COMP5349 – Cloud Computing

Week 3: Container Technology

Dr. Ying Zhou
School of Computer Science

Outline

- **Container Brief Intro**

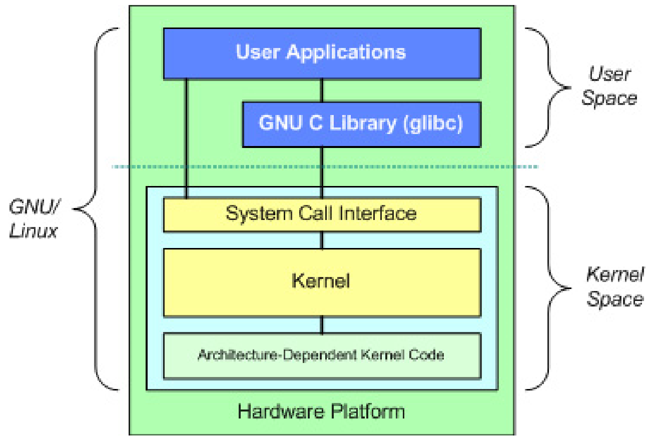
- Docker Overview
  - Images
  - Containers
  - Storage
  - Networking
  - Security

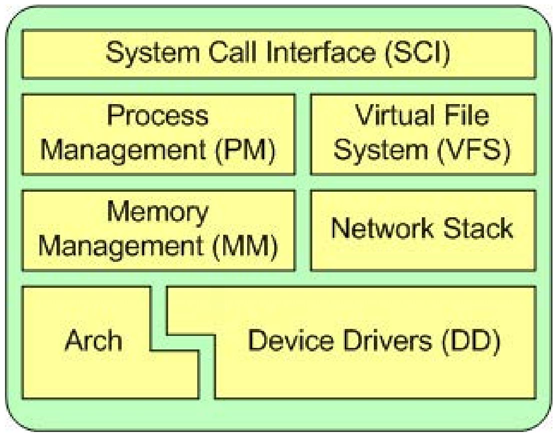
Last Week：Virtualization

- The key motivation for virtualization
  - Maintain isolation, increase utilization
- Two components in server virtualization
  - Hypervisor and virtual machine
- The design principle of hypervisor
  - Similar to OS design principle
    - Kernel and other modules
  - Can be implemented in different ways
- Different options for managing critical instructions
  - Full virtualization: keep OS unchanged
    - With software emulation
    - With hardware assisted execution mode
  - Modify OS
    - Paravirtualization

Containerization

- "**Operating-system-level virtualization**, also known as **containerization**, refers to an **operating system feature** in which the kernel allows the existence of multiple isolated user-space instances."
- "Such instances, called **containers**, partitions, virtualization engines (VEs) or jails (FreeBSD jail or chroot jail), may look like real computers from the point of view of programs running in them."

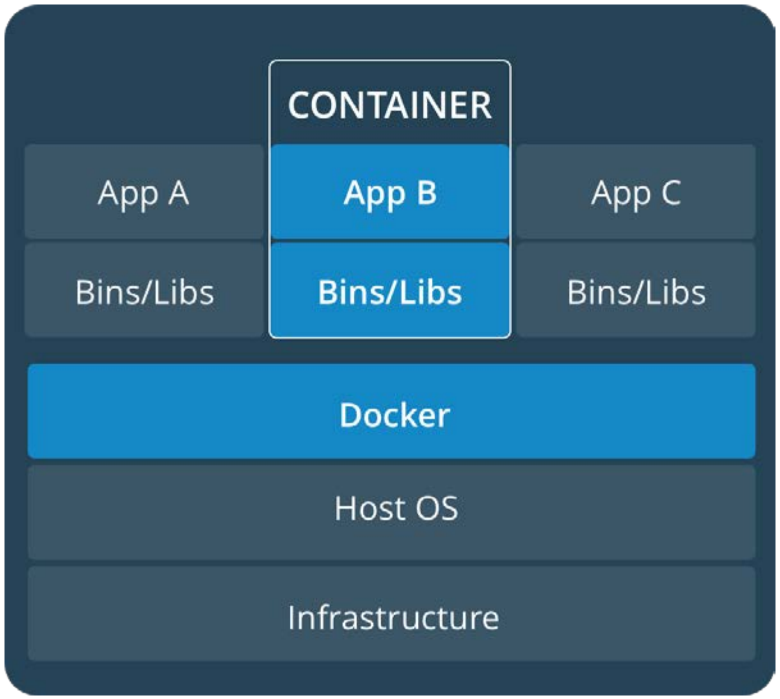


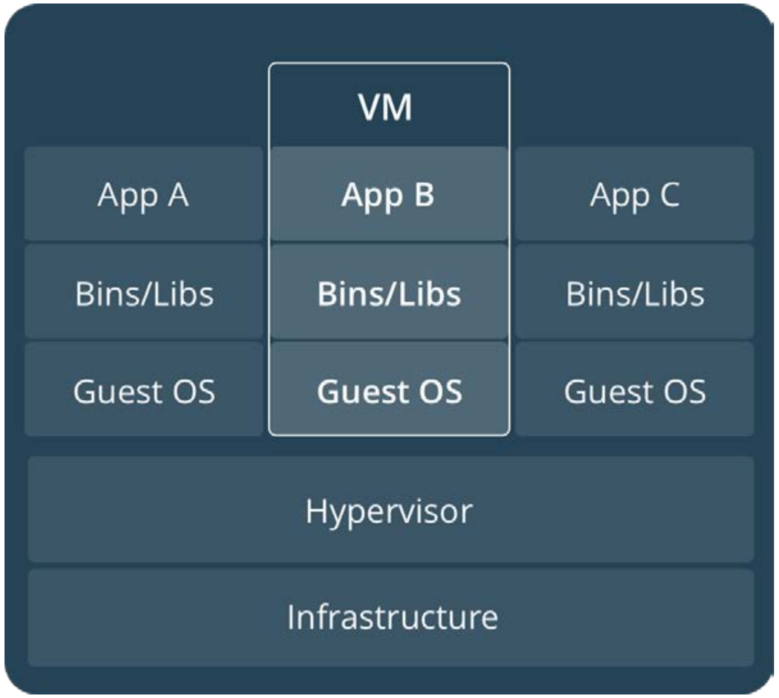


https://www.ibm.com/developerworks/library/l-linux-kernel/index.html

https://en.wikipedia.org/wiki/Operating-system-level_virtualization#Implementations

Container vs. System Virtual Machine





All containers use the kernel of the host machine

Each VM contains a full OS

You cannot run windows container on Linux machine

All Linux distros, share more or less the “same” upstream kernel, running Ubuntu container on Redhat would not cause any drama

Linux Kernel, System and Distribution

- Linux kernel is the most basic component of a Linux Operating System. It does the four basic jobs
  - Memory Management
  - Process Management
  - Device Drivers
  - System Calls and Security
- Linux System
  - Kernel + system library and tools
    - E.g. GNU tools like gcc,
- Linux distribution
  - Pre-packaged Linux system + more applications
    - E.g. news servers, web browsers, text-processing and editing tools, etc.
  - Redhat, Debian, Amazon Linux, Android(?)

Container vs. System Virtual Machine

- Operating Systems and Resources
  - Running full fledged OS inside VM takes up certain amount of resources even if no app is running in the VM
  - Starting OS takes some time
  - Container exits when the process inside it finishes
  - Container is much faster to start, similar to starting an application
- Isolation for performance and security
  - VMs have very good isolation and security
  - They enjoy hardware support and mature technology
  - Containers offer reasonable level of isolation using *kernel techniques like <u>namespace</u> and <u>control groups</u>*
- Containers can run inside VM

Linux Kernel Feature: Namespace

- To create an illusion of full computer system for each container, we need to give each a “copy” of the kernel resources
  - An independent file system starting with a root directory: /
  - An independent set of process ids, with id 1 assigned to *init* process
  - An independent set of user uids, with id *0* assigned to root user
  - Etc..
- To avoid conflict, OS provides a feature called **namespace**
- The namespace provides containers their own view of the system

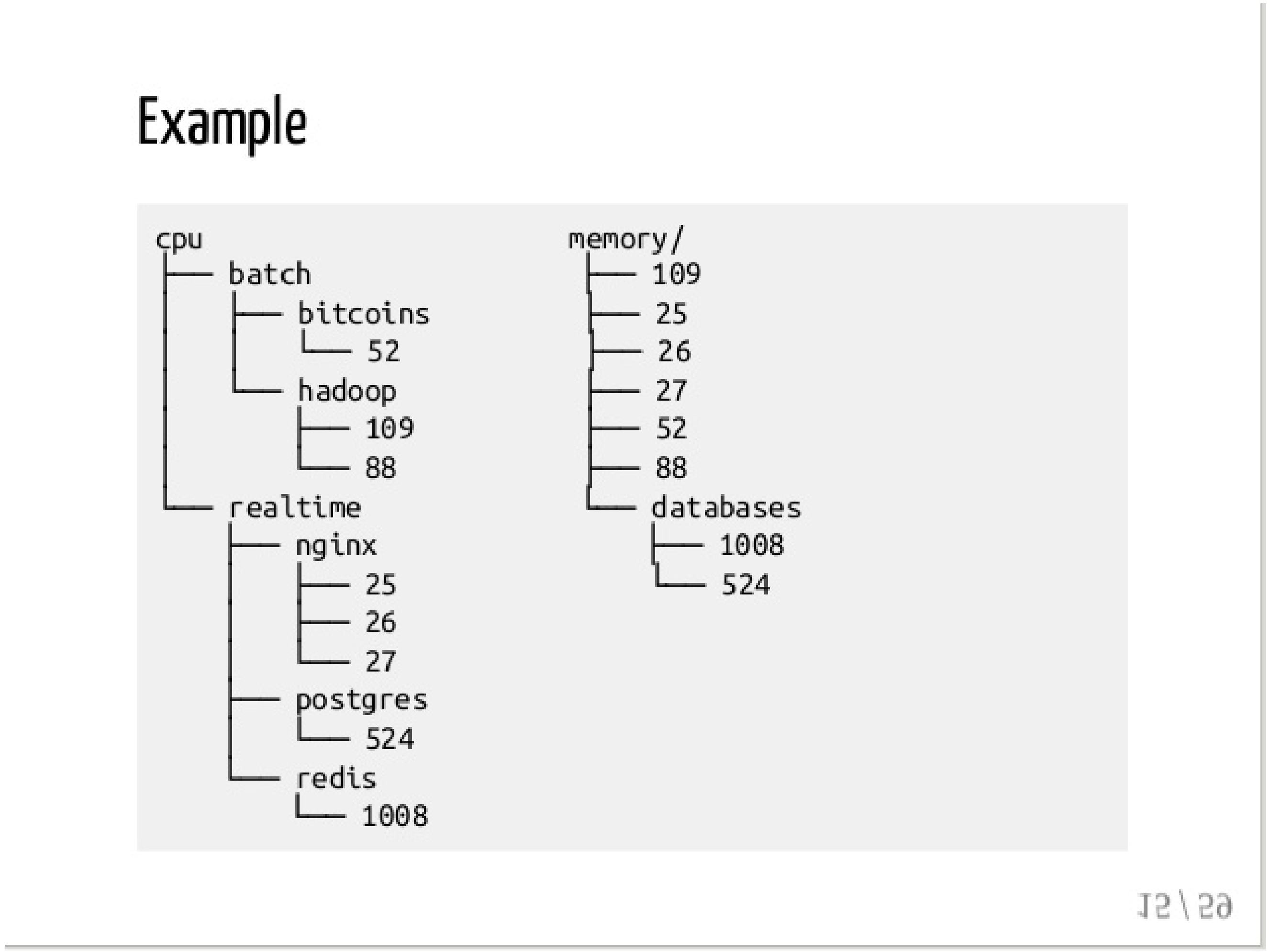
Namespace Kinds

- Early Linux has a single namespace for each resource type
- Gradually, namespaces are added to different resources
  - Mount (mnt)
    - This deals with file system
    - Each container can have its own rootfs
    - Each container manages its own mount points
  - Process ID (pid)
    - Each container has its own numbering starting at 1
    - When PID 1 goes away, all other processes exit immediately
    - PID name spaces are nested. The same process may have different PIDs in different namespaces
  - User ID
    - Provide user segregation and privilege isolation
    - There is a mapping between container UID to host OS UID
    - UID 0 (root) in container may have a different UID in the host
  - Net, IPC, etc..

Linux Kernel Feature: Cgroup

- Control Groups (cgroups) is another kernel feature to enable isolated container
- It is used to control kernel resource allocated to each container/process
  - Metering and limiting
- The resources include:
  - Memory
  - CPU
  - I/O (File and Network)
- Cgroups are organized hierarchically for each resource type
  - Each process belongs to 1 node in each hieararchy

Cgroups Hierarchy Example



Example

```
cpu                                memory/
├── batch                          ├── 109
│   ├── bitcoins                   ├── 25
│   │   └── 52                     ├── 26
│   └── hadoop                     ├── 27
│       ├── 109                    ├── 52
│       └── 88                     ├── 88
└── realtime                       └── databases
    ├── nginx                          ├── 1008
    │   ├── 25                         └── 524
    │   ├── 26
    │   └── 27
    ├── postgres
    │   └── 524
    └── redis
        └── 1008
```

https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon

Container Runtimes

- LXC
- Systemd-nspawn
- OpenVZ
- Sandboxie
- etc…

Outline

- Container Brief Intro
- Docker Overview
  - Images
  - Containers
  - Storage
  - Networking

Docker Overview

- Docker is the most “famous” container
- It is not just a container, it is a packaging and deployment system build on container technology
  - There is a large ecosystem of various components
  - The container part is usually presented as a black box where docker users do not need to know a lot about how it works
  - For most users, the dependency management and deploy everywhere are the most prominent features
- Its success partly relies on good use case

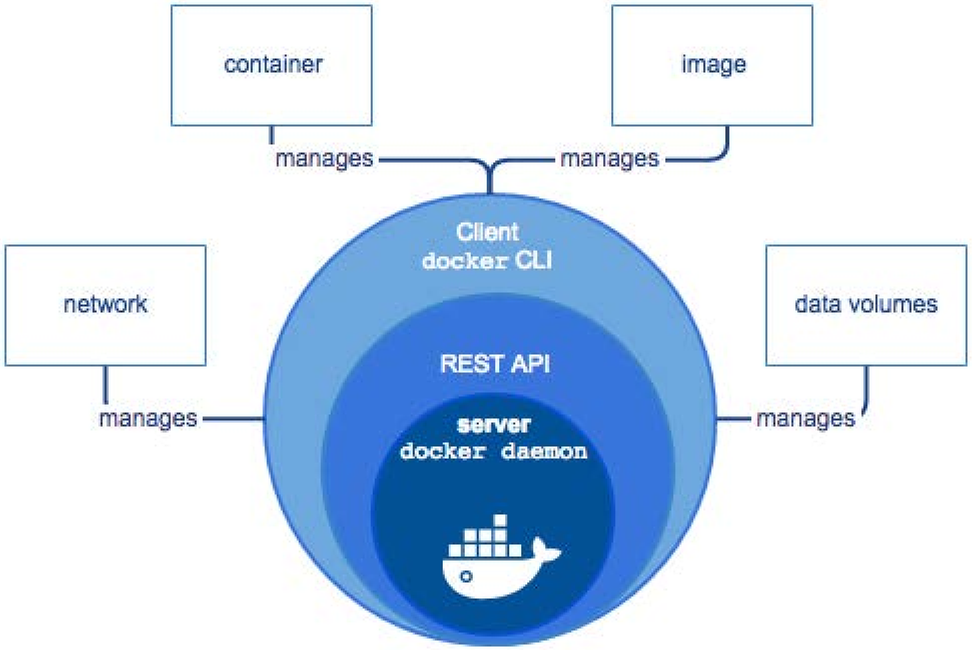
The dependency hell

- Modularization and layering are key principles in computer science .
- But it has an unwanted by product “dependency hell”
  - APIs may change across versions
  - Conflicting dependencies
    - Some app runs on Python 2 while others on Python 3, and some may be very particular on the exact version
    - Alternative solution?
  - Missing dependencies
    - You may install some app successfully, but may encounter problem in execution
    - Alternative solution?
  - Platform differences
    - Development and production have different environments
- Docker is promised to solve all those problems

Docker Components

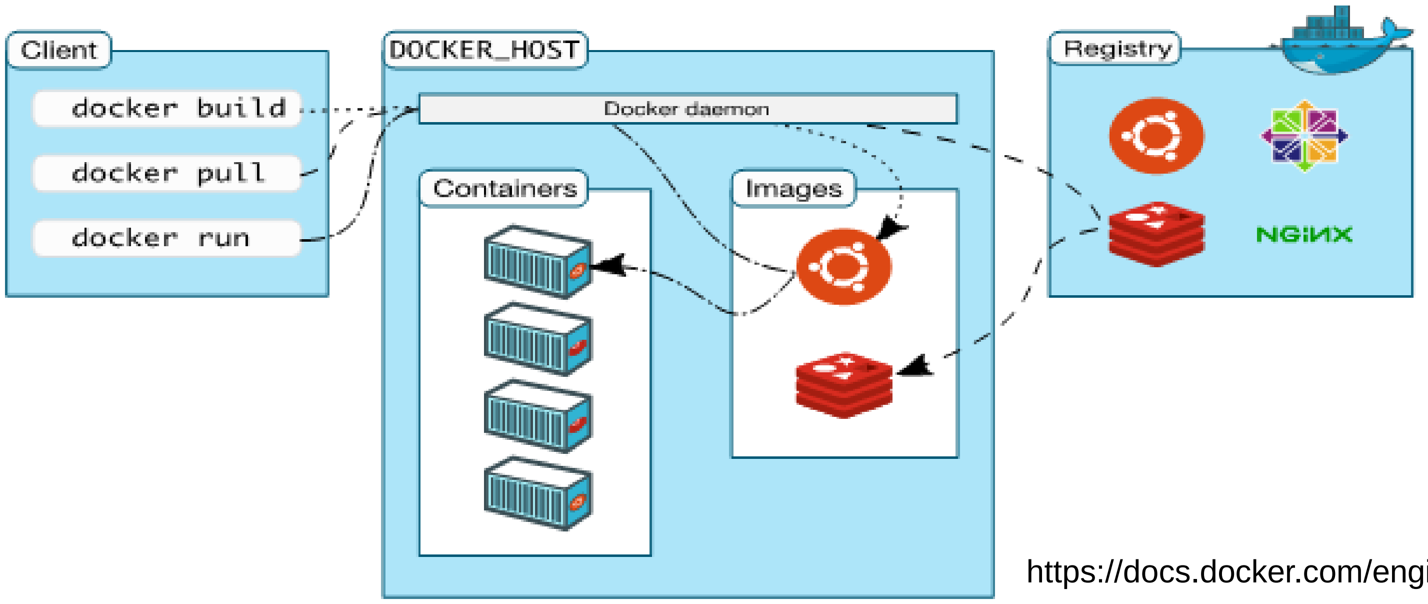
- Docker daemon is a long running process to provide overall control
  - images, containers, network and data volumes
- The daemon exposes a REST API to allow remote control
- A command line interface (CLI) to interact with docker daemon

Any similarity with system VM?
Any similarity with JVM?
Which component is managing access to the critical resources like cpu and memory?



https://docs.docker.com/engine/docker-overview/

Docker Architecture



https://docs.docker.com/engine/docker-overview/

- The client and daemon can run on the same or different system
- Docker daemon has command for managing the life cycle of a container
- The docker registries are where docker images are stored and can be used

Docker Objects

- Container
  - A relatively isolated running environment for user’s applications, e.g. a web application. Container uses kernel technologies to manage resource allocation and isolation.
- Images
  - A read-only template with instructions for building and executing some application inside container
- Network
  - Mechanisms for connection among Docker and non-Docker workload
- Data Volume
  - Is the standard mechanism for persisting data for container

Docker image

- Docker images are defined in Dockerfile
  - A text document containing a sequence of instructions/commands to build and run your application
  - If you ever written something like a *make* file, *ant* build file, Maven *POM* file, etc. etc. the Dockerfile is designed for similar purpose
  - You declare dependencies, set environment variables and other configurations in it
- The images are created by calling the command **`docker build`**
- The images are stored in a local Docker image registry and can be published in a public registry

Example Dockerfile

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

https://docs.docker.com/get-started/part2/#dockerfile

Image Layering

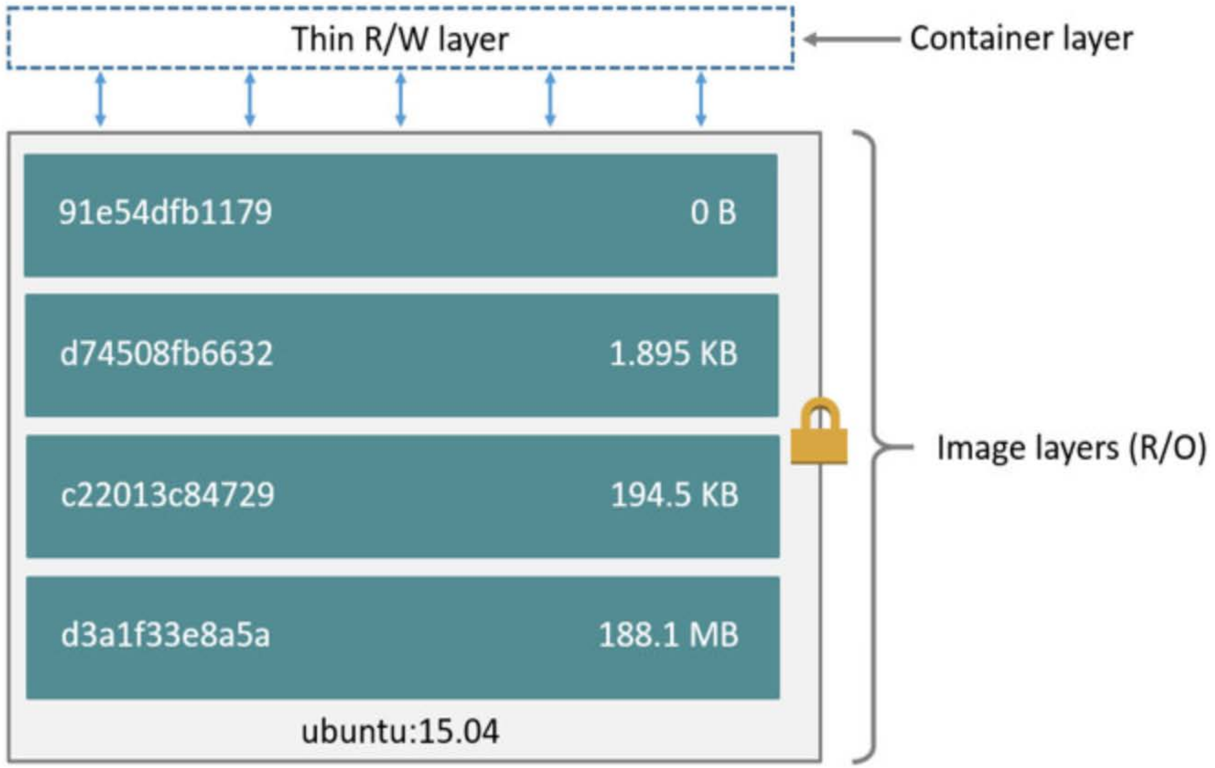
- Docker uses *Copy-on-Write* strategy to organize storage inside containers to guarantee the lightweight feature
  - Small space requirement and fast start-up time
  - Various drivers can be used aufs, Btrfs, ZFS, etc
- Copy-on-Write
  - Storages are organized into multiple logical layers
  - If a file or directory exists in the lower layers, the upper layers can use it
  - If an upper layer needs to modify anything (write) on the lower layer, it creates a copy on that layer and modify it.
  - Common lower layers can be shared by many images
- Docker images are build on top of each other, upper images do not need to copy lower image files if nothing is changed.

Image Layers

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Obtain some layers from parent image

Create a layer each



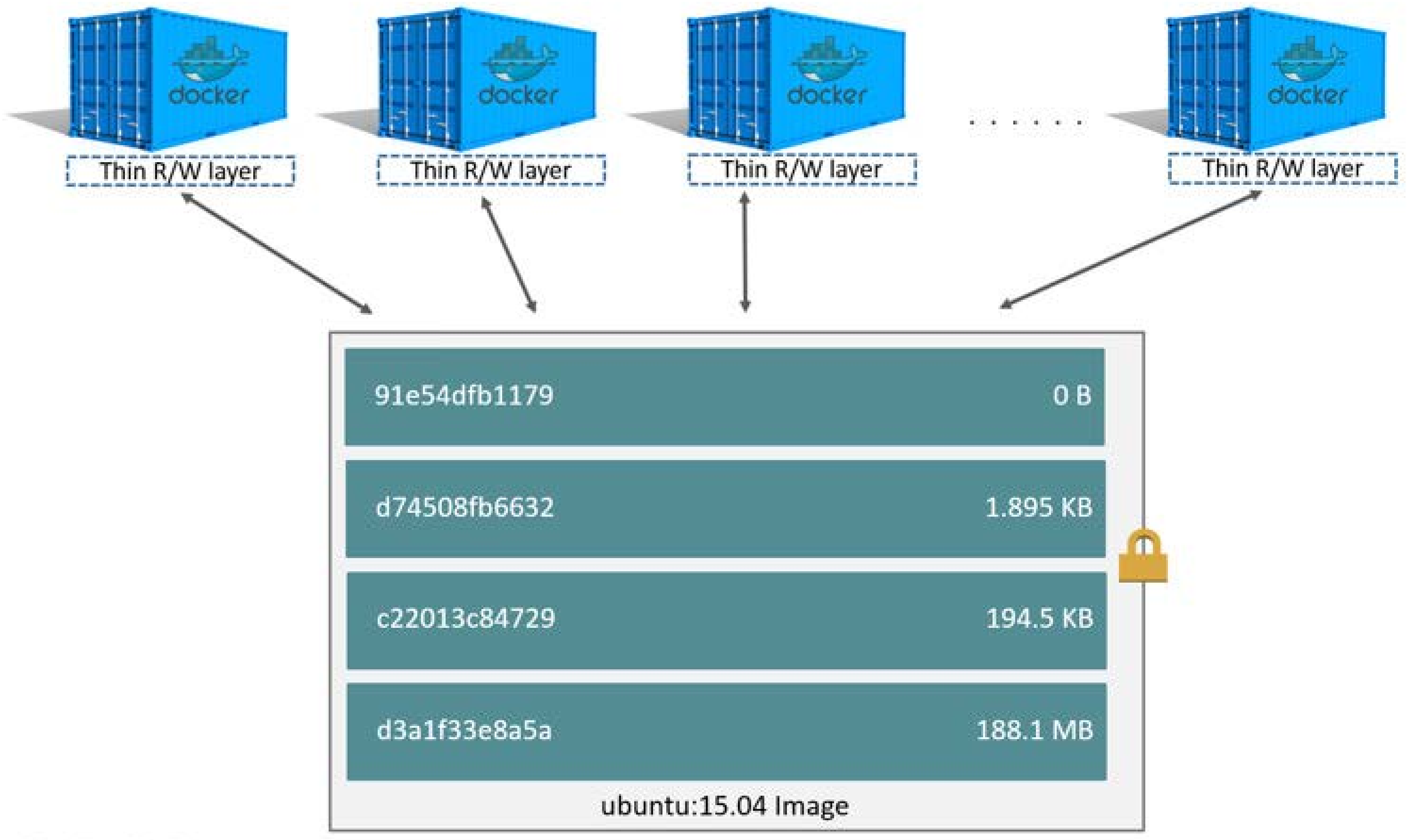
Container
(based on ubuntu:15.04 image)

https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/

Container and image layers

- Images are read-only templates, all image layers are read-only
- When an image is loaded into a container to run, the container add a thin writable layer on top of it.
  - All writes to the container are stored in this layer
  - It is deleted when container exits (is deleted)
  - Should be used as temporary storage
- If multiple images use a same base image, only one copy of the base image is required
  - Small space
- When container starts, only a new writable layer is added on top of the existing image layers
  - Fast start up
- Application persistence should be handled differently

Multiple containers sharing the same image

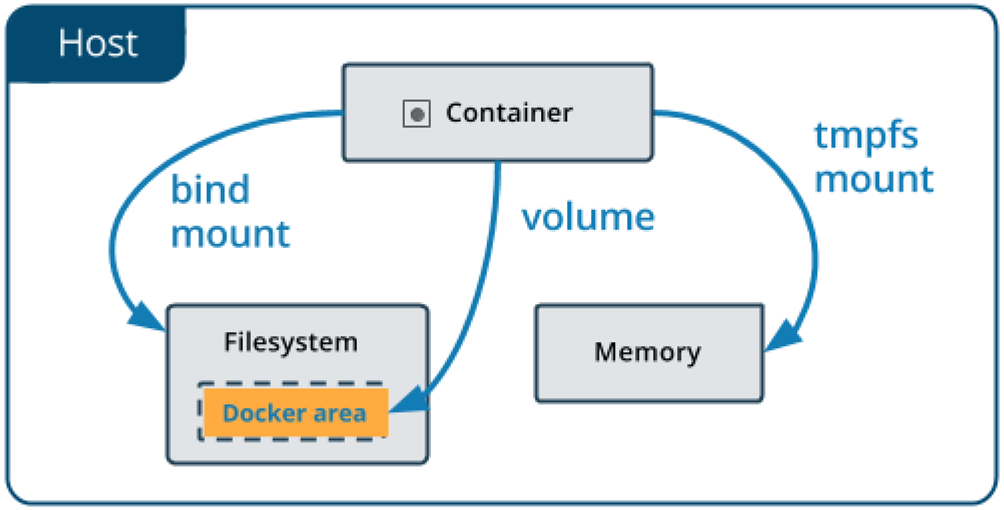


Docker Container

- The actual container uses many OS technologies to provide isolation and to allocation resources: cpu, memory, i/o
- This is achieved by utilized various features provided by linux kernel
  - Namespaces
  - Cgroups
  - Others
- **Linux Containers (LXC)** was used as the execution driver
- The current driver **Libcontainer** is more general, allowing Docker to run on platform other than Linux

Data Volume

- Apart from storing data within the writable layer of a container, there are other preferred ways to persist data
  - Bind mount is an early version storage option, it allows a client to mount any file or directory on the host machine into a container
  - Volumes uses a designated location in the host machine as container storage, it is completely managed by Docker
  - tmpfs is a rarely used option, which uses host memory to simulate storage



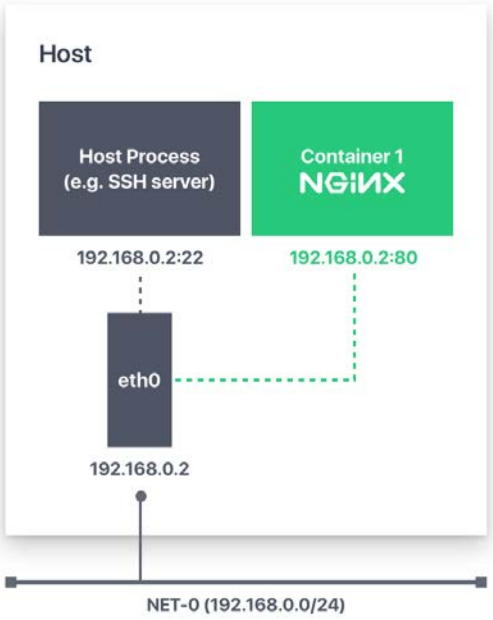
Networking

- Docker provides a few options for container to decide how it wants to connect with outside
  - **Host**
  - **Bridge**: default/user defined
  - **None**
  - Overlay
  - Others…
- These are specified during the start of the container, if nothing is specified, the default bridge driver is used.

Networking: Host

- Host driver is the simplest option,
  - It removes the isolation between container and host
  - Container is treated in the same way as a process in the host
  - It connects directly to the host NIC (host networking namepace)

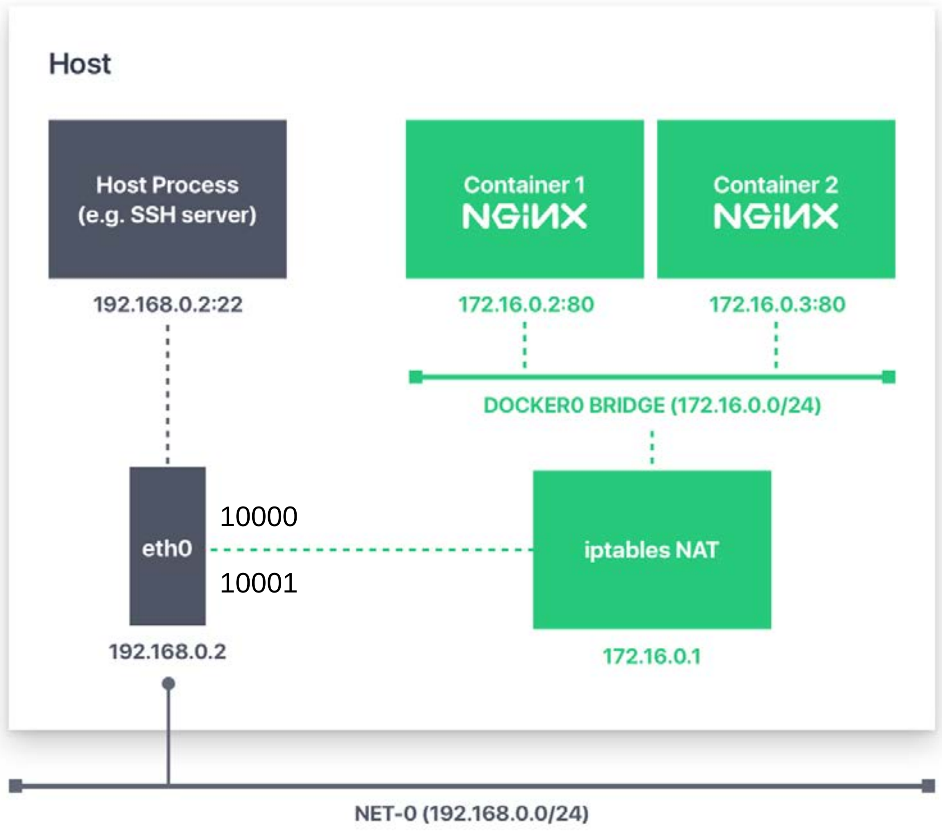
```
docker run –d --name nginx-1 –net=host nginx
```



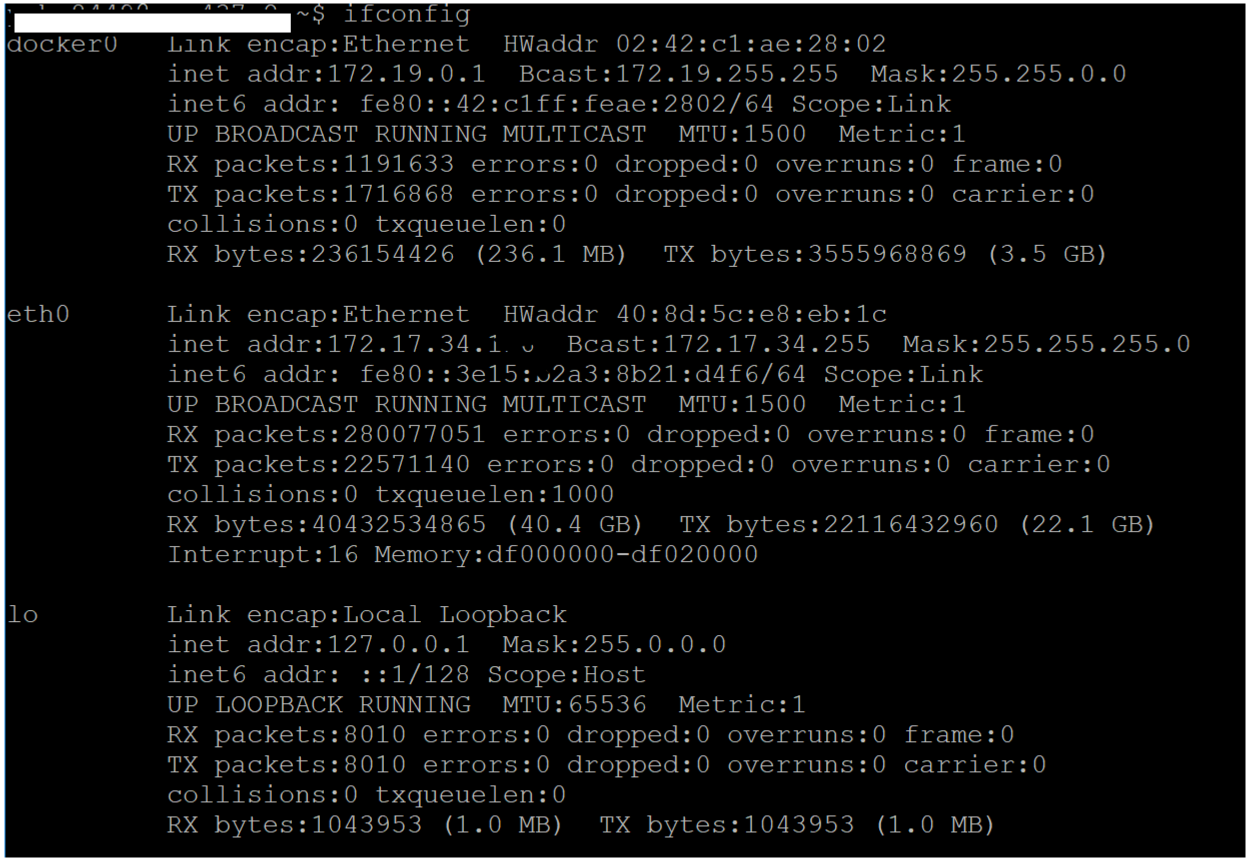
https://mesosphere.com/blog/networking-docker-containers/

Networking: Bridge

- Docker manages its own private network
  - Adding a software bridge to the host
  - The mapping from internal/private address to public/host based address is done through Network Address Translation (NAT)



docker run –d —-name nginx-1 -p 10000:80 nginx
docker run –d —-name nginx-2 -p 10001:80 nginx



```
[redacted]~$ ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:c1:ae:28:02
          inet addr:172.19.0.1  Bcast:172.19.255.255  Mask:255.255.0.0
          inet6 addr: fe80::42:c1ff:feae:2802/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1191633 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1716868 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:236154426 (236.1 MB)  TX bytes:3555968869 (3.5 GB)

eth0      Link encap:Ethernet  HWaddr 40:8d:5c:e8:eb:1c
          inet addr:172.17.34.1[redacted]  Bcast:172.17.34.255  Mask:255.255.255.0
          inet6 addr: fe80::3e15:[redacted]2a3:8b21:d4f6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:280077051 errors:0 dropped:0 overruns:0 frame:0
          TX packets:22571140 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:40432534865 (40.4 GB)  TX bytes:22116432960 (22.1 GB)
          Interrupt:16 Memory:df000000-df020000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:8010 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8010 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1043953 (1.0 MB)  TX bytes:1043953 (1.0 MB)
```

Docker Security Issues

- **Namespaces provide the first and most straightforward form of isolation**
- But for various reasons, a container might not follow strict name space based isolation
  - E.g. the Host networking driver puts container and host in the same namespace
  - Even though each container has its own file system. The host file system is not completely invisible to container.
- The special requirements to run Docker daemon as root largely compromises security
- Docker containers are started with restricted capabilities and can be further restricted, but the default profile does not provide complete isolation
  - It is relatively easy for a user to escalate to some level of ‘root’ privilege through container.

Next Week

- This is the end of cloud enabling technology content
- Starting from next week, we will focus on “big data analytic” frameworks
- The topic for next week is basic MapReduce programming
  - Dean, Sanjay Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters*. In OSDI'04,
  - Tom White, Hadoop, the definitive Guide, O’reilly, 2009

References

- Container's Anatomy,
  - https://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon
- Dirk Merkel, Docker: Lightweight Linux Containers for Consistent Development and Deployment, Linux Journal, May 19, 2014
  - https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment
- Docker Documentation: Docker Overview
  - https://docs.docker.com/engine/docker-overview/#the-docker-platform
- Docker File Systems
  - https://docs.docker.com/storage/storagedriver/#images-and-layers
  - https://docs.docker.com/storage/volumes/
- Docker Networking
  - https://mesosphere.com/blog/networking-docker-containers/
- Docker Security
  - https://docs.docker.com/engine/security/security/