UNIVERSITÉ DU QUÉBEC À MONTRÉAL

LA COMMUNICATION ET LE PLACEMENT DE CONTENEURS DOCKER DANS LE FOG COMPUTING

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

EL HOUSSINE BOURHIM

JANVIER 2020

REMERCIEMENTS

Au termes de ce mémoire je présente mes sincères remerciements à Madame Pr. Halima ELBIAZE ma directrice de recherche, pour son soutien scientifique et financier, ses encouragements et ses orientations précieuses tout au long de ma maîtrise. Je tiens à lui exprimer ma reconnaissance pour sa disponibilité ainsi que ses conseils avisés, et que ce travail soit le modeste témoignage de ma haute considération profond du respect.

Je remercie aussi Dr. Obaid ABBASI et Dr. Wael JAAFAR qui m'ont énormément aidé dans la réalisation de ce mémoire à travers leurs conseils et leurs remarques.

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 remercie de tout coeur la Fondation de l'UQAM, pour les bourses d'excellence qui m'ont été attribuées ; ces bourses ont grandement facilité mes activités de recherche.

Sans oublier de remercier vivement le corps professoral de la faculté des sciences de l'UQAM. Et tous mes collègues au sein du laboratoire TRIME et de l'Université du Québec à Montréal (UQAM) avec qui j'ai passé de très bons moments.

Mes remerciements s'adressent également à mes parents, frères et soeurs et amis qui ont été toujours présents pour moi et qui m'ont toujours soutenu et encouragé.

El Houssine BOURHIM

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ACRONYMES

API	*Application Programming Interface*
CNI	*Container Network Interface*
CNM	*Container Network Model*
CPGA	*Container Placement based Genetic Algorithm*
CPU	*Central Processing Unit*
DPDK	*Data Plane Development Kit*
FSC	*Fog Service Controller*
GA	*Genetic Algorithm*
GPL	*General Public License*
IdO	Internet des Objets
ILP	*Integer Linear Programming*
IoT	*Internet of Things*
IP	*Internet Protocol*
IPC	*Inter-Process Communication*
iWARP	*Internet Wide Area RDMA Protocol*
JVM	*Java Virtual Machine*
LISP	*Locator ID Separation Protocol*

MB	*Megabytes*
MIPS	*Millions of Instructions Per Second*
NAT	*Network Address Translation*
NFV	*Network Function Virtualization*
NIC	*Network Interface Card*
ODP	*Open vSwitch datapath*
OVS	*Open Virtual Switch*
PC	*Personal Computer*
PCI	*Peripheral Component Interconnect*
PCIe	*Peripheral Component Interconnect express*
PMD	*Poll Mode Driver*
QoS	*Quality of Service*
RAM	*Random Access Memory*
RDMA	*Remote Direct Memory Access*
RoCE	*RDMA over Converged Ethernet*
SDN	*Software-Defined Networking*
TCP	*Transmission Control Protocol*
TRIME	Télécommunication, Réseaux, Informatique Mobile et Embarquée
UDP	*User Datagram Protocol*
vEth	*Virtual Ethernet*
VIA	*Virtual Interface Architecture*

VLAN	*Virtual Local Area Network*
VM	*Virtual Machine*
VMM	*Virtual Machines Monitor*
VNIC	*Virtual Network Interface Card*
VPN	*Virtual Private Network*
VTEP	*VXLAN Tunnel End Point*
VXLAN	*Virtual Extensible Local Area Network*

RÉSUMÉ

Au cours des dernières années, l'informatique en brouillard (*fog computing*) est devenue de plus en plus populaire avec l'avènement des applications Internet des objets (IdO). Ces dernières sont caractérisées par des exigences strictes de qualité de service (QoS), que l'infrastructure actuelle du cloud ne peut pas satisfaire.

Le *fog computing* permet d'étendre l'architecture du *cloud computing* vers un réseau de noeuds périphériques proches des utilisateurs, composé d'un grand nombre d'appareils distribués et hétérogènes qui communiquent et coopèrent entre eux afin d'exécuter des services. Cette infrastructure s'appuie sur la virtualisation applicative par conteneurs Docker. Ceci est dû principalement aux avantages des conteneurs en termes d'utilisation de ressources et de temps de déploiement et d'exécution des services. Par conséquent, les fournisseurs de *fog computing* adoptent les conteneurs Docker comme étant l'outil de virtualisation des applications.

Toutefois, l'hétérogénéité des noeuds du *fog* et des protocoles de communication entre conteneurs ont un impact majeur sur les processus d'allocation des ressources aux applications IdO, et sur les performances globales de l'infrastructure du *fog*.

L'objectif de ce mémoire est de proposer une solution de placement de conteneurs dans une infrastructure de *fog computing*. Vu que cette plateforme prend en considération l'aspect hétérogène des noeuds et des protocoles de communication entre conteneurs, notre solution vise à minimiser le délai de réponse aux requêtes des utilisateurs.

Nous commençons par proposer un mécanisme de communication entre conteneurs RDMA (*Remote Direct Memory Access*), qui satisfait les exigences de performances de communication entre conteneurs. Ensuite, nous proposons une stratégie de placement de conteneurs dans le *fog* baptisée CPGA (*Container Placement based Genetic Algorithm*), qui tient en compte de l'aspect hétérogène des noeuds et des protocoles de communication entre conteneurs.

Les résultats montrent que RDMA surpasse les performances des solutions existantes, et réalise d'excellentes performances en termes d'utilisation du processeur, de débit et de latence. En outre, la stratégie de placement CPGA montre un bon compromis de performances en étant capable de fournir une solution presque optimale en un temps de calcul acceptable.

Mots clés : Fog computing, conteneurs Docker, communication inter-conteneurs, RDMA, algorithmes génétiques

INTRODUCTION

Contexte

Le nombre d'utilisateurs et d'objets connectés à Internet a augmenté d'une façon significative au cours des dernières années. D'ici 2025, près de 75 milliards d'objets seront connectés et produiront un véritable déluge de données pour différents services (Department, 2019).

En conséquence, les infrastructures actuelles de *cloud*, très centralisées, pourraient atteindre leur limite. Les capacités des réseaux et des centres de données existants risquent en effet la saturation, ce qui ferait exploser le temps de latence et rendrait le traitement des données incompatible avec les besoins d'analyse en temps réel. Avec le *cloud*, les serveurs capables de traiter les données sont distants et leur temps de réponse dépend de facteurs que l'utilisateur ne peut pas contrôler, par exemple la charge de travail des serveurs, l'encombrement du réseau ..etc.

C'est ici que le *cloud computing* cède la place au *fog computing*. Ce dernier se distingue par sa proximité aux utilisateurs. Il leur offre des services hébergés (ressources de traitement, espaces de stockage, applications...) géographiquement situés en périphérie immédiate des réseaux locaux, sans recourir au *cloud* ou à un centre de données. Ce faisant, le *fog computing* réduit le temps de gestion des données et améliore la qualité du service rendu.

Motivation et problématique

En 2025, l'Internet des objets (en anglais, *Internet of Things IoT*) qui définit l'interconnexion entre Internet et des objets, des lieux et des environnements phy-

siques, devrait inclure environ 75 milliards d'objets connectés dans le monde entier (Department, 2019). Tous ces objets génèrent des données en continu, qui nécessitent d'être stockées et évaluées en temps réel pour des applications critiques. Ceci représente une tâche que les solutions actuelles de *cloud* centralisé ne sont pas en mesure de maîtriser (Dastjerdi et Buyya, 2016; Yannuzzi *et al.*, 2014).

Par conséquent, le *fog computing* est proposé pour résoudre ce problème comme étant une infrastructure de traitement et de stockage de données dans des noeuds à proximité des utilisateurs finaux. Cela permettra de réduire le temps de réponse des requêtes des utilisateurs, et satisfaire les exigences du temps réel des applications IoT.

Actuellement, les fournisseurs du *fog* utilisent la conteneurisation Docker comme support technologique pour résoudre les problèmes de performances causés par la virtualisation traditionnelle (utilisation abusive des ressources) (Sri Raghavendra et Chawla, 2018; Puliafito *et al.*, 2019). Ainsi, l'unité d'allocation de ressources dans le *fog* est le conteneur Docker.

Cependant, quant à la performance de mise en réseau de conteneurs, nous constatons que les mécanismes fournis par Docker sont limités. Docker utilise des logiciels virtuels (*docker bridge*, weave, flannel) pour transmettre les données entre les conteneurs, cela induit une surcharge de traitement du processeur, ce qui dégrade les performances réseau de la communication entre les conteneurs (Yu *et al.*, 2016).

Afin de surmonter cette limitation, nous proposons dans ce travail une solution qui se base sur les mécanismes de zéro-copie et l'accès direct à la mémoire à savoir RDMA. Cette dernière permet d'éviter toute limitation logicielle ou du système d'exploitation dans l'environnement de Docker, ce qui permet d'améliorer les performances réseau de la communication entre les conteneurs.

De plus, nous proposons une stratégie de placement de conteneurs Docker dans l'environnement du *fog*, qui se base sur l'algorithme génétique, nommé CPGA, notre stratégie permet de trouver le meilleur plan de placement de conteneurs dans le *fog*, tout en tenant compte de l'hétérogénéité des noeuds et des protocoles de communication entre conteneurs.

Contributions

Nous proposons une plateforme de *fog computing* améliorant les performance réseaux. Nos contributions dans ce projet se résument en :

1. Un mécanisme de communication entre les conteneurs améliorant les performances de conteneur en termes de l'utilisation de ressource du CPU, de débit, et de latence. Cette contribution est détaillée dans le chapitre 2.
2. Une nouvelle stratégie de placement de conteneurs dans le *fog computing* qui tient en compte les protocoles de communication entre les conteneurs, et qui répond aux exigences de qualité de service des applications IoT. Cette contribution est détaillée dans le chapitre 3.

Plan du mémoire

Ce mémoire se compose de quatre chapitres et s'organise comme suit :

Le premier chapitre étant l'introduction, nous présentons le contexte du sujet, les éléments de la problématique ainsi que les objectifs de recherche.

Le deuxième chapitre fournit un aperçu des concepts associés à la conteneurisation dans un environnement d'informatique géodistribué (*fog computing*). Plus précisément, nous décrivons le déploiement des conteneurs Docker dans le *fog*. À partir de là nous tirons les problématiques et l'intérêt de l'utilisation des conteneurs Docker dans le *fog computing*.

Le troisième chapitre sera consacré aux protocoles de communication entre les conteneurs dans un environnement d'informatique géodistribué (*fog computing*). Nous décrivons les différents modes et technologies de communication entre les conteneurs, ensuite nous comparons les performances de notre solution par rapport aux solutions existantes.

Le quatrième chapitre sera consacré à l'optimisation de placement des conteneurs dans le *fog computing*. Nous commençons par formuler le problème de placement des conteneurs comme un programme linéaire en nombre entier. Ensuite nous décrivons notre solution heuristique basée sur un algorithme génétique. Finalement, nous comparons les performances de trois stratégies de placement de conteneurs par rapport aux performances de notre solution et de la solution optimale.

Enfin, le dernier chapitre sera consacré à la conclusion du mémoire.

CHAPITRE I

LA CONTENEURISATION ET LE FOG COMPUTING

1.1 Introduction

Afin de mieux cerner l'étendu de notre problématique, nous présentons dans ce premier chapitre les concepts liés à l'informatique géodistribuée (*fog 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'informatique géodistribué (*fog computing*)

1.2.1 Définition

L'informatique géodistribuée (appelé aussi informatique en brouillard, en anglais *fog computing*) est une infrastructure informatique de traitement de données qui a été proposée par Cisco en 2012 (Bonomi *et al.*, 2012). Le *fog computing* fait référence à une infrastructure décentralisée dans laquelle les ressources de calcul et analytiques sont distribuées aux emplacements les plus logiques et les plus efficaces, en tout point du continuum entre la source de données (les objets connectés) et le centre de données du Cloud (Qian *et al.*, 2009).

Dans l'environnement du *fog computing*, la majeure partie du traitement a lieu au sein des noeuds à la périphérie du réseau tels que les appareils mobiles, les points d'accès, les routeurs, les stations de base, etc. (Cisco, 2016).

Le *fog computing* a pour but d'optimiser les communications entre un grand nombre d'objets connectés (IoT) et les services de traitement distants (*Cloud*), en tenant compte d'une part des volumes de données considérables engendrés par ce type d'architecture (mégadonnées) et d'autre part de la variabilité de la latence dans un réseau distribué, tout en donnant un meilleur contrôle sur les données transmises (Cisco, 2016).

1.2.2 Architecture

L'architecture du *fog computing* est composée de trois couches, comme montré dans la figure 1.1 :

1. Couche IoT/Utilisateur : cette couche comprend tous les dispositifs d'une architecture d'internet des objets tels que les appareils mobiles, les caméras de surveillance, les véhicules, etc. (Cisco, 2016). Les données générées au niveau de la couche IoT/Utilisateur sont soit traitées directement sur le périphérique, soit transmises à un noeud de la couche *fog*.

2. Couche Fog : cette couche comprend un certain nombre de noeuds performants et proches des noeuds IoT qui reçoivent les données de la couche périphériques (IoT) tels que les routeurs, les stations de base, les ordinateurs, etc. (Cisco, 2016), les traitent, ou les acheminent au *cloud* si nécessaire.

3. Couche Cloud : cette couche comprend l'ensemble des serveurs qui composent les centres de données du *cloud computing*.

Dans l'architecture du *fog computing*, les ressources de stockage et de traitement des données sont externalisées du *cloud* et fournies de manière décentralisée sur une couche intermédiaire (Couche *Fog*).

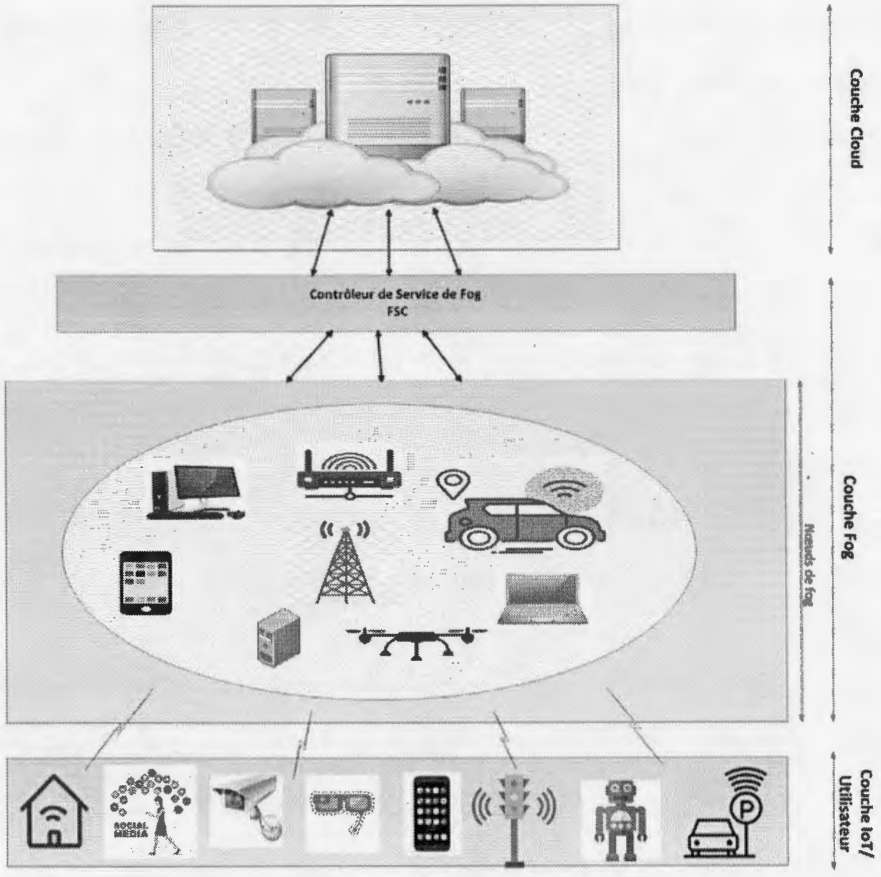

Figure 1.1 Architecture du Fog Computing (Bourhim *et al.*, 2019)

1.2.3 Les caractéristiques

Les caractéristiques du *fog computing* peuvent être résumées comme suit (Ai *et al.*, 2018; Yi *et al.*, 2015) :

- Localisation et faible latence : le *fog computing* prend en charge la connaissance de l'emplacement où les nœuds du *fog* sont déployés. De plus, comme le *fog* est plus proche des terminaux (les utilisateurs finaux), cela permet de réduire le temps de latence lors du traitement des données dans les terminaux.
- Répartition géographique : contrairement au *cloud* centralisé, les appli-

cations et les services fournis par le *fog* sont distribués et peuvent être déployés n'importe où.

- Évolutivité : le *fog* supporte des réseaux de capteurs à grande échelle qui surveillent leur environnement. Le *fog* fournit des ressources de calcul et de stockage distribuées qui peuvent fonctionner dans un tel environnement.
- Prise en charge de la mobilité : l'un des aspects importants des applications du *fog* est la possibilité de se connecter directement à des périphériques mobiles et d'activer des méthodes de mobilité, telles que le protocole LISP (*Locator ID Separation Protocol*) (Meyer, 2008).
- Interactions en temps réel : les applications du *fog computing* permettent des interactions en temps réel entre les nœuds du *fog*, plutôt que le traitement par lots utilisé dans le *cloud*.
- Hétérogénéité : les nœuds du *fog computing* sont conçus par différents fabricants. Ils se présentent donc sous différentes formes et doivent être déployés en fonction de leurs plateformes. Le *fog* a la capacité de travailler sur différentes plateformes.
- Interopérabilité : les composants du *fog* peuvent interagir et fonctionner avec différents domaines et entre différents fournisseurs de services.

1.2.4 Les avantages du *fog computing*

Le *fog computing* étend le modèle du *cloud computing* à la périphérie du réseau. Bien que le *fog* et le *cloud* utilisent des ressources similaires (traitement, stockage et réseau) et utilisent les mêmes mécanismes et attributs (virtualisation, multi-entité) (Li *et al.*, 2018), le *fog computing* apporte de nombreux avantages aux appareils IoT (terminaux finaux) (Liu *et al.*, 2017) . Ces avantages peuvent être résumés comme suit :

- Faible latence : le *fog computing* peut prendre en charge des services en

temps réel (par exemple vidéo streaming, jeux en ligne, etc.) (Peralta *et al.*, 2017).

- Distribution géographique et à grande échelle : le *fog computing* peut fournir des ressources de calcul réparties (CPU, RAM, Stockage, etc.) pour des applications volumineuses et distribuées (Peralta *et al.*, 2017).
- Frais d'exploitation moins élevés : économie de la bande passante du réseau, en traitant les données localement au lieu de les envoyer vers le *cloud* (Cisco, 2016).
- Flexibilité et hétérogénéité : le *fog computing* permet la collaboration de différents environnements physiques et infrastructures (Cisco, 2016).
- Une plus grande agilité de l'entreprise : les applications de *fog computing* peuvent être rapidement développées et déployées. En outre, ces applications peuvent programmer la machine pour qu'elle fonctionne en fonction des besoins du client (Cisco, 2016).

1.2.5 Le *fog computing* et les applications IoT

L'architecture du *cloud computing* centralisée actuelle est confrontée à de grands défis pour les applications IoT. Par exemple, le *cloud* ne prend pas en charge les applications IoT sensibles au délai, telles que le vidéo streaming, les jeux en ligne, les véhicules connectés (Mouradian *et al.*, 2017). En outre, l'environnement de *cloud* ne supporte pas la localisation des noeuds car le modèle de cloud est centralisé. Le *fog computing* est donc conçu pour être en mesure de relever ces défis. Le tableau 1.1 résume les différences entre le *cloud* et le *fog* (Atlam *et al.*, 2018).

