Docker Data Storage

Florent Gluck - Florent.Gluck@hesge.ch

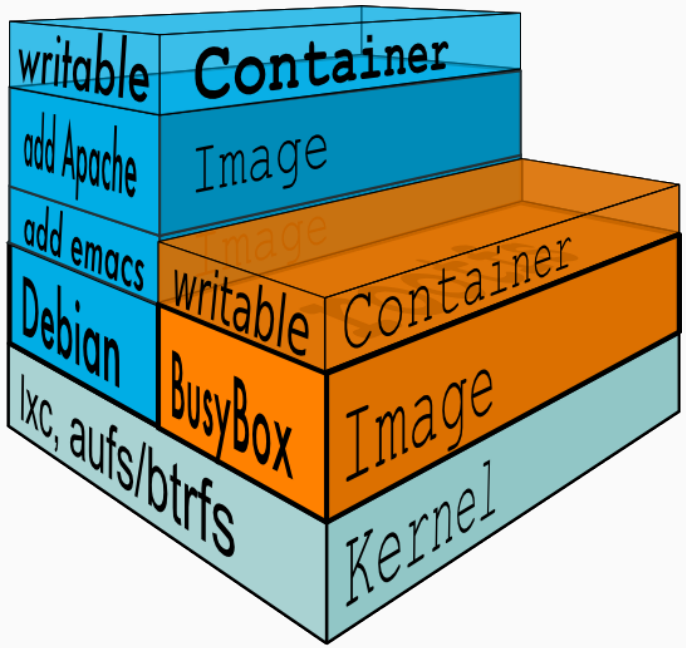
April 7, 2022

Docker images

- Docker images are akin to a root filesystem for containers:
  - they do not need a kernel + modules: containers share the host kernel
  - they do not need initialization tools or scripts
  - usually minimal: only includes dependencies (libs) to run the application(s)
- Images = ***source code*** for containers
  - portable and can be shared, stored and updated
- **Images are layered** = made of different *stacked* layers, so that lower layers can be reused and shared

Images' layers (1/2)

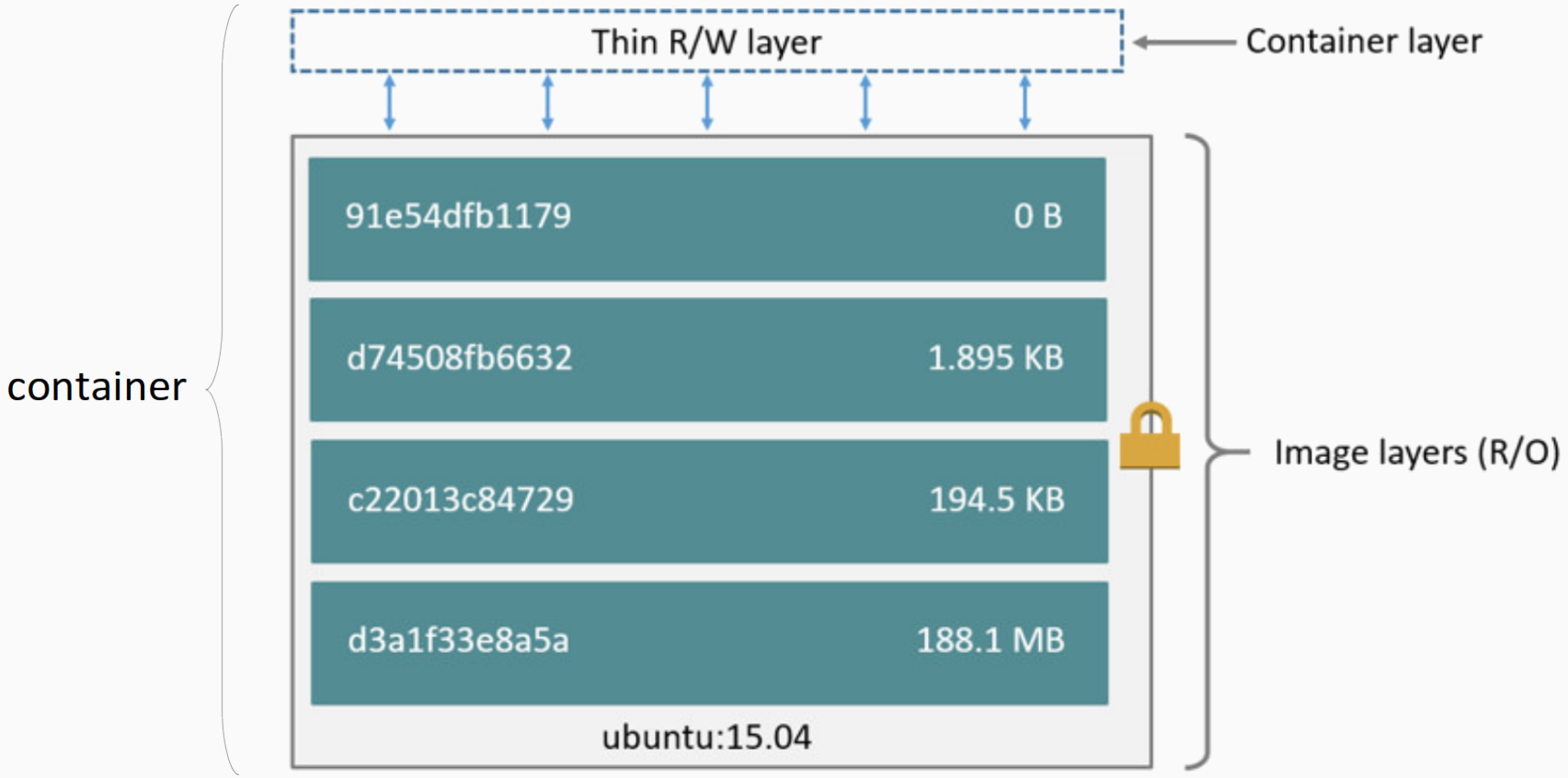
- Images are built up from series of layers, created using series of instructions, e.g.
  - add a file
  - run a command
- **Each layer represents changes from the underlying layer**
- Layers are immutable and referenced by hashes and optional tags
- Containers use the filesystem from an image + a top read-write layer



