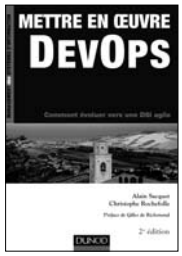
DOCKER

Pratique des architectures
à base de conteneurs

K. Hightower, B. Burns, J. Beda
208 pages
2019



A. Sacquet, C. Rochefolle
288 pages
2018

C. Legrenzi, P. Rosé
272 pages
2016

C. Aubry
384 pages
2018

DOCKER

Pratique des architectures à base de conteneurs

Pierre-Yves Cloux
Director of Software Development
chez SITAONAIR

Thomas Garlot
Responsable d'équipe DevOps
chez SITAONAIR

Johann Kohler
Architecte logiciel
chez SITAONAIR

2e édition

DUNOD

Illustration de couverture : © Tryaging – iStock

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, 2016, 2019
11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-078970-2

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Avant-propos

Pendant longtemps, déployer du code en production revenait à tenter de transporter de l'eau entre ses mains : c'était fonctionnel, mais pas vraiment optimal. Comme l'eau filant entre les doigts, il manquait presque nécessairement une partie des données de configuration lors du déploiement, ceci en dépit d'efforts méthodologiques, documentaires et humains conséquents.

◆ *L'apport de la virtualisation*

La virtualisation a tenté de répondre à cette problématique (parmi d'autres) sans apporter une réponse complètement satisfaisante. En effet, bien qu'offrant une isolation vis-à-vis de l'architecture matérielle, qui se standardise de plus en plus, la machine virtuelle reste… une machine. Elle exécute un système d'exploitation dont les paramètres peuvent différer d'un environnement à l'autre.

Des outils comme Chef ou Puppet résolvent une partie du problème, mais n'offrent encore une fois qu'une couche d'abstraction partielle. Enfin, le déploiement d'applications sur la base d'une image de VM (pour *Virtual Machine*) est lourd (plusieurs gigaoctets, y compris pour les OS les plus compacts).

◆ *Les architectures à base de conteneurs*

Avec Docker, nous sommes entrés dans l'ère des architectures à base de « conteneur ». On parle aussi de « virtualisation de niveau système d'exploitation » par opposition aux technologies à base d'hyperviseurs (comme VMWare ou VirtualBox) qui cherchent à émuler un environnement matériel.

Contrairement à une VM, un conteneur n'embarque pas un système d'exploitation complet. Il repose pour l'essentiel sur les fonctionnalités offertes par l'OS sur lequel il s'exécute. L'inconvénient de cette approche est qu'elle limite la portabilité du conteneur à des OS de la même famille (Linux dans le cas de Docker).

Nous verrons dans le chapitre 1 que l'implémentation Windows de Docker autorise l'exécution de conteneurs Windows et Linux, ce qui contredit l'affirmation précédente. Néanmoins Microsoft fait usage d'une solution un peu particulière qui, quoique fonctionnelle, s'éloigne du concept usuel de conteneur tel qu'il a été imaginé à l'origine.

Cette approche, en revanche, a l'avantage d'être beaucoup plus légère (les conteneurs sont nettement plus petits que les VM et plus rapides au démarrage) tout en offrant une isolation satisfaisante en termes de réseau, de mémoire ou de système de fichiers.

Étonnamment, le concept de conteneur et son implémentation ne sont pas nouveaux. La version de OpenVZ pour Linux date, par exemple, de 2005, de même que Solaris Zone ou FreeBSD jail. Docker a changé la donne en simplifiant l'accès à cette technologie par l'introduction d'innovations, comme la mise à disposition d'un langage de domaine (DSL ou *Domain Specific Language*) permettant, au travers du fameux « Dockerfile », de décrire très simplement la configuration et la construction d'un conteneur.

◆ *Le propos de cet ouvrage*

L'objet de cet ouvrage est d'offrir une approche à 360 degrés de l'écosystème Docker.

Docker, plus qu'une technologie, est en effet aujourd'hui un écosystème de solutions fourmillantes : Docker Compose, Kubernetes (et ses nombreuses implémentations chez les leaders du cloud public), ou encore Docker Swarm.

Autour du *runtime* (moteur d'exécution) qui exécute les conteneurs (le Docker Engine), des outils complémentaires visent à conditionner, assembler, orchestrer et distribuer des applications à base de conteneurs. Timidement, des initiatives de standardisation voient le jour, laissant espérer une meilleure interopérabilité que celle qui prévaut aujourd'hui dans le domaine de la virtualisation matérielle.