Tableau 1.1 Comparaison entre le *cloud* et le *fog computing.*

Paramètres	**Cloud computing**	**Fog computing**
Latence	Élevée	Faible
Matériel	Stockage et puissance de calcul immense (évolutif)	Stockage et puissance de calcul limitée
Localisation des nœuds	Via Internet	Au bord du réseau local
Distance entre le client et le serveur	Plusieurs sauts	Un saut
Déploiement	Centralisé	Distribué
Mesures de sécurité	Définies	Difficiles à définir

Le *fog computing* agit comme un pont entre les dispositifs IoT et les ressources de stockage et traitement de *cloud.* Selon Cisco (Cisco, 2016), le *fog computing* fait partie du paradigme de l'informatique en nuage (*cloud*) qui rapproche le nuage de la périphérie du réseau. Il fournit un modèle virtualisé de ressources de calcul, de stockage et de réseau entre les équipements terminaux de la couche IoT et les serveurs de cloud classique (Mouradian *et al.*, 2017).

Pour accroître l'efficacité des applications IoT, la plupart des données générées par ces objets/appareils IoT doivent être traitées et analysées en temps réel. Le *fog computing* rapproche les ressources de traitement, stockage et réseau de la périphérie du réseau, ce qui permet de résoudre le problème de temps réel des dispositifs IoT et fournira des applications IoT sécurisées et efficaces (Ketel, 2017).

Le *fog computing* est considéré comme le meilleur choix pour les applications nécessitant une faible latence, telles que le vidéo streaming, les jeux en ligne, les véhicules connectés, etc. (Skarlat *et al.*, 2016). Il joue aussi un rôle essentiel dans différentes applications IoT dans de nombreux domaines importants, tels que les

applications de santé, construction, transport, ville intelligente, éducation, sécurité publique, énergie, etc.

Comme dans tout système informatique du *cloud* ou du *fog computing*, les applications des utilisateurs sont virtualisées en utilisant différents outils de virtualisation, et c'est ce que nous présenterons dans la section suivante.

1.3 La virtualisation

1.3.1 Définition de la virtualisation

La virtualisation consiste à créer une version virtuelle d'un dispositif ou d'une ressource, comme un serveur, un dispositif de stockage, un système d'exploitation ou une ressource réseau (Goth, 2007).

Cela permet aux organisations de partitionner un ordinateur ou serveur physiques en plusieurs machines virtuelles. Chaque machine virtuelle peut alors interagir de manière indépendante, et exécuter différents systèmes d'exploitation ou applications tout en partageant les ressources d'un seul ordinateur hôte. En créant plusieurs ressources à partir d'un seul ordinateur ou serveur, la virtualisation améliore l'extensibilité et les charges de travail, tout en réduisant le nombre de serveurs utilisés, la consommation d'énergie, les coûts d'infrastructure et la maintenance (Microsoft, 2019; Menascé, 2005).

1.3.2 Types de virtualisation

Dans la virtualisation, on peut distinguer quatre catégories. La première est la virtualisation serveur, qui permet à un serveur centralisé de fournir et gérer des postes individuels. La deuxième est la virtualisation applicative, qui sépare les applications du matériel et du système d'exploitation. La troisième est la virtualisation réseau, conçue pour diviser la bande passante réseau en canaux indépendants

affectés ensuite à des systèmes virtuels. La quatrième est la virtualisation du stockage, qui combine plusieurs ressources de stockage réseau en un seul dispositif de stockage auquel plusieurs utilisateurs peuvent accéder (Hess et Newman, 2010). Dans la suite de ce mémoire, nous nous intéressons à la virtualisation du serveur et la virtualisation du réseau.

1.3.3 La virtualisation serveur

La 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 *VMs* (Virtual Machines), tandis que l'hyperviseur est connu sous le nom de *VMM* (*Virtual Machines Monitor*), c'est-à-dire gestionnaire de *VM* (Diouf, 2019). Ces nouvelles notions sont définies comme suit :

1. Machine virtuelle (VM) : est un conteneur logiciel totalement isolé capable d'exécuter son système d'exploitation et ses applications comme s'il s'agissait d'un véritable ordinateur. Une VM se comporte exactement comme un ordinateur physique. Elle contient son propre matériel virtuel : *CPU*, mémoire *RAM*, disque dur et carte d'interface réseau (*NIC*) basés sur du logiciel.

2. Hyperviseur (*VMM*) : est un programme qui permet à plusieurs systèmes d'exploitation de différents types (Linux, Mac ou Windows) de partager un seul hôte matériel. Il implante les mécanismes d'isolation et de partage des ressources matérielles. L'hyperviseur contrôle les accès aux ressources, allouant à chaque VM ce dont elle a besoin et s'assurant que ces systèmes invités (ou machines virtuelles) n'interfèrent pas l'un avec l'autre. Quant à la communication avec l'extérieur, l'hyperviseur peut fournir plusieurs techniques pour rendre accessible ou non les *VMs*. Il peut réaliser cette

tâche par l'assignation d'adresses *IP* (*Internet Protocol*) et numéros de port en utilisant différentes techniques d'accès réseau aux *VMs* (mode pont (*bridge*), mode hôte (*host only*), *NAT - Network Address Translation*, réseau privée *VLAN - Virtual Local Area Network*) (Chowdhury et Boutaba, 2010).

1.3.4 La virtualisation applicative

La virtualisation applicative est un procédé permettant d'ajouter une couche d'abstraction entre le système d'exploitation et les applications. Cette technique permet d'exécuter des applications sur un poste client sans les installer sur ce poste. En effet, ces applications sont réellement installées sur un serveur virtuel distant (Gurr, 2008). La virtualisation des applications permet d'exécuter des applications depuis un poste client indépendamment du système d'exploitation. Ce procédé permet d'éliminer le problème d'incompatibilité des applications.

Les machines virtuelles applicatives les plus connues sont la *JVM (Java Virtual Machine)* et les conteneurs (Diouf, 2019).

— *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* fournit un environnement d'exécution pour les applications Java quelle que soit la plateforme. Elle assure l'indépendance du matériel et du système d'exploitation lors de l'exécution des applications Java (Czajkowski, 2000).

— Conteneurs : Meilleurs 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 (Diouf, 2019).

1.3.5 La virtualisation réseau

La virtualisation des réseaux consiste à créer des réseaux logiques qui s'appuient sur des ressources matérielles. Ce concept permet à plusieurs instances virtuelles (ou réseaux virtuels) de coexister et de partager les ressources d'une ou de plusieurs infrastructures physiques de réseaux (Barachi *et al.*, 2010). En effet, plusieurs technologies existantes de réseau se basent sur ce concept. Dans le chapitre suivant nous allons décrire en détails certaines de ces techniques, qui permettent de déployer des réseaux virtuels à travers une même infrastructure réseau physique. Parmi ces techniques, nous retrouvons : les réseaux locaux virtuels (*Virtual Local Area Network - VLAN*), les réseaux privés virtuels (*Virtual Privat Network - VPN*), les réseaux de superposition ou de recouvrement (*overlay networks*) (Chowdhury et Boutaba, 2010).

La virtualisation de réseau peut s'avérer utile à héberger des machines virtuelles et des conteneurs, ou idéalement à un fournisseur de *cloud ou fog computing*, en permettant à ceux-ci de mettre à disposition de leurs clients des environnements réseaux totalement isolés en se basant sur une infrastructure réseau commune.
Selon (Wang *et al.*, 2013), les composantes de base de la virtualisation du réseau sont :

— Cartes réseaux virtuelles (Virtual Network Interface Card - VNIC) : ces cartes sont des périphériques réseaux virtuels avec les mêmes interfaces de liaison de données qu'une carte d'interface réseau physique (*Network Interface Card - NIC*). En outre, les ressources du système traitent les VNIC comme s'il s'agissait de cartes d'interface réseau physiques.
— Commutateurs virtuels : il s'agit des logiciels intégrés à l'hyperviseur et qui se comportent comme des commutateurs matériels, ces logiciels

permettent l'interconnexion des systèmes virtuels (Machines virtuelles, Conteneurs) entre eux ou avec des éléments réseaux physiques (cartes réseaux, commutateurs, etc).

1.4 Les conteneurs Docker

1.4.1 Définition

La virtualisation par conteneurs (également connue sous le nom de conteneurisation) constitue une alternative à la virtualisation serveur. Il s'agit d'une approche qui repose sur le noyau Linux et ses fonctionnalités de virtualisation par conteneurs LXC (Linux Container) notamment :

— Le composant *cgroups* pour contrôler et limiter l'utilisation des ressources (CPU, RAM) pour un groupe de processus associé au système d'initialisation *systemd*, qui permet de définir l'espace utilisateur et de gérer les processus associés.

— Les espaces de noms ou *namespaces* qui permettent de créer des environnements sécurisés de manière à isoler les conteneurs et empêcher par exemple qu'un groupe puisse accéder aux ressources des autres groupes.

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 Docker

Docker est l'incarnation la plus moderne de la conteneurisation (Madhumathi, 2018). Il s'agit d'un projet libre et ouvert développé par la société *Dotcloud* (renommée par la suite Docker).

Les conteneurs Docker sont construits à partir des images Docker. Une image Docker est un package léger, autonome et exécutable d'un logiciel qui inclut tout ce qui est nécessaire pour l'exécuter (code de l'application, environnement d'exécution (*runtime*), outils système et librairies, etc).

Les conteneurs Docker permettent de simplifier la mise en œuvre de systèmes distribués en permettant à de multiples applications et processus de s'exécuter de façon autonome sur une seule machine physique ou à travers de multiples machines isolées. Ceci permet de déployer des nœuds en tant que ressources sur besoin.

La figure 1.2 présente l'architecture de base de Docker (Nguyen et Bein, 2017).

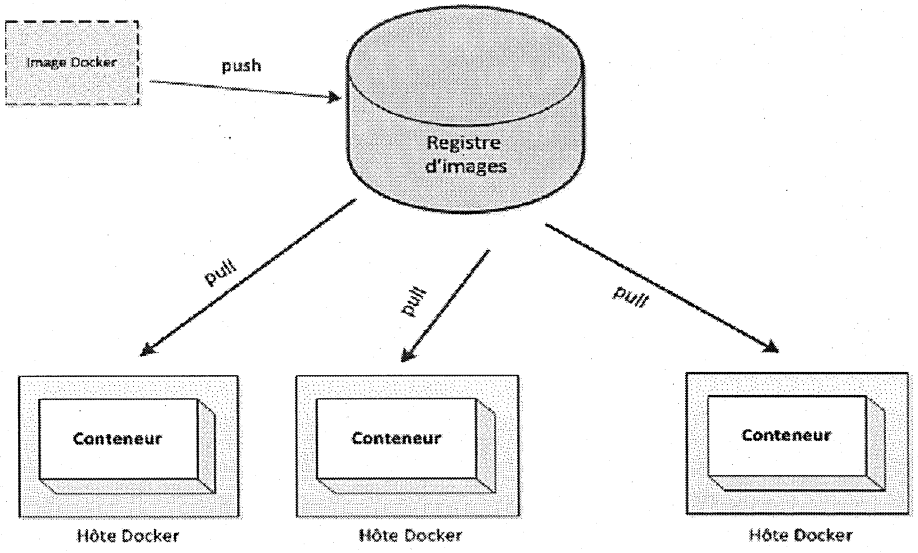

Figure 1.2 Architecture de base de Docker.

1.4.3 Comparaison entre machine virtuelle et conteneur Docker

La virtualisation traditionnelle utilise un hyperviseur pour créer de nouvelles machines, et de les exécuter sur le serveur hôte sous forme de machines virtuelles *VM*. Ces VM intègrent elles-mêmes un système d'exploitation complet sur lequel les applications qu'elles contiennent sont exécutées. Toutefois, les conteneurs Docker s'exécutant sur une machine hôte partagent le noyau du système d'exploitation de cette machine et font directement appel à celui-ci pour exécuter leurs applications.

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'exécutent (Diouf, 2019). La figure 1.3 montre la structure d'une *VM* et celle d'un conteneur Docker.

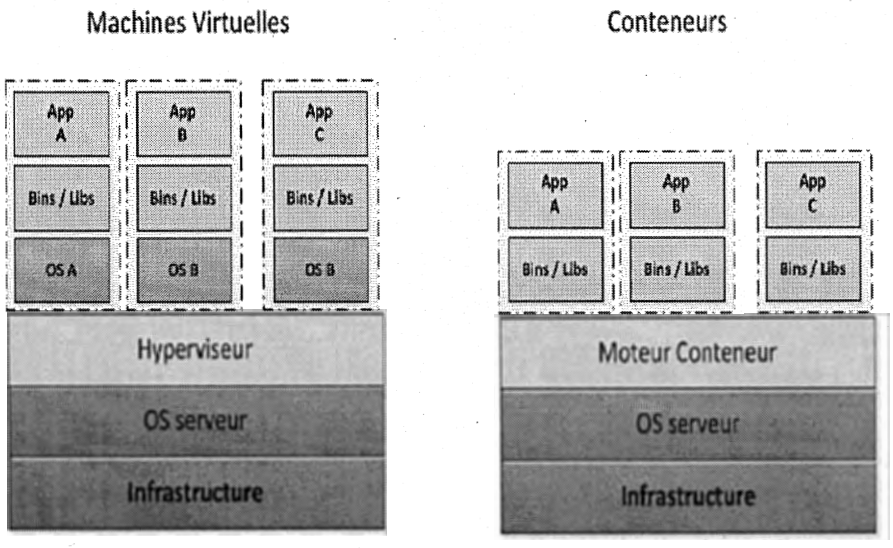

Figure 1.3 Structures d'une machine virtuelle et d'un conteneur Docker.

Le conteneur Docker représente une amélioration des capacités de *LXC*. Ainsi, il n'émule pas l'environnement matériel hôte (Felter *et al.*, 2015a; Diouf, 2019). Il peut être déployé facilement sur une machine virtuelle comme sur une machine physique.

Comme le montre le tableau 1.2 ci-après, un conteneur peut être créé et démarré très rapidement (en quelques secondes) et utilise peu de ressources (processeur, mémoire vive, etc.) (Felter *et al.*, 2015a; Dua *et al.*, 2014; Joy,

2015; Sharma *et al.*, 2016). Du fait que le conteneur n'embarque pas de système d'exploitation et qu'il peut partager des fichiers communs, il est très léger. Cela réduit l'utilisation du disque, ce qui facilite sa migration d'une machine physique à une autre et le téléchargement d'images est beaucoup plus rapide. Les conteneurs présentent des avantages en termes de gestion de mémoire vive, de stockage, de réseau, de vitesse de démarrage, de déploiement et de migration (Joy, 2015; Amaral *et al.*, 2015; Tay *et al.*, 2017). Les conteneurs sont donc plus polyvalents et performants que les VMs.

Tableau 1.2 Caractéristiques des technologies de virtualisation.

Paramètres	**Machine Virtuelle**	**Conteneur Docker**
OS invité	Chaque *VM* utilise son propre *OS*	Les conteneurs utilisent directement l'*OS* hôte.
Utilisation des ressources (CPU, RAM)	Forte utilisation	Utilisation presque native
Temps de démarrage	Quelques minutes	Quelques secondes
Stockage	Quelques Gigaoctets	Quelques Mégaoctets
Communication	Via des commutateurs virtuels	Via les mécanismes *IPC* (*Inter-Process Communication*) et les commutateurs virtuels
Isolation	Isolation complète	Isolation partielle
Sécurité	Dépend de la configuration de l'hyperviseur	Nécessite un contrôle des accès

1.5 Déploiement des conteneurs dans le *fog computing*

Les solutions basées sur hyperviseur imposent un surcoût important en termes d'utilisation des ressources (Ramalho et Neto, 2016). C'est pourquoi les conteneurs ont été reconnus comme une option de virtualisation plus légère. De nombreuses entreprises informatiques ont commencé à utiliser les conteneurs pour déployer et exécuter leurs applications. L'utilisation de ces outils dans le *fog* est possible (Bellavista et Zanni, 2017) et confère davantage à cet environnement dans lequel le temps de réponse à une requête est très important et parfois critique.

1.5.1 Vue d'ensemble

La conteneurisation et le *fog computing* sont encore des paradigmes récents, qui visent à améliorer les performances du système informatique en termes d'utilisation des ressources et de temps de réponse aux requêtes des utilisateurs (Skarlat *et al.*, 2016).

Selon l'étude (Bellavista et Zanni, 2017), le déploiement des conteneurs Docker sur les noeuds de *fog* est devenu possible, et apportent de grands avantages aux divers domaines d'application IoT. Les conteneurs sont déployés sur les noeuds de *fog* via le moteur de Docker (*Docker engine*). Ils peuvent communiquer entre eux à travers le réseau virtuel qui est implémenté sur le noeud. Un exemple de déploiement de conteneurs sur les noeuds de *fog* est présenté dans la figure 1.4.

1.5.2 Défis et problématiques

Il n'est pas trivial de concevoir des plateformes informatiques pour le *fog* répondant aux exigences des applications et aux ressources offertes par le fournisseur du *fog computing*. Plusieurs défis sont identifiés selon (Yi *et al.*, 2015).

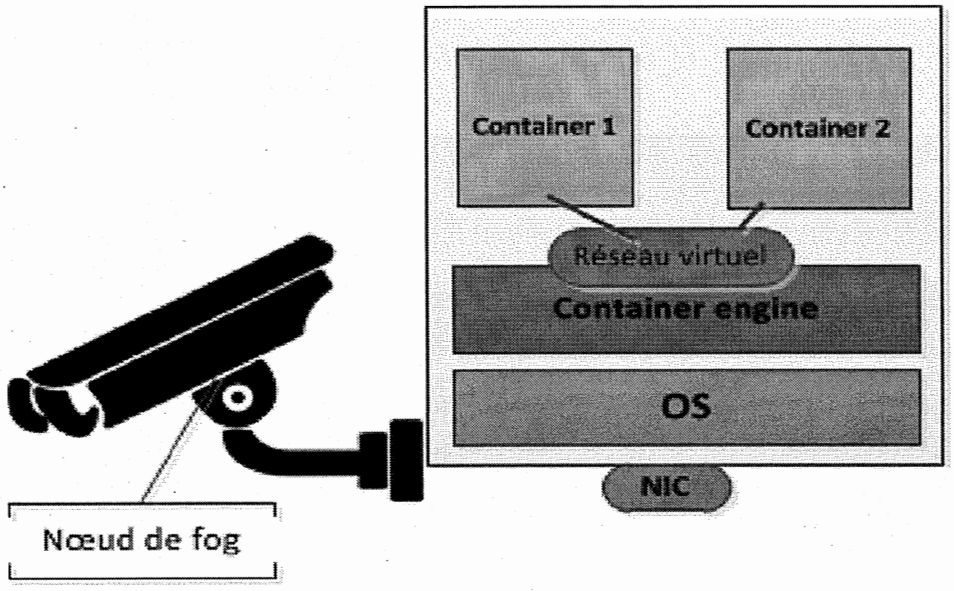

Figure 1.4 Déploiement de conteneurs sur un noeud de fog

- Gestion de réseau : la gestion de réseau constitue un fardeau pour le *fog*, à moins de tirer parti des avantages de l'application des techniques SDN et NFV (Ordonez-Lucena *et al.*, 2017). Cependant, l'intégration transparente de SDN et de NFV dans le *fog* n'est pas facile et constituera certainement un défi majeur. Une intégration naïve peut ne pas atteindre les objectifs de conception en termes de latence et d'efficacité.
- Sécurité et confidentialité : la sécurité et la confidentialité doivent être prises en compte à chaque étape de la conception d'une plateforme informatique pour le *fog*.
- Qualité de service : de nombreux facteurs entraînent une latence élevée des performances des applications sur les plateformes du *fog computing*. Une latence élevée ruinera la qualité de l'expérience et la satisfaction des utilisateurs, car le *fog computing* cible principalement les applications et les services sensibles aux délais.

Dans ce mémoire on s'intéresse à relever ce défi. Pour remédier à ce problème nous étudions deux pistes :

(a) Protocoles de communication entre les conteneurs : les protocoles de communication entre les conteneurs utilisent différents mécanismes de communication, qui ont un impact significatif sur l'utilisation des ressources en termes d'utilisation du CPU, de la mémoire et de la bande passante. La conception et l'adaptation de ces mécanismes de communication permettra de mieux utiliser les ressources de traitement (CPU, mémoire et bande passante) des noeuds, et ainsi de diminuer le délai de communication entre les conteneurs. Notre contribution dans ce sens est présenté dans le chapitre 2.

(b) Allocation de ressources : une bonne planification et placement des conteneurs dans les endroits les plus optimaux dans le réseau de *fog*, permettra de mieux utiliser les ressources et de répondre aux exigences des requêtes des utilisateurs tout en minimisant la latence. Notre contribution dans ce sens est présenté dans le chapitre 3.

