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 se 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 a permis aux entreprises de diffuser des applications très complexes sans problèmes 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 lui même. Cela signifie que

les interactions avec le système (fichiers, réseau, liaison avec des serveurs de données, ...) restent dans le conteneur ou sont transmis à un autre conteneur. De plus, on peut créer une image de cet environnement et l'exécuter ailleur sans avoir à le modifier.

C'est presque aussi souple que la virtualisation. Cependant, contrairement à la virtualisation, on n'utilise pas une pseudo machine propre sur laquelle on installe le système de son choix. Ce sont des processus, mais ils sont isolés. Il faut donc utiliser le noyau linux. À cause de cela, le conteneur doit donc rester sous linux.

- le système de fichiers est séparé, on peut (re)installer une bibliothèque précise compatible avec le logiciel de notre choix ;
- les ressources réseaux sont dédiées, on peut choisir si le conteneur dialogue avec l'extérieur, avec un autre conteneur ou créer un réseau propre aux différents conteneurs ;
- les processus sont isolés, il doivent passer par le réseau ou des sockets systèmes 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 systèmes peuvent :

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

En tant normal, les programmes sont exécutés dans un environnement *protégé. I.E.* la plupart du temps ils travaillent en mode utilisateur avec une mémoire virtuelle séparée des autres et un jeu de commandes réduit. Ils ne peuvent accéder à tout : pilotes de périphériques, 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 systèmes 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)
# si vous n'avez pas cette distribution mais avez installé docker
# utilisez docker !
# docker run --rm -it ubuntu:16.04 bash
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/libselinux.so.1 /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/libpcre.so.3 /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/libpthread* /tmp/toto/lib/x86_64-linux-gnu/
cp /lib/x86_64-linux-gnu/libtinfo.so.5 /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/libattr.so.1 /tmp/toto/lib/x86_64-linux-gnu/
# exécuter un processus en changeant le répertoire /
chroot /tmp/toto/ bash
```

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 : permettent d'exécuter des codes python en modifiants les répertoires où se trouvent les bibliothèques de fonctions. Cela permet de développer plusieurs projet dans un environnement stables malgrès l'utilisation de bibliothèques incompatibles.

— Les *sandbox* qui sont 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]

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.

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

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, ...). Sous linux, c'est ce qu'on fait via les *Espaces de noms*

Définition 2 (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.

Ce type de séparation est très utile pour la sécurité. C'est l'utilisation principale de `chroot`. En effet, même si il est corrompu :

— Un programme *chrooté* n'accède qu'aux commandes qui ont été copiées dans l'environnement. On peut limiter ces commandes à celles qui sont nécessaires au fonctionnement du logiciel sans rien de plus.

— 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 aussi 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 vecteurs mathématiques.

Ici, ils sont utilisés pour séparer les processus. Par exemple, 2 processus qui ne sont pas dans le même espace de nom utilisateurs ont des utilisateurs différents. Il peut donc y avoir un administrateur dans le premier processus qui est différent du second. Cette technique est utilisée notamment par défaut dans docker pour que l'administrateur du docker ne soit pas administrateur de l'hôte.

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 fichiers, leur carte réseau ...

Mais, ce n'est pas la seule fonctionnalité utilisée par docker. En effet, en plus d'isoler les processus entre eux, il faut être capable de contrôler l'utilisation des ressources : mémoire, processeur, disque, ...

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 de celui des machines virtuelles à qui on attribue via, le gestionnaire d'hypervision, une partie de la mémoire, un nombre de CPU... Elles vont aussi permettre de comptabiliser ces ressources pour une éventuelle facturation.

2.3 Sécurité

Sécurité

L'isolation permet une certaine sécurité, mais pour des raison pratique, cette isolation peut être rompue.

- Un utilisateur à le droit de créer des docker dans lequel il est administrateur.
- Un docker peut partager n'importe quel répertoire de la machine hôte, par exemple `/etc/`.
- Donc un utilisateur peut lancer un docker qui partage le répertoire `/etc/`, comme il esst `root` dans le docker, il peut lire le contenu de `/etc/shadow` !
- **Attention, seul les utilisateur de confiance doivent pouvoir créer et configurer les docker**.

Mais tout n'est pas possible pour les docker grâce au *CAPABILITIES* de linux.

Sécurité POSIX

Le système de sécurité unix de base est d'autoriser ou non une action en fonction du propriétaire courant d'un processus ou de son groupe.

- Un utilisateur normal à des droits plus ou moins limité.
- `root` a tout les droits.
- Un utilisateur peut lancer des commandes avec l'identité (et donc les droits) d'un autre si le fichier exécutable à un drapeau spécial (`setuid bit`).

Mais donc l'utilisateur `root` du docker a tout les droits !

Sécurité les *CAPABILITIES*

Les *CAPABILITIES* ou capacités sont un moyen d'associer à un processus ou un fichier des permis d'utiliser certaines fonctionnalités :

- changer d'identité
