What is Docker all about?

Simon Pietro Romano

spromano@unina.it

Agenda

- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- Building images
- Container networking
- Docker-compose
- Docker-compose networking

Credits & references

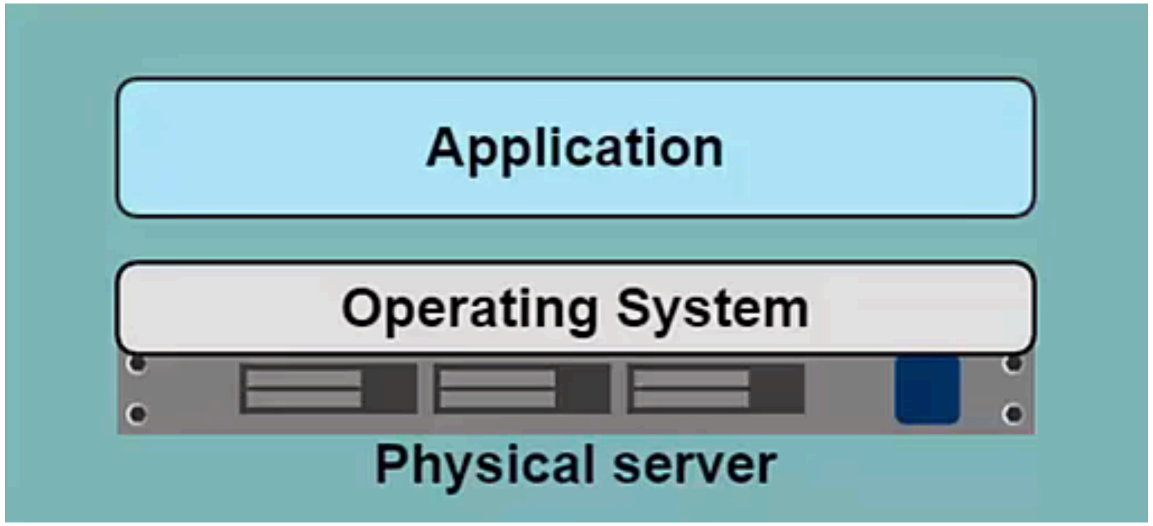
- https://docs.docker.com
- Introduction to Docker tutorial by Johnny Tu
  - https://training.docker.com/introduction-to-docker

Agenda

- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- Building images
- Container networking
- Docker-compose
- Docker-compose networking

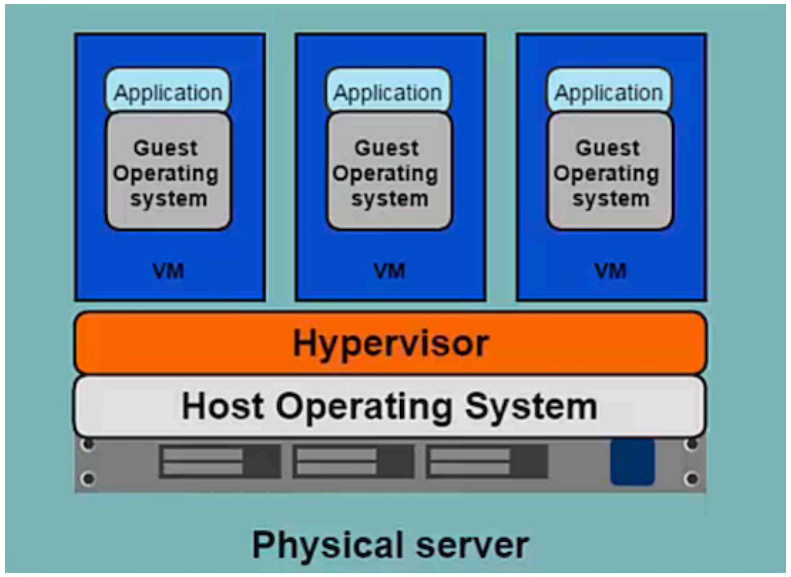
History - One Application on one physical server

- Problems:
  - Slow deployment times
  - Huge costs
  - Wasted resources
  - Difficult to scale
  - Difficult to migrate
  - Vendor lock in



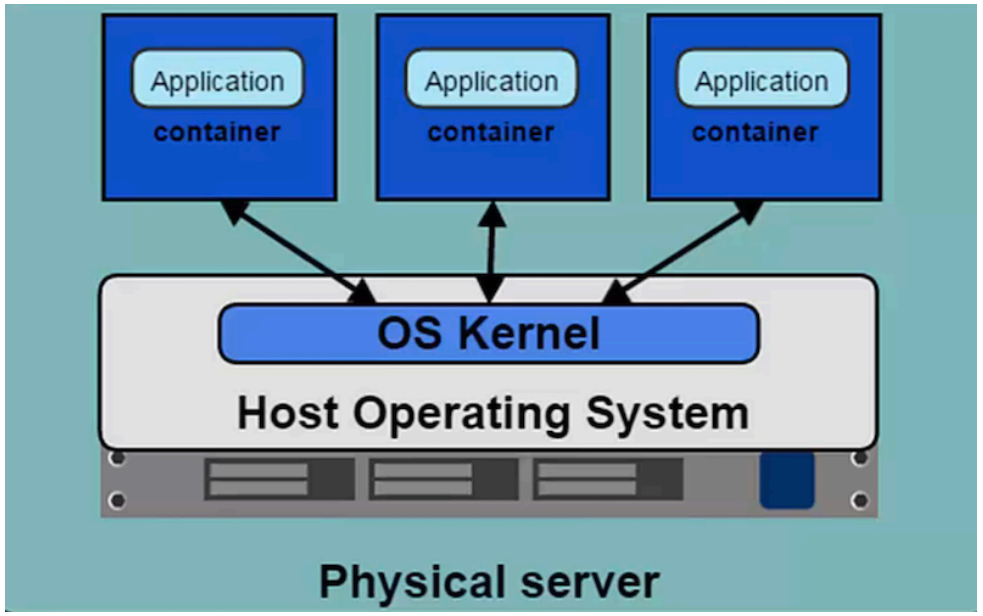
History - Hypervisor-based virtualization

- One physical server can contain multiple applications
- Each application runs in a virtual machine
- Benefits:
  - Better resource usage
    - One physical machine divided into multiple VM
  - Easier to scale
  - VM's in the cloud
    - Pay as you go
- Limitations:
  - Each VM still requires
    - CPU allocation
    - Storage
    - RAM
    - An entire guest operating system
  - The more VM's you run, the more resources you need
  - Guest OS means wasted resources



Containers

- Container-based virtualization uses the kernel on the host's operating system to run multiple guest instances
  - Also known as Operating-System-level virtualization
- The kernel of an operating system allows the existence of multiple isolated user-space instances
- LXC (Linux Containers) as first example (2008)
- Become very popular with the Docker project (2013)
- Each guest instance is called a *container*
- Each container has its own
  - Root filesystem
  - Processes
  - Memory
  - Devices
  - Network ports
- Containers isolate runtime environments



Containers vs. VMs

- Containers
  - Are more lightweight
  - No need to install guest OS
  - Less CPU, RAM, storage space required
  - More containers per machine than VMs
  - Greater portability

