Performance et gestion de ressources dans un cloud multi-virtualisé

Mathieu Bacou

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :

M. MATHIEU BACOU

le mardi 12 mai 2020

Titre :

Performance et gestion de ressources dans un cloud multi-virtualisé

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. DANIEL HAGIMONT

M. ALAIN TCHANA

Rapporteurs :

M. JEAN-MARC MENAUD, IMT ATLANTIQUE

M. PIERRE SENS, UNIVERSITE PARIS 6

Membre(s) du jury :

M. ANDRE LUC BEYLOT, TOULOUSE INP, Président

M. ALAIN TCHANA, ECOLE NORMALE SUP LYON ENS DE LYON, Membre

M. DANIEL HAGIMONT, TOULOUSE INP, Membre

Mme FABIENNE BOYER, UNIVERSITE GRENOBLE ALPES, Membre

Thèse

Performance et gestion de ressources dans un cloud multi-virtualisé

Mathieu Bacou

12 mai 2020

Directeurs : Pr Daniel Hagimont
Pr Alain Tchana

Co-directeur CIFRE*: M. Thierry Caminel

Doctorat de l'Université de Toulouse
Délivré par l'Institut National Polytechnique de Toulouse

*Thèse CIFRE n° 2016/1561 en collaboration avec Atos Origin Intégration.

Cette thèse est mise en page par LaTeX et KOMA-Script.

À mes parents, pour leur soutien sans limite.
À Mymy, pour tout ce qu'elle voit en moi.

Remerciements

Mes remerciements vont d'abord à mes parents, Colette et Didier. Vous m'avez toujours soutenu sans faille, me permettant de réaliser avec une grande sérénité mes longues études jusqu'à ce doctorat. Je vous en suis infiniment reconnaissant.

Je me tourne ensuite vers Valentin et Alexandre, mes meilleurs amis qui sont restés près de moi malgré mon caractère certain et mon opiniâtreté. Mais je préfère penser à vous avec nos histoires de cœur, nos voyages et nos vacances !

Bien sûr, c'est ici que je me tourne vers toi, Mylène. Bien que tu occupes une place dans ma vie depuis plus récemment, cette place est très importante pour moi. Les derniers mois de ma thèse auraient été bien différents sans toi, et je ne pourrais jamais assez te remercier pour ce point d'accroche que tu représentes pour moi.

Évidemment, c'est à vous tous Tom, Jade, Anthony, Laura, Alvin, que je dois ce bon temps passé en excellente compagnie, parfois à me questionner sur mon avenir, et surtout à simplement rire de tout ça. Merci à vous. Merci aussi à Grégoire, avec qui je me suis retrouvé dans cette aventure difficile mais dont nous avons réussi à voir le bout !

Je présente mes remerciements à mes directeurs de thèse, Daniel Hagimont et Alain Tchana pour leurs conseils efficaces et leur supervision ; ainsi qu'Atos Intégration et ses employés Thierry Caminel et Jean-Pierre Belmonte qui m'ont suivi durant cette thèse CIFRE. Merci bien sûr à nos secrétaires Sylvie Armengaud, Muriel Pernier et Annabelle Sansus, sans qui nos recherches seraient bien moins aisées. Je veux aussi exprimer toute ma gratitude envers MM. Pierre Sens et Jean-Marc Menaud, qui ont accepté d'être rapporteurs de ma thèse, ainsi qu'à tous les membres du jury.

Je remercie enfin tous mes collègues du laboratoire, à commencer par Boris Teabe qui ne m'a pas lâché depuis mon premier stage, ainsi que Vlad Nitu pour ce beau partage de bureau pendant un temps, et Brice Ekane pour sa bonne humeur immuable ; et puis Tu Ngoc, Patrick Lavoisier Wapet, Bao Bui, Djob Mvondo, Stella Bitchebe, Kevin Jiokeng…

Merci à vous tous pour vos rôles respectifs dans ma vie et dans ma thèse !

Résumé

Le cloud computing permet aux entreprises de réduire la barrière et les coûts d'utilisation de l'informatique, en mutualisant les besoins avec d'autres utilisateurs. Cette mutualisation est permise par la technologie de la virtualisation. Il s'agit de vendre les ressources physiques, concrètes, d'un centre d'hébergement, comme des ressources virtuelles, logiques. L'efficacité de la solution de virtualisation, sur différents critères, est au cœur des préoccupations à la fois des fournisseurs de cloud, et des clients.

Les premiers veulent servir le plus grand nombre de clients possibles avec les ressources physiques déjà disponibles. Il faut donc que la solution de virtualisation permette d'allouer les ressources sans gaspillage. La consommation énergétique est aussi un poste de dépense important qu'ils cherchent à minimiser. Les clients quant à eux, recherchent pour un coût minimal, des garanties de performances et de prédictibilité de celles-ci : la solution de virtualisation doit offrir le même niveau de performance malgré l'abstraction des ressources physiques en ressources virtuelles.

Les deux solutions de virtualisation principales sont les machines virtuelles et les conteneurs. Elles ont chacune leurs avantages et leurs inconvénients sur les axes de la gestion des ressources et des performances. Mais il est possible d'imbriquer les conteneurs dans les machines virtuelles, produisant ainsi un cloud multi-virtualisé, avec deux niveaux. Comment alors profiter au mieux des caractéristiques des deux solutions dans ce nouvel environnement ?

Cette thèse explore les problématiques de ces deux solutions et de leur combinaison, et propose des systèmes pour obtenir avec la multi-virtualisation de meilleures performances, une gestion des ressources approfondie et un coût réduit. Elle décrit premièrement une méthode de consolidation de la charge de travail au premier niveau, pour la réduction de la consommation énergétique ; deuxièmement, un algorithme d'allocation des ressources aux conteneurs du deuxième niveau, pour corriger leur problème inhérent de prédictibilité des performances ; et troisièmement, deux systèmes joints pour l'optimisation du réseau multi-virtualisé afin d'en améliorer les performances et l'utilisation des ressources, ainsi que de réduire le coût du cloud pour le client. Ces travaux agissent à tous les niveaux de la virtualisation imbriquée afin de propulser plus avant la technologie de la multi-virtualisation.

Abstract

Companies use cloud computing to lower entry and usage costs of using information technologies as a resource. The main feature of cloud computing that enables these lower costs is the pooling of resources with other users. It is the technology of virtualization that enables the pooling of resources. Its principle is to sell physical, concrete resources from a data-center as virtual, abstract resources. The core concern for both cloud providers and clients is the efficiency, on various axes, of the virtualization solution.

The former wish to serve as many clients as possible with the given physical resources of their data-centers. This puts an emphasis on the capability of the virtualization solution to allocate resources with limited waste. Another target for optimization is the power usage of data-centers. Efficient management of resources has a great effect on it. As for clients of cloud computing, they seek guarantees on performance, including predictability. Indeed, the virtualization solution must provide them with the same performance level despite abstraction of physical resources into virtual resources. Buying cloud resources as cheap as possible is also a priority.

There are two main virtualization solutions: virtual machines and containers. Both have their own sets of benefits and drawbacks, on the axes of resource management and performance. However, containers can be nested inside of virtual machines, thus building a two-level multi-virtualized cloud. How best to use both solutions in this new environment?

This thesis explores issues of both solutions and of their combination. It proposes new systems to gain better performance, to improve resource management and to provide cheaper cloud services using multi-virtualization. First, it describes a novel workload consolidation method at the first level to further reduce power usage; second, an algorithm to allocate resources to containers of the second level that fixes their intrinsic issue of performance predictability; and third, two joint systems that optimize multi-virtualized networking to improve performance and resource utilization, and save money on cloud usage. These works play at every level of nested virtualization in order to move the technology of multi-virtualization forward.

Table des matières

Résumé ix

Abstract xi

1. Introduction 1

2. Contexte et problématiques 5
2.1. Le cloud . . . 5
2.1.1. Présentation . . . 5
2.1.2. Modèle de déploiement . . . 8
2.1.3. Type de service . . . 10
2.1.4. Technologies . . . 13
2.2. La virtualisation . . . 16
2.2.1. Présentation . . . 16
2.2.2. Mode de virtualisation . . . 16
2.2.3. Comparaison des modes de virtualisation . . . 24
2.2.4. Applications de la virtualisation . . . 27
2.3. Problématiques . . . 28
2.3.1. Gestion des ressources virtualisées . . . 28
2.3.2. Performance et prédictibilité . . . 29
2.3.3. Problématiques dans le cloud doublement virtualisé 30
2.3.4. Contributions . . . 31

3. La consolidation de la charge de travail : Drowsy-DC 33
3.1. Introduction . . . 33
3.1.1. Efficacité énergétique . . . 34
3.1.2. Limites de la consolidation . . . 36
3.1.3. Problème de la mémoire virtualisée . . . 36
3.1.4. Nouvelle vision de la consolidation . . . 37
3.2. Notions préliminaires . . . 38
3.2.1. Caractérisation des VM : SLMU, LLMU, LLMI . . 40
3.2.2. États énergétiques . . . 40
3.3. Vue d'ensemble . . . 43

3.4. Consolidation en fonction de l'inactivité 44
3.4.1. Principe 44
3.4.2. Construction du modèle d'inactivité (MI) 45
3.4.3. Calcul de la probabilité d'inactivité (PI) 46
3.4.4. Intégration dans OpenStack 47
3.4.5. Synthèse 50
3.5. Mise à jour du modèle d'inactivité 51
3.5.1. Calcul des scores SI_X 51
3.5.2. Calcul des poids w_X 53
3.6. Suspension des serveurs 54
3.6.1. Détection de l'inactivité 55
3.6.2. Délai de sursis 56
3.7. Reprise des serveurs 56
3.7.1. Reprise sur requête entrante 57
3.7.2. Reprise planifiée 57
3.7.3. Reprise de serveurs doublement virtualisés 58
3.8. Optimisation de la reprise des serveurs 58
3.8.1. Reprise du noyau 60
3.8.2. Reprise de la carte réseau 61
3.9. Évaluation 61
3.9.1. Évaluation en environnement réel 61
3.9.2. Évaluation par simulation 71
3.10. État de l'art 74
3.10.1. États de faible énergie 74
3.10.2. Consolidation de la charge de travail 75
3.11. Conclusion 76

4. L'auto-configuration des applications conteneurisées 77
4.1. Introduction 77
4.2. Allocation du processeur aux conteneurs 80
4.2.1. Allocation par le moteur de conteneurs 80
4.2.2. Allocation par l'orchestrateur 81
4.3. Analyse : violation du principe tel-tel et solutions 82
4.3.1. Installation 82
4.3.2. Méthodologie 82
4.3.3. Résultats 83
4.3.4. Synthèse 85
4.4. Algorithme d'allocation hybride de la ressource processeur 85
4.4.1. Conception 87
4.4.2. Algorithme 88

4.4.3. Détermination de la catégorie de conteneur 92
4.4.4. Intégration à Kubernetes 92
4.4.5. Limites de l'algorithme et pistes d'amélioration . . 93
4.5. Évaluation 94
4.5.1. Méthodologie 95
4.5.2. Résultats 97
4.6. État de l'art 101
4.6.1. Performance des conteneurs 101
4.6.2. Violation du principe tel-tel 102
4.6.3. Gestion de la ressource processeur virtualisée . . . 102
4.7. Conclusion 103

5. La double virtualisation du réseau : BrFusion et HostLo 105
5.1. Introduction 105
5.2. Notions préliminaires 108
5.2.1. Pods 108
5.2.2. Virtualisation imbriquée du réseau 110
5.3. Analyse des défauts de la virtualisation réseau imbriquée . 110
5.3.1. Origine 112
5.3.2. Déploiement mono-VM de pods 112
5.3.3. Duplication du réseau virtualisé 114
5.4. HostLo : déploiement inter-VM de pods 115
5.4.1. Vue d'ensemble 115
5.4.2. Intégration 117
5.4.3. Implémentation 119
5.4.4. Mise à jour de l'algorithme de placement des pods 119
5.4.5. Gestion des autres ressources 120
5.4.6. Évaluation 121
5.5. BrFusion : déduplication de la virtualisation du réseau . . 133
5.5.1. Vue d'ensemble 133
5.5.2. Intégration 135
5.5.3. Implémentation 135
5.5.4. Évaluation 136
5.6. État de l'art 146
5.6.1. Réseautage entre les VM 146
5.6.2. Réseautage dans les environnements virtualisés . . 147
5.7. Conclusion 148

6. Conclusion et perspectives 149
6.1. Multi-virtualisation du cloud 149

6.2. Pistes d'amélioration des travaux 150
6.3. Vers le cloud Function-as-a-Service multi-virtualisé 151

A. Simulateur de Drowsy-DC **153**

B. Simulateur d'allocation hybride pour le principe tel-tel **155**

C. Pilote d'interface HostLo **157**

D. Modifications de QEMU pour HostLo **159**

E. Simulateur d'économies financières avec HostLo **161**

Bibliographie **163**

Table des figures

2.1. Ancien et nouveau paradigmes informatiques. 7
2.2. Modèles de déploiement du cloud. 8
2.3. Types de service cloud. 11
2.4. Technologies du cloud. 13
2.5. Virtualisation du matériel et types d'hyperviseurs. 18
2.6. Types de virtualisation. 18
2.7. Paravirtualisation assistée par le matériel. 20
2.8. Virtualisation du système d'exploitation. 21
2.9. Virtualisation imbriquée. 23

3.1. Proportionnalité énergétique. 35
3.2. Concept de Drowsy-DC. 39
3.3. Traces d'activité de VM LLMI dans un DC de cloud privé. 41
3.4. Architecture de Drowsy-DC. 44
3.5. Processus de consolidation avec Drowsy-DC. 49
3.6. Processus de reprise d'un serveur. 59
3.7. Phases de reprise du noyau Linux. 60
3.8. Proportion de temps de colocalisation de chaque VM. . . . 64
3.9. Utilisation des ressources du module de reprise. 67
3.10. Temps de reprise d'un serveur. 67
3.11. Efficacité du modèle d'inactivité. 70
3.12. Économies d'énergie. 72
3.13. Nombre de migrations. 73

4.1. Violation du principe tel-tel et conséquences. 79
4.2. Durées d'exécution de Stream. 84
4.3. Durées d'exécution du banc d'essai de CloudSuite. 84
4.4. Durées d'exécution des bancs d'essai PARSEC. 86
4.5. Groupes de processeurs pour l'allocation hybride. 89
4.6. Exemple du processus d'allocation. 90
4.7. Effets de l'optimisation de l'orchestrateur. 98
4.8. Passage à l'échelle de l'optimisation de l'orchestrateur. . . 100

5.1. Défauts de la virtualisation imbriquée. 106

5.2. Réseautage des pods. 109
5.3. Virtualisation imbriquée du réseau. 111
5.4. Défauts de la virtualisation imbriquée du réseau. 113
5.5. Performances de la virtualisation simple et imbriquée. . . . 116
5.6. Illustration de HostLo. 118
5.7. Économies financières de HostLo. 123
5.8. Installations des expériences d'évaluation de HostLo. . . . 125
5.9. Performance de Hostlo avec Netperf. 127
5.10. Performance de HostLo avec Memcached. 128
5.11. Latence de HostLo avec Memcached. 129
5.12. Performance de HostLo avec NGINX. 130
5.13. Utilisations processeur de Memcached et NGINX. 131
5.14. Illustration de BrFusion. 134
5.15. Intégration de BrFusion et HostLo. 136
5.16. Installations des expériences pour l'évaluation de BrFusion. 138
5.17. Performance de BrFusion avec Netperf. 141
5.18. Performance de BrFusion avec Kafka et NGINX. 142
5.19. Utilisations processeur de Kafka et NGINX. 144
5.20. Temps de démarrage d'un conteneur. 145

Liste des tableaux

2.1. Comparaison des principales techniques de virtualisation. . 25

3.1. États de sommeil ACPI. 42
3.2. Composants du modèle d'inactivité d'une VM. 47
3.3. Capacités des systèmes évalués. 62
3.4. Proportion de temps passé suspendu par chaque serveur. . 64
3.5. Traces d'activité de VM pour l'évaluation du MI. 68
3.6. Métriques d'efficacité de la prédiction. 68

4.1. Comparaison des mécanismes d'allocation de processeur. . 81

5.1. Tarifs des VM `m5` d'Amazon Web Services EC2. 122
5.2. Macro-bancs d'essai pour l'évaluation de HostLo. 126
5.3. Macro-bancs d'essai pour l'évaluation de BrFusion. 139

Liste des algorithmes

1. Allocation hybride de ressource processeur. 91

Chapitre 1

Introduction

Le support de l'informatique est fondamental pour tous les acteurs de notre société contemporaine. Pour les entreprises, pour les agents publics, pour les associations, il constitue tour à tour un outil de gestion, un support de travail ou le cœur même de l'activité. Son utilisation nécessite dans tous ces cas l'achat, la maintenance et la mise à disposition de la ressource informatique, ce qui constitue toujours un engagement de moyens important. La force de ces contraintes incite alors à externaliser.

C'est le principe du *cloud* : utiliser des ressources informatiques distantes. Les utilisateurs louent une part de *cloud* pour leurs usages personnels ; et les contraintes d'utilisation sont déportées sur le fournisseur du *cloud*, qui mutualise les besoins et les ressources. La location de ces ressources mutualisées correspond à une virtualisation, car à l'échelle des besoins d'un utilisateur, les ressources sont virtuellement infinies et toujours disponibles.

Le fournisseur de *cloud* s'assure de la garantie de ces deux caractéristiques. Il faut aussi qu'il tienne des promesses d'efficacité afin de réellement servir les besoins des utilisateurs. En effet, le déplacement de leurs services informatiques vers le *cloud* doit se faire sans heurt ; c'est un prérequis à sa pertinence. Ce sont donc deux problématiques fondamentales du *cloud* qui se dévoilent ici : d'une part, la bonne gestion des ressources virtuelles ; et d'autre part, la performance des applications qui y sont déplacées.

Par voie de conséquence, la virtualisation est une technologie pilier du *cloud*. Elle s'y retrouve sous plusieurs formes, appelées machines virtuelles et conteneurs, qui ont des avantages et des inconvénients variés et complémentaires. Afin de profiter de cette complémentarité, ou par simple contrainte technique, ces formes peuvent être imbriquées : les machines virtuelles constituent un premier niveau, et les conteneurs un second niveau de virtualisation. L'interface de cet empilement crée de nouvelles problématiques, mais aussi de nouvelles opportunités, autour des deux axes du *cloud* que sont la performance des applications et la gestion des ressources.

Comment les problématiques classiques du *cloud* virtualisé évoluent-elles dans un *cloud* multi-virtualisé ?

D'une part, les problèmes classiques de gestion des ressources dans les plate-formes cloud subsistent malgré l'empilement des couches de virtualisation. Il s'agit par exemple de la consolidation des machines virtuelles, du partage des ressources entre elles, ou encore du maintien de la prédictibilité des performances. D'autre part, de nouveaux problèmes apparaissent du fait de l'empilement des couches, notamment la duplication des services entre les différents niveaux de la multi-virtualisation, ou le manque de coordination entre ceux-ci au sujet de la gestion des ressources. Cette thèse se positionne dans le contexte d'une plate-forme multi-virtualisée, mais elle ne prétend pas en corriger tous les problèmes. Elle donne une analyse et propose une solution pour trois d'entre eux.

La première contribution concerne la consolidation de la charge de travail au niveau inférieur de la multi-virtualisation, c'est-à-dire les machines virtuelles. Son but est d'exploiter la capacité de mise en veille des serveurs inactifs. Pour ce faire, une extension au système de consolidation classique est conçue, qui vise à favoriser le regroupement de machines virtuelles avec des périodes d'inactivité correspondantes. Elle crée ainsi des serveurs aux périodes d'inactivité conséquentes, et ceux-ci peuvent être mis en veille et relancés à la reprise de l'activité.

La deuxième contribution s'attaque au problème de l'allocation de la ressource processeur aux conteneurs, dans la couche supérieure de la multi-virtualisation. On constate en effet une disparité entre l'allocation de processeur donnée à un conteneur, et l'information sur la ressource processeur disponible relevée par l'application conteneurisée. Cette disparité entraîne une auto-configuration erronée de l'application, dont l'effet est l'imprédictibilité de ses performances. La solution proposée est un algorithme hybride d'allocation de la ressource processeur, qui utilise deux mécanismes d'allocation différents pour corriger ce problème avec un minimum de contraintes.

Enfin, la troisième contribution consiste en deux systèmes joints qui corrigent des problèmes du réseau multi-virtualisé. Le premier système apporte une meilleure utilisation des ressources doublement virtualisées. Il étend la virtualisation du réseau des conteneurs au travers du cloisonnement des machines virtuelles, et permet en conséquence de mieux répartir les conteneurs sur les ressources qu'elles fournissent. Le second déduplique la virtualisation du réseau qui est réalisée indépendamment par chaque couche de la multi-virtualisation. Il élimine ainsi l'impact de cette virtualisation dupliquée du réseau, qui est une dégradation des performances.

Production scientifique

Les résultats et travaux scientifiques de cette thèse ont été publiés sous les références suivantes :

— Mathieu Bacou, Grégoire Todeschi, Alain Tchana, Daniel Hagimont, Baptiste Lepers et Willy Zwaenepoel. « Drowsy-DC : Data Center Power Management System ». In : *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019*. IEEE, mai 2019, p. 825-834

— Mathieu Bacou, Alain Tchana et Daniel Hagimont. « Your Containers Should be WYSIWYG ». In : *2019 IEEE International Conference on Services Computing, SCC 2019*. IEEE, juil. 2019, p. 56-64

— Mathieu Bacou, Grégoire Todeschi, Alain Tchana et Daniel Hagimont. « Nested Virtualization Without the Nest ». In : *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019*. ACM, août 2019, 12:1-12:10

Contenu

Le chapitre 2 propose une description en profondeur du contexte de la thèse, en décrivant le *cloud* et la technologie qui le meut : la virtualisation. Il conclut sur les problématiques auxquelles la thèse tente de répondre.

Les contributions sont exposées dans les chapitres 3 à 5. Ils en présentent chacun le contexte spécifique, avant d'exposer les travaux correspondants et de finir avec leur évaluation.

Tout d'abord, le chapitre 3 est dédié à un nouveau système de consolidation de la charge de travail au premier niveau de la multi-virtualisation. Ensuite, le chapitre 4 propose une solution au problème de la prédictibilité des performances dû à la virtualisation du deuxième niveau. Il est suivi du chapitre 5 qui décrit un double système pour améliorer la situation du réseau dans la multi-virtualisation.

Enfin, le chapitre 6 conclut ces développements, avant de proposer une vision pour le futur de la multi-virtualisation dans le *cloud*.

Chapitre 2

Contexte et problématiques

2.1. Le cloud 5
2.2. La virtualisation 16
2.3. Problématiques 28

On décrit ici le contexte de la thèse : le *cloud computing*, qui repose sur la virtualisation. Ensuite de quoi on donne les problématiques qu'elle traite.

2.1. Le cloud

Le nom *cloud computing* est utilisé dans de nombreux contextes, pour désigner des services très divers. Cette section s'attache à éclaircir sa définition, sa place dans l'industrie et les technologies qui le rendent possible.

2.1.1. Présentation

L'informatique est un pilier technologique de l'économie et de la société moderne. C'est une ressource, exploitée par la vie de chaque personne et par l'activité de chaque institution. Pour les entreprises, pour les agents publics, pour les associations, elle constitue tour à tour un outil de gestion, un support de travail ou le cœur même de l'activité.

C'est une ressource, produite par tout ce que l'on peut considérer comme un ordinateur : téléphone intelligent, poste personnel, serveur, centre d'hébergement, etc. Sa demande ne cesse de croître ; et si sa production augmente elle aussi, les acteurs peinent à assouvir seuls leurs propres besoins. Les coûts d'acquisition, de maintien et de mise à disposition de la ressource informatique restent prohibitifs, d'autant plus lorsque ces tâches n'ont pas

de rapport direct avec leurs activités. Elle est pourtant nécessaire. La puissance de ces contraintes incite alors à *l'externaliser*.

C'est le principe de l'informatique en nuage, plus couramment désignée par l'expression anglaise *cloud computing* et simplement raccourcie en *cloud*. Le paradigme du cloud est *d'utiliser par l'intermédiaire d'un réseau, des ressources informatiques distantes.* Concrètement, l'utilisateur d'une solution de cloud loue une quantité de ressource informatique auprès de son centre d'hébergement. Cette location est prise le temps d'assouvir le besoin de l'utilisateur. Ce scénario présente une grande différence avec l'ancien paradigme : au lieu d'acheter physiquement, lorsque le besoin émerge, des serveurs informatiques ou d'en demander l'accès de manière quasi-exclusive ; le cloud propose de *mutualiser* tous les besoins en une seule plate-forme qui met à disposition l'ensemble de ses ressources à tous les utilisateurs en même temps. Dans ce cadre, un fournisseur de cloud a pour rôle de fournir ou d'aider à fournir cette plate-forme de cloud. Ce changement de paradigme est illustré en figure 2.1.

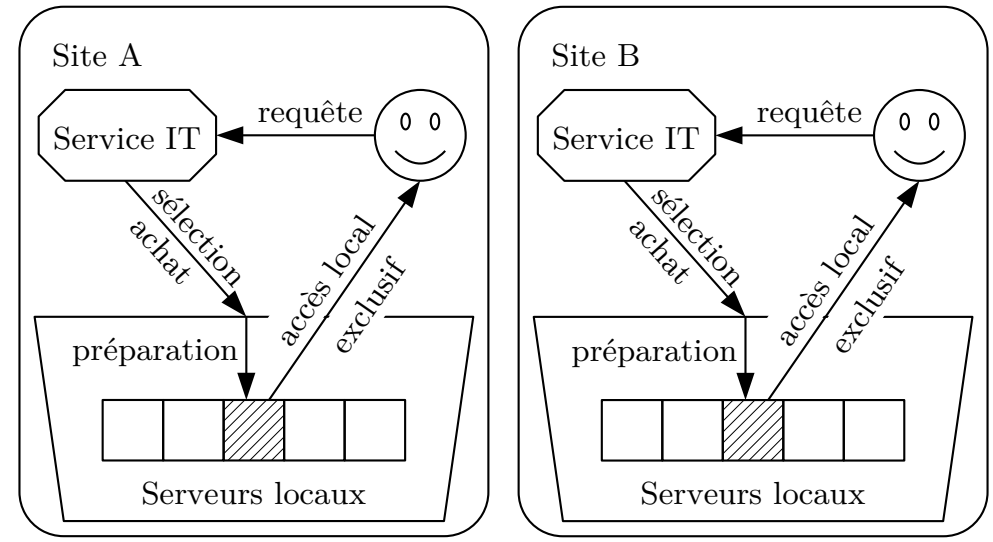
On voit que c'est la mutualisation qui fait la force du cloud. Les contraintes liées au support informatique mentionnées plus haut sont affaiblies par la centralisation de la ressource informatique : le centre d'hébergement s'occupe de toujours avoir assez de ressource informatique disponible, et gère sa mise à disposition. La mutualisation a donc un intérêt organisationnel, en simplifiant et en optimisant l'accès à la ressource ; et surtout financier, en l'externalisant à différents niveaux.

Citons deux exemples d'utilisation du cloud. Ils décrivent des situations radicalement différentes, dont le cloud est à chaque fois une solution. Il faut y distinguer d'une part le modèle de déploiement de la plate-forme cloud ; et d'autre part le type de service qu'elle met à disposition.

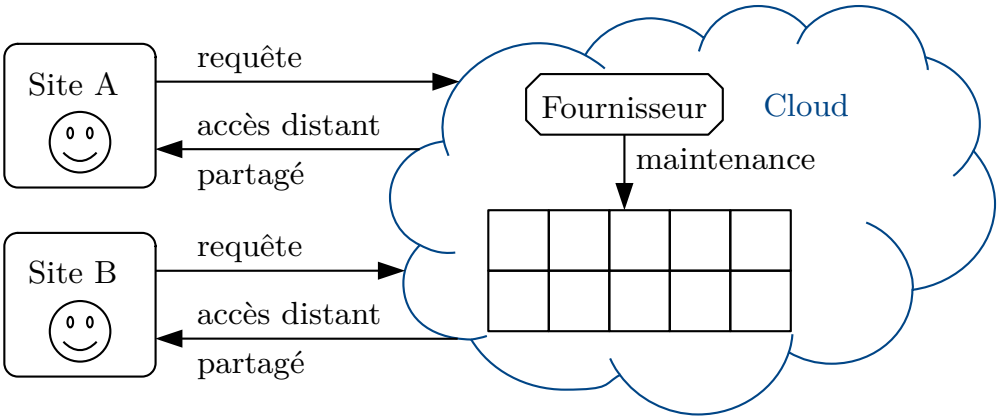
Exemple 1. *Une association municipale souhaite organiser un évènement. Pour ce faire, elle doit en établir le planning avec la participation de tous ses membres. Un simple document texte suffit, mais son partage et son édition par plusieurs parties éparpillées dans toute la ville est un problème difficile qui nécessite une solution informatique. L'association n'a aucun intérêt à gérer son propre serveur à cette fin.*

Ils utilisent donc un service cloud d'édition de documents en ligne. Le document est hébergé sur un serveur distant appartenant au fournisseur de cloud, et ce dernier fournit aussi le logiciel d'édition, hébergé de même sur ses serveurs.

Exemple 2. *Les employés d'une entreprise d'ingénierie hydraulique, avec plusieurs sites partout en France, ont régulièrement besoin d'exécuter des*



(a) Ancien paradigme : centres d'hébergement locaux



(b) Nouveau paradigme : le cloud

FIGURE 2.1. – Comparaison de l'ancien paradigme informatique (haut), avec des centres d'hébergement locaux et peu flexibles ; et du nouveau paradigme informatique du cloud (bas), avec un centre d'hébergement mutualisé flexible.

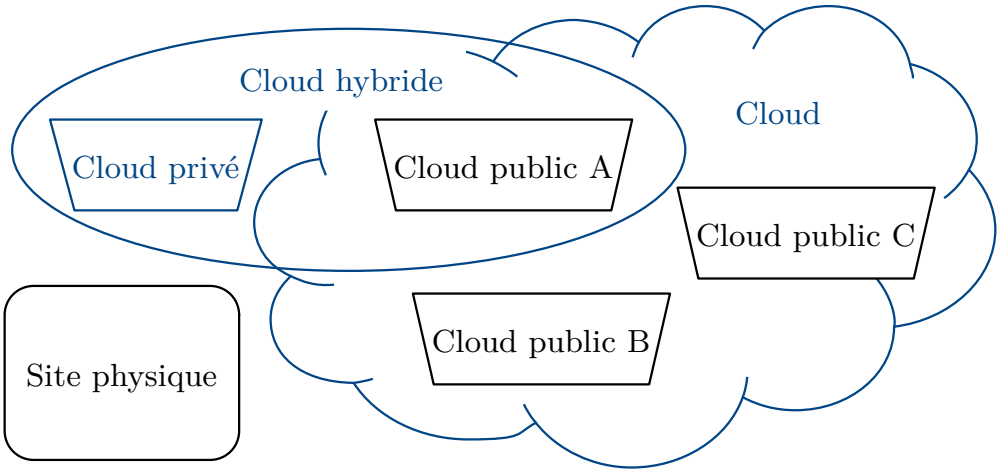


FIGURE 2.2. – Modèles de déploiement du cloud : privé, public et hybride.

simulations. Auparavant, chaque site possédait son propre centre d'hébergement à taille réduite, c'est-à-dire une pièce climatisée avec une poignée de serveurs de calcul. Les employés qui souhaitaient simuler leurs travaux devaient demander un accès temporaire à un serveur libre de leur site, au service dédié à l'administration des serveurs. Ils devaient s'assurer avant de lancer la simulation, que l'environnement d'exécution n'avait pas été perturbé par l'utilisateur précédent.

Désormais, c'est le seul site de Toulouse qui possède un centre d'hébergement dimensionné pour l'ensemble de l'entreprise. Cette plate-forme de cloud est mise à disposition de tous les employés, quel que soit leur site de rattachement. Ceux-ci ont accès à une interface en ligne par laquelle ils peuvent déployer leur simulation dans un environnement déjà préparé et toujours neuf. Leur simulation est placée dans une file d'attente et sera exécutée automatiquement dès qu'assez de ressources seront disponibles.

On décrit dans les sections qui suivent, d'abord les modèles de déploiement en section 2.1.2, puis les types de services en section 2.1.3.

2.1.2. Modèle de déploiement

Le modèle de déploiement correspond à l'ouverture de l'accès à la plate-forme. Les différents modèles sont illustrés en figure 2.2.

Cloud privé

Une plate-forme de cloud privé est mise en place uniquement pour *un seul client*. Le fournisseur de cloud peut être celui qui fournit la solution logicielle pour convertir le centre d'hébergement interne existant en plate-forme cloud (et peut être l'utilisateur de la plate-forme lui-même) ; ou alors c'est lui qui héberge la plate-forme en donnant l'exclusivité à son client.

L'intérêt est ici l'aspect *organisationnel* de la mutualisation plus que l'aspect financier, puisque l'utilisateur possède le centre d'hébergement cloud, ou achète une solution personnalisée. Le caractère privé aide à la sécurité des traitements et des données.

L'exemple 2 décrit un cloud privé. [1] Des exemples de l'industrie sont le cloud privé d'Intel [25], ou les solutions de cloud privé OpenNebula [100].

Cloud public

À l'inverse, la plate-forme de cloud public est proposée *à tout le monde* sans distinction, et l'offre est plus générique en offrant un grand panel de fonctionnalités. En général, le fournisseur de cloud possède un centre d'hébergement qu'il met à disposition en tant que plate-forme cloud à des clients qui se partagent ses ressources et ses services.

C'est l'aspect *financier* de la mutualisation qui prime ici, puisque la ressource informatique est totalement externalisée. Les coûts de maintenance du centre d'hébergement disparaissent, et seuls les besoins finaux de l'utilisateur sont facturés (si le service n'est pas gratuit). Du fait du partage entre tous les utilisateurs cependant, le cloud public peut être moins fiable et moins sécurisé que le cloud privé.

L'exemple 1 décrit l'utilisation d'un cloud public. Des exemples de l'industrie sont les clouds publics Amazon Web Service [2], Microsoft Azure [122] ou encore Google Cloud [121].

Cloud hybride

Une combinaison de cloud privé et de cloud public est parfois souhaitable. Il s'agit d'allier un cloud privé, plus onéreux et limité au strict nécessaire, à un cloud public, qui apportent bien plus les intérêts du cloud. Le cloud public sert ici à *étendre les capacités limitées du cloud privé* (en termes

1. On peut éventuellement considérer cet exemple comme un cloud communautaire, c'est-à-dire un cloud semi-public réservé à un nombre restreint de membres, ici les sites physiques de l'entreprise. Cette distinction n'a pas d'importance.

de ressource informatique ou de fonctionnalités), celui-ci restant nécessaire pour des raisons de sécurité par exemple.

L'exemple 2 pourrait être étendu pour décrire un cloud hybride : l'entreprise pourrait choisir d'utiliser ponctuellement un cloud public afin d'étendre sa capacité de simulation pendant des périodes d'activité intense.

2.1.3. Type de service

Le type de service correspond à la forme de la ressource informatique proposée. Les différentes formes se distinguent notamment par le *niveau d'abstraction* de la ressource informatique. Plus elle est abstraite au client, et moins ce dernier a de gestion à sa charge ; c'est le fournisseur de cloud qui gère plus d'aspects de la ressource à sa place. Cela passe par des technologies décrites en section 2.1.4. Cet équilibre des préoccupations entre le client et le fournisseur est illustré en figure 2.3.

Infrastructure-as-a-Service (IaaS)

Le niveau le plus bas consiste à louer l'infrastructure, c'est-à-dire *un accès quasi-direct au serveur*. Concrètement, l'utilisateur se voit attribuer une portion de ressource du centre d'hébergement cloud sous une forme telle qu'il semble avoir la possession d'un véritable serveur. Ainsi, il se trouve responsable du déploiement et de la maintenance de l'environnement qu'il va utiliser (installation d'un système d'exploitation par exemple). Ce niveau de contrôle le plus grand apporte donc une plus grande charge à l'utilisateur, ce qui diminue l'intérêt du cloud. Il reste néanmoins désirable pour simplement externaliser le centre d'hébergement.

L'exemple 2 ne décrit pas un cloud IaaS ; ce pourrait être le cas si les ingénieurs n'avaient accès qu'à la ressource informatique brute et devaient gérer eux-mêmes leur environnement de simulation. Des exemples dans l'industrie de l'offre IaaS sont les services Amazon Elastic Compute Cloud [7], ceux du Google Compute Engine [29] ou encore IBM Cloud Virtual Servers [56].

Platform-as-a-Service (PaaS)

À un niveau plus élevé d'abstraction de la ressource, le fournisseur de cloud peut vendre des *environnements de déploiement*. C'est lui qui les maintient, et l'utilisateur déploie simplement son application conçue avec des technologies supportées par la plate-forme (qu'il aura développée lui-même ou bien même obtenue du fournisseur de cloud ou d'une tierce partie).

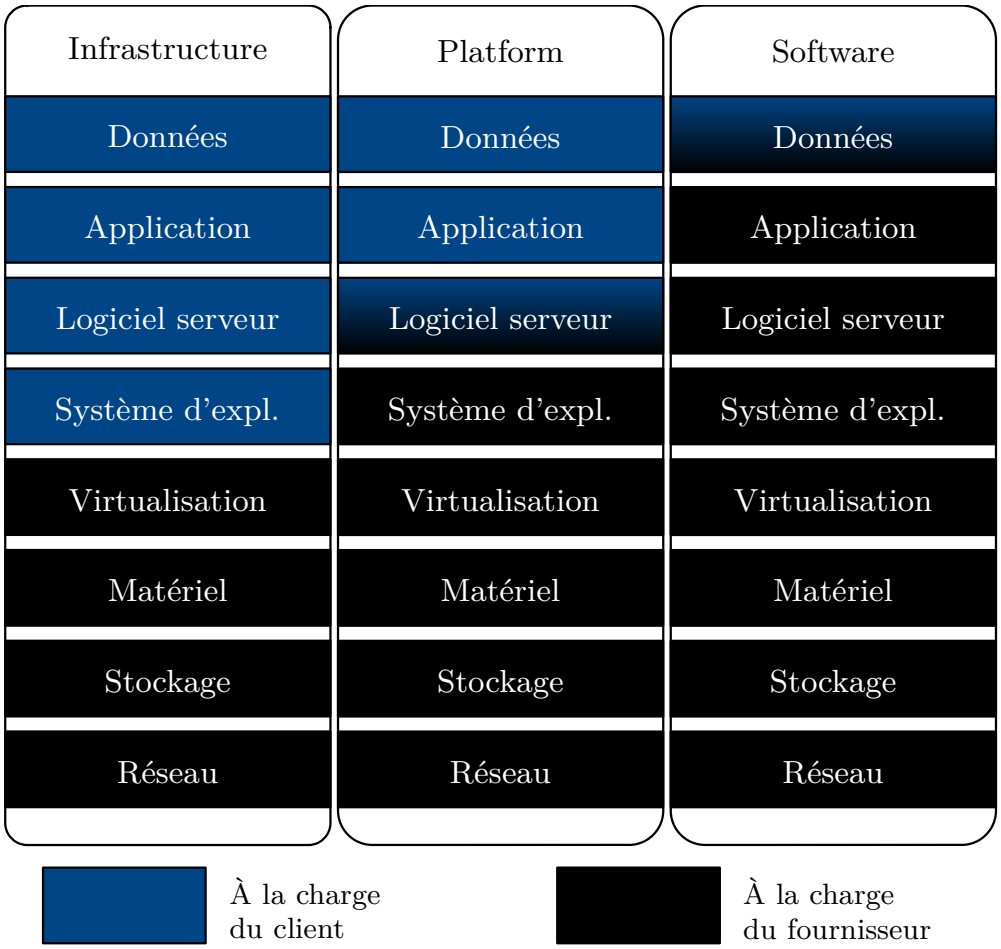


FIGURE 2.3. – Types de service cloud : Infrastructure-, Platform- et Service-as-a-Service ; et répartition des préoccupations.

Ce type de service voit le plus d'évolution actuellement, car c'est celui qui décharge le plus l'utilisateur des tâches de gestion, tout en lui permettant d'utiliser le cloud pour ses propres applications.

L'exemple 2 décrit un cloud PaaS : les ingénieurs ont accès à un environnement contrôlé par le fournisseur pour exécuter leurs simulations, et c'est aussi le fournisseur qui planifie l'exécution de celles-ci en cas d'affluence. Des exemples dans l'industrie de l'offre PaaS sont Amazon Elastic Beanstalk [8], Microsoft App Service [9] ou encore Google App Engine [6].

Par ailleurs, l'implémentation du type PaaS peut prendre de nombreuses formes, qui ont une influence sur le paradigme de déploiement. Lorsque la plate-forme fournie permet de déployer une application construite en micro-services dans des conteneurs (voir section 2.2.2), on peut l'appeler *Container-as-a-Service* (CaaS).

Les plates-formes *Function-as-a-Service* (FaaS) proposent quant à elles de déployer une application comme un ensemble de fonctions élémentaires. C'est le chaînage et l'exécution massivement parallèle de ces fonctions qui assurent le service de l'application. En fait, l'utilisateur n'a plus à se soucier de la répartition des requêtes reçues par l'application (voir plus bas, section 2.1.4), de la gestion des données produites, ni de la communication entre les divers composants, etc. C'est la plate-forme qui fournit ces fonctionnalités de serveurs, on parle ainsi de *serverless* (littéralement, « sans serveur »). On peut donc voir le FaaS comme un PaaS qui fournit des *environnements d'exécution* plutôt que des environnements de déploiement.

Software-as-a-Service (SaaS)

Enfin, le niveau le plus haut d'abstraction de la ressource informatique propose directement à l'utilisateur des applications, voire des données, hébergées par le fournisseur de cloud. La location de ressource est indirecte : c'est le fait d'utiliser le service qui consomme de la ressource.

L'exemple 1 décrit un cloud SaaS : les membres de l'association utilisent un éditeur de documents en ligne, qui stocke le document sur un service de stockage cloud géré par le même fournisseur. Des exemples dans l'industrie de l'offre SaaS sont la suite bureautique Google G Suite [48], incluant par exemple GMail et Google Documents ; celle de Microsoft Office 365 [87] ; ou encore tous les services de Framasoft [33].

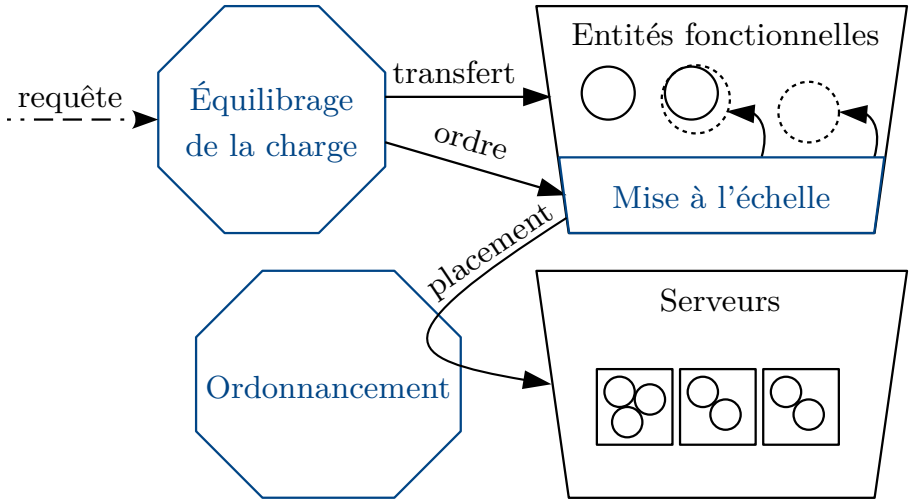


FIGURE 2.4. – Technologies du cloud : rôles du partage de la ressource informatique, de l'équilibrage de la charge, de la mise à l'échelle et de l'ordonnancement.

2.1.4. Technologies

L'avantage de la mutualisation proposée par le cloud est possible grâce à différentes technologies, principalement de partage de la ressource informatique et d'accès distant. Bien que le réseau soit au centre du chapitre 5, les technologies de l'accès distant ne seront pas abordées dans ce document, de même que l'on n'approchera pas le sujet du stockage de masse des données pour le cloud. D'autres technologies annexes mais essentielles, souvent proposées par les fournisseurs de cloud, sont présentées ici. Leurs rôles dans le cloud sont illustrés en figure 2.4.

Partage de la ressource informatique

La technologie fondamentale du cloud est le partage des ressources du cloud. Les fonctionnalités présentées ci-dessous reposent sur elle. Son objectif est de partager entre tous les utilisateurs, les *ressources physiques* qui ont été mutualisées. Ils sont en quelque sorte en compétition pour les ressources, et ne peuvent souvent pas se faire confiance entre eux (par exemple dans le cas d'un cloud public). De plus, les types IaaS et PaaS (voir section 2.1.3) proposent aux utilisateurs de déployer leurs propres applications, voire leurs propres environnements, comme s'ils avaient le monopole d'un véritable serveur. En fait, les utilisateurs louent auprès

du fournisseur de cloud des *ressources logiques*, qui n'existent pas telles quelles en réalité. Ces ressources sont virtuelles ; la technologie qui permet le partage de la ressource informatique du cloud est la *virtualisation*, à laquelle la section 2.2 est dédiée.

Mise à l'échelle et équilibrage de la charge

Maintenant que la ressource peut être partagée, il faut pouvoir effectivement l'utiliser pour déployer une application. C'est l'aspect du *dimensionnement de l'application*, qui englobe deux problématiques : la mise à l'échelle, et l'équilibrage de la charge.

mise à l'échelle capacité à allouer plus de ressource pour continuer à assurer le service malgré une augmentation de la charge de travail (*scaling*) ;

équilibrage de la charge répartition intelligente de la charge de travail entre les répliques d'entités fonctionnelles de l'application (*load balancing*).

Concernant la mise à l'échelle, on en distingue deux types :

horizontale répliquer les entités fonctionnelles de l'application,

verticale allouer plus de ressource aux entités fonctionnelles existantes ;

L'exemple 2 permet d'illustrer ces problématiques. On suppose que des simulations peuvent être exécutées simultanément dans le même environnement. Sous cette hypothèse, une affluence subite de simulations peut être gérée soit en augmentant les ressources allouées à un environnement de simulation déjà déployé, soit en déployant un nouvel environnement.

Ces deux technologies reposent sur deux caractéristiques du cloud :

1. flexibilité : les ressources sont louées et libérées à la demande ;
2. infinité : les ressources semblent infinies, car celles du centre d'hébergement cloud sont plusieurs ordres de grandeur plus vastes que les demandes des clients.

En fournissant des ressources logiques, c'est le partage de la ressource informatique par la virtualisation (voir section 2.1.4 ci-dessus) qui fournit ces caractéristiques.

On notera que l'établissement *d'une politique combinant mise à l'échelle et équilibrage de la charge* est très important, afin de :

— minimiser les ressources : un mauvais dimensionnement de l'application introduit un gaspillage, qui a des impacts économiques sur le fournisseur, et environnementaux par la consommation électrique ;
— minimiser le coût du cloud pour le client : le gaspillage a aussi un impact économique sur le client ;
— garantir de bonnes performances : par exemple, la mise à l'échelle horizontale nécessite d'attendre la création de la nouvelle entité fonctionnelle de l'application, son utilisation trop fréquente introduit donc une latence qui impacte les performances des applications du client.

Ordonnancement

Le partage des ressources du cloud entre tous ses utilisateurs nécessite un *arbitrage* : comment servir les requêtes en ressource de tous les utilisateurs ?

Cette tâche est l'ordonnancement (en anglais *scheduling*), qui est accomplie par un ordonnanceur (*scheduler*). Elle consiste à décider quel serveur du centre d'hébergement cloud va fournir les ressources pour une requête (par exemple à la création d'une nouvelle entité fonctionnelle d'une application, demandée par la mise à l'échelle horizontale, voir section 2.1.4). C'est-à-dire que le rôle de l'ordonnanceur est *d'attribuer des ressources physiques, aux ressources logiques* vendues par la plate-forme cloud.

Un but de l'ordonnancement est de servir toutes les requêtes, en allouant intelligemment les ressources de chaque serveur pour éviter le gaspillage ; il en va de l'efficacité de la mutualisation permise par le cloud. Mais l'ordonnancement est un problème difficile qui inclut notamment des notions d'équité et de latence de service dès lors que les profils de requêtes des utilisateurs se complexifient. De plus, il doit prendre en compte d'autres contraintes que la quantité de ressources disponibles : par exemple, certaines catégories de charges de travail ne devraient pas être placées sur le même serveur physique au risque de dégrader les performances.

L'objectif d'allocation intelligente est facilité par le partage de la ressource informatique en ressources logiques (voir section 2.1.4) pour :

— allouer plus finement, en estimant le besoin réel de l'utilisateur pendant l'exécution de son application ;
— sur-allouer les ressources des serveurs, en supposant que les utilisateurs demandent en réalité plus de ressource qu'ils n'en ont besoin afin de soutenir des pics de charge.

2.2. La virtualisation

La virtualisation est la technologie fondamentale du cloud, comme décrit dans la section 2.1.4. Elle permet de décorréler les ressources logiques vendues à l'utilisateur, des véritables ressources physiques du serveur dans le centre d'hébergement. C'est en cela que les ressources sont virtualisées.

2.2.1. Présentation

De manière générale, la virtualisation consiste donc à exposer des ressources physiques sous la forme de ressources logiques. Par exemple, les processeurs d'un serveur sont virtualisés en processeurs virtuels (vCPU) ; et une interface réseau est multiplexée en utilisant un pont (*bridge*), c'est-à-dire un commutateur réseau (*network switch*) virtuel.

La virtualisation est assurée par une couche d'abstraction matérielle. Elle porte différents noms selon le mode de virtualisation (voir section 2.2.2 plus bas), mais c'est toujours elle qui produit les ressources logiques à partir des ressources concrètes. De par la nature du cloud, qui propose un accès distant à des ressources logiques, la virtualisation efficace du réseau est essentielle. C'est donc une préoccupation majeure de la couche d'abstraction.

Celle-ci a de plus pour rôle d'administrer des *invités* : ce sont les entités virtuelles qui utilisent les ressources logiques. Ces entités portent aussi des noms différents selon le mode de virtualisation. Ce rôle d'aministration consiste à isoler les invités les uns par rapport aux autres, mais aussi à les isoler de la couche d'abstraction elle-même. Ce cloisonnement répond à des enjeux de sécurité et de performance par le partitionnement des ressources, qui sont plus amplement décrits dans la section 2.3.

Enfin, le principe général d'utilisation de la virtualisation repose sur des *images*. Celles-ci sont déployées dans une entité virtualisée, pour installer l'invité où résident l'application de l'utilisateur ainsi que tout ce qui est nécessaire à son fonctionnement. Une fois de plus, la notion d'image dépend fortement du mode de virtualisation.

2.2.2. Mode de virtualisation

La virtualisation est une technologie dont le domaine d'utilisation est plus vaste que le cloud. On ne présente ici que les modes de virtualisation couramment rencontrés comme supports au cloud.

Virtualisation du matériel

Le mode de virtualisation que l'on trouve le plus souvent dans le cloud, quel que soit le type de service, est la virtualisation du matériel. Ce mode permet de *faire cohabiter plusieurs systèmes d'exploitation (SE) en même temps, sur un même serveur physique*. C'est une tâche difficile car le rôle d'un système d'exploitation est de prendre le contrôle de l'entièreté de la machine et d'arbitrer l'accès à ses ressources entre les processus qu'il exécute. La virtualisation du matériel prend donc en quelque sorte la place d'un système d'exploitation pour d'autres SE, à un niveau plus proche du matériel que ces derniers.

Cette place de « SE pour des SE » est remplie par un *hyperviseur*. Il gère des *machines virtuelles* (en anglais *virtual machines*, ou VM), qui fournissent donc des représentations virtuelles des ressources à des SE *invités*. Comme l'hyperviseur permet d'administrer des VM, on le désigne aussi comme *gestionnaire de machines virtuelles* (en anglais *virtual machine manager*, ou VMM).

