

TIW - Cloud computing

Docker

Fabien RICO (fabien.rico@univ-lyon1.fr)

Jean Patrick GELAS

Univ. Claude Bernard Lyon 1

séance 2



Qu'est-ce que c'est

C'est :

- un moyen de faire fonctionner plusieurs processus dans un environnement isolé et/ou limité ;
- un moyen d'installer et de configurer des logiciels dans cet environnement ;
- un moyen de gérer différentes images et de les distribuer.

Ce n'est pas :

- de la virtualisation au sens général : moins de souplesse ;
- une simple exécution de programme : séparation du système hôte.

1 Introduction

2 Fonctionnement

- Séparation
- Contrôle des ressources
- Système de fichiers
- Résultats

3 Les commandes

- Images
- Gestions des conteneurs

4 Autour de docker

- docker compose

Virtualisation niveau système

C'est le fait, pour un système d'exploitation, de pouvoir exécuter des processus dans un environnement isolé (cf wikipedia).

Rappel

- mémoire virtuelle ;
- isolation des processus ;
- mode utilisateur/mode noyau ;
- appels systèmes.

Pour séparer ou limiter les processus, les appels systèmes peuvent :

- gérer des espaces de nom ;
- refuser certaines opérations ;
- appliquer des quotas.

Exemple historique : chroot

Définition (chroot)

chroot (*Change Root*) est une commande qui permet de changer le répertoire racine du système pour un processus et tous ses processus fils.

Normalement

```
$ ls /
```

- 1 le programme demande à ouvrir /
- 2 le programme obtient un descripteur de /
- 3 le programme liste /

Le résultat est :

```
bin boot dev etc home lib  
lib64 media ...
```

Dans un chroot

```
$ chroot /tmp/plop/ ls /
```

- 1 le programme demande à ouvrir /
- 2 le programme obtient un descripteur de /tmp/toto/
- 3 le programme liste /tmp/toto


Le résultat est :

```
bin lib lib64
```



Autres exemples

- La *machine virtuelle JAVA (JVM)* : un bytecode peut être exécuté sur des ordinateurs très différents car il est interprété par un programme isolé, la machine virtuelle java.
- Les *environnements virtuels* de python : permet d'exécuter des code python en modifiant les répertoires où se trouvent les bibliothèques de fonctions. Cela permet de développer plusieurs projet dans un environnement stables malgré l'utilisation de bibliothèques incompatibles.
- Les *sandbox* qui est un système de sécurité permettant d'exécuter du code *douteux* dans un environnement dédié et séparé du reste du système (pour le code javascript des sites web, l'isolation des applications des téléphones ...) ¹

1. See <http://pittsburgh.issa.org/Archives/Android-vs-iOS-MayUpdate> pdf 

Description d'un conteneur

Un conteneur docker peut être vu comme une extension du `chroot`. Il est composé :

- d'une description ;
- d'un ou plusieurs répertoire(s) de l'hôte qui forment son système de fichiers ;
- d'interfaces réseaux virtuelles ;
- éventuellement des processus en train de s'exécuter dans l'environnement.

S'il n'y a pas de processus, le conteneur est éteint, mais peut être relancé. S'il est en fonctionnement on peut lui attacher un nouveau processus. On peut sauvegarder un conteneur sous la forme d'une *image* qui permettra de créer d'autre conteneur.



Espace de nom

`chroot` est une commande unix qui permet de sécuriser certaines applications car le système ajoute automatiquement un préfix à tous les fichiers manipulés. Cela apporte :

- une limitation des actions ;
- un cloisonnement des applications.

On peut généraliser cela à d'autres objets du système (utilisateur, processus, ...). Sous linux, c'est ce qu'on fait via les *Espaces de noms*

Définition (Espace de nom)

Un espace de nom est un préfixe que l'on ajoute au nom des objet du système lorsqu'ils sont accédés par certains processus. Il est ainsi possible de séparer des processus qui ont un espace de nom différent.

Par exemple, si un processus est dans l'espace de nom `toto` pour les fichiers

`open("truc")` devient `open ("toto::truc")`

Ce processus ne peut accéder qu'aux objets du même espace.