Notre objectif dans cet ouvrage est donc multiple :

- ✓ aborder le concept de conteneur et d'architecture à base de conteneurs en décryptant les avantages proposés par cette approche ;
- ✓ apprendre à installer Docker sur un poste de travail ou dans un environnement serveur ;
- ✓ apprendre à utiliser Docker pour créer des images et manipuler des conteneurs ;
- ✓ étudier des architectures plus complexes au travers d'exemples complets : architectures multi-conteneurs, développement, intégration continue et implémentation d'un cluster multi-hôtes.

◆ *La structure du livre*

Dans une première partie, nous expliquerons ce qu'est un conteneur, sur quels principes et quelles technologies il repose. Nous verrons aussi comment Docker se positionne parmi un nombre croissant d'acteurs et de logiciels importants.

Nous aborderons ensuite le concept de « CaaS », pour « *Container as a Service* », au travers de divers exemples. À cette occasion, nous parlerons d'outils tels que Kubernetes, Swarm, Mesos ou Azure Service Fabric. Nous nous intéresserons aux liens entre les approches conteneurs et celles d'outils de gestion de configuration comme Puppet, Chef ou Ansible.

La seconde partie de cet ouvrage se focalise sur la prise en main de Docker en étudiant son installation sur un poste de travail ou dans un environnement virtualisé. Nous commencerons alors à « jouer » avec des conteneurs pour comprendre par la pratique les concepts abordés dans la première partie.

La troisième partie du livre est consacrée à l'apprentissage de Docker. Nous y aborderons les commandes du client et les instructions du Dockerfile. Cette troisième partie peut être lue séquentiellement, mais aussi servir de référence.

La quatrième partie du livre se consacre à la mise en œuvre des concepts appris précédemment autour d'exemples pratiques et réalistes. Nous étudierons ainsi le développement, le conditionnement et le déploiement d'une architecture à base de conteneurs en nous limitant au cas mono-hôte.

La cinquième et dernière partie est consacrée aux solutions d'orchestration de conteneurs que sont Swarm et Kubernetes. Nous montrerons comment l'application étudiée dans la quatrième partie peut être déployée sur plusieurs hôtes sans modification majeure des conteneurs.

◆ *À qui s'adresse ce livre*

Cet ouvrage s'adresse à un public mixte « DevOps » :

- ✓ si vous êtes engagé dans une organisation de développement (en tant que développeur, architecte ou manager), vous trouverez les informations nécessaires pour maîtriser rapidement la conception d'images Docker, mais aussi pour la réalisation d'architectures multi-conteneurs ;
- ✓ si votre domaine est plutôt celui de l'exploitation, vous acquerrez les compétences nécessaires au déploiement de Docker sous Linux.

L'objectif de ce livre est double :

- ✓ il permet d'accélérer la prise en main de cette technologie en présentant des cas d'usages illustrés par des exemples didactiques ;
- ✓ il offre aussi une référence illustrée d'exemples du langage DSL et des commandes de Docker.

Compléments en ligne

Le code source et les exemples de ce livre sont distribués via GitHub sur le dépôt public :

https://github.com/dunod-docker/docker-examples-edition2/

Le dépôt contient, outre les exemples de code, toutes les commandes chapitre par chapitre (permettant un copier/coller), l'ensemble des liens vers les sites externes ainsi qu'un errata et des articles complémentaires.

Veuillez vous reporter à la procédure d'installation du chapitre 3.

◆ *Remerciements*

L'écriture d'un livre est une activité consommatrice de temps qu'il n'est pas possible de mener à son terme sans l'assistance de nombreuses personnes.

Nous voudrions donc adresser nos remerciements à :

- ✓ Patrick Moiroux, pour son expertise de Linux en général, des architectures web distribuées et de Docker en particulier ;
- ✓ Bernard Wittwer, pour avoir attiré notre attention, il y a plus de trois ans, sur un petit projet open source sympathique nommé Docker dont il percevait déjà le potentiel ;
- ✓ Marc Schär, notre stagiaire de compétition, devenu expert de Kubernetes ;
- ✓ tous les contributeurs au projet open source BizDock (http://www.bizdock-project.org/), qui a été notre premier contact avec le monde des conteneurs et Docker ;
- ✓ nos épouses et nos enfants pour leur patience au regard du temps que nous avons passé isolés du monde extérieur à expérimenter des logiciels et à en décrire le fonctionnement.

Table des matières

TROISIÈME PARTIE
Apprendre Docker

QUATRIÈME PARTIE
Développer, déployer et opérer avec Docker

CINQUIÈME PARTIE
Orchestration de conteneurs

© Dunod - Toute reproduction non autorisée est un délit.

