Virtualisation et Conteneurisation

Guillaume Urvoy-Keller - Novembre 2022
Amaury Boller - Révision Octobre 2025 AIRFRANCE

Partie 2 — De la virtualisation système à la conteneurisation moderne

UNIVERSITÉ CÔTE D'AZUR

1. Introduction

La virtualisation classique (machines virtuelles) repose sur un **hyperviseur** qui isole des systèmes d'exploitation complets.
Mais pour certains usages — développement, microservices, déploiement rapide — cette approche devient **trop coûteuse** :

- Chaque VM embarque un OS complet,
- Démarrage lent,
- Forte consommation de RAM et d'espace disque.

➡️ **La virtualisation légère**, ou **conteneurisation**, est née pour répondre à ces limites.

2. Principe de la virtualisation légère

2.1 Définition

La virtualisation légère consiste à **isoler plusieurs environnements d'exécution** (conteneurs) **au sein du même noyau Linux**.
Chaque conteneur possède :

- ses **processus**,
- son **système de fichiers**,
- sa **pile réseau**,

mais partage le **même noyau** que l'hôte.

🧩 **Différence majeure avec les VM :**

- VM → virtualisation *du matériel*
- Conteneur → virtualisation *du système d'exploitation*

2.2 Objectifs

- Démarrer des applications **instantanément**,
- Réduire la **consommation de ressources**,
- Faciliter la **portabilité** des environnements logiciels,
- Simplifier le **déploiement et la maintenance**.

3. Les fondations techniques sous Linux

3.1 Namespaces

Introduits dans le noyau Linux à partir de la version 2.4, ils permettent d'isoler ce qu'un processus peut « voir » :

Namespace	Role	Exemple
PID	Isoler les processus	Chaque conteneur a son propre PID 1
NET	Isoler la pile réseau	Interfaces virtuelles, tables de routage propres
MNT	Isoler les montages de fichiers	Chaque conteneur a son rootfs
UTS	Isoler les noms d'hôtes	Chaque conteneur a son hostname
IPC	Isoler la mémoire partagée	Pas de conflit entre processus interconteneurs
USER	Isoler les identités utilisateurs	UID/GID mappés sur l'hôte
*TIME (since Linux kernel 5.6)	Permet d'isoler l'horloge système	

3.2 Control Groups (cgroups)

Les **cgroups** (apparus en 2008) permettent de **limiter et contrôler** l'usage des ressources :

- CPU, mémoire, disque, réseau.
- Organisation hiérarchique : chaque processus appartient à un groupe.

Exemple :

Limiter un conteneur à 512 Mo de RAM et 50 % du CPU.
Les cgroups assurent que ce conteneur n'impacte pas les autres.

Depuis 2020 : adoption du **cgroups v2**, système unifié plus simple et performant.

4. Conteneurs : l’écosystème moderne

4.1 Docker : moteur de conteneurs

Lancé en 2013, **Docker** a rendu la virtualisation légère accessible :

- Simplifie la création et la distribution d’images,
- Automatise l’isolation (namespaces, cgroups),
- Offre un format standard et portable (image Docker).

Exemple :

```
$ docker run -it debian /bin/bash
root@container:/# echo "Bonjour, BUT Info !"
```

➡️ Lancement instantané d’un environnement Debian isolé.