Hyperviseur On catégorise les hyperviseurs en deux types,[2] illustrés en figure 2.5 :

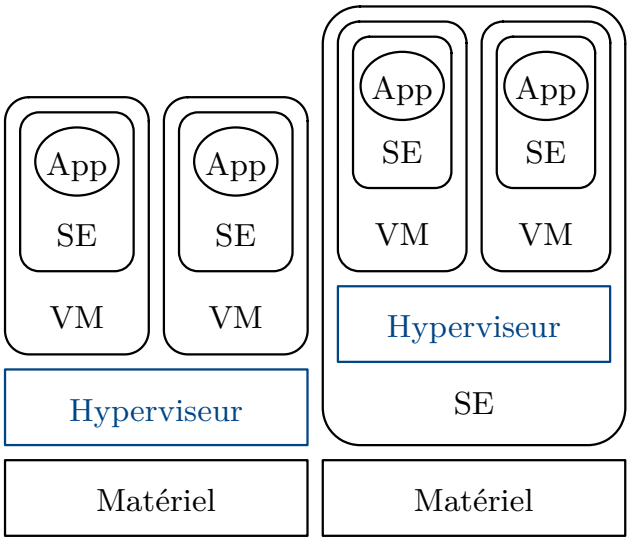
Type 1, natif l'hyperviseur s'exécute directement sur le matériel (en anglais *bare metal*, ou « métal nu »), il est comme un SE et les SE invités sont comme ses processus ;

Type 2, hébergé l'hyperviseur est un processus d'un SE normal, les SE invités sont donc à un troisième niveau au-dessus du matériel.

Des exemples d'hyperviseurs de type 1 sont Xen [139], KVM [68], Microsoft Hyper-V [55] ou encore VMware ESXi [43]. Le type 2 n'est généralement pas rencontré dans le cloud : il est utile pour une utilisation locale sur un ordinateur personnel, mais un fournisseur de cloud préfèrera gérer ses serveurs avec un hyperviseur natif. En effet, le gain de performance est majeur et la relative difficulté de gestion du type 1 n'est pas rédhibitoire dans ce cas. Des exemples d'hyperviseurs de type 2 sont VMware Workstation [137], VirtualBox [102] ou encore QEMU [110].

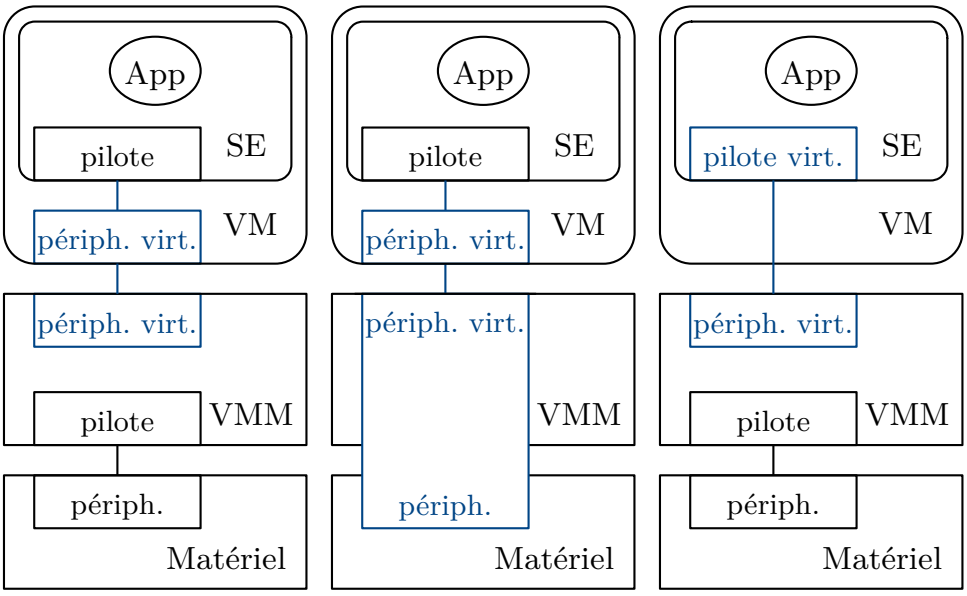
Orthogonalement aux types d'hyperviseur, il existe trois types de virtualisation pour les VM, illustrés en figure 2.6 :

2. La distinction entre les deux est parfois floue, comme pour KVM qui est un module du noyau Linux.



(a) Type 1 : natif (b) Type 2 : hébergé

FIGURE 2.5. – Virtualisation du matériel et types d'hyperviseurs.



(a) Totale (b) Assistée du matériel (c) Paravirtualisation

FIGURE 2.6. – Types de virtualisation. Les éléments en bleu sont des pilotes ou périphériques modifiés pour la virtualisation.

totale le matériel est simulé par l'hyperviseur jusqu'au point où un SE invité peut s'exécuter dans une VM sans aucune modification ;

assistée par le matériel le matériel inclut des capacités de virtualisation afin d'aider l'hyperviseur à administrer des VM qui peuvent accueillir des SE invités non modifiés ;

paravirtualisation l'hyperviseur ne simule pas le matériel mais propose une interface de virtualisation à travers une VM, qui est utilisée par le SE invité qui aura été spécialement modifié en ce sens.

La virtualisation totale est la moins performante puisque l'hyperviseur doit simuler tout le matériel requis par le SE invité, y compris par exemple la mémoire, rendant coûteux chaque accès à celle-ci. L'assistance du matériel permet comme son nom l'indique, de décharger une partie de cette simulation de l'hyperviseur vers le matériel, offrant une efficacité accrue. Enfin, *la paravirtualisation est généralement la technique la plus efficace*, en évitant le besoin d'émuler le matériel pour le SE invité car celui-ci a connaissance de sa virtualisation. Cette connaissance se traduit par la conversion d'instructions d'accès au matériel, en instructions d'accès aux ressources virtuelles de l'hyperviseur.[3] On notera que la paravirtualisation peut être couplée à l'assistance du matériel, et il est en fait courant que l'hyperviseur implémente son interface paravirtualisée avec l'assistance du matériel. Cette combinaison est illustrée en figure 2.7.

Machine virtuelle Une machine virtuelle en cours d'exécution est représentée par l'état du matériel qu'elle virtualise : données en mémoire centrale, queues d'entrées-sorties, drapeaux et registres du processeur, etc. Mais une VM simule en particulier un support de stockage à partir duquel s'exécute le SE invité (généralement et pour citer un exemple simple, un disque dur). En fait, on résume souvent une VM à son *image* disque, car elle suffit pour la dupliquer et la recréer. L'état de la VM en cours d'exécution est utilisé pour la suspendre et reprendre son exécution au même point (c'est-à-dire sans la redémarrer), notamment pour la migration (voir section 2.2.4).

3. Au lieu d'utiliser des instructions matérielles, le SE invité utilise des hypercalls, c'est-à-dire des instructions privilégiées à l'accès contrôlé, pour communiquer avec l'hyperviseur. Le parallèle est fait avec les processus d'un SE qui accèdent aux ressources par le biais de syscalls.

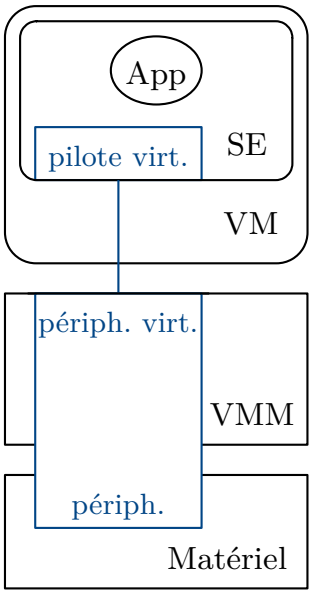


FIGURE 2.7. – Paravirtualisation assistée par le matériel. Combinaison des types de virtualisation illustrés en figures 2.6b et 2.6c.

Système d'exploitation invité L'objectif de la virtualisation est de déployer facilement une application, mais la virtualisation du matériel requiert un SE invité dans la VM. Cet intermédiaire nécessaire peut être *optimisé pour la virtualisation* de l'application. Une direction d'optimisation est la paravirtualisation présentée ci-dessus, puisque le SE invité a la connaissance d'être virtualisé et ne requiert donc pas la coûteuse émulation du matériel.

Une autre direction est l'utilisation d'un système d'exploitation bibliothèque (en anglais *library operating system*) afin de créer un *unikernel* (soit un « uni-noyau », un noyau à usage unique). Il s'agit de compiler l'application avec une bibliothèque qui inclut les fonctionnalités habituellement proposées par un SE (accès aux ressources ou exécution parallèle par exemple). Ainsi, le résultat de cette compilation est un unikernel : une image exécutable comme un SE mais qui ne fait fonctionner que l'application. Si l'on sait que cet unikernel sera exécuté de manière virtualisée, on peut l'optimiser grâce à la paravirtualisation afin d'améliorer les performances ; mais aussi réduire l'empreinte disque et mémoire ainsi que le temps de démarrage, afin de rendre plus efficace la mise à l'échelle horizontale (voir section 2.1.4).

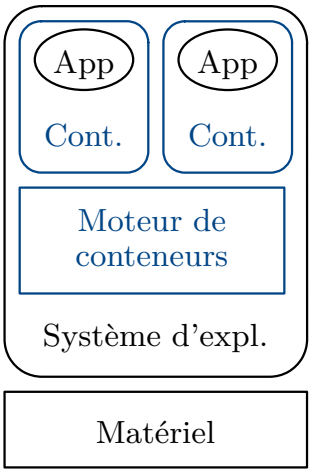


FIGURE 2.8. – Virtualisation du système d'exploitation.

Virtualisation du système d'exploitation

Contrairement à la virtualisation du matériel qui virtualise une machine entière, la virtualisation du système d'exploitation *virtualise un environnement d'exécution pour une application*, comme illustré en figure 2.8. Il s'agit de présenter à l'application un environnement d'exécution spécialement préparé et dédié à celle-ci, mais cloisonné du reste du SE hôte. Bien que d'autres termes existent en fonction de l'écosystème, on désigne ici cet environnement virtualisé comme un *conteneur*.

Un conteneur consiste en un processus du SE, mais qui est particulier en ce qu'il n'a pas accès à toutes ses ressources. Par exemple, il ne peut voir qu'une arborescence limitée du système de fichiers, ou ne peut pas utiliser toutes les interfaces réseau ; ou encore, sa capacité d'allocation mémoire ou son débit d'entrée-sortie disque sont limités.

Ces contraintes apposées sur le conteneur sont en fait des fonctionnalités du SE hôte, mais qui sont intégrées en une solution cohérente appelée *moteur de conteneurs*. Elles varient donc en fonction de la solution et du SE hôte. On distingue cependant deux catégories de ces contraintes :

isolation cacher au conteneur des ressources ou composants du système (utilisateurs, sections du système de fichiers, interfaces réseau…) ;

limitation contrôler l'accès aux ressources (limite d'allocation mémoire, de débit entrée-sortie réseau, d'utilisation du processeur…).

Moteur de conteneurs Souvent, le moteur de conteneurs ajoute des fonctionnalités pour faciliter le développement, le déploiement et la gestion des conteneurs. Par exemple, il peut aider à la manipulation de l'image du conteneur (comme l'image d'une VM) qui contient au minimum l'environnement d'exécution de l'application conteneurisée ; ou à régler le réseautage du conteneur pour qu'il ait accès à d'autres conteneurs locaux ou qu'il soit accessible depuis l'extérieur du centre d'hébergement cloud. On notera d'ailleurs que puisque la virtualisation est au niveau du SE, l'image d'un conteneur ne contient pas de noyau.

Des exemples de moteurs de conteneurs sont Docker [42], LXC [74], OpenVZ [99] ou encore systemd-nspawn [129].

Orchestrateur La section 2.1.3 explique que le PaaS prend souvent la forme d'un CaaS ou d'un FaaS, et est basé sur des conteneurs. Le déploiement d'une application complexe sur ces plate-formes ainsi que sa gestion restent difficiles, c'est pourquoi ces tâches sont souvent déléguées à un *orchestrateur*. Un orchestrateur offre généralement une abstraction au-dessus des conteneurs, afin de représenter l'application comme des blocs logiques. Il peut aussi abstraire d'autres fonctionnalités du moteur de conteneurs, comme la virtualisation du réseau.

Des exemples d'orchestrateurs sont Kubernetes [23], Docker Swarm [128] ou encore Ansible [52].

Imbrication des deux modes

On peut remarquer qu'il est possible de virtualiser un système d'exploitation qui s'exécute dans une machine virtuelle. Cette *virtualisation imbriquée*[4] permet de mitiger les inconvénients des deux modes tout en conservant certains avantages (voir section 2.2.3). Techniquement, elle fonctionne sans modification des modes de virtualisation, comme illustré en figure 2.9. Cependant, et de la même manière que la virtualisation à un seul niveau a révélé des problématiques spécifiques, cette virtualisation à deux niveaux apporte son lot de nouveaux défis à relever.

La virtualisation imbriquée (ou multi-virtualisation) peut être mise en place soit par l'utilisateur du cloud, qui loue des VM et déploie des conteneurs qu'il administre lui-même ; soit par le fournisseur de cloud, qui utilise les VM comme une première couche d'administration et de sécurité mais

4. Il existe aussi des moyens techniques pour imbriquer des VM dans des VM, mais cette utilisation reste rare car elle n'apporte que l'avantage de pouvoir administrer ses propres VM dans un cloud qui est déjà virtualisé et propose des VM.

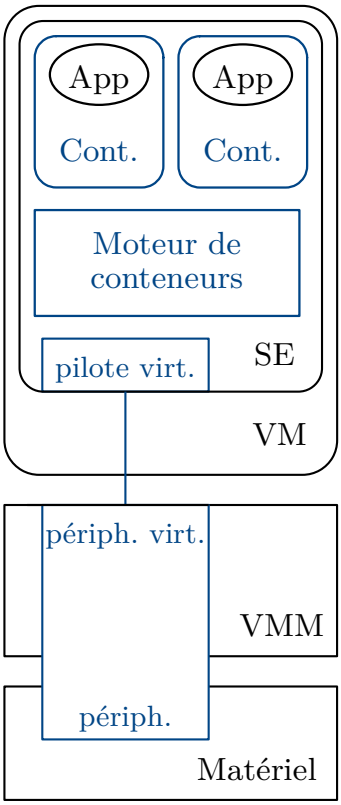


FIGURE 2.9. – Virtualisation imbriquée (virtualisation du SE dans une VM paravirtualisée assistée par matériel). Combinaison des modes de virtualisation illustrés en figures 2.7 et 2.8.

propose à ses clients un déploiement de conteneurs. En fait, on l'observe en pratique pour deux raisons majeures.

La sécurité Des études montrent que les conteneurs sont moins sécurisés que les VM [28, 49, 126]. Leur isolation des ressources est aussi moins efficace. Le tableau 2.1 ci-dessous résume la comparaison des deux modes de virtualisation, notamment sur ces aspects de sécurité.

Dans le contexte d'une offre publique de cloud qui héberge les applications de différents clients, *les VM sont utilisées pour isoler leurs applications entre elles*. Dans chaque VM, les conteneurs implémentent *l'isolation des composants des applications*. C'est une solution très répandue dans toutes les formes de cloud qui font intervenir les conteneurs. On peut citer l'offre FaaS (voir section 2.1.3) telle qu'implémentée dans Amazon Web Services Lambda : les fonctions s'exécutent dans des conteneurs qui contiennent l'environnement d'exécution, ceux-ci étant déployés dans des VM [135].

La nécessité technique L'utilisation des VM peut être une contrainte, par exemple si elles sont la seule forme d'achat de ressources.

Un exemple est le *cloud dérivé* : le principe est d'acheter à bas prix des VM chez un fournisseur de premier ordre, et de revendre ces ressources sous une forme mieux adaptée à un profil de client. Cette revente prend souvent la forme de conteneurs, et on retrouve donc de la virtualisation imbriquée.

Un autre cas de figure est celui des *clouds communautaires*. L'exemple 2 de la section 2.1.1 peut être considéré comme un cloud communautaire. Ce sont des clouds où le besoin de garder l'offre attrayante est moins présent, et ou l'inertie administrative freine aussi les mises à jour technologiques. Ainsi, bien que les utilisateurs souhaitent utiliser des conteneurs, ils sont contraints de louer des ressources sous la forme de VM, et ils déploient leurs conteneurs dans celles-ci. Un autre exemple provient d'Atos Integration, il est décrit en section 2.3.3.

2.2.3. Comparaison des modes de virtualisation

La virtualisation du matériel et la virtualisation du SE sont des techniques différentes mais qui ont un objectif similaire : le partage de la ressource informatique du cloud. Le tableau 2.1 les compare sur des points importants pour le développeur, l'utilisateur et le fournisseur de cloud. Il les compare aussi à la virtualisation imbriquée, afin de montrer comment elle permet de réduire les inconvénients des deux principaux modes.

TABLE 2.1. – Comparaison des principales techniques de virtualisation : du système d'exploitation « SE », du matériel « Matériel » (para-virtualisation assistée par le matériel), et imbriquée « Imbriquée » (conteneurisation dans une VM dont l'image est optimisée pour l'imbrication).

	SE	Matériel	Imbriquée
Sécurité	+	+ +	+ +
Performances	+	+ +	+ +
De la plate-forme	+	+ +	+ +
De l'invité	~	+	+
Utilisabilité	+ +	+	+ +
Développement	+	+ +	+
Déploiement	+ +	+	+ +
Gestion	+ +	+	+ +
Mise à l'échelle	+ +	+	+ +
Performance	−	−	−
Processeur	~	−	−
Mémoire	~	−	−
Mémoire de masse	~	−	−
Réseau	− −	−	− −
Prédictibilité	− −	−	− −
Démarrage	~	− −	−
Mémoire de masse	−	− −	−
Mémoire vive	~	− −	−

L'appréciation de chaque point de comparaison court de $--$, influence de la technique de virtualisation la plus néfaste ; à $++$, influence la plus positive de la technique. $\sim$ signifie que la technique n'a pas d'impact par rapport au déploiement de l'application en natif.

Il faut interpréter cette comparaison pour la *virtualisation d'une application donnée par rapport au déploiement natif dans un SE*. Par exemple, pour la comparaison de l'empreinte mémoire de masse, il est tout à fait possible pour une VM d'avoir une image plus petite qu'un conteneur, mais c'est peu probable pour des images de la même application [79].

De manière générale, on remarque que la virtualisation, quelle que soit la technique, apporte des gains de sécurité et d'utilisabilité. Elle dégrade les performances, bien que la virtualisation niveau SE s'approche d'un déploiement natif. On notera par ailleurs comment l'imbrication permet d'obtenir le niveau de sécurité de la virtualisation du matériel, avec l'utilisabilité et d'autres caractéristiques positives de la virtualisation du système d'exploitation ; bien que cela soit au détriment des performances de cette dernière.

Les points de comparaison sont détaillés ci-après. Au sujet des trois derniers points, on considère pour l'évaluation de la virtualisation imbriquée une situation moyenne : il faut parfois créer une nouvelle VM, et parfois une VM est déjà disponible, ce qui en mutualise les inconvénients pour plusieurs conteneurs. Autrement dit, ces points sont parfois limités par les caractéristiques de la virtualisation du matériel, et sont parfois limités seulement par la virtualisation du SE ; d'où une évaluation moyenne pour la virtualisation imbriquée.

Sécurité $--$ dénote la plus mauvaise sécurité

- **Performances** capacité des invités à ne pas voir leurs performances être influencées par l'exécution des autres,
- **De la plate-forme** capacité de la plate-forme à résister à des attaques ou intrusions en provenance de ses invités,
- **De l'invité** capacité de l'invité à résister à des attaques ou intrusions en provenance de sa plate-forme ;

Utilisabilité $--$ dénote la plus grande difficulté d'utilisation de la technique de virtualisation

- **Développement** facilité de création d'une image qui peut être déployée pour exécuter des invités,

	Déploiement	facilité d'exécution des invités à partir de leurs images,
	Gestion	facilité d'administration des invités par la plate-forme,
	Mise à l'échelle	facilité de mise à l'échelle horizontale et verticale des invités ;
Performance		– – dénote le pire impact sur les performances des invités, naturellement la virtualisation ne peut pas apporter de bénéfice sur ce point ;
Démarrage		– – dénote la création des invités la plus lente ;
Mémoire de masse		– – dénote la plus grande taille d'image des invités ;
Mémoire vive		– – dénote la plus grande utilisation de mémoire vive.

2.2.4. Applications de la virtualisation

Quel que soit le mode de virtualisation, elle est la technologie au cœur du cloud pour implémenter les techniques présentées en section 2.1.4.

Une entité virtuelle (VM ou conteneur) correspond à une certaine quantité de ressources louées auprès du cloud. Cette quantité de ressources virtuelles peut être *modifiée dynamiquement* pour subvenir aux besoins changeants de l'application par rapport à sa charge de travail, par exemple en lui ajoutant des processeurs virtuels ou en diminuant son allocation mémoire. [5] C'est ainsi que la mise à l'échelle verticale est implémentée.

De plus, une entité virtuelle est créée à partir d'une image, il est donc aisé de la *répliquer* pour assurer la mise à l'échelle horizontale. Cette dernière va de pair avec l'équilibrage de la charge. Cette fonctionnalité est possible parce que tous les composants du cloud, y compris les entités virtuelles répliquées mais aussi des serveurs de bases de données et autres supports, sont reliés en *réseaux virtuels*, gérés par la plate-forme de cloud. La virtualisation touche en effet le réseautage du centre d'hébergement.

Par ailleurs, du point de vue du fournisseur de cloud, la *charge de travail est découpée en entités virtuelles* de différentes tailles suivant différentes dimensions (les types de ressources allouées). À leur création, ces entités virtuelles peuvent être placées sur n'importe quel serveur physique du centre d'hébergement, qui répond bien sûr à des contraintes de ressources suffisantes et à d'autres contraintes plus subtiles. De plus, ces entités

5. Ce ne sont pas toujours des opérations simples, notamment pour les machines virtuelles, mais cela reste néanmoins possible.

virtuelles peuvent être *migrées*, c'est-à-dire qu'il est possible de déplacer une VM ou un conteneur d'un serveur physique à l'autre (ou d'une VM à l'autre dans le cas de la virtualisation imbriquée), et ce sans impact visible sur l'exécution de l'application virtualisée. [6] C'est ainsi que la virtualisation permet l'ordonnancement de la charge de travail dans le cloud.

Enfin, la section 2.2.3 montre des caractéristiques de sécurité liées au mode de virtualisation. En effet, grâce à la virtualisation, le cloud apporte des garanties de sécurité via un *cloisonnement* des applications des différents utilisateurs. Il est aussi possible grâce à la virtualisation de produire des *instantanés* de l'état d'une entité virtuelle en cours d'exécution. Cela est utilisé pour la migration, mais c'est aussi une fonctionnalité utile pour la sûreté de fonctionnement de l'application virtualisée.

2.3. Problématiques

La section 2.2 a présenté le rôle fondamental de la virtualisation pour le cloud. La présente section présente les grandes problématiques qu'elle apporte, d'abord au niveau de la virtualisation du matériel, puis au niveau de celle du système d'exploitation. Enfin, elle les met en perspective dans le contexte de cette thèse : la multi-virtualisation.

2.3.1. Gestion des ressources virtualisées

La section 2.2.4 explique que la virtualisation permet l'administration des ressources pour les partager entre les utilisateurs. C'est un enjeu crucial pour le fournisseur de cloud. En effet, résoudre au mieux le difficile problème d'assurer les requêtes en ressources variables des clients, avec les serveurs aux ressources fixes du centre d'hébergement, permet *d'économiser des ressources*. Cette économie se traduit de deux manières :

1. aspect financier : le fournisseur de cloud peut servir plus de clients (plus de revenus), et a besoin d'un nombre minimal de serveurs physiques (moins de frais opérationnels) ;
2. aspect énergétique : il faut moins de serveurs actifs pour assurer le service, ce qui se répercute sur la consommation électrique (directement, sur la consommation des serveurs ; et indirectement, sur

6. La migration de VM est différente de la migration de conteneurs ; en pratique, migrer un conteneur consiste souvent à l'arrêter et à le recréer à la destination sans réellement migrer son état.

la consommation des équipements tiers comme la climatisation des centres d'hébergement).

L'utilisation de l'ordonnancement pour répondre à un objectif d'économie des ressources s'appelle la *consolidation* de la charge de travail. Grossièrement, on cherche à répartir la charge sur les serveurs physiques pour optimiser l'utilisation des ressources et en utiliser un minimum.

2.3.2. Performance et prédictibilité des applications virtualisées

Idéalement, la virtualisation est une abstraction des ressources pour en faciliter l'administration et l'accès, de manière invisible du point de vue de l'utilisateur. En pratique, elle apporte un coût en termes de *performance* et de *prédictibilité* de celle-ci (voir tableau 2.1).

En effet, les performances sont diminuées par la virtualisation du matériel. Cette dégradation peut avoir deux origines :

1. l'accès au processeur, à la mémoire, aux entrées-sorties, etc. est indirect, car il est administré par la solution de virtualisation (hyperviseur ou moteur de conteneurs), et cette ingérence se traduit nécessairement par l'exécution d'instructions qui ne profitent pas à l'application virtualisée, mais qui sont nécessaires pour l'administration virtuelle ;
2. la virtualisation, en ce qu'elle virtualise les ressources, a un effet inattendu sur l'application virtualisée.

Une illustration de la première origine est la virtualisation du réseau : il faut partager une interface réseau physique entre plusieurs entités virtuelles concurrentes, ce qui est souvent implémenté par l'ajout d'une interface réseau virtuelle en amont qui est capable de multiplexer l'interface physique.

En ce qui concerne la seconde origine, un exemple est l'effet de la virtualisation du processeur : dans le cas d'une VM, le SE invité et l'hyperviseur ont leurs propres ordonnanceurs de tâches. Le premier gère des processus, le second gère des processeurs virtuels sur lesquels s'exécutent les processus du SE invité. Il peut arriver que l'ordonnanceur du SE invité élise un processus à l'exécution sur un processeur virtuel, mais que l'ordonnanceur de l'hyperviseur n'élise pas ce processeur virtuel à l'exécution. Lorsque le processus en question est prioritaire sur l'accès à une ressource de la VM, cela peut être catastrophique pour les performances de celle-ci.

On notera que la dégradation des performances varie grandement en force et en nature selon la ressource. Cela peut être une simple diminution du débit

d'entrée-sortie ; une augmentation significative du temps d'exécution de l'application ; ou bien même des blocages longs et répétés de son exécution. Par ailleurs, le second exemple ci-dessus montre un effet particulièrement imprévisible de dégradation des performances. Celle-ci se manifeste souvent, mais semble être aléatoire puisqu'elle provient d'une conjoncture de deux acteurs qui n'ont pas de rapport direct entre eux.

De manière générale, les performances d'une application dans le cloud sont *imprévisibles* car l'utilisateur ne contrôle pas le support physique d'exécution. Par exemple, il peut spécifier la quantité de processeur virtuel allouée, mais la performance de calcul dépend finalement des caractéristiques des processeurs physiques du serveur choisi par l'administrateur du cloud pour exécuter l'application. Les centres d'hébergement cloud incluent d'ailleurs souvent des serveurs hétérogènes, ce qui rend pratiquement impossible l'estimation précise des performances.

2.3.3. Évolution des problématiques dans le cloud doublement virtualisé

Les sections 2.3.1 et 2.3.2 ci-dessus décrivent les problématiques du cloud virtualisé, mais comment ces problématiques évoluent-elles dans un cloud doublement virtualisé ? En effet, la section 2.2.2 explique que les deux modes de virtualisation (niveau matériel et niveau système d'exploitation) peuvent être imbriqués. Plusieurs raisons peuvent motiver ce choix, mais on décrit ici le cas d'usage d'Atos Intégration qui a motivé cette thèse.

Exemple 3. *La European Space Agency (ESA) souhaite aider à la valorisation des données photographiques terrestres produites par ses programmes d'étude. Atos Intégration, avec d'autres partenaires, s'est vue confier la mission de développer une plate-forme cloud pour accomplir cet objectif. Un utilisateur de cette plate-forme doit pouvoir accéder à l'ensemble massif des données en y déployant ses applications de traitement et d'analyse.*

Il s'agit donc d'un cloud Platform-as-a-Service (PaaS), avec une composante serverless puisque le stockage et la mise à disposition des données sont à la charge du fournisseur. Afin de faciliter le développement, le déploiement et le partage des applications utilisateurs, le choix des conteneurs, c'est-à-dire de la virtualisation niveau système d'exploitation, a été retenu.

Cependant, Atos Intégration n'est pas le fournisseur du cloud. Celui-ci sera servi par un centre d'hébergement géré au niveau européen et dédié à ce genre de projets. C'est un cloud communautaire, dont les ressources sont

vendues sous la forme de machines virtuelles. Ainsi, la virtualisation des VM est une contrainte, et la virtualisation imbriquée est nécessaire.

Ainsi, dans le contexte de la virtualisation imbriquée, les préoccupations sont réparties sur les deux modes de virtualisation, mais aussi sur leur combinaison. En particulier, voici les problématiques traitées par ce document :

1. niveau virtualisation du matériel : gestion des ressources virtualisées en exploitant la mise en veille de serveurs qui hébergent des machines virtuelles inactives ;
2. niveau virtualisation du système d'exploitation : prédictibilité des performances en corrigeant l'allocation de processeur aux conteneurs ;
3. combinaison des niveaux : performance du réseau et gestion des ressources multi-virtualisées, en utilisant le niveau des VM pour virtualiser le réseau des conteneurs.

2.3.4. Contributions

La section 2.3.1 suggère que l'objectif classique de la consolidation est l'économie des ressources physiques. Le chapitre 3 propose une nouvelle approche qui cherche à consolider la charge de travail (représentée par des VM, c'est-à-dire au premier niveau de la virtualisation imbriquée), de sorte à optimiser la durée de mise en veille des serveurs qui les hébergent. L'objectif est donc l'économie de consommation énergétique. Ce travail a été publié sous la référence suivante :

> Mathieu Bacou, Grégoire Todeschi, Alain Tchana, Daniel Hagimont, Baptiste Lepers et Willy Zwaenepoel. « Drowsy-DC : Data Center Power Management System ». In : *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019*. IEEE, mai 2019, p. 825-834

Le chapitre 4 concerne quant à lui une atteinte à la prédictibilité des performances des applications conteneurisées (donc au deuxième niveau de la virtualisation imbriquée), qui subissent des difficultés à s'auto-configurer dans cet environnement. Ce travail a été publié sous la référence suivante :

> Mathieu Bacou, Alain Tchana et Daniel Hagimont. « Your Containers Should be WYSIWYG ». In : *2019 IEEE International Conference on Services Computing, SCC 2019*. IEEE, juil. 2019, p. 56-64

Enfin, le chapitre 5 propose une solution au problème de la virtualisation du réseau des applications doublement virtualisées, en prenant en compte les deux niveaux. La contribution cherche à simplifier la virtualisation du réseau, et à abstraire la couche de virtualisation des machines virtuelles dans ce contexte de virtualisation imbriquée ; cette abstraction apporte aussi une amélioration de la gestion des ressources et une diminution du coût d'utilisation du cloud. Ce travail a été publié sous la référence suivante :

> Mathieu Bacou, Grégoire Todeschi, Alain Tchana et Daniel Hagimont. « Nested Virtualization Without the Nest ». In : *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019*. ACM, août 2019, 12:1-12:10

Chapitre 3

La consolidation de la charge de travail surtout inactive : Drowsy-DC

3.1. Introduction 33
3.2. Notions préliminaires 38
3.3. Vue d'ensemble 43
3.4. Consolidation en fonction de l'inactivité 44
3.5. Mise à jour du modèle d'inactivité 51
3.6. Suspension des serveurs 54
3.7. Reprise des serveurs 56
3.8. Optimisation de la reprise des serveurs 58
3.9. Évaluation 61
3.10. État de l'art 74
3.11. Conclusion 76

3.1. Introduction

Maintenant qu'il est établi que la mutualisation du cloud apporte des avantages considérables en termes de gestion de la ressource informatique, il est hautement intéressant de chercher à l'optimiser. Avec la mutualisation, les ressources disponibles ne sont pas en corrélation avec les besoins. En effet, les besoins totaux sont plus grands que les ressources physiques ; le fournisseur de cloud s'assure seulement que les besoins en un instant donné sont couverts par la provision actuelle des ressources. Pour ce faire, leur *économie* est primordiale, et on cible ici l'aspect énergétique.

3.1.1. Efficacité énergétique

La réduction de la consommation énergétique des centres d'hébergement (en anglais *data centers*, ou DC), et de manière plus générale leur *efficacité énergétique*, est une préoccupation majeure des fournisseurs de cloud. Les travaux traitant de ce sujet peuvent être répartis en deux catégories selon le niveau auquel ils agissent :

1. niveau serveur, en minimisant la consommation électrique du matériel d'un seul serveur ;
2. niveau DC, en minimisant la consommation électrique nécessaire au DC pour servir le cloud.

Dans la première catégorie, le but ultime est d'atteindre la *proportionnalité énergétique* [15]. Un serveur présentant cette caractéristique consomme de l'énergie proportionnellement à son utilisation. Ainsi, un serveur sans charge de travail ne consommerait pas d'énergie. Ce concept est illustré en figure 3.1 par la droite noire. Atteindre l'objectif de la proportionnalité énergétique requiert le contrôle indépendant des états énergétiques de chaque composant matériel. Ces états énergétiques sont décrits plus amplement en section 3.2.2.

De plus, un tel système modulaire est capable d'adapter la consommation énergétique de chaque composant à l'utilisation courante de celui-ci. Bien que les solutions existantes améliorent visiblement l'efficacité énergétique du processeur, des disques durs et des interfaces réseau, l'état de l'art de ce domaine est encore loin de son idéal des serveurs à proportionnalité énergétique, comme on peut le voir en figure 3.1. Par conséquent, il existe des techniques de gestion de l'énergie au niveau du DC.

Les travaux de la seconde catégorie exploitent la *consolidation* des VM. Elle consiste à ordonnancer dynamiquement les VM sur un nombre minimal de serveurs, dans le but de placer ces serveurs désormais inactifs dans un état de faible énergie. Cet état peut être :

— la suspension, c'est-à-dire que le serveur suspend son exécution et conserve son état en mémoire vive ;
— le sommeil, aussi appelé « hibernation », c'est-à-dire que le serveur suspend son exécution mais conserve son état en mémoire persistante, comme sur un disque dur.

Placer les serveurs inactifs dans des états de faible énergie permet non seulement de réduire la consommation électrique du DC ; mais en augmentant la charge de travail des serveurs actifs, elle améliore aussi leur efficacité énergétique, ce qui est aussi illustré sur la figure figure 3.1.

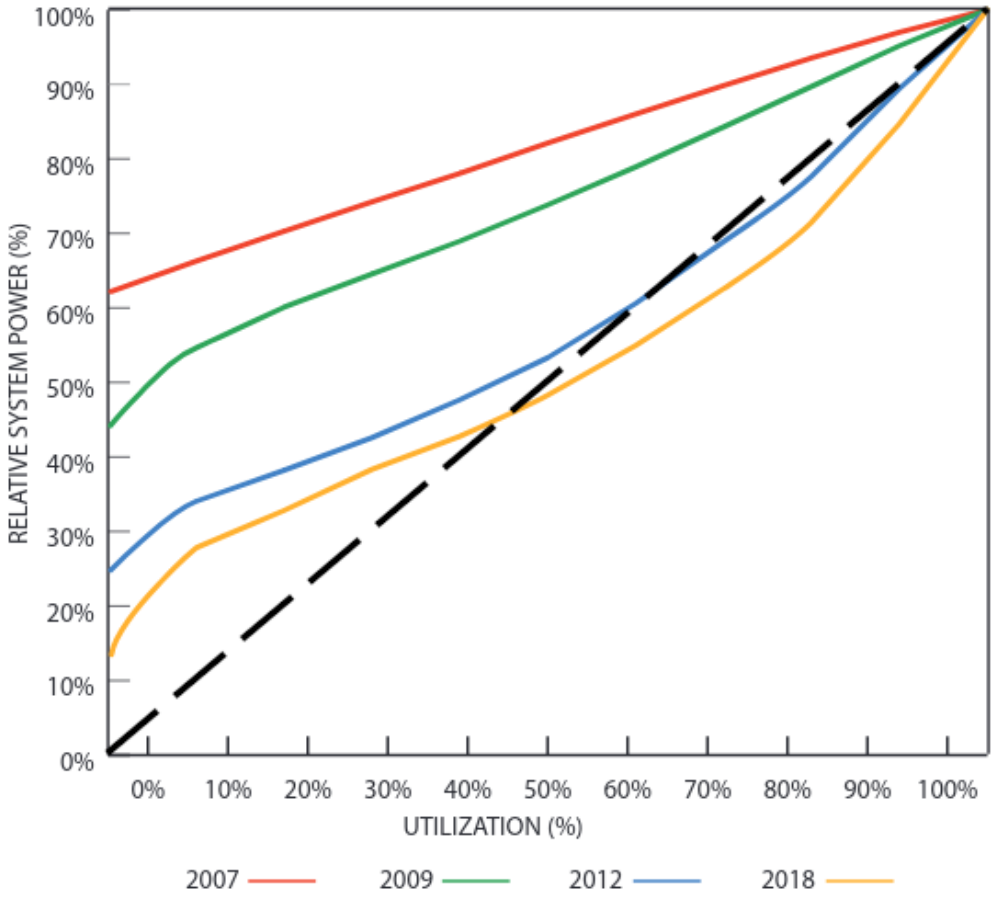


FIGURE 3.1. – Proportionnalité énergétique : consommation normalisée des générations de systèmes, en fonction de leur utilisation normalisée. Données pour des serveurs Intel à deux *sockets*. Tiré de BARROSO, HÖLZLE et RANGANATHAN [16]

3.1.2. Limites de la consolidation

Cependant, le gain d'utilisation du DC est d'environ 5 à 10%, et en pratique, on constate rarement des serveurs avec un taux d'utilisation supérieur à 50% même dans le cas des charges de travail les mieux adaptées [15, 31, 35, 82]. Les explications sont multiples.

D'abord, la section 2.1.4 suggère que l'ordonnancement est un problème difficile. En fait, la consolidation dans son rôle d'ordonnancement des VM est un problème de *bin packing* (que l'on peut traduire comme un problème de rangement d'objets dans un nombre minimum de boîtes). Ce problème d'optimisation combinatoire est dit NP-difficile, c'est-à-dire qu'il n'est pas soluble avec un algorithme de complexité polynomiale (sous-entendu, il faut un algorithme de complexité exponentielle). La conséquence de cette caractérisation est que trouver, en un temps raisonnable, une solution parfaite à ce problème, est impossible en pratique. Autrement dit, calculer un placement optimal des VM sur les serveurs du DC, qui permettrait d'atteindre le niveau d'utilisation maximal des serveurs, est impossible.

Ensuite, la méthode de partage des ressources du DC dépend de leur nature. Par exemple, contrairement au processeur qui est partagé temporellement, la mémoire est partagée spatialement entre les VM et ne peut pas être préemptée facilement. [1] Par conséquent, *la mémoire est souvent la ressource limitante* lors du processus de consolidation [96].

3.1.3. Problème de la mémoire virtualisée

Ce constat mène à de nombreuses solutions qui cherchent à minimiser l'empreinte mémoire des VM (on peut citer le partage des pages mémoire [14, 88, 134] et la compression des pages [41, 145]). Une autre piste est la surallocation de la mémoire (par exemple avec le *ballooning* [26, 117, 134]) : puisque les VM utilisent de la mémoire logique, l'hyperviseur peut leur attribuer plus de mémoire que le serveur n'en possède physiquement. Cependant, cette solution vient avec tous les problèmes que l'on peut imaginer en cas de pénurie de mémoire. En fait, toutes ces solutions ne sont généralement pas déployées dans les DC cloud grand public, pour deux raisons.

Premièrement, malgré les gains substantiels de mémoire obtenus par ces méthodes, l'impact sur les performances des applications est important. Les pénalités encourues lorsque les pires cas de ces solutions surgissent

1. De manière générale pour le processeur, la préemption désigne le fait d'arrêter de donner du temps processeur à un processus qui l'utilisait actuellement, pour le donner à un autre processus.

sont trop grandes. Secondement, et en particulier pour le *ballooning*, ces solutions dépendent de l'estimation de l'empreinte mémoire des applications et des VM. Cette estimation est très difficile à faire à cause de la variabilité temporelle de l'espace mémoire de travail (*working set*), c'est-à-dire de la mémoire strictement nécessaire au bon fonctionnement de la VM [60, 75].

3.1.4. Nouvelle vision de la consolidation

Le constat est donc le suivant : d'une part, la route vers la proportionnalité énergétique est longue ; et d'autre part, l'évolution de la consolidation est bloquée par la gestion de la mémoire virtualisée. On propose dans ce travail de combiner les deux catégories de solutions, niveau serveur et niveau DC, en redéfinissant l'utilisaton de la consolidation afin d'améliorer l'efficacité énergétique des serveurs.

L'observation de départ est que les traces d'activité de VM montrent des motifs. En fonction de ces motifs, il est possible de classer les VM en trois types [143] (décrits plus en détails en section 3.2) :

1. courte durée de vie, et donc toujours actives ;
2. longue durée de vie, surtout actives ;
3. longue durée de vie, surtout inactives.

C'est le troisième type qui est le centre d'intérêt ici. On le note LLMI, pour la terminologie anglaise *long-lived, mostly idle*. Les motifs d'activité des VM LLMI présentent des pics d'utilisation isolés, séparés par de longues périodes d'inactivité. [2] Ces VM sont devenues une réelle préoccupation pour les fournisseurs de clouds publics, puisque Amazon Web Service et Microsoft Azure [124] par exemple, proposent désormais des tarifications adaptées. C'est aussi le cas pour les clouds privés, qui hébergent en fait une majorité de VM LLMI : en effet, plus de la moitié de la charge de travail de Nutanix, un fournisseur de cloud privé [97], est constituée d'applications d'entreprises qui sont des cas typiquement LLMI [98].

Le constat de la présence des VM LLMI mène à l'idée suivante : consolider dans le but d'obtenir des serveurs *qui peuvent être suspendus alors qu'ils hébergent des VM*, ce qui permet d'obtenir une efficacité énergétique presque optimale. Concrètement, on cherche à consolider pour *colocaliser les VM qui présentent des motifs d'inactivité similaires*, tout en prenant bien sûr en compte les contraintes de consolidation classiques telles que les besoins

2. Elles sont parfois appelées *burstable*, en référence à leur activité soudaine et éphémère.

en ressources. En faisant cela, on produit des serveurs qui peuvent être suspendus pendant ces périodes d'inactivité synchrones. De tels serveurs sont qualifiés de *somnolents*, c'est-à-dire en anglais *drowsy*, et le système a donc été baptisé Drowsy-DC. Le concept de Drowsy-DC est illustré en figure 3.2.

Son implémentation nécessite trois composants :

1. une extension de la consolidation afin d'implémenter cette nouvelle vision, ce qui nécessite notamment de modéliser l'inactivité des VM ;
2. un sous-système pour la suspension des serveurs, pour maximiser la durée de suspension sans dégrader le service ainsi que pour éviter d'alterner rapidement entre état suspendu et état actif ;
3. un sous-système pour la reprise des serveurs, qui minimise la dégradation de la latence de service.

Organisation du chapitre

La section 3.2 décrit les concepts et notions spécifiques à ce chapitre. Une vision d'ensemble du travail de ce chapitre est donnée en section 3.3. On présente ensuite l'algorithme de consolidation basée sur l'inactivité des VM dans la section 3.4, et le détail de la modélisation de l'inactivité des VM, au cœur de cet algorithme, est rapporté en section 3.5. Les sous-systèmes logiciels pour la suspension et la reprise des serveurs sont décrit en sections 3.6 et 3.7 respectivement. La section 3.8 montre comment la reprise des serveurs a été optimisée. Pour continuer, une évaluation du système Drowsy-DC est conduite en section 3.9. Enfin, la section 3.10 suit en présentant l'état de l'art du sujet, avant de conclure sur ce travail dans la section 3.11.

3.2. Notions préliminaires

Drowsy-DC concerne principalement les machines virtuelles et la virtualisation matérielle, présentées en section 2.2.2. Il agit aussi au niveau des serveurs et du centre d'hébergement à travers l'ordonnancement, une technologie de cloud abordée en section 2.1.4. Celui-ci repose sur la consolidation, c'est-à-dire la migration de VM, décrite en section 2.2.4. Les concepts spécifiques à Drowsy-DC sont d'abord les différents types de VM évoqués dans l'introduction (LLMI et consorts) ; et ensuite les états énergétiques des serveurs, qui sont à la base de la gestion de la consommation électrique.

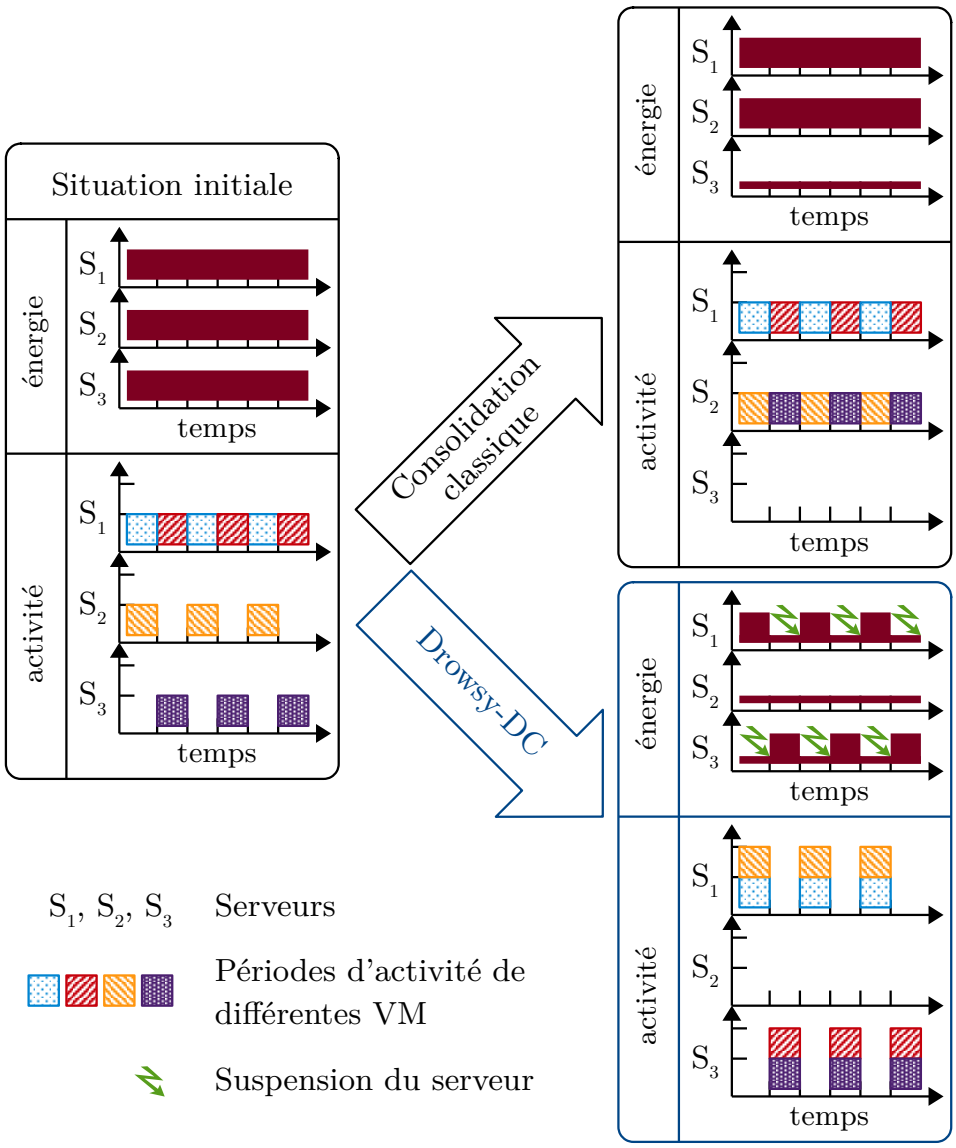


FIGURE 3.2. – Concept de Drowsy-DC. La consolidation classique cherche à vider des serveurs ; tandis que Drowsy-DC cherche à synchroniser les périodes d'inactivité des VM pour suspendre des serveurs en plus d'en vider d'autres.

3.2.1. Caractérisation des VM : SLMU, LLMU, LLMI

L'introduction explique que l'observation des traces d'activité des VM d'un DC dévoile qu'une VM peut être d'un type parmi trois :

à courte durée de vie, surtout active (SLMU) tâches de traitement des données, comme du MapReduce [32]…

à longue durée de vie, surtout active (LLMU) services Web à forte affluence…

à longue durée de vie, surtout inactive (LLMI) services Web rarement sollicités, avec des taux de fréquentation saisonniers…

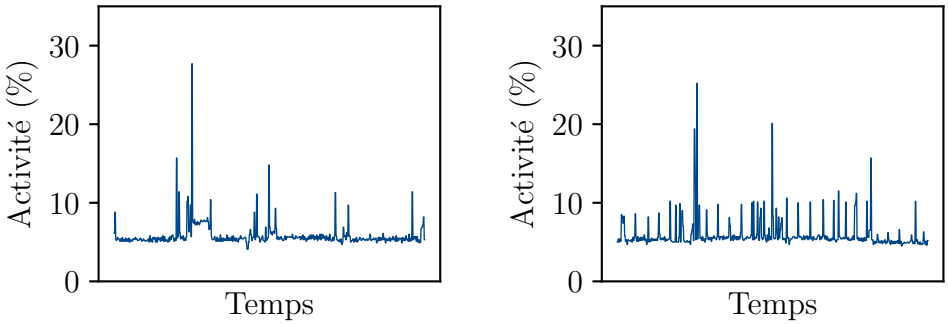
Le principe de Drowsy-DC est d'exploiter les phases inactives des VM LLMI (de l'anglais *long-lived, mostly idle*). Ce sont donc les VM d'intérêt pour ce système. Il doit cependant composer avec les autres types dans le DC : les SLMU (de l'anglais *short-lived, mostly used*) et les LLMU (de l'anglais *long-lived, mostly used*). Heureusement, ils sont facilement identifiables de par leur activité permanente aux motifs très différents de ceux d'une VM LLMI (voir section 3.9.1).

La figure 3.3 montre des traces d'activité de VM LLMI. Les noms des VM V_x correspondent aux VM utilisées pour l'évaluation, en section 3.9.1.

3.2.2. États énergétiques

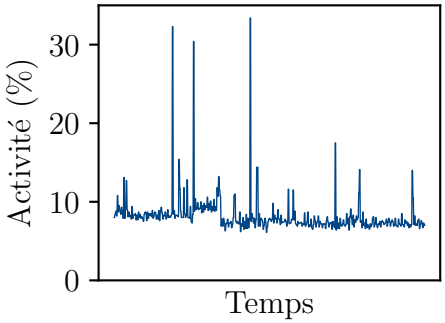
Les économies d'énergie réalisées par Drowsy-DC proviennent de la suspension des serveurs. C'est-à-dire que le système cherche à produire des serveurs inactifs qui peuvent alors être mis dans un état de faible énergie. Cet état consomme moins d'énergie parce que la plupart des fonctions et périphériques du serveur sont désactivées. Par exemple, le processeur et le contrôleur disque sont éteints.

Les états énergétiques d'un ordinateur en général, sont gérés par l'interface ACPI, pour *Advanced Configuration and Power Interface* (c'est-à-dire « Interface avancée de configuration et d'alimentation »). C'est une interface standard utilisée par les SE pour gérer l'état énergétique des composants matériels qui l'implémentent. ACPI permet le contrôle de nombreux aspects d'une machine par le SE, y compris donc les états énergétiques. Il y a plusieurs catégories d'états en fonction de la cible (état global, état du processeur, état d'un périphérique…). Les états énergétiques dont on parle ici sont les états de sommeil S_{x}. Le tableau 3.1 liste les états de sommeil tels que définis par ACPI.

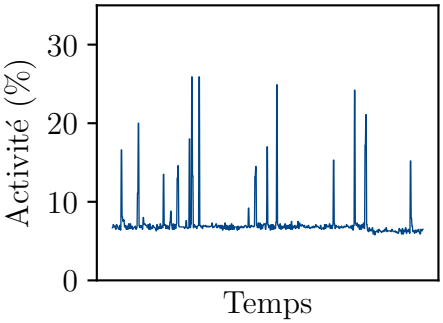


(a) V_1

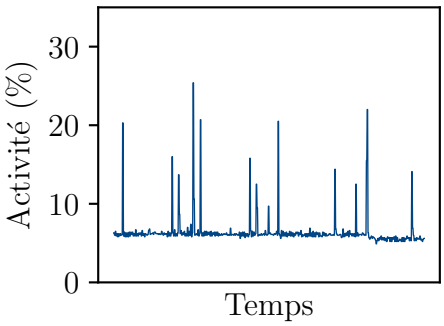
(b) V_2



(c) V_3 et V_4



(d) V_5



(e) V_6

FIGURE 3.3. – Traces d'activité de VM LLMI dans un DC de cloud privé.

