TIW - Cloud computing

Docker

Fabien Rico (fabien.rico@univ-lyon1.fr)
Jean Patrick Gelas

séance 2

Table des matières

1 Introduction

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.

Un peu comme la machine virtuelle java, cela signifie qu'on exécute des programmes dans un environnement maitrisé, donc sans avoir à ce soucier de l'intégration de ces programmes au système. Cette propriété est une partie de ce qui a fait le succès du java. En effet, cela permet aux entreprises de diffuser des applications très complexes sans problème de compatibilité avec les différents systèmes d'exploitations.

Ici, ce qu'on peut faire va plus loin. Non seulement les programmes sont exécutés dans un environnement séparé, mais cet environnement est un petit système et les interactions avec le système (fichier, réseau, autres commandes ...) restent dans le conteneur. De plus, on peut créer une image de cette environnement et l'exécuter sur un autre système sans avoir à le modifier.

C'est presque aussi souple que la virtualisation. Cependant, on n'utilise pas une machine virtuelle propre sur laquelle on installe un nouveau système. Il faut conserver le noyau linux, le conteneur doit donc rester sous linux.

— le système de fichiers est séparé, on peut (re)installer une bibliothèque précise compatible avec le logiciel ;

— les ressources réseaux sont dédiées, on peut choisir quel conteneur dialogue avec l'extérieur ou avec un autre conteneur ;
— les processus sont isolés, il doivent passer par le réseau ou des sockets système pour dialoguer (ce qui assure une certaine sécurité).

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 système peuvent :
— gérer des espaces de nom ;
— refuser certaines opérations ;
— appliquer des quotas.

En tant normal, les programmes sont exécutés dans un environnement *protégé*. La plupart du temps ils travaillent en mode utilisateur avec une mémoire virtuelle séparée des autres et un jeu de commande réduit. Ils ne peuvent accéder aux pilotes de périphériques, à certaines zones mémoire, ...

Pour communiquer ou interagir avec le système, ces processus doivent utiliser des *appels systèmes*, c'est à dire des fonctions proposées par le système. Ces appels système peuvent très bien ne pas dire la vérité ou appliquer des règles de séparation.

Exemple historique : chroot

Définition 1 (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

Pour que cela fonctionne, il faut que l'environnement *chrooté* ait accès aux commandes qu'il va utiliser (bash et ls) ainsi qu'à leurs librairies.

```
# création de l'environnement (sous ubuntu 15.04)
mkdir -p /tmp/toto/bin/
mkdir -p /tmp/toto/lib/x86_64-linux-gnu/
mkdir -p /tmp/toto/lib64/
# copie des programmes à utiliser
cp /bin/bash /tmp/toto/bin/
```

```
cp /bin/ls /tmp/toto/bin/
# copie de l'éditeur de lien
cp /lib64/ld-linux-x86-64.so.2 /tmp/toto/lib64/
# copie des librairies utiles (connues grâce à la commande ldd /bin/bash)
cp /lib/x86_64-linux-gnu/libtinfo.so.5    /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libdl.so.2       /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libc.so.6        /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libselinux.so.1  /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libacl.so.1      /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libpcre.so.3     /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libattr.so.1     /tmp/toto/lib/x86_64-linux-gnu/
```

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 réseau 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.

2 Fonctionnement

2.1 Séparation

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, ...). C'est ce qu'on fait via les *Espace de noms*

Définition 2 (Espace de nom)**.** Un espace de nom est un préfixe que l'on ajoute aux noms de certains objets du système. Il permet de séparer des processus qui ont un espace de nom différent.

Si un processus demande à accéder à un objet, l'espace de nom est automatiquement ajouté à la requête. Par exemple, si le processus est dans l'espace de nom `toto` `open("truc")` devient `open ("toto::truc")`

Il ne peux accéder qu'aux objets du même espace.

Ce type de séparation est très utile pour la sécurité. C'est l'utilisation proncipale de `chroot`. Même si il est corrompu :

- Un programme *chrooté* n'accèdent qu'aux commandes qui ont été copiées dans l'environnement. On peut donc limiter ces commandes à celles qui sont nécessaires.
- Un programme *chrooté* ne modifie que des fichiers de l'environnement. Il y a beaucoup moins de risques d'obtenir des informations sur un autre composant du système, ou d'utiliser une faille pour acquérir de nouveaux droits.

