

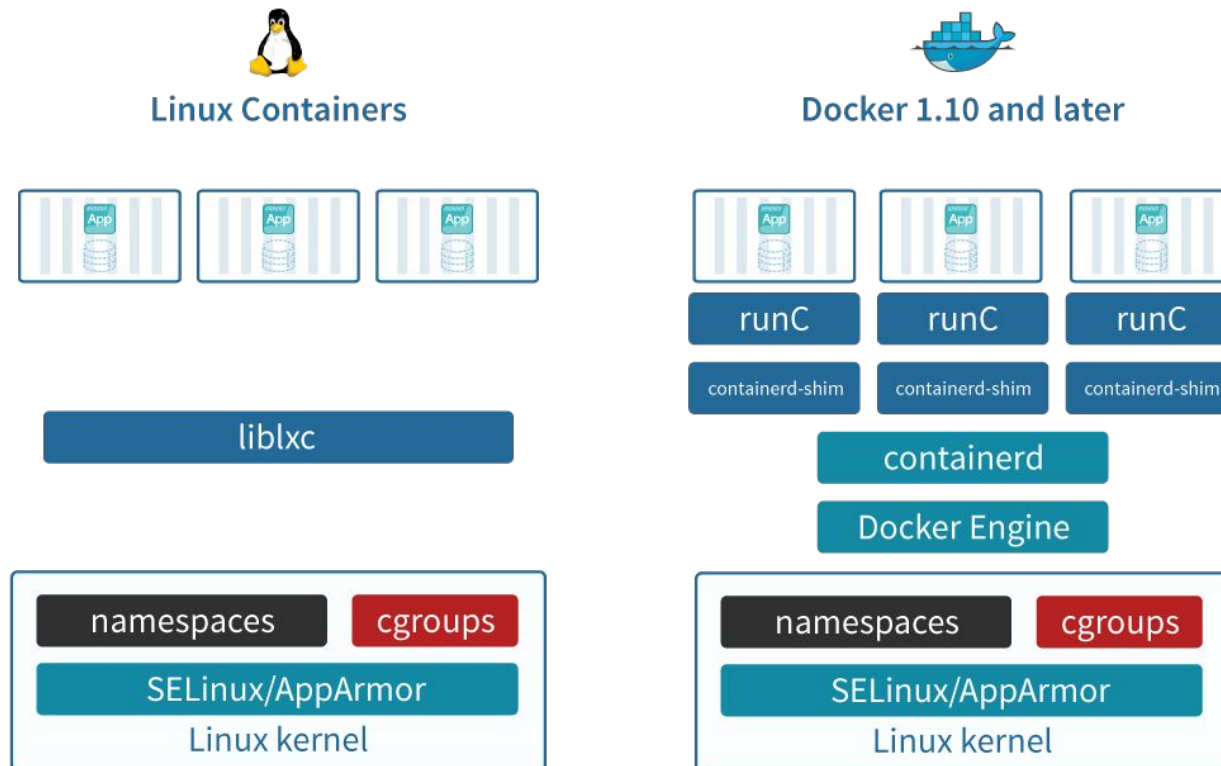
An introduction to Docker

Ing. Vincenzo Maffione

Operating Systems Security

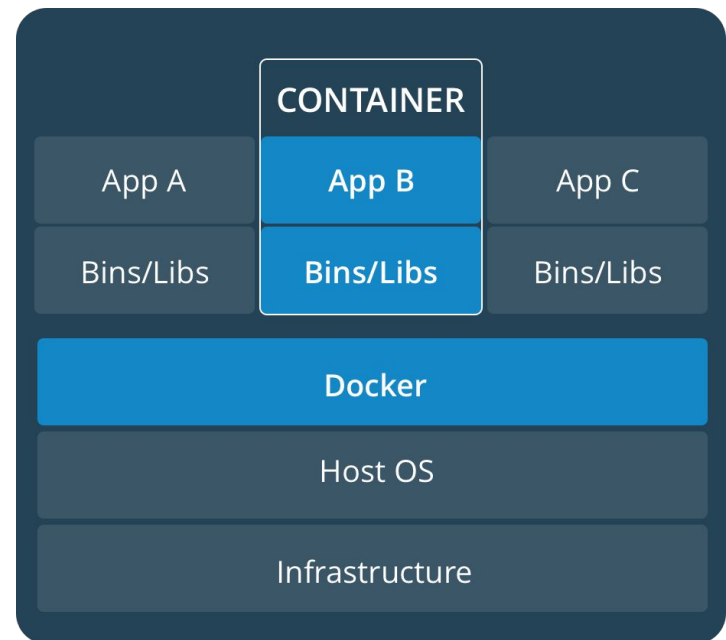
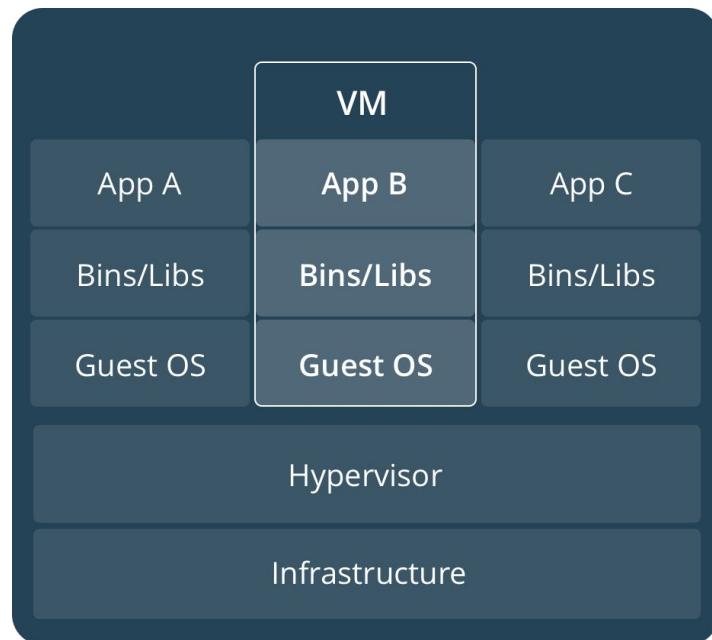
Container technologies on Linux

- Several *light virtualization* technologies are available for Linux
 - They build on *cgroups*, *namespaces* and other containment functionalities
 - LXC (Linux Containers) and Docker are the most popular products



Container vs Virtual Machine

- A VM has emulated hardware and hosts a whole Operating System (*guest*), which is separate from the host O.S.
- A container does not emulate any hardware, and shares the O.S. *kernel* with the host → **less isolation, more efficiency**



LXC - Linux Containers

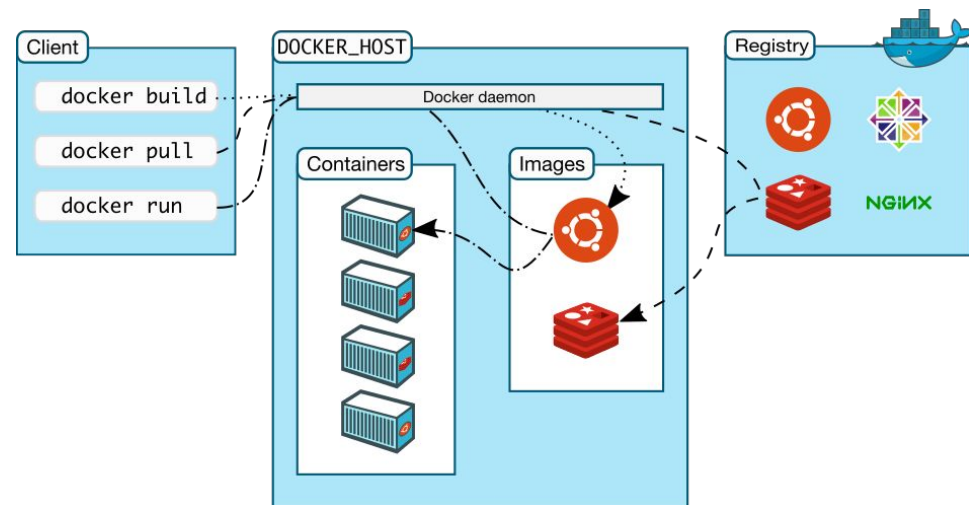
