UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PLACEMENT DES MICROSERVICES DANS LE CONTINUUM NUAGE-PÉRIPHÉRIE AVEC L'INFORMATIQUE EN RÉSEAU

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

SOUKAINA OULEDSIDI ALI

FÉVRIER 2023

UNIVERSITÉ DU QUÉBEC À MONTRÉAL
Service des bibliothèques

Avertissement

La diffusion de ce mémoire se fait dans le respect des droits de son auteur, qui a signé le formulaire *Autorisation de reproduire et de diffuser un travail de recherche de cycles supérieurs* (SDU-522 – Rév.04-2020). Cette autorisation stipule que «conformément à l'article 11 du Règlement no 8 des études de cycles supérieurs, [l'auteur] concède à l'Université du Québec à Montréal une licence non exclusive d'utilisation et de publication de la totalité ou d'une partie importante de [son] travail de recherche pour des fins pédagogiques et non commerciales. Plus précisément, [l'auteur] autorise l'Université du Québec à Montréal à reproduire, diffuser, prêter, distribuer ou vendre des copies de [son] travail de recherche à des fins non commerciales sur quelque support que ce soit, y compris l'Internet. Cette licence et cette autorisation n'entraînent pas une renonciation de [la] part [de l'auteur] à [ses] droits moraux ni à [ses] droits de propriété intellectuelle. Sauf entente contraire, [l'auteur] conserve la liberté de diffuser et de commercialiser ou non ce travail dont [il] possède un exemplaire.»

REMERCIEMENTS

Je voudrais dans un premier temps exprimer ma profonde reconnaissance à ma directrice de recherche, professeure Halima Elbiaze, et je tiens à la remercier pour son soutien moral et financier, sa disponibilité, et ses conseils et critiques qui ont guidé mes réflexions.

J'adresse mes sincères remerciements à tous les professeurs qui ont répondu à mes questions et m'ont donné beaucoup de conseils.

Je remercie chaque membre de ma petite famille pour son soutien inconditionnel et ses encouragements qui ont été d'une grande aide.

Je désire aussi remercier les professeurs de l'Université du Québec à Montréal, qui m'ont fourni les outils nécessaires à la réussite de mes études universitaires.

TABLE DES MATIÈRES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES v

LISTE DES FIGURES ix

LISTE DES TABLEAUX xi

RÉSUMÉ xii

CHAPITRE I INTRODUCTION 1

1.1 Mise en contexte 1

1.2 Motivation 2

1.3 Problématique 4

1.4 Méthodologie 7

1.5 Contribution 8

1.6 Organisation du mémoire 8

CHAPITRE II CONCEPTS PRÉLIMINAIRES 10

2.1 Introduction 10

2.2 Microservices 10

2.2.1 Architecture monolithique vs architecture de microservices . . 11

2.2.2 Principes de conception 14

2.2.3 Avantages 15

2.2.4 Défis de l'architecture de microservices dans le contexte de NFV 16

2.2.5 La migration d'une application monolithique vers un microservice 18

2.2.6 Modèles de déploiement de microservices 19

2.2.7 les anti-patrons 22

2.3 Les conditions d'une architecture de microservices 23

2.4 L'informatique en réseau (*In-network Computing*) 26

2.4.1 Définition 26

2.4.2 L'architecture de l'informatique en réseau 27

2.4.3 Les avantages de l'informatique en réseau 28

2.5 Conclusion 29

CHAPITRE III ÉTAT DE L'ART 31

3.1 Introduction 31

3.2 Le placement des entités de calcul 31

3.3 Le placement des microservices 35

3.4 Conclusion 39

CHAPITRE IV CAMP-INC : MÉCANISME DE PLACEMENT DES MICROSERVICES 40

4.1 Introduction 40

4.2 Le modèle du système 40

4.2.1 Description du cas d'utilisation *holopatient* 40

4.2.2 L'architecture du système 42

4.2.3 Composants et fonctionnement de CaMP-INC 43

4.3 Formulation mathématique de la problématique 45

4.3.1 Description du problème 45

4.3.2 Définition des variables 45

4.3.3 La programmation linéaire en nombres entiers 46

4.3.4 Complexité du problème 50

4.4 Algorithmes proposés 52

4.4.1 Gestionnaire de placement (*Placement Manager*) 52

4.4.2 Détecteur de défaillance (*Failure Detector*) 55

4.4.3 Gestionnaire de registre (*Registry Manager*) 55

4.5 Conclusion 56

CHAPITRE V ÉVALUATION DE PERFORMANCES 58

5.1 Introduction 58

5.2 Environnement de simulation 58
5.3 Topologie et paramètres de simulation 59
5.4 L'algorithme de comparaison 60
5.5 Résultats et discussions 61
5.5.1 Le coût total 61
5.5.2 Le temps d'exécution 62
5.5.3 Le coût de communication 63
5.5.4 Le nombre de noeuds utilisés pour le placement 64
5.5.5 La latence moyenne des requêtes de service 65
5.5.6 Le délai de communication moyen entre les microservices et les registres de services 66
5.6 Conclusion 67
CHAPITRE VI CONCLUSION 68

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

6G Sixième génération.

ADMD *Application Deployment using Microservice and Docker container* (Déploiement d'application en utilisant microservice et conteneur Docker).

API *Application Programming Interface* (Interface de programmation d'application).

AR *Augmented Reality* (Réalité augmentée).

CaMP-INC *Components-aware Microservices Placement for In-Network Computing Cloud-Edge Continuum*.

CPU *Central Processing Unit* (Central Processing Unit).

E/S Entrées/Sorties.

E2E *End to End* (Bout en bout).

EC *Edge Computing* (Informatique en périphérie).

eNFs *embedded Network Functions* (Fonctions réseau intégrées).

EPTA *Execution container Placement with Task assignment Algorithm*.

ETSI *European Telecommunications Standards Institute* (Institut européen des normes de télécommunications).

FASE *Flow-Aware Service Embedding*.

FD *Failure Detector* (Détecteur de défaillance).

GAP *Generalized Assignment Problem* (Problème d'affectation généralisé).

HTTP *HyperText Transfer Protocol* (Protocole de transfert hypertexte).

ILP *Integer Linear Programming* (Programmation linéaire en nombres entiers).

INC *In-Network Computing* (Informatique en réseau).

IoT *Internet of Things* (Internet des objets).

IP *Internet Protocol* (Protocole Internet).

LaECP *Latency-aware Edge-cloud Collaborative Placement*.

MEC *Multi-access Edge Computing* (Informatique en périphérie multiaccès).

NFV *Network Function Virtualization* (Virtualisation des fonctions réseau).

NP *Nondeterministic Polynomial* (Non déterministe polynomial).

PDP *Programmable Data Plane* (Plan de données programmable).

PIaFFE *A Place-as-you-go In-network Framework for Flexible Embedding of VNFs*.

PM *Placement Manager* (Gestionnaire de placement).

QoS *Quality of Service* (Qualité de service).

RAM *Random Access Memory* (Mémoire vive).

REST *REpresentational State Transfer* (Transfert d'état représentatif).

RM *Registry Manager* (Gestionnaire de registre).

RPC *Remote Procedure Call* (Appel de procédure à distance).

SAA *Sample Average Approximation* (Approximation moyenne d'échantillon).

SDN *Software Defined Networking* (Réseau définit par logiciel).

SF *Service Function* (Fonction de service).

SFC *Service Function Chain* (Chaînage de fonctions de service).

smartNIC *smart Network Interface Card* (Carte d'interface réseau intelligente).

SNR *Signal to Noise Ratio* (Rapport signal sur bruit).

SOA *Service-Oriented Architecture* (Architecture orientée service).

SR *Service Registry* (Registre de service).

VM *Virtual Machine* (Machine Virtuelle).

VNF *Virtual Network Function* (Fonction de réseau virtualisée).

VNFC *Virtual Network Function Component* (Composant de fonction de réseau virtualisée).

VR *Virtual Reality* (Réalité virtuelle).

WAN *Wide Area Network* (Réseau à large zone).

LISTE DES FIGURES

Figure Page

2.1 un exemple de flux monolithique dans une application *e-commerce* Gos et Zabierowski (2020) 12

2.2 un exemple de flux de microservices dans une application *e-commerce* Gos et Zabierowski (2020) 12

2.3 architecture monolithique vs architecture de microservices 13

2.4 modèle de plusieurs instances de service par hôte Richardson (2016) 19

2.5 modèle d'instance de service par machine virtuelle Richardson (2016) 20

2.6 modèle d'instance de service par conteneur Richardson (2016) . . 21

2.7 les microservices et leurs bases de données Stec (2022) 24

2.8 l'inscription des microservices auprès du registre de service Richardson (2015) 25

2.9 application avec microservices indépendants vs application avec microservices dépendants 26

2.10 architecture de l'informatique en réseau Ali *et al.* (2021) 28

4.1 l'architecture globale du système 43

4.2 modèle du système 44

5.1 la topologie du réseau 59

5.2 le coût total 61

5.3 le temps d'exécution 62

5.4 le coût de communication 64

5.5 le nombre de noeuds utilisés pour le placement 65

5.6 la latence moyenne des requêtes de service 66

5.7 le délai de communication moyen entre les microservices (MS) et les registres de services (RS) 67

LISTE DES TABLEAUX

Tableau Page

3.1 classement des travaux concernant le placement 35

4.1 tableau des notations 51

5.1 les paramètres de simulation 60

RÉSUMÉ

Les microservices sont une technologie prometteuse pour les réseaux futurs, et plusieurs efforts ont été consacrés pour placer de manière optimale les microservices dans les centres de données infonuagiques (*cloud data centers*). Cependant, le déploiement des microservices dans les serveurs de périphérie (*edge servers*) et les dispositifs du réseau (*network devices*) est plus coûteux que le déploiement dans les serveurs infonuagiques en termes de budget. De plus, plusieurs travaux ne considèrent pas les exigences principales de l'architecture de microservices, telles que le registre de service (*service registry*), la détection des défaillances et la base de données spécifique de chaque microservice.

Dans de ce mémoire, nous définissons quelques concepts tels que les microservices et les exigences de leur architecture ainsi que l'informatique en réseau. Par la suite, nous dressons un état de l'art sur le placement des entités de calcul et les microservices. Aussi, nous étudions le problème du placement des composants (c.-à-d. les microservices et leurs bases de données correspondantes) tout en tenant compte de la défaillance des nœuds physiques et de la distance aux registres de services. Nous proposons également un mécanisme de placement des composants pour le continuum nuage-périphérie (*cloud-edge continuum*) et l'informatique en réseau (*in-network computing*) nommé CaMP-INC (*Components-aware Microservices Placement for In-Network Computing Cloud-Edge Continuum*).

Nous formulons notre problème en utilisant la programmation linéaire en nombres entiers. L'objectif est de minimiser le coût. Comme le problème est $\mathcal{NP}$-difficile, nous proposons une solution heuristique. Les résultats numériques démontrent que notre solution proposée CaMP-INC réduit le coût total (c. -à-d., le coût de déploiement, de communication et d'exécution) de 15,8 % en moyenne et a une performance supérieure en matière de minimisation de la latence par rapport à une solution existante.

Mots clés : Placement des microservices, l'architecture de microservices, l'informatique en réseau, le continuum nuage-périphérie.

CHAPITRE I

INTRODUCTION

1.1 Mise en contexte

Compte tenu du nombre croissant d'appareils connectés à Internet et de diverses applications distribuées, y compris les applications pour l'analyse de données et l'apprentissage automatique, l'informatique en nuage (en anglais *cloud computing*) centralisé a récemment été critiqué par la communauté d'Internet Kulatunga *et al.* (2017). Cela est dû au fait que l'informatique en nuage a été initialement conçue pour des applications monolithiques, portant l'idée de les faire fonctionner sur des centres de données avec des ressources hautement disponibles.

Cependant, les nouvelles applications d'analyse de données et d'apprentissage automatique souffrent d'une dégradation des performances en raison des goulots d'étranglement du réseau. Pour résoudre ce défi, une nouvelle architecture réseau et informatique est nécessaire.

Récemment, avec l'apparition des commutateurs de réseau programmables (*programmable network switches*), l'informatique en réseau (INC) a reçu une grande attention de la part de l'industrie et du milieu universitaire. Étant donné que l'informatique en réseau effectue le calcul au sein du réseau, le traitement des paquets est effectué sur le chemin et, par conséquent, les latences imprévisibles

dans le chemin de communication pourraient être évitées.

Néanmoins, les commutateurs ont une mémoire sur puce limitée et pourraient rapidement manquer de mémoire si nous voulons y traiter une application monolithique. De telles applications ont souvent une élasticité limitée, alors que les décomposer nous permet d'avoir des services plus petits, plus ciblés et plus modulaires Kecskemeti *et al.* (2016).

C'est là que les microservices viennent en aide. Le style architectural des microservices décompose une application en une collection de services Krylovskiy *et al.* (2015a) et surmonte ainsi les inconvénients de l'architecture monolithique inélastique traditionnelle Liang et Lan (2021). Un aspect crucial de l'architecture de microservices est que les composantes du grand service se concentrent sur une seule tâche Mazzara *et al.* (2018), ce qui en fait un choix favorable pour les appareils aux ressources limitées tels que les commutateurs ou les cartes réseau intelligentes.

Nous considérons que la traduction du terme *in-network computing* en français est l'informatique en réseau et nous l'utilisons dans le reste de ce mémoire.

1.2 Motivation

Les réseaux traditionnels ont été conçus principalement pour effectuer la transmission des données. C'est pour cette raison que les dispositifs réseau comme les routeurs et les commutateurs conventionnels fournissent des fonctions de transfert de données et ne sont pas capables d'effectuer des calculs sur ces données. Néanmoins, selon Hu *et al.* (2021), les futurs réseaux 6G amélioreront considérablement les capacités de calcul et de transmission des nœuds du réseau. De cette manière, les données peuvent être traitées pendant le processus de transmission, si les ressources de calcul des nœuds du réseau peuvent être pleinement utilisées. Par

conséquent, la pression de traitement des données dans le système infonuagique et la consommation d'énergie vont être réduites. Également, les raisons principales pour lesquelles l'informatique en réseau est plus adaptée aux réseaux 6G que l'informatique en périphérie (EC) ou en nuage sont les suivantes :

— Les capacités des dispositifs réseau peuvent être utilisées pour accélérer le traitement des applications afin d'offrir une meilleure expérience aux utilisateurs finaux.
— Les dispositifs réseau récents peuvent exécuter des tâches informatiques complexes.
— La transmission des données peut être réduite et ainsi alléger le trafic.

D'un autre côté, les environnements hétérogènes et limités en ressources, notamment ceux de l'informatique en périphérie et en réseau nécessitent des architectures modulaires et distribuées comme l'architecture de microservices afin que les ressources des dispositifs limités puissent être utilisées pour les modules d'application critiques en latence et gourmands en bande passante tout en plaçant le reste des modules dans les ressources infonuagiques Pallewatta *et al.* (2022).

De plus, de nos jours, un nombre croissant d'entreprises informatiques et de fournisseurs d'applications développent des applications complexes à l'aide de techniques de l'architecture de microservices Deng *et al.* (2020). Ces applications peuvent bénéficier de cette architecture grâce aux caractéristiques des microservices Pallewatta *et al.* (2019). De la part de leur conception, les microservices sont faciles à conteneuriser, car ils sont faiblement couplés, indépendants et autosuffisants. Aussi, les microservices prennent en charge la mise à l'échelle verticale et horizontale. Dans la mise à l'échelle verticale, les ressources sont allouées en fonction de la charge, tandis que la mise à l'échelle horizontale est obtenue en répliquant plusieurs fois un seul microservice pour satisfaire la charge. Cette dernière peut considérablement améliorer les performances des applications déployées

dans les environnements hétérogènes où les nœuds sont limités en ressources.

Bien que les microservices soient une solution adaptée à différents environnements, leur déploiement doit être fait avec soin. Les fournisseurs d'applications peuvent louer de nombreux appareils et déployer de nombreuses instances de microservices afin d'offrir de meilleures expériences aux utilisateurs, mais le coût de location devient un grand défi Deng *et al.* (2020). Par la suite, ces fournisseurs ne seront pas satisfaits si les microservices sont déployés dans des appareils rarement utilisés par les utilisateurs. La société Flexera fle (2022), spécialisée dans les services infonuagiques, a constaté qu'environ 26 % des entreprises comptant au moins 1 000 employés dépensent plus de 6 millions de dollars par an dans le nuage public, mais 35 % de ces dépenses sont considérées comme du gaspillage. D'autre part, les ressources en périphérie et en réseau peuvent être tarifées plus cher que les ressources en nuage en raison des améliorations de la latence et de la bande passante, ce qui conduit de nombreuses politiques de placement existantes à envisager de minimiser le coût total Pallewatta *et al.* (2022).

1.3 Problématique

Avec l'évolution de l'Internet des objets, l'informatique en nuage a commencé à atteindre ses limites. Les objets tels que les capteurs nécessitent un traitement ultra rapide pour pouvoir réagir dans un délai acceptable. En outre, l'informatique en nuage a été considérée comme une solution viable pour traiter et stocker les données générées par les appareils intelligents. Cependant, transmettre une quantité importante de données des appareils IoT vers le nuage cause une congestion du réseau et une latence élevée étant donné que les centres de données infonuagiques se situent à plusieurs noeuds des sources de données Pallewatta *et al.* (2022). C'est pour cette raison que l'informatique en périphérie est née afin de rapprocher le

calcul et le stockage des utilisateurs finaux. En d'autres termes, l'informatique en périphérie fournit un calcul à proximité des appareils IoT qui aide à réduire la latence et la taille des données générées qui peuvent être classifiées comme *big data* Ali *et al.* (2020). Cependant, l'informatique en réseau, qui fait référence au calcul au sein du réseau, pourrait être plus efficace dans les scénarios de l'Internet des objets, par exemple, dans les scénarios critiques qui nécessitent un traitement en temps réel tels que les applications de réalité augmentée et réalité virtuelle AR/VR Scherb *et al.* (2018). Les dispositifs utilisés dans l'informatique en réseau tels que les routeurs, les commutateurs et les points d'accès installés dans l'infrastructure de communication ont des capacités de calcul et de stockage sous-utilisées et qui peuvent être suffisantes pour effectuer quelques traitements en plus de leur tâche habituelle de transfert et de routage des paquets de données Ali *et al.* (2020).