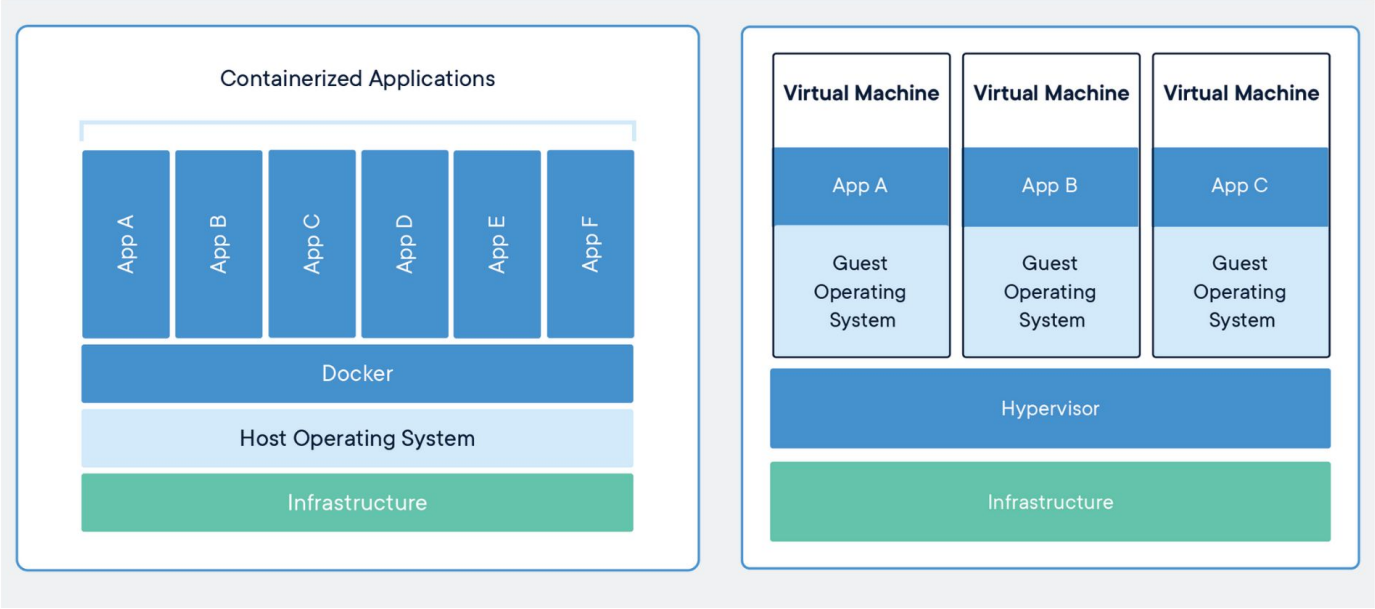
4.2 Containerd et Podman

Docker s’appuie aujourd’hui sur **containerd**, le moteur de conteneurs standardisé par la CNCF.

La Cloud Native Computing Foundation est un projet de la Linux Foundation qui a été fondé en 2015 pour aider à faire progresser la technologie des conteneurs et rassembler les industries technologiques autour de son évolution.

Podman, alternative “rootless” (sans privilèges administrateur), est de plus en plus utilisé dans les environnements sécurisés et pédagogiques.

4.3 Architecture simplifiée



Source: docker.com

5. Stockage et réseaux

5.1 Volumes

Les conteneurs sont éphémères : leurs données disparaissent à l'arrêt.
Les **volumes** permettent de rendre les données persistantes :

docker run --volume <host-path>:<container-path>

Exemple:

```
$ docker run -v $(pwd):/data myimage
```

5.2 Réseaux

Chaque conteneur dispose de sa propre interface virtuelle (veth).
Docker crée un **bridge réseau par défaut** (`docker0`) reliant les conteneurs entre eux et à l'hôte.

```
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo

2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP
   link/ether 00:0c:29:3b:72:9a brd ff:ff:ff:ff:ff:ff
   inet 192.168.1.50/24 brd 192.168.1.255 scope global ens33

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN
   link/ether 02:42:29:5a:9b:7d brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0

5: veth1c2f3a0@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
   link/ether 2a:e3:b1:41:37:2c brd ff:ff:ff:ff:ff:ff link-netnsid 0


...
```

5.2 Réseaux (suite)

Interprétation

- `docker0` : **bridge réseau virtuel créé par Docker**, commun à tous les conteneurs par défaut.
- `vethXXXX@ifYY` : paires d'interfaces **veth** :
  - une dans le **namespace réseau du conteneur**,
  - l'autre dans le **namespace de l'hôte**, attachée au bridge `docker0`.
- Chaque conteneur reçoit une IP dans le réseau **172.17.0.0/16** (natif Docker).

5.3 Réseaux: Exposition et mapping des ports

Principe

Les conteneurs Docker sont isolés :
➡️ leurs ports **ne sont pas accessibles** depuis l'extérieur par défaut.

Le mapping de ports

```
docker run -p 8080:80 nginx
            |  |   |
            |  |   └─ Port du conteneur (interne)
            |  └──── Port de l'hôte (externe)
            └──────── Option de mapping
```

5.3 Réseaux: Exposition et mapping des ports (suite)

Que se passe-t-il réellement ?

- Docker crée une **règle NAT (iptables)** :
  → le port **8080** de l'hôte est redirigé vers le port **80** du conteneur.
- Le conteneur écoute sur son interface `eth0` interne (ex : `172.17.0.2:80`).
- L'hôte devient l'interface publique :
  → `http://<IP_HOTE>:8080` atteint le conteneur.

Cas d'usage

- Serveur web : `-p 8080:80`
- Serveur SSH dans un conteneur : `-p 2222:22`
- Serveur VoIP (SIP/RTP) : multiples ports à mapper
- API interne exposée à un LAN

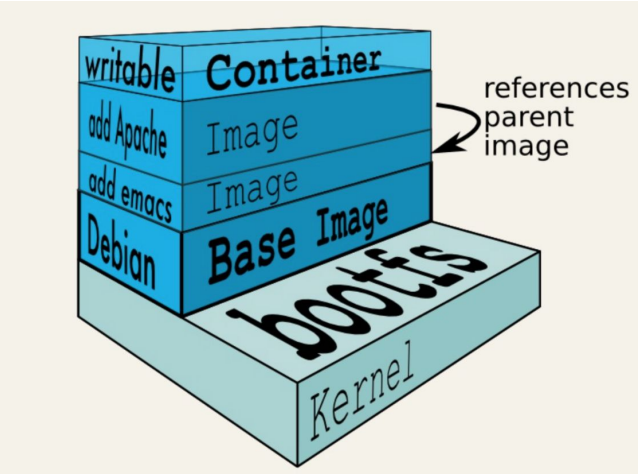
6. Images Docker

6.1 Structure et fonctionnement

Qu’est-ce qu’une image Docker ?

