UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UNE PLATEFORME D'ORCHESTRATION DE CONTENEURS DOCKER TOLÉRANTE AUX FAUTES BYZANTINES

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

GOR MACK DIOUF

AVRIL 2019

REMERCIEMENTS

En préambule à ce rapport de mémoire, je souhaiterais adresser mes remerciements les plus sincères aux personnes qui m'ont apporté leur aide et qui ont contribué à la réalisation de ce travail.

Je voudrais tout d'abord exprimer ma profonde gratuite à Madame Pr. Halima ELBIAZE, ma directrice de recherche. Tout au long de ma maîtrise, j'ai eu l'honneur de bénéficier de son soutien scientifique, moral et financier. Je tiens à lui exprimer ma reconnaissance pour sa disponibilité ainsi que ses conseils avisés.

C'est également du fond du cœur que je remercie Monsieur Wael JAAFAR qui a supervisé ce travail en me prodiguant d'excellents conseils, avec un professionnalisme hors pair, toujours inspiré par l'essentiel. Sa clairvoyance et sa disponibilité font de lui un superviseur idéal qui a toujours répondu, avec beaucoup de plaisir, mais aussi et surtout de manière très pertinente, à mes nombreuses questions.

Mes vifs remerciements vont également aux membres du jury pour l'intérêt qu'ils ont porté à notre recherche en acceptant d'examiner notre travail et de l'enrichir par leurs propositions.

Je désire aussi témoigner ma reconnaissance à Madame Catherine Truchan et toute la direction de Ericsson Canada pour avoir mis à notre disposition l'infrasture cloud qui nous a permis d'expérimenter notre recherche.

Je tiens particulièrement à remercier la Fondation de l'UQAM, pour les bourses d'excellence qui m'ont été attribuées ; ces bourses ont grandement facilité mes activités de recherches.

J'adresse aussi mes remerciements à tous mes collègues au sein du laboratoire TRIME et de de l'Université du Québec à Montréal (UQAM) avec qui j'ai passé de très bons moments.

Un grand merci à toute ma famille, mon épouse et ma fille pour leur amour, leur patience et l'estime qu'elles portent pour moi.

Gor Mack DIOUF

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

VMM	*Virtual Machines Monitor*
μs	Microseconde
Gbps	Giga bit par seconde
LXC	*LinuX Container*
AAA	*Access Anywhere Anytime*
API	*Application Programming Interface*
BFT-SMaRt	*Byzantin Fault-Tolerance State Machine Replication*
CNCF	*Cloud Native Computing Foundation*
CPU	*Central Processing Unit*
DDoS	*Distributed Denial of Service*
E/S	Entrées/Sorties
EBFT	*Efficient Byzantin Fault-Tolerance*
FLP	*Fischer Lynch Problem*
GPL	*General Public License*
IaaS	*Infrastructure as a Service*
IBM	*International Business Machines Corporation*

IP	*Internet Protocol*
IPC	*Inter-Process Communication*
JVM	*Java Virtual Machine*
K8s	Kubernetes
kmMC	Kubernetes multi-maîtres classique
kmMR	Kubernetes multi-maîtres robuste
KVM	*Kernel-based Virtual Machine*
MAC	*Message Authentication Code*
NAS	*Network Attached Storage*
NIST	*National Institute of Standards and Technology*
OS	*Operating System*
PaaS	*Plateform as a Service*
PBFT	*Pratical Byzantin Fault-Tolerance*
RAM	*Random Access Memory*
SaaS	*Software as a Service*
SAN	*Storage Area Network*
SMR	*State Machine Replication*
TRIME	Télécommunication, Réseaux, Informatique Mobile et Embarquée
VCPU	*Virtual Central Processing Unit*

VE	*Virtual Environment*
VM	*Virtual Machine*
VPN	*Virtual Private Network*

RÉSUMÉ

Au cours des dernières années, l'utilisation des technologies de virtualisation dans l'infonuagique a considérablement augmenté. Ceci est dû principalement aux avantages en termes d'efficacité d'utilisation des ressources et de robustesse que procure la virtualisation. La virtualisation par conteneurs Docker et la virtualisation par hyperviseurs sont les deux principales technologies apparues sur le marché. Parmi elles, la virtualisation par conteneurs Docker se distingue par sa capacité de fournir un environnement virtuel léger et efficace, mais qui nécessite la présence d'un orchestrateur. Par conséquent, les fournisseurs de l'infonuagique adoptent la plateforme ouverte Kubernetes comme étant le gestionnaire standard des applications conteneurisées.

Afin de garantir la disponibilité des applications dans Kubernetes, ce dernier fait appel au mécanisme de réplication des machines Raft. Malgré sa simplicité, Raft suppose que les machines échouent uniquement à l'arrêt. Cet évènement d'arrêt est rarement la seule raison d'un disfonctionnement en pratique. En effet, des erreurs logicielles ou des attaques malicieuses peuvent amener des machines à présenter un comportement byzantin (c'est-à-dire aléatoire) et par conséquent corrompre l'exactitude et la disponibilité du protocole non-byzantin Raft.

Dans ce mémoire, nous proposons une plateforme Kubernetes multi-maîtres robuste (kmMR) pour surmonter cette limitation. Notre solution kmMR se base sur l'adaptation et l'intégration d'un protocole byzantin BFT-SMaRt dans l'environnement Kubernetes pour la duplication des machines (noeuds maîtres dans ce contexte). Contrairement au protocole Raft, BFT-SMaRt permet de résister aux fautes byzantines et non-byzantines.

Nous commençons par proposer une approche d'implantation, afin d'équiper la plateforme Kubernetes du protocole de réplication byzantin BFT-SMaRt. Ensuite, nous proposons de modéliser des fautes byzantines en simulant des attaques de deni-de-service distribuées (DDoS). Ces dernières vont servir dans l'évaluation des performances de KmMR, ainsi que dans sa comparaison à la plateforme classique utilisant le protocole Raft.

Les résultats montrent que KmMR est capable de garantir la continuité du service, même en présence de plusieurs fautes. De plus, KmMR assure en moyenne

un délai de consensus 1000 fois plus court que celui réalisé par la plateforme classique. Enfin, nous montrons que KmMR génère un faible surcoût en terme de consommation des ressources par rapport à la plateforme classique.

Mots clés : cloud computing, conteneurs Docker, Kubernetes, tolérance aux fautes byzantines, continuité de service

INTRODUCTION

Contexte

Confrontées à l'augmentation continuelle des coûts de mise en place et de maintenance des systèmes informatiques accessibles depuis partout et à tout moment, les entreprises externalisent de plus en plus leurs services informatiques en les confiant à des entreprises spécialisées comme des fournisseurs services nuagiques (*cloud*). L'intérêt principal de cette stratégie pour les entreprises réside dans le fait qu'elles peuvent se prévaloir d'un excellent niveau de service, en ne payant que les ressources nécessaires et effectivement consommées. Quant au fournisseur du *cloud*, son but est de répondre aux besoins des clients en déployant le minimum de ressources nécessaires. Une approche courante du fournisseur consiste à mutualiser ses ressources (*slicing*) pour les partager entre plusieurs entreprises clients. Dans ce contexte, plusieurs défis se dressent afin d'offrir un environnement *cloud* efficace. Nous pouvons citer parmi elles la garantie de bonnes performances, la gestion des ressources et la continuité de service.

Motivations et problématique

Actuellement, les fournisseurs du *cloud* utilisent la conteneurisation Docker comme support technologique pour résoudre les problèmes de performances causés par la virtualisation traditionnelle (utilisation abusive des ressources) (Bernstein, 2014; Abdelbaky *et al.*, 2015; Docker, 2016a; Peinl *et al.*, 2016; Manu *et al.*, 2016b; Naik, 2016; Peinl *et al.*, 2016; Madhumathi, 2018; Kovács *et al.*, 2018). Ainsi, l'unité d'allocation de ressources dans le *cloud* est le conteneur Docker.

Face au besoin de gestion, les fournisseurs de *clouds* (Google, Docker, Mesosphere, Microsoft, Vmware, IBM , Oracle, etc.) (Foundation, 2017) ont mis en place la fondation de l'informatique nuagique native, la *CNCF* (*Cloud Native Computing Foundation* (CloudNativeCon, 2018)) dans le but de standardiser l'utilisation des conteneurs dans le *cloud*. La *CNCF* a adopté Kubernetes comme la plateforme standard d'orchestration d'applications conteneurisées (Bernstein, 2014; Sill, 2015; Rensin, 2015; Burns *et al.*, 2016; Google, 2018; Heidari *et al.*, 2016). Kubernetes, un projet ouvert initié par Google (Bernstein, 2014; Google, 2018), prône la vision d'une plateforme d'orchestration modulaire, personnalisable et évolutive.

Cependant, quant à la continuité de service, nous constatons que les mécanismes fournis par Kubernetes sont limités. Kubernetes utilise le protocole Raft (Ongaro et Ousterhout, 2014) pour répliquer les états entre ses composants afin de garantir la disponibilité des applications hébergées (Google, 2018; Oliveira *et al.*, 2016; CoreOS, 2018; Blake Mizerany et Qin, 2018). Toutefois, Raft suppose que les composants échouent uniquement à l'arrêt. Cet évènement d'arrêt est rarement la seule raison d'un échec en pratique. Des erreurs logicielles ou des attaques malicieuses peuvent amener un composant à présenter un comportement byzantin (c'est-à-dire arbitraire) et par conséquent corrompre l'exactitude et la disponibilité garanties par le protocole Raft.

Afin de surmonter cette limitation, nous proposons dans ce travail une solution qui se base sur l'adaptation et l'intégration d'un protocole byzantin *BFT-SMaRt* (Sousa *et al.*, 2013) dans l'environnement Kubernetes. Contrairement au protocole Raft, *BFT-SMaRt* permet de résister à toute faute.

Contributions

Ce mémoire apporte les contributions suivantes :

- Nous formulons le problème de tolérance aux fautes byzantines dans Kubernetes.

- Nous proposons notre solution Kubernetes multi-maîtres robuste (KmMR), une plateforme Kubernetes tolérante aux fautes byzantines. KmMR est basée sur l'intégration du protocole byzantin *BFT-SMaRt* dans Kubernetes pour résister à toute sorte de faute, y compris les fautes byzantines. KmMR est la première plateforme Kubernetetes tolérante aux fautes byzantines dans la littérature des systèmes distribués.

- Nous proposons une méthode efficace pour adapter et intégrer le protocole byzantin *BFT-SMaRt* (écrit en Java, un langage de programmation orientée objet) dans Kubernetes (écrit en Go, un langage de programmation orientée service).

- Nous mettons en œuvre notre solution KmMR et une autre plateforme Kubernetetes multi-maîtres classique (kmMC) dans un environnement *cloud* basé sur OpenStack. Contrairement à notre solution, KmMC se base simplement sur le protocole de réplication Raft qui est déjà intégré dans Kubernetes.

- Nous avons conduit une évaluation expérimentale dans le cas d'un environnement non-byzantin, c'est-à-dire le nombre de nœuds défaillants est strictement inférieur au seuil fixé par le protocole de réplication en place, et dans un environnement byzantin où le nombre de nœuds défaillants est égal au seuil fixé par le protocole de réplication en place. Des attaques de déni de service distribuées (*DDoS*) ont été menées afin d'étudier les performances de notre plateforme KmMR par rapport à la plateforme classique (KmMC). Les résultats obtenus confirment l'efficacité et la robustesse de notre solution.

Plan du mémoire

La suite de ce mémoire s'organise de la manière suivante.

Le premier chapitre fournit un aperçu des concepts associés à la conteneurisation. Plus précisément, nous décrivons les interactions entre le *cloud*, la virtualisation traditionnelle et la virtualisation moderne. À partir de là, nous discutons de l'intérêt de l'utilisation des conteneurs Docker dans l'informatique moderne.

Ensuite, le chapitre 2 pose les défis de l'utilisation à l'échelle des conteneurs Docker. Nous définissons les fonctionnalités requises pour orchestrer les conteneurs. Puis, nous présentons les outils gratuits capables de gérer les conteneurs Docker. Nous discutons également de l'intérêt de Kubernetes par rapport aux autres plateformes d'orchestration.

Le chapitre 3 étudie le problème de tolérance aux fautes dans Kubernetes. Nous identifions les pannes qui peuvent survenir dans un environnement Kubernetes et nous présentons les travaux connexes pertinents à notre étude. Nous concluons ce chapitre en démontrant les limitations de ces travaux.

Le chapitre 4 se concentre sur la présentation et sur l'analyse de notre contribution. Nous proposons KmMR comme une plateforme d'orchestration de conteneurs Docker tolérante à toute sorte de faute. Nous évaluons ensuite les performances de notre solution et nous la comparons à la plateforme Kubernetes multi-maîtres classique (KmMC).

Enfin, le dernier chapitre conclut ce mémoire et expose nos travaux futurs.

CHAPITRE I

LA CONTENEURISATION

1.1 Introduction

Afin de mieux cerner l'étendu de notre problématique, nous présentons dans ce premier chapitre les concepts liés à l'infonuagique (*cloud computing*), à la virtualisation traditionnelle et à la conteneurisation. L'objectif est de les décrire et expliquer les liens qui existent entre eux.

1.2 L'infonuagique (le *cloud computing*)

Le *cloud computing* (Qian *et al.*, 2009), ou l'infonuagique, est une infrastructure informatique distribuée de processeurs et de mémoires, accessible à des usagers via un réseau, généralement l'Internet. Ainsi, n'importe quel matériel connecté (ordinateur de bureau, portable, tablette, téléphone, ou autre objet connecté) peut servir d'outil pour exécuter diverses applications ou consulter des données hébergées sur des serveurs *cloud*. Le *NIST* (*National Institute of Standards and Technology*) définit ce concept comme étant « l'accès via le réseau, à la demande et en libre-service, à des ressources informatiques virtualisées et partagées » (Mell *et al.*, 2011). L'informatique nuagique permet donc de décharger les serveurs et les services traditionnellement localisés sur des serveurs locaux vers un ou plusieurs fournisseurs tiers dotés de la compétence et des ressources requises au maintien

de cette architecture réseau et de ces services.

Le *cloud computing* privilégie la haute disponibilité. Ses services doivent être accessibles depuis partout et à tout moment (*access anywhere anytime-AAA*). Afin de garantir cette haute disponibilité, le *NIST* a défini pour le *cloud* cinq caractéristiques essentielles, trois niveaux de service et quatre modèles de déploiement dans « *The Nist definition of cloud computing* » (Mell *et al.*, 2011).

1.2.1 Les caractéristiques du *cloud*

Le *cloud* se distingue des environnements informatiques classiques par les caractéristiques suivantes :

- Service à la demande : les services sont fournis au client automatiquement et sans intervention humaine ;
- Élasticité rapide : les ressources de stockage, de calcul, et de bande passante peuvent être rapidement ajustées statiquement ou dynamiquement en fonction des besoins immédiats de chaque client ;
- Mutualisation : les différents clients bénéficient de ressources de serveurs allouées de manière dynamique et rapide dans le cadre d'un modèle de *cloud* communautaire ;
- Résilience : le *cloud computing* se doit d'apporter des mécanismes hétérogènes où les technologies présentes supportent une multitude de clients légers (ordiphones, tablettes) et de clients lourds (ordinateurs) ;
- Paiement à l'utilisation : contrairement aux centres d'hébergement web classiques dans lesquels le paiement se fait à l'avance sous forme de forfait, le *cloud* propose une facturation à l'usage. Les services se facturent en fonction

de l'utilisation des ressources et en toute transparence pour le client et le fournisseur de services infonuagiques. Selon Tchana (2011), cette facturation peut être implémentée de deux façons. La première consiste à facturer à l'entreprise la durée d'utilisation d'un ensemble de ressources quelle que soit l'utilisation effective réalisée. Par exemple, soit r l'ensemble des ressources réservées par l'entreprise. Soient t_d et t_f respectivement l'instant de début et de fin d'utilisation des ressources. Soit C_u le coût d'utilisation d'une ressource durant une unité de temps. Ainsi, le coût total d'utilisation de l'ensemble des ressources r est :

$$C_r = C_u * r * (t_f - t_d) \tag{1.1}$$

Quant à la deuxième façon, elle est beaucoup plus précise que la première. Elle consiste à facturer les véritables instants pendant lesquels les ressources ont été utilisées. En partant des mêmes paramètres que précédemment, soit $T = \{t_{u1}, t_{u2}, ..., t_{un}\}$ avec $t_{ui} \in [t_d, t_f]$ $(1 \leq i \leq n)$ l'ensemble des unités de temps pendant lesquelles les ressources ont été réellement utilisées. Dans ce cas, le coût total d'utilisation de l'ensemble des ressources r est :

$$C_r = C_u * r * \sum_{i=0}^{n} t_{ui} \tag{1.2}$$

1.2.2 Les niveaux de service

Le terme *cloud computing* est apparu au début des années 2000 mais, historiquement, le concept de l'informatique en tant que service existe depuis bien plus longtemps. Il remonte aux années 1960, quand des sociétés de services informatiques permettaient aux entreprises de louer du temps sur une plateforme matérielle, au lieu de devoir en acheter une elles-mêmes. Les années 1990 voient l'explosion de l'Internet ; l'accès aux services peut s'effectuer via le réseau. Le *cloud computing* distribue à présent ces services sous la dénomination *Software as a Service*

(*SaaS*). Le concept n'utilise plus de clients lourds comme des logiciels dédiés, mais un simple navigateur Internet (Schmitt, 2014).

Le *cloud* peut offrir trois niveaux de service : le *IaaS*, le *PaaS* et le *SaaS*.
Dans le niveau *IaaS* (*Infrastructure as a Service*), le matériel est l'unique élément à être décentralisé. Les clients s'abstraient donc de toute contraintes de gestion du matériel physique.
Le niveau PaaS (*Plateform as a Service*) constitue sur le niveau supérieur à *IaaS*. Il offre aux entreprises un environnement de développement ou d'hébergement de leurs applications. L'entreprise client ne se soucie plus de l'infrastructure sous-jacente.
Le niveau le plus commun est le *SaaS* (*Software as a Service*). Il qui apporte aux clients une application (logiciel) à la demande et sous la forme d'un service prêt à l'emploi. Les utilisateurs du service ne se préoccupent plus de l'architecture matérielle (installation, maintenance) et logicielle (mises à jour) sous-jacente.
La figure 1.1 montre l'interaction entre le fournisseur du *cloud* et le client en fonction des niveaux de service. Ces derniers doivent être déployés sur des infrastructures qui possèdent les caractéristiques essentielles présentées dans la section 1.2.1 pour être considérées comme étant des plateformes de *cloud computing*.