L'hétérogénéité des dispositifs dans un continuum nuage-périphérie avec l'informatique en réseau fait appel à l'architecture de microservice puisque les microservices peuvent être déployés dans des hôtes de tailles différentes Mazzara *et al.* (2018). L'architecture de microservice réduit considérablement l'effort impliqué dans la gestion d'une application en la décomposant en une collection de services Krylovskiy *et al.* (2015b), surmontant ainsi les lacunes des architectures monolithiques traditionnelles Liang et Lan (2021). Étant donné que les microservices sont des services autonomes, dédiés à l'exécution d'une tâche spécifique, cela en fait une excellente option pour l'informatique en réseau où le calcul est effectué dans des appareils à ressources limitées.

Bien que le placement des microservices ait été largement étudié au cours des dernières années dans le contexte de l'informatique en nuage, la plupart de ces travaux de recherche ne peuvent pas être appliqués à l'informatique en périphérie et en réseau en raison de ses caractéristiques distinctes (par exemple, limitations de ressources et hétérogénéité). Avec l'émergence de nouveaux types d'applications

et de nouveaux paradigmes (par exemple, les applications critiques à la latence), le besoin d'une politique de placement des microservices optimale et adaptative augmente. Une politique de placement de microservices détermine quels microservices sont déployés sur quels dispositifs pour atteindre les objectifs de performance à un coût acceptable.

Cependant, la dynamique et l'hétérogénéité du continuum nuage-périphérie avec l'informatique en réseau compliquent davantage le problème de placement des microservices. En fait, les serveurs en périphérie et les dispositifs du réseau ont tendance à avoir des ressources strictement limitées en ce qui concerne le traitement, le stockage et la mémoire RAM, la bande passante et l'énergie Filali *et al.* (2020).

De plus, l'architecture de microservices a plusieurs exigences pour assurer son bon fonctionnement :

1. La réplication des microservices.
2. Le registre de service.
3. Une base de données dédiée.
4. La résilience.
5. L'équilibrage de charge.

Bien que plusieurs efforts ont été déployés pour aborder le placement des microservices, quelques travaux comme Kaur *et al.* (2022) et Wang *et al.* (2020a) se concentrent principalement sur la satisfaction des exigences de performance et ne répondent pas aux exigences de l'architecture de microservices. Aussi, les méthodes de placement des microservices existantes se concentrent généralement sur ces derniers en tant que composants uniques et ignorent leurs bases de données et le registre de service. De plus, de nombreuses études se limitent au paradigme de l'informatique en nuage Ding *et al.* (2022), en périphérie ou en réseau et ignorent

la possibilité de les combiner.

1.4 Méthodologie

La méthodologie adoptée afin de réussir ce travail de recherche et d'élaborer ce mémoire comporte quatre étapes :

1. Revue de la littérature pour acquérir des connaissances approfondies sur l'architecture de microservice et l'informatique en réseau. Ensuite, l'élaboration d'un état de l'art sur les différentes méthodes de placement des microservices dans le but de trouver des lacunes et proposer une solution pour les combler.
2. Formulation mathématique du problème de placement des microservices qui prend en considération le placement des bases de données également. Cette formulation vise à minimiser le coût tout en satisfaisant certaines contraintes, notamment de délai et de capacité. La complexité du problème est ensuite analysée afin de déterminer sa faisabilité.
3. Proposition d'une heuristique pour le placement des microservices et de leurs bases de données correspondantes en tenant compte de la possibilité de défaillance des nœuds physiques et de la distance entre les microservices déployés et les registres de services existants.
4. Évaluation de la performance en comparant la formulation en programmation linéaire en nombres entiers, l'heuristique proposée et une solution existante dans la littérature. Le but étant d'observer le comportement de la solution proposée ainsi qu'évaluer l'impact de considérer certaines exigences de l'architecture de microservices.

1.5 Contribution

Dans ce mémoire, nous étudions le problème de placement des microservices dans le continuum nuage-périphérie avec l'informatique en réseau pour bénéficier d'un accès de faible latence aux nœuds de périphérie et de réseau et de la puissance de calcul élevée du nuage.

Nos objectifs de conception incluent :

1. Chaque microservice doit avoir un composant de base de données s'il utilise des données comme entrée ou stocke les résultats dans une base de données.
2. Un mécanisme de résilience pour détecter les nœuds défaillants et éviter de les utiliser pour le placement.
3. Garantir la communication entre les microservices et le registre de service sans violer le seuil de délai.

Nous proposons une solution pour le placement des microservices et leurs bases de données en tenant compte des nœuds défaillants et du registre de service. Cette solution est constituée de trois composantes :

1. Un gestionnaire de placement capable de placer les microservices et leurs bases de données tout en minimisant le coût.
2. Un détecteur de défaillance responsable de détecter les nœuds du réseau qui sont tombés en panne et d'informer le gestionnaire de placement.
3. Un gestionnaire de registre qui décide de placer un nouveau registre de service si les registres existants ne respectent pas un certain délai.

1.6 Organisation du mémoire

Le présent mémoire est organisé comme suit :

Premièrement, nous commençons dans le chapitre 2 par définir les concepts préliminaires nécessaires pour la compréhension de la suite du mémoire. Nous expliquons le concept de microservices en mettant le point sur ses principes de conception et ses avantages. Ensuite, nous détaillons les exigences d'une architecture de microservices et la notion de l'informatique en réseau. Deuxièmement, dans le chapitre 3 nous survolons des travaux qui ont été faits dans le cadre du placement des entités de calcul et des microservices tout en présentant une analyse critique de ces travaux. Troisièmement, le chapitre 4 détaille le problème et la solution proposée. Dans ce chapitre, nous décrivons tout d'abord notre modèle de système. Par la suite, nous analysons le problème et le formulons à l'aide de la programmation linéaire en nombres entiers puis nous démontrons que le problème est $\mathcal{NP}$-difficile. Dans le même chapitre, nous présentons les algorithmes proposés. Quatrièmement, dans la première partie du chapitre 5, nous décrivons l'environnement, la topologie et les paramètres de la simulation et l'algorithme de comparaison. Et dans la deuxième partie, nous présentons les résultats et nous les discutons. Finalement, le chapitre 6 conclut ce mémoire.

CHAPITRE II

CONCEPTS PRÉLIMINAIRES

2.1 Introduction

Dans ce chapitre, nous présentons quelques concepts préliminaires relatifs au travail de recherche effectué dans ce mémoire. Tout d'abord, nous expliquons la différence entre une architecture monolithique et une architecture de microservices. Ensuite, nous décrivons les principes de conception, les avantages, les défis et les modèles de déploiement des microservices et nous discutons la migration d'une application monolithique vers des microservices et les anti-patrons (*anti-patterns*). L'autre concept que nous abordons dans ce chapitre est les cinq exigences d'une architecture de microservices. Finalement, nous définissons le paradigme de l'informatique en réseau et nous expliquons son architecture et ses avantages.

2.2 Microservices

Au cours des dernières décennies, les modèles d'architecture logicielle ont évolué vers la conception modulaire, le couplage faible et la distribution. L'objectif principal de ce changement est de fournir la possibilité de réutiliser le code et d'augmenter la robustesse du service. Dans ce contexte, nous définissons les microservices comme des composants faiblement couplés et déployables de manière indépendante qui suivent l'approche architecturale infonuagique native Balalaie

et al. (2016). Ces composants communiquent à travers des interfaces de programmation d'application (API) légères et chacun exécute une fonction unique. Comme ils fonctionnent indépendamment les uns des autres, les microservices peuvent être mis à jour et mis à l'échelle avec moins de complexité que les applications monolithiques. Il existe également des langages de programmation émergents, tels que Jolie Jolie-lang.org (2022), qui conviennent au paradigme des microservices.

2.2.1 Architecture monolithique vs architecture de microservices

Une architecture monolithique encapsule toutes les fonctionnalités dans un seul composant exécutable, ce qui signifie que ses modules individuels ne peuvent pas être exécutés indépendamment. Bien que ce soit une bonne idée de commencer un projet en utilisant ce type d'architecture, car cela vous permet d'explorer à la fois la complexité du système ainsi que les limites de ses composants, à mesure que l'application se développe et que l'équipe s'élargit, les inconvénients de ce type d'architecture deviennent de plus en plus évidents Ponce *et al.* (2019). Il est souvent difficile de comprendre et de modifier l'application, c'est pourquoi le développement est généralement lent. Un autre inconvénient est que l'ensemble de l'application monolithique doit être reconstruit et déployé à chaque modification même si elle n'affecte qu'une petite partie de l'application. Aussi, il peut être difficile de mettre à l'échelle une application, car une architecture monolithique ne peut être mise à l'échelle qu'horizontalement.

La figure 2.1 montre un exemple de flux monolithique dans une application de commerce en ligne (*e-commerce*) dont tous les composants sont combinés en un seul programme.

D'un autre côté, l'architecture de microservices est une approche pour développer une application unique sous la forme d'une série de petits services qui s'exécutent

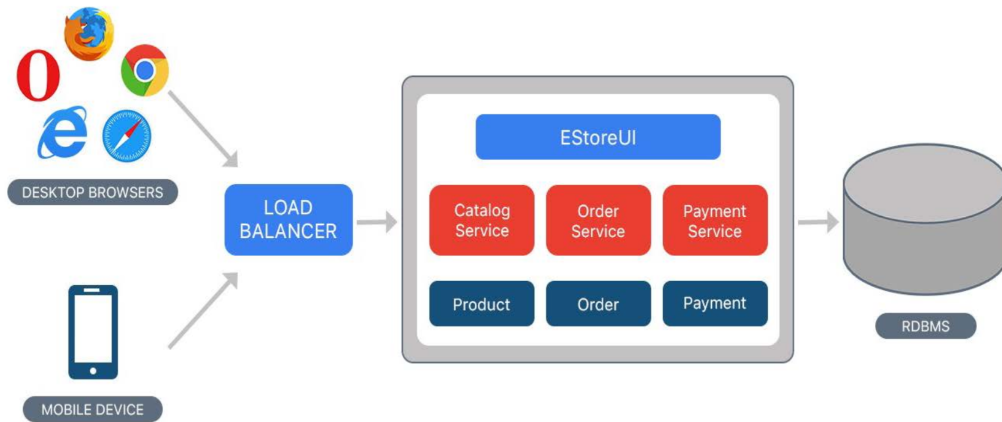

FIGURE 2.1 – un exemple de flux monolithique dans une application *e-commerce* Gos et Zabierowski (2020)

indépendamment et communiquent de manière légère. En plus d'être plus faciles à maintenir, les microservices sont également plus tolérants aux pannes, car la défaillance d'un service ne perturbera pas l'ensemble du système, contrairement à une architecture monolithique Ponce *et al.* (2019).

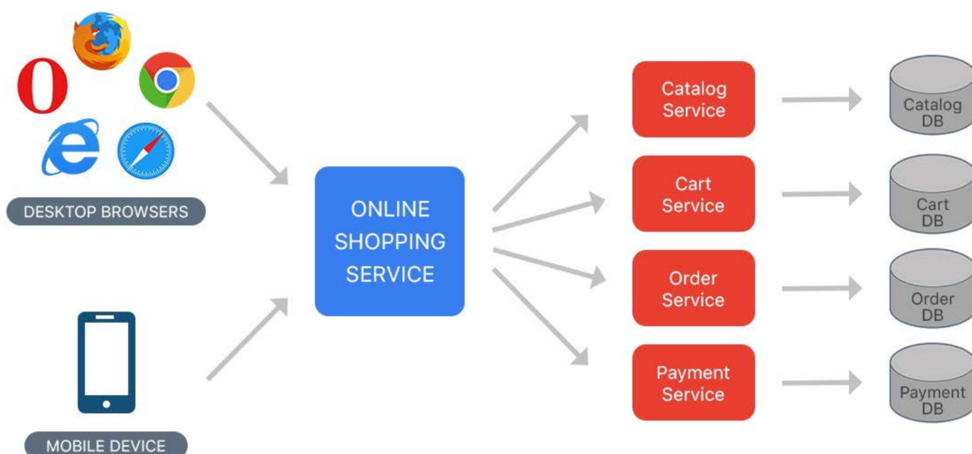

FIGURE 2.2 – un exemple de flux de microservices dans une application *e-commerce* Gos et Zabierowski (2020)

La figure 2.2 présente un exemple de flux de microservices dans la même appli-

cation de commerce en ligne présentée dans la figure précédente. Il y a quatre microservices de base qui fournissent les services de l'application et un autre service supplémentaire qui expose les fonctionnalités comme s'il s'agissait d'une seule composante.

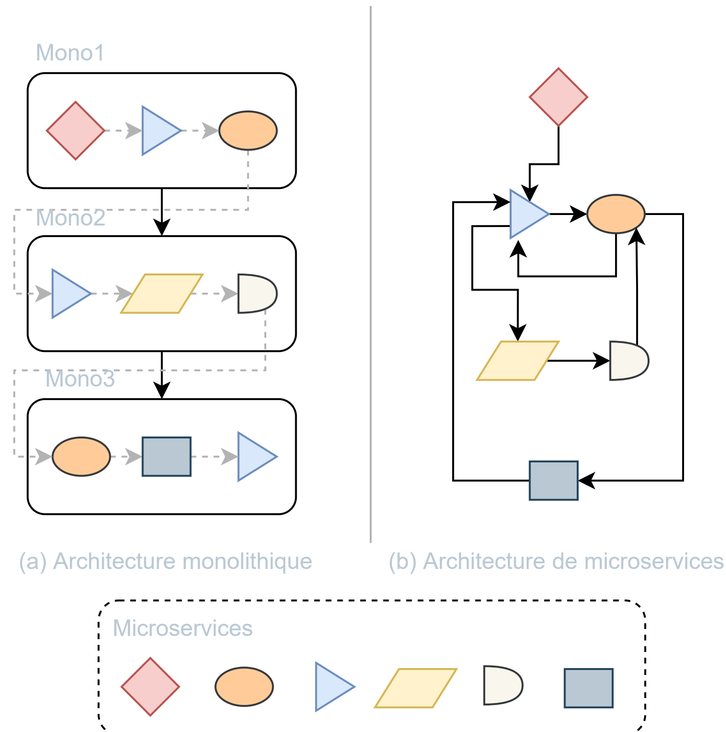

FIGURE 2.3 – architecture monolithique vs architecture de microservices

Comme le montre la figure 2.3, il convient de mentionner que le nombre total de microservices dans le réseau n'est pas nécessairement égal à la somme du nombre de microservices de toutes les applications monolithiques décomposés, car, compte tenu de la capacité de l'architecture de microservices à partager des fonctionnalités communes à différentes applications monolithiques, certains microservices peuvent être couramment utilisés par différentes applications. Par conséquent, dans le cas d'un déploiement antérieur sur le même serveur, le microservice n'a besoin que d'être mis à l'échelle. Cependant, le nombre total de connexions entre les micro-

services du réseau est égal à la somme des connexions de tous les microservices des applications monolithiques décomposés.

2.2.2 Principes de conception

Le paradigme des microservices repose sur quelques principes simples Dragoni *et al.* (2017) Mazzara *et al.* (2018) :

— Le contexte délimité : ce concept capture l'une des propriétés clés du paradigme des microservices. L'accent devrait être mis sur la capacité opérationnelle (*business capability*). Une capacité opérationnelle est créée en combinant des éléments fonctionnels connexes, qui sont ensuite déployés en tant que service.

— La taille : Le concept de taille est critique pour les microservices et présente des avantages majeurs en ce qui concerne le maintien et l'extension des services. L'architecture des microservices stipule que si un service est trop volumineux, il doit être décomposé en plusieurs services pour préserver la granularité et maintenir l'accent sur la capacité opérationnelle fournie.

— L'indépendance : Dans l'architecture de microservices, un service est indépendant d'un autre en matière de pile technologique, y compris son modèle de gestion de données et sa base de données. Cela encourage le faibe couplage (*loose coupling*) et la cohésion. De plus, seules les interfaces publiées des services peuvent être utilisées pour la communication entre eux.

Il convient de mentionner que la communication s'effectue généralement à travers une combinaison d'API REST, de diffusion d'événements et de courtiers de messages. L'API n'a pas besoin d'être dans un format particulier, mais le transfert d'état représentatif (REST) est populaire, en partie parce que sa lisibilité humaine et sa nature sans état le rendent utile pour les interfaces *Web*. Les autres formats d'API courants incluent gRPC et GraphQL.

2.2.3 Avantages

Dans cette partie, nous présentons les avantages de l'architecture de microservices par rapport à l'architecture monolithique. En fait, chacun de ces éléments peut également être considéré comme un défi ou un inconvénient dans l'architecture monolithique Mazzara *et al.* (2018) Krylovskiy *et al.* (2015b) Balalaie *et al.* (2015) Hasselbring et Steinacker (2017) Dragoni *et al.* (2017).

— La mise à l'échelle : l'une des principales motivations pour passer à une architecture de microservices est la mise à l'échelle. Les composants peuvent être mis à l'échelle indépendamment les uns des autres, ce qui réduit le gaspillage et les coûts associés à la mise à l'échelle d'applications entières lorsqu'une seule fonctionnalité fait face à une charge élevée Education (2021). La nature indépendante des déploiements dans les microservices ouvre de nouveaux modèles pour améliorer la mise à l'échelle et la robustesse des systèmes Newman (2019) . Cependant, l'effet de la mise à l'échelle sur les composants rarement utilisés est négligeable. Les composants qui sont utilisés par le plus grand nombre d'utilisateurs finaux doivent donc être considérés en priorité Frye (2020).

— Les technologies mixtes : l'isolement des processus permet de varier les choix technologiques, en mélangeant différents langages de programmation, styles de programmation, plates-formes de déploiement ou bases de données pour trouver la bonne combinaison Newman (2019) . Il s'agit d'une capacité très favorable pour les réseaux qui englobent une variété d'appareils avec différentes capacités matérielles. Chaque service peut être déployé sur un matériel qui correspond le mieux à ses besoins en ressources. C'est assez différent de l'architecture monolithique où des composants avec des besoins en ressources très différents doivent être déployés ensemble.

— La facilité de mise à jour : de nouvelles fonctionnalités peuvent être ajou-

tées sans toucher à l'ensemble de l'application et le code peut être mis à jour facilement sans affecter la fonctionnalité d'autres parties du système Education (2021).

- La facilité de développement : comme les services peuvent travailler en parallèle, il est possible d'amener plus de développeurs à se pencher sur un problème sans qu'ils ne se gênent mutuellement, en utilisant différentes piles et différents langages de programmation Newman (2019) . Il peut également être plus facile pour ces développeurs de comprendre leur partie du système, car ils peuvent concentrer leur attention sur une seule partie de celui-ci, alors que dans les applications monolithiques, l'architecture est vulnérable aux périls du couplage de la mise en œuvre et du déploiement.
- la minimisation du domaine de défaillance : l'architecture de microservice réduit le domaine de défaillance et améliore l'isolation des pannes, facilitant ainsi le dépannage et la haute disponibilité Roseboro et Reading (2016) . Dans ce cas, si un service se comporte mal, les autres services continuent de traiter les requêtes normalement, alors que dans une architecture monolithique, un composant qui se comporte mal peut faire tomber tout le système Richardson (2018) .
- L'agilité accrue : grâce aux microservices, les membres d'une équipe peuvent travailler sur des modules individuels. Chaque individu peut créer un module et le déployer indépendamment, réduisant ainsi les frictions opérationnelles de l'équipe et augmentant l'agilité de la livraison du logiciel Carrales (2021).

