a union file system.

(ii) A CernVM-FS server and repository with additional "satellite services" to receive new layers from clients and publish them.

(iii) The utility docker2cvmfs for converting images from a Docker registry to thin images stored on CernVM-FS.

Our plugin is an extension of the built-in graphdrivers, so conventional Docker images are supported seamlessly. For CernVM-FS provided images, we introduce the concept of "thin images".

4.1. Running thin images

A "thin image" has all properties of a standard Docker image but instead of the actual data, it carries just the "thin image descriptor". The thin image descriptor is a JSON file with a list of layers that it replaces and the location (i.e., the name of the CernVM-FS repository) where these layers are available. The change in behavior happens once the graphdriver plugin recognizes that the imported Docker image is carrying the thin image descriptor instead of the actual layer data. Upon starting a container from a thin image, the information from the thin image descriptor is used to mount the corresponding CernVM-FS repositories, which are then used as part of the union mount. The graphdriver plugin comes with CernVM-FS client built into the plugin container image so no additional software except the plugin container itself needs to be set up to use thin images.

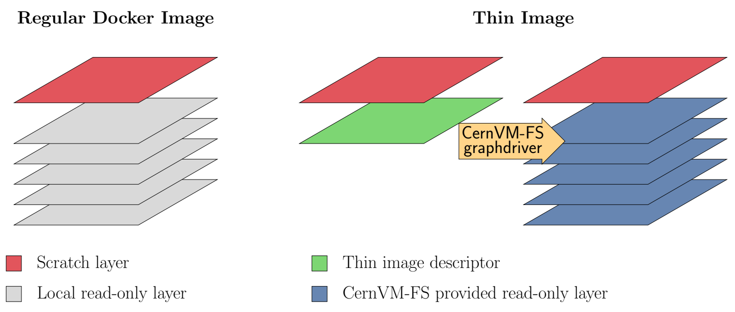

Figure 1. Comparing internal organization of normal and thin images. The original image layers are moved to a CernVM-FS repository and replaced by a "thin image descriptor".

4.2. Updating thin images
The Docker pull and run commands are straightforward, but there are some important details about Docker commit and build commands. The difficulty comes from the fact that CernVM-FS is a read-only file system from the client's viewpoint. Publishing new contents in a CernVM-FS repository takes place on a central server where a transaction is started, changes are made and then the transaction is committed or published. After changes are made and committed on the server, they will eventually propagate to clients. Publishing new contents from client to server is a new scenario for CernVM-FS that requires additional services. For the transport of the tarballs we choose Amazon S3 API [15], a widely used industry standard, in its open source implementation Minio [16]. The concept of active storage has recently become popular and usually is available alongside S3 storage (e.g., Amazon Lambda [17]). The purpose of active storage is to manipulate stored data on certain events (e.g., data is removed or uploaded). Minio supports active storage by calling a webhook. We introduce a "publishing service" which implements a webhook handler for Minio event notifications together with a status API. When new content is uploaded to Minio, the webhook is triggered and the publisher service is notified. Processing of notifications consists of the serialization of requests into a single queue, starting a CernVM-FS transaction, extracting the uploaded layer tarball into the repository and committing the transaction. On error, the transaction gets aborted and the next notification is processed. The status API allows for following the publication process from the client side. Clients remount the CernVM-FS repository once the new content is published.

4.3. Configuring the plugin
Configuration for the CernVM-FS and Minio clients can be supplied by bind mounting directories from a host file system to plugin's container file system. The Docker command line tool provides a subcommand for setting which directories should be bind mounted. The configuration format for the CernVM-FS client reuses the layout of the /etc/cvmfs directory, that usually provides repository public keys and repository configuration. The external configuration will overwrite the internal configuration. It is possible to supply default configuration together with the plugin.

It is not necessary to configure and use the Minio S3 client for running thin images. However, each worker needs to have the credentials and address of the publish status API in order to modify the thin images. The Minio client configuration is a JSON file that is created in the bind mounted directory.

