Série modulaire sur les

Systèmes distribués

Fascicule 2:

Les conteneurs : Théorie

Frédéric Mailhot

Université de Sherbrooke

Mai 2022

Note : En vue d'alléger le texte, le masculin est utilisé pour désigner les femmes et les hommes.

Document fascicule02.pdf
Version 1.0, Mai 2022

Département de génie électrique et de génie informatique, Université de Sherbrooke.
Réalisé à l'aide de LaTeX et TeXstudio.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction aux conteneurs

Depuis 2013, la technologie des conteneurs logiciels a révolutionné le développement, le déploiement et la maintenance des systèmes informatiques. Les conteneurs sont maintenant utilisés dans une multitude de contextes où des applications distribuées disponibles à grande échelle sont nécessaires. Cette technologie permet de déployer et de gérer facilement des quantités d'applications qui, tout en étant indépendantes les unes des autres, peuvent interagir, être démarrées et s'exécuter rapidement et en utilisant une quantité de mémoire minimale. Dans ce chapitre, nous étudierons les structures et éléments précurseurs ayant permis l'émergence de ce type de système.

Les conteneurs tels qu'on les connaît aujourd'hui ont été développés à partir de 2008 par l'ingénieur franco-américain Solomon Hykes [DC13], initialement pour son entreprise dotCloud. Par la suite, en 2013, la technologie sous-jacente a été distribuée sous licence *open source*, supportée commercialement par l'entreprise Docker Inc. Depuis 2015, le projet Open Container Initiative (OCI) est responsable de l'élaboration de standards ouverts visant le fonctionnement et l'interopérabilité des technologies permettant l'utilisation de conteneurs [Fou15].

La technologie des conteneurs a été élaborée à partir d'un ensemble d'éléments pré-existants sur le système d'exploitation Linux. Initialement, il n'était possible d'utiliser le tout qu'avec Linux (ou dans des machines virtuelles Linux), mais il existe maintenant des versions spécifiques pour Windows (qui utilisent le Windows Subsystem for Linux (WSL)) ou pour macOS.

Les éléments importants pour le fonctionnement des conteneurs sont:

- l'isolation de l'accès à une partie du système:
  - fichiers (historiquement, avec la commande *chroot*)
  - fichiers, réseau, etc. (maintenant avec les *namespaces* et les *cgroups*)

- l'isolation de la mémoire accessible par un processus ou un ensemble de processus (systèmes *cgroup* et *LXC*, basé sur les *cgroups*)

- l'utilisation d'un système de fichiers en couches (de type *union*, tels *OverlayFS* ou *Btrfs*)

Dans ce qui suit, nous étudierons un plus en détail chacun de ces éléments, ainsi que l'évolution qui a mené à ces derniers.

1.1 Historique - les éléments précurseurs

1.1.1 Les débuts de l'histoire

Dès 1945 et le premier ordinateur (ENIAC, ou Electronic Numerical Integrator and Computer [Bur47, BB81]), l'idée de généraliser l'utilisation du système pour en faciliter l'exploitation était présente. Au départ, l'emploi du ENIAC était limité à un seul usager à la fois. Rapidement, il a été question d'éliminer cette contrainte, pour optimiser le taux d'utilisation du matériel. En 1954, John Backus du MIT a proposé l'idée de permettre à plusieurs usagers d'utiliser simultanément un même ordinateur, par ce qu'il a appelé le temps partagé (*time-sharing*) [Bac54]. En 1961, le *Compatible Time-Sharing System* (*CTSS*) a été proposé par le MIT comme premier système d'exploitation permettant le temps partagé.

En 1964, le développement du système *MULTICS* (*Multiplexed Information and Computing Service*), un effort de développement d'un système d'exploitation à temps partagé, a été démarré conjointement par le MIT, l'entreprise General Electric et le laboratoire de recherche Bell Labs [Rit80]. Très novateur, MULTICS a introduit de nombreux concepts permettant d'isoler les uns des autres les processus en exécution. Malgré les innovations importantes introduites dans ce système, il était complexe, avec un code source volumineux, et n'eut pas énormément de succès. Ayant contribué au développement de MULTICS, deux programmeurs des Bell Labs, Ken Thompson et Dennis Ritchie, décidèrent d'implémenter un nouveau système d'exploitation beaucoup plus simple, mais intégrant certains des éléments novateurs des systèmes CTSS et MULTICS. Leur objectif était de produire rapidement un système peu exigeant en mémoire et s'exécutant facilement sur les ordinateurs disponibles aux Bell Labs (au départ, un PDP-7 de Digital Equipment[Rit80]). Le système *UNICS* (un clin d'oeil au système MULTICS), rapidement renommé *UNIX*, était né [RT74]. Ce système d'exploitation relativement compact a proposé une vue unifiée de la mémoire et du système de fichiers, les processus pouvant utiliser un ensemble de services offerts par le noyau (*kernel*) et permettant la communication entre eux et avec le système.

1.1.2 Les débuts de la virtualisation

Le développement des systèmes d'exploitation *temps partagé* a certainement facilité l'utilisation des ordinateurs par de multiples usagers simultanément. Cependant, il est ensuite apparu clairement qu'il serait intéressant que les usagers puissent utiliser des ordinateurs distincts, ou à tout le moins qu'ils puissent utiliser les systèmes ordinés monolithiques comme s'il s'agissait de multiples ordinateurs indépendants. En 1964, alors que chez IBM le Système 360, modèle 67 était en développement [PJP91, Bro09], les systèmes d'exploitation CP-40, puis CP-67 furent développés pour produire des *Conversational Monitor Systems (CMS)*, permettant à des usagers distincts d'utiliser des systèmes d'exploitation indépendants. Ces systèmes furent ainsi les premiers *hypervisors*, avec des espaces mémoire virtualisés et indépendants pour chaque usager. Les ordinateurs IBM Système 360-67 n'ont jamais été commercialisés, mais les technologies sous-jacentes sont apparues au début des années 70 dans les ordinateurs IBM Système 370-145.