TABLE 3.1. – États de sommeil ACPI.

S_x	Nom courant	Description
S_0	actif	le serveur est en exécution normale, il sert des requêtes
S_1	suspension active	le processeur et la mémoire centrale restent alimentés mais ne font rien, et les autres périphériques peuvent être suspendus
S_2	*néant*	comme S_1 mais le processeur est éteint ; rarement utilisé
S_3	suspension (*suspend to RAM*)	seule la mémoire centrale reste alimentée pour maintenir l'état, tous les autres périphériques sont suspendus
S_4	hibernation (*suspend to disk*)	la mémoire centrale est copiée vers de la mémoire non volatile (comme un disque dur), et tout le système est éteint ; un serveur peut être relancé depuis cet état
S_5	extinction	comme S_4 mais sans la sauvegarde de la mémoire centrale ; un serveur peut être redémarré à distance même dans cet état

La consolidation classique cherche à produire des serveurs vides de VM afin de les placer dans l'état d'extinction S_4. Et de fait, l'état de suspension S_3 était rarement implémenté dans les serveurs. Drowsy-DC quant à lui, veut exploiter l'état S_3, que l'on désigne simplement comme « la suspension », par opposition à l'hibernation S_4. La consommation d'énergie de cet état est supérieure à celles de S_4 et S_5 ; mais cette augmentation est négligeable.

Néanmoins, la suspension a l'avantage de permettre une reprise très rapide du serveur. En effet, si la relance d'un serveur depuis S_3 ne nécessite grossièrement que de relancer les périphériques matériels (voir section 3.8) ; relancer le serveur depuis l'état S_4 requiert la copie de l'état du serveur avant hibernation depuis le disque dur vers la mémoire centrale. Cette copie devient de plus en plus lente à mesure que la quantité de mémoire centrale des serveurs augmente. Relancer le serveur depuis l'état S_5 revient simplement à redémarrer totalement la machine : le SE reprend de zéro.

En résumé, la consolidation en général cherche à placer des serveurs dans l'état d'extinction S_4 pour ne pas consommer d'énergie ; et Drowsy-DC va plus loin et cherche en plus, à placer des serveurs dans l'état de suspension S_3, qui représente un équilibre très intéressant entre la consommation d'énergie et la performance.

3.3. Vue d'ensemble

Drowsy-DC gère un centre d'hébergement grâce à deux modules logiciels :

Module de suspension extension du système de surveillance des serveurs qui existe déjà dans le DC, dont le rôle est de prendre la décision, en fonction de différents critères, de suspendre un serveur ;

Module de reprise extension du gestionnaire du DC qui existe déjà en son sein, dont le rôle est de relancer les serveurs suspendus lorsqu'une des VM qu'ils hébergent ont besoin de s'exécuter.

Par ailleurs, le module de reprise comporte des optimisations pour garantir une durée de reprise minimale. Ces optimisations dépendent des informations fournies par le module de suspension. Les deux modules sont décrits respectivement en section 3.6 et section 3.7.

Avec l'ajout de ces modules, le système de consolidation existant est amélioré par l'incorporation de l'algorithme de consolidation de Drowsy-DC basé sur l'inactivité des VM. Le résultat est un système, illustré en figure 3.4,

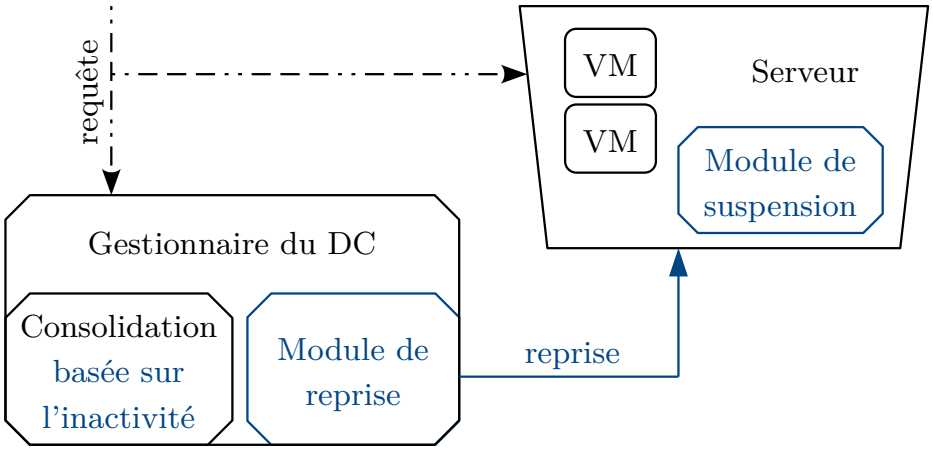


FIGURE 3.4. – Architecture de Drowsy-DC. Les éléments apportés pour l'intégration de Drowsy-DC sont en bleu.

qui peut prendre des décisions de placement de VM en fonction des critères classique, tels que les requêtes en ressources, mais aussi en fonction d'un nouveau critère : les motifs d'inactivité des VM. Cet algorithme est décrit dans la section 3.4 qui suit.

3.4. Consolidation en fonction de l'inactivité

Le concept au centre du placement des VM par Drowsy-DC est la notion de *périodes d'inactivité*. Le but est de colocaliser les VM qui seront vraisemblablement inactives durant le prochain intervalle de temps (par exemple la prochaine heure), pour faire en sorte que leur serveur reste dans son état de faible énergie (suspendu) pendant cet intervalle de temps.

3.4.1. Principe

Drowsy-DC construit continuellement un *modèle d'inactivité* (noté MI) pour chaque VM. Ce modèle peut être vu comme un résumé de l'inactivité observée dans le passé. Ainsi, chaque fois qu'une VM est examinée pour être placée ou déplacée, Drowsy-DC calcule à partir de son MI une *probabilité d'inactivité* (notée PI). Cette PI quantifie la possibilité que cette VM soit inactive durant le prochain intervalle de temps. Ce n'est pas à strictement parler une probabilité, mais elle est une prédiction quantitative qu'une VM soit inactive pendant une heure donnée, en fonction de son passé observé.

On définit aussi la probabilité d'inactivité d'un serveur, par la moyenne des PI de ses VM. Le choix de la moyenne pour résumer les PI des VM d'un serveur vient de l'ajout d'une étape de consolidation (voir section 3.4.4) qui optimise le placement pour limiter la taille de l'intervalle des PI sur un serveur. Cette étape signifie qu'il est préférable d'utiliser la moyenne des PI de ses VM pour représenter le comportement global d'un serveur.

Finalement, l'algorithme de consolidation va sélectionner le serveur de destination pour une VM en fonction d'une part des contraintes de placement traditionnelles (par exemple la disponibilité en ressources) pour satisfaire l'accord de qualité de service (*Service-Level Agreement*, ou SLA) ; et d'autre part en fonction d'une contrainte de proximité entre la PI de la VM et la PI du serveur, tout en cherchant à maximiser cette dernière.

On décrit maintenant comment le MI est construit, et la PI calculée à partir de celui-ci. Ensuite, on détaille l'intégration de cet algorithme dans un système de gestion de DC existant basé sur OpenStack.

3.4.2. Construction du modèle d'inactivité (MI)

L'objectif du MI d'une VM est de fournir des données afin de calculer sa PI pour les intervalles de temps futurs. Une solution directe est de considérer que la PI du prochain intervalle de temps ne dépend que de l'intervalle de temps courant. Cependant, cette méthode produit un taux de faux positifs (prédiction d'inactivité alors que la VM est active) très haut.

Le MI propose une approche plus raffinée inspirée par l'observation des motifs d'inactivité des VM s'exécutant dans le DC de Nutanix. On peut y identifier les trois types de VM présentés en section 3.2.1, et en particulier une proportion significative de VM LLMI. Ce sont en effet les VM de cette catégorie qui sont au centre de Drowsy-DC. En étudiant leurs traces, on remarque de l'inactivité périodique à quatre échelles :

1. l'heure du jour (par exemple, le matin) ;
2. le jour de la semaine (par exemple, le week-end) ;
3. le jour du mois (par exemple, la fin du mois) ;
4. le mois de l'année (par exemple, l'été).

Cette observation est confirmée par d'autres travaux qui analysent les traces d'activité de VM [3, 17, 31].

Cette étude guide le choix de l'heure comme intervalle de temps. Cet intervalle peut être vu comme la résolution du MI, et donc comme la résolution des décisions de placement en fonction de la PI. Néanmoins, ce

sont bien les quatre échelles de temps listées ci-dessus qui sont prises en compte pour définir l'inactivité d'une VM.

En fait, avec le MI, on cherche à exprimer le type d'information suivant :

> La probabilité que la VM soit inactive à l'heure h, lors du jour j_s de la semaine, qui est aussi le jour j_m du mois m, est X.

Par exemple, le site Web qui affiche les résultats d'un diplôme national sera particulièrement visité à des heures spécifiques (14h et 15h) pendant un jour précis (le 20ème) d'un mois précis (juillet), et ce tous les ans.

Concrètement, chaque serveur du DC exécute un constructeur de modèle, qui collecte chaque heure le niveau d'activité de chacune de ses VM. Cette collecte sert à mettre à jour les scores de *synthèse d'inactivité* (notée SI) qui constituent le modèle d'une VM (voir section 3.5 pour le processus de mise à jour du MI). Le niveau d'activité collecté est basé sur le nombre de quanta de temps processeur que l'ordonnanceur de l'hyperviseur a alloué à la VM durant l'heure passée.

On définit quatre types de scores SI, un pour chaque échelle temporelle :

- $\mathrm{SI}_j(h)$: score de l'inactivité en fonction de la position de l'heure h dans le jour ;
- $\mathrm{SI}_s(h, j_s)$: score de l'inactivité en fonction de la position de l'heure h dans la semaine (combinaison de h et j_s) ;
- $\mathrm{SI}_m(h, j_m)$: score de l'inactivité en fonction de la position de l'heure h dans le mois (combinaison de h et j_m) ;
- $\mathrm{SI}_a(h, j_m, m)$: score de l'inactivité en fonction de la position de l'heure h dans l'année (combinaison de h, j_m et m).

De plus, le MI doit prendre en compte le rôle de l'échelle de temps (c'est-à-dire de chaque type de score SI) dans l'inactivité de la VM. Par exemple pour l'exemple précédent du site Web des résultats de diplôme, on remarque que la position de l'heure dans la semaine (SI_s) a peu d'importance dans la périodicité de l'inactivité. Par conséquent, on définit quatre poids w_X, un pour chaque échelle de temps. Leurs valeurs sont corrigées après la mise à jour des scores SI (voir section 3.5).

Les composants du modèle d'inactivité pour les différentes échelles de temps sont résumés dans le tableau 3.2.

3.4.3. Calcul de la probabilité d'inactivité (PI)

À partir du MI d'une VM, on calcule sa PI pour l'heure h du jour j_s de la semaine, qui est aussi le jour j_m du mois m ; avec la formule donnée par l'équation (3.1).

TABLE 3.2. – Composants du modèle d'inactivité d'une VM.

Échelle	Poids	Scores SI	Variable
jour	w_j	24 $\mathrm{SI}_j(h)$	$0 \leq h < 24$
semaine	w_s	24×7 $\mathrm{SI}_s(h, j_s)$	$0 \leq j_s < 7$
mois	w_m	24×31 $\mathrm{SI}_m(h, j_m)$	$0 \leq j_m < 31$
année	w_a	24×365 $\mathrm{SI}_y(h, j_m, m)$	$0 \leq m < 12$

$$\begin{aligned} \mathrm{PI}(h, j_s, j_m, m) &= w_j \cdot \mathrm{SI}_j(h) + w_s \cdot \mathrm{SI}_s(h, j_s) + \\ &\quad w_m \cdot \mathrm{SI}_m(h, j_m) + w_a \cdot \mathrm{SI}_a(h, j_m, m) \\ &= \mathbf{w}^\intercal \cdot \mathrm{SI} \end{aligned} \tag{3.1}$$

$\mathbf{w}$ est le vecteur des poids, et $^\intercal$ est l'opérateur de transposition. SI est le vecteur des quatre scores SI associés à l'intervalle de temps pour lequel la PI est calculée.

3.4.4. Intégration dans OpenStack

Il s'agit donc d'intégrer l'utilisation du MI et de la PI dans un gestionnaire de DC. L'algorithme de consolidation de Drowsy-DC peut s'incorporer à n'importe quel système. OpenStack [101] est choisi à des fins d'illustration. On présente d'abord ses composants dédiés au placement des VM avant de décrire comment l'algorithme de Drowsy-DC s'y insère.

Placement et consolidation de VM avec OpenStack

Une opération de placement de VM peut être provoquée par deux raisons : la création d'une VM, et la consolidation dynamique.

Dans OpenStack, la première est prise en charge par le gestionnaire de la grappe, appelé Nova. Celui-ci inclut un `Filter Scheduler` (« ordonnanceur filtrant ») qui sélectionne les serveurs appropriés au placement de la VM, en exécutant les étapes suivantes :

1. éliminer les serveurs inappropriés en fonction d'un large éventail de paramètres tels que les ressources disponibles ;
2. pondérer et trier les serveurs restants en fonction de paramètres tels que le taux de colocation.

En ce qui concerne la consolidation dynamique d'une VM, OpenStack s'appuie sur Neat, qui divise ce problème en quatre sous-problèmes selon les travaux de Beloglazov et Buyya [18] :

1. identifier les serveurs sous-chargés : toutes leurs VM devraient être migrées ailleurs avant de les placer dans un état de faible énergie ;
2. identifier les serveurs surchargés : certaines de leurs VM devraient être migrées ailleurs pour assurer les critères de qualité de service ;
3. sélectionner les VM à migrer depuis les serveurs identifiés ;
4. placer les VM sélectionnées sur d'autres serveurs.

Nova et Neat sont particulièrement flexibles, et il est facile d'implémenter l'algorithme de placement en fonction de l'inactivité.

Intégration de l'algorithme de Drowsy-DC

On décrit donc ici l'intégration de Drowsy-DC dans le système de consolidation d'OpenStack. Le processus de consolidation final est illustré en figure 3.5.

Placement initial d'une VM Nova permet aisément d'ajouter des filtres et des pondérateurs. L'intégration de notre solution se fait en ajoutant un pondérateur qui va favoriser les serveurs dont la PI est la plus proche de la PI de la VM (voir section 3.4.1).

Migration d'une VM pour la consolidation dynamique Neat quant à lui, est conçu pour permettre l'insertion de nouveaux algorithmes de consolidation, en personnalisant l'algorithme en quatre étapes présenté ci-dessus. L'algorithme de Drowsy-DC touche aux points 3 et 4 :

Sélection des VM sélection des VM aux PI les plus éloignées de celle de leur serveur hôte, et examen des critères classique (vitesse de migration…) pour départager deux VM aux PI proches ;

Placement des VM en commençant par la plus grosse VM, filtrage des serveurs capables d'assurer ses requêtes, puis sélection du serveur avec la PI la plus proche.

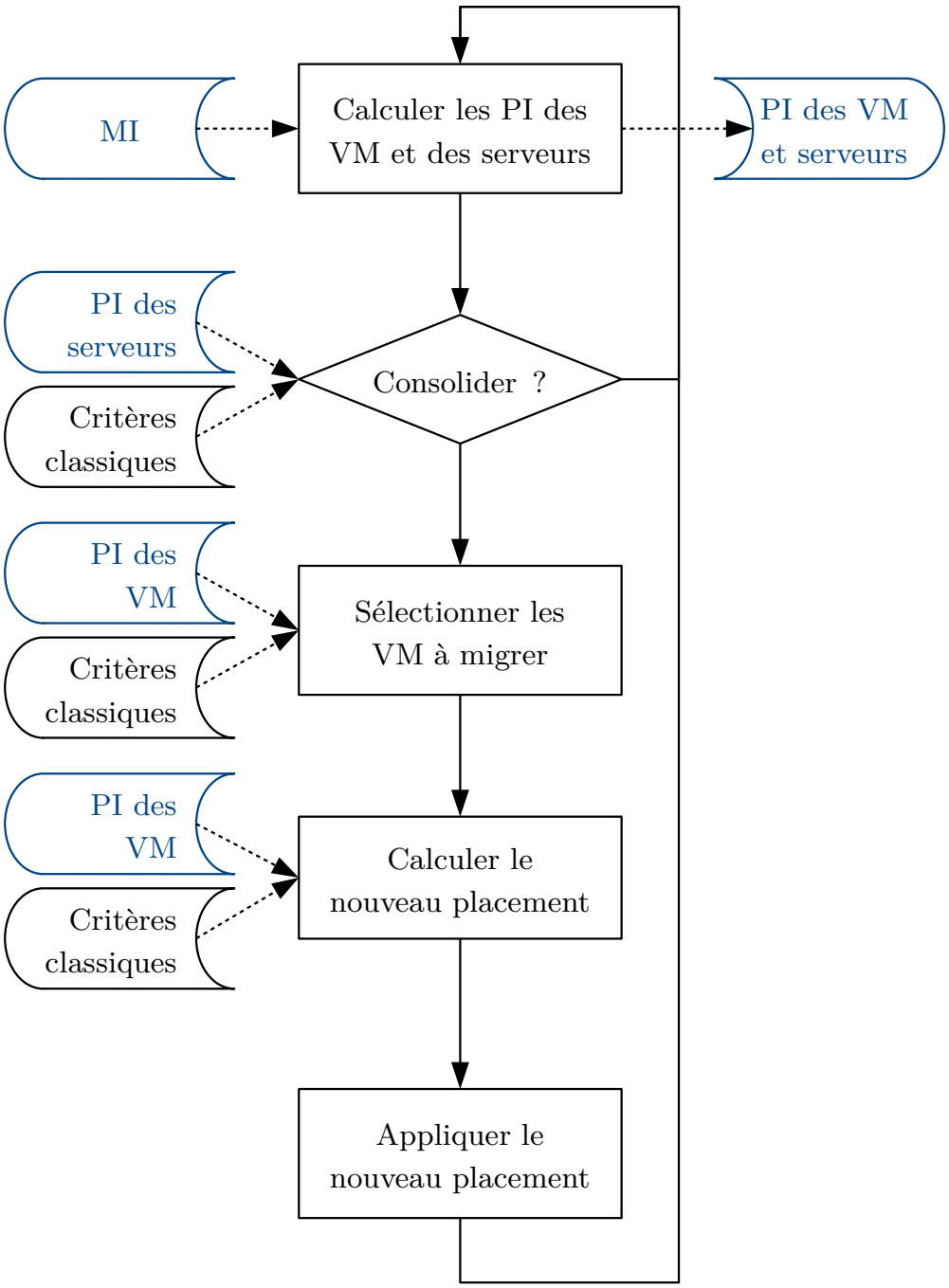


FIGURE 3.5. – Processus de consolidation de OpenStack, avec l'intégration de Drowsy-DC en bleu.

Étape supplémentaire : optimisation des PI des serveurs À ce stade, Neat a corrigé les problèmes de placement de VM qu'il a identifiés, c'est-à-dire les serveurs surchargés et sous-chargés. Drowsy-DC a seulement influencé les corrections avec la PI. Les VM occupent donc un ensemble de serveurs considéré comme minimal.

Drowsy-DC ajoute cependant une étape de consolidation supplémentaire, afin de corriger les problèmes de placement de VM liés à la PI. Elle reproduit la procédure de Neat avec les actions suivantes, détaillées ci-après :

1. identifier les serveurs aux intervalles de PI des VM trop grands ;
2. sélectionner les VM à migrer hors des serveurs identifiés ;
3. placer les VM sélectionnées sur d'autres serveurs.

Le point de départ est similaire à l'identification par Neat des serveurs sous-chargés et surchargés : sur un serveur donné, si la PI de la VM la plus active (c'est-à-dire la PI la plus basse) et la PI de la VM la plus inactive (c'est-à-dire la PI la plus haute) sont trop éloignées, alors Drowsy-DC doit migrer les VM avec les PI les plus extrêmes jusqu'à ce que l'intervalle des PI retombe sous un certain seuil. Ce seuil est défini empiriquement comme 7σ, avec σ le facteur appliqué au niveau d'activité tel que défini en section 3.5. Il représente environ l'impact d'une semaine entière d'activité maximale constante sur un SI_j.

Les VM sélectionnées pour la migration sont donc celles avec la PI la plus éloignée de la PI de leurs serveurs. Ces VM sont placées sur des serveurs choisis de la manière décrite précédemment : selon la proximité des PI des VM et des serveurs, mais en prenant bien sûr en compte l'avis de Neat afin de ne pas défaire la consolidation précédente en surchargeant ou souschargeant des serveurs.

3.4.5. Synthèse

L'objectif général de la consolidation basée sur la PI est *de placer sur les mêmes serveurs, les VM avec des PI similaires*. La logique derrière ce but est que les serveurs avec des VM présentant des PI fortes ont de fortes chances d'être suspendus. À l'inverse, les serveurs avec des VM aux PI faibles ne dormiront probablement jamais. Ces derniers sont des serveurs pour lesquels Drowsy-DC ne peut rien, parce qu'ils n'hébergent finalement pas de VM LLMI, et les problématiques courantes de performance et de gestion des ressources sont gérées par l'algorithme de consolidation classique.

Par ailleurs, les serveurs avec des VM aux PI moyennes, c'est-à-dire aux comportements indéterminés, sont séparés des serveurs avec des VM aux PI

fortes. Ils conservent cependant de meilleures chances d'être suspendus que les serveurs avec des VM aux PI faibles. Ils servent aussi de premiers hôtes aux VM nouvellement créées, jusqu'à ce que leur nature, plutôt active ou inactive, soit apprise.

Une dernière remarque concerne l'utilisation de la PI. De mauvaises estimations de la PI n'entraînent aucun impact sur les performances des applications. En effet, le MI et la PI sont uniquement employés à des fins de consolidation, comme des influences sur l'algorithme de placement. Les décisions de suspension ou de reprise des serveurs sont toujours prises en fonction de facteurs réels et observés, tels que l'activité des serveurs ou l'arrivée d'une requête. Ces décisions sont évaluées par les modules de suspension et de reprise décrit dans les chapitres qui suivent.

Pour aller plus loin, Drowsy-DC pourrait être intégré à l'orchestrateur de conteneurs dans un environnement multi-virtualisé. De la même manière que Drowsy-DC consolide les VM en fonction de leurs PI et de celles des serveurs, un orchestrateur devrait placer ses conteneurs en fonction de leurs PI et de celles des VM. En fait, on modéliserait l'inactivité des conteneurs plutôt que des VM, et la PI d'une VM deviendrait la moyenne des PI de ses conteneurs, de la même manière que la PI d'un serveur est la moyenne des PI de ses VM. Ainsi, il serait possible de consolider la charge de travail à deux niveaux grâce à la multi-virtualisation, ce qui complexifierait la tâche mais apporterait certainement de meilleurs résultats grâce à une flexibilité et une finesse de placement accrues.

3.5. Mise à jour du modèle d'inactivité

La section 3.4.2 explique que le MI d'une VM est modifié à chaque heure : ses scores SI sont mis à jour, et ses poids sont corrigés.

3.5.1. Calcul des scores SI_X

À la création de la VM, tous les SI_X sont initialisés à 0. C'est la valeur qui représente le comportement indéterminé. Ils sont ensuite toujours compris dans l'intervalle $[-1, 1]$. De plus, ils sont mis à jour à chaque fin d'heure de la manière suivante : si la VM a été observée inactive durant l'heure entière, les SI_X sont incrémentés, sinon ils sont décrémentés.

La valeur absolue de l'incrément ou du décrément, notée $v(\mathrm{SI}_X)$, est obtenue par l'équation (3.2). Dans celle-ci, deux facteurs interviennent :

1. a^* : valeur de base pour l'incrément, elle représente le niveau d'activité de la VM ;
2. $u(|\mathrm{SI}_X|)$: valeur correctrice du comportement du SI_X

$$v(\mathrm{SI}_X) = a^* \cdot u(|\mathrm{SI}_X|) \tag{3.2}$$

On décrit ci-dessous ces deux facteurs.

Valeur de base : niveau d'activité a^*

La valeur de base a^* est obtenue à partir du niveau d'activité a de la VM. Il y a deux cas :

VM active $a = a_h$, qui est le niveau d'activité durant l'heure observée ;

VM inactive $a = \overline{a}$, qui est le niveau moyen de l'activité de la VM observé pendant toutes les heures d'activité passées de la VM.

a_h est en fait le ratio de quanta de temps processeur alloué à la VM par l'orchestrateur de l'hyperviseur, par rapport au nombre total de quanta disponibles sur une heure. Un filtrage est opéré sur la chronologie des allocations de quanta à la VM afin d'éliminer les allocations de quanta très courtes. Celles-ci ne sont pas indicatrices d'une réelle activité de la VM, et ce filtrage est nécessaire pour identifier des cas où $a_h = 0$, c'est-à-dire où la VM est inactive.

Ce choix de la valeur de a quand la VM est inactive, permet d'incrémenter rapidement les SI_X d'une heure inactive de la VM quand la VM était observée très active le reste du temps, puisque cette inactivité est significative.

Finalement, le choix de la valeur du niveau d'activité a pour la valeur de mise à jour v est résumé par l'équation (3.3).

$$a = \begin{cases} a_h, & \text{si } a_h > 0. \\ \overline{a}, & \text{si } a_h = 0. \end{cases} \tag{3.3}$$

Cependant, a est une valeur brute qu'il convient de traiter avant de l'incorporer au SI_X. On applique donc ensuite un facteur à a, ce qui produit a^* par l'équation (3.4), afin de contrôler son impact sur l'évolution du SI_X. Le facteur $\sigma = \frac{1}{365 \times 24}$ est défini de telle sorte qu'une VM ait besoin d'une activité maximale ($a_h = 1$) pendant toute une année pour porter son SI_j de sa valeur initiale 0 à son minimum -1 (indépendamment du coefficient u qui est fonction de SI_X, décrit plus bas).

$$a^* = \sigma \cdot a = \frac{a}{365 \times 24} \tag{3.4}$$

Correction du comportement du SI_X : $u(|\mathrm{SI}_X|)$

Le second facteur de v le fait dépendre de la valeur actuelle du SI_X. C'est le coefficient $u(|\mathrm{SI}_X|)$ qui exprime cette dépendance. On remarquera qu'il utilise la valeur absolue du SI_X. Son but est double :

1. faire varier rapidement le SI_X lorsqu'il est encore indéterminé, afin d'apprendre rapidement le comportement de la VM ;
2. empêcher le SI_X d'atteindre des valeurs extrêmes, pour permettre au MI de répondre à un changement dans le comportement de la VM.

Autrement dit, plus le SI_X est proche de 0, et plus v doit être grand [3] ; et à l'inverse, plus le SI_X est proche de ses bornes -1 ou 1, et plus v doit être petit. Ce comportement est donc obtenu grâce à ce coefficient u.

Dans l'équation (3.5), α et β paramétrisent l'impact de u :

— α peut être interprété comme la vitesse à laquelle la valeur de mise à jour v décroît, lorsque le seuil définit par β est atteint par $|\mathrm{SI}_X|$;

— de fait, β peut être interprété comme la valeur seuil de $|\mathrm{SI}_X|$ à partir de laquelle on considère que le SI_X atteint des valeurs extrêmes.

On définit empiriquement $\alpha = 0.7$, ainsi que $\beta = 0.5$ parce que c'est la valeur médiane entre la détermination totale du comportement de la VM, et son indétermination. Ces deux paramètres servent aussi de voies d'amélioration du MI, en ajustant leurs valeurs, voire en les définissant dynamiquement en fonction des variations constatées du niveau d'activité.

$$u(|\mathrm{SI}_X|) = \frac{1}{1 + \mathrm{e}^{\alpha(|\mathrm{SI}_*| - \beta)}} \tag{3.5}$$

3.5.2. Calcul des poids w_X

Maintenant que les SI_X ont été mis à jour, il faut corriger les poids w_X.

Les poids sont appris tout au long de la vie de la VM. Ils sont recalculés et corrigés à la fin de chaque heure. Pour ce faire, on cherche à minimiser la fonction d'erreur quadratique Q définie en équation (3.6).

$$Q(\mathbf{w}) = (\mathrm{PI}' - \mathrm{PI})^2 \tag{3.6}$$

3. On rappelle que v est toujours positif, et sa valeur est ajoutée ou ôtée du SI_X selon que la VM était, respectivement, inactive ou active.

PI' est la PI qui aurait dû être calculée si le modèle était parfait, et PI est la PI qui est calculée avec les poids actuellement en cours de mise à jour. Cependant, étant donné la nature de la PI qui est une estimation de la probabilité, il n'existe pas de véritable valeur correcte pour PI'. Ainsi, ce terme est remplacé par l'expression mixte donnée par l'équation (3.7).

$$\mathrm{PI}' = \mathbf{w_0}^\intercal \cdot \mathrm{SI}' \tag{3.7}$$

$\mathbf{w_0}^\intercal$ est la transposée du vecteur des poids avec leurs valeurs au début de h, et SI' est le vecteur des quatre scores SI dont les valeurs ont été mises à jour.

Par conséquent, l'expression de la fonction d'erreur quadratique qui doit être minimisée pour l'apprentissage des poids est donnée par l'équation (3.8).

$$Q(\mathbf{w}) = \left(\mathbf{w_0}^\intercal \cdot \mathrm{SI}' - \mathbf{w}^\intercal \cdot \mathrm{SI}\right)^2 \tag{3.8}$$

Une méthode simple de minimisation de cette fonction est l'algorithme du gradient (ou de la plus forte pente, *steepest descent* en anglais) [76]. Il s'agit de faire progresser $\mathbf{w}$ itérativement par pas proportionnels au négatif du gradient de Q, ce qui finit par converger vers une valeur de $\mathbf{w}$ qui minimise localement l'erreur quadratique. Cette méthode a l'avantage d'être rapide tout en produisant de bons résultats, malgré son défaut connu de se perdre parfois sur un minimum local. On peut de plus régler sa précision de sorte à ne pas pénaliser la vitesse du système de consolidation.

3.6. Suspension des serveurs

L'introduction de l'algorithme de consolidation de Drowsy-DC dans le système permet de créer des serveurs qui hébergent des VM inactives. Ils sont donc eux-mêmes inactifs et peuvent être suspendus. Comme expliqué en section 3.3, c'est le module de suspension qui est chargé de suspendre les serveurs.

C'est un ajout logiciel sur chaque serveur. Il transmet des informations cruciales au module de reprise, notamment une date de reprise, comme décrit plus loin en section 3.7.2, au moment de la suspension. Mais son rôle est avant tout de prendre la décision de suspendre le serveur, et d'amorcer cette suspension.

3.6.1. Détection de l'inactivité

De prime abord, on peut considérer qu'un système est inactif si aucun de ses processus n'est en cours d'exécution. Cette approche produit des *faux négatifs et des faux positifs*.

Les faux négatifs sont des processus qui sont en cours d'exécution mais qui devraient être ignorés dans ce contexte. Ils incluent les programmes de surveillance qui s'exécutent sur le serveur, ainsi que les services d'arrière-plan liés au noyau. Ils sont éliminés par l'introduction d'une liste noire.

Les faux positifs sont les processus d'une VM qui ne sont pas en cours d'exécution, mais qui fournissent un service qui ne doit pas être considéré comme inactif. D'une part, un processus peut être bloqué en attente de ressources, par exemple sur une lecture disque. Dans ce cas, le serveur ne doit pas être suspendu. On en déduit qu'il faut déterminer la *raison* pour laquelle un processus n'est pas en cours d'exécution. D'autre part, un processus d'une VM peut être inactif sans que le service auquel il correspond le soit effectivement. Par exemple, il peut avoir des sessions en cours, ou des connexions SSH ou TCP ouvertes. Si aucune donné n'est échangée sur celles-ci, la VM semble inactive mais la suspension du serveur provoquerait une importante latence.

L'identification de ce second type de faux positifs nécessite une méthode d'introspection de la VM [65] pour obtenir des paramètres qui reflètent l'activité réelle de la VM. Cette technique n'est pas implémentée dans Drowsy-DC pour deux raisons :

1. le système ne doit pas nécessiter de modification des SE invités [4] ni des applications ;
2. l'impact potentiel sur les performances est réduit par l'optimisation de la reprise des serveurs (voir section 3.8).

Il existe aussi des heuristiques qui prennent en compte la proportion de ressources actuellement utilisées par une VM. On peut par exemple considérer le taux de salissement des pages (en anglais *page dirtying*), qui peut être suivi depuis l'hyperviseur [146].

Inactivité de la double virtualisation

On remarque que les difficultés de détection de l'inactivité proviennent en fait de l'aspect « boîte noire » des VM. Comme Drowsy-DC agit au niveau de l'hyperviseur, il n'a pas d'information sur les applications qui s'exécutent dans les VM, et il faut trouver des méthodes détournées. Ce

4. Indépendamment de la para-virtualisation, voir section 2.2.2.

problème pourrait être résolu par la double virtualisation. En effet, le moteur de conteneurs (qui permet la virtualisation du système d'exploitation, imbriquée dans la virtualisation du matériel, voir section 2.2.2) gère des entités virtualisées beaucoup plus transparentes. Il possède toutes les informations sur les ressources actuellement utilisées par les applications, leurs connexions ouvertes, etc. Il suffirait donc que le moteur de conteneurs transmette ces informations à Drowsy-DC ; c'est-à-dire que le système de virtualisation niveau SE communique ces informations au niveau matériel.

3.6.2. Délai de sursis

Pour peser la décision de suspendre un serveur, le module prend en compte un *délai de sursis*. Après la reprise d'un serveur, ce dernier ne peut pas être suspendu à nouveau pendant une certaine durée, même s'il est avéré inactif. Ce sursis permet d'éviter aux serveurs d'alterner trop rapidement entre les états actif et de faible énergie, ce qui provoquerait une qualité de service très dégradée et une consommation énergétique augmentée.

Le délai de sursis est calculé par le module de suspension au moment de la reprise de son serveur, en fonction de la probabilité d'inactivité (PI, voir section 3.4). Si la PI indique que le serveur est probablement actif pendant la période actuelle, le délai est allongé afin d'éviter une dégradation des services qui sont prédits actifs. Le délai de sursis est défini empiriquement entre 5 s et 2 min. Il augmente exponentiellement lorsque la PI diminue, pour des raisons de précaution vis-à-vis de la qualité de service des VM actives ou au comportement indéterminé.

3.7. Reprise des serveurs

Un pré-requis essentiel de Drowsy-DC est de garantir une reprise rapide du serveur. C'est la responsabilité du *module de reprise*, un ajout logiciel placé sur un serveur qui gère le DC, et va donc toujours être actif (il ne sert pas à héberger des VM). Ce système peut facilement passer à l'échelle en déployant un module de reprise par étagère de serveurs (par *rack*). Dans le prototype présenté ici, il est placé sur le commutateur de réseau défini par logiciel (en anglais *Software Defined Network* (SDN) *switch*). De plus, étant donné le rôle critique du module de reprise pour Drowsy-DC, son implémentation doit être résistante aux défaillances. Pour ce faire, tous les modules agissent en collaboration en se surveillant mutuellement (mécanisme de pulsation régulière) et en se répliquant.

Le rôle du module de reprise est donc de sortir les serveurs de leur suspension. La reprise d'un serveur peut être déclenchée pour deux raisons :

1. une requête réseau entrante ;
2. une reprise plannifiée.

3.7.1. Reprise sur requête entrante

Le module de reprise inclut un analyseur de paquets simplifié. Chaque requête reçue par le commutateur réseau est inspectée. Il s'agit de déterminer si la VM à laquelle elle est destinée, est hébergée sur un serveur actuellement suspendu. La vérification de la relation entre les VM et les serveurs est rendue efficace par une table de hachage (en anglais *hashmap*). Celle-ci fait le lien entre les adresses IP des VM et les adresses Ethernet MAC des serveurs qui les hébergent. Cette table est mise à jour à partir des informations qu'un module de suspension envoie au moment de suspendre son serveur (voir section 3.6). Enfin, si le serveur qui héberge la VM destination de la requête est suspendu, le module de reprise met fin à la suspension en envoyant un paquet Ethernet Wake-on-LAN (WoL).

3.7.2. Reprise planifiée

Au moment de la suspension de son serveur, un module de suspension détermine une *date de reprise*. Pour ce faire, il liste les minuteurs haute définition (en anglais *high resolution timers*) [5] enregistrés dans le noyau. En effet, lorsqu'un processus se suspend, il enregistre un tel minuteur afin de planifier sa reprise par le noyau. Par conséquent, la date de reprise d'un serveur est le minuteur qui va se déclencher le plus tôt. En pratique, cette information est extraite du noyau Linux à l'aide d'un module spécialement développé, qui parcourt la structure en arbre rouge-noir (en anglais *red-black tree*) utilisée par le noyau pour conserver les minuteurs.

Cette méthode peut produire de faux positifs, c'est-à-dire sélectionner les minuteurs de processus qui ne devraient pas déclencher la reprise d'un serveur. De tels processus sont vraisemblablement les mêmes que les faux négatifs que le module de suspension ignore avec une liste noire lorsqu'il vérifie l'inactivité de son serveur (voir section 3.6.1). Il faut donc filtrer les minuteurs en fonction des processus qui les ont enregistrés.

5. Ces minuteurs sont dits « haute définition » parce qu'ils sont implémentés par le noyau Linux avec une résolution de quelques nanosecondes. Leur résolution n'a pas d'intérêt pour Drowsy-DC.

On remarque qu'après ce filtrage, le module de suspension peut tout à fait se retrouver sans aucun minuteur valide lors de la détermination de la date de reprise. Cela signifie simplement qu'aucun traitement digne d'intérêt n'est prévu. Le serveur peut donc rester suspendu indéfiniment jusqu'à sa reprise pour une requête entrante (voir ci-dessus section 3.7.1).

Finalement, avant de suspendre son serveur, le module de suspension envoie au module de reprise la date de reprise qu'il a déterminée. Ce dernier maintient une autre table de hachage, qui fait le lien entre les dates de reprise et les adresses Ethernet MAC des serveurs correspondants. Ainsi, lorsqu'une date de reprise approche, le module envoie un paquet WoL au serveur indiqué et retire l'entrée de la table. Cette reprise est commandée en avance afin de compenser la durée du processus de reprise du serveur (voir section 3.8).

3.7.3. Reprise de serveurs doublement virtualisés

La double virtualisation pourrait aider le module de reprise à jouer son rôle.

D'abord pour la reprise sur requête entrante, le moteur de conteneurs a une notion plus précise du trafic réseau vers ses conteneurs, puisque c'est lui qui virtualise leur réseau. Comme il connaît aussi leur placement sur les VM, cela faciliterait l'analyse de la requête pour déterminer la destination. Cela ne dispenserait cependant pas de maintenir le lien entre les VM et les serveurs qui les hébergent.

En ce qui concerne la reprise planifiée, la détermination de la date de reprise serait grandement simplifiée de la même manière que pour la détection de l'inactivité pour le module de suspension (voir section 3.6.1). En effet, le moteur de conteneurs connaît exactement quels sont les processus des applications qui ont un intérêt, et le filtrage des minuteurs serait donc absolument exact et éviterait tous les faux positifs.

3.8. Optimisation de la reprise des serveurs

La réduction de la durée de reprise des serveurs est essentielle à Drowsy-DC. Le système ne cherche pas à exploiter la moindre période creuse pour suspendre un serveur, donc l'importance de la durée n'est pas fondamentale pour les économies d'énergie. Minimiser la durée de reprise est cependant une nécessité pour ne pas dégrader les performances des applications. En effet, la reprise d'un serveur peut être déclenchée par l'arrivée d'une requête

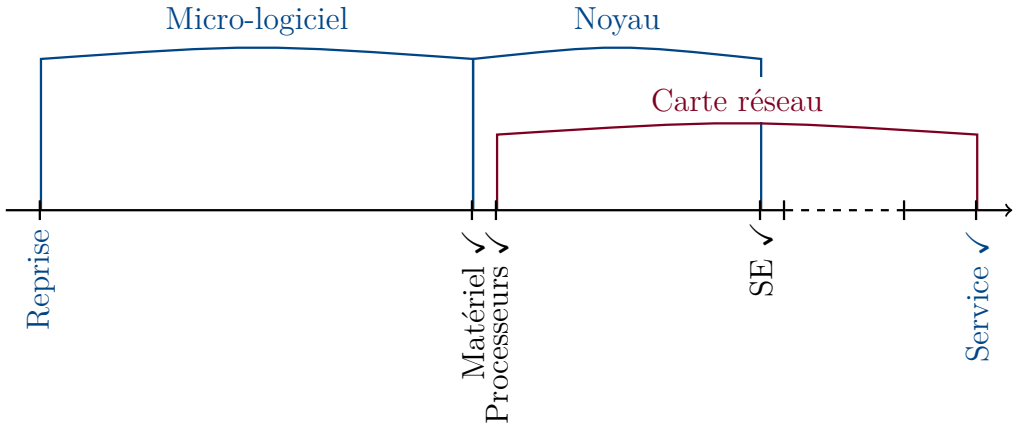


FIGURE 3.6. – Processus de reprise d'un serveur et latence jusqu'à la reprise du service hébergé.

(voir section 3.7). Il faut donc servir cette requête le plus rapidement possible. On notera d'ailleurs que seule la première requête d'une série sera ainsi retardée, puisque le serveur sera bien actif pour les suivantes.

Tout d'abord, la figure 3.6 représente les phases du processus de reprise d'un serveur sous Linux et leurs durées. Il y a en particulier trois phases importantes :

1. reprise du micro-logiciel (en anglais *firmware*) : environ 475 ms ;
2. reprise du noyau : environ 300 ms ;
3. réinitialisation de la carte réseau [6] : 1,5 s.

Il y a un certain chevauchement de la réinitialisation de la carte réseau sur la reprise du noyau. En fin de compte, la latence totale de réveil d'un service est d'environ 2 s.

La première phase de la reprise est cependant indépendante du noyau, car il s'agit de relancer les composants très bas niveau du serveur. La participation active des constructeurs du matériel est nécessaire pour réduire la durée de cette phase. On décrit donc maintenant l'optimisation des deux autres phases.

6. Les temps sont donnés pour une carte réseau Ethernet, qui est le matériel le plus couramment accessible.

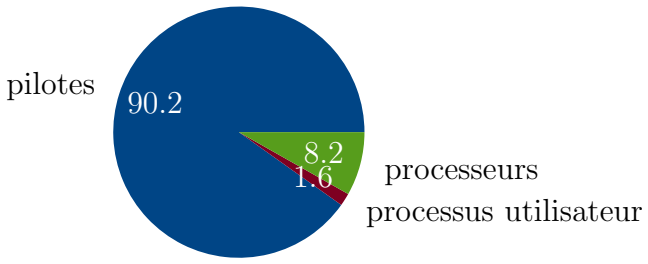


FIGURE 3.7. – Phases de reprise du noyau Linux.

3.8.1. Reprise du noyau

Cette phase se divise en trois parties, dont les durées relatives sont illustrées en figure 3.7 :

1. reprise des processeurs : environ 25 ms ;
2. reprise des pilotes : environ 275 ms ;
3. reprise des processus dans l'espace utilisateur : environ 5 ms.

Comme la troisième a une durée négligeable face aux deux autres, ce sont ces dernières qui ont été optimisées pour Drowsy-DC.

Premièrement, l'outil `pm-graph` d'Intel [107] permet l'analyse de toute la phase de reprise du noyau. On y observe notamment que les processeurs sont relancés séquentiellement. La séquentialité est nécessaire pour des raisons techniques. Cependant, seule la VM à qui la requête est destinée aura besoin de ressources. On peut donc réduire cette durée en ne relançant que le nombre de processeurs nécessaires à cette VM ; les processeurs restants seront relancés ultérieurement.

Deuxièmement, la reprise des pilotes représente en fait la plus longue partie de la reprise du noyau. Elle est parallélisée, cependant certains pilotes peuvent être désactivés étant donné qu'ils sont utilisés par des périphériques inutilisés. Par exemple, le module du noyau Linux `i915` gère la puce graphique VGA, largement optionnelle pour un serveur. La désactivation de tous ces modules inutiles réduit effectivement la durée de reprise du noyau à un minimum : du fait du parallélisme, c'est la durée de reprise du pilote le plus lent. Sur la machine d'expérimentation, c'est celui de la carte réseau, il est bien sûr nécessaire.

Le gain de vitesse réel dépend des modules exacts utilisés sur un serveur. Mais étant donné que le serveur est relancé pour une VM en particulier, on peut éliminer plus de pilotes en ne relançant que ceux qui fournissent les ressources à cette VM : le disque qui contient son image, l'interface réseau qu'elle utilise, etc.

3.8.2. Reprise de la carte réseau

C'est la phase la plus longue de la reprise du serveur, elle dure environ 1,5 s. Bien qu'il faille effectivement relancer son pilote (voir ci-dessus), celui-ci provoque la *réinitialisation de la carte réseau*. La conséquence est que les procédures d'auto-négociation du protocole au plus bas niveau doivent être effectuées à nouveau, par exemple pour déterminer la bande passante du lien physique.

Pour corriger ce problème, des cartes réseau Intel (comme le modèle `I350` [57]) sont construites autour de puces électroniques capables de maintenir le lien physique alors que le serveur est suspendu. Cette fonctionnalité est nommée « session critique » [7] (en anglais *critical session*) [67]. Maintenir le lien physique actif a un impact négligeable sur la consommation électrique du serveur suspendu. Pour le modèle `I350`, la différence est inférieure à 2 mW [57].

La combinaison de toutes les optimisations de la reprise du serveur devrait porter sa durée totale à moins de 800 ms.

3.9. Évaluation

On évalue dans cette section le système Drowsy-DC : sa capacité à suspendre des serveurs, et donc les économies d'énergie qu'il permet ; mais aussi son impact sur le DC, avec la consommation en ressources de ses modules logiciels et le nombre de migrations de VM.

3.9.1. Évaluation en environnement réel

Méthodologie

Cette expérience doit démontrer l'efficacité de Drowsy-DC. On force le système à reconsolider les VM périodiquement, plutôt que d'attendre

7. On retrouve aussi cette fonctionnalité sous les noms de *veto bit*, *manageability session*, ou encore sous la capacité *keep PHY link up*, c'est-à-dire de garder le lien physique actif.

TABLE 3.3. – Capacités des systèmes évalués.

Nom	Consolidation	Serveur inactif	Serveur vide
Neat	Neat	*néant*	hibernation
Neat & susp.	Neat	suspension	hibernation
Drowsy-DC	Drowsy-DC	suspension	hibernation

que la migration soit nécessaire comme dans le cas normal (par exemple lorsqu'un serveur est surchargé). Ce comportement n'est pas applicable dans un véritable DC parce qu'il provoquerait un trop grand nombre de migrations. La conséquence en serait une dégradation des performances de tout le DC. Mais dans le cadre de cette expérience, il permet d'observer l'effet de Drowsy-DC.

On évalue ici Drowsy-DC, avec l'état de faible énergie (suspension) activé sur les serveurs. On le compare à Neat, le composant de consolidation dynamique de OpenStack, avec la possibilité d'éteindre les serveurs vides, mais pas de suspendre les serveurs inactifs. Mais on le compare aussi à Neat en lui permettant de les suspendre, afin de démontrer l'importance de la consolidation basée sur l'inactivité. Dans ce cas, on utilise la même méthode que Drowsy-DC pour prendre les décisions de suspension et de reprise, mais sans le délai de sursis puisqu'il repose sur la PI, qui est une propriété de Drowsy-DC (voir section 3.6). Dans tous les cas, les serveurs peuvent aussi être endormis, c'est-à-dire placés dans un état de plus faible énergie que la suspension (voir section 3.2). Les capacités en termes d'états de faible énergie des trois systèmes sont résumées dans le tableau 3.3.

Installation

L'expérience se déroule sur une grappe gérée par OpenStack, constituée de six machines HP, notées P_1 à P_6 qui intègrent chacune un processeur Intel® Core® i7-3770 cadencé à 3,40 GHz, 16 Gio de mémoire, une carte réseau Ethernet 10 Gbit/s et exécutant la distribution Linux Ubuntu Server 14.04. Toutes les machines sont capables d'être suspendues, c'est-à-dire d'atteindre le niveau de faible énergie ACPI S3 *suspend to RAM*. Dans cet état, elles consomment environ 5 W de puissance instantanée, ce qui représente 10% de la consommation de la machine dans son état actif mais sans charge de travail. De plus, les serveurs sont virtualisés avec le couple QEMU et KVM. Ils sont reliés par un commutateur de réseau défini par logiciel (*SDN switch*) fourni par P_1. Ce serveur héberge aussi le module de

reprise ainsi que les contrôleurs de OpenStack. Les serveurs P_2 à P_5 sont utilisés par OpenStack pour placer des VM.

La grappe héberge au total 8 VM, qui se voient allouées 6 Gio de mémoire et 5 vCPU chacune, ce qui représente un maximum de 2 VM par serveur. Parmi ces 8 VM, il y a :

— 2 VM à longue durée de vie, surtout actives (LLMU), notées V_1 et V_2 ;
— 6 VM à longue durée de vie, surtout inactives (LLMI), notées V_3 à V_8.

Au départ, les 2 VM LLMU sont placées sur des machines distinctes.

Chacune de ces 8 VM exécute une application de la suite de banc d'essai CloudSuite [46] : `Media Streaming` pour les 2 VM LLMU, et `Web Search` pour les 6 VM LLMI. C'est le serveur P_6 qui héberge les simulateurs de clients de CloudSuite pour générer une charge de travail vers ces applications. Les simulateurs de clients pour `Web Search` sont configurés pour répéter des traces d'utilisation relevées pendant sept jours dans le DC privé de Nutanix. En particulier, V_3 et V_4 rejouent en fait l'exacte même trace. Ces traces ont été exposées en figure 3.3.

Résultats globaux

On montre d'abord en figure 3.8 la proportion de temps pendant laquelle une VM a été colocalisée avec chaque autre VM dans la grappe gérée par Drowsy-DC uniquement. La dernière colonne donne le nombre de migrations subies par chaque VM.

Ces résultats confirment en premier lieu que Drowsy-DC a bien identifié V_1 et V_2 comme étant des VM LLMU, et les a donc rangées sur le même serveur pour la majeure partie de l'expérience. De plus, le système a aussi bien identifié les motifs d'inactivité semblables des VM LLMI, en particulier V_3 et V_4 qui recevaient exactement la même charge de travail comme mentionné plus haut. On voit en effet qu'elles aussi, ont été colocalisées sur le même serveur pendant une très longue durée, et après une seule migration. Enfin, on voit que le nombre de migrations subies par chaque VM est relativement faible, ce qui signifie qu'une VM migrée atteint facilement un état stable.

On donne en tableau 3.4 la proportion de temps que chaque serveur (hormis P_1 et P_6 bien sûr) a passé dans l'état suspendu, avec Drowsy-DC et avec Neat. Étant donné qu'il y avait 8 VM pour 4 serveurs, et que chaque serveur ne peut accueillir qu'exactement 2 VM, aucun serveur n'a été vide de VM, et donc n'a été endormi. Au total, les serveurs gérés par Drowsy-DC ont été suspendus 35% plus longtemps que lorsqu'ils étaient gérés par

	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	#mig
V_1	100	85	0	0	0	15	0	0	1
V_2	85	100	0	0	0	0	0	15	0
V_3	0	0	100	76	0	0	24	0	0
V_4	0	0	76	100	23	0	0	1	1
V_5	0	0	0	23	100	77	0	0	2
V_6	15	0	0	0	77	100	0	8	1
V_7	0	0	24	0	0	0	100	76	1
V_8	0	15	0	1	0	8	76	100	3

FIGURE 3.8. – Proportion de temps de colocalisation de chaque VM. Cases rouges : VM V_1 et V_2, qui sont LLMU ; cases bleues : VM V_3 et V_4, qui sont des VM LLMI avec exactement la même charge de travail. La dernière colonne donne le nombre de migrations subies par chaque VM.

TABLE 3.4. – Proportion de temps (pourcentage) passé par chaque serveur dans l'état suspendu.

Algorithme	P_2	P_3	P_4	P_5	Total
Drowsy-DC	0	94	79	91	66
Neat & susp.	89	7	8	93	49

Neat. Autrement dit, l'algorithme de consolidation de Drowsy-DC basé sur l'inactivité, qui cherche à optimiser le placement des VM pour maximiser les périodes de suspension des serveurs, a augmenté la durée totale de ces périodes de 35% par rapport à Neat. On remarque bien dans le cas de l'évaluation de Drowsy-DC que le serveur P_2 est celui qui a rapidement hébergé les deux VM LLMU (V_2 était initialement placée ici, et V_3 l'a rejointe par consolidation), puisqu'il n'a jamais été suspendu.

