Chapitre 3
Les containers

1. Cahier des charges

Le chapitre Application standalone a révélé les défauts d'une architecture monolithique. Le chapitre Infrastructure et services de base a fourni les bases d'une infrastructure tolérante aux pannes et capable d'accueillir des applications dans de bien meilleures conditions. Il faut maintenant pouvoir déployer l'application sur les serveurs, le plus simplement et le plus rapidement possible. Il y a de nombreuses étapes à respecter pour l'installer, la configurer et la démarrer. Il est temps d'ajouter des éléments au cahier des charges :

– L'application doit pouvoir être installée simplement.
– L'application doit être réutilisable.
– L'application doit pouvoir être déployée sur plusieurs serveurs.
– Un même serveur doit pouvoir faire tourner plusieurs instances de l'application.
– La maintenance de l'application doit être minimale.
– L'application ne doit pas accaparer toutes les ressources du serveur.

Ce dernier point peut surprendre, et pourtant c'est aussi un élément de disponibilité et de tolérance aux pannes. Un processus qui consomme toutes les ressources CPU empêche le bon fonctionnement des autres applications et services.

S'il consomme toute la mémoire, il risque d'être tué par Linux, via le mécanisme OOMKiller (*Out-Of-Memory Killer*). Il ne faudrait pas oublier que le rôle d'un système d'exploitation est de préserver à tout prix le bon fonctionnement global du système et de préserver les ressources, quitte à sacrifier certains services et applications.

Plusieurs solutions classiques pourraient être envisagées. Toutes les distributions Linux diffusent les composants logiciels sous forme de packages. On pourrait imaginer un package dpkg ou rpm pour notre application, ce qui permettrait de l'installer plus simplement, et le package serait réutilisable. Mais le cahier des charges ne serait pas entièrement rempli : comment l'installer plusieurs fois sur le même serveur ? Comment l'empêcher de mettre en danger le serveur dans son ensemble ?

Il se trouve que le noyau Linux dispose depuis de nombreuses années de tout le nécessaire pour résoudre le problème : les mécanismes d'isolation et les containers.

2. Isolation et container

2.1 Principe

L'isolation est une technique consistant à exécuter des applications dans leurs propres contextes, isolés des autres. C'est une forme de virtualisation. La première implémentation de ce type sous Unix date de 1979, avec **chroot**.

L'isolation d'un ou de plusieurs processus est assurée par le noyau Linux à l'aide des deux mécanismes : les espaces de noms ou **namespaces**, et les groupes de contrôle ou **cgroups**. Les namespaces sont apparus en 2002, et les cgroups en 2007. Tout noyau Linux intègre ces fonctionnalités par défaut. L'arrivée des espaces de noms utilisateurs ou **users namespaces** dans le noyau 3.8 en février 2013 a ouvert la voie au support complet des conteneurs ou containers.

Les namespaces regroupent des processus dans un espace isolé des autres. Ils permettent d'avoir :

– une isolation des processus et un processus de PID 1 comme premier processus dans son espace (PID 1 du premier processus d'un espace de noms) ;
– une isolation du réseau, avec des adresses IP et des ports qui leur sont propres ;
– une isolation des volumes de données : le stockage ;
– une isolation des droits et des utilisateurs par rapport à l'hôte : être root au sein d'un container peut ainsi correspondre à un utilisateur sans pouvoir sur l'hôte.

Les cgroups contrôlent les ressources du système qu'un groupe de processus peut utiliser, avec par exemple les contrôles suivants :

– limiter la consommation de la mémoire ;
– limiter l'utilisation des processeurs ;
– gérer les priorités ;
– obtenir des informations « comptables » sur ce groupe ;
– contrôler dans leur ensemble (arrêter, par exemple) plusieurs processus ;
– isoler ce groupe, en l'associant à un espace de noms.

L'association de ces deux mécanismes est la base du principe du container.

2.2 Container et machine virtuelle

Les containers sont souvent comparés à des machines virtuelles pour en faciliter la compréhension, mais il s'agit de deux types de technologies de virtualisation bien distinctes. Le concept est différent.

