Docker Data Storage

Florent Glück - florent.gluck@hesge.ch

April 15, 2025

ISC - HEPIA

Docker images

- Docker **images are root filesystems (rootfs) for containers**
- Docker images:
  - are **immutable (read-only)**
  - **should be minimal**: only include an application and its dependencies
  - do not need a kernel + modules: containers share the host kernel
  - do not need initialization tools or scripts
- Images are portable and can be shared, stored and updated

Image inheritance

- New images can be created from existing images
- Images are usually created from images of well-known Linux distributions (e.g. Ubuntu, Alpine, Debian, etc.)
- Starting from an existing image $\longrightarrow$ easy and no significant overhead
- Images can also be created from scratch (from an archive)
  - such images are called **base images**

Image composition

- **Images are layered = made of different stacked layers** that can be re-used by other images and shared by containers
- Every image **extends** a **parent image** (its first layer)
- Layers are created from Dockerfile instructions: `RUN`, `COPY`, `ADD`

Image layers

A layer is a collection of files and directories

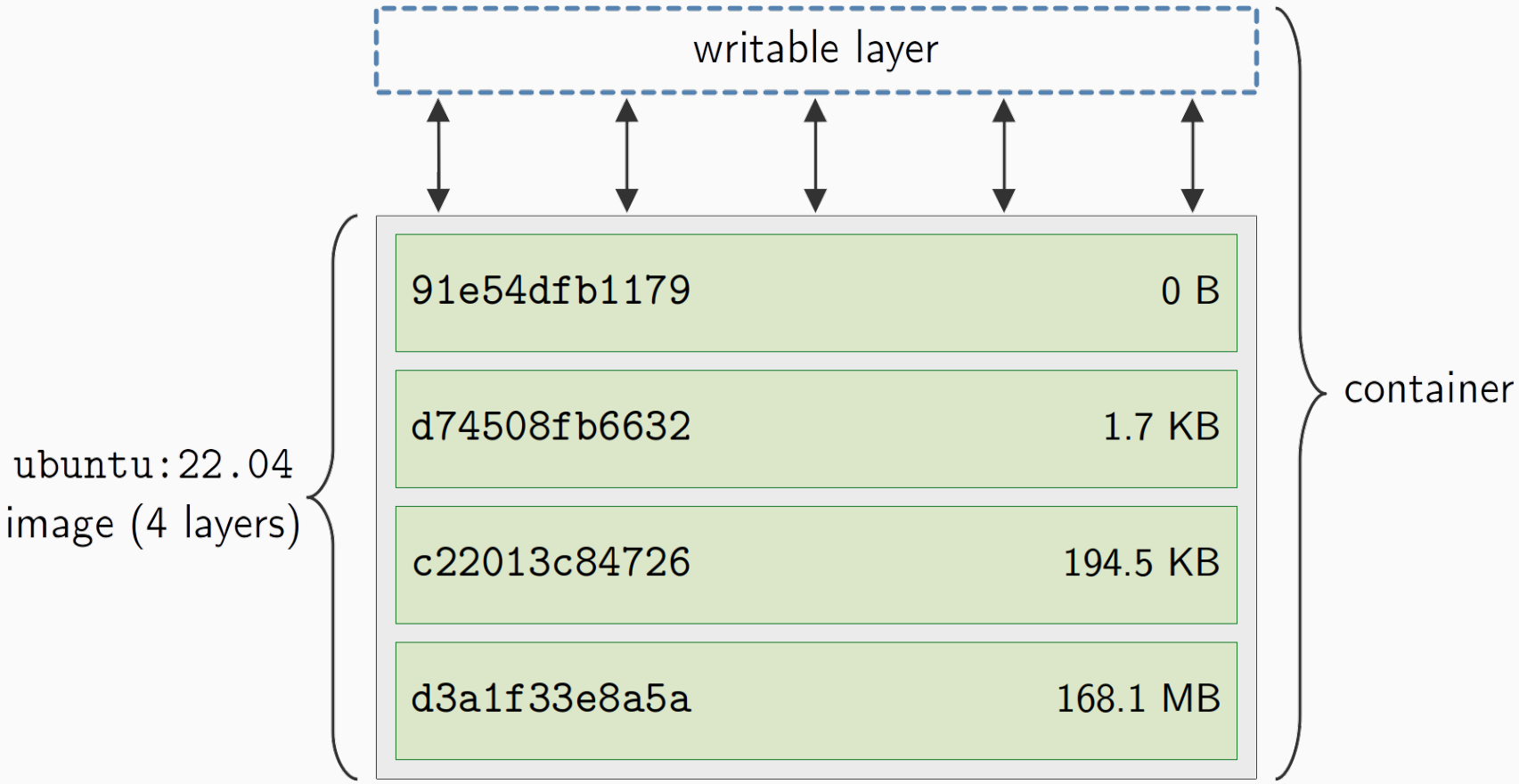
- Each layer is immutable (read-only)
- Each layer represents a delta of the changes from the previous layer
- Each layer is associated and referenced by a **hash generated from the layer's content**
  - allows `dockerd` to know whether a layer has already been downloaded
- When downloading an image ⟶ each missing or *non-up-to-date* layer is downloaded separately

