TELECOM SudParis

INSTITUT POLYTECHNIQUE DE PARIS

Operating system-level virtualization

Mathieu Bacou
mathieu.bacou@telecom-sudparis.eu

Actually, VMs are bad

- A guest OS is required
  - Overhead of deployment and maintenance
- Very slow: a new VM starts in minutes
  - Allocate disk, deploy image, create VM, boot guest OS
  - Not quick enough for workload bursts
- Coarse grained:
  - In resource management: allocate to a full OS
  - In application architecture: monolithic layout
    - Horizontal scaling must replicate whole VM instead of components

Introducing: OS-level virt.

- Cloud users don’t want to run OSes
  - They want to run their applications
- How to share cloud resources closer to the applications?
  - Virtualization layer just between the OS and the application
- Virtualize the OS for multiple applications at the same time!
- An OS executes a container engine that runs containers
  - Docker, LXC, OpenVZ...

Actors of OS-level virtualization

I. Container engine

II. Container

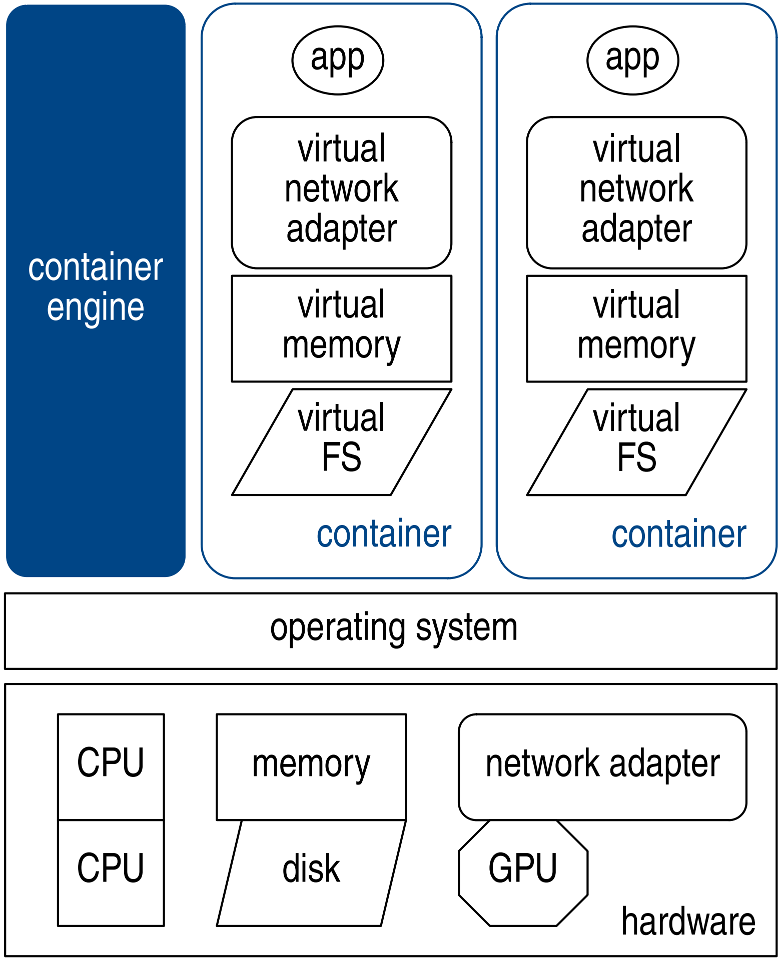
Containers and engines

- Engine:
  - Manage container lifecycle: create container from image, start and stop containers...
  - Handle out-of-container tasks: virtual networking…
  - Many engines for many uses: generic, HPC, scientific…
    - With interchangeable underlying container engine cores
- A container image packages an application and its runtime
  - Business core, dependencies, semi-static configuration
  - Registries of reusable images (DockerHub, local…)
    - Typically written in a portable, constant manner