- VMs
  - More consolidated technology
  - Multitenancy
  - Guest Operating Systems other than Linux
  - Live migration

Agenda

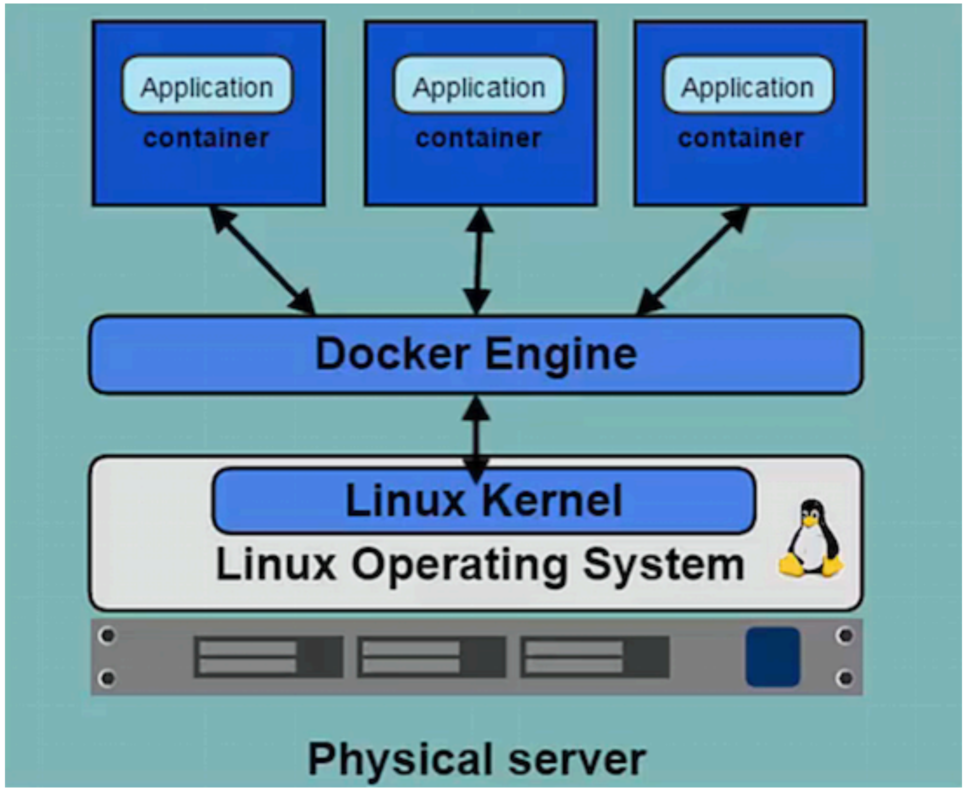
- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- Building images
- Container networking
- Docker-compose
- Docker-compose networking

What is Docker?

- Docker is an open source platform for developing, shipping and running applications using container virtualization technology
- The Docker Platform consists of multiple products/tools
  - Docker Engine
  - Docker Hub
  - Docker Machine
  - Docker Compose
  - Docker Swarm
  - Kitematic

Docker and the Linux Kernel

- Docker Engine (daemon) is the program that enables containers to be built, shipped and run
- Docker Engine uses Linux Kernel namespaces and control groups (cgroups)
- Namespaces give us the isolated workspace
- Cgroups limit, account for, and isolate the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes



Docker installation

- Docker needs Linux kernel
  - You may need a Linux Virtual Machine
  - The Docker Toolbox is an installer to quickly and easily install and setup a Docker environment on your Windows or Mac computer
- Follow the instructions at https://docs.docker.com/engine/installation/
- Verify your installation

```
$ sudo docker version
```

- Add your user account to the docker group (logout and re-login required)

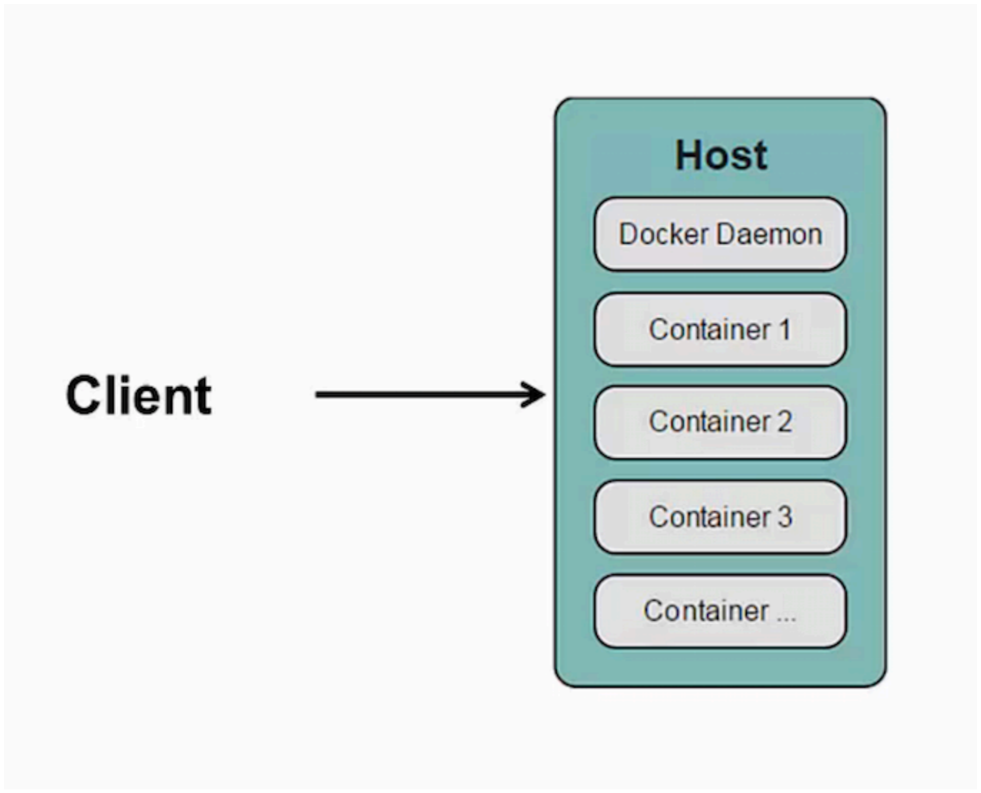
```
$ sudo usermod -aG docker <user>
```

- Run your first container

```
$ sudo docker run hello-world
```

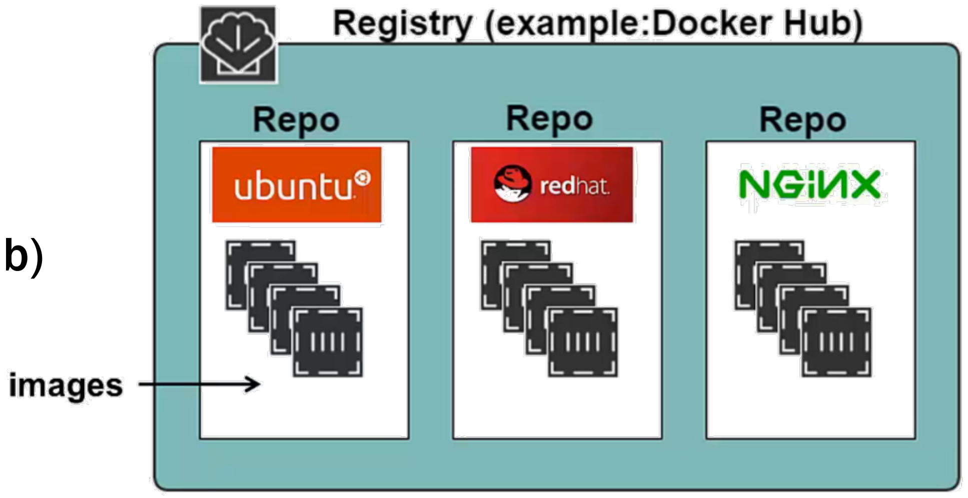
Docker client and Daemon

- Client/Server architecture
- Client takes user inputs and sends them to the daemon
- Daemon builds, runs, and distributes containers
- Client and daemon can run on the same or different hosts