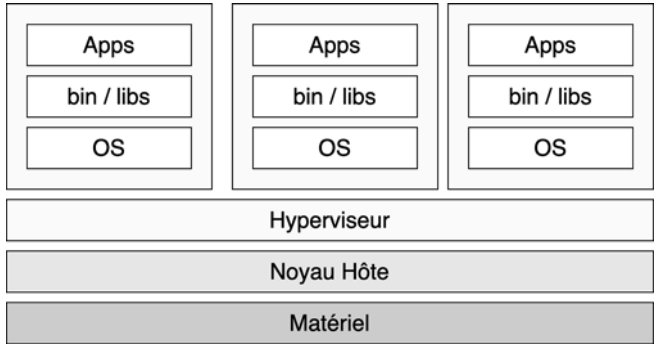

Figure 1 : Hyperviseur et machines virtuelles

Les machines virtuelles fonctionnent sur des hyperviseurs. Qu'elles utilisent l'émulation ou non, les machines virtuelles exécutent un système d'exploitation complet, avec un noyau et des pilotes de périphériques (même si ceux-ci utilisent la paravirtualisation). Les applications s'installent sur le système d'exploitation de la machine virtuelle. Vue de l'hyperviseur, la machine virtuelle est un processus unique.

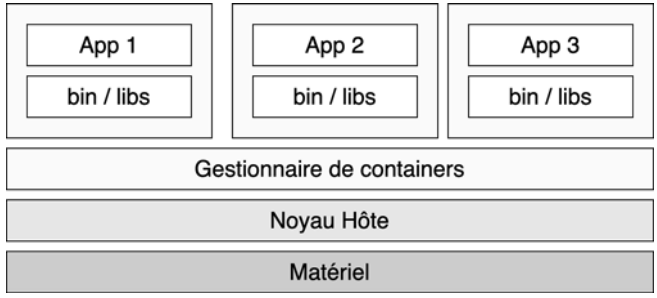

Figure 2 : Hôte et containers

Les containers fonctionnent directement sur l'hôte, sur le même noyau. Les processus qu'ils contiennent sont des processus comme les autres, juste isolés et limités en ressources. Si un container contient dix processus, on voit bien dix processus sur son hôte.

2.3 Namespace

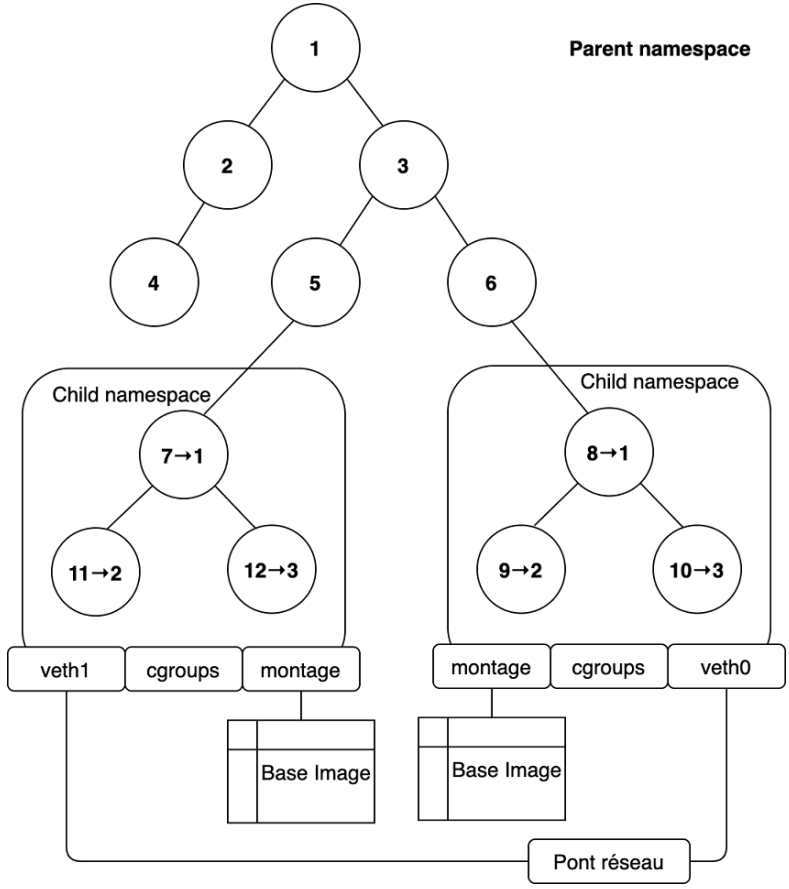

Figure 3 : Containers : namespaces et cgroups

Tout premier processus d'un namespace porte un PID 1 (*Process Identifier 1*). Tous les fils de ce processus font automatiquement partie de ce namespace. Tous les processus de cet espace sont isolés des autres namespaces. Ils ne voient pas ce qui se passe à l'extérieur de leur espace. C'est le **PID namespace**.