2.2.4 Défis de l'architecture de microservices dans le contexte de NFV

Pour les applications IoT, nous pouvons avoir une application composée de plusieurs éléments et ceux-ci sont implémentés sous forme de VNFs. Les fonctions

de réseau virtualisées VNFs sont des versions virtualisées des fonctions réseau traditionnelles (par exemple, un équilibreurs de charge ou un pare-feu).

En outre, les VNFs ont tendance à être monolithiques, ce qui peut compliquer leur déploiement dans les dispositifs réseau. Cependant, ils peuvent être décomposés en petites fonctions Chowdhury *et al.* (2019) nommées VNFC selon la description architecturale de VNF de l'*ETSI* ISG (2014). Ces petits composants peuvent être facilement intégrés dans des dispositifs à capacité limitée.

D'un autre côté, pour assurer une adoption plus large de l'architecture de virtualisation des fonctions réseau (NFV) par l'industrie des technologies de l'information et de la communication, NFV devrait surmonter les défis introduits par l'architecture de microservices. Les principaux défis incluent les problèmes suivants Hawilo *et al.* (2019) :

- La charge de communication : l'architecture de microservices est basée sur la création de petits composants indépendants qui sont chaînés à l'aide de différents protocoles *Web*. Cette approche peut entraîner des activités de réseau complexes qui sont difficiles à gérer et imposent rapidement un effet négatif sur la gestion du réseau. Les applications du monde réel peuvent être décomposées en centaines de microservices et en dizaines de milliers d'instances en cours d'exécution, comme c'est le cas avec Netflix et Twitter.
- Le déploiement des services : malgré les avantages que l'architecture de microservices apporte aux NFV, la gestion et le développement de VNFC deumeurent des défis complexes. Une tâche telle que le déploiement d'applications est triviale avec des applications monolithiques, mais avec des sous-tâches supplémentaires d'architecture de microservices, cela devient une tâche compliquée.
- La sécurité : l'architecture de microservices présente de nouveaux défis de

sécurité qui ne touche pas les applications monolithiques traditionnelles. Ces problèmes de sécurité sont exacerbés par l'utilisation intensive de divers canaux de communication qui créent davantage d'opportunités de détournement de données et d'interception pendant leur transit.
— La latence : il peut être difficile d'atteindre le même niveau de performances qu'avec une approche monolithique en raison de la latence entre les services, qui découle de la latence du réseau que les microservices utilisent pour communiquer entre eux.
— Le placement : Le critère utilisé pour placer les machines virtuelles (VMs) et les conteneurs sur des serveurs physiques est le principal contributeur à l'augmentation du trafic de signalisation entre les serveurs. L'allocation de VMs et de conteneurs est l'un des principaux facteurs qui affectent les exigences des applications de classe opérateur telles que la qualité de service (QoS), la fiabilité et la haute disponibilité. Avoir le placement optimal ou presque optimal est une étape indispensable pour satisfaire aux exigences de QoS.

2.2.5 La migration d'une application monolithique vers un microservice

La transition vers une architecture de microservices ne se révèle pas être une opération simple Newman (2021). Il faut choisir quelques services pour commencer la migration, apprendre de cette expérience et faire avancer cet apprentissage le plus rapidement possible. En créant et en publiant de nouveaux microservices de manière incrémentielle, il est plus facile de détecter et gérer les problèmes au fur et à mesure qu'ils surviennent. Lors de la migration, il faut assurer la coexistence de composants monolithiques et microservices. En outre, il existe l'option d'une méthode de secours, qui peut permettre au système de retomber dans un flux d'exécution alternatif si un microservice n'est pas disponible. La méthode de

secours peut être une méthode de suivi distincte ou un service hérité existant.

2.2.6 Modèles de déploiement de microservices

Une application à base de microservices peut comporter des dizaines, voire des centaines de services, et chacun est une mini application avec des exigences de déploiement uniques. Aussi, ces services doivent être déployés rapidement, de manière fiable et rentable. Il existe trois modèles de déploiement de microservices Richardson (2016) :

1. Plusieurs instances de service par hôte : dans ce modèle, plusieurs instances du service sont exécutées sur un ou plusieurs hôtes physiques ou virtuels. La figure 2.4 montre la structure de ce modèle.

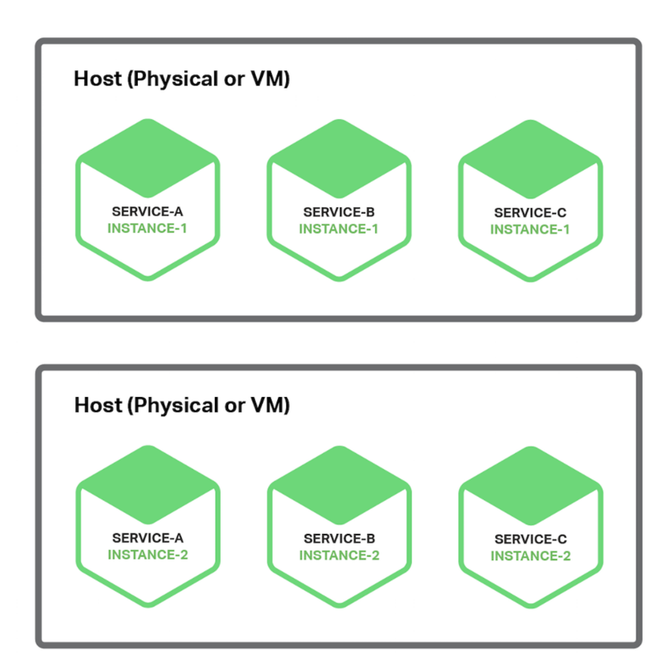

FIGURE 2.4 – modèle de plusieurs instances de service par hôte Richardson (2016)

Parmi les avantages de ce modèle, c'est qu'il est relativement facile de

déployer une instance de service et les ressources sont utilisées relativement de manière efficace. Le principal inconvénient est le manque d'isolement des instances de service.

2. Instance de service par machine virtuelle : Chaque service est encaplsulé en tant que machine virtuelle et chaque instance de service est une machine virtuelle lancée à partir d'une image VM. La figure suivante 2.5 représente la structure de ce modèle.

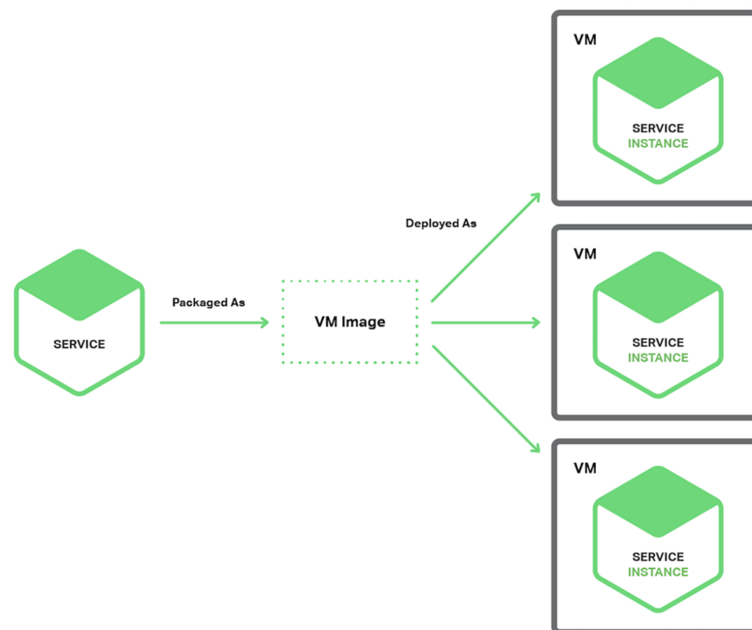

FIGURE 2.5 – modèle d'instance de service par machine virtuelle Richardson (2016)

Avec ce modèle, les instances de service sont isolées et la technologie qui implémente le service est encapsulée. Par contre, l'utilisation des ressources est inefficace et le déploiement de nouvelles versions des services est généralement lent.

3. Instance de service par conteneur : Chaque service est une image de conteneur. La figure 2.6 ci-dessous montre la structure de ce modèle.

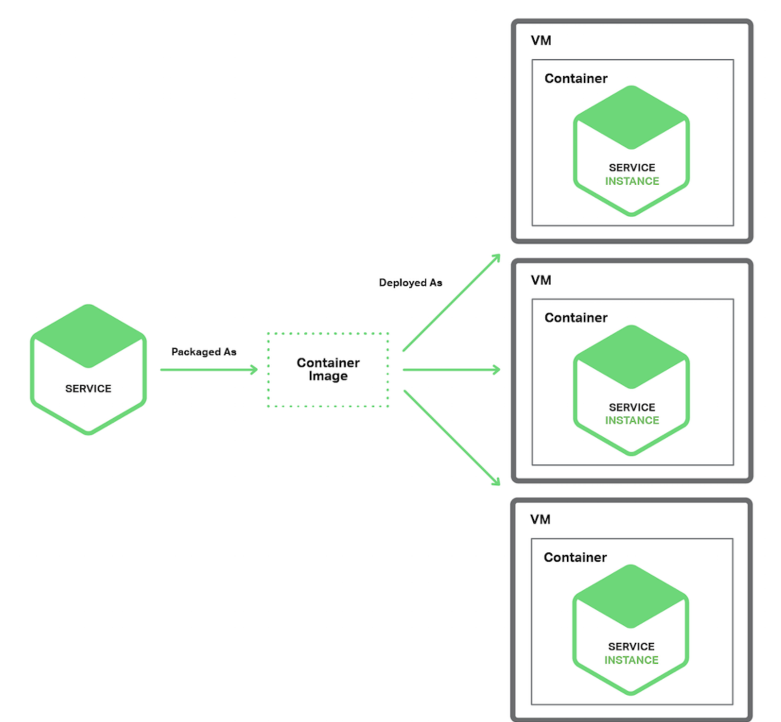

FIGURE 2.6 – modèle d'instance de service par conteneur Richardson (2016)

Ce modèle a plusieurs avantages. Premièrement, il garantit que les instances de service sont isolées les unes des autres. Deuxièmement, la consommation de ressources de chaque conteneur peut être facilement surveillée. Troisièmement, ce modèle encapsule la technologie utilisée pour l'implémentation des services. Finalement, les conteneurs sont une technologie légère et très rapide à construire. Mais aussi, ce modèle a quelques inconvénients principalement au niveau de la sécurité. Les conteneurs ne sont pas aussi sécurisés que les machines virtuelles, car ils se partagent le noyau du système d'exploitation hôte.

Dans de nombreux articles de recherche, les microservices sont souvent implémentés dans des conteneurs en raison de leur légèreté, de leurs performances et de leur déploiement rapide par rapport à la virtualisation

traditionnelle basée sur des machines virtuelles. De plus, la technologie des conteneurs peut être utilisée pour minimiser les effets de la virtualisation sur les ressources système et réduire son coût Kang *et al.* (2016).

2.2.7 les anti-patrons

Bien qu'il existe de nombreux patrons pour bien développer les microservices, il existe un nombre tout aussi important de patrons qui peuvent rapidement causer des problèmes à toute l'équipe de développement, appelés anti-patrons. Ici, nous mentionnons les anti-patrons les plus importants Education (2021).

- Commencer par l'architecture de microservices : comme première règle des microservices, il ne faut pas commencer par les microservices. Les microservices sont un moyen de gérer la complexité uniquement lorsque les applications deviennent trop volumineuses et trop lourdes pour être mises à jour et maintenues facilement.
- Concevoir des microservices sans *DevOps* ni services infonuagiques : créer des microservices signifie créer des systèmes distribués. Tenter de faire des microservices sans une automatisation appropriée du déploiement et de la surveillance ou des services infonuagiques gérés pour prendre en charge l'infrastructure hétérogène, engendre beaucoup de problèmes inutiles.
- Diviser un service en plusieurs microservices trop petits : si vous allez trop loin avec le "micro" dans les microservices, les coûts généraux et la complexité l'emportent sur les gains globaux. Il est préférable de se pencher sur les services plus importants et de ne les découper que lorsqu'il devient difficile et lent de déployer des modifications, ou que différentes parties du service ont des exigences de charge/d'échelle différentes.
- Transformer les microservices en SOA : les projets de microservices im-

pliquent généralement la réusinage d'une application afin qu'elle soit plus facile à gérer, tandis que la SOA vise à modifier la façon dont les services informatiques fonctionnent à l'échelle de l'entreprise. Un projet de microservices qui se transforme en projet SOA risque de céder sous son propre poids.

- Construire de nombreux codes et services personnalisés : les premiers pionniers de l'architecture de microservices, comme Netflix, construisent et gèrent une sorte d'architecture parfaite avec beaucoup de code personnalisé et de services inutiles pour l'application moyenne. Pour les applications moyennes, il est préférable d'utiliser autant d'outils prêts à l'emploi que possible.

2.3 Les conditions d'une architecture de microservices

Le succès d'une architecture de microservices implique cinq exigences principales :

1. La réplication des microservices : Pour assurer la réplication des microservices, un mécanisme doit être défini pour permettre la mise à l'échelle des microservices sur différents nœuds en fonction des demandes entrantes Pallewatta *et al.* (2019) Liu *et al.* (2019) Hawilo *et al.* (2019) Sampaio *et al.* (2019) Kakivaya *et al.* (2018).
2. Les bases de données : Chaque microservice doit avoir sa base de données dédiée afin d'être découplé des autres microservices Messina *et al.* (2016). Cependant, les microservices n'ont pas à être déployés dans le même noeud que leurs bases de données, ce qui permettra de les dupliquer indépendamment. Comme le montre la figure 2.7, les microservices communiquent avec leurs bases de données à l'aide d'opérations de lecture/écriture et ces opérations peuvent être effectuées à l'aide de requêtes HTTP GET/POST. La

fréquence de l'échange entre les deux dépend du nombre d'opérations de lecture/écriture qui doivent être effectuées par les microservices. Une base de données peut être un ensemble de tables privées par microservice, un schéma par microservice ou un serveur de base de données par microservice.

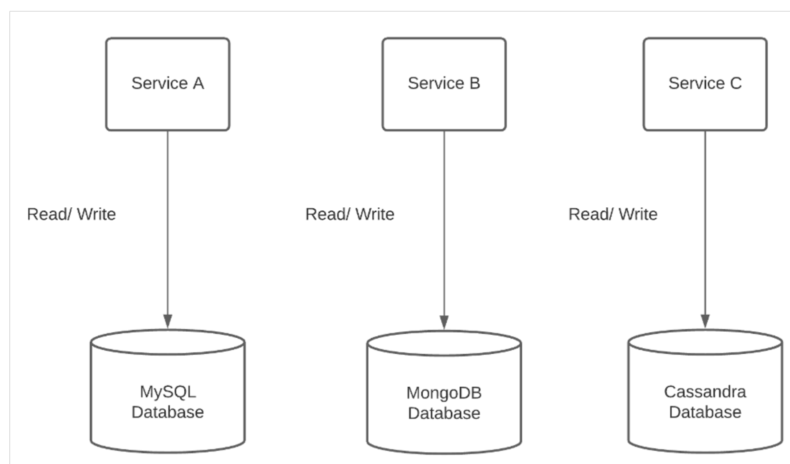

FIGURE 2.7 – les microservices et leurs bases de données Stec (2022)

3. Le registre de service (SR) : Un élément important de l'architecture de microservices est le registre de service Pallewatta *et al.* (2019) Hawilo *et al.* (2019) Messina *et al.* (2016) qui permet la découverte de services. Le registre de service est une base de données contenant l'emplacement des microservices disponibles. Lorsqu'un nouveau microservice est déployé, il doit d'abord s'inscrire (c-à-d. envoyer son adresse IP et numéro de port) auprès de ce registre de service (comme le montre la figure 2.8).
4. La résilience : La nature distribuée des microservices les rend plus tolérants aux pannes et résilients Hawilo *et al.* (2019) Kakivaya *et al.* (2018). De plus, pour renforcer cette résilience, l'architecture de microservices devrait être dotée d'un mécanisme capable de détecter les défaillances des nœuds du réseau afin d'éviter le placement dans ces nœuds-là. Si nous prenons l'exemple de la figure 2.9, nous avons deux applications, une pour le

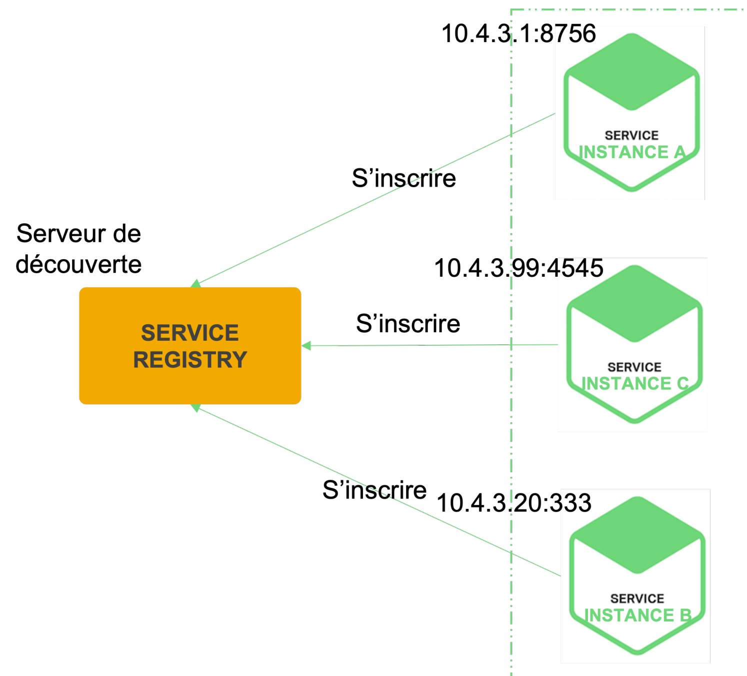

FIGURE 2.8 – l'inscription des microservices auprès du registre de service Richardson (2015)

partage d'images et une autre pour le traitement des données audio. Dans la première application, les microservices sont indépendants. Dans ce cas, si le microservice responsable de télécharger l'image tombe en panne, les autres microservices (c.-à-d. éditer et afficher l'image) restent fonctionnels. Par conséquent, les utilisateurs peuvent quand même afficher et éditer leurs images. D'un autre côté, nous avons l'application de traitement des données audio qui représente un cas particulier d'une application où les microservices sont dépendants et leur exécution doit être faite suivant un ordre précis. Si le microservice qui convertit l'analogique au numérique tombe en panne, alors, le microservice qui traite les données ne peut pas fonctionner et par la suite celui qui convertit le numérique en analogue n'aura pas des données traitées pour faire la conversion. Dans ce cas, une erreur survenue dans un microservice peut affecter des parties du système. D'où l'intérêt

de la détection des défaillances.