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* et le *fog computing*. L'étude comparative entre l'utilisation des machines virtuelles et les conteneurs Docker montre que ces derniers ont des avantages importants par rapport aux VMs en raison de l'amélioration de plusieurs métriques de performance (CPU, RAM, stockage, temps de démarrage, etc.). En général, la conteneurisation par Docker permet de créer et de déployer des applications (services) sous forme de conteneurs dans une machine physique ou virtuelle.

Nous avons ensuite présenté les défis majeurs de l'utilisation des conteneurs dans le *fog computing* plus précisément le problème associé à la qualité de services exigée par les applications. Afin de résoudre ces problèmes, nous avons proposé deux contributions majeures.

Dans le chapitre suivant, nous présenterons la première contribution qui consiste en un mécanisme de communication entre les conteneurs qui améliore les performances du système.

CHAPITRE II

LES PROTOCOLES DE COMMUNICATION ENTRE LES CONTENEURS DANS LE CLOUD ET LE FOG COMPUTING

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 (*Cloud* ou *Fog computing*).

L'utilisation de conteneurs dans le *fog* est possible et confère davantage à cet environnement dans lequel le temps de réponse aux requêtes des utilisateurs et l'utilisation de ressources sont très importants. Cependant, un des plus gros défis des conteneurs Docker est de gérer les communications entre les conteneurs dans un système distribué tel que celui du *Fog computing.*

L'objectif de ce chapitre est d'expliquer les exigences des environnements de conteneurs, les modes de communication entre les conteneurs et enfin de proposer un mécanisme de communication entre les connecteurs qui améliore les performances du système en termes d'utilisation du CPU, du débit et de la latence.

2.2 Motivation et énoncé du problème

2.2.1 Motivation

Les conteneurs ont apporté une incroyable souplesse aux développeurs, permettant un déploiement efficace des applications. Pour des raisons de haute disponibilité, les applications peuvent être décomposées en micro-services déployés sur des clusters de conteneurs (Singh et Peddoju, 2017). Il est donc évident que les conteneurs doivent ensuite être en mesure de communiquer de manière efficace leur état et leurs données pour assurer une exécution adéquate de l'application.

2.2.2 Problème et objectif

Il existe actuellement de nombreuses solutions logicielles fournissant une structure de réseau virtuel pour les conteneurs, notamment Weave (Weave, 2017), Flannel (Flannel, 2015), Calico (Calico, 2019), etc. De manière générale, dans ces approches, les deux hôtes hébergeant les conteneurs se rejoignent via un tunnel de réseau de superposition VXLAN (appelé aussi réseau de recouvrement, en anglais *overlay network*) (Zismer, 2016). Les conteneurs communiquent avec les interfaces réseau virtuelles de leurs hôtes à l'aide d'un pont de conteneur logiciel (*software bridge*). Mais, cela induit une surcharge supplémentaire due aux processus d'encapsulation et de décapsuplation induit par le logiciel de superposition (*software bridge*).

La surcharge du processeur est identifiée comme le principal problème de performance dans les réseaux de superposition (*overlay networks*). Pour confirmer cela, nous avons mené une expérience pour révéler la relation entre la surcharge du lien de communication et l'utilisation des ressources du processeur.

Dans cette expérience, nous avons déployé deux conteneurs, chacun sur une ma-

chine, caractérisée par un processeur Intel Xeon E5 avec 12 cœurs cadencés à 2.4 GHz, et 16 Go de RAM. Le moteur Docker 18.05.0-ce est installé sur les deux machines. Un conteneur est exécuté en mode client et l'autre en mode serveur, les deux conteneurs sont connectés en mode superposition (*overlay mode*). Le conteneur client envoie des messages de taille 1024 octets chacun au conteneur serveur à l'aide de l'outil de mesure du trafic Qperf (George, 2018). Ensuite, nous avons fait varier la capacité du lien de communication de 200 Mo à 60 Go, et nous avons mesuré l'utilisation du CPU et le débit engendré. Nous avons effectué la même expérience pour les deux mécanismes de communication, ethernet standard (sans zéro-copie) et ethernet avec zéro-copie (Choi *et al.*, 2017).

La figure 2.1 illustre nos résultats d'évaluation du débit et de l'utilisation du CPU de deux conteneurs connectés en mode superposition (*Overlay mode*) de Docker, en fonction de la bande passante du lien. Nous observons que quand l'utilisation du processeur est très élevée, aux environs de 90%, le débit reste presque constant à 15 Go/s. De plus, l'augmentation de la capacité du lien (bande passante) n'a pas d'effet sur le débit en raison de l'utilisation très élevée du CPU.

Par contre, l'utilisation du processeur diminue de manière significative avec le mécanisme zéro-copie (Choi *et al.*, 2017), le mécanisme zéro-copie permet de transmettre les données entre les tampons d'interface réseau et la mémoire d'application sans impliquer le noyau ou le système d'exploitation de la machine hôte, ce qui réduit considérablement le nombre de commutations de mode entre l'espace utilisateur et l'espace du noyau (Bröse, 2008). En conséquence, l'utilisation du CPU et le débit du réseau sont considérablement améliorés.

En résumé, le mécanisme zéro-copie diminue l'utilisation du CPU, et cela permet une augmentation considérable du débit de communication entre les conteneurs.

Dans ce travail, nous avons choisi RoCE (*RDMA over Converged Etherent*) (Infi-

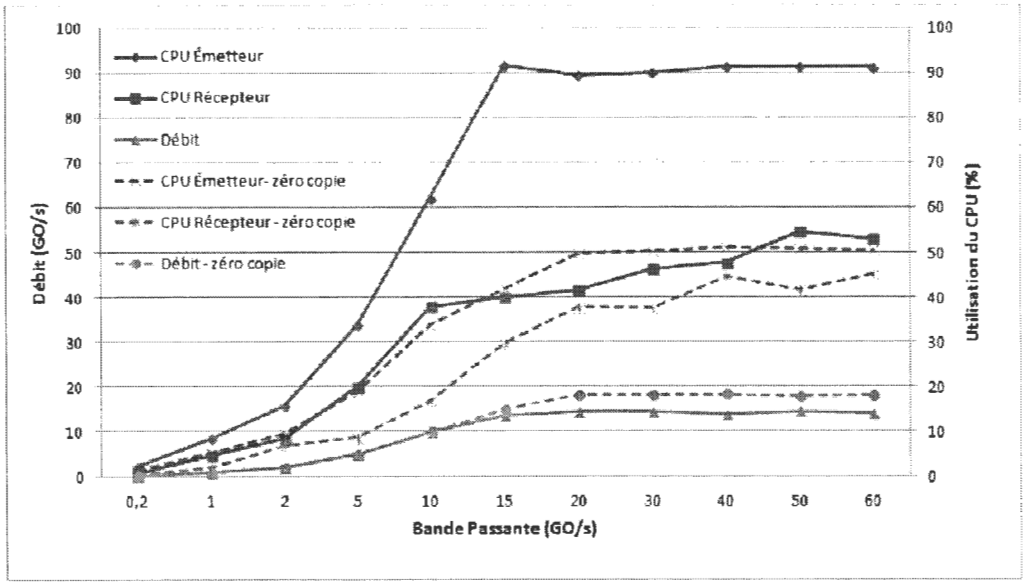

Figure 2.1 L'utilisation du CPU dans le mode superposition de Docker

niband, 2014) pour la communication entre les conteneurs car il combine la fonctionnalité de zéro-copie et l'accès direct à la mémoire à distance (RDMA) dans les réseaux Ethernet, réduisant ainsi l'utilisation du CPU qui est la principale raison derrière le goulot d'étranglement des performances dans la communication entre les conteneurs.

Les mécanismes de communication entre les conteneurs doivent répondre aux trois exigences clés des environnements de virtualisation par conteneur, qu'on discutera dans la section suivante.

2.2.3 Les contraintes

Dans cette partie, nous discutons brièvement les trois contraintes ou exigences clés pour la communication entre les conteneurs.

1. Portabilité :

   La portabilité répond au besoin de distribuer et de déployer des applications indépendamment de la couche infrastructure. C'est un concept qui permet aux applications d'être adaptées facilement en vue de fonctionner dans différents environnements d'exécution. Les différences peuvent porter sur l'environnement matériel (processeur) comme sur l'environnement logiciel (système d'exploitation).

   Avec le développement très rapide des plateformes de déploiement des applications, les développeurs doivent donc faire face à un monde extrêmement dynamique.

   Pour cette raison, et comme les micro-services sont généralement répartis sur plusieurs centres de données avec des environnements d'infrastructure variés, il est essentiel pour les développeurs de pouvoir déplacer des charges de travail et exécuter des applications avec un minimum d'ajustements tout en répondant aux exigences de performance de l'application.

   L'un des avantages principaux de l'utilisation des conteneurs est que les applications conteneurisées sont encapsulées avec leurs fichiers, librairies et dépendances nécessaires de manière à masquer les disparités d'infrastructure, leur permettant ainsi d'être relocalisées et exécutées sur un autre hôte exécutant un moteur de conteneur (*container engine*).

2. Isolation :

   Le concept de base de la conteneurisation repose sur le principe de partager le même noyau du système d'exploitation hôte par tous les conteneurs. Bien que cela apporte de nombreux avantages en matière de performance de communication entre les conteneurs (Felter *et al.*, 2015b; Li *et al.*, 2017), cela signifie également que tous les conteneurs se disputent pour les mêmes ressources du système, ce qui peut entraîner des situations indésirables. Par conséquent, les

plateformes de conteneur exploitent les fonctionnalités du noyau Linux, à savoir les espaces de noms (*namespace*) et les groupes de contrôle (*cgroups*), afin de fournir la fonctionnalité d'isolation en veillant à ce que chaque conteneur puisse afficher et accéder uniquement aux ressources (réseau, nom d'hôte, mémoire, ..etc.) allouées à son espace de noms (Pahl, 2015).

Nous notons toutefois que certains paramètres relatifs aux systèmes multi-entités (ou multi-locataires) (AlJahdali *et al.*, 2014) sont liés à des impératifs de sécurité et d'isolation strictes, en raison de considérations de performance, et du fait que les conteneurs s'exécutent sur un système d'exploitation hôte et un pont réseau identiques, ce qui constitue potentiellement une menace de sécurité plus grande comparé à une approche de virtualisation traditionnelle (Combe *et al.*, 2016). En tant que telles, les solutions typiques préconisées lorsqu'une isolation forte est requise consistent à embarquer des conteneurs dans une machine virtuelle ou à utiliser des hyperconteneurs (Hypercontainer, 2017) au détriment d'une surcharge du lien de communication.

Une nouvelle solution de virtualisation est proposée pour remédier au problème d'isolation dans les conteneurs, nommé KataContainer, qui consiste à combiner la rapidité des conteneurs et la sécurité des machines virtuelles (katacontainers, 2019).

3. Performance :

   Les machines virtuelles (VM) sont susceptibles de consommer plus de ressources que les conteneurs à cause de la rigidité de leur allocation, car des ressources telles que le processeur, la mémoire et le stockage doivent être réservés exclusivement avant leur exécution. Cette stratégie entraîne un gaspillage de ressources lorsque les machines virtuelles ne les utilisent pas complètement, car la charge de travail qui leur est associée fluctue dans le temps.

   Par contre, les conteneurs ne nécessitent pas de réserver les ressources à l'avance,

ce qui permet une meilleure utilisation des ressources. De plus, comme les conteneurs ne comprennent que les dépendances pertinentes pour l'exécution d'une application, cela entraîne moins de duplication de processus dans le système de la machine hôte (Fernando, 2018).

Néanmoins, les performances du réseau constituent l'aspect le plus important pour les conteneurs, en particulier du fait que les conteneurs sont de plus en plus associés à des micro-services en raison de tâches d'orchestration, de mise en cluster et de réplication (Borgione, 2017). Pourtant, les solutions existantes basées sur le mécanisme de superposition telles que Docker *bridge* introduisent une surcharge pour les charges de travail avec un débit de paquets plus élevé (Yu *et al.*, 2016; Felter *et al.*, 2015b), ce qui nécessite d'explorer des solutions basées sur le mécanisme zéro-copie telles que l'accès direct à la mémoire à distance (*RDMA*), décrit plus en détail dans la section suivante.

Notre objectif est de proposer un mécanisme de communication entre les conteneurs, qui donne une meilleure performance en termes d'utilisation de ressources CPU et de débit.

2.3 Les modes de communication entre les conteneurs

2.3.1 Vue d'ensemble

Comme dans tout système informatique, les machines physiques, virtuelles ou processus doivent communiquer entre eux pour échanger de l'information sur leur état en plus des données.

Deux modèles de réseaux sont utilisés dans la communication entre les conteneurs : le modèle de réseau de conteneur (*Container Network Model - CNM*) et l'interface de réseau de conteneur (*Container Network Interface - CNI*) (Zeng *et al.*, 2017), qui proposent un ensemble de plugins et d'interfaces permettant une gestion

efficace de la communication entre les conteneurs. Puisque nous menons nos expériences avec Docker, nous nous concentrons donc sur la norme *CNM* et détaillons ci-dessous les modes de communication entre les conteneurs, à savoir, mode hôte, mode de superposition et OvS avec DPDK. Bien que chaque mode présente des avantages pour des scénarios spécifiques, il en résulte également une ou plusieurs exigences essentielles de l'application (en termes d'isolation, de portabilité et de performance) (Abbasi *et al.*, 2019).

2.3.2 Mode hôte (*Host mode*)

Dans ce mode, les conteneurs sont confinés dans leur espace de noms (*namespace*) de la machine hôte, ce qui permet d'éviter l'utilisation des logiciels de traduction d'adresses de réseau (*NAT*), et donc d'améliorer les performances de la communication entre les conteneurs en termes d'utilisation du CPU et de délai (Abbasi *et al.*, 2019; Agarwal, 2015). Néanmoins, ce mode de communication ne permet pas de sortir vers un réseau extérieur, ni d'accéder au réseau local par l'intermédiaire de la carte réseau physique de la machine hôte. Ce mode permet uniquement d'établir une connexion entre le conteneur et la machine physique. Cela par l'intermédiaire de l'adaptateur virtuel du conteneur et l'adaptateur virtuel de la machine physique.

Les conteneurs utilisent l'adresse IP de leur hôte local pour communiquer avec les conteneurs qui sont déployées sur un autre hôte, comme montré dans la figure 2.2. Cela a pour conséquence une mauvaise isolation due aux conflits de ports, et une limitation sévère du nombre de conteneurs (donc de services) à exécuter sur un seul hôte.

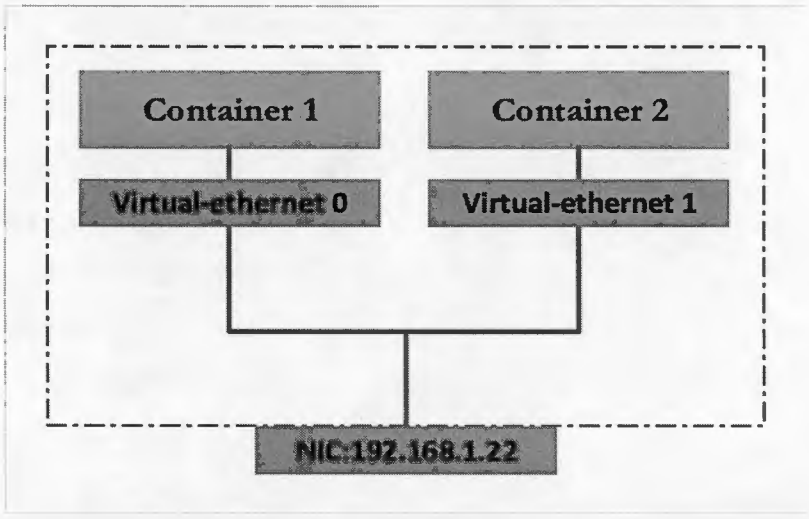

Figure 2.2 La communication entre les conteneurs en mode hôte

2.3.3 Mode de superposition (*Overlay mode*)

Le mode de superposition (en anglais *Overlay mode*) constitue l'approche de premier plan pour les réseaux de conteneurs, utilisé principalement pour des fins de portabilité et d'isolation dans les environnements multi-entités (Borgione, 2017). Ce mode utilise des tunnels de réseau pour permettre la communication entre les hôtes, permettant ainsi aux conteneurs de fonctionner comme dans un réseau intra-hôte.

Une technologie de tunnelisation courante utilisée dans Docker est VXLAN (*Virtual Extensible Local Area Network*) (Mahalingam *et al.*, 2014), qui encapsule les trames dans des datagrammes UDP. Les éléments chargés de cette encapsulation (et de la désencapsulation) sont appelés VTEP (*VXLAN Tunnel End Point*). Un réseau de superposition VXLAN (*VXLAN overlay network*) est composé de deux VTEP, ces VTEP communiquent en UDP, sur un numéro de port dédié.
Un VTEP peut être matériel, lorsqu'il s'agit d'un équipement réseau physique disposant de cette fonctionnalité, ou logiciel lorsqu'il s'agit d'une implémentation

déployée sur un serveur (classiquement un hyperviseur ou un serveur hébergeant des conteneurs) (Petit, 2018).

En résumé, il existe trois interfaces principales pour un hôte en mode superposition, comme illustré dans la figure 2.3 : le pont virtuel et l'interface veth et vtep. Nous notons cependant que le débit et la latence du mode superposition sont nettement inférieurs à ceux obtenus lorsque la communication est basée sur le mode hôte et la mémoire partagée. L'inefficacité de cette dernière est due au fait que les paquets doivent parcourir toute la pile TCP/IP deux fois, une fois à partir de l'interface réseau local (*NIC Network Interface Card*) et une fois à partir du commutateur virtuel (*veth*). Il convient également de noter que Docker prend en charge quelques autres technologies de tunnelisation détaillées ci-après.

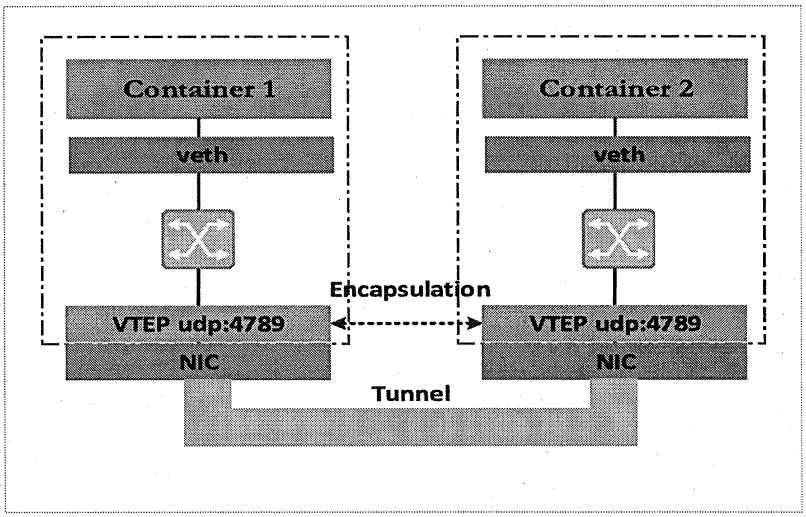

Figure 2.3 La communication entre les conteneurs en mode superposition

— **Flannel :**

Flannel est une solution réseau orientée conteneurs, ouverte, issue de CoreOS (Coreos, 2019). Flannel implémente une encapsulation de couches logicielles via un tunnel afin d'encapsuler le paquet IP dans un paquet UDP (Flannel,

2015). Elle permet de mettre en place une fabrique de couches réseaux (un maillage de routeurs, censés permettre automatiquement à l'infrastructure de s'étendre) pour donner la possibilité à des conteneurs répartis sur plusieurs machines de communiquer, spécifiquement s'ils sont orchestrés par Kubernetes (Google, 2018).

Flannel propose plusieurs *backends*. Un *backend* est une implémentation choisie pour le plan de données (*data plane*). Le *backend* principal recommandé est VXLAN (YANG Yong et Zhenjiang, 2017) , et c'est celui que nous utiliserons dans notre évaluation.

