B3 - Conteneurisation

Hey yo.
Tout le monde avec des machines CentOS (et mises à jour !) pendant mes TPs.
Ce TP va assez loin sur certains points. N'hésitez pas à m'appeler pour ne pas rester bloqués dans le flou le plus total, si un point n'est pas clair ou simplement mal compris.
Most important : have fun.

TP 1 - Containerization Basics

But :

- appréhender et comprendre la conteneurisation
  - en particulier la conteneurisation avec le kernel GNU/Linux
- créer des conteneurs (plus ou moins) à la main (il existe tant de façon de faire ça une fois qu'on a compris le concept)
- appréhender les éléments standardisés autour des conteneurs
  - image (ACI)
  - runtime (runc)
- comprendre certains aspects de sécurité de Docker et savoir le configurer
- utilisez `docker` + `docker-compose`
- répondre au use case de devoir faire tourner des apps packagées sous forme de conteneurs, de façon simple et relativement robuste

Conteneurs : utilisation simple, connaissances avancées

Dans cette partie, vous allez installer et utiliser très brièvement Docker. Puis vous utiliserez d'autres outils afin de mettre en place de la conteneurisation. L'idée est de détacher les mots "conteneurs" et "Docker" dans votre vocabulaire, en essayant de vraiment comprendre ce qu'est un conteneur.

Docker

Installation

Simple : suivez la **doc officielle**. Tout est dans la doc pour l'installation, **LISEZ DE HAUT EN BAS** et exécutez ensuite.

Détails de l'installation :

- utilisez `repoquery` pour voir les fichiers installés par ce paquet
- démarrez le service `docker` à l'aide d'une commande `systemctl`
- mettez en place une configuration afin de pouvoir utilisez docker **sans être** `root`. Expliquez en quoi cela pourrait être dangereux d'un point de vue sécurité (c'est en rapport avec le groupe d'utilisateurs `docker`)
- utilisez `docker info` pour s'assurer du bon fonctionnement de Docker

Utilisation élémentaire :

- téléchargez l'image `ubuntu` avec `docker pull`
- créez un conteneur en utilisant l'image `ubuntu` (avec `docker run`)
- créez un conteneur en utilisant l'image `ubuntu`, faites en sorte que le conteneur reste en vie, et rentrez à l'intérieur de celui-ci avec un shell `bash` (vous devez donc pouvoir passer des commandes dans le conteneur)

- une fois en possession du `bash` dans le conteneur, baladez-vous un peu. Regardez ses cartes réseau (avec des commandes `ip`), ses processus (commandes `ps`), etc.
- faites de même avec l'image `alpine`

Ok mais comment ça marche ?
Ici, on va regarder d'un peu plus près les `cgroups` et les `namespaces`.

- lancez un conteneur en tâche de fond (peu importe lequel), ce conteneur doit lancer un processus facilement identifiable (par exemple `sleep 999999`)
- depuis votre machine hôte, trouvez le processus lancé à l'intérieur du conteneur (à l'aide d'un `ps`)
- mettez en évidence l'utilisation des `namespaces` et des `cgroups` par ce conteneur
  - il existe plusieurs façons de faire ça, je vous en ai déjà montré, ça se trouve aussi sur l'ami Internet
  - indices en vrac : `systemd` pourra vous aider, le répertoire `/proc` aussi

Autres moyens de conteneurisation

Ok donc un conteneur Docker c'est un process COMME LES AUTRES, à part qu'on le fait tourner dans des `namespaces` spécifiques, en limitant ses ressources à l'aide des `cgroups`. Donc n'importe quel outil qui sait lancer des process, les mettre dans des `namespaces` et les gérer avec des `cgroups` sait faire des conteneurs non ? A peu de choses près, oui...

Dans cette partie, n'hésitez pas à jouer en regardant ce qu'il se passe niveau `namespaces`/`cgroups` à chaque fois que vous lancez des conteneurs (en regardant `/proc` ou avec des commandes comme `ip a` ou `ps`) et faites m'en part dans le compte-rendu du TP si vous le faites.