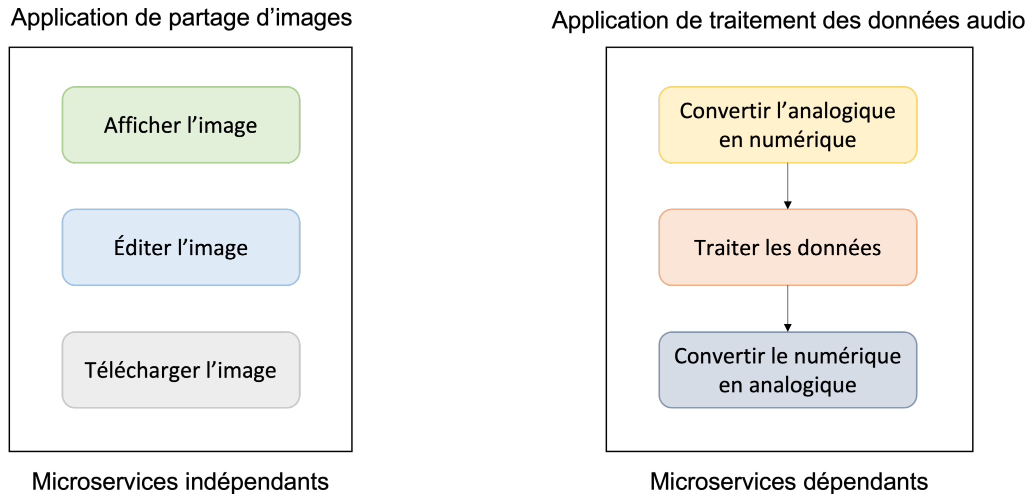

FIGURE 2.9 – application avec microservices indépendants vs application avec microservices dépendants

5. l'équilibrage de charge : un équilibreur de charge est nécessaire pour équilibrer la charge des requêtes entre les instances du même microservice Kakivaya *et al.* (2018) Pallewatta *et al.* (2019) Liu *et al.* (2019).

2.4 L'informatique en réseau (*In-network Computing*)

2.4.1 Définition

L'informatique en réseau comme son nom l'indique fait référence à l'exécution de programmes au sein des dispositifs réseau tels que les commutateurs et les routeurs. Elle utilise les dispositifs du réseau déjà existants et qui sont principalement dédiés pour le transfert du trafic. Ceci implique qu'aucun coût ni aucun espace supplémentaire ne sont nécessaires pour le déploiement de cette technologie. Aussi, l'informatique en réseau permet d'alléger le trafic et de réduire la latence étant donné que le traitement se fait à proximité des utilisateurs finaux

Zilberman (2019). L'informatique en réseau peut être efficace dans les scénarios de l'Internet des objets, par exemple, dans un réseau mobile Scherb *et al.* (2018) ou dans des scénarios qui nécessitent un traitement en temps réel comme les applications AR/VR. Aussi, des travaux comme Sifalakis *et al.* (2014) et Król et Psaras (2017) ont montré l'importance de l'exécution des fonctions au sein du réseau, en particulier lorsqu'il s'agit d'applications qui sont sensibles au délai et gourmandes en bande passante. Le traitement dans le réseau peut se faire par l'intermédiaire d'un langage de programmation dédié (par exemple, P4 Bosshart *et al.* (2014)). Ce langage est utilisé pour définir des traitements qui peuvent être utilisés par les périphériques réseau.

2.4.2 L'architecture de l'informatique en réseau

La figure 2.10 décrit une architecture à 3 niveaux Ali *et al.* (2021). Le niveau 1 comprend les appareils IoT, tels que des capteurs ou des appareils intelligents (par exemple, un téléphone intelligent ou une montre intelligente). Ces appareils envoient des requêtes pour récupérer des données ou exécuter des services. Le niveau 2 comprend des serveurs de périphérie responsables de répondre aux demandes des utilisateurs finaux. Le niveau 3 est le nuage et est situé à plusieurs sauts du niveau 1. Le nuage dispose de suffisamment de ressources et d'une capacité de stockage très élevée, mais il requiert un temps important pour répondre aux demandes des utilisateurs finaux. L'informatique en réseau peut être effectuée dans le réseau d'infrastructure par des dispositifs de réseau ou dans le réseau d'accès sans fil par les points d'accès. Les dispositifs réseau tels que les routeurs peuvent traiter les requêtes nécessitant des ressources limitées et renvoyer les résultats en peu de temps.

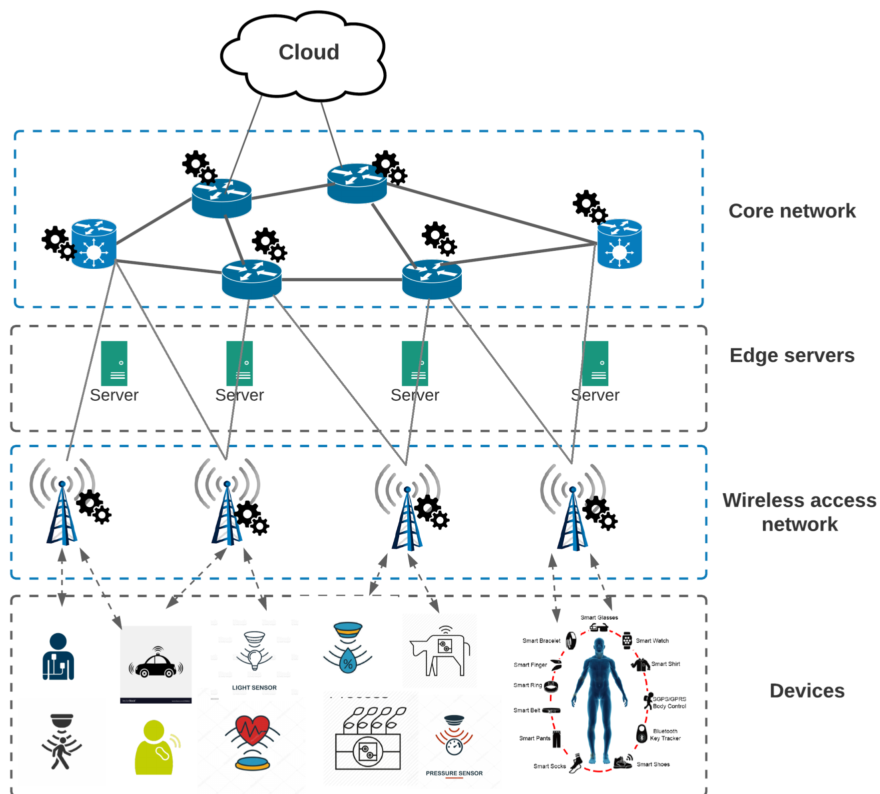

FIGURE 2.10 – architecture de l'informatique en réseau Ali *et al.* (2021)

2.4.3 Les avantages de l'informatique en réseau

L'informatique en réseau a plusieurs avantages Hu *et al.* (2021). Parmi lesquels nous avons :

— Une faible latence : L'informatique en réseau repose sur le concept selon lequel les données d'application peuvent être traitées pendant la transmission, réduisant ainsi les délais de traitement globaux.

— Un haut débit : Le traitement des paquets de données par les dispositifs réseau programmables peut atteindre 10 milliards paquets par seconde lorsqu'ils sont basés sur des ASICs programmables de hautes performances. Aussi, les performances des applications implémentées à l'aide de l'infor-

matique en réseau sont 10 000 fois meilleures que celles implémentées sur un hôte.

- Un taux d'efficacité énergétique élevé : Il est possible de réduire le taux d'inactivité du dispositif réseau lorsque le déchargement du traitement des applications est fait sur la couche réseau, et davantage de traitement de données peut être effectué avec la même consommation d'énergie.
- Une réduction des données transmises : En plus de réduire la consommation énergétique globale du centre de données, la quantité de données est progressivement réduite au cours du processus de transmission.

Dans le cadre de nos travaux de recherche Ali *et al.* (2021) , nous avons mis en avant les avantages du traitement au sein du réseau relativement à un traitement en périphérie à travers un cas d'utilisation. Ce dernier consiste à détecter les personnes suspectées d'être malades du Covid-19, tout au long d'un corridor de débarquement dans un aéroport. Le principe consiste à récolter les sons, les analyser à la recherche de toux infectieuses et enfin identifier la personne suspectée. Lors de l'évaluation des performances d'une architecture avec le traitement au sein du réseau, nous avons exécuté des fonctions de filtrage de données dans les nœuds du réseau qui sont les plus proches des microphones et des caméras de surveillance. Les résultats ont montré qu'avec l'informatique en réseau le traitement des paquets se fait en 2.2 secondes en moyenne contrairement au traitement dans les nœuds en périphérie où le délai excède 11 secondes. Aussi, nous avons montré que nous pouvons gagner presque 22.5 Go dans les serveurs en périphérie si nous utilisons le paradigme de l'informatique en réseau.

2.5 Conclusion

Dans ce chapitre, nous avons présenté deux concepts qui représentent les piliers de notre travail de recherche. Le premier concept est celui des microservices qui

représentent une technologie émergente qui a beaucoup de succès dans l'industrie et le milieu universitaire. L'architecture de microservices a été également adoptée par plusieurs compagnies, notamment Netflix et Uber. L'autre concept est l'informatique en réseau qui représente un paradigme très adapté aux réseaux futurs. Dans le chapitre qui suit, nous présentons un état de l'art.

CHAPITRE III

ÉTAT DE L'ART

3.1 Introduction

Ce chapitre présente l'état de l'art sur des travaux qui ont été déjà faits concernant le placement. Nous présentons, en premier lieu, une revue de littérature sur le placement des entités de calcul telles que les machines virtuelles, conteneurs, fonctions et tâches. En deuxième lieu, nous survolons les travaux faits pour traiter le placement des microservices en particulier.

3.2 Le placement des entités de calcul

Le placement des entités de calcul a été largement étudié dans les environnements infonuagiques dans le but de distribuer efficacement les services aux centres de données en fonction de leurs besoins en ce qui concerne la latence, le stockage ou le calcul Steiner *et al.* (2012). Cependant, les techniques développées pour les environnements infonuagiques ne sont pas applicables aux environnements hétérogènes et dynamiques de l'informatique en périphérie. De même, les mécanismes de placement proposés pour l'informatique en périphérie et dans le nuage peuvent ne pas être appropriés pour l'informatique en réseau compte tenu des ressources et capacités de calcul considérablement limitées des dispositifs réseau Lia *et al.* (2021) Cooke et Fahmy (2020).

Les applications de l'informatique en réseau et en périphérie peuvent être d'une grande variété incluant les applications AR/VR, les jeux en ligne et la robotique autonome, par conséquent, l'orchestrateur doit être équipé d'une stratégie de placement adaptative qui permet des décisions efficaces. Les algorithmes de placement doivent tenir compte de plusieurs défis tels que Sonkoly *et al.* (2021) :

1. L'hétérogénéité des entités de calcul incluant les machines virtuelles, les conteneurs, les fonctions et les tâches
2. La variété des équipements pouvant héberger ces entités (par exemple, serveurs, stations de base, points d'accès)
3. La dynamique des conditions du réseau (par exemple, la mobilité ou l'immobilité)
4. Les différentes exigences des utilisateurs finaux.

Ces dernières années, de nombreux efforts ont été déployés pour résoudre ces problèmes, allant de simples algorithmes et heuristiques à des techniques d'intelligence artificielle.

Placer une tâche informatique dans un continuum nuage-périphérie avec l'informatique en réseau nécessite deux composants essentiels :

1. Un nœud exécuteur avec des ressources de traitement disponibles.
2. Des données d'entrée à transférer de la source au nœud exécuteur sélectionné pour le traitement.

Dans les environnements informatiques hautement distribués, où les composants d'application consomment des ressources de calcul, de stockage et de mise en réseau, la tâche et les algorithmes de placement correspondants ont un impact considérable sur les performances. Les stratégies de placement font face à deux défis principaux. Premièrement, une grande quantité d'échange de données entre les nœuds d'exécution de tâches nécessite la mise à disposition d'une bande pas-

sante appropriée. Deuxièmement, les délais d'exécution des tâches sont intrinsèquement liés aux délais de traitement de bout en bout imposés par l'emplacement des nœuds exécuteurs de tâches.

De même, le chaînage de fonctions de service (SFC) est reconnu comme une technique essentielle pour connecter une séquence de fonctions de service (SF) basées sur SDN/NFV Bhamare *et al.* (2016). L'obtention de services à faible latence exige de réduire le temps d'exécution du SFC grâce à l'accélération du traitement des fonctions de service. La reconfigurabilité du plan de données et les performances de traitement des paquets à débit de ligne (*line rate*) activées par le plan de données programmable (PDP) sont essentielles pour atteindre cet objectif. L'utilisation de SFC dans un plan de données programmable peut se faire de deux manières Lee *et al.* (2021). Premièrement, une approche de fonction de service redondante qui permet le traitement facile des SF ordonnées qui composent un SFC est discutée dans Chen *et al.* (2019). Dans cette approche, les tables des SF sont intégrées de manière redondante dans le commutateur de plan de données programmable afin de satisfaire l'ordre de traitement du SFC au débit de ligne. Cependant, les ressources limitées des dispositifs de plan de données programmables peuvent ne pas permettre efficacement la redondance des fonctions de service intégrées. Deuxièmement, les auteurs de Lee *et al.* (2019); Wu *et al.* (2019) ont proposé l'approche de recirculation lorsque l'ordre des fonctions SFC est différent de celui des SF embarquées. Un paquet est remis en circulation de la sortie vers le port d'entrée. Cependant, il est difficile de garantir les performances de débit de ligne en raison de la recirculation. Lee *et al.* (2021) propose une fonction d'intégration de service sensible au flux (FASE) qui équilibre ces deux approches. Il combine la redondance des fonctions de service et l'approche de recirculation pour trouver l'intégration optimale de la fonction de service qui minimise le temps d'exécution du SFC tandis que les ressources de commutation de plan de données programmables sont

utilisées efficacement.

D'un point de vue architectural, les VNFs ont tendance à être monolithiques tels les équilibreurs de charge et les optimiseurs WAN, ce qui complique leur déploiement dans les dispositifs réseau. Comme l'informatique en réseau dans le continuum *Cloud-Edge-Mist* englobe une infrastructure hétérogène avec des hôtes et des dispositifs réseau de tailles différentes, dont beaucoup ont des ressources limitées telles que des commutateurs ou des smartNIC, l'architecture de micro-service offre de nombreuses caractéristiques attrayantes telles l'hétérogénéité et l'évolutivité qui s'adaptent au déploiement de services dans un tel environnement. Les applications peuvent être décomposées en fonctionnalités telles que l'analyse d'en-tête de paquet et la classification de paquet Chowdhury *et al.* (2019). Ces petits composants peuvent être facilement intégrés dans des dispositifs réseau avec une capacité limitée.

Mafioletti *et al.* (2020) ont proposé un framework, nommé PIaFFE (*A Place-as-you-go In-network Framework for Flexible Embedding of VNFs*), pour le placement des VNFs en utilisant le traitement en réseau. Dans ce travail, la logique des VNFs est décomposée en fonctions de réseau embarquées eNFs pour identifier les fonctions qui se chevauchent et les placer efficacement dans des dispositifs programmables en réseau. eNFs sont des VNFs compacts et légers qui utilisent un langage de programmation de haut niveau et s'intègrent dans des smartNICs. De plus, Chen *et al.* (2021) propose LightNF, un système qui permet de décharger les fonctions réseau dans des commutateurs programmables. Le système proposé dans LightNF est équipé d'un analyseur qui examine les caractéristiques et les options de déchargement des fonctions réseau et les ressources consommées par chaque fonction. Ensuite, sur la base de cette analyse, il construit un cadre d'optimisation qui décide du placement des SFC.

Article	**Entités de calcul**	**Paradigme informatique**		**Dépendance des entités**		**Métriques de performance**					
		INC	**EC**	**avec CXN**	**sans CXN**	**Latence**	**Débit**	**Bande passante**	**Énergie**	**UR**	**Coût**
Chen *et al.* (2021)	Fonctions réseau	✓		✓		✓	✓	✓			
Cooke et Fahmy (2020)	Tâches et ressources informatiques	✓	✓	✓		✓	✓	✓	✓		✓
Lee *et al.* (2021)	Fonctions de service	✓		✓		✓					
Lia *et al.* (2021)	Tâches	✓	✓		✓	✓				✓	
Mafioletti *et al.* (2020)	Fonctions réseau embarquées	✓		✓		✓	✓			✓	

INC : informatique en réseau, EC : informatique en périphérie, CXN : connexion, UR : Utilisation des ressources.

TABLEAU 3.1 – classement des travaux concernant le placement

Le tableau 3.1 classe les articles de placement selon quatre critères. Le premier critère en question concerne les unités de calcul. Il existe des blocs de construction fonctionnels des fonctions réseau tels que les pare-feux et les équilibreurs de charge. Le concept de fonction de service est principalement utilisé dans la chaîne de fonctions de service, où les fonctions de service sont liées pour former un service répondant à une demande spécifique. Une tâche est une unité qui se concentre sur une seule action. Enfin, les fonctions de réseau embarquées sont des composants légers et petits d'une VNF. Le deuxième critère est le paradigme informatique de l'informatique en réseau ou en périphérie considéré dans chaque article. Le troisième critère est la dépendance des entités qui fait référence à la connexion (par exemple, SFC) ou à la non-connexion (par exemple, un seul composant) entre les composants. Le dernier critère concerne les métriques de performance objectives prises en compte dans le placement des entités de calcul.

3.3 Le placement des microservices

De nombreux efforts de recherche ont été déployés pour étudier le placement des microservices. Zhijun Ding et al. Ding *et al.* (2022) s'attaquent aux limites des travaux existants en tenant compte de la concurrence dynamique et de la disponibilité des microservices ainsi que du problème de la dépendance partagée des bibliothèques entre plusieurs instances de microservices. À cette fin, les auteurs proposent un modèle non linéaire entier de placement de microservices pour *Kubernetes* dans le but de minimiser les coûts. Pour résoudre rapidement le modèle

de placement, un algorithme génétique amélioré est proposé pour calculer la solution optimale approximative. Le modèle proposé peut optimiser le schéma de placement de *Kubernetes* et réduire les coûts de placement d'applications à grande échelle. En outre, il peut répondre aux exigences de performances d'une certaine plage de charges de requêtes. Cependant, les ressources et les coûts sont gaspillés lorsque les applications de microservice font face à une petite charge de demandes et ne peuvent pas s'adapter à des charges de demandes soudaines.