Docker Images, Containers, Registry and Repositories

- **Images**
  - Read only template used to create containers
  - Built by you or other Docker users
  - Stored in the Docker Hub or your local Registry
- **Containers**
  - Isolated application platform
  - Contains everything needed to run your application
  - Based on images
- **Registry**
  - Is where we store images
  - Can be private or public (Docker Hub)
- **Repositories** are inside Registry

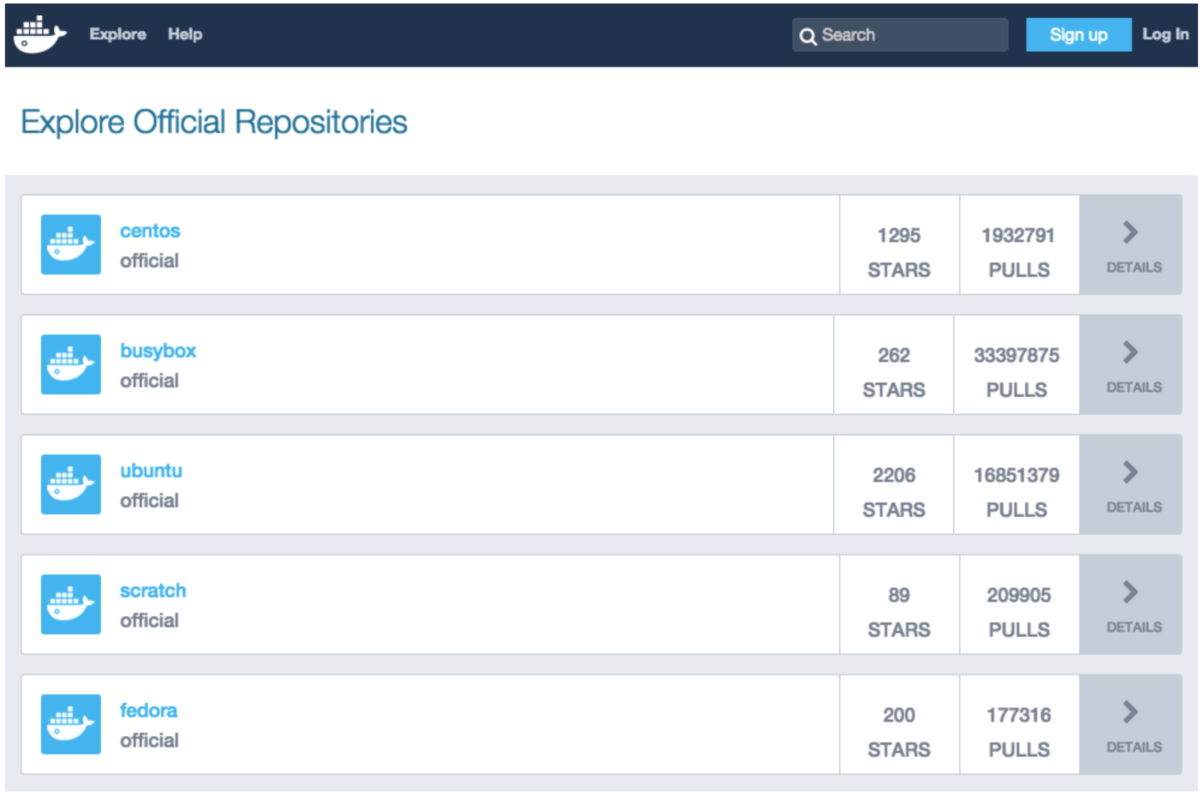


Benefits of Docker

- Separation of concerns
  - Developers focus on building their apps
  - System admins focus on deployment
- Fast development cycle
- Application portability
  - Build in one environment, ship to another
- Scalability
  - Easily spin up new containers if needed
- Run more apps on one host machine

Docker Hub (1/2)

- Official Registry maintained by Docker (the Company)
- Lots of images available for use
  - User-provided images (be careful!): `username/repository:tag`
  - Official images: `repository:tag`
  - Default tag is `latest`



Explore Official Repositories

centos official	1295 STARS	1932791 PULLS	DETAILS
busybox official	262 STARS	33397875 PULLS	DETAILS
ubuntu official	2206 STARS	16851379 PULLS	DETAILS
scratch official	89 STARS	209905 PULLS	DETAILS
fedora official	200 STARS	177316 PULLS	DETAILS

Docker Hub (2/2)

- Images can be downloaded from Docker Hub at any time:

```
$ docker pull ubuntu:14.04
```

- When downloaded, images are stored locally. Local images can be displayed:

```
$ docker images
```

- When creating a container, Docker will attempt to use a local image first
- If no local image is found, the Docker daemon will look in Docker Hub unless another registry is specified

Agenda

- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- **Working with containers**
- Building images
- Container networking
- Docker-compose
- Docker-compose networking

Working with containers (1/2)

- Spin up a container through the “run” command of the Docker CLI
  ```
  $ docker run [OPTIONS] IMAGE [COMMAND] [ARG...]
  ```
  Example:
  ```
  $ docker run ubuntu:14.04 ps aux
  ```
- Container with terminal
  - `-i` option (interactive mode) tells Docker to connect to STDIN on the container
  - `-t` option (TTY mode) specifies to get a pseudo-terminal
- «Detached» container
  - `-d` flag tells Docker to run the container as a daemon
  - Prints the id of the container created
- Observe container’s STDOUT
  ```
  $ docker logs <container id/name>
  ```
  - To follow the output, add the `-f` option
- Find your containers
  - Use `docker ps` to list running containers
  - Use `docker ps -a` to list all containers (includes containers that are stopped)

Working with containers (2/2)

- Stop a running container

  ```
  $ docker stop [OPTIONS] CONTAINER [CONTAINER...]
  ```

  or

  ```
  $ docker kill [OPTIONS] CONTAINER [CONTAINER...]
  ```

- Start a stopped container

  ```
  $ docker start [OPTIONS] CONTAINER [CONTAINER...]
  ```

- Execute a new process withing a running container

  ```
  $ docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
  ```

  e.g., to obtain a shell within a running container:

  ```
  $ docker exec -i -t <container id/name> /bin/sh
  ```

- Delete a (stopped) container

  ```
  $ docker rm [OPTIONS] CONTAINER [CONTAINER...]
  ```