Images' layers (2/2)

- Layers are stacked on top of each other
- **When a new container is created → a new writable layer is added on top of the underlying layers**
- This top layer is called the *container layer*
- Underlying layers are **read-only**
- All changes made in a running container, e.g. write, modify, delete files → written to the writable layer

Layers illustrated

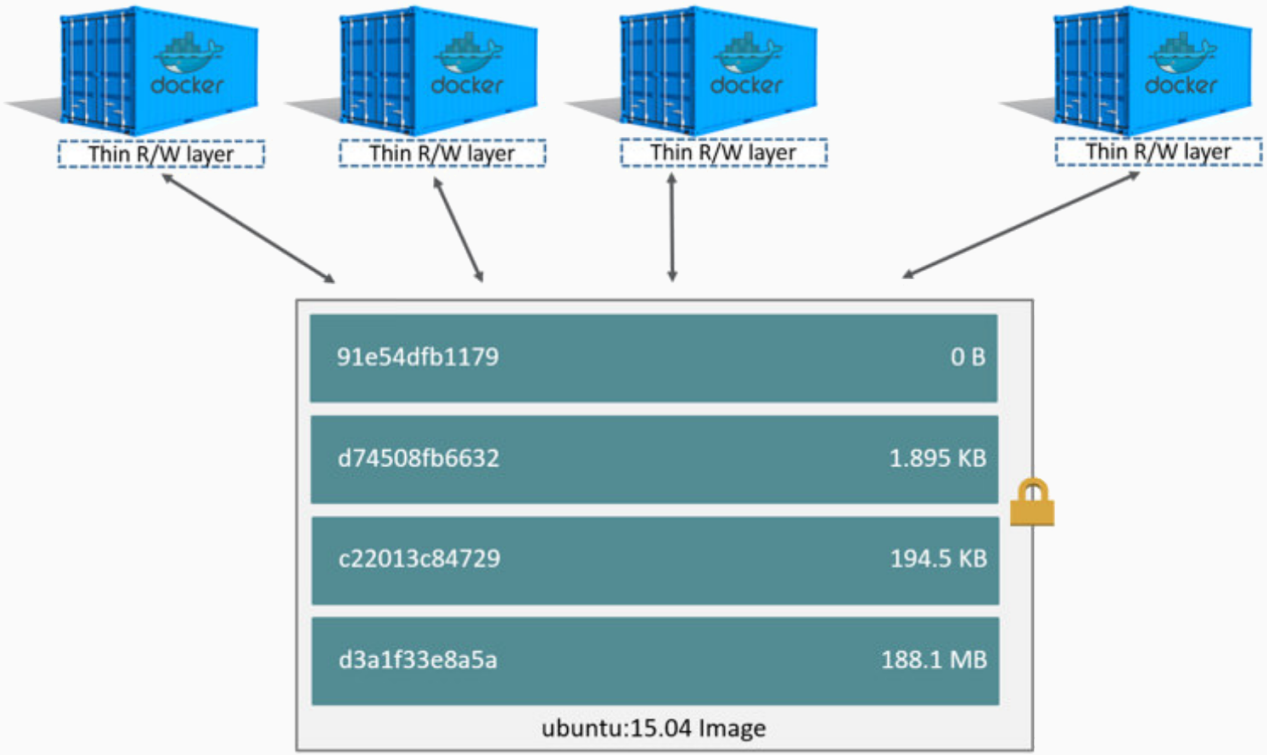


Container layer (1/2)

- **A major difference between container and image = the top writable layer**
- All file modifications (additions, deletions, modifications) of an image are stored in the top writable layer