PREMIÈRE PARTIE

Les conteneurs : principes, objectifs et solutions

Cette première partie vise à présenter :

- ✓ les origines du concept de conteneur ;
- ✓ l'apport de Docker à cette technologie déjà ancienne ;
- ✓ comment les conteneurs autorisent la réalisation de nouvelles architectures informatiques ;
- ✓ comment les conteneurs colonisent les solutions de gestion d'infrastructure.

Cette première partie comprend deux chapitres. Le premier décrit ce qu'est un conteneur d'un point de vue technique et conceptuel. Le second chapitre présente un survol complet des solutions de gestion d'infrastructure à base de conteneurs : des plus modernes (que l'on nomme également CaaS[1]) aux plus classiques pour lesquelles les conteneurs apportent les avantages décrits dans le chapitre 1.

1. CaaS est l'acronyme en anglais de *Container as a Service*, soit en français « conteneur comme un service ».

Les conteneurs et le cas Docker

Objectif

L'objectif de ce chapitre est de décrire les concepts qui ont présidé à l'émergence de la notion de conteneur logiciel. Nous présenterons les briques de base sur lesquelles les conteneurs (et plus spécifiquement les conteneurs Docker) reposent. Puis nous expliquerons comment sont construits et distribués les conteneurs Docker. Enfin, nous nous intéresserons aux différents éléments des architectures à base de conteneurs : Docker et les autres.

À l'issue de ce chapitre vous comprendrez ce qu'est un conteneur, ce qu'est Docker et quels sont les concepts architecturaux et logiciels qu'il implémente. Vous saurez aussi quels sont les acteurs de cet écosystème.

1. 1 LA CONTENEURISATION

Les conteneurs logiciels cherchent à répondre à la même problématique que les conteneurs physiques aussi appelés conteneurs intermodaux, ces grandes boîtes de métal standardisées qui transportent de nos jours l'essentiel des biens matériels par camion, train, avion ou bateau.

Nous allons donc faire un peu d'histoire.

Docker et les conteneurs

Dans ce chapitre, comme dans la suite de cet ouvrage, nous allons aborder la question de la conception des architectures à base de conteneurs. Néanmoins, nous le ferons par le prisme de Docker qui en est l'implémentation la plus populaire. De ce fait, nous nous focaliserons sur les OS qui sont officiellement supportés par Docker (soit Linux, Mac OS X et Windows). Nous tenons à indiquer que des implémentations de conteneurs existent aussi pour d'autres systèmes, comme FreeBSD ou Solaris. Néanmoins, nous ne les aborderons pas dans le cadre de ce livre.

1.1.1 L'histoire des conteneurs intermodaux

Avant l'apparition de ces conteneurs standardisés, les biens étaient manipulés manuellement. Les Anglo-Saxons utilisent le terme de *break bulk cargo*, ce que nous pourrions

traduire par « chargement en petits lots ». Plus simplement, les biens étaient chargés individuellement par des armées de travailleurs.

Les biens étaient, par exemple, produits dans une usine, chargés un par un dans un premier moyen de transport jusqu'à un hangar (généralement près d'un port ou d'une gare) où avait lieu un déchargement manuel. Un second chargement survenait alors depuis ce premier lieu de stockage jusque dans le moyen de transport longue distance (par exemple un bateau). Ce second chargement était lui aussi manuel. En fonction des destinations, des contraintes géographiques et légales, cette séquence de chargement et déchargement pouvait avoir lieu plusieurs fois. Le transport de biens était donc coûteux, lent et peu fiable (biens abîmés ou perdus).

Les premières tentatives de standardisation du transport de biens sont intervenues en Angleterre à la fin du XVII^e^ siècle, essentiellement pour le transport du charbon. L'un des exemples bien connus est celui de la ligne de chemin de fer Liverpool – Manchester qui utilisait des boîtes en bois de taille standardisée qui étaient ensuite déchargées par grues sur des charrettes tirées par des chevaux.

Dans les années 1930, la Chambre internationale de commerce a mené des tests afin de standardiser les conteneurs en vue de faire baisser les coûts de transport dans le contexte de la grande dépression de 1929. En 1933, le Bureau international des conteneurs ou BIC est établi à Paris (et il s'y trouve toujours) pour gérer la standardisation de ces grosses boîtes métalliques dont le succès n'a fait que croître depuis.

Figure 1.1 – Conteneurs intermodaux

Aujourd'hui, on estime qu'il y a plus de 20 millions de conteneurs en circulation dans le monde.

1.1.2 Les avantages d'un transport par conteneur

Les avantages du transport par conteneur intermodal sont liés à ce que les Anglo-Saxons nomment *separation of concerns* (en français ségrégation des problèmes ou ségrégation des responsabilités).