Les espaces de noms sont utilisés en programmation pour séparer les noms des objets. Par exemple en C++ :

- Les vecteurs de la classe `std`, std::vector sont des tableaux dynamiques.
- Les vecteurs de la classe `ublas`, boost::numeric::ublas::vector sont des vecteur mathématiques.

Ici, ils sont utilisés pour séparer les processus. Par exemple, 2 processus qui ne sont pas dans le même espace de noms utilisateur ont des utilisateurs différents. Il peut donc y avoir un administrateur dans le premier processus qui est différent du second.

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.

Grâce à cette isolation, les conteneurs vont pouvoir fonctionner comme de petites machines virtuelles indépendantes avec leur propre adresse, leur propre système de fichier, leur carte réseau ...

Mais, ce n'est pas la seule fonctionnalité utilisée par docker. En effet, il faut être capable de controler l'utilisation des ressources.

2.2 Contrôle des ressources

Contrôle des ressources : les *cgroups*

Définition 3 (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.

Docker utilise les cgroups pour appliquer une gestion fine des ressources matérielles. Cela permet aussi bien de limiter la bande passante réseau que d'interdire l'utilisation d'un processeur à l'un des conteneurs.

Ces fonctionnalités vont ajouter à docker un fonctionnement proche des machines virtuelles à qui on attribue à la création une partie de la mémoire, un nombre de CPU... Elles vont aussi permettre de comptabiliser ces ressources pour une éventuelle facturation.

2.3 Système de fichiers

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 standard ;
  - 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, combient de fois faut-il chaque outil ?

Gestion du stockage : *Copy on Write*

On peut « *empiler* » les conteneurs.

Définition 4 (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.

Dans l'exemple précédant, la base des conteneurs ne serait présente qu'une seule fois dans le système de fichiers de l'hôte. Ce qui serait dupliqué sont : les configurations, les fichiers des différents site web et éventuellement les outils installés dans un conteneur particulier par son utilisateur. Le coût supplémentaure est beaucoup plus faible que 30 copie du système.

Le COW est une propriété qui est disponible dans beaucoup de domaines (mémoire, base de données, ...). Il y a même plusieurs systèmes de fichiers qui intègrent directement le copy on write. Cela peut donc être fait de manière efficace. Attention, le COW ne permet pas totalement de mutualiser les fichiers communs. En effet, 2 fichiers modifiés en parallèle de la même manière sont considérés comme différents.

2.4 Résultats

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.

Dans un système, 2 processus peuvent accéder au même fichier, partager une zone mémoire (tube,IPC,...). Cela reste possible pour 2 conteneurs différents sans perte d'efficacité. On peut avoir des comportements similaires entre des machines virtuelles ou entre une machine et son hôte, mais cela demande d'adapter le système de la machine virtuelle en reprogrammant certains pilotes de périphérique. Par exemple, pour accélérer le partage de fichiers, on crée un type de fichiers spécial qui permet à la machine virtuelle d'écrire directement dans le disque de l'hôte. C'est le rôle des extensions fournie par les hyperviseurs (`vmware-tools`...)

Le coût d'un conteneur est très faible car il n'y a pratiquement pas de différence entre lancer 2 fois un procesus et lancer 2 fois ce processus dans 2 conteneurs différents. Dans le cas des machines virtuelles, ce cout est plus important. Pour avoir 2 VMs contenant 2 serveurs de base de données, chaque machine a besoin dans sa mémoire d'une copie du système d'exploitation et de toutes les bibliothèques qu'elle utilise.

De même, les processus d'un conteneur accèdent au matériel comme les autres. Il n'y a pas de notion de cartes virtuelles avec un driver spécialisé.

Mais, le niveau de sécurité est moindre par rapport à une virtualisation complète. Par exemple, s'il n'y a pas de limitation de l'utilisation de la mémoire, un docker peut remplir la mémoire de son hôte et donc perturber le fonctionnement des autres.

De même si on n'utilise pas la séparation des utilisateurs, l'administrateur du conteneur est aussi l'administrateur de l'hôte. Si un utilisateur standard peut créer un docker et l'exécuter en choisissant les partages, il peut lancer un docker partageant le répertoire /etc/ et, puisqu'iil est administrateur du docker, il peut modifier les configurations de la machine hôte.