- Delete a local image

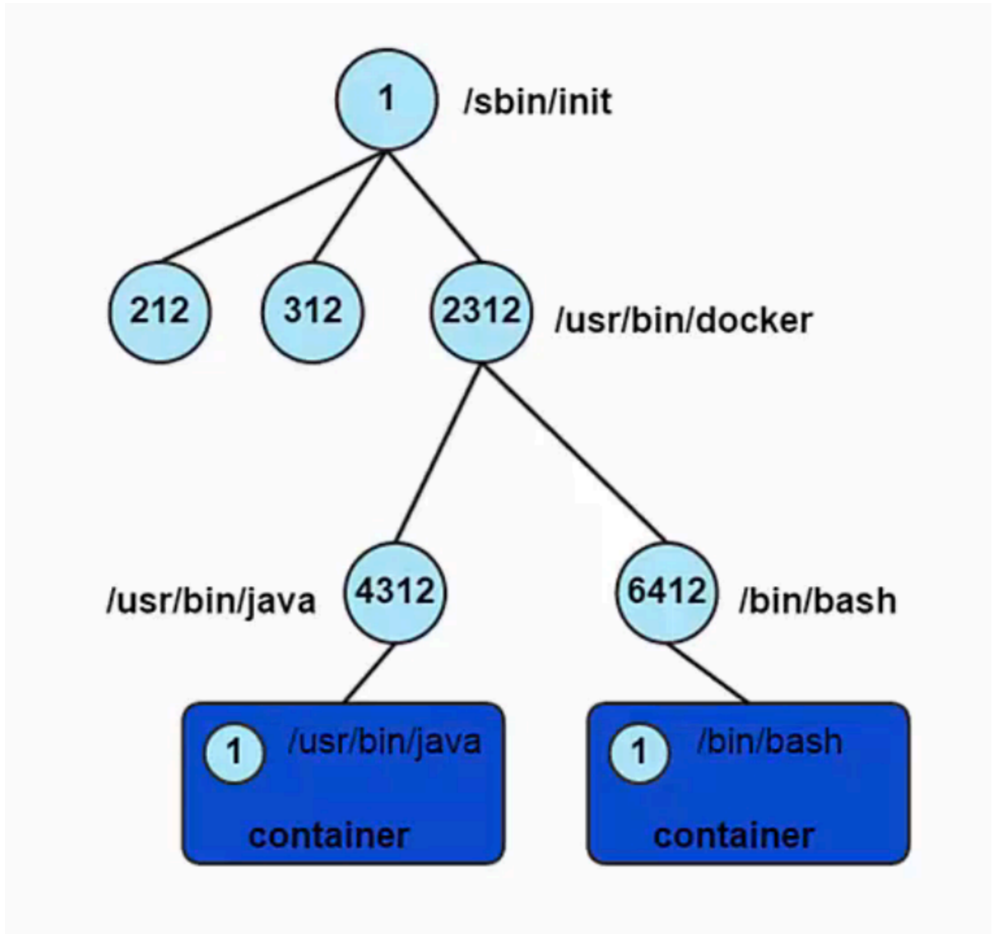
  ```
  $ docker rmi [OPTIONS] IMAGE [IMAGE...]
  ```

- Display containers' stats

  ```
  $ docker stats [OPTIONS] [CONTAINER...]
  ```

Container processes

- A container only runs as long as the process from your command is running
- Your command's process is always PID 1 inside the container



Volumes

- Special directories within a container’s file system, designed to persist data
- Independent from the containers life cycle
- Survive to containers deletion
- Can be mapped to a host folder
- A container can “mount” one or more volumes when created by using the `-v` option

```
$ docker run -it -v /home/pippo:/myvolumes/pippo
ubuntu:14.04 bash
```

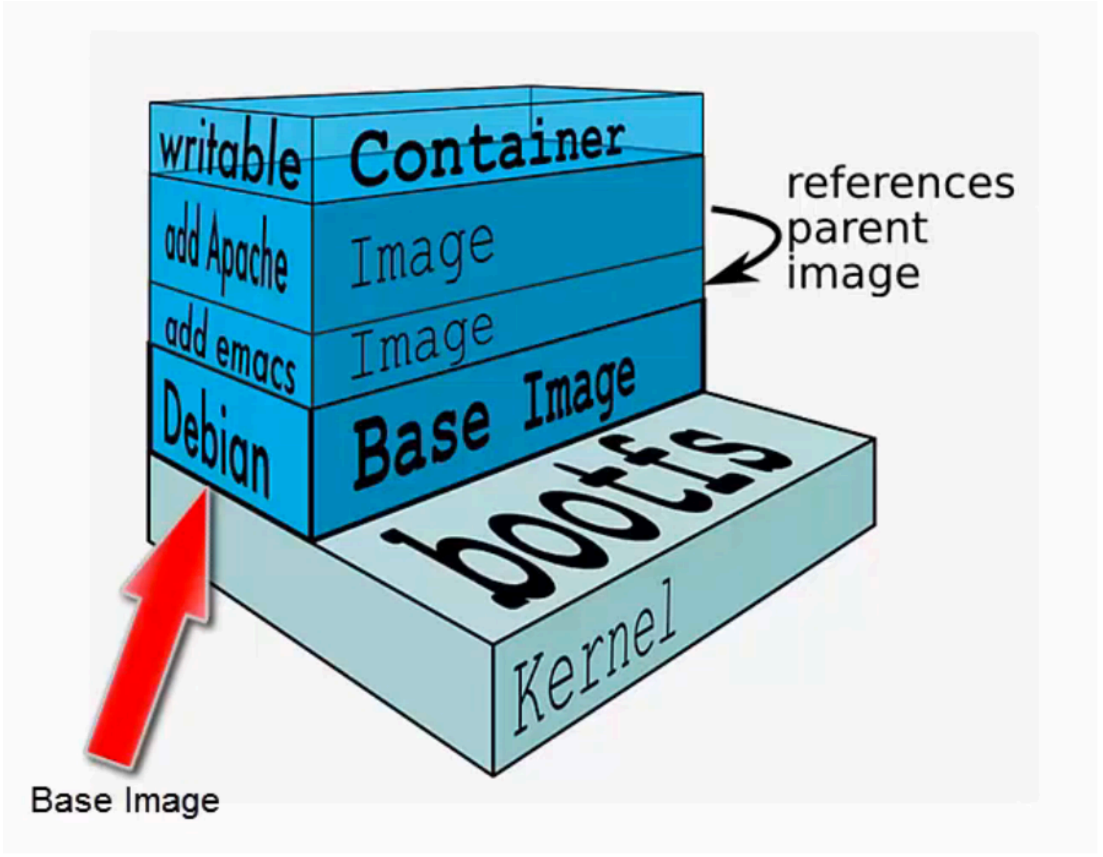
  - Paths specified must be absolute
- Can be shared among containers
  - `--volumes-from` option to `docker run`

Agenda

- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- **Building images**
- Container networking
- Docker-compose
- Docker-compose networking

Building Docker images

- Images consist of multiple *layers*
- A layer is itself an image
- Every image contains a base layer on top of which the image is built
- Layers are read-only
- Top layer is writable