— **Weave :**

Weave (Weave, 2017) est une solution réseau orientée conteneurs, elle implémente également un réseau virtuel utilisant un tunnel UDP. Sa principale différence par rapport à Flannel est que plusieurs paquets de conteneur sont fusionnés en un seul paquet, puis transférés via un canal UDP (Stowe, 2017). Le réseau de superposition Weave comprend plusieurs routeurs logiciels qui établissent une connexion TCP afin d'échanger des informations sur la topologie du réseau. En outre, les routeurs établissent également des connexions UDP pour transférer les paquets encapsulés. Weave crée son propre pont réseau sur chaque hôte pour connecter des conteneurs au réseau de superposition.

La communication entre les différents conteneurs locaux est réalisée via le routeur logiciel du système hôte. Tandis que les paquets destinés à un conteneur externe sont capturés par le routeur Weave et encapsulés dans un paquet UDP qui est ensuite envoyé au routeur Weave de l'hôte destinataire.

2.3.4 *Open vSwitch avec DPDK*

Open vSwitch avec DPDK (*Data Plane Development Kit*) est un autre mécanisme de communication, permettant de fournir une mise en réseau de haute performance entre conteneurs (Robin, 2016).

DPDK est un ensemble de bibliothèques d'espace utilisateur qui permet à un utilisateur de créer des applications de traitement de paquets optimisées, et offre une série de pilotes en mode interrogation (PMD *Poll Mode Driver*), qui permettent le transfert direct de paquets entre l'espace utilisateur et l'interface physique de la machine locale, cela en contournant la pile réseau du noyau. En éliminant à la fois la gestion des interruptions et la traversée de la pile réseau du noyau, des avantages considérables en termes de performances sont obtenus (c'est-à-dire, haute débit et faible latence) (Robin, 2016).

Une architecture de haut niveau OvS-DPDK est illustrée dans la figure 2.4.

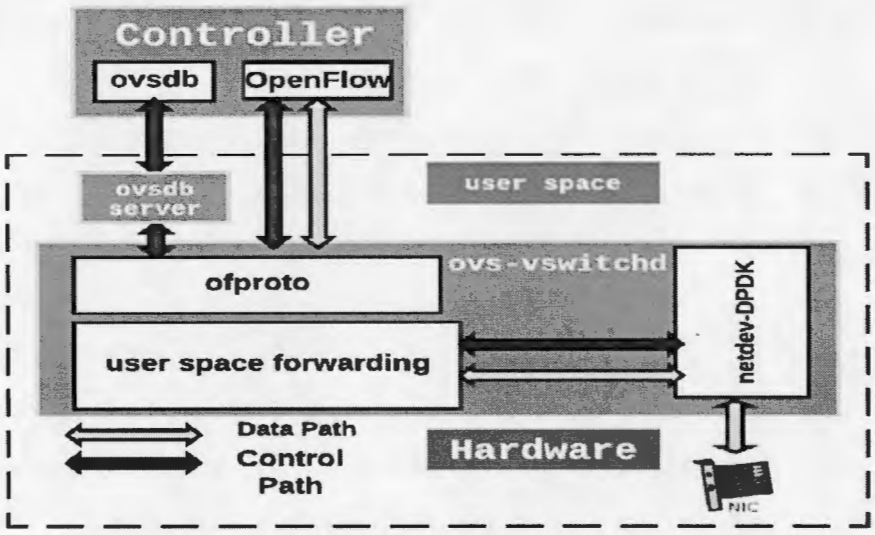

Figure 2.4 Architecture de OvS avec DPDK (Abbasi *et al.*, 2019)

2.4 La solution proposée : RDMA

2.4.1 Définition

L'accès direct à la mémoire à distance (RDMA *Remote Direct Memory Access*) est un mécanisme de communication inter-processus (*IPC*) qui permet à un processus d'accéder directement à la mémoire de l'ordinateur sans impliquer le système d'exploitation. Cela permet une mise en réseau à haut débit et à faible latence (Shpiner *et al.*, 2017; Larsen et Lee, 2014).

2.4.2 Vue d'ensemble

Les solutions de mise en réseau de conteneurs existants basés sur la superposition sacrifient les performances pour une bonne portabilité, car le trafic transite par une pile logicielle profonde, comme expliqué dans la section précédente. La solution clé pour atteindre des performances élevées est d'éviter dans le plan de données toute limitation de performances, telles que les ponts, les routeurs logiciels et le noyau du système d'exploitation hôte, comme montré dans notre évaluation de performance de communication entre conteneurs, dans la section précédente (figure 2.1).

Les canaux de communication fournis par les mécanismes *IPC* basés sur l'hôte et les transports réseau déchargés en matériel (c'est-à-dire décharger les tâches de transport réseau sur une carte d'interface réseau), tels que RDMA, offrent de nombreuses options pour la création de meilleures solutions réseau inter-conteneurs (Yu *et al.*, 2016). Il est à noter que RDMA sacrifie légèrement l'isolation du conteneur. Un tel compromis en échange d'une augmentation considérable des performances réseau est considéré acceptable lorsque les conteneurs en communication appartiennent à la même application du même utilisateur (Yu *et al.*, 2016).

Les implémentations RDMA courantes incluent l'architecture d'interface virtuelle (VIA) (Dunning *et al.*, 1998), RDMA sur Ethernet convergé (RoCE) (Infiniband, 2010), InfiniBand (Mellanox, 2014), Omni-Path (Birrittella *et al.*, 2015) et iWARP (Borkar *et al.*, 1988). Dans ce travail, nous avons utilisé la technologie RoCE.

2.4.3 Architecture et fonctionnement

RDMA sur Ethernet convergé (RoCE) (Infiniband, 2014) est une norme de l'association professionnelle InfiniBand (*InfiniBand Trade Association*) conçue pour fournir des services de transport InfiniBand sur des réseaux Ethernet. RoCE utilise l'interface de programmation (API) appelée *verbs InfiniBand* (Mellanox, 2014), ainsi que ses protocoles de transport et de réseau, et remplace le lien InfiniBand et les couches physiques par ceux d'Ethernet.

RoCE prend en charge la mise en réseau sans copie en permettant à la carte réseau de transférer des données du fil directement vers la mémoire d'application (*Application buffer*) ou de la mémoire d'application directement vers le fil, éliminant ainsi le besoin de copier des données entre la mémoire d'application et les tampons de données du système d'exploitation (*OS buffer*), comme montré dans la figure 2.5. De tels transferts ne nécessitent aucune intervention de système d'exploitation, caches ou commutateurs de contexte, et les transferts se poursuivent en parallèle avec les autres opérations du système. Cela réduit la latence dans le transfert de données.

Dans ce travail, nous avons utilisé RoCE dans nos expériences : Hard RoCE (*Hardware RocE*) où la carte réseau matérielle prend en charge RDMA, et Soft RoCE (*Software RoCE*) où RDMA est pris en charge par un pilote logiciel du système d'exploitation.

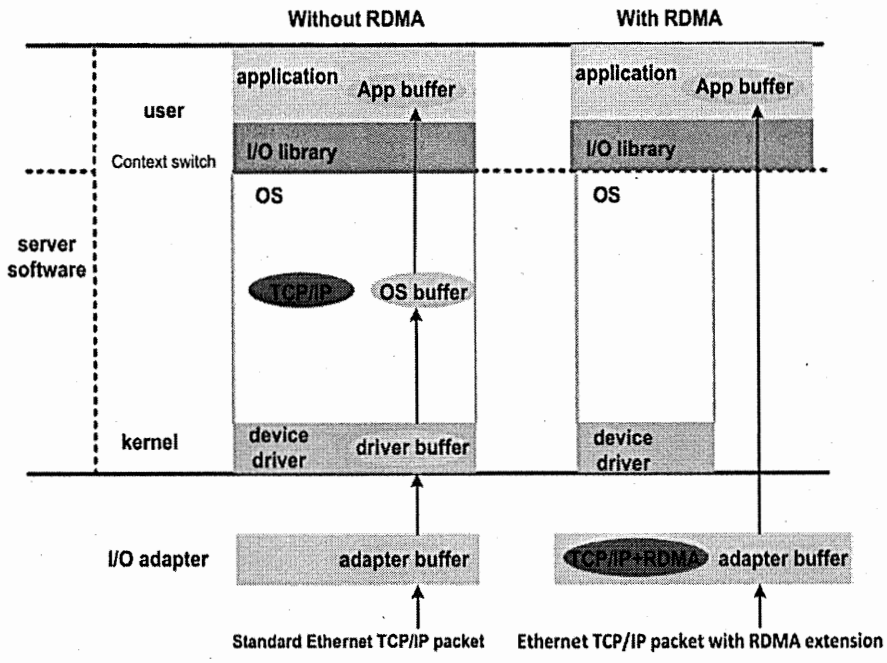

Figure 2.5 Modèle de RDMA sur Ethernet Convergé (RoCE)

2.5 Évaluation des performances de la solution proposée

Nous avons comparé les performances de différentes technologies de communication entre conteneurs en termes d'utilisation du processeur, de débit et de latence. Les performances des technologies basées sur RDMA (Soft-RoCE et Hard-RoCE) sont comparées à celles non basées sur RDMA, à savoir Weave, Flannel et OvS-DPDK.

2.5.1 L'environnement de simulation

Notre environnement de test est composé de deux machines physiques exécutant Ubuntu 16.04, chacune est équipée d'un processeur Intel Xeon E5 avec 12 cœurs cadencés à 2.4 GHz, 16 Go de RAM et de cartes Mellanox ConnectX-3 40 GbE connecté à une voie PCI Express Gen 3x16. Le moteur Docker 18.05.0-ce est ins-

tallé sur les deux machines pour les besoins d'instanciation de conteneurs. Ensuite, nous avons déployé un conteneur Docker sur chacune de ces deux machines.

Dans nos expériences, Qperf v0.4.9 (George, 2018) a été utilisé pour mesurer l'utilisation du CPU, le débit et la latence entre les deux conteneurs.

Étant donné que les mesures sont effectuées entre des conteneurs, l'outil de mesure Qperf a été adapté pour fonctionner dans des conteneurs Docker. Nous avons donc lancé Qperf depuis le conteneur et nous avons effectué des mesures via les interfaces virtuelles (veth) et de superposition associé au conteneur afin de minimiser les interférences. Qperf est exécuté en mode serveur sur la machine réceptrice, tandis que le client Qperf qui s'exécute sur la machine émettrice envoie du trafic TCP au serveur Qperf, avec des tailles de message variant de 32 octets à 64 Kilooctets. Nous avons répété cette expérience dix fois pour faire des calculs statistiques et calculer les intervalles de confiance. Ensuite, nous avons évalué et tracé la variabilité de chaque mesure, illustrée par les diagrammes en forme de boîte à moustaches dans les figures 2.6, 2.7 et 2.8, montrant les valeurs minimales et maximales, le premier et troisième quartiles, la médiane ainsi que les barres d'erreur.

2.5.2 Les résultats de simulation

Dans cette partie, nous comparons les performances de la technologie Hard-RoCE en mode orienté connexion à Weave, Flannel, Soft-RoCE et OvS-DPDK, en termes d'utilisation du CPU, de débit et de délai. Étant donné que les machines utilisées dans notre expérimentation sont multicœurs, les valeurs d'utilisation du CPU peuvent dépasser 100%. Par exemple, une utilisation de 200% du processeur correspond à une utilisation complète de deux cœurs. En ce qui concerne l'utilisation du CPU, nous avons observé que l'utilisation du CPU diminue lorsque la taille du message augmente, cela est dû au fait que la surcharge de traitement de la pile TCP est amortie à mesure que la taille du message augmente.

Comme on s'y attendait, la solution RDMA (Hard-RoCE) réduit considérablement l'utilisation du CPU par rapport aux solutions en mode superposition telles que Weave et Flannel, comme le montre la figure 2.6. RDMA contourne le noyau du système d'exploitation en permettant à une machine de lire et d'écrire directement sur la mémoire d'une autre machine sans impliquer le système d'exploitation de l'une ou l'autre des machines, ce qui réduit le temps de travail du processeur. En outre, on peut voir sur la figure 2.6, que la solution RDMA est la plus stable en termes de variabilité d'utilisation du CPU. Ceci est dû au fait que le CPU est peu impliqué dans le processus de communication. Par contre, les solutions logicielles (Weave, Flannel) sont plus sujettes à la variance. L'utilisation élevée du CPU par Weave et Flannel est due au processus de traitement des paquets par les commutateurs virtuels associés aux conteneurs et le système d'exploitation hôte.

En ce qui concerne les résultats de OvS-DPDK, notez que nous avons configuré DPDK en mode pilote d'interrogation PMD (*Poll Mode Driver*) pour éliminer la surcharge liée à la commutation de contexte, qui est provoquée par des interruptions dans le noyau Linux.

La figure 2.7 présente le délai en échelle logarithmique en fonction de la taille du message. Les solutions Flannel et Weave présentent une latence élevée, ce qui est attendu vu que la transmission de données dans ces solutions reposant sur le mécanisme de superposition, où les données traversent à la fois le commutateur logiciel et la pile TCP/IP, ce qui induit plus de latence. Nous constatons que Weave présente une performance légèrement supérieure à celle de Flannel, en particulier avec des tailles de messages inférieures à 1 Ko. Weave étant configuré pour tirer parti du chemin de données rapide, il tire profit du sous-système ODP (*Open vSwitch datapath*) du noyau, ce qui permet au routeur Weave de transférer les instructions de traitement de paquets directement au noyau afin de réduire le temps de latence. OvS-DPDK surpasse les commutateurs virtuels natifs (Weave et Flannel)

de trois ordres de grandeur. Cela est dû au fait que le plan de données est exécuté dans l'espace utilisateur et que les copies de mémoire entre l'espace utilisateur et l'espace noyau, ainsi que les commutateurs de contexte associés sont explicitement contournés. Enfin, comparés aux autres solutions, les solutions RoCE (soft et hard) montrent une latence plus faible car ils exploitent le mécanisme de zéro copie, selon lequel les données sont directement transférées entre les conteneurs sans impliquer la pile de système d'exploitation.

La figure 2.8 compare le débit en fonction de la taille du message. Étant donné que dans Hard-RoCE, le processeur hôte n'est pas très impliqué dans la transmission des paquets, le débit maximum atteint presque la capacité maximale de la carte réseau. L'implémentation soft-RoCE utilise un pilote dans le système d'exploitation, ce qui entraîne un débit relativement faible à cause de l'implication du noyau. D'autre part, les solutions de réseau basées sur la superposition (Weave et Flannel) introduisent un surcoût de débit, cela est dû au processus d'encapsulation et désencapsulation des paquets. Le processus de fragmentation des données et d'encapsulation des fragments est très gourmand en utilisation des ressources de traitement (CPU, mémoire), et cela réduit considérablement le débit.

2.6 Conclusion

Dans ce chapitre, nous avons évalué le gain de performance lié à l'utilisation de RoCE (*RDMA over Converged Ethernet*) pour la communication entre conteneurs. Nous avons démontré que RoCE surpasse les solutions de réseau de conteneurs existantes, en particulier Flannel, Weave et OvS-DPDK, en termes de surchage du processeur, de débit et de latence. En outre, RoCE réalise d'excellentes performances et une excellente portabilité, en sacrifiant peu d'isolation.

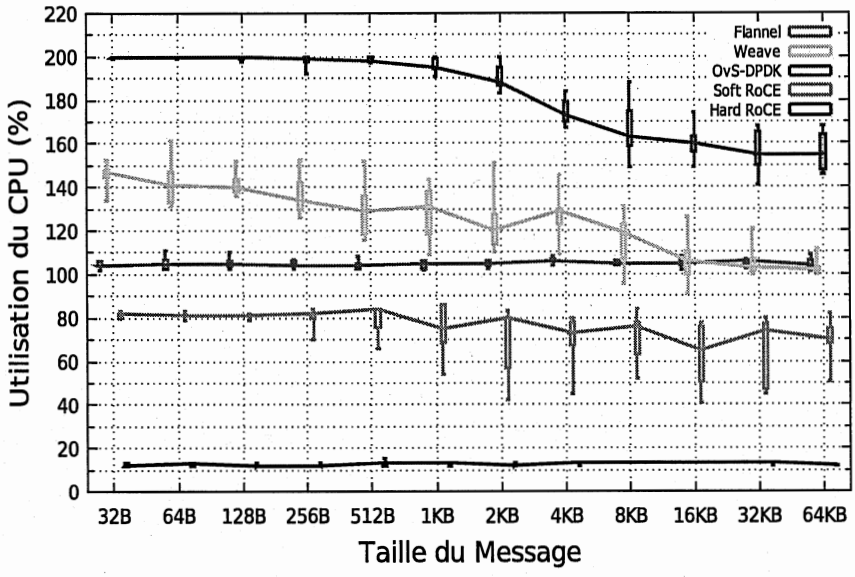

Figure 2.6 Variation de l'utilisation du CPU en fonction de la taille du message.

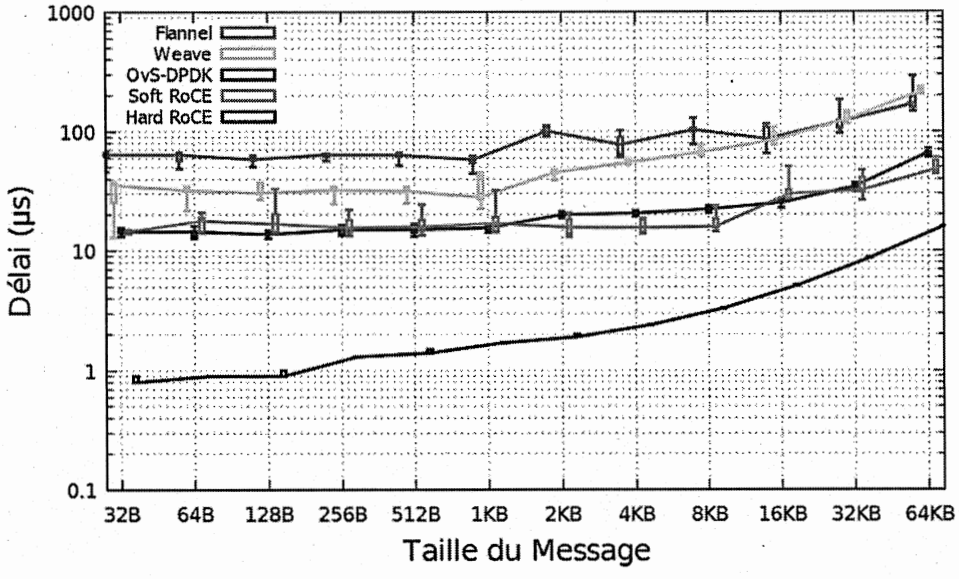

Figure 2.7 Variation du délai de communication entre les conteneurs en fonction de la taille du message

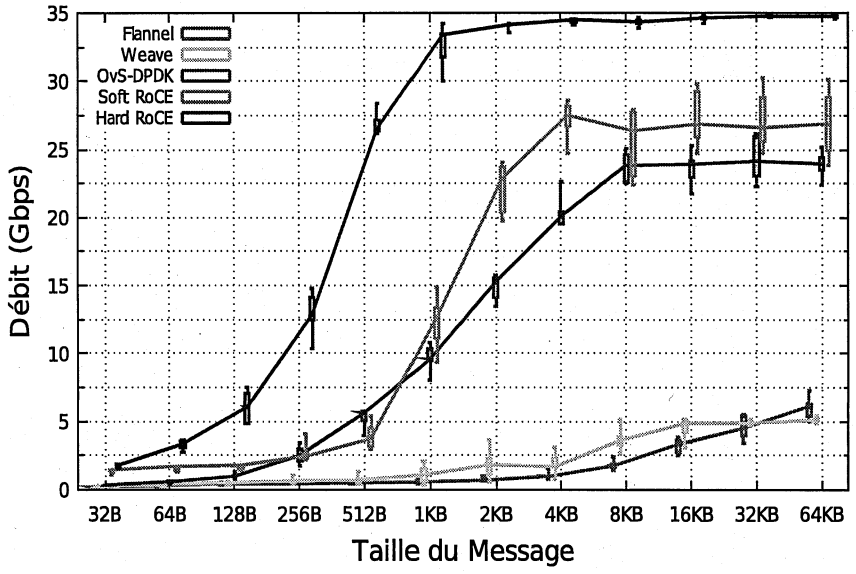

Figure 2.8 Variation du débit de communication entre les conteneurs en fonction de la taille du message.

