Chapitre 2
Docker en réseau

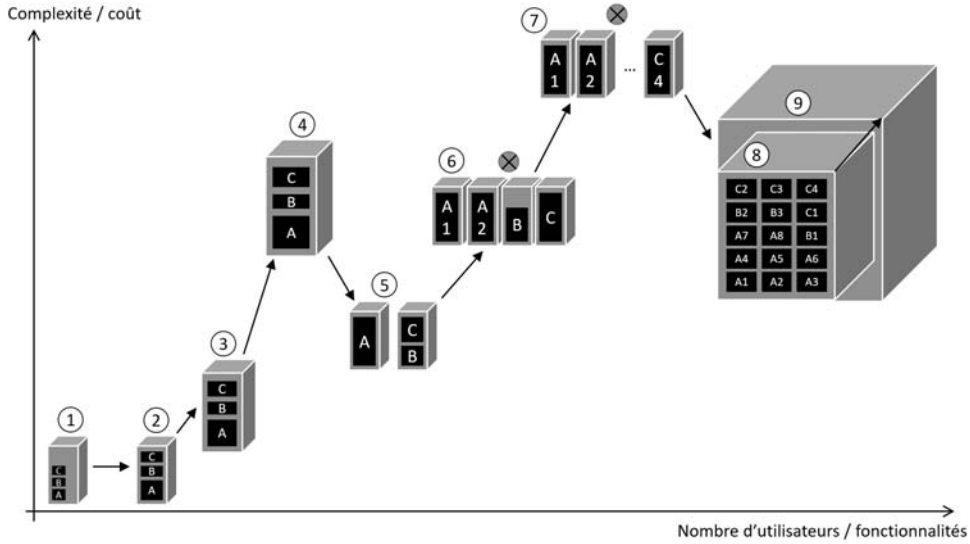
1. Mise en réseau de Docker

1.1 Approche théorique

1.1.1 Problématique de montée en charge

Un problème auquel est confronté presque tout administrateur d'une application web au cours de sa vie est l'augmentation forte des ressources consommées par une application. Des optimisations logicielles ou d'organisation peuvent parfois être réalisées, mais globalement, la ressource utilisée croît avec le nombre d'utilisateurs et les fonctionnalités exposées.

Le schéma ci-dessous montre les différentes étapes par lesquelles un administrateur "traditionnel" peut théoriquement passer :



– L'étape 1 est la mise en production pour un ensemble d'utilisateurs restreints, à savoir généralement les seules personnes connaissant l'application avant qu'elle ne soit globalement répertoriée sur Internet : typiquement, les testeurs et personnes intéressées de l'entreprise publiant le site ou l'application. À ce niveau de charge, tout se passe généralement bien. Chaque module de l'application (des conteneurs Docker, par exemple) est à l'aise avec la part de ressources qu'il peut consommer.

– Imaginons que l'application fonctionne correctement et que les utilisateurs sont au rendez-vous. La croissance aboutira à l'étape 2, à savoir que les conteneurs augmentant leur besoin en ressource, ils occuperont au final toute la puissance de la machine hôte.

– À ce moment-là, un facteur limitant apparaîtra sur la mémoire vive, le stockage, la bande passante réseau ou la puissance de calcul (voire sur plusieurs de ces caractéristiques). Une des approches les plus simples pour éviter des pertes de performance – ou même des réponses en erreur dans le pire de cas – est de réaliser l'opération nommée "scale up" et qui consiste à augmenter la taille de la machine utilisée. C'est ce que nous représentons à l'étape 3. Si le système n'est pas virtualisé, ceci nécessite de réinstaller la totalité des briques logicielles, et si l'application n'a pas été structurée pour gérer cette éventualité, le coût peut être élevé.