1.2.3 Les modèles de déploiement

Les trois niveaux de service cités dans la section précédente peuvent être déployés chez le client via :

1. un *cloud* public géré par un prestataire de services privé, par exemple Google, Amazon, etc. ;

2. un *cloud* privé prévu pour les besoins de l'entreprise. Il y a les *clouds* privés

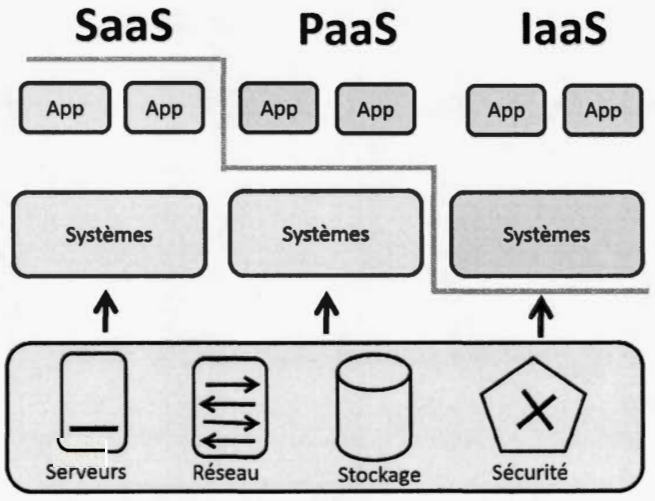

Figure 1.1 Les niveaux de service dans le *cloud*.

internes, où l'entreprise bâtit son propre centre de données pour ses propres besoins, et les *clouds* privés externes, où un fournisseur réserve une partie de ses serveurs à l'usage exclusif d'un client. Dans ce dernier cas, l'infrastructure est entièrement dédiée au client et accessible via des réseaux sécurisés de type *VPN* (*Virtual Private Network*).

3. un *cloud* communautaire où l'infrastructure est partagée par plusieurs organisations qui ont des intérêts communs (par exemple des exigences de sécurité, de conformité, etc.). Comme le *cloud* privé, il peut être géré par les organisations elles-mêmes ou par un tiers.

4. un *cloud* hybride où l'entreprise peut gérer des ressources internes via deux nuages ou plus (privé, communautaire ou public), qui restent des entités uniques, mais qui sont liées par une technologie normalisée ou propriétaire. Ceci permet la portabilité des données ou des applications.

1.2.4 Le *cloud* face au défi de l'isolation

Comme dans toute infrastructure informatique, la mutualisation de ressources dans le *cloud* implique la mise en place de divers mécanismes de sécurité, de comptage de ressources, de gestion des conflits d'accès, etc. L'isolation a été proposée afin de garantir la mutualisation des ressources. Isoler revient à gérer les contraintes suivantes :

1. Garantir au client l'exclusivité d'accès à un ensemble de ressources durant toute sa présence dans le *cloud* (malgré la mutualisation).
2. Donner à chaque client du *cloud* l'illusion d'être le seul utilisateur. Rien ne doit le laisser présager la présence d'autres utilisateurs ou applications.
3. Assurer la non-monopolisation des ressources globales du *cloud* par un seul client.
4. Assurer la non-violation des espaces utilisateurs dans le *cloud.*

1.3 La virtualisation

La mise en place de l'isolation a rendue possible effectuée grâce à l'introduction des techniques de virtualisation dans la création du *cloud.* Toutefois, il existe plusieurs manières de mettre en œuvre l'isolation :

- La première façon consiste à allouer la ressource matérielle entière à une entreprise même si celle-ci ne souscrit qu'à une fraction de cette ressource. Cette méthode ne permet qu'une résolution partielle des problèmes d'isolation. En effet, certaines ressources comme le réseau et sa bande passante demeurent partagées entre les entreprises dans le *cloud.* À moins que le

fournisseur alloue exclusivement à chaque entreprise des équipements et une bande passante dédiée (ce qui n'est pas raisonnable), il serait impossible avec cette méthode d'éviter des situations de monopolisation des ressources par une entreprise. Cette solution matérielle doit être complétée par une solution logicielle.

- La seconde méthode consiste à laisser la responsabilité aux entreprises d'implanter les mécanismes d'isolation. Cette méthode n'est pas envisageable dans la mesure où le *cloud* ne dispose d'aucun moyen d'introspection des applications des entreprises afin de s'assurer de l'implantation de ces mécanismes.

- La dernière méthode est hybride (matérielle et logicielle). Tout en implémentant les mécanismes d'isolation, elle donne l'illusion aux clients d'avoir un accès direct et exclusif à la ressource matérielle. Parallèlement, elle garantie au *cloud* un accès contrôlé des clients aux ressources matérielles. Il s'agit de l'isolation par virtualisation.

1.3.1 Définition de la virtualisation

La virtualisation (Huang *et al.*, 2006; Goth, 2007; Danny, 2013; Liu et Zhao, 2014) consiste à créer une version virtuelle d'un dispositif ou d'une ressource, comme un système d'exploitation, un serveur, un dispositif de stockage ou une ressource réseau. Elle peut donc être considérée comme l'abstraction physique des ressources informatiques. En d'autres termes, les ressources physiques allouées à une machine virtuelle sont abstraites à partir de leurs équivalentes physiques. Chaque dispositif virtuel, qu'il s'agisse d'un disque, d'une interface réseau, d'un réseau local, d'un commutateur, d'un processeur ou d'une mémoire, correspond à une ressource physique sur un système informatique. Les machines virtuelles hébergées par la

machine hôte sont donc perçues par ce dernier comme des applications auxquelles il est nécessaire de dédier ou distribuer ses ressources.

Cependant, il existe plusieurs types de virtualisation (Hess et Newman, 2010). Chaque type correspond à un cas d'utilisation spécifique, comme cela sera présenté dans la section suivante.

1.3.2 Types de virtualisation

Il existe de nombreux domaines d'application à la virtualisation, s'agissant généralement de la virtualisation du stockage, du réseau, ou du serveur.

- La virtualisation du stockage. Elle est obtenue en isolant les données de leur localisation physique. Elle permet de masquer les spécificités physiques des unités de stockage. Vues de l'extérieur, les unités de stockage sont perçues comme étant un volume unique. La virtualisation du stockage est couramment utilisée dans un réseau de stockage (*Storage Area Network-SAN*, *Network Attached Storage-NAS*).

- La virtualisation du réseau. Elle consiste à combiner des ressources réseaux physiques (commutateurs, routeurs, etc.) et logiques (bande passante, adressage, etc.) pour créer des réseaux virtuels, découplés du matériel réseau sous-jacent. L'objectif est de garantir aux systèmes et aux utilisateurs un partage efficace et sécurisé des ressources physiques du réseau.

- La virtualisation du serveur. Ce type de virtualisation se définit comme l'ensemble des techniques matérielles ou logicielles qui permettent de faire fonctionner plusieurs systèmes, serveurs ou applications sur une seule machine en donnant l'illusion qu'ils fonctionnent sur différentes machines.

Dans la suite de ce mémoire, nous nous intéressons à la virtualisation du serveur. Plus précisément, nous étudions ci-après les deux techniques de virtualisation du serveur, à savoir la virtualisation système et la virtualisation applicative.

1.3.3 La virtualisation système

Cette technique de virtualisation du serveur fait référence à l'utilisation d'un hyperviseur pour permettre à la machine hôte d'exécuter plusieurs instances virtuelles en même temps. Ces dernières sont communément appelées des *VM*s (Virtual Machines), tandis que l'hyperviseur est connu sous le nom de *VMM* (*Virtual Machines Monitor*), c'est-à-dire gestionnaire de *VM*. Ces nouvelles notions sont définies comme suit :

1. Machine virtuelle (*VM*) : il s'agit d'une version virtuelle d'un matériel dédié avec son propre système d'exploitation (*Operating System-OS*). Comme dans une machine réelle, chaque *VM* conserve un fonctionnement normal, défini comme suit : une *VM* gère les accès mémoire (*Random Access Memory-RAM*), disque, réseau, processeurs (*Central Processing Unit-CPU*) et autres périphériques de ses processus.

2. Hyperviseur (*VMM*) : il implante les mécanismes d'isolation et de partage des ressources matérielles. Il est capable de démarrer simultanément plusieurs machines virtuelles de différents types (Linux, Mac ou Windows) sur le même matériel. Quoique les *VM* gèrent les accès aux ressources, aucun accès aux ressources n'est possible sans l'aval du *VMM*. Ce dernier décide notamment des attributions de temps processeurs aux machines virtuelles. Quant à la communication avec l'extérieur, l'hyperviseur peut fournir plusieurs techniques pour rendre accessible ou non les *VM*. Il peut réaliser cette tâche par l'assignation d'adresses *IP* (*Internet Protocol*) et par im-

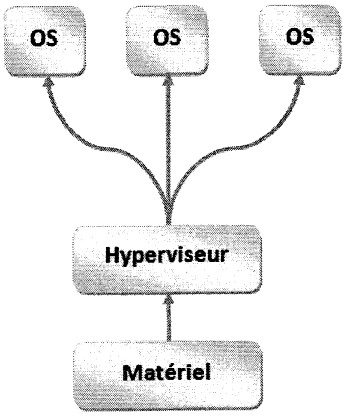

Figure 1.2 Hyperviseur de type 1.

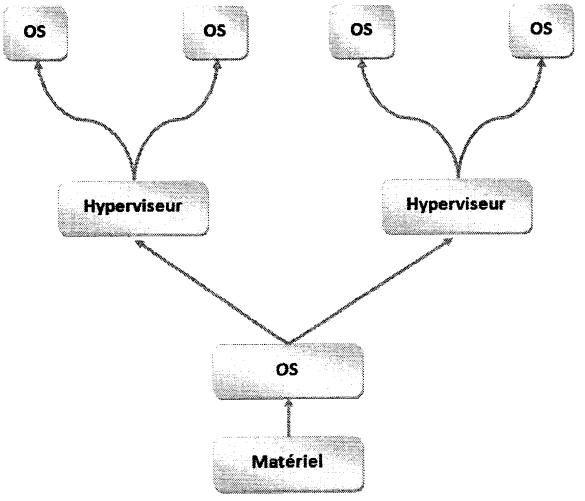

Figure 1.3 Hyperviseur de type 2.

plantation de mécanismes d'accès réseau aux *VM* (par routage, filtrage de communication, etc.). On peut distinguer deux types d'hyperviseur (Soriga et Barbulescu, 2013; Hwang *et al.*, 2013) :

- Hyperviseur de type 1 (natif) : Il s'installe directement sur la couche physique du matériel du serveur. Ainsi, au démarrage de ce dernier, ce *VMM* prend le contrôle exclusif du matériel. Hyper-V, Xen, VMware, ESXi, KVM sont des hyperviseurs de type 1 (Fayyad-Kazan *et al.*, 2013).
- Hyperviseur de type 2 (hébergé) : Comparable à un émulateur, il s'installe sur l'*OS* installé sur le matériel du serveur. Il démarre comme un logiciel installé sur le système d'exploitation. Comme exemples, on peut citer *Oracle VM* (*Virtual Box*), *Workstation*, *Parallels* et *Fusion*.

Les figures 1.2 et 1.3 présentent les deux types d'hyperviseur de la virtualisation système.

1.3.4 La virtualisation applicative

Contrairement à la virtualisation système, le principe de cette technique est de mettre en œuvre une couche de virtualisation sous forme d'une application qui elle-même crée des instances virtuelles et les isole des spécificités de la machine hôte, c'est-à-dire de celles de son architecture ou de son système d'exploitation. Ceci permet au concepteur de l'application de ne pas avoir à rédiger plusieurs versions de son logiciel pour qu'il soit exécutable dans tous les environnements. Les machines virtuelles applicatives les plus connues sont la *JVM* (*Java Virtual Machine*) et les conteneurs.

- *Java Virtual Machine* (*JVM*) : Elle est gratuite, propriétaire jusqu'à la version 6 (stable) et libre sous la *GPL* (*General Public License*) à partir de la version 7. La *JVM* permet aux applications Java compilées de produire les mêmes résultats quelle que soit la plateforme, tant que celle-ci est pourvue de la machine virtuelle Java adéquate (Czajkowski, 2000).

- Conteneurs : Mieux que la *JVM*, les conteneurs fonctionnent avec toutes les technologies compatibles avec Linux (par exemple Java, Python, Ruby, Nodejs et Go). Les conteneurs sont isolés les uns des autres et peuvent optimiser l'utilisation des ressources (*CPU*, *RAM*, stockage) de la machine hôte.

1.4 La virtualisation par conteneurs

1.4.1 Définition

La virtualisation par conteneurs (également connue sous le nom de conteneurisation) constitue une alternative à la virtualisation système. Il s'agit d'une approche

qui s'appuie directement sur les fonctionnalités du noyau (*Kernel*) pour créer des environnements virtuels (*Virtual Environment-VE*) isolés les uns des autres. Ces *VE* sont appelés conteneurs, tandis que les fonctionnalités fournies par le noyau du système d'exploitation sont les groupes de contrôles (*cgroups*) et les espaces de noms (*namespaces*) (Pahl, 2015; Moga *et al.*, 2016). Les *namespaces* permettent de contrôler et de limiter la quantité de ressources utilisée pour un processus, tandis que les *cgroups* gèrent les ressources d'un groupe de processus.
Un conteneur fournit donc les ressources nécessaires pour exécuter des applications comme si ces dernières étaient les seuls processus en cours d'exécution dans le système d'exploitation de la machine hôte.

1.4.2 Vue d'ensemble des conteneurs

Les conteneurs constituent une solution pour la gestion des ressources entre les applications (Moga *et al.*, 2016). Le terme est dérivé des conteneurs d'expédition, une méthode standard pour stocker et expédier tout type de cargaison.

Le concept de base des conteneurs est né en 1979 avec la fonction Unix *chroot*. Cette dernière permet de créer plusieurs sessions isolées sur une même machine, de façon à partager parallèlement les ressources de processus et de stockage de la machine. Vu cette caractéristique particulière de la fonction *chroot*, celle-ci peut être considérée comme étant le précurseur de la conteneurisation.
Introduite en l'année 2000, la solution *FreeBSD Jail* a permis de partitionner le système d'exploitation de la machine hôte en des sous-systèmes isolés. Par rapport à chroot, la fonctionnalité supplémentaire de l'environnement *FreeBSD* était la possibilité d'isoler également le système de fichiers, les utilisateurs et le réseau, en les rendant plus sécurisés. Mais cette solution présente des difficultés en termes de mise en œuvre. L'année suivante a vu le développement d'une solution sem-

blable à *FreeBSD Jail* appelée *Linux VServer*. Cette dernière utilise un système *VPS* (*Virtual Private Server*) pour partitionner les ressources d'une machine en des contextes de sécurité (Beloussov *et al.*, 2008). Mettant les bases d'une utilisation optimale des fonctionnalités du noyau du système d'exploitation, le projet *VServer* conduira, en 2008, au projet *LinuX Container* (*LXC*). Dans ce nouveau projet, chaque distribution Linux importe ses propres bibliothèques qui supportent les conteneurs *LXC*, mais qui ne sont pas compatibles entre elles. Cependant, c'est à partir de 2013 que Docker (Eberbach et Reuter, 2015) a rendu simple le développement et le déploiement des conteneurs tout en les standardisant. Docker résout un problème de longue date dans le *cloud computing* : la portabilité des applications (Abdelbaky *et al.*, 2015).

Vu l'importance de cette propriété de portabilité dans le contexte du *cloud computing*, nous nous concentrons dans la suite sur la technologie Docker.

1.4.3 Les conteneurs Docker

Docker est l'incarnation la plus moderne de la conteneurisation (Madhumathi, 2018). Il s'agit d'un projet ouvert développé par la société *Dotcloud* (renommée par la suite Docker) sur la base de *LXC*. Fondamentalement, Docker étend *LXC* avec le *Docker daemon* (Bernstein, 2014; Dua *et al.*, 2014; Joy, 2015a). Ce dernier est une *API* (*Application Programming Interface*) installée entre le noyau du système d'exploitation et les *VEs*, qui, ensemble, exécutent des processus isolés : processeur, mémoire, entrées/sorties (E/S), réseau, etc. (Combe *et al.*, 2016). Depuis la version 0.9 en 2014, Docker n'utilise plus *LXC* comme environnement d'exécution par défaut (Bui, 2015). Il l'a remplacé par sa propre bibliothèque Libcontainer (Docker, 2016b), implémentée en Go, un langage de programmation orientée service. Docker est un écosystème complexe mais intuitif pour le déve-

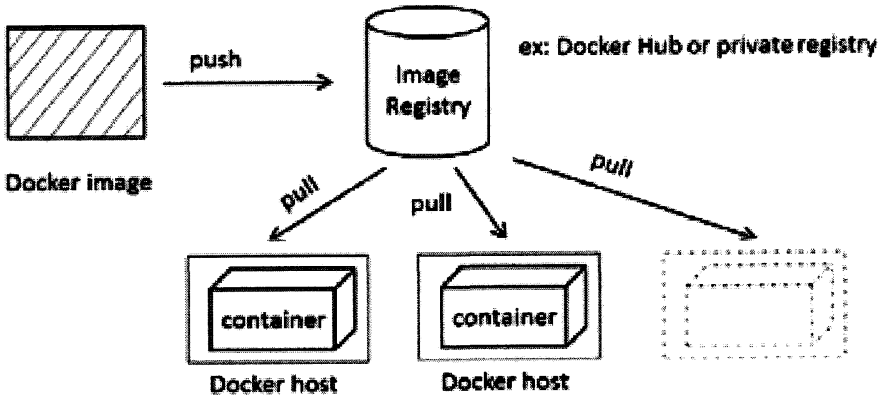

Figure 1.4 Architecture de base de Docker.

loppement des conteneurs, Il est riche en fonctionnalités, notamment son système d'image, ses registres et son interface de lignes de commande. La figure 1.4 présente l'architecture de base de Docker (Nguyen et Bein, 2017).