Dans le but de minimiser la latence de service de bout en bout, Kiranpreet Kaur et al. Kaur *et al.* (2022) présentent une nouvelle stratégie de placement des microservices tenant compte de la composition du service interne, notamment la communication entre microservices. Le problème de placement est formulé comme un programme linéaire en nombres entiers, et une combinant deux heuristiques (un algorithme glouton et un algorithme génétique) est proposée. La solution réduit considérablement le temps d'exécution de l'algorithme de placement. De plus, l'algorithme glouton rapide fournit une allocation initiale de services, qui est ensuite améliorée par un algorithme génétique avancé ; ce dernier restructure les microservices par des mutations et des croisements afin de réduire la latence globale, tout en préservant le placement des microservices. Néanmoins, les auteurs considèrent une configuration statique au lieu de considérer l'arrivée et le départ aléatoires des services dans le système.

Yimeng Wang et al. Wang *et al.* (2020b) se sont concentré sur l'étude des méthodes d'estimation de la latence des services de bout en bout (E2E) et de son rôle dans l'amélioration de l'efficacité du placement des microservices. Les auteurs ont proposé un algorithme de placement collaboratif *edge-cloud* (LaECP) sensible à la latence qui place gloutonnement les microservices sur les appareils périphériques dans le but de réduire la latence globale. Ils ont utilisé un modèle d'apprentissage automatique (XGBoost) pour créer un estimateur de latence de service de bout

en bout basé sur les données au lieu d'utiliser des estimateurs de latence mathématiques traditionnels. Cet estimateur de latence plus précis a ensuite été intégré à leur algorithme de placement (XGB-LaECP) pour prouver son impact sur la réduction de la latence globale. Bien que la méthode de filtrage de placement heuristique pour le placement des services soit pertinente, elle présente encore certaines limites liées à l'adaptation rapide aux changements et à l'amélioration continue de leur estimateur. En effet, leur heuristique de placement des services est statique et ne tient pas compte de la dynamique des services.

Xili Wan et al. Wan *et al.* (2018) ont étudié le placement d'applications basées sur des microservices dans des centres de données infonuagiques à l'aide de conteneurs *Docker*. L'objectif des auteurs est de minimiser le coût de déploiement ainsi que le coût opérationnel tout en satisfaisant les exigences de délai. Ils ont d'abord proposé le déploiement d'applications à l'aide de microservices et *Docker* (*ADMD framework*) dans lequel le contrôleur de microservices ajuste dynamiquement les ressources affectées à chaque application. Ensuite, un algorithme efficace nommé EPTA est proposé afin de décider du placement des conteneurs d'exécution (c.-à-d., des conteneurs contenant des microservices). L'algorithme proposé est utilisé pour la comparaison dans le chapitre 5.

D'autre part, les dispositifs de réseau sans fil sont limités en matière de calcul et de stockage. Ainsi, la conteneurisation des microservices légers est essentielle Ali *et al.* (2020). L'un des défis consiste à mapper chaque conteneur sur le nœud approprié sans perturber ses fonctionnalités de base c.-à-d., le routage et le transfert de paquets. À cette fin, il est nécessaire d'étudier les caractéristiques sous-jacentes du réseau et les capacités de calcul résiduelles telles que le nombre de cycles CPU, le stockage, la RAM et la puissance de la batterie. De nombreuses caractéristiques de réseau sans fil sous-jacentes doivent être prises en compte pour mapper les microservices en tant que conteneurs sur les nœuds du réseau brouillard (*fog net-*

work). Ces paramètres qui dépendent de la nature dynamique des réseaux sans fil sont : le rapport signal sur bruit (SNR), le coût du chemin entre la source et la destination, le nombre de stations associées, les erreurs de réception et de transmission, le taux de perte de paquets et la vitesse de transmission. Enfin, la décision de placement pour les microservices devrait optimiser les métriques susmentionnées.

Zhao *et al.* (2020) proposent une approche de placement de redondance pour les applications basées sur des microservices dans l'informatique de périphérie à accès multiple (*Multi-access Edge Computing* ou MEC). Dans le but de minimiser la latence globale et d'assurer une haute disponibilité, ils ont envisagé la redondance dans le placement des microservices. Ils ont développé un cadre d'application basé sur l'approximation moyenne d'échantillons (SAA) qui utilise un algorithme génétique pour placer de manière redondante des instances de microservice dans une périphérie distribuée. Des points importants ont été pris en considération tels que la possibilité pour plusieurs microservices d'être candidats pour la même tâche, ainsi que la récupération d'informations située à plusieurs sauts du demandeur. Cependant, cette solution présente trois limitations principales liées à la dynamique de l'environnement et au manque de généralité pour les autres services. En fait, la solution est limitée aux applications avec une séquence linéaire de microservices c.-à-d., des applications sans ramification dans leur flux de contrôle. Et bien que leur cadre d'application proposé puisse être exécuté périodiquement, il est toujours hors ligne et n'est pas flexible aux changements rapides.

Tous les travaux susmentionnés résolvent le problème de placement des microservices de différentes manières et avec différentes technologies. Néanmoins, plusieurs d'entre eux ne considèrent qu'une seule des exigences de l'architecture de microservices mentionnées précédemment dans le chapitre 2 comme Kaur *et al.* (2022) qui prend en compte l'équilibrage de charge. Kaur *et al.* (2022) et Wang

et al. (2020b) de leur côté, se concentrent principalement sur la satisfaction des exigences de performances et ne répondent pas aux exigences d'architecture de microservices. De plus, les méthodes de placement de microservices existantes les considèrent comme des composants uniques et ignorent leurs bases de données et le registre de service. Aussi, de nombreuses études se limitent au paradigme *cloud*, *edge* ou INC et ignorent la possibilité de les combiner.

3.4 Conclusion

Dans ce chapitre, nous avons survolé un ensemble d'articles qui ont proposé des solutions pour le placement des entités de calculs telles que les fonctions réseau et le placement des microservices. Dans ce mémoire, nous considérons le placement des microservices en tenant compte des limitations discutées ci-dessus. Dans le prochain chapitre, nous allons analyser la problématique et proposer notre propre mécanisme de placement des microservices.

CHAPITRE IV

CAMP-INC: MÉCANISME DE PLACEMENT DES MICROSERVICES

4.1 Introduction

Le présent chapitre est composé de plusieurs parties. La première partie décrit un cas d'utilisation de *holopatient* pour illustrer notre modèle du système. Ensuite, dans la deuxième partie, nous expliquons l'architecture réseau du système constituée de plusieurs niveaux. Puis, nous détaillons les composants et le fonctionnement du mécanisme proposé. La troisième partie est la formulation mathématique du problème dans laquelle nous définissons le problème ainsi que les variables et nous présentons la formulation en programmation linéaire en nombres entiers et une analyse de la complexité. La dernière partie présente les algorithmes proposés.

4.2 Le modèle du système

4.2.1 Description du cas d'utilisation *holopatient*

Dans le cas d'utilisation *holopatient*, les médecins peuvent explorer le corps, parcourir le patient holographique et se concentrer sur les indications visuelles et les données visuelles afin d'acquérir une compréhension approfondie de l'état du patient et de diagnostiquer les problèmes de santé à travers un examen à distance. Aussi, les médecins peuvent recevoir un retour haptique du côté patient tout en

entendant, voyant et évaluant des patients holographiques atteints de nouvelles pathologies qui présentent divers symptômes de différentes maladies ou blessures. Un autre avantage des simulations holographiques standardisées est qu'elles permettent à un groupe de médecins d'accéder au contenu de la réalité augmentée n'importe où et d'observer le même patient, de le consulter ensemble, d'examiner des informations complémentaires telles que des graphiques ou des résultats de tests et d'échanger des points de vue sur l'évaluation et le diagnostic. De plus, un *holopatient*, capable d'imiter avec précision l'état physique d'un patient, aide les étudiants en médecine à développer des compétences cliniques dans un environnement sûr et pratique dirigé par un instructeur. Ils peuvent examiner un patient en réalité augmentée et sont capables de pratiquer l'évaluation visuelle et les compétences d'observation ou d'utiliser des études de cas de pathologie pour une option *plug-and-play* dans un cadre où l'instructeur contrôle entièrement le scénario. Ce cas d'utilisation peut être utilisé à la fois à des fins de diagnostic et de formation. Cependant, nous nous concentrons principalement sur la partie diagnostic ici, car il s'agit d'un aspect plus difficile et exigeant du cas d'utilisation. Afin que l'expérience soit agréable pour le médecin et le patient, il faut satisfaire les exigences suivantes :

- Le médecin doit recevoir des images et des vidéos holographiques de haute qualité.
- Le médecin doit recevoir un retour haptique à temps.
- Le médecin doit recevoir un retour haptique naturel avec un positionnement extrêmement précis.
- Le patient doit subir des contacts inoffensifs du robot pendant l'examen.

Ce cas d'utilisation est utilisé pour mieux illustrer notre modèle du système dont les composants et le fonctionnement sont détaillés dans la sous-section suivante.

4.2.2 L'architecture du système

L'architecture comme représentée dans la figure 4.1 est pour le cas d'utilisation du *holopatient*. Dans ce type d'architectures, nous avons deux côtés, le côté du médecin et le côté du patient qui communiquent à distance. Le médecin porte des lunettes VR de réalité virtuelle qui lui permettent de voir un hologramme du patient. Le médecin porte également des gants haptiques avec des capteurs qui transmettent des données au robot du côté du patient. Le robot est responsable d'opérer sur le patient. Il y a également une caméra du côté du patient pour capturer les images qui vont être transformées et affichées dans les lunettes VR du médecin. Les microphones et les haut-parleurs sont utilisés pour assurer une communication audio entre le médecin et les assistants dans le côté du patient. L'architecture se compose de quatre niveaux, le premier niveau est constitué d'un ensemble d'appareils IoT. Il y a deux types d'appareils :

1. Les demandeurs (par exemple, haut-parleur)
2. Les générateurs de contenu (par exemple, caméra).

Les demandeurs envoient des requêtes pour récupérer du contenu (par exemple, l'enregistrement de la caméra) qui va être traité par les microservices. Le contenu est généré par les appareils générateurs de contenu. Le deuxième niveau est connecté au premier niveau à travers une connexion sans fil et contient des stations de bases et des serveurs périphériques (*Edge Servers*). Le troisième niveau représente le réseau central constitué de plusieurs dispositifs réseau (par exemple, commutateurs et routeurs). Ensuite, le dernier niveau contient un serveur infonuagique avec une grande capacité de traitement et de stockage.

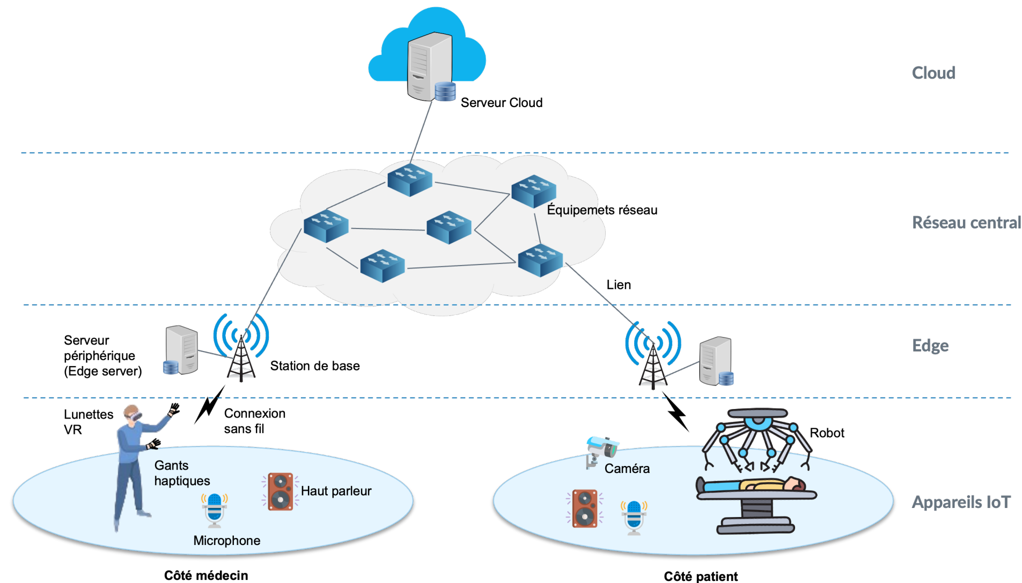

FIGURE 4.1 – l'architecture globale du système

4.2.3 Composants et fonctionnement de CaMP-INC

Dans le système, le contenu est généré par des dispositifs générateurs de contenu tels que des caméras, des microphones, des robots et des gants haptiques. Dans le cas d'utilisation holopatient, les demandeurs, par exemple, haut-parleurs et lunettes de réalité virtuelle envoient des requêtes pour exécuter des services sur le contenu. Le service est une séquence de microservices qui doivent être exécutés. Chaque microservice peut communiquer avec une base de données privée à travers des opérations de lecture/écriture. Les microservices sont déployés dans les serveurs et les dispositifs réseau.

Comme le montre la figure 4.2, une requête de service arrive à l'orchestrateur (étape 1), qui est le composant responsable de la gestion des microservices. Il y a trois éléments principaux dans l'orchestrateur :

1. Le détecteur de défaillance (FD)
2. Le gestionnaire de placement (PM)

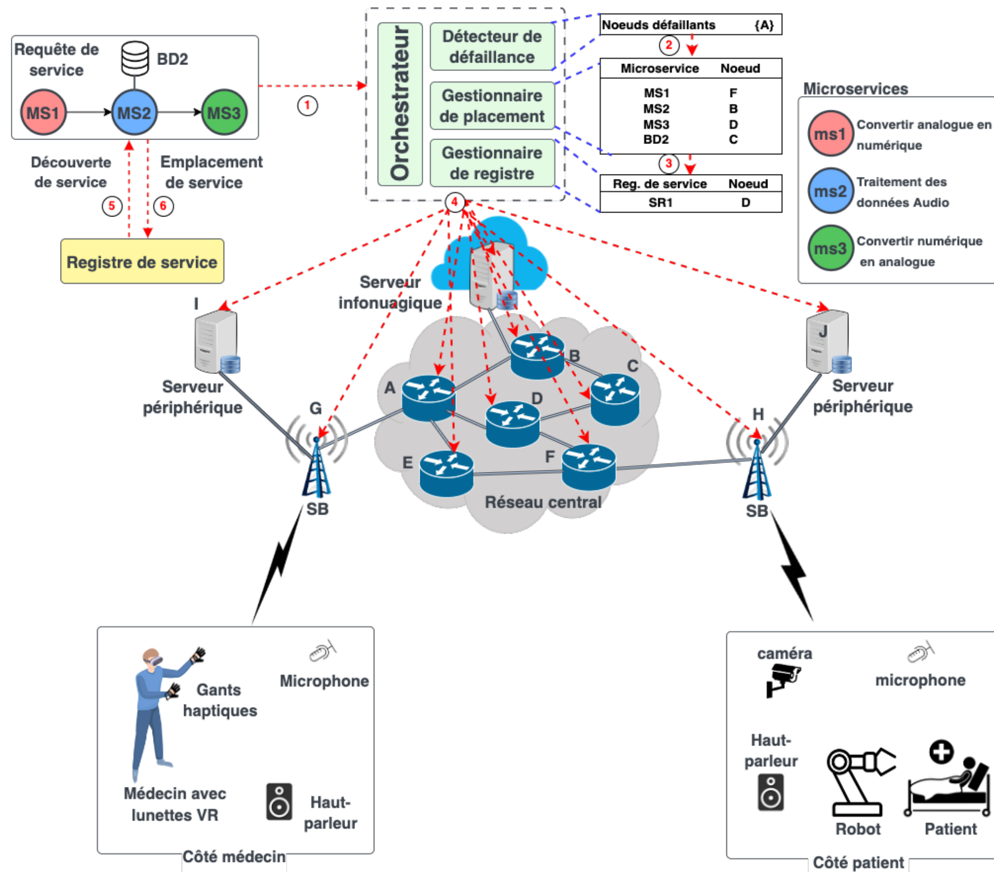

FIGURE 4.2 – modèle du système

3. Le gestionnaire de registre (RM)

Le détecteur de défaillance vérifie périodiquement les nœuds défaillants du réseau. Il vérifie régulièrement l'état du réseau et assure le suivi des nœuds défaillants pour les sauvegarder dans une liste. Chaque fois que cette liste est mise à jour, le détecteur de défaillance notifie le gestionnaire de placement (étape 2). Le gestionnaire de placement décide ensuite du placement des microservices et de leurs bases de données sur les nœuds du réseau en conséquence. Après avoir placé la chaîne de microservices, le gestionnaire de registre vérifie la distance entre les nœuds hébergeant les nouveaux microservices et les registres de services existants (étape 3). Si la distance dépasse un certain seuil, alors le gestionnaire de registre décide de pla-

cer un nouveau registre de service au voisinage des microservices. L'orchestrateur distribue ensuite les microservices et les bases de données sur les nœuds décidés par gestionnaire de placement (étape 4). Après le placement, le registre de service envoie des requêtes aux microservices nouvellement déployés pour la découverte de service (étape 5). Enfin, les microservices envoient leur emplacement (c.-à-d. l'adresse IP) au registre de service (étape 6).

4.3 Formulation mathématique de la problématique

4.3.1 Description du problème

Notre problème est défini comme le placement dynamique des microservices et de leurs bases de données correspondantes tout en minimisant le coût total incluant le coût de déploiement, le coût opérationnel et le coût de communication. Cette optimisation doit être faite tout en satisfaisant les exigences de la qualité de service telles que la capacité des dispositifs, le délai et le mappage.

4.3.2 Définition des variables

Soit V l'ensemble de tous les dispositifs du système et U l'ensemble des utilisateurs finaux. Le réseau physique est représenté par un graphe $G = (N, E)$, où $N = V \cup U$ est l'ensemble des nœuds physiques et E est l'ensemble des liens de communication entre les nœuds. Chaque dispositif $k \in V$ a un vecteur de ressources $r_k = (r_1^k, r_2^k, r_3^k, r_4^k)$, où $r_x^k \geq 0$ $(1 \leq x \leq 4)$ est la quantité de ressources en matière de calcul, de mémoire, de stockage et de communication (E/S) respectivement.