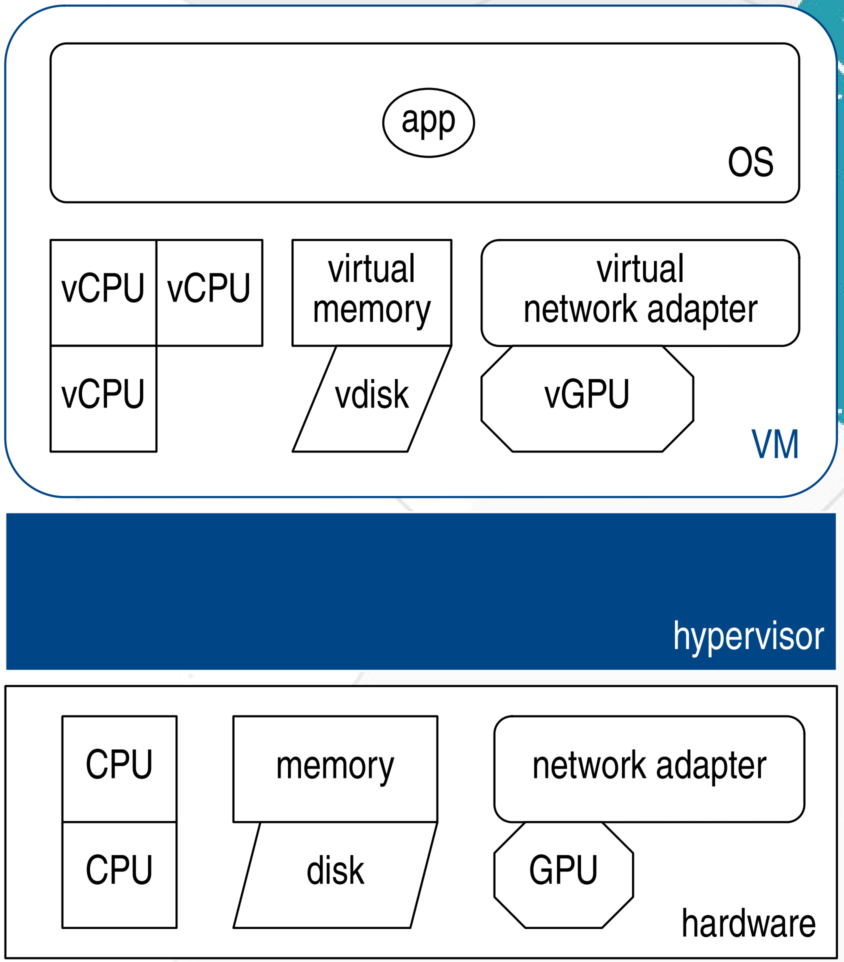
Containers

- Container: isolated and limited virtual copy of the host OS
  - Deploys the image to “fill” the virtual copy
- Isolation: users, devices, processes…
  - Virtual filesystem: built from container image
- Limits: CPU, memory, I/O…
  - Also monitoring

Comparison with HW virt.



Stack for OS-level virtualization



Stack for hardware virtualization

Comparison with HW virt.

	OS-level virtualization	Hardware virtualization
Security	-	+
Usability	+	-
Performance	0	0
Startup time	+	-
Image size	+	-
Memory overhead	+	-

- Containers are better overall for cloud-native applications
  - Applications architectured to be deployed on the cloud
- With reduced security
- VMs still have use cases: interactive environment, robustness...

Demo: Docker

- Creation and usage of a Docker container:
  - Run an interactive image
  - Pull and run a daemon service
  - List images, monitor containers
- Docker is a bit low-level for applications: docker-compose for multi-component apps

Build containers: two ways

1) Interactively
   - From a base distribution image (Ubuntu, Alpine…)
   - Use package manager
   - `docker commit` to tag the current state of the container as an image
   - Testing and experimenting
2) Dockerfile
   - DSL to describe how to install and configure app
   - Proper method: clean, reusable, reproducible

Build containers: Dockerfile

Dockerfile for docker/cowsay

Start from base image	`FROM alpine`
Execute commands to build and configure the image	`RUN apk add --no-cache perl`
Add external files	`COPY cowsay /usr/local/bin/cowsay` `COPY docker.cow /usr/local/share/cows/default.cow`
Set default executable	`ENTRYPOINT ["/usr/local/bin/cowsay"]`