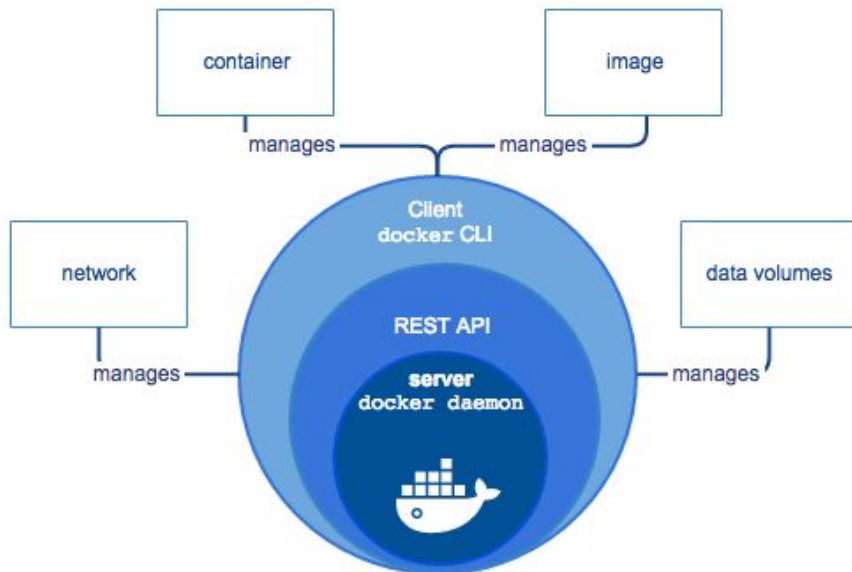
- A set of tools for creating and managing *system containers* (or, to a lesser extent, *application containers*)
 - Supported by *virt-manager* as a possible hypervisor (even if it's not)
- The goal of LXC is to provide an *execution environment as close as possible to a standard GNU/LINUX installation*, without the need to have a separate Linux kernel
- Designed for system administrators that are used to work with VMs
 - A LXC (system) container looks like a VM without a *boot loader*
 - The administrator can move its application stack from a VM to a container *without the need to modify the applications or their configuration*
 - Switching from VMs to LXC gives performance gains
 - Storage technologies for LXC and VMs are similar