Soit S l'ensemble des microservices disponibles. Chaque microservice $v \in S$ peut avoir une base de données $db \in DB$ et nécessite un vecteur de ressources noté

$c_v = (c_1^v, c_2^v, c_3^v, c_4^v)$, où $c_x^v \geq 0$ $(1 \leq x \leq 4)$ est la quantité de ressources requise en matière de calcul, de mémoire, de stockage et de communication (E/S) respectivement. Nous supposons que chaque microservice v est empaqueté dans un conteneur qui nécessite une quantité de ressources ϕ_v en termes de calcul, mémoire, stockage et communication (E/S). La demande de service du demander $u \in U$ est notée $Q_u \in Q$ et modélisée comme un graphe orienté $Q_u = (N^u, E^u)$, où N^u représente l'ensemble de tous les microservices et leurs bases de données correspondantes à exécuter pour la requête Q_u qui doivent être placés sur des nœuds physiques N, et E^u est l'ensemble des arcs virtuels. Nous considérons deux types d'arcs :

1. Les arcs entre microservices qui représentent l'ordre d'exécution des microservices.
2. Les arcs entre chaque microservice et sa base de données correspondante qui représentent les opérations de lecture ou d'écriture dans la base de données.

Nous définissons également les variables suivantes :

— v_f^u : premier microservice à exécuter pour la requête Q_u,

— v_l^u : dernier microservice à exécuter pour la requête Q_u,

— Z^u : l'ensemble des appareils hébergeant le contenu demandé par Q_u.

4.3.3 La programmation linéaire en nombres entiers

Notre objectif est de minimiser le coût total, incluant le coût opérationnel, le coût de déploiement et le coût de communication. Soit λ_v le coût d'exécution d'une instance du microservice v et λ_{db} le coût d'exécution d'une instance de la base de données db. Le coût d'exécution est le prix payé pour l'allocation des ressources nécessaires pour l'exécution d'un microservice ou l'utilisation d'une base de données.

Le coût opérationnel est défini comme suit :

$$C_O = \sum_{k \in N,\ v \in N^u} \lambda_v x_{v,k} + \sum_{k \in N,\ db \in DB^u} \lambda_{db} y_{db,k} \tag{4.1}$$

Où $x_{v,k}$ et $y_{db,k}$ sont les indicateurs de placement des microservices et base de données respectivement. $x_{v,k} = 1$ si le microservice v est placé dans l'appareil k et 0 sinon et $y_{db,k} = 1$ si la base de données db est placée dans l'appareil k.

Le déploiement d'un nouveau microservice v entraîne un coût de déploiement γ_v (c.-à-d. le coût de la licence) de chaque microservice et le déploiement d'une nouvelle base de données db coûte γ_{db} qui est le prix du contrat de licence.

Le coût total de déploiement est alors :

$$C_D = \sum_{k \in N,\ v \in N^u} \gamma_v x_{v,k} + \sum_{k \in N,\ db \in DB^u} \gamma_{db} y_{db,k} \tag{4.2}$$

Comme précédemment mentionné, $x_{v,k}$ et $y_{db,k}$ sont les indicateurs de placement des microservices et base de données respectivement.

Nous considérons également $\beta_{(p,q)}$ comme le coût d'utilisation d'un lien physique $(p, q) \in E$ avec $p, q \in N$ comme la paire de dispositifs hébergeant les microservices qui doivent être exécutés pour satisfaire la requête Q_u du demandeur $u \in U$. Le coût de la communication inclut également le coût des liens $\beta_{(z,k)}$ entre le dispositif $z \in Z^u$ hébergeant le contenu demandé par un demandeur $u \in U$ et le dispositif $k \in V$ hébergeant le premier microservice à exécuter pour Q_u et le coût du lien $\beta_{(k,u)}$ entre l'appareil $k \in V$ hébergeant le dernier microservice à exécuter pour Q_u et le demandeur $u \in U$. Soit $\beta_{(i,j)}$ le coût du lien entre le dispositif hébergeant un microservice et un dispositif hébergeant sa base de données.

Alors le coût global de la communication est de :

$$\begin{aligned} C_C &= \sum_{\substack{(u,v)\in E^u \\ (p,q)\in E}} \beta_{(p,q)} m_{(p,q)}^{(u,v)} + \sum_{\substack{(z,k)\in E \\ z\in Z^u}} \beta_{(z,k)} x_{v_f^u,k} \\ &+ \sum_{\substack{(k,u)\in E \\ u\in U}} \beta_{(k,u)} x_{v_l^u,k} + \sum_{\substack{(v,db)\in E^u \\ (i,j)\in E}} \beta_{(i,j)} m_{(i,j)}^{(v,db)} \end{aligned} \tag{4.3}$$

Où $m_{(p,q)}^{(u,v)} = 1$ si le lien physique $(p,q) \in E$ héberge le lien virtuel $(u,v) \in E^u$ et 0 sinon et $m_{(i,j)}^{(v,db)} = 1$ si l'arc $(i,j) \in E$ héberge le lien virtuel $(v,db) \in E^u$ et 0 sinon.

L'objectif est de trouver les matrices de placement : $\mathbf{X} = [x_{v,k}]$ de taille $N^u \times V$, $\mathbf{Y} = [y_{db,k}]$ de taille $DB^u \times V$ des microservices et bases de données respectivement et la matrice $\mathbf{M} = [m_{(p,q)}^{(u,v)}]$ de taille $E^u \times E$ de mappage des liens virtuels vers les liens physiques qui minimisent le coût total.

Ainsi, le problème peut être formulé comme suit :

$$\min_{\mathbf{X},\mathbf{Y},\mathbf{M}} \quad C_{total} = C_O + C_D + C_C \tag{4.4}$$

$$\text{s. à.} \quad \sum_{v \in N^u} (c_v + \phi_v) x_{v,k} + \sum_{db \in DB^u} (c_{db} + \phi_{db}) y_{db,k} \leq r_k, \forall k \in N \tag{4.5}$$

$$\sum_{\substack{(u,v) \in E^u \\ (p,q) \in E}} d_{(p,q)} m_{(p,q)}^{(u,v)} + \sum_{v \in N^u} p_v x_{v,k} + \sum_{(z,k) \in E} (p_{v_f^u} + d_{(z,k)}) x_{v_f^u,k} + \sum_{(k,u) \in E} (p_{v_l^u} + d_{(k,u)}) x_{v_l^u,k} + \sum_{v \in N^u} (\eta_{v,k} + \psi_{v,k}) + \sum_{v \in N^u} Reg_v x_{v,k} \leq \Delta_{th}, \forall k \in N \tag{4.6}$$

$$\sum_{(u,v) \in E^u} m_{(p,q)}^{(u,v)} = 1, \forall (p,q) \in E \tag{4.7}$$

$$\sum_{v \in N^u} x_{v,k} = 1, \forall k \in N \tag{4.8}$$

La contrainte (4.5) garantit que les ressources requises par le microservice $v \in N^u$, la base de données $db \in DB^u$, le conteneur empaquetant le microservice v et le conteneur empaquetant la base de données db ne dépassent pas la capacité des dispositifs où ils ont été placés. La contrainte (4.6) garantit que le temps de réponse à la demande de service du demandeur ne dépasse pas le seuil de délai prédéfini Δ_{th}. La contrainte du délai est la somme des termes suivants :

1. Le délai de chaque lien physique entre chaque couple de dispositifs hébergeant des microservices exécutés pour fournir le service demandé,
2. Le délai de traitement de ces microservices,
3. Le délai de communication entre le générateur de contenu (c.-à-d. celui sélectionné pour fournir le contenu à la demande de l'usager) et le dis-

positif hébergeant le premier microservice et le délai de traitement de ce microservice,

4. Le délai de communication entre le demandeur et le dispositif hébergeant le dernier microservice et le délai de traitement de ce dernier,
5. Le temps nécessaire pour créer un conteneur et le libérer $\eta_{v,k}$ et $\psi_{v,k}$ respectivement pour le microservice v sur le dispositif k, et
6. Le délai entre le registre de service et les microservices.

Les contraintes (4.7) et (4.8) sont des contraintes de mappage. La contrainte (4.7) garantit que les microservices et les liens virtuels entre eux sont affectés à une paire d'appareils et le lien physique entre eux. La contrainte (4.8) garantit que chaque microservice est placé dans exactement un nœud physique.

Le tableau 4.1 définit toutes les notations utilisées dans la formulation du problème.

4.3.4 Complexité du problème

Afin de prouver la $\mathcal{NP}$-difficulté du problème (4.4), nous le réduisons au problème d'affectation généralisé (GAP) Cattrysse et Van Wassenhove (1992). Étant donné n bacs et m articles, le bac i a une capacité C_i, $1 \leq i \leq n$, l'article j a un coût c_{ij} avec un poids w_j si assigné au bac i, $1 \leq j \leq m$. Le GAP vise à minimiser le coût total en affectant les articles aux bacs, en respectant les capacités des bacs. Nous considérons un cas particulier du problème (4.4) où seuls les microservices et leurs bases de données sont placés et il n'y a pas de dépendance entre eux. Par conséquent, si le microservice ou la base de données j est placé dans le dispositif i, il consomme w_j de ressources et entraîne un coût total de c_{ij}. Pour cela, le problème (4.4) vise à minimiser le coût total de placement des microservices et des bases de données, sujet au respect de la capacité C_i de chaque microservice ou

Notation	Description
V	ensemble de tous les dispositifs du système
U	ensemble d'utilisateurs finaux
E	ensemble de tous les arcs de communication entre les dispositifs
S	ensemble de microservices disponibles
DB	ensemble de bases de données disponibles
Q	ensemble de requêtes de services demandés par les demandeurs
Q_u	requête de service du demandeur $u \in U$
N^u	ensemble de microservices pour la requête $Q_u \in Q$
DB^u	ensemble de bases de données pour la requête $Q_u \in Q$
E^u	ensemble d'arcs entre les microservices pour la requête $Q_u \in Q$
Z^u	ensemble de dispositifs hébergeant du contenu pour la requête $Q_u \in Q$
r_k	quantité de ressources disponibles dans l'appareil $k \in V$
c_v	quantité de ressources requises pour le microservice $v \in S$
ϕ_v	quantité de ressources requises pour le conteneur exécutant le microservice $v \in S$
v_f^u	le premier microservice à exécuter pour la requête $Q_u \in Q$
v_l^u	le dernier microservice à exécuter pour la requête $Q_u \in Q$
λ_v	coût d'exécution d'une instance de microservice $v \in S$
λ_{db}	coût d'exécution d'une instance de base de données $db \in DB$
γ_v	coût de déploiement du microservice $v \in S$
γ_{db}	coût de déploiement de la base de données $db \in DB$
$\eta_{v,k}$	temps de créer un conteneur pour le microservice v dans le dispositif $k \in V$
$\psi_{v,k}$	temps de libérer un conteneur pour le microservice v dans le dispositif $k \in V$
$\beta_{(p,q)}$	coût d'utilisation d'un lien physique $(p,q) \in E$
$d_{(p,q)}$	délai du lien physique $(p,q) \in E$
p_v	délai de traitement du microservice $v \in S$
Δ_{th}	seuil de délai
Reg_v	délai entre le registre de service et le nœud hébergeant le microservice $v \in S$

TABLEAU 4.1 – tableau des notations

base de données i. Ce cas particulier de notre problème est équivalent au GAP. Par conséquent, le problème (4.4) est $\mathcal{NP}$-difficile, en raison de la $\mathcal{NP}$-difficulté du GAP.

4.4 Algorithmes proposés

La solution CaMP-INC proposée se compose de trois éléments principaux, à savoir :

1. Le gestionnaire de placement (PM), qui place les microservices et leurs bases de données.
2. Le détecteur de défaillance (FD), qui détecte les dispositifs défaillants.
3. Le gestionnaire de registre (RM), qui décide s'il est nécessaire de placer un nouveau registre de service.

4.4.1 Gestionnaire de placement (*Placement Manager*)

Le gestionnaire de placement communique avec le détecteur de défaillance et le gestionnaire de registre pour décider du placement des microservices, de leurs bases de données et, si nécessaire, d'un nouveau registre de service. L'algorithme de placement (Algorithme 1) prend en entrée le graphe du réseau G, la requête q du demandeur et le seuil de délai Δ_{th}. Il commence par récupérer les nœuds défaillants détectés par le détecteur de défaillance et les sauvegarde dans une liste F. Puis, il calcule les K chemins les plus courts entre le nœud générateur de contenu (c.-à-d., $q.NC$) et le demandeur (c.-à-d., $q.UF$) en tenant compte du seuil de délai Δ_{th} (lignes 1–2). Les K chemins sont stockés dans une liste L. Les variables $CoutMin$ et C sont initialisées avec l'infini ∞ (lignes 3–4) et sont utilisées pour calculer le coût du placement minimal. Pour chaque chemin $l \in L$ (lignes 5), nous initialisons la variable P avec l'ensemble vide $\emptyset$ (lignes 6) afin d'y stocker les stra-

Algorithm 1 placement

Entrée: G : graphe du réseau, q : requête de l'UF, Δ_{th} : seuil de délai
Sortie: StrategiePlacement, CheminAvecCoutMin

1: $F \leftarrow$ NoeudsDefaillants(G)
2: $L \leftarrow$ K-CheminMin($G \setminus F, q.UF, q.NC, \Delta_{th}, K$)
3: $CoutMin \leftarrow \infty$
4: $C \leftarrow \infty$
5: **pour** $l \in L$ **faire**
6: $P \leftarrow \emptyset$
7: $MauvaisChemin \leftarrow faux$
8: **pour** $m \in q.S$ **faire**
9: $lRank \leftarrow$ rank(l)
10: $MeilleurNoeud \leftarrow$ NoeudAvecRankMax($lRank$)
11: **si** $MeilleurNoeud = null$ **alors**
12: $MauvaisChemin \leftarrow vrai$
13: $break$
14: **fin si**
15: $P.ajouter((m, MeilleurNoeud))$
16: **fin pour**
17: **si** $MauvaisChemin = faux$ **alors**
18: $C \leftarrow$ CoutPlacement(P)
19: **si** $C < CoutMin$ **alors**
20: $CoutMin \leftarrow C$
21: $PlacementMin \leftarrow P$
22: $CheminMin \leftarrow l$
23: **fin si**
24: **fin si**
25: **fin pour**
26: **si** $C = \infty$ **alors**
27: **retourner** $\emptyset$
28: **sinon**
29: **si** DecisionRegistre($G \setminus F, CheminMin, \Delta_{th}$) **alors**
30: **pour** $n \in$ Voisins($PlacementMin$) **faire**
31: **si** SatisfaitDelai($n, PlacementMin, \Delta_{th}$) **alors**
32: $PlacementMin.ajouter((Registre, n))$
33: $break$
34: **fin si**
35: **fin pour**
36: **fin si**
37: **retourner** $PlacementMin, CheminMin$
38: **fin si**

tégies de placement calculées. Nous initialisons également la variable booléenne $MauvaisChemin$ par $faux$ (lignes 7) qui est utilisée pour s'assurer de ne calculer le coût de placement minimal que si nous trouvons un placement pour tous les microservices et les bases de données de la requête. Le classement des nœuds est utilisé pour affecter chaque microservice ou sa base de données ((c.-à-d., les composants des microservices et les bases de données de la chaîne de service $q.S$) au nœud avec le score le plus élevé, ensuite, la paire de microservices et le meilleur nœud est ajoutée à la liste P (lignes 8–15). Si tous les microservices sont affectés à des nœuds dans le chemin, le coût de placement est calculé (lignes 17–18). Si ce dernier est minimal, il est sauvegardé avec le placement et le chemin correspondants (lignes 19–22) dans les variables $CoutMin$, $PlacementMin$ et $CheminMin$ respectivement. Un ensemble vide est retourné si aucun placement n'est trouvé (lignes 21–22). Sinon, l'algorithme vérifie la nécessité de placer un nouveau registre de service à l'aide du gestionnaire de registre. S'il satisfait le seuil de délai pour un des voisins des nœuds hébergeant les microservices placés, alors le registre y est placé (lignes 28–33). Enfin, le placement avec un coût minimum et le chemin utilisé dans ce placement sont retournés (lignes 37).

Le classement des nœuds est fait en utilisant l'algorithme *PageRank* Page *et al.* (1999). L'algorithme *PageRank* est utilisé par le moteur de recherche *Google* Brin et Page (1998). Le classement des pages est basé sur l'idée que l'importance d'une page *Web* est déterminée par le nombre de pages auxquelles elle est liée. Par analogie, les pages *Web* représentent dans notre cas les microservices et les bases de données à placer.

Algorithm 2 détection de défaillance

Entrée: G : graphe du réseau, t : période de temps

1: **pour** $noeud \in G$ **faire**
2: MessageVerification($noeud$)
3: **si** $\neg$Ack($noeud$) **alors**
4: MarquerDefaillant($noeud$)
5: informer le gestionnaire de placement
6: **fin si**
7: **fin pour**
8: Attendre(t)
9: **Aller à** 1

4.4.2 Détecteur de défaillance (*Failure Detector*)

Le Détecteur de défaillance vérifie périodiquement les nœuds défaillants comme décrit dans Algorithme 2. Il prend en entrée le graphe du réseau G et une certaine période de temps t. Il envoie d'abord un message de vérification de l'état à chaque nœud dans G (ligne 2). Ce message de vérification est une requête envoyée à chaque nœud pour examiner s'il est tombé en panne ou pas. Ensuite, si aucun accusé de réception n'est reçu, il marque le nœud comme défaillant pour la période et informe le gestionnaire de placement en lui envoyant l'ensemble des nœuds défaillants (lignes 3–5). L'algorithme attend une période de temps t avant de revérifier (ligne 8).

4.4.3 Gestionnaire de registre (*Registry Manager*)

Le gestionnaire de registre utilise l'algorithme 3 pour vérifier s'il est nécessaire de placer un nouveau registre de service. Les entrées de la fonction DecisionRegistre() sont le graphe du réseau G, le chemin *chemin* et le seuil de délai Δ_{th}. Tout d'abord, le gestionnaire de registre récupère les emplacements des registres de services existants et les sauvegarde dans une liste L_{reg} (ligne 2). Après, il initialise la variable booléenne $delaiDepasse$ par la valeur $faux$ et un compteur cpt par zéro (lignes

Algorithm 3 décision de placement du registre

1: **fonction** DecisionRegistre(G : graphe du réseau, $chemin$: chemin, Δ_{th} : seuil de délai) : booléen
2: $L_{reg} \leftarrow$ TrouverRegistries(G)
3: $delaiDepasse \leftarrow faux$
4: $cpt \leftarrow 0$
5: **pour** $reg \in L_{reg}$ **faire**
6: $D \leftarrow$ Delai($chemin, reg$)
7: **si** $D > \Delta_{th}$ **alors**
8: $delaiDepasse \leftarrow vrai$
9: $cpt \leftarrow cpt + 1$
10: **fin si**
11: **fin pour**
12: **si** $delaiDepasse = vrai \wedge cpt = LongueurListe(L_{reg})$ **alors**
13: **retourner** vrai
14: **fin si**
15: **retourner** faux
16: **fin fonction**

3–4). Ensuite, il calcule le délai entre chaque registre et nœud du chemin, si ce délai dépasse le seuil, la variable $delaiDepasse$ prend la valeur $vrai$ et le compteur cpt s'incrémente de 1 (lignes 4–9). Si tous les registres existants ne satisfont pas le seuil de délai et le compteur cpt est égal au nombre de registres existants, la valeur $vrai$ est retournée (lignes 12–13). Sinon, la fonction retourne la valeur $faux$ (ligne 15). Cette valeur est ensuite vérifiée par le gestionnaire de placement afin de décider de placer un nouveau registre de service.