Dans cette expérience, l'effet de Drowsy-DC est une réduction de la consommation énergétique totale d'environ 55%, avec 18 kWh au lieu de 40 kWh avec Neat sans suspension des serveurs. L'activation de la suspension des serveurs pour Neat, comme décrit plus haut, réduit la consommation à 24 kWh, on peut donc considérer que l'algorithme de consolidation permet d'économiser 27% d'énergie en plus de la simple utilisation de la suspension des serveurs. On notera aussi que l'efficacité de Drowsy-DC augmente avec le temps. D'une part, les modèles d'inactivité deviennent toujours plus précis ; et d'autre part, l'algorithme de consolidation continuera de prendre

des décisions de placement optimisées pour l'économie d'énergie.

Enfin, cette expérience permet de constater que Drowsy-DC garantit le SLA des applications. Par exemple, plus de 99% des requêtes pour le service `Web Search` ont été servies en moins de 200 ms, comme requis par CloudSuite. Cependant, le temps de réponse des requêtes qui ont provoqué une reprise d'un serveur a été observé jusqu'à 1500 ms. Ce constat n'impacte pas le SLA général parce que ces requêtes sont une minorité. Par ailleurs, l'optimisation de la reprise porterait le temps de reprise d'un serveur plus bas encore, à environ 800 ms (voir section 3.8).

Applications basées sur des minuteurs

On ajoute ici une expérience supplémentaire réalisée avec une application basée sur des minuteurs pour son service. Dans notre cas, c'est un simple service de sauvegarde qui est déclenché périodiquement. Le bon fonctionnement des modules de suspension et de réveil a été observé. Le premier envoie la bonne date de reprise au second lors de la suspension ; et le second anticipe la date de reprise du serveur afin de compenser la durée de reprise en relançant le serveur en avance de 2 s.

Résultats spécifiques

Module de suspension

Le module de suspension est évalué sous trois angles :

1. efficience : détection de l'inactivité, prévention des alternances d'états énergétique et détermination de la date de reprise ;
2. impact négatif : consommation de ressources et durée de transition vers la suspension ;
3. passage à l'échelle : évolution de l'impact négatif avec l'augmentation du nombre de VM. [8]

L'efficience du module en ce qui concerne la détection de l'inactivité et la détermination de la date de reprise est proche de la perfection puisqu'il s'appuie directement sur les informations du noyau Linux qui héberge les VM avec KVM. L'apport de la double virtualisation permettrait l'exacte perfection puisqu'il n'y aurait plus besoin de recourir à des listes noires de processus. Pour ce qui est du délai de sursis, on a constaté que celui-ci évite environ 63% des alternances d'états énergétiques trop rapides ; et il ne gâche qu'environ 9% de temps de suspension potentiel. On définit une

8. Jusqu'à 32 VM, qui est le nombre maximum de VM qu'un hôte dédié de Amazon Web Services EC2 peut héberger.

alternance trop rapide comme une situation où le serveur est suspendu puis repris en moins de 2 min.

La durée de transition vers la suspension n'est pas rallongée par le module de suspension et reste d'environ 1 s. Ce temps est constant quel que soit le nombre de VM parce que la transition vers la suspension ne nécessite pas d'enregistrer sur le disque l'état des VM et du serveur, ce qui aurait pu poser des problèmes de passage à l'échelle [58]. Enfin, le module consomme moins de 0.1% des ressources processeur et mémoire du serveur quel que soit le nombre de VM, ce qui correspond bien aux tâches très simples dont il est responsable.

Module de reprise

Le module de reprise s'exécute sur le commutateur réseau de l'étagère de serveurs. Il est en gros évalué sur les mêmes points que le module de suspension, mais pour la reprise des serveurs.

Sur le sujet de la détection du besoin de relancer un serveur, on observe des résultats parfaits étant donné que cette fonctionnalité est basée sur des informations exactes. L'état d'un hôte et la liste des VM qui y sont hébergés sont envoyé par le module de suspension au module de reprise ; et la VM ciblée par une requête entrante est déterminée par l'adresse IP du paquet réseau.

Pour ce qui est du passage à l'échelle du module, plusieurs expériences faisant varier le volume de trafic réseau qu'il doit traiter (de 1 Gbit/s à 10 Gbit/s) ont été réalisées. Leurs résultats sont montrés en figure 3.9. On constate que l'utilisation mémoire est plutôt constante, à environ 9% de la machine qui l'héberge, tandis que l'utilisation du processeur augmente de 10% jusqu'à 16%.

Enfin, la durée de reprise d'un serveur est en fait l'addition de trois durées :

1. durée de détection : temps nécessaire au module de reprise pour déterminer le serveur de destination et qu'il faut le relancer ;
2. durée du réseau : temps de transmission du paquet Ethernet Wake-on-LAN au serveur à relancer ;
3. durée de reprise de la machine serveur.

La figure 3.10 montre que les durées de détection et de reprise du serveur sont à peu près constantes. Elles sont respectivement d'environ 50 ms et 2 s. La durée du réseau quant à elle augmente avec le volume du trafic, de 0,05 ms à 0,25 ms. Cependant, ce n'est pas un problème parce que cette durée est négligeable face à la durée de reprise du serveur. Cette dernière peut d'ailleurs être réduite à moins de 800 ms comme décrit en section 3.8.

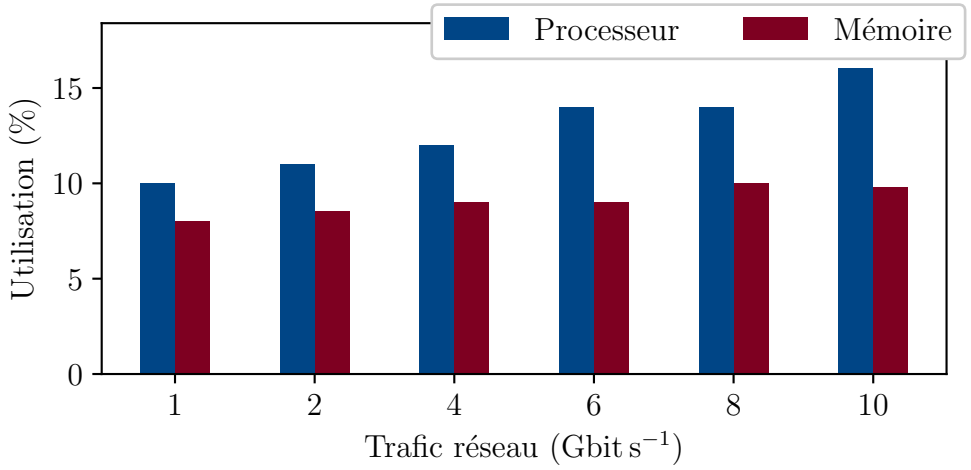


FIGURE 3.9. – Utilisation des ressources du module de reprise.

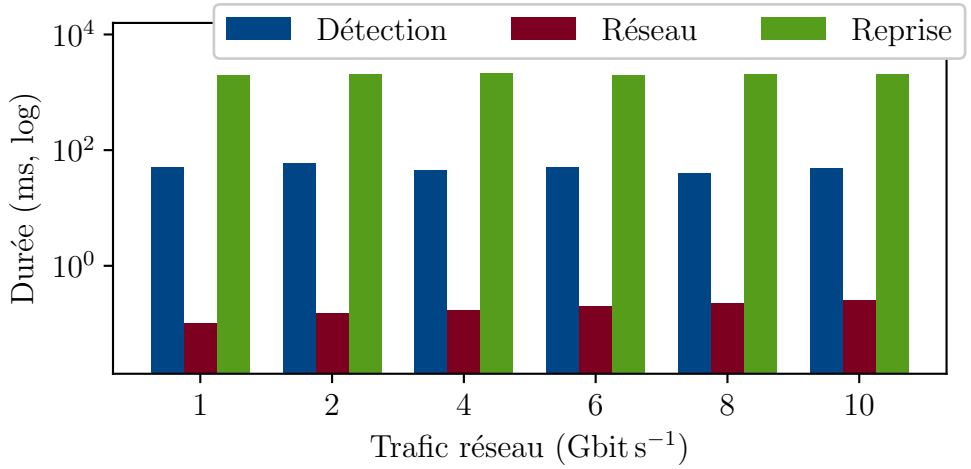


FIGURE 3.10. – Temps de reprise d'un serveur.

TABLE 3.5. – Traces d'activité de VM pour l'évaluation du MI.

Sous-fig.	Échelle de périodicité	Description
a	quotidienne	service de sauvegarde quotidien à 2h du matin
b	hebdomadaire, annuelle	publication de BD en ligne, sauf en juillet et en août
c – g	quotidienne, hebdomadaire	traces réelles d'un DC de cloud privé (voir figure 3.3)
h	aucune	VM à longue durée de vie, surtout active

TABLE 3.6. – Métriques d'efficacité de la prédiction. VP : nombre de vrais positifs ; VN : nombre de vrais négatifs ; inversement avec F pour les faux positifs et négatifs. Un cas est positif lorsque la VM est inactive ; ou que la VM est prédite inactive, c'est-à-dire que sa PI est supérieure à 0,5.

Rappel	Précision	F-mesure	Spécificité
$\frac{VP}{VP+FN}$	$\frac{VP}{VP+FP}$	$\frac{2\times rappel\times précision}{rappel+précision}$	$\frac{VN}{VN+FP}$

Modèle d'inactivité (MI)

On évalue ici la capacité du modèle à prédire les périodes d'inactivité. Pour ce faire, les MI de différents motifs d'inactivité sont construits sur trois ans, afin d'expérimenter avec des motifs annuels, plus complexes. Les motifs d'évaluation sont listés en tableau 3.5, où la première colonne renvoie aux sous-figures de la figure figure 3.11. Les traces a, b et h sont générées artificiellement, à partir des observations de traces comme décrit en section 3.4 ; les autres sont des traces réelles. Ces dernières durent initialement une semaine et ont été répétées sur trois ans.

L'objectif du MI est de prédire la possibilité que la VM soit inactive durant la prochaine heure. Par conséquent, on utilise quatre métriques classiques de l'efficacité des prédictions, décrites en tableau 3.6.

rappel sensible aux faux négatifs, c'est-à-dire aux cas où le modèle prédit de l'activité alors que la VM est en réalité inactive ;

précision sensible aux faux positifs, c'est-à-dire aux cas où le modèle prédit de l'inactivité alors que la VM est en réalité active ;

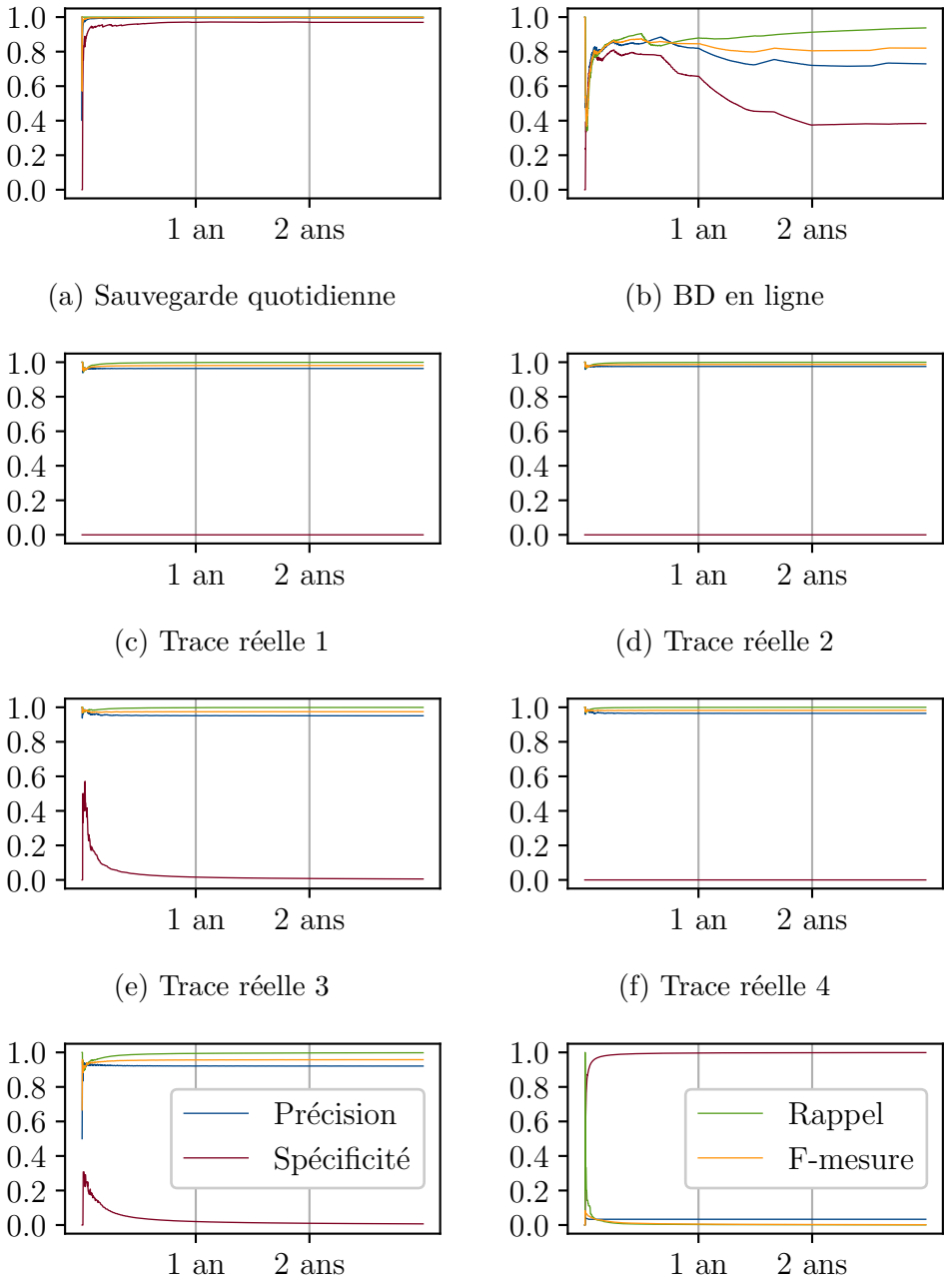
F-mesure résumé du rappel et de la précision ;

spécificité équivalent du rappel pour les cas négatifs.

La spécificité est en quelque sorte la mesure négative. Elle est importante parce qu'elle caractérise la capacité du modèle à bien prédire les périodes actives, afin d'identifier rapidement les VM LLMU. Mais dans le cas des VM LLMI, elle n'est pas significative. De plus, si la F-mesure est la métrique d'évaluation principale, la précision et le rappel restent importants individuellement. La précision indique l'efficacité du modèle à éviter les faux positifs, c'est-à-dire les prédictions d'inactivité alors que la VM est active. En effet, ces erreurs de prédiction peuvent mener à colocaliser une VM active avec des VM inactives, empêchant ainsi leur serveur d'être suspendu. Autrement dit, une bonne précision du modèle signifie qu'il ne détruit pas des opportunités d'économies. De manière similaire, le rappel caractérise la capacité du modèle à prédire toutes les périodes d'inactivité. Cette métrique indique donc si le modèle est capable de saisir toutes les opportunités d'économies. Les résultats des évaluations du modèle sont donnés en figure 3.11.

Résultats pour les VM LLMI (sous-figures a à g) La première observation est le temps nécessaire au modèle pour gagner en efficacité. En effet, il y a une courte période au début de chaque courbe où les métriques « positives » (rappel, précision et donc F-mesure) augmentent. C'est dû à l'apprentissage de zéro du modèle. Pour les traces de VM les plus simples, cet apprentissage initial est suffisant pour toute leur durée de vie, par exemple pour les traces réelles, sous-figures c à g.

Dans un cas plus complexe, comme pour la sous-figure b, il faut environ deux ans au modèle pour saisir la complexité du motif de périodicité. On observe en effet un changement dans la qualité de prédiction une fois que les mois de vacances de la deuxième année se sont écoulés (voir tableau 3.5 pour la description de cette trace). De plus, toujours sur la sous-figure b, la dégradation de la précision au début de la deuxième année provient d'une nouvelle phase d'apprentissage pour le modèle, qui doit déterminer que le jour de l'année n'a pas d'influence sur le comportement de la VM. On peut en effet observer que le début de la troisième année est plus stable après



(a) Sauvegarde quotidienne

(b) BD en ligne

(c) Trace réelle 1

(d) Trace réelle 2

(e) Trace réelle 3

(f) Trace réelle 4

(g) Trace réelle 5

(h) LLMU

FIGURE 3.11. – Efficacité du modèle d'inactivité. Évaluation selon quatre métriques de la qualité des prédictions du modèle pendant trois ans. Pour toutes les sous-figures sauf la h, la métrique d'intérêt est principalement la F-mesure.

cet apprentissage. Malgré tout, la F-mesure ne descend pas en dessous de 82% à partir de quelques semaines.

Pour revenir au cas simple de la sous-figure a, ainsi qu'aux traces réelles de c à g, le modèle montre de bons résultats de prédiction. La F-mesure atteint plus de 97% après seulement quelques semaines.

Résultats pour les VM LLMU (sous-figure h) Enfin, le modèle a été évalué sur une trace de VM surtout active. L'évaluation rassure sur le fait que le modèle reconnaît très vite la nature de cette VM, puisque la spécificité est très proche de 1 étant donné qu'elle est presque toujours active.

3.9.2. Évaluation par simulation

Pour compléter l'évaluation, on évalue Drowsy-DC par simulation sur de vraies traces de VM à l'aide de CloudSim [24]. Le code source de la simulation basée sur CloudSim est donné en annexe A. Les traces de VM LLMU proviennent du DC cloud privé de Google [114] tandis que les traces LLMI sont celles du cloud privé décrit en section 3.9.1. Celles-ci sont artificiellement étendues pour simuler six mois d'exécution. Quant au DC simulé, il est composé de 14 serveurs et héberge 55 VM. Les tailles, en termes de ressources processeur et mémoire, des serveurs et des VM sont générées aléatoirement parmi les types de serveurs dédiés et de VM disponibles sur Amazon Web Service EC2.

Lors de cette simulation, on compare Drowsy-DC à OpenStack Neat [19], déjà présenté précédemment, ainsi qu'à Oasis [146], décrit en section 3.10. Ces comparaisons se font en faisant varier la proportion de VM LLMI dans le DC, c'est-à-dire que l'on panache les traces LLMU et les traces LLMI selon différentes proportions. Elle va de 10%, pour un DC composé essentiellement de VM de traitement de données, à 90%, pour un DC composé surtout de petits services Internet d'entreprises, par exemple.

Les résultats sur la consommation énergétique sont donnés en figure 3.12.

Tout d'abord, on observe que Neat consomme en fait environ la même quantité d'énergie quelle que soit la fraction de VM LLMI dans le DC. L'exploitation de la suspension des serveurs économise au mieux environ 10 kWh. Ce constat met en exergue tout le potentiel de la consolidation basée sur l'inactivité de Drowsy-DC. En effet, se contenter d'utiliser la suspension des serveurs inactifs avec Neat ne produit pas d'économie énergétiques significatives. C'est parce que Neat ne sait pas exploiter cette possibilité.

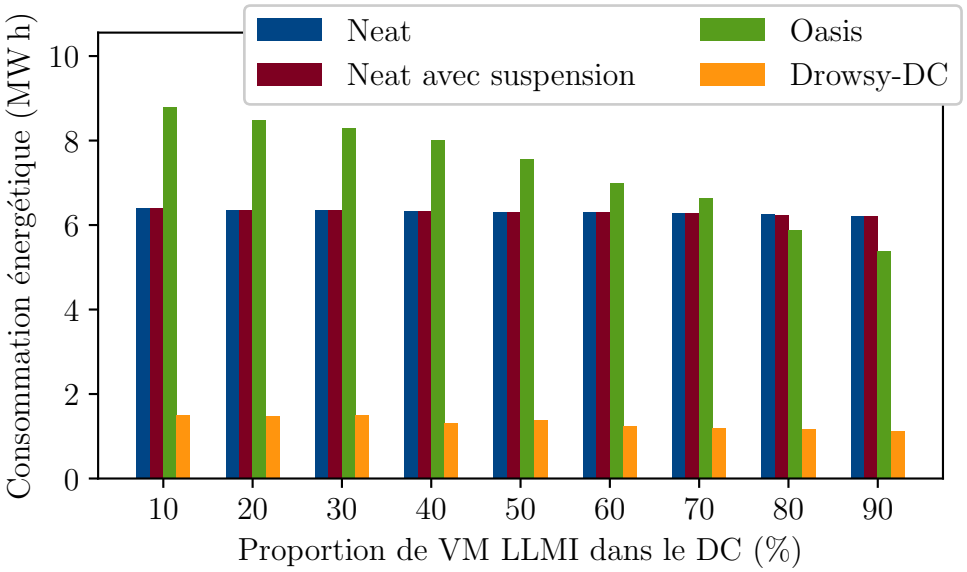


FIGURE 3.12. – Économies d'énergie réalisées par Drowsy-DC, Neat, Neat avec la suspension des serveurs inactifs, et Oasis.

Le système Oasis quant à lui, montre des cas où il consomme en fait plus d'énergie qu'avec Neat. Cela s'explique pour deux raisons :

1. les serveurs de mémoire ajoutés à chaque serveur normal consomment eux-mêmes de l'énergie de manière non négligeable ;
2. Oasis provoque un très grand nombre de migrations de VM (voir figure 3.13), coûteuses en énergie parce qu'il faut que les serveurs soient actifs.

La consommation diminue cependant avec l'augmentation de la proportion de VM LLMI dans le DC. L'énergie supplémentaire dédiée aux serveurs de mémoire est compensée au-delà de 70% de VM LLMI.

Enfin, Drowsy-DC montre des économies d'énergie très importantes par comparaison avec les autres solutions. Celles-ci oscillent entre 76% (pour 30% de VM LLMI) et 81% (pour 70% de VM LLMI) par rapport à Neat. Par comparaison avec Oasis, Drowsy-DC économise environ 81% d'énergie en moyenne.

Il est intéressant de constater que les économies réalisées par Drowsy-DC sont en fait plutôt stables quelle que soit la proportion de VM LLMI dans le DC ; l'écart type relatif à la moyenne sur toutes les proportions est de 2%. Ainsi, le fait que Drowsy-DC semble économiser le plus d'énergie avec 70%

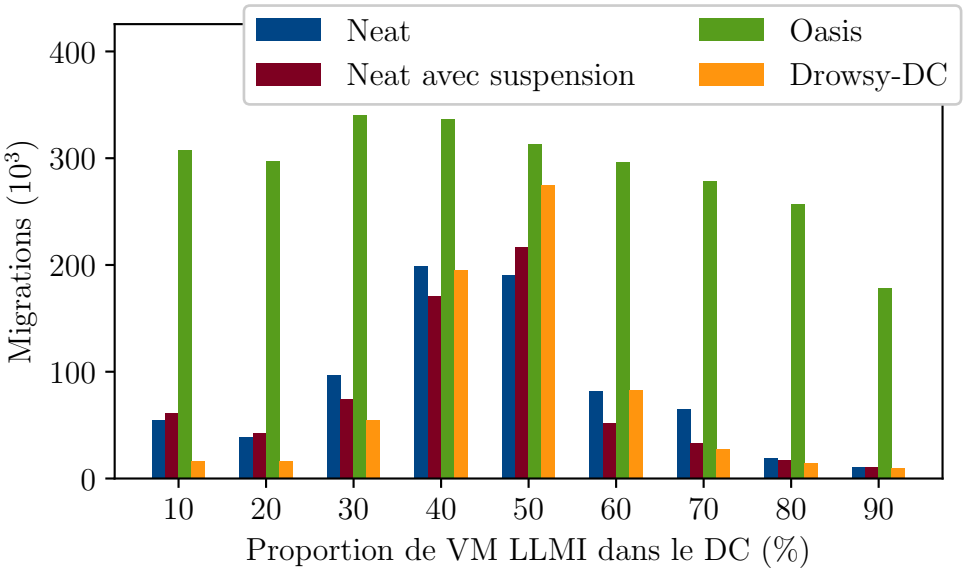


FIGURE 3.13. – Nombre de migrations effectuées par Drowsy-DC, Neat, Neat avec la suspension des serveurs inactifs, et Oasis.

de VM LLMI dans le DC n'est pas impactant. La différence d'économie entre les extrema est en fait inférieure à 0,5 kWh.

L'observation d'économies d'énergie les plus faibles pour 50% de VM LLMI dans le DC correspond, lors de cette simulation, à une explosion du nombre de migrations de VM. La comparaison du nombre de migrations est illustrée en figure 3.13.

De manière générale, Neat avec ou sans suspension des serveurs ne nécessite que peu de migrations. En fait, Drowsy-DC et Neat montrent les mêmes variations du nombre de migrations en fonction de la proportion de VM LLMI dans le DC. Cela vient du fait que Drowsy-DC demeure une modification de l'algorithme de consolidation de Neat. Les trois méthodes montrent donc une augmentation du nombre de migrations avec les proportions 40% et 50%. L'explication la plus probable est un « accident » dans la répartition des VM dû aux mélanges spécifiques de VM LLMI et LLMU. Quoiqu'il en soit, Oasis provoque avec toutes les proportions une grande quantité de migrations. La conséquence est une augmentation de la consommation énergétique, comme expliqué précédemment ; c'est un effet valable pour toutes les solutions.

3.10. État de l'art

Drowsy-DC repose donc sur deux axes : l'optimisation de l'efficacité énergétique des serveurs par l'utilisation d'état de faible énergie, et l'optimisation de la consommation énergétique du DC par la consolidation de la charge de travail. Cette section liste les travaux qui ont précédemment exploré ces voies, et comment Drowsy-DC comble leurs lacunes.

3.10.1. États de faible énergie

L'implémentation des états de faible énergie (suspension et hibernation des serveurs) a émergé tardivement, car il était difficile de concevoir l'idée de rendre indisponible un serveur, qui doit par essence servir sans interruption.

Une première proposition est SleepServer par Savage et Gupta [118], appliquée dans le contexte des ordinateurs d'entreprise qui sont alimentés en continu malgré leur inactivité. SleepServer cherche à mettre en veille ces ordinateurs. Son efficacité est prouvée [113], cependant la solution consiste à assurer les services hébergés sur ces ordinateurs par un serveur mandataire (en anglais *proxy*) qui les réimplémente de manière minimale. Cette solution n'est pas donc pas applicable au cloud.

En ce qui concerne les DC de cloud, Meisner, Gold et Wenisch [82] ont imaginé PowerNap, une modélisation du problème d'exploitation des périodes d'inactivité, parfois très courtes. Elle montre notamment que la durée de transition de l'état de faible énergie à l'état actif doit être inférieure à 1ms pour qu'un système qui l'implémente n'impose pas de dégradation significative. On peut considérer qu'Agile, par Isci et al. [58], implémente cette modélisation sur les premiers serveurs capable d'atteindre les niveaux de faible énergie. Les économies d'énergie sont importantes mais là aussi, la durée de transition entre états est trop longue et impose une dégradation des performances trop importante.

Bien que la vitesse de transition vers l'état actif soit importante aussi pour Drowsy-DC, la contrainte est moins forte parce que le système est plus conservateur grâce à sa modélisation des périodes d'inactivité et à son système de réveil programmé. De plus, inspiré par le travail de Lentz, Litton et Bhattacharjee [71] pour les smartphones, Drowsy-DC utilise l'abstraction de la virtualisation pour réveiller les périphériques matériels strictement nécessaires au service qui provoque le réveil, afin de l'accélérer.

Meisner et Wenisch [83] ont conçu DreamWeaver, qui ordonnance les instructions exécutées par le processeur en lots afin de garder le serveur inactif (et donc dans un état de faible énergie) le plus longtemps possible.

Cet ordonnancement bas niveau qui retarde les instructions le plus longtemps possible requiert un co-processeur, et donc des serveurs développés spécialement avec cette fonctionnalité ; là où Drowsy-DC est une solution de haut niveau basée sur la modélisation du comportement des VM et leur consolidation en conséquence.

3.10.2. Consolidation de la charge de travail

La consolidation est permise par la capacité de migration des VM, c'est pourquoi de nombreux travaux cherchent à optimiser cette opération dans le but d'économiser l'énergie.

Oasis, développé par Zhi, Bila et Lara [146], exploite la migration partielle des VM [22] pour consolider avec une plus grande densité. Ce système s'applique lors de la migration d'une VM inactive : seul son espace mémoire de travail est migré, et ses autres pages mémoire seront migrées à la demande si la VM redevient active et peut rester à sa destination. Comme la migration de la mémoire de la VM est partielle, on obtient une plus grande densité de consolidation, ce qui se traduit par plus de serveurs vides, qui peuvent être placés dans un état de faible énergie. Cependant, le système de la migration partielle requiert des « serveurs de mémoire » pour servir aux VM partiellement migrées, leurs pages mémoire qui n'ont pas été déplacées, c'est-à-dire qui sont restées sur un serveur désormais suspendu. À cette fin, chaque serveur hébergeant des VM se voit attaché un « serveur de mémoire ». Ce dernier consomme moins d'énergie, mais cette consommation n'est à la fin pas négligeable. De plus, si la VM partiellement migrée redevient active et requiert des pages mémoire qui n'avaient pas été migrées, elle peut dépasser la capacité de son serveur, et il faut la ramener sur le serveur d'origine ; ce qui provoque une dégradation des performances.

Zhang et al. [143] ont proposé Picocenter, qui est un système similaire à Oasis mais qui utilise du stockage cloud pour migrer les VM inactives. Le même problème de la dégradation des performances due à la latence de rapatriement est constaté, et ce malgré une approche prédictive qui détermine l'espace mémoire de travail pour accélérer la reprise. On notera cependant que le concept de VM LLMI, central à Drowsy-DC, provient de ce travail. Drowsy-DC utilise un processus de migration classique pour la consolidation, et ne connaît donc pas ces limites.

Enfin, Meng et al. [86] ont imaginé un concept similaire à Drowsy-DC : la consolidation des VM doit considérer les VM en combinaison plutôt qu'individuellement, en croisant les motifs d'inactivité. Il est intéressant de constater que ce travail considère la stratégie opposée à Drowsy-DC : le

but est ici de colocaliser les VM deux par deux, de sorte à faire coïncider les phases d'activité de l'une avec les phases d'inactivité de l'autre. Ainsi, les ressources inutilisées par la VM inactive sont utilisées par l'autre VM, active, et l'on peut augmenter la densité de consolidation, et donc le nombre de serveurs inactifs qui sont placés dans un état de faible énergie.

L'objectif de Drowsy-DC est donc à l'opposé : il s'agit de synchroniser les périodes d'inactivité afin de créer des serveurs inactifs bien qu'hébergeant des VM. Le travail de Meng et al. [86] repose aussi sur un algorithme de prédiction afin de déterminer l'état actif ou inactif de chaque VM. Cependant, Drowsy-DC est beaucoup plus efficace car il n'est pas limité à considérer les VM deux par deux ; et la complexité de son algorithme est en $\mathcal{O}(n)$ (avec n le nombre de VM), tandis que celle de l'autre est en $\mathcal{O}(n^2)$, permettant à Drowsy-DC de gérer un grand nombre de VM LLMI.

3.11. Conclusion

Ainsi, les efforts actuels d'amélioration des performances énergétiques des centres d'hébergement rencontrent des obstacles difficiles à surmonter. Les serveurs ne sont réellement efficaces énergétiquement qu'au maximum de leur charge, ou bien sûr lorsqu'ils sont éteints. Mais le mécanisme de consolidation actuel ne permet pas d'atteindre cette efficacité en plaçant la charge de travail intelligemment. C'est pourquoi Drowsy-DC propose d'utiliser un état de faible énergie, la suspension. Dans cet état, un serveur est efficace énergétiquement. La transition vers cet état peut se faire alors même que le serveur héberge des machines virtuelles, avec un impact minimal sur les performances des applications. Il faut bien entendu que le serveur soit inactif, c'est-à-dire que ses machines virtuelles soient inactives.

Drowsy-DC propose donc en supplément, un nouvel algorithme de consolidation qui suit un nouveau paradigme. Désormais, l'objectf est de colocaliser des machines virtuelles avec des motifs d'inactivité qui correspondent, afin de maximiser les périodes d'inactivité des serveurs. Cela produit ainsi des serveurs efficaces énergétiquement : soit parce qu'ils sont au maximum de leurs charges, soit parce qu'ils sont inactifs et sont donc suspendus. L'intégration de cet algorithme s'accompagne de deux modules logiciels pour supporter la suspension et le réveil optimisés des serveurs.

Ces modules pourraient par ailleurs bénéficier des informations sur la charge de travail fournies par la double virtualisation. De même, la consolidation basée sur l'inactivité serait plus puissante avec l'intégration à l'orchestrateur dans un contexte de multi-virtualisation.

Chapitre 4

L'auto-configuration des applications conteneurisées

4.1. Introduction 77
4.2. Allocation du processeur aux conteneurs 80
4.3. Analyse : violation du principe tel-tel et solutions 82
4.4. Algorithme d'allocation hybride de la ressource processeur 85
4.5. Évaluation 94
4.6. État de l'art 101
4.7. Conclusion 103

4.1. Introduction

L'utilisation du type de cloud IaaS (voir section 2.1.3) et des VM est souvent pénible pour l'utilisateur. En effet, *il doit s'occuper de toutes les fonctionnalités annexes* nécessaires à son service, comme la tolérance aux pannes, le dimensionnement automatique, etc. De plus, le déploiement même de l'application peut être complexe. C'est pour ces raisons que le type PaaS devient le favori, puisque c'est le fournisseur de cloud qui propose ces services, et les conteneurs y sont souvent utilisés.

Le tableau 2.1 vu précédemment indique que la virtualisation niveau système d'exploitation (c'est-à-dire les conteneurs) offre *une mauvaise prédictibilité des performances* [34, 95]. Dans un PaaS basé sur des conteneurs, les utilisateurs qui cherchent à obtenir un minimum de garantie sur la prédictibilité, assignent à chaque conteneur une quantité inflexible de ressources. En effet, l'attribution de ressources à un conteneur est généralement implémentée par deux paramètres :

1. la requête : minimum garanti ;
2. la limite : maximum de ressources permis à ce conteneur.

Dans la recherche de la prédictibilité des performances, la requête et la limite sont définies égales.

Malgré cette astuce, les performances de calcul d'un conteneur restent imprévisibles à cause du *mécanisme d'allocation*. En effet, il existe deux mécanismes pour allouer de la capacité de processeur : les ensembles de processeurs [1] (en anglais *CPU sets*), et le quota d'ordonnanceur (en anglais *scheduler quota*). D'abord, les ensembles de processeurs restreignent l'exécution des instructions d'un conteneur à un ensemble de processeurs donné. Ensuite, le quota d'ordonnanceur n'impose pas de restriction sur les processeurs utilisables, mais agit au niveau de l'algorithme d'ordonnancement. Celui-ci limite le temps processeur qu'il alloue au conteneur afin de garder cette quantité sous le quota défini.

Le mécanisme du quota est le plus facile à utiliser, parce qu'il s'agit simplement de modifier un paramètre de l'algorithme implémenté dans le SE. À l'inverse, utiliser les ensembles de processeurs demande à l'administrateur des conteneurs, c'est-à-dire à l'orchestrateur (voir section 2.2.2), de gérer lui-même ces ensembles pour respecter les requêtes de ressources. C'est la raison pour laquelle en pratique, *les orchestrateurs ne gèrent les ressources qu'avec le quota d'ordonnanceur*, bien que les moteurs de conteneurs soient capable d'utiliser les ensembles de processeurs.

Cependant, l'utilisation du quota dégrade totalement la prédictibilité des performances pour certaines applications. En effet, un grand nombre d'entre elles *s'auto-configurent* [70, 117] en fonction de la quantité de ressources perçue. C'est-à-dire qu'elles observent les ressources de leur système hôte, et se basent sur cette information pour déterminer ensuite leurs propres réglages. Par exemple, le nombre de processus ouvriers (en anglais *worker processes*) dépend du nombre de processeurs.

Il en découle qu'une application qui s'auto-configure, déployée dans un conteneur soumis à une limite de ressource processeur par le mécanisme du quota d'ordonnanceur, obtiendra une vision erronée des ressources disponibles. Il s'agit d'une violation du *principe tel-tel* (en anglais *What You See Is What You Get*, raccourci en WYSIWYG), illustrée en figure 4.1.

1. On utilise le terme « processeur » pour désigner en fait les cœurs d'un processeur vus par le SE hôte, voire les fils d'exécution matériels (en anglais *hardware threads*) provenant de la technologie *Simultaneous Multi Threading* (SMT), c'est-à-dire l'exécution de plusieurs fils sur le même cœur.

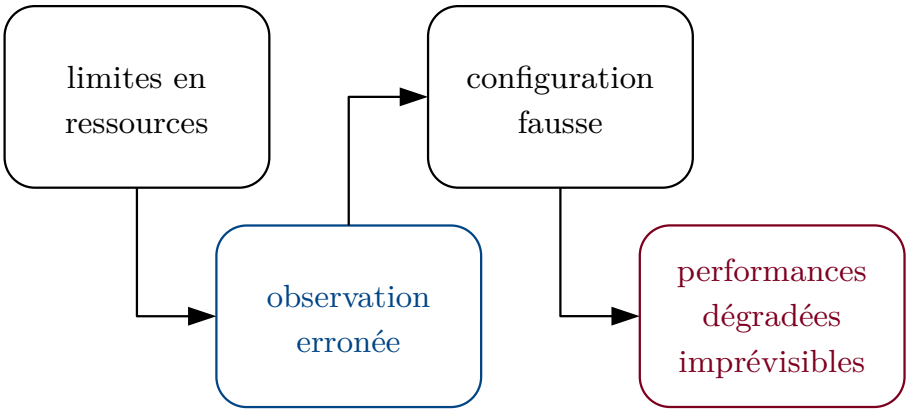


FIGURE 4.1. – Violation du principe tel-tel et conséquences.

Tel-tel. *Le principe tel-tel établit qu'une application conteneurisée qui observe son environnement, doit obtenir une vue fidèle des ressources qui lui sont disponibles.*

On prend l'exemple d'un conteneur limité à 200% d'utilisation processeur, c'est-à-dire qu'il obtient l'équivalent en temps processeur de deux processeurs à chaque période d'ordonnancement. Il est placé sur un serveur possédant 56 processeurs. L'application qu'il virtualise observera ces 56 processeurs, et s'auto-configurera de manière erronée avec 56 processus ouvriers. Cette *mauvaise configuration* a un impact très négatif sur les performances et leur prédictibilité, comme présenté dans les sections suivantes. Il apparaît que l'utilisation des ensembles de processeurs est une bonne solution à ce problème.

Bien que la violation du principe tel-tel affecte d'autres types de ressources, tels que la mémoire [45], le travail présenté dans ce chapitre se concentre sur l'utilisation du processeur et propose une solution conçue spécifiquement pour cette ressource.

Afin de comprendre et délimiter le problème et ses solutions potentielles, une analyse a été menée. Il s'agit de comparer les mécanismes d'allocation processeur (ensembles de processeurs et quota d'ordonnanceur) entre eux, avec des bancs d'essai micro et macro pour les performances ; mais aussi du point de vue de leur utilisation par l'orchestrateur.

La solution qui est proposée, basée sur cette analyse, est un algorithme hybride pour l'orchestrateur, qui lui permet d'utiliser au mieux les deux mécanismes. Il requiert au préalable l'identification du type d'application conteneurisée : est-elle sensible à la violation du principe tel-tel ou non ?

Organisation du chapitre

La section 4.2 décrit les concepts et notions spécifiques à ce chapitre. L'analyse du problème et de ses solutions potentielles est reproduite dans la section 4.3. La section section 4.4 détaille l'algorithme final qui implémente la solution retenue. Cet algorithme est évalué en section 4.5, avant de présenter l'état de l'art du sujet en section 4.6 et de conclure sur ce travail en section 4.7.

4.2. Notion préliminaire : allocation du processeur aux conteneurs

Dans le contexte de la virtualisation du système d'exploitation, deux systèmes logiciels interviennent dans l'allocation de la ressource du processeur aux conteneurs (voir section 2.2.2) :

1. le moteur de conteneurs ;
2. l'orchestrateur.

À des fins d'exemple et d'intégration du travail de ce chapitre, on considère respectivement Docker [42] et Kubernetes [23].

4.2.1. Allocation par le moteur de conteneurs

Comme expliqué en introduction, le moteur de conteneurs peut utiliser deux mécanismes d'allocation : le quota d'ordonnanceur, et les ensembles de processeurs.

Dans le cas du premier mécanisme, c'est *l'ordonnanceur du SE hôte* qui agit : à chaque période d'ordonnancement, il attribue du temps processeur en fonction du quota défini par le moteur sur le conteneur. On notera que dans le cas le plus courant du noyau Linux comme SE hôte, on parle de quota CFS, pour *Completely Fair Scheduler* [103], c'est-à-dire « ordonnanceur totalement équitable » qui est l'ordonnanceur de Linux. Avec le mécanisme du quota, le conteneur voit tous les processeurs du serveur hôte, ce qui provoque une violation du principe tel-tel (voir section 4.1).

Pour ce qui est du second mécanisme, les ensembles de processeurs, il s'agit de *limiter l'exécution du conteneur à un ensemble concret de processeurs*. L'utilité première est généralement de fixer un processus sur des cœurs de processeur précis, par exemple sur un nœud NUMA (pour *Non Uniform Memory Access*, c'est-à-dire « à accès mémoire non uniforme »). Mais dans le contexte du présent chapitre, cela a pour effet de cacher au conteneur

TABLE 4.1. – Comparaison des mécanismes d'allocation de processeur.

	quota d'ordonnanceur	ensemble de processeurs
Utilisation	facile	difficile
Flexibilité	bonne	mauvaise
Tel-tel	non	oui

les processeurs qui ne lui sont pas alloués. Autrement dit, il ne voit que *son allocation de ressource processeur réelle*, ce qui permet de respecter le principe tel-tel.

Les descriptions des deux mécanismes laissent entrevoir *des différences d'utilisabilité*. En effet, le quota d'ordonnanceur est bien plus flexible puisqu'il permet une allocation très fine. Il est capable d'allouer une portion de capacité processeur, avec finesse d'allocation allant jusqu'à 1% du processeur. De plus, son utilisation est bien plus simple pour les orchestrateurs parce que le véritable travail d'allocation des ressources revient à l'ordonnanceur. Une comparaison des mécanismes d'allocation par quota d'ordonnanceur et par ensemble de processeurs est donnée en tableau 4.1.

Pour finir, on peut citer le mécanisme des parts de processeur (en anglais *CPU shares*). Elles représentent une allocation minimale des ressources processeur du serveur à un conteneur. Cependant, toutes les parts sont relatives entre elles ; et ce mécanisme est imprévisible parce qu'il fixe un minimum. Ainsi, un conteneur peut utiliser plus de ressource processeur si le serveur en a la capacité. Les parts de processeur ne sont donc pas considérées pour atteindre l'objectif de prévisibilité des performances.

4.2.2. Allocation par l'orchestrateur

L'orchestrateur *contrôle le moteur de conteneurs*, et utilise donc les mécanismes d'allocation de ce dernier, présentés ci-dessus. Pour ce qui est de la configuration, il propose généralement de spécifier l'allocation en ressources d'un conteneur en termes de requête et de limite. Ces deux réglages sont en pratique égaux lorsque la prédictibilité est recherchée, comme expliqué en introduction. On ignore donc ici le réglage de la requête, d'autant plus qu'elle est implémentée par les parts de processeur décrites plus haut, et seule la limite est considérée.

Ainsi, pour implémenter la limite en ressource processeur d'un conteneur, *les orchestrateurs utilisent le quota d'ordonnanceur*. Les raisons sont effectivement sa simplicité d'utilisation et sa flexibilité d'allocation.

4.3. Analyse : violation du principe tel-tel et solutions

Cette section démontre par l'évaluation pourquoi la solution actuelle, qui utilise seulement le quota d'ordonnanceur pour allouer du processeur aux conteneurs, est néfaste pour la prédictibilité des performances.

4.3.1. Installation

Les deux mécanismes d'allocation de processeur (ensembles et quota) sont comparés sur différents bancs d'essai : d'abord Stream [81], et la suite PARSEC [21] ; puis l'application d'analyse en mémoire de CloudSuite [46, 104], qui utilise Apache Spark [141] pour calculer des recommandations de films. Le serveur d'évaluation est une machine Dell avec 12 cœurs de processeur Intel® Xeon® E5–2420 v2 non SMT et NUMA. Cependant, afin d'éviter les effets de l'architecture NUMA, les applications d'évaluation ne s'exécutent que sur le même nœud NUMA, donc sur 6 cœurs. Le système d'exploitation est la distribution Arch Linux, avec un noyau Linux 4.15.15. Le moteur de conteneurs est Docker en version 18.01-CE.

4.3.2. Méthodologie

L'objectif de ces expériences est de *tester la capacité de limitation de l'allocation processeur* à un conteneur. Cette limite est fixée à 350% pour que l'allocation :

— inclue plus de deux processeurs, pour les traitements parallèles ;
— soit plus petite que le maximum de ressource processeur disponible, afin de laisser de la marge ;
— ne soit pas un nombre de processeurs exact, afin de mettre en lumière les avantages et inconvénients des deux mécanismes.

Par ailleurs, le conteneur est observé dans trois configurations différentes :

`quota` le conteneur a accès aux 6 processeurs du nœud NUMA, mais le mécanisme du quota d'ordonnanceur est utilisé pour limiter le temps processeur alloué à 350% ;

`quota+config.` comme `quota`, cependant l'application conteneurisée est configurée manuellement pour n'utiliser que 4 fils d'exécution au maximum, ce qui est l'entier supérieur le plus proche de l'allocation ;

`ensemble` un ensemble de processeurs est utilisé pour limiter le conteneur à 4 processeurs du nœud NUMA, et on applique aussi un quota d'ordonnanceur afin d'affiner l'allocation à 350%, le tout sans configuration manuelle.

Ainsi, `quota` représente la situation actuelle ; `quota+config.` représente une configuration effectuée par celui qui déploie l'application, avec une connaissance fine de l'application ; enfin, `ensemble` représente une utilisation des ensembles de processeurs pour limiter l'allocation.

4.3.3. Résultats

La figure 4.2 donne les résultats de l'évaluation avec le banc d'essai Stream. Elle montre la distribution des durées d'exécution sur les 100 essais. On observe sous `quota` *deux niveaux de performance* distincts, visibles dans les deux renflements. Cette observation ne se retrouve pas sous `ensemble` : la distribution est rassemblée autour du même niveau de performance. Cette imprévisibilité sous `quota` provient bien de la fonctionnalité d'auto-configuration du banc d'essai Stream. Celui-ci décide du nombre de fils d'exécution à créer en fonction du nombre de processeurs. Ce nombre est le total des cœurs du nœud NUMA sous `quota`, mais l'application n'a pas connaissance de la limite de 350% appliquée par un quota d'ordonnanceur. Cette différence, qui viole le principe tel-tel, a un impact très néfaste sur une application aussi dépendante du matériel, comme expliqué plus bas en section 4.3.4.

On confirme cette analyse du problème avec les résultats de la configuration `quota+config`. En effet, cette configuration produit bien des résultats prévisibles, reproduits sur le graphe central de la figure 4.2. Ils sont par ailleurs similaires à ceux de la configuration `ensemble`, dans le dernier graphe. Cela suggère que *le mécanisme des ensembles de processeurs est une bonne solution* au problème de l'imprévisibilité des performances. De plus, l'utilisation seule du quota dégrade les performances de l'application. Stream est 31,8% plus performant sous `ensemble`.

Les résultats de l'autre banc d'essai, l'application Spark d'analyse en mémoire de CloudSuite, sont rapportés en figure 4.3. La limite processeur est placée sur le conteneur ouvrier Spark. Dans le cas de cette application réelle, on observe des résultats similaires : la configuration `ensemble` est plus performante que `quota` de 68%, et montre une meilleure prédictibilité.

Pour finir, la figure 4.4 montre les résultats des bancs d'essai de la suite PARSEC. Les gains de performance relatifs de `ensemble` par rapport à

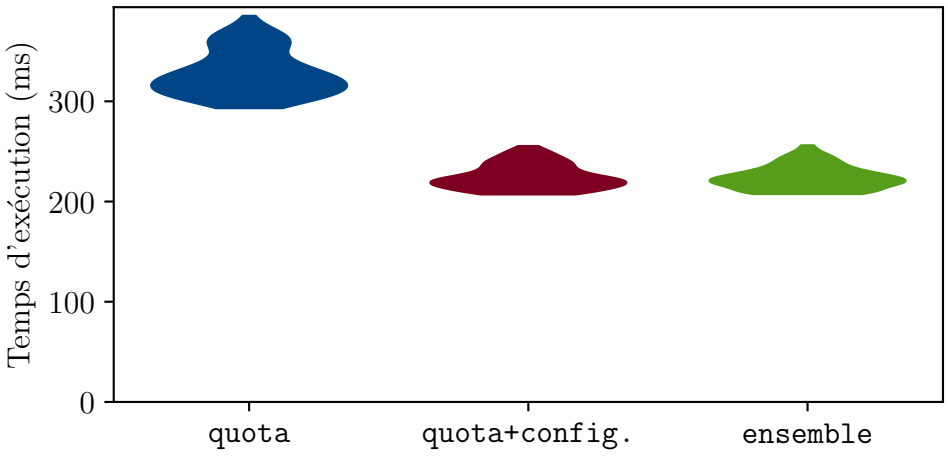


FIGURE 4.2. – Durées d'exécution de Stream sur 100 mesures.

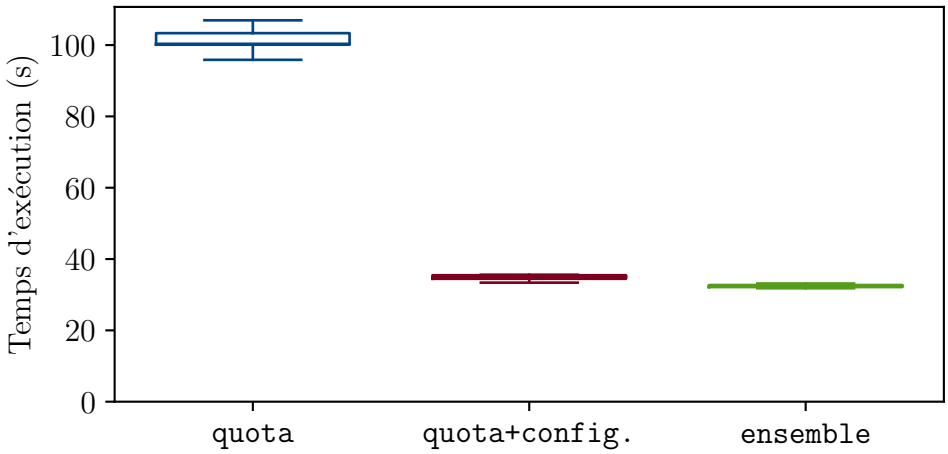


FIGURE 4.3. – Durées d'exécution du banc d'essai d'analyse en mémoire.

`quota` vont de 5,5% pour `ferret`, à 27,6% pour `dedup`. C'est cependant pour `streamcluster` que l'impact absolu de la violation du principe tel-tel est le plus grand : environ 60 s (soit 22,5% de gains pour `ensemble` par rapport à `quota`). Il est intéressant en effet, de constater que pour certaines applications la violation du principe tel-tel n'entraîne pas une dégradation des performances ou de leur prédictibilité significative.

