Large-Scale Analysis of Docker Registry Dataset

Your N. Here
Your Institution

Second Name
Second Institution

Abstract

Docker containers are becoming increasingly popular due to their isolation properties, low overhead, and efficient packaging of execution environment. Containers are created from images which preserve software dependencies, environment configuration, and other parameters that affect application's runtime. Docker registry stores images and allows clients to push images to and pull images from it. The total amount of images stored in Docker Hub registry is 457,627 now and is continuing to increase. As the amount of images stored in public and private Docker registries increases it becomes important to study images' characteristics. Such knowledge can help to improve storage performance both at Docker registry and Docker client sides.

Our goal is to collect statistics from a large amount of Docker images and perform a large-scale characterization of Docker images. So far, we downloaded 143,784 images from Docker Hub registry (20TB, 31% of the whole dataset) and performed the first in-depth analysis for these images. We characterize images using multiple metrics, e.g., image size distribution, file size and type distribution, the number of layers per image, the amount of redundant data between images and layers, temporal trends.

1 Introduction

outline:
1. container and registry are becoming populary
2. but we dont know the statistic characterization about registry
3. This paper provide a first in-depth analysis
4. contribution: interesting findings

2 Background

TODO:
1. complete:
2. search for reference

Hypervisors based server virtualization technologies (e.g., VMware [?], Xen [?], and KVM [?]) have been extensively used by most of cloud platforms such as Amazon EC2, which consists of a virtual machine monitor (VMM) on top of a host operating system that allows dynamically partitioning of a machine and sharing the available physical resources such as CPU, storage, memory and I/O devices to support the concurrent execution of multiple guest operating systems instances within virtual machines (VMs) and provide users with benefits ranging from application isolation through server consolidation to improve disaster recovery and faster server provisioning [?].

Nevertheless, hypervisor-based virtualization has a high performance overhead because of execution time overhead caused by executing privileged instructions and memory overhead caused by running multiple VMs. Guest OSs are normally executed at a reduced privilege level []. Hypervisor intercepts traps from guest OSs and emulates the trapping instructions, which incurs execution time overheads, specially for applications which relies on I/O operations since I/O interrupt overhead is amplified for nested virtual machine. Memory overhead includes space reserved for the VMs buffer and various virtualization data structures, such as shadow page tables [], which increases with the number of virtual CPUs and the configured memory for the guest OSs [].

Recent container-based virtualization (such as Linux Containers(LXC) [?], OpenVZ [?], and Docker []) emerges as a lightweigh virtualization, which promises a near-native performance. As opposed to virtual machines (VMs), container based virtualization works at operating system level and do not emulate another operating system. In this case, all the virtual instances share a single operating system kernel, which significantly reduces the overhead imposed through VMs. LXC offers isolation (of PIDs, IPCs, mount pints, and network) through (PID and network) *namespaces* while manages resource and controls processes via *cgroups* [].

Docker container is a new popular container-based virtualization technology that extends LXC with higher level APIs and additional functionality. It also uses namespaces to isolate applications inside containers. Moreover, Docker incorporates copy-on-write union filesystems (UnionFS) to avoid duplication and enable versioning. It couples the above two components with a number of features, like portability, re-use, and reproducibility.

2.1 Docker & Docker container

1. TODO: add a docker architecture pic

Docker containers create a wrapped, controlled environment on the host machine in which applications can be run in isolated manner via two main Linux kernel features – kernel namespaces which are used to split the view that processes have of the system and control groups (cgroups) that restricts the resource usage of a process or group of processes. Currently, Linux kernel provides six different namespaces, PID, IPC, NET, MNT, UTS, and USER for process IDs, IPC requests, networking, file-system mount points, host names, and user IDs [**?**] [**?**]. Controlled resources include CPU shares, RAM, network bandwidth, and disk I/O.

As Figure **??** shows, Docker ecosystem includes various components, i.e., Docker daemon, Docker container, Docker image, and Docker Hub. Docker Deamon, known as Docker engine, can create images from Dockerfiles (through docker build), launch containers (through docker run), and fetch non-local images from Docker registry as well as publish new images to Docker registry such as Docker Hub (through docker pull or docker push). It controls isolation levels of containers including cgroups, namespaces, capabilities restrictions, and SELinux/Apparmorprofiles, monitor them to trigger actions such as restart, and spawn shells into running containers for administration purposes.

Docker container is created from Docker images which package an application with its runtime dependencies, such as binaries, and libraries. Images are read-only copies of file system data. All modifications to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state [].

2.2 Docker image & Docker storage drivers

2.2.1 Docker image

Docker images are composed of a set of individual layers along with metadata in the JavaScript Object Notation (JSON) format called manifest.

1. Layers.
2. Manifest.

2.2.2 Copy-on-write storage

1. aufs.
2. zfs.
3. overlay/overlay2.
4. devicemapper.

2.3 Docker Registry

The Registry is a stateless, highly scalable server side application that stores and lets you distribute Docker images.

1. Registry versions.
2. Content addressability.
3. Pulling images/pushing images.

3 Methodology

TODO:
1. Complete fig and table

Our methodology has two steps. The first step is to massively download the Docker images from Docker registry. When the images are downloaded, we analyze them and calculate statistics distribution for different metrics. The details of each step are covered in the following sections.

3.1 Downloader

Instead of using Docker (or Docker Engine) to download images, we wrote our own downloader python script that utilizes Docker Registry API [] to simultaneously download **original** manifests and layers. There are two reasons. First, Docker Engine (starting from version 1.10) automatically converts the manifests from schema version 1 to schema version 2, which affects our results about manifest version statistics; Second, layer content directories are not visible for some Docker storage drivers, e.g., devicemapper, which is not feasible to analyze the layer content.

