MemOpLight : Vers une consolidation mémoire pour les conteneurs grâce à un retour applicatif

Francis Laniel

THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

École Doctorale EDITE de Paris (*ED*130)
Informatique, Télécommunication et Électronique

Spécialité :
Ingénierie / Systèmes Informatiques

Présentée par :
Francis LANIEL

Pour obtenir le grade de :
DOCTEUR DE SORBONNE UNIVERSITÉ

MEMOPLIGHT : VERS UNE CONSOLIDATION MÉMOIRE POUR LES CONTENEURS GRÂCE À UN RETOUR APPLICATIF

Soutenue le 9 novembre 2020 devant le jury composé de :

Mme. Sonia BEN MOKHTAR	*Rapportrice*
M. Gaël THOMAS	*Rapporteur*
M. Thomas LEDOUX	*Examinateur*
M. Pierre SENS	*Examinateur*
M. Marc SHAPIRO	*Directeur de thèse*
M. Jonathan LEJEUNE	*Encadrant*
M. Julien SOPENA	*Encadrant*
M. Franck WAJSBURT	*Encadrant*

SORBONNE UNIVERSITÉ LIP6 Inria

REMERCIEMENTS

Ce manuscrit marque l'achèvement de mon doctorat mais aussi la fin d'une tranche de vie parisienne.

Concernant le doctorat, l'absence de proposition de titularisation à la suite de celui-ci est, pour moi, synonyme de précarité. Je doute sincèrement que les «chaires de professeur junior», proposées par la Loi de Programmation de la Recherche et instaurant la «pré-titularisation conditionnelle [sic]», apporte une solution à ce problème [121, Article 3][121, Alinéa 167 du Rapport Annexé]. Si cette même loi assure que «la rémunération des nouveaux contrats doctoraux sera progressivement revalorisée de 30% entre 2021 et 2023», la rémunération actuelle peut, entre autres, mener à des situations de mal-logement, problème dont Paris est, malheureusement, concernée [121, Alinéa 146 du Rapport Annexé][59, Onglet "Contexte régional"]...

Même si au cours ma thèse il me fut difficile de "regarder le ciel quand [j'avais] la tête dans le guidon", mon aventure à Paris fut néanmoins rythmée de nombreux événements et rencontres [140]. Certains auront peut-être marqué la France sans la changer fondamentalement... Tandis que d'autres m'auront permis d'avancer humainement parlant ! Vivre à Paris aura donc contribué à faire de moi la personne que je suis aujourd'hui.

Ainsi, je tiens, en premier lieu, à remercier les rapporteurs **Sonia Ben Mokhtar** et **Gaël Thomas** pour leurs évaluations de ce manuscrit. Je souhaite aussi adresser mes remerciements à **Thomas Ledoux** et **Pierre Sens** pour avoir été examinateurs. D'une manière plus générale, je remercie tous les membres du jury pour la discussion scientifique qui a suivi ma présentation et a permis de faire émerger des pistes d'améliorations novatrices.

Je n'aurai pu mener cette thèse à son terme sans les précieux conseils de mes encadrants. Je remercie donc **Marc Shapiro** pour avoir été le directeur de celle-ci et pour m'avoir conseillé, notamment quant à la manière de rédiger pour partager mes connaissances. Je veux aussi remercier **Jonathan Lejeune**, entre autres, pour son expertise sur les questions de qualité de service et **Franck Wajsburt** pour avoir amené un regard extérieur bienvenu lors de l'avancement des travaux de cette thèse. Je tiens à remercier **Julien Sopena** pour son investissement dans la thèse. Je ne compte plus le nombre de samedi passé au téléphone avec toi à peaufiner, un peu au dernier moment parfois, la rédaction d'un article. Tu es là pour tes doctorants et je te remercie donc pour ton engagement.

J'aimerai aussi remercier le personnel du LIP6 et de Sorbonne Université sans qui certaines tâches capitales, comme la signature du contrat doctoral, ne pourraient être réalisées : **Jean-Claude Bajard**, **Lelia Blin**, **Ophélie Da Costa**, **Bertrand Granado**, **Sylvie Gonçalves**, **Karine Heydemann**, **Irphane Khan**, **Fabrice Kordon**, **Marie Véronique Lam**, **Aliénor Le Conte**, **Shahin-Léa Mahmoodian**, **Aurore Marcos**, **Habib Mehrez**, **Jean-Luc Mounier**, **Belaïd Naït Sidnas**, **Diem Phuong Nguyen**, **Danny Richard**, **Samiha Taba** et **Sabrina Vacheresse**.

Tout au long de cette thèse, j'ai pu compter sur le soutien indéfectible de ma famille. Je vais donc remercier ma mère pour ses encouragements tout au long de cette aventure ainsi que mon père pour ses conseils. Je remercie aussi ma sœur pour son avis éclairé sur la pédagogie. Enfin, je souhaite remercier tout particulièrement un autre membre de ma famille qui m'a fait découvrir les ordinateurs et sans qui je n'aurai, sans doute, jamais été intéressé par l'informatique.

Mes amis ont toujours été là pour m'épauler tout au long de ce parcours. Les moments que nous avons passés ensemble resteront à jamais gravés dans ma mémoire et dans mon cœur ! Je remercie donc les amis d'ici : **Cyril**, **Guillaume**, **Jocelyn**, **Kevin** et **Nathan** ! Ainsi que ceux de Saint-Étienne : **Benoît**, **Fabien**, **Julien**, **Loïc**, **Marie D.**, **Marie G.**, **Michaël**, **Sammy**, **Théo D.**, **Théo L.**, **Thomas**, **Yohann** et **Youri** !

D'après la charte du doctorat de Sorbonne Université, celui-ci consiste en un "[t]ravail personnel réalisé dans un environnement collectif" [151]. Force est de reconnaître que le collectif aide beaucoup à surmonter la solitude de la tâche à affronter ! Les échanges que j'ai pu avoir avec mes collègues étaient parfois animés, nous n'étions pas tout le temps d'accord sur tout, mais surtout intéressants ! J'ai ainsi beaucoup appris de vous, que ce soit humainement ou scientifiquement. C'est pourquoi, je tiens à remercier mes collègues des équipes DELYS, MoVe et WHISPER : **Arnaud**, **Benoît**, **Daniel Wi.**, **Daniel Wl.**, **Darius**, **Dimitrios**, **Gabriel**, **Guillaume**, **Ilyas**, **Jonathan**, **Laurent**, **Lucas**, **Mathieu**, **Neha**, **Redha**, **Saalik**, **Sreeja**, **Vincent** et **Yoann**. La recherche française n'avancerait pas aussi vite sans vous alors gardez espoir et courage !

RÉSUMÉ

Le déploiement et l'exécution d'applications dans le *cloud* sont aujourd'hui une réalité. L'existence de celui-ci est intrinsèquement liée à celle de la virtualisation. Ce concept consiste à découper une machine physique en plusieurs sous-machines, dites machines virtuelles, isolées les unes des autres.

Plus récemment, les conteneurs se sont posés comme une alternative viable aux machines virtuelles. Les conteneurs sont plus légers que ces dernières et apportent les mêmes garanties d'isolation et de sécurité. Néanmoins, l'isolation, un conteneur ne peut affamer ses congénères, proposée est peut-être trop poussée. En effet, les mécanismes existants permettant l'isolation mémoire ne s'adaptent pas aux changements de charge de travail. Il n'est donc pas possible de consolider la mémoire, c'est-à-dire récupérer la mémoire inutilisée d'un conteneur pour en faire un meilleur usage.