Espace de nom (suite)

Sous linux, les *espaces de noms* permettent une séparation :

- `mnt` des points de montage (des disques) ;
- `pid` des numéros de processus ;
- `net` des outils de communication réseau ;
- `ipc` des outils de communication inter processus ;
- `user` des utilisateurs ;
- `uts` du nom de la machine et de son domaine ;
- `cgroups` des groupes de contrôle de processus.

Contrôle des ressources : les *cgroups*

Définition (*cgroups*)

Les *control groups* (*cgroups*) sont une fonctionnalité du noyau linux qui permet de rassembler des processus dans un groupe et de :

- limiter leur accès aux ressources ;
- comptabiliser l'utilisation de ces ressources ;
- appliquer des priorités ;
- geler, créer des points de sauvegarde ou restaurer des groupes.

Cela permet d'appliquer à un groupe de processus ce que l'ont fait déjà sur une machine virtuelle classique.

Gestion du stockage : le problème

Le système de fichiers du conteneur est séparé de l'hôte. *Il est vide !*

- Pour fonctionner, un logiciel a besoin
 - ▶ de dialoguer avec le noyau du système ;
 - ▶ d'utiliser des logiciels standards ;
 - ▶ d'utiliser des bibliothèques de fonctions.
- Seul le noyau est déjà présent dans un docker : *il faut installer le reste.*

Par exemple, un conteneur basé sur ubuntu 12.01 avec le serveur apache doit avoir dans son système de fichiers :

- les outils de base : apt, bash, vi ...
- les configurations minimales : /etc/
- les bibliothèques essentielles : ld, libc, ...
- le serveur apache.

Si un serveur contient 30 conteneurs basés sur le même modèle, combien de fois faut-il chaque outil ?



Gestion du stockage : *Copy on Write*

On peut « *empiler* » les images de conteneurs.

Définition (COW)

Le *Copy-on-Write* est la capacité de maintenir 2 copies d'un ensemble de données

- en gardant une seule copie de ce qui est commun ;
- en dupliquant uniquement ce qui est modifié.

Grâce au COW on peut :

- mutualiser les systèmes de fichiers basés sur le même modèle ;
- créer rapidement des conteneurs qui spécialisent un conteneur existant.

Que permet docker ?

- Créer des images avec un mini système d'exploitation
⇒ un disque de machine virtuel ?
- Créer un conteneur basé sur ce disque auquel on attribue des ressources
⇒ une instance de VM ?
- Allumer, éteindre, sauvegarder le conteneur
⇒ même opération que sur les VMs ?

Le comportement est très proche de celui des hyperviseurs. Mais c'est de la virtualisation au niveau du système, c'est à dire que ce n'est que de la gestion de processus différents dans un seul système d'exploitation.



Différence avec la virtualisation

Comme ce ne sont que des processus dans un seul système :

- le partage de ressources est facilité ;
- il y a moins de surcout que la virtualisation ;
- il n'y pas de gestion du matériel ;
- il y a moins de sécurité.

A quoi cela sert ?

Les éléments

- images : le *template* (modèle), les données de l'application prêtes à l'emploi ;
- conteneur : instance qui fonctionne ou qui peut être démarrée ;
- volumes : répertoire qu'on peut partager entre conteneurs ou avec l'hôte ;
- *registry* ou dépôt : service permettant de déposer ou télécharger des images.



Hub et images

<https://hub.docker.com/> contient les images proposées par les autres :

- beaucoup de choix ;
- un système de notation (les étoiles) ;
- possibilité de télécharger des versions (les *tags*).

Quelques commandes :

- `docker search mot_clef` chercher une image ;
- `docker pull nom:TAGdeVersion` télécharger une image ;
- `docker images` lister les images présentes localement ;
- `docker rmi` effacer une image locale.

Récupération, dépôt d'image

Les images sont sur un serveur *registry* par défaut le *Docker store* : <https://store.docker.com/> On peut télécharger ou déposer des images sur ces serveurs avec les commandes `pull` et `push`

```
docker pull [registry/]repository[:tag]
```

- Le *registry* est le serveur qui est contacté (par défaut `registry.docker.io :443`).
- Le *repository* est la famille d'images que vous voulez obtenir (par exemple `debian`, `ubuntu`, `nginx`,...).
- Le *tag* est la version (par défaut *latest*).

Par exemple

```
$ docker pull debian:8.5
```

permet d'obtenir un docker basé sur la version 8.5 de la débien.

```
$ docker pull debian
```

permet d'obtenir un docker basé sur la dernière version de la débien.