- And then: `docker build -t namespace/name:tag .`
- Can start from empty image: `FROM scratch`
  - Used by distribution base images: build from archive
- Also declare users, volumes, network ports

Internals of Docker

I. Isolation
II. Limit
III. Operation control
IV. Virtual filesystem

Isolation: namespaces

- Provide an isolated view of the OS
  - `chroot` on steroids (CHange ROOT of a process)
- 8 dimensions:

1) `mnt`: mount points
   - I.e. filesystem
2) `pid`: PID hierarchy
   - First process in the container is PID 1
3) `net`: network facilities
   - Interfaces, ports, protocol stack…
4) `ipc`: interprocess communication
   - Semaphore, message queue, shared mem
5) `user`: users, groups and privileges
   - Mappings of UIDs/GIDs between host and container
     - UID 0 is root, available in container: if you escape the container, you are root!
6) `uts`: hostname
   - Stands for "UNIX TimeSharing", or said otherwise: multi-user in UNIX
7) `time`: clock
8) `cgroup`: control groups (next slide)

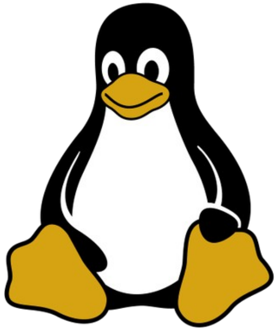
Limit: control groups (cgroups)

- Constrain resource usage
  - Also monitoring facilities
- 12 dimensions:

1) `cpu`: CPU time
2) `cpuacct`: CPU accounting
3) `cpuset`: CPU pinning
4) `memory`: memory and swap
5) `devices`: access rights to devices
6) `freeze`: freeze, suspend processes
7) `net_cls`: network packets classes
8) `net_prio`: network packets priority
9) `blkio`: block devices (disk) I/O
10) `perf_event`: performance mon.
11) `hugetlb`: huge pages usage
12) `pids`: number of processes

Demo: namespaces & cgroups

- Spawn new process in namespaces
- Put process in control groups
  - Set limit and monitor resource usage
- Demonstrated filesystem interface
  - Also a programmatic interface with syscalls



Operation control: caps and MAC

- Capabilities: selectively drop root privileges
  - Remove privileges from a “root” container
- Mandatory Access Control (MAC): system-level operational policies
  - SELinux, AppArmor...
- 40 capabilities (CAP_XXX):

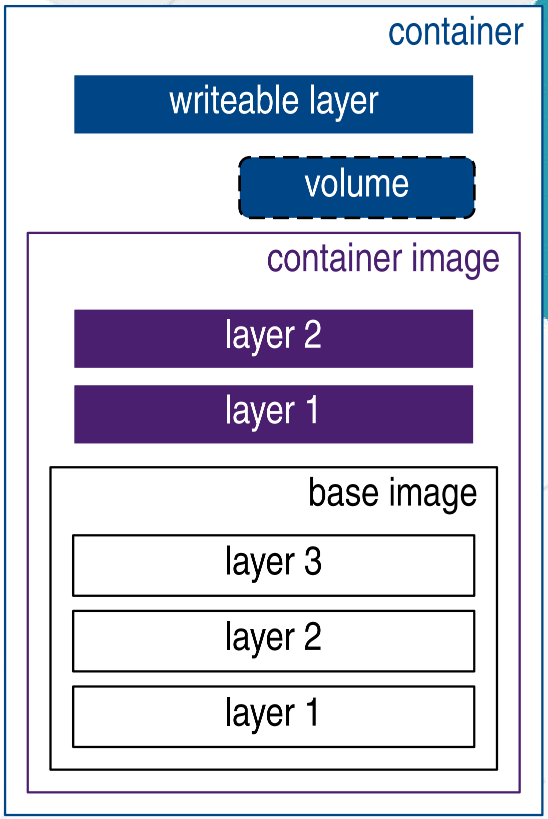
1) CHOWN: change owner
2) SETGID/SETUID: change process GIDs/UIDs
3) KILL: send signals
4) NET_ADMIN: network admin
5) NET_RAW: use RAW sockets
6) SYS_ADMIN: system admin (mount...)
7) SYS_CHROOT: change root path of process
8) SYS_MODULE: (un)load kernel modules
9) SYS_NICE: change process niceness
10) SYS_TIME: change system clock