Une **image** est un *modèle immuable* contenant tout le nécessaire pour exécuter une application :

- système minimal (ex : Debian slim),
- binaires, librairies, dépendances,
- configuration, commande de lancement.



Caractéristiques clés

- **Immuable** : une fois construite, elle ne change pas.
- **Portable** : identique sur toute machine Docker.
- **Layered** : composée de *couches* superposées (layers).
- **Stockée** dans un **registry** : Docker Hub, GitHub Container Registry, registry local…

6. Images Docker

6.2 Création et optimisation d’une image

Le Dockerfile

Fichier texte décrivant les instructions pour construire une image.

Exemple:

```
FROM debian:12-slim
RUN apt-get update && apt-get install -y nginx
COPY index.html /var/www/html/
CMD ["nginx", "-g", "daemon off;"]
```

6. Images Docker

6.3 Les layers (couches)

Docker crée une couche *par instruction*.
Exemple :

- `FROM` → couche base
- `RUN apt-get install` → couche logiciel
- `COPY` → couche fichiers
- `CMD` → métadonnée (pas une couche)

Optimisation

- Grouper les commandes `RUN` pour réduire les layers.
- Utiliser des bases légères : `alpine`, `slim`.
- Ajouter un `.dockerignore` pour éviter les fichiers inutiles dans l'image.

Les images légères réduisent le stockage, la RAM et améliorent les temps de déploiement.

7. Orchestration

6.1 Docker Swarm

- Mode cluster intégré à Docker.
- Facile à mettre en œuvre, utile pour la pédagogie.
- Moins utilisé en production (abandonné depuis 2023).

6.2 Kubernetes

- Standard industriel.
- Gère des centaines de conteneurs sur un cluster.
- Concepts : Pods, Services, Ingress, Namespace (au sens K8s).

💡 **Lien avec le réseau :** Kubernetes s'appuie sur des **CNI (Container Network Interface)** pour gérer la connectivité (Flannel, Calico, Cilium...).

8. Kubernetes

8.1 Présentation

Kubernetes (K8s) est une plateforme open source permettant :

- Le déploiement,
- L’orchestration,
- L’auto-scaling,
- L’auto-réparation
  de conteneurs sur un cluster de machines.

8. Kubernetes

8.2 Pourquoi Kubernetes ?

- Gestion automatique des pannes,
- Mise à l’échelle transparente,
- Réseau uniforme et cohérent (CNI),
- Déploiement reproductible (manifests YAML),
- Support massif dans les clouds (AKS, EKS, GKE, OVH Managed K8s).

8. Kubernetes

8.3 Concepts clés

- **Pod** : plus petite unité déployable (un ou plusieurs conteneurs).
- **Node** : machine (VM ou physique) qui exécute les pods.
- **Cluster** : ensemble cohérent de nodes.
- **Service** : expose un pod / un groupe de pods via une IP stable.
- **CNI (Container Network Interface)** : plugin réseau (Calico, Flannel, Cilium...).

8. Kubernetes

8.4 Les objets essentiels

Pods

- Unité d’exécution : 1 ou plusieurs conteneurs partageant réseau et volumes.

Deployment

- Gère la création, mise à jour et réplication des pods.
- Exemple : s’assurer qu'il y a toujours **3 pods nginx** actifs.

Service

Permet d'exposer un ensemble de pods :

- **ClusterIP** : réseau interne Kubernetes.
- **NodePort** : port ouvert sur les nodes.
- **LoadBalancer** : intégré au cloud.

ConfigMap / Secret

- Configuration externalisée, sensible ou non.

8. Kubernetes

8.5 Kubernetes et le réseau (CNI)

- Chaque pod reçoit une **IP routable dans le cluster**.
- Communication pod↔pod via plugins CNI :
  - **Flannel** (simple, overlay VXLAN),
  - **Calico** (routage BGP),
  - **Cilium** (eBPF).
- Le composant **kube-proxy** gère la mise en place des règles NAT/conntrack.

💡 Kubernetes repose sur des mécanismes réseau avancés : overlay, routage, filtrage, eBPF, load balancing, BGP…

9. Ouverture : convergence VM / Conteneurs

Les infrastructures modernes combinent :

- VMs pour l'isolation forte et la sécurité,
- Conteneurs pour la flexibilité et la scalabilité.