Docker

- Docker is a tool for creating, managing and orchestrating *application containers*
- The goal of Docker is to optimize the process of *development, testing, delivery and deployment* of applications, by means of packaging each application component in a separate *container*
- Designed for software developers:
 - Takes care of all the steps involved in software development
- Switching from VM to Docker containers **is not immediate**
 - It may be necessary to modify the application or its configuration
 - The root of the problem is that the execution environment of a Docker container **it's not normally a complete UNIX system**

Docker architecture

- Client-server architecture to manage *images*, *containers*, *volumes* and *virtual networks*
 - Client and server may run on different machines
 - Architecture is similar to *libvirt*, but with more functionalities, including the capability to interact with an *image registry* (<https://hub.docker.com/>)



Docker components (I)

- **Image**

- An image is a portable template, accessible in read only mode, that has all the instructions necessary to create a Docker container
- Contains all the dependencies needed for a software component (code or binary, run-time, libraries, configuration files, environment variables, ...)
- An image can be defined from a file system archive (tarball)
- Or it can be defined by extending a previous image with a list of instructions specified in a text file (known as *Dockerfile*)

- **Docker registry**

- Database for efficient storage of images
- Registries can be public (like DockerHub) or private to an organization

Docker components (II)

- **Container**

- A Docker *container* is an executable instance of a Docker image
 - Defined by the image and the configuration specified at creation time
- A container can be created, started, stopped, migrated, destroyed, connected to one or more virtual networks, associated to one or more data volumes ...
- The container is the unit of application development, testing, delivery and deployment, assuming that Docker is used as operating support
- Any modification to the file system visible to a container are not reflected on the image (image is read-only)
- It's possible to define to what extent a container is isolated from the host
 - Access to the host file system and special devices, limitations on memory allocation and CPU utilization.

Docker components (III)

- Network

- Virtual networks, implemented by means of *virtual switches* and iptables
- *Bridge* networks limit connectivity to the containers on a single host
- *Overlay* networks allow for containers connectivity among different hosts
 - Typically using VXLAN encapsulation

- Volume

- A volume is a directory that can be associated to one or more containers
- Its lifespan is independent of the containers that use it
- Used to share storage across different containers, or anyway storage that can outlive the user container

Docker components (IV)

- **Service**

- A Docker *service* is a set of containers that are replicas of the same image, and which together provide a *load balanced* service
- *Services* are used to deploy containers “in production”
- A service can be scaled up or down depending on the input load

- **Stack**

- A set of interdependent services that interact to implement a complete application:
 - Ex: A web application to share pictures could be made of (i) a service for the storage and search of pictures; (ii) a service for the web interface for the users; and (iii) a service to encode/decode pictures

Single-host mode vs swarm mode

- By default, the containers of a Docker stack are deployed only on the host that runs the `dockerd` daemon
- However, Docker can also be configured in *swarm mode*
 - In this case the containers that make up the stack can be placed on all the nodes of a cluster (a.k.a. *swarm*)
 - A swarm consists of a *swarm manager* node and a set of *swarm worker* nodes

Docker under the hood

- Linux namespaces
 - Normally, each containers comes with an instance of each type of namespace (pid, net, ipc, mnt, uts), to limit the scope of host kernel objects visible to the container
- Linux cgroups
 - Used to limit the amount of resources assigned to the containers
- Union File Systems
 - File systems that are defined by composition, overlapping different *layers*
 - Common layers (e.g. base installation of Ubuntu or Fedora) are reused by many images and containers
 - New containers and new images consume only a small space
 - They can be created very quickly!