Images, layers and containers (1/3)

When a container is created, a new **writable layer is added on top of the image**

- This top layer is:
  - called the **container layer**
  - initially empty
- All changes made in a running container (write, modify, delete files, etc.) are **written to the writable layer**

Images, layers and containers (2/3)



Images, layers and containers (3/3)

- Each container has its **own writable** container layer where changes are stored
- Multiple containers running the same image **share the same read-only underlying layers**

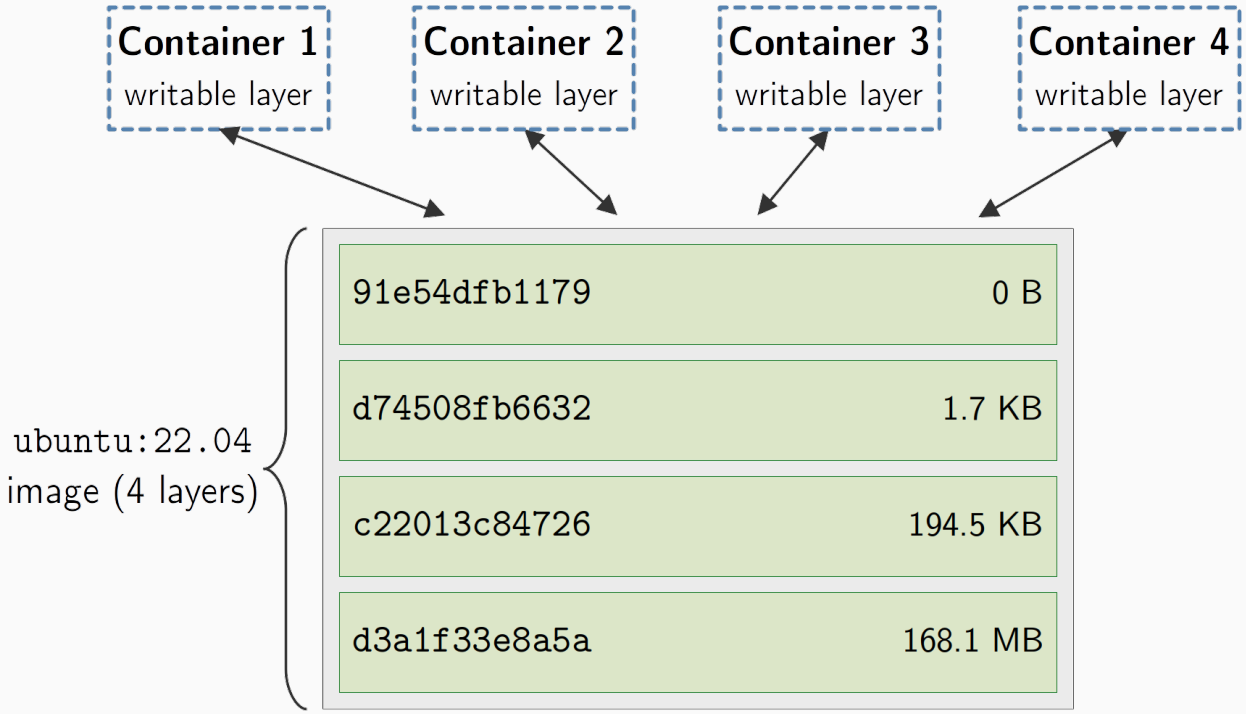


Image vs container

What is the **major difference** between a **container** and an **image**?

