BORDEAUX INP Enseirb-Matmeca

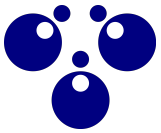
Document de travail
réalisé par des élèves.

Introduction à la conteneurisation

Maxime TUEUX et Sam VIE

RSI – 2022/2023

© Ian Battaglia / Unsplash



Mis en page à l'aide de LuaLaTeX

Édition du 2 octobre 2022 à 16:14

© 2022 Maxime TUEUX, Sam VIE. Tous droits réservés.

La diffusion partielle ou non de ce document, quel qu'en soit le format (matériel ou non), est interdite sans l'accord de son auteur. Les droits des illustrations sont la propriété de leurs auteurs respectifs.

Glossaire

VM : Virtual Machine : Machine virtuelle
CI : Continuous Integration : Intégration Continue
CD : Continuous Deployment/Delivery : Déploiement continu
NB : Nota Bene
SGBD : Système de Gestion de Base de Données
UFS : Union File System

Table des matières

Chapitre 1

Définition et intérêts d'un environnement conteneurisé

1.1 Définition

Un environnement conteneurisé désigne un environnement virtualisé dont les éléments constitutifs sont isolés au sein de briques appelés « conteneurs ».

Ces conteneurs, à l'instar d'une machine virtuelle (ou VM), regroupent en plus de l'application à lancer, les bibliothèques et binaires nécessaire à son exécution. En revanche, un conteneur n'inclut pas de système d'exploitation, il s'appuie sur le système hôte et sur le moteur de conteneurisation pour accéder aux ressources matérielles du système [1] (figure 1.1).

Enfin, en plus de ces éléments, le conteneur contient directement le moyen d'exécuter l'application qu'il embarque, ainsi que quelques menus éléments de configuration.

1.2 Intérêts

Ce fonctionnement, que l'on pourrait qualifier de virtualisation simplifiée ou allégée, est porteur d'un grand nombre d'avantages, tant pour le développement d'applications que pour leur déploiement. Nous tâcherons ici d'en présenter un maximum, sans pour autant prétendre être exhaustifs.

1. Pour être plus précis, le conteneur n'embarque que la plus haute des quatre couches du système d'exploitation (aussi appelée couche « Applications ») sous laquelle va s'exécuter l'application qu'il contient. Cela permet, par exemple, de choisir la distribution la plus adaptée à un contexte d'exécution. Pour les trois autres couches, le conteneur s'appuiera sur le système hôte.

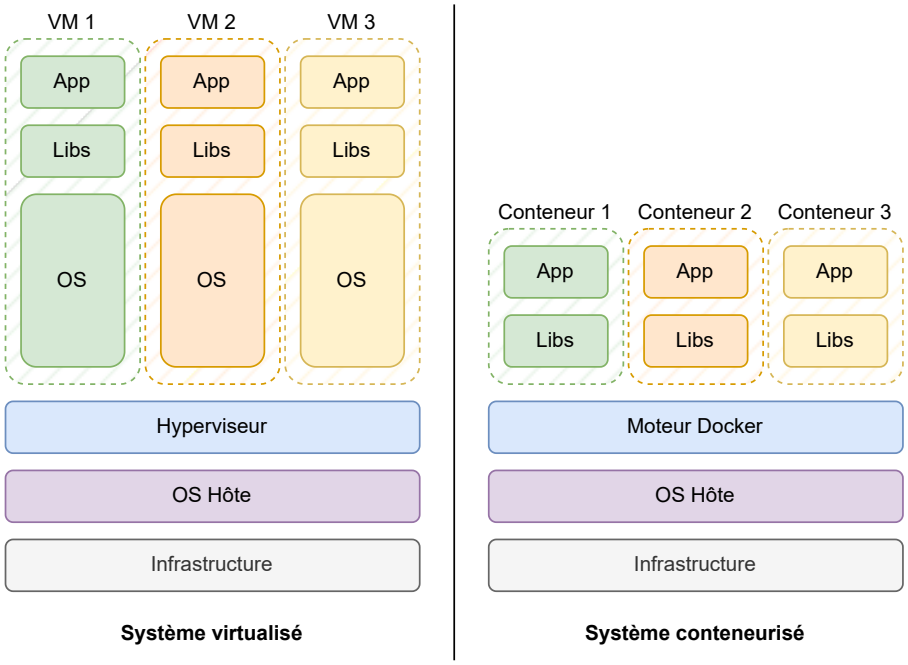


Figure 1.1 – Comparaison entre un système utilisant des VM et un système conteneurisé

1.2.1 En phase de déploiement

Il suffit de regarder la figure 1.1 pour remarquer le premier avantage des environnements conteneurisés : l'impact sur les performances est amplement moindre par rapport à des VM, puisqu'on élimine l'un des défauts majeurs de ces dernières : la nécessité pour chaque VM de redéployer l'ensemble des couches de son système d'exploitation.

Un autre avantage est la possibilité d'affecter aux conteneurs leurs ressources de manière dynamique, voire de borner ces allocations [2].