Pour répondre à ce problème et garantir, à la fois, l'isolation et la consolidation mémoire, nous proposons MemOpLight. Ce mécanisme s'adapte aux changements de charge de travail grâce à un retour applicatif. Chaque conteneur indique donc s'il a de bonnes performances pour guider la répartition de la mémoire. Celle-ci est récupérée aux conteneurs ayant de bonnes performances pour que les autres conteneurs puissent augmenter celles-ci. L'idée étant de trouver un équilibre mémoire où tous les conteneurs ont des performances satisfaisantes.

MemOpLight permet d'augmenter la satisfaction des conteneurs de 13% comparé aux mécanismes existants.

ABSTRACT

Nowadays, deploying and executing applications in the cloud is a reality. The cloud can not exist without virtualization. This concept consists of slicing physical machines into several sub-machines, isolated from one another, known as virtual machines.

Recently, containers emerged as a viable alternative to virtual machines. Containers are lighter than virtual machines and bring the same isolation and security guarantees. Nonetheless, the isolation they offer is maybe too important. Indeed, existing mechanisms enforce memory isolation by ensuring that no container starves the others ; however, they do not adapt to changes in workload. Thus, it is impossible to consolidate memory, *i.e.* to reclaim memory unused by some containers to make a better use of it.

To answer this problem and ensure both isolation and consolidation, we introduce MemOpLight. This mechanism adapts to workload changes thanks to application feedback. Each container tells the kernel whether it has good or bad performance to guide memory reclaim. Memory is first reclaimed from containers with good performance in the hope that the others can improve their own performance. The idea is to find a balance where all containers have satisfying performance.

MemOpLight increases container satisfactions by 13% compared to existing mechanisms.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES ALGORITHMES

LISTE DES TABLEAUX

1

INTRODUCTION

Aujourd'hui, le *cloud* (l'informatique dans le nuage) est une réalité. Selon l'un des plus gros acteurs du marché, celui-ci peut être défini comme suit [15] :

> [...] la mise à disposition de ressources informatiques à la demande via Internet, avec une tarification en fonction de votre utilisation.

Les exemples d'entreprises, comme Netflix ou Airbnb, s'étant tournées, avec succès, vers le *cloud* ne manquent pas [147, 79, 16]. Si le *cloud* attire autant, c'est parce qu'il présente de multiples avantages pour le client qui, d'après ce même acteur[1], sont [15] :

- L'AGILITÉ : L'agilité permet d'innover plus rapidement grâce à la facilité d'accès aux ressources.
- L'ÉLASTICITÉ : Elle permet d'allouer la juste quantité de ressources nécessaires à la volée. Ainsi, il n'est plus besoin de sur-approvisionner, en continu, une application pour faire face à d'hypothétiques pics de requêtes.
- LA RÉDUCTION DES COÛTS : Le *cloud* permet d'économiser l'achat de serveurs physiques puisque la puissance de calcul est celle du fournisseur.
- LE DÉPLOIEMENT MONDIAL EN QUELQUES MINUTES : Les géants du *cloud* sont présents partout sur la planète [17]. Il devient donc plus facile pour une entreprise de conquérir de nouveau marché [79].

[1] *Il n'aura échappé à personne que cette entreprise capitaliste n'a aucun intérêt financier à être objective au sujet de son produit. Aucune technologie ne vient sans inconvénient et le cloud n'y échappe pas [101].*

Dans les années 2010, la plupart des géants du numérique ont revêtu la casquette de fournisseur *cloud* [2, 126, 114, 12, 64]. Amazon a initié le mouvement, en 2008, en rendant disponible pour tous son service de *cloud* : *Elastic Cloud Compute* (EC2) [9].

En théorie, le modèle économique du *cloud* est très simple[2]. Un client souhaitant exécuter son service dans le *cloud* choisit d'abord l'option qui correspond le mieux à ses besoins par rapport aux fonds dont il dispose. Une option correspond à un service logiciel offert par le fournisseur de *cloud*, comme une base de données, ou bien à des ressources physiques, il est alors commun de parler d'«instances» [7]. Dans un *cloud* dit *Infrastucture as a Service* (IaaS), une instance est un système d'exploitation virtualisé, ou système invité, auquel est alloué de la puissance de calcul, c'est-à-dire des processeurs, de la mémoire vive et du stockage [112]. Le client est ensuite facturé au temps passé, la tarification dépendant de l'instance choisie [10]. Par exemple, une instance disposant de peu de mémoire vive est bon marché comparée à une instance en ayant plus. Le service du client est ensuite exécuté sur l'un des serveurs physiques du fournisseur de *cloud* à côté de services appartenant à d'autres clients. La mise à disposition de ressources aux

[2] *Dans les faits, certains frais peuvent apparaître comme cachés, la vigilance du client est donc de mise [35]...*

clients par le fournisseur est garantie par «une entente de niveau de service», *Service-level Agreement* (SLA). Dans les faits, la plupart des SLA des fournisseurs de *cloud* garantissent un taux de disponibilité mensuelle d'environ 99,9% [117, 125, 6].

Le concept d'instance, qui est à la base du *cloud*, s'appuie sur la virtualisation. Celle-ci permet l'exécution de plusieurs systèmes d'exploitation, dits systèmes invités, sur un autre système d'exploitation, dit système hôte, tout en donnant l'impression aux systèmes invités qu'ils utilisent directement le matériel de la machine [128]. On parle alors de machine virtuelle (VM pour *virtual machine*). Le système invité et le système hôte sont complètement isolés, il n'est pas possible, sauf accord explicite, pour un système de récupérer les données de l'autre système [138].

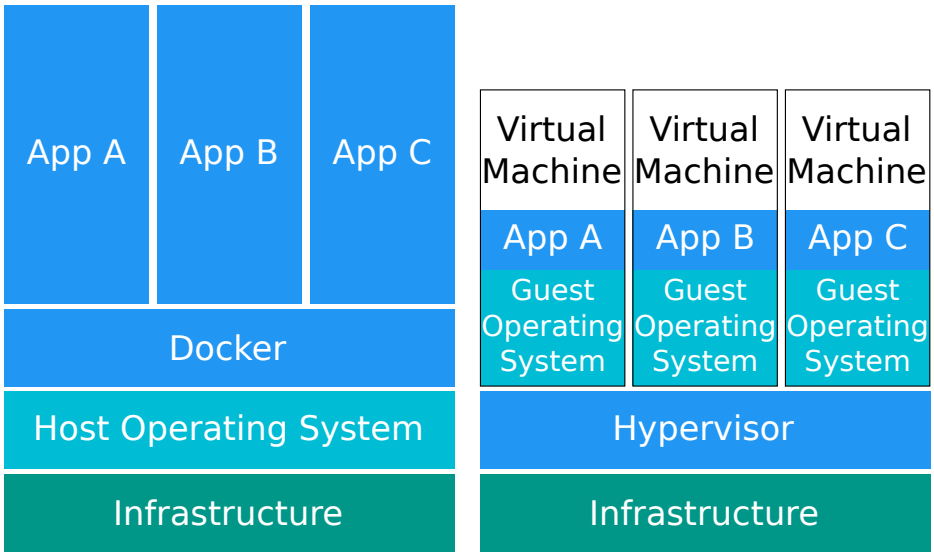
Du point de vue du client, la virtualisation garantit que son service n'est ni dérangé, le service d'un autre client ne pourra pas s'accaparer ses ressources, ni espionné, un autre client n'a pas connaissance des VM des autres clients. Cette isolation permet ainsi le respect de la SLA signée par le client et le fournisseur. Pour le fournisseur, la virtualisation permet l'*isolation* des différents services des clients tout en assurant la *consolidation*. Celle-ci consiste à exécuter plusieurs VM sur une même machine physique. Le fournisseur a d'ailleurs tout intérêt à maximiser le nombre d'instances sur une même machine physique pour tirer profit de celle-ci [1]. En effet, les VM des clients peuvent être assez légères tandis que les machines physiques utilisées dans le *cloud* disposent de processeurs puissants[3] offrant un haut niveau de parallélisme. Par exemple, si la machine physique dispose d'un processeur 4 cœurs et de 4 GB de mémoire vive il est parfaitement possible d'y exécuter deux instances s'étant vues allouer 2 cœurs et 2 GB de mémoire vive sans qu'elles ne se gênent mutuellement.