Construction d'image

On peut recréer une image à partir d'un docker

```
docker commit NomDuConteneur NomImage:Tag
```

Cette image peut ensuite être

- utilisée pour relancer un conteneur

```
docker run -d --name NomNouveau NomImage:Tag command
```

- sauvegardée sous la forme d'une archive

```
docker save -o fichier.tar NomImage:Tag
```

- envoyée sur un serveur

```
docker push NomImage:Tag
```

Dockerfile

On peut construire une nouvelle image à partir d'un fichier de description appelé Dockerfile.

```
docker build -t nom:tag RepertoireDuDockerfile
```

A partir du fichier de description, la commande :

- télécharge une image de base ;
- applique une série de commandes qui modifient l'image ;
- crée une image docker utilisable
 - ▶ avec une description de l'environnement
 - ▶ avec une commande à exécuter au lancement

La construction via Dockerfile utilise le COW pour construire les différentes images générées.

Exemple de Dockerfile

```
FROM ubuntu:vivid  
MAINTAINER Fabien Rico <fabien.rico@univ-lyon1.fr>
```

image de base

```
RUN apt-get update \  
    apt-get -y install apache2 && apt-get clean
```

installation d'apache

```
COPY serveur.conf /etc/apache2/sites-enabled/  
    /etc/apache2/sites-enabled/000-default.conf
```

ajout d'un fichier

```
WORKDIR /var/www/html
```

répertoire de travail

```
ENV APACHE_RUN_USER www-data  
ENV APACHE_RUN_GROUP www-data  
ENV APACHE_LOG_DIR /var/log/apache2
```

variables d'environnement

```
EXPOSE 80
```

port exposé

```
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

commande à exécuter



Création du conteneur

C'est au moment de la création que l'on peut donner la configuration du docker

```
docker create --name nom [options] NomImage:tag [commande]
```

- `-p HostPort:DockePort` : mapping de port entre l'hôte et le conteneur
- `--link NomAutreDocker` : ajoute un lien avec un autre docker (c'est à dire fixe des variables d'environnement permettant de contacter le docker
- `-v VolumeHôte:VolumeCont` crée un volume partagé entre l'hôte et le conteneur.
- `--name nom` nom du conteneur
- `NomImage` le nom de l'image à exécuter
- `commande` la commande à exécuter dans le docker (par défaut celle du Dockerfile)



Gestion du conteneur

- Lancer un conteneur docker `start nomDuConteneur`
le conteneur s'exécute en lançant la commande prévue dans le *create* ou définie dans le Dockerfile.
- stopper un contenu docker `stop nom`
- Lister les conteneurs exécutés `docker ps`
- Lister les conteneurs existant `docker ps -a`
- exécuter une commande dans un docker existant
`docker exec -it nomDuConteneur commande`
- Supprimer un conteneur docker `rm nom`
- Créer et lancer un conteneur
`docker run -d ...` avec les mêmes options que `docker create`.



Gestion du réseau des dockers

- On peut créer des réseaux spécifique aux docker ce qui permet de les isoler des autres :

```
docker network create [--subnet 192.168.23.0/24]  
nom_du_réseau
```

- Dans ce cas, on peut alors créer le docker de manière à ce qu'il soit dès le départ dans ce réseau :

```
docker create ... --net=nom_du_réseau [--ip  
192.168.23.45] ...
```

- Ou alors ajouter à un docker existant une interface dans le réseau :