Virtual filesystem

- Isolated filesystem: `mnt` namespace
  - Also with `chroot`
- Two parts:
  - Container image: basis for virtual filesystem
    - Docker specifics, see next
  - Volumes: external data storage
    - Mounted into the virtual FS of the container

Container image with Docker

- An image has layers
  - Like git commits
  - Reusable by other images, caching
  - `docker image history IMAGE_NAME`
- Layers from Dockerfile are read-only
  - For execution, add a writeable layer
  - Use copy-on-write to modify files from lower layers
- Union file system: virtual FS driver for layers
  - Many drivers: AUFS, OverlayFS, devicemapper...



Container image layers and volumes

Union FS and copy-on-write

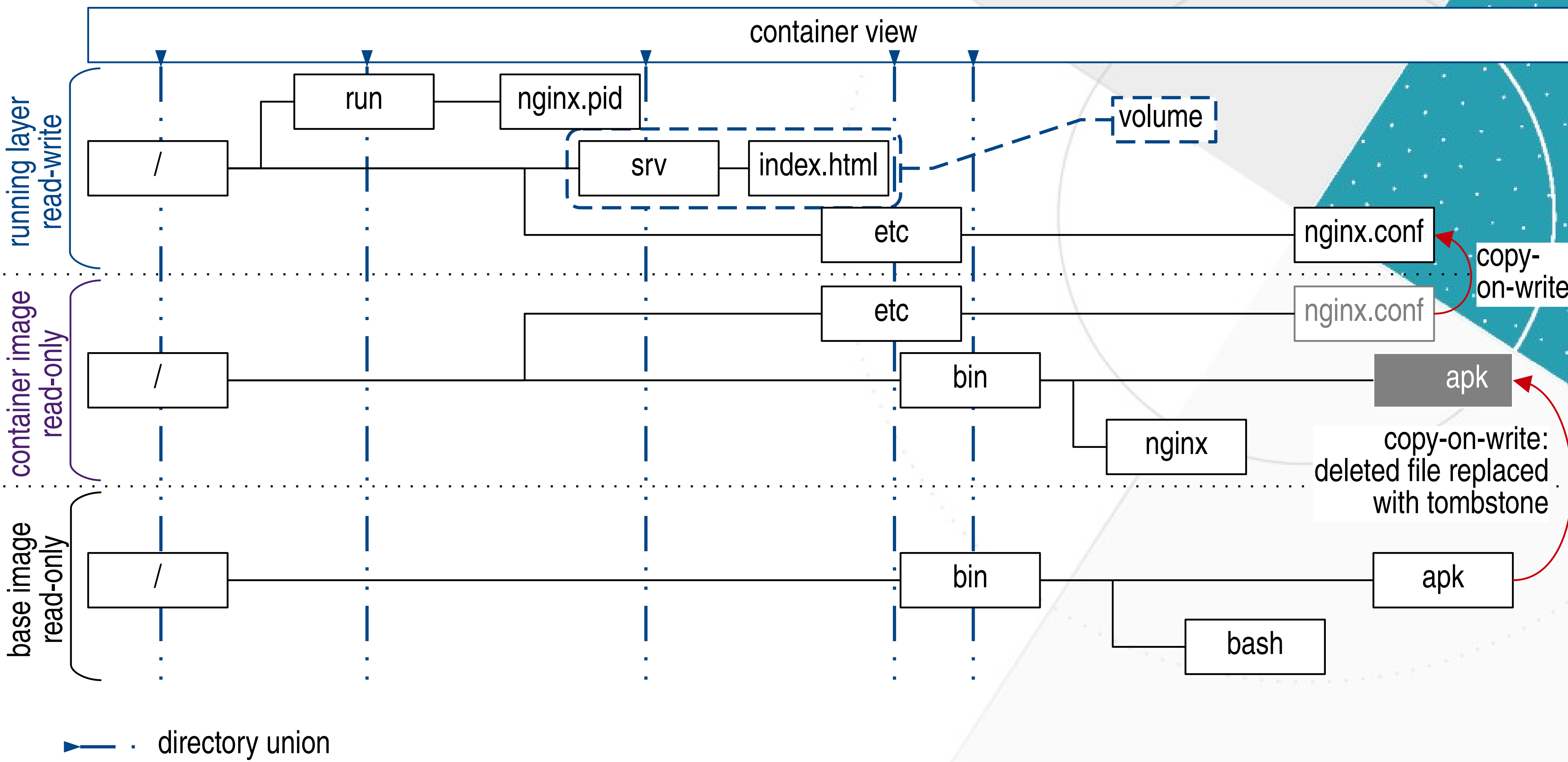
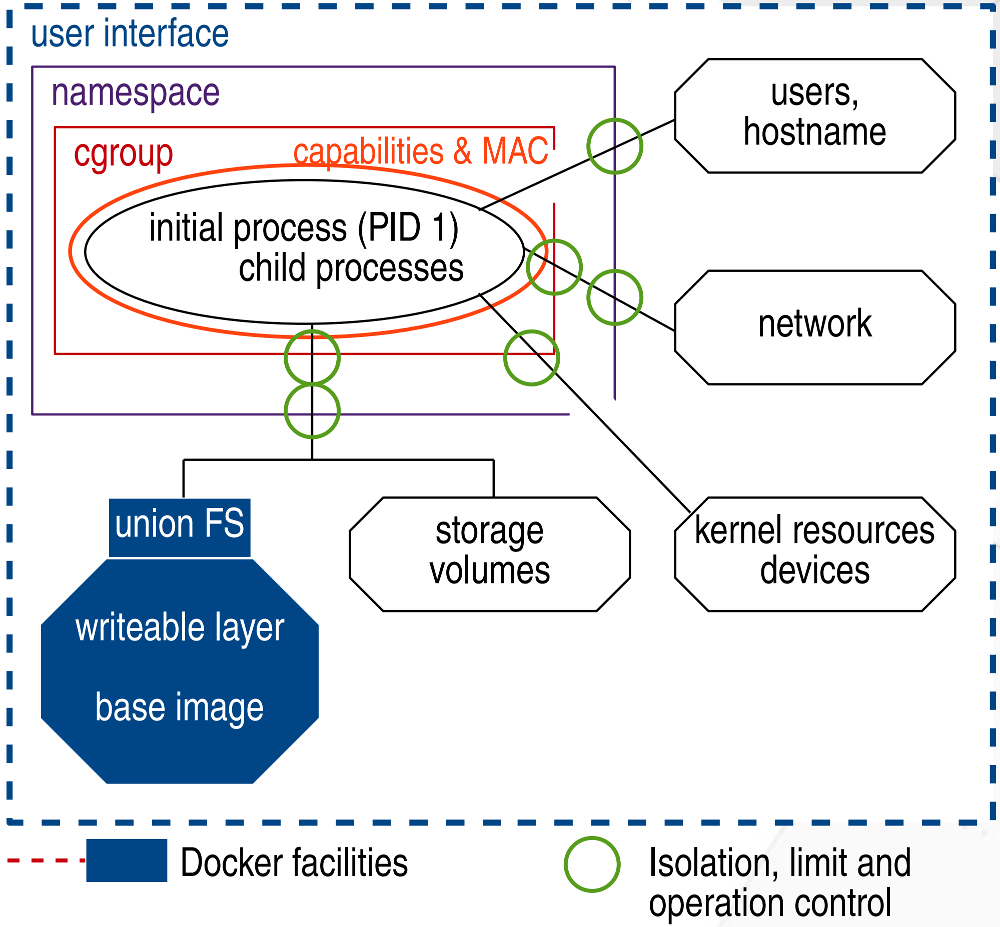


Illustration of union filesystem and copy-on-write

Docker container engine



Everything else are features from Linux kernel!