La virtualisation est alors devenue très populaire. Elle reposait sur des infrastructures matérielles spécifiquement développées à cet effet (en particulier, un support matériel pour la mémoire virtuelle [SN05]). Les coûts associés aux ordinateurs "sérieux" étaient particulièrement élevés à cette époque [1]. En conséquence, l'idée de permettre à de nombreux usagers l'utilisation simultanée des systèmes ordinés était très intéressante. Cependant, avec la venue de microprocesseurs relativement peu dispendieux [2], la virtualisation a alors perdu beaucoup de son attrait. En effet, plutôt que d'investir dans un ordinateur coûteux partagé entre de nombreux usagers, il est devenu plus attrayant d'utiliser un grand nombre de systèmes à capacités plus limitées, mais à un coût total significativement plus faible.

1.1.3 La virtualisation sur microprocesseur

À partir de la fin des années 90, la puissance de calcul des microprocesseurs est devenue suffisante pour penser de nouveau à la virtualisation. À ce moment, il est apparu qu'on pouvait maintenant faire de façon logicielle l'équivalent de ce qui était fait auparavant avec un support matériel dédié [Blo20].

Pour vraiment produire un ordinateur virtualisé sur microprocesseur, il est nécessaire d'émuler de façon logicielle l'ensemble des éléments matériels du système. Dans les années 1990, Mendel Rosenblum, jeune professeur à l'Université Stanford, a travaillé au développement de SimOS, un simulateur de systèmes

[1] En 1970, les coûts d'un *mainframe* allaient jusqu'à 1,8 millions de dollars pour un ordinateur IBM Système 370-145 [Arc70], et jusqu'à 12 millions de dollars pour un ordinateur IBM Système 360-195 [Arc69] soit l'équivalent de 12 à 80 millions de dollars en 2022 [Inf22]

[2] En 1977, un Apple II se vendait 1 300 dollars [Com18], soit l'équivalent d'environ 6 000 dollars en 2022 [Inf22]

ordinés [RHWG95]. En 1998, il a co-fondé l'entreprise VMware, dont l'objectif était de développer un émulateur logiciel complet fonctionnant sur l'architecture x86 [Yag04, AA06]. Le premier produit de l'entreprise, VMware Workstation, est apparu en mai 1999. D'autres systèmes logiciels de virtualisation complète sont ensuite apparus, tels Xen (2003), KVM (Kernel-based Virtual Machine) pour Linux (2006) et VirtualBox (2007) [Blo20]. L'intérêt de ces divers systèmes de virtualisation est leur capacité à émuler parfaitement un ordinateur matériel. Il est alors possible de faire fonctionner un système complet dans l'un de ces environnements, qui se comporte comme s'il s'agissait d'un système matériel indépendant. On retrouve ainsi la fonctionnalité des systèmes de virtualisation des années 70, mais en utilisant des microprocesseurs modernes (et peu coûteux) plutôt que des *mainframes* beaucoup plus onéreux. Avec le succès des systèmes de virtualisation VMware (et autres), les concepteurs de microprocesseurs Intel et AMD ont commencé à supporter la virtualisation à l'aide de matériel et d'instructions dédiées, en 2005 et 2006 respectivement [Lo07, FO06, Cor07e, AMD09].

1.1.4 Les éléments précurseurs à l'élaboration de conteneurs

En parallèle avec la virtualisation purement matérielle des années 60 et 70 et de la virtualisation logicielle "complète" du début des années 2000 (à la VMware, Xen, KVM, VirtualBox, etc.), une autre avenue a aussi été explorée: utiliser les ressources d'un système d'exploitation unique, mais où certains groupes de processus ont des permissions restreintes. De cette manière, il est inutile de dupliquer les ressources fournies par le système ordiné. Il suffit d'en protéger l'accès.

Assez tôt dans l'histoire du système d'exploitation Unix, certains éléments logiciels facilitant une forme simplifiée de virtualisation ont été développés. Ces ajouts ont été faits en parallèle et de façon indépendante du support matériel utilisé pour la virtualisation telle qu'effectuée à ce moment.

1.1.4.1 La fonctionnalité *chroot*

L'appel système *chroot* a été introduit en 1978 dans la version 7 du système d'exploitation Unix. Cette première fonctionnalité logicielle a permis de modifier la racine du système de fichier (\ sous Unix) pour un ensemble de processus. Ainsi *chroot* a permis de démarrer certains processus de telle sorte qu'ils ne puissent connaître que la portion du système de fichier qui leur était assignée. Dans la pratique, la commande *chroot* a été utilisée de façon assez limitée, puisque ce système n'isolait que l'accès aux fichiers et laissait les processus accéder aux autres ressources de l'ordinateur (réseau, ports d'entrées/sorties, autres disques, mémoire, etc.).

Au début des années 90, l'équipe de recherche des Bell Labs travaillant sur le système d'exploitation *Plan 9* a proposé la généralisation du concept *chroot* pour supporter l'isolation des processus [PPT+92]. Cette idée a été reprise plus tard sous différentes formes, permettant de sécuriser les accès aux ressources d'un système d'exploitation par les processus qui s'y exécutent.

1.1.4.2 L'isolation des processus: *emprisonnement* (*jail*) et approches similaires

En 1998, cette idée d'isoler complètement un ensemble de processus est reprise par Poul-Henning Kamp pour le système FreeBSD [KW00, KW04, Kam16], en créant le mécanisme d'*emprisonnement* (*jail*). Ce système permet de restreindre l'accès de processus aux fichiers, autres processus et réseau qui leur sont accessibles. De plus, cette technologie permet un accès asymétrique aux ressources: les processus situés à l'extérieur de la *prison* peuvent observer ce qui s'y passe, mais à l'inverse ceux qui sont à l'intérieur ne voient rien d'autre. En plus d'un mécanisme de virtualisation, c'est aussi une technique qui permet aux administrateurs d'un système de vérifier que tout s'y déroule sans problème. Par la suite de nombreuses initatives ont suivi et proposé des solutions similaires au mécanisme de *jails* de FreeBSD [Hog14]: OpenVZ [Ope15], VServer de Jacques Gélinas [VSe01], et plusieurs autres.