– En règle générale, l'administrateur prévoyant qui a été confronté à ce besoin de "scale up" prend ses précautions et, si l'application est prévue pour augmenter encore son empreinte, il choisira une machine suffisamment bien dotée pour voir venir les prochains pics d'exploitation. Pourtant, quelles que soient les ressources financières à disposition, il arrive un moment où il n'est plus possible d'acheter une machine plus grosse (en pratique, la limite où l'application n'est plus en mesure d'exploiter correctement des ressources plus abondantes peut parfois être atteinte avant, typiquement lorsqu'une application ne peut tirer parti des processeurs multiples). C'est l'étape 4 qui représente ce moment dans la croissance théorique d'une application.

– Le coût a augmenté de manière forte sur ces dernières étapes, mais la complexité est toutefois contenue. Le passage à l'étape 5, obligatoire car le "scale up" ne peut plus fonctionner, va permettre de réduire drastiquement les coûts, mais au prix d'une complexité en hausse : il consiste à passer au "scale out", c'est-à-dire à l'augmentation du nombre de machines. Dans un premier temps, comme précédemment, l'approche est relativement simple : il suffit de mettre en place deux machines et de répartir intelligemment les conteneurs entre celles-ci, en fonction de la connaissance que l'administrateur a de leur fonctionnement (connaissance qui nécessite une communication avec les développeurs ayant créé ladite application, ce qui est parfois en soi un problème).

– Ensuite, le "scale out" va lui aussi connaître sa phase de complexification, lorsqu'il va falloir gérer plusieurs instances d'un même service pour continuer à augmenter en performance. Pour cela, il faudra introduire dans le système un mécanisme de répartition de charge (*load balancing* en anglais), comme HAProxy. L'étape 6 montre ce cas de figure, avec deux instances du conteneur A, et le symbole du load balancer au-dessus des machines logiques.

– Si la croissance se poursuit, de plus en plus de machines seront nécessaires pour gérer les multiples instances de chacun des services (comme montré à l'étape 7). De plus, si l'habitude a été prise de gérer des gros conteneurs occupant toute la ressource d'un serveur logique, le nombre de ces derniers va commencer à poser problème.

– La solution proposée à ce problème serait de disposer d'un système qui, bien que composé de multiples machines logiques (physiques ou virtuelles), apparaîtrait comme un seul hôte Docker, ce qui permettrait une gestion plus simple, tout en rendant possible la présence de nombreux conteneurs. C'est cette solution, schématisée par l'étape 8, que nous allons étudier plus avant dans ce chapitre.

– Le principal avantage de cette solution est que la mise à l'échelle étant complètement disjointe entre le nombre de machines ou leur taille et le nombre de conteneurs que l'ensemble va pouvoir porter, il est très simple de faire évoluer indépendamment ces deux caractéristiques. À l'augmentation des performances attendues correspondra l'augmentation du nombre de conteneurs Docker. Ceci se traduira par la nécessité d'augmenter la taille de l'hôte Docker "virtuel", ce qui se fera par ajout de machines dans ce qu'on appelle communément le cluster (grappe en français, mais ce vocabulaire est très peu utilisé). C'est l'étape 9, d'agrandissement du parallélépipède représentant un cluster, qui illustre ceci.

1.1.2 Solution découplée

Revenons un peu plus en détail sur la solution proposée à l'étape 8 ci-dessus. Elle consiste à séparer (on parle parfois de "découplage") la taille et le nombre des machines physiques et virtuelles supportant le démon Docker du dimensionnement de l'application utilisant cette infrastructure. Pour arriver à cette séparation des deux notions, il est nécessaire qu'un client Docker soit capable d'adresser un ensemble d'hôtes Docker de la même manière qu'il appellerait le démon positionné sur une machine hôte seule. Pour cela, il convient que le mode d'appel du démon ne soit pas lié strictement à des caractéristiques d'une machine. L'utilisation du protocole TCP pour échanger avec les démons permet précisément ceci, car seul un identifiant de type IP est nécessaire pour indiquer au client Docker quel démon ou quel ensemble de démons (en passant par une des machines) il doit contacter. C'est ce qui est réalisable en modifiant la valeur de la variable d'environnement DOCKER_HOST :

```
set DOCKER_HOST = tcp://172.99.79.150:2376
```