[3] *Les instances EC2 T3a s'appuient sur des processeurs AMD EPYC 7000 [18]. Certains AMD EPYC disposent de 64 cœurs et 128 threads comme l'EPYC 7702P [19].*

Plus récemment, les conteneurs se sont posés comme une alternative aux hyperviseurs pour la virtualisation [55]. Ceux-ci permettent, comme les premiers, d'isoler les applications tout en permettant un contrôle fin des ressources allouées à un conteneur [134]. Néanmoins, ils sont plus légers que les hyperviseurs, notamment car ils suppriment le système d'exploitation invité [163]. En effet, les applications «conteneurisées» s'exécutent directement au-dessus d'un système d'exploitation hôte comme le montre la Figure 1. Plusieurs logiciels permettent de lancer des conteneurs et de les gérer, l'un des plus connus est `docker` [52]. Amazon a rendu public son service d'exécution de conteneur dans AWS en 2015 [8]. La grande majorité des fournisseurs de *cloud* s'est adaptée et propose donc des services permettant d'exécuter des conteneurs sur leurs plateformes [13, 4, 115].

1. Amazon et Google proposent un type d'instance s'exécutant sur des machines peu chargées, par contre ces instances peuvent être arrêtées par le fournisseur à tout moment [11, 65]. Néanmoins, et Amazon l'assure, pour des instances normales, les ressources allouées à un client ne sont allouées qu'à ce client [142, Slide 14] :

> All resources assigned to you are dedicated to your instance with no over commitment



(a) Docker (b) Hyperviseur avec plusieurs VM

FIGURE 1 – Docker comparé à un hyperviseur

Pour gérer les ressources allouées, les conteneurs s'appuient sur des fonctionnalités offertes par le système d'exploitation hôte. Pour Linux, cette fonctionnalité porte le nom de `cgroup` [71]. Il existe un `cgroup` pour chaque type de ressource (nombre de cœurs, tranche de temps CPU, mémoire physique, bande passante disque, etc.). Il est donc possible, par exemple, de restreindre l'exécution d'une application à 2 cœurs et avec 2 GB de mémoire. Plus spécifiquement, le `cgroup` mémoire propose deux limites :

UNE LIMITE DITE «DURE» : Un conteneur ne peut pas allouer plus de mémoire que cette limite. S'il dépasse cette limite, le conteneur se voit réclamer sa mémoire aussitôt.

UNE LIMITE DITE «MOLLE» : Quand la mémoire physique disponible se fait rare, on parle alors de pression mémoire, le système d'exploitation hôte essaye d'assurer cette quantité de mémoire au conteneur. Néanmoins, cette limite vient sans aucune garantie de fonctionnement.

Ces deux limites garantissent l'isolation, puisqu'un conteneur ne pourra pas s'accaparer toute la mémoire au détriment de ses congénères. Elles se posent, par contre, en opposition totale avec la consolidation. Nous avons mené des expériences qui montrent que si un conteneur n'utilise pas sa mémoire, il n'est pas possible pour un autre conteneur de la lui réclamer afin d'en faire un meilleur usage. Avec une charge de travail dynamique, il y a donc nécessité d'une intervention humaine. Mais le client a l'illusion que son service s'exécute seul sur la machine et il n'a donc pas une vue globale du système comme l'a le fournisseur.

Dans cette thèse, notre but est d'apporter la consolidation aux conteneurs tout en garantissant une isolation nécessaire au respect de la SLA. Ceci pose plusieurs questions auxquelles nous allons répondre dans ce manuscrit :

1. Tout d'abord, pourquoi est-ce que les limites proposées par Linux n'arrivent pas à garantir et l'isolation et la consolidation ?
2. Ensuite, existe-t-il un mécanisme prenant en compte les fluctuations dans les charges de travail des applications tout en ne fixant pas de limite dure à leurs empreintes mémoires ?
3. Enfin, est-il possible, dans le même temps, de prendre en compte l'utilisation globale des ressources du système ?

Nous répondons à ces questions en proposant les contributions suivantes :

1. Dans un premier temps, nous montrons par l'expérience que les mécanismes actuels de limitation du `cgroup` mémoire permettent l'isolation mais pas la consolidation.
2. Dans un deuxième temps, il est possible d'utiliser une boucle autonomique en espace utilisateur indiquant au noyau le niveau de satisfaction des applications.
3. Dans un dernier temps, une seconde boucle autonomique, prenant place dans le noyau, permet de réclamer la mémoire des applications tout en ayant une vue globale de l'utilisation de la mémoire par celles-ci.

MemOpLight, notre contribution, permet de répondre aux deux derniers points. Ce mécanisme garantit le respect de la SLA tout en améliorant les performances des applications. Nous présentons notre mécanisme dans ce manuscrit dont le plan est le suivant :

CHAPITRE 2 : Ce chapitre présente un état de l'art des techniques permettant la consolidation ou le suivi des performances des applications. Dans celui-ci, sont abordées les différentes théories permettant de dimensionner aux mieux l'environnement d'exécution d'une application. Pour chacune de ces théories, des articles de recherche y faisant référence sont étudiés. Des travaux de recherche n'utilisant pas les approches susnommées sont examinés [14, 33].

CHAPITRE 3 : Ce chapitre porte sur la mémoire. Celle-ci est d'abord étudiée au niveau matériel en abordant la pyramide mémoire et le compromis entre taille et vitesse. Ensuite, la gestion de la mémoire par le système d'exploitation est décrite, notamment l'allocation et la réclamation de celle-ci. Puis, le concept de virtualisation est présenté en étudiant les hyperviseurs et les conteneurs.

CHAPITRE 4 : Dans la lignée du Chapitre 3, ce chapitre met en exergue l'impossibilité des conteneurs à garantir et l'isolation et la consolidation mémoire en montrant expérimentalement ce problème. Avant cela, les spécificités, dans la gestion mémoire, propres à ces derniers, sont détaillées.