Dockerfile

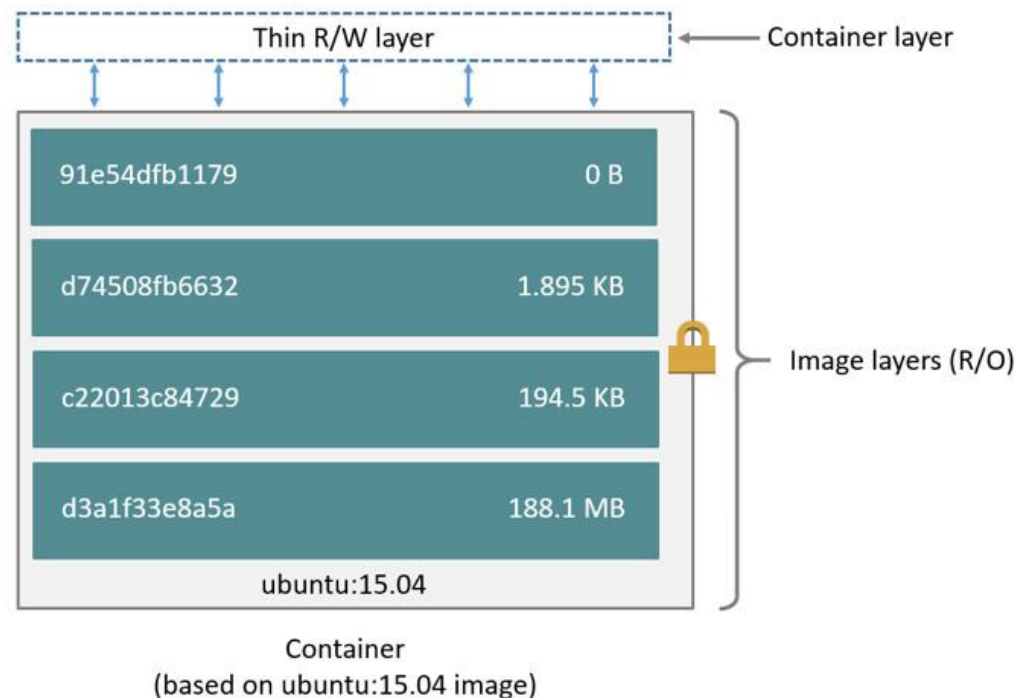
- A text file that contains a **recipe to build an image**
- An image should be a well-defined component and contain only the software actually needed for a well-defined task

```
# Start from an official image with the Python runtime
FROM python:2.7-slim
# Set the container current working directory (PWD) to “/chess”
WORKDIR /chess
# Copy files from current host directory to the /chess directory in the container
ADD . /chess
# Install some packages
RUN apt-get update && apt-get install -y libcgroup acl
# Flag that the software inside this image listens on port 9000
EXPOSE 9000
# Define an environment variable
ENV PLAYER Ghost
# Specify the command to be run inside the container when it's started
CMD ["python", "./chess.py"]
```

Structure of a Docker image

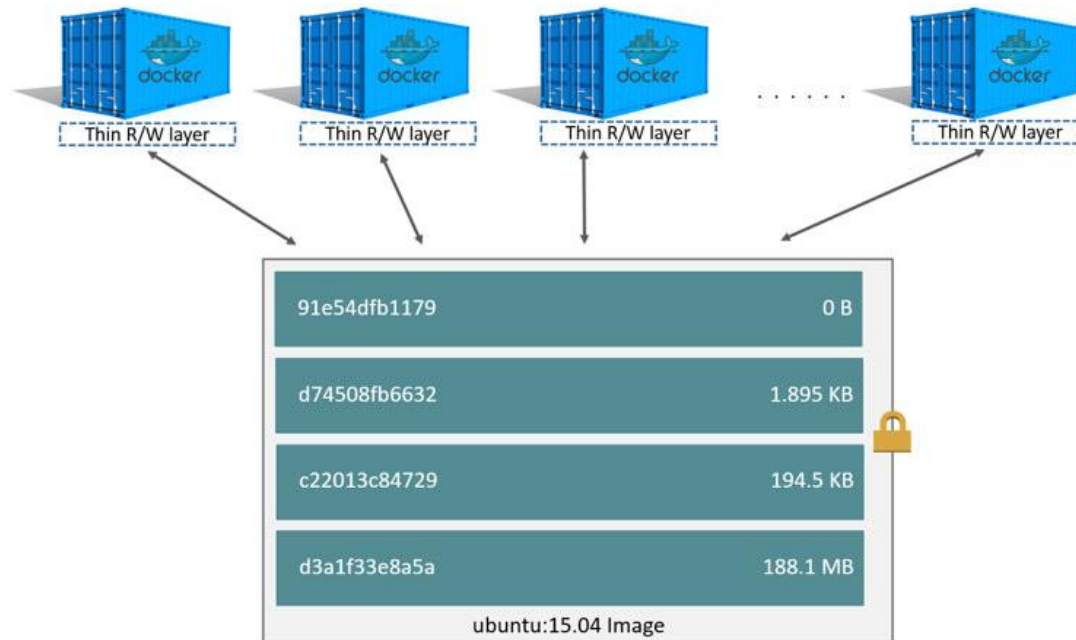
- An image is a *stack* of layers (*onion*-like structure)
- Each instruction in the Dockerfile adds a layer
 - Each layer stores only the difference w.r.t. the previous layers.
- A read/write *container layer* gets created on containers creation

```
# This is a Dockerfile
FROM ubuntu:15.10
COPY . /app
RUN make /app
CMD python /app/app.py
```