Downloader can download multiple images simultaneously and within each image downloading process, layers are downloaded in parallel. To download the original manifests and layers from Docker registry, downloader embeds a Docker registry client API [**?**] which only encapsulates manifest, config file and layer downloading functions in Docker engine without extracting layer tarball and converting manifest version.

To download an image, the name should be provided. To the best of our knowledge, Docker Hub (i.e., Docker registry) doesn't provide a method to list their public images. Public images in Docker registry can be divided into official images and non-official images. The amount of official images is only xxx. While estimating and listing all the non-official images requires crawling Docker Hub. We created Crawler python script to crawl Docker Hub websites and list both official and non-official images.

3.1.1 Crawler

Docker Hub website provides search engine which indexes public images for users to search for a specific image or a list of images that contains a certain letter or string. The name of non-official public image is comprised of "$\langle namespace\rangle/\langle repositoryname\rangle$", where *namespace* is the user name. In this case, we search for '/' and obtain a list of images which contains '/'. In other words, this method lists all the non-official public images in Docker Hub. Crawler downloads all the pages which contains '/'. Once web pages are downloaded, it parses the web content and build a list of non-official images.

A interesting observation is that Crawler can get a similar list of images if we replace '/' with '*'. Note that from 5/30/2017-7/11/2017, Crawler used above method to obtain the total amount of images in Docker Hub. But after 7/11/2017, the Docker Hub removed the index of '/'. Thus, currently we search for '*' instead of '/' to obtain a list of non-official public images.

3.1.2 Downloading images

As shown in Figure **??**, downloader first obtains a list of public images through crawling Docker Hub. Then, it starts downloading process. It downloads two components: manifest and individual layer files.

The first step in downloading an image is to fetch the manifest by using the following url: $GET/v2/\langle name\rangle/manifests/\langle reference\rangle$, where *name* parameter refers to $\langle namespace\rangle/\langle repositoryname\rangle$. Note that the *namespace* of official is *library*. The reference can include a tag or digest. Note that we only downloaded the images with *latest* tag to shorten the downloading process.

Layers are stored as compressed tar archive in the registry, indexed by digest. As discussed in Section **??**, manifest consist of multiple layer digests. Note that Schema 2 version also contains a config file digest. Once the manifest is downloaded, the downloader will then use the digests to download individual layers (including config file for Schema 2 version) by using the following url: $GET/v2/\langle name\rangle/blobs/\langle digest\rangle$, where *name* refers to the image name while *digest* refers to the layer digest or config file digest.

1. TODO add a figure, downloader and analyzer

3.1.3 Docker image dataset statistics

Crawler delivered a list of xxx images on 5/30/2017. However, duplicated images exist in the list. After reducing the repeated images, our image dataset consists of xxx distinct images.

The downloading process took roughly 30 days to finish. Overall, we downloaded xxx TB of xxx images with xxx layers as shown in Table **??**. xxx of images couldn't be downloaded. There are two reasons: first, xxx of images were either deleted or empty. Second, xxx of images doesn't have tag: *latest*. As we discussed in Section **??**, we only downloaded the images with latest version to shorten the downloading process.

2. TODO: add a table discribe:
how many images, duplicate ratio
how many cannot download, (removed, no latest)
how many layers, config, manifest
dataset

3.2 Analyzer

Analyzer analyzes the images we downloaded and creates two kinds of files for each image: image profile and individual layer profiles. Image profile includes image metadata information, such as image pull count, layer count, etc., and image configuration information, such as os, architecture, etc.; While layer profile contains layer metadata information, such as layer size, file count, etc.;

and directory metadata information for each subdirectory, such as directory depth, directory size, etc.; and file metadata information for each file, such as file size, file type, etc.

3.2.1 Layer profile

As discussed in Section **??**, the layers we downloaded are compressed tar archive files. To analyze the layer content, analyzer first decompress and extract each layer tarball to a layer directory. Then, analyzer recursively goes through each subdirectory and obtains the metadata information for each subdirectory and each file as shown in figure **??**.

3. TODO: add a fig: discribe all layer metadata, config metadata, and manifest metadata structure

4. TODO: add a table
layer tarball format statistics, tar, compressed, non compressed
config statistics, txt, json
manifest statistics, txt, json

3.2.2 Image Profile

As shown in figure **??**, Analyzer parses the manifest and obtains the configuration information such as os, architecture etc.. Note that manifest Schema version 2 stores configuration information in a config file as discussed in Section **??**. As shown in table **??**, xxx of manifests are Schema version 2 while the rest are Schema version 1.

Once individual layers are analyzed, analyzer can build the whole image profile by including pointers to its layer profiles as shown in figure **??**. Table **??** summaries the layer archive file, config file, and manifest statistics.

4 Results

4.1 Image

4.1.1 Growth of images at docker registry

4.1.2 Image size distribution

4.1.3 Image popularity distribution

4.1.4 Layer depth distribution

4.1.5 Repeat layer count distribution

4.1.6 File count distribution

4.1.7 Repeat file count distribution

4.1.8 Compression rate distribution

4.1.9 Other metrics, exectution envirnoment

4.2 Layer

4.2.1 Layer size distribution

4.2.2 File count distribution

4.2.3 Directory count distribution

4.2.4 Directory depth distribution

4.2.5 Repeat file distribution

4.2.6 File size distribution

4.2.7 File type distribution

4.2.8 Compression rate distribution

4.2.9 Other metrics, layer age

5 Relatedwork

do we need a relatedwork?

6 Conclusion