Les transporteurs délèguent à leurs clients le remplissage du conteneur qui est ensuite scellé avant le transport. De ce fait, le transporteur n'a pas à s'occuper de la nature précise des biens transportés. Il doit s'assurer que le conteneur dans son ensemble arrive intact. Il assure donc la traçabilité d'un objet dont les caractéristiques physiques sont standardisées. Il peut les empiler, choisir le mode de transport adapté à la destination, optimiser les coûts et la logistique. Camions, bateaux, trains et grues vont pouvoir traiter ces biens, les arrimer correctement, qu'ils transportent des armes ou des couches-culottes.

La personne ou l'entreprise cliente a de son côté la garantie, pour peu que le conteneur ne soit pas perdu ou ouvert, que ses biens arriveront dans le même état et arrangement que lors du chargement. Le client n'a donc pas à se soucier de l'infrastructure de transport.

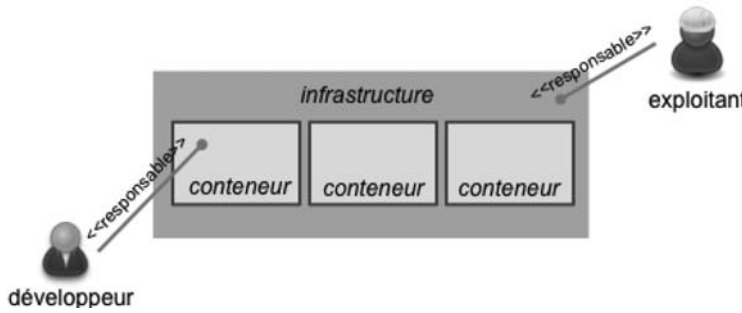
Cette séparation des responsabilités est à l'origine de la baisse drastique des coûts du transport de marchandises. Grâce à la standardisation et à l'automatisation qu'elle rend possible, les infrastructures de transport se sont mécanisées, automatisées et donc fiabilisées.

1.1.3 Extrapolation au monde logiciel

Comme nous l'avons évoqué en introduction, le monde informatique reste profondément organisé autour de l'aspect artisanal de la livraison du produit logiciel. Qu'il s'agisse d'un éditeur livrant un logiciel à un client ou de la mise en exploitation d'un développement, de très nombreux incidents survenant au cours du cycle de vie d'un logiciel sont liés à cette problématique.

C'est justement ce que la technologie des conteneurs logiciels cherche à résoudre.

Figure 1.2 – Séparation des responsabilités dans une architecture à base de conteneurs



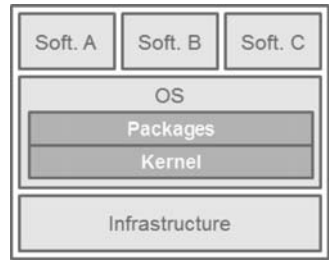
Dans une architecture à base de conteneurs :

- ✓ le contenu du conteneur, c'est-à-dire le code et ses dépendances (jusqu'au niveau de l'OS), est de la responsabilité du développeur ;
- ✓ la gestion du conteneur et les dépendances que celui-ci va entretenir avec l'infrastructure (soit le stockage, le réseau et la puissance de calcul) sont de la responsabilité de l'exploitant.

1.1.4 Les différences avec la virtualisation matérielle

Étudions l'empilement des couches dans le cas du déploiement de trois logiciels sur un même hôte (*host*) sans machine virtuelle ou conteneur.

Figure 1.3 – Installation native de trois logiciels sur un même hôte



© Dunod - Toute reproduction non autorisée est un délit.

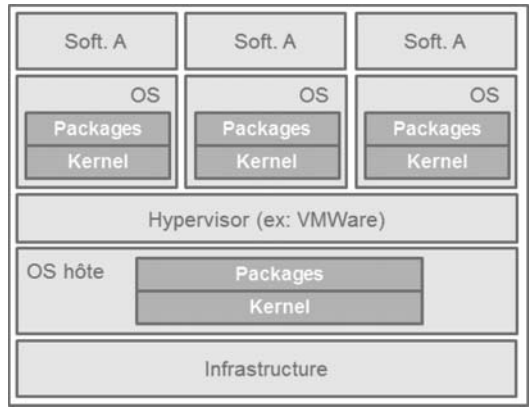
Dans ce type d'installation, on imagine que plusieurs situations problématiques peuvent survenir :

- ✓ les différents logiciels peuvent interagir entre eux s'ils n'ont pas été conçus par le même éditeur. Ils pourraient, par exemple, nécessiter des packages (bibliothèques, extensions) ou des versions de système d'exploitation différentes. Ils peuvent aussi ouvrir des ports réseaux identiques, accéder aux mêmes chemins sur le système de fichiers ou encore entrer en concurrence pour les ressources I/O ou CPU ;
- ✓ toute mise à jour de l'OS hôte va nécessairement impacter tous les logiciels qui tournent dessus ;
- ✓ chaque mise à jour d'un logiciel pourrait entraîner des impacts sur les autres.