Sharing layers

- When a container is running, any modification to its disk are reflected to the *container layer*.
- All the other layers are read-only and can be shared among many container

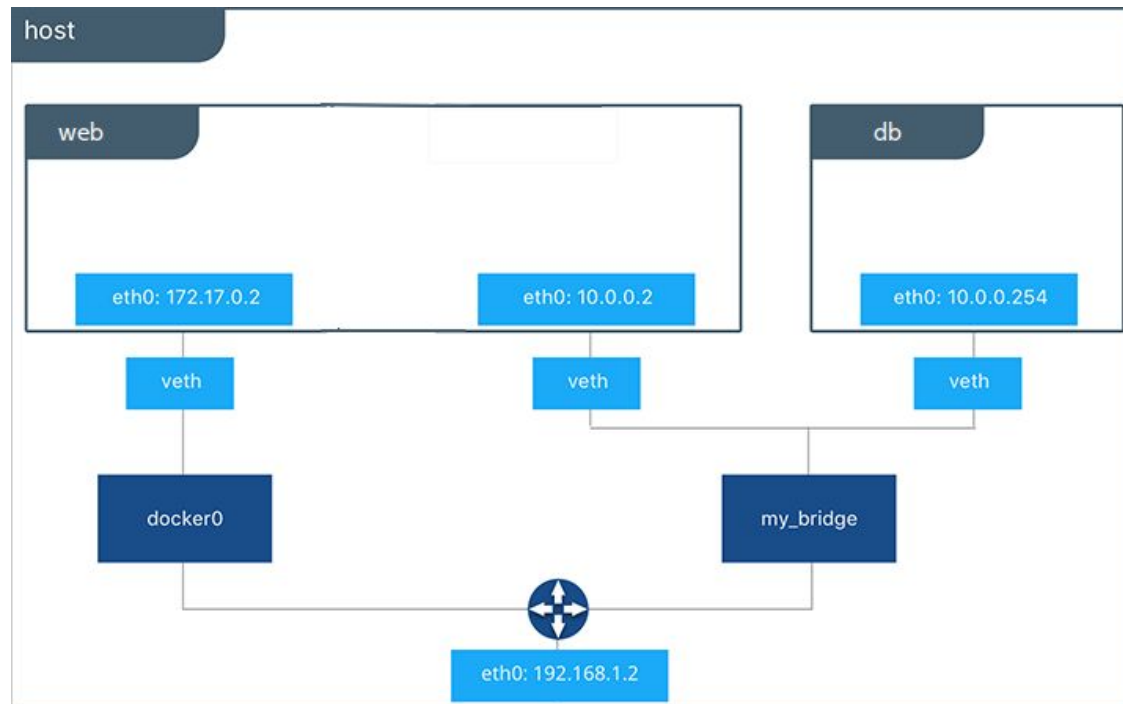


Implications of the onion structure

- A container does not take any disk space until it performs some write operation on the file system
 - In any case it takes only the space needed to store the difference
 - Huge **disk space savings compared to VMs and LXC**, which both store images with a *monolithic* format (e.g. qcow2)
- Creation of new images and containers is extremely fast compared to VMs and LXC
 - To create a new container it is sufficient to create an empty container layer

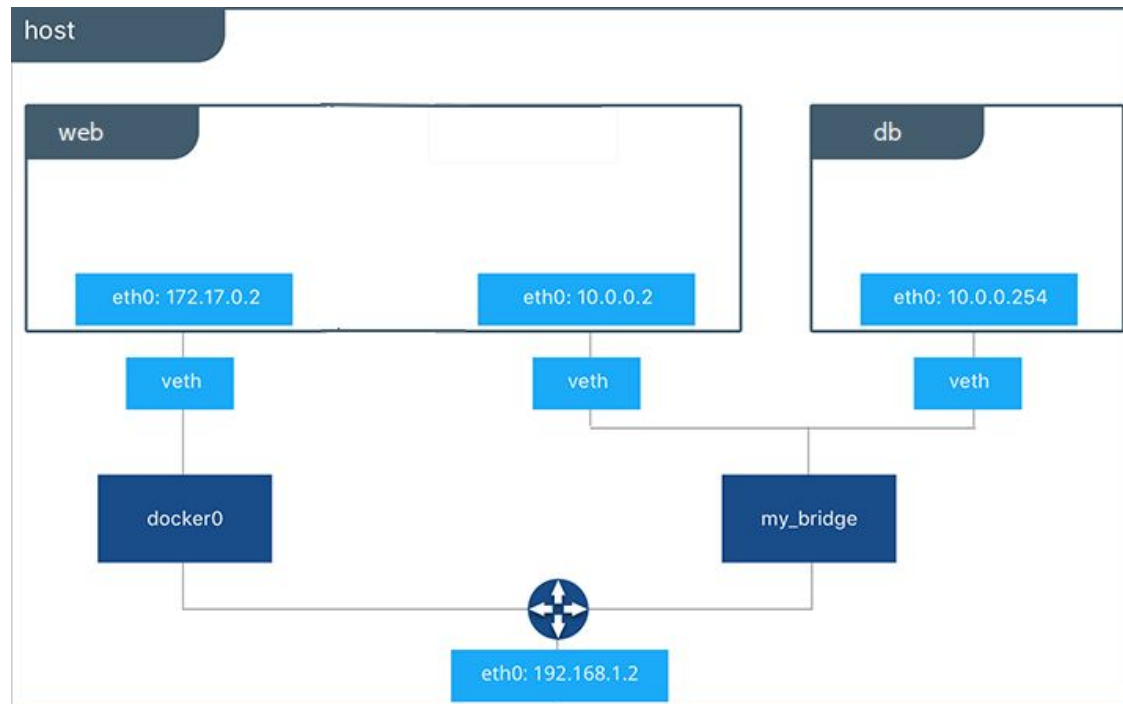
Docker networking (I)

- Standard software bridges are used to connect the containers by means of *virtual interfaces* (*veth* pairs in this case)
- The user can create and manage new networks, connect a container to one or more networks (even while the container is running)



Docker networking (II)

- Each *bridge* network uses a different IP subnet
- The IP subnet is visible to the host
- Networks use the *bridge* driver (host-only + NAT) by default



Main Docker commands

How to install Docker

- Install the latest Docker release on Ubuntu 16.04

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -  
$ sudo add-apt-repository "deb [arch=amd64]  
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"  
$ sudo apt-get update  
$ sudo apt-get install docker-ce  
$ sudo usermod -aG docker ${YOUR_USERNAME}
```

- Check that Docker is up and running:

```
$ sudo systemctl status docker  
[...]  
Active: active (running) since ...  
[...]
```

Search images in a registry

- Search in a registry (e.g., the default public one)

```
[user@host ~]$ docker search nginx
```

- The output shows a list of images matching the keyword
- Images are sorted by decreasing number of *votes* given by Docker users
 - Most popular images first

- *Note: An incomplete command shows and help with all the possible options*

```
[user@host ~]$ docker image
```

```
[...]
```

```
Commands:
```

```
build      Build an image from a Dockerfile
```

```
history    Show the history of an image
```

```
import     Import the contents from a tarball to create a filesystem image
```

```
inspect    Display detailed information on one or more images
```

```
[...]
```

Image management

- List the existing images

```
[user@host ~]$ docker image ls
```

- Import an image from a registry

```
[user@host ~]$ docker image pull base/archlinux
```

- Remove an image (locally)

```
[user@host ~]$ docker image rm ubuntu
```

- Show detailed information about an image

```
[user@host ~]$ docker image inspect ubuntu
```

- Remove unused images

```
[user@host ~]$ docker image prune
```

Building an image from a Dockerfile

- Move to the directory containing the Dockerfile

```
[user@host ~]$ cd /path/to/dockerfiledir
```
- Build an image from the Dockerfile in the current directory, giving it a name (tag) “myimg”

```
[user@host ~]$ docker build -t myimg .
```
- The new image will be stored together with the other ones already available on the host
- Each Dockerfile is normally stored in a separate directory
 - The file name must be “Dockerfile”

Creating and launching containers

- Create a container and launch it (within the same command)

```
[user@host ~]$ docker run -it --name ubu1 ubuntu /bin/bash
```

- The `ubuntu` argument refers to the name of an available image
 - The `/bin/bash` argument specifies the command to be run by the container
 - If present it overrides the command specified within the image (CMD)
 - The `-t` option specifies the allocation of a terminal
 - The `-i` option specifies that the command is interactive (it's a shell)
 - The `-d` option is used to run the container in background
- It is possible to create and launch with separate commands:

```
[user@host ~]$ docker create -it --name ubu1 ubuntu /bin/bash
```

```
[user@host ~]$ docker start -i ubu1
```


Publication of exposed ports

- An image can expose a TCP/UDP port through the EXPOSE directive in the Dockerfile
- When a container is launched, it is possible to map each exposed port to an host port, to enable access from the host external network
- This mapping is specified through the -p option of the run or create commands
 - Ex: Launch a Web server container exposing port 80, mapping it on the port 8000 of the host

```
[user@host ~]$ docker run -p 8000:80 apache /usr/bin/apachectl --daemon
```

Container management

- Show all the running containers
`[user@host ~]$ docker ps`
- Show all the containers (in any state)
`[user@host ~]$ docker ps -a`
 - Includes containers that are not currently running
- Reboot a container (specified by name or ID)
`[user@host ~]$ docker restart ubu1`
- Stop a container
`[user@host ~]$ docker stop ubu1`
- Remove a container
`[user@host ~]$ docker rm ubu1`
- More commands: `kill`, `inspect`, `pause`, `unpause`, ...
- Equivalent commands in *canonical form*:
`[user@host ~]$ docker container COMMAND`

Volumes management

- Create a volume called “myvol”

```
[user@host ~]$ docker volume create myvol
```

- Remove “myvol”

```
[user@host ~]$ docker volume rm myvol
```

- Show the list of volumes available on the host

```
[user@host ~]$ docker volume ls
```

- Run a container, making the content of the “myvol” volume available in the /mntvol path inside the container

```
[user@host ~]$ docker run -v myvol:/mntvol -it ubuntu /bin/bash
```

- Run a container, making the content of the host directory “/home/user/tmp” available in the /mnt path inside the container

```
[user@host ~]$ docker run -v /home/user/tmp:/mnt -it ubuntu /bin/bash
```

Data volume containers

- It is possible to create unnamed volumes, implicitly associated to a container (but still independent on the container lifespan)
 - Ex: Create an unnamed volume, making it available in “/myvol”

```
[user@host ~]$ docker run -v /myvol --name ubu1 -it ubuntu /bin/bash
```
 - Actually, a volume name is assigned automatically
- Unnamed volumes can be attached to other containers
 - Ex: Launch a container importing volumes from another container called ubu1

```
[user@host ~]$ docker run --volumes-from ubu1 --name ubu2 -it ubuntu /bin/bash
```
 - The imported volumes are mounted in the ubu2 file system at the same mountpoints used inside ubu1
- A container like ubu1 is called “*Data volume container*”

Management of Docker virtual networks

- Show current networks

```
[user@host ~]$ docker network ls
```

- Create an user-defined *bridge* network (“mynet”)

```
[user@host ~]$ docker network create --subnet=192.168.13.0/24 mynet
```

- Remove a network

```
[user@host ~]$ docker network rm mynet
```

- Attach the *ubu2* container to *mynet*

```
[user@host ~]$ docker network connect mynet ubu2
```

- Detach the *ubu2* container from *mynet*

```
[user@host ~]$ docker network disconnect mynet ubu2
```

- Launch a container attached to *mynet*

```
[user@host ~]$ docker run --network=mynet [options] imagename command
```

- Launch a container attached to *mynet*, specifying a static IP

```
[user@host ~]$ docker run --network=mynet --ip 192.168.13.4 [options] imagename  
command
```

Other commands

- Show total Docker disk usage

```
[user@host ~]$ docker system df
```

- System clean up: remove stopped containers, and unused volumes, networks and images

```
[user@host ~]$ docker system prune
```

- Remove all the images and all containers (including the ones in use)

```
[user@host ~]$ docker rm $(docker ps -a -q)
```

```
[user@host ~]$ docker image remove $(docker images -q)
```

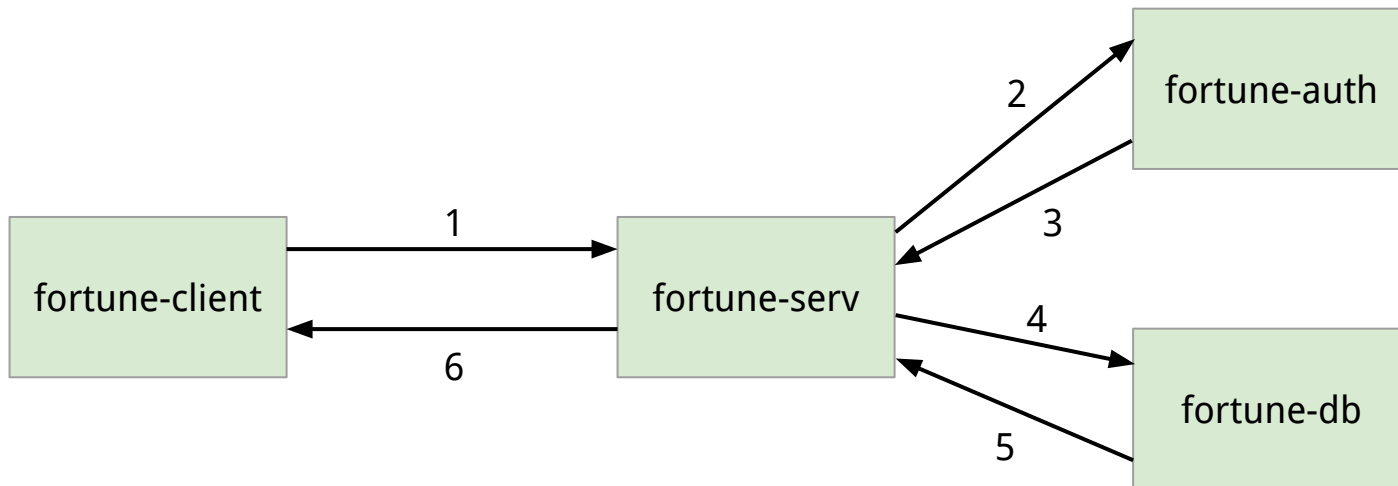
- Show per-container real-time statistics, including utilization of CPU, memory, network and disk

```
[user@host ~]$ docker stats
```

Exercise

Deployment of the *fortune* service (I)

- The *fortune* service provides random proverbs and sayings to its clients
- Composed of three server-side programs and a client-side program
- When the client (`fortune-client`) connects to the main server program (`fortune-serv`), the latter first contacts an authentication server (`fortune-auth`) to authenticate the client; then it contacts a database server (`fortune-db`) to fetch the proverb. Finally `fortune-serv` forwards the response back to the client.



Deployment of the *fortune* service (II)

- The `fortune-serv`, `fortune-auth` and `fortune-db` programs listen for TCP connections from any network interface
- They take the TCP port number to listen on as a command line argument
- The `fortune-serv` also accepts the following command line arguments:
 - IP address and port where the `fortune-auth` program is listening
 - IP address and port where the `fortune-db` program is listening
- The `fortune-client` programs takes as command line arguments the IP address and port where the `fortune-serv` program is listening.
- All the programs provide an help (and terminate) if invoked with the “-h” option

Exercise - Part A

1. Deploy the *fortune* service with Docker in single-host mode, creating a container for each of the three components (`fortune-db`, `fortune-auth` e `fortune-serv`)
2. To create the containers, first build an image for each component, defining proper Dockerfiles.
 - a. The `fortune-serv` program needs the `fortune-mod` package.
3. Run a client from the host
4. Run a client from within an additional container, using a Docker volume to import the `fortune-client` program inside the container

Suggestions

- Start from the Dockerfile example in the previous slides (with python)
- Use a different directory for each Dockerfile
- Put all the containers on an user-defined network, and assign them static IP addresses
 - To check what is the IP address of a container called xyz, use

```
[user@host ~]$ docker container inspect xyz | grep IPAddress
```
- Run the containers in interactive mode, to check the terminal output (-it options of the run command)
- To remove non-running containers (e.g. to make up for mistakes) use:

```
[user@host ~]$ docker container prune
```
- Use the Dockerfile ENV directive to extend the PATH environment variable to include “/usr/games”

Lurking bugs

- The `fortune-client` program also takes an optional “`--cookie`” command-line argument
 - A string of alphabetic lowercase character used (somehow) by the server to identify the anecdote to be selected
- Unfortunately, some bugs in the server-side code can cause the failure of one or more components with various symptoms
 - Kernel panic
 - Memory exhaustion on the machine where they run

Exercise - Part B

- Figure out how bugs are triggered.

Suggestions:

- Users reported that there is more than one bug, and they seem to depend on the value of the first character of the cookie
- Use per-container CPU stats to check for anomalies/crashes
- On a component crash, take all the actions necessary to restore the functionality of the *fortune* service
- Verify that the crash of a component is isolated to its container (the other containers and the host machine must not be affected)
 - Suggestion: use the `--cpus`, `--memory` and `--pids-limit` options of `docker run` to limit the resources assigned to a container