CHAPITRE 5 : Ce chapitre présente MemOpLight. Notre mécanisme s'appuie sur deux composants :

— D'un côté, sur des contrôleurs applicatifs, propres à chaque conteneur, qui informent le noyau des performances de ces derniers

— et, d'un autre côté, sur un mécanisme de réclamation mémoire, intégré au noyau, se basant sur les informations collectées par les contrôleurs.

CHAPITRE 6 : Les performances de MemOpLight sont comparées aux mécanismes existants dans ce chapitre. Pour ce faire, nous réalisons une expérience reprenant le scénario d'exécution d'un site de commerce en ligne mais avec plus de conteneurs. Ensuite, nous étudions l'impact des différents paramètres de MemOpLight sur ses performances.

CHAPITRE 7 : Enfin, ce chapitre conclut nos travaux en les discutant et en apportant des pistes de travaux futurs.

2

DIMENSIONNEMENT DES RESSOURCES

Le dimensionnement consiste en l'ajustement des ressources nécessaires à l'application d'un client en fonction de ses besoins. Si elle fait face à un pic de requêtes, elle a besoin de plus de ressources pour y répondre. *A contrario*, si elle ne reçoit que peu de requêtes, elle n'a plus besoin de toutes ses ressources.

Pour un client, dimensionner correctement son ou ses instances lui permet d'économiser de l'argent. En effet, celui-ci est facturé pour les ressources qu'il utilise proportionnellement à la durée de leur utilisation [10]. Pour un fournisseur, le dimensionnement dynamique des applications pourrait lui rapporter de l'argent en facturant ce service.

Prenons l'exemple de l'entreprise, fictive, «P'tit Nuage» dont la plateforme *cloud* comprend une seule machine physique équipée d'un processeur 4 cœurs et de 4 GB de mémoire. Si le client A loue une instance possédant 3 cœurs et 3 GB de mémoire, «P'tit Nuage» ne pourra pas répondre au besoin du client B souhaitant exécuter son application avec 2 cœurs et 2 GB. «P'tit Nuage» pourrait, bien entendu, survendre ses ressources... Mais cette entreprise familiale refuse de voir son nom sali !

Elle a donc fait preuve d'ingéniosité en développant un système de dimensionnement automatique. Grâce à ce système, l'entreprise verra que le client A n'utilise pas toutes ses ressources, seulement 2 cœurs et 2 GB. Elle pourra donc accueillir l'instance du client B. Ainsi, l'utilisation de la machine physique de «P'tit Nuage» passera de 75% à 100%. L'entreprise aura aussi augmenté son nombre de clients et, puisque ses services sont payants, accru son chiffre d'affaires. Pour les clients, A aura vu sa facture baisser et B pourra exécuter son application dans le *cloud* [152]. «*Tout est au mieux*» dans le «*meilleur des mondes possibles*» !

Dans la suite de ce chapitre, nous étudions les solutions qu'aurait pu utiliser l'entreprise «P'tit Nuage» pour développer son système de dimensionnement automatique. Nous commençons par définir les deux types de dimensionnement existant, à savoir horizontal et vertical. Puis, nous nous concentrons sur les diverses théories permettant de décider le dimensionnement. Pour chacune de celles-ci, nous la détaillons avant de voir ses applications dans la recherche et/ou l'industrie. Nous continuons notre étude avec des travaux orignaux ne s'appuyant pas sur l'une des approches décrites précédemment.

2.1 LES DEUX TYPES DE DIMENSIONNEMENT

Une solution de dimensionnement cherche à dimensionner parfaitement, c'est-à-dire sans surdimensionner ni sous-dimensionner. Une telle solution doit aussi éviter les oscillations dans l'ajout ou le retrait

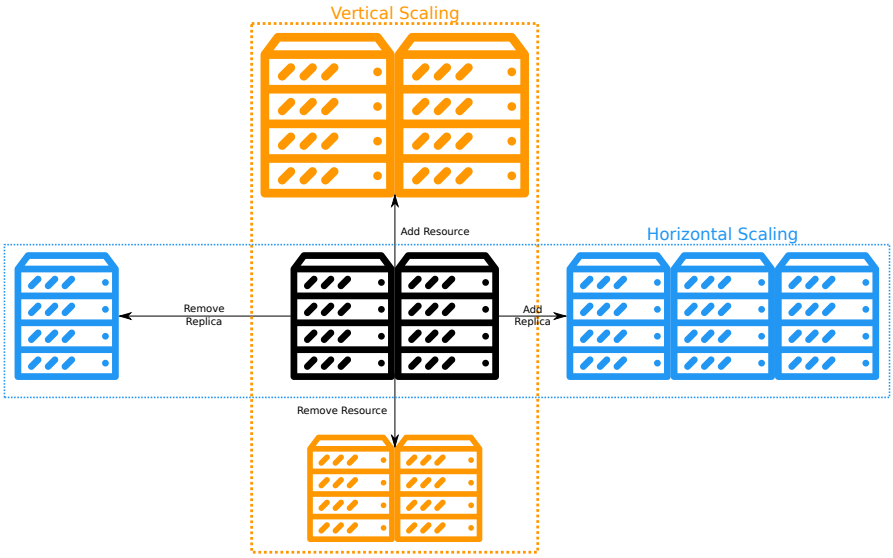


FIGURE 2 – Le dimensionnement horizontal et vertical [57, Figure 1]

de ressources. Le dimensionnement peut se faire selon deux axes, dépeints dans la Figure 2, qui sont :

LE DIMENSIONNEMENT HORIZONTAL : Ce dimensionnement consiste à ajouter ou enlever des réplicas, des VM ou des conteneurs, pour suivre au mieux la charge de l'application [57].

LE DIMENSIONNEMENT VERTICAL : Il revient à allouer plus, respectivement moins, de ressources à un réplica qui s'exécute déjà, par exemple, ajouter de la mémoire à un conteneur [57].

La solution que nous présentons dans cette thèse propose un dimensionnement vertical de la mémoire.

Une approche intuitive pour dimensionner consiste à se baser sur des seuils comme ajouter un réplica si la charge CPU est supérieure à n%. De telles solutions existent dans l'industrie, comme celle proposée par Kubernetes. Kubernetes est un logiciel, *open source*, permettant l'orchestration de conteneurs [92]. L'élément de base de Kubernetes est un *pod* qui est un ensemble de conteneurs. Les *pods* s'exécutent sur des nœuds qui sont des machines virtuelles ou physiques.

Ce logiciel intègre un mécanisme de dimensionnement dynamique [93]. Si un nouveau *pod* ne peut être lancé, par exemple car il nécessite plus de mémoire que ce qui peut lui être offert, un nouveau nœud est ajouté pour permettre l'exécution du *pod*. À l'inverse, un nœud non nécessaire pourra être supprimé. Un nœud est considéré ainsi, entre autres, quand la somme des requêtes d'utilisation de son processeur et de sa mémoire est inférieure à la moitié de sa capacité.