4.5 Conclusion

Dans ce chapitre, nous avons expliqué notre solution proposée CaMP-INC. Nous avons commencé par définir un cas d'utilisation. Puis, nous avons détaillé l'architecture du système ainsi que les composants et le fonctionnement de CaMP-INC. Après, nous avons décrit le problème et analysé sa complexité. Et finalement, nous avons détaillé trois algorithmes qui composent le mécanisme global de placement

des microservices. Dans le prochain chapitre, nous allons évaluer la performance de notre solution proposée CaMP-INC.

CHAPITRE V

ÉVALUATION DE PERFORMANCES

5.1 Introduction

Dans ce chapitre, nous évaluons la performance de notre solution proposée CaMP-INC. Pour ce faire, nous la comparons avec notre formulation en programmation linéaire en nombres entiers nommée ILP et un algorithme de placement des micro-services nommé EPTA. Tout d'abord, nous détaillons l'environnement ainsi que la topologie et les paramètres de simulation. Par la suite, nous expliquons brièvement l'algorithme de comparaison EPTA. Pour finir, nous présentons les résultats des simulations et nous les discutons.

5.2 Environnement de simulation

Nous avons utilisé le solveur Gurobipy v9.5.1 PyPI (2022) qui est une bibliothèque logicielle d'optimisation mathématique pour résoudre des problèmes d'optimisation linéaire et quadratique à nombres entiers mixtes. Nous avons utilisé le langage de programmation Python pour coder les algorithmes proposés et l'algorithme de comparaison.

Nous avons fait tourner les simulations sur une machine HP Ubuntu 20.04 avec les caractéristiques suivantes : Intel Core(TM) i5-8250U CPU @1.60 GHz 1.80 GHz,

8.00 Go de mémoire RAM et un système d'exploitation 64 bits, processeur x64.

5.3 Topologie et paramètres de simulation

La topologie utilisée pour les simulations est illustrée dans la figure 5.1 et se compose de 8 utilisateurs (4 sont des demandeurs de contenu et 4 sont des générateurs de contenu), 2 stations de base connectant les utilisateurs à 2 serveurs périphériques et le réseau central composé de 6 dispositifs du réseau et un serveur infonuagique connecté au réseau central.

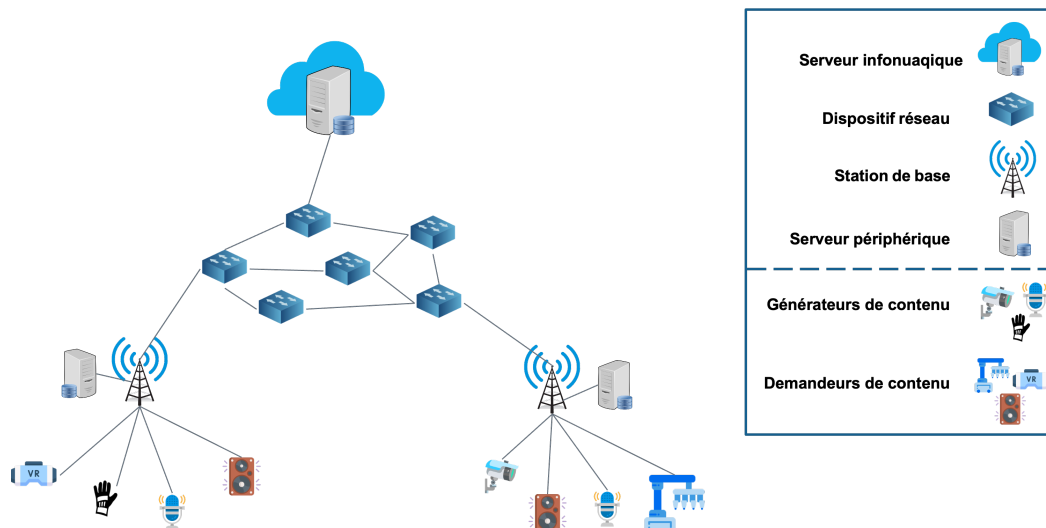

FIGURE 5.1 – la topologie du réseau

La capacité des nœuds physiques varie en {250, 500, 1000, 10000} cycles CPU/s selon si le nœud physique est un dispositif réseau, une station de base, un serveur périphérique ou un serveur infonuagique respectivement. Le coût de la licence d'un microservice est de 100$ et le nombre de microservices par requête varie dans l'intervalle [3-5]. Puisque nous considérons des applications contraintes par le délai, le seuil de délai varie dans l'intervalle [10-100] ms. Nous faisons varier le nombre de requêtes en {5, 10, 15, 20, 25, 30} et calculons les caractéristiques de performance de notre formulation en programmation linéaire en nombres entiers

Paramètre	Valeur
Nombre de nœuds du réseau	19 nœuds
Capacité des nœuds physiques	{250, 500, 1000, 10000} cycles CPU/s
Le coût de licence d'un microservice	100$
Nombre de microservices par requête	[3-5] microservices
Seuil de délai	[10-100] ms

Tableau 5.1 – les paramètres de simulation

ILP, de notre mécanisme de placement proposé CaMP-INC et d'un algorithme de placement des microservices nommé EPTA Wan *et al.* (2018). Le tableau 5.1 regroupe tous les paramètres de simulation.

5.4 L'algorithme de comparaison

Nous comparons notre formulation en programmation linéaire en nombres entiers ILP et notre mécanisme de placement des microservices CaMP-INC avec l'algorithme proposé dans Wan *et al.* (2018). L'algorithme EPTA (Execution container Placement with Task assignment Algorithm) permet le placement du conteneur d'exécution empaquetant un microservice avec l'affectation de tâche. Cet algorithme de placement des microservices utilise la programmation linéaire de façon distribuée sur plusieurs nœuds du réseau. Wan *et al.* (2018) partage le réseau en plusieurs régions, et distribue l'ensemble des requêtes sur plusieurs contrôleurs, chaque contrôleur optimise la fonction objectif du placement sur un nombre restreint de requêtes et une zone limitée du réseau. Lors de notre implémentation, nous avons simplifié l'algorithme Wan *et al.* (2018), la version simplifiée fait le placement des requêtes une par une au lieu de faire le placement de toutes les requêtes d'un coup. Ceci permet de diminuer le temps d'exécution de l'algorithme.

5.5 Résultats et discussions

Dans cette section, nous présentons et discutons les résultats des simulations. Nous commençons par calculer le coût total incluant le coût de déploiement, le coût opérationnel et le coût de communication. Après, nous évaluons le temps d'exécution de chaque approche. Ensuite, nous calculons le coût de communication uniquement, puis, le nombre de nœuds utilisés. Pour montrer l'impact de la prise en compte des exigences de l'architecture de microservices, nous calculons la latence moyenne et la distance moyenne entre les microservices et le registre de service.

5.5.1 Le coût total

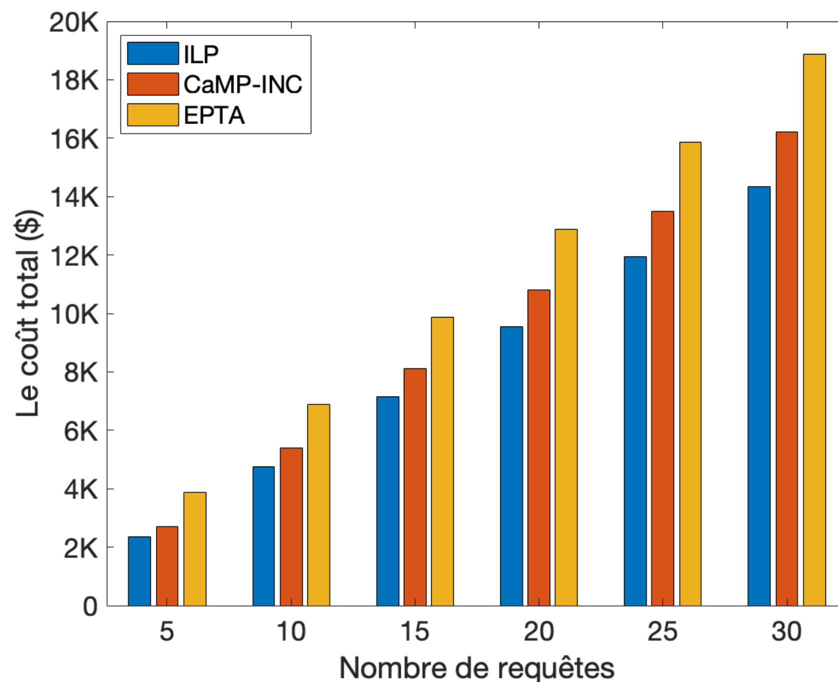

FIGURE 5.2 – le coût total

La figure 5.2 représente le coût total (c.-à-d., la somme des coûts de déploiement,

opérationnel et de communication) des différentes approches pour chaque nombre de requêtes. Nous observons que la programmation linéaire en nombres entiers ILP peut réduire le coût de plus de 13% et 39% par rapport aux heuristiques CaMP-INC et EPTA respectivement. De plus, CaMP-INC représente un bon compromis entre la programmation linéaire en nombres entiers ILP et l'algorithme EPTA. En revanche, EPTA a les moins bonnes performances, car il préfère sélectionner les nœuds proches du contrôleur, même si le coût de communication entre le demandeur et le contenu est très élevé.

5.5.2 Le temps d'exécution

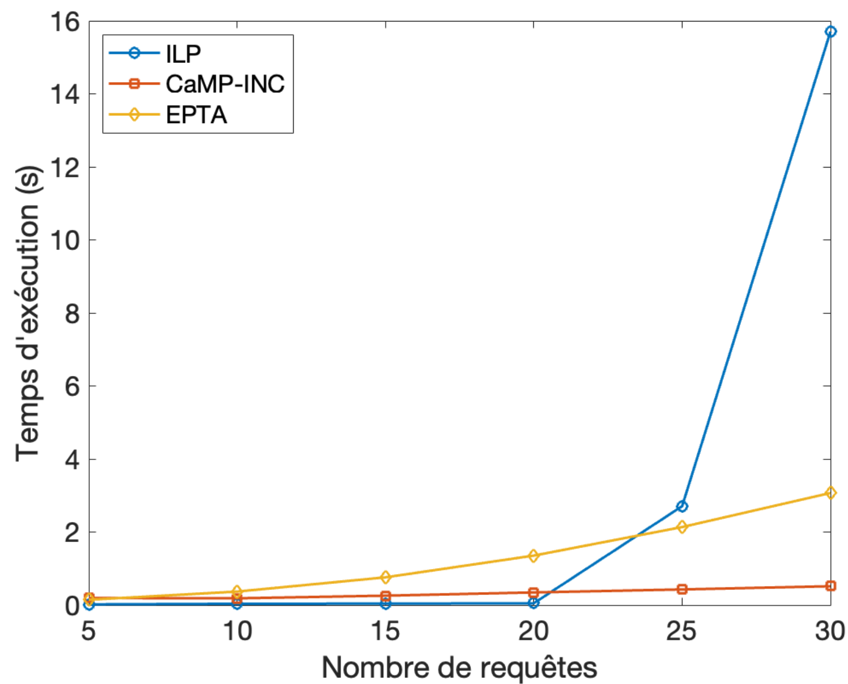

FIGURE 5.3 – le temps d'exécution

Nous faisons varier le nombre de requêtes et nous calculons le temps d'exécution en seconde de chaque approche. La figure 5.3 compare le temps d'exécution du ILP, de CaMP-INC et du EPTA. Le temps d'exécution du ILP augmente signifi-

cativement avec l'augmentation du nombre de requêtes. À partir de 35 requêtes, le temps d'exécution prend de nombreuses heures, ce qui est peu pratique à l'usage. Néanmoins, les deux heuristiques ont un temps d'exécution acceptable (c.-à-d., inférieur à 4s). CaMP-INC surpasse EPTA, car les auteurs de l'algorithme EPTA utilisent une version allégée de leur formulation de programmation linéaire en nombres entiers dans l'heuristique ce qui explique l'augmentation du temps d'exécution par rapport au nombre de requêtes.

La complexité temporelle de l'algorithme CaMP-INC dans le pire des cas est $O(K * m * log(n))$, où n est le nombre de nœuds dans le graphe G, m est le nombre de micorservices dans la requête de l'utilisateur final, et K est le nombre de chemins les plus courts calculés. L'algorithme effectue une boucle à travers K chemins les plus courts, puis effectue une boucle à travers les micorservices m, puis calcule le coût de placement, ensuite effectue d'autres opérations qui prennent un temps logarithmique dans le pire des cas.

5.5.3 Le coût de communication

On peut observer sur la figure 5.4 que ILP surpasse CaMP-INC et EPTA. Étant donné que la programmation linéaire en nombres entiers donne toujours le résultat le plus optimal, la performance de CaMP-INC et EPTA est inférieure à celle du ILP. De plus, EPTA a le coût de communication le plus élevé par rapport aux deux autres approches. La différence de coût entre EPTA et CaMP-INC est due au fait que les microservices dans EPTA communiquent entre eux à travers le contrôleur, ce qui n'est pas le cas dans CaMP-INC où les microservices sont capables de communiquer entre eux sans aucun intermédiaire. Cette communication entre chaque microservice et le contrôleur engendre un délai supplémentaire.

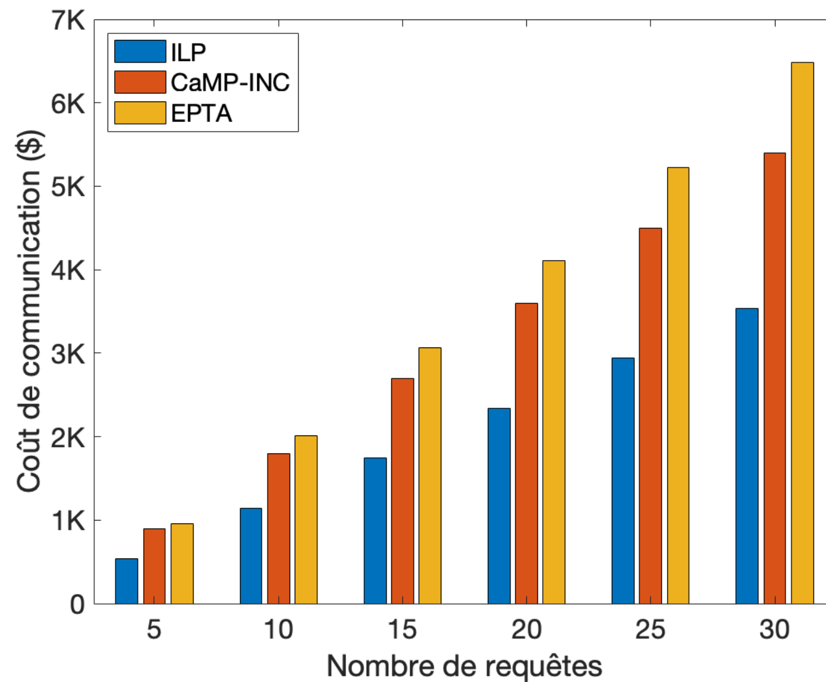

FIGURE 5.4 – le coût de communication

5.5.4 Le nombre de noeuds utilisés pour le placement

La figure 5.5 montre le nombre de nœuds utilisés pour placer les microservices. Nous observons que CaMP-INC permet en moyenne d'avoir moins de nœuds pour le placement comparé aux ILP et EPTA. Ceci peut être important au cas où il serait nécessaire de conserver certains nœuds pour d'autres traitements. Cependant, ILP utilise en moyenne le plus grand nombre de nœuds, ce qui n'est pas incorrect puisque l'objectif principal est d'avoir le coût optimal. Autrement dit, ILP se concentre uniquement sur la solution oprimale et ne prête pas attention au nombre de noeuds utilisés.

Dans ce qui suit, nous comparons les performances de CaMP-INC et EPTA en ce qui concerne les exigences de l'architecture de microservices, en particulier la détection des pannes et la gestion du registre de services. L'objectif étant de

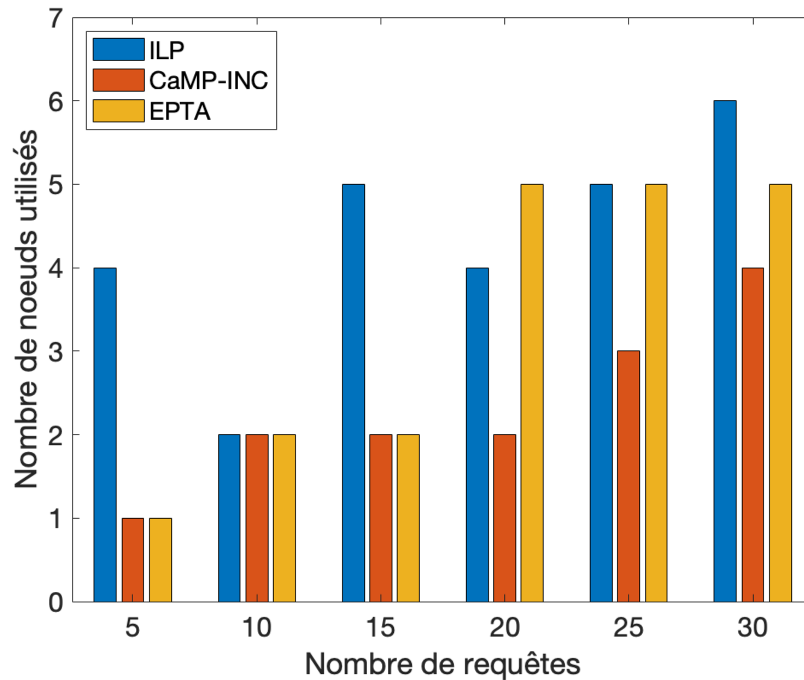

FIGURE 5.5 – le nombre de noeuds utilisés pour le placement

montrer l'importance de considérer ces exigences précédemment discutées en ce qui concerne le temps de latence et le délai de communication.