1.1.4.3 Les fonctionnalités *namespaces* et *cgroups*

Au début des années 2000, les *namespaces* sont repris pour le système d'exploitation Linux, directement inspirés du système *Plan 9*. Au départ limités au système de fichier (à la *chroot*), on y ajoute progressivement plusieurs types de *namespaces* [Wik22, Ker13], pour:

- le système de fichiers (utilisé à partir de 2001 [Vir01]),
- les caractéristiques *UTS* (*Unix Time-Sharing*) pour les noms de domaines et de machines (2006 [Hal06]),
- les communications entre processus (2006 [Kor06]),
- les identificateurs de processus (2007 [PE07]),
- le réseau (2007 [Cor07c]),
- les usagers (2012 [Ker12]),
- le controle du système lui-même, avec les *cgroups* (2016 [Heo16, Tor16]),
- le temps (accès à l'horloge du système) (2020 [Lar20])

Au milieu des années 2000, deux développeurs de chez Google conçoivent les *container groups* [Men04] [Set06], qu'on appelle par la suite les *control groups* ou *cgroups* [Cor07c]. Ce système est intégré dans le noyau Linux depuis 2007 [Cor07d]. Il est repris et ré-écrit par Tejun Heo en 2016 [Heo16]. Il permet de gérer un ou plusieurs processus et de contrôler:

- l'utilisation de la mémoire [Cor07a, SS07],
- l'accès au système de fichiers [Heo14],
- la priorisation de l'exécution [Cor07b]
- l'accès aux entrées/sorties
- l'exécution des processus (et de les "geler" au besoin) [Han06]

Les *cgroups* et les *namespaces* sont deux piliers importants des conteneurs tels qu'on les connaît aujourd'hui [Pet15, Mac15, vK21]:

- les *cgroups* contrôlent et limitent l'utilisation des ressources du système par les processus
- les *namespaces* contrôlent et limitent ce qu'un processus peut "voir" du système

1.1.4.4 Les systèmes de fichiers à union

Les systèmes de fichiers organisent l'information persistante sous forme de répertoires et de fichiers, à l'aide d'une structure hiérarchique qui permet de rapidement retrouver l'information se trouvant sur un "disque" (support matériel persistant pour l'information). Le concept de fichiers et de répertoires existe depuis le début des années 60, étant apparu dans le système d'exploitation CTSS (Compatible Time-Sharing System) [CDD+63]. Cependant, ce n'est qu'au milieu des années 80, que l'idée d'ajouter une couche d'abstraction au-dessus du système de fichier fait son apparation, dans le système SunOS 2 et ses successeurs SunOS 3 et Solaris[Pat03], chez Sun Microsystems. Le *Virtual File System* (*VFS*) permet alors de découpler l'accès aux systèmes de persistance physique (tels les disques locaux et les tout nouveaux disques réseaux) de la logique d'accès à l'information, service du système d'exploitation offert aux processus s'y exécutant. En 2004, le développement du système *ZFS* débute chez Sun Microsystems, permettant une gestion efficace, unifiée et sécuritaire d'un ensemble de ressources d'entreposage de données [Wat09]. Sous Linux, plusieurs projets sont proposés, parfois en parallèle, pour supporter les systèmes de fichiers à union (union file systems): Unionfs [Uni14], OverlayFS [Bro12b, Bro12a], Btrfs [Rod08].

L'intérêt des systèmes de fichiers à union est multiple. Ces systèmes permettent de modifier (en apparence) des fichiers qui ne sont accessibles qu'en lecture

seule. Ils ont été très utilisés pour intégrer des substrats qu'on ne peut écrire (tel les CD-ROMs) à un système où il est important de modifier certaines configuration, par exemple les *Live CDs*. Ces systèmes sont aussi particulièrement utiles pour les conteneurs, puisqu'ils permettent de partager (sans les modifier) un ensemble de fichiers. Ainsi, plusieurs conteneurs peuvent utiliser une certaine base commune de fichiers, chacun pouvant modifier cette base commune sans affecter les autres conteneurs [Hei21].

1.2 Historique - les premiers conteneurs

Les conteneurs tels qu'on les connait maintenant ont été proposés en 2013 par la compagnie Docker. Un organisme de standardisation, le *Open Container Initiative* (*OCI*) a rapidement vu le jour et est maintenant responsable de la définition des fonctionnalités de base ainsi que des méthodes à définir pour qu'un système de conteneurisation soit reconnu. Comme on l'a vu plus haut, les conteneurs sont basés sur des technologies qui existaient déjà auparavant. Ce qui rend la technologie de conteneurisation particulièrement intéressante est la façon dont ces technologies ont été imbriquées. De plus, un ensemble de fonctionnalités facilitant grandement l'utilisation de ces technologies a aussi été mis sur pieds. Enfin, le modèle proposé (images, conteneurs et orcherstration) représente de façon facilement compréhensible ce que l'ensemble représente.

1.2.1 Pourquoi les conteneurs, alors qu'il y avait déjà les machines virtuelles?

Il est intéressant de comprendre pourquoi la technologie des conteneurs a été proposée et en quoi elle diffère des systèmes de virtualisation (à la VMware, Xen, KVM, VirtualBox, Parallels, etc.). Les machines virtuelles qui sont apparues à la fin des années 90 et au début des années 2000 avaient pour objectif l'exécution efficace et rapide d'un système d'exploitation complet, à partir d'un système d'exploitation "hôte". Ces systèmes de virtualisation, qui ont maintenant accès à des instructions matérielles facilitant leur exécution, doivent être complets et indépendants des systèmes "hôtes". Par exemple, si un système "hôte" Windows exécute une machine virtuelle Linux, cette dernière doit contenir la totalité du système d'exploitation Linux, comme s'il s'exécutait directement sur un ordinateur physique. L'avantage de cette façon de procéder est que la machine virtuelle est entièrement indépendante du système hôte, qui peut alors être totalement différent (par exemple, une machine virtuelle Linux s'exécutant dans un système d'exploitation Windows). L'inconvénient est qu'il faut dupliquer en entier le système virtualisé, ce qui exige beaucoup d'espace disque, beaucoup de mémoire, et un partage des ressources matérielles qui ne sont utilisables que par un seul système d'exploitation à la fois.

L'idée derrière les conteneurs est d'exécuter directement les tâches désirées dans le système d'exploitation hôte. Cependant, pour la sécurité, l'intégrité et l'équité de l'exécution des tâches sur le système, il est nécessaire d'isoler ces tâches "désirées" du reste du système d'exploitation. L'utilisation conjointe des *namespaces*, des *cgroups* et de *systèmes de fichiers à union* permettent d'effectuer facilement cette isolation. L'intérêt des conteneurs repose sur l'intégration de ces trois technologies, de concert avec des fonctionnalités qui en facilitent l'usage et la compréhension (voir figure 1.1).