- When container is deleted $\rightarrow$ writable layer is deleted
  - however: **underlying image remains unchanged!**

Container layer (2/2)

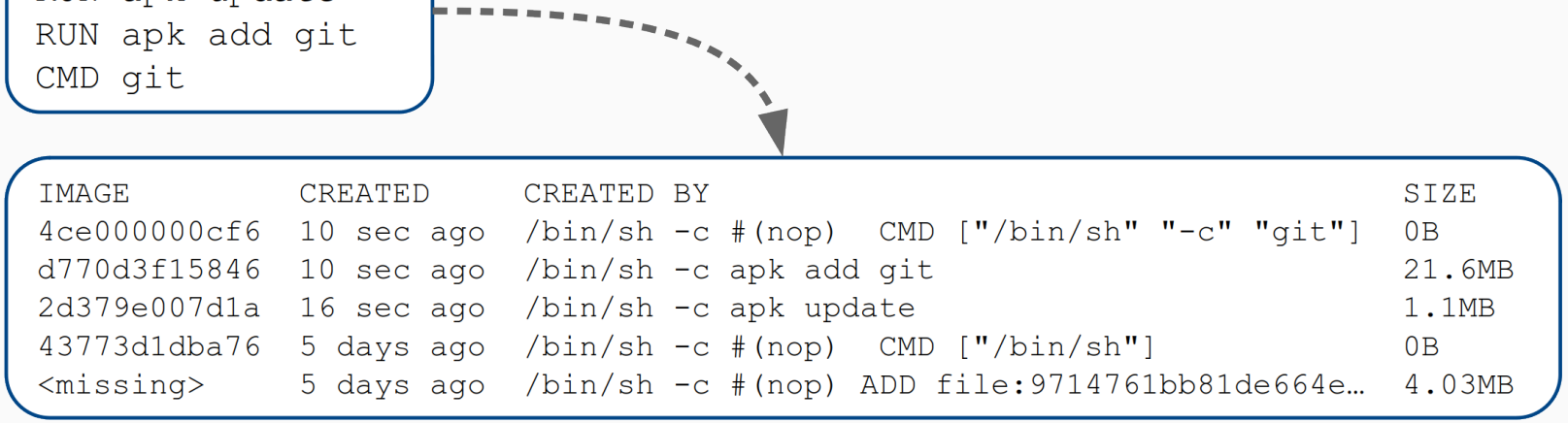
- Multiple containers running the same image **share the same read-only underlying layers**
- Each container overlays its **own read-write** container layer
- All changes are stored in this read-write container layer



Images and layers

- A Docker image is built up from a series of layers
- Example of a very simple Dockerfile:

```
FROM alpine:3.6
RUN apk update
RUN apk add git
CMD git
```