Low-level facilities of Docker container engine

Containers for the cloud

I. Application architecture in the cloud

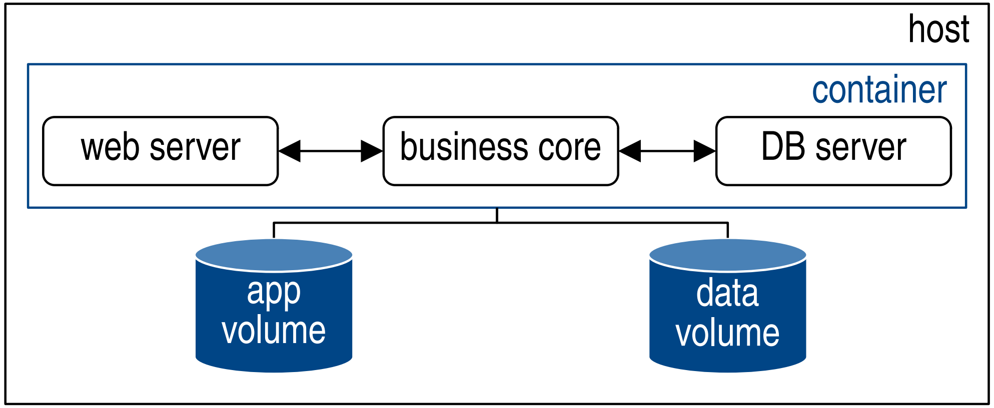
II. Micro-services

III. Orchestration

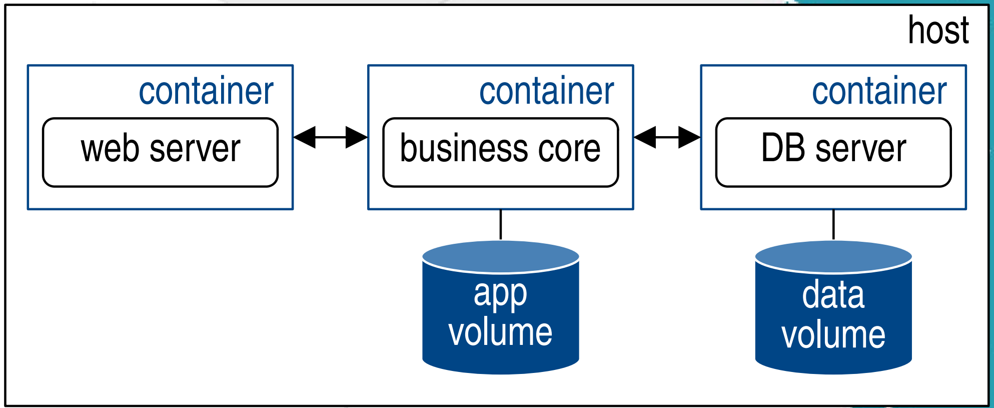
Cloud application architecture

- Historic pattern: monolithic application
  - All components are ad-hoc, tightly coupled
- Unfit for the cloud
  - Must manage all components at once for scalability, deployment, service quality
  - Hard to reconfigure
- New paradigm enabled by container: micro-services

Micro-services



❌ Monolithic container



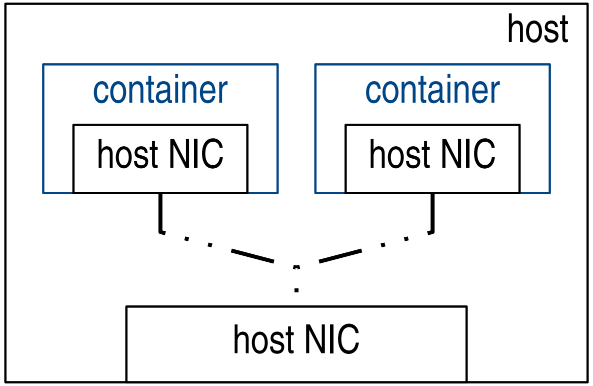
✅ Composition of containers: micro-services