L'expérience montre que l'exécution, sur le même système, de logiciels fournis par des éditeurs différents qui n'auraient pas testé cette cohabitation est très souvent problématique.

La virtualisation matérielle de type VMWare, VirtualBox ou HyperV n'assure-t-elle pas déjà cette séparation des responsabilités ?

Figure 1.4 – Virtualisation matérielle : trois VM sur le même hôte

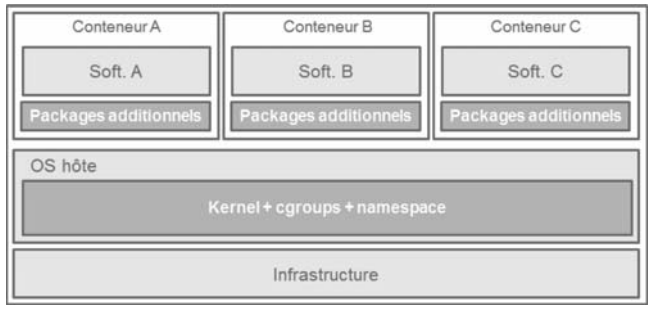


La réponse est « oui, mais… ».

Dans les faits, comme nous le voyons sur la figure précédente, cette virtualisation matérielle offre un niveau d'isolation élevé. Chaque logiciel se trouve dans son bac à sable (*sandbox* en anglais). Les problèmes évoqués précédemment sont donc résolus, mais d'autres apparaissent :

- ✓ le poids d'une machine virtuelle est tout d'abord très important. Une machine virtuelle est une machine et, même avec un système d'exploitation minimal, un OS moderne consommera difficilement moins de quelques Go de mémoire. La distribution de ce type de package demandera une bande passante réseau conséquente ;
- ✓ la machine virtuelle embarque trop d'éléments. Elle ne laisse pas le choix à l'exploitant (selon la manière dont elle aura été configurée) de choisir librement ses caractéristiques, comme le type de stockage, le nombre de CPU utilisés, la configuration réseau. Évidemment, les solutions de gestion d'environnements virtualisés (par exemple, vCenter de VMWare) offrent des solutions, mais celles-ci ont presque toujours des impacts sur le logiciel. Ce dernier ne peut pas être conçu sans savoir comment il va être exécuté.

Figure 1.5 – Architecture à base de conteneurs



Une architecture à base de conteneurs offre une solution de compromis. Le conteneur offre l'isolation permettant à un développeur d'embarquer l'ensemble des dépendances logicielles dont il a besoin (y compris les dépendances de niveau OS). De plus, un conteneur s'appuie sur le noyau (*kernel*) du système d'exploitation hôte[1]. Il est donc très léger et démarre presque aussi vite que le processus qu'il encapsule. Le nombre de conteneurs qu'un même hôte peut exécuter est donc nettement plus élevé que son équivalent en machines virtuelles.

Des conteneurs dans une VM

Qu'en est-il du risque d'impact sur le logiciel en cas de mise à jour du système d'exploitation hôte ? Effectivement, dans le cas d'une architecture à base de conteneurs, si le noyau de l'OS est mis à jour, il peut affecter les conteneurs qu'il exécute. Rien n'empêche donc de revenir dans ce type de cas à l'option VM (*Virtual Machine*), qui va augmenter l'isolation tout en conservant la possibilité d'exécuter plusieurs conteneurs dans la même VM. C'est d'ailleurs ce type d'option que Microsoft met en avant avec les conteneurs HyperV, tout en conservant aussi la possibilité de faire tourner des conteneurs nativement sur l'OS hôte. C'est également l'approche du projet Kata Containers[2] ou du projet HyperKit[3].

Il existe par ailleurs des projets visant à offrir des OS hôtes minimaux dédiés à l'exécution de conteneurs et se focalisant uniquement sur les problématiques d'infrastructure. L'usage de ces OS hôtes spécialisés va limiter leur exposition aux attaques et aux bugs pouvant nécessiter des mises à jour. Nous en verrons quelques exemples plus loin dans ce chapitre.

Cette idée de simplifier au maximum le système d'exploitation en éliminant les services non utilisés est aussi appliquée dans une technologie techniquement différente des conteneurs mais cherchant à atteindre les mêmes objectifs : Unikernel. Le lecteur intéressé pourra notamment regarder le projet https://github.com/solo-io/unik, initialement développé par EMC avant d'être confié à la communauté.