Selon ce principe, le premier processus démarré par le noyau (init, systemd...) et qui porte le PID 1 est aussi dans un namespace : le **parent namespace**. Depuis le parent, les processus au sein d'un espace sont vus comme des processus ordinaires, avec des PID « normaux ». Ainsi, le processus de PID 1 d'un namespace pourrait avoir le PID 12345, vu de l'extérieur. Par exemple, ici le processus de PID 1 de ce container est le PID 1148 de l'hôte sur lequel il tourne :

```
# Dans le container :
# cat /proc/1/cmdline | tr '\0' ' ' ; echo
/usr/local/openjdk-11/bin/java .

# Vu du serveur :
eni@node2:~$ ps auxww |grep java
root      1148 28.0 21.1 4909116 412356 ?
Ssl  20:22   3:08 /usr/local/openjdk-11/bin/java
```

Les mécanismes des namespaces permettent aussi de modifier la racine rootfs du processus, à la manière d'un chroot. On installe tout le nécessaire au fonctionnement des processus dans un répertoire, puis on remplace le système de fichiers racine / du namespace par ce répertoire. C'est le **mount namespace**.

Chaque namespace peut disposer d'une interface réseau virtuelle. Cette interface est souvent pontée sur une interface réseau de l'hôte ou sur une autre interface gérée par un service spécifique. L'interface virtuelle reçoit sa propre adresse IP et assure la communication entre le namespace et l'extérieur. C'est le **network namespace**.

Par défaut, les processus d'un namespace partagent les utilisateurs et groupes avec ceux du système. On peut pourtant faire en sorte de les isoler, de les dissocier, ou au contraire de les associer à d'autres utilisateurs. Ainsi, il devient possible d'avoir des utilisateurs de même UID (*User IDentifier*) dans deux espaces, alors que ces utilisateurs sont pourtant différents, ou encore d'avoir un compte root au sein de l'espace qui sera associé à un utilisateur sans privilège en dehors de celui-ci. C'est le **user namespace**.

Chaque namespace peut être associé à un cgroup afin de limiter les ressources des processus de l'espace. C'est le **cgroup namespace**.

Il existe deux autres types de namespaces : **IPC** (*InterProcess Communication*), qui permet d'isoler les communications interprocessus au sein de l'espace, et **UTS** (*Unix Timesharing System*), permettant de changer les noms d'hôtes (*hostname*) et de domaines (*domainname*, NIS (*Network Information System*)) de l'espace.

2.4 cgroup

Si on place des processus au sein d'un cgroup, on peut contrôler les ressources qu'ils peuvent utiliser. On peut ainsi limiter ces processus à une quantité de mémoire utilisable donnée, par exemple 256 Mo, et à une quantité de temps processeur donnée, exprimée en cœurs ou millicœurs, par exemple 500 millicœurs ou 0,5 cœur. Ces limites sont pour l'ensemble de l'espace de noms, et non des limites par processus.

Les groupes de contrôle permettent non seulement de limiter l'utilisation des ressources d'un groupe de processus, mais aussi d'empêcher un groupe de processus d'accaparer toutes les ressources du système, et donc de cannibaliser les autres applications et le système lui-même. Ainsi, si un processus du groupe tente d'utiliser plus de mémoire que le quota alloué, il sera tué par les mécanismes de protection cgroups du noyau Linux.

2.5 Montage en union

Le montage en union (*union mount*) réunit plusieurs systèmes de fichiers – généralement un empilement ordonné de répertoires –, en un point de montage unique, créant ainsi un système de fichiers virtuel, fruit de l'union de tous les autres. Chaque couche de l'empilement ajoute ou supprime des fichiers. Les systèmes de fichiers les plus connus sont **AUFS** (*Another Union FileSystem*) et **Overlay**. Un système de fichiers de ce type peut être aussi implémenté en utilisant LVM (*Logical Volume Manager*) ou les instantanés *snapshots* proposés par certains systèmes de fichiers comme BTRFS (*B-TRee File System*). Overlay est maintenant le plus utilisé.