```
docker network connect [--ip 192.168.23.46]  
nom_du_réseau nom_du_docker
```

Il est possible de créer des réseaux partagés entre les dockers de plusieurs machines hôtes, ce qui permet de gérer de l'équilibrage de charge par exemple. Mais cela sera vu plus tard avec l'orchestration des dockers.



Configuration des dockers

Il doit être possible de relier les docker entre eux, de donner certaines configuration comme les mots de passes.

- L'ancienne méthode (*link*) est dépréciée.
- On utilise :
 - ▶ les adresses fixées par l'utilisateur ;
 - ▶ la possibilité d'ajouter des adresses dans le fichier `/etc/hosts` grâce à l'option `--add-host` à la création ;
 - ▶ la possibilité de donner des variables d'environnement grâce à l'option `--env` ou `-e`.



Difficultés d'utilisation

- Le docker est exécuté dans un environnement séparé :
 - ▶ Les fichiers de configurations ne se trouvent pas au même endroit pour le docker et pour son hôte.
 - ▶ Les logs d'erreur peuvent être difficiles à voir (utilisez `docker logs nom_du_docker`) ou des utilitaire comme *SysDig*.
 - ▶ Ce n'est pas parce que l'hôte peut contacter un serveur que le docker le peut.

Autour de docker

Il y a plusieurs utilitaires qui se sont ajouté à docker et commencent à interagir :

- *Swarm* pour créer des clusters de machines utilisant docker.
- *docker-compose* pour organiser simplement la construction et le lancement de plusieurs conteneur qui dépendent les un des autres.
- *docker-machine* pour automatiser la création de machines virtuelles contenant le logiciel docker.

Swarm fera l'objet du prochain cours.

docker-compose

Docker permet d'implémenter des micro-services, cela signifie que les conteneurs se limite souvent à une tâche simple :

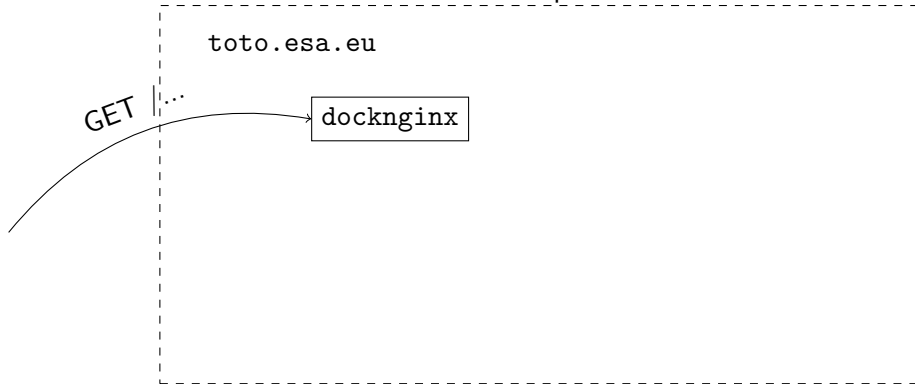
```
toto.esa.eu
```

Ces dockers doivent être lancé dans un certain ordre et avec des liaisons déterminées entre eux.



docker-compose

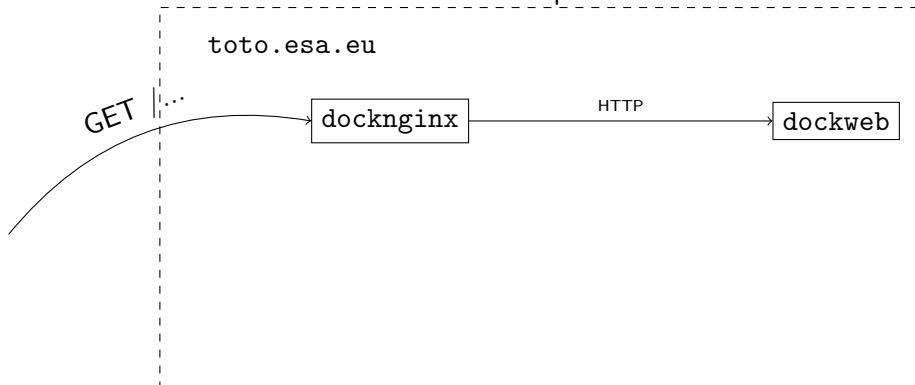
Docker permet d'implémenter des micro-services, cela signifie que les conteneurs se limite souvent à une tâche simple :



Ces dockers doivent être lancé dans un certain ordre et avec des liaisons déterminées entre eux.

docker-compose

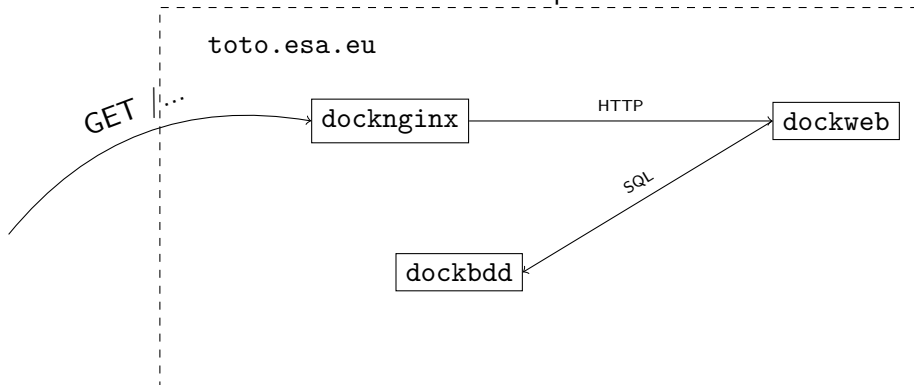
Docker permet d'implémenter des micro-services, cela signifie que les conteneurs se limite souvent à une tâche simple :



Ces dockers doivent être lancé dans un certain ordre et avec des liaisons déterminées entre eux.

docker-compose

Docker permet d'implémenter des micro-services, cela signifie que les conteneurs se limite souvent à une tâche simple :



Ces dockers doivent être lancé dans un certain ordre et avec des liaisons déterminées entre eux.

Ce que fait docker-compose

Docker compose :

- lance pour vous la construction de l'image (`docker build`);
- crée pour vous les docker (`docker run`) :
 - ▶ en tenant compte des contraintes,
 - ▶ en gérant les nom et adresses interne des docker ;
- gère la mise à jour des dockers :
 - ▶ si vous modifier le `Dockerfile` de l'un deux, l'image est recrée,
 - ▶ si vous modifier les configuration de création, il est reconstruit,
 - ▶ s'il y a des dépendances, elles sont répercutée.

docker-compose.yml

Tout est décrit dans un ensemble de répertoires et dans le fichier de description docker-compose.yml.