1. Sur la figure 1.5, le lecteur attentif aura noté les termes « cgroups » et « namespaces ». Nous les définirons dans le paragraphe suivant. À ce stade, nous dirons qu'il s'agit d'extensions du noyau Linux qui rendent possible la réalisation de conteneurs « isolés » les uns des autres.

2. https://katacontainers.io

3. https://github.com/moby/hyperkit

© Dunod - Toute reproduction non autorisée est un délit.

Nous verrons aussi plus tard que Docker apporte deux autres avantages aux solutions à base de conteneurs : la possibilité de décrire formellement comment construire le conteneur et un format d'image standardisé. Cette capacité fondamentale est probablement la clé du succès de Docker.

Maintenant que nous sommes convaincus des avantages des architectures à base de conteneurs, nous allons en étudier les fondements techniques.

1. 2 LES FONDATIONS : LINUX, CGROUPS ET NAMESPACES

Docker est une solution open source de conteneurs Linux qui s'appuie elle-même sur d'autres composants eux aussi ouverts. Ces briques de base sont en fait communes à tous les types de conteneurs Linux.

Initialement, Docker utilisait notamment LXC (*Linux Containers*) comme implémentation (on parle de *driver*), mais a ensuite développé sa propre bibliothèque de bas niveau nommée libcontainer[1] pour enfin migrer vers runC, le standard de l'OCI (Open Container Initiative), dont nous parlerons plus tard. Ce composant encapsule les fonctionnalités fondamentales, proches du noyau du système d'exploitation, dont la combinaison permet la virtualisation de niveau OS. Aujourd'hui le moteur Docker est construit au-dessus de containerd[2] qui lui-même intègre runC. Pas d'inquiétude néanmoins, la maîtrise de cet empilement de projets et de composants n'est pas fondamentalement utile pour tirer parti de Docker.

Nous allons étudier les différents composants historiques et fondamentaux sur lesquels s'est construite la technologie des conteneurs pour :

- ✓ en comprendre le rôle fonctionnel ;
- ✓ comprendre les différentes étapes qui ont abouti au concept de conteneur.

1.2.1 Docker = Linux

Il est en premier lieu essentiel de rappeler que Docker est initialement une technologie liée à Linux, son noyau (*Linux Kernel*) et certains de ses services orchestrés par containerd.

Comment Docker marche-t-il sous Windows ou Mac OS X ?

Eh bien, initialement il ne marchait pas réellement sous ces OS, du moins pas nativement.

Pour faire tourner Docker sous Windows, il était, encore jusqu'à récemment, nécessaire d'y installer un OS Linux. La solution usuellement pratiquée consistait à s'appuyer sur un logiciel de virtualisation matérielle : VirtualBox[3] dans le cas de Docker (Docker Toolbox).

---

1. Notons que ce driver d'exécution est ce qui fait le pont entre le moteur d'exécution des conteneurs et le système d'exploitation. Dans le cas de l'implémentation native sous Windows, c'est ce composant que les équipes de Microsoft ont dû réécrire pour l'adapter à Windows.

2. https://containerd.io/

3. https://www.virtualbox.org/

VirtualBox est un hyperviseur open source (disponible en licence GPL) aujourd'hui géré par Oracle. Il est donc très fréquemment utilisé par des solutions open source souhaitant disposer de fonctionnalités de virtualisation sans contraintes de licences.

L'installation n'était donc pas native et la complexité de gestion des terminaux, comme les performances d'accès au système de fichiers, limitait très fortement l'intérêt de Docker sous ces OS Desktop.

Mais les choses ont changé en 2016.

1.2.2 Docker = Linux + Windows + Mac

En 2016, Microsoft et Docker ont annoncé que la version éponyme de Windows Server inclurait la possibilité d'exécuter des conteneurs nativement. Les équipes de Microsoft, avec le support des ingénieurs de Docker, ont pour ce faire porté runC en s'appuyant sur les capacités natives de Windows en lieu et place de Linux. Cette fonctionnalité a ensuite été étendue à Windows 10 desktop pro 64 bits avec pour objectif de cibler les développeurs.

Microsoft propose aussi des images de base pour ces conteneurs « natifs » Windows (Windows Server Core et la très compacte Windows Nano Server).

Question brûlante : est-il possible d'exécuter un conteneur Linux sous Windows ?

Oui mais via un « truc » déjà évoqué dans ce chapitre : en encapsulant le conteneur Linux dans une VM Hyper-V exécutant un Linux sous Windows. Le Linux utilisé par Microsoft est nommé LinuxKit LCOW (pour Linux Containers On Windows).