Ce type de système de fichiers représente un avantage majeur monté en tant que système de fichiers racine d'un container : toutes les couches sont en lecture, sauf la dernière – un répertoire volontairement vide, qui est en écriture. Le container peut ainsi écrire sur le point de montage sans modifier le système de fichiers de base. Lorsque le container est détruit, les modifications sont perdues par la destruction de la dernière couche.

Un second avantage de cette technologie est de pouvoir capitaliser et mutualiser certaines couches. Ainsi, si on n'a qu'un fichier de 20 Mo de différence entre deux images de container de 100 Mo, ces deux images n'occupent que 120 Mo, et non 200 Mo.

2.6 Image applicative

Pour démarrer une application au sein d'un container, toutes ses dépendances doivent être installées sur son système de fichiers racine : l'application est totalement isolée et ne peut pas accéder aux fichiers hors de son espace. Une méthode consiste à copier dans un répertoire une arborescence Linux standard contenant les fichiers (exécutables, bibliothèques et fichiers de configuration) de base, les dépendances de l'application à exécuter dans le container, et l'application elle-même.

L'arborescence Linux de base peut être celle d'une quelconque distribution.

On peut créer une copie de cette arborescence afin d'en faire une image de base réutilisable. On peut la stocker, la répliquer, la distribuer. On peut créer des catalogues, des dépôts. On peut la décliner pour l'adapter à ses besoins, en faire plusieurs versions.

On peut monter cette arborescence en lecture seule en union avec un répertoire vide en écriture, et l'utiliser comme système de fichiers racine dans l'espace de noms.

Créer un container, c'est instancier une image de système de fichiers, puis démarrer un ou plusieurs processus, au sein d'un namespace.

2.7 Couches d'images

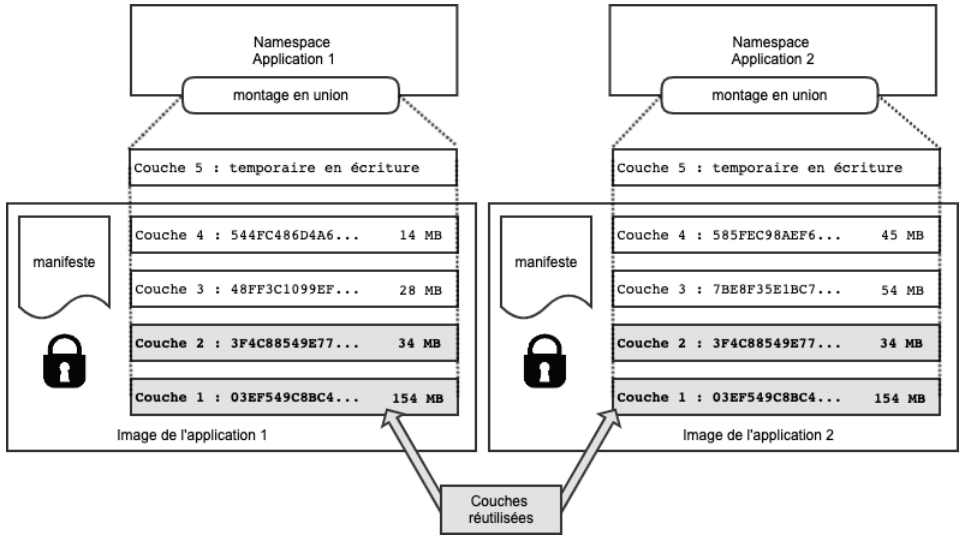

Figure 4 : Images en couche et montage en union

Quand on construit une image, on part du minimum utilisable.

Par-dessus cette couche minimale, on peut ajouter les mises à jour des packages, puis installer les dépendances de l'application qu'on souhaite démarrer, puis l'application elle-même.

À chaque étape, on ajoute une couche (*layer*) supplémentaire, par exemple :

– couche 1 : l'image de base ;
– couche 2 : les mises à jour ;
– couche 3 : les dépendances de l'application ;
– couche 4 : l'application.