5.5.5 La latence moyenne des requêtes de service

La figure 5.6 illustre l'impact de la détection des défaillances pour le placement des microservices. Nous pouvons observer que la latence moyenne des requêtes de service pour CaMP-INC augmente légèrement avec l'augmentation du nombre de requêtes. Puisque, plus il y a de requêtes, plus les microservices sont placés dans des chemins plus longs et donc des délais de liaison sont ajoutés. D'autre part, EPTA a une latence moyenne de requête de service plus élevée que CaMP-INC, car il ne prend pas en compte les nœuds défaillants et décide ensuite de placer des microservices dans des nœuds qui ne fonctionnent pas. Par conséquent, lorsque la

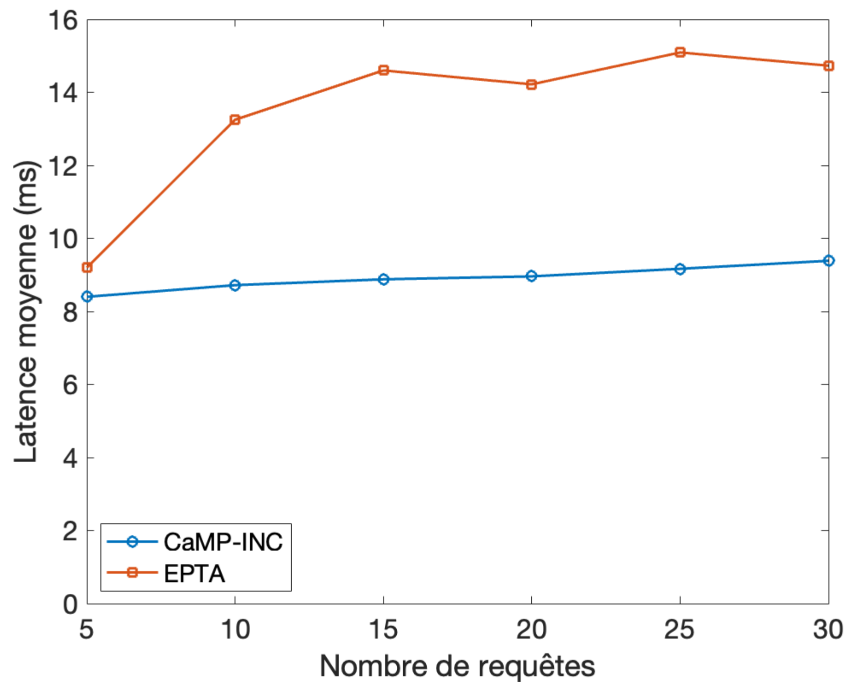

FIGURE 5.6 – la latence moyenne des requêtes de service

demande arrive dans un nœud défaillant, le traitement échoue et une requête est envoyée à l'orchestrateur pour prendre une autre décision de placement.

5.5.6 Le délai de communication moyen entre les microservices et les registres de services

Afin de comparer le placement des registres de services entre EPTA et CaMP-INC, nous calculons le délai de communication moyen entre les microservices placés et les registres de services (figure 5.7). CaMP-INC place d'une meilleure façon les registres de services par rapport à EPTA et réduit la distance en conséquence. Cette distance réduite se traduit par un faible temps de communication entre les microservices et les registres de services, ce qui est très avantageux dans les scénarios de délais limités où la latence doit être aussi faible que possible.

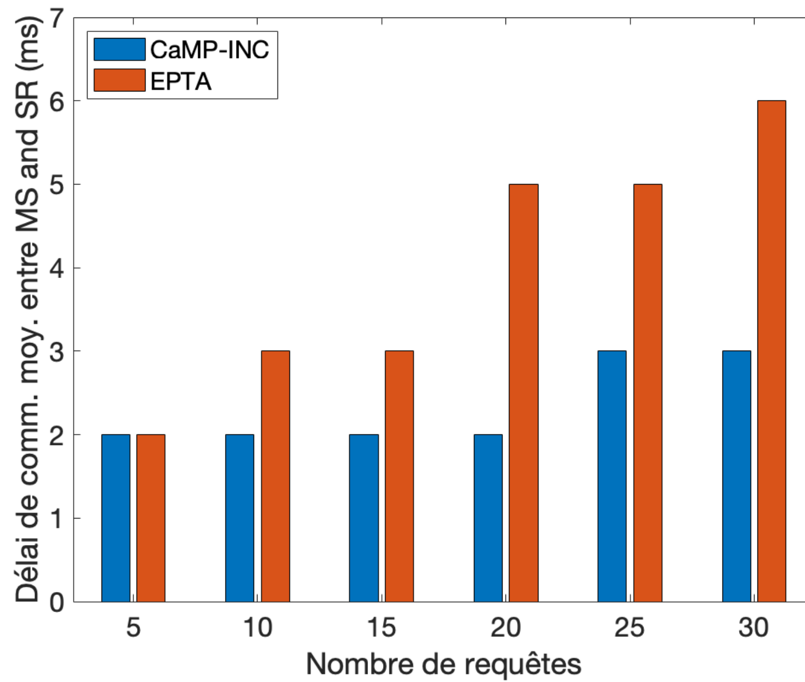

FIGURE 5.7 – le délai de communication moyen entre les microservices (MS) et les registres de services (RS)

5.6 Conclusion

Ce chapitre a présenté l'environnement ainsi que la topologie et les paramètres de simulation. Il présente également les différents résultats obtenus et leur discussion. Les résultats montrent que notre solution proposée représente un bon compromis entre une solution existante et la solution optimale. De plus, à travers ces résultats, nous avons montré l'avantage de prendre en considération les exigences de l'architecture de microservices, notamment la détection de défaillance et le placement du registre de service.

CHAPITRE VI

CONCLUSION

En conclusion, notre étude sur l'adoption des microservices dans les continuums nuage-périphérie de l'informatique en nuage et en périphérie reflète notre engagement envers la compréhension approfondie de cette architecture en évolution rapide. Nous avons exploré les défis inhérents à la gestion des microservices et proposé une solution, CaMP-INC, qui offre une performance accrue et un bon équilibre entre les coûts. Ce travail nous a permis de réfléchir sur les avantages potentiels de l'architecture de microservices, tout en reconnaissant les défis qui doivent être surmontés pour en tirer le meilleur parti. Nous continuerons à explorer de nouvelles opportunités pour améliorer CaMP-INC en intégrant des fonctionnalités telles qu'un équilibrage de charge et une gestion intelligente des répliques.

RÉFÉRENCES

(2022). Cloud migration stats - 2022 flexera state of the cloud report. https ://info.flexera.com/CM-REPORT-State-of-the-Cloud.

Ali, S., Pandey, M. et Tyagi, N. (2020). Wireless-fog mesh : A framework for in-network computing of microservices in semipermanent smart environments. *International Journal of Network Management*, *30*(6), e2125.

Ali, S. O., Hmitti, Z. A., Elbiaze, H. et Glitho, R. (2021). On computing in the network : Covid-19 coughs detection case study. Dans *International Symposium on Ubiquitous Networking*, 186–198. Springer.

Balalaie, A., Heydarnoori, A. et Jamshidi, P. (2015). Migrating to cloud-native architectures using microservices : an experience report. Dans *European Conference on Service-Oriented and Cloud Computing*, 201–215. Springer.

Balalaie, A., Heydarnoori, A. et Jamshidi, P. (2016). Migrating to cloud-native architectures using microservices : an experience report. Dans *Advances in Service-Oriented and Cloud Computing : Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers 4*, 201–215. Springer.

Bhamare, D., Jain, R., Samaka, M. et Erbad, A. (2016). A survey on service function chaining. *Journal of Network and Computer Applications*, *75*, 138–155.

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. *et al.* (2014). P4 : Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, *44*(3), 87–95.

Brin, S. et Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, *30*(1-7), 107–117.

Carrales, A. V. (2021). Microservices vs monolith. https ://faun.pub/microservices-vs-monolith-the-ultimate-comparison-2021-8747201ed53.

Cattrysse, D. G. et Van Wassenhove, L. N. (1992). A survey of algorithms for the generalized assignment problem. *European journal of operational research*, *60*(3), 260–272.

Chen, X., Huang, Q., Wang, P., Meng, Z., Liu, H., Chen, Y., Zhang, D., Zhou, H., Zhou, B. et Wu, C. (2021). Lightnf : Simplifying network function offloading in programmable networks. Dans *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, 1–10. `http://dx.doi.org/10.1109/IWQOS52092.2021.9521329`

Chen, X., Zhang, D., Wang, X., Zhu, K. et Zhou, H. (2019). P4sc : Towards high-performance service function chain implementation on the p4-capable device. Dans *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 1–9.

Chowdhury, S. R., Salahuddin, M. A., Limam, N. et Boutaba, R. (2019). Rearchitecting nfv ecosystem with microservices : State of the art and research challenges. *IEEE Network*, *33*(3), 168–176.

Cooke, R. A. et Fahmy, S. A. (2020). A model for distributed in-network and near-edge computing with heterogeneous hardware. *Future Generation Computer Systems*, *105*, 395–409.

Deng, S., Xiang, Z., Taheri, J., Khoshkholghi, M. A., Yin, J., Zomaya, A. Y. et Dustdar, S. (2020). Optimal application deployment in resource constrained distributed edges. *IEEE Transactions on Mobile Computing*, *20*(5), 1907–1923.

Ding, Z., Wang, S. et Jiang, C. (2022). Kubernetes-oriented microservice placement with dynamic resource allocation. *IEEE Transactions on Cloud Computing*, (01), 1–1.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. et Safina, L. (2017). Microservices : yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195–216.

Education, I. C. (2021). microservices. https ://www.ibm.com/cloud/learn/microservices.

Filali, A., Abouaomar, A., Cherkaoui, S., Kobbane, A. et Guizani, M. (2020). Multi-access edge computing : A survey. *IEEE Access*, *8*, 197017–197046.

Frye, B. (2020). 8 steps for migrating existing applications to microservices. https ://insights.sei.cmu.edu/blog/8-steps-for-migrating-existing-applications-to-microservices/.

Gos, K. et Zabierowski, W. (2020). The comparison of microservice and monolithic architecture. Dans *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, 150–153. IEEE.

Hasselbring, W. et Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. Dans *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 243–246. IEEE.

Hawilo, H., Jammal, M. et Shami, A. (2019). Exploring microservices as the architecture of choice for network function virtualization platforms. *IEEE Network*, *33*(2), 202–210.

Hu, N., Tian, Z., Du, X. et Guizani, M. (2021). An energy-efficient in-network computing paradigm for 6g. *IEEE Transactions on Green Communications and Networking*, *5*(4), 1722–1733.

ISG, N. (2014). *Network Functions Virtualisation (NFV) ; Virtual Network Functions Architecture.* Rapport technique, ETSI, Tech. Rep.

Jolie-lang.org (2022). Jolie, the service-oriented programming language. https ://www.jolie-lang.org.

Kakivaya, G., Xun, L., Hasha, R., Ahsan, S. B., Pfleiger, T., Sinha, R., Gupta, A., Tarta, M., Fussell, M., Modi, V. *et al.* (2018). Service fabric : a distributed platform for building microservices in the cloud. Dans *Proceedings of the thirteenth EuroSys conference*, 1–15.

Kang, H., Le, M. et Tao, S. (2016). Container and microservice driven design for cloud infrastructure devops. Dans *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 202–211. IEEE.

Kaur, K., Guillemin, F., Rodriguez, V. Q. et Sailhan, F. (2022). Latency and network aware placement for cloud-native 5g/6g services. Dans *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, 114–119. IEEE.

Kecskemeti, G., Marosi, A. C. et Kertesz, A. (2016). The entice approach to decompose monolithic services into microservices. Dans *2016 International Conference on High Performance Computing Simulation (HPCS)*, 591–596. http://dx.doi.org/10.1109/HPCSim.2016.7568389

Król, M. et Psaras, I. (2017). Nfaas : named function as a service. Dans *Proceedings of the 4th ACM Conference on Information-Centric Networking*, 134–144.

Krylovskiy, A., Jahn, M. et Patti, E. (2015a). Designing a smart city internet of things platform with microservice architecture. Dans *2015 3rd International Conference on Future Internet of Things and Cloud*, 25–30. http://dx.doi.org/10.1109/FiCloud.2015.55

Krylovskiy, A., Jahn, M. et Patti, E. (2015b). Designing a smart city internet of things platform with microservice architecture. Dans *2015 3rd international conference on future internet of things and cloud*, 25–30. IEEE.

Kulatunga, C., Bhargava, K., Vimalajeewa, D. et Ivanov, S. (2017). Cooperative in-network computation in energy harvesting device clouds. *Sustainable Computing : Informatics and Systems*, *16*, 106–116.

Lee, H., Lee, J., Ko, H. et Pack, S. (2019). Resource-efficient service function chaining in programmable data plane. Dans *Proc. EuroP4*.

Lee, J., Ko, H., Lee, H. et Pack, S. (2021). Flow-aware service function embedding algorithm in programmable data plane. *IEEE Access*, *9*, 6113–6121. http://dx.doi.org/10.1109/ACCESS.2020.3048421

Lia, G., Amadeo, M., Campolo, C., Ruggeri, G. et Molinaro, A. (2021). Optimal placement of delay-constrained in-network computing tasks at the edge with minimum data exchange. Dans *2021 IEEE 4th 5G World Forum (5GWF)*, 481–486. IEEE.

Liang, Y. et Lan, Y. (2021). Tclbm : A task chain-based load balancing algorithm for microservices. *Tsinghua Science and Technology*, *26*(3), 251–258. http://dx.doi.org/10.26599/TST.2019.9010032

Liu, M., Peter, S., Krishnamurthy, A. et Phothilimthana, P. M. (2019). E3 :{Energy-Efficient} microservices on {SmartNIC-Accelerated} servers. Dans *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 363–378.

Mafioletti, D. R., Dominicini, C. K., Martinello, M., Ribeiro, M. R. N. et Villaça, R. d. S. (2020). Piaffe : A place-as-you-go in-network framework for flexible embedding of vnfs. Dans *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 1–6. http://dx.doi.org/10.1109/ICC40277.2020.9149240

Mazzara, M., Dragoni, N., Bucchiarone, A., Giaretta, A., Larsen, S. T. et Dustdar, S. (2018). Microservices : Migration of a mission critical system. *IEEE Trans. Serv. Comput.*, 1–1. http://dx.doi.org/10.1109/TSC.2018.2889087

Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M. et Urso, A. (2016). The database-is-the-service pattern for microservice architectures. Dans *International Conference on Information Technology in Bio-and Medical Informatics*, 223–233. Springer.

Newman, S. (2019). *Monolith to microservices : evolutionary patterns to transform your monolith.* O'Reilly Media.

Newman, S. (2021). Mmigrating monoliths to microservices with decomposition and incremental changes. https ://www.infoq.com/articles/migrating-monoliths-to-microservices-with-decomposition/.

Page, L., Brin, S., Motwani, R. et Winograd, T. (1999). *The PageRank citation ranking : Bringing order to the web.* Rapport technique, Stanford InfoLab.

Pallewatta, S., Kostakos, V. et Buyya, R. (2019). Microservices-based iot application placement within heterogeneous and resource constrained fog computing environments. Dans *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 71–81.

Pallewatta, S., Kostakos, V. et Buyya, R. (2022). Qos-aware placement of microservices-based iot applications in fog computing environments. *Future Generation Computer Systems*, *131*, 121–136.

Ponce, F., Márquez, G. et Astudillo, H. (2019). Migrating from monolithic architecture to microservices : A rapid review. Dans *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 1–7. IEEE.

PyPI (2022). gurobipy : Python interface to gurobi. https ://www.gurobi.com.

Richardson, C. (2015). Service discovery in a microservices architecture. https ://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/.

Richardson, C. (2016). Deploying microservices : Choosing a strategy. https ://www.nginx.com/blog/deploying-microservices/.

Richardson, C. (2018). *Microservices patterns : with examples in Java.* Simon and Schuster.

Roseboro, R. et Reading, H. (2016). Cloud-native nfv architecture for agile service creation & scaling. *White paper, Jan.*

Sampaio, A. R., Rubin, J., Beschastnikh, I. et Rosa, N. S. (2019). Improving

microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, *10*(1), 1–30.

Scherb, C., Grewe, D., Wagner, M. et Tschudin, C. (2018). Resolution strategies for networking the iot at the edge via named functions. Dans *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 1–6. IEEE.

Sifalakis, M., Kohler, B., Scherb, C. et Tschudin, C. (2014). An information centric network for computing the distribution of computations. Dans *Proceedings of the 1st ACM Conference on Information-Centric Networking*, 137–146.

Sonkoly, B., Czentye, J., Szalay, M., Németh, B. et Toka, L. (2021). Survey on placement methods in the edge and beyond. *IEEE Communications Surveys & Tutorials*.

Stec, A. (2022). Database design in a microservices architecture. https ://www.baeldung.com/cs/microservices-db-design.

Steiner, M., Gaglianello, B. G., Gurbani, V., Hilt, V., Roome, W., Scharf, M. et Voith, T. (2012). Network-aware service placement in a distributed cloud environment. Dans *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, p. 73–74., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/2342356.2342366

Wan, X., Guan, X., Wang, T., Bai, G. et Choi, B.-Y. (2018). Application deployment using microservice and docker containers : Framework and optimization. *Journal of Network and Computer Applications*, *119*, 97–109.

Wang, Y., Zhao, C., Yang, S., Ren, X., Wang, L., Zhao, P. et Yang, X. (2020a). Mpcsm : Microservice placement for edge-cloud collaborative smart manufacturing. *IEEE Transactions on Industrial Informatics*, *17*(9), 5898–5908.

Wang, Y., Zhao, C., Yang, S., Ren, X., Wang, L., Zhao, P. et Yang, X. (2020b). Mpcsm : Microservice placement for edge-cloud collaborative smart manufacturing. *IEEE Transactions on Industrial Informatics*, *17*(9), 5898–5908.

Wu, D., Chen, A., Ng, T. S. E., Wang, G. et Wang, H. (2019). Accelerated service chaining on a single switch asic. Dans *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, p. 141–149., New York, NY, USA. Association for Computing Machinery. http://dx.doi.org/10.1145/3365609.3365849

Zhao, H., Deng, S., Liu, Z., Yin, J. et Dustdar, S. (2020). Distributed redundancy scheduling for microservice-based applications at the edge. *IEEE Transactions on Services Computing*, 1–1. http://dx.doi.org/10.1109/TSC.2020.3013600

Zilberman, N. (2019). In-network computing. https ://www.sigarch.org/in-network-computing-draft.