1.5 Comparaison entre machine virtuelle et conteneur Docker

La virtualisation traditionnelle utilise un hyperviseur pour créer de nouvelles machines virtuelles (*VM*) et pour assurer l'isolation entre elles. Toutefois, la conteneurisation Docker ne nécessite que l'installation du logiciel Docker sur le noyau du système d'exploitation de la machine hôte. Néanmoins, il existe une certaine similitude entre ces deux instances virtuelles : elles sont toutes des systèmes autonomes qui, en réalité, utilisent un système supérieur (celui de la machine hôte) pour réaliser leurs tâches. Cependant, la principale différence entre elles est que la *VM* doit contenir tout un système d'exploitation (un système invité) alors que les conteneurs se contentent d'utiliser le système d'exploitation sur lequel ils s'exé-

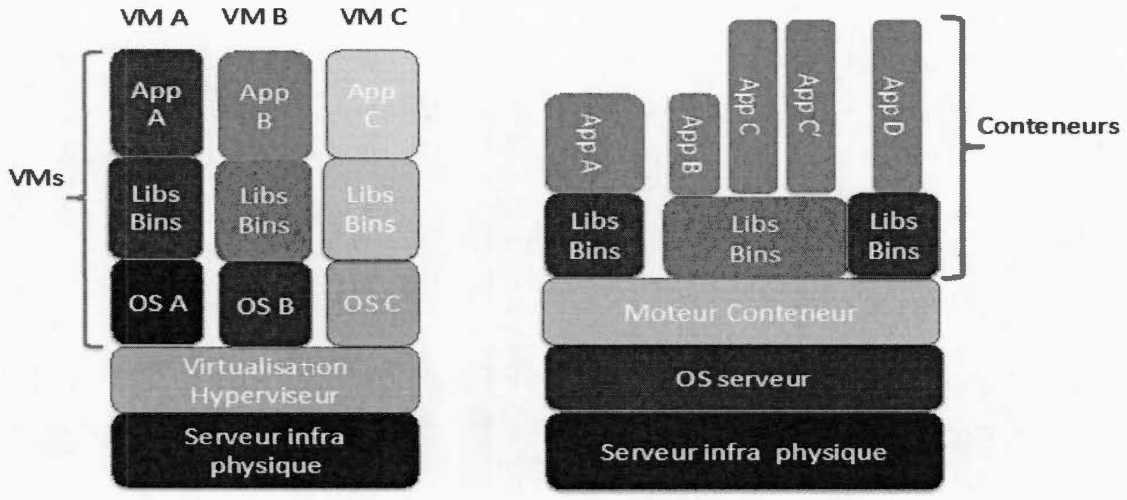

Figure 1.5 Structures d'une machine virtuelle et d'un conteneur Docker.

cutent. La figure 1.5 montre la structure d'une *VM* et celle d'un conteneur Docker. Le conteneur Docker représente une amélioration des capacités de *LXC*. Ainsi, il n'émule pas l'environnement matériel hôte (Felter *et al.*, 2015; Naik, 2016). Il peut être déployé facilement sur une machine virtuelle comme sur une machine physique.

Comme le montre le tableau 1.1 ci-après, un conteneur peut être créé et détruit presque en temps réel et introduisant ainsi une surcharge de tâches négligeable vis-à-vis de l'utilisation des ressources (*CPU*, *RAM*, E/S) de la mémoire de la machine hôte (Felter *et al.*, 2015; Dua *et al.*, 2014; Joy, 2015b; Sharma *et al.*, 2016). Par rapport aux *VM*, les conteneurs présentent des avantages en termes de gestion de réseau, de mémoire vive, de stockage, de vitesse de démarrage, de déploiement et de migration (Joy, 2015a; Amaral *et al.*, 2015; Tay *et al.*, 2017). Par contre, ces derniers souffrent du faible isolement de la machine hôte. En effet, si le conteneur Docker est compromis, alors l'attaquant peut obtenir un accès complet au système

d'exploitation de la machine hôte (Li et Kanso, 2015; Manu *et al.*, 2016b,a). D'où le besoin d'environnements robustes et sécurisés pour les conteneurs Docker.

1.6 Conclusion

Dans ce chapitre, nous avons présenté les notions de base associées à la conteneurisation. Plus précisément, nous avons défini le domaine d'application de la conteneurisation, qui n'est autre que le *cloud computing*.
L'étude comparative entre l'utilisation des machines virtuelles et les conteneurs Docker montre que ces derniers ont un avantage inhérent sur les *VM*s en raison de l'amélioration de plusieurs métriques de performances (ex : *CPU*, *RAM*, stockage, temps de démarrage, etc.).
En général, Docker permet de créer et de déployer des logiciels (services) sous forme de conteneurs dans une machine physique ou virtuelle grâce à une collection d'outils gratuits. Toutefois, il n'est pas adapté au déploiement de plusieurs conteneurs sur plusieurs machines distribuées (système distribué) (Li et Kanso, 2015). Des plateformes d'orchestration de conteneurs ont été créées afin de remédier à cette limitation. Elles seront discutées dans le chapitre suivant.

H

Tableau 1.1 Caractéristiques des technologies de virtualisation.

Paramètres	Machine Virtuelle	Conteneur Docker
OS invité	Chaque *VM* virtualise le matériel hôte et charge son propre *OS*	Aucun conteneur n'émule le matériel hôte. Il utilise directement l'*OS* hôte.
Communication	Via des périphériques Ethernet	Via les mécanismes *IPC* (*Inter-Process Communication*) standard tels que les sockets, les pipes, la mémoire partagée, etc
Utilisation des ressources (*CPU* et *RAM*) de la machine hôte	Forte utilisation	Utilisation presque native
Temps de démarrage	La *VM* prend quelques minutes pour démarrer	Ne prend que quelques secondes pour démarrer
Stockage	Nécessite beaucoup plus de stockage car son *OS* tout entier et ses programmes associés doivent être installés et exécutés	Utilise moins d'espace de stockage car le système d'exploitation de base est partagé
Isolation	Le partage de librairies, fichiers, etc. entre *VM* est impossible	Les sous-répertoires peuvent être montés de manière transparente et peuvent être partagés
Sécurité	Dépend de la configuration de l'hyperviseur	Nécessite un contrôle des accès

CHAPITRE II

PLATEFORMES D'ORCHESTRATION DE CONTENEURS DOCKER

2.1 Introduction

Les conteneurs Docker constituent une méthode de virtualisation légère au niveau du système d'exploitation qui permet de lancer une application et ses dépendances à travers un ensemble de processus isolés du reste du système. Cette méthode permet d'assurer le déploiement rapide et stable des applications dans n'importe quel environnement informatique.

En plein essor depuis quelques années, les conteneurs Docker ont modifié la façon dont nous développons, déployons et maintenons des logiciels. Leur légèreté et flexibilité ont favorisé l'apparition de nouvelles formes d'architectures qui organisent les applications au sein de conteneurs Docker, prêts à être déployés sur un *cluster* (groupe) de machines virtuelles ou physiques. Toutefois, cette nouvelle approche requiert des nouveaux outils d'orchestration de conteneurs Docker (Bernstein, 2014).

2.2 Caractéristiques d'un orchestrateur de conteneurs

Manipuler quelques conteneurs Docker sur une seule machine est une tâche facile. Lorsqu'il s'agit de faire passer ces conteneurs en production (sur un ensemble

d'hôtes distribués), de nombreuses questions se posent :

- Comment gérer les déploiements et leurs emplacements ?
- Comment gérer l'équilibrage des charges ?
- Comment gérer la communication entre les conteneurs ?
- Comment gérer la découverte de services ?
- Comment gérer les mises à jour ?
- Comment gérer la montée en échelle ?
- Comment gérer le stockage nécessaire à la persistance des données ?
- Comment gérer la configuration et les secrets ?
- Mais aussi, comment gérer les dysfonctionnements ?

Tout gérer de manière manuelle n'est pas concevable, car il serait difficile d'assurer la viabilité, la maintenance et la pérennité du système. Un outil autonome capable de répondre à ces besoins est donc requis. Cet outil constituera la plateforme d'orchestration de conteneurs.

Il existe plusieurs orchestrateurs, par exemple, Fleet (CoreOS, 2014), Mesos Apache (2014), Swarm (Luzzardi et Victor, 2014) et Kubernetes (K8s, 2014). Toutefois, dans la suite de ce chapitre, nous nous intéressons à Swarm et à Kubernetes. Ces derniers se présentent comme étant des solutions stables, gratuites et capables d'automatiser le déploiement, la migration, la surveillance, la mise en réseau, l'extensibilité et la disponibilité des applications basées sur la technologie des conteneurs Docker (Peinl *et al.*, 2016; Burns *et al.*, 2016). Cependant, chacun d'eux propose ses propres architectures et méthodologies pour la gestion de conteneurs.

Leurs définitions, architectures et fonctionnements seront détaillés ci-après.

2.3 L'orchestrateur Swarm

2.3.1 Définition

Swarm (Luzzardi et Victor, 2014; Soppelsa et Kaewkasi, 2016) est le premier gestionnaire de conteneurs Docker, lancé par la société Docker en 2014. Il est devenu une fonctionnalité native et intégré au démon Docker depuis la version 1.12 en 2016. L'activation du mode Swarm peut pallier les lacunes de Docker en termes de déploiement, d'exploitation et de gestion sur plusieurs hôtes (Naik, 2016).

2.3.2 Architecture de Swarm

L'orchestration avec Swarm est axée sur une simple architecture en *cluster*(s)(Luzzardi et Victor, 2014; Cérin *et al.*, 2017; Kovács *et al.*, 2018). Un *cluster* Swarm est constitué d'un nœud gestionnaire (manager Swarm) et de nœud(s) de travail (*worker* Swarm) comme le présente la figure 2.1.

- Le *manager* Swarm : Il effectue les fonctions d'orchestration et de gestion nécessaires à la conservation de l'état souhaité du *cluster*. Il planifie et déploie les conteneurs vers les nœuds de travail. L'initialisation du *cluster* se fait au niveau du *manager*.

- Les *workers* Swarm : Ils reçoivent et exécutent les conteneurs répartis à partir du *manager*. Un agent Swarm s'exécute sur chaque *worker* et rend compte des tâches qui lui sont assignées.

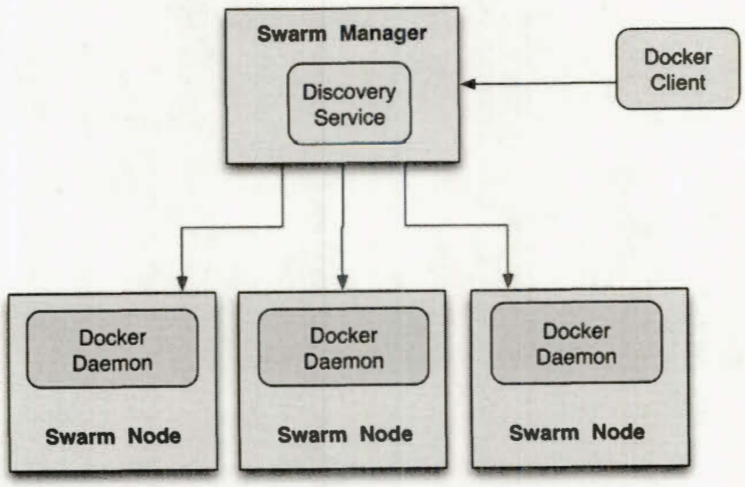

Figure 2.1 Architecture de Swarm.

2.3.3 Méthodologies de placement de conteneurs

Docker Swarm utilise des stratégies de classement pour déployer un conteneur sur un nœud de travail particulier et des filtres pour restreindre le domaine des nœuds. Pour comprendre ces critères, nous modélisons le déploiement d'un conteneur $c_{t,m}$ qui requiert t temps CPU et m quantité RAM.

Considérons un *cluster* Swarm constitué d'un nœud *manager* et d'un ensemble de nœuds de travail $W = \{W_1, W_2, ..., W_n\}$.

Afin de mieux comprendre le fonctionnement de Swarm, nous modélisons à travers le tableau 2.1 les ressources de chaque nœud de travail W_i ($1 \leq i \leq n$).

Pour déployer ce conteneur, le nœud *manager* doit sélectionner un *worker*. Pour ce faire, le *manager* applique la stratégie et les filtres définis sur la liste des nœuds de travail qui ont les ressources demandées par le conteneur, c'est-à-dire les W_i

Tableau 2.1 Modélisation des ressources du *worker* Swarm W_i $(1 \leq i \leq n)$.

Ressources	Paramètres	Description
Quantité *RAM*	m_i^T	La quantité totale *RAM* de W_i
	m_i^d	La quantité *RAM* disponible dans W_i
Temps *CPU*	t_i^T	Le temps total *CPU* de W_i
	t_i^d %	Le pourcentage *CPU* disponible dans W_i
Conteneurs	n_i^T	Le nombre total de conteneurs instanciés dans W_i

qui vérifient les contraintes suivantes :

$$t_i^d \geq t \tag{2.1}$$

$$m_i^d \geq m \tag{2.2}$$

Les équations 2.1 et 2.2 déterminent si le nœud W_i satisfait les exigences du conteneur $c_{t,m}$ en termes de temps *CPU* et de mémoire vive, respectivement.

1. Stratégies de classement dans Swarm (Docker, 2018) : Elles sont au nombre de trois : *spread*, *binpack* et *random*.

   Les deux premières stratégies calculent le rang en fonction des ressources (*CPU* et *RAM*) disponibles et du nombre total de conteneurs (actifs et inactifs) placés dans le nœud. Par contre, la troisième sélectionne au hasard un nœud de travail parmi ceux qui satisfont les contraintes 2.1 et 2.2. *spread*, la stratégie par défaut, sélectionne le nœud ayant le plus de ressources disponibles, alors que la stratégie *binpack* choisit le nœud le plus saturé.

2. Les filtres dans Swarm (Docker, 2018) : Au nombre de cinq, les filtres permettent d'ajouter d'autres types de contraintes sur la stratégie de classement. Ils peuvent être répartis en deux catégories : ceux qui se basent sur les propriétés du nœud de travail et ceux qui se basent sur les propriétés du conteneur à déployer.

   Les filtres du nœud sont des contraintes telles que son système d'exploitation ou son état. Par contre, les filtres de conteneurs se focalisent sur la relation entre le conteneur à déployer et les conteneurs déployés dans le passé. Cette relation peut être leur affinité, leur dépendance et le port à utiliser. L'affinité permet de déployer le conteneur à côté de ceux qui portent, par exemple, la même image Docker que lui. La dépendance déploie le conteneur à côté de ceux dont il dépend. Enfin, le filtre de port permet d'instancier le conteneur à un nœud où ses conteneurs n'utilisent pas le port spécifié.

2.4 L'orchestrateur Kubernetes

2.4.1 Définition

Kubernetes, K8s en abrégé (Bernstein, 2014; Rensin, 2015; Burns *et al.*, 2016; Google, 2018), est un projet initié par Google en 2014 quand il a perçu l'avantage des conteneurs Docker par rapport à la virtualisation traditionnelle. Il s'agit d'une innovation après plus d'une décennie d'expérience avec Borg (Burns *et al.*, 2016), son premier gestionnaire de conteneurs. L'orchestrateur Kubernetes automatise le déploiement et la gestion d'applications conteneurisées à grande échelle. La plateforme K8s permet d'exécuter et de coordonner des conteneurs sur un ensemble de machines physiques et/ou virtuelles. Elle est conçue pour gérer entièrement le cycle de vie des applications conteneurisées en utilisant des méthodes de prédictibilité, d'extensibilité et de haute disponibilité (Kratzke et Quint, 2017). D'autres

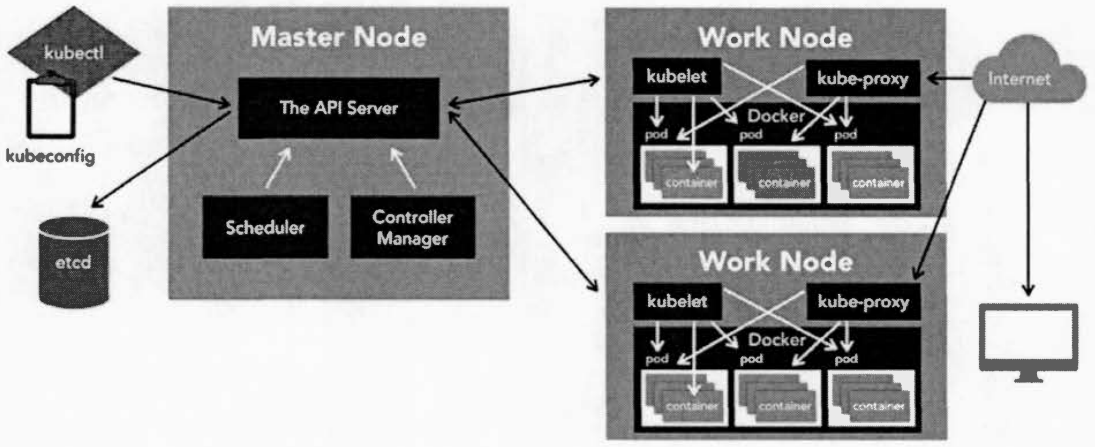

Figure 2.2 Architecture de Kubernetes.

entreprises, telles que Red Hat, contribuent également au développement de Kubernetes, disponible sur *GitHub* (K8s, 2014), la plateforme de partage de code la plus populaire au monde.

2.4.2 Architecture de Kubernetes

Comme le montre la figure 2.2, Kubernetes orchestre les conteneurs Docker suivant une architecture maître/esclaves (Bila *et al.*, 2017) constituée d'un nœud maître et de nœud(s) de travail (*minion*). Un agent K8s est installé sur chaque nœud client (les exécuteurs) et renvoie des rapports au nœud maître. Ce dernier est le responsable du *cluster* Kubernetes.

- Le maître Kubernetes : Ce nœud est responsable de la gestion globale et de la disponibilité du *cluster* Kubernetes. Ses composants, c'est-à-dire le serveur *API*, le contrôleur et le planificateur prennent en charge les tâches d'interaction, de surveillance et de planification au sein du *cluster*. Le serveur *API* fournit l'interface à l'état partagé du *cluster* via laquelle tous les autres com-

posants interagissent. Le contrôleur surveille l'état partagé du *cluster* via le serveur *API* et apporte des modifications afin de ramener le *cluster* de l'état actuel instable vers un état stable. Le planificateur gère la charge du *cluster*. Il prend en compte les besoins en ressources individuels et collectifs, les exigences de qualité de service, les contraintes matérielles/logicielles, les politiques, etc. Les données du *cluster* Kubernetes sont stockées dans une base de données, (etcd (CoreOS, 2018) par exemple). L'administration du *cluster* se fait donc au niveau du maître via la *kubectl*, l'interface de ligne de commande de K8s. La *kubectl* a un fichier de configuration appelé $kubeconfig$. Ce dernier contient les informations de configuration et d'authentification pour accéder au serveur *API*.