```
dockerfront:
  image: nginx
  ports:
    - "80:80"
    - "443:443"
  links:
    - "dockweb:engine"
  volumes:
    - "./front/default.conf: \
      /etc/nginx/conf.d/default.conf:ro"
```

description du docker dockerfront

basé sur l'image nginx

qui reçoit les port web de la machine

mise en place de lien entre les dockers

engine correspondra à l'adresse du
docker dockweb

partage de volumes (ici la configuration de nginx)

```
dockweb:
  build: ./dockweb/
  links:
    - "db:db"
  ...
```

description du docker dockweb

Il est construit à partir de ce qui se trouve
dans le répertoire ./dockweb/

lien avec le docker de base de données

```
db:
  image: mysql:5.7
  ...
  environment:
    - "MYSQL_ROOT_PASSWORD=totoplop"
    - "MYSQL_USER=truc"
    - "MYSQL_PASSWORD=motdepasse"
    - "MYSQL_DATABASE=truc"
```

description du docker de la base de donnée

construit à partir de l'image mysql dans la version 5.7

variables fixées à la création du docker

pour cette image, ces variables

permettent de choisir le mot de passe et créer
un utilisateur et une base de données.



Les commandes

- **docker**—compose build construit les dockers décrits dans le fichier `docker-compose.yml` ;
- **docker**—compose create crée les conteneurs ;
- **docker**—compose up fait l'ensemble des opérations ;
- **docker**—compose down détruit l'ensemble des conteneur et leur volumes.

Ces commandes gèrent la mise à jour, si les conteneurs tournent et qu'un seul des contenueurs est modifié, c'est ce dernier qui est recrée et/ou relancé.



docker-machine

docker machine est un script qui crée pour vous des machines virtuelles capables d'utiliser docker.

- Il dialogue avec les système de cloud computing (openstack, amazone2c, azure ...) pour créer les machines virtuelles.
- Il met en place les moyens de vous y connecter.
- Il installe docker et swarm sur ces machines.

Il ne vous reste alors plus qu'à configurer et lancer vos services.



Conclusion

Pourquoi utiliser docker ?

- parce que cela permet de travailler confortablement
 - ▶ sans changer de machine ;
 - ▶ en maîtrisant les librairie, version de système, dépendances...
- parce que cela permet de distribuer votre travail simplement mais efficacement.
- parce que cela permet d'utiliser le travail des autres.

Objectifs à venir :

- être capable de manipuler les conteneurs, les ensembles de conteneur ;
- être capable de monter un cluster ;
- être capable de mettre en place un cluster hadoop sans rien comprendre à son fonctionnement.