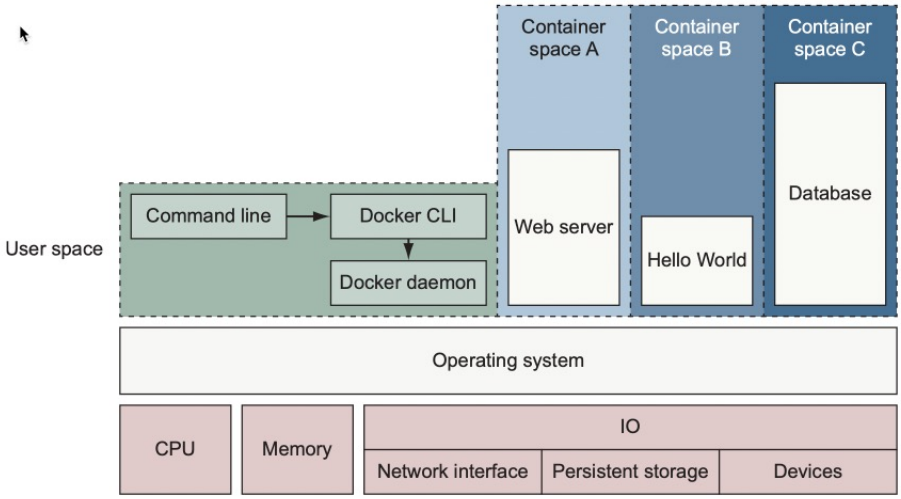

Figure 1.1: Exemple d'exécution de trois conteneurs simultanés (tiré de [NK19])

Puisque les conteneurs représentent des processus qui, quoiqu'isolés du reste du système, s'exécutent tout de même directement dans le système sous-jacent, les exigences en espace mémoire et en espace disque sont minimisés. De plus, puisqu'il s'agit de processus "natifs", leur démarrage est quasi instantanée, tout comme tout autre processus du système. La technologie des conteneurs est donc très différente de celle des machines virtuelles traditionnelles, où il est nécessaire de démarrer un système d'exploitation complet avant l'exécution des tâches requises [aqu22] (Voir figure 1.2).

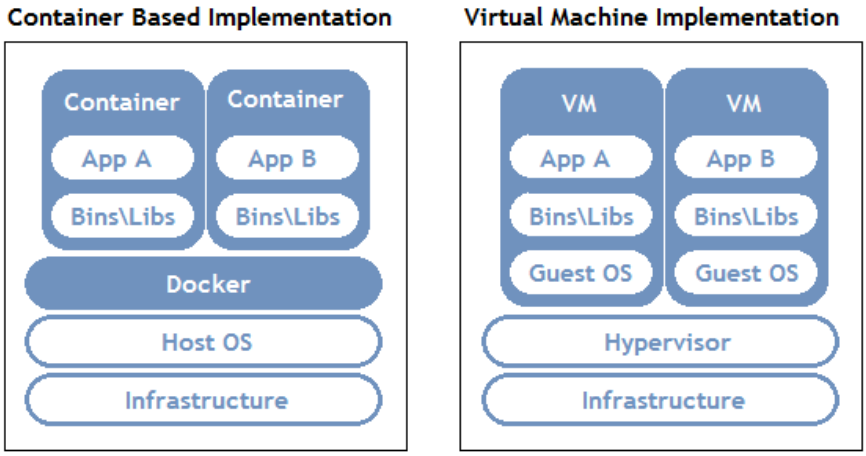

Figure 1.2: Différence entre conteneur et machine virtuelle (tiré de [aqu22])

1.2.2 Docker, la première implémentation des conteneurs modernes

Faisant suite aux systèmes développés pour l'entreprise dotCloud, le système de conteneurs tels qu'on les connaît maintenant a été proposé à l'origine en 2013 par la compagnie Docker Inc. et supportée par le projet en code ouvert (*open source*) Open Container Initiative (OCI). Tel que présenté plus haut, trois technologies déjà existantes sous Linux ont été combinées pour produire les premiers conteneurs:

- *namespaces*
- *cgroups*
- *union filesystems*

Ces éléments ont été intégrés ensemble, en y ajoutant un système de gestion à deux niveaux des *union filesystems*, les *images* (l'ensemble des répertoires et fichiers de niveaux inférieurs, qui ne sont accessibles qu'en lecture) et les *conteneurs (*containers*)* (le niveau supérieur de répertoires et de fichiers, dans

lequel il est possible d'écrire). De plus,dès le départ le système de conteneurs a automatisé l'utilisation des *namespaces* et des *cgroups* pour simplifier la gestion de l'accès au réseaux et de l'utilisation de la mémoire. Le modèle conceptuel proposé est associé à celui des conteneurs destinés au transport (maritime, ferroviaire, routier). Ainsi, un conteneur Docker offre un environnement protégé et isolé du reste du système. On y exécute un ou plusieurs processus, qui sont automatiquement encapsulés par le système. Pour permettre la personnalisation d'un conteneur, on y permet la modification de fichiers locaux au conteneur. Pour y arriver, on déploie automatiquement un *union file system* où la couche supérieure se trouve directement dans le conteneur, alors que les couches inférieures ne sont accessibles qu'en lecture seule. C'est ce qu'on nomme des *images*. Dans le modèle de conteneurisation proposé par Docker (et standardisé depuis), les images représentent un système de fichiers en couches qui ne peut être modifié. L'intérêt de cette façon de procéder est qu'une image peut être partagée entre de nombreux conteneurs, ce qui réduit significativement les besoins en espace disque et duplication. De plus, puisqu'un conteneur contient la couche supérieure (et modifiable) du système de fichiers complet, l'ensemble des conteneurs et des images qu'ils utilisent forment un tout cohérent. Il est donc possible de transformer un conteneur actif (avec sa couche de fichiers modifiables) en une nouvelle image, qui capture les modifications effectuées dans le conteneur et les rend accessible en lecture seule pour d'éventuels nouveaux conteneurs.