Dans le chapitre suivant, nous étudions les stratégies de placement de conteneurs sur les noeuds du réseau de fog, en tenant compte des technologies de communication entre les conteneurs, et ensuite nous montrons leurs impacts sur les performances du système de fog computing.

CHAPITRE III

LE PLACEMENT DE CONTENEURS DANS LE *FOG COMPUTING*

Dans ce chapitre, nous étudions le problème de placement de conteneurs dans le fog computing. Nous commençons par décrire la motivation derrière l'étude du problème de placement de conteneurs. Ensuite, nous modélisons à l'aide de la programmation linéaire en nombres entiers ILP (*Integer Linear Programming*) le problème de placement de conteneurs dans le *fog computing*. Par la suite, nous présentons notre heuristique CPGA. Et nous finissons par évaluer les performances de notre heuristique CPGA par rapport aux performances de la technique ILP et l'algorithme glouton.

3.1 Motivations

Fog computing est une plateforme informatique distribuée émergente qui vise à rapprocher le calcul de ses sources de données, ce qui peut réduire le temps de latence et le coût de transmission des données vers un *cloud* distant (Mouradian *et al.*, 2017). Cette infrastructure est composée d'un grand nombre de périphériques distribués et hétérogènes qui communiquent et coopèrent pour exécuter des services.

Un tel environnement est principalement dédié aux applications sensibles au délai. Par conséquent, les conteneurs sont plus adaptés que les machines virtuelles

pour déployer des applications et réduisent le temps de déploiement et d'exécution des applications (Barik *et al.*, 2016). Une application est composée de plusieurs services, ces services imposent des contraintes importantes à l'opérateur de réseau du *fog* qui doit s'assurer que les données sont effectivement acheminées vers les services appropriés, dans le bon ordre (Livi *et al.*, 2016). Par conséquent, notre travail se concentre sur le chaînage des services (*service chaining*) conteneurisées, dans le *fog*, en prenant en compte les technologies de communication inter-conteneurs.

Le temps de réponse aux requêtes des utilisateurs est affecté par la distribution des nœuds de *fog* et par les technologies de communication inter-conteneurs mises en œuvre dans les nœuds. Cependant, la stratégie consistant à placer le conteneur dans l'environnement de *fog* reste une opération critique, qui est effectuée pour déterminer le noeud le plus approprié pour héberger le conteneur. La sélection de nœuds appropriés est très importante pour améliorer l'efficacité énergétique, l'utilisation des ressources et la prise en charge de la qualité de service dans un environnement de *fog computing*.

Le placement de conteneurs dans un environnement de *fog* est très important et peut constituer une tâche complexe à cause de deux raisons : premièrement, l'arrivée des demandes d'instanciation de conteneurs peuvent ne pas être prévisibles. Deuxièmement, la taille du réseau est souvent très grande, et pour une charge donnée, il a été démontré que le scénario optimal ou quasi optimal est NP difficile (Tang *et al.*, 2018; Huang *et al.*, 2009).

Le but de notre travail est de proposer et de mettre en œuvre une stratégie de placement de conteneurs sur la base du chaînage de services pour l'infrastructure informatique de *fog*. Cette stratégie prend en compte les nœuds physiques hétérogènes et largement répartis, ainsi que les différentes technologies de com-

munication inter-conteneurs. Notre stratégie vise à minimiser le temps de réponse moyen du système. Pour ce faire, nous modélisons notre problème en utilisant différentes approches d'optimisation combinatoire.

Les principales contributions de ce chapitre sont :

— Un algorithme de placement de conteneurs qui permet d'obtenir de bons résultats de mappage entre les conteneurs et les noeuds du réseau de fog, sous des contraintes des ressources.

— Une stratégie de placement de conteneurs prenant en compte les protocoles de communication entre les conteneurs et le chaînage de services dans un environnement réaliste de *fog computing.*

— Une comparaison détaillée entre tous les algorithmes proposés pour déterminer les compromis quantitatifs en termes de qualité du plan de placement et d'utilisation des ressources.

3.2 Modèle du système

Nous présentons dans cette section notre modèle du système, basé sur l'architecture hiérarchique du *fog* à trois couches, décrite dans la figure 1.1.

Notre processus de placement de conteneurs est illustré dans la figure 3.1, dans lequel notre processus est centré autour d'un contrôleur de service de fog (*Fog Service Controller*), qui se situe entre la couche de *fog* et la couche de *cloud.* Son rôle principal est de contrôler et de gérer les ressources de *fog* dans une zone géographique déterminée (domaine de fog). Nous considérons un scénario dans lequel une application est composée d'un ensemble de microservices conteneurisés (c'est-à-dire un service déployé dans un conteneur) et enchaînés, à déployer dans les noeuds de la couche *fog.*

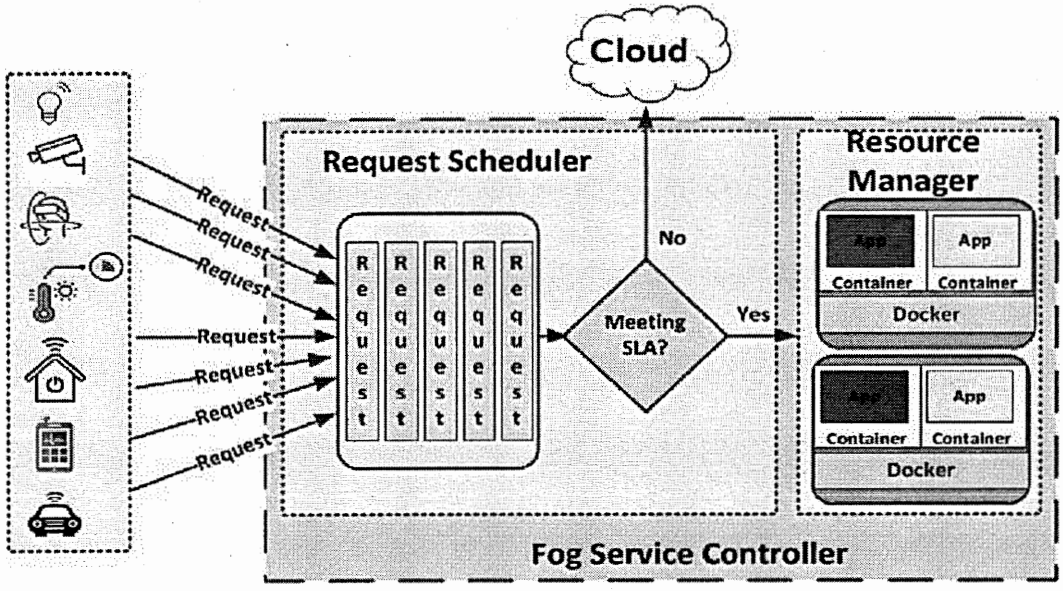

Figure 3.1 Modèle du système

Le contrôleur de service est constitué de deux composants : le planificateur de requêtes (*Request Scheduler*) et le gestionnaire de ressources (*Resource Manager*).

Le planificateur de requêtes sert à diriger les requêtes des utilisateurs vers la couche *cloud* ou vers la couche *fog*, selon qu'il peut ou non satisfaire les exigences de qualité de service de la requête demandée par l'utilisateur. Dans le cas où les exigences de qualité de service de la requête peuvent être satisfaites dans la couche de *fog*, le gestionnaire de ressources détermine les nœuds de *fog* dans lesquels les services seront exécutés et alloue les ressources correspondantes à la requête.

Une requête est considérée comme une application composée de plusieurs conteneurs (Singh et Peddoju, 2017) et chaque conteneur a ses besoins en ressources CPU, RAM et stockage.

Nous notons que chaque conteneur est caractérisé par différentes exigences de CPU, de RAM et de stockage pour un déploiement et une exécution adéquate.

En outre, les nœuds de *fog* pouvant être hétérogènes en termes de protocoles de communication pris en charge en fonction de leurs interfaces réseau physique (Hong *et al.*, 2018). Le chaînage de microservices conteneurisés requiert également diverses techniques de communication inter-conteneurs (par exemple, sur la base d'un accès direct à la mémoire RDMA, mode de superposition, mode hôte). Ces techniques sont pris en compte dans le processus de chaînage de service, du fait de leurs impacts potentiels sur les performances des applications (Yu *et al.*, 2016; Suo *et al.*, 2018).

La technologie basée sur RDMA pourrait être disponible uniquement sur des nœuds de grande capacité (en termes de CPU, RAM, PCIe, etc.) tels que les stations de base, les routeurs et les stations de travail. Contrairement aux solutions basées sur le mode hôte et sur la superposition qui sont disponibles sur des nœuds de faible capacité, tels que les points d'accès, les caméras de surveillance et les PCs .

Un résumé des types de nœud de *fog* par technique de communication inter-conteneurs est proposé dans le tableau 3.1.

Tableau 3.1 Les noeuds de fog et les techniques de communication entre conteneurs

Noeud de fog	**Technique de communication entre conteneurs**
Point d'accès, caméra de vidéo de surveillance	Mode hôte
Routeur, switch, PC	Mode de superposition (flannel, weave)
Station de base, Serveur	RDMA

Notre objectif est de trouver les emplacements optimaux pour déployer des micro-services conteneurisés dans la couche de *fog*, dans lesquels les nœuds de *fog* sont hétérogènes (en termes de capacité) et des protocoles de communication inter-conteneurs, de manière à garantir les exigences de délai de l'application.

3.3 Formulation du problème

Dans cette section, nous formulons le problème de placement de conteneurs et de chaînage de services dans un environnement de fog computing en tant que problème d'optimisation ILP.

3.3.1 Énoncé du problème

L'environnement de *fog* est constitué de nombreux noeuds qui sont connectés entre eux, et qui sont prêts à déployer des services sous forme de conteneurs. Cependant, les ressources informatiques dans le *fog* sont limitées, et différents types de services auront des demandes différentes en termes de ressources (CPU, RAM et stockage). Dans ce travail, nous proposons une stratégie de placement de conteneurs orientée service dans le *fog*.

3.3.2 Notations

Dans cette partie, nous décrivons les notations utilisées dans le document. U désigne l'ensemble des utilisateurs, qui font référence à un hôte ou à un appareil IoT connecté au dispositif du *fog*.

S est l'ensemble de services pour lesquels le placement de conteneurs dans le *fog* est requis. Et C est l'ensemble des conteneurs à déployer sur les nœuds de *fog*.

Les nœuds de *fog* définis par F désignent tous les nœuds de *fog* constituant le

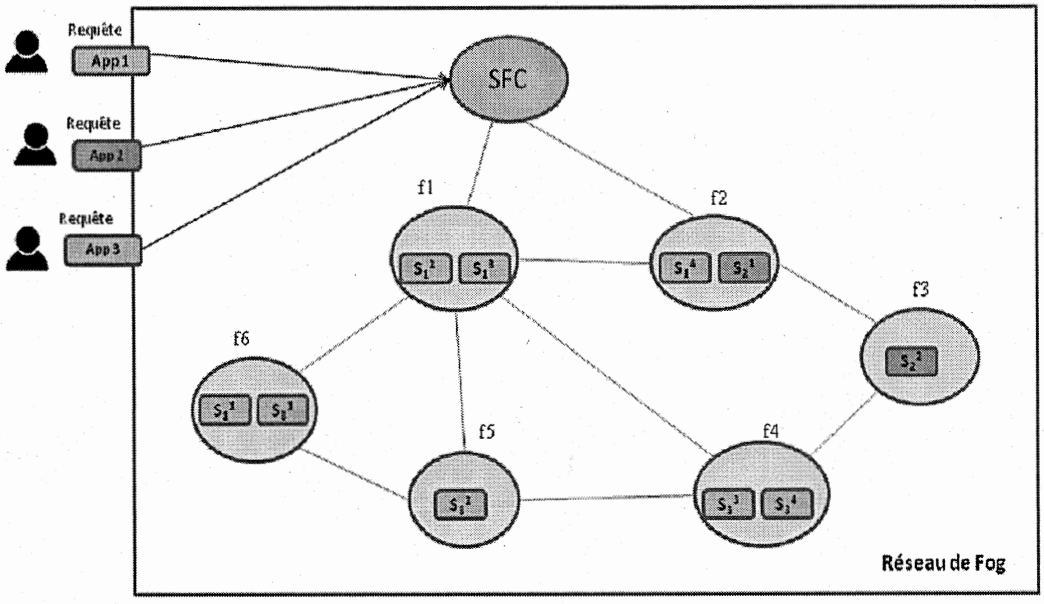

Figure 3.2 Exemple typique d'un placement de conteneurs dans le *fog*

réseau. Un nœud de *fog* est caractérisé par ses capacités CPU P, RAM M et Stockage L.

Notre modèle vise à déterminer un mappage optimal entre les applications IoT et les ressources informatiques du *fog* dans le but d'optimiser le temps de réponse du système, tout en satisfaisant les exigences de la qualité de service (QoS) des applications. Le modèle concret de placement de conteneurs sur les ressources de *fog* est présenté dans la section suivante.

Nous spécifions ci-après les variables et contraintes prises en compte pour notre problème de placement.

Nous considérons par souci du simplicité du langage, qu'un conteneur héberge un seul service. Par conséquent, nous considérons $s_{k,i} \in \mathrm{S}_k$ un service donné i d'une application a_k est déployé dans un conteneur i.

Tableau 3.2 Paramètres d'entrée et variables

Paramètres d'application	
U	Ensemble des utilisateurs
A	Ensemble des applications
S_k	Ensemble des services d'une application a_k
D_{a_k}	*Deadline* d'une application a_k
$D^t_{a_k}$	Temps de déploiement d'une application a_k
M_{a_k}	Temps d'exécution d'une application a_k
R_{a_k}	Temps de réponse d'une application a_k
$C^P_{s_i}$	Demande de CPU d'un service
$C^M_{s_i}$	Demande de RAM d'un service
$C^L_{s_i}$	Demande de stockage d'un service
Paramètres de ressources de fog	
F	Ensembles des noeuds de fog
f_j	Noeud de fog j
$R^P_{f_j}$	Capacité de CPU de f_j
$R^M_{f_j}$	Capacité de RAM de f_j
$R^L_{f_j}$	Capacité de stockage de f_j
d_{f_i,f_j}	Délai de liaison entre les nœuds i et j
$d_{U,f}$	Délai de liaison entre l'utilisateur U et le noeud f

Soit $X^{f_j}_{s_{k,i}} \in \{0,1\}$ la variable de décision qui indique si un service i d'une application a_k est placé sur un nœud de fog f_j.

3.3.3 La fonction objective

Notre fonction objective vise à minimiser le temps de réponse de chaque application de l'ensemble des applications A, tout en garantissant les exigences de

qualité de service en termes de délai, et respectant les contraintes de capacité et de communication des noeuds de fog.

$$\min R_{a_k}, \quad \forall\, a_k \in A \tag{3.1}$$

3.3.4 Les contraintes du problème

Nous spécifions ci-après les contraintes du problème de placement de conteneurs dans le *fog computing* :

La première contrainte, exige que tous les services composant l'application doivent être placés dans le réseau de *fog*.

$$\sum_{f_j}^{F} \sum_{s_{k,i}}^{S_k} X_{s_{k,i}}^{f_j} = |S_k|, \quad \forall\, a_k \in A \tag{3.2}$$

où $|S_k|$ désigne le nombre de services composant l'application a_k.

La deuxième contrainte, les services placés ne doivent pas dépasser les capacités de CPU, de RAM et de stockage de la ressource de fog correspondante.

$$\sum_{s_{k,i}}^{S_k} C_{s_{k,i}}^{\alpha} \cdot X_{s_{k,i}}^{f_j} \leq R_{f_j}^{\alpha}, \quad \alpha = \{C, M, L\}, \forall\, f_j \in F \tag{3.3}$$

La troisième contrainte, le placement des services composant l'application a_k doit être effectué de manière à ce que le temps de réponse qui en résulte ne dépasse pas le deadline de l'application D_{a_k}.

$$R_{a_k} \leq D_{a_k}, \quad \forall\, a_k \in A \tag{3.4}$$

Le temps de réponse de l'application R_{a_k} est calculé par

$$R_{a_k} = D_{a_k}^{t} + M_{a_k} + d_{a_k}, \quad \forall\, a_k \in A \tag{3.5}$$

Où $D^t_{a_k}$ est le temps de déploiement de tous les services qui compose l'application a_k. Le temps d'exécution M_{a_k} est le temps nécessaire pour exécuter tous les services de l'application a_k, et d_{a_k} correspond au temps de communication nécessaire pour transmettre et échanger les données entre l'utilisateur et les différents noeuds déployant les services de l'application a_k, tel que décrit par l'équation 3.6.

Dans le délai d_{a_k} est inclus le délai entre l'utilisateur final U_k et le noeud de *fog* sur lequel est placé le conteneur hébergeant le premier service de la chaîne de services de l'application a_k, le délai entre les conteneurs suivants, le délai entre le dernier conteneur et le premier conteneur et enfin le délai entre le premier conteneur et l'utilisateur final U_k. Un exemple de la chaîne de communication d_{a_k} est montré dans la figure 3.3. Dans notre modèle, nous supposons que seul le nœud hébergeant le premier conteneur connaît les informations de routage nécessaires pour atteindre l'utilisateur final.

Nous considérons en outre dans notre formulation, le délai intra-noeud ainsi que le délai de communication inter-noeud. Notez également que le délai entre noeuds de *fog* comprend le délai induit par les différents mécanismes de communication entre conteneurs.

$$d_{a_k} = 2 \cdot \sum_{j=1}^{F} d(U_k, X^{f_j}_{s_{k,1}}) + \sum_{i=1}^{n-1} \sum_{j=1}^{F} \sum_{j'=1}^{F} d(X^{f_j}_{s_{k,i}}, X^{f_{j'}}_{s_{k,i+1}}) + d(X^{f_j}_{s_{k,n}}, X^{f_{j'}}_{s_{k,1}}), \quad (3.6)$$

$$d(X^{f_j}_{s_{k,i}}, X^{f_{j'}}_{s_{k,i+1}}) = \begin{cases} d_{f_j,f_{j'}} & \text{si } X^{f_j}_{s_{k,i}} = 1 \text{ et } X^{f_{j'}}_{s_{k,i+1}} = 1 \\ d_{f_j,f_j} & \text{si } X^{f_j}_{s_{k,i}} = 1 \text{ et } X^{f_j}_{s_{k,i+1}} = 1 \end{cases}$$

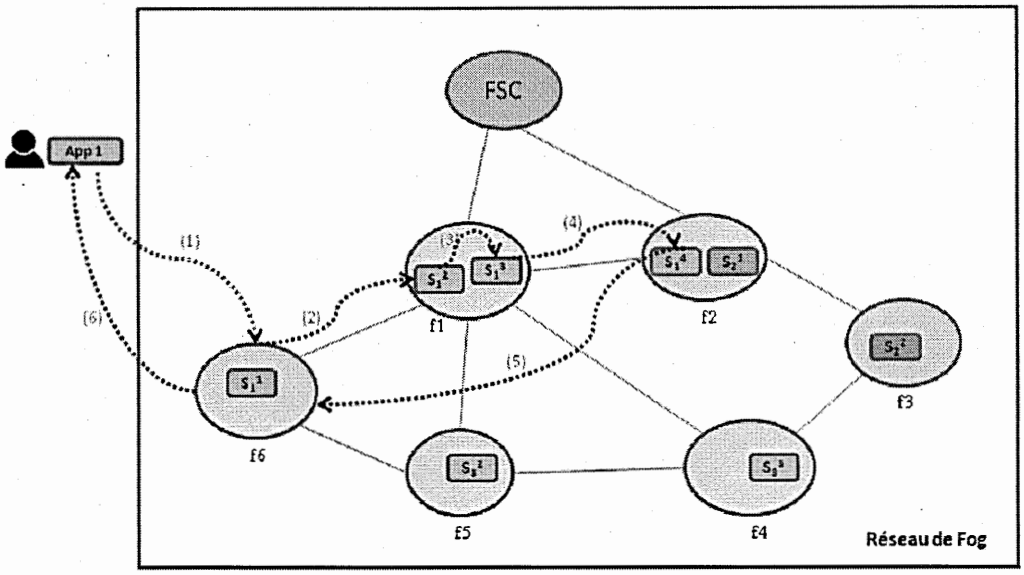

Figure 3.3 Temps de réponse d'une requête

3.4 Stratégies d'optimisation

Dans cette partie, nous présentons les différentes stratégies d'optimisation que nous avons utilisées pour résoudre notre problème de placement de conteneurs dans le *fog computing.*

En optimisation combinatoire, une méta-heuristique est une méthode approchée qui permet de trouver une solution réalisable pour un problème NP-difficile pas forcément optimale en un temps polynomial. En ce qui concerne le problème de placement de conteneurs dans le *fog computing*, trouver le meilleur plan de placement prendrait un temps énorme vu que c'est un problème NP-difficile (Tang *et al.*, 2018; Huang *et al.*, 2009).