- **The top writable layer!**

Additionally:

- All file modifications (additions, deletions, modifications) of a container are **only** stored in the writable layer
- When a container is deleted $\longrightarrow$ only its writable layer is deleted
  - but the underlying immutable image (layers) **remains!**

Inspecting a container's layers (1/2)

- Layers composing a container's rootfs can be determined with:

```
docker inspect <container>
```

- Inspect the `"Data"` field under the `"GraphDriver"` node:
  - `LowerDir` defines the immutable image layers (also called "lower directories")
  - `UpperDir` defines the container writable layer (also called "upper directory")
  - `MergedDir` defines the unified view of all layers, i.e. the container's rootfs (also called "merged directory")

Inspecting a container's layers (2/2)

Example, inspecting layers of container **nginx:1.27**:

```
docker inspect --format "{{json .GraphDriver.Data}}" nginx:1.27 | aeson-pretty
```

```
"LowerDir":
"/var/lib/docker/overlay2/5d8248138b1d493c0abedc31c05dd6df4f7e5bedcd8f83ab66f493b9f7ee7fe1/diff:
 /var/lib/docker/overlay2/1dc5c7e86993317073aa97b3696162c0fd5f51a13e0fe1aa0df30844e6d7a04f/diff:
 /var/lib/docker/overlay2/1ba1551810e5c914317b7f94eb170bd9462c2c2f41457b165341fdbfc6935dcc/diff:
 /var/lib/docker/overlay2/1cd81882657893b678faa3dbd0945fb3e694a95a88e3368a4f7387f3b47d65ea/diff:
 /var/lib/docker/overlay2/5d2bc99960ac393db06c988866803e2ba4217294d1f065c6d084f18720a8e5d9/diff:
 /var/lib/docker/overlay2/ccc1998b60ebc3ec952dddb4e76b2542964724d1d46c37635770e468f83bb51a/diff",

"UpperDir":
"/var/lib/docker/overlay2/a5f552406943edb1b7a456a10aec1155c7437e4e86a0f13b73b9fb6cb3f701ed/diff",

"MergedDir":
"/var/lib/docker/overlay2/a5f552406943edb1b7a456a10aec1155c7437e4e86a0f13b73b9fb6cb3f701ed/merged",

"WorkDir":
"/var/lib/docker/overlay2/a5f552406943edb1b7a456a10aec1155c7437e4e86a0f13b73b9fb6cb3f701ed/work"
```