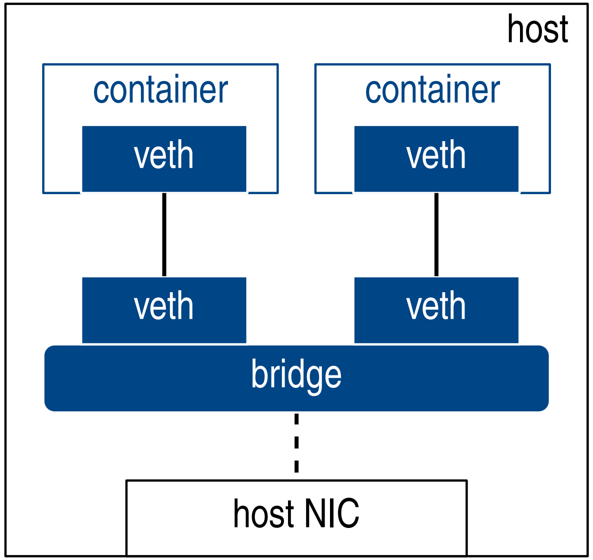
- Components as processes
  - Manual interfacing
    - Need in-container PID 1 to run multiple processes
  - Cons of monolithic apps
- Components as containers
  - Max reuse of images
  - High flexibility, easy configuration
  - Fine-grained scalability

Network for micro-services

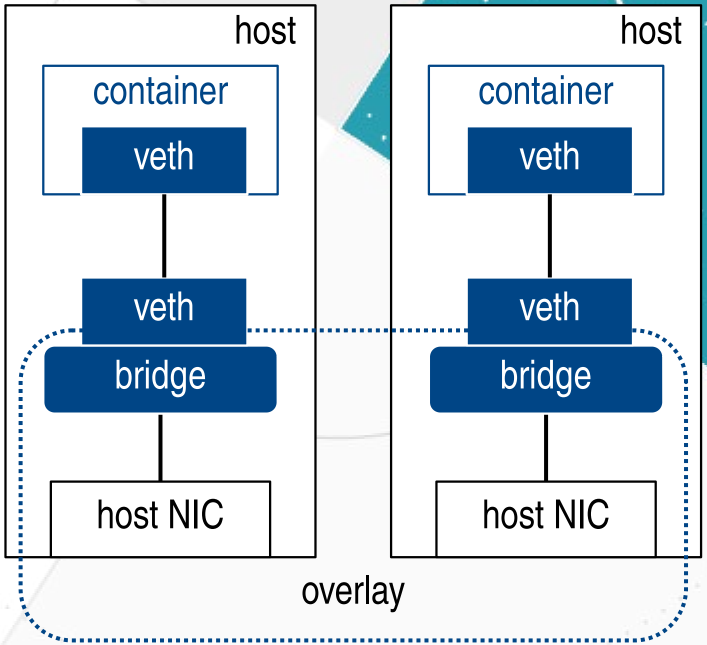
- Configuration of network by Docker
  - Dedicated links between component containers
  - Controlled link to the Internet
- Network drivers:
  - Host: expose host network devices to the container
  - Bridge: local virtual network
    - Can be exposed to the Internet
  - Overlay: inter-host inter-container network
  - None: no networking