Petite précision : nous allons souvent travailler avec Alpine Linux, c'est un OS extrêmement léger. Ca vous évitera de tous télécharger un truc de 800Mo en même temps. `bash` n'est pas présent par défaut dans Alpine, vous utilisezez `sh` si vous voulez un shell.

Quelques rappels dans cette optique :

- la suite de commande `ip` est très puissante (visualisation de la stack réseau du système : les infos sur les carte réseau)
- `mount` (entre autres) vous permet d'afficher les points de montage
- `ps -ef` ou `ps aux` permet de listez les processus au sein d'un système
- le répertoire `/proc` contient énormément d'informations quant à l'état de l'OS et du kernel

1. `systemd-nspawn`

Outil bas-niveau. Utilisation de `systemd` pour créer des conteneurs. Utilisation de la commande `systemd-nspawn`. En bref :

- *`chroot` on steroids* (chroot + namespacing)
- peut faire boot un OS complet (je parle de VRAIMENT dérouler la séquence de boot)
- impossible de reboot, ou de toucher au kernel (chargement de modules, etc..)

Création et manipulation d'un conteneur Alpine :

- récupérez le fiilesystem de Alpine, "à plat", dans un dossier
  - `mkdir alpine`
  - `docker export $(docker create alpine) | tar -C alpine -xvf -`
  - ceci remplira le répertoire `alpine` avec le filesystem complet de la dernière version d'Alpine
  - regardez un peu dedans pour comprendre ce qu'il vient de se passer
- lancer un conteneur simple avec `systemd`, en utiliisant le répertoire Alpine : `systemd-nspawn -D ./alpine`
- tout ça c'est du systemd hein. `systemctl | grep machine`, vous devriez voir un scope cgroup dans lequel évolue votre conteneur. `systemd-cgls` ou `systemd-cgtop` pour mieux le voir
- explorez les options de `systemd-nspawn` :
  - créer un répertoire temporaire dans le conteneur avec `tmpfs` (bonus : expliquez l'utilité de `tmpfs`)

    - isoler complètement le conteneur en terme de réseau
- Debian container et boot complet (le téléchargement peut être très long à Ingésup, vous pouvez passez outre cet exercice si c'est le cas)
    - utilisez `debootstrap` (besoin des dépôts EPEL pour l'installer)
    - choisissez une distrib dans `/usr/share/debootstrap/scripts/`
    - pour un debian wheezy `debootstrap wheezy /deb http://deb.debian.org/debian/` vous pouvez changer l'url vers `http://archive.ubuntu.com/ubuntu/` pour une distrib ubuntu
    - `systemd-nspawn -D /deb` pour avoir un shell dans le conteneur
    - en utilisant `systemd-nspawn -D /deb -b` vous pouvez lancer la séquence de boot
    - avec ce répertoire, vous pouvez aussi faire un *chroot* dedans : `chroot /deb`

2. avec `runc` et le standard ACI

Outils standards portés par l'**OCI**, jetez un oeil **ici** et **ici** pour saisir un peu mieux. Je ferai un point là-dessus en cours.

- **caractéristiques** :
    - outil standard
    - très léger et autonome
    - grosse communauté
    - standard de l'OCI
    - initié par Docker sous le nom de *libcontainer* puis légué à l'OCI
    - c'est aussi le runtime utilisé par docker
- installer `runc` (juste un paquet)
- démarrez le service systemd `docker` si ce n'est pas fait
    - `docker info | grep -i runtime`
- créer un fs alpine Linux + les metadatas standardisées (standard ACI de l'initiative OCI). Pour ce faire :
    - `mkdir -p <WORK_DIR>/rootfs`
    - se déplacer dans le répertoire `<WORKDIR>`
    - créer un fs alpine `docker export $(docker create alpine) | tar -C rootfs -xvf -`
    - toujours dans le même répertoire, générer les metadatas `runc spec`, un fichier json a pop dans le répertoire courant
    - ouvrez ce fichier `JSON` et expliquez brièvement son contenu (**brièvement**, pas une description ligne par ligne)
    - `runc run <NAME>` et vous avez votre conteneur
    - `runc list` pour voir les conteneurs

3. avec... rien ?