Pour simplifier encore plus l'utilisation des images et des conteneurs, la compagnie Docker a aussi offert une infrastructure de gestion et de persistance des images. Cette infrastructure peut être publique ou privée. Il existe des registres (*registry*) publics officiels, mais il est aussi possible de définir un registre privé local à une organisation, un groupe, etc. (voir figure 1.3).

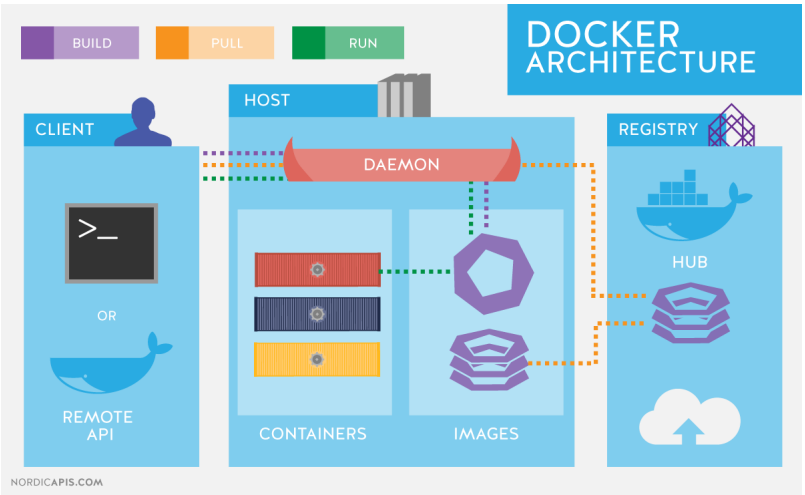

Figure 1.3: Architecture Docker standard (tiré de [aqu22])

Au départ, la technologie des conteneurs a été mise en oeuvre sur des systèmes Linux, puisque ce sont ces systèmes qui offraient les technologies *namespaces*, *cgroups* et *sytèmes de fichiers à union*. Assez rapidement cependant, la technologie des conteneurs a été adaptée pour les systèmes d'exploitation Windows et macOS. Il est intéressant de comprendre comment ces premières adaptations ont été réalisées, puisque les conteneurs sont simplement des tâches s'exécutant sur le système d'exploitation "natif" et que les conteneurs fonctionnaient sur des sytèmes Linux. En fait, les premières adaptations des conteneurs reposaient simplement sur leur exécution dans une machine virtuelle Linux. Au départ, pour les sytèmes d'exploitation Windows et macOS, on y démarrait tout d'abord une machine virtuelle Linux (en utilisant des technologies telles que VMware, ou autres), machine qui demeurait active en continu sur le système "hôte". L'utilisation de conteneurs était alors simplement déplacée dans ces machines virtuelles continuellement présentes, mais invisibles. Depuis, le système d'exploitation Windows a intégré directement une couche noyau compatible avec Linux, le Windows Subsystem for Linux (*WSL*) [Sta16].

Chapitre

2

Les images

Dans l'environnement de conteneurisation, les images représentent un système de fichiers à union (*union file system*) accessible en lecture seule. Les images comportent un ensemble d'une ou de plusieurs couches distinctes, représentant des modifications successives d'un système de fichiers sous-jacent original (voir 2.1).

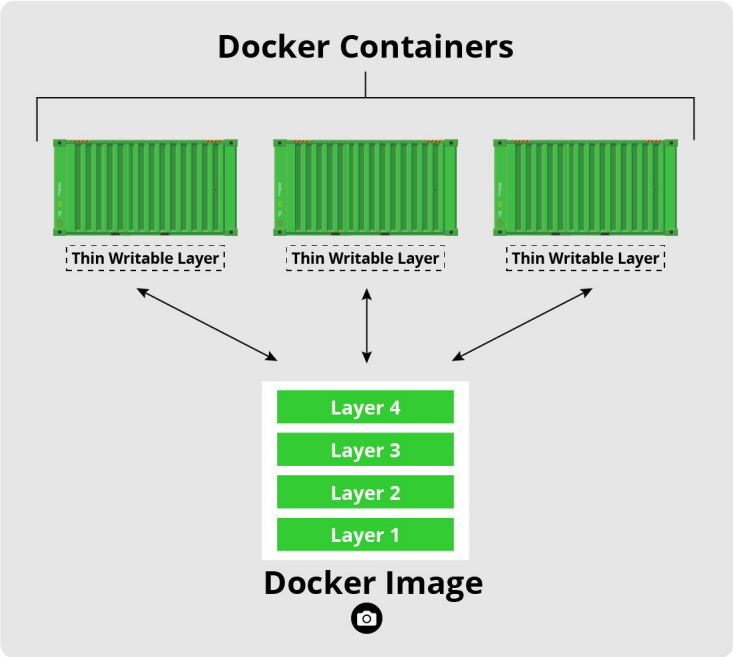

Figure 2.1: Architecture Docker standard (tiré de [Kis21])

Il est à noter que les modifications successives du système de fichiers sont conservées dans la structure même des images. Ceci implique qu'une image

représente en fait une succession de modifications de répertoires et de fichiers, eux-mêmes pouvant provenir d'autres images. Cette hiérarchie d'images est très utile à de nombreux égards. Ainsi, si une image sous-jacente est réutilisée à plusieurs reprises, son contenu n'a pas à être dupliqué sur le système hôte. Puisqu'une image n'est toujours accessible qu'en lecture seule, une même image sous-jacente peut être partagée entre de nombreuses images (ou des conteneurs) de niveaux supérieurs.

Puisque de nombreuses images distinctes peuvent exister, il est important de pouvoir les distinguer et reconnaître. Pour ce faire, une signature numérique (un *hash*) est calculée pour chaque image. Lorsqu'une image est bâtie au-dessus d'une image déjà existante, cette dernière est indiquée par sa signature numérique unique au niveau de l'image supérieure.

2.1 Exploration des images docker

Dans ce qui suit, nous étudierons les différents éléments qui constituent une image docker. En parallèle, nous passerons en revue les commandes utilisables pour observer ces éléments dans un système en fonction.