Google Cloud Scaling permet le dimensionnement horizontal de VM en fonction de la charge de travail [66]. Ce dimensionnement se base sur l'utilisation du processeur, le nombre de requêtes HTTP reçues ou bien des métriques personnalisées par l'utilisateur. Par exemple, si l'utilisateur a défini le taux d'utilisation du processeur comme étant 80%, le dimensionnement automatique ajoutera ou enlèvera des VM pour que cette métrique respecte le taux défini. Pour revoir le nombre de VM à la baisse, Google Cloud Scaling calcule le nombre de VM

cible en fonction de la charge maximale reçue sur les 10 dernières minutes. Si le nombre cible est inférieur au nombre courant de VM alors des VM seront supprimées.

Comme Google Cloud Scaling, le *cloud* Azure permet, par exemple, l'ajout d'une VM si l'utilisation moyenne du processeur est supérieure à 70% [118]. Ce *cloud* offre la possibilité de planifier les dimensionnements, il devient donc possible, par exemple, d'ajouter une VM les week-ends ou à certaines heures de la journée.

Kriushanth et Arockiam se basent sur un équilibreur de charge [90]. En fonction de seuils statiques, une VM pourra être enlevée ou ajoutée. La valeur du seuil peut aussi être modifiée au cours de l'exécution du système par une formule donnée par les auteurs.

Les solutions sus-citées fonctionnent, mais elles peuvent parfois manquer de réalisme du fait qu'elles sont basées sur des seuils. En effet, que signifie la charge CPU pour une application orientée mémoire ?

Pour pallier cela, il existe des théories permettant de fixer au mieux les seuils pour procéder à un dimensionnement. La première d'entre elles est l'autonomique, elle est basée sur une boucle de surveillance du système à dimensionner. Ainsi, en cas de changement dans la charge de travail, la boucle pourra dimensionner adéquatement. La théorie du contrôle permet, comme son nom l'indique, de contrôler un système. Elle est donc une parfaite candidate au dimensionnement d'applications s'exécutant dans le *cloud*. Une autre théorie, basée sur l'apprentissage par renforcement, peut aussi convenir au dimensionnement de telles applications. La principale force de cette méthode est qu'elle peut s'améliorer en apprenant de ce qui s'est passé. Si un événement ayant mené à un dimensionnement se reproduit, l'apprentissage pourra dimensionner le système de manière à répondre correctement à cet événement. Enfin, l'analyse des séries temporelles peut être utilisée comme un oracle afin de prédire les événements à venir et donc de dimensionner convenablement le système.

Dans les sections suivantes, nous examinons les théories susnommées. Pour chacune d'entre elles, nous détaillons son fondement théorique puis nous étudions comment elle est réellement utilisée. Nous avons choisi de les étudier indépendamment les unes des autres même si elles sont parfois imbriquées et parfois complémentaires.

2.2 L'AUTONOMIQUE APPLIQUÉE AU DIMENSIONNEMENT

L'entreprise IBM est pionnière dans l'autonomique, c'est elle qui a porté ce domaine au début des années 2000 [72]. L'autonomique est donc une spécialité relativement jeune.

Dans les sous-sections à venir, nous commençons par étudier l'élément de base de l'autonomique, à savoir la boucle autonomique, puis nous regardons ses utilisations dans la recherche.

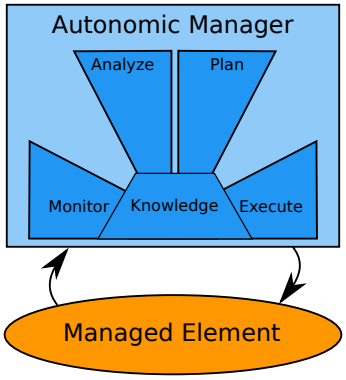


FIGURE 3 – Les quatre étapes de la boucle autonomique [85, Figure 2]

2.2.1 *Théorie de l'autonomique*

L'élément central de celle-ci, décrit dans la Figure 3, est la boucle autonomique.

Elle vise à observer un système donné pour prendre, et appliquer, les bonnes décisions pour son évolution. De base, cette boucle comporte quatre étapes que nous allons détailler :

MONITORING (OBSERVATION) : Dans cette étape, le système est observé. Plus précisément et dans le cadre du *cloud*, on peut imaginer que les performances de l'application d'un client sont mesurées. S'il s'agit d'un serveur web, cette étape peut revenir à calculer le 95^e centile de la latence des requêtes traitées sur la dernière seconde.

ANALYSIS (ANALYSE) : Les données collectées lors de l'étape M sont ici analysées. Dans notre exemple, la comparaison du centile obtenu avec une valeur de référence peut correspondre à cette étape.

PLAN (PLANIFICATION) : Selon le résultat de la phase A, une décision est prise et planifiée. Par exemple, si la latence des requêtes de notre serveur web s'avère supérieure à la valeur de référence, il faudra alors ajouter des ressources à notre serveur web. En effet, celui-ci ne semble pas s'exécuter avec les performances attendues.

EXECUTE (EXÉCUTION) : La décision prise dans la phase précédente est ici mise en application. Dans notre exemple, cette étape consisterait à ajouter des ressources pour accroître les performances de notre serveur web, par exemple, en lançant une nouvelle instance ou en ajoutant de la mémoire à une instance existante.

KNOWLEDGE (CONNAISSANCE) : La décision prise, suite à l'analyse des données collectées, est enregistrée afin d'être réutilisée plus tard. Chaque étape peut faire appel à cette connaissance, celle-ci n'est donc pas une étape en elle-même. Si la latence de notre serveur web passe, à nouveau, en dessous du seuil de référence, il faudra alors ajouter une nouvelle instance ou de la mémoire à une instance existante.

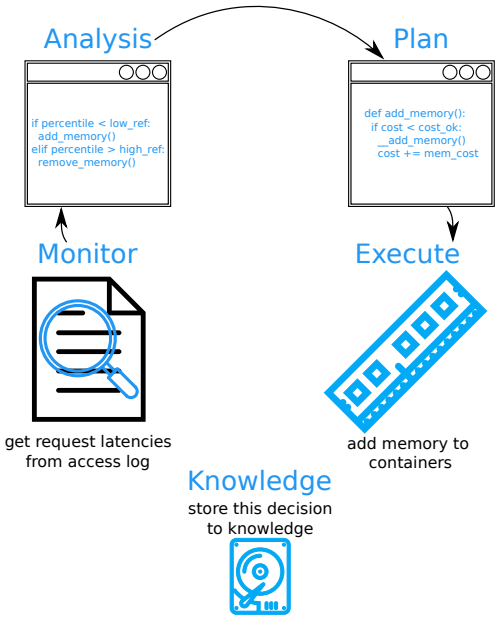


FIGURE 4 – La boucle autonomique appliquée à un serveur web s'exécutant dans le *cloud*

Dans la littérature, cette boucle est désignée sous l'appellation «boucle *MAPE-K*» d'après les initiales des différentes phases la composant. La Figure 4 applique la boucle autonomique à l'exemple détaillé.

2.2.2 *L'autonomique en pratique*

Pour Dupont *et al.*, il est trop compliqué, pour un humain, de gérer une application dans le *cloud* à cause de la multitude des paramètres [57]. Ils utilisent donc une boucle *MAPE-K*. Celle-ci peut ajouter/retirer une VM, mais aussi augmenter/dégrader le logiciel. Ainsi, la qualité du contenu offert par une plateforme de vidéo à la demande peut être abaissée pour faire face à un pic de charge. Néanmoins, une décision de dimensionnement est prise à partir de seuils.