Let's hack this shit.

- **partie PAS obligatoire**
- **l'idée** :
    - un *conteneur* c'est juste un concept, on peut l'implémenter soi-même de mille et une façons différentes
- on peut réutilisez les filesystems créés juste au dessus pour faire un conteneur applicatif juste avec du *systemd* (juste une unité de type *service*) :
    - mise en place d'isolation *namespaces* et restrictions *cgroups*
    - set de *capabilities* limité
    - un applicatif qui se lance en `ExecStart` (comme un nginx) dans un filesystem créé plus haut (on peut `chroot` à l'intérieur pour le configurer facilement
    - ça tient sur quelques lignes et on crée un process complètement cloisonné

- c'est aussi gérable de coder rapidement un petit programme (en Python par exemple) qui crée des conteneurs très basiques (en utilisant les appels système `unshare` et `nsenter`, il exiiste une commande bash pour les deux, et une lib Python)

4. avec rkt

- **caractéristiques** :
  - réputé robuste & secure
  - grosse communauté
  - conforme aux standards
  - initié par coreOS
- *rkt* est une alternative à Docker (initiative de CoreOS pas mal poussée par d'autres grands acteurs)
- récupérez le rpm pour centos `https://github.com/rkt/rkt/releases/download/v1.30.0/rkt-1.30.0-1.x86_64.rpm`
- installez le (`rpm -ivh <PACKAGE>`)
- *rkt* n'utilise pas de démon pour gérer les conteneurs (contrairement à docker). Rien à démarrer donc !
- `rkt list` pour voir les conteneurs lancés
- pour lancer une image Docker alpine dans un conteneur rkt : `rkt --stage1-name=coreos.com/rkt/stage1-fly:1.29.0 --insecure-options=image run docker://alpine`
- **Note** : le paramètre `--stage1-name` est obligatoire sur nos machines CentOS (vous devriez obtenir une erreur sans lui). N'hésitez pas à me poser la question du pourquoi du comment, je ferai aussi sûrement un point là-dessus une fois que tout le monde l'aura fait

rkt n'utilise pas de démon donc impossible de mettre un conteneur en fond (il serait à la charge du démon... qui n'existe pas donc !). On peut missionner `systemd`, pour ceci :

- utilisez la commande suivante pour lancer exécuter un `rkt run` qui sera géré par `systemd`

```
systemd-run rkt --insecure-options=image --stage1-name=coreos.com/rkt/stage1-fly:1.29.0
run docker://alpine --exec /bin/sleep -- 9999
# systemd-run est très puissant
```

- expliquez cette ligne de commande (n'hésitez pas à poser des questions, encore une fois)
- utilisez la commande `systemctl` pour voir l'état du service lancé
- **note** : on peut aussi lancer les conteneurs dans des VM avec *rkt* (voir modification du "*stage1*" *rkt*)

Docker focus

Caractéristiques de Docker (en quelques mots) :

- très répandu
- léger et désormais **vraiment** cross-platform
- très utile pour des environnements de développement
- intégration avec énormément d'outils

Basic configuration

- installation de docker : suivez **la doc officielle**, **c'est pas dans vos dépôts**
- unités **systemd** de type *service* et *socket* dédiées
  - pendant le démarrage du *service*, population de `/var/lib/docker`
  - et création du *socket* UNIX dédié à Docker
- une fois le service démarré vous pouvez utilisez Docker avec `root`. Ou avec les membres du groupes `docker`, et c'est pas de la magie, cette restriction est apposée quelque part. Cherchez où est apposée cette restriction (réfléchissez à ce que vous cherchez : on parle des droits d'accès à Docker, ou plutôt, au poiint

d'accès pour échanger de la donnée avec Docker...)
- changer la configuration de base du démon docker `dockerd`
  - utilisez un répertoire de travail (l'habituel `/var/lib/docker`) sur une partition LVM à la racine (`/data` par exemple)
  - changez le *OOM score* du démon (plus il est haut et plus il y a des chances qu'il se fasse détruire en premier)
- créer une autre unité systemd `docker-tcp.service`
  - lancez une deuxième instance du démon Docker, accessible à travers un socket TCP (à travers le réseau donc)
  - testez cette configuration avec TCP (depuis votre machine hôte, une deuxième VM, ou en local en forçant une connexion TCP)
  - quelles autres options du démon docker (`dockerd`) peuvent être utilisées pour le rendre plus sécurisé et/ou robuste et/ou restreint ?