- Les *minions* Kubernetes : Ceux-ci sont les nœuds sur lesquels s'exécutent les applications conteneurisées. D'une part, les nœuds clients communiquent avec le nœud maître via leur *kubelet*. Le *kubelet* reçoit les ordres du nœud maître et les exécute via son moteur Docker. Il rapporte également l'état du nœud client au serveur *API* du nœud maître. D'autre part, le *kube-proxy* s'exécute sur chaque nœud client afin de gérer l'accès des utilisateurs finaux aux services déployés. Chaque service est compilé dans un ou plusieurs *pod*. Un *pod* est un ensemble logique d'un ou plusieurs conteneurs. Il s'agit de la plus petite unité pouvant être programmée en tant que déploiement dans Kubernetes. Les conteneurs d'un même *pod* partagent, entre autres, la capacité de stockage et l'adresse IP sur le réseau.

Le placement de ces *pods* se fait en suivant une stratégie propre à Kubernetes. Elle sera décrite au point suivant.

2.4.3 Méthodologies de placement des *pods*

Considérons un *cluster* Kubernetes constitué d'un nœud maître et d'un ensemble fini de *minions* $M = \{M_1, M_2, ..., M_n\}$.

Soit $pod_{t,m,p,v}$, un *pod* à déployer dans le *cluster* et qui demande t temps de CPU, m quantité RAM, un port spécifique p et une quantité v de volume de stockage.

Afin de sélectionner le *minion* sur lequel le *pod* va être instancié, le nœud maître K8s procède en deux étapes. D'abord, il filtre les *minions*. Ensuite, il classe les *minions* restants afin de déterminer le mieux adapté pour le *pod*.

1. Filtrage des *minions* (Kubernetes, 2014b; Medel *et al.*, 2018) : Cette opération a pour but de filtrer les nœuds qui ne répondent pas aux exigences du $pod_{t,m,p,v}$. Kubernetes utilise plusieurs prédicats pour effectuer ce filtrage, notamment :

   - *podFitsRessources* : le nœud a-t-il suffisamment de ressources (CPU et RAM) pour accueillir le *pod* ?
   - *podFitsHostPorts* : le nœud est-t-il en mesure d'exécuter le *pod* via le port p sans conflits ?
   - *NoVolumeZoneConflict* : le nœud dispose-t-il de la quantité de stockage v que demande le *pod* ?
   - *MatchNodeSelector* : le nœud correspond-il aux paramètres de la requête du sélecteur défini dans la description du *pod* ?

   Tous ces prédicats peuvent être combinés pour mettre en place des filtres sophistiqués.

2. Classement des *minions* restants (Kubernetes, 2014a) : Après le filtrage, Kubernetes utilise des fonctions de priorité pour calculer le meilleur *minion*

parmi les nœuds qui sont aptes à l'hébergement du *pod*. Pour chacun de ces nœuds, la fonction de priorité attribue un score compris entre 0 et 10 où 10 désigne le préféré. Chaque fonction de priorité est pondérée par un nombre positif et le score final d'un nœud est calculé en additionnant ses scores pondérés. Par exemple, supposons qu'il existe deux fonctions prioritaires f et g avec les facteurs de pondération $poids_f$ et $poids_g$ respectivement. Le score final $ScoreFinalNud_A$ du nœud A est :

$$ScoreFinalNud_A = (poids_f * f) + (poids_g * g) \tag{2.3}$$

Les principales fonctions de priorité activables dans Kubernetes sont :

- *BalancedResourceAllocation* : cette fonction vise à équilibrer la charge des *minions*. Elle essaie de mettre le *pod* dans un nœud de telle sorte que le taux d'utilisation des ressources (*CPU* et mémoire) soit équilibré après le déploiement du *pod*.
- *LeastRequestedPriority* : favorise le nœud qui a le plus de ressources disponibles.
- *CalculateSpreadPriority* : répartit les *pods* en réduisant le nombre de *pods* appartenant au même service sur le même nœud.
- *CalculateAntiAffinityPriority* : répartit les *pods* en minimisant le nombre de *pods* appartenant au même service sur des nœuds partageant un attribut (*label*) particulier. Par exemple des nœuds de test.
- *CalculateNodeLabelPriority* : préfère les nœuds qui ont un attribut spécifié.

Une fois que les scores finaux de tous les nœuds sont calculés, le *minion* ayant le score le plus élevé est choisi comme l'hôte qui va instancier le *pod*. S'il y a plus d'un *minion* qui a le score le plus élevé, le nœud maître du *cluster* Kubernetes en choisit un arbitrairement.

2.5 Comparaison entre Swarm et Kubernetes

Bien que les deux orchestrateurs soient implémentés en Go (Go, 2018) (un langage de programmation orientée service initié par Google) et fournissent à peu près les mêmes fonctionnalités, il existe des différences fondamentales entre les deux approches (Heidari *et al.*, 2016).
Le tableau 2.2 ci-après énumère les points de différences les plus remarquables.

À la suite de cette étude comparative (tableau 2.2), il est possible de noter que Swarm se focalise sur la facilité d'utilisation avec l'intégration au *Docker Deamon*, tandis que Kubernetes prône la vision d'une plateforme d'orchestration modulaire, personnalisable et donc évolutive.

En plus, pour encourager l'adoption des conteneurs dans les environnements modernes et dynamiques (tels que les *clouds* publics, privés et hybrides), des entreprises (Google, Docker, *Microsoft*, *Mesosphere*, *Vmware*, IBM, Oracle, etc.) (Foundation, 2017) ont mis en place la fondation de l'informatique nuagique native, la *CNCF* (*Cloud Native Computing Foundation* (CloudNativeCon, 2018)) (Sill, 2015). L'objectif principal de la *CNCF* est la création de normes pour le fonctionnement des conteneurs. Kubernetes a été adopté comme l'orchestrateur standard de conteneurs. Ainsi le projet K8s est désormais maintenu par la *CNCF* (Github, 2018).

2.6 Conclusion

À travers ce chapitre, nous avons présenté des plateformes d'orchestration de conteneurs Docker. Plus précisément, nous nous sommes intéressés à Swarm et Kubernetes, deux solutions gratuites d'orchestration de conteneurs.
L'étude montre que Kubernetes est la solution qui attire l'attention de la com-

munauté *cloud*. Toutefois, la tolérance aux pannes demeure une préoccupation majeure dans le *cloud computing*. Le chapitre suivant étudie sa prise en compte dans Kubernetes.

Tableau 2.2 Différences fondamentales entre Swarm et Kubernetes.

Paramètres	Swarm	Kubernetes
Mise en œuvre	Facile et rapide à installer	Nécessite quelques configurations manuelles. De plus, les instructions d'installation varient en fonction de l'OS du serveur
Vitesse de déploiement	Se basant sur des critères qui ne tiennent pas compte de l'état réel du *cluster*, Swarm est capable de déployer rapidement des conteneurs	Kubernetes a besoin de connaître à l'avance l'état du *cluster*, ce qui ralentit le déploiement des *pods*
Modularité	Swarm utilise l'*API Docker deamon* pour gérer tout le *cluster*	Kubernetes est modulaire. Il offre d'un ensemble unifié d'*API*s qui collaborent pour gérer le *cluster*
Surveillance	Pas de garanties solides sur l'état du *cluster*	Grâce à son ensemble unifié d'*API*s, le nœud maître connaît en temps réel l'état global du *cluster*
Tolérance aux fautes (pannes)	Faible tolérance aux pannes	Kubernetes est tolérant aux pannes franches comme l'arrêt d'un nœud de travail, d'un *pod* ou d'un conteneur

CHAPITRE III

TOLÉRANCE AUX FAUTES DANS KUBERNETES : ÉTAT DE L'ART

3.1 Introduction

Nous présentons dans ce chapitre les concepts de fautes et de tolérance aux fautes. Nous débutons par la définition de système robuste. Puis, nous identifions les fautes (pannes) qui peuvent survenir dans un système Kubernetes. Par la suite, nous présentons la notion de machines d'états, ainsi que son utilisation dans le cadre de la tolérance aux fautes. Nous formulons le problème du consensus, témoignant des principaux enjeux abordés par les protocoles de réplication de machines d'états. Ensuite, nous présentons les travaux connexes à la tolérance aux fautes dans les systèmes orchestrés par Kubernetes. Nous finissons ce chapitre par l'identification des limites de ces travaux.

3.2 Robustesse d'un système

La robustesse d'un système fait référence à sa capacité à poursuivre son fonctionnement lorsqu'une partie du système est défaillante (Hayes, 1976; Schlichting et Schneider, 1983; Johnson, 1984; Cristian, 1991). Un système est dit défaillant lorsque les sorties ne sont plus conformes à la spécification initiale. Les fautes sont définies comme les causes d'une erreur. Une erreur est attachée à un état du nœud et sa propagation peut entraîner la défaillance du système. L'occurrence

d'une faute peut être : 1) transitoire : peut apparaître, disparaître et ne plus jamais se reproduire ; 2) intermittente : reproductible dans un contexte donné ; 3) persistante : apparaître jusqu'à la réparation du système. Un nœud (également appelé processus) qui n'est pas défaillant est dit correct. Un processus correct suit ses spécifications, tandis qu'un processus défaillant peut s'arrêter ou présenter un comportement arbitraire. La défaillance peut être causée par un défaut logiciel, une attaque malicieuse, une erreur d'interaction personne-machine ou autre.

Au niveau des systèmes distribués/répartis orchestrés par Kubernetes, une défaillance peut survenir au niveau des minions comme au niveau du nœud maître. Une défaillance peut être interprétée par l'ensemble du réseau de deux manières différentes comme nous le verrons dans la section suivante.

3.3 Classification des fautes dans Kubernetes

Les fautes qui peuvent survenir dans un cluster peuvent être classées en deux catégories : fautes de *crash* ou fautes byzantines.

3.3.1 Les fautes de *crash* (*fail-stop*)

Les fautes de *crash* se caractérisent par l'arrêt complet de l'activité d'un nœud. Cet état est perçu par les autres nœuds comme étant l'absence de messages attendus à partir d'un instant donné et ce, jusqu'à l'éventuelle terminaison de l'application.

Un système qui n'est capable de détecter que les fautes de *crash* considère qu'un processus peut être dans un des deux états suivants : soit il fonctionne et donne le résultat correct, soit il ne fait rien. La défaillance peut être par exemple une panne franche de processeur, une coupure de voie physique, certains types de programmes

erronés (exemple, une boucle infinie), un système d'exploitation interbloqué, etc.

3.3.2 Les fautes byzantines

Les fautes byzantines font référence à l'existence de nœuds malicieux dont le but est de semer la confusion dans le réseau. Cela peut être dû à une erreur matérielle, un virus, la corruption du code de l'algorithme, une attaque, etc. Tout comportement s'écartant des spécifications, en produisant des résultats non conformes, est qualifié de comportement byzantin (Lamport *et al.*, 1982).

Nous distinguons entre les fautes byzantines naturelles, par exemple, une erreur physique non détectée sur une transmission de message, la mémoire, ou une instruction, et les fautes byzantines malicieuses visant à faire échouer le système comme un virus, un ver, une instruction de sabotage, etc.

Lorsque la taille d'un système devient importante ou lorsque ce système est déployé dans un environnement non contrôlé, la probabilité que certains éléments du système subissent des fautes devient non négligeable. Dans la suite de ce chapitre, nous nous intéressons aux solutions qui peuvent assurer le bon fonctionnement de tels systèmes, malgré la présence de fautes.

3.4 Réplication de machines d'états

Une manière de garantir la continuité de service est d'utiliser la réplication (Aublin, 2014). La réplication de machines d'états (*State Machine Replication-SMR*) consiste à utiliser plusieurs copies d'un système, implanté sous forme d'une machine d'états, afin de tolérer les fautes (Lamport, 1978; Schneider, 1986, 1990), quelque soit leur type, de façon à ce que le système reste disponible. Chaque copie du système, appelée réplica ou processus, est placée sur un nœud différent.

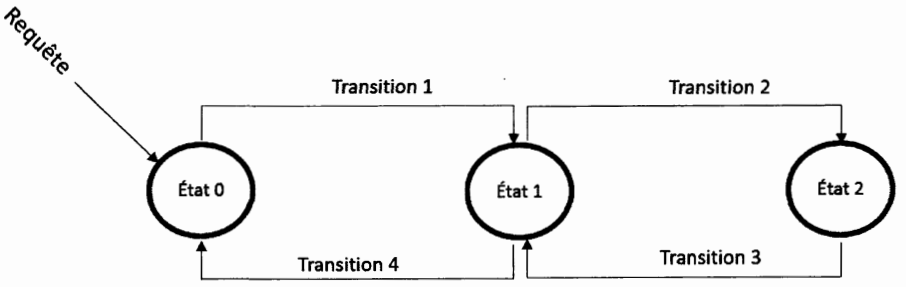

Figure 3.1 Illustration d'une machine d'états composée de trois états distincts.

3.4.1 Machines d'états

Une machine d'états est la représentation d'un système sous forme d'automate. Une telle représentation facilite la modélisation de nombreux systèmes et permet d'en simplifier l'étude. Le fonctionnement d'une machine d'états repose sur deux concepts essentiels : celui d'état et celui de transition. Un état décrit la configuration d'un système, tandis qu'une transition est une séquence d'instructions à exécuter lorsqu'une condition particulière est remplie, ou lorsqu'un événement spécifique survient. Une machine d'états est donc caractérisée par un ensemble d'états et un ensemble de transitions comme le montre la figure 3.1.

Les machines d'états peuvent être déterministes ou non déterministes. Dans le premier cas, leurs sorties ne dépendent que de leurs entrées, c'est-à-dire, une machine à états déterministe retournera toujours la même sortie pour une même entrée. Pour des machines non déterministes, les sorties sont aléatoires. Dans le cadre de la réplication, notre intérêt se porte sur les machines d'états déterministes. En effet, celles-ci permettent de garantir que l'état de plusieurs copies du système sera identique lorsque soumis aux mêmes entrées.

La section suivante présente les différentes approches pour la réplication.

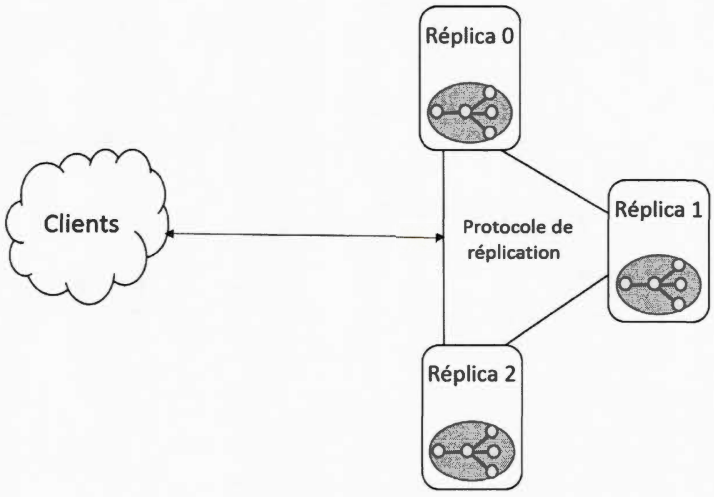

Figure 3.2 Vue d'ensemble de la réplication de machines d'états.

3.4.2 Les formes de réplication

La *SMR* est une technique fondamentale pour la mise en œuvre de systèmes capables d'assurer la continuité de leur fonctionnement malgré la présence de fautes (Sousa *et al.*, 2013). Elle permet à un ensemble de nœuds d'exécuter les mêmes séquences d'instructions sur chaque requête transmise par un *client* (figure 3.2).

Il existe deux approches pour l'exécution des requêtes : 1) la réplication active où tous les réplicas corrects exécutent les requêtes, mettent à jour leur machines d'états et répondent aux clients ; 2) la réplication passive où un seul réplica, appelé leader, exécute les requêtes et transmet les modifications aux autres nœuds, puis répond au client. La figure 3.3 présente l'échange de messages pour chaque forme de réplication. Cependant, pour éviter toute incohérence dans la réplication, les réplicas ont besoin d'être sûrs que leurs machines d'états sont identiques avant de répondre aux *clients*. La section suivante décrit ce problème de réplication de machine d'états, appelé problème de consensus.

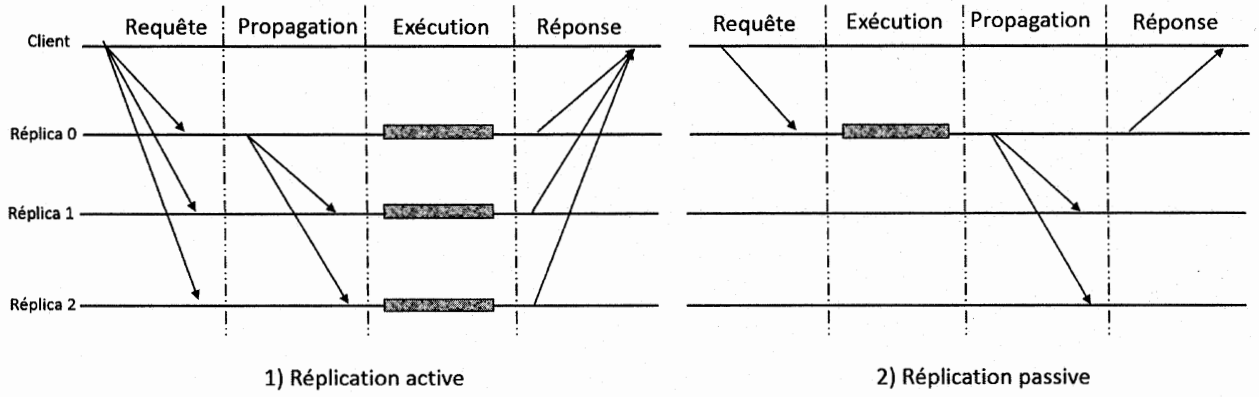

Figure 3.3 Les deux formes de réplication.

3.4.3 Le problème de consensus

Le consensus est un problème fondamental dans les systèmes distribués à tolérance de fautes. Il s'agit de mettre d'accord les réplicas sur une même valeur proposée par un des nœuds. Dans la partie suivante, nous formulons ce problème.

- Formulation du problème de consensus :

  Considérons un système constitué d'un ensemble N de n nœuds répliqués.
  $N = \{N_0, N_1, ..., N_{n-1}\}$ $(n > 1)$.
  Supposons qu'au maximum f $(f \leq n-1)$ nœuds peuvent être défaillants.
  Soit P, un sous-ensemble de m $(m \geq 1)$ nœuds du système.
  $P = \{P_0, P_1, ..., P_{m-1}\} \subseteq \mathbb{N}$ $(0 \leq m \leq n-1)$.