- ouvrir des socket en écoute sur les ports réservés
- ...

Docker utilise ces *capabilities* pour limiter les droits des processus internes aux conteneurs. Par exemple par défaut :

— Ouvrir des ports réservés (80, 443 ...) ou changer l'identité d'un processus est autorisé.
— Ajouter ou retirer un module de noyau est interdit.

Le choix des capacités autorisées doit permettre au docker de fonctionner en réduisant au maximum les risque d'attaque si le docker est compromis. Il est possible d'ajouter ou de retirer des capacité à la création du docker.

Attention le droit de passer outre les permissions de lecture est autorisé par défaut. Attention au utilisateur qui peuvent lancer des docker.

Voir un article de *Linux Magazine* sur le sujet et la documentation sur la sécurité des dockers et sur la création des dockers.

2.4 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 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, combient de fois faut-il chaque outil ?

Gestion du stockage : *Copy on Write*

On peut « *empiler* » les images de 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. Ne seraient dupliqués que : les configurations, les fichiers des différents site web et éventuellement les outils installés dans un conteneur particulier par son utilisateur. Le coût supplémentaire 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 (btrfs). 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 sont considérés comme différents, même si on a fait la même modification. Être capable de retrouver que 2 fichiers sont identiques car ils ont été modifiés de la même manière est un autre problème plus complexe : la *déduplication*.

2.5 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 aux mêmes fichiers, partager une zone mémoire (tube,IPC,...). Cela reste possible pour 2 conteneurs différents sans perte d'efficacité. Bien sur, on peut avoir des comportements similaires entre des machines virtuelles ou entre une machine virtuelle et son hôte. Mais cela demande d'adapter le système de la machine virtuelle en reprogrammant certains pilotes de périphériques. 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 ce processus dans 2 conteneurs différents. Dans le cas des machines virtuelles, ce coût 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 processus.

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'il est administrateur du docker, il peut modifier les configurations de la machine hôte.

A quoi cela sert-il ?

— Isolation de serveurs pour des raisons de sécurité ou de robustesse.

— Séparation des applicatifs, des données, des configurations, ...

— Utilisation de différentes versions de bibliothèques (comme java ?).

— Simplification du déploiement d'applications.

— Environnent de développement.

3 Les commandes

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.

3.1 Images

Hub et images

Les images sont disponibles dans des dépots.

Par défaut `https://hub.docker.com/` ou `https://store.docker.com/` contiennent les images publiques 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.

Utilisation des images publiques

Il y a beaucoup d'images proposées.

— Le site web ou le dépot git d'une image sont une bonne indication de la qualité d'une image publique.
  — présence de documentation ;
  — options possibles et expliquées ;
  — Par exemple l'image docker de Mysql par oracle et celle de mysql par docker
— Il y a débat sur le fait d'utiliser ou non des images publiques.
  — problème de maintient ;
  — problème de sécurité ;
  — problème de compétence ;
  — problème de temps de développement.

Récupération, dépot d'image

On peut télécharger ou déposer des images sur les 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*).

Le *Docker store* est un dépot (*registry*) par défaut. On peut utiliser d'autre serveur (par exemple un registry privé).

Par exemple `$ docker pull debian:8.5` permet d'obtenir un docker basé sur la version 8.5 de la débian.

`$ docker pull localhost:5000/debian` permet d'obtenir un docker basé sur la dernière version de l'image débian disponible sur le dépot local en écoute sur le port 5000.

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                                          ← image de base
MAINTAINER Fabien Rico <fabien.rico@univ-lyon1.fr>

RUN apt-get update \                                       ← installation d'apache
    apt-get -y install apache2 && apt-get clean

COPY serveur.conf /etc/apache2/sites-enabled/              ← ajout d'un fichier
    /etc/apache2/sites-enabled/000-default.conf

WORKDIR /var/www/html                                      ← répertoire de travail

ENV APACHE_RUN_USER www-data                               ← variables d'environnement
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2

EXPOSE 80                                                  ← port exposé

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]           ← commande à exécuter
```

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:tag* `[`*commande*`]`

- `-p` *HostPort*`:`*DockerPort* : mapping de port entre l'hote et le conteneur
- `--link` *NomAutreDocker* : **deprécié** 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.
- `--network` choisir le réseau du docker
- `--network-alias` choisir un nom DNS dans le réseau des dockers
- `--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 dockers 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 ... --network=`*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 dockers 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` ;
  - le possibilité de donner des nom DNS précis dans le réseau des dockers

Par exemple, considérons 2 dockers :

— un docker *wordpress* qui a besoin d'une base de donnée ;

— un docker *mysql* qui servira de base de donnée au premier.