4.3.4. Synthèse

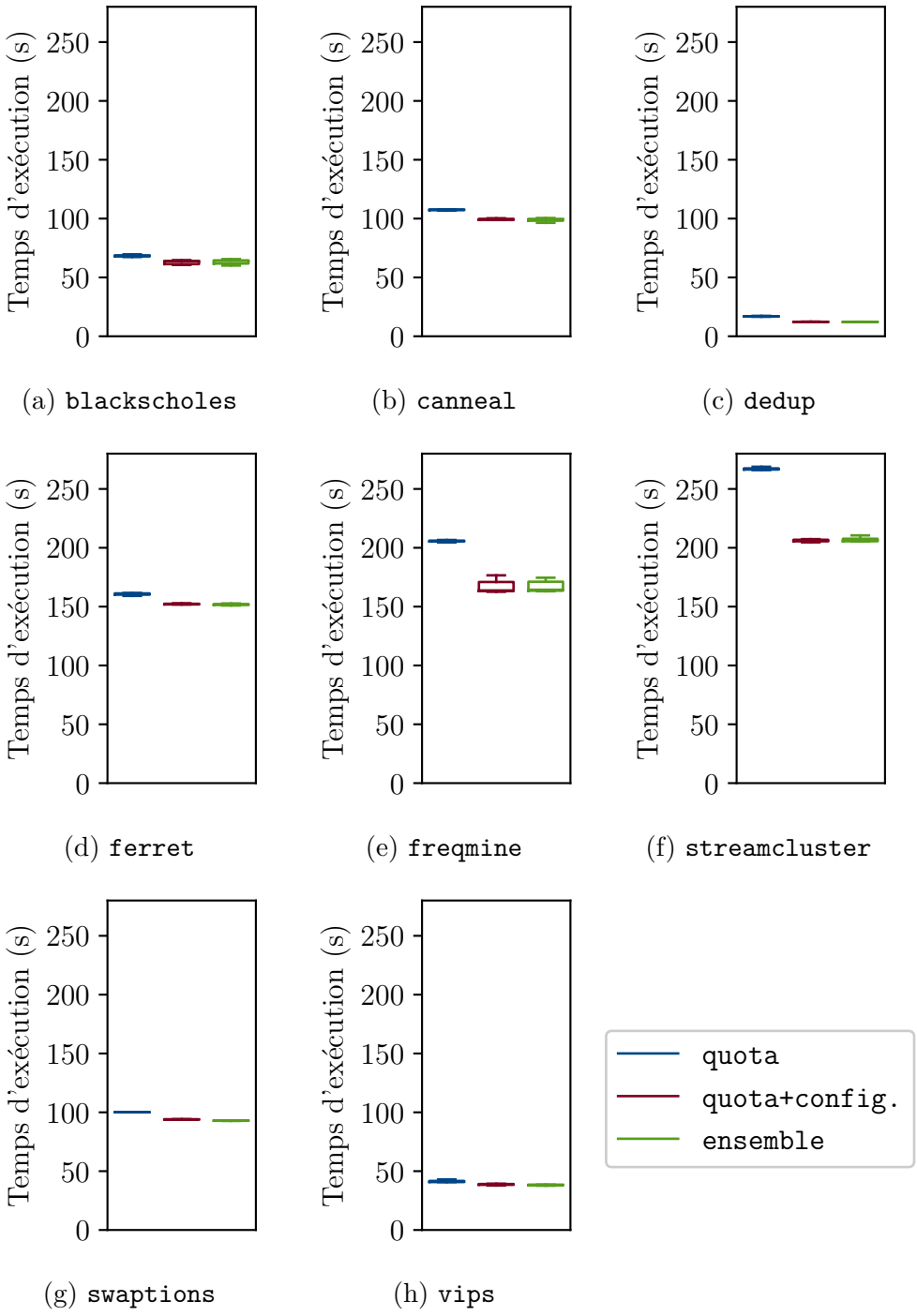
Ces expériences démontrent qu'une application conteneurisée qui s'auto-configure prend des décisions erronées à cause de la *différence entre son allocation en ressources et l'observation de son environnement*. La conséquence est la sur-création de fils d'exécution, caractérisée par un surcoût à la gestion de ces fils, une pollution des caches, etc. [44, 50, 59, 69, 111].

Cependant, on remarque aussi que *le mécanisme d'allocation n'a pas d'impact sur certaines applications*. Par ailleurs, les orchestrateurs déploient souvent des conteneurs pour des tâches secondaires, comme la journalisation, pour lesquels la dégradation des performances n'est pas un problème. Dans ces deux cas, il est préférable d'utiliser le mécanisme du quota d'ordonnanceur, pour sa meilleure utilisation des ressources. En effet, les ensembles de processeurs ont par nature une granularité grossière (un processeur entier), et ne permettent pas par eux-mêmes d'appliquer des limites fractionnelles fines. De plus, allouer du processeur par le biais d'ensembles revient à gérer les ressources en se plaçant comme un adversaire de l'ordonnanceur du SE hôte. La raison est que ce dernier perd en liberté d'ordonnancement puisque chaque conteneur devra être ordonnancé sur l'ensemble de processeurs donné, plutôt que n'importe où sur le système.

Ainsi, le travail développé dans ce chapitre consiste à utiliser conjointement les deux mécanismes d'allocation, en un *algorithme hybride*. L'objectif est d'empêcher les violations du principe tel-tel à l'aide des ensembles de processeurs lorsque nécessaire ; tout en utilisant les quotas d'ordonnanceur lorsque c'est possible, et pour permettre les allocations de processeur à une granularité plus fine.

4.4. Algorithme d'allocation hybride de la ressource processeur

Ainsi, l'analyse précédente mène à la considération d'un algorithme hybride d'allocation de la ressource processeur. Celui-ci doit exploiter les



(a) `blackscholes` (b) `canneal` (c) `dedup`

(d) `ferret` (e) `freqmine` (f) `streamcluster`

(g) `swaptions` (h) `vips`

FIGURE 4.4. – Durées d'exécution des bancs d'essai de la suite PARSEC.

caractéristiques des deux mécanismes d'allocation : le quota d'ordonnanceur et les ensembles de processeurs.

4.4.1. Conception

Le problème a été identifié comme étant *le mensonge du conteneur à l'application qu'il héberge*. Il lui indique un nombre de processeurs erroné. On peut donc envisager que la solution la plus directe est de corriger ce comportement. C'est tout à fait possible et souhaitable, mais cela requiert de modifier le noyau du SE hôte. En effet, la conteneurisation repose directement sur des fonctionnalités permettant l'isolation et la limitation (voir section 2.2.2). L'allocation de ressource concerne la limitation, et elle est implémentée par le noyau Linux avec les groupes de contrôle (en anglais *control groups*, raccourcis en cgroups). Il s'agirait donc de corriger les sources d'information utilisées par toutes les applications, pour qu'elles renvoient la bonne quantité de ressource en fonction de leurs cgroups. Mais une telle solution nécessitant de corriger les noyaux hôtes est difficile à appliquer dans un environnement cloud.

Une autre solution moins contraignante est de construire l'application conteneurisée *avec une bibliothèque d'auto-configuration*. Celle-ci serait capable d'identifier la quantité de ressource réelle à partir du cgroup du conteneur. Cependant, ce n'est pas non plus une bonne solution parce qu'elle va à l'encontre de la philosophie de la conteneurisation. Cette dernière stipule que l'application native et l'application conteneurisée sont exactement les mêmes ; autrement dit, que la conteneurisation n'intervient pas dans la construction de l'application et n'a pas d'influence sur elle. Inclure une bibliothèque spécifiquement pour l'application conteneurisée reviendrait à en créer une version spéciale.

Enfin, l'analyse de la section 4.3 montre que *spécifier manuellement la configuration* de l'application corrige le problème. Malheureusement, cela revient à inclure une configuration statique dans le conteneur de l'application, qui devrait être réalisée pour chaque variation de l'allocation en ressources. Cette méthode annule la généricité de l'image de conteneur et ses avantages. Ainsi, la configuration manuelle n'est pas une solution.

C'est pour ces trois raisons que la réponse proposée aux violations du principe tel-tel, consiste en un *allocateur de ressources pour les conteneurs, au niveau de l'orchestrateur*. Elle ne nécessite que la modification de la couche de gestion la plus haute, qui est souvent un intergiciel développé par le fournisseur de cloud lui-même, ce qui en facilite l'intégration. En fait, la solution décrite ci-dessous corrige le problème de la violation du principe

tel-tel, tout en étant totalement indépendante de l'application conteneurisée et facilement intégrable.

4.4.2. Algorithme

On décrit dans cette section l'algorithme d'allocation hybride. Il utilise les deux mécanismes d'allocation de ressource processeur : le quota d'ordonnanceur et les ensembles de processeurs.

Tout d'abord, selon l'analyse de la section 4.3, il faut distinguer *deux catégories de conteneurs* :

1. $\mathcal{C}_{\text{tel}}$ correspond aux conteneurs sensibles au principe tel-tel ;
2. $\mathcal{C}_{\overline{\text{tel}}}$ correspond aux autres conteneurs.

La méthode de détermination de la catégorie d'un conteneur est expliquée en section 4.4.3

Vue d'ensemble

Le principe est de gérer les processeurs d'un serveur en *trois groupes*, illustrés en figure 4.5 :

1. $\mathcal{P}_{\text{excl}}$: processeurs alloués entiers et exclusivement à un conteneur $\mathcal{C}_{\text{tel}}$, chacun d'eux est inclus dans exactement un ensemble de processeurs d'un seul conteneur ;
2. $\mathcal{P}_{\text{frac}}$: processeurs utilisés pour allouer une fraction de cœur à au moins un conteneur $\mathcal{C}_{\text{tel}}$, ils peuvent être inclus simultanément dans plusieurs ensembles de processeurs ;
3. $\mathcal{P}_{\text{part}}$: processeurs partagés entre tous les conteneurs $\mathcal{C}_{\overline{\text{tel}}}$.

Voici comment ces trois groupes sont utilisés. $\mathcal{P}_{\text{part}}$ est vu comme une réserve de processeurs disponibles. Le but de l'algorithme est d'allouer à un conteneur $\mathcal{C}_{\text{tel}}$ son propre ensemble de processeurs exclusifs. Cet ensemble doit inclure le nombre correct de processeurs pour respecter le principe tel-tel. Ces processeurs sont choisis dans $\mathcal{P}_{\text{part}}$, et sont donc déplacés dans $\mathcal{P}_{\text{excl}}$. Par exemple, si un conteneur $\mathcal{C}_{\text{tel}}$ fait la requête de 300% de processeur, trois processeurs de $\mathcal{P}_{\text{part}}$ sont déplacés vers $\mathcal{P}_{\text{excl}}$ et alloués dans un ensemble de processeurs à ce conteneur.

De plus, si la requête n'est pas un nombre entier de cœurs, c'est un processeur de $\mathcal{P}_{\text{frac}}$ qui est utilisé pour fournir la partie fractionnelle, en étant ajouté à l'ensemble de processeurs du conteneur qui contient déjà les processeurs exclusifs. S'il n'y a pas de processeur capable de fournir

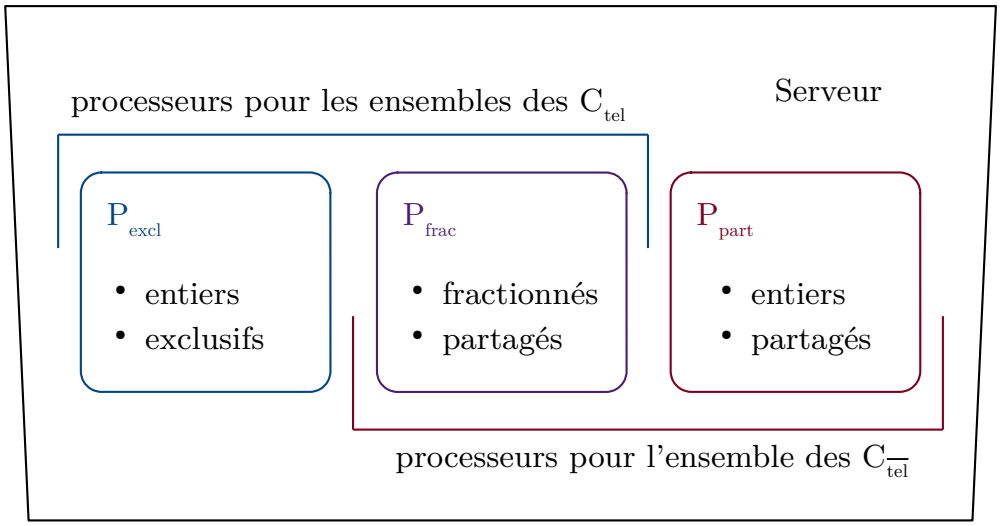


FIGURE 4.5. – Groupes de processeurs pour l'allocation hybride.

cette requête dans $\mathcal{P}_{\text{frac}}$, c'est un processeur de $\mathcal{P}_{\text{part}}$ qui est utilisé (et il est déplacé dans $\mathcal{P}_{\text{frac}}$). Par-dessus cette allocation par un ensemble de processeurs, un quota d'ordonnanceur est appliqué au conteneur pour partager son cœur fractionnel de $\mathcal{P}_{\text{frac}}$.

La gestion des conteneurs $\mathcal{C}_{\overline{\text{tel}}}$ est bien plus simple. Ils partagent tous le même ensemble de processeurs. Celui-ci contient tous les processeurs de $\mathcal{P}_{\text{frac}}$ et de $\mathcal{P}_{\text{part}}$ réunis. Ce sont leurs quotas d'ordonnanceur respectifs qui arbitrent le partage.

Pour résumer, les conteneurs $\mathcal{C}_{\text{tel}}$ se voient alloués des processeurs entiers de $\mathcal{P}_{\text{excl}}$, et peuvent avoir plusieurs cœurs fractionnels de $\mathcal{P}_{\text{frac}}$. Quant aux conteneurs $\mathcal{C}_{\overline{\text{tel}}}$, l'algorithme leur alloue les processeurs de $\mathcal{P}_{\text{frac}}$ et de $\mathcal{P}_{\text{part}}$.

Un exemple du processus d'allocation est illustré en figure 4.6. Il montre comment les allocations successives de deux conteneurs $\mathcal{C}_{\text{tel}}$ et des conteneurs $\mathcal{C}_{\overline{\text{tel}}}$ sont servies en utilisant les trois groupes de processeurs décrits ci-dessus.

Implémentation

Une implémentation en pseudo-code est donnée par l'algorithme 1. Elle montre la configuration d'un conteneur c avec une requête d'allocation processeur r, exprimée en milli-processeur (mCPU) comme Kubernetes le fait. Son but est d'allouer un ensemble de processeurs minimal à chaque $\mathcal{C}_{\text{tel}}$ tout en évitant de partager des processeurs entre plusieurs conteneurs $\mathcal{C}_{\text{tel}}$.

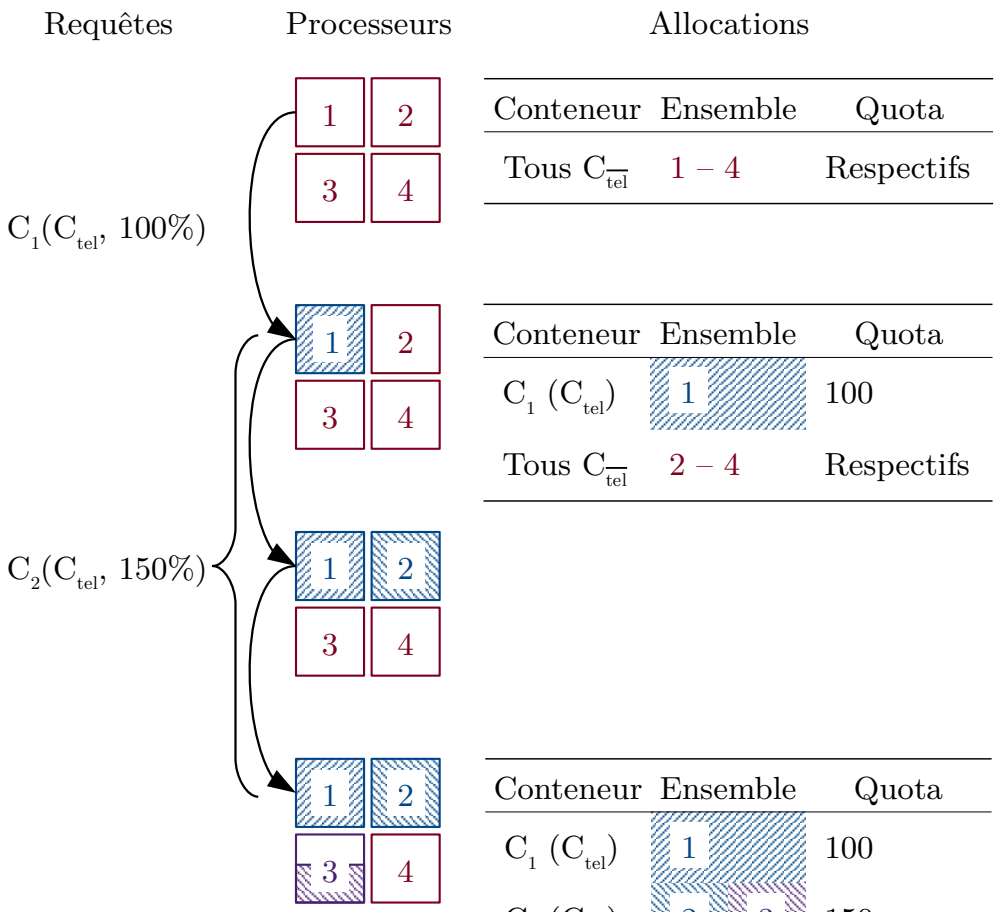


FIGURE 4.6. – Exemple du processus d'allocation hybride utilisant les trois groupes de processeurs : $\mathcal{P}_{\text{excl}}$ en bleu, $\mathcal{P}_{\text{frac}}$ en violet et $\mathcal{P}_{\text{part}}$ en rouge. Les motifs hachurés représentent les ensembles de processeurs pour les deux différents conteneurs $\mathcal{C}_{\text{tel}}$.

Algorithme 1 Allocation hybride de ressource processeur.

Précondition : l'hôte a assez de ressource processeur au total pour allouer r à c

procédure ALLOUER(c, r)
  si hôte vide **alors** ▷ initialisation
    $\mathcal{P}_{\text{excl}} \leftarrow \{\}$
    $\mathcal{P}_{\text{frac}} \leftarrow \{\}$
    $\mathcal{P}_{\text{part}} \leftarrow \{\text{tous les processeurs}\}$
  fin si
  si c est de type $\mathcal{C}_{\overline{\text{tel}}}$ **alors**
    APPLIQUER_ENSEMBLE(c, $\mathcal{P}_{\text{frac}} \cup \mathcal{P}_{\text{part}}$)
  sinon
    $entier \leftarrow \lfloor r/1000 \rfloor$
    $frac \leftarrow r \bmod 1000$
    $p^c_{\text{excl}} \leftarrow \{\min(entier, \#\mathcal{P}_{\text{part}})$ processeurs de $\mathcal{P}_{\text{part}}\}$
    $\mathcal{P}_{\text{part}} \leftarrow \mathcal{P}_{\text{part}} \setminus p^c_{\text{excl}}$
    $\mathcal{P}_{\text{excl}} \leftarrow \mathcal{P}_{\text{excl}} \cup p^c_{\text{excl}}$
    $p^c_{\text{frac}} \leftarrow \{\}$
    si $\#p^c_{\text{excl}} \neq entier$ **alors** ▷ il manque des cœurs entiers
      si $frac = 0$ **alors**
        abandon : impossible d'assurer tel-tel
      sinon ▷ tel-tel toujours possible (p.ex. 150% sur deux cœurs)
        $manque \leftarrow (entier - \#p^c_{\text{excl}}) \times 1000 + frac$
        $p^c_{\text{frac}} \leftarrow$ CHOISIR_PROCS($\mathcal{P}_{\text{frac}}$, $\lceil manque/1000 \rceil$, $manque$)
        si $p^c_{\text{frac}} = \{\}$ **alors**
          abandon : impossible d'assurer tel-tel
        fin si
      fin si
    sinon si $frac \neq 0$ **alors** ▷ cœurs entiers alloués, allocation de la fraction
      $p^c_{\text{frac}} \leftarrow$ CHOISIR_PROCS($\mathcal{P}_{\text{frac}}$, 1, $frac$)
      si $p^c_{\text{frac}} = \{\}$ **alors**
        si $\#\mathcal{P}_{\text{part}} = 0$ **alors**
          abandon : impossible d'assurer tel-tel
        sinon
          $p^c_{\text{frac}} \leftarrow \{1$ processeur de $\mathcal{P}_{\text{part}}\}$
          $\mathcal{P}_{\text{part}} \leftarrow \mathcal{P}_{\text{part}} \setminus p^c_{\text{frac}}$
          $\mathcal{P}_{\text{frac}} \leftarrow \mathcal{P}_{\text{frac}} \cup p^c_{\text{frac}}$
        fin si
      fin si
    fin si
    APPLIQUER_ENSEMBLE(c, $p^c_{\text{excl}} \cup p^c_{\text{frac}}$)
  fin si
  APPLIQUER_QUOTA(c, r) ▷ application du quota dans tous les cas
fin procédure
fonction CHOISIR_PROCS(s, n, r)
  choisir au plus n processeurs dans s pour allouer r, retourne {} si impossible
fin fonction

Relaxe du principe tel-tel

L'algorithme 1 garantit le principe tel-tel pour tous les conteneurs $\mathcal{C}_{\text{tel}}$. Si c'est impossible à cause de la fragmentation des ressources processeur, il abandonne l'allocation. Mais on peut aussi considérer la relaxe de ce principe, c'est-à-dire *accepter une allocation de conteneurs $\mathcal{C}_{tel}$ qui viole le principe tel-tel*, si le serveur a quand même suffisamment de ressources. Concrètement, au lieu d'abandonner l'allocation aux trois endroits où il le fait, l'algorithme alloue la ressource processeur manquante à partir de $\mathcal{P}_{\text{frac}}$ en supprimant la limite n lors des appels de CHOISIR_PROCS. On obtient ainsi une allocation imparfaite du point de vue du principe tel-tel, mais qui améliore quand même les performances et leur prédictibilité.

4.4.3. Détermination de la catégorie de conteneur

La base de cet algorithme est donc l'identification d'un conteneur comme étant de type $\mathcal{C}_{\text{tel}}$ ou $\mathcal{C}_{\overline{\text{tel}}}$. Cette détermination peut s'intégrer aux *processus d'intégration continue* (en anglais *Continuous Integration*, CI) [54]. Ils comportent une étape de calibration des performances. Elle consiste généralement à déterminer la quantité de ressources demandées par l'application, pour obtenir les performances désirées en fonction de la charge de travail. C'est-à-dire que l'application est évaluée sous les charges de travail w_i et avec différentes allocations de ressources r_j.

L'intégration de la détermination de la catégorie de conteneur consiste à exécuter deux sous-évaluations pour chaque évaluation (w_i, r_j). La première applique la limite r_j avec un quota d'ordonnanceur, et la seconde avec un ensemble de processeurs. Ainsi, le conteneur est de type $\mathcal{C}_{\text{tel}}$, et donc requiert le respect du principe tel-tel, si ses performances avec l'ensemble de processeurs sont meilleures qu'avec le quota d'ordonnanceur dans la majorité des évaluations réalisées.

L'efficacité de cette méthode est montrée par l'analyse de la section 4.3. Elle montre en effet une différence significative de performance entre les deux mécanismes d'allocation. Cette intégration dépend bien sûr aussi de la représentativité des charges de travail, qui est un problème de la CI.

4.4.4. Intégration à Kubernetes

À titre d'exemple, on décrit ici une possible intégration de cette solution à l'orchestrateur de conteneurs Kubernetes. Lorsqu'il ordonnance des conteneurs, celui-ci détermine d'abord sur quel serveur les déployer, et va

ensuite les y placer et leur allouer les ressources. Par défaut, Kubernetes utilise le quota d'ordonnanceur pour allouer la ressource processeur.

Le premier niveau d'intégration à Kubernetes est donc de *remplacer son système d'allocation niveau serveur*, basé sur le quota, par l'algorithme hybride décrit plus haut en section 4.4.2. Le second niveau consiste à modifier et à *optimiser l'orchestrateur global*, et plus précisément la logique de décision du serveur où les conteneurs sont déployés.

L'objectif de cette optimisation niveau orchestrateur est d'aider à déployer les conteneurs $\mathcal{C}_{\text{tel}}$ sur *des machines qui pourront leur garantir le principe tel-tel*. Pour aller plus loin, cette optimisation cherche en fait à :

— maximiser le nombre de processeurs $\mathcal{P}_{\text{excl}}$ alloués aux conteneurs $\mathcal{C}_{\text{tel}}$;
— minimiser le temps processeur partagé entre les cœurs de $\mathcal{P}_{\text{frac}}$ alloués aux conteneurs $\mathcal{C}_{\text{tel}}$.

Bien entendu, il faut continuer à prendre en compte les critères de placement classiques, comme la disponibilité en ressources et d'autres critères d'affinité spécifiques à Kubernetes.

L'optimisation niveau orchestrateur est intégrée comme suit. Au déploiement de conteneurs, Kubernetes détermine une liste des serveurs pouvant les accueillir, selon ses critères classiques. Ensuite, il *simule l'allocation* en suivant l'algorithme 1 sur chaque serveur. Il peut ainsi sélectionner celui qui maximise les allocations avec des processeurs $\mathcal{P}_{\text{excl}}$, et minimise le temps processeur partagé des cœurs $\mathcal{P}_{\text{frac}}$. Lorsque le serveur de destination est trouvé, il suffit d'appliquer l'allocation ainsi précalculée.

4.4.5. Limites de l'algorithme et pistes d'amélioration

En l'état, l'algorithme d'allocation hybride est un prototype. Il permet néanmoins de se convaincre qu'une telle solution d'allocation existe (voir section 4.5), et permet de résoudre les problématiques soulevées en section 4.3. On donne ici des pistes d'amélioration.

Tout d'abord, *la gestion des allocations fractionnelles sur les processeurs dans $\mathcal{P}_{frac}$ est un problème difficile*. C'est un problème de *bin packing* (problème de rangement d'objets en un nombre optimisé de boîtes, déjà abordé en section 3.1), très souvent rencontré dans le domaine du cloud dès lors qu'il s'agit de gérer des ressources virtuelles dans le but d'héberger autant d'applications que possible [20]. Cependant, l'objectif classique est de consolider la charge de travail pour utiliser aussi peu de ressources physiques que possible et en maximiser l'utilisation. Ici, le but est inverse : on cherche à éviter la colocation des allocations fractionnelles, en les répartissant sur les processeurs dans $\mathcal{P}_{\text{frac}}$. En effet, et c'est illustré en section 4.5 plus loin,

deux conteneurs $\mathcal{C}_{\text{tel}}$ qui partagent un processeur ne verront pas autant de bénéfices sur leurs performances qu'attendu en respectant le principe tel-tel. Quoiqu'il en soit, l'implémentation actuelle utilise une stratégie de type « première correspondance » (en anglais *first fit*), légèrement modifiée pour préférer les processeurs de $\mathcal{P}_{\text{frac}}$ les plus vides.

Une autre politique pour assurer une allocation fractionnelle est d'utiliser un processeur de $\mathcal{P}_{\text{part}}$ plutôt que de chercher un processeur dans $\mathcal{P}_{\text{frac}}$. On garantit ainsi la séparation des fractions, mais cela consomme très vite les processeurs dans $\mathcal{P}_{\text{part}}$. Ainsi, quand un conteneur $\mathcal{C}_{\text{tel}}$ doit être alloué, et s'il manque des processeurs entiers dans $\mathcal{P}_{\text{part}}$, il faut revoir les allocations fractionnelles afin de libérer un processeur de $\mathcal{P}_{\text{part}}$ et l'utiliser dans $\mathcal{P}_{\text{excl}}$. Autrement dit, des allocations dynamiques peuvent être envisagées pour éviter encore plus la colocation des allocations fractionnelles.

De plus, une contrainte importante lors de l'allocation de la ressource processeur est l'architecture NUMA (voir section 4.2.1). Elle implique que tous les cœurs de processeur ne sont pas égaux du point de vue d'une application. Le placement de ses fils d'exécution est donc primordial pour ses performances. L'algorithme présenté plus haut en section 4.4.2 n'intègre pas cette contrainte.

Enfin, dans le contexte de la multi-virtualisation, les ensembles de processeurs incluent en fait des vCPU (voir section 2.2.1). C'est-à-dire que l'algorithme attribue des processeurs qui ne correspondent pas nécessairement à des processeurs physiques. Les effets particulièrement imprévisibles de ce double ordonnancement forment un problème connu de la virtualisation du matériel [63, 130, 133], que l'algorithme ne prend actuellement pas en compte.

4.5. Évaluation

Le travail présenté dans ce chapitre se compose de trois piliers :

1. l'efficacité des ensembles de processeurs à assurer le principe tel-tel ;
2. la détermination de la catégorie ($\mathcal{C}_{\text{tel}}$ ou $\mathcal{C}_{\overline{\text{tel}}}$) d'un conteneur ;
3. l'algorithme d'allocation hybride.

L'efficacité du premier est prouvée dans la section 4.3 : sous un ensemble de processeurs, les applications sensibles à la violation du principe tel-tel retrouvent leurs performances nominales. Les expériences de cette même section montrent aussi que la détermination de la catégorie d'un conteneur en fonction de ses évaluations de performance est possible (voir section 4.4.3).

En fin de compte, la présente section évalue *les impacts de l'algorithme d'allocation hybride décrit en section 4.4 et sa capacité à assurer le principe tel-tel aux conteneurs.*

4.5.1. Méthodologie

L'objectif de cet algorithme est d'allouer au mieux un ensemble de processeurs minimal à chaque conteneur $\mathcal{C}_{\text{tel}}$, tout en évitant de partager des cœurs entre les conteneurs $\mathcal{C}_{\text{tel}}$ (voir section 4.4). Une contrainte annexe est la minimisation du gaspillage de ressource. En effet, cela peut entraîner le rejet de conteneurs à cause de l'impossibilité de respecter le principe tel-tel, malgré la présence de ressource processeur en quantité suffisante. Ainsi, on définit deux métriques d'intérêt :

1. p est la proportion de temps processeur, sur tout le centre d'hébergement, qui est alloué aux conteneurs $\mathcal{C}_{\text{tel}}$ à partir de processeurs partagés entre plusieurs conteneurs $\mathcal{C}_{\text{tel}}$;
2. r est le taux de rejet de conteneurs $\mathcal{C}_{\text{tel}}$ à cause de l'impossibilité de fournir une allocation respectant le principe tel-tel.

L'algorithme possède deux versions : l'une est stricte, et rejette les conteneurs $\mathcal{C}_{\text{tel}}$ pour lesquels il ne peut pas garantir le principe tel-tel ; et la seconde est lâche, et accepte l'allocation de tous les conteneurs tant que le serveur a assez de ressources pour eux. Autrement dit, pour la version lâche $r = 0$.

Par ailleurs, la section 4.4.4 explique qu'il y a deux parties à l'intégration de l'algorithme dans un ordonnanceur :

1. l'allocation hybride au niveau du serveur ;
2. l'optimisation de la sélection du serveur de déploiement par l'orchestrateur en simulant l'allocation hybride.

On évalue donc aussi l'impact sur p et r de cette sélection du meilleur serveur au niveau de l'orchestrateur. Enfin, on vérifie le passage à l'échelle des deux parties par rapport à la proportion de conteneurs $\mathcal{C}_{\text{tel}}$, c'est-à-dire de conteneurs qui nécessitent un traitement spécial.

L'évaluation est une simulation d'allocations extraites de traces d'un centre d'hébergement. Le code source de la simulation est donné en annexe B. Les traces d'allocations proviennent de la surveillance par Google de 12500 serveurs pendant environ un mois [114].

Chaque conteneur a une certaine probabilité d'être déterminé comme $\mathcal{C}_{\text{tel}}$, c'est-à-dire de nécessiter un ensemble de processeurs, sinon d'être $\mathcal{C}_{\overline{\text{tel}}}$. La valeur de cette probabilité varie pendant les expériences.

Une caractéristique importante des traces de Google est que leur centre d'hébergement sur-alloue les ressources, ce qui n'est pas le cas de Kubernetes. Ce n'est de fait pas non plus le cas de notre système d'allocation hybride. Par conséquent, le centre d'hébergement simulé reçoit une très lourde charge de travail et ne peut pas allouer tous les conteneurs des traces. On notera que la métrique r ne compte que les rejets dus à la violation du principe tel-tel. Les rejets dus au simple manque de ressources absolues sont ignorés.

De l'importance de p La métrique de temps processeur partagé p est importante pour la raison suivante. Ce partage entre des conteneurs $\mathcal{C}_{\text{tel}}$ mène à une dégradation de leurs performances par rapport au respect du principe tel-tel grâce à un ensemble exclusif de processeurs. Par exemple, dans une situation où deux conteneurs $\mathcal{C}_{\text{tel}}$ font des requêtes pour 350% de processeur, chacun d'eux se voit allouer un ensemble de 4 processeurs sous un quota de 350%. Cependant, ces ensembles peuvent tout à fait avoir un processeur en commun, qui sert 50% à chaque conteneur. Ce cœur est réparti équitablement entre les deux conteneurs, mais son partage provoque, de manière tout à fait logique, des interférences d'ordonnancement et d'accès aux autres ressources. La conséquence de ces interférences est une dégradation des performances et de leur prédictibilité. Celle-ci est cependant inattendue par l'utilisateur qui déploie une application nécessitant le respect du principe tel-tel. Une évaluation avec le banc d'essai Stream présenté en section 4.3 le montre s'exécutant 1,7 fois plus lentement avec un cœur partagé avec une autre instance de lui-même, qu'avec l'exclusivité de son ensemble de processeurs.

Ce problème d'interférence n'est pas au cœur de ce travail, parce qu'il est très connu indépendamment de l'utilisation des ensembles de processeurs ; et il l'est tout particulièrement pour les applications dont l'utilisation de la mémoire est centrale, telles que Stream [91, 92]. Néanmoins, les ensembles de processeurs représentent une meilleure alternative que le quota d'ordonnanceur pour ce qui est de préserver les performances : dans l'exemple de Stream décrit ci-dessus, le banc d'essai reste en moyenne 12% plus rapide sous un ensemble de processeurs partagé, que sous un quota d'ordonnanceur.

Pour résumer au sujet de la métrique p, celle-ci révèle en fait *la perte de flexibilité d'ordonnancement* inhérente à l'utilisation d'ensembles de

processeurs. Cela représente en effet une contrainte sur les décisions de l'ordonnanceur, comme expliqué en section 4.3.4.

4.5.2. Résultats

On présente ici les résultats d'évaluation de l'algorithme d'allocation.

Optimisation niveau orchestrateur

Les résultats des expériences de comparaison de p et r avec et sans l'optimisation niveau orchestrateur, sont rapportés en figure 4.7. L'allocation hybride au niveau du serveur est stricte.

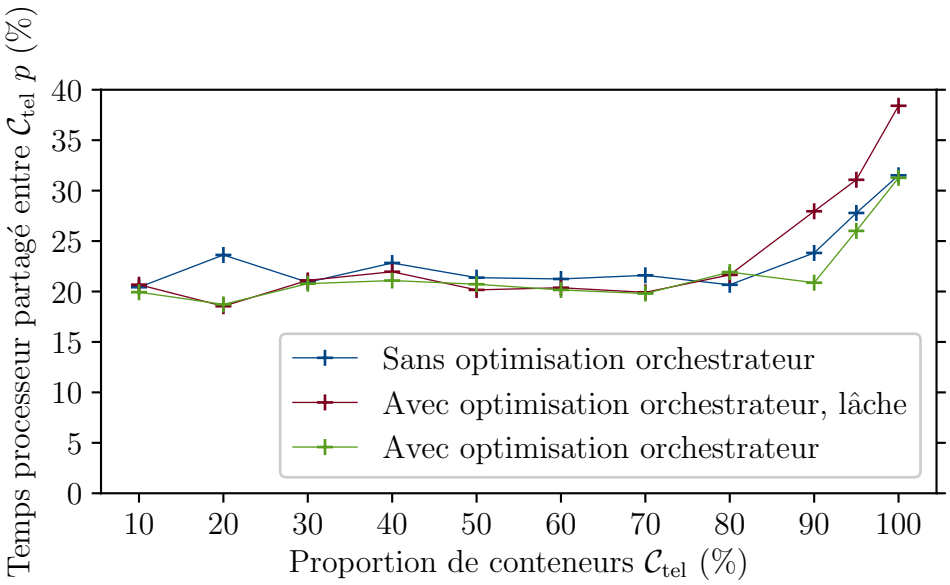
On observe dans les deux cas une certaine stabilité de p entre 10% et 90% de conteneurs $\mathcal{C}_{\text{tel}}$. L'algorithme optimisé provoque une proportion de temps processeur partagé entre plusieurs conteneurs $\mathcal{C}_{\text{tel}}$ d'environ 20,4% ; sans l'optimisation, cette proportion est d'environ 21,8%. Elle a donc un impact positif, bien que limité, sur s, en réduisant le partage de temps processeur d'environ 1,4 points (6,8%) pour les proportions de conteneur $\mathcal{C}_{\text{tel}}$ entre 10% et 90%. Les comportements particuliers observés au-delà de 90% de conteneurs $\mathcal{C}_{\text{tel}}$ sont commentés dans un paragraphe dédié plus bas.

En ce qui concerne le taux de rejets r, il est nul pour les deux versions entre 10% et 60% de conteneurs $\mathcal{C}_{\text{tel}}$. À partir de 70%, la version optimisée augmente le taux de rejet d'environ 0,3 points par rapport à la version non optimisée. Ce comportement est attendu puisque l'algorithme optimisé fait de son mieux pour éviter le partage de temps processeur entre les conteneurs $\mathcal{C}_{\text{tel}}$, c'est-à-dire pour minimiser p. Cet objectif provoque une fragmentation plus importante de l'allocation processeur parmi les cœurs de $\mathcal{P}_{\text{frac}}$; et cette fragmentation signifie qu'il y a moins de processeurs entiers disponibles dans $\mathcal{P}_{\text{part}}$ pour les nouveaux ensembles de processeurs (voir section 4.4). Cette dégradation est cependant très faible.

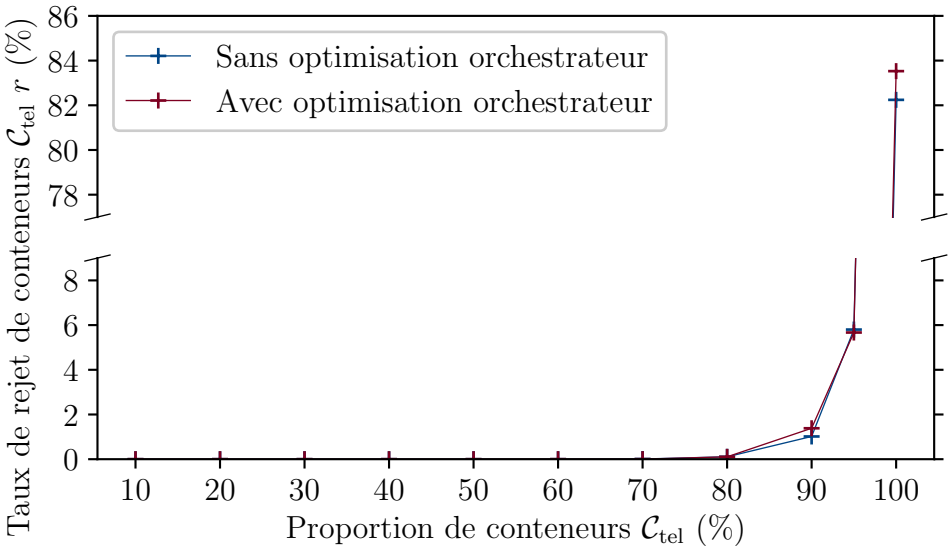
Allocation hybride stricte et lâche niveau serveur

Les résultats de p en figure 4.7a comparent aussi les politiques stricte et lâche de l'allocation hybride au niveau du serveur (avec dans les deux cas l'optimisation niveau orchestrateur). On rappelle que par définition, $r = 0$ avec la politique lâche ; le taux de rejet r pour la politique stricte est montré en figure 4.7b.

Dans les deux cas, on observe des résultats proches entre 10% et 80% de conteneurs $\mathcal{C}_{\text{tel}}$. Cependant, pour les proportions plus élevées, la politique



(a) Partage de temps processeur entre conteneurs $\mathcal{C}_{\text{tel}}$.



(b) Rejet des conteneurs $\mathcal{C}_{\text{tel}}$ pour impossibilité de respecter le principe tel-tel.

FIGURE 4.7. – Effets de l'optimisation de la sélection du serveur par l'orchestrateur pour minimiser p ; et comparaison des versions stricte et lâche de l'allocation hybride au niveau du serveur.

lâche alloue beaucoup de conteneurs $\mathcal{C}_{\text{tel}}$ avec des cœurs partagés. La proportion de temps partagé p augmente de 33,9%. Cela correspond à l'augmentation du taux de rejet des conteneurs que l'on observe pour la politique stricte.

Ainsi, la politique stricte ne rejette pas de conteneurs $\mathcal{C}_{\text{tel}}$ jusqu'à une proportion de 60%, et la relaxe du principe tel-tel n'est donc pas nécessaire. Il est par conséquent préférable de n'utiliser la politique lâche qu'en cas d'obligation d'accepter autant de conteneurs que possible.

Pour les proportions de conteneurs $\mathcal{C}_{\text{tel}}$ supérieures à 90% On remarque que lorsque les conteneurs sont presque uniquement de type $\mathcal{C}_{\text{tel}}$, les valeurs de p et de r explosent ($s = 31,3\%$ et $r = 83,5\%$, politique stricte et orchestrateur optimisé). C'est un argument en faveur de l'allocation hybride, qui montre pourquoi utiliser exclusivement les ensembles de processeurs n'est pas viable. En effet, la réserve de processeurs disponibles dans $\mathcal{P}_{\text{part}}$ s'épuise très rapidement avec une grande proportion de conteneurs $\mathcal{C}_{\text{tel}}$. Il ne reste donc plus de processeurs entiers à allouer, ce qui rend impossible la garantie du principe tel-tel ; en conséquence de quoi le taux de rejet r augmente. De manière similaire, la proportion de temps processeur partagé p augmente. C'est parce que les conteneurs pour lesquels l'algorithme hybride arrive encore à trouver une allocation qui respecte le principe tel-tel, partagent obligatoirement au moins un cœur.

Pour résumer, *l'absence de conteneurs de type $\mathcal{C}_{\overline{tel}}$ provoque une famine des processeurs entiers* sur les serveurs. De cette famine résulte des allocations globalement mauvaises, et même des allocations rejetées. Si on utilise la politique lâche, c'est-à-dire que l'on autorise les allocations qui violent le principe tel-tel pour les conteneurs $\mathcal{C}_{\text{tel}}$, p augmente encore à 38,4%.

Passage à l'échelle avec la proportion de conteneurs $\mathcal{C}_{\text{tel}}$

Lors de l'ordonnancement d'un conteneur $\mathcal{C}_{\overline{\text{tel}}}$, les algorithmes présentés en section 4.4 ne font rien de plus que l'algorithme classique. Cependant, un conteneur $\mathcal{C}_{\text{tel}}$ nécessite plus de travail.

Algorithme d'allocation hybride niveau serveur Le meilleur cas pour l'algorithme au niveau du serveur est de trouver tous les processeurs nécessaires dans $\mathcal{P}_{\text{part}}$, qui est donc une opération en $O(1)$. Le pire cas consiste à trouver des cœurs partagés dans $\mathcal{P}_{\text{frac}}$. La complexité est liée au nombre de cœurs dans $\mathcal{P}_{\text{frac}}$, dont une borne supérieure est c, le nombre de cœurs du serveur. Ainsi on considère que la complexité du pire cas est $O(c)$.

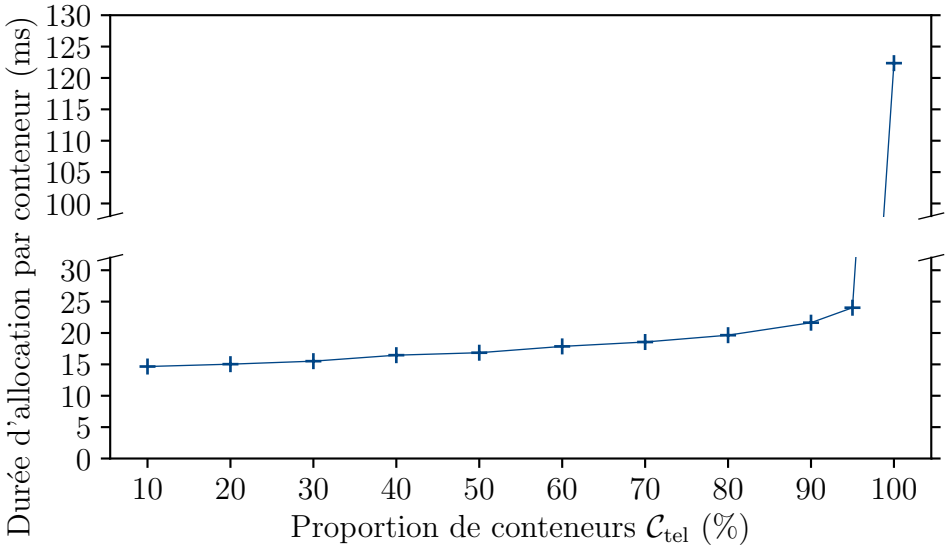


FIGURE 4.8. – Passage à l'échelle de l'optimisation de l'orchestrateur.

En fin de compte, l'algorithme niveau serveur n'a pas de problème de passage à l'échelle avec le nombre de conteneurs $\mathcal{C}_{\text{tel}}$. Il faut remarquer en revanche, qu'un grand nombre de ces conteneurs diminue statistiquement le nombre de processeurs dans $\mathcal{P}_{\text{part}}$, ce qui favorise l'occurrence du pire cas.

Optimisation de l'allocation niveau orchestrateur La figure 4.8 donne les résultats de l'évaluation du temps nécessaire à l'orchestrateur optimisé pour allouer les conteneurs, en fonction de la proportion de conteneurs $\mathcal{C}_{\text{tel}}$. On observe une augmentation de cette durée avec la proportion de ces conteneurs. Elle s'explique par le coût supérieur pour allouer un conteneur $\mathcal{C}_{\text{tel}}$ qu'un conteneur $\mathcal{C}_{\overline{\text{tel}}}$. En effet, dans le cas du dernier, l'orchestrateur ne fait que sélectionner les serveurs en fonction des ressources disponibles. Alors que pour le premier, l'orchestrateur simule l'allocation sur chaque serveur. Avec l'augmentation de la proportion de conteneurs $\mathcal{C}_{\text{tel}}$, la durée moyenne de cette simulation augmente.

Il faut aussi noter la très forte augmentation au-delà de 90% de conteneurs $\mathcal{C}_{\text{tel}}$. Elle provient de la famine en processeurs $\mathcal{P}_{\text{part}}$, déjà évoquée plus haut. Celle-ci implique que l'algorithme d'allocation niveau serveur rencontre plus souvent son pire cas. Ainsi, les simulations deviennent presque toujours de complexité $O(c)$, avec c le nombre de cœurs sur le serveur.

Par ailleurs, la durée d'exécution de l'optimisation de l'orchestrateur dépend aussi du nombre de serveurs n considérés pour l'ordonnancement d'un conteneur. C'est parce que l'orchestrateur simule l'allocation du conteneur $\mathcal{C}_{\text{tel}}$ sur chaque serveur. L'impact de cette simulation peut cependant être largement réduit en déchargeant celle-ci sur chaque serveur. Il suffit alors que le serveur renvoie à l'orchestrateur le résultat de sa simulation. Il ne reste plus à ce dernier qu'à comparer la qualité des allocations du point de vue du respect du principe tel-tel. C'est une architecture courante dans ce genre de cadriciels de gestion de centre d'hébergement [19], qui délèguent la surveillance aux serveurs eux-mêmes.

De plus, l'algorithme d'allocation niveau serveur nécessite environ 40 µs. Cette durée est négligeable face à la durée de création d'un conteneur, qui se compte en centaines de millisecondes [123]. Par conséquent, le déchargement de la simulation sur chaque serveur réduira énormément le temps de sélection du serveur au moment de l'ordonnancement d'un conteneur $\mathcal{C}_{\text{tel}}$. On notera que les résultats de la figure 4.8 proviennent de notre simulation prototype, qui n'intègre justement pas cette optimisation de décharge de la simulation.

4.6. État de l'art

La problématique de ce chapitre est donc l'impact de la virtualisation niveau SE sur les performances. Cet impact est causé par la violation du principe tel-tel. La solution suggérée est la conception d'un algorithme d'allocation des ressources CPU, qui est hybride en ce qu'il utilise le mécanisme du quota d'ordonnanceur en même temps que les ensembles de processeurs. On présente dans cette section des travaux concernant ces trois aspects que sont l'étude de l'impact de la virtualisation, la violation du principe tel-tel et la gestion de la ressource processeur pour la virtualisation.

4.6.1. Performance des conteneurs

Plusieurs études comparent les conteneurs aux VM [45, 90, 120, 123]. Ils offrent généralement de meilleures performances aux applications virtualisées. On peut aussi citer d'autres études de performance plus ciblées, comme le travail de Morabito [89] sur la consommation énergétique, qui ne montre en fait pas de différence majeure entre les conteneurs et les VM. Enfin, l'étude de Li et Kanso [72] teste la capacité des conteneurs à fournir un service haute disponibilité ; et celle de Dua, Raja et Kakadia [39] les évalue en remplacement des VM dans les plate-formes PaaS.

4.6.2. Violation du principe tel-tel

Une remarque intéressante de l'étude de Felter et al. [45] est que l'ignorance par l'application de sa virtualisation représente une limitation fondamentale à la conteneurisation. Cette ignorance mène effectivement à la violation du principe tel-tel. De plus, la différence de performance entre l'application d'un quota d'ordonnanceur et celle d'un ensemble de processeurs a aussi été observée dans les expériences de Sharma et al. [123]. Ils n'ont cependant pas plus analysé ce problème.

Cependant, Oracle a aussi remarqué cette difficulté, et a proposé une solution sous la forme d'une bibliothèque support [77]. Son but est d'abstraire l'obtention des informations sur les ressources disponibles. En particulier, son but est de prendre en compte la conteneurisation. [2] L'inconvénient est que l'application doit être mise à jour afin d'intégrer cette bibliothèque.

Une autre initiative de l'industrie contre la violation du principe tel-tel dans les conteneurs, est la mise à jour de la machine virtuelle Java (JVM, pour *Java Virtual Machine*) pour qu'elle soit consciente de sa conteneurisation [62]. Ainsi, à partie de la version 10, la JVM s'auto-configure en fonction de la mémoire et du processeur alloués à son conteneur.

4.6.3. Gestion de la ressource processeur virtualisée

On compte de nombreux travaux cherchant à améliorer la gestion des ressources pour le mode de virtualisation spécifique que sont les conteneurs.

Tout d'abord, le travail de Hoenisch et al. [53] donne une modélisation du dimensionnement des VM et des conteneurs comme un problème d'optimisation. Il existe d'autres modélisations des conteneurs pour faciliter la gestion de leurs ressources du point de vue de l'orchestrateur [105].

À partir de la première modélisation [53], ElasticDocker [36] a été développé pour apporter la mise à l'échelle verticale aux conteneurs, qui est une fonctionnalité déjà présente pour les VM. Bien que ce projet gère à la fois le nombre de vCPU (quand l'hôte est une VM) et le quota d'ordonnanceur des conteneurs, il n'a pas étudié l'impact d'un nombre de vCPU erroné, ce qui constituerait une violation du principe tel-tel. ElasticDocker pourrait tout à fait être intégré à la solution présentée dans ce chapitre afin d'améliorer le calcul des allocations en ressources des conteneurs.

2. Pour prendre en compte la conteneurisation, la bibliothèque utilise en fait les informations de ressources du cgroup du conteneur. Voir la section 4.4.1 au sujet des cgroups.

Similairement, Baresi et al. [13] ont aussi développé une solution pour la mise à l'échelle verticale automatique des conteneurs et de leurs VM hôtes. Notamment, leur projet implémente un système d'attaches (*hooks*) spécifiques aux applications conteneurisées. Leur but est justement de résoudre le problème de la mise à l'échelle verticale dynamique de l'application, en tandem avec son conteneur. Autrement dit, ces attaches sont une solution au problème de la violation du principe tel-tel. Le travail présenté dans ce chapitre est quant à lui indépendant des applications conteneurisées. L'allocation dynamique de ressources au conteneur requerrait néanmoins un moyen de notifier l'application du changement de son allocation.

Enfin, on peut rapprocher l'utilisation des ensembles de processeurs qui est faite par l'algorithme hybride présenté dans ce chapitre, de la technique d'épinglage des vCPU (en anglais *vCPU pinning*). Il s'agit de fixer l'ordonnancement des vCPU des VM sur les processeurs physiques. Cette technique a des effets bénéfiques sur la consommation énergétique, les performances, et la réduction des interférences entre les applications virtualisées [109]. On peut imaginer intégrer cette technologie pour mieux attribuer les processeurs aux conteneurs (voir le détail en section 4.4) [73].