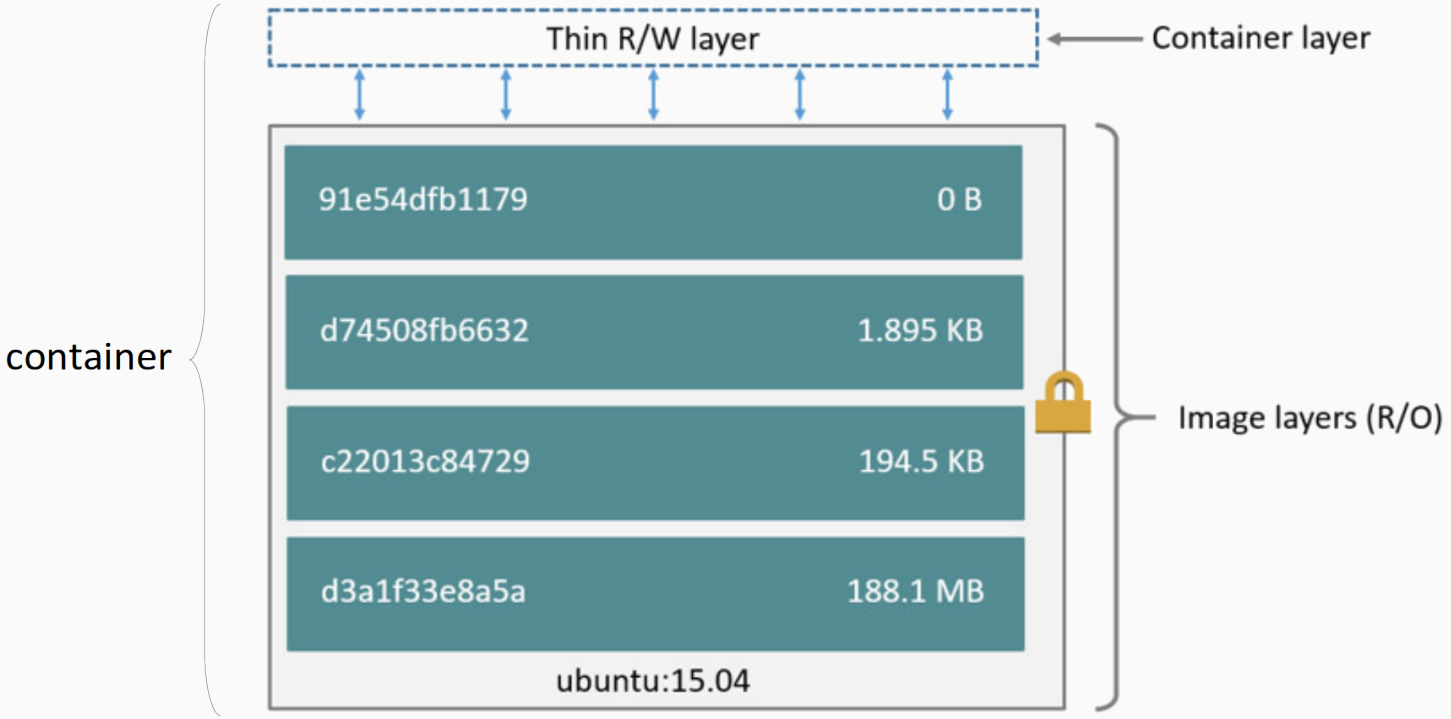
```
IMAGE          CREATED      CREATED BY                                      SIZE
4ce000000cf6   10 sec ago   /bin/sh -c #(nop)  CMD ["/bin/sh" "-c" "git"]   0B
d770d3f15846   10 sec ago   /bin/sh -c apk add git                          21.6MB
2d379e007d1a   16 sec ago   /bin/sh -c apk update                           1.1MB
43773d1dba76   5 days ago   /bin/sh -c #(nop)  CMD ["/bin/sh"]              0B
<missing>      5 days ago   /bin/sh -c #(nop) ADD file:9714761bb81de664e…   4.03MB
```

- Dockerfile commands often create layers
- **Each layer = set of differences from previous layer**

Docker storage driver

- How Docker builds and stores images?
- How are these images used by containers?
- Storage drivers allow to create data in container's writable layer
- Files in the container **do NOT persist** after container is deleted
- Read and write speeds in container's writable layer are **slower** than on native filesystem