Ensuite, les interfaces construites pour le travail avec les conteneurs sont dans l'ensemble plus simples d'utilisation que des systèmes de virtualisation avancés comme RedHat OpenStack, mais offrent à peu près les mêmes fonctions. En somme, il n'est pas nécessaire de disposer de connaissances avancées en administration système ou en réseaux pour proposer un déploiement consistant et cohérent en s'appuyant sur la conteneurisation.

Enfin, on retrouve les avantages des systèmes virtualisés plus classiques : confiner les différentes composantes d'une pile applicative dans des environnements séparés et relativement étanches permet de réduire les conséquences d'une compromission d'un élément, qui n'impactera de ce fait pas les autres composantes de la pile. Également, dans un contexte de déploiement continu (CD), il est plus simple de manipuler et de transférer des applications « empaquetées » sous la forme d'images de conteneurs, plutôt que de manipuler directement le code et les exécutables des applications que l'on souhaite déployer.

2. Par exemple, il est possible borner entre 512 MB et 1024 MB la consommation de RAM d'un conteneur, voire de ne le border que dans un sens, de manière à lui allouer un minimum syndical de ressources mais à l'autoriser à aller piocher plus si des ressources sont disponibles.

1.2.2 En phase de développement

En phase de développement, l'intérêt principal de la conteneurisation ne réside pas le plus souvent dans le lancement et le fonctionnement du produit développé lui-même. En revanche, les applications tierces qui servent au développement d'un produit [3] ont elles tout intérêt à être déployées via des conteneurs !

En effet, cela permet un haut niveau de flexibilité dans l'utilisation des ressources du poste de développement, ainsi que dans la nature même des services tiers utilisés.

Ainsi, imaginons que nous soyons en train de développer une application web pour un projet A, applications architecturée comme suit : un frontend, un backend, une base de données relationnelle PostgreSQL version 12 et une base Redis 6 pour les données d'état. À côté de ce projet A, nous avons un projet B, composé d'une API REST, d'une base PostgreSQL 13 et d'une base Redis 6. Ces projets utilisent globalement les mêmes technologies, mais ont besoin de deux versions distinctes du même logiciel : PostgreSQL. Or, il n'existe pas de moyen simple de faire cohabiter sur le même poste de travail deux versions du même logiciel, et quant bien même ce serait possible, les deux SGBD [4] consommeraient simultanément de la ressource système, ce qui pèsera nécessairement sur la fluidité de la machine.

Imaginons maintenant que nos services tiers sont conteneurisés : en lieu et place des trois SGBD, nous installons Docker et montons trois containers : PostgreSQL 12, PostgreSQL 13 et Redis 6, à l'aide de trois petites commandes. Non seulement un simple mappage des ports nous permet de faire cohabiter sans souci nos deux versions de PostgreSQL, mais nous pouvons stopper le conteneur inutile quand il n'est pas utilisé et le réactiver à tout moment. Nous pourrons le réactiver plus tard et il reprendra son exécution là où il s'était arrêté. Nous pouvons également stopper complètement Docker pour économiser les ressources du système si les conteneurs sont inutiles, et relancer à tout moment le runtime qui fera remonter tout seul tous les conteneurs actifs avant son arrêt.

Mais ce n'est pas le seul intérêt de la conteneurisation : elle permet également de simplifier très largement les procédures de paquetage et de déploiement dans un contexte de CI/CD, ce qui se montre très utile pour tester rapidement le produit développé dans un environnement iso-production.

3. Une ou des base(s) de données par exemple.
4. Système de Gestion de Base de Données

Chapitre 2

Les éléments constitutifs d'un environnement conteneurisé

2.1 Les images

Une image est un instantané inerte d'un conteneur, qui contient l'ensemble des informations constitutives de ce dernier, c'est-à-dire son environnement logiciel (distribution linux, packages nécessaires, etc...), le logiciel en lui-même et sa commande d'amorce, ainsi que potentiellement certains éléments de configuration, comme son port d'exposition, ou la description d'une sonde de santé.

Une image est décrite par un *Dockerfile*, qui s'inscrit directement au sein des fichiers du projet. Une fois construite, elle est stockée soit localement, soit dans un registre de conteneurs (voir section 2.5). **Une fois créée, le contenu d'une image est inaltérable.**

2.2 Les conteneurs

À partir d'une même image, on peut produire autant de conteneurs que l'on souhaite. On pourrait dégrossir le tableau en disant qu'un conteneur est « une image qui tourne », mais ce ne serait pas tout à fait exact : au moment d'exécuter un conteneur à partir d'une image, on peut lui passer toute une flopée d'éléments de configuration facultatifs qui s'avèrent déterminants vis-à-vis de son environnement final d'exécution.

On peut ainsi :
- Lui monter un ou plusieurs volumes (voir section 2.3)
- Le connecter à un ou plusieurs réseaux (voir section 2.4)
- Lui passer des variables d'environnement
- Lui configurer une sonde de santé
- Surcharger sa commande d'amorce

En faisant varier ces paramètres, il est possible d'adapter totalement le fonctionnement du programme contenu dans le conteneur à un usage ou à un environnement spécifiques, **à condition que le programme concerné ait été développé en tenant compte de ces éléments de configuration**.