4.7. Conclusion

Ainsi, les systèmes de gestion des conteneurs actuels allouent les ressources avec un mécanisme qui a un impact négatif fort sur les performances des applications et leur prédictibilité. En effet, ce mécanisme du quota d'ordonnanceur trompe une application qui s'auto-configure, en lui montrant tous les processeurs du serveur hôte au lieu de lui indiquer la véritable limite en ressources. *Il viole le principe tel-tel*. C'est cependant le mécanisme privilégié pour sa facilité d'utilisation et sa flexibilité de gestion.

Ce chapitre propose d'adjoindre l'utilisation d'un autre mécanisme d'allocation, les ensembles de processeurs, via un algorithme d'allocation processeur combinant les deux mécanismes. Il s'agit d'abord de déterminer si l'application a besoin du respect du principe tel-tel, c'est-à-dire si elle est sensible au nombre de processeurs effectivement visibles. Dans ce cas, ses ressources processeurs doivent être allouées avec le mécanisme qui permet la garantie du principe tel-tel. Les expériences montrent que cet algorithme corrige bien le problème de violation du principe tel-tel tout en bénéficiant de la flexibilité d'allocation permise par le premier mécanisme.

Bien que cet algorithme soit conçu pour la ressource processeur, la violation du principe tel-tel affecte aussi les ressources mémoire et réseau.

Chapitre 5

La double virtualisation du réseau : BrFusion et HostLo

5.1. Introduction 105
5.2. Notions préliminaires 108
5.3. Analyse des défauts de la virtualisation réseau imbriquée . 110
5.4. HostLo : déploiement inter-VM de pods 115
5.5. BrFusion : déduplication de la virtualisation du réseau . . 133
5.6. État de l'art 146
5.7. Conclusion 148

5.1. Introduction

Les chapitres 3 et 4 précédents illustrent les rôles qu'occupent désormais les deux modes de virtualisation. La virtualisation du matériel sert principalement au fournisseur de cloud pour gérer les ressources de son centre d'hébergement sous la forme de VM ; tandis que la virtualisation du système d'exploitation facilite pour l'utilisateur le déploiement dans le cloud, sous la forme de conteneurs. On imagine aisément que les fournisseurs de cloud cherchent à attirer les clients en leur proposant des solutions toujours plus faciles, et qui sont donc basées sur les conteneurs. On remarque pourtant que *les VM restent omniprésentes* dans les offres cloud, quand bien même elles seraient seulement un détail d'implémentation plutôt qu'une partie intégrante de l'offre. En fait, la pratique montre de nombreux cas d'utilisation de la multi-virtualisation, qui ont été abordés dans la section 2.2.2.

Pourtant, l'étude des mises en application et de l'utilisation de la virtualisation imbriquée montre les limites très fortes de cette pratique. Le présent

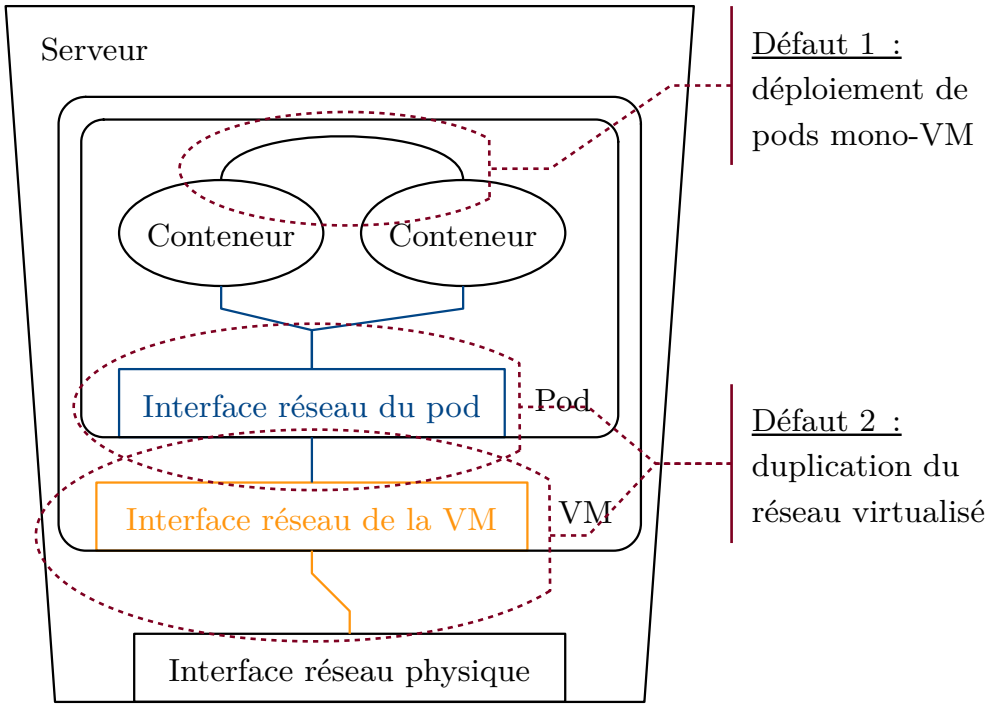


FIGURE 5.1. – Vue simplifiée des défauts de la virtualisation imbriquée. Les éléments en bleu participent au réseautage des pods ; les éléments en orange participent au réseautage des VM.

chapitre met en lumière deux défauts dans sa conception courante. La figure figure 5.1 les illustre dans l'architecture de la virtualisation imbriquée ; on les évoque ici, ils seront mieux détaillés dans la section 5.3.

Déploiement mono-VM de pods Malgré la différence du niveau de virtualisation, *le fait d'utiliser des VM a un impact sur le déploiement des conteneurs* par les orchestrateurs. En fait, plutôt que d'administrer des conteneurs individuellement, ils les contrôlent par groupes : les pods (voir section 5.2). Or, une limitation technique empêche les conteneurs qui composent un pod d'être séparés sur différentes VM. Elle provient de la virtualisation du réseau privé des pods, réalisée indépendamment dans chaque SE invité. La conséquence est la *fragmentation des ressources* : au lieu de les allouer finement à des éléments petits, les conteneurs, elles sont allouées par lot à des groupes plus gros, les pods. Cette perte de finesse de l'allocation est ce qui provoque la fragmentation des ressources et leur gaspillage : les allocations en lot laissent des trous dans les ressources des VM, qui sont

trop petits pour accueillir un pod et ne seront donc jamais utilisés.

Duplication du réseau virtualisé Les deux niveaux de virtualisation abstraient chacun le réseau de manière indépendante. En effet, le SE du serveur qui héberge les VM le virtualise une première fois pour celles-ci ; et ensuite, chaque VM va à nouveau virtualiser ses interfaces pour les pods qui y sont déployés. Comme les deux couches de virtualisation du réseau utilisent les mêmes composants logiciels pour implémenter ce routage virtuel, on constate exactement une *duplication du réseau virtuel*. Les expériences montrent une dégradation du débit allant jusqu'à 68%, ainsi qu'une latence augmentée jusqu'à 31%, par rapport à une seule couche de virtualisation.

On présente dans ce chapitre une solution pour chacune de ces deux problématiques. La première solution est nommée HostLo, pour *host localhost*. C'est un type d'interface réseau virtuelle conçue spécialement pour la communication entre les conteneurs d'un pod, et provisionnée par le serveur hôte pour les pods. Elle permet ainsi des *déploiements de pod inter-VM*, en utilisant l'hôte physique pour fournir aux pods une nouvelle interface réseau virtuelle spécialement dédiée à la communication intra-pod.

La seconde est baptisée BrFusion, pour *bridge fusion*. Son principe est de *fusionner les deux couches de virtualisation du réseau*, implémentées principalement comme des ponts (en anglais *bridges*), c'est-à-dire des commutateurs réseau virtuels. Cette fusion est accomplie en virtualisant les ressources réseau du serveur hôte physique afin de fournir directement des interfaces réseau aux pods, plutôt qu'aux VM comme des étapes intermédiaires. Les deux solutions sont orthogonales : HostLo fournit à un pod une interface réseau spéciale pour la communication interne, tandis que BrFusion simplifie la provision de l'interface réseau virtuelle du pod pour communiquer avec l'extérieur.

Finalement, la correction de ces deux défauts de la virtualisation imbriquée peut être interprétée comme sa *désimbrication*. En effet, il s'agit d'éliminer du point de vue de l'utilisateur, la couche de virtualisation des VM. L'objectif plus grand est d'enfin placer l'orchestrateur au cœur du centre d'hébergement cloud ; l'hyperviseur devient son subordonné, utilisé pour la gestion des ressources offertes aux conteneurs.

Afin de caractériser plus précisément l'impact de ces défauts de la virtualisation imbriquée, une analyse a d'abord été menée. Elle met en lumière leurs effets sur les performances réseau ainsi que sur l'utilisation des ressources. Ces conclusions ont dirigé la conception des deux solutions présentées ici.

Organisation du chapitre

La section 5.2 décrit les concepts et notions spécifiques à ce chapitre. L'analyse des défauts de la virtualisation imbriquée est reproduite dans la section 5.3. Ensuite, la section 5.4 présente d'abord HostLo, qui permet le déploiement de pods inter-VM ; puis la section 5.5 présente BrFusion, visant à corriger le défaut de la double virtualisation du réseau. Chacune de ces deux sections conclut sur une évaluation de la solution qui y est détaillée. Pour finir, la section 5.6 présente l'état de l'art du sujet, et la section 5.7 donne les conclusions de ce chapitre.

5.2. Notions préliminaires

On se place dans le contexte de la virtualisation imbriquée, et on s'intéresse plus particulièrement au réseau dans cet environnement. Cette section détaille d'abord la notion de pod, qui est au centre du second défaut de la virtualisation imbriquée ; puis elle clarifie la terminologie utilisée pour désigner les deux niveaux de virtualisation, en décrivant le réseautage virtuel dans la multi-virtualisation.

5.2.1. Pods

L'intermédiaire de gestion du cloud présenté aux utilisateurs est l'orchestrateur de conteneurs. En fait, dans le modèle d'orchestrateur de Kubernetes [23], celui-ci gère des pods (littéralement, une cosse ou gousse de pois, ou alors un banc de baleines comme celle du logotype de Docker [42]), c'est-à-dire des *groupes logiques de conteneurs*. Concrètement, dans un modèle de déploiement d'une application en micro-services, un pod fournit un de ces micro-services. Par exemple, un pod peut inclure un conteneur qui sert effectivement la fonctionnalité ; ainsi qu'un autre conteneur dédié à la surveillance du conteneur principal, et on peut encore y adjoindre un conteneur pour la journalisation, ou bien un conteneur adaptateur pour standardiser l'interface du service, etc.

Ainsi, le pod est *l'unité atomique de déploiement et de réplication* pour l'orchestrateur. Cela implique que l'orchestrateur alloue des ressources au pod entier de manière indivisible, et non pas à chaque conteneur. Les ressources allouées sont simplement la somme des requêtes de chaque conteneur individuel du pod. Cette mise en commun des ressources correspond en fait à une différence dans leur isolation.

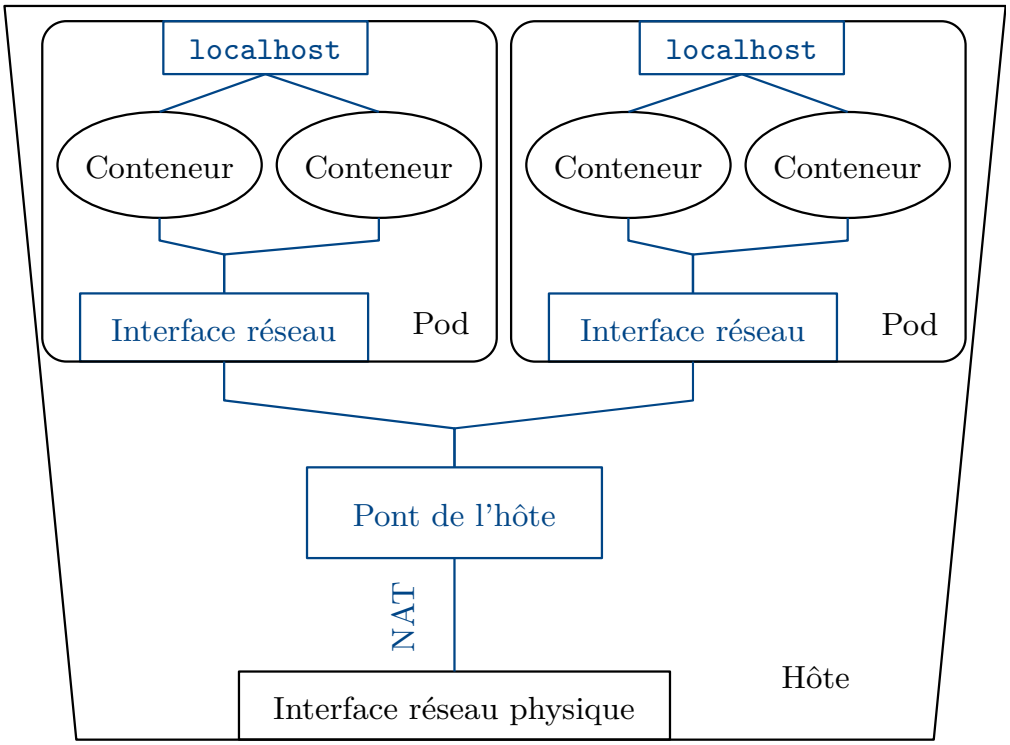


FIGURE 5.2. – Réseautage des pods sur un hôte (serveur ou VM).

En effet, l'isolation des ressources des conteneurs évolue lorsqu'on considère les pods. En pratique, les conteneurs d'un pod *partagent la même isolation réseau*, c'est-à-dire qu'ils accèdent aux mêmes interfaces réseau virtuelles. Ils partagent notamment la même interface `localhost`, comme illustré en figure 5.2. Il s'agit d'une interface virtuelle fournie par le SE hôte, et reproduite de manière privée dans chaque pod. Elle agit comme une boucle de retour (en anglais *loopback*) extrêmement efficace. Les conteneurs d'un pod l'utilisent pour communiquer entre eux par connexion IP.

De plus, de la même manière que les conteneurs d'un pod partagent leurs interfaces réseau, ils ont aussi en commun des volumes disques et un domaine de mémoire partagée [108]. Les volumes disques sont nécessaires parce que les conteneurs sont généralement non persistants. Le stockage des données se fait en écrivant dans un volume disque monté dans un pod dans le contexte présent. Concernant la mémoire partagée, comme l'isolation est au niveau du pod plutôt que des conteneurs individuels, ces derniers peuvent l'utiliser (généralement via une bibliothèque logicielle et un protocole dédiés) pour communiquer de manière encore plus efficace entre eux.

5.2.2. Virtualisation imbriquée du réseau

Le contexte de ce chapitre est donc la virtualisation imbriquée, évoquée en section 2.2.2, c'est-à-dire que l'on a un empilement des modes de virtualisation :

1. niveau machines virtuelles : c'est l'hyperviseur qui a le contrôle des ressources physiques ;
2. niveau conteneurs : l'orchestrateur est en quelque sorte client de l'hyperviseur, il utilise les VM comme hôtes pour déployer ses pods.

Cet empilement se répercute dans la virtualisation du réseau, comme illustré en figure 5.3. C'est d'ailleurs la raison du premier défaut de la virtualisation imbriquée, évoqué en introduction.

Cette figure montre donc trois interfaces virtuelles et deux ponts pour la commutation virtuelle du réseau. Leurs rôles sont décrits ci-dessus, du plus bas au plus haut :

interface réseau physique carte réseau physique du serveur physique, virtualisée par l'hyperviseur ;

pont de l'hôte pont utilisé par l'hyperviseur pour virtualiser l'interface réseau physique ;

interface réseau de la VM carte réseau virtuelle de la VM, c'est-à-dire fournie par l'hyperviseur pour la VM, virtualisée ensuite par l'orchestrateur ;

pont de la VM pont utilisé par l'orchestrateur pour virtualiser l'interface réseau de la VM ;

interface réseau du pod interface réseau virtuelle du pod, c'est-à-dire fournie par l'orchestrateur pour le pod.

Entre chaque pont et l'interface réseau qu'il virtualise, on trouve des règles de routage NAT (pour *Network Address Translation*, c'est-à-dire « traduction d'adresse réseau »). Elles permettent de router les paquets entre le pont et l'interface afin de fournir au pod l'accès au monde extérieur.

5.3. Analyse des défauts de la virtualisation réseau imbriquée

L'introduction a présenté deux défauts majeurs de la virtualisation imbriquée, du point de vue du réseau :

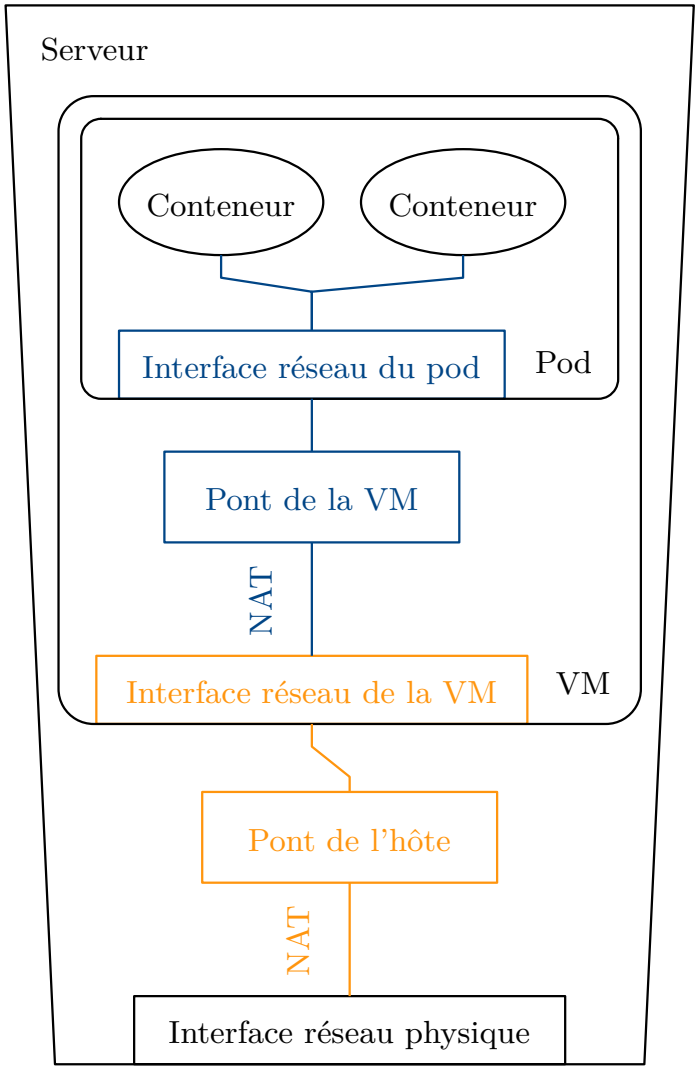


FIGURE 5.3. – Virtualisation imbriquée du réseau. Les éléments en bleu participent au réseautage des pods ; les éléments en orange participent au réseautage des VM.

1. déploiement mono-VM de pods ;
2. duplication du réseau virtualisé.

Cette section pousse l'analyse de ces défauts plus avant, et quantifie leurs impacts sur les applications doublement virtualisées.

5.3.1. Origine

Le réseau en tant que ressource virtuelle, est très différent des autres ressources comme le processeur ou la mémoire. Cette différence provient de ce que le réseau est en fait constitué de deux ressources :

1. provision matérielle et logicielle pour l'envoi et la réception de données : piles protocolaires, queues de paquets, ports physiques, etc. ;
2. identité sur les réseaux : communément, l'adresse Ethernet MAC et les adresses IP.

La mutualisation de la provision matérielle et logicielle, dans le but de la virtualiser, présente des problématiques proches de celles du processeur et de la mémoire : arbitrage de l'utilisation des ressources, isolation de celles-ci, etc. Mais c'est *la mutualisation de l'identité réseau* qui est spéciale. Elle demande des technologies spécifiques au réseau et à sa finalité. Par exemple, pour fournir le réseau à ses conteneurs, Docker s'appuie d'abord sur un pont afin de mutualiser l'adresse Ethernet MAC de la carte réseau physique. Ensuite, il configure NAT (voir section 5.2.2) pour transférer les paquets entre ce pont et l'interface physique au niveau IP. Cette configuration à base d'un pont et de NAT est très courante dans les installations de Docker mais aussi pour virtualiser le réseau des VM. C'est de cette installation, composée de la virtualisation du réseau de manière séparée entre les deux niveaux, que proviennent les défauts de la virtualisation imbriquée.

On remarque dans la figure 5.4 que ces défauts ciblent différentes sections du réseau d'un pod. Corriger ces défauts répond néanmoins au même objectif global *d'abstraction de la couche des VM* du point de vue de l'orchestrateur, et donc des utilisateurs.

5.3.2. Déploiement mono-VM de pods

En ce qui concerne le second défaut, on constate que le réseau est virtualisé indépendamment dans chaque VM. Cela devient un problème lorsque l'on considère les déploiements de pods, puisque tous les conteneurs d'un pod doivent communiquer les uns avec les autres par l'interface privée `localhost`

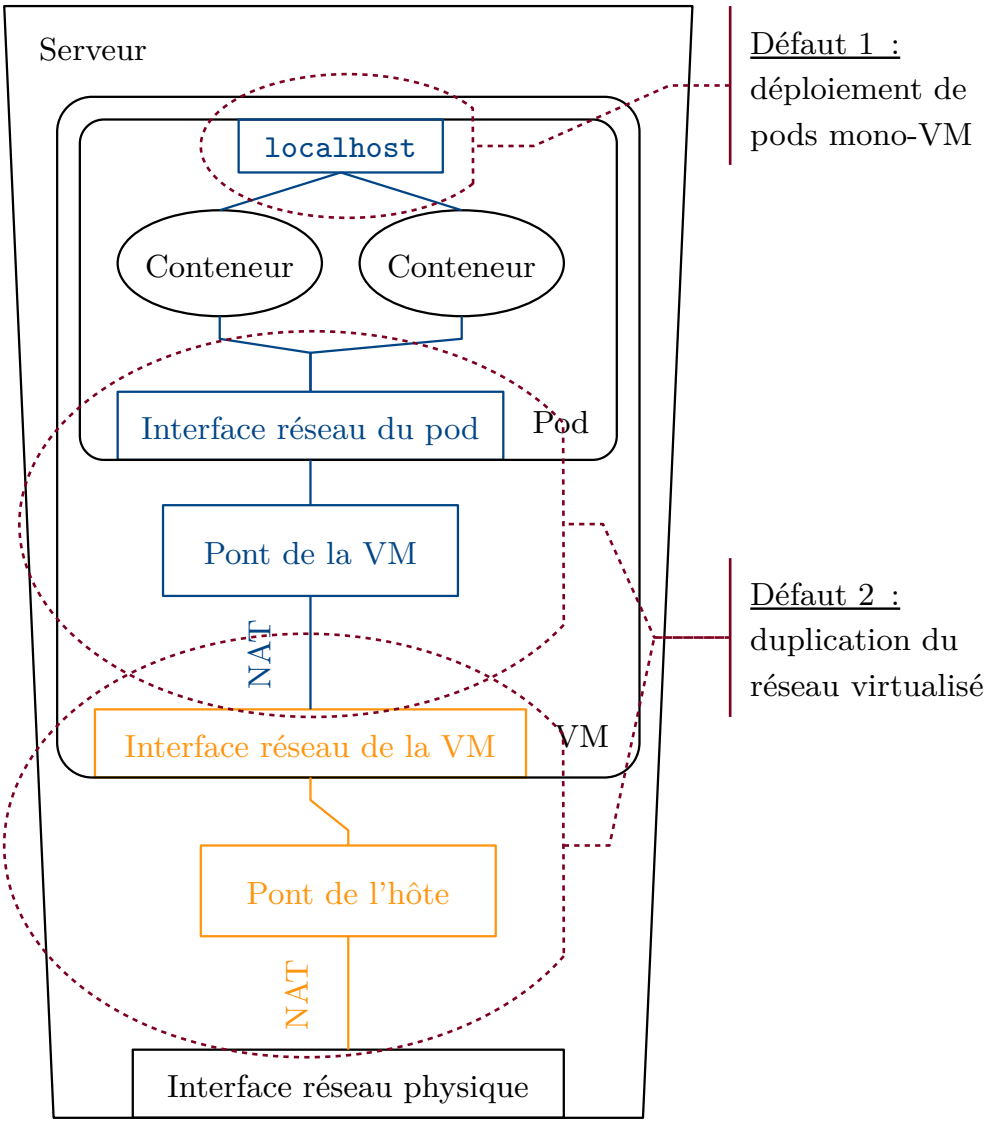


FIGURE 5.4. – Défauts de la virtualisation imbriquée du réseau. Les éléments en bleu participent au réseautage des pods ; les éléments en orange participent au réseautage des VM.

(voir section 5.2.1). Celle-ci est locale au pod, elle est donc aussi locale à la VM qui l'héberge. Ainsi, ce réseautage local à la VM est une contrainte au *déploiement inter-VM de pods*.

Cette contrainte constitue d'une part une diminution massive du taux d'utilisation global des ressources au niveau du centre d'hébergement ; et d'autre part constitue une augmentation significative du coût d'utilisation payé par les utilisateurs. Ces effets proviennent de la *fragmentation des ressources* provoquée par les déploiements mono-VM de pods.

Par exemple, on considère les dimensions en ressources des VM proposée par Amazon Web Services EC2 [40]. Si un pod requiert 6 vCPU et 24 Go de mémoire, il faut utiliser une instance `m5.2xlarge` pour l'héberger, au tarif de 0,448 $/h. C'est la dimension la plus petite qui permet d'héberger entièrement le pod, ce qui est actuellement une contrainte. Pourtant, il est possible d'obtenir ces mêmes ressource (6 vCPU et 24 Go de mémoire) à l'aide d'une instance `m5.large` associée à une autre instance `m5.xlarge`. Dans ce cas, le tarif total est de 0,336 $/h. Ainsi, la capacité à déployer un même pod sur plusieurs VM apporte des économies de ressources et d'argent importantes.

La solution décrite plus loin dans ce chapitre, HostLo, fournit cette fonctionnalité au prix d'une dégradation des performances acceptable.

5.3.3. Duplication du réseau virtualisé

Ainsi, le réseau est virtualisé deux fois :

1. au niveau de l'hôte physique, pour les VM ;
2. au niveau de la VM, pour les conteneurs.

Cette double virtualisation est illustrée en figure 5.3.

Concrètement, *elle double la longueur des chemins d'émission et de réception* des paquets entre le conteneur et l'extérieur de sa VM. De plus, les deux couches de virtualisation du réseau sont gérées par deux entités séparées qui ne communiquent pas ensemble : le gestionnaire de VM et l'orchestrateur de pods. Cette ségrégation rend difficile toute optimisation.

Ainsi, un paquet envoyé depuis l'application conteneurisée va suivre le chemin décrit ci-dessous ; un paquet reçu suivra le même chemin à l'envers.

1. le paquet est déposé sur l'interface réseau interne au pod, et traverse sa frontière ;
2. il est reçu sur le pont dans la VM : comme sa destination n'est pas sur les autres interfaces réseau sur le pont, il n'est pas commuté dessus, et sera géré par les règles NAT à l'étape suivante ;

3. le paquet est routé selon les règles NAT définies par l'orchestrateur, et atteint alors l'interface réseau virtuelle interne de la VM, où il traverse sa frontière ;
4. il est reçu sur le pont dans l'hôte : l'étape suivante dépend de sa destination ;
5. si le paquet est destiné à une autre VM, il est commuté sur le pont vers l'interface réseau virtuelle de la VM destination, sinon il est routé selon les règles NAT définies par le gestionnaire de VM et atteint l'interface réseau physique du serveur.

Une expérience avec le micro-banc d'essai Netperf montre l'impact de cette conception en figure 5.5. La partie serveur du banc d'essai s'exécute dans un contexte de virtualisation imbriquée (dans un conteneur, lui-même dans une VM), tandis que son client s'exécute nativement dans le SE hôte. Il écoute sur une interface virtuelle reliée au pont de l'hôte par NAT.

On observe une diminution du débit de 68% et une augmentation de la latence de 31% avec une taille de messages de 1280 o, comparativement avec la virtualisation à un seul niveau, c'est-à-dire lorsque le serveur Netperf s'exécute nativement dans la VM, sans conteneurisation. La solution décrite plus loin dans ce chapitre, BrFusion, permet d'atteindre un niveau de performance comparable à la virtualisation à un seul niveau, mais dans un contexte de virtualisation imbriquée.

5.4. HostLo : déploiement inter-VM de pods

Le second défaut de la virtualisation imbriquée est de ne pas permettre le déploiement inter-VM de pods. HostLo en est la solution.

5.4.1. Vue d'ensemble

En fait, proposer *un canal de communication efficace entre des VM*, qui soit privé pour le pod, permet de corriger ce défaut.

Tout d'abord, on rappelle que la communication interne au pod, c'est-à-dire entre les conteneurs qui composent un pod, repose sur l'interface `localhost` privée du pod (voir section 5.2). Comme elle est fournie par la VM hôte, le déploiement inter-VM du pod est impossible. En pratique, cette interface agit au niveau du lien comme une boucle de retour (en anglais *loopback*). Autrement dit, elle renvoie simplement les trames Ethernet qu'elle

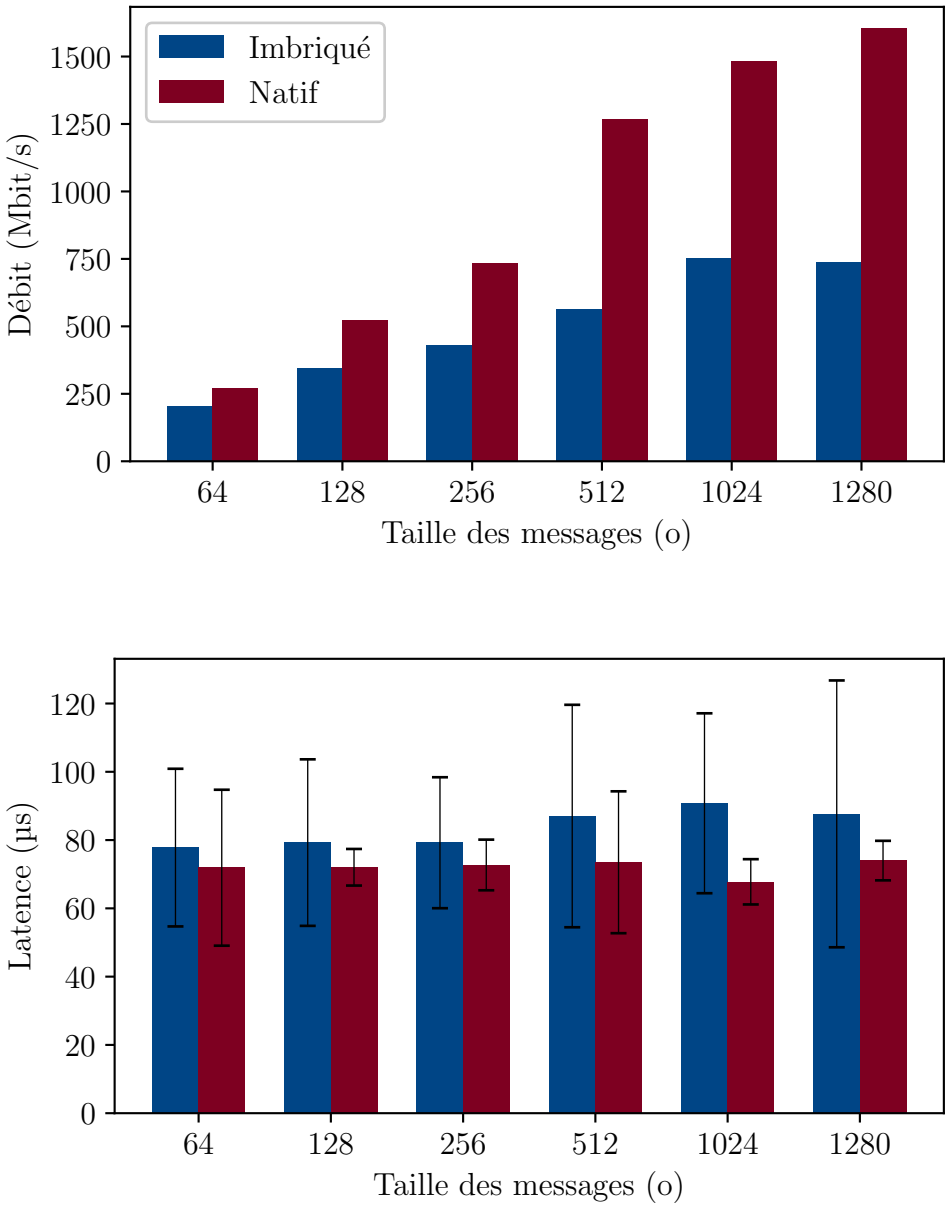


FIGURE 5.5. – Comparaison des performances réseau entre la virtualisation à un seul niveau (sans conteneur, natif dans la VM) et la virtualisation imbriquée. Extrait de la figure 5.17.

reçoit ; et ce bien qu'elle soit utilisée par le biais de son adresse IP. [1]

La solution HostLo, pour *host localhost*, c'est-à-dire « `localhost` côté hôte », est illustrée en figure 5.6. Il s'agit comme son nom l'indique, de *créer dans le serveur hôte une interface réseau virtuelle* particulière. Ses caractéristiques sont d'agir comme une interface de boucle retour, mais qui peut être multiplexée entre plusieurs VM. Ainsi, on crée dans chaque VM où des conteneurs du pod seront placés, une terminaison de cette interface. Cette terminaison sera utilisée exclusivement par les conteneurs du pod, et servira à leur fournir l'interface `localhost` qu'ils s'attendent à pouvoir utiliser pour communiquer entre eux.

5.4.2. Intégration

Comme pour BrFusion, l'intégration de HostLo repose sur la *communication entre l'orchestrateur de pods et le gestionnaire de VM*. Le premier décide du placement du pod, éventuellement inter-VM : dans ce cas, il demande au gestionnaire de VM de provisionner une nouvelle interface HostLo. Voici comment peut se dérouler cet échange :

1. l'orchestrateur demande au gestionnaire de VM la provision d'une nouvelle interface HostLo, et lui indique les VM auxquelles l'ajouter (celles où le pod sera placé) ;
2. le gestionnaire de VM crée la nouvelle interface, et insère des terminaisons dans les VM spécifiées, c'est-à-dire qu'il les ajoute comme de nouvelles interfaces réseau dans les VM ;
3. puis il retourne à l'orchestrateur une identification des terminaisons de l'interface HostLo dans les VM, pour que l'agent local de ce dernier les utilise pour le pod désagrégé ;
4. finalement, l'orchestrateur via son agent local dans la VM, y configure l'interface réseau (c'est-à-dire la terminaison de l'interface HostLo) et l'insère dans le pod en remplacement de son interface `localhost`.

Une illustration de cette séquence a été donnée en figure 5.15. L'intégration de HostLo est similaire à celle de BrFusion du point de vue de la communication entre l'orchestrateur et le gestionnaire de VM.

1. Techniquement, c'est tout le sous-réseau `127.0.0.0/8` qui peut être utilisé.

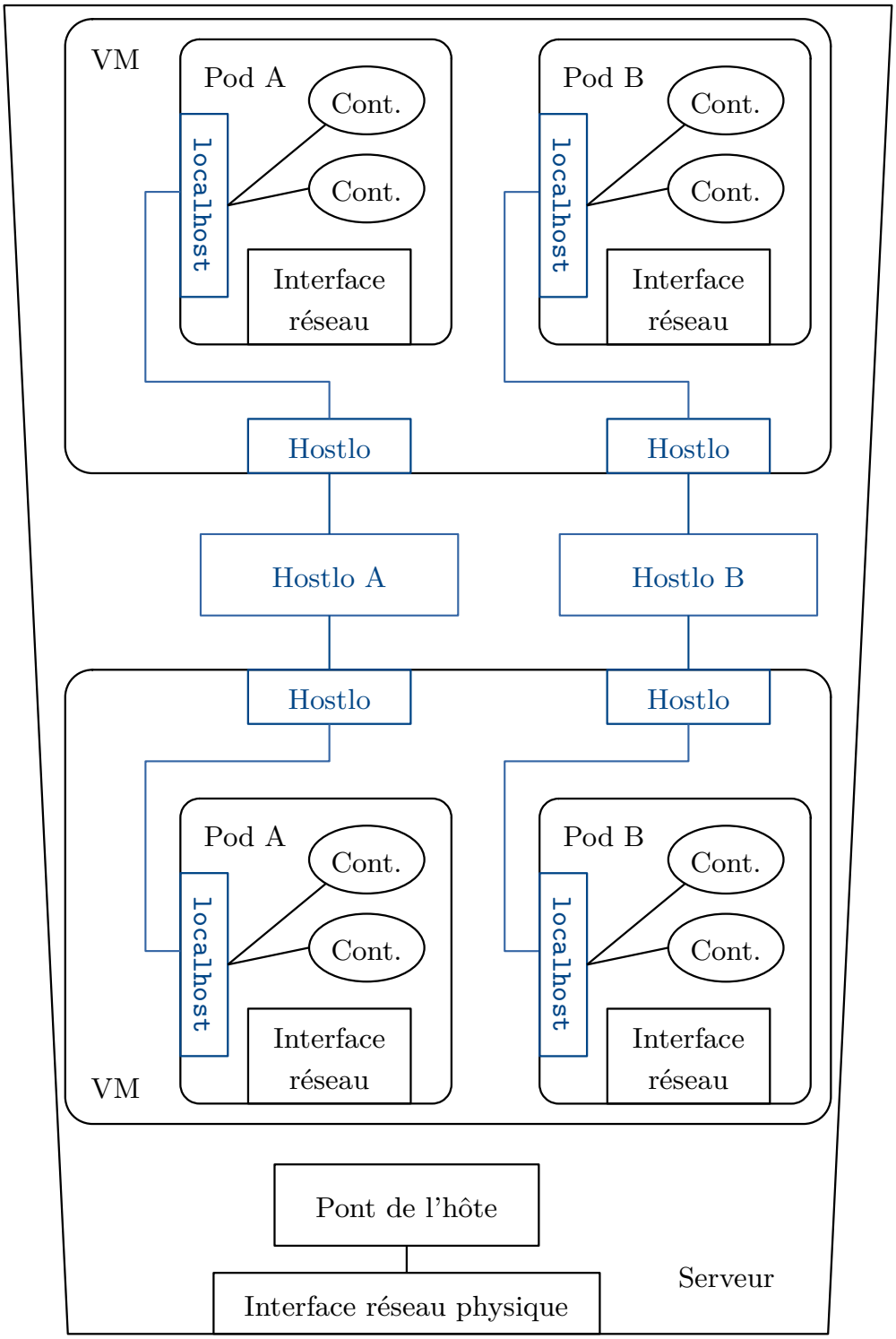


FIGURE 5.6. – Illustration de HostLo.

5.4.3. Implémentation

On propose une implémentation prototype de HostLo dans QEMU et le noyau Linux. Comme pour BrFusion (voir section 5.5.3), l'orchestrateur est peu impacté par l'implémentation, qui serait réalisée par un greffon CNI.

C'est la description de la communication interne au pod, donnée ci-dessus, qui guide la conception de HostLo. Une interface HostLo est une interface réseau virtuelle qui renvoie tous les paquets qu'elle reçoit, au niveau de la trame Ethernet. L'utilisation des interfaces TAP [132] est donc tout indiqué. Il s'agit d'interfaces réseau virtuelles, une fonctionnalité du noyau Linux. Elles traitent des trames Ethernet comme une interface physique d'un côté, et lisent et écrivent des trames Ethernet dans un descripteur de fichier de l'autre. Elles sont déjà utilisées par QEMU pour servir d'arrière-boutique à ses interfaces réseau virtualisées.

Ainsi, HostLo utilise une interface TAP modifiée pour agir comme une boucle de retour. Elle peut de plus être multiplexée entre plusieurs VM, ce qui n'est pas possible à l'origine. Concrètement, HostLo est une modification du pilote de périphériques TAP du noyau Linux :

— il fournit au moins une queue d'envoi et une queue de réception pour chaque VM où l'interface sera utilisée ;
— il renvoie toutes les trames Ethernet reçues à toutes ses queues.

Le code source de ce pilote est donné en annexe C.

De plus, il faut implémenter l'utilisation de HostLo dans QEMU, afin de lui permettre de provisionner ce nouveau type d'interface. La provision d'une interface HostLo consiste en fait à créer une interface TAP, mais avec le nouveau mode spécial `loopback` (boucle de retour), et d'ajouter une paire de queues d'envoi et de réception de cette interface à chaque VM ciblée. Concrètement, QEMU va passer des appels `ioctl` sur une interface `devfs` proposée par le noyau Linux. Pour finir, l'insertion des terminaisons de l'interface HostLo dans les VM s'effectue de la même manière que l'insertion de l'interface provisionnée dynamiquement pour BrFusion (voir section 5.5.3) : par le biais de l'interface de gestion parallèle QMP. Le code source de cette modification de QEMU est donné en annexe D.

5.4.4. Mise à jour de l'algorithme de placement des pods

Au-delà de l'implémentation de la fonctionnalité de HostLo, c'est-à-dire la capacité de déploiement inter-VM des pods, il faut mettre à jour l'algorithme de placement et de gestion des VM de Kubernetes. En effet, il n'a pas été conçu avec la possibilité de déployer ainsi des pods, ni avec la capacité

d'acheter une nouvelle VM si nécessaire. [2] On présente ici une mise à jour prototype de cet algorithme pour lui permettre d'exploiter le potentiel de HostLo et réduire le coût pour l'utilisateur.

Celle-ci est très simple. Il s'agit dans un premier temps de laisser Kubernetes déterminer un placement de manière classique, c'est-à-dire de placer le pod entier sur une VM déjà achetée. La meilleure VM selon Kubernetes, lorsque l'on utilise la politique « utilisation maximale » [66], est celle qui a actuellement le plus de ressources déjà allouées à des pods, autrement dit celle qui est la plus utilisée. [3] Autrement dit, c'est une stratégie de groupement. Si Kubernetes ne trouve pas de VM pouvant héberger ce pod entier, il prévoit d'acheter la VM la moins chère qui permet d'héberger le pod.

Ensuite, ce placement est optimisé par rapport à la capacité de HostLo : les conteneurs du pod sont déplacés sur les VM qui ont le plus de ressources gâchées, en commençant par les plus petits. Le principe est d'essayer d'éliminer le gaspillage, et réduire le nombre de VM ou leur taille. En effet, si Kubernetes a prévu d'acheter une VM, cette passe d'optimisation peut permettre de réduire sa taille, et donc d'en acheter une moins chère.

Cette mise à jour prototype reste simple, mais son évaluation en section 5.4.6 montre déjà des résultats satisfaisants.

5.4.5. Gestion des autres ressources

HostLo cible le problème du réseautage interne au pod pour permettre le déploiement inter-VM. Pourtant, les conteneurs d'un pod ont aussi besoin de *partager des volumes disques, ainsi que de communiquer par mémoire partagée*, comme décrit en section 5.2.1. Par conséquent, le déploiement inter-VM de pods nécessite en fait plus qu'une interface `localhost` inter-VM efficace comme HostLo. Heureusement, les problèmes des volumes et de la mémoire partagés ont déjà été résolus par des travaux présentés ci-dessous.

Volumes disques

On rappelle ici qu'un volume disque est une section limitée du système de fichiers de l'hôte. Elle peut être montée dans un pod, et sera ainsi partagée

2. En fait, Kubernetes n'a pas de comportement particulier dans un environnement de virtualisation imbriquée : il ne différencie par les serveurs physiques des VM pour héberger ses pods.

3. Le taux d'utilisation d'une VM est déterminé par la moyenne des sommes des requêtes en ressource processeur, et des requêtes en mémoire, relatives aux ressources totales de la VM.

entre tous les conteneurs du pod. La principale difficulté est que la lecture et l'écriture dans ce volume est gérée par le SE hôte du pod. Dans le cas d'un pod inter-VM, il y a plus d'un SE impliqué. C'est un problème parce que les SE des différentes VM ne sont pas conçus pour partager un système de fichiers avec un autre SE. Cela créerait des incohérences dans le système de fichiers parce que les SE maintiennent son état dans leurs mémoires, ainsi que les caches. On voit donc que simplement monter le système de fichier du volume du pod dans les différentes VM n'est pas une solution.

Jujjuri et al. [61] ont développé VirtFS, un système de fichiers paravirtualisé (voir section 2.2.2) pour QEMU/KVM, basé sur VirtIO. Il est capable entre autres, d'être monté plusieurs fois dans différentes VM, grâce au fait qu'il est géré par QEMU via la paravirtualisation plutôt que par les SE invités. Ainsi, il suffit de modifier l'orchestrateur pour communiquer avec le gestionnaire de VM afin de monter dans les VM avec VirtFS, le disque qui contiendra le volume du pod ; et d'ensuite utiliser ce volume virtuel dans le pod inter-VM.

Mémoire partagée

Bien souvent, pour la communication entre les conteneurs d'un même pod, c'est la mémoire partagée qui est utilisée. Il s'agit de mettre en commun une région mémoire entre les conteneurs, afin d'échanger des données en suivant un protocole adapté, comme présenté en section 5.2.1. Le partage de la mémoire entre des VM est une problématique ancienne [115]. Des solutions existent, et si elles peuvent être utilisées au niveau des VM de manière transparente du point de vue des applications, alors elles sont applicables aux pods. C'est parce que les pods ne modifient pas la gestion de la mémoire, ce ne sont que des groupes logiques de processus.

Ainsi, Ren et al. [115] propose une étude comparative très complète de toutes les techniques de mémoire partagée entre des VM. La mieux adaptée à la problématique du déploiement inter-VM des pods est MemPipe [144]. En effet, MemPipe implémente la mémoire partagée entre des VM KVM au niveau de la couche de transport ; autrement dit, d'une manière qui est transparente pour les application conteneurisées du pod.

5.4.6. Évaluation

Le but de HostLo est de corriger le défaut de la virtualisation imbriquée qui contraint à déployer un pod entièrement dans une VM. HostLo autorise donc le déploiement des conteneurs d'un pod sur plusieurs VM, de manière

TABLE 5.1. – Tarifs des VM `m5` d'Amazon Web Services EC2.

Modèle	vCPU	Mém. (Gio)	vCPU (rel.)	Mém. (rel.)	Tarif
`large`	2	8	0.0208	0.0208	0,112 $/h
`xlarge`	4	16	0.0417	0.0417	0,224 $/h
`2xlarge`	8	32	0.0833	0.0833	0,448 $/h
`4xlarge`	16	64	0.1667	0.1667	0,896 $/h
`12xlarge`	48	192	0.5	0.5	2,688 $/h
`24xlarge`	96	384	1	1	5,376 $/h

désagrégée. Cette désagrégation permet *d'améliorer l'utilisation des ressources* en permettant des placements à une granularité plus fine (celle des conteneurs, plutôt que celle des pods).

Cependant, ce déploiement inter-VM implique potentiellement une dégradation des performances réseau entre les conteneurs du pod. En effet, au lieu d'être liés directement par une interface `localhost` extrêmement rapide, ils communiqueront alors par une connexion à travers les VM.

Enfin, comme pour BrFusion, on évalue l'utilisation du processeur.

Économies financières

En permettant de déployer un pod à cheval sur plusieurs VM, l'orchestrateur peut faire des allocations à une granularité plus fine. Cela réduit la fragmentation, et permet d'économiser des ressources. Cette économie se traduit en un nombre réduit de VM, et aussi en une réduction de la taille des VM nécessaires à l'hébergement des pods. En conclusion, HostLo permet de réduire le coût d'utilisation de la virtualisation imbriquée.

Méthodologie

Pour évaluer ces économies, on simule pour chaque utilisateur dans les traces du centre d'hébergement de Google [114], le coût total d'hébergement de ses pods sur des VM. Pour ce faire, on utilise les spécifications et les tarifs des VM de Amazon Web Service EC2, plus précisément les modèles « à la demande » `m5` [40]. Le tableau 5.1 reproduit ces informations. On remarquera aussi les spécifications en ressources en valeurs relatives au plus gros modèle de VM, `24xlarge`, pour correspondre aux informations de ressources dans les traces de Google.

Le code source la simulation est donnée en annexe E. Elle consiste à placer les pods efficacement, et à créer de nouvelles VM aux bonnes

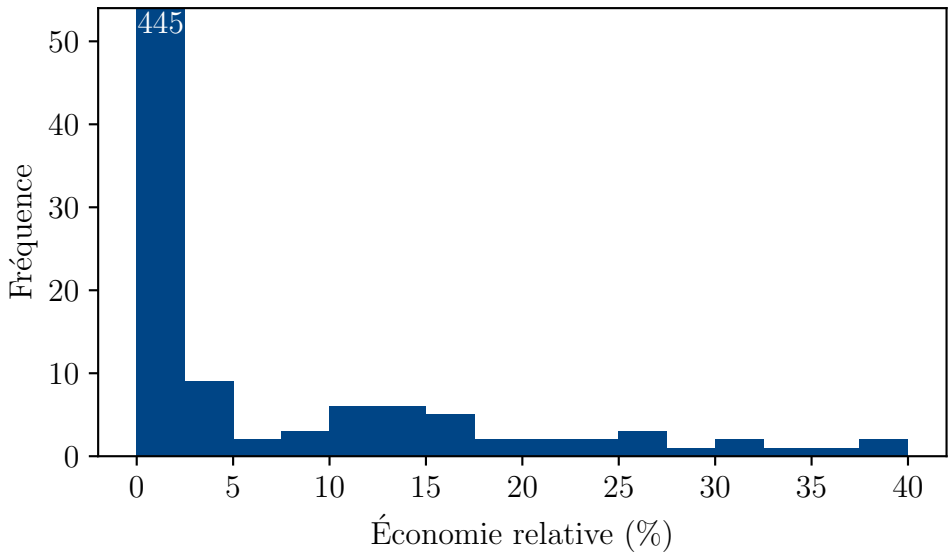


FIGURE 5.7. – Économies financières de HostLo : fréquence des économies par utilisateur, relativement à un placement par Kubernetes.

dimensions si nécessaire. On compare Kubernetes et son algorithme de placement classique, qui ne peut pas déployer un pod à cheval sur plusieurs VM ; et HostLo avec la désagrégation des pods, dont l'algorithme est décrit plus haut en section 5.4.4. Dans le cadre de la simulation, on demande à l'orchestrateur de placer tous les pods d'un utilisateur les uns après les autres, en commençant par le plus gros. [4]

Résultats

Les résultats de la simulation sont reproduits en figure 5.7. Ce graphique montre les fréquences des économies financières réalisées pour chacun des 492 utilisateurs dans les traces Google. Les économies sont relatives au coût des VM pour chaque utilisateur avec un placement fait par Kubernetes, sans HostLo.

On observe des réductions de coût pour environ 11,4% des utilisateurs. Parmi ceux-ci, environ 66,7% bénéficient d'une réduction de plus de 5%. L'économie relative la plus importante est d'environ 40%. La réduction du coût la plus grande est d'environ 237 $/h, ce qui correspond pour cet utilisateur à une réduction de 35%.

4. La taille d'un pod est définie comme la somme des valeurs relatives de ses requêtes en processeur et en mémoire.

Performances réseau

Bien que HostLo apporte des réductions non négligeables du coût d'utilisation du cloud, la désagrégation d'un pod provoque une certaine dégradation des performances réseau que l'on quantifie ici.

On compare les performances réseau de HostLo à trois autres solutions :

1. NAT : voir la description pour BrFusion en section 5.5.4 ;
2. surcouche : la solution de réseau en surcouche de Docker, actuellement la seule alternative viable pour la désagrégation d'un pod ;
3. groupé : tous les conteneurs d'un pod s'exécutent dans la même VM, et communiquent donc par leur interface `localhost` privée.

Parmi ces solutions, groupé est l'étalon. Cependant, contrairement à BrFusion, HostLo ne vise pas le niveau de performance de cet étalon. Il faut seulement des performances acceptables par rapport aux gains apportés par la solution. En particulier, HostLo concerne les communications entre conteneurs d'un même pod. Celles-ci sont généralement composées de petits messages tels que des notifications d'évènements pour la journalisation et la surveillance. Par conséquent, la latence est la métrique de premier ordre.

Environnement Le même environnement d'évaluation que pour BrFusion est utilisé. Il est décrit en section 5.5.4.

Installation Les bancs d'essai utilisés pour l'évaluation de HostLo sont similaires à ceux de BrFusion, et sont des applications de type client-serveur. La partie serveur et la partie client d'un banc d'essai sont placées dans leurs propres conteneurs, dans deux VM différentes ; sauf bien sûr pour la situation nominale où les deux sont placées dans la même VM et communiquent par `localhost`. Les installations des quatre solutions comparées ici sont décrites en figure 5.8.