Basic `docker run`

- quelques commandes utiles :
  - lister les conteneurs `docker ps` ou `docker container ls`
  - lister les images `docker images` ou `docker image ls`
  - lister les réseaux `docker network ls`
  - `docker stats`
- vérifiez configuration/installation : `docker info`
  - le runtime est bien `runc` btw
- lancez quelques conteneurs avec `docker run`
  - `docker run -d alpine sleep 9999`
  - `docker run -it alpine sh`
- exploration des options de `docker run`
  - trouvez comment donner un nom à vos conteneurs, ainsi qu'un hostname
  - jusqu'alors, vos conteneurs lançaient des processus en tant que root (on peut le voir en lannçant un `ps` sur l'hôte, pendant qu'un conteneur tourne). Faites en sorte de modifier ce comportement
  - quelles autres options de `docker run` peuvent être utilisées pour réduire les droits d'un conteneur ?
  - utilisez `-v` pour monter le répertoire `/home` de l'hôte dans un conteneur alpine
- utilisation d'une image du Hub
  - utilisez l'image `nginx` du dépôt library et accéder à la page d'accueil de *NGINX* sur le port 8888 de l'hôte (votre PC)
  - optionnel : lancez le conteneur *NGINX* depuis une unité *systemd* de type *service*. L'idée est de pouvoir interagir avec le conteneur (le service) en utilisant le binaire `systemctl` (par exemple `systemctl start super-nginx` pour démarrer le conteneur)

Création de Dockerfile

Pour rappel, les Dockerfiles servent à créer des images, elles-mêmes utilisées pour lancer des conteneurs.

Les Dockerfiles (à part cas spécifiques) sont le plus souvent créés à partir d'autres Dockerfiles. Vous voulez packager Python ? Prenez une image de base légère (alpine, debian, autres) et ajoutez-y Python.

- le but ici sera de créer un Dockerfile qui lance une application Web simple, exposée vers l'extérieur
  - créez un Dockerfile basé sur debian
  - créez, dans l'image, un répertoire à la racine qui contiendra votre site : `/web`
  - définissez un répertoire de travail au sein du Dockerfile (clause `WORKDIR`)
  - déposez un fichier HTML simple dans le répertoire de travail (il faudra le créer en dehors de l'image pour pouvoir l'ajouter dedans)
  - faites en sorte que la commande lancée par le conteneur soit un `sleep`, avec une valeur par défaut de 60, modifiable par l'utilisateur au lancement du conteneur (utilisation de `CMD` et `ENTRYPOINT`)
  - assurez-vous que le paquet permettant d'avoir Python 3 est présent dans l'image
  - modifiez la commande lancée par le conteneur pour être : `python -m http.server 8888` avec la possibilité de changer la valeur `8888` au lancement du conteneur

    - mettez en place l'utilisation d'un utilisateur applicatif dans l'image (clause `USER`)
    - enfin, indiquez au sein du Dockerfile que le port 8888 sera exposé
- lancez votre conteneur et faites une requête `curl` (ou un screen avec un navigateur) pour voir le contenu de votre page HTML

HTTP API

Comment ça c'est un truc de dév ?
Nan sans déc y'a un vrai intérêt. Même plusieurs. Montrer que :