Comme les auteurs précédents, Kouki et Ledoux s'appuient sur une telle boucle [89]. Mais, ils basent leur solution sur la gestion des configurations des applications s'exécutant dans le *cloud*. Pour les auteurs, une configuration est composée du nombre de tiers de l'application, et, pour chaque tiers, du nombre de réplicas ainsi que du niveau de parallélisme. Une configuration est jugée, par leur fonction d'utilité, optimale si elle maximise celle-ci.

Contrairement aux approches susnommées, la phase K de la solution de Maurer *et al.* s'appuie sur le *Case Base Reasoning* (CBR) [109]. Cette technique consiste à résoudre un cas en s'appuyant sur le passé : si le cas s'est déjà produit alors la solution est simplement réutilisée. Leur approche est donc proactive, elle essaie de prédire le futur, plutôt que réactive qui, elle, ne répond qu'à l'état courant du système [104].

2.2.3 *Conclusion*

Comme nous l'avons vu avec les articles susnommés, l'autonomique prend tout son sens dans le *cloud* pour aider au dimensionnement des applications. En effet, en surveillant souvent l'état du système il devient possible de faire face à des pics de charge en augmentant, que ce soit horizontalement ou verticalement, les ressources allouées au système.

Comme nous le verrons dans le Chapitre 5, notre solution s'inscrit dans la continuité de ces travaux puisqu'elle peut être vue comme une double boucle MAPE[4].

[4] *Nous n'avons pas indiqué le «K» car MemOpLight ne s'appuie pas une base de connaissances*

2.3 THÉORIE DU CONTRÔLE

La théorie du contrôle est un domaine large qui ne s'applique pas qu'à l'informatique [69]. Entre dans ce domaine tout système, soumis à des perturbations, dont la valeur de sortie doit être régulée à une valeur désirée, ce à l'aide d'un contrôleur d'entrée.

Pour l'informatique, Hellerstein donne l'exemple d'un administrateur système souhaitant fixer la charge de 3 serveurs à 66% chacun. Ainsi, en cas de crash, les deux serveurs restants pourraient absorber la charge.

Comme l'autonomique, cette théorie se base sur une boucle. Néanmoins, les différents éléments la composant diffèrent de ceux présents dans la boucle autonomique. Dans les sous-sections à venir, nous présentons les éléments de la théorie du contrôle puis nous examinons les articles de recherche ayant trait à cette théorie.

2.3.1 *Définition de la théorie du contrôle*

La théorie du contrôle s'appuie sur une boucle de rétroaction dont les éléments sont, d'après Hellerstein, les suivants :

LE SYSTÈME SURVEILLÉ : C'est le système dont on souhaite réguler la sortie. Dans notre exemple, le système surveillé sera un serveur web.

LA SORTIE : C'est la valeur mesurée en sortie du système surveillé, par exemple, la bande passante du serveur web.

LA RÉFÉRENCE : La sortie doit être égale à cette valeur. L'administrateur système souhaite que la bande passante de son serveur soit égale à $1.5\,\mathrm{GB\,s^{-1}}$.

L'ERREUR : L'erreur correspond à la différence entre la référence et la sortie. Si l'erreur du serveur web est positive, ceci implique que celui-ci a des performances inférieures à la référence.

L'ENTRÉE : L'entrée est le paramètre affectant le comportement du système surveillé, elle peut être ajustée dynamiquement. Par exemple, le nombre de *threads* qu'un serveur web gère à un moment donné.

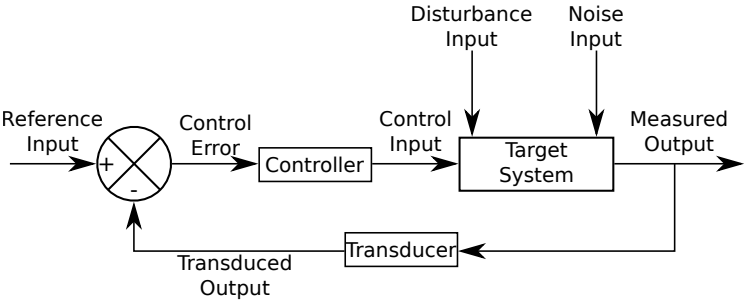


FIGURE 5 – Schéma bloc d'une boucle de rétroaction [69, Figure 1.1]

CONTRÔLEUR : Ce composant modifie l'entrée, en se basant sur les valeurs passées et actuelles de l'erreur, afin d'obtenir la valeur de référence. En reprenant notre exemple, il est possible d'imaginer que plus l'erreur est grande plus il faudra augmenter le nombre de *threads* pour espérer obtenir une bande passante égale à la référence.

LA PERTURBATION : Elle représente n'importe quel élément affectant l'influence de l'entrée sur la sortie mesurée. Dans notre exemple, la température pourrait être une perturbation, car elle forcerait le processeur à diminuer sa fréquence et donc ses performances.

LE BRUIT : Il change la valeur mesurée pour la sortie. Si la bande passante est mesurée en valeur flottante, l'erreur accumulée au fil des opérations peut être considérée comme du bruit.

LE TRANSDUCTEUR : Ce composant transforme la valeur de la sortie pour qu'elle soit comparée avec la référence. Par exemple, la bande passante en sortie peut être donnée en $\mathrm{MB\,s^{-1}}$ alors que la valeur de référence est donnée en $\mathrm{GB\,s^{-1}}$. Il faudra donc transformer la sortie pour la comparer à l'entrée.

La Figure 5 dépeint une boucle de rétroaction contenant les éléments décrits ci-dessus. Malgré les différences avec ceux présentés dans la Figure 3, il nous semble possible d'établir un parallèle entre ces deux boucles :

MONITOR : Dans la boucle autonomique, *la sortie* est collectée dans cette étape.

ANALYSIS : Cette phase peut comporter *un transducteur* pour changer l'unité de la sortie pour pouvoir effectuer une comparaison. Elle va aussi calculer *l'erreur*, la différence entre *la sortie* et *la référence*, en prenant idéalement en compte *la perturbation* et *le bruit*.

EXECUTE : Cette phase agit directement sur *l'entrée* en modifiant sa valeur pour que le système réagisse correctement à l'événement. Elle agit donc comme *le contrôleur* de la théorie du contrôle

2.3.2 *La théorie du contrôle en pratique*

Malrait *et al.* s'intéressent, comme Kouki et Ledoux, à l'impact de la configuration d'un serveur sur le compromis entre performance

et disponibilité [108]. Mais contrairement aux seconds, Malrait et ses co-auteurs s'appuient sur la théorie susmentionnée, car ils définissent plusieurs lois de contrôle. La première vise à maximiser la disponibilité pour une latence donnée en ajustant le nombre de *threads* que le serveur peut gérer. La deuxième loi tente de minimiser la latence pour un taux d'abandon donné, là aussi, en réglant le nombre de *threads*.

Dans des travaux plus récents, Malrait *et al.* étendent leur solution en ajoutant deux nouvelles lois [107]. Celles-ci viennent combler les faiblesses des premières puisque leurs performances sont, au pire, égales à celles-ci. Ainsi, en variant le nombre de *threads*, les auteurs peuvent dimensionner au mieux afin de minimiser la latence et le taux d'abandon des requêtes.