2.1.1 Les images locales

Le système docker conserve en mémoire l'ensemble des images qui ont été utilisées auparavant sur le système hôte. Nous procéderons en supposant qu'un nouveau système docker vient d'être installé. Tout d'abord, ajoutons une première image:

```
$Shell> docker image pull postgres
```

Vérifions que la nouvelle image est bien présente:

```
$Shell> docker images
```

ou

```
$Shell> docker image list
```

On obtient une sortie qui s'apparente à:

```
REPOSITORY                          TAG       IMAGE ID       CREATED         SIZE
postgres                            latest    4ec279110c4d   6 days ago      355MB
```

Regardons maintenant ce que contient cette image:

```
$Shell> docker image inspect postgres
```

On obtient alors une grande quantité d'informations, parmi lesquelles on trouve:

- ID: la signature numérique (*hash*) de cette image
- RepoTags: l'étiquette associée à cette image
- Created: date de création de l'image
- Architecture: type de matériel auquel cette image est destinée
- Os: type de système d'exploitation requis
- GraphDriver: information sur le système de fichiers à union utilisé:
  - LowerDir: images sous-jacentes (et en lecture seule), sous *.../docker/overlay2/...*
  - MergeDir: répertoire résultant de l'union, visible au niveau de cette image
  - UpperDir: répertoire utilisé pour les modifications (avant la création de l'image en lecture seule)
  - WorkDir: répertoire de travail, où s'effectuent les adaptations et unifications des systèmes sous-jacents
- RootFS et Layers: ensemble des images sous-jacentes utilisées (avec leurs signatures numériques)

2.1.2 Les registres d'images

Pour obtenir une image qu'on pourra ensuite utiliser dans un conteneur, on utilise typiquement un registre d'images (*registry*). Il existe des registres publics, mais il est aussi possible de créer un registre local à une organisation.

Lorsqu'on utilise *docker*, le registre utilisé par défaut pour trouver des images est le *Docker Hub*. On y trouve un très grand nombre d'images (plus de 240 milliards d'image en juillet 2020 [Sup21]), dont nombre de versions officielles de systèmes connus tels Postgresql, Apache, nginx, Ubuntu, Alpine, etc. On retrouve dans le registre officiel docker non seulement un ensemble d'images, mais aussi un ensemble de versions pour chacune des images. On peut toujours spécifier la version d'une image en utilisant une commande du type:

```
$Shell> docker image pull postgres:10
```

Par convention, on peut toujours obtenir la version la plus récente d'une image en utilisant l'étiquette *latest*. Par exemple:

```
$Shell> docker image pull postgres:latest
```

Par défaut, si aucune étiquette n'est utilisée, le système suppose qu'il s'agit de la dernière version de l'image demandée.

2.1.3 La gestion des images locales

Il peut arriver que le système local contienne un très grand nombre d'images, ce qui peut exiger un espace disque important. On peut alors épurer la liste des images locales à conserver, à l'aide de différentes commandes:

```
$Shell> docker image prune
```

Cette commande permet de retirer toutes les images transitoires créées temporairement mais maintenant inutilisées . On reconnait ces images par leur étiquette (*<none>*) dans la liste d'images locales.

```
$Shell> docker image prune -a
```

Cette commande retire toutes les images qui ne sont pas utilisées par un conteneur. Attention: si une image doit être utilisée ultérieurement par un conteneur, il faudra que le système la téléverse de nouveau, impliquant des délais et une certaine consommation de bande passante sur le réseau. Il est préférable de conserver les images utilisées fréquemment, mais si à un instant donné elles ne sont pas actives.

```
$Shell> docker rmi une_image
```

Cette commande permet de retirer l'image spécifiée (*une_image*). Si l'image est en cours d'utilisation par un conteneur, elle ne pourra être retirée jusqu'à ce que le conteneur soit lui-même éliminé.

Chapitre 3

Les conteneurs

Les conteneurs représentent ce qui est exécuté par le système de conteneurisation, par exemple *docker*. Ils représentent à la fois la couche active (dans laquelle il est possible d'écrire) du système de fichiers, l'ensemble de couches en lectures seules du système de fichiers (l'image utilisée par le conteneur), la configuration du réseau utilisée spécifiquement pour un groupe de processus encapsulés par le conteneur (ports "internes", ports "externes", addresse réseau, etc.), les liens (s'il y en a) avec le reste du système de fichiers du système hôte, etc.

Le système qui permet l'exécution des conteneurs est lui-même configurable et détermine, par exemple, le nom du utilisé, la quantité de mémoire allouée, le nombre de processeurs, la version du système de fichier à union (typiquement, *overlay2*, mais peut aussi être *Btrfs* ou *zfs*, ou autres).

Pour connaître la configuration du système d'exécution des conteneurs, on peut utiliser la commande suivante (en supposant qu'on utilise le système *docker*):

```
$Shell> docker info
```

3.1 Démarrage et utilisation d'un conteneur

Pour la suite, nous supposerons que le système *docker* est utilisé pour faire fonctionner les conteneurs.

Pour démarrer un conteneur, il suffit d'exécuter la commande *docker run* suivie du nom d'une image. Par exemple:

```
$Shell> docker run hello-world
```

Pour savoir quels conteneurs sont actifs, on peut utiliser les commandes:

```
$Shell> docker ps
$Shell> docker ps -a
```

La première *docker ps* imprime une liste des conteneurs qui sont en exécution, alors que la deuxième commande (*docker ps -a*) indique tous les conteneurs présents dans le système, incluant ceux qui sont arrêtés.

Pour arrêter un conteneur, on peut utiliser les commandes:

```
$Shell> docker stop nom_du_conteneur
$Shell> docker stop ID_du_conteneur
```

Cependant, l'arrêt n'implique pas l'élimination du conteneur du système (il est possible de le redémarrer dans certains cas). Pour éliminer complètement un conteneur du système, on doit, après avoir arrêter un conteneur, invoquer la commande:

```
$Shell> docker rm nom_du_conteneur
$Shell> docker rm ID_du_conteneur
```

Lorsqu'on démarre un conteneur, on peut indiquer qu'on désire démarrer un certain exécutable. En particulier, on peut demander le démarrage interactif d'un *shell*, ce qui nous donne accès (en ligne de commandes) à l'intérieur d'un conteneur. Par exemple, en utilisant l'image *alpine*, une version minimaliste du système d'exploitation *Ubuntu*, on peut obtenir un "terminal" qui s'y exécute de l'intérieur en donnant la commande:

```
$Shell> docker run --rm -it alpine sh
```

Bibliographie

[AA06] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proc. of the ASPLOS XII*, pages 2–13, San Jose, 2006. ACM.