Il est donc possible, sous Windows, d'exécuter deux types de conteneurs différents :

- ✓ des conteneurs Windows « natifs » (utilisant des images Windows) et tournant directement sur l'OS Windows (et apparaissant donc comme des processus Windows) ;
- ✓ des conteneurs Windows ou Linux s'exécutant dans des « mini-VM » Hyper-V contenant des versions « simplifiées » de Linux et Windows.

Sous Mac, l'approche a été similaire à cette dernière option. Docker pour Mac utilise une VM HyperKit très légère qui joue le rôle d'Hyper-V pour Windows.

Bien que tout ceci puisse sembler un peu compliqué, nous verrons, dans le chapitre 3, que ces différences restent pour l'essentielle transparentes pour l'utilisateur de Docker.

Il faut néanmoins noter que Linux reste le citoyen de première classe du monde Docker. Que ce soit sous Windows ou MacOS, le moteur Docker sert souvent à exécuter des conteneurs Linux.

Pour comprendre comment fonctionnent les conteneurs, il est d'ailleurs utile de décrire les services Linux historiques sur lesquels se sont appuyées les premières implémentations (LXC par exemple). Docker (plus exactement runC) est basé sur deux extensions originales du kernel Linux :

- ✓ CGroups ;
- ✓ Namespaces.

© Dunod - Toute reproduction non autorisée est un délit.

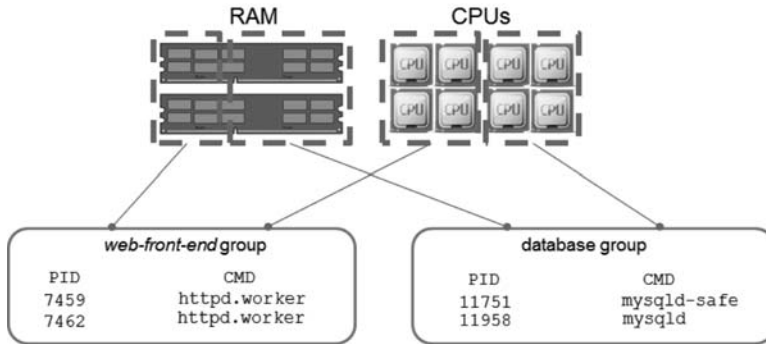
1.2.3 CGroups

CGroups[1] (pour *Control Groups*) permet de partitionner les ressources d'un hôte (processeur, mémoire, accès au réseau ou à d'autres terminaux). L'objectif est de contrôler la consommation de ces ressources par processus.

On doit cette bibliothèque aux ingénieurs de Google, qui a été probablement la première entreprise commerciale à utiliser des conteneurs, bien avant l'apparition de Docker.

Prenons par exemple une machine sur laquelle serait installée une application web avec un front-end PHP et une base de données MySQL. CGroups permettrait de répartir la puissance de calcul (CPU) et la mémoire disponible entre les différents processus, afin de garantir une bonne répartition de la charge (et donc probablement des temps de réponse).

Figure 1.6 – Répartition de ressources grâce à CGroups



CGroups introduit une notion de contrôleur qui, comme son nom l'indique, a pour objectif de contrôler l'accès à une ressource pour une hiérarchie de processus. En effet, par défaut, les processus fils d'un processus associé à un groupe donné héritent des paramètres de leur parent.

1.2.4 Namespaces

Les Namespaces sont indépendants de CGroups, mais fonctionnent de concert. Ils permettent de faire en sorte que des processus ne voient pas les ressources utilisées par d'autres. Si CGroups gère la distribution des ressources, Namespaces apporte l'isolation nécessaire à la création de conteneurs.

L'ancêtre du mécanisme des Namespaces est la commande `chroot`, qui existe au sein du système Unix depuis 1979 !

La fonction de cette commande est de changer pour un processus donné le répertoire racine du système. Le processus en question a l'impression d'être à la racine du système. L'objectif étant, par exemple, pour des raisons de sécurité, d'empêcher l'utilisateur de se promener dans des sections du système de fichiers dont il n'aurait a priori pas l'usage. Il s'agit donc conceptuellement d'une sorte de virtualisation.

Namespaces étend ce concept à d'autres ressources.

1. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

Tableau 1.1 – Namespaces Linux

Namespace	Isole
IPC	Communication interprocessus
Network	Terminaux réseau, ports, etc.
Mount	Point de montage (système de fichiers)
PID	Identifiant de processus
User	Utilisateurs et groupes
UTS	Nom de domaines

1.2.5 Qu'est-ce qu'un conteneur finalement ?

Un conteneur est tout simplement un système de fichiers sur lequel s'exécutent des processus (de préférence un par conteneur) de manière :

- ✓ contrainte : grâce à CGroups qui spécifie les limites en termes de ressources ;
- ✓ isolée : grâce notamment à Namespaces qui fait en sorte que les conteneurs ne se voient pas les uns les autres.