Berekmeri et ses co-auteurs s'intéressent aussi à la théorie du contrôle [26]. Ils proposent donc un modèle qu'ils utilisent dans leurs contrôleurs. Celui-ci permet de calculer le temps de traitement d'une requête en fonction du nombre de nœuds et de clients. Plutôt que de jouer sur le nombre de *thread*, comme Maltrait *et al.*, pour garantir un temps de traitement des requêtes, Berekmeri *et al.* font varier le nombre de nœuds alloués au système pour minimiser le temps de traitement des requêtes.

De Cicco *et al.* ciblent, comme Kriushanth et Arockiam, un équilibreur de charge auquel ils appliquent la théorie du contrôle [48]. Celui-ci route les requêtes des clients vers les VM les moins chargées. Périodiquement, la charge des VM est analysée pour décider si un dimensionnement doit avoir lieu, leur finalité consistant à minimiser le nombre de VM tout en maximisant la bande passante de serveurs de flux vidéos.

2.3.3 *Conclusion*

Dimensionner correctement un système revient à le contrôler. Il est donc naturel que la théorie du contrôle s'exprime pleinement dans ce domaine.

Dans le Chapitre 5, nous verrons que même si notre mécanisme ne comporte pas tous les éléments attendus par la théorie du contrôle, il existe des similitudes entre ce domaine et notre solution.

2.4 APPRENTISSAGE PAR RENFORCEMENT

Pour Sutton et Barto, l'apprentissage par renforcement est un sous domaine de l'apprentissage machine [145]. L'apprentissage par renforcement consiste à apprendre comment associer des actions à des situations, et ce pour maximiser une récompense. Pour ce faire, un agent apprenant va devoir exploiter ce qu'il a appris tout en explorant de nouvelles pistes pour améliorer ses connaissances.

Contrairement aux deux approches susmentionnées, les fondations de l'apprentissage par renforcement ne reposent pas sur une boucle. Nous développons donc, dans les sous-sections suivantes, les éléments

de base de cet apprentissage avant d'étudier ses applications dans la recherche et l'industrie.

2.4.1 *Théorie de l'apprentissage par renforcement*

Un système d'apprentissage par renforcement comporte plusieurs éléments [145, Section 1.3] :

AGENT APPRENANT : C'est l'élément de base du système d'apprentissage par renforcement. Nous pouvons imaginer qu'un serveur web s'exécutant dans le *cloud* peut être étendu pour apprendre.

ENVIRONNEMENT : L'agent évolue dans un environnement. Dans notre exemple, le serveur web évolue dans un environnement auquel appartiennent aussi les requêtes qu'il reçoit.

POLITIQUE : Une politique peut être abstraite comme l'association entre des actions à effectuer et des états de l'environnement. Par exemple, si le serveur web fait face à un pic de requêtes, il faudra lui ajouter des ressources.

RÉCOMPENSE : En retour de ces actions l'agent reçoit une valeur de récompense. Par ses actions, il devra maximiser celle-ci. L'action prise par une politique pourra être changée si elle a mené à une faible récompense. Dans notre exemple, la récompense consiste en l'inverse de la latence des requêtes.

FONCTION DE VALEUR : Là où la récompense consiste en une évaluation immédiate de l'action, la fonction de valeur indique la satisfaction sur la durée. Ainsi, dans un environnement *cloud*, diminuer le coût peut être une fonction de valeur. Par exemple, le système a décidé de retirer des ressources au serveur, ceci a mené à une faible récompense, car la latence des requêtes a augmenté. Mais sur le long terme, cette action a permis de diminuer le coût, la fonction de valeur renvoie donc une valeur conséquente.

MODÈLE DE L'ENVIRONNEMENT : Il modélise l'environnement afin de prédire l'état futur de celui-ci en fonction de son état courant et de l'action qui a été effectuée. À noter qu'un système d'apprentissage par renforcement ne comporte pas toujours de modèle. Par contre, l'absence de celui-ci consiste en une solution purement réactive. Dans notre exemple, le modèle pourrait prédire qu'ajouter des ressources au serveur lors d'un pic de charge mènera à une diminution de la latence des requêtes.

À partir de ces éléments, il est possible d'utiliser différents modèles mathématiques pour implémenter un agent apprenant. Ces modèles sont très divers les uns des autres, mais sont souvent probabilistes, comme les bandits manchots ou les méthodes de Monte-Carlo.

Comme pour la théorie du contrôle, il nous semble aussi possible de réaliser une correspondance du modèle des bandits manchots avec la boucle *MAPE-K* [145, Chapter 2] :

MONITOR : L'observation de la valeur obtenue suite à une action est réalisée dans cette étape. Une action est, par exemple, l'actionnement du levier d'une certaine machine à sous. La valeur associée à cette action correspond alors au gain obtenu suite au fait d'avoir joué.

ANALYSIS : Dans cette étape, l'action à réaliser est choisie. Son choix se base sur une valeur estimée calculée, pour chaque action, comme la moyenne obtenue suite à plusieurs exécutions de celle-ci. Un algorithme simple pour réaliser ce choix consiste en la sélection de l'action ayant la valeur estimée maximale.

EXECUTE : Dans le modèle des bandits manchots, cette phase correspond à l'exécution de l'action précédemment choisie. Dans notre exemple, cette phase effectue donc l'actionnement du levier de la machine à sous connue pour maximiser la valeur.

KNOWLEDGE : La connaissance acquise par un modèle de bandit manchots peut être exploitée, si l'action choisie est celle ayant la valeur maximale. Mais elle peut aussi être explorée dans le cas où l'action choisie ne maximise pas la valeur. Le résultat d'une exploration peut remplacer l'action maximisant la valeur. Si cette nouvelle action est utilisée, le modèle réalise alors une exploitation. Il existe donc un conflit entre l'exploitation et l'exploration puisqu'une action ne peut être les deux à la fois.

2.4.2 *L'apprentissage par renforcement en pratique*

Rao et ses coauteurs font état de la complexité à établir un lien entre l'utilisation des ressources et la performance des applications, plus particulièrement pour la mémoire [135]. En effet, si la mémoire totale utilisée par une application est inférieure à la taille de son *Working Set* [1] (WS) alors ses performances seront moindres [50]. De plus, estimer la taille du WS n'est pas aisé [68]. Lorsqu'iBalloon, la solution des auteurs, prend une décision de dimensionnement, celle-ci donne lieu à une récompense. Celle-ci punit un paramètre ne respectant pas la SLA tout en essayant de maximiser l'usage des ressources. La décision sera aussi apprise grâce à cette réaction au moyen d'un apprentissage basé sur l'apprentissage par renforcement.

Amazon propose une solution de dimensionnement visant à optimiser la disponibilité ou le coût d'une application, ou encore, un équilibre des deux [23, 14]. Comme la solution de Rao *et al.*, ce dimensionnement est basé sur l'apprentissage automatique. Il analyse jusqu'à 14 jours d'historique d'une métrique pour prédire sa valeur pour les 2 jours à venir.

1. Peter J. Denning définit ainsi le WS [50] :

   a working set of pages is the minimum collection of pages that must be loaded in main memory for a process to operate efficiently

2.4.3 *Conclusion*

Comme la boucle *MAPE-K*, l'apprentissage par renforcement est une solution viable pour dimensionner des applications s'exécutant dans le *cloud*. Comme la première, elle permet de réagir à un cas déjà observé. Le système de récompense autorise aussi à corriger les erreurs de décision du système afin d'améliorer celles-ci.