4.4. Creating new thin images using docker2cvmfs
We developed a utility, docker2cvmfs, to fetch the layer tarballs from a Docker registry and create the thin image descriptor. This tool reads an image manifest and downloads each layer

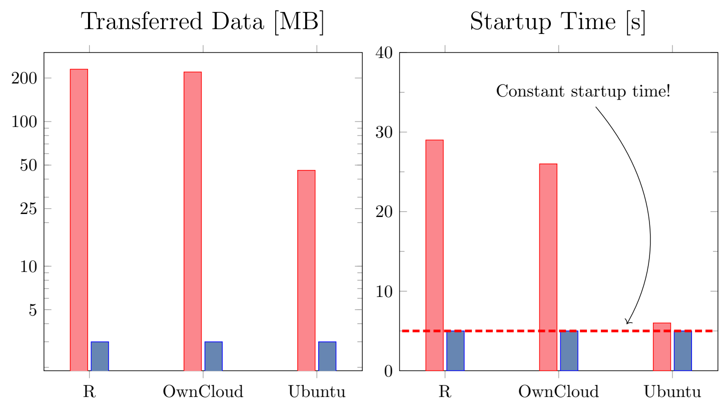

Figure 2. Transfer volume and startup time with the normal and corresponding thin images.

tarball which later needs to be published in a CernVM-FS repository. We use this tool also for debugging because it provides a convenient way to access image manifests.

4.5. Improvements in CernVM-FS

CernVM-FS is a special purpose file system that is already optimized for serving software and operating system binaries (e.g., the root file system of the CernVM virtual appliance). In particular, CernVM-FS supports POSIX operations needed by the software use case, which is often overlooked in other distributed file systems. Only a few tweaks were needed to support hosting container images, namely an improved handling of special files (device nodes) to support serving Docker image layers.

5. Evaluating the results

Figure 2 shows that the cold startup time with a thin image requires only a fraction of the image to be actually transferred. It is important to notice that no matter how large the original image is, the startup time and the amount of data transferred is constant and the same for all images in these tests.

For instance, the R-base Docker image of 200 MB compressed will result in only 6 MB transferred when converted to the thin image format.

6. Summary

In this contribution, we have shown how to integrate the Docker container engine with the CernVM File System for container image distribution. As a result, we removed the Docker Registry as a bottleneck for using Docker containers at the scale of a data center. As Docker image contents are distributed by CernVM-FS on demand and with file-level granularity, typical container start-up times are reduced by one to two orders of magnitude.

The CernVM-FS integration is implemented by means of a Docker graphdriver plugin. The integration is seamless in so far as regular Docker images can be used side-by-side with images from CernVM-FS and all Docker commands and workflows remain independent of the image source. To our knowledge, our Docker graphdriver implementing CernVM-FS access is the first 3rd party Docker graphdriver plugin.

Ongoing work is focused on a first exploitation phase under production conditions in the CERN data center. The CernVM-FS server side “satellite services” used to publish new Docker image layers is currently being generalized in order to support remote upload of contents to a CernVM-FS repository.

References

[1] Docker Inc Docker Engine URL `https://www.docker.com/`

[2] CoreOS Inc rkt URL `https://coreos.com/rkt/`

[3] Kurtzer G M, Sochat V and Bauer M W 2017 *PloS one* **12** e0177459

[4] Blomer J, Aguado-Sanchez C, Buncic P and Harutyunyan A 2011 *Journal of Physics: Conference Series* **331**

[5] Harter T, Salmon B, Liu R, Arpaci-Dusseau A C and Arpaci-Dusseau R H 2016 *Proc. 14th USENIX Conference on File and Storage Technologies (FAST'16)*

[6] Docker Inc Docker engine managed plugin system URL `https://docs.docker.com/engine/extend/`

[7] Docker Inc Docker image manifest specification URL `https://docs.docker.com/registry/spec/manifest-v2-2/`

[8] Docker Inc Dockerfile reference URL `https://docs.docker.com/engine/reference/builder/`

[9] Docker Inc About docker graphdrivers URL `https://docs.docker.com/storage/storagedriver/`

[10] Okajima J AUFS URL `https://aufs.sourceforge.net/`

[11] Brown N OverlayFS doc. URL `https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt`

[12] Project Atomic URL `https://projectatomic.io/`

[13] Alpine Linux URL `https://alpinelinux.org/`

[14] Blomer J, Berzano D, Buncic P, Charalampidis I, Ganis G, Lestaris G, Meusel R and Nicolaou V 2014 *Journal of Physics: Conference Series* vol 513 (IOP Publishing) p 032009

[15] Amazon Amazon S3 URL `https://aws.amazon.com/s3/`

[16] Minio, Inc Minio URL `https://minio.io/`

[17] Amazon Amazon Lambda URL `https://aws.amazon.com/lambda/`