  Le problème de consensus consiste à trouver un protocole qui permet :

  1. À tout nœud P_i $(0 \leq i \leq m-1)$ de proposer aux autres nœuds du système une valeur v_i $(0 \leq i \leq m-1)$.
  2. Ensuite, de mettre d'accord ces derniers sur le choix d'une même valeur $v_{consensus} \in \{v_i\}$ $(0 \leq i \leq m-1)$.

- Solutions possibles :

  Un protocole qui résout ce problème de consensus doit fournir les quatre propriétés suivantes (Pease *et al.*, 1980) :

  (i) Chaque nœud correct décide à *terme* une valeur v_j $(0 \leq j \leq m-1)$ (*terminaison*).

  (ii) Si un nœud décide une valeur v_j $(0 \leq j \leq m-1)$ alors v_j a été proposée par un certain nombre de nœuds du système (*validité*).

  (iii) Aucun nœud ne décide deux fois (*intégrité*).

  (iv) Il n'existe pas deux nœuds corrects qui décident une valeur différente (*accord*).

  Selon Schneider (1990), tout protocole qui vérifie les conditions de sûreté et de vivacité ci-dessous fournira ces quatre propriétés :

  - Sûreté : tous les réplicas corrects exécutent les requêtes qu'ils reçoivent dans le même ordre.
  - Vivacité : chaque requête est exécutée correctement par des réplicas corrects.

Un tel protocole est communément appelé protocole de consensus ou de réplication. Il base ses décisions sur les messages échangés entre les nœuds du système. Néanmoins, certains parmi ces derniers peuvent ne pas participer aux prises de décision. En effet, les protocoles de consensus utilisent une notion de quorum. Il s'agit du nombre minimum de nœuds corrects que le protocole requiert pour établir un consensus. Le quorum dépend de la taille du système et du nombre maximal de fautes que le système tolère.

Le théorème énoncé par Fischer *et al.* (1982), connu sous l'appellation FLP (*Fischer Lynch Problem*) *impossibility result* soutient que la résolution du problème

de consensus est impossible dans un environnement asynchrone (c'est-à-dire si les délais de communication ne sont pas bornés) en présence d'un nœud défaillant. Cette affirmation a été contredite par Bracha et Toueg (1985) qui ont prouvé que l'établissement du consensus ne dépend que du modèle de fautes à tolérer et du quorum du protocole. Ce résultat a conduit au développement de plusieurs protocoles avec des objectifs différents. Ceux-ci sont répartis en deux catégories (Schneider, 1990) : les protocoles de consensus non-byzantins et les protocoles de consensus byzantins. La section suivante présente ces protocoles.

3.5 Les types de protocoles de consensus

Un protocole de consensus ou de tolérance aux fautes permet à un système de n nœuds d'établir un consensus sur chaque action, même si certains nœuds sont défectueux.

3.5.1 Les protocoles non-byzantins

Les protocoles non-byzantins supposent que les nœuds échouent uniquement s'ils s'arrêtent toute activité. Ces protocoles requièrent $n = 2f+1$ réplicas pour tolérer au plus $f = \frac{n-1}{2}$ fautes de *crash* (Bracha et Toueg, 1985). Par exemple, Paxos (Lamport *et al.*, 2001; Lamport, 2011; CoreOS, 2018), Raft (Ongaro et Ousterhout, 2014), Zab (Van Renesse *et al.*, 2015) sont des protocoles non-byzantins.

3.5.2 Les protocoles byzantins

Les protocoles byzantins sont capables de tolérer n'importe quel type de fautes. Toutefois, les protocoles byzantins exigent typiquement $n = 3f + 1$ pour tolérer au plus $f = \frac{n-1}{3}$ fautes byzantines (Bracha et Toueg, 1985).

Dans le cas des protocoles byzantins, seule la réplication active est utilisée. En effet, si la réplication est passive, un *leader* malicieux peut décider de ne pas exécuter les requêtes correctement. Or, ni le client ni les autres réplicas ne s'en rendront compte. Cela n'est pas le cas avec la réplication active, où chaque réplica exécute la requête et répond au client. Le client peut ainsi comparer les réponses des différents réplicas afin de s'assurer de leur validité.

Comme exemples, nous citons *PBFT* (*Pratical Byzantin Fault-Tolerance*) (Castro et Liskov, 2002), *Prime*(Amir *et al.*, 2011), (Clement *et al.*, 2009b,a; Aublin *et al.*, 2015; Kotla *et al.*, 2009), *EBFT* (*Efficient Byzantin Fault-Tolerance*) (Veronese *et al.*, 2013) et *BFT-SMaRt* (*Byzantin Fault-Tolerance State Machine Replication*) (Sousa et Bessani, 2012; Sousa *et al.*, 2013, 2018).

3.6 Exemple de protocole non-byzantin : Raft

3.6.1 Description

Raft (Ongaro et Ousterhout, 2014) est un protocole dérivé de Paxos (Lamport *et al.*, 2001). Il est conçu pour être facile à comprendre et à mettre en œuvre, en comparaison aux autres protocoles de réplication. Par conséquent, de nombreuses implémentations de Raft sont disponibles (Blake Mizerany et Qin, 2018).

Intégré dans plusieurs systèmes à réplication (Blake Mizerany et Qin, 2018), le protocole Raft veille à ce que les réplicas maintiennent des machines d'états identiques (Oliveira *et al.*, 2016). Faisant partie des protocoles non-byzantins, Raft ne tolère que les fautes de crash.

Raft suit une réplication passive. Ses réplicas peuvent avoir le statut de *leader*, *suiveur* ou *candidat*.

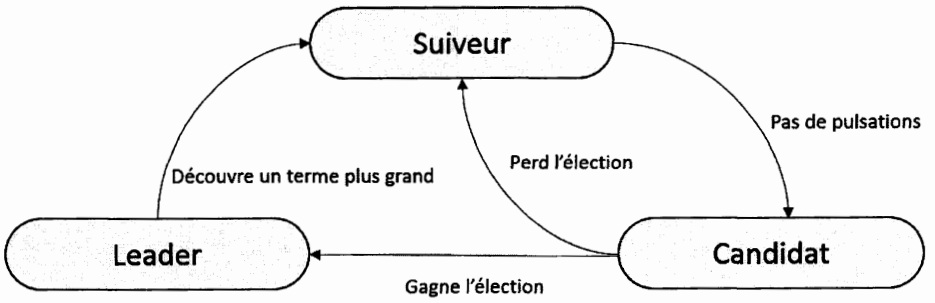

Figure 3.4 Flux électoral du *leader* du protocole Raft.

- *Le leader* : Dans le cluster, un unique nœud actif dirige la communication. Il reçoit les demandes des *clients*, les traite et réplique les modifications de sa machine d'états aux autres nœuds. Après quoi, il envoie la réponse aux *clients*.

- *Les suiveurs* : Quand le *leader* est actif, tous les autres nœuds sont des *suiveurs*. Ils attendent les modifications envoyées par le *leader* pour mettre à jour leurs machines d'états. Si un *client* transmet sa requête à un nœud *suiveur*, ce dernier ignore la demande et lui notifie l'adresse du *leader*.

- *Les candidats* : Quand le *leader* tombe en panne, les *suiveurs* se portent *candidats* et déclenchent des votes pour élire un nouveau leader.

3.6.2 Élection du *leader*

Le *mandat* d'un *leader* $Raft$ commence par son élection jusqu'à ce qu'il tombe en panne. Afin d'organiser des élections de leader, Raft assigne un numéro à chaque mandat. Ces mandats, appelés *termes*, sont représentés par des entiers positifs et consécutifs. Tout nœud (leader ou candidat) inclut le numéro du *terme* dans ses messages.

De plus, chaque nœud *suiveur* dispose d'un délai d'attente (généralement entre

150 et 300 ms) avant de se porter *candidat*. Tant que le *leader* est actif, il envoie périodiquement des messages de pulsations ($AppendEntriesMessage$) à tous les nœuds du cluster. Tout nœud recevant un tel message réinitialise son délai d'attente à une valeur arbitraire. Si, toutefois, le nœud *leader* échoue et que le *délai d'attente* d'un suiveur expire, ce dernier change de statut en candidat et déclenche une nouvelle élection de leader. Dans ce cas, le candidat procède comme suit :

1. Incrémente son numéro de *terme* actuel.

2. Vote pour lui-même.

3. Envoie des demandes de votes ($RequestVoteMessages$) à tous les nœuds du cluster.

Ces derniers répondent à la demande en votant pour la première requête contenant un numéro de *terme* supérieur au leur. Ils mettent également à jour leur numéro de *terme* et reviennent au statut *suiveur*. Une fois qu'un *candidat* reçoit les votes de la majorité ($\lceil f + 1 \rceil$) des nœuds, il se présente comme le nouveau *leader* du cluster. Toutefois, si aucun candidat n'obtient la majorité des votes (par exemple, lorsqu'il y a égalité), aucun *leader* n'est élu pour ce *terme*. Ainsi, le premier qui verra son *délai d'attente* expiré commence une nouvelle élection de leader.

L'exigence d'une majorité de votes garantit qu'un seul *leader* sera sélectionné dans un *terme* (sécurité), tandis que les *délais d'attente* des *suiveurs* garantissent qu'un certain *leader* sera éventuellement sélectionné (vivacité).
Le flux électoral du *leader* Raft est résumé dans la figure 3.4.

3.7 Mécanismes de tolérance aux fautes intégrés dans Kubernetes

3.7.1 Les solutions existantes

Raft (Ongaro et Ousterhout, 2014) est le protocole de réplication intégré dans Kubernetes (Google, 2018; Oliveira *et al.*, 2016; CoreOS, 2018; Blake Mizerany et Qin, 2018). Quelques modifications ont été apportées à Raft pour permettre son exécution dans l'environnement Kubernetes. Celles-ci concernent notamment les caractéristiques suivantes :

1. Acceptation des requêtes par tout réplica : Dans le protocole Raft (Ongaro et Ousterhout, 2014), seul le nœud *leader* peut interagir avec les clients. Lorsqu'un réplica non-leader reçoit des demandes du client, celles-ci sont ignorées et le client est redirigé à l'adresse du leader. Une telle approche est contradictoire à Kubernetes qui établit un équilibrage de charge (via le *kube-proxy*), en envoyant les demandes à tout nœud ou conteneur répliqué. Par conséquent, Raft est converti en une réplication active (section 3.4.2) pour se conformer à Kubernetes (Oliveira, 2017; Blake Mizerany et Qin, 2018).

2. Réimplémentation en Go : Conçu pour être facile à comprendre, à adopter et à mettre en œuvre, Raft est réimplémenté dans le langage de programmation Go (Blake Mizerany et Qin, 2018). Ce dernier constitue le langage de Kubernetes ainsi que celui des conteneurs Docker.

Netto *et al.* (2017) proposent un autre protocole de réplication non-byzantin appelé *DORADO*. Il est conçu pour être intégré dans Kubernetes. *DORADO* est similaire à Raft, mais il exige que la mémoire du nœud maître Kubernetes soit partagée (en lecture et en écriture) à tous les conteneurs instanciés au niveau des

nœuds de travail afin qu'ils y conservent leurs machines d'états. Cette approche permet à *DORADO* d'avoir des délais de consensus plus courts que *Raft* mais en aggravant le problème d'isolation.

Oliveira *et al.* (2016) montrent que les performances de Raft dans un cluster Kubernetes mono-maître sont similaires à celles obtenues dans un système non virtualisé (Ongaro et Ousterhout, 2014).

3.7.2 Limites des solutions existantes

Malgré ses propriétés souhaitables pour tout système à réplication, Raft est particulièrement impuissant devant des comportements byzantins (Lim *et al.*, 2014; Buchman, 2016). En effet, un réplica défaillant peut ne pas s'arrêter, et plutôt adopter continuellement un comportement arbitraire, comme ne pas suivre le protocole, corrompre son état local ou produire des sorties incorrectes ou incohérentes (Schneider, 1990). Par exemple, des attaques de déni de service distribuées (*Distributed Denial of Service-DDoS*) (Gilbert *et al.*, 2012; Lonea *et al.*, 2013; Somani *et al.*, 2017) peuvent amener un nœud à présenter un comportement byzantin.

3.8 Conclusion

Nous avons présenté dans ce chapitre les fautes susceptibles d'être présentes dans tout système Kubernetes. Celles-ci peuvent être divisées en deux catégories : les fautes de *crash* qui caractérisent les arrêts de processus uniquement, et les fautes byzantines qui modélisent toute sorte de faute. Un système n'est robuste que s'il est capable de poursuivre son fonctionnement normal malgré la présence de fautes. Nous avons étudié les mécanismes de tolérance à ces fautes. Plus précisément, nous nous sommes intéressés au protocole Raft, la solution de tolérance aux fautes intégrée dans Kubernetes.

Toutefois, Raft suppose que les composants échouent uniquement à l'arrêt de tout traitement. Par conséquent, Kubernetes ne garantit pas la disponibilité des applications conteneurisées en cas d'erreurs logicielles ou d'attaques malicieuses.

Dans le chapitre suivant, nous présentons KmMR, la plateforme Kubernetes multi-maîtres robuste que nous proposons pour surmonter cette limitation dans l'orchestrateur standard des conteneurs Docker.

CHAPITRE IV

KMMR : PLATEFORME D'ORCHESTRATION DE CONTENEURS DOCKER ROBUSTE

4.1 Introduction

Kubernetes permet de déployer et d'orchestrer des groupes de conteneurs avec un seul nœud maître. Cependant, certains systèmes critiques nécessitent des garanties de continuité de service plus strictes. C'est le cas des systèmes de télécommunications et des services de distribution d'énergie. Ces derniers doivent être disponibles 24h/24 et 7j/7.

Le nœud maître Kubernetes se charge de répliquer les conteneurs du cluster sur différents nœuds de travail pour assurer cette continuité de service en cas de panne au niveau d'un nœud de travail. Toutefois, le nœud maître K8s peut lui-même échouer. Une telle situation entraînera l'indisponibilité totale du cluster, c'est-à-dire la perte de toutes les données de gestion. Dans ce cas, le déploiement de clusters Kubernetes multi-maîtres où plusieurs nœuds maîtres coopèrent devient nécessaire. La duplication de nœuds maîtres à elle seule ne permet pas d'assurer une tolérance totale aux pannes (Perronne, 2016). En effet, elle doit être associée à un protocole de réplication (également appelé protocole de consensus), chargé d'assurer la cohérence entre les états des nœuds maîtres. Ce dernier permet de définir le degré de robustesse du système multi-maîtres.

Dans ce chapitre, nous proposons de créer une plateforme Kubernetes multi-maîtres résistante à tout type de fautes. Elle est basée sur la duplication du nœud maître et l'intégration d'un protocole de réplication tolérant aux fautes non-byzantines et byzantines dans Kubernetes.

4.2 Modèle de système

Nous définissions dans cette section le modèle du système incluant le type de système, le modèle de communication et les fautes considérées.

4.2.1 Type de système

Nous considérons un cluster Kubernetes constitué de n nœuds maîtres K8s répliqués entre eux et de c nœuds clients (également appelés nœuds de travail K8s). Les nœuds de travail traitent les demandes de service venant des utilisateurs finaux et envoient leurs états aux nœuds maîtres sous forme de requêtes. De plus, nous supposons qu'au plus f nœuds maîtres K8s peuvent tomber simultanément en panne. À l'état initial du cluster, nous supposons que la condition byzantine $n \geq 3f + 1$ est vérifiée (Schneider, 1990; Malkhi et Reiter, 1998; Castro et Liskov, 2002; Aublin *et al.*, 2015). De plus, le système peut être configuré pour vérifier aussi $n \geq 2f+1$ réplicas pour tolérer f fautes non-byzantines (de crash) (Lamport *et al.*, 2001; Lamport, 2011; Ongaro et Ousterhout, 2014; CoreOS, 2018).

La figure 4.1 présente l'architecture de la plateforme Kubernetes multi-maîtres robuste, la solution que nous proposons dans ce mémoire.

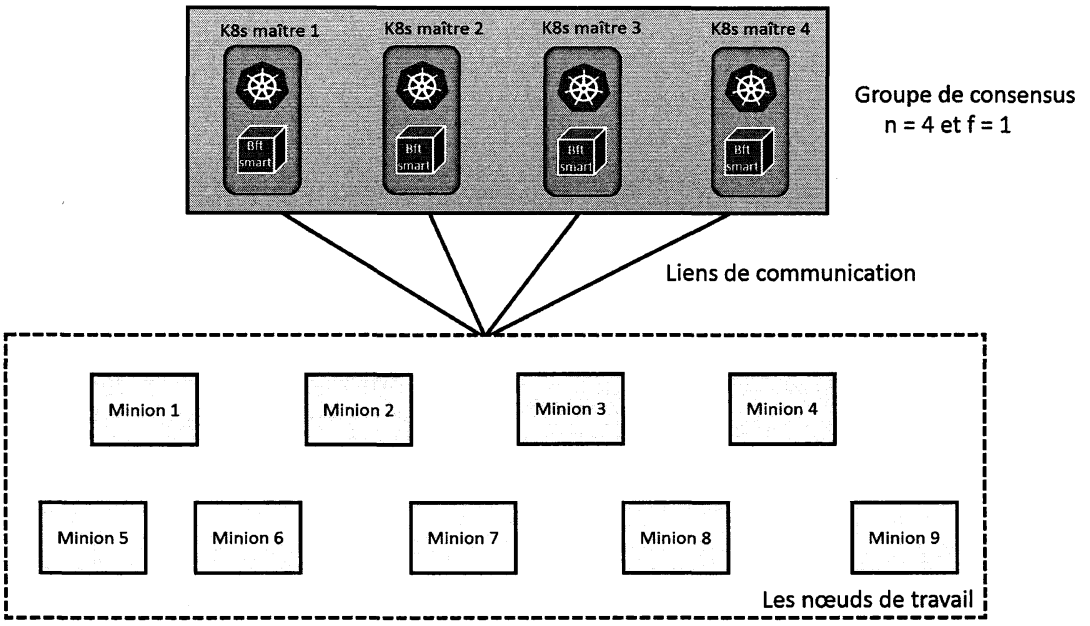

Figure 4.1 Plateforme Kubernetes multi-maîtres robuste.