Il faut être capable simplement, à la création du docker *mysql* de choisir un nom d'utilisateur et un mot de passe pour gérer la base et, à la création du docker *wordpress*, de donner l'adresse de la base de donnée, l'utilisateur et le mot de passe nécessaire à son fonctionnement. Pour cela, on peut :

— pour le docker *mysql* fixer son nom DNS `--network-alias baseWP` et le mot de passe de l'administrateur par une variable d'environnement `-e MYSQL_PASSWD=toto` ;

— pour le docker *wordpress* ajouter le nom du serveur de base de donnée et le mot de passe vie des variables d'environnement `-e WP_MYSQL_PASSWD=toto`, `-e WP_MYSQL_HOST=baseWP`.

Un docker bien conçut doit être configurable en grande partie via ce type de méthode. Si vous utiliser des docker recommandé, il faut regarder sur la description du docker hub ce qui est utilisé.

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 utilitaires comme *SysDig*.

  — Ce n'est pas parce que l'hôte peut contacter un serveur que le docker le peut.

4 Autour de docker

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 conteneurs qui dépendent les un des autres.

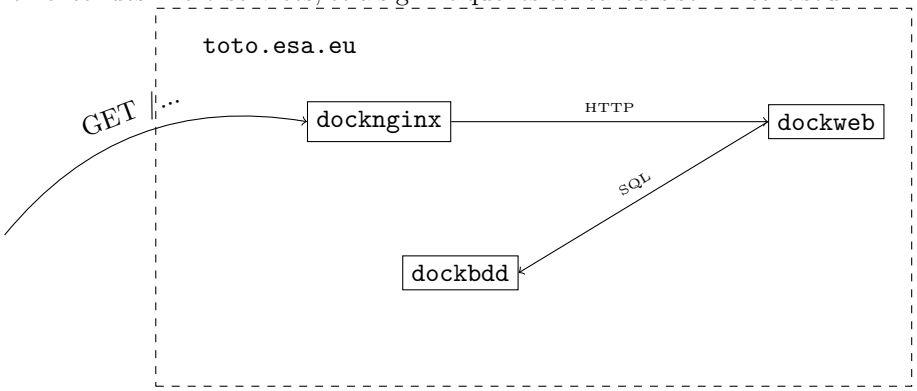
— *docker-machine* pour automatiser la création de machines virtuelles contenant le logiciel docker.

Swarm fera l'objet du prochain cours.

4.1 docker compose

`docker-compose`

Docker permet d'implémenter des micro-services, cela signifie que les conteneurs se limitent sou-



vent à une tâche simple :

Ces dockers doivent être lancés 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 dockers (`docker run`) :
  - en tenant compte des contraintes,
  - en gérant les noms et adresses internes des dockers ;
- gère la mise à jour des dockers :
  - si vous modifiez le `Dockerfile` de l'un deux, l'image est recrée ;
  - si vous modifiez les configurations de création, il est reconstruit ;
  - s'il y a des dépendances, elles sont répercutées.

`docker-compose.yml`

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

```
dockfront:
      image: nginx
      ports:
            -"80:80"
            -"443:443"
      links:
            -"dockweb:engine"

      volumes:
            -"./front/default.conf: \\
                  /etc/nginx/conf.d/default.conf:ro"


dockweb:
      build: ./dockweb/

      links:
            -"db:db"
      ...


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

description du docker `dockfront`
basé sur l'image `nginx`

qui reçoit les ports 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)

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

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 contenuer est modifié, c'est ce dernier qui est recrée et/ou relancé.

4.2 docker-machine

`docker-machine`

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

- Il dialogue avec les systèmes 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.

5 Conclusion

Conclusion

Pourquoi utiliser docker ?

- parce que cela permet de travailler confortablement
  - sans changer de machine ;
  - en maitrisant les bibliothèques, 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/spark sans rien comprendre à son fonctionnement.