1.3 LES APPORTS DE DOCKER : STRUCTURE EN COUCHES, IMAGES, VOLUMES ET REGISTRY

Comme nous l'avons précédemment expliqué, les conteneurs existent depuis longtemps, mais Docker a apporté des nouveautés décisives qui ont clairement favorisé leur popularisation.

L'une d'entre elle a trait à la manière dont Docker optimise la taille des conteneurs en permettant une mutualisation des données. C'est ce que nous allons expliquer dans cette section.

Nous allons aussi aborder la question de la persistance des données dans un conteneur, en expliquant la notion de volume.

Notez que nous expérimenterons ces concepts ultérieurement, à l'aide des outils Docker.

> Attention : l'apport du projet Docker à la technologie des conteneurs ne se limite pas aux concepts décrits dans ce paragraphe. Nous verrons par la suite, dans la seconde partie de cet ouvrage, que Docker a aussi changé la donne grâce à ses outils et en particulier grâce aux fameux Dockerfiles.

1.3.1 La notion d'image

La notion d'image dans le monde conteneur est pratiquement synonyme de Docker. Comme nous l'avons déjà vu, un conteneur est un ensemble de fichiers sur lequel s'exécutent un ou plusieurs processus. On peut alors se poser la question suivante : mais d'où viennent ces fichiers ?

Construire un conteneur pourrait se faire à la main, en reconstruisant dans une partie du système de fichiers Linux une arborescence de fichiers : par exemple, un répertoire

© Dunod - Toute reproduction non autorisée est un délit.

pourrait stocker les fichiers spécifiques au conteneur qui ne font pas partie du noyau de l'OS. Mais cette pratique serait fastidieuse et poserait le problème de la distribution et de la réutilisation.

L'un des apports essentiels de Docker est d'avoir proposé une manière de conditionner le contenu d'un conteneur en blocs réutilisables et échangeables : les images.

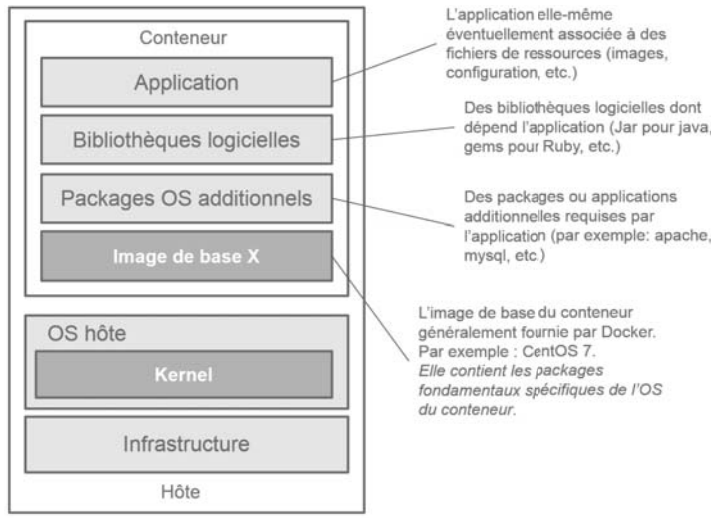
Ces images sont donc des archives qui peuvent être échangées entre plusieurs hôtes, mais aussi être réutilisées. Cette réutilisation est rendue possible par une seconde innovation : l'organisation en couches.

1.3.2 Organisation en couches : *union file system*

Les conteneurs Docker sont en fait des millefeuilles constitués de l'empilement ordonné d'images. Chaque couche surcharge la précédente en y apportant éventuellement des ajouts et des modifications.

En s'appuyant sur un type de système de fichiers Linux un peu particulier nommé *union file system*[1], ces différentes couches s'agrègent en un tout cohérent : le conteneur ou une image elle-même réutilisable.

Figure 1.7 – Structure d'un conteneur au sein d'un hôte



Admettons que nous construisions une application s'appuyant sur deux conteneurs tournant sur le même hôte :

- ✓ un front-end basé sur une application JSP, tournant dans un serveur d'applications Tomcat[2], exécuté par une machine virtuelle Java 1.8 sur un OS CentOS 7 ;
- ✓ un back-end constitué d'une base de données Cassandra[3] s'exécutant elle aussi sur Java 1.8 et CentOS 7.

1. Nous verrons dans la suite de ce chapitre qu'il existe plusieurs implémentations de ce type de système de fichiers.
2. http://tomcat.apache.org/
3. http://cassandra.apache.org/