Dockerfile

- It’s a configuration file that contains instructions for building a Docker image
- Instructions specify what to do when building the image
- FROM instruction specifies what the base image should be
- RUN instruction specifies a command to execute
  - Each RUN instruction will execute the command on the top writable layer
  - Can aggregate multiple RUN instructions by using &&
- ADD instruction copies files from the local filesystem or from the network
- CMD instruction defines a default command to execute when a container is created
  - Performs no action during the build process
  - Can only be specified once in a Dockerfile
  - Can be overridden at runtime
- ENTRYPOINT instruction, like CMD, defines the command that will run when a container is executed
  - Run time arguments and CMD instruction are passed as parameters to the ENTRYPOINT instruction
  - Cannot be overridden at runtime
  - Container essentially runs as an executable

Building from Dockerfile – example

- Dockerfile:

```
FROM ubuntu:14.04
RUN apt-get update && apt-get -y install traceroute
CMD /bin/bash
```

- Build instruction:

```
$ docker build -t [repository:tag] [path]
```

e.g.,

```
$ docker build –t spromano/traceroute:0.0.1 .
```

- Push to Docker Hub:

```
$ docker push spromano/traceroute:0.0.1
```

Agenda

- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- Building images
- **Container networking**
- Docker-compose
- Docker-compose networking

Container networking – default networks

- When you install Docker Engine, it creates three networks automatically
- List all networks available on a Docker host

```
$ docker network ls
NETWORK ID          NAME                DRIVER
7fca4eb8c647        bridge              bridge
9f904ee27bf5        none                null
cf03ee007fb4        host                host
```

- The *bridge* network represents the `docker0` interface which is automatically created during Docker installation
  - It's the default networking mode
  - Containers connected to the *bridge* network are "NAT-ted" through the docker0 interface
  - Containers connected to the *bridge* network are on the same LAN
- The *host* network represents the network stack of the host machine
  - Containers connected to the *host* network share the network stack with the host (and among each other)
- Containers connected to the *none* network have no networking capabilities

Container networking – default networks

- A container can be connected to one of the three default networks through the `--net` option to the `docker run` command

  e.g.,

  ```
  $ docker run -ti --net host ubuntu:14.04 bash
  ```

- The `docker network inspect` command returns information about a network and containers connected to it
- Containers connected to the *bridge* network can talk to each other on the same LAN
  - Need to know other containers' IP addresses!
- A container can be linked to one or more running containers through the `--link` option to the `docker run` command

  e.g.,

  ```
  $ docker run -td --name database mysql
  $ docker run -td --name tomcat --link database:db tomcat:7
  ```

Port mapping/publishing

- Containers connected to a bridged network have their own network stack (IP address and ports)
- Ports on the container can be mapped to ports on the host machine through the `-p` option to the `docker run` command

  ```
  $ docker run -td -p 8000:8080 tomcat:7
  ```

  - All packets received by the host on port 8000 are forwarded to the container's port 8080
- Port ranges can be mapped as well

  ```
  $ docker run -td -p 10000:20000/UDP tomcat:7
  ```

- Mapped ports are shown by `docker ps`

User-defined networks

- You can create your own user-defined networks that better isolate containers
- Docker provides some default **network drivers** for creating these networks
  - *bridge* network driver
  - *overlay* network driver
- You can write your own network driver plugin
- You can create multiple networks
- You can add containers to more than one network
- Containers can only communicate within networks but not across networks
- Docker daemon runs an embedded DNS server to provide automatic service discovery for containers connected to user-defined networks
  - Name resolution requests from the containers are handled first by the embedded DNS server
  - If the embedded DNS server is unable to resolve the request it will be forwarded to any external DNS servers configured for the container
  - Within a user-defined network, container names are resolvable through the embedded DNS

User-defined networks – bridge driver (1/2)

- A bridge network is the easiest user-defined network

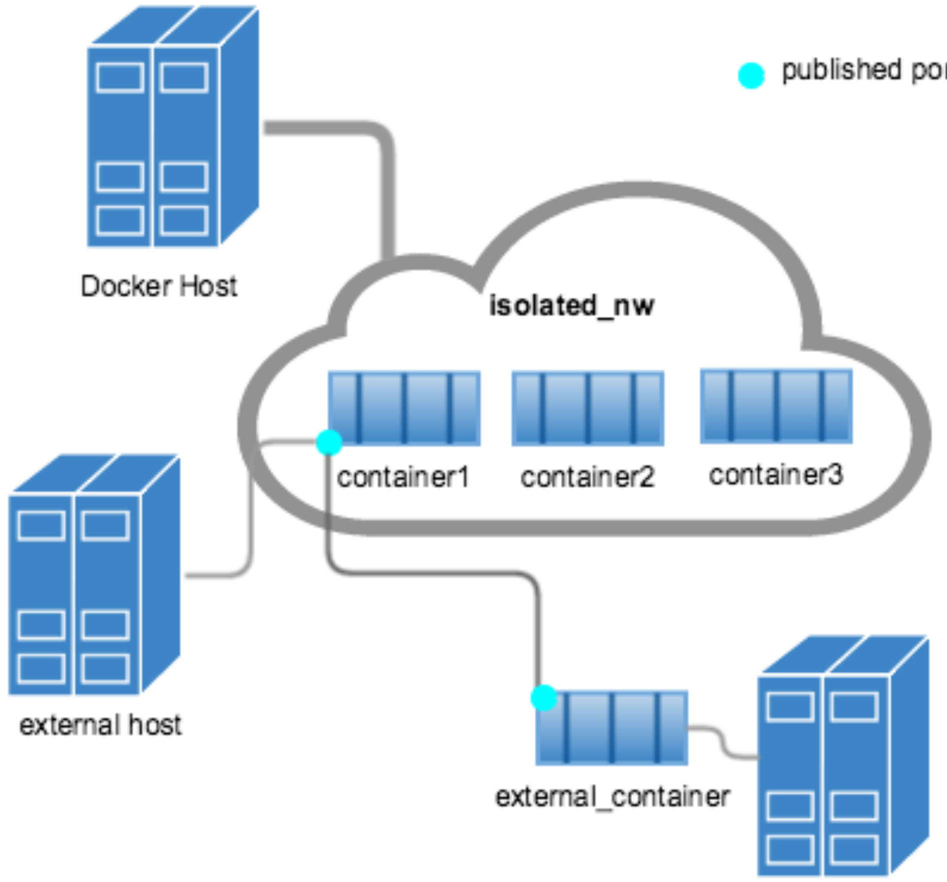
```
$ docker network create --driver=bridge <network_name>
$ docker network inspect <network_name>
```

- Uses the *bridge* network driver provided by Docker
- It's similar to the default `docker0` bridge network
- After you create the network, you can launch containers on it

```
$ docker run --net=<network_name> ...
```

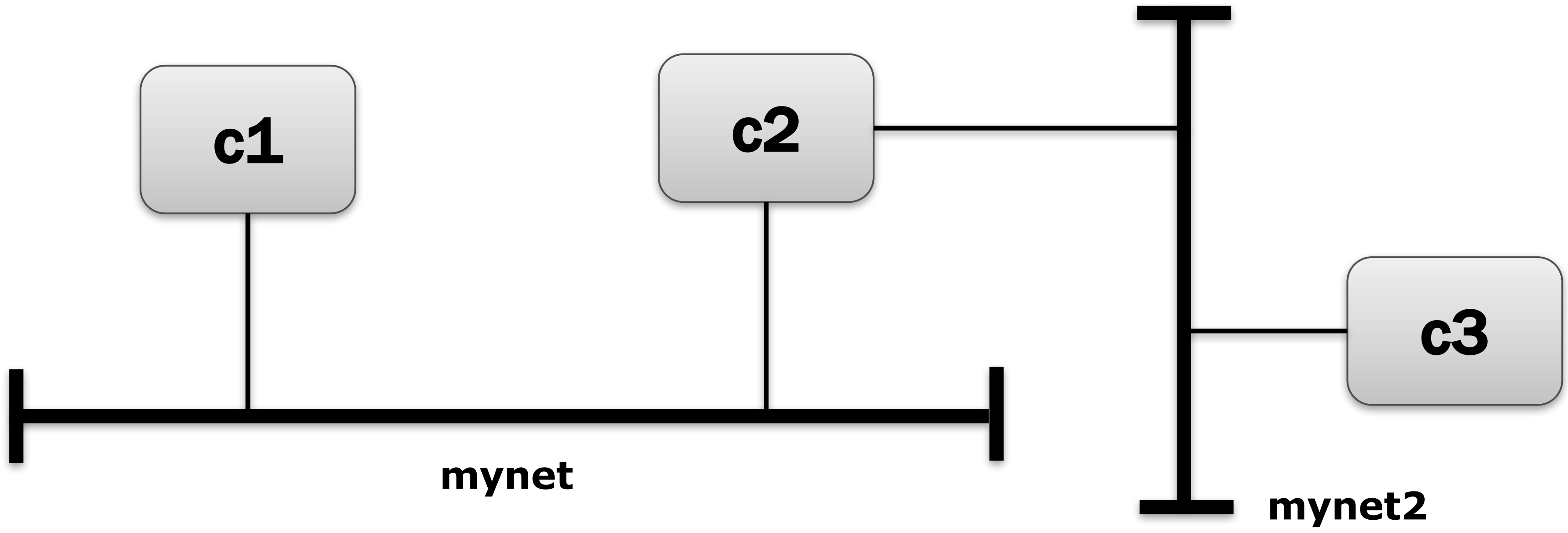
- The containers you launch into this network must reside on the same Docker host
- Each container in the network can immediately communicate with other containers in the network
- The network itself isolates the containers from external networks
- You can expose and publish container ports on containers in this network
- Containers can be attached to a network at any time

User-defined networks – bridge driver (2/2)



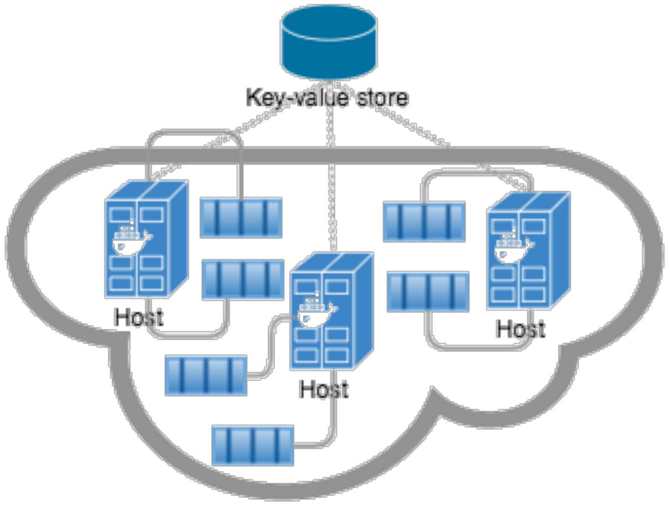
User-defined networks – example

```
$ docker network create --driver=bridge mynet
$ docker run -td --name c1 --net mynet ubuntu:14.04 bash
$ docker run -td --name c2 --net mynet ubuntu:14.04 bash
$ docker network create --driver=bridge mynet2
$ docker run -td --name c3 --net mynet2 ubuntu:14.04 bash
$ docker network connect mynet2 c2
```



User-defined networks – overlay driver

- Docker's *overlay* network driver supports multi-host networking natively out-of-the-box
- The overlay network requires a valid key-value store service
  - Consul, Etcd, and ZooKeeper are currently supported
  - Before creating a network you must install and configure your chosen key-value store service



```
$ docker network create -d overlay
```

Sharing the network stack

- Containers running on the same host can share the network stack
- The option `--net=container:NAME_or_ID` to the `docker run` command tells Docker to put container's processes inside a network stack that has already been created for another container
- The new container's processes will be confined to their own filesystem and process list and resource limits, but will share the same IP address and port numbers as the first container
- Processes on the two containers will be able to connect to each other over the loopback interface
- Example:

```
$ docker run -td --name cont1 ubuntu:14.04 bash
$ docker run -td –name cont2 --net=container:cont1 ubuntu:14.04 bash
```

Pipework

- https://github.com/jpetazzo/pipework
- Lets you connect together containers in arbitrarily complex scenarios
- Works with "plain" LXC containers and with Docker
- Allows to create a new network interface inside a container and to set networking parameters (IP address, netmask, gateway)
    - This new interface becomes the default one for the container
- Sintax:

```
$ pipework <hostinterface> [-i containerinterface] <guest>
<ipaddr>/<subnet>[@default_gateway] [macaddr][@vlan]


$ pipework <hostinterface> [-i containerinterface] <guest> dhcp
[macaddr][@vlan]
```

Agenda

- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- Building images
- Container networking
- Docker-compose
- Docker-compose networking

Docker-compose

- Compose is a tool for defining and running multi-container Docker applications
- https://docs.docker.com/compose/install/
- Applications are made of (micro)services
- Use a single file (i.e., the Compose file) to configure your application's services
  - docker-compose.yml
  - YAML… Yet Another Markup Language!
- Also useful to run a single container
  - Options, volumes, ports mapping, …
- The `docker-compose up` command looks for the Compose file in the working directory and starts your entire app
  - Pass the `-d` parameter to daemonize
- Compose has commands for managing the whole lifecycle of your application
  - Start, stop and rebuild services
  - View the status of running services
  - Stream the log output of running services
  - Run a one-off command on a service

docker-compose.yml example

```yaml
version: "2"
services:
  tomcat:
    image: alexamirante/tomcat7
    volumes:
      - ./webapps:/var/lib/tomcat7/webapps
      - ./logs:/var/log/tomcat7
    ports:
      - "80:8080"
      - "8022:22"
    links:
      - mysql:db
  mysql:
    image: mysql
    env_file: mysql.env
    volumes:
      - ./mysql:/var/lib/mysql
```

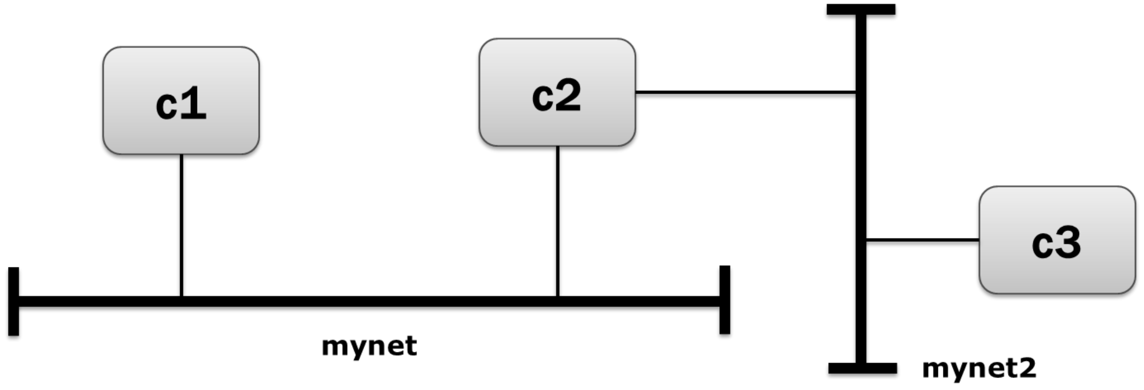
Agenda

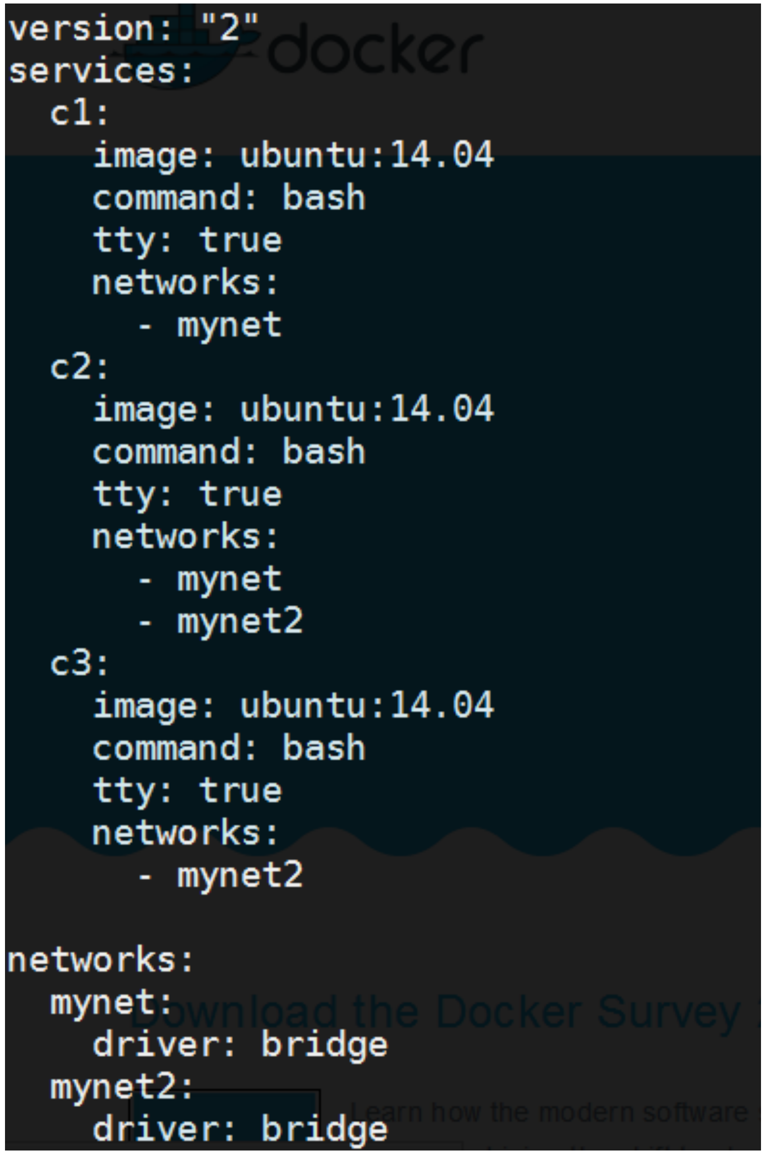
- Containers vs. Virtual Machines
- Docker Platform Overview and Terminology
- Working with containers
- Building images
- Container networking
- Docker-compose
- Docker-compose networking

Docker-compose networking

- By default Compose sets up a single network for your app
- Each container for a service joins the default network and is both *reachable* by other containers on that network, and *discoverable* by them at a hostname identical to the container name
- Your app's network is given a name based on the "project name", which is based on the name of the directory it lives in
  - If the docker-compose.yml is inside a directory called "myapp", a network called `myapp_default` is created
  - You can override the project name with the `--project-name` flag
- Containers belonging to different apps will join different networks and won't be able to communicate by default
- You can specify custom networks in the Compose file with the top-level `networks` key
  - Lets you create more complex topologies and/or specify custom drivers and options
- Each service can specify what network to connect to with the service-level `networks` key
  - Can also connect to "external" networks, i.e., networks defined outside of Compose

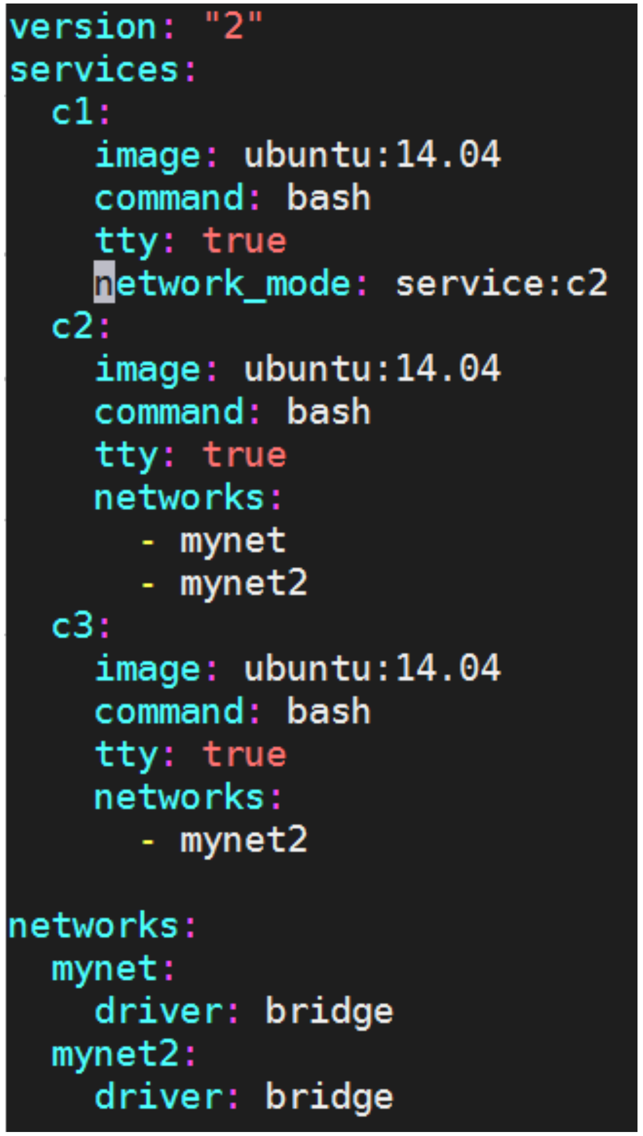
Docker-compose networking: example



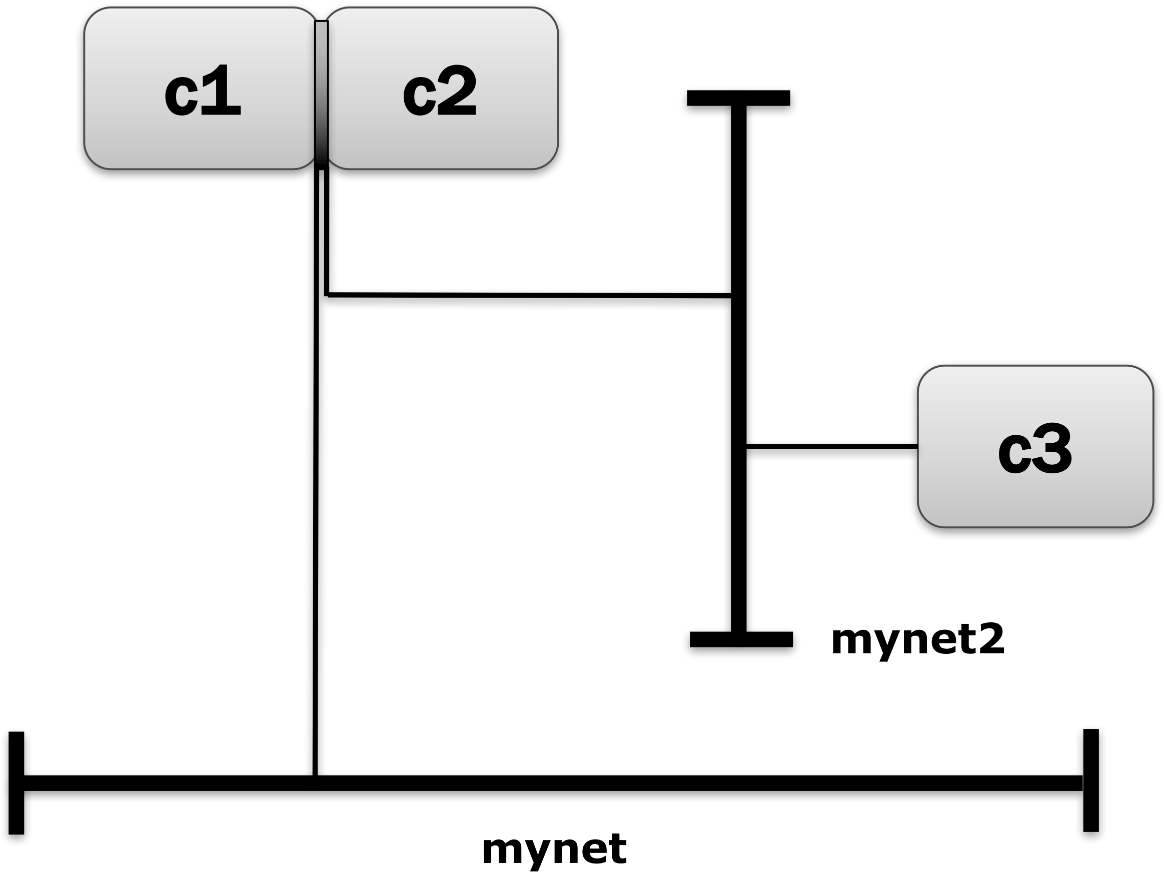


```
version: "2"
services:
  c1:
    image: ubuntu:14.04
    command: bash
    tty: true
    networks:
      - mynet
  c2:
    image: ubuntu:14.04
    command: bash
    tty: true
    networks:
      - mynet
      - mynet2
  c3:
    image: ubuntu:14.04
    command: bash
    tty: true
    networks:
      - mynet2

networks:
  mynet:
    driver: bridge
  mynet2:
    driver: bridge
```

Docker-compose networking: example 2



```
version: "2"
services:
  c1:
    image: ubuntu:14.04
    command: bash
    tty: true
    network_mode: service:c2
  c2:
    image: ubuntu:14.04
    command: bash
    tty: true
    networks:
      - mynet
      - mynet2
  c3:
    image: ubuntu:14.04
    command: bash
    tty: true
    networks:
      - mynet2

networks:
  mynet:
    driver: bridge
  mynet2:
    driver: bridge
```



Containers clustering

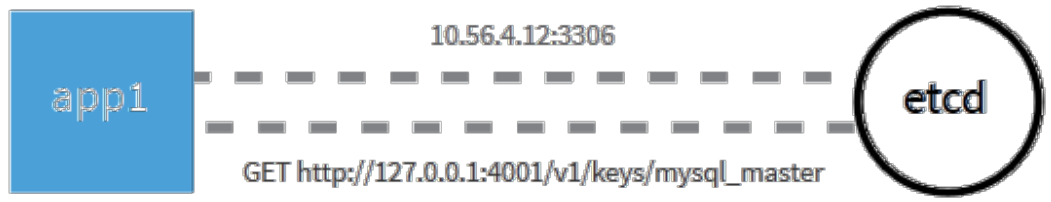
- Clustering solutions aim at turning a pool of Docker hosts into a single, virtual Docker host
- Several orchestration tools
  - Docker Swarm
  - CoreOS Fleet
  - Kubernetes
  - Apache Mesos
- Provide mechanisms to
  - Schedule/start containers on appropriate hosts
  - Handle conflicts
  - Scale
  - Failover
  - …

CoreOS

- A minimal Linux distro (~100MB) designed for security, consistency, and reliability
- Uses Linux containers (LXC/libcontainer) to manage your services at a higher level of abstraction
  - A single service's code and all dependencies are packaged within a container that can be run on one or many CoreOS machines
  - Containers don’t run their own Linux kernel or require a hypervisor
    - Almost no performance overhead
- Runs on almost any platform, including Vagrant, Amazon EC2, QEMU/KVM, VMware and OpenStack and your own hardware
- Main building blocks:
  - Docker
  - Etcd
  - Fleet
  - Cloud-Config
- You should construct a docker container for each of your services, start them with fleet and connect them together by reading and writing to etcd

Etcd

- Key value store, written in go
- Used for configuration and service discovery
  - Each host provides a local endpoint for etcd, which is used for service discovery and reading/writing configuration values
- All changes are reflected across the entire cluster



Read the master database IP address from etcd.

Fleet

- Tool that presents your entire cluster as a single init system
- Uses systemd
  - Receives systemd *unit files* and schedules them onto machines in the cluster based on declared conflicts and other preferences encoded in the unit file
- Can deploy high availability services by ensuring that service containers are not located on the same machine, availability zone or region
- fleetctl control tool

```
$ fleetctl list-machines
$ fleetctl list-units
$ fleetctl start <unit_file.service>
$ fleetctl destroy <unit_file.service>
```

Cloud-config

- Yser-supplied configuration file
- Allows to declaratively customize various OS-level items
    - Configure machine parameters
    - Launch systemd units on start-up
    - ...
- Processed during each boot
- Will, at a minimum, tell the host how to join an existing cluster and command the host to boot up two services called etcd and fleet

Quick demo

- Use Vagrant to locally create a cluster
  - Vagrant CoreOS repo:

    `git clone https://github.com/coreos/coreos-vagrant.git`
- The *config.rb* file contains a few useful settings about Vagrant environment
  - E.g., create 3 machines in the cluster