Network for micro-services





veth: Virtual ETHernet



Host

Bridge

Overlay

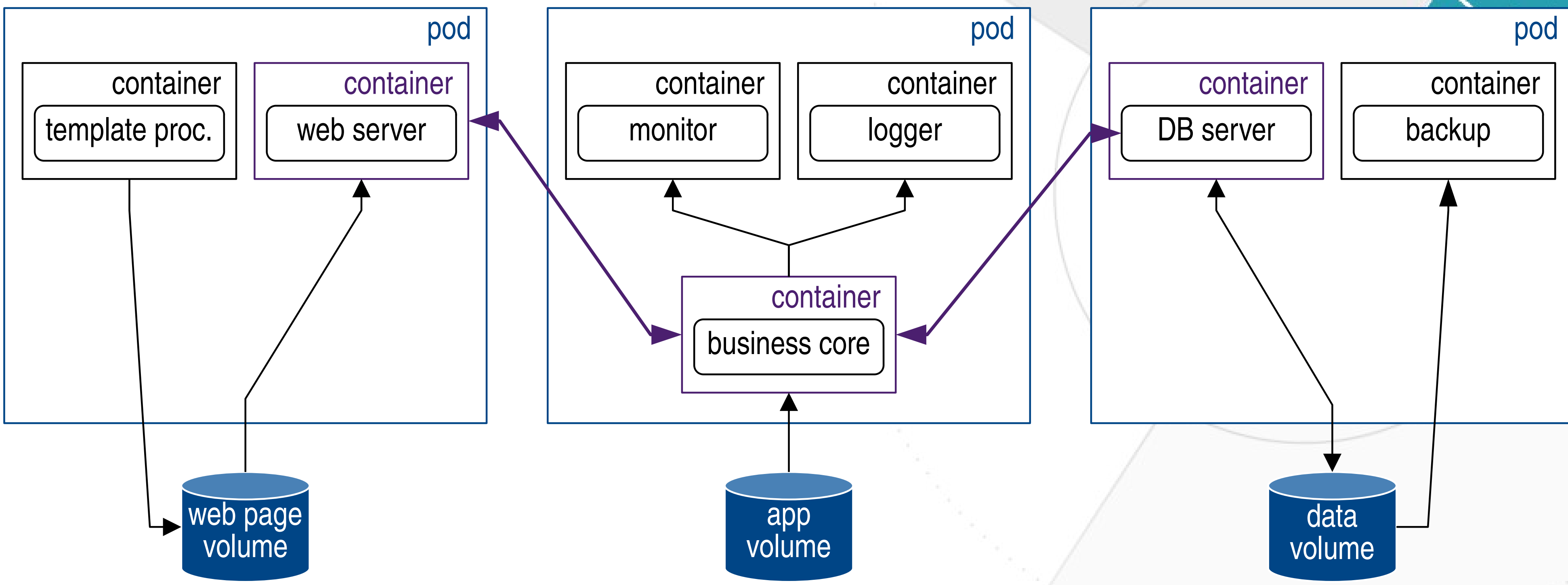
Orchestration

- Composition: build application as micro-services
  - Example: docker-compose
- Orchestration: manage micro-services
  - Distribution
  - Replication
  - Load-balancing
  - Availability
  - Higher-level interfaces to composition features
    - Acts as the user front-end
  - Examples: Kubernetes, Docker Swarm
- Abstraction of management unit: the pod

Orchestration: scheduling

- Manual criteria: filters
  - Handle host heterogeneity
    - Settings of Docker engine, host OS…
  - Container affinity: force placement for resource access
    - Image availability, volume placement, other container...
- Strategies for deployment on physical hosts
  - Spread: balance load over hosts
  - Binpack: colocate as much as possible
- Handle colocation of tightly-coupled containers: pods
  - Containers in a pod share the same network namespace and same volumes
  - Pod = service container + helper containers (logging, interfacing...)

Orchestration of pods



Application architecture with pods

Demo: Kubernetes

- Create and use a pod
- Create and use a deployment
  - Scalability
  - Roll-out

kubernetes

Kubernetes app.: deployment.yml

Scalability: set number of replicas

Pod composition (containers)

```yaml
kind: Deployment
# [ ... ]
spec:
  replicas: 3
  selector:
    matchLabels:
      app: simpleserver
  template:
    metadata:
      labels:
        app: simpleserver
    spec:
      containers:
      - name: pythonserver
        image: python:simpleserver
        resources:
          requests:
            cpu: 0.5
        ports:
          - containerPort: 8080
```

OS-level virtualization

- Virtualize the OS instead of the hardware
  - Containers: lighter, faster, simpler
- Based on Linux kernel: namespaces, cgroups
  - Container engines bring usability and networking
- Enable new cloud-native application architecture: micro-services
  - Managed with orchestrators