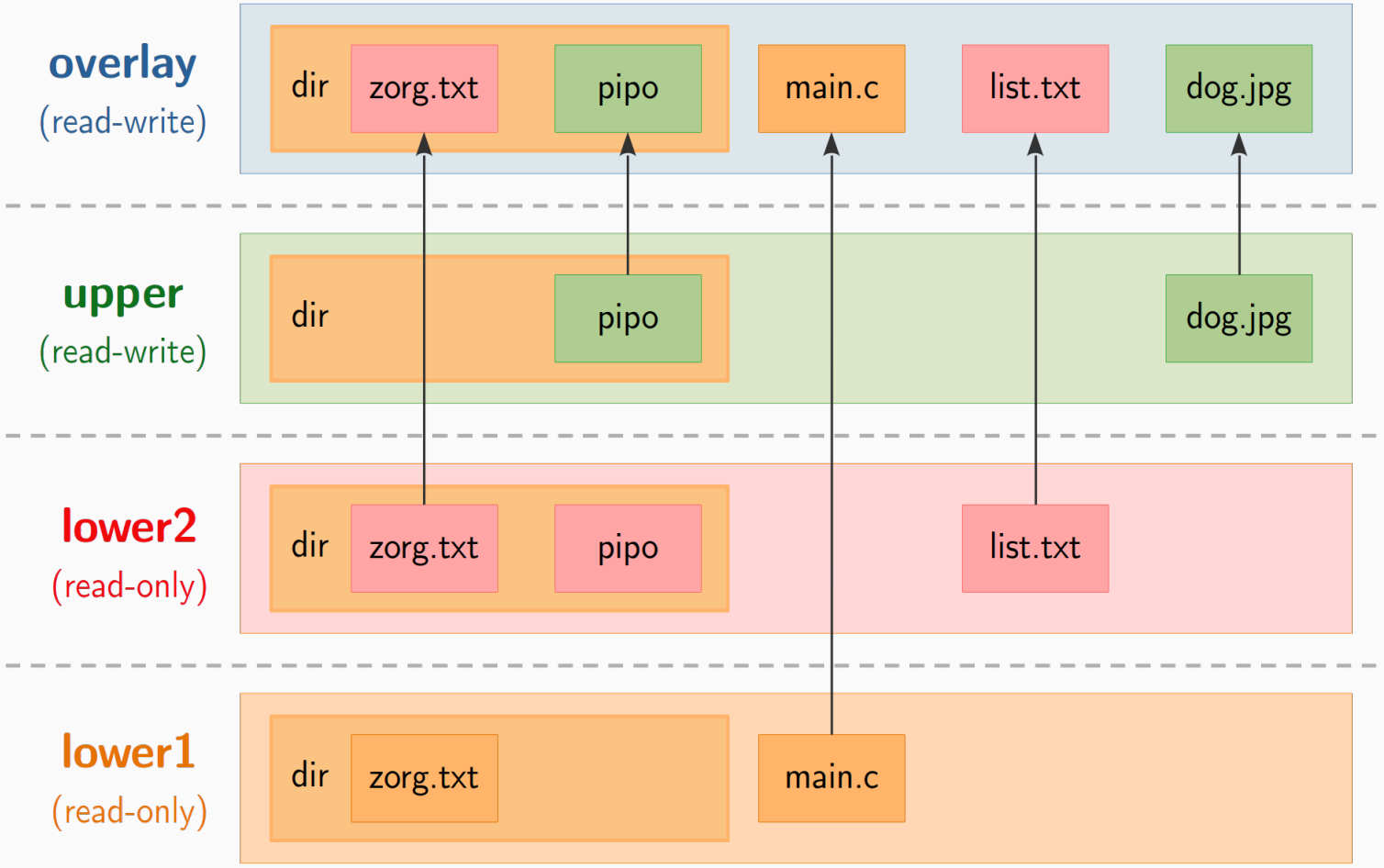
Docker storage drivers

- Docker uses a storage driver to present a rootfs composed of immutable image layers + a writable layer
- The default storage driver is overlay2
- Overlay2 uses Linux kernel's **overlay filesystem**
- Layers **contents** are stored in `dockerd` daemon local storage directory in `/var/lib/docker/overlay2/`

Overlay filesystem

- **Merges** multiple **lower** file tree layers with a single **upper** file tree, then **presents a unified view** of these file trees in an **overlay** file tree
- Behavior when the same file/directory name exists in multiple trees:
  - the one in the highest layer is exposed in the **overlay** layer
- The **lower** layers are accessed in read-only
- The **upper** layer is accessed in read-write
- The **overlay** is accessed in read-write as a mounted filesystem

Overlay filesystem: behavior example



Overlay filesystem: usage

```
mount -t overlay LABEL
      -o lowerdir=LOW_DIRS,upperdir=UPPER_DIR,workdir=WORK_DIR MERGED_DIR
```

LABEL	arbitrary label to name the filesystem (visible in mount)
LOW_DIRS	list of lower directories, separated by :
UPPER_DIR	upper directory
WORK_DIR	working directory; required by kernel; must be empty and on the same filesystem as UPPER_DIR
MERGED_DIR	merged or *overlay* directory