- un socket c'est juste l'endroit où la donnée s'échange. On peut faire transiter n'importe quoi comme info, par exemple de l'HTTP. Le démon docker attend de l'HTTP à travers un socket UNIX (par défaut)
- c'est "programmatique" comme approche. On pourrait construire nous-même un binaire ou un client pour faire ce que l'on fait d'habitude avec la commande `docker`
- cette API, elle est conforme aux standards. Une API similaire est présente, par exemple, sous `rkt` ou les `VIC` (d'ailleurs, avec un VIC engine, on utilise le binaire `docker` quand même pour taper dessus : VIC engine expose la même API que docker, on peut donc utiliser le même outil pour l'utiliser.)
- utilisez l'option `--unix-socket` de `curl` pour requêter l'API HTTP de docker
    - `curl --unix-socket <PATH_TO_SOCKET> http://<URI>`
        - quelle URI ? Cette fois-ci je vous laisse chercher la doc tout seuls
    - récupérez la liste des conteneurs actifs
    - récupérez la liste des images
    - lancez un conteneur et récupérez son IP depuis une requête `curl`

Docker : configuration avancée du démon

Cette partie est dédiée à la configuration du démon docker `dockerd`, l'applicatif avec lequel on discute en tapant des commandes `docker` (en passant par le socket toujours).
Pour modifier la configuration du démon, on peut directement modifier l'unité systemd `docker.service` ou aussi, plutôt, le fichier `/etc/docker/daemon.json` (vous trouverez les infos pour réaliser cette partie dans la doc dédiée à `dockerd`)

- votre démon Docker doit utilisez **la politique `seccomp` recommandée par le projet Moby**
    - expliquez brièvement, avec vos mots, ce qu'est `seccomp` et le rapport avec `docker`
- suivez **la doc officielle** pour mettre en place l'utilisation d'une backend `device-mapper` en `direct-lvm` ("*CONFIGURE DIRECT-LVM MODE MANUALLY*") côté stockage
    - expliquez l'utilité d'utiliser ce driver de stockage
    - mettez en place une configuration valide avec le driver `device-mapper`
    - test : `docker info`, ou `df -h` à chaque lancement de conteneur
    - test2 : lancer un conteneur, exécuter un shell dedans, remplir le disque complètement. Plus aucune opération est réalisable. On peut tuer le conteneur depuis l'extérieur (le disque de l'hôte n'est pas rempli)
- optionnel : utilisation des *user namespaces*
    - ceci permettra d'accéder à un très haut niveau de sécurité (via une isolation quasi-totale des utilisateurs de l'hôte par rapport à ceux des conteneurs). Expliquez avec vos mots en quoi consiste l'utilisation des *user namespaces* par `docker`
    - activez l'utilisation des *user namespaces* par votre kernel
    - utilisez le *user namespace remapping* du démon docker
    - test : vérifiez l'appartenance de votre répertoire Docker de data (`/data` si vous l'avez changé comme demandé plus haut)

Déploiement d'application n-tiers avec `compose`

- installez `docker-compose` en suivant la doc officielle
- packagez le code python fourni (créer une image) (deux fichiers : un qui a l'app, l'autre qui a les dépendances. Pour installer les dépendances, il faut faire `pip install <FILE>` où `FILE` est le fichier `requirements`. Je vous conseille de partir de l'image `python:3.5.4-alpine`
- créez un compose qui contient :
  - un conteneur *Redis* (stockage clé/valeur)
  - un conteneur avec l'app Python packagée (qui écrit/lit des valeurs dans Redis)
  - l'app Python doit pouvoir joindre un hôte Redis avec le hostname `db` sur le port 6379
- ouvrez un navigateur sur votre machine, rdv à `http://<IP_VM>:5000`
- modifiez le fichier `yml` et ajouter un troisième conteneur reverse proxy NGINX qui redirige vers l'interface web de l'app Python
- à la fin :
  - un conteneur frontal `front` : NGINX, qui écoute sur le port 80, accessible depuis l'extérieur
  - un conteneur applicatif `app` : l'app Python, packagée par vos soins, joignable uniquement depuis le réseau de votre `docker-compose`
  - un conteneur de bdd Redis `db`, qui écoute sur le port 6379, dans lequel `app` vient écrire

Utilisation d'un GUI avec Portainer

Partie pour faire joujou.

- Déployez Portainer sur votre hôte existant (ça se trouve sur github et ça se monte en 2/2 !)
- créez un deuxième hôte Docker et le piloter depuis Portainer (socket TCP)