Le fonctionnement en environnement conteneurisé demande donc des changements de pratiques de développement non négligeables afin de permettre une finalité d'usage la plus souple possible.

2.3 Les volumes

Le système de fichier des conteneurs Docker est composé d'un ensemble de couches (layers) appelé Union File System (UFS). Toutes les layers définies dans l'image sont en **lecture seule**. Par-dessus celles-ci, Docker rajoute un layer supplémentaire, qui est quant à lui **en lecture-écriture**, et qui est supprimé en même temps que le conteneur.

Cela signifie que par défaut, **l'ensemble des modifications que le conteneur effectue sur son système de fichier sont éphémères** et ne survivront pas à une suppression ou une recréation du conteneur.

Afin de rendre les données d'un conteneur persistantes, on crée des **volumes**, que l'on pourrait apparenter à des volumes virtuels, que l'on monte sur le système de fichier d'un ou plusieurs conteneurs.

Il existe deux manières de déclarer des volumes : soit on déclare la création d'un volume que l'on nomme. Docker lui définira un emplacement dans le système de fichier hôte [1], que l'on monte ensuite dans le système de fichiers d'un conteneur en l'appelant par son nom, soit on peut directement utiliser un emplacement de système de fichiers hôte en donnant directement son chemin relatif ou absolu à l'exécution du conteneur. Il est à noter qu'il est tout à fait possible de partager un même volume entre plusieurs conteneurs.

Un conteneur considère donc deux informations pour chaque volume auquel il est lié : d'une part son chemin dans le stockage hôte (mis sous alias par Docker ou non), et d'autre part le chemin sous lequel il sera monté dans son propre système de fichiers.

2.4 Les réseaux

Les réseaux Docker sont des réseaux virtuels qui permettent l'interconnexion des conteneurs entre eux et avec le monde extérieur. Il existe cinq types de réseaux par défaut, sachant qu'il est possible d'en créer des personnalisés.

En marge de ce système, si deux conteneurs sont sur le même réseau et souhaitent communiquer, on pourra utiliser le nom du conteneur que l'on souhaite contacter en lieu et place de son adresse IP. Cela est particulièrement utile dans les cas d'un driver `bridge` ou `overlay`.

1. En général sous `/var/lib/docker/volumes/`

Le driver *none* sert à interdire toute communication (entre conteneurs comme avec l'extérieur) au conteneur auquel il est attaché.

Le driver *host* expose directement les conteneurs sur l'interface hôte. Ce driver supprime toute forme d'isolation réseau entre les conteneurs et avec l'hôte, il n'est donc pas recommandé de l'utiliser dans un environnement de production vulnérable.

Le driver *bridge* est le driver du réseau par défaut, nommé « `bridge` ». Celui-ci est connecté sur l'interface réseau `docker0` de l'hôte. Les conteneurs y sont affectés par défaut s'il n'ont pas de configuration réseau précisée.

Dans cette configuration réseau, chaque conteneur se voit attribuer une adresse IP, et est connecté à un sous-réseau dont le point de sortie est l'interface `docker0` de l'hôte. Les conteneurs sont libres de communiquer entre eux et avec l'extérieur, mais l'extérieur ne peut communiquer avec les conteneurs. La communication de conteneur à hôte est impossible.

Le driver *overlay* est utilisé dans le cas d'un système multi-nœuds où les conteneurs sont répartis sur plusieurs serveurs. Il de créer un réseau distribué sur l'ensemble des hôtes. À part cette caractéristique supplémentaire, son fonctionnement est en tous points identique au driver `bridge`.

Le driver *macvlan* permet d'affecter une adresse MAC aux conteneurs qu'il contient, les faisant apparaître sur le réseau comme des périphériques physiques.

2.5 Les registres

Ce sont des dépôts d'images. Il en existe de trois sortes :

Les registres locaux Chaque instance de Docker en possède un, qui stocke localement les images utilisées et construites par un poste / un serveur.

Les registres publics À l'instar de Docker Hub, ils mettent à disposition des images publiquement.

Les registres privés Leur fonctionnement est identique aux registres publics, mais ils nécessitent une authentification préalable.

Par défaut, si l'image requêtée n'est pas trouvée localement, elle est recherchée sur le Docker Hub [2].

Pour requêter une image qui est présente sur un autre registre que le Docker Hub, il faut préciser son URL [3].

2. NB : Le nombre de téléchargements d'images depuis le Docker Hub est limité à 600 par heure. (09/2022)
3. Par exemple, `postgres` devient `registry.example.com/postgres`

2.6 Les piles applicatives

Une pile applicative, ou « stack » est un ensemble de conteneurs, volumes, réseaux et configurations prévus pour fonctionner dans un ensemble logique. Leur configuration est regroupée au sein d'un unique fichier, pour faciliter sa gestion et son déploiement.

Cette configuration se présente sous la forme d'un fichier YAML, et est mise en œuvre via l'utilitaire Docker Compose.

Table des figures