[AMD09] AMD. AMD I/O virtualization technology (IOMMU) specification. Accessible en ligne, 2009. Voir: `https://web.archive.org/web/20110124134140/http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf`.

[aqu22] aqua. Docker architecture. Disponible sur cloud native wiki, 2022. Voir: `https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/`.

[Arc69] IBM Archive. System/360 Model 195. Accessible en ligne, 1969. Voir: `https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP2195.html`.

[Arc70] IBM Archive. System/370 Model 145. Accessible en ligne, 1970. Voir: `https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP3145.html`.

[Bac54] John Backus. *Digital Computers - Advanced Coding Techniques*, pages 16–2. MIT Press, 1954. Voir: `https://web.archive.org/web/20180929021555/http://www.bitsavers.org/pdf/mit/summer_session_1954/Digital_Computers_Advanced_Coding_Techniques_Summer_1954.pdf`.

[BB81] Arthur Burke and Alice R. Burke. The ENIAC: The first general-purpose electronic computer. *Annals of the History of Computing*, 3:4:310–389, 1981. Voir: `https://ieeexplore.ieee.org/document/4640745`.

[Blo20] The Dion Research Blog. Data infrastructures for the rest of us - III - softwaree. Accessible en ligne, 2020. Voir: `https://blog.dionresearch.com/2020/05/data-infrastructures-for-rest-of-us-iii.html/`.

[Bro09] Josh Brodkin. With long history of virtualization behind it, IBM looks to the future. Accessible en ligne, 2009. Voir: https://www.networkworld.com/article/2254433/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html.

[Bro12a] Neil Brown. Document the overlay filesystem. Accessible en ligne, 2012. Voir: https://www.uwsg.indiana.edu/hypermail/linux/kernel/1209.2/02574.html.

[Bro12b] Neil Brown. Overlay filesystem. Accessible en ligne, 2012. Voir: https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[Bur47] Arthur Burke. Electronic computing circuits of the ENIAC. *Proceedings of the I.R.E.*, 35:8:756–767, 1947. Voir: https://ieeexplore.ieee.org/document/1697434.

[CDD+63] F.J. Corbato, M. M. Daggett, R. C. Daley, R. J. Creasy, J. D. Hellwig, R. H. Orenstein, and L. K. Korn. *The Evolution of the Unix Time-sharing System, in Language Design and Programming Methodology*, pages 23–25. M.I.T. Press, 1963. Voir: http://people.csail.mit.edu/saltzer/CTSS/CTSS-Documents/CTSS_ProgrammersGuide_1963.pdf.

[Com18] Evan Comen. Check out how much a computer cost the year you were born. Accessible en ligne, 2018. Voir: https://www.usatoday.com/story/tech/2018/06/22/cost-of-a-computer-the-year-you-were-born/36156373/.

[Cor07a] Jonathan Corbet. Controlling memory use in containers. Disponible en ligne, 2007. Voir: https://lwn.net/Articles/243795/.

[Cor07b] Jonathan Corbet. Kernel space: Fair user scheduling for Linux. LinuxWorld, 2007. Voir: http://www.networkworld.com/news/2007/101207-kernel.html.

[Cor07c] Jonathan Corbet. Network namespaces. Disponible en ligne, 2007. Voir: https://lwn.net/Articles/219794/.

[Cor07d] Jonathan Corbet. Notes from a container. Disponible en ligne, 2007. Voir: https://lwn.net/Articles/256389/.

[Cor07e] Intel Corporation. Intel virtualization technology. Accessible en ligne, 2007. Voir: https://web.archive.org/web/20150521014611/https://software.intel.com/sites/default/files/m/0/2/1/b/b/1024-Virtualization.pdf.

[DC13] Julien Dupont-Calbo. Solomon Hykes: un frenchy au "Y Combinator". Accessible en ligne, 2013. Voir: https://www.lemonde.fr/economie/article/2013/04/08/un-frenchy-au-y-combinator_3155747_3234.html#6o4IpY8ri7Id2kLR.99.

[FO06] John Fisher-Ogden. Hardware support for efficient virtualization. Accessible en ligne, 2006. Voir: https://cseweb.ucsd.edu//~jfisherogden/hardwareVirt.pdf.

[Fou15] The Linux Foundation. Open container initiative. Accessible en ligne, 2015. Voir: https://opencontainers.org/.

[Hal06] Serge E. Hallyn. UTS namespaces: Introduction. Disponible en ligne, 2006. Voir: https://lwn.net/Articles/179345/.

[Han06] Dave Hansen. Resource management. Disponible en ligne, 2006. Voir: https://events.static.linuxfound.org/slides/lfcs09_hansen2.pdf.

[Hei21] Martin Heinz. Deep dive into docker internals - union filesystem. Disponible en ligne, 2021. Voir: https://martinheinz.dev/blog/44.

[Heo14] Tejun Heo. kernfs, sysfs, driver-core: implement synchronous self-removal. Accessible en ligne, 2014. Voir: https://lwn.net/Articles/584019/.

[Heo16] Tejun Heo. [GIT PULL] cgroup namespace support for v4.6-rc1. Accessible en ligne, 2016. Voir: https://lkml.org/lkml/2016/3/18/564.

[Hog14] Scott Hogg. Software containers: Used more frequently than most realize. Accessible en ligne, 2014. Voir: https://www.networkworld.com/article/2226996/software-containers--used-more-frequently-than-most-realize.html.

[Inf22] InflationTool. Inflation calculator. Accessible en ligne, 2022. Voir: https://www.inflationtool.com/.

[Kam16] Poul-Henning Kamp. Jails - High value but shitty Virtualization. Accessible en ligne, 2016. Voir: http://phk.freebsd.dk/sagas/jails/.

[Ker12] Michael Kerrisk. User namespaces progress. Disponible en ligne, 2012. Voir: https://lwn.net/Articles/528078/.