4.2.2 Modèle de communication

Nous considérons un système de nœuds distribués et asynchrone. Le système est asynchrone dans le sens où les nœuds peuvent ne pas être en mesure d'envoyer des messages et d'obtenir des réponses dans les délais. En d'autres termes, les pannes de communication sont possibles. De plus, nous supposons l'existence de fonctions cryptographiques fournissant des signatures numériques, des résumés de messages et des codes d'authentification de messages (*Message Authentication Code-MAC*).

4.2.3 Modèle de fautes

Nous supposons que tout nœud maître peut être défaillant s'il arrête de fonctionner (faute de crash c'est-à-dire non-byzantine), comme décrit à la section 3.3.1 ou bien s'il présente un comportement arbitraire (faute byzantine), selon la section 3.3.2.

4.3 BFT-SMaRt : protocole de réplication pour KmMR

Chaque nœud maître correct dans KmMR doit avoir, en tout temps, une copie conforme de l'état global du cluster. Pour cela, les propriétés de *sûreté* et de *vivacité* (Schneider, 1990) doivent être satisfaites. La *sûreté* stipule que tous les nœuds maîtres corrects exécutent les requêtes qu'ils reçoivent dans le même ordre alors que la *vivacité* garantit que chaque requête est exécutée par des nœuds maîtres corrects.

Ces deux conditions peuvent être satisfaites si tous les nœuds maîtres communiquent entre eux à travers un protocole de réplication. Le protocole est en charge de garantir le fait que tout le système se comporte en accord avec sa spécification et que l'état entre les maîtres reste cohérent, malgré la présence d'au plus f fautes que le protocole est en mesure de tolérer.
Kubernetes intègre Raft comme protocole de réplication (Netto *et al.*, 2017; Google, 2018; Oliveira *et al.*, 2016; CoreOS, 2018; Blake Mizerany et Qin, 2018). Ce protocole non-byzantin ne garantit aucune cohérence si un nœud du système présente un comportement byzantin. Par conséquent, il n'est pas recommandé d'utiliser uniquement Raft (de même que tout autre protocole non-byzantin) dans un environnement non fiable ou présentant des risques d'instabilité. Dans ce travail, nous proposons de mettre en place une plateforme Kuberetes multi-maîtres robuste, via l'intégration du protocole byzantin *BFT-SMaRt* dans Kubernetes.

En effet, parmi les protocoles byzantins connus, seuls *PBFT* (Lamport, 2011), *UpRight* (Clement *et al.*, 2009a) et *BFT-SMaRt* (Sousa *et al.*, 2013; Sousa et Bessani, 2012) mettent en œuvre un système de réplication fonctionnel. Le choix de *BFT-SMaRt* est motivé par les raisons suivantes :

1. L'architecture de *PBFT* n'exploite pas pleinement le matériel moderne contrai-

rement à *BFT-SMaRt* qui s'adapte bien aux systèmes multi-cœurs (Sousa *et al.*, 2013).

2. *UpRight* présente des performances nettement inférieures à celles de *BFT-SMaRt* (Sousa *et al.*, 2013).

3. *BFT-SMaRt* cible à la fois des performances élevées en termes de temps d'exécution et l'exactitude des données répliquées si des nœuds défaillants présentent un comportement byzantin.

4. *BFT-SMaRt* est une bibliothèque modulaire, extensible et robuste (GitHub, 2018). Son objectif principal est de fournir une bibliothèque pouvant être adoptée pour mettre en place des services fiables.

5. *BFT-SMaRt* prend en charge les reconfigurations du jeu de réplicas, c'est-à-dire l'ajout et la suppression de nœuds dans le système (Bessani *et al.*, 2013) contrairement aux autres protocoles byzantins.

6. *BFT-SMaRt* fournit un support efficace et transparent pour des services durables (Lamport *et al.*, 2010).

4.3.1 Établissement du consensus

La figure 4.2 présente l'ensemble des étapes pour établir un consensus avec *BFT-SMaRt*. Les nœuds clients envoient leurs requêtes à tous les nœuds maîtres, ce qui déclenche l'exécution du protocole de consensus. Chaque instance de consensus commence par un nœud maître — le *leader* — proposant un lot de demandes dans le cadre de l'établissement d'un consensus. Cela se fait en envoyant un message $PROPOSE$ contenant le lot susmentionné aux autres nœuds maîtres. Tous les maîtres qui reçoivent le message $PROPOSE$ vérifient s'il provient du *leader* et si le lot proposé est valide. Si tel est le cas, ils enregistrent le lot proposé et

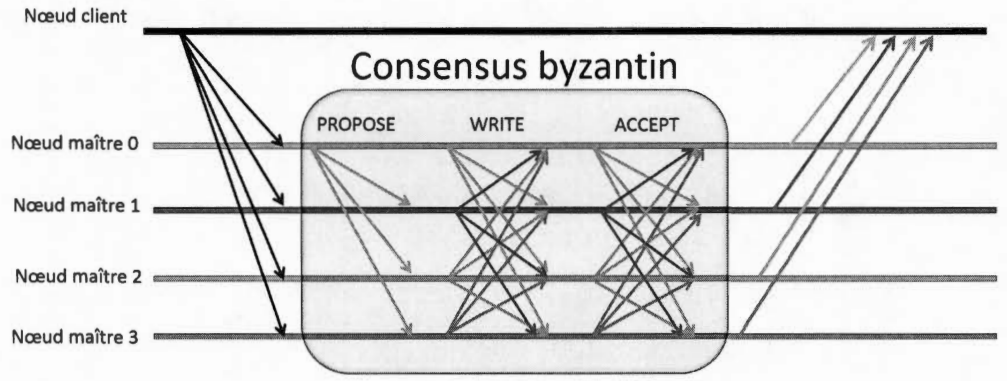

Figure 4.2 Processus d'établissement du consensus par BFT-SMaRt.

envoient un message $WRITE$ à tous les autres réplicas contenant un hachage cryptographique du lot proposé. Si un réplica reçoit $\lceil \frac{n+f+1}{2} \rceil$ messages $WRITE$ avec le même hachage, il envoie un message $ACCEPT$ à tous les autres réplicas. Si un réplica reçoit $\lceil \frac{n+f+1}{2} \rceil$ messages $ACCEPT$ pour le même hachage, il envoie son lot correspondant comme décision de son instance de consensus.
La séquence des messages qui vient d'être décrite est exécutée si le nœud maître *leader* est correct. Si cette condition n'est pas remplie, le protocole doit élire un nouveau *leader* et forcer tous les réplicas à converger vers la même exécution par consensus. Cette procédure est décrite en détail par Sousa et Bessani (2012).

4.4 Intégration de la bibliothèque BFT-SMaRt dans K8s

Le protocole *BFT-SMaRt* est implémenté en Java, un langage de programmation orienté objet, tandis que Kubernetes et le moteur Docker sont écrits en Go (Go, 2018). Ainsi, nous avons deux options différentes pour intégrer *BFT-SMaRt* dans K8s. La première consiste à réécrire tout le code source de la bibliothèque *BFT-SMaRt* en Go. La seconde est d'envelopper la bibliothèque *BFT-SMaRt* dans un conteneur Docker. Contrairement à Raft qui contient moins de 3000 lignes de code

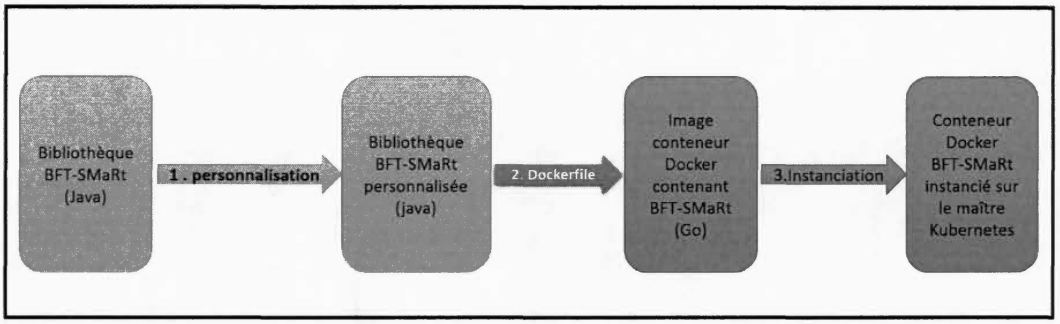

Figure 4.3 Intégration du protocole BFT-SMaRt dans K8s.

(Ongaro et Ousterhout, 2014) et qui a été réécrit en Go pour fonctionner dans l'environnement Kubernetes (Blake Mizerany et Qin, 2018), nous optons pour la seconde approche pour les raisons suivantes :

- Un conteneur Docker s'exécute rapidement (Joy, 2015a).
- Le surcoût introduit par un conteneur Docker est négligeable (Felter *et al.*, 2015).
- Le code source du projet *BFT-SMaRt* contient environ 100 fichiers, pour un total de 13500 lignes de code Java (Sousa *et al.*, 2013), ce qui serait fastidieux à réécrire en langage Go.

La figure 4.3 montre notre procédure d'intégration de *BFT-SMaRt* dans Kubernetes. Nous récupérons la bibliothèque *BFT-SMaRt* et toutes ses dépendances (GitHub, 2018). Puis, nous la personnalisons en définissant les paramètres des nœuds maîtres. Ensuite, nous créons notre fichier Docker $Docker file$ (figure 4.4). Après, nous exécutons le fichier pour produire l'image Docker conteneurisant *BFT-SMaRt*. Par la suite, nous instancions dans chaque nœud maître Kubernetes l'image Docker possédant ses informations.

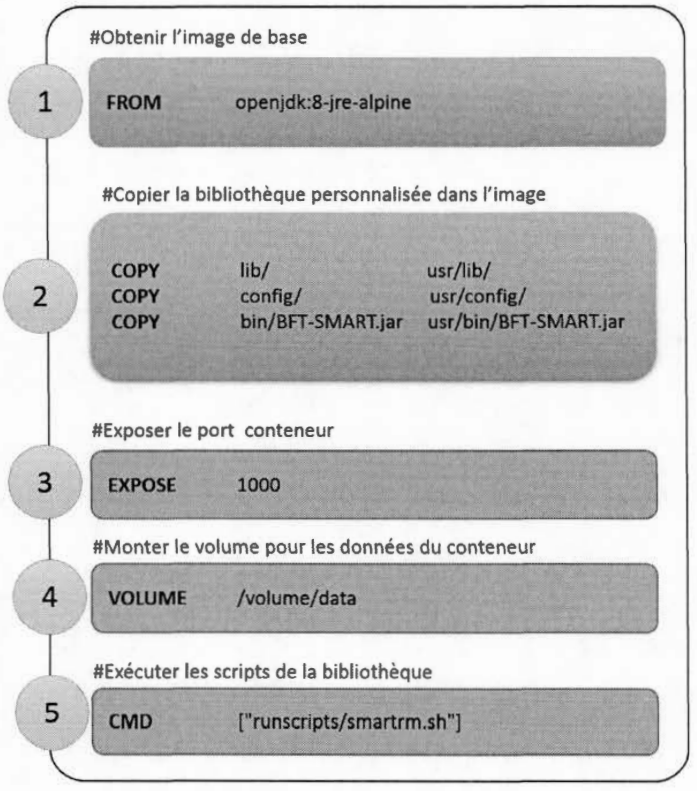

Figure 4.4 Dockerfile de création du conteneur BFT-SMaRt.

4.5 Évaluation expérimentale

4.5.1 Configuration de la simulation

Nous avons mis en œuvre notre solution KmMR dans un environnement *cloud* OpenStack (Sefraoui *et al.*, 2012) fourni par Ericsson Canada. Les ressources disponibles dans cet environnement étaient comme suit : 50 Go de mémoire vive (*RAM*) et 20 processeurs virtuels (VCPU) utilisables sur un nombre maximal de 10 machines (figure 4.5).

L'expérimentation a été réalisée sur des *clusters* composés de plusieurs nœuds maîtres Kubernetes ($n = 5$ ou $n = 7$), connectés entre eux via le réseau Giga Ethernet d'OpenStack et accessibles depuis Internet. Chaque nœud est une machine virtuelle équipée du système d'exploitation Ubuntu serveur 18.04 TLS 64 bits et possède deux processeurs virtuels (*VCPU*) core i7 cadencés à 2.4 GHz, avec 4 Go de *RAM* et 20 Go pour le stockage. Le moteur Docker 18.05.0-ce est installé sur les nœuds Kubernetes pour les besoins d'instanciation des conteneurs. Ensuite, nous avons déployé Kubernetes 1.11.0 pour assurer l'orchestration de ces conteneurs Docker. Le rôle de maître Kubernetes *kubeadm* a été activé sur tous les nœuds maîtres (configuration multi-maîtres). Les machines restantes ont été utilisées pour représenter les nœuds clients et les générateurs d'attaques. *BFT-SMaRt* a été conteneurisé et intégré dans les nœuds maîtres pour assurer la coordination et l'établissement du consensus entre eux suite aux requêtes envoyées par les nœuds clients. Ces derniers envoient leurs requêtes en boucle fermée, comme défini par Schroeder *et al.* (2006), c'est-à-dire qu'ils attendent la réponse d'une requête avant d'en envoyer une nouvelle. Dans chaque cluster, nous commençons par l'initialisation du protocole de réplication sur chaque nœud maître et nous attendons que tous les nœuds maîtres s'identifient mutuellement et qu'ils ne sont pas défaillants. Ensuite, deux clients diffusent des requêtes à l'ensemble des nœuds

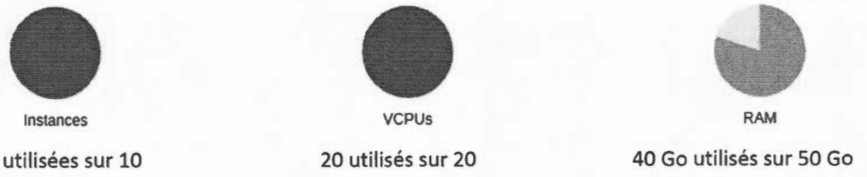

Figure 4.5 Taux d'utilisation des ressources *cloud openstack*.

maîtres. Ces derniers reçoivent les demandes et s'échangent des messages afin d'établir un consensus. Pour mesurer les performances de KmMR, nous avons utilisé le micro-benchmark 0/0 (Castro et Liskov, 2002). Ceci nécessite que les requêtes des nœuds clients et les réponses des nœuds maîtres Kubernetes soient vides.

Afin de modéliser un comportement byzantin, *Hping*3 (Sanfilippo, 2014; Ops, 2016; Kolahi *et al.*, 2015) a été choisi comme outil de génération d'attaques distribuées de déni-de-service (*DDoS*). Les machines d'attaque *DDoS* ciblent simultanément un seul nœud maître du cluster. En effet, chaque machine d'attaque lui envoie en boucle ouverte des requêtes de 65495 octets à travers la commande d'attaque `hping3 -f adresse IP du nœud maître ciblé -d 65495`, c'est-à-dire les requêtes sont envoyées successivement sans attente de réponse. La figure 4.6 montre les débits d'attaques *DDoS* en fonction du nombre de machines d'attaque *DDoS* utilisées simultanément. Dans le cas de trois attaquants, la troisième machine génère des requêtes de taille 16000, 32000, 48000 et 65495 octets selon le débit d'attaque requis.

Nous avons évalué les performances de notre solution et nous l'avons comparée à la plateforme Kubernetes multi-maîtres classique (KmMC). Pour ce faire, nous avons testé les deux scénarios suivants, chacun répété une vingtaine de fois.

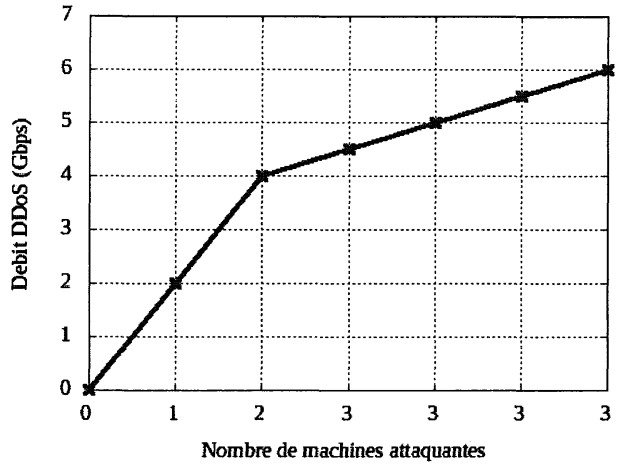

Figure 4.6 Variation du débit d'attaque DDoS.

- Scénario 1 : Plateforme Kubernetes dans un environnement non-byzantin. L'objectif de ce scénario consiste à étudier le comportement du délai de consensus lorsqu'initialement le nombre de fautes dans le cluster est inférieur au nombre maximal de fautes tolérées par le protocole de réplication en place. Ceci correspond à $f < \frac{n-1}{3}$ pour KmMR et $f < \frac{n-1}{2}$ pour KmMC. Afin d'étudier la robustesse de chaque plateforme, nous avons orchestré des attaques *DDoS* sur un seul nœud maître. Pour chaque débit d'attaque, nous avons récupéré le délai de consensus établi par le nœud maître leader.

- Scénario 2 : Plateforme Kubernetes dans un environnement byzantin. Dans ce scénario, nous avons étudié le comportement du délai de consensus et l'utilisation des ressources (*CPU*, *RAM* et bande passante) du cluster lorsque la plateforme est initialisée avec le nombre de fautes maximal qui peuvent être tolérées par le protocole de réplication en place. Ceci correspond à $f = \frac{n-1}{3}$ pour KmMR et à $f = \frac{n-1}{2}$ pour KmMC. Afin d'étudier la robustesse et le surcoût de KmMR et KmMC, nous avons orchestré des attaques *DDoS* sur un seul nœud maître. Pour chaque débit d'attaque, nous avons obtenu le

délai de consensus établi par le nœud maître leader, ainsi que la consommation des ressources par le nœud maître victime de l'attaque *DDoS*. *IPerf3* (iPerf, 2018) a été utilisé pour mesurer la bande passante tandis que *top* (BISWAS, 2018) sert à mesurer les consommations en *CPU* et *RAM*.

4.5.2 Résultats et discussions

Afin d'évaluer la pertinence de notre solution, nous avons considéré quatre métriques : les délais observés par les nœuds maîtres pour établir un consensus, l'utilisation des ressources *CPU*, *RAM* et bande passante.