Plusieurs algorithmes de métaheuristiques ont été élaborés afin de résoudre le problème de placement des conteneurs. On peut citer les algorithmes génétiques, les algorithmes de colonies de fourmis, l'optimisation par essaims particulaires et la

recherche tabou (Ait Salaht *et al.*, 2019). Dans ce qui suit, nous détaillons la technique de programmation linéaire, l'algorithme glouton et enfin l'algorithme génétique. Ces algorithmes sont utilisés dans ce travail pour la résolution du problème de placement des conteneurs dans un environnement de *fog computing.*

3.4.1 L'optimisation linéaire

L'optimisation linéaire (appelé aussi programmation linéaire) est une technique mathématique d'optimisation (maximisation ou minimisation) de fonction objectif linéaire sous des contraintes ayant la forme d'inéquations linéaires (Nabli, 2012).

Les méthodes de résolution les plus efficaces d'un modèle de programmation linéaire en nombres entiers sont basées généralement sur les algorithmes de recherche arborescente par séparation et évaluation *(Branch and Bound)* et les méthodes des coupes (Mancel, 2004). Dans notre travail, nous avons utilisé la méthode *Branch and Bound*, détaillé ci-après.

L'algorithme de *Branch and Bound* (Roucairol, 1989) consiste à construire noeud par noeud l'arbre d'énumération de l'ensemble des solutions du modèle IP (*Integer Programming*) d'une manière intelligente, tout en faisant usage des bornes inférieure et supérieure afin d'éviter la génération de tous les noeuds de l'arbre. Il s'agit principalement de diviser récursivement le problème initial en sous-problèmes plus petits et plus faciles à résoudre : c'est la phase de séparation *(branching)*. Quand à la phase d'évaluation *(bound)*, elle consiste à identifier les sous-ensembles qui peuvent contenir la solution optimale et à supprimer ceux qui ne la contiennent pas (Grainia, 2015).

L'algorithme de *Branch and Bound* construit et parcourt l'arbre de recherche. La racine de cette dernière est le domaine réalisable du problème initial, le modèle IP, tandis que les nœuds représentent les domaines réalisables des sous-problèmes

(Grainia, 2015).

La programmation linéaire permet de tester tous les choix de solutions possibles. Étant donnée la NP-complitude du problème de placement de conteneurs (Tang *et al.*, 2018; Huang *et al.*, 2009), la résolution du problème que nous avons formulé en nombres entiers est uniquement proposé pour un petit nombre de conteneurs. Cependant, le *fog computing* héberge un nombre élevé de conteneurs. C'est pour cette raison que nous proposons dans les sous sections suivantes une heuristique visant à fournir une solution sous-optimale pour le problème de placement de conteneurs dans le fog.

3.4.2 L'algorithme glouton

Un algorithme glouton (*greedy algorithm*) est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local. Dans certains cas cette approche permet d'arriver à un optimum global, mais dans le cas général c'est une heuristique (Cormen *et al.*, 2009).

Notre stratégie gloutonne pour résoudre le problème de placement de conteneurs consiste à chercher et placer les conteneurs dans les nœuds de fog les plus proches des utilisateurs finaux, en tenant compte des contraintes (3.2), (3.3) et (3.4). Nous effectuons ensuite une recherche exhaustive pour déterminer les nœuds de fog les plus appropriés pour héberger les conteneurs restants, jusqu'à ce que tous les conteneurs soit placés. Le pseudo-code de notre algorithme glouton est décrit dans l'algorithme 1.

Algorithm 1: Algorithme glouton

Input: C, F, Matrice de délai

Output: $X_{s_{k,i}}^{f_j}$

$F_{alloc} \leftarrow \emptyset$

$C_{dep} \leftarrow \emptyset$

for *chaque* $c_i \in C$ **do**

  # les noeuds sont triés en ordre croissant selon leurs distance avec les utilisateurs

  for *chaque* $f_j \in F$ **do**

    # vérifier les contraintes (3.2), (3.3) et (3.4)

    if f_j *peut héberger* c_i **then**

      $F_{alloc} \leftarrow F_{alloc} \cup \{f_j\}$

      $C_{dep} \leftarrow C_{dep} \cup \{c_i\}$

      $C \leftarrow C \setminus \{c_i\}$

    end

  end

  if $C_{dep} \neq \emptyset$ **then**

    Mettre à jour $Placement(C_{dep}, F_{alloc})$

  end

  return $Placement(C_{dep}, F_{alloc}) : X_{s_{k,i}}^{f_j}$

end

3.4.3 L'algorithme génétique : CPGA

À présent, nous détaillons la conception de notre heuristique CPGA, plus particulièrement l'heuristique sous-jacente, basée sur un algorithme génétique (Genetic

Algorithm - GA) et qui vise à trouver un schéma de placement de conteneurs au sein de l'infrastructure de *fog computing.*

L'algorithme génétique est considéré comme une technique d'optimisation discrète qui convient aux problèmes combinatoires tels que le problème de placement de conteneurs (Anderson et Ferris, 1994).

Description du fonctionnement de l'heuristique

Un GA est une variante d'algorithmes de recherche évolutionnaire, popularisés sous l'impulsion des travaux de John Holland (Holland, 1992). Le but est d'obtenir une solution approchée à un problème d'optimisation. Cette famille d'algorithmes est construite autour des principes de l'évolution biologique naturelle pour résoudre des problèmes d'optimisation complexes. Selon ces principes, chaque individu d'une population est contraint d'améliorer ou encore d'adapter ses propriétés biologiques s'il veut espérer survivre face aux nombreux défis des générations (Mitchell, 1996).

Un des avantages principaux d'un GA est sa capacité d'utiliser des informations antérieures pour réorienter de manière efficiente la recherche d'informations vers des régions plus prometteuses du vaste espace de recherche, pour obtenir des solutions quasi-optimales (Ye *et al.*, 2011).

Les éléments de base des algorithmes génétiques reposent sur la représentation des solutions du problème. Leur principe est d'encoder les paramètres du problème d'optimisation comme étant une combinaison de gènes formant des chromosomes. On associe les solutions potentielles du problème d'optimisation à un chromosome qui représente un point de l'espace de recherche. Par l'intermédiaire d'une fonction d'évaluation (appelé aussi fonction *fitness*), ce chromosome est évalué afin de quantifier sa proximité par rapport à la solution optimale (Mitchell, 1996).

Ainsi, les chromosomes non adaptés (c'est-à-dire ceux qui ne représentent pas une solution réalisable) sont éliminés à chaque génération, et en raffinant les qualités des chromosomes subsistants par un couplage de chromosomes parents ayant les meilleurs atouts génétiques, il peut être prouvé que l'algorithme génétique converge progressivement vers une solution optimale ou quasi optimale (Cerf, 1994).

La première étape d'exécution d'un algorithme génétique consiste à transposer adéquatement le problème d'optimisation ainsi que ses paramètres en des termes génétiques (chromosome, gène, etc.) étant donné que l'efficacité de l'algorithme génétique en dépend fortement (Wu *et al.*, 2017; Gan et Learmonth, 2016).

Ainsi, chaque solution potentielle du problème d'optimisation sera encodé sous la forme d'un chromosome avec des gènes (paramètres de la solution) sur lequel on applique un processus évolutif. Ce codage est représenté sous forme de chaîne de bits contenant toute l'information nécessaire à la description d'un point dans l'espace de recherche. L'avantage de ce processus de codage est qu'il offre une abstraction du problème d'optimisation dans la mesure où il permet d'appliquer un processus plus ou moins générique à des problèmes spécifiques (Mouhamad, 2016).

Notre modèle de codage consiste à associer chaque schéma candidat de placement de conteneur à un chromosome, représentant ainsi une solution potentielle du problème.

Le problème de placement des conteneurs dans les noeuds de *fog* est un problème qui a pour objectif de trouver un emplacement adéquat pour chaque conteneur dans les noeuds de *fog*. De ce fait, une représentation de l'individu par une liste des noeuds hébergeant les conteneurs pourrait être appliquée. Le premier élément de la liste représente le noeud hébergeant le premier conteneur, le dernier élément de

la liste représente le noeud hébergeant le dernier conteneur. La figure 3.4 montre la représentation d'un chromosome.

Nous tirons alors profit du codage par valeur où chaque gène représente l'identifiant du noeud hébergeant un conteneur d'une application.

Le codage par valeur réelle constitue souvent la forme de représentation du chromosome la plus convenable et réaliste pour la résolution de problème d'optimisation dans la mesure où il permet de coder le chromosome en utilisant une séquence de valeurs liées au problème (Mouhamad, 2016).

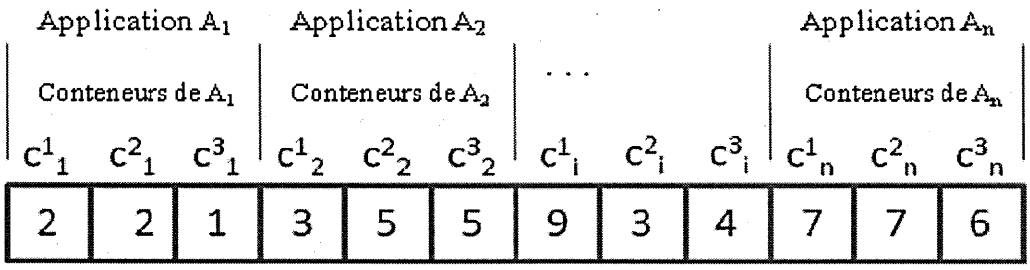

Figure 3.4 Représentation d'un individu dans CPGA

Description des étapes de l'heuristique CPGA

- **Sélection**

  Le processus de sélection consiste à choisir parmi les individus présents dans la population initiale, ceux ayant la plus forte probabilité de contribuer à l'obtention de meilleurs solutions au fil des générations à travers le processus de croisement et de mutation.

  Au début, CPGA génère une population initiale composée de schémas candidats de placement de conteneurs construits en effectuant une sélection aléatoire de noeuds de fog au sein de l'espace de recherche jusqu'à satisfaction des contraintes (3.2), (3.3) et (3.4), pour chaque schéma candidat de placement de conteneurs.

Dans notre algorithme CPGA, nous avons utilisé l'opérateur de sélection par tournoi, en raison de ses performances élevées en termes de temps d'exécution et de pertinence des solutions sélectionnées (Jinghui Zhong *et al.*, 2005; Razali *et al.*, 2011). Son principe est de faire un tirage de n individus (taille du tournoi) dans la population donnant lieu à une compétition, l'individu dont la fonction d'évaluation (fitness) est la plus élevée est désigné comme parent.
Notre stratégie de sélection par tournoi est résumée dans l'algorithme 3.

- **Croisement**

  La prochaine étape de l'exécution de CPGA consiste à effectuer des opérations de croisement de chromosomes. Son but est d'enrichir la diversité de la population en manipulant la structure des chromosomes. Classiquement, les croisements sont envisagés avec deux parents et génèrent deux enfants. Le croisement est un processus de reproduction à travers lequel deux parents ayant réussis à traverser le processus de sélection génèrent un ou des enfants biologiquement mieux adaptés en combinant leurs caractéristiques. Dans le contexte de CPGA, il s'agira de combiner deux schémas de placements pour en construire un schéma fils de placement qui optimisera mieux la fonction objectif. Le croisement consistera également à déterminer la manière dont les enfants héritent des caractéristiques de leurs parents.
  Pour CPGA, nous avons adopté l'opérateur de croisement en un point (Bridges et Goldberg, 1987), qui consiste à choisir un point de croisement au hasard. Un enfant prend une section avant le point de croisement d'un parent et prend l'autre section après le point de croisement de l'autre parent, puis il recombine les deux sections pour former un nouvel individu. L'autre enfant se construit inversement, tel qu'illustré sur la figure 3.5. Ce type de croisement à découpage de chromosomes est très efficace pour les problèmes discrets (Magalhaes-Mendes, 2013).
  Notre stratégie de croisement est résumée dans l'algorithme 4

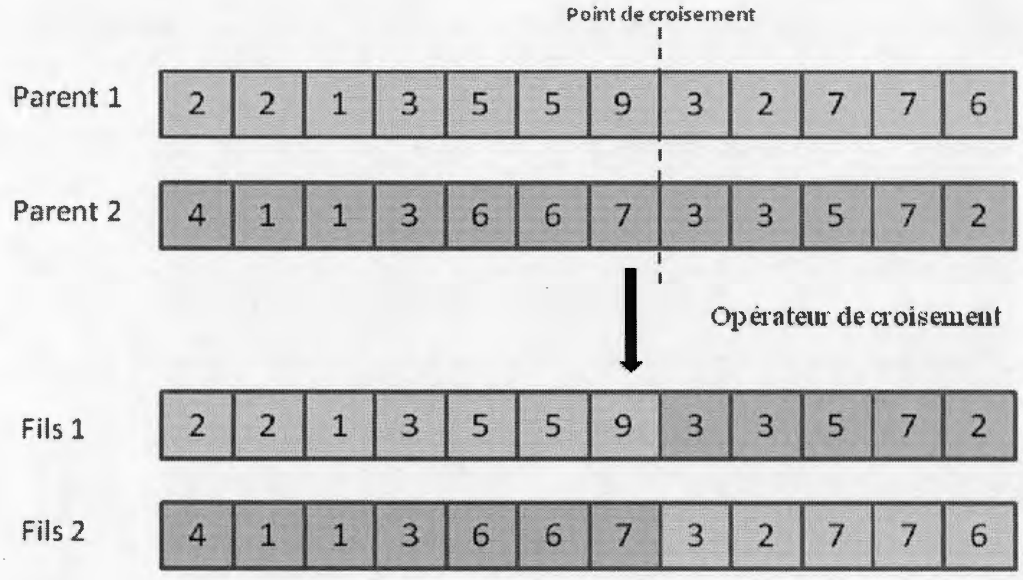

Figure 3.5 Représentation du croisement dans CPGA

- **Mutation**

  Le rôle de cet opérateur est de modifier aléatoirement les caractéristiques d'une solution. Il joue le rôle d'élément perturbateur, qui permet une exploration efficiente de nouvelles régions de l'espace de recherche avec comme objectif d'éviter une convergence prématurée vers des optimums locaux (Mitchell, 1996), en s'assurant qu'il y ait un minimum de diversité génétique au sein de la population. Le principe de la mutation est de modifier aléatoirement une portion de bits ou un groupe de gènes du chromosome de sorte à générer un nouvel individu.

  On note également que la mutation ne s'applique pas forcement à tous les individus de la population. En effet, la fréquence selon laquelle les individus subissent une mutation de gènes est déterminée par la probabilité de mutation. Afin d'éviter une recherche aléatoire, la valeur de la probabilité de mutation

retenue (de manière statique ou dynamique) doit rester relativement faible (typiquement moins de 2%) (Pandey *et al.*, 2014).

Dans notre algorithme, nous avons utilisé l'opérateur de mutation à un point, où on choisit un point aléatoire dans le chromosome puis on change sa valeur par un autre identifiant d'un noeud, tel qu'illustré sur la figure 3.6. Notre mécanisme de mutation est utilisé principalement pour corriger les individus qui ne sont pas valides. Notre stratégie de mutation est résumée dans l'algorithme 5.

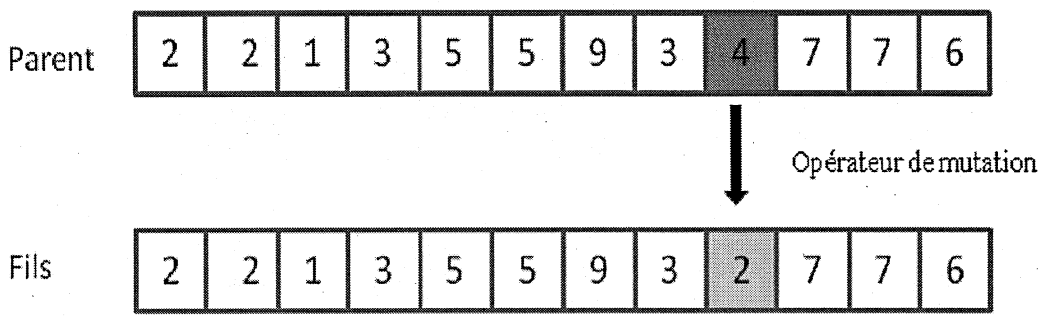

Figure 3.6 Représentation du mutation dans CPGA

Nous procédons ensuite à déterminer la qualité des schémas fils à travers une fonction d'évaluation correspondant à l'équation 3.1. Dès lors, nous trions chaque schéma fils de placement par ordre décroissant en fonction de son score d'évaluation.

À la fin des itérations, le schéma de placement ayant le meilleur score d'évaluation est adopté comme solution finale pour l'ensemble de conteneurs concernés.

Nous résumons notre heuristique de placement de conteneurs basée sur l'algorithme génétique dans les algorithmes 2, 3, 4, 5 et 6.

Algorithm 2: CPGA Container Placement based on Genetic Algorithm

Input: C, F, Matrice de délai

Output: $X_{s_{k,i}}^{f_j}$

1 *# Paramètres : taille de population (pop_size), nombre de génération (gen_num), taille du tournoi (k), probabilité de croissement (p_c), probabilité de mutation (p_m)*

2 *# L'ensemble des conteneurs C, et l'ensemble des noeuds F servent à générer la population initiale*

3 $POP \leftarrow$ population *pop_size* d'individu générer aléatoirement (avec test de faisabilité)

4 $Best \leftarrow \infty$

5 $Iter \leftarrow 1$

6 **while** *la condition d'arrêt n'est pas vérifié (Iter < gen_num)* **do**

7 *# Sélection*

8 $P_1 \leftarrow$ Selection_tournoi(POP)

9 $P_2 \leftarrow$ Selection_tournoi(POP)

10 *# Opérations de croisement et de mutation*

11 $R \leftarrow random(0, 1)$

12 **if** $R < p_c$ **then**

13 $POP\prime \leftarrow$ Croisement(P_1, P_2)

14 **end**

15 **if** $R < p_m$ **then**

16 $POP\prime \leftarrow$ Mutation(P_1)

17 **end**

18 *# Mettre à jour la population courante*

19 $POP \leftarrow POP \cup POP\prime$

20 # Évaluation ;

21 $newBest \leftarrow \min_{i \in POP} fitness(i)$;

22 **if** $newBest < Best$ **then**

23 $Best \leftarrow newBest$;

24 $Meilleur\_individu \leftarrow individu\ i$

25 **end**

26 $Iter \leftarrow Iter + 1$;

27 **end**

28 **return** $Meilleur\_individu : X_{s_{k,i}}^{f_j}$

Algorithm 3: Selection_tournoi

Input: Population courante POP

Output: Meilleur individu P

1 $BestScore \leftarrow \infty$

2 $BestInd \leftarrow Null$

3 **for** *i=1 to k* **do**

4 $newInd \leftarrow$ sélectionner au hasard un individu de POP

5 **if** *newInd vérifier les contraintes (3.2), (3.3) et (3.4)* **then**

6 $newScore \leftarrow$ fitness($newInd$)

7 **if** $newScore < BestScore$ **then**

8 $BestScore \leftarrow newScore$

9 $BestInd \leftarrow newInd$

10 **end**

11 **end**

12 **end**

13 **return** $BestInd$

Algorithm 4: Croisement

Input: Deux individus parents P_1, P_2

Output: Deux individus fils C_1, C_2

```
1  newInds ← Null
2  position ← random(1, size(P1))
3  for i = 1 to position do
4  |  C1(i) ← P1(i)
5  |  C2(i) ← P2(i)
6  end
7  for i = position + 1 to size(P1) do
8  |  #échanger les gênes
9  |  C1(i) ← P2(i)
10 |  C2(i) ← P1(i)
11 end
12 #vérifier que C1 et C2 respectent les contraintes (3.2), (3.3) et (3.4)
13 newInds ← C1 ∪ C2
14 return newInds
```

Algorithm 5: Mutation

Input: Un individu parent P_1

Output: Un individu fils C_1

$position \leftarrow random(1, size(P_1))$

$newGene \leftarrow random(F)$

$P_1(position) \leftarrow newGene$

#vérifier que P_1 respect les contraintes (3.2), (3.3) et (3.4)

$C_1 \leftarrow P_1$

return C_1

Algorithm 6: fitness

Input: Un schéma de placement Ind

Output: Score

$Score \leftarrow 0$

for *i=1 to size(Ind) - 1* **do**

    # f_i désigne la valeur du gène à la position i

    # $f_i = Ind(i)$, $f_{i+1} = Ind(i+1)$

    $Score \leftarrow Score + d_{f_i, f_{i+1}}$

end

return *Score*

3.5 Évaluation des performances

3.5.1 Environnement de simulation

Dans nos simulations, nous supposons que tous les nœuds de *fog* appartiennent au même domaine, donc les noeuds sont sous le même contrôle administratif. De plus, nous considérons que les nœuds de *fog* sont statiques. Nous considérons que

les requêtes des utilisateurs sont traitées en fonction de leur ordre d'arrivée.