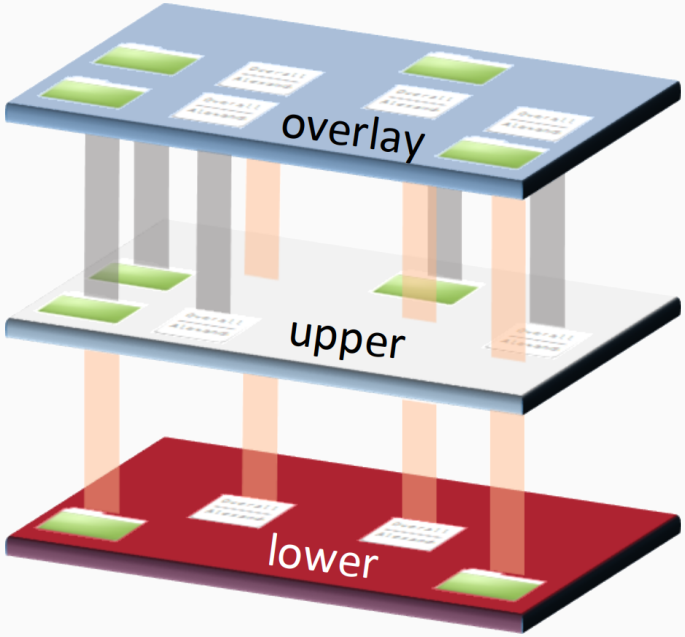
Images and layers



- A storage driver handles details about the way these layers interact with each other
- Default storage driver $\rightarrow$ overlay2 (overlay filesystem v2)

Overlay filesystem

- **Combines** upper and lower directory trees and **presents a unified view**
- Behavior when same name files/dirs exist in both directory trees:
  - files: only files in upper directory tree visible in overlay
  - dirs: upper and lower dirs contents merged and visible in overlay
- Might be several lower directories
  - Lower dirs are usually read-only
- Upper dir is usually writable



Overlay filesystem: usage

Exemple of an overlay using 3 lower layers (`low1`, `low2`, `low3`), an upper directory in `/upper` and the merged overlay in `/merged`:

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

- The working directory (`workdir`) must be empty and on the same filesystem mount as the `upper` directory
- Lower directories (if more than one) are separated by ":"
- In this example, the layer order is the following:

```
/upper
/low1
/low2
/low3
```

Layer sharing

- When downloading a Docker image → each layer is pulled down separately
- Layers stored in Docker's local storage area
- Layers located in `/var/lib/docker/<storage-driver>/`

Image inheritance

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

Writable layer & performance

- Ideally, very little data should be written to a container's writable layer
- Use Docker volumes for write-heavy workloads instead of the container's writable layer
- **Volumes** write directly to the host filesystem $\rightarrow$ **better performances than writing to the writable layer!**

Committing changes

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

Data sharing

- Files created inside a container are stored on a writable container layer:
  - data not persistent when container destroyed
  - 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

Bind mount

- Container can read-write files outside the container's writable layer
- A file or directory on the host machine is mounted into a container
- The file or directory is **referenced by its full absolute path** on the host machine
- Efficient, but rely on the host machine's filesystem having a specific directory structure available (*mount point*)
- Exemple:

```
mkdir shared_mount
docker run --mount
       type=bind,src=$(pwd)/shared_mount,dst=/shared
       alpine:3.12
```

Volume mount

- Container can read-write files outside the container's writable layer
- A volume (local, but possibly remote) is mounted into a container
- **Preferred** mechanism for persisting data generated by and used by Docker containers
- The volume is **referenced by its name** on the host machine
- Volumes are **fully managed** by Docker
- Exemple:

```
docker run --mount
       type=volume,src=my_vol,dst=/shared
       alpine:3.12
```

- Use `docker help volume` on how to manage volumes

Volumes vs bind mounts

- Advantages of volumes over bind mounts:
  - Volumes manageable via Docker CLI or Docker API
  - Easier to backup or migrate
  - Work on both Linux and Windows containers
  - Can be stored on remote hosts (e.g. Cloud), supports encrypted contents, etc.
  - New volumes can have their contents pre-populated by a container

Volumes vs writing to the container rw layer

- Volumes often better choice than persisting data in a container's writable layer:
  - Better read-write performance
  - Does not increase the container's size
  - Contents exist outside the container's lifecycle

Transfering data to/from a volume

- How to copy data from the local filesystem to a volume?
- How to copy data from a volume to the local filesystem?
- Use `docker cp`
- From local filesystem to container:

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

- From container to local filesystem:

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

Resources

- Docker storage documentation
  https://docs.docker.com/storage/
- 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