Le tableau 4.1 présente les délais de consensus requis dans chaque plateforme sur des clusters composés de cinq (5) et de sept (7) nœuds maîtres respectivement. La première colonne contient les débits des attaques *DDoS*, les deux colonnes qui suivent contiennent les délais de consensus obtenus avec la plateforme Kubernetes classique (KmMC), les deux dernières colonnes présentent ceux obtenus avec KmMR, la solution que nous proposons. Pour les deux plateformes, les délais de consensus augmentent légèrement et proportionnellement aux débits d'attaque *DDoS*. En effet, dans un environnement fiable, et même avec une faute byzantine supplémentaire, la plateforme du cluster respecte toujours le seuil de nombre de fautes maximal, établi à $\frac{n-1}{3}$ pour KmMR et à $\frac{n-1}{2}$ pour KmMC. Par conséquent, le fonctionnement de la plateforme continuera sans dégradation significative des performances.Toutefois, KmMC présente des délais de consensus plus petits que KmMR. Ceci est directement lié à la nature du protocole utilisé par la plateforme. En effet, KmMC se base sur le protocole Raft conçu de façon à avoir peu d'échanges de messages de consensus entre les nœuds maîtres (Ongaro et Ousterhout, 2014). Par contre, KmMR se base sur *BFT-SMaRt* qui nécessite un échange

Tableau 4.1 Variation du délai de consensus (μs) en fonction du débit de l'attaque *DDoS* (*Gbps*) dans un environnement non-byzantin (n = 5 et n = 7).

Débit d'attaque *DDoS*	KmMC		KmMR	
	5-maîtres K8s	7-maîtres K8s	5-maîtres K8s	7-maîtres K8s
0.0 *Gbps*	1701.91	2048.25	2746.45	3161.83
2.0 *Gbps*	2004.38	2132.93	2940.87	3179.45
4.0 *Gbps*	2178.72	2471.39	3362.42	4521.79
4.5 *Gbps*	2201.37	2501.73	3525.17	4632.38
5.0 *Gbps*	2287.65	2623.87	3612.93	4729.98
5.5 *Gbps*	2304.12	2702.99	3867.32	4970.93
6.0 *Gbps*	2331.12	2732.25	4053.53	4970.93

plus important de messages afin d'établir le consensus (Sousa et Bessani, 2012; GitHub, 2018). On déduit que, dans un environnement fiable, une faute byzantine supplémentaire (générée ici par les attaques *DDoS*) n'affecte en rien la stabilité du protocole de réplication en place. Il est donc judicieux de favoriser la plateforme KmMC si le risque d'excéder le nombre de fautes maximal, dicté par Raft, est faible.

Nous présentons dans la figure 4.7 la variation du délai de consensus en fonction du débit d'attaque *DDoS* dans un environnement byzantin sur un cluster composé de 5 nœuds maîtres. Les résultats obtenus montrent que la durée du consensus augmente avec le débit de l'attaque *DDoS*. Lorsque le débit de l'attaque *DDoS* est plus petit que 4.25 *Gbps*, KmMC fournit un temps de consensus légèrement meilleur que notre solution. Par contre, lorsque le débit d'attaque dépasse cette valeur, KmMC se dégrade de façon rapide et importante. Ceci est dû principa-

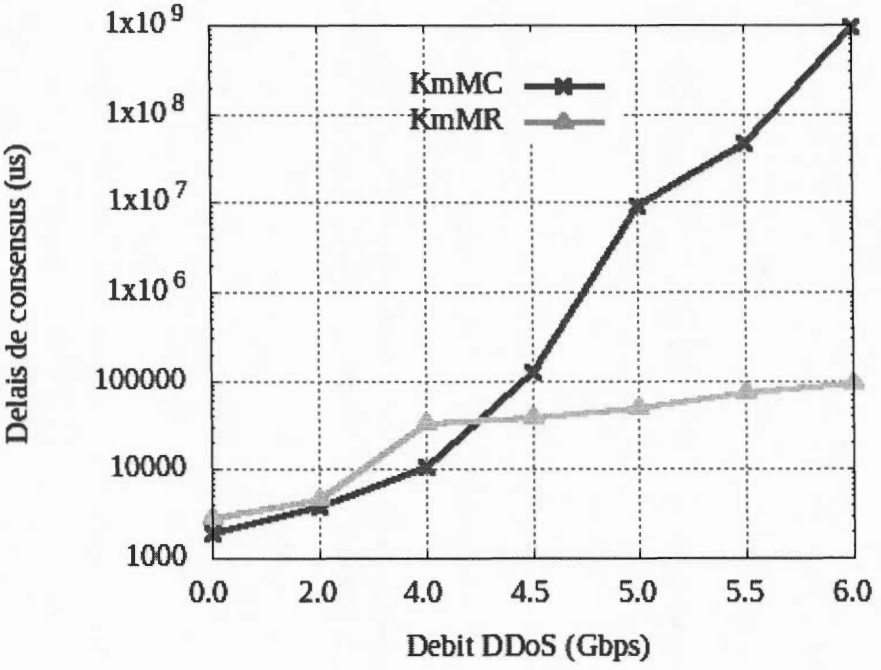

Figure 4.7 Variation du délai de consensus (μs) en fonction du débit de l'attaque *DDoS* (*Gbps*) dans un environnement byzantin (n = 5).

lement au fait que le protocole de réplication Raft de KmMC n'est pas capable de se protéger contre les fautes byzantines. En fait, dans une telle situation, le nœud victime des attaques adopterait un comportement byzantin, par exemple il ne répondrait plus aux autres nœuds dans les temps. Ainsi, à partir du moment où l'ampleur de l'attaque *DDoS* devient importante, le protocole de réplication Raft déclenche des changements dans le leadership du cluster puisqu'il n'arrive plus à établir de consensus avec son nœud leader actuel. Ceci ralentit considérablement l'établissement des consensus dans la plateforme KmMC. Face à l'ampleur de l'attaque *DDoS* (débit >= 4.25 *Gbps*), notre solution KmMR résiste et assure un délai de consensus 1000 fois meilleur que KmMC.

La figure 4.8 illustre la variation du délai de consensus dans le même environ-

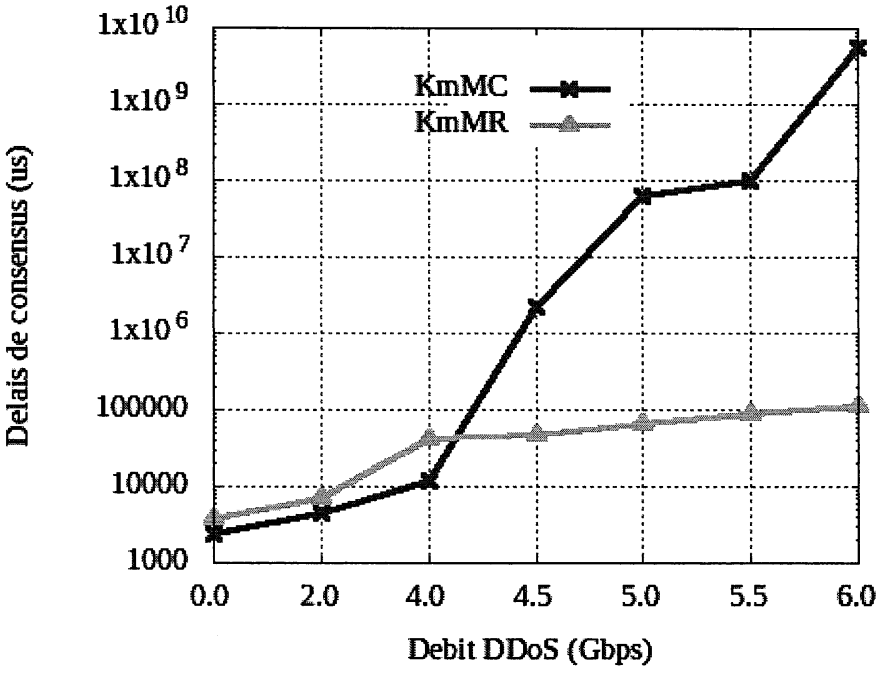

Figure 4.8 Variation du délai de consensus (μs) en fonction du débit de l'attaque *DDoS* (*Gbps*) dans un environnement byzantin (n = 7).

nement mais sur un cluster composé de 7 nœuds maîtres. Nous constatons que les variations sont similaires dans les deux clusters. Toutefois, l'établissement de consensus est plus rapide dans un cluster composé de 5 maîtres. Ceci est dû au nombre de messages moins importants afin d'établir le consensus. En outre, dans un cluster composé de 7 maîtres, KmMC devient plus sensible aux attaques *DDoS*. En effet, la dégradation rapide des performances de KmMC commence pour un débit d'attaque aux alentours de 4.1 *Gbps* au lieu de 4.25 *Gbps* pour le cluster à 5 maîtres. Par contre, KmMR, basé sur le protocole de réplication *BFT-SMaRt*, est capable d'établir son consensus dans un environnement byzantin, même face à des attaques *DDoS* réussies.

Les figures 4.9, 4.10 et 4.11 présentent le taux d'utilisation du *CPU*, la consom-

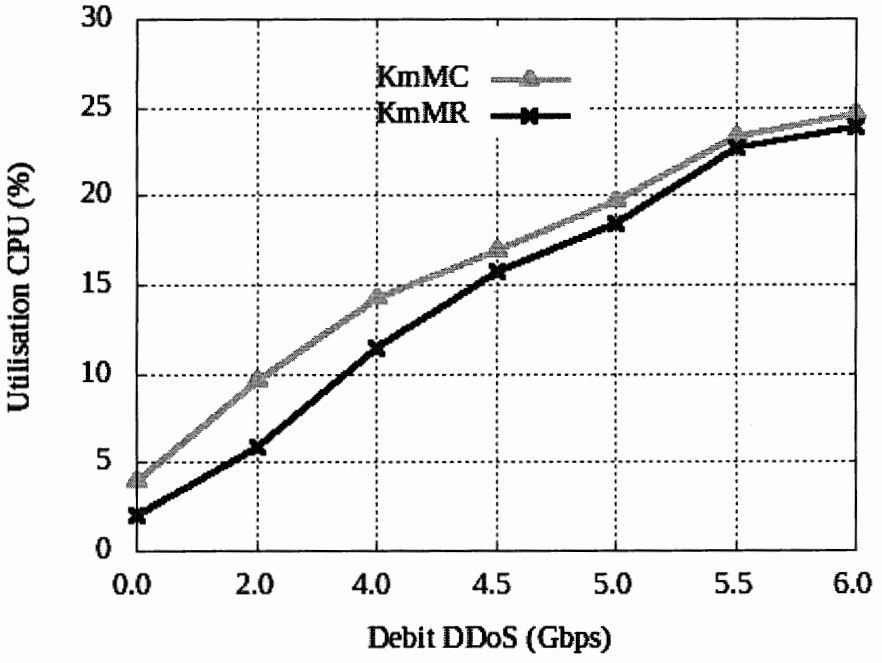

Figure 4.9 Pourcentage d'utilisation du CPU dans un environnement byzantin (n=7).

mation de la mémoire *RAM* et celle de la bande passante en fonction du débit de l'attaque *DDoS* exercée sur un nœud maître. Les résultats montrent que lorsque le débit de l'attaque *DDoS* est inférieur à 4.5 *Gbps*, KmMR utilise généralement autant ou plus de ressources que KmMC. Ceci est attendu puisque l'établissement d'un consensus dans KmMR, en utilisant *BFT-SMaRt*, nécessite un plus grand nombre d'échanges de messages que KmMC. Cependant, pour des débits d'attaque supérieurs à 4.5 *Gbps*, KmMR et KmMC ont presque le même niveau d'utilisation des ressources. En effet, Raft se met à effectuer des changements dans le cluster en vue de retrouver sa stabilité. Ceci engendre une consommation plus importante des ressources, le rendant aussi gourmand en *CPU*, *RAM* et bande passante que *BFT-SMaRt*.

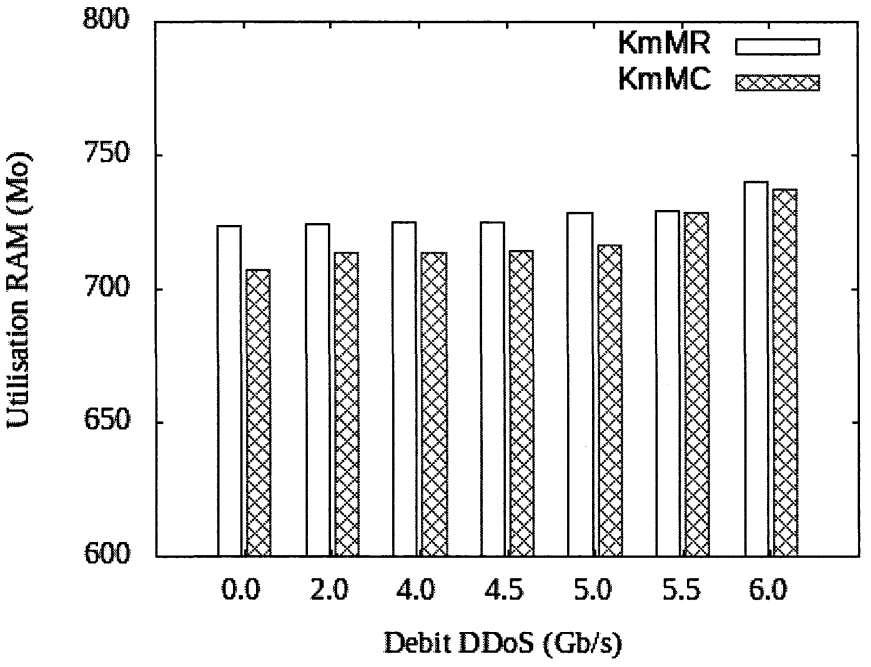

Figure 4.10 Utilisation de la RAM dans un environnement byzantin (n=7).

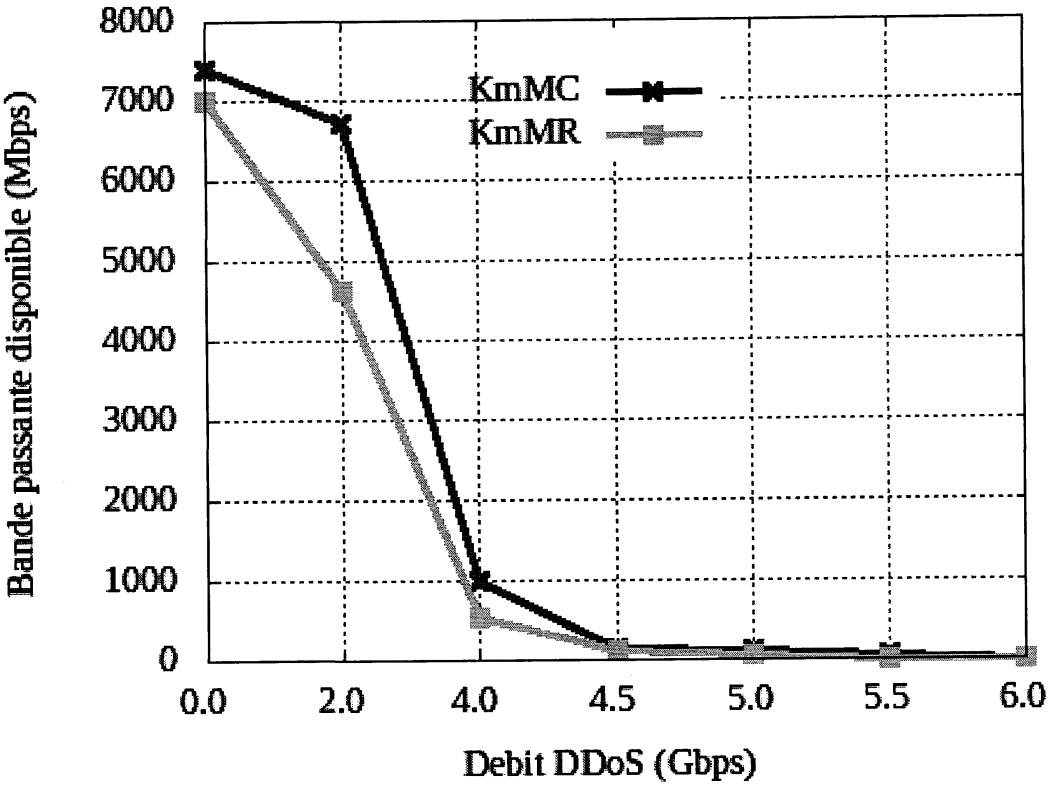

Figure 4.11 Utilisation de la bande passante dans un environnement byzantin (n=7).

4.6 Conclusion

Dans ce chapitre, nous avons présenté KmMR, notre plateforme d'orchestration de conteneurs Docker. Plus précisément, nous avons modélisé, mis en œuvre et évalué la première plateforme d'orchestration de conteneurs tolérante à tout type de faute. KmMR se base sur l'adaptation et l'intégration du protocole byzantin *BFT-SMaRt* dans Kubernetes avec la duplication du nœud maître.

L'analyse des performances de notre approche a permis de démontrer la faisabilité de notre solution, son efficacité et sa capacité à garantir la continuité de service aux applications conteneurisées malgré la présence de n'importe quelle faute dans le système.

CONCLUSION

Ces cinq dernières années ont vu le développement d'une nouvelle méthode de virtualisation dans le *cloud computing* : la conteneurisation Docker. Le principe fondateur de cette approche moderne est de créer des environnements virtuels au niveau applicatif. Ceci révolutionne la virtualisation traditionnelle qui consiste à la création d'environnements virtuels au niveau de l'infrastructure. Cette approche de virtualisation légère a pour principaux avantages l'utilisation efficace des ressources du fournisseur, l'amélioration des performances et la réduction des coûts pour les utilisateurs. Ceci implique l'apparition de nouvelles formes d'architectures qui organisent les services du *cloud* au sein de conteneurs prêts à être instanciés dans des machines virtuelles ou physiques.

Toutefois, l'utilisation des conteneurs Docker dans l'environnement *cloud* n'est pas sans défi. En effet, plusieurs difficultés doivent être résolues, parmi elles on peut citer l'orchestration (gestion des déploiements, migrations, mises à jour, etc.) et la garantie de la continuité de service (tolérance aux fautes). Comme nous l'avons présenté, le défi d'orchestration de conteneurs Docker est résolu par l'adoption de Kubernetes comme plateforme évolutive d'orchestration des applications conteneurisées. Quant à la continuité de service, l'étude des travaux de recherche effectués dans ce domaine montre que Kubernetes ne prend pas en compte toutes les fautes qui peuvent être à l'origine d'un dysfonctionnement. Nous regroupons ces fautes en deux catégories : les fautes non-byzantines qui caractérisent les arrêts de processus, et les fautes byzantines qui modélisent toute sorte de faute, comme par exemple des attaques malicieuses. Kubernetes utilise Raft, un proto-

cole de consensus non-byzantin. Par conséquent, il n'est pas résistant face aux fautes byzantines.