Dans notre implementation de CPGA, nous avons défini le nombre de générations à 500 générations, et la taille de la population est fixée à 100.

Comme mentionné précédemment, le mécanisme de sélection utilisé est la stratégie de sélection par tournoi d'une taille égale à 2. Le taux d'opérations de croisement et de mutation sont respectivement fixés à 0,07 et 0,01. Un plan de placement de conteneurs optimal est sélectionné après 500 générations.

Nos simulations sont effectuées sur une machine physique fonctionnant sous Ubuntu 18.04, avec un processeur Intel Xeon (R) E5 de 32 cœurs de 2 GHz, et 64 Go de RAM.

3.5.2 Scénario de simulation

Dans notre évaluation, nous considérons un réseau de *fog* composé de dix nœuds, la topologie du réseau de *fog* est générée à l'aide de l'outil de génération de topologie réseau dénommé BRITE (Medina *et al.*, 2001), nous considérons vingt applications différentes, chaque application est constituée de trois services déployés sur trois conteneurs.

Dans ce travail, nous avons comparé les performances de trois algorithmes d'optimisation décrits précédemment : ILP implémenté à l'aide de la bibliothèque CPLEX (IBM, 2019) et servant également de point de référence à nos évaluations, l'algoritme glouton et enfin CPGA. Un résumé de nos paramètres expérimentaux est présenté dans les tableaux 3.3, 3.4 et 3.5.

Tableau 3.3 Scénarios d'évaluation

Scénario	Nombre de noeuds (RDMA)	Nombre de noeuds (Overlay)	Nombre de noeuds (Host)
Scénario 1	4	4	2
Scénario 2	2	4	4
Scénario 3	0	5	5

Tableau 3.4 Ressources du Fog (Gupta *et al.*, 2017)

Noeuds du fog	CPU (MIPS)	RAM (MB)	DISK (MB)
Noeud1	1000	1000	3000
Noeud2	500	250	500
Noeud3	2000	1500	4000
Noeud4	1000	1000	3000
Noeud5	1500	1000	2000
Noeud6	1000	1000	1000
Noeud7	500	500	1000
Noeud8	2000	2000	4000
Noeud9	1000	1000	2000
Noeud10	500	500	1000

Tableau 3.5 Ressources demandées

Services	CPU (MIPS)	RAM(MB)	STOCKAGE(MB)	$\text{Temps}_{exec}(s)$
Application a_1 : $D_{a_1} = 20$ s, $D^t_{a_1} = 15$s (Ahmed et Pierre, 2018)				
Conteneur 1	200	50	50	0.8
Conteneur 2	200	50	50	0.8
Conteneur 3	200	50	50	0.8

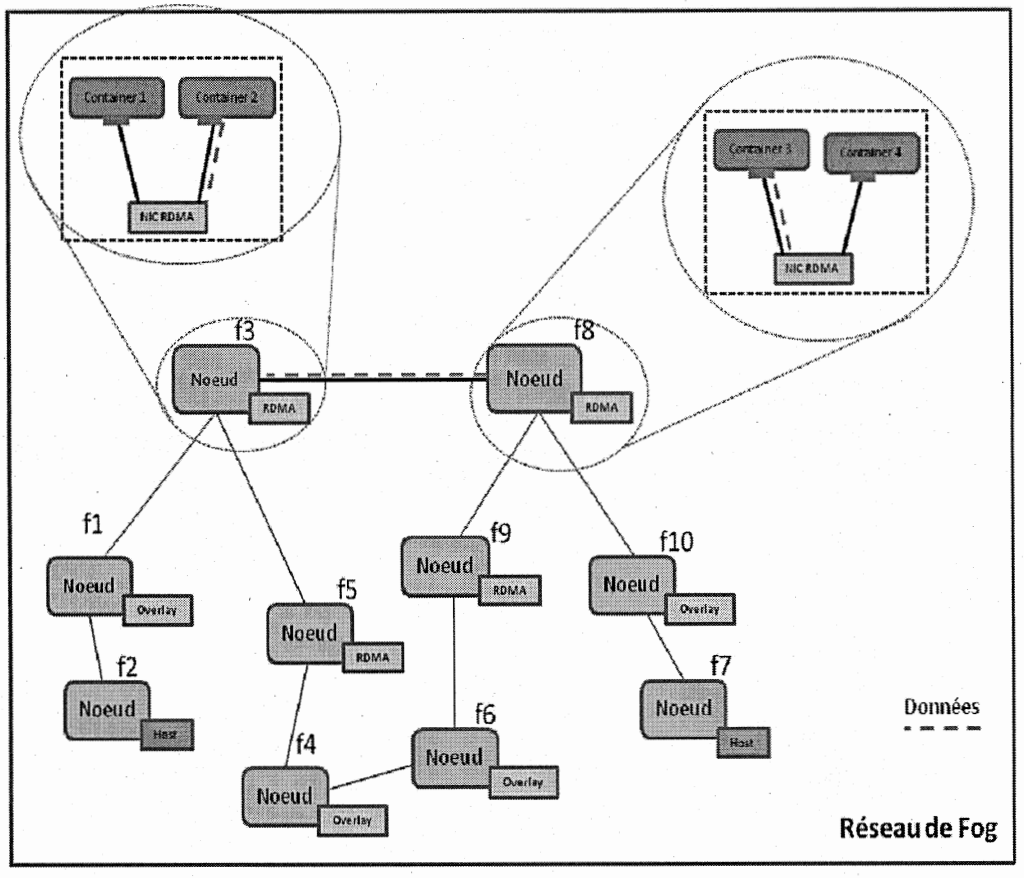

Figure 3.7 Scénario d'évaluation

3.5.3 Résultats et discussions

Dans ce travail, nous nous sommes concentrés principalement sur l'étude de l'impact de différentes technologies de communication inter-conteneurs sur les performances des applications en termes de temps de réponse. Plus précisément, nous examinons les métriques suivantes pour évaluer l'efficacité d'un plan de placement de conteneurs : (1) le temps de réponse du système aux requêtes des utilisateurs ; (2) le nombre de nœuds de *fog* actives, afin de mesurer l'utilisation des ressources ainsi que la consommation d'énergie qui en résulte ; et enfin, le temps d'exécution de l'algorithme pour indiquer le potentiel d'évolutivité à mesure que le nombre de

requêtes augmente. Nous avons simulé trois scénarios, résumés dans le tableau 3.3, dans lesquels le protocole de communication inter-conteneurs implémenté dans les noeuds varie. Nous avons d'abord étudié, en utilisant ILP, le temps de réponse moyen du système pour chaque scénario simulé en fonction du nombre de requêtes reçues par le contrôleur de *fog*, comme montré sur la figure 3.8.

Nous observons que le temps de réponse moyen du système diminue avec l'augmentation du nombre de noeuds qui implémente la technique RDMA. L'impact de RDMA en tant que protocole de communication inter-conteneurs est donc évident en termes d'amélioration des performances du réseau. Le gain en performances peut s'expliquer par la capacité de RDMA à exploiter les mécanismes de zéro-copie avec le contournement du noyau pour réduire les traitements du processeur.

Nous avons également comparé les performances de chaque algorithme de placement de conteneur (à savoir ILP, glouton et CPGA) précédemment décrit en utilisant le scénario 2 comme base de simulation. La figure 3.9 illustre sur une échelle logarithmique le temps d'exécution de chaque algorithme en fonction du nombre de requêtes. Nous observons que ILP devient de plus en plus complexe en temps de calcul au fur et à mesure que les variables d'entrée augmentent avec le nombre de requêtes.

Par contre, Greedy et CPGA montrent une évolution plus stable en termes de temps d'exécution, indiquant une compatibilité pour les environnements de *fog* à grande échelle. Il convient de noter que Greedy montre le temps d'exécution le plus faible, ce qui s'explique par la simplicité relative de la recherche de placement de conteneurs, qui consiste simplement à rechercher les nœuds de *fog* compatibles (c'est-à-dire les noeuds de *fog* qui disposent des ressources) les plus proches des utilisateurs. Cependant, les limites de l'algorithme *Greedy* sont entièrement illus-

trées dans la figure 3.10 où nous observons que le temps de réponse moyen obtenu est supérieur à celui de CPGA et ILP. Ainsi nous soulignons l'incapacité de l'algorithme *Greedy* de trouver une stratégie de placement de conteneurs quasi optimale à mesure que le nombre de requêtes augmente. Par contre, CPGA révèle une meilleure capacité à découvrir et à placer des conteneurs dans des noeuds de fog, présentant un plan de placement proche de l'optimum.

En fait, nous observons une disparité moyenne de 10-15 ms du temps de réponse entre CPGA et ILP. Nous émettons l'hypothèse que les opérateurs de croisement et de mutation mis à profit par CPGA permettent une meilleure exploration de l'espace de recherche, évitant ainsi de rester bloqués dans des optima locaux. Il convient de noter que le temps d'exécution de CPGA dépend également du nombre de générations considérées. Le compromis étant que CPGA pourrait ne pas converger vers des plans optimaux avec un nombre réduit de générations.

Pour illustrer cela, nous montrons dans la figure 3.12 le temps de réponse moyen obtenu par CPGA en fonction du nombre de générations. Nous notons que CPGA converge vers des schémas de placement de conteneurs quasi optimaux après 500 générations, ce qui explique pourquoi nous l'avons défini à 500 générations dans les autres simulations (3.9, 3.10 et 3.11).

Nous avons également évalué le nombre de nœuds utilisés en fonction du nombre de requêtes pour chaque algorithme. Les résultats sont illustrés dans la figure 3.11, où ILP semble consommer le moins de ressources de *fog*. Comme Greedy cherche de manière exhaustive des nœuds de *fog* voisins pour le placement de conteneurs, cela en résulte une consommation de ressources élevée. De toute évidence, une consommation élevée de ressources a des impacts importants, notamment une grande consommation d'énergie et des dépenses importantes de fonctionnement et de maintenance de l'infrastructure du *fog*.

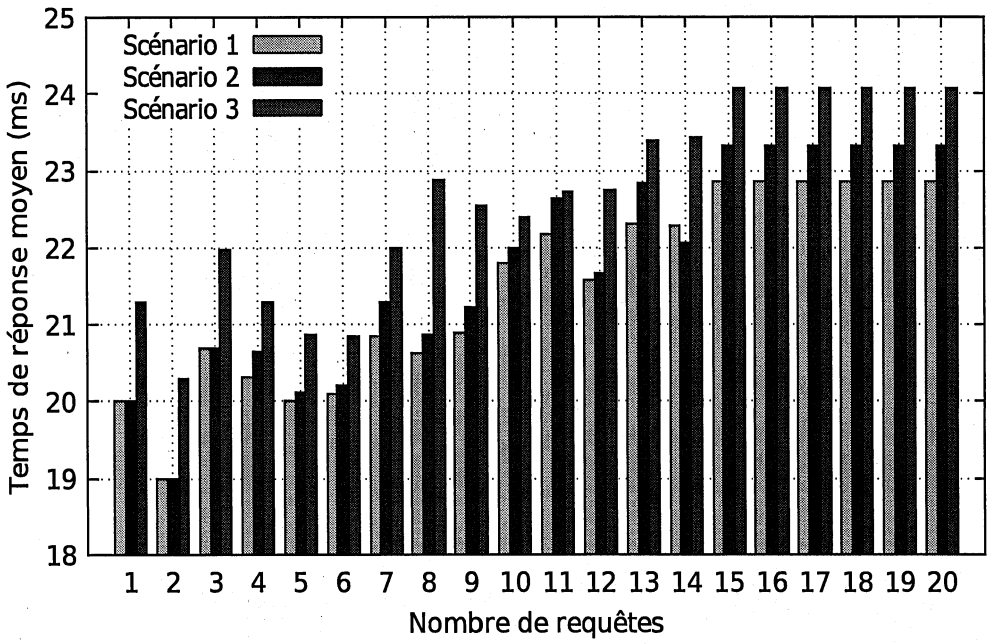

Figure 3.8 Variation du temps de réponse en fonction du scénario.

3.6 Conclusion

Nous avons présenté dans ce travail une stratégie de placement de conteneurs basée sur l'algorithme génétique qui prend en compte les nœuds de fog hétérogènes en plus de diverses technologies de communication entre conteneurs. Nous avons également étudié leurs impacts sur les performances des applications, en utilisant des approches ILP, algorithme glouton et CPGA. Les évaluations expérimentales ont montré que notre algorithme CPGA surpasse les performances des autres approches testées. Nous avons également mis en évidence les avantages considérables du mécanisme RDMA en termes de temps de réponse du système.

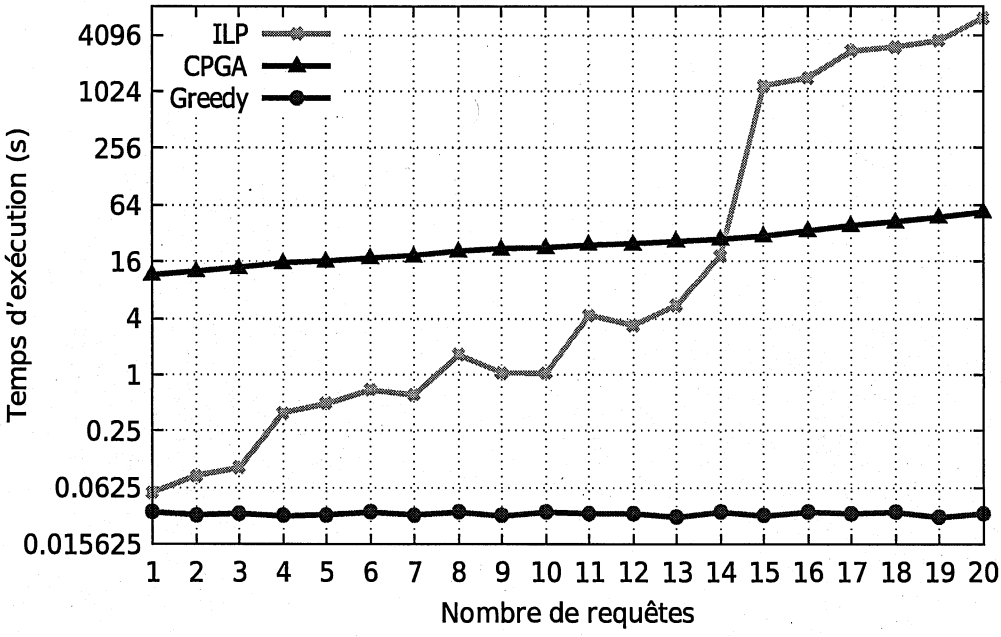

Figure 3.9 Variation du temps d'exécution en fonction du nombre de requêtes.

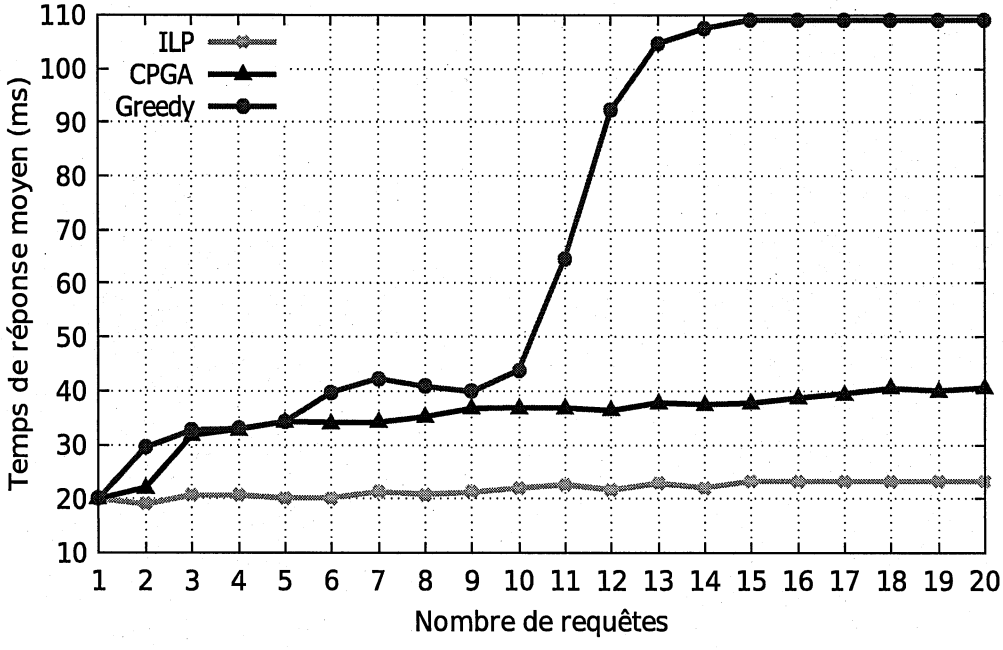

Figure 3.10 Variation du temps de réponse en fonction du nombre de requêtes.

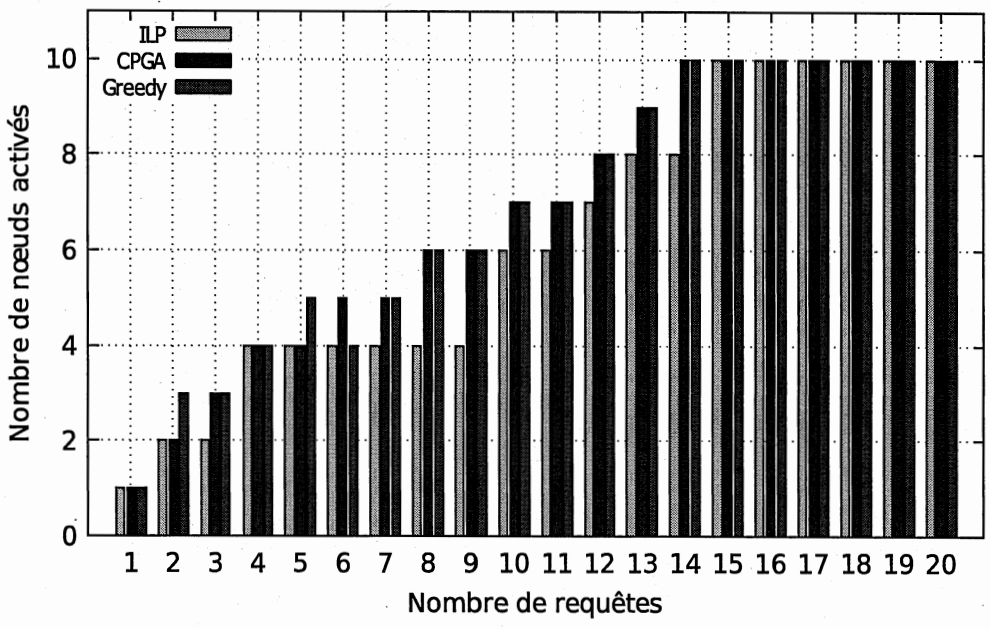

Figure 3.11 Variation du nombre de noeuds actives en fonction du nombre de requêtes.