Le fait que le contrat entre un client Docker et un serveur Docker soit aussi simple est une première partie de la solution. La seconde tient dans le fait qu'un ensemble de démons Docker en réseau peut être adressé par un seul couple adresse IP + port réseau, correspondant à n'importe lequel des nœuds de gestion du cluster (le vocable anglais de "manager node" est souvent utilisé). Ce nœud de gestion se charge ensuite de "pousser" les bonnes instructions Docker sur les nœuds de travail (on parle de "worker nodes"). Enfin, la troisième partie de la solution tient dans le fait que l'API Docker est contractuelle, c'est-à-dire qu'elle reste la même, qu'on accède à un démon ou à un réseau de démons. Ces trois caractéristiques aboutissent ensemble au découplage précité entre la taille du cluster et le nombre de démons Docker vus par le client, à savoir systématiquement un.

Remarque

L'API Docker est désormais standardisée par le biais de deux spécifications de l'Open Container Initiative, un consortium web ouvert de normalisation de la gestion des conteneurs auquel adhèrent Docker ainsi que de nombreuses autres grandes sociétés d'informatique. La première de ces spécifications concerne la gestion de l'exécution des conteneurs (et donc l'API pour les démarrer, les arrêter, etc.) et la seconde, la définition des images. Plus d'informations sur https://www.opencontainers.org/.

1.1.3 Conséquences sur l'approche initiale

Nous avons montré l'existence d'un mode que l'on qualifie couramment d'élastique, dans le sens où les conteneurs n'ont pas à se poser la question de la structure des machines virtuelles ou physiques sous-jacentes au démon Docker qui les accueille. Dès lors, la question se pose de directement créer les infrastructures de déploiement dans ce mode, et ce sans attendre que la montée en charge ne nous y contraigne.

Plusieurs arguments vont dans ce sens. Tout d'abord, le fait que le cluster puisse n'être composé que d'une machine permet de limiter au maximum les coûts de fonctionnement en première étape. Ce n'est certes pas le cas pour la part d'investissement de ces coûts, mais nous verrons un peu plus loin qu'elle est très limitée depuis les dernières versions de Docker, l'installation d'un cluster étant à peine plus complexe que l'installation d'un démon Docker sur une seule machine. Elle peut même être limitée encore plus fortement en faisant appel à un fournisseur dédié, qui proposera ce service en ligne (on parle alors de CaaS, pour *Container as a Service*).

De plus, le coût des changements de machines et d'infrastructures tels que montrés en théorie dans les étapes décrites plus haut est en général très élevé, non seulement pour le matériel mais aussi à cause de la mise en œuvre, potentiellement complexe, et ceci, sans même compter les coûts en termes d'image à cause des possibles ruptures de services dans ces moments à risque pour la production.

À l'inverse, il ne faut pas négliger le coût de formation à cette nouvelle façon de gérer les serveurs, qui va à l'encontre de ce que bon nombre d'administrateurs ont mis en pratique pendant des années. Dans le mode traditionnel, l'administrateur connaît chacun des serveurs par son nom, il sait ce qui se trouve dessus, a l'habitude de gérer telle ou telle spécificité. Lorsque le serveur montre des signes de fatigue, sa connaissance de ce serveur particulier lui permet de rapidement déterminer la criticité du problème et l'urgence qu'il y a à le régler par rapport à d'autres. Dans ce nouveau mode où les machines ne sont que des supports d'un cluster, l'administrateur les gère de manière indifférenciée. Il en va d'ailleurs de même pour les conteneurs, puisqu'il ne peut pas savoir, dans un cluster, sur quelle machine quels conteneurs vont s'exécuter. Le système peut même arrêter dynamiquement un conteneur sur un serveur en même temps qu'il le redémarre sur un autre.