Méthodologie

L'évaluation des performances de HostLo est similaire à celle de BrFusion, avec des installations différentes, illustrées ci-dessus.

Notamment, on examine le débit et la latence avec le même micro-banc d'essai, Netperf (voir section 5.5.4). On réutilise aussi le macro-banc d'essai NGINX, mais Kafka est ici remplacé par Memcached, un registre clé-valeur. (en anglais *key-value store*). Le tableau 5.2 donne les paramètres spécifiques à ces bancs d'essai ainsi que leurs métriques.

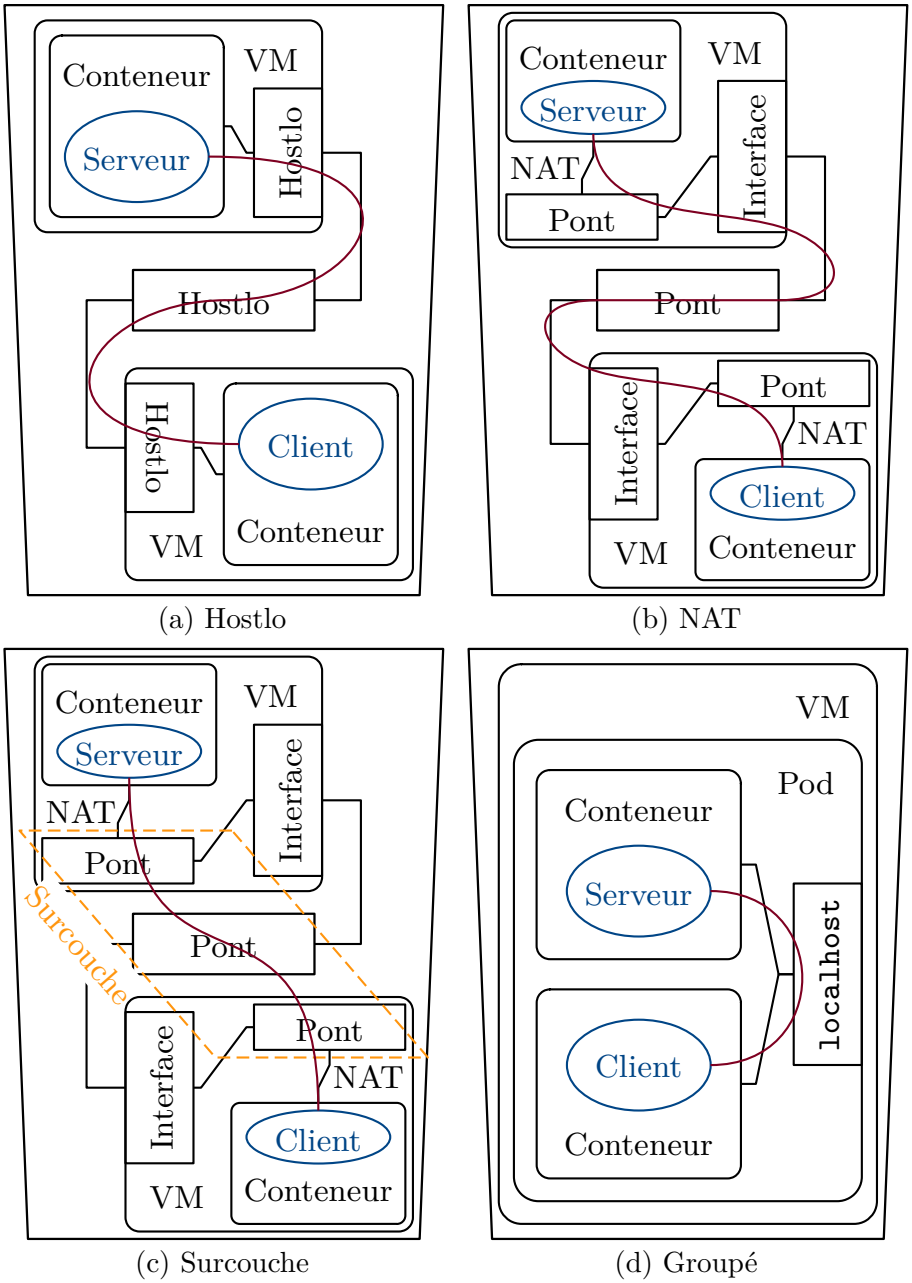


FIGURE 5.8. – Installations des expériences d'évaluation de HostLo, NAT, surcouche et groupé.

TABLE 5.2. – Macro-bancs d'essai pour l'évaluation comparative de BrFusion avec NAT et natif.

Application, banc d'essai	Paramètres	Métrique
NGINX [94], wrk2 [138]	2 fils d'exécution, 50 conn/fil, 10 000 req/s sur fichier de 1 ko	latence
Memcached [84], memtier_benchmark [85]	4 fils d'exécution, 50 conn/fil, ratio `SET` : `GET` de 1:10	rép/s, latence

Résultats : micro-banc d'essai Netperf

Les résultats avec le micro-banc d'essai Netperf sont reproduits en figure 5.9. Tout d'abord, en ce qui concerne le débit, celui de HostLo augmente avec la taille des messages aussi bien qu'avec NAT et la surcouche. Ces deux derniers montrent cependant des pics de performance inattendus avec certaines tailles de messages. Comme attendu cependant, aucune de ces trois solutions n'atteint le niveau de performance en groupé. À titre d'exemple, avec une taille de message de 1024 o, le débit de HostLo est 17,9% plus grand que celui de NAT, 27% plus petit que celui de la surcouche, et 5,3 fois plus petit que celui en groupé.

Malgré une performance en débit moyenne, on relève des résultats prometteurs en latence pour HostLo. Elle reste stable indépendamment de la taille des messages, comme en groupé, bien qu'elle soit deux fois plus grande. Concernant NAT et la surcouche, leurs latences varient énormément et de manière inattendue. Par exemple, avec des messages de 1024 o, la latence de HostLo est 87,3% plus faible que celle de NAT, et 89,8% plus faible que celle de la surcouche. L'écart type de la latence de HostLo reste aussi plutôt bas, à environ 27,9% de la latence moyenne, à comparer avec l'écart type de 20,5% de la latence moyenne en groupé. Quant à NAT et la surcouche, ils montrent des écarts types allant de 25,8% à 95,4%.

Résultats : macro-bancs d'essai

Les résultats des macro-bancs d'essai sont visibles en figures 5.10 à 5.12.

Concernant les résultats avec Memcached, HostLo atteint de manière inattendue les niveaux de débit et de latence observés en groupé. Cela s'explique en partie par le fait que les résultats en groupé montrent une grande variabilité de latence, visible en figure 5.11. À l'inverse, les requêtes passées avec HostLo ont une latence stable. Quand aux résultats avec

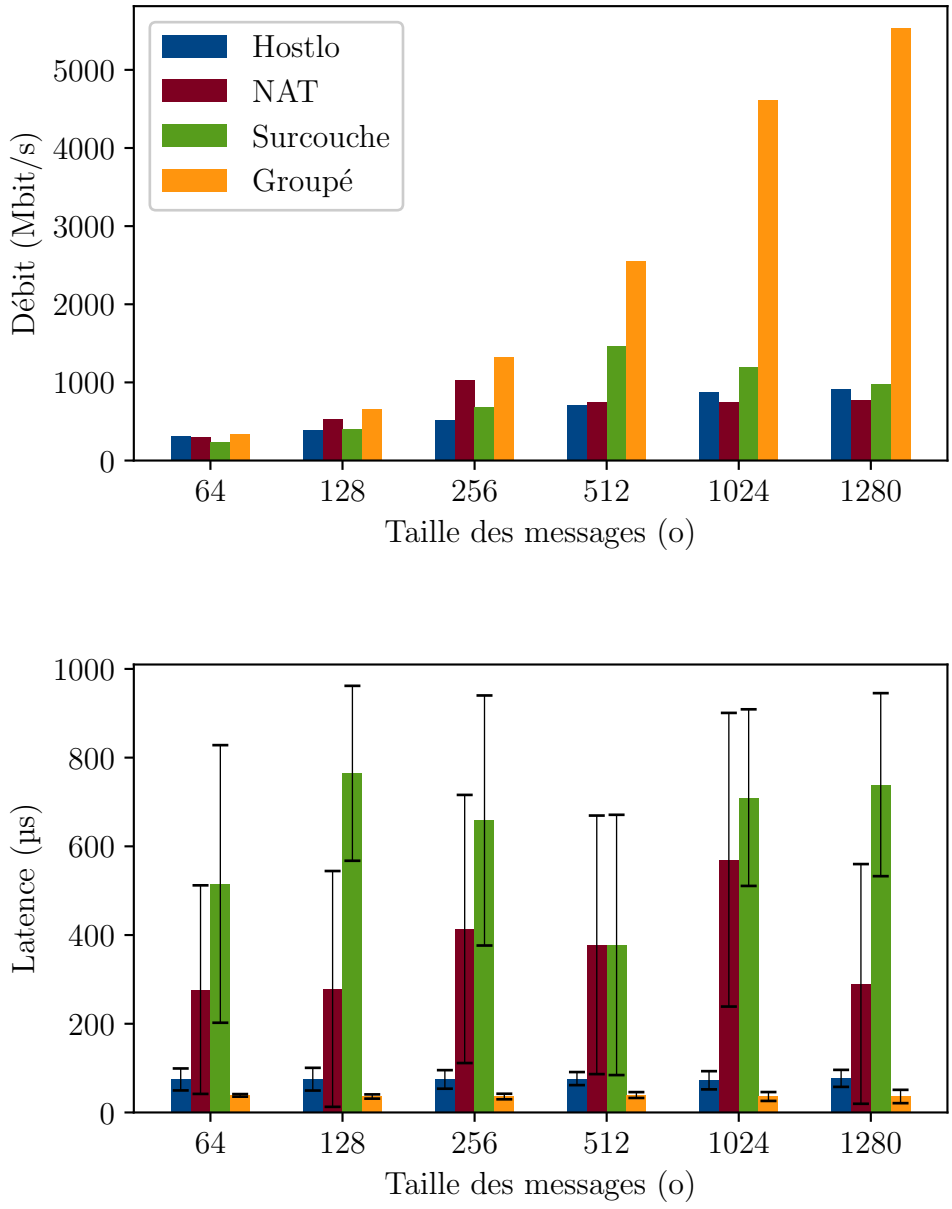


FIGURE 5.9. – Performance réseau de HostLo, NAT, surcouche et groupé, par Netperf. Les barres donnent l'écart type de la latence.

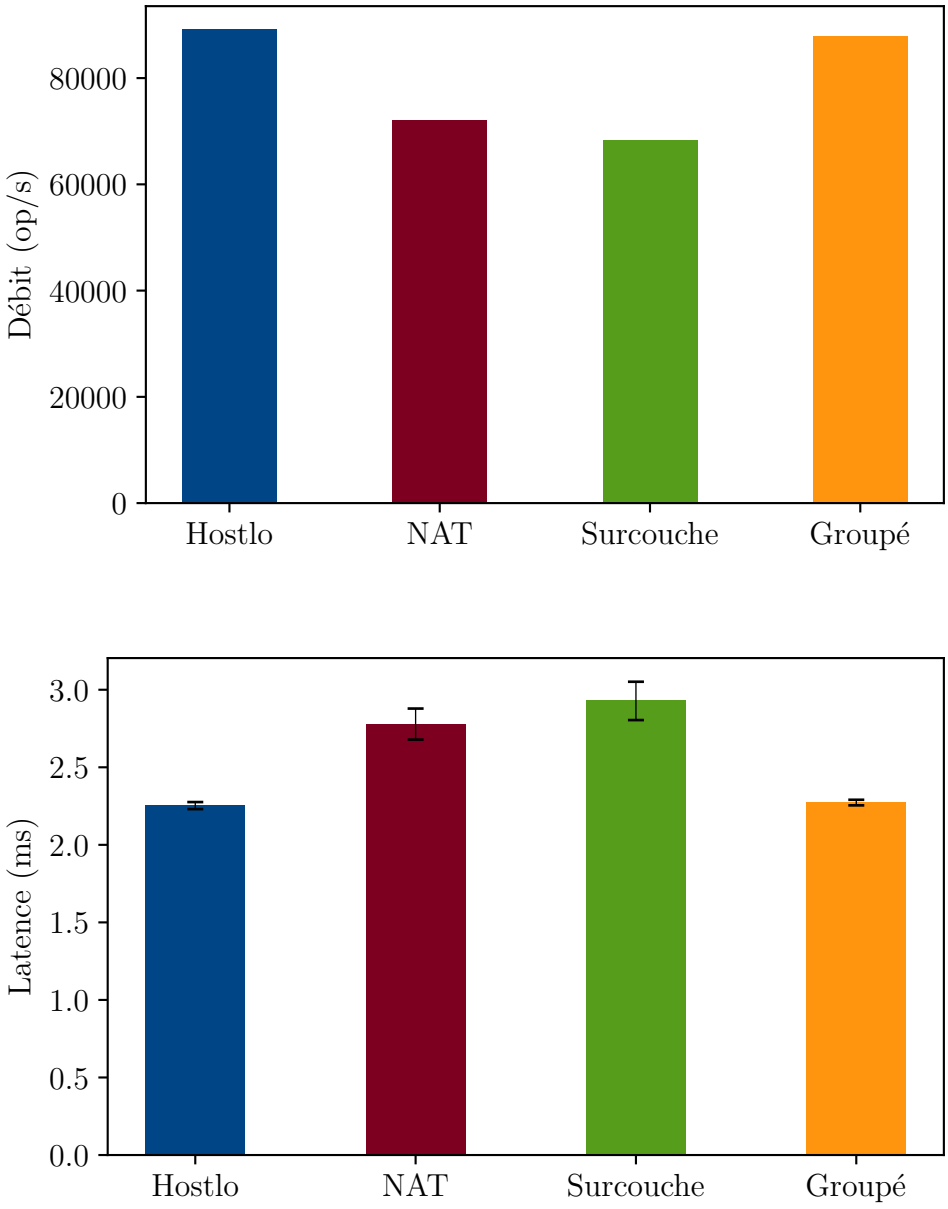
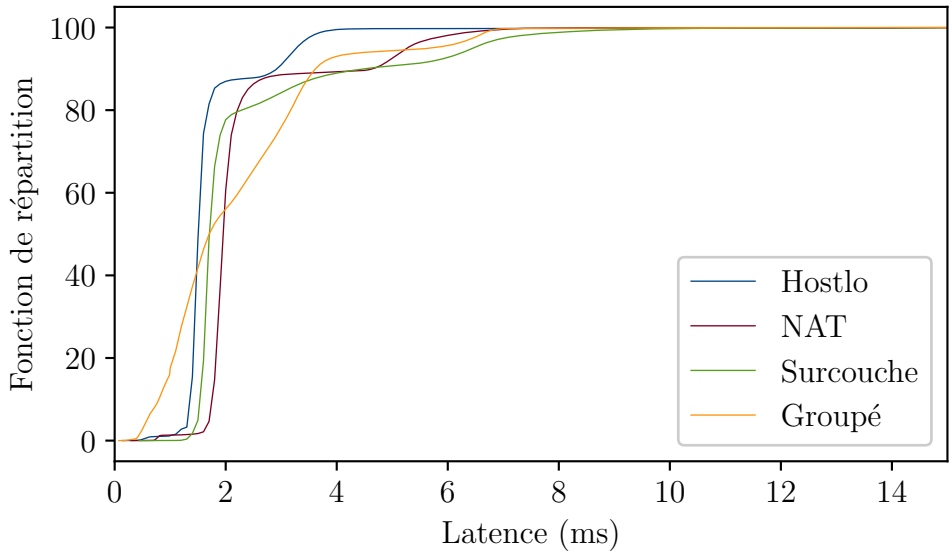
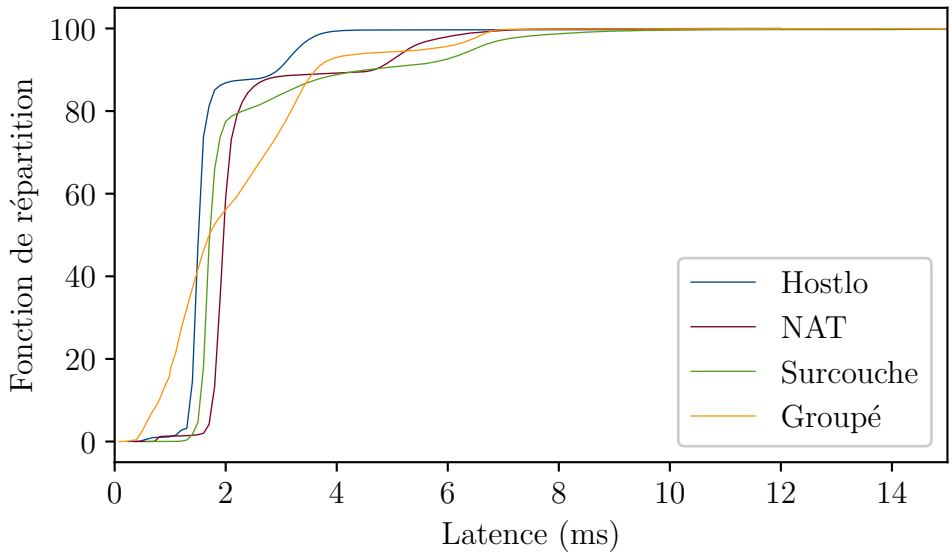


FIGURE 5.10. – Performance réseau de HostLo avec NAT, surcouche et groupé, par Memcached.



(a) GET



(b) SET

FIGURE 5.11. – Performance réseau de HostLo avec NAT, surcouche et groupé, par Memcached (distribution des latences pour les opérations `GET` et `SET`).

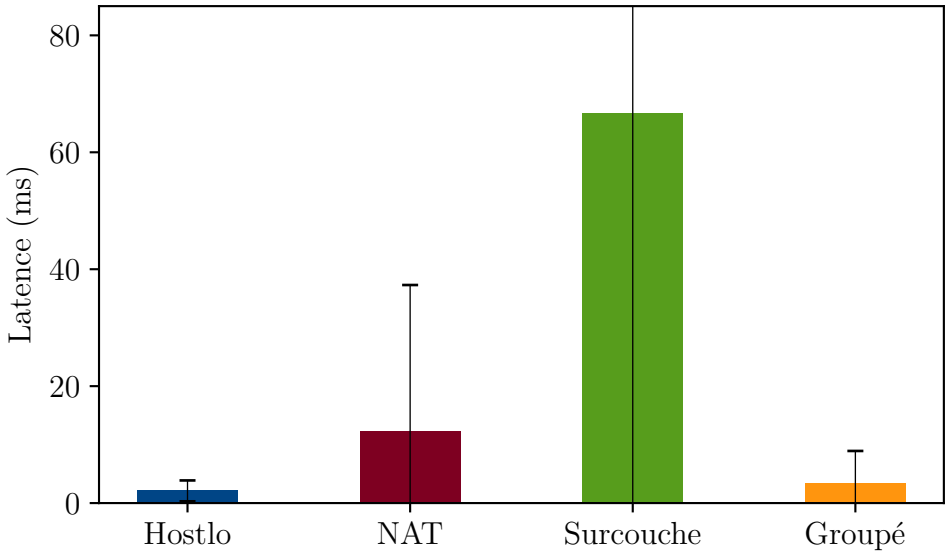


FIGURE 5.12. – Performance réseau de HostLo avec NAT, surcouche et groupé, par NGINX.

NGINX, HostLo produit une latence supérieure de 49,4% à celle en groupé, mais qui reste bien plus faible que celle de NAT ou de la surcouche. On remarque par ailleurs un écart type très grand pour les quatre solutions.

Utilisation du processeur

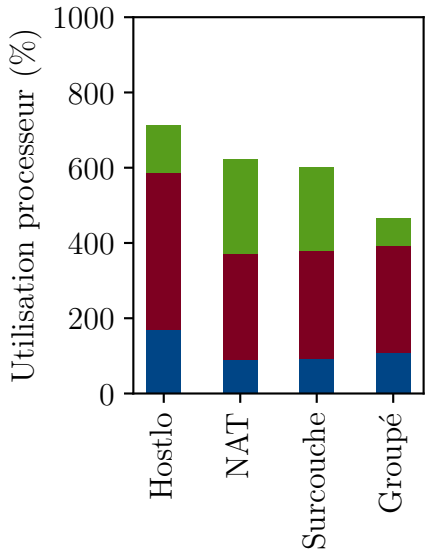
Méthodologie

Cette expérience est en tout point similaire à celle réalisée pour BrFusion, décrite en section 5.5.4.

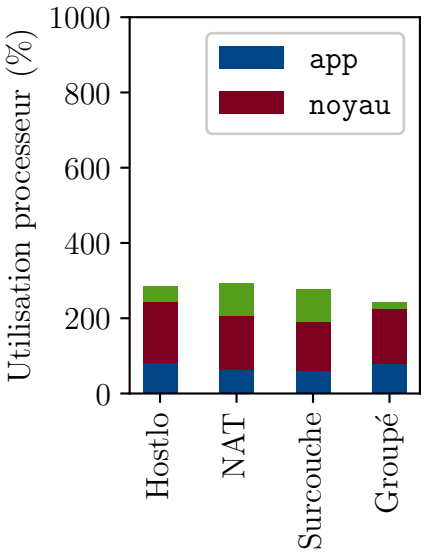
Résultats

Les résultats d'utilisation du processeur par le banc d'essai Memcached sont donnés en figures 5.13a et 5.13b, et par NGINX en figures 5.13c et 5.13d.

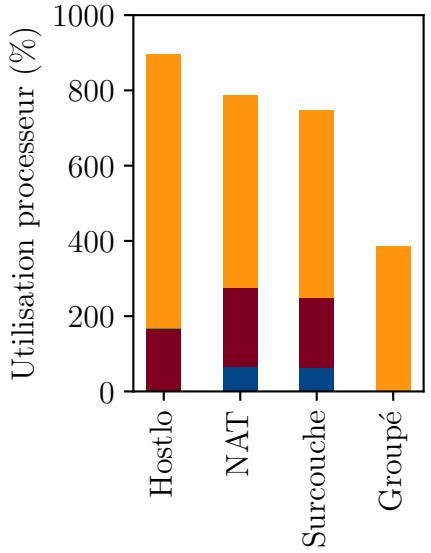
Concernant Memcached, l'augmentation principale de l'utilisation processeur est visible au niveau des utilisations par les noyaux des SE invités qui exécutent le serveur et le banc d'essai. Elle est de 46,7% par rapport au mode groupé. L'utilisation totale du processeur par le client et le serveur augmente de 53,2%. Depuis le serveur hôte, le temps processeur alloué aux deux VM augmente de 89,8%. On notera cependant que par nature, il n'y a en groupé qu'une seule VM tandis que les trois autres solutions en né-



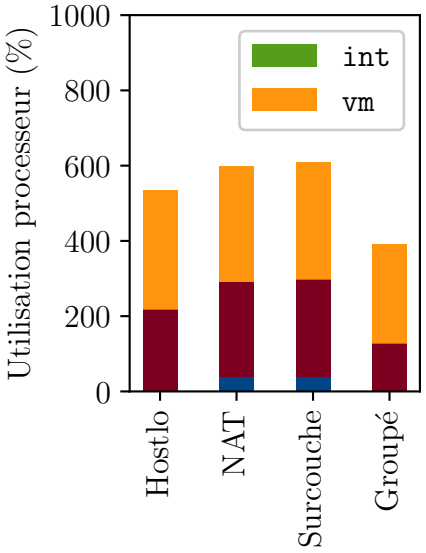
(a) Banc d'essai et serveur Memcached



(c) Banc d'essai et serveur NGINX



(b) VM du banc d'essai et VM du serveur Memcached dans l'hôte



(d) VM du banc d'essai et VM du serveur NGINX dans l'hôte

FIGURE 5.13. – Utilisations du processeur pour les bancs d'essai Memcached (gauche) et NGINX (droite).
Haut : somme des utilisations processeur du banc d'essai et du serveur dans leurs VM respectives. Bas : somme des utilisations processeur de leurs deux VM. En groupé, il n'y a qu'une seule VM.

cessitent obligatoirement deux. Cette augmentation de la consommation processeur est donc largement attendue du fait de ce second SE invité.

On observe par ailleurs avec HostLo, qu'une quantité de temps processeur (environ 1,68 cœurs) est utilisée par le noyau du serveur hôte pour le compte des VM (source `noyau`). C'est cependant aussi le cas avec NAT et la surcouche, en quantité comparable. On en conclut que ce temps processeur provient de la gestion par ce noyau, des paquets qui transitent par les interfaces virtuelles des VM et qui sont fournies avec Vhost (voir la description de l'environnement en section 5.5.4). En effet, Vhost prend la forme d'un composant du noyau, utilisé par KVM pour accélérer les entrées-sorties des VM en les traitant directement dans le noyau plutôt que de laisser QEMU, en espace utilisateur, les traiter avec un périphérique virtio classique. Ainsi, puisqu'il n'y a pas de différence entre HostLo et les deux autres solutions, on conclut que l'utilisation processeur du noyau hôte spécifiquement pour fournir HostLo est correctement attribuée aux VM.

Enfin pour NGINX, l'augmentation de l'utilisation processeur de Hostlo par rapport à l'étalon en groupé est réduite : 17,1% pour le client et le serveur, et 36,9% en ce qui concerne l'utilisation processeur des VM. On retrouve par ailleurs les mêmes observations que pour Memcached.

Résumé de l'évaluation et discussion

Pour conclure sur les avantages et les inconvénients de HostLo, cette solution permet de réaliser des économies financières substantielles pour le client de la virtualisation imbriquée. L'implémentation de HostLo vient au prix d'une certaine augmentation de l'utilisation processeur, ainsi que d'une dégradation des performances réseau entre les conteneurs d'un pod, qui est inévitable. Il convient donc de prendre en compte tous ces paramètres lors du placement d'un pod, pour décider s'il est envisageable de le désagréger.

L'implémentation actuelle de HostLo montre des performances satisfaisantes dans le contexte où cette interface est utilisée, mais qui restent cependant faibles. Il serait pertinent d'explorer d'autres implémentations pour la communication inter-VM entre les conteneurs d'un pod. Par exemple, DPDK [119] (pour *Data Plane Development Kit*, c'est-à-dire « trousse de développement du niveau données ») est prouvé très efficace pour des implémentations réseau dans l'espace utilisateur d'un SE. Il pourrait être épaulé par vhost-user [38], qui permet de traiter des paquets provenant d'invités virtuels de manière optimisée sans jamais passer par le noyau du SE.

Par ailleurs, l'implémentation de HostLo est actuellement *limitée à un serveur physique*. Cela va à l'encontre d'une fonctionnalité majeure de la

virtualisation niveau matériel : la migration des VM (voir section 2.2.4). L'implémentation actuelle permet certes d'externaliser le trafic qui traverse une interface HostLo pour l'envoyer à un autre serveur physique. Cependant, la dégradation des performances du réseau deviendra inacceptable. On peut alors imaginer que l'orchestrateur indique à l'hyperviseur quelles VM vont de pair et doivent être migrées ensemble, ou alors ne peuvent pas être migrées. Une solution au niveau de l'orchestrateur est aussi de migrer des pods soit vers un autre serveur physique, pour éviter le besoin de migrer ; ou vers une autre VM pour relâcher le couplage entre les VM et autoriser la migration. Migrer des pods est possible avec du *checkpoint and restart* (c'est-à-dire « point de contrôle et reprise ») [1]. Néanmoins, dans une architecture en micro-services, les conteneurs sont généralement sans état. Ainsi, leur « migration » revient à les détruire, et à les redéployer à la destination. Pour résumer, adapter HostLo à la migration des VM requiert un certain niveau de coopération entre l'orchestrateur et l'hyperviseur.

En plus de l'implémentation de l'interface réseau spéciale de HostLo, *la mise à jour de l'algorithme de placement de l'orchestrateur* est importante pour obtenir les économies financières permises par les déploiements inter-VM de pods. L'algorithme de la section 5.4.4, évalué plus haut, n'a pas permis de réaliser des économies pour la majeure partie des utilisateurs. Il reste donc du travail à ce sujet.

5.5. BrFusion : déduplication de la virtualisation du réseau

Cette section décrit BrFusion, qui est la correction du premier défaut de la virtualisation imbriquée : la duplication de la virtualisation du réseau.

5.5.1. Vue d'ensemble

On a vu que l'interface réseau du serveur hôte était déjà virtualisée par le pont dans celui-ci. Cette situation de départ est illustrée en figure 5.14. Par conséquent, *le pont dans la VM, qui sert à re-virtualiser son interface réseau, est en réalité inutile.* Le principe de BrFusion est donc d'éliminer le pont dans la VM, en provisionnant une interface réseau pour chaque pod.

Concrètement, lorsque le pod est créé dans la VM, *une nouvelle interface réseau virtuelle est provisionnée* par le gestionnaire de VM. Cette interface est insérée dans la VM pour l'usage exclusif du pod. Cela permet d'inclure

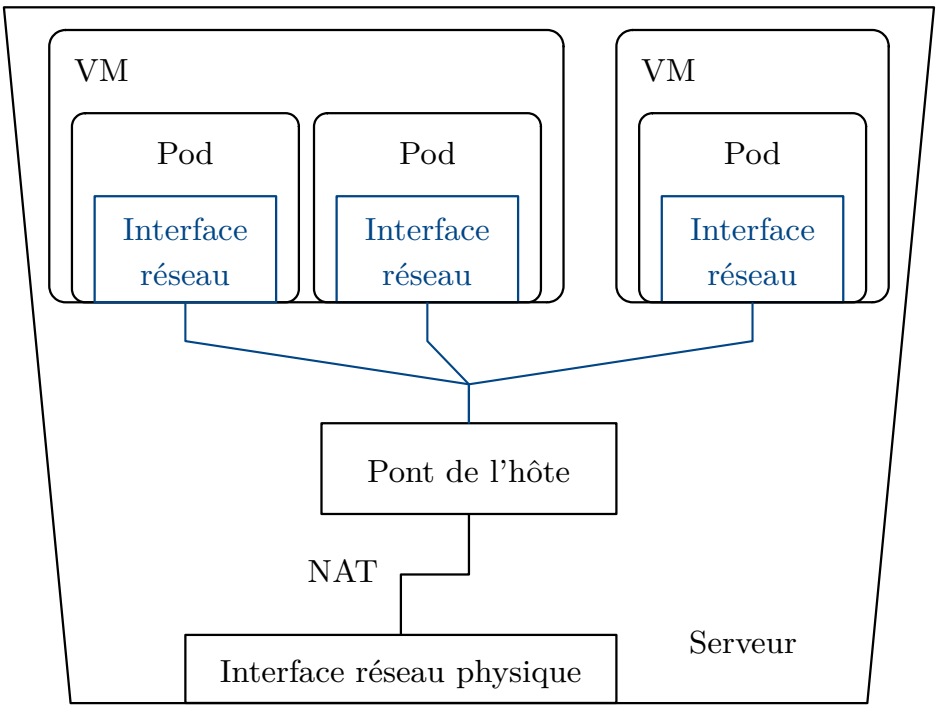


FIGURE 5.14. – Illustration de BrFusion. Comparer avec la virtualisation imbriquée du réseau en figure 5.3.

la nouvelle interface directement dans celui-ci. Il n'est ainsi plus nécessaire d'utiliser NAT et un pont pour virtualiser le réseau pour le pod.

5.5.2. Intégration

L'interface virtuelle dédiée au pod est donc administrée par le gestionnaire de VM. Il l'ajoute à un pont du serveur hôte pour fournir la connexion au pod. Selon la politique d'administration, ce peut être un pont commun à toutes les VM, ou un pont provisionné pour le propriétaire de la VM, etc. Dans tous les cas, la configuration de l'interface virtuelle reste la même que la configuration actuelle, c'est-à-dire basée sur NAT dans le serveur.

En fait, la solution BrFusion repose sur la *communication* entre l'orchestrateur, qui demande la provision d'une interface réseau pour un nouveau pod ; et le gestionnaire de VM, qui contrôle véritablement les ressources et fournit donc cette interface. Voici comment peut se dérouler cet échange :

1. l'orchestrateur demande au gestionnaire de VM d'ajouter une nouvelle interface à la VM choisie pour déployer le pod, il spécifie aussi éventuellement un domaine réseau au niveau hôte (c'est-à-dire un pont) qui possèdera la nouvelle interface ;
2. le gestionnaire de VM ajoute la nouvelle interface à la VM et la configure comme demandé ;
3. puis il retourne à l'orchestrateur une identification de la nouvelle interface dans la VM (par exemple, l'adresse Ethernet MAC), pour que l'agent local de ce dernier l'utilise pour le nouveau pod ;
4. finalement, l'orchestrateur via son agent local dans la VM, y configure l'interface réseau et l'alloue au pod.

Une illustration de cette séquence est donnée en figure 5.15. Elle reste générale parce que les séquences pour l'intégration de BrFusion et pour celle de HostLo (voir section 5.4.2) sont similaires.

5.5.3. Implémentation

On propose une implémentation prototype de BrFusion dans QEMU. [5] À la création d'une VM, QEMU propose aussi une interface de gestion parallèle appelée QMP (pour *QEMU Machine Protocol*, soit « protocole

5. La virtualisation est généralement réalisée par le couple QEMU et KVM, le dernier étant utilisé par le premier pour optimiser certains aspects avec de la virtualisation aidée par le matériel. L'implémentation de BrFusion ne concerne que QEMU.

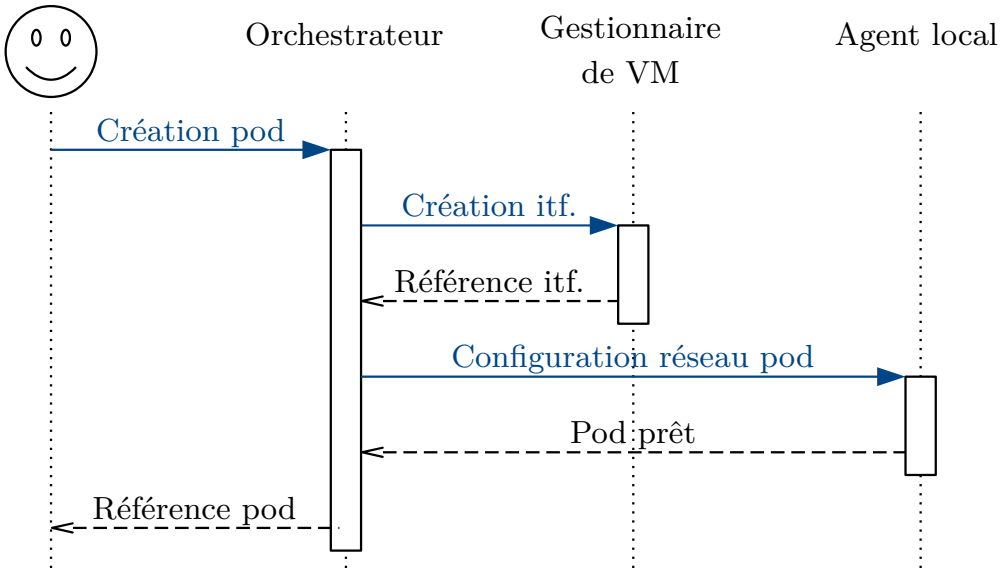


FIGURE 5.15. – Séquence d'échange entre l'orchestrateur, son agent local dans une VM, et le gestionnaire de VM pour l'intégration de BrFusion et de HostLo.

machine QEMU ») [37]. En particulier, on peut obtenir un socket UNIX connecté à cette interface, que le gestionnaire de VM va utiliser pour exécuter des opérations telles que l'ajout ou le retrait à une VM d'interfaces réseau virtuelles. Il est ainsi possible d'obtenir une interface réseau en tout point identique à une interface provisionnée lors de la création de la VM. Le SE invité va ensuite configurer ce nouveau périphérique ajouté à chaud.

Il faut ensuite étendre l'orchestrateur, par exemple Kubernetes, pour qu'il communique avec QEMU. Dans ce cas, la tâche est rendue aisée par l'implémentation d'un greffon CNI [30] (pour *Container Network Interface*, soit « interface réseau pour les conteneurs »). De manière générale, les greffons CNI sont une spécification standardisée pour implémenter de nouveaux modèles de réseautage dans Kubernetes.

5.5.4. Évaluation

BrFusion agit sur le chemin des paquets entre un pod et l'extérieur de la VM qui l'héberge. Son but premier est de *corriger la dégradation des performances réseau*.

On examine aussi la façon dont cette simplification du chemin réseau

impacte l'utilisation du processeur par le pod et la VM. En effet, en éliminant une couche de virtualisation du réseau dans la VM, on s'attend à ce que BrFusion réduise la quantité de processeur utilisée par la VM pour le réseautage de ses pods. Cela se traduirait par la libération de temps processeur pour les autres applications dans la VM.

Enfin, un impact négatif potentiel de BrFusion est le *ralentissement de la création d'un pod*, à cause de la provision dynamique d'une interface réseau.

Environnement Le serveur physique utilisé pour les expériences est une machine Dell® avec 12 cœurs répartis en NUMA sur deux processeurs Intel®Xeon® E5-2420 v2 cadencés à 2,20 GHz (fréquence fixe sous le gouverneur `performance`), sans SMT (pour *Simultaneous multithreading*, c'est-à-dire les fils d'exécution matériels). Les VM sont toujours provisionnées avec 5 vCPU et 4 Go de mémoire, avec QEMU et son accélérateur KVM. Concernant le réseau, leurs interfaces sont para-virtualisées avec virtio [116], et utilisent Vhost [136] en arrière-boutique. Les SE invités sont tous identiques, et exécutent la distribution Arch Linux avec un noyau en version 4.19.9. Le moteur de conteneurs est Docker 18.09.0-CE.

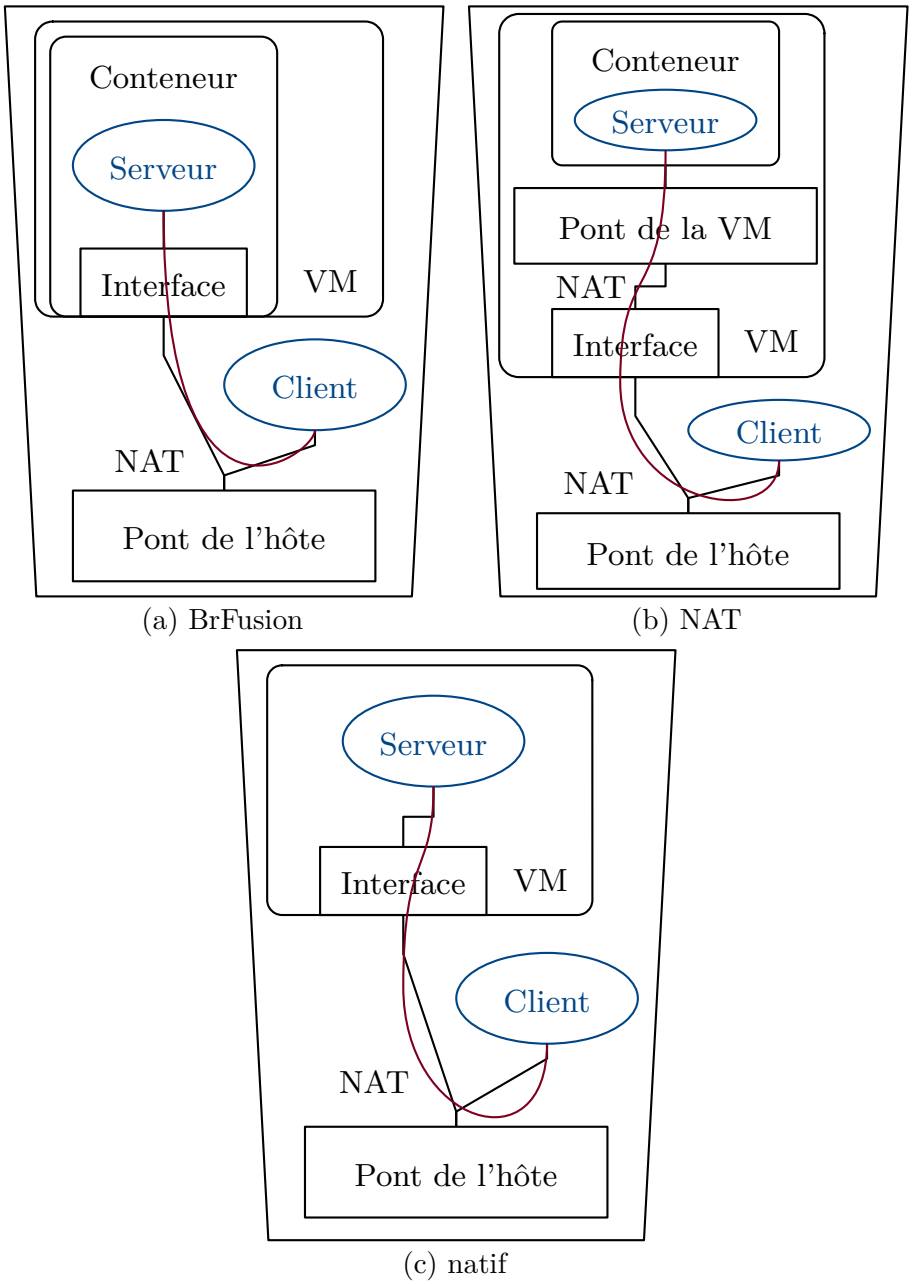
Installation Toutes les expériences utilisent des applications de type client-serveur. La partie serveur du banc d'essai est placée dans une VM. Sa partie client s'exécute sur des processeurs du serveur physique différents de ceux de la VM. Elle est reliée au pont de l'hôte, et donc à la VM, par NAT. Les installations pour les trois solutions sont décrites en figure 5.16.

Performances réseau

Tout d'abord concernant la correction des performances réseau, BrFusion est comparé à deux autres solutions :

1. NAT : il s'agit de la solution de virtualisation du réseau par défaut, basée sur l'empilement de deux ponts avec du routage NAT (voir section 5.2.2) ;
2. natif : pas de virtualisation imbriquée, c'est-à-dire que l'application s'exécute nativement dans la VM, sans conteneurisation.

Parmi ces deux solutions, natif est l'étalon. C'est son niveau de performance qui est visé par BrFusion.



(a) BrFusion (b) NAT

(c) natif

FIGURE 5.16. – Installations des expériences pour l'évaluation de BrFusion.

Table 5.3. – Macro-bancs d'essai pour l'évaluation comparative de BrFusion avec NAT et natif.

Application, banc d'essai	Paramètres	Métrique
NGINX [94], wrk2 [138]	2 fils d'exécution, 50 conn/fil, 10 000 req/s sur fichier de 1 ko	latence
Kafka [5], kafka-producer-per-test.sh	120 000 msg/s, 100 o/msg, 8192 o/lot	latence

Méthodologie

Les trois solutions (BrFusion, NAT et natif) sont évaluées de manière similaire (voir ci-dessous pour l'installation des expériences). On examine leur débit et leur latence réseau, grâce à différents bancs d'essai.

Le premier est Netperf [51], un micro-banc d'essai. Il fonctionne de la façon suivante. Le client établit une connexion de contrôle vers le serveur. Il prépare ensuite une connexion pour les données, qui sera mesurée pour obtenir les résultats de l'expérience. On utilise deux de ses modes de fonctionnement : `UDP_RR` pour évaluer la latence, et `TCP_STREAM` pour évaluer le débit. `UDP_RR` consiste à mesurer le temps pour envoyer une requête et recevoir sa réponse, en effectuant des transactions synchrones ; c'est-à-dire qu'il n'y a qu'une seule requête à la fois sur la connexion. La métrique ainsi obtenue est la latence de requête moyenne. Pour ce qui est de `TCP_STREAM`, il envoie autant de données que possible pendant une durée définie (ici, 20 s). On obtient ici comme métrique, le débit moyen. En évaluant chaque solution, on fait varier la taille des messages.

Ensuite, on expérimente à l'aide des macro-bancs d'essai listés dans le tableau 5.3. Ce dernier indique aussi les paramètres d'évaluation spécifiques à chaque benchmark, ainsi que leurs métriques.

Résultats : micro-banc d'essai Netperf

Les résultats avec le micro-banc d'essai Netperf sont reproduits en figure 5.17. Pour le débit comme pour la latence, BrFusion montre un niveau de performance similaire à la configuration native. Autrement dit, BrFusion élimine effectivement la dégradation des performances réseau due à l'imbrication de la virtualisation. Par exemple, avec des paquets de 1280 o, le débit de BrFusion est 2,1 fois plus grand que celui de NAT ; la latence moyenne est aussi 18,4% plus faible ; et on observe une différence de 3,5%

entre les performances de BrFusion et du natif. Enfin, on remarque que les performances de BrFusion, comme celles du natif, augmentent avec la taille des messages, tandis que celles de NAT augmentent plus lentement et stagnent entre 1024 o et 1280 o.

Résultats : macro-bancs d'essai

Les résultats avec les macro-bancs d'essai sont reproduits en figure 5.18. Tout d'abord pour Kafka, on observe une amélioration de la latence moyenne des requêtes de 11,8% avec BrFusion par rapport à NAT. La latence moyenne avec BrFusion reste cependant 13,1% plus élevée qu'en natif. BrFusion réduit aussi l'écart type : il est de 5,9% de la latence moyenne, contre 6,6% avec NAT et 4,9% en natif.

Ensuite pour NGINX, BrFusion améliore la latence moyenne des requête de 30,1% par rapport à NAT, mais elle est cependant 120,3% plus élevée qu'en natif. Similairement, l'écart type est d'environ deux fois la latence moyenne pour BrFusion et NAT, contre seulement 47% de la latence moyenne en natif. On explique donc cette dégradation des performances avec BrFusion et NAT par rapport au natif, à l'application elle-même plutôt qu'à la couche réseau.

Utilisation du processeur

On compare ici aussi BrFusion à NAT et au natif, sur leur consommation de processeur respective.

Méthodologie

Pour ce qui est de l'impact sur le processeur, on en surveille l'utilisation, divisée entre quatre sources :

1. application, notée `app` ;
2. noyau du SE de la VM, notée `noyau` ;
3. service des interruptions logicielles par le noyau, notée `int` ;
4. temps processeur alloué par le serveur physique à la VM, notée `vm`.

Résultats

Les résultats d'utilisation du processeur par le macro-banc d'essai Kafka sont donnés en figures 5.19a et 5.19b, et par NGINX en figures 5.19c et 5.19d.

Concernant Kafka, les niveaux d'utilisation avec BrFusion et avec NAT pour la VM sont tous deux environ 9,6% plus haut qu'en natif (visible en figure 5.19a). Cependant, l'utilisation du processeur par le serveur Kafka dans la VM (visible en figure 5.19a) montre que BrFusion réduit le temps de

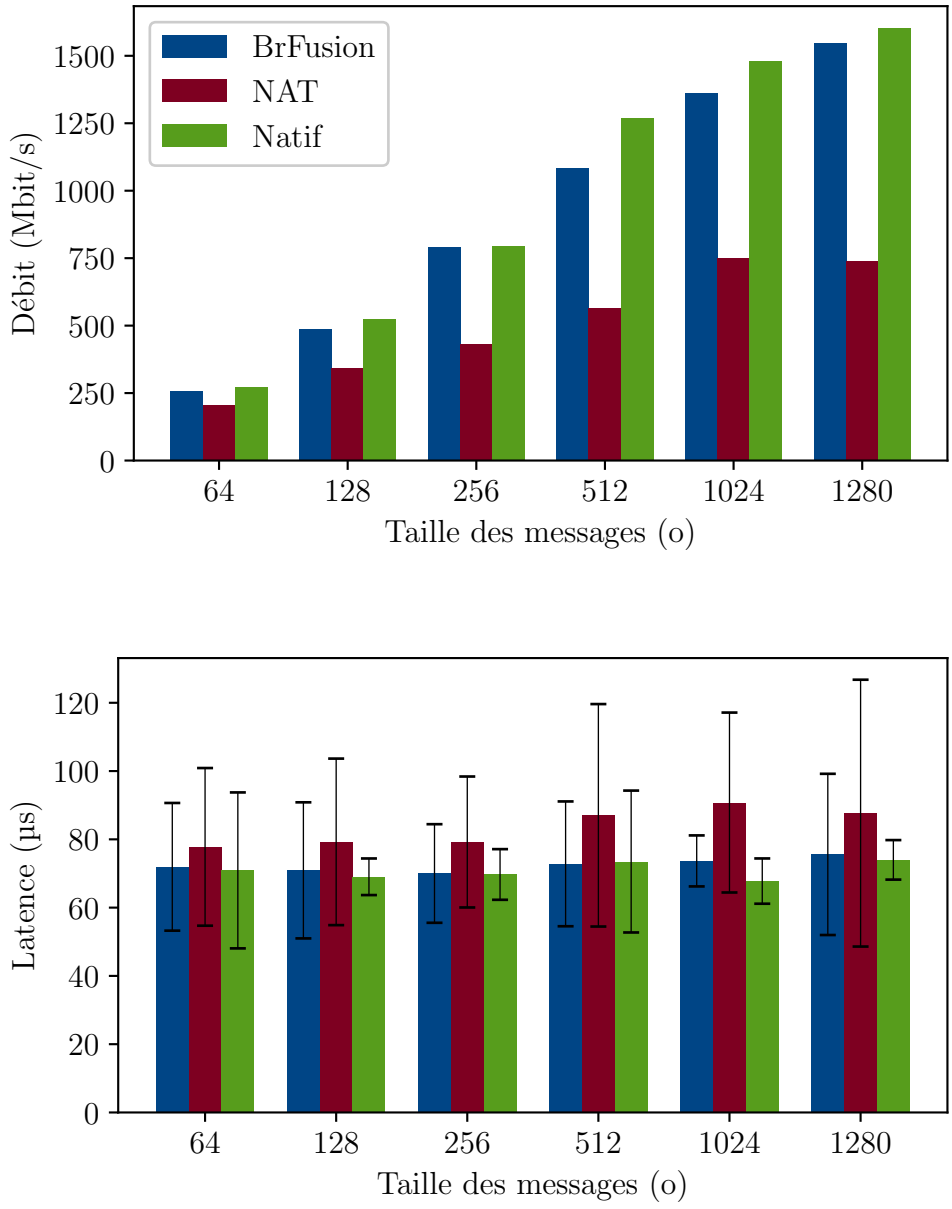
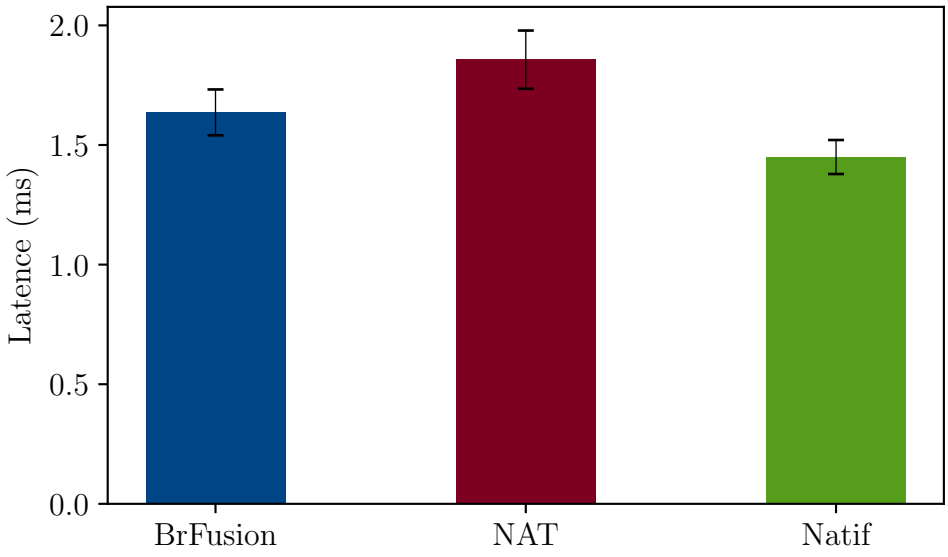
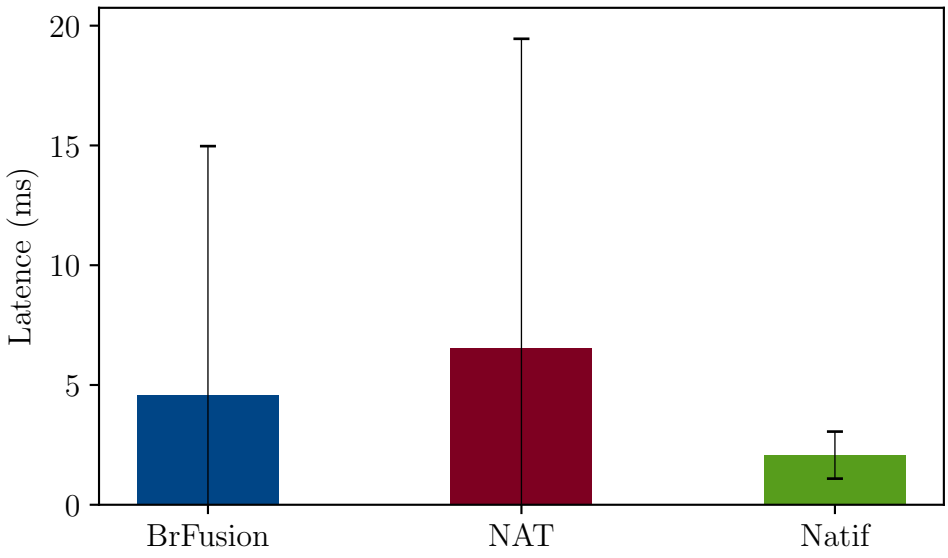


FIGURE 5.17. – Performance réseau de BrFusion, de NAT et du mode natif, par Netperf. Les barres donnent l'écart type de la latence.