[Ker13] Michael Kerrisk. Namespaces in operation, part 1: namespaces overview. Disponible en ligne, 2013. Voir: https://lwn.net/Articles/531114/.

[Kis21] Edward Kisller. A beginners guide to understanding and building docker images. Accessible en ligne, 2021. Voir: https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/.

[Kor06] Kirill Korotaev. IPC namespaces. Disponible en ligne, 2006. Voir: https://lwn.net/Articles/187274/.

[KW00] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proc. of the SANE 2000*, page 15, Maastricht, 2000. SANE (System Administration and Networking). Voir: http://www.sane.nl/events/sane2000/papers/kamp.pdf.

[KW04] Poul-Henning Kamp and Robert N. M. Watson. Building systems to be shared securely. *ACMqueue*, 2:5:42–51, 2004. Voir: https://queue.acm.org/detail.cfm?id=1017001.

[Lar20] Michael Larabel. It's finally time: The time namespace support has been added to the Linux 5.6 kernel. Disponible en ligne, 2020. Voir: https://www.phoronix.com/scan.php?page=news_item&px=Time-Namespace-In-Linux-5.6.

[Lo07] Jack Lo. VMware and CPU virtualization technology. Accessible en ligne, 2007. Voir: https://web.archive.org/web/20110717231306/http://download3.vmware.com/vmworld/2005/pac346.pdf.

[Mac15] Duncan Macrae. How linux kernel cgroups and namespaces made modern containers possible. Accessible en ligne, 2015. Voir: https://www.silicon.co.uk/software/open-source/linux-kernel-cgroups-namespaces-containers-186240?inf_by=59f18086681db813098b456c.

[Men04] Paul Menage. CGROUPS. Accessible en ligne, 2004. Voir: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[NK19] Jeff Nicloloff and Stephen Kuenzli. *Docker in Action*. Manning Publications, 2019. Voir: https://www.manning.com/books/docker-in-action-second-edition.

[Ope15] OpenVz. Openvz history. Accessible en ligne, 2015. Voir: https://wiki.openvz.org/History.

[Pat03] Steve D. Pate. *UNIX Filesystems: Evolution, Design, and Implementation*. Wiley Publishing, 2003.

[PE07] Kir Kolyshkin Pavel Emelyanov. PID namespaces in the 2.6.24 kernel. Disponible en ligne, 2007. Voir: https://lwn.net/Articles/259217/.

[Pet15] Jerome Petazzoni. Container's anatomy. Accessible en ligne, 2015. Voir: `https://fr.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon from_action=save`.

[PJP91] Emerson W. Pugh, Lyle R. Johnson, and John H. Palmer. *IBM's 360 and Early 370 Systems*. MIT Press, 1991. Voir: `https://mitpress.mit.edu/books/ibms-360-and-early-370-systems`.

[PPT+92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Proc. of the fifth ACM SIGOPS European Workshop*, page 1, Mont Saint-Michel, 1992. ACM. Voir: `http://doc.cat-v.org/plan_9/4th_edition/papers/names`.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmet Witchel, and Anoop Goopta. Complete Computer System Simulation: The SimOS Approach. *IEEE Paralell & Distributed Technology*, 3:4:34–43, 1995. Voir: `https://www.cs.toronto.edu/~demke/2227/S.14/Papers/rosenblum-simos.pdf`.

[Rit80] Dennis Ritchie. *The Evolution of the Unix Time-sharing System, in Language Design and Programming Methodology*, pages 25–35. Springer-Verlag, 1980. Voir: `https://www.read.seas.harvard.edu/~kohler/class/aosref/ritchie84evolution.pdf`.

[Rod08] Ohad Rodeh. B-trees, shadowing, and clones. *Transactions on Storage*, 3:4:15:1–27, 2008. Voir: `https://dl.acm.org/doi/pdf/10.1145/1326542.1326544`.

[RT74] Dennis Ritchie and Ken Thompson. The Unix Time-Sharing System. *Communications of the ACM*, 17:7:365–375, 1974. Voir: `https://dsf.berkeley.edu/cs262/unix.pdf`.

[Set06] Rohit Seth. Containers: Introduction. Disponible en ligne, 2006. Voir: `https://lwn.net/Articles/199643/`.

[SN05] James E. Smith and Ravi Nair. *Virtual Machines*. Morgan Kaufmann, 2005.

[SS07] Balbir Singh and Vaidyanathan Srinivasan. Containers: Challenges with the memory resource controller and its performance. In *Proc. of the Linux Symposium*, pages 209–222, Ottawa, 2007. USENIX. Voir: `https://www.kernel.org/doc/mirror/ols2007v2.pdf`.

[Sta16] Ars Staff. Why microsoft needed to make windows run linux software. Publié par Ars Technica, 2016. Voir: `https://arstechnica.com/information-technology/2016/04/why-microsoft-needed-to-make-windows-run-linux-software/`.

[Sup21] JFrog Support. Docker hub and docker registries: A beginners guide. Accessible en ligne, 2021. Voir: https://jfrog.com/knowledge-base/docker-hub-and-docker-registries-a-beginners-guide/.

[Tor16] Linus Torvalds. Linux 4.6-rc1. Disponible en ligne, 2016. Voir: https://lkml.org/lkml/2016/3/26/132/.

[Uni14] Unionfs. Unionfs: A stackable unification file system. Accessible en ligne, 2014. Voir: https://unionfs.filesystems.org/.

[Vir01] Alexander Viro. He's back. And this time he's got a chainsaw. Accessible en ligne, 2001. Voir: https://lwn.net/2001/0301/a/namespaces.php3.

[vK21] Scott van Kalken. What are namespaces and cgroups, and how do they work? Accessible en ligne, 2021. Voir: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/.

[VSe01] Linux VServer. Overview. Accessible en ligne, 2001. Voir: http://linux-vserver.org/Overview.

[Wat09] Scott Watanabe. *Solaris 10 ZFS Essentials*. Prentice Hall, 2009.

[Wik22] Wikipedia. Linux namesapces. Accessible en ligne, 2022. Voir: https://en.wikipedia.org/wiki/Linux_namespaces.

[Yag04] Tom Yager. Sending software to do hardware's job. Infoworld, 2004. Voir: https://www.infoworld.com/article/2664741/sending-software-to-do-hardware-s-job.html.