A quoi cela sert ?

3 Les commandes

Les éléments

- images : le template, 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 conteneur ou avec l'hôte ;
- dépot : le docker hub ou tout le monde peut déposer une image de docker à télécharger et tester.

3.1 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 présentent localement ;
- `docker rmi` effacer une image locale.

Récupération, dépot d'image

Les images sont sur un serveur *registry* par défaut le *Docker Hub*. 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 hub.docker.com :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 débian.

$ docker pull debian permet d'obtenir un docker basé sur la dernière version de la débian.

Construction d'image

On peut recréer une image à partir d'un docker `docker commit` *`NomDuConteneur NomImage:Tag`*

Cette image peut ensuite être

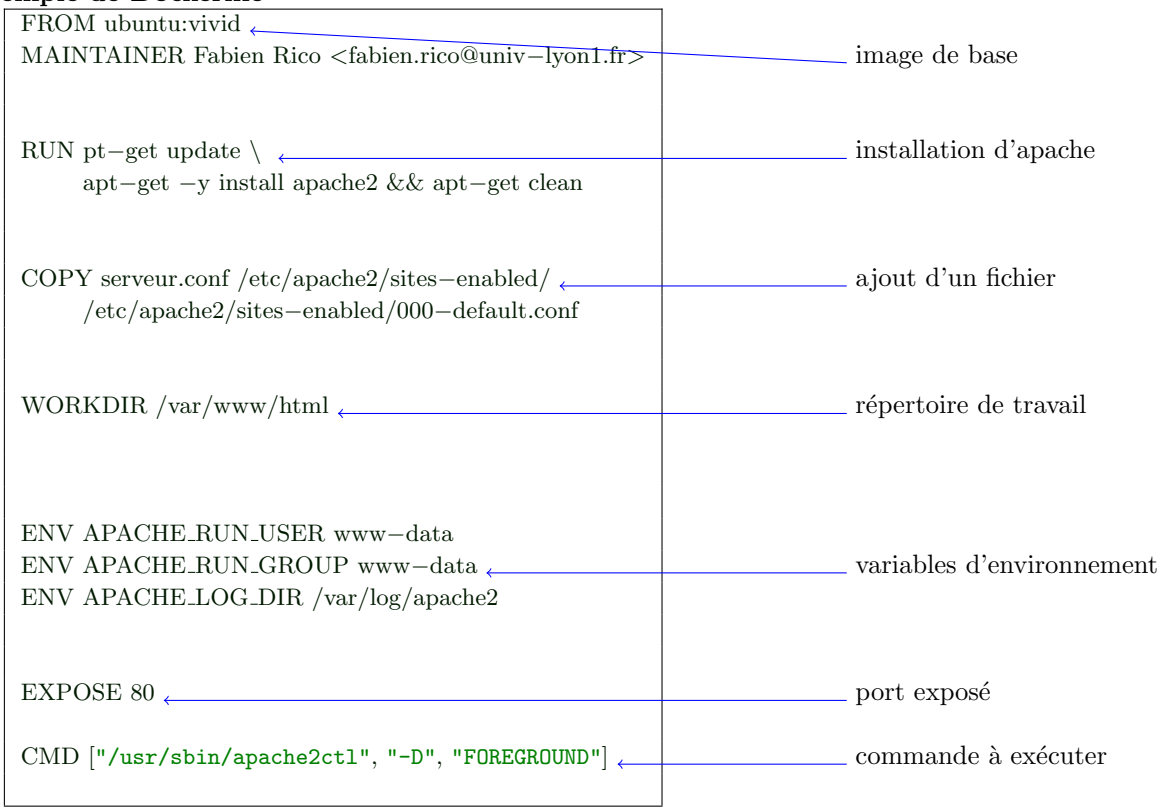
- utilisée pour relancer un conteneur
  `docker run --name` *`NomNouveau NomImage:Tag command`*
- sauvegardée sous la forme d'une archive
  `docker save -o` *`fichier.tar NomImage:Tag`*

— 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 modifie 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



3.2 Gestions des conteneurs

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 [commande]`
— `-p HostPort:DockerPort` : mapping de port entre l'hote 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 VolumeHote: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 ... avec les même options que `docker create`.