(a) Kafka



(b) NGINX

FIGURE 5.18. – Performance réseau de BrFusion, de NAT et du mode natif, par les macro-bancs d'essai. Les barres donnent l'écart type de la latence moyenne sur dix exécutions.

processeur dédié au service des interruptions logicielles de 67% par rapport à NAT. Cela correspond à une réduction totale du temps de processeur de 4,7%. En effet, les règles permettant le routage NAT sont appliquées sur les paquets via des attaches logicielles exécutées dans ces interruptions logicielles. BrFusion élimine effectivement l'exécution de ces attaches.

On peut faire les mêmes observations, mais avec une plus grande amplitude, avec les résultats de NGINX.

Temps de création du pod

Pour finir, on compare le temps de démarrage d'un conteneur sous BrFusion, avec le temps de démarrage sous la solution de virtualisation imbriquée originelle, c'est-à-dire NAT.

Méthodologie

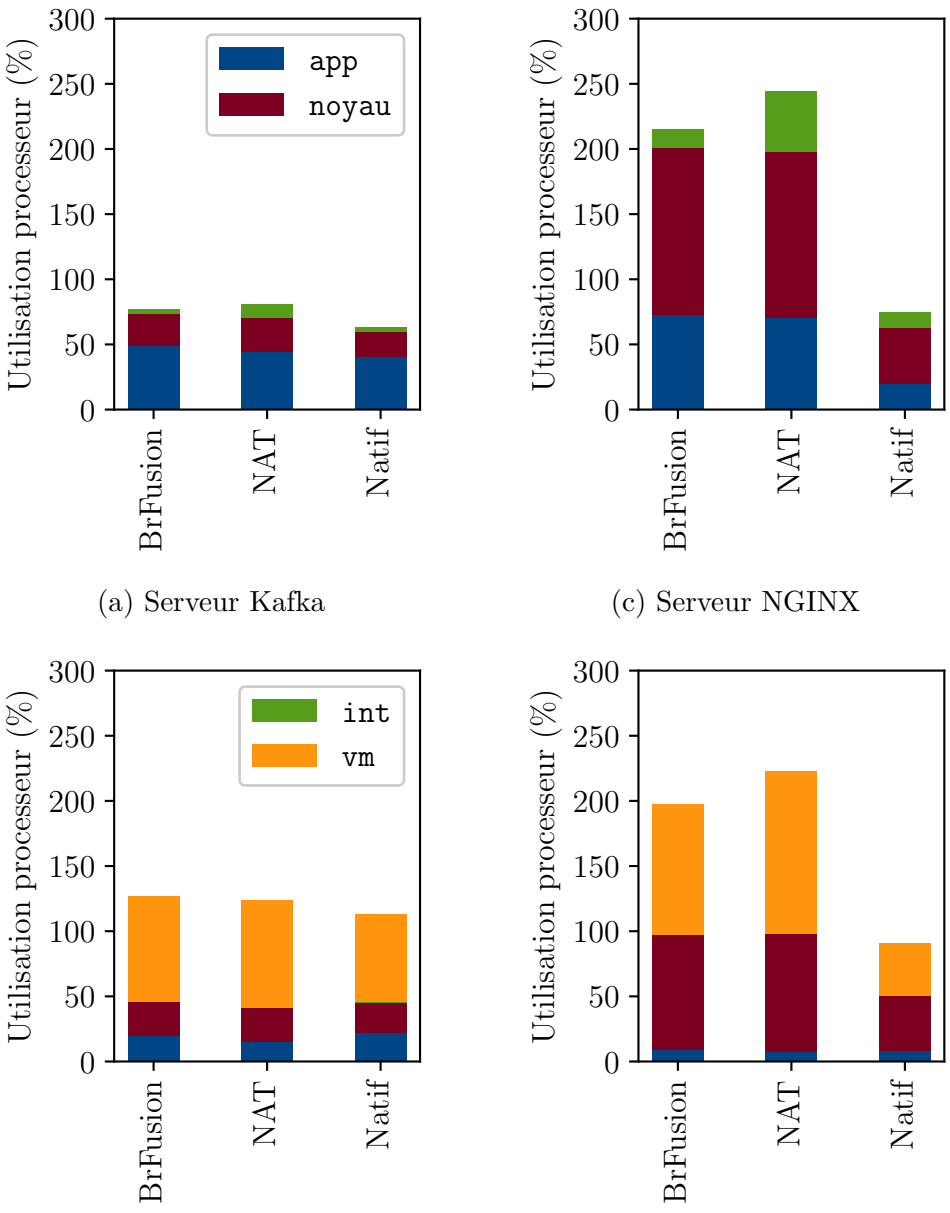
Pour simplifier l'expérimentation, on considère le temps de création d'un conteneur plutôt qu'un pod. Techniquement, les mêmes actions sont exécutées, sauf toutes les opérations de gestion d'un pod effectuées par un orchestrateur. Ainsi, cette simplification n'impacte pas ces expériences.

On définit donc le temps de création d'un conteneur, comme la durée entre l'envoi de l'ordre de création du conteneur au moteur de conteneurs, et l'envoi par le conteneur d'un message par un socket TCP. Les mesures sont basées sur le Time Stamp Counter (c'est-à-dire « compteur dateur », TSC) [131]. L'émulation du processeur par QEMU a été modifiée pour passer le TSC du serveur physique tel quel à la VM. Ce faisant, le TSC devient une horloge absolue, d'une grande précision, à travers la frontière de la VM.

Ici, la partie « serveur » dans le conteneur est un script Python qui ouvre une connexion TCP vers le client et transmet le TSC lu grâce à un exécutable codé en C qui récupère la valeur depuis les registres du processeur virtuel. Au total, cette expérience est exécutée 100 fois.

Résultats

Les résultats de cette expérience sont reproduits en figure 5.20. On peut voir en figure 5.20a que 75% des temps de démarrage mesurés sont légèrement plus courts avec BrFusion qu'avec NAT. Le figure 5.20b donne des statistiques plus précises. Par exemple, le temps de démarrage minimum est environ 5,2% plus court avec BrFusion. Pour résumer, BrFusion ne ralentit par le démarrage des conteneurs.



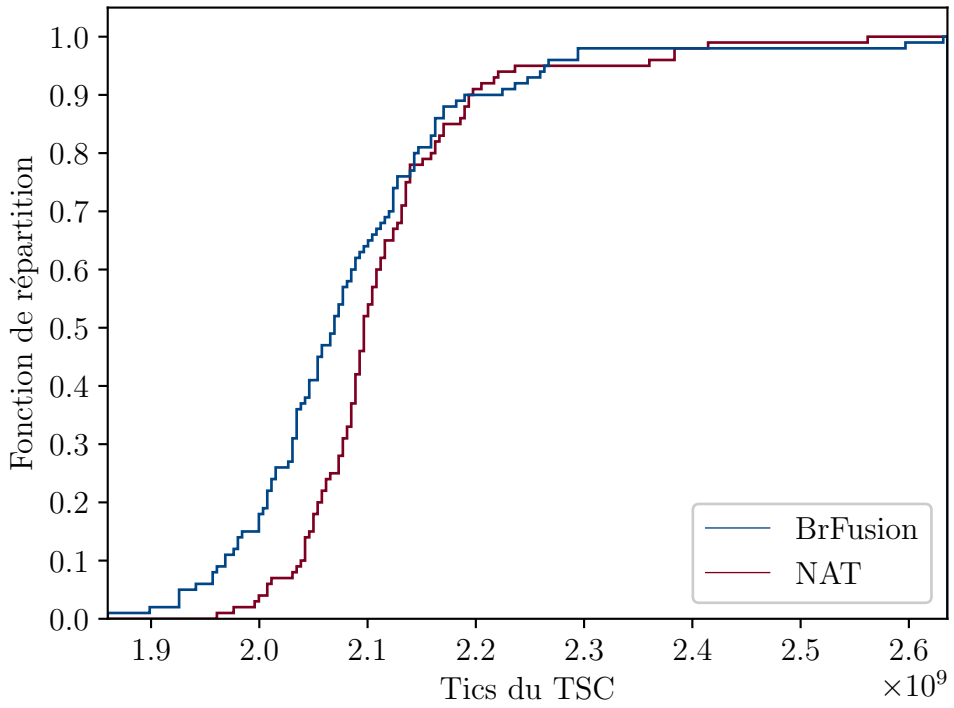
(a) Serveur Kafka

(c) Serveur NGINX

(b) VM Kafka dans l'hôte

(d) VM NGINX dans l'hôte

FIGURE 5.19. – Utilisations processeur pour les bancs d'essai Kafka (gauche) et NGINX (droite).
Haut : utilisation processeur du serveur. Bas : utilisation processeur pour la VM qui héberge le serveur.



(a) Répartition des temps de démarrage

	NAT (10^9 tics)	BrFusion (10^9 tics)	Différence
Min	1.96	1.86	-5.2%
50%	2.10	2.07	-1.4%
95%	2.24	2.27	+1.3%

(b) Statistiques

FIGURE 5.20. – Temps de démarrage d'un conteneur avec BrFusion et NAT.

Résumé de l'évaluation et discussion

Pour conclure sur les avantages et les inconvénients de BrFusion, cette solution réduit grandement la dégradation des performances du réseau en virtualisation imbriquée. Elle peut même montrer des résultats similaires à ceux observés avec un seul niveau de virtualisation. Elle réduit de plus l'impact sur la consommation processeur, ce qui a un effet positif sur les performances des applications. De plus, BrFusion ne ralentit pas la création des conteneurs, et il semble donc que ce soit *une amélioration nette dans le paysage de la virtualisation imbriquée*.

BrFusion représente un changement dans la gestion des ressources réseau. En effet, les ponts des différents utilisateurs du cloud doublement virtualisé étaient auparavant séparés dans leurs différentes VM. Ils sont désormais tous gérés par le même SE, celui du serveur hôte physique. Il faut donc s'assurer plus qu'avant, de l'équité et de la sécurité entre ces ponts.

5.6. État de l'art

Le contexte de ce chapitre est donc la virtualisation imbriquée. En particulier, le sujet est le réseau et sa virtualisation en général. Les solutions proposées aux deux défauts de la virtualisation imbriquée décrits en introduction reposent respectivement sur le réseautage entre les VM, et le réseautage dans les environnements virtualisés (VM et conteneurs).

5.6.1. Réseautage entre les VM

La communication entre des applications placées dans deux VM différentes est un réel besoin, et des solutions ont été étudiées en ce sens. On peut citer le système de Zhang, Lu et Panda [142]. Il combine la technique de la mémoire partagée implémentée par Nahanni [78] pour les VM colocalisées, avec la technologie SR-IOV par Infiniband pour les VM sur des serveurs hôtes différents. Ce système est cependant conçu spécifiquement pour les applications basées sur MPI (pour *Message Passing Interface*, ou « interface de transmission de message »), c'est-à-dire un protocole pour le calcul parallèle et distribué. Il faut de plus modifier spécialement les applications pour l'utiliser. À l'inverse, HostLo est une solution au niveau de la couche de transport du réseau, qui est totalement transparente pour le conteneur.

Un autre système est MemPipe [144] : il se place sous le niveau IP afin d'échanger les paquets entre des VM colocalisées via mémoire partagée. Bien que cette solution n'aie pas besoin d'une modification de l'application,

elle n'inclut aucune notion d'isolation. En effet, elle analyse simplement les paquets pour déterminer la VM de destination, qui doit être notifiée de la transmission. De plus, utiliser cette solution pour implémenter de manière transparente l'interface réseau locale à un pod serait une tâche difficile.

5.6.2. Réseautage dans les environnements virtualisés

La plupart du temps, la virtualisation du réseau est basée sur des commutateurs réseaux virtuels, appelés simplement des ponts [80] (en anglais *bridges*). Ces ponts sont toujours utilisés même avec les réseaux en surcouche. Ils peuvent cependant être remplacés avantageusement par la commutation virtuelle dans le matériel (en anglais *hardware switching*) grâce à la technologie de macvlan [4]. Ce remplacement est totalement compatible avec BrFusion (voir section 5.5). Grossièrement, le principe de BrFusion est d'éliminer le pont dans la VM, et de fournir à cette dernière une interface réseau déjà virtualisée pour le pod par le serveur hôte. Cette interface peut être créée avec macvlan. Cependant, macvlan n'est pas utilisé en pratique par les fournisseurs de cloud. Son intégration nécessite en effet de changer les pratiques de routage interne, et de gérer directement au niveau du centre d'hébergement les adresses IP des conteneurs [147].

Ainsi, il est possible d'utiliser de meilleures technologies de commutation virtuelle du réseau, comme macvlan ou même encore Open vSwitch [106]. Ces solutions s'appliquent autant aux VM qu'aux conteneurs, mais elles ne corrigent pas le problème fondamental de la duplication du réseau virtuel, ni celui du déploiement inter-VM des pods.

Concernant les conteneurs, on trouve des études générales sur leurs performances réseau [27, 127]. L'étude de Suo et al. [127] observe de manière notable la dégradation des performances due à la virtualisation imbriquée du réseau.

Par la suite, des travaux se sont penchés sur l'amélioration du réseautage pour les conteneurs. Kim et al. [64] utilisent du réseautage virtuel par RDMA (pour *Remote Direct Memory Access*, ou « accès mémoire direct à distance »). Le support dans le noyau Linux pour les réseaux en surcouche a été développé par Zhuo et al. [147]. Le réseautage en surcouche au niveau applicatif a aussi été proposé par Subhraveti et al. [125]. Ces trois solutions simplifient et optimisent le réseautage en surcouche, mais elles ne résolvent finalement pas la duplication du réseau virtuel.

Le travail de Nakamura, Sekiya et Tazaki [93] développe la greffe de sockets : il s'agit d'outrepasser les piles réseau des conteneurs, en greffant les sockets créés par les conteneurs sur des sockets provenant directement

de la pile réseau du SE hôte. Il élimine une couche de virtualisation du réseau, comme BrFusion ; mais il nécessite la modification des applications ou bien la mise en place de pièges pour les syscalls liés aux sockets.

Enfin, aucun de ces travaux ne prend en compte la virtualisation imbriquée. Ils cherchent à optimiser le lien entre le conteneur et son hôte, qui est ici une VM. En allant plus loin, le concept derrière BrFusion est d'éliminer purement et simplement ce lien, en permettant à un conteneur d'utiliser directement le réseau virtualisé par le serveur hôte physique.

5.7. Conclusion

Ainsi, la virtualisation imbriquée dans sa forme actuelle comporte des défauts majeurs liés à la virtualisation du réseau. Le premier est la contrainte du déploiement de pods entiers sur une seule VM. Elle limite l'orchestrateur dans sa gestion des ressources, qui sont alors fragmentées. Cet effet se répercute sur leur taux d'utilisation et sur le coût du cloud pour l'utilisateur. Le second est la duplication du réseau virtuel, qui a pour conséquence une dégradation des performances, parce qu'elle rallonge le chemin des données échangées.

Le travail présenté dans ce chapitre propose une solution à chacun de ces défauts. D'abord, HostLo permet à l'orchestrateur de réaliser des déploiements inter-VM de pods en proposant une interface réseau inter-VM efficace pour la communication entre les conteneurs d'un pod. L'adaptation de l'algorithme de placement des pods de l'orchestrateur produit ainsi des économies substantielles. Ensuite, BrFusion simplifie le chemin du réseau virtuel dans la virtualisation imbriquée et corrige ainsi la dégradation des performances de manière presque parfaite, et sans impact négatif.

En fait, la base de ces deux solutions réside dans la *communication entre l'orchestrateur et l'hyperviseur*. Le modèle classique place l'hyperviseur au centre du centre d'hébergement, parce qu'il a le contrôle sur les ressources physiques ; ici, l'orchestrateur est contraint de faire au mieux avec les VM fournies. On veut dans ce chapitre proposer une inversion de ces positions. L'orchestrateur doit reprendre le contrôle du centre d'hébergement, et donner des ordres d'allocation de ressources (interfaces réseau, VM) à l'hyperviseur. Autrement dit, si la couche de virtualisation des VM est actuellement déjà camouflée à l'utilisateur qui ne voit que les conteneurs, il s'agit de l'abstraire aussi dans la gestion du centre d'hébergement.

Chapitre 6

Conclusion et perspectives

Ainsi, des solutions ont été proposées pour résoudre certaines problématiques du cloud multi-virtualisé. Ce chapitre donne une conclusion à l'ensemble de ce travail, et dresse un portrait des perspectives futures.

6.1. Multi-virtualisation du cloud

On a vu que le cloud est le principal fournisseur de la ressource informatique. Il est basé sur deux formes de virtualisation, dont les avantages peuvent être combinés dans la multi-virtualisation. Mais on y retrouve aussi les inconvénients de chacune, ainsi que de nouvelles problématiques.

La consolidation des machines virtuelles est à un stade très avancé, et seul un changement de paradigme peut permettre de nouvelles améliorations. C'est ce que propose Drowsy-DC, en incluant l'inactivité de la charge de travail comme critère de consolidation.

De plus, malgré sa relative maturité, les différents niveaux de la virtualisation souffrent encore de certains maux. Ainsi, la conteneurisation a un impact très néfaste sur la prédictibilité des performances. Le travail présenté ici cherche à réduire cet impact en introduisant une nouvelle méthode d'allocation des ressources.

Enfin, la multi-virtualisation n'est pas triviale à implémenter. Cette difficulté est bien visible dans la situation du réseau. D'une part, celui-ci est dédoublé dans cet environnement, et les performances sont considérablement réduites ; BrFusion intègre les virtualisations du réseau des deux niveaux, afin d'en simplifier la multi-virtualisation et corriger le problème des performances. D'autre part, la virtualisation du réseau effectuée par les VM gêne sa multi-virtualisation ; en fait, les deux niveaux sont mal intégrés dans cette imbrication, et il en résulte une mauvaise gestion des ressources. Cette contrainte est levée grâce à HostLo, qui propose une communication

entre les niveaux de virtualisation pour mieux utiliser les ressources du centre d'hébergement et réduire le coût du cloud.

Finalement, les problématiques du cloud virtualisé se trouvent autant dans chacun des niveaux de virtualisation qu'à l'interface de ceux-ci ; mais des solutions innovantes émergent dans leur coopération.

6.2. Pistes d'amélioration des travaux

Tout d'abord, Drowsy-DC, qui ne concerne actuellement que le premier niveau de virtualisation, pourrait bénéficier de la double virtualisation. Par exemple, la détection précise de l'inactivité des serveurs pourrait être grandement facilitée grâce aux informations sur la charge de travail dont dispose l'orchestrateur. De même, comme l'orchestrateur contrôle le réseautage de ses pods avec des ajouts logiciels, la reprise d'un serveur ne nécessiterait plus d'analyser les requêtes entrantes pour déterminer leurs serveurs de destination. Enfin, l'orchestrateur pourrait participer à l'effort de consolidation dans le but d'obtenir des serveurs inactifs, en prenant en compte la PI lors de la répartition des pods sur les VM.

En ce qui concerne le deuxième travail, celui-ci corrige un problème intrinsèquement lié au deuxième niveau de virtualisation, c'est-à-dire les conteneurs. L'algorithme d'allocation hybride qu'il propose pourrait être amélioré pour prendre en compte l'architecture NUMA dans la construction des ensembles de processeurs. Par ailleurs, dans un contexte multi-virtualisé, ces ensembles contiendraient des vCPU plutôt que des processeurs réels. Cette distinction a des implications difficiles à identifier sur les performances des applications, et il faudrait établir une communication entre l'hyperviseur et l'orchestrateur afin d'allouer au mieux les ressources processeur.

Enfin, on envisage facilement des extensions pour HostLo et BrFusion. À propos du premier, il est limité dans son implémentation à un serveur physique, et on pourrait donc étendre la désagrégation des pods au-delà, sur plusieurs serveurs. On peut aussi constater que l'amélioration de l'algorithme de placement des pods dans l'orchestrateur, pour prendre en compte cette capacité, apporterait des économies financières supplémentaires. Enfin pour BrFusion, comme son principe est d'indiquer à l'orchestrateur de déléguer la provision du réseau des pods au gestionnaire de VM, il faut aussi envisager comment mieux intégrer les mesures de sécurité et d'isolation entre ces deux niveaux de virtualisation.

6.3. Vers le cloud Function-as-a-Service multi-virtualisé

Que la multi-virtualisation existe par contrainte ou par choix technique, elle continuera d'être présente dans le cloud. On constate de plus en plus la combinaison, sous différentes formes, de la virtualisation du matériel avec la virtualisation du système d'exploitation.

Par exemple, Kata Containers [112] intègre la virtualisation du matériel dans un système de conteneurisation, afin d'obtenir les avantages de sécurité des machines virtuelles tout en conservant les avantages de facilité d'utilisation et de vitesse des conteneurs. Un autre exemple proche est gVisor [140]. Ce projet développé par Google propose un unikernel dédié à la charge de travail conteneurisée. Il s'agit d'un nouveau moteur de conteneurs qui aide à sécuriser le système d'exploitation hôte contre les applications qui y sont déployées. D'une autre manière, Amazon développe Firecracker [47], qui est un hyperviseur minimaliste. Cet aspect minimaliste correspond à des optimisations pour exécuter des machines virtuelles dédiées au Function-as-a-Service. Les systèmes d'exploitations invités sont eux-mêmes dédiés et optimisés pour servir des ressources pour le Function-as-a-Service, leur hyperviseur peut donc l'être également.

Ces exemples montrent en fait la combinaison des différents niveaux de virtualisation dans la technique qui sert le cloud. Plus précisément, ils décrivent des améliorations de la communication entre les acteurs des niveaux de virtualisation. Certes, des problématiques spécifiques à chaque niveau persistent, notamment sur les axes de la performance et de la sécurité. Mais c'est dans leur collaboration qu'existe le futur du cloud.

Le motif général apparaît : il s'agit de transformer l'hyperviseur en une méthode d'allocation des ressources au service des orchestrateurs. C'est parce qu'il reste très efficace dans la gestion des ressources du point de vue du fournisseur de cloud ; mais à l'inverse, ce sont bien les orchestrateurs qui présentent la meilleure interface à l'utilisateur.

Pour aller plus loin, il faut développer un environnement où la notion de machine virtuelle disparaît. En effet, une VM constitue un ensemble rigide de toutes les ressources qui constituent une machine, pourtant une application conteneurisée décrit précisément ses besoins : processeur, mémoire, réseau, disque… C'est-à-dire que la virtualisation des machines virtuelles consiste à proposer du matériel général qui couvre tous les besoins, notamment par le biais d'un SE invité, tandis que la virtualisation basée sur des conteneurs correspond exactement aux besoins de l'utilisateur du cloud, très

spécifiques. Par conséquent, la couche d'abstraction de la VM devient inutile. L'hyperviseur évolue alors vers la provision de ressources désagrégées, plutôt que la création de VM complètes, à destination d'orchestrateurs qui allouent directement les ressources virtualisées à des applications cloud. Les défis s'expriment alors en termes de caractéristiques de ce matériel virtuel fourni aux applications cloud : performance et prédictibilité, interface, partage, et particulièrement la sécurité. En effet, la disparition de la VM, du matériel séparé qu'elle virtualise et du SE invité élimine aussi leur protection.

Annexe A

Code source du simulateur de Drowsy-DC avec CloudSim

La simulation de Drowsy-DC pour l'évaluation repose sur CloudSim [24]. Son code source est disponible à l'adresse suivante : `https://git.bacou.me/Drowsy-DC/SimulationCloudSim`. En fait, elle repose sur une version légèrement modifiée et corrigée de CloudSim, elle-même disponible à cette adresse : `https://git.bacou.me/Drowsy-DC/SimulationCloudSim`.

Il s'agit de simuler la consommation énergétique d'un centre d'hébergement, ce qui est possible grâce à CloudSim. Le code source de la simulation ajoute l'algorithme de Drowsy-DC, ainsi que ceux de Neat et d'Oasis (voir section 3.9). Il inclut aussi des spécifications de serveurs capables de passer dans des états d'énergie plus faible.

La simulation repose sur le suivi de traces d'utilisation d'un centre d'hébergement. Les traces utilisées pour l'évaluation de Drowsy-DC sont disponibles ici : `https://gitlab.com/Drowsy-DC/dataset`.

Annexe B

Code source du simulateur d'allocation hybride pour le respect du principe tel-tel

La simulation de l'algorithme d'allocation hybride pour son évaluation est disponible ici : `https://git.bacou.me/UnnestedVirtualization/KubernetesCPUSets`. Elle repose sur un simulateur plus général qui rejoue des allocations de conteneurs, trouvable à l'adresse suivante : `https://git.bacou.me/UnnestedVirtualization/KubeSim`.

Il s'agit de simuler les allocations de ces conteneurs avec différents algorithmes au niveau de l'orchestrateur et au niveau de chaque serveur. La simulation donne les métriques décrites en section 4.5 pour l'évaluation de l'algorithme d'allocation hybride. Le simulateur de base est flexible et permet d'implémenter facilement différents comportements pour tous les acteurs du centre d'hébergement, ainsi que de rejouer différentes traces.

Annexe C

Code source du pilote d'interface HostLo pour le noyau Linux

Le système HostLo repose sur une nouvelle interface réseau virtuelle, qui est fournie par le noyau Linux. Il s'agit d'une modification du pilote TAP de ce noyau, disponible à l'adresse suivante : `https://gitlab.com/Thrar/linux/-/tree/vtap` (branche `vtap` de la bifurcation du code source du noyau Linux). Le principe de l'implémentation de l'interface HostLo est décrit en section 5.4.3.

Cette modification consiste en l'ajout de deux nouveaux modes de fonctionnement d'une interface TAP :

1. `HUB` : une interface en mode *hub*, c'est-à-dire en mode concentrateur réseau, envoie les paquets qu'elle reçoit depuis une queue, sur toutes ses autres queues d'émission ;
2. `LOOPBACK` : une interface en mode *loopback*, c'est-à-dire en mode boucle de retour, renvoie les paquets qu'elle reçoit depuis une queue, sur la queue d'émission correspondante.

Ainsi, créer une interface TAP avec ces deux modes permet d'obtenir une interface se comportant comme l'interface `localhost`.

Annexe D

Code source des modifications de QEMU pour l'utilisation de HostLo

En plus de la modification du noyau Linux pour fournir des interface TAP `localhost` (voir annexe C), HostLo requiert la modification du gestionnaire de VM QEMU pour utiliser ces nouvelles interfaces. Le code source ajoutant cette fonctionnalité est disponible à l'adresse suivante : `https://git.bacou.me/maba/qemu/src/branch/hostloop` (branche `hostloop` de la bifurcation du code source de QEMU).

Cette modification consiste d'une part en l'ajout de la capacité de QEMU à configurer une interface TAP avec cette nouvelle fonctionnalité ; et d'autre part en l'introduction d'un nouveau programme d'aide dédié à l'activation des interfaces HostLo.

Annexe E

Code source du simulateur d'économies financières permises par HostLo

La simulation pour l'évaluation des économies d'argent apportées par HostLo est disponible à l'adresse suivante : `https://git.bacou.me/UnnestedVirtualization/HostloSim`. Il s'agit de rejouer les traces d'allocation de pods d'un centre d'hébergement, en les allouant avec la possibilité d'un déploiement inter-VM ou sans cette possibilité. Les algorithmes d'allocation utilisés sont celui de Kubernetes, et une modification de celui-ci pour prendre en compte la capacité d'un déploiement inter-VM (voir section 5.4.6).

Bibliographie

[1] Moustafa Abdelbaky et al. « Docker containers across multiple clouds and data centers ». In : *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2015, p. 368-371.

[2] *Amazon Web Services (AWS) – Cloud Computing Services*. 2020. URL : https://aws.amazon.com/ (visité le 17/01/2020).

[3] George Amvrosiadis et al. « On the diversity of cluster workloads and its impact on research results ». In : *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, p. 533-546.

[4] Jason Anderson et al. « Performance considerations of network functions virtualization using containers ». In : *2016 International Conference on Computing, Networking and Communications (ICNC)*. IEEE. 2016, p. 1-7.

[5] *Apache Kafka*. 2020. URL : https://kafka.apache.org (visité le 06/02/2020).

[6] *App Engine | Google Cloud*. 2020. URL : https://cloud.google.com/appengine/ (visité le 17/01/2020).

[7] *AWS | Amazon EC2 – Servie d'hébergement cloud évolutif*. 2020. URL : https://aws.amazon.com/fr/ec2/ (visité le 17/01/2020).

[8] *AWS | Elastic Beanstalk – PaaS et gestion d'applications*. 2020. URL : https://aws.amazon.com/fr/elasticbeanstalk/ (visité le 17/01/2020).

[9] *Azure App Service – app hosting | Microsoft Azure*. 2020. URL : https://azure.microsoft.com/en-us/services/app-service/ (visité le 17/01/2020).

[10] Mathieu Bacou, Alain Tchana et Daniel Hagimont. « Your Containers Should be WYSIWYG ». In : *2019 IEEE International Conference on Services Computing, SCC 2019*. IEEE, juil. 2019, p. 56-64.

[11] Mathieu Bacou, Grégoire Todeschi, Alain Tchana et Daniel Hagimont. « Nested Virtualization Without the Nest ». In : *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019*. ACM, août 2019, 12:1-12:10.

[12] Mathieu Bacou, Grégoire Todeschi, Alain Tchana, Daniel Hagimont, Baptiste Lepers et Willy Zwaenepoel. « Drowsy-DC : Data Center Power Management System ». In : *2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019*. IEEE, mai 2019, p. 825-834.

[13] Luciano Baresi et al. « A discrete-time feedback controller for containerized cloud applications ». In : *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, p. 217-228.

[14] Sean Barker et al. « An empirical study of memory sharing in virtual machines ». In : *Proceedings of the 2012 USENIX conference on Annual Technical Conference*. USENIX Association. 2012, p. 25-25.

[15] Luiz André Barroso et Urs Hölzle. « The case for energy-proportional computing ». In : *Computer* 40.12 (2007), p. 33-37.

[16] Luiz André Barroso, Urs Hölzle et Parthasarathy Ranganathan. « The datacenter as a computer : Designing warehouse-scale machines ». In : *Synthesis Lectures on Computer Architecture* 13.3 (2018), p. i-189.

[17] Olivier Beaumont, Lionel Eyraud-Dubois et Juan-Angel Lorenzo-del-Castillo. « Analyzing real cluster data for formulating allocation algorithms in cloud platforms ». In : *Parallel Computing* 54 (2016), p. 83-96.

[18] Anton Beloglazov et Rajkumar Buyya. « Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints ». In : *IEEE Transactions on Parallel and Distributed Systems* 24.7 (2013), p. 1366-1379.

[19] Anton Beloglazov et Rajkumar Buyya. « OpenStack Neat : a framework for dynamic and energy-efficient consolidation of virtual machines in OpenStack clouds ». In : *Concurrency and Computation : Practice and Experience* 27.5 (2015), p. 1310-1333.

[20] Anton Beloglazov et Rajkumar Buyya. « Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers ». In : *Concurrency and Computation : Practice and Experience* 24.13 (2012), p. 1397-1420.

[21] Christian Bienia et Kai Li. *Benchmarking modern multiprocessors*. Princeton University Princeton, NJ, 2011.

[22] Nilton Bila et al. « Jettison : efficient idle desktop consolidation with partial VM migration ». In : *Proceedings of the 7th ACM european conference on Computer Systems*. ACM. 2012, p. 211-224.

[23] Eric A Brewer. « Kubernetes and the path to cloud native ». In : *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 2015, p. 167-167.

[24] Rodrigo N Calheiros et al. « CloudSim : a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms ». In : *Software : Practice and experience* 41.1 (2011), p. 23-50.

[25] Sudip Chahal et al. « An enterprise private cloud architecture and implementation roadmap ». In : *IT@ Intel White Paper, USA* (2010).

[26] Jui-Hao Chiang, Han-Lin Li et Tzi-cker Chiueh. « Working Set-based Physical Memory Ballooning. » In : *ICAC*. 2013, p. 95-99.

[27] Joris Claassen, Ralph Koning et Paola Grosso. « Linux containers networking : Performance and scalability of kernel modules ». In : *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE. 2016, p. 713-717.

[28] Theo Combe, Antony Martin et Roberto Di Pietro. « To docker or not to docker : A security perspective ». In : *IEEE Cloud Computing* 3.5 (2016), p. 54-62.

[29] *Compute Engine : Virtual Machines (VMs) | Google Cloud*. 2020. URL : `https://cloud.google.com/compute/` (visité le 17/01/2020).

[30] *Container Network Interface – networking for Linux containers*. 2020. URL : `https://github.com/containernetworking/cni` (visité le 06/02/2020).

[31] Eli Cortez et al. « Resource central : Understanding and predicting workloads for improved resource management in large cloud platforms ». In : *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, p. 153-167.

[32] Jeffrey Dean et Sanjay Ghemawat. « MapReduce : simplified data processing on large clusters ». In : *Communications of the ACM* 51.1 (2008), p. 107-113.

[33] *Dégooglisons Internet (Framasoft)*. 2020. URL : `https://degooglisons-internet.org/` (visité le 17/01/2020).

[34] Christina Delimitrou et Christos Kozyrakis. « Hcloud : Resource-efficient provisioning in shared cloud systems ». In : *ACM SIGOPS Operating Systems Review* 50.2 (2016), p. 473-488.

[35] Christina Delimitrou et Christos Kozyrakis. « Quasar : resource-efficient and QoS-aware cluster management ». In : *ACM SIGPLAN Notices* 49.4 (2014), p. 127-144.

[36] Yahya Al-Dhuraibi et al. « Autonomic vertical elasticity of docker containers with ElasticDocker ». In : *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE. 2017, p. 472-479.

[37] *Documentation/QMP – QEMU*. 2020. URL : `https://wiki.qemu.org/Documentation/QMP` (visité le 06/02/2020).

[38] *DPDK vHost User Ports – Open vSwitch Documentation*. 2020. URL : `https://docs.openvswitch.org/en/latest/topics/dpdk/vhost-user/` (visité le 07/02/2020).

[39] Rajdeep Dua, A Reddy Raja et Dharmesh Kakadia. « Virtualization vs containerization to support paas ». In : *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, p. 610-614.

[40] *EC2 Instance Pricing – Amazon Web Services (AWS)*. 2020. URL : `https://aws.amazon.com/ec2/pricing/on-demand/` (visité le 05/02/2020).

[41] Magnus Ekman et Per Stenstrom. « A robust main-memory compression scheme ». In : *ACM SIGARCH Computer Architecture News*. T. 33. 2. IEEE Computer Society. 2005, p. 74-85.

[42] *Empowering App Development for Developers | Docker*. 2020. URL : `https://www.docker.com/` (visité le 21/01/2020).

[43] *ESXi | Bare Metal Hypervisor | VMware*. 2020. URL : `https://www.vmware.com/products/esxi-and-esx.html` (visité le 21/01/2020).

[44] Stijn Eyerman et Lieven Eeckhout. « The Benefit of SMT in the Multi-core Era : Flexibility Towards Degrees of Thread-level Parallelism ». In : *ASPLOS*. 2014.

[45] Wes Felter et al. « An updated performance comparison of virtual machines and linux containers ». In : *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2015, p. 171-172.

[46] Michael Ferdman et al. « Clearing the clouds : a study of emerging scale-out workloads on modern hardware ». In : *ACM SIGPLAN Notices*. T. 47. 4. ACM. 2012, p. 37-48.

[47] *Firecracker*. 2020. url : `https://firecracker-microvm.github.io/` (visité le 19/02/2020).

[48] *G Suite : applications de collaboration et de productivité pour les entreprises*. 2020. url : `https://gsuite.google.fr/` (visité le 17/01/2020).

[49] Xing Gao et al. « ContainerLeaks : Emerging security threats of information leakages in container clouds ». In : *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2017, p. 237-248.

[50] Md E. Haque et al. « Few-to-Many : Incremental Parallelism for Reducing Tail Latency in Interactive Services ». In : *ASPLOS*. 2015.

[51] *HewlettPackard/netperf*. 2020. url : `https://github.com/HewlettPackard/netperf` (visité le 06/02/2020).

[52] Lorin Hochstein et Rene Moser. *Ansible : Up and Running : Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, Inc., 2017.

[53] Philipp Hoenisch et al. « Four-fold auto-scaling on a contemporary deployment platform using docker containers ». In : *International Conference on Service-Oriented Computing*. Springer. 2015, p. 316-323.

[54] Jez Humble et David Farley. *Continuous Delivery : Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[55] *Hyper-V on Windows 10 | Microsoft Docs*. 2020. url : `https://docs.microsoft.com/virtualization/hyper-v-on-windows/` (visité le 21/01/2020).

[56] *IBM Cloud Virtual Servers | IBM*. 2020. URL : https://www.ibm.com/cloud/virtual-servers-cloud/ (visité le 17/01/2020).

[57] *Intel Ethernet Controller I350 Datasheet*. Intel. 2017. URL : http://www.intel.com/content/www/us/en/embedded/products/networking/ethernet-controller-i350-datasheet.html (visité le 30/01/2020).

[58] Canturk Isci et al. « Agile, efficient virtualization power management with low-latency server power states ». In : *ACM SIGARCH Computer Architecture News*. T. 41. 3. ACM. 2013, p. 96-107.

[59] Myeongjae Jeon et al. « Adaptive parallelism for web search ». In : *EuroSys*. ACM. 2013, p. 155-168.

[60] Stephen T Jones, Andrea C Arpaci-Dusseau et Remzi H Arpaci-Dusseau. « Geiger : monitoring the buffer cache in a virtual machine environment ». In : *ACM Sigplan Notices* 41.11 (2006), p. 14-24.

[61] Venkateswararao Jujjuri et al. « Virtfs–a virtualization aware file system pass-through ». In : *Ottawa Linux Symposium (OLS)*. Citeseer. 2010, p. 109-120.

[62] Efe Kahraman. *Docker Awareness in Java*. Rapp. tech. 2018.

[63] Sanidhya Kashyap, Changwoo Min et Taesoo Kim. « Scaling Guest OS Critical Sections with eCS ». In : *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, p. 159-172.

[64] Daehyeok Kim et al. « FreeFlow : Software-based Virtual RDMA Networking for Containerized Clouds ». In : *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, p. 113-126.

[65] In Kee Kim et al. « A Supervised Learning Model for Identifying Inactive VMs in Private Cloud Data Centers ». In : *Proceedings of the Industrial Track of the 17th International Middleware Conference*. ACM. 2016, p. 2.

[66] *Kubernetes – Production-Grade Container Scheduling and Management*. 2020. URL : https://github.com/kubernetes/kubernetes/ (visité le 06/02/2020).

[67] Patrick KUTCH. *Maintaining the Ethernet Link to the BMC During Server Power Actions*. Rapp. tech. Intel, 2012. URL : http://www-ssl.intel.com/content/dam/www/public/us/en/documents/guides/maintaining-the-ethernet-link-to-the-BMC.pdf (visité le 30/01/2020).

[68] *KVM*. 2020. URL : https://www.linux-kvm.org/page/Main_Page (visité le 21/01/2020).

[69] Jihye KWON et al. « Multicore scheduling of parallel real-time tasks with multiple parallelization options ». In : *RTAS*. IEEE. 2015, p. 232-244.

[70] Kelvin Roderick LAWRENCE. « Method and system for dynamically adjustable and configurable garbage collector ». B1 6 629 113 (US). 2003.

[71] Matthew LENTZ, James LITTON et Bobby BHATTACHARJEE. « Drowsy power management ». In : *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM. 2015, p. 230-244.

[72] Wubin LI et Ali KANSO. « Comparing containers versus virtual machines for achieving high availability ». In : *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, p. 353-358.

[73] Zhi LI et al. « Affinity-aware dynamic pinning scheduling for virtual machines ». In : *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. IEEE. 2010, p. 242-249.

[74] *Linux Containers - LXC - Introduction*. 2020. URL : https://linuxcontainers.org/lxc (visité le 14/12/2019).

[75] Pin LU et Kai SHEN. « Virtual Machine Memory Access Tracing with Hypervisor Exclusive Cache. » In : *Usenix Annual Technical Conference*. 2007, p. 29-43.

[76] David G LUENBERGER, Yinyu YE et al. *Linear and nonlinear programming*. T. 2. Springer, 1984.

[77] *lxc/libresource : CGroup aware resource querying library*. 2019. URL : https://github.com/lxc/libresource (visité le 04/02/2020).

[78] A Cameron MACDONELL. « Shared-memory optimizations for virtual machines ». In : (2011).

[79] Filipe MANCO et al. « My VM is Lighter (and Safer) than your Container ». In : *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, p. 218-233.

[80] Victor Marmol, Rohit Jnagal et Tim Hockin. « Networking in containers and container clusters ». In : *Proceedings of netdev 0.1* (2015).

[81] John D McCalpin et al. « Memory bandwidth and machine balance in current high performance computers ». In : *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2.19–25 (1995).

[82] David Meisner, Brian T Gold et Thomas F Wenisch. « PowerNap : eliminating server idle power ». In : *ACM SIGARCH Computer Architecture News* 37.1 (2009), p. 205-216.

[83] David Meisner et Thomas F Wenisch. « DreamWeaver : architectural support for deep sleep ». In : *ACM SIGPLAN Notices*. T. 47. 4. ACM. 2012, p. 313-324.

[84] *memcached - a distributed memory object caching system*. 2020. url : `https://memcached.org` (visité le 06/02/2020).

[85] *memtier_benchmark – NoSQL Redis and Memcache traffic generation and benchmarking tool*. 2020. url : `https://github.com/RedisLabs/memtier_benchmark` (visité le 06/02/2020).

[86] Xiaoqiao Meng et al. « Efficient resource provisioning in compute clouds via VM multiplexing ». In : *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, p. 11-20.

[87] *Microsoft Office 365 – Pack Office 2019 – Microsoft 365*. 2020. url : `https://products.office.com/` (visité le 17/01/2020).

[88] Grzegorz Miłó s et al. « Satori : Enlightened page sharing ». In : *Proceedings of the 2009 conference on USENIX Annual technical conference*. 2009, p. 1-1.

[89] Roberto Morabito. « Power consumption of virtualization technologies : an empirical investigation ». In : *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE. 2015, p. 522-527.

[90] Roberto Morabito, Jimmy Kj ä llman et Miika Komu. « Hypervisors vs. lightweight virtualization : a performance comparison ». In : *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, p. 386-393.

[91] Onur Mutlu et Thomas Moscibroda. « Parallelism-aware batch scheduling : Enhancing both performance and fairness of shared DRAM systems ». In : *ACM SIGARCH Computer Architecture News*. T. 36. 3. IEEE Computer Society. 2008, p. 63-74.

[92] Onur Mutlu et Thomas Moscibroda. « Stall-time fair memory access scheduling for chip multiprocessors ». In : *MICRO*. IEEE Computer Society. 2007, p. 146-160.

[93] Ryo Nakamura, Yuji Sekiya et Hajime Tazaki. « Grafting sockets for fast container networking ». In : *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*. 2018, p. 15-27.

[94] *nginx news*. 2020. URL : `https://nginx.org` (visité le 06/02/2020).

[95] Vlad Nitu et al. « Swift birth and quick death : Enabling fast parallel guest boot and destruction in the Xen hypervisor ». In : *ACM SIGPLAN Notices* 52.7 (2017), p. 1-14.

[96] C Norris, HM Cohen et B Cohen. « Leveraging IBM EX5 systems for breakthrough cost and density improvements in virtualized x86 environments ». In : *White paper* (2011).

[97] *Nutanix Enterprise Cloud - Run Any Application at Any Scale*. Nutanix. 2020. URL : `https://www.nutanix.com/` (visité le 28/01/2020).

[98] *Nutanix investor data sheet*. Nutanix. Oct. 2018. URL : `https://s21.q4cdn.com/380967694/files/doc_financials/2019/Q1/Nutanix-Q119-Earnings-Infographics_vFINAL_11272018.pdf` (visité le 27/01/2020).

[99] *Open source container-based virtualization for Linux*. 2020. URL : `https://openvz.org/` (visité le 21/01/2020).

[100] *OpenNebula – opennebula open-source cloud management platform*. 2020. URL : `https://opennebula.org/` (visité le 17/01/2020).

[101] *OpenStack - Build the future of Open Infrastructure*. OpenStack. 2020. URL : `https://www.openstack.org/` (visité le 28/01/2020).

[102] *Oracle VM VirtualBox*. 2020. URL : `https://www.virtualbox.org/` (visité le 21/01/2020).

[103] Chandandeep Singh Pabla. « Completely fair scheduler ». In : *Linux Journal* 2009.184 (2009), p. 4.

[104] Tapti Palit, Yongming Shen et Michael Ferdman. « Demystifying cloud benchmarking ». In : *2016 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2016, p. 122-132.

[105] Fawaz Paraiso et al. « Model-driven management of docker containers ». In : *2016 IEEE 9th International Conference on cloud Computing (CLOUD)*. IEEE. 2016, p. 718-725.

[106] Ben Pfaff et al. « The design and implementation of open vswitch ». In : *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, p. 117-130.

[107] *pm-graph | 01.org*. Intel. 2020. url : `https://01.org/pm-graph` (visité le 30/01/2020).

[108] *Pods – Kubernetes*. 2020. url : `https://kubernetes.io/docs/concepts/workloads/pods/pod/` (visité le 06/02/2020).

[109] Andrej Podzimek et al. « Analyzing the impact of cpu pinning and partial cpu loads on performance and energy efficiency ». In : *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE. 2015, p. 1-10.

[110] *QEMU*. 2020. url : `https://www.qemu.org/` (visité le 21/01/2020).

[111] Arun Raman et al. « Parallelism orchestration using DoPE : the degree of parallelism executive ». In : *PLDI*. 2011.

[112] Alessandro Randazzo et Ilenia Tinnirello. « Kata Containers : An Emerging Architecture for Enabling MEC Services in Fast and Secure Way ». In : *2019 Sixth International Conference on Internet of Things : Systems, Management and Security (IOTSMS)*. IEEE. 2019, p. 209-214.

[113] Joshua Reich et al. « Sleepless in Seattle No Longer. » In : *USENIX Annual Technical Conference*. 2010.

[114] Charles Reiss et al. « Heterogeneity and dynamicity of clouds at scale : Google trace analysis ». In : *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, p. 1-13.

[115] Yi Ren et al. « Shared-memory optimizations for inter-virtual-machine communication ». In : *ACM Computing Surveys (CSUR)* 48.4 (2016), p. 49.

[116] Rusty Russell. « virtio : towards a de-facto standard for virtual I/O devices ». In : *ACM SIGOPS Operating Systems Review* 42.5 (2008), p. 95-103.

[117] Tudor-Ioan Salomie et al. « Application level ballooning for efficient server consolidation ». In : *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, p. 337-350.

[118] Yuvraj Agarwal Stefan Savage et Rajesh Gupta. « Sleepserver : A software-only approach for reducing the energy consumption of PCs within enterprise environments ». In : *Power (KW)* 100.150 (2010), p. 200.

[119] Dominik Scholz. « A look at Intels dataplane development kit ». In : *Network* 115 (2014).

[120] Kyoung-Taek Seo et al. « Performance comparison analysis of linux container and virtual machine for building cloud ». In : *Advanced Science and Technology Letters* 66.105-111 (2014), p. 2.

[121] *Services de cloud computing | Google Cloud*. 2020. URL : `https://cloud.google.com/` (visité le 17/01/2020).

[122] *Services de cloud computing | Microsoft Azure*. 2020. URL : `https://azure.microsoft.com/` (visité le 17/01/2020).

[123] Prateek Sharma et al. « Containers and virtual machines at scale : A comparative study ». In : *Proceedings of the 17th International Middleware Conference*. 2016, p. 1-13.

[124] Andrew Silver. « Mostly idle at work ? Microsoft Azure has some bursty VMs it'd love to sell you ». In : *The Register* (2017). URL : `https://www.theregister.co.uk/2017/09/12/microsoft_azure_offers_bursty_vms`.

[125] Dinesh Subhraveti et al. « AppSwitch : Resolving the Application Identity Crisis ». In : *arXiv preprint arXiv:1711.02294* (2017).

[126] Yuqiong Sun et al. « Security namespace : making Linux security frameworks available to containers ». In : *27th USENIX Security Symposium (USENIX Security 18)*. 2018, p. 1423-1439.

[127] Kun Suo et al. « An analysis and empirical study of container networks ». In : *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, p. 189-197.

[128] *Swarm mode overview | Docker Documentation*. Docker. 2020. URL : `https://docs.docker.com/engine/swarm/` (visité le 31/01/2020).

[129] *systemd-nspawn*. 2020. URL : https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html (visité le 21/01/2020).

[130] Boris TEABE et al. « The lock holder and the lock waiter preemption problems : nip them in the bud using informed spinlocks (I-Spinlock) ». In : *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, p. 286-297.

[131] Guo-Song TIAN, Yu-Chu TIAN et Colin FIDGE. « High-precision relative clock synchronization using time stamp counters ». In : *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs 2008)*. IEEE. 2008, p. 69-78.

[132] *Universal TUN/TAP device driver*. 2020. URL : https://elixir.bootlin.com/linux/v4.19.34/source/Documentation/networking/tuntap.txt (visité le 06/02/2020).

[133] Gauthier VORON et al. « An interface to implement NUMA policies in the Xen hypervisor ». In : *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, p. 453-467.

[134] Carl A WALDSPURGER. « Memory resource management in VMware ESX server ». In : *ACM SIGOPS Operating Systems Review* 36.SI (2002), p. 181-194.

[135] Liang WANG et al. « Peeking behind the curtains of serverless platforms ». In : *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, p. 133-146.

[136] Wei WANG et al. « Design of Vhost-pci ». In : *KVM Forum*. 2016.

[137] *Windows VM | Workstation Pro | VMware*. 2020. URL : https://www.vmware.com/products/workstation-pro.html (visité le 21/01/2020).

[138] *wrk2 – A constant throughput, correct latency recording variant of wrk*. 2020. URL : https://github.com/giltene/wrk2 (visité le 06/02/2020).

[139] *Xen Project*. 2020. URL : https://xenproject.org/ (visité le 21/01/2020).

[140] Ethan G YOUNG et al. « The true cost of containing : A gVisor case study ». In : *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. 2019.

[141] Matei ZAHARIA et al. « Apache Spark : A Unified Engine for Big Data Processing ». In : *Commun. ACM* 59.11 (oct. 2016), p. 56-65.

[142] Jie Zhang, Xiaoyi Lu et Dhabaleswar K Panda. « Designing locality and NUMA aware MPI runtime for nested virtualization based HPC cloud with SR-IOV enabled InfiniBand ». In : *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2017, p. 187-200.

[143] Liang Zhang et al. « Picocenter : Supporting long-lived, mostly-idle applications in cloud environments ». In : *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 37.

[144] Qi Zhang et Ling Liu. « Workload adaptive shared memory management for high performance network I/O in virtualized cloud ». In : *IEEE Transactions on Computers* 65.11 (2016), p. 3480-3494.

[145] Jishen Zhao et al. « Buri : Scaling big-memory computing with hardware-based memory expansion ». In : *ACM Transactions on Architecture and Code Optimization (TACO)* 12.3 (2015), p. 31.

[146] Junji Zhi, Nilton Bila et Eyal de Lara. « Oasis : energy proportionality with hybrid server consolidation ». In : *Proceedings of the Eleventh European Conference on Computer Systems*. ACM. 2016, p. 10.

[147] Danyang Zhuo et al. « Slim : OS Kernel Support for a Low-Overhead Container Overlay Network ». In : *16th USENIX Symposium on Networked Systems Design and Implementation*. 2019, p. 331-344.