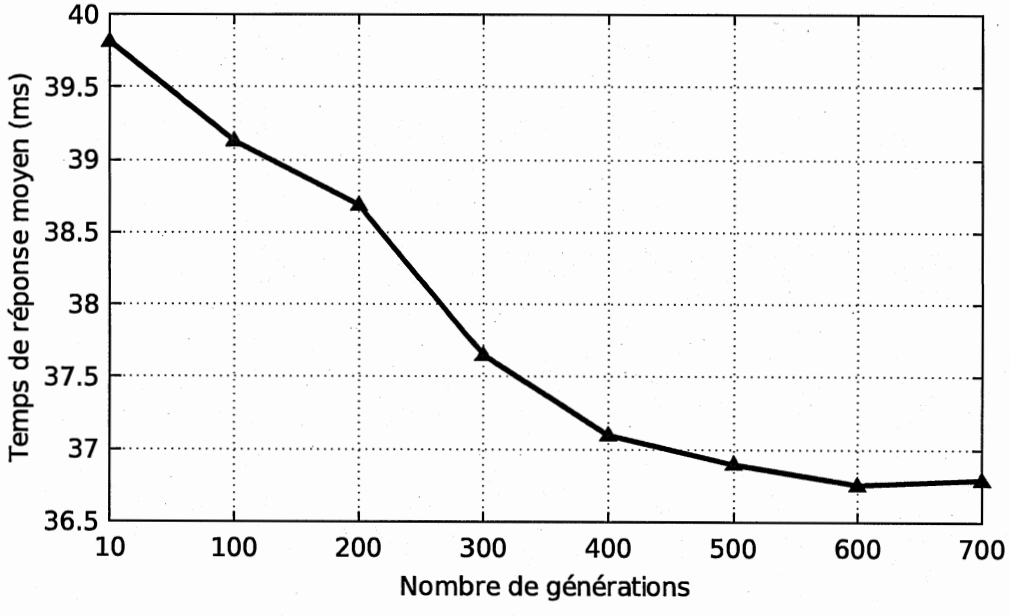

Figure 3.12 Performance du CPGA.

CONCLUSION

Les dernières années ont vu l'augmentation du nombre d'objets connectés à Internet IoT, demandant des services différents pour différents domaines d'applications. Cela a entraîné une hausse augmentation de la quantité de données à traiter dans les environnements informatique traditionnelle comme le *cloud computing*. Cependant, l'environnement du *cloud computing* n'est pas convenable pour les applications IoT sensibles aux délais. En fait, cela a donné naissance à une nouvelle infrastructure : *fog computing*, qui traite les données des applications toutes proches des utilisateurs finaux ce qui permet de réduire le délai et donc répondre aux exigences de délai des applications IoT. Cette infrastructure du *fog computing* s'appuie sur la virtualisation applicative par conteneur Docker, comme étant une technologie de virtualisation léger permettant ainsi de réduire le délai de déploiement des applications.

Toutefois, le déploiement des conteneurs dans le *fog computing* n'est pas trivial. En effet, plusieurs difficultés doivent être résolues, parmi elles nous pouvons citer la sécurité et la confidentialité, la gestion du réseau, l'amélioration de la qualité de service (protocoles de communication entre les conteneurs, l'allocation des ressources, etc).

Comme nous l'avons présenté, le défi de performance de communication entre conteneurs Docker est résolu par l'adoption du mécanisme d'accès direct à la mémoire RDMA comme protocole de communication entre les applications conteneurisées. Quant à l'allocation des ressources, nous avons proposé une stratégie de placement de conteneurs basée sur l'algorithme génétique CPGA qui prend en

compte les nœuds de *fog* hétérogènes en plus de mécanismes de communication réseau entre conteneurs. Nous avons étudié leurs impacts sur les performances des applications à l'aide de scénarios réalistes, en utilisant des approches ILP, gloutonne et CPGA.

Les évaluations expérimentales ont montrés que le mécanisme de communication RDMA, et la stratégie de placemenet CPGA surpassent les performances des autres approches testées.

Nous envisageons dans nos travaux futurs de mettre en œuvre un scénario plus réaliste tenant compte de la mobilité des nœuds de *fog*. Nous comptons aussi d'améliorer le modèle de système d'allocation des ressources en termes de fiabilité et de disponibilité des services.

RÉFÉRENCES

Abbasi, U., Bourhim, E. H., Dieye, M. et Elbiaze, H. (2019). A performance comparison of container networking alternatives. *IEEE Network*, *33*(4), 178–185.

Agarwal, K. (2015). *MA study of virtualization overheads.* (Thèse de doctorat). Stony Brook University.

Ahmed, A. et Pierre, G. (2018). Docker container deployment in fog computing infrastructures. Dans *2018 IEEE International Conference on Edge Computing (EDGE)*, 1–8.

Ai, Y., Peng, M. et Zhang, K. (2018). Edge computing technologies for internet of things : a primer. *Digital Communications and Networks*, *4*(2), 77 – 86.

Ait Salaht, F., Desprez, F. et Lebre, A. (2019). *An overview of service placement problem in Fog and Edge Computing.* Research Report RR-9295, Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, LYON, France

AlJahdali, H., Albatli, A., Garraghan, P., Townend, P., Lau, L. et Xu, J. (2014). Multi-tenancy in cloud computing. Dans *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, 344–351.

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.

Anderson, E. et Ferris, M. (1994). Genetic algorithms for combinatorial optimization : The assemble line balancing problem. *INFORMS Journal on Computing*, *6*, 161–173.

Atlam, H., Walters, R. et Wills, G. (2018). Fog computing and the internet of things : A review. *Big Data and Cognitive Computing*, *2*.

Barachi, M. E., Kara, N. et Dssouli, R. (2010). Towards a service-oriented network virtualization architecture. Dans *2010 ITU-T Kaleidoscope : Beyond the Internet ? - Innovations for Future Networks and Services*, 1–7.

Barik, R. K., Lenka, R. K., Rao, K. R. et Ghose, D. (2016). Performance analysis of virtual machines and containers in cloud computing. Dans *2016 International Conference on Computing, Communication and Automation (ICCCA)*.

Bellavista, P. et Zanni, A. (2017). Feasibility of fog computing deployment based on docker containerization over raspberrypi. Dans *Proceedings of the 18th International Conference on Distributed Computing and Networking*, ICDCN '17, 16 :1–16 :10., New York, NY, USA. ACM.

Birrittella, M. S., Debbage, M., Huggahalli, R., Kunz, J., Lovett, T., Rimmer, T., Underwood, K. D. et Zak, R. C. (2015). Intel® omni-path architecture : Enabling scalable, high performance fabrics. Dans *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 1–9.

Bonomi, F., Milito, R., Zhu, J. et Addepalli, S. (2012). Fog computing and its role in the internet of things. Dans *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, 13–16., New York, NY, USA. ACM.

Borgione, G. (2017). Container networking deep dive. Récupéré le 2019-06-13 de `https://blogs.vmware.com/cloudnative`

Borkar, S., Cohn, R., Cox, G., Gleason, S., Gross, T., Kung, H. T., Lam, M., Moore, B., Peterson, C., Pieper, J., Rankin, L., Tseng, P. S., Sutton, J., Urbanski, J. et Webb, J. (1988). iwarp : an integrated solution to high-speed parallel computing. Dans *Proceedings. SUPERCOMPUTING '88.*

Bourhim, E. H., Elbiaze, H. et Dieye, M. (2019). Inter-container communication aware container placement in fog computing. Dans *15th International Conference on Network and Service Management(CNSM)*. IFIP/IEEE.

Bridges, C. L. et Goldberg, D. E. (1987). An analysis of reproduction and crossover in a binary-coded genetic algorithm. *Grefenstette, 878*, 9–13.

Bröse, E. (2008). Zerocopy : Techniques benefits and pitfalls.

Calico (2019). Project calico. Récupéré le 2019-07-20 de `http://www.projectcalico.org/`

Cerf, R. (1994). *Une théorie asymptotique des algorithmes génétiques.* (Thèse de doctorat). Récupéré de `http://www.theses.fr/1994MON20056`

Choi, H.-H., Kim, K. et Kang, D.-J. (2017). Performance evaluation of a remote block device with high-speed cluster interconnects. Dans *Proceedings of the 8th International Conference on Computer Modeling and Simulation*, ICCMS '17.

ACM.

Chowdhury, N. M. K. et Boutaba, R. (2010). A survey of network virtualization. *Computer Networks*, *54*(5), 862 – 876.

Cisco (2016). Fog computing and the internet of things : Extend the cloud to where the things are. Récupéré le 2019-06-22 de `https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf`

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

Coreos (2019). Coreos : Open source, containers, and kubernetes. Récupéré le 2019-05-08 de `https://coreos.com/`

Cormen, T. H., Leiserson, C. E., Rivest, R. L. et Stein, C. (2009). *Introduction to algorithms*. MIT press.

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

Dastjerdi, A. V. et Buyya, R. (2016). Fog computing : Helping the internet of things realize its potential. *Computer*, *49*(8), 112–116.

Department, S. R. (2019). Statista research department. Récupéré le 2019-12-26 de `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`

Diouf, G. M. (2019). *Une pateforme d'orchestration de conteneurs Docker tolérant aux fautes byzantines*. (Mémoire de maîtrise). Université du Québec à Montréal.

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.

Dunning, D., Regnier, G., McAlpine, G., Cameron, D., Shubert, B., Berry, F., Merritt, A. M., Gronke, E. et Dodd, C. (1998). The virtual interface architecture. *IEEE Micro*, *18*(2), 66–76.

Felter, W., Ferreira, A., Rajamony, R. et Rubio, J. (2015a). 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.

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

(ISPASS), 171–172.

Fernando, A. (2018). Why it's important to keep your containers small and simple. Récupéré le 2019-05-07 de `https://hackernoon.com/why-its-important-to-keep-your-containers-small-and-simple-618ced7343a5`

Flannel (2015). Flannel. Récupéré le 2019-05-08 de `https://github.com/coreos/flannel`

Gan, C. C. et Learmonth, G. (2016). An improved chromosome formulation for genetic algorithms applied to variable selection with the inclusion of interaction terms. *ArXiv, abs/1604.06727*.

George, J. (2018). qperf - Measure RDMA and IP performance. Récupéré le 2019-07-01 de `https://linux.die.net/man/1/qperf`

Google (2018). Kubernetes. Récupéré le 2019-05-16 de `https://kubernetes.io/`

Goth, G. (2007). Virtualization : Old technology offers huge new potential. *IEEE Distributed Systems Online*, *8*(2), 3–3.

Grainia, S. (2015). *L'algorithme de Branch and Price and Cut pour le problème de conception de réseaux avec coûts fixes et sans capacité.* (Mémoire de maîtrise). Université de Montréal.

Gupta, H., Dastjerdi, A. V., Ghosh, S. K. et Buyya, R. (2017). ifogsim : A toolkit for modeling and simulation of resource management techniques in internet of things, edge and fog computing environments. *Softw., Pract. Exper.*, *47*, 1275–1296.

Gurr, C. (2008). Facilitating Microsoft Windows Vista Migration Through Application Virtualization. Récupéré le 2019-08-16 de `https://www.dell.com/downloads/global/power/ps1q08-20080154-LANDesk.pdf`

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

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems : An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA : MIT Press.

Hong, C.-H., Lee, K., Kang, M. et Yoo, C. (2018). qcon : Qos-aware network resource management for fog computing. *Sensors*, *18*(10).

Huang, X., Ganapathy, S. et Wolf, T. (2009). Evaluating algorithms for composable service placement in computer networks. Dans *2009 IEEE International Conference on Communications*, 1–6.

Hypercontainer (2017). About hypercontainer. Récupéré le 2019-07-10 de `https://github.com/hyperhq/docs.hypercontainer.io`

IBM (2019). CPLEX Optimizer. Récupéré le 2019-06-09 de `https://www.ibm.com/analytics/cplex-optimizer`

Infiniband (2010). *RDMA over converged Ethernet (RoCE)*. Rapport technique, Infiniband Trade Association. Supplement to InfiniBand architecture specification :Volume 1 Release 1.2.1, Annex A16.

Infiniband (2014). *RoCEv2 (IP routable RoCE)*. Rapport technique, Infiniband Trade Association. Supplement to InfiniBand architecture specification : volume 1 release 1.2.2, Annex A17.

Jinghui Zhong, Xiaomin Hu, Jun Zhang et Min Gu (2005). Comparison of performance between different selection strategies on simple genetic algorithms. Dans *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*, volume 2, 1115–1121.

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

katacontainers (2019). The speed of containers, the security of vms. Récupéré le 2019-06-12 de `https://katacontainers.io/`

Ketel, M. (2017). Fog-cloud services for iot. Dans *Proceedings of the SouthEast Conference*, ACM SE '17, 262–264., New York, NY, USA. ACM.

Larsen, S. et Lee, B. (2014). Survey on system i/o hardware transactions and impact on latency, throughput, and other factors. In *Advances in Computers*, volume 92 67–104. Elsevier.

Li, J., Jin, J., Yuan, D. et Zhang, H. (2018). Virtual fog : A virtualization enabled fog computing framework for internet of things. *IEEE Internet of Things Journal*, *5*(1), 121–131.

Li, Z., Kihl, M., Lu, Q. et Andersson, J. A. (2017). Performance overhead comparison between hypervisor and container based virtualization. Dans *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, 955–962.

Liu, Y., Fieldsend, J. E. et Min, G. (2017). A framework of fog computing : Architecture, challenges, and optimization. *IEEE Access*, *5*, 25445–25454.

Livi, S., Jacquemart, Q., Pacheco, D. L. et Urvoy-Keller, G. (2016). Container-based service chaining : A performance perspective. Dans *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, 176–181.

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.

Magalhaes-Mendes, J. (2013). A comparative study of crossover operators for genetic algorithms to solve the job shop scheduling problem. *WSEAS transactions on computers*, *12*(4), 164–173.

Mahalingam, M., Dutt, D. G., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M. et Wright, C. (2014). Virtual extensible local area network (vxlan) : A framework for overlaying virtualized layer 2 networks over layer 3 networks. *RFC*, *7348*, 1–22.

Mancel, C. (2004). *Modelisation et resolution de problemes d'optimisation combinatoire issus d'applications spatiales.* (Thèse de doctorat). INSA de Toulouse.

Medina, A., Lakhina, A., Matta, I. et Byers, J. (2001). Brite : an approach to universal topology generation. Dans *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 346–353.

Mellanox (2014). *Introduction to InfiniBand.* Rapport technique.

Menascé, D. A. (2005). Virtualization : Concepts, applications, and performance modeling. Dans *Int. CMG Conference*, 407–414.

Meyer, D. (2008). The locator identifier separation protocol (lisp). *The Internet Protocol Journal*, *11*, 23–36.

Microsoft (2019). What is virtualisation ? Récupéré le 2019-08-16 de `https://azure.microsoft.com/en-in/overview/what-is-virtualization/`

Mitchell, M. (1996). *An Introduction to Genetic Algorithms.* Cambridge, MA, USA : MIT Press.

Mouhamad, D. (2016). *Disponibilité des données dans les centres de données à caractéristiques hétérogènes.* (Mémoire de maîtrise). Université du Québec à Montréal.

Mouradian, C., Naboulsi, D., Yangui, S., Glitho, R. H., Morrow, M. J. et Polakos, P. A. (2017). A comprehensive survey on fog computing : State-of-the-art

and research challenges. *IEEE Communications Surveys & Tutorials*, *20*(1), 416–464.

Nabli, H. (2012). Optimisation linéaire. Récupéré le 2019-07-20 de `http://wims.unice.fr/wims/fr_U3~opresearch~doclinearoptim.fr.html`

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.

Ordonez-Lucena, J., Ameigeiras, P., Lopez, D., Ramos-Munoz, J. J., Lorca, J. et Folgueira, J. (2017). Network Slicing for 5G with SDN/NFV : Concepts, Architectures, and Challenges. *IEEE Communications Magazine*, *55*(5), 80–87.

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

Pandey, H. M., Chaudhary, A. et Mehrotra, D. (2014). A comparative review of approaches to prevent premature convergence in ga. *Appl. Soft Comput.*, *24*(C), 1047–1077.

Peralta, G., Iglesias-Urkia, M., Barcelo, M., Gomez, R., Moran, A. et Bilbao, J. (2017). Fog computing based efficient iot scheme for the industry 4.0. Dans *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, 1–6.

Petit, B. (2018). Les réseaux d'overlay : principes et fonctionnement. Récupéré le 2019-08-12 de `https://blog.wescale.fr/2018/02/15/les-reseaux-doverlay-principes-et-fonctionnement/`

Puliafito, C., Vallati, C., Mingozzi, E., Merlino, G., Longo, F. et Puliafito, A. (2019). Container migration in the fog : A performance evaluation. *Sensors*, *19*, 1488.

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

Ramalho, F. et Neto, A. (2016). Virtualization at the network edge : A performance comparison. Dans *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 1–6.

Razali, N. M., Geraghty, J. *et al.* (2011). Genetic algorithm performance with different selection strategies in solving tsp. Dans *Proceedings of the world congress on engineering*, volume 2, 1–6. International Association of Engineers Hong Kong.

Robin, G. (2016). Open vSwitch with DPDK Overview. Récupéré le 2019-08-27 de `https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview`

Roucairol, C. (1989). *Parallel branch and bound algorithms- an overview.* Rapport technique RR-0962, INRIA

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.

Shpiner, A., Zahavi, E., Dahley, O., Barnea, A., Damsker, R., Yekelis, G., Zus, M., Kuta, E. et Baram, D. (2017). Roce rocks without pfc : Detailed evaluation. 25–30.

Singh, V. et Peddoju, S. K. (2017). Container-based microservice architecture for cloud applications. Dans *2017 International Conference on Computing, Communication and Automation (ICCCA)*, 847–852.

Skarlat, O., Schulte, S., Borkowski, M. et Leitner, P. (2016). Resource provisioning for iot services in the fog. Dans *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, 32–39.

Sri Raghavendra, M. et Chawla, P. (2018). A review on container-based lightweight virtualization for fog computing. Dans *2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, 378–384.

Stowe, M. (2017). Breaking down the boxes : Containers. Récupéré le 2019-07-20 de `https://www.projectcalico.org/breaking-down-the-boxes-containers/`

Suo, K., Zhao, Y., Chen, W. et Rao, J. (2018). An analysis and empirical study of container networks. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 189–197.

Tang, L., Yang, H., Ma, R., Hu, L., Wang, W. et Chen, Q. (2018). Queue-aware dynamic placement of virtual network functions in 5g access network. *IEEE Access*, *6*, 44291–44305.

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.

Wang, A., Iyer, M., Dutta, R., Rouskas, G. N. et Baldine, I. (2013). Network

virtualization : Technologies, perspectives, and frontiers. *Journal of Lightwave Technology, 31*(4), 523–537.

Weave (2017). Weave net. Récupéré le 2019-05-08 de `https://www.weave.works/`

Wu, M.-C., Lin, C.-S., Lin, C.-H. et Chen, C.-F. (2017). Effects of different chromosome representations in developing genetic algorithms to solve dfjs scheduling problems. *Computers Operations Research, 80*, 101 – 112.

YANG Yong, D. X. et Zhenjiang, D. (2017). Technical analysis of network plugin flannel for containers. *ZTE Communications magazine, 4*.

Yannuzzi, M., Milito, R., Serral-Gracià, R., Montero, D. et Nemirovsky, M. (2014). Key ingredients in an iot recipe : Fog computing, cloud computing, and more fog computing. Dans *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, 325–329.

Ye, Z., Zhou, X. et Bouguettaya, A. (2011). Genetic algorithm based qos-aware service compositions in cloud computing. Dans *Proceedings of the 16th International Conference on Database Systems for Advanced Applications : Part II*, 321–334. Springer-Verlag.

Yi, S., Hao, Z., Qin, Z. et Li, Q. (2015). Fog computing : Platform and applications. Dans *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, 73–78.

Yu, T., Noghabi, S. A., Raindel, S., Liu, H., Padhye, J. et Sekar, V. (2016). Freeflow : High performance container networking. Dans *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, New York, NY, USA. ACM.

Zeng, H., Wang, B., Deng, W. et Zhang, W. (2017). Measurement and evaluation for docker container networking. Dans *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 105–108.

Zismer, A. (2016). Performance of docker overlay networks. *University of Amsterdam.*