Notre contribution consiste à surmonter cette limitation constatée dans Kubernetes. Nous avons commencé par formuler le problème de tolérance aux fautes byzantines dans Kubernetes. Puis, nous avons proposé KmMR (Kubernetes multi-maîtres robuste) comme une nouvelle plateforme d'orchestration de conteneurs Docker. Cette nouvelle plateforme est tolérante à toute sorte de faute. Notre approche se base sur l'adoption et l'intégration du protocole byzantin *BFT-SMaRt* dans l'environnement Kubernetes avec la duplication du nœud maître.

Les performances de KmMR ont été évaluées et comparées à celles de la plateforme Kubernetes multi-maîtres classique (KmMC). Les résultats obtenus montrent que l'approche classique (KmMC) est efficace et robuste dans un environnement non-byzantin. Cependant, dans un environnement byzantin, KmMR garantit la continuité du service, contrairement à KmMC qui s'effondre devant l'ampleur des fautes byzantines. Dans un tel environnement, lorsque la charge de la faute byzantine est faible, KmMR utilise généralement autant ou plus de ressources que KmMC. Toutefois, pour une charge importante de la faute byzantine, les deux protocoles consomment des niveaux de ressources équivalents.

Nous envisageons dans nos travaux futurs de concevoir un algorithme de consensus hybride et intelligent, capable d'agir comme Raft quand l'environnement Kubernetes est non-byzantin et comme *BFT-SMaRt* lorque l'environnement est byzantin.

Nous comptons aussi varier davantage les types des attaques exercées sur notre plateforme.

RÉFÉRENCES

Abdelbaky, M., Diaz-Montes, J., Parashar, M., Unuvar, M. et Steinder, M. (2015). Docker containers across multiple clouds and data centers. Dans *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, 368–371. http://dx.doi.org/10.1109/UCC.2015.58

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. et Steinder, M. (2015). Performance evaluation of microservices architectures using containers. Dans *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*, 27–34. IEEE.

Amir, Y., Coan, B., Kirsch, J. et Lane, J. (2011). Prime : Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, *8*(4), 564–577.

Apache (2014). Mesos project. Récupéré le 2018-11-19 de https://github.com/apache/mesos

Aublin, P.-L. (2014). *Vers des protocoles de tolérance aux fautes Byzantines efficaces et robustes.* (Thèse de doctorat). Université de Grenoble.

Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V. et Vukolić, M. (2015). The next 700 BFT protocols. *ACM Transactions on Computer Systems (TOCS)*, *32*(4), 12.

Beloussov, S. M., Protassov, S. S. et Tormasov, A. G. (2008). Virtual private server with enhanced security. US Patent 7,461,144.

Bernstein, D. (2014). Containers and cloud : From lxc to Docker to Kubernetes. *IEEE Cloud Computing*, (3), 81–84.

Bessani, A. N., Santos, M., Felix, J., Neves, N. F. et Correia, M. (2013). On the efficiency of durable state machine replication. Dans *USENIX Annual Technical Conference*, 169–180.

Bila, N., Dettori, P., Kanso, A., Watanabe, Y. et Youssef, A. (2017). Leveraging the serverless architecture for securing linux containers. Dans *Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on*, 401–404. IEEE.

BISWAS, S. (2018). A guide to the linux top command. Récupéré le 2018-10-01 de `https://www.booleanworld.com/guide-linux-top-command/`

Blake Mizerany, X. L. et Qin, Y. (2018). The raft consensus algorithm. Récupéré le 2018-10-01 de `https://raft.github.io//#implementations`

Bracha, G. et Toueg, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, *32*(4), 824–840.

Buchman, E. (2016). *Tendermint : Byzantine fault tolerance in the age of blockchains.* (Thèse de doctorat).

Bui, T. (2015). Analysis of Docker security. *arXiv preprint arXiv :1501.02967.*

Burns, B., Grant, B., Oppenheimer, D., Brewer, E. et Wilkes, J. (2016). Borg, omega, and Kubernetes. *Queue*, *14*(1), 10.

Castro, M. et Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, *20*(4), 398–461.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M. et Riche, T. (2009a). Upright cluster services. Dans *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 277–290. ACM.

Clement, A., Wong, E. L., Alvisi, L., Dahlin, M. et Marchetti, M. (2009b). Making byzantine fault tolerant systems tolerate byzantine faults. Dans *NSDI*, volume 9, 153–168.

CloudNativeCon (2018). Cncf : The cloud native computing foundation. Récupéré le 2018-11-13 de `https://www.cncf.io/`

Combe, T., Martin, A. et Di Pietro, R. (2016). To docker or not to docker : A security perspective. *IEEE Cloud Computing*, *3*(5), 54–62.

CoreOS (2014). Fleet project. Récupéré le 2018-11-19 de `https://github.com/coreos/fleet`

CoreOS (2018). Coreos etcd. Récupéré le 2018-10-01 de `https://coreos.com/etcd/`

Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, *34*(2), 56–78.

Czajkowski, G. (2000). Application isolation in the Java virtual machine. Dans *ACM Sigplan Notices*, volume 35, 354–366. ACM.

Cérin, C., Menouer, T., Saad, W. et Abdallah, W. B. (2017). A new Docker swarm scheduling strategy. Dans *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, 112–117. http://dx.doi.org/10.1109/SC2.2017.24

Danny, A. (2013). Virtualisation.

Docker (2016a). Docker hub. Récupéré le 2018-09-10 de https://hub.docker.com/

Docker (2016b). Libcontainer project. Récupéré le 2018-11-19 de https://github.com/docker/libcontainer

Docker (2018). Docker swarm strategies. Récupéré le 2018-11-16 de https://docs.docker.com/swarm/scheduler/strategy/

Docker (2018). Swarm filters. Récupéré le 2018-11-16 de https://docs.docker.com/swarm/scheduler/filter/

Dua, R., Raja, A. R. et Kakadia, D. (2014). Virtualization vs containerization to support PaaS. Dans *2014 IEEE International Conference on Cloud Engineering*, 610–614. http://dx.doi.org/10.1109/IC2E.2014.41

Eberbach, E. et Reuter, A. (2015). Toward el dorado for cloud computing : Lightweight VMs, containers, meta-containers and oracles. Dans *Proceedings of the 2015 European Conference on Software Architecture Workshops*, p. 13. ACM.

Fayyad-Kazan, H., Perneel, L. et Timmerman, M. (2013). Benchmarking the performance of microsoft hyper-v server, VMware ESXi and Xen hypervisors. *Journal of Emerging Trends in Computing and Information Sciences*, *4*(12), 922–933.

Felter, W., Ferreira, A., Rajamony, R. et Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. Dans *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, 171–172. IEEE.

Fischer, M. J., Lynch, N. A. et Paterson, M. S. (1982). *Impossibility of Distributed Consensus with One Fault Process*. Rapport technique, YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE.

Foundation, T. L. (2017). Cncf members. Récupéré le 2018-11-16 de https://www.cncf.io/about/members/

Gilbert, S., Saia, J., King, V. et Young, M. (2012). Resource-competitive analysis : A new perspective on attack-resistant distributed computing. Dans *Proceedings of the 8th International Workshop on Foundations of Mobile Computing*, p. 1. ACM.

GitHub (2018). BFT-SMaRt library. Récupéré le 2018-10-01 de `https://github.com/bft-smart/library`

Github (2018). Cncf k8s. Récupéré le 2018-11-20 de `https://github.com/cncf/k8s-conformance`

Go (2018). The go programming language. Récupéré le 2018-10-01 de `https://golang.org/`

Google (2018). Kubernetes. Récupéré le 2018-11-16 de `https://kubernetes.io/`

Goth, G. (2007). Virtualization : Old technology offers huge new potential. *IEEE Distributed Systems Online*, *8*(2).

Hayes, J. P. (1976). A graph model for fault-tolerant computing systems. *IEEE Transactions on Computers*, (9), 875–884.

Heidari, P., Lemieux, Y. et Shami, A. (2016). QoS assurance with light virtualization - a survey. Dans *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 558–563. `http://dx.doi.org/10.1109/CloudCom.2016.0097`

Hess, K. et Newman, A. (2010). *Virtualisation en pratique*. Pearson.

Huang, W., Liu, J., Abali, B. et Panda, D. K. (2006). A case for high performance computing with virtual machines. Dans *Proceedings of the 20th annual international conference on Supercomputing*, 125–134. ACM.

Hwang, J., Zeng, S., Wu, F. et Wood, T. (2013). A component-based performance comparison of four hypervisors. Dans *IM*, 269–276.

iPerf (2018). Iperf. Récupéré le 2018-10-01 de `https://iperf.fr/fr/iperf-doc.php`

Johnson, B. (1984). Fault-tolerant microprocessor-based systems. *IEEE Micro*, (6), 6–21.

Joy, A. M. (2015a). Performance comparison between linux containers and virtual machines. Dans *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, 342–346. IEEE.

Joy, A. M. (2015b). Performance comparison between linux containers and virtual machines. Dans *2015 International Conference on Advances in Computer Engineering and Applications*, 342–346. http://dx.doi.org/10.1109/ICACEA.2015.7164727

K8s (2014). Code source Kubernetes. Récupéré le 2018-11-16 de https://github.com/kubernetes/kubernetes/

Kolahi, S. S., Treseangrat, K. et Sarrafpour, B. (2015). Analysis of UDP DDoS flood cyber attack and defense mechanisms on web server with linux ubuntu 13. Dans *Communications, Signal Processing, and their Applications (ICCSPA), 2015 International Conference on*, 1–5. IEEE.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A. et Wong, E. (2009). Zyzzyva : Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, *27*(4), 7.

Kovács, J., Kacsuk, P. et Emődi, M. (2018). Deploying Docker swarm cluster on hybrid clouds using occopus. *Advances in Engineering Software*, *125*, 136 – 145. http://dx.doi.org/https://doi.org/10.1016/j.advengsoft.2018.08.001. Récupéré de http://www.sciencedirect.com/science/article/pii/S0965997817308918

Kratzke, N. et Quint, P.-C. (2017). Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. *Journal of Systems and Software*, *126*, 1–16.

Kubernetes (2014a). Les fonctions de priorité Kubernetes. Récupéré le 2018-11-18 de https://github.com/kubernetes/kubernetes/tree/release-1.1/plugin/pkg/scheduler/algorithm/priorities/

Kubernetes (2014b). Les prédicats Kubernetes. Récupéré le 2018-11-18 de https://github.com/kubernetes/kubernetes/tree/release-1.1/plugin/pkg/scheduler/algorithm/predicates

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*(7), 558–565.

Lamport, L. (2011). Byzantizing paxos by refinement. Dans *International Symposium on Distributed Computing*, 211–224. Springer.

Lamport, L. *et al.* (2001). Paxos made simple. *ACM Sigact News*, *32*(4), 18–25.

Lamport, L., Malkhi, D. et Zhou, L. (2010). Reconfiguring a state machine. *ACM SIGACT News*, *41*(1), 63–73.

Lamport, L., Shostak, R. et Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *4*(3), 382–401.

Li, W. et Kanso, A. (2015). Comparing containers versus virtual machines for achieving high availability. Dans *2015 IEEE International Conference on Cloud Engineering*, 353–358. http://dx.doi.org/10.1109/IC2E.2015.79

Lim, J., Suh, T., Gil, J. et Yu, H. (2014). Scalable and leaderless byzantine consensus in cloud computing environments. *Information Systems Frontiers*, *16*(1), 19–34.

Liu, D. et Zhao, L. (2014). The research and implementation of cloud computing platform based on docker. Dans *2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing(ICCWAMTIP)*, 475–478. http://dx.doi.org/10.1109/ICCWAMTIP.2014.7073453

Lonea, A. M., Popescu, D. E. et Tianfield, H. (2013). Detecting DDoS attacks in cloud computing environment. *International Journal of Computers Communications & Control*, *8*(1), 70–78.

Luzzardi, A. et Victor, V. (2014). Swarm : a Docker-native clustering system. Récupéré le 2018-11-13 de https://github.com/docker/swarm/

Madhumathi, R. (2018). The relevance of container monitoring towards container intelligence. Dans *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 1–5. http://dx.doi.org/10.1109/ICCCNT.2018.8493766

Malkhi, D. et Reiter, M. (1998). Byzantine quorum systems. *Distributed computing*, *11*(4), 203–213.

Manu, A., Patel, J. K., Akhtar, S., Agrawal, V. et Murthy, K. B. S. (2016a). Docker container security via heuristics-based multilateral security-conceptual and pragmatic study. Dans *Circuit, Power and Computing Technologies (ICCPCT), 2016 International Conference on*, 1–14. IEEE.

Manu, A., Patel, J. K., Akhtar, S., Agrawal, V. et Murthy, K. B. S. (2016b). A study, analysis and deep dive on cloud PaaS security in terms of Docker container security. Dans *Circuit, Power and Computing Technologies (ICCPCT), 2016 International Conference on*, 1–13. IEEE.

Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U. et Rana, O. F. (2018). Characterising resource management performance in Kubernetes. *Computers & Electrical Engineering*, *68*, 286–297.

Mell, P., Grance, T. *et al.* (2011). The NIST definition of cloud computing.

Moga, A., Sivanthi, T. et Franke, C. (2016). OS-level virtualization for industrial automation systems : are we there yet ? Dans *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 1838–1843. ACM.

Naik, N. (2016). Building a virtual system of systems using Docker swarm in multiple clouds. Dans *Systems Engineering (ISSE), 2016 IEEE International Symposium on*, 1–3. IEEE.

Netto, H. V., Lung, L. C., Correia, M., Luiz, A. F. et de Souza, L. M. S. (2017). State machine replication in containers managed by Kubernetes. *Journal of Systems Architecture*, *73*, 53–59.

Nguyen, N. et Bein, D. (2017). Distributed MPI cluster with Docker swarm mode. Dans *2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC)*, 1–7. `http://dx.doi.org/10.1109/CCWC.2017.7868429`

Oliveira, C., Lung, L. C., Netto, H. et Rech, L. (2016). Evaluating Raft in docker on Kubernetes. Dans *International Conference on Systems Science*, 123–130. Springer.

Oliveira, C. P. (2017). Raft on Kubernetes. Récupéré le 2018-10-20 de `https://github.com/caiopo/raft`

Ongaro, D. et Ousterhout, J. K. (2014). In search of an understandable consensus algorithm. Dans *USENIX Annual Technical Conference*, 305–319.

Ops, B. (2016). Denial-of-service attack–dos using Hping3 with spoofed IP in kali linux. *BlackMORE Ops. BlackMORE Ops*, *17*.

Pahl, C. (2015). Containerization and the PaaS cloud. *IEEE Cloud Computing*, *2*(3), 24–31.

Pease, M., Shostak, R. et Lamport, L. (1980). Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, *27*(2), 228–234.

Peinl, R., Holzschuher, F. et Pfitzer, F. (2016). Docker cluster management for the cloud-survey results and own solution. *Journal of Grid Computing*, *14*(2), 265–282.

Perronne, L. (2016). *Vers des protocoles de tolérance aux fautes byzantines efficaces et robustes.* (Thèse de doctorat). Université Grenoble Alpes.

Qian, L., Luo, Z., Du, Y. et Guo, L. (2009). Cloud computing : An overview. Dans *IEEE International Conference on Cloud Computing*, 626–631. Springer.

Rensin, D. K. (2015). Kubernetes-scheduling the future at cloud scale.

Sanfilippo (2014). Hping3. Récupéré le 2018-10-01 de `http://www.hping.org/hping3.html`

Schlichting, R. D. et Schneider, F. B. (1983). Fail-stop processors : an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)*, *1*(3), 222–238.

Schmitt, M. (2014). *Architecture de sécurité pour les données dans un contexte d'informatique en nuage.* (Mémoire de maîtrise). Université du Québec à Montréal.

Schneider, F. B. (1986). *Abstractions for Fault Tolerance in Distributed Systems.* Rapport technique, CORNELL UNIV ITHACA NY DEPT OF COMPUTER SCIENCE.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys (CSUR)*, *22*(4), 299–319.

Schroeder, B., Wierman, A. et Harchol-Balter, M. (2006). Open versus closed : a cautionary tale. USENIX.

Sefraoui, O., Aissaoui, M. et Eleuldj, M. (2012). Openstack : toward an open-source solution for cloud computing. *International Journal of Computer Applications*, *55*(3), 38–42.

Sharma, P., Chaufournier, L., Shenoy, P. et Tay, Y. (2016). Containers and virtual machines at scale : A comparative study. Dans *Proceedings of the 17th International Middleware Conference*, p. 1. ACM.

Sill, A. (2015). Emerging standards and organizational patterns in cloud computing. *IEEE Cloud Computing*, *2*(4), 72–76.

Somani, G., Gaur, M. S., Sanghi, D., Conti, M. et Buyya, R. (2017). DDoS attacks in cloud computing : Issues, taxonomy, and future directions. *Computer Communications*, *107*, 30–48.

Soppelsa, F. et Kaewkasi, C. (2016). *Native Docker Clustering with Swarm.* Packt Publishing Ltd.

Soriga, S. G. et Barbulescu, M. (2013). A comparison of the performance and scalability of Xen and KVM hypervisors. Dans *Networking in Education and Research, 2013 RoEduNet International Conference 12th Edition*, 1–6. IEEE.

Sousa, J., Alchieri, E. et Bessani, A. (2013). State machine replication for the masses with BFT-SMaRt.

Sousa, J. et Bessani, A. (2012). From byzantine consensus to BFT state machine replication : A latency-optimal transformation. Dans *Dependable Computing Conference (EDCC), 2012 Ninth European*, 37–48. IEEE.

Sousa, J., Bessani, A. et Vukolic, M. (2018). A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. Dans *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 51–58. IEEE.

Tay, Y. C., Gaurav, K. et Karkun, P. (2017). A performance comparison of containers and virtual machines in workload migration context. Dans *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 61–66.
`http://dx.doi.org/10.1109/ICDCSW.2017.44`

Tchana, A.-B. (2011). *Système d'Administration Autonome Adaptable : application au Cloud.* (Thèse de doctorat). Institut National Polytechnique de Toulouse.

Van Renesse, R., Schiper, N. et Schneider, F. B. (2015). Vive la différence : Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, *12*(4), 472–484.

Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C. et Verissimo, P. (2013). Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, *62*(1), 16–30.