Overlay filesystem: mount example

Example of an overlay filesystem with 3 lower dirs (`low1`, `low2`, `low3`), an upper dir in `/upper` and presenting the overlay dir in `/merged`:

```
mount -t overlay some_label -o lowerdir=/low3:/low2:/low1,upperdir=/
upper,workdir=/work /merged
```

- In this example, directories are ordered as follow:

```
/upper  # highest
/low3
/low2
/low1   # lowest
```

In a container, `merged` would be the container's rootfs (from the user's point of view), `upper` the writable layer, and `low1`, `low2`, `low3` the image layers

Committing changes

- **`docker commit`** commits the current filesystem state of a container into an image file
  - **current state** of a container = immutable layers + top writable layer
  - useful when modifying a container by hand and wanting to make these changes permanent
  - better to use dockerfiles, but `commit` is useful for testing and preparing
- **`docker diff`** useful to display filesystem changes between a container and its image (similar to `git diff`)

Writable layer & performance

- **Issue with overlay filesystem**: reading and writing in container's writable layer is **slower** than on native filesystem
- Ideally, very little data should be written to a container's writable layer
- Use Docker volumes[1] for write-heavy workloads instead of the container's writable layer
- **Volumes** write directly to the host filesystem $\longrightarrow$ **better performances than writing to the writable layer!**

[1] More on volumes in the next slides

Data sharing

- Files created inside a container are stored on a writable container layer:
  - data not persistent when container is deleted
  - can be difficult to get the data out of the container
- How to share data between host and container?
- How to share data between multiple containers?
- Two possibilities:
  1. **bind** mount
  2. **volume** mount

(1) Bind mount

- Container can read-write files outside the container's writable layer
- A directory on the **server** (where `dockerd` runs) is mounted into a container
- The file or directory is **referenced by its full absolute path** on the server
- Efficient, but rely on the host machine's filesystem having a specific directory structure available (*mount point*)
- **Only works** if client and server run on the **same host**!
- Example:

```
mkdir shared_mount
docker run --mount type=bind,src=$(pwd)/shared_mount,dst=/shared alpine:3.21
```

(2) Volume mount

- Container can read-write files outside the container's writable layer
- A volume (possibly remote) is mounted into a container
- The volume is **referenced by its name** on the **server** machine
- Volumes are **fully managed** by Docker
- **Client and server can be on different hosts!**
- Example (more info with `docker help volume`):

```
docker volume create my_vol
docker run --mount type=volume,src=my_vol,dst=/shared alpine:3.21
```

Volumes: tricky behavior

- Mounting a non-existing volume automatically and implicitly creates it!
- Mounting an empty volume in a non-empty directory in the container copies the directory's files into the volume!

Volumes vs bind mounts

Benefits of volumes over bind mounts

- Volumes manageable via Docker CLI or Docker API
- Work on both Linux and Windows containers
- Can be stored on **remote** hosts
- Support encrypted contents, etc.

Volumes vs writing to the container writable layer

Volumes are often a better choice than persisting data in a container's writable layer:

- **Contents exist outside the container's lifecycle!**
  - contents remain when container is deleted
- **Better read-write performance**
- Does not increase the container's size

Transfering data to/from a volume

- How to copy data from:
  - the client's filesystem to a volume (on the server)?
  - a volume to the client filesystem?
- Copy content of the current directory on the client to the volume mounted in `/shared` in the container:

```
docker cp . my_container:/shared/
```

- Copy volume's content (mounted in `/shared` in the container) to the current directory on the client:

```
docker cp my_container:/shared/ .
```

Resources

- Docker storage documentation
  https://docs.docker.com/storage/
- Docker storage drivers documentation
  https://docs.docker.com/storage/storagedriver/
- The Overlay Filesystem
  https://windsock.io/the-overlay-filesystem/
- Julia Evans on containers & overlayfs
  https://jvns.ca/blog/2019/11/18/how-containers-work--overlayfs/
- OverlayFS Linux kernel documentation
  https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt