Docker and containers

...

O. Sallou - IRISA - CC-BY-SA 4.0

Linux containers

- Based on cgroups, namespaces, …
- Isolation of processes, network, …
-
- Execute some processes in isolation using host Linux kernel
  - different from virtual machines (no emulation)
  - executes Linux only (Linux on Linux)
  - Mechanism implemented on Windows Server (windows on windows)
- Works on Windows/Mac via a VM

Docker but...

- Several software provide the same mechanism: lxc, Rkt, singularity
-
- Not all with same features and ease of use
- Docker is the actual “standard” but could change
- OCI (Open Container Initiative) to create standards on image format and components

Docker

- Makes use of Linux containers
- Ease of use and install
- Executes a daemon
  - CLI talks to daemon
  - API available to talk to daemon
  - Daemon executes/stops containers
- Daemon can listen on a TCP port (WARNING: anyone with access to the port can execute Docker)
- Need root access or be in docker group to execute Docker commands
- Executes a Linux operating system above host (can be different flavor, ie Debian, Fedora, ...), you cannot use the host system direclty (some others do)

Process isolation

- Processes are put in a group set sharing the same CPU and MEM space limits/rate.
- If a process executes a new thread/process afterward, it is placed in the same group.
- Killing a group will kill all processes within

Process isolation

- Memory is an optional hard limit
  - Procs won’t be allowed to allocate more memory
- CPUs is also optional but:
  - There is no hard limit on number of CPU but a CPU share limitation (kernel time sharing on the task) and if more CPU is available, it is used.
  - Not linked by default to a specific CPU or CPU set

Time sharing

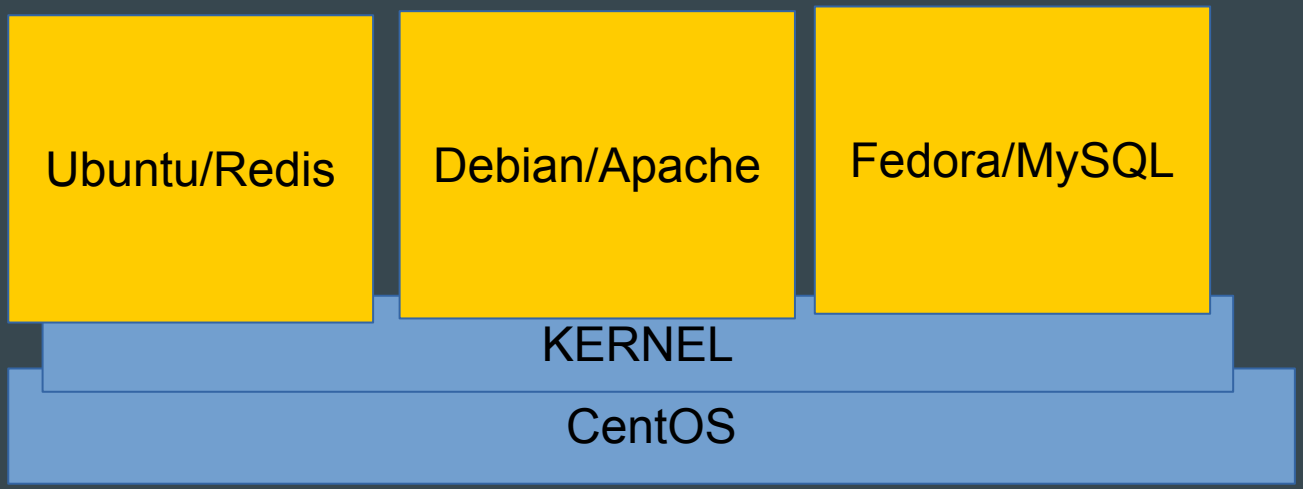
- Let’s suppose with have 1 core only
- Task 1 ask for 1024 shares
  - Task 1 will use the whole CPU time if needed
- Task 2 now ask for 2048 shares
  - Task 1 will be allocated 1/3 of CPU time
  - Task 2 will be allocated 2/3 of CPU time
- Values (1024, 2048) have no meaning and can be any number, they are also used as a ratio
- Docker asks for values > 1024

Process isolation - namespaces

- Namespace changes process PIDs, main process gets PID 1
- PIDs are “internal PIDs” within the namespace and maps to a system PID.

An OS on host OS?

- Docker executes an OS (full OS or minimal, like Busybox or Alpine)
- But OS does not execute any service (systemd, ...)
- Container starts 1 process only (but process can start/execute other ones)
- A host can supports hundreds of containers (as if they processes run directly on host)



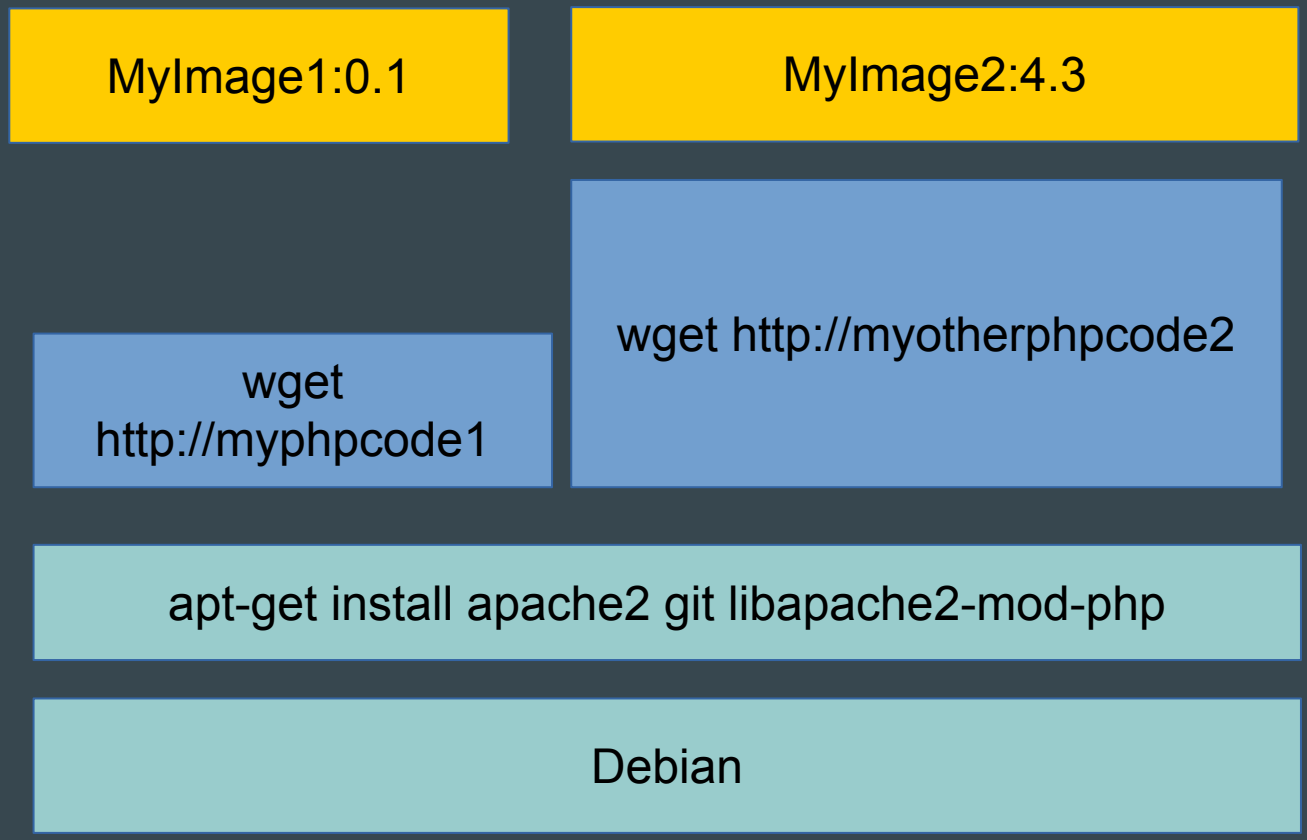
Images

- Docker executes a process based on an image
  - “docker images”
- Images are quite small compared to VM (50Mb or a few 100’s Mb)
- Local or remote
- Version-ed (Ubuntu:14.10, MySoft:0.1)
- Images are read-only

Layers

- Images are made of layers
- Layers can be shared by multiple images
  - Limits the disk space needed, 2 Debian images can use the same base image
- Layers are immutable
- You can’t delete a layer used by an other image
- Identified with sha ids, similar to git mechanisms.

Layers



MyImage1 and MyImage2 will share 2 layers, and will get 1 specific layer each
When instantiating images, shared layers will be installed once, with no duplication, limiting disk requirements.

Image tags

- Tags give a name and a version to an image
  - docker tag XYZ mysuperimage:1.0
  - docker tag XYZ mysuperimage => mysuperimage:latest
- There are some conventions on image name/tags
- Image name/versions are URIs, they give a unique identifer and an address where to download the image
  - docker pull redis
    - download redis image from DockerHub, the default address
  - docker pull docker-registry.genouest.org/debian/acabas
    - download debian/acabas from https://docker-registry.genouest.org

Containers

- Container is an instance of an image
- Creates a new layer above image layers, this layer is writable but not persistent (deleted with container)
- Container starts 1 process which get id 1
- Possibility to attach a tty for interactivity (-it)

Root in container

- In container, process is started as root
- As a new isolated image is executed, user does not have access to the host filesystem/processes.
- Any write in a volume (see later) will be done as root too
- Users existing on host (local users, ldap users…) DO NOT exists in container, it is a different OS/configuration, no mapping possible.
- If a user can execute docker, you give him root privileges in the container in YOUR network, this implies security considerations.

Life cycle / commands

- docker pull XY
  - download the XY locally
  -
- docker run XY /usr/bin/dosomething
  - Executes a container ie image instance and executes *dosomething* (program MUST be in container)
  - Returns a container ID
  - When *dosomething* ends, container stops
  -
- docker logs containerId
  - See stdout logs of container
  -
- docker rm containerId
  - Deletes container and all its content

Life cycle / commands

- docker “start/stop”
  - can be executed on previously created containers
  -
- docker run –rm -it XY /bin/bash
  - Execute XY in interactive mode, in a bash shell and deletes container at exit.

Access to the container

- Docker exec will execute a command in the container
- No SSH, command is executed directly in the container
- tty allows interactive sessions

Volumes

- User cannot “mount” a filesystem (or need explicit privileges)
- Can specify a local host dir to be bind-mounted, will be seen in container like a local directory
  - docker -v /home/path:/opt/path:rw
    - /home/path: local host directory
    - /opt/path: directory in container

Volumes

- A container can be a data volume and attached to a container
- But size is limited to container limits (10Go by default)
- Useful for data persistence
- A data container must be on the same host than attached container, and is not moved automatically with the container

Network

- Containers use a local bridge with NAT
- Ports (http, ssh, …) are not accessible by default
- Port mapping allows to open a local host port mapped to a container port (example maps port 8080 to container port 80)
  - Need to manage local ports to avoid conflicts
  - Ports mapped via command line
    - docker … -p 8080:80 ...

Network

- Containers can use different network mechanisms to get their own public/private network IP (see CNI)
- Docker networks allows containers within the same network to communicate without port mapping using internal IP address (iptables managed by Docker)
  - docker network create XXX
  - docker run -net XXX ….
  - Networks can be used to link a container with its database container for example

Dockerfile

- To create a container, “docker build .”
- With a Dockerfile
  - Like a Makefile
  - One line per command to execute = one image layer per line
  - No process running during build (cannot insert in a database for example)

Dockerfile example

FROM debian:latest <= base image
MAINTAINER me <me@example.org> <= author info
LABEL TEST=”123” DESCRIPTION=”this is a test” <= labels..
ENV CODEENV=”dev” <= set some env variables
RUN apt-get update
RUN apt-get install -y git wget
RUN wget http://…./mydata
RUN git clone https://…/mycode.git
ENTRYPOINT [‘/root/mycode/bin/test’] <= default program to execute (can be overriden)

Build and run an image

docker build .
….
Built image 0A23443FEFE
docker tag 0A23443FEFE me/myimage:1.0
docker run me/myimage:1.0 -v /tmp:/data -i /data/in
-o /data/out

Plugins

- Docker supports external plugins to manage networks and volumes
- Easy to create your own plugin with a simple web server serving a REST API.
- Docker calls API during container life cycle
- Some plugins:
  - Calico, Weave, ...: network
  - Flocker, Convoy, ….: volume

Security

- No mount in container
- Restricted capabilities can be overridden
- User namespace can map users (including root) in container to a different set of user ids on local host but cannot be specified on a per container basis
- Network can be disabled

Security

- A user with Docker access can mount any volume from local host, with root access in the container => ok on local computer, not in a shared resource
- Images: should care about used images, could contain anything….

CNI – Container Network Interface

- CNI modifies the base network mechanisms of containers
- Network management is delegated to an other provider (CNI compliant) to provide an IP address to the container and manage network access to the container and between the containers on multi-hosts systems.
- Containers get their own IP address, removing the need to map the ports with local host.
- Provided via network plugins

Registries

- Docker registry (DockerHub) is a “store” for container images
  - docker push me/myimage
  - docker pull me/myimage
- Default registry is docker one but can specify other registries, image name is a URI:
  - docker pull docker-registry.genouest.org/me/myimage
  - If first part of name contains dots, it is a server name
- Registries manage ACLs to allow/deny access to the container images

Other registries

- Hosted: Quay.io
- Open source:
  - port.us.org (general usage)
  - bioshadock.genouest.org (for bioinformatics)

Ecosystem

- Swarm, kubernetes, mesos
  - Cluster/scheduling management for containers (manage resources and placement on multiple hosts)
  - Manage network links among multiple hosts
- Simulate time sharing with number of CPUs (1 cpu on 3 will get 1/3 of whole CPU time but not a real CPU)

That’s all ! :-)