La solution décrite dans le Chapitre 5 de cette thèse n'intègre pas d'apprentissage. Néanmoins, cette faculté pourrait lui être apportée afin d'améliorer ses performances.

2.5 SÉRIE TEMPORELLE

Une série temporelle est une séquence de valeurs observées séquentiellement au cours du temps [30]. La finalité étant de prédire, en se basant sur les séries temporelles observées à l'instant t ou antérieures, les valeurs qui seront observées à l'instant $t + l$ où $l \geq 1$. Théoriquement, les séries temporelles s'appuient sur des fonctions de prédiction pour chaque délai l. Cette prédiction devra minimiser l'erreur quadratique moyenne entre la valeur générée et la valeur mesurée.

Cette théorie tranche, elle aussi, avec les précédentes dans le sens où elle est très mathématique. De plus, comparée à la boucle autonomique, son étude porte sur des éléments ayant eu lieu bien avant l'itération courante. Nous détaillons donc ses spécificités avant de nous attaquer à son emploi dans différents travaux de recherche.

2.5.1 *Théorie de la série temporelle*

Plusieurs modèles existent permettant de générer la valeur d'une série temporelle à l'instant t. Ceux-ci sont dits stationnaires, si les valeurs générées varient autour d'une certaine moyenne et avec un écart-type donné, ou, dans le cas contraire, non stationnaires.

L'un des modèles stationnaires, dit *autoregressive* (AR), consiste à exprimer la valeur actuelle z_t comme une somme linéaire des précédentes valeurs du modèle, z_{t-n} où $n \geq 1$ et d'un choc aléatoire a_t. La Formule 1 exprime $\tilde{z}_t = z_t - \mu$, soit la série de dérivation par rapport à la moyenne μ, pour le modèle AR [30, Équation 1.2.2] :

$$\tilde{z}_t = \phi_1 \tilde{z}_{t-1} + \phi_2 \tilde{z}_{t-2} + ... + \phi_p \tilde{z}_{t-p} + a_t \tag{1}$$

À la suite de raisonnements, il apparaît que $\tilde{z}_t$ peut être exprimé uniquement en fonction de a_t et a_{t-j} où $1 \leq j \leq m$.

Un autre modèle stationnaire, dit *moving average* (MA), exprime la série de dérivation $\tilde{z}_t$ en fonction d'un certain nombre de chocs aléatoires. Ce modèle est représenté par la Formule 2 [30, Équation 1.2.3] :

$$\tilde{z}_t = a_t - \theta_1 a_{t-1} - \theta_2 a_{t-2} - ... - \theta_q a_{t-q} \tag{2}$$

Le modèle stationnaire, dit *autoregressive-moving average* (ARMA), combine les modèles précédant pour calculer $\tilde{z}_t$. Son équation est donnée par la Formule 3 [30, Équation 1.2.4] :

$$\tilde{z}_t = \phi_1 \tilde{z}_{t-1} + ... + \phi_p \tilde{z}_{t-p} + a_t - \theta_1 a_{t-1} - ... - \theta_q a_{t-q} \quad (3)$$

Le modèle non stationnaire, dit *autoregressive integrated moving average process* (ARIMA), est une variante du modèle ARMA. Ce modèle s'appuie sur $w_t = \nabla^d z_t$, où ∇^d est l'opérateur de différence arrière défini par $\nabla^d = z_t - ... - z_{t-d}$ [30, Section 1.2.1, Équation 1.2.6]. Son équation est livrée dans la Formule 4 [30, Équation 1.2.7].

$$\tilde{z}_t = \phi_1 w_{t-1} + ... + \phi_p w_{t-p} + a_t - \theta_1 a_{t-1} - ... - \theta_q a_{t-q} \quad (4)$$

Contrairement aux autres méthodes, nous ne pensons pas que puisse être établi un parallèle entre la boucle *MAPE-K* et les séries temporelles. Néanmoins, celles-ci peuvent être utilisées lors de la phase *Analysis* pour décider d'une action à effectuer.

2.5.2 *Série temporelle en pratique*

Comme De Cicco *et al.*, Li et Xia s'appuient sur un équilibreur de charge afin de router la requête vers la plateforme *cloud* la plus proche géographiquement [102]. Contrairement à De Cicco et ses co-auteurs, leur solution analyse l'historique des requêtes pour prédire, au moyen du modèle ARMA, le nombre de requêtes pour la prochaine période. À partir de cette prédiction, le nombre de conteneurs s'exécutant dans le *cloud* est calculé.

Jing-Gang et ses associés s'intéressent, comme Li et Xia, aux conteneurs, via les *pods* `kubernetes` [82]. Ici aussi, une prédiction sur le nombre de requêtes de la prochaine période est réalisée. Mais celle-ci se base sur le modèle ARIMA et non ARMA comme utilisé par Li et Xia. Si la différence entre la prédiction et la réalité est supérieure à un seuil donné, des *pods* sont ajoutés.

2.5.3 *Conclusion*

Comme le montrent les travaux de Li et Xia ainsi que ceux de Jin-Gang *et al.*, l'analyse de série temporelle peut être utilisée pour prédire la charge reçue par une application. En se basant sur cette prédiction, il devient possible de dimensionner les ressources allouées à l'application pour qu'elle réponde au mieux aux requêtes à venir.

L'approche proposée dans cette thèse, MemOpLight, se base sur un historique réduit contrairement à ce que prévoient ces théories. Nous pourrions étendre notre solution avec celles-ci, notamment car elles semblent légères.

2.6 APPROCHES ORIGINALES

Dans cette section, nous présentons des solutions qui ne font pas appel aux théories étudiées précédemment. Ces solutions sont néanmoins tout autant intéressantes.

Ge *et al.* s'intéressent aux *pods* `kubernetes` avec des charges de travail MapReduce [63]. Ils fixent le nombre de *pods* associés à chaque tâche via des formules. D'après leurs expériences, il est plus intéressant d'instancier le nombre de *pods* au début des calculs que pendant ceux-ci, car l'instanciation entrave les performances.

Le *memory ballooning* est une technique permettant de réclamer de la mémoire à une VM pour l'allouer à une autre. Capitulino présente un projet visant à rendre le *ballooning* automatique [133, 94, 32]. L'automatisation de ce mécanisme repose sur des événements de pression mémoire basés sur des seuils statiques.

Kim et ses co-auteurs constatent que les solutions actuelles de *memory ballooning* présentent un surcoût trop important [88]. Pour cela, ils proposent donc un *memory ballooning* conscient de la pression mémoire. Ils définissent alors trois seuils de pression mémoire.

Pour Carver *et al.*, et comme le montre le *memory ballooning*, il est plus compliqué de récupérer des ressources à une VM qu'à un conteneur [34]. Ils ont modifié la réclamation mémoire de Linux de sorte que les conteneurs soient réclamés chronologiquement. Bien entendu, cette réclamation s'arrête lorsqu'une certaine quantité a été récupérée.

Comme Carver *et al.*, notre solution cible les conteneurs. En effet, nous avons constaté un problème, que nous détaillons dans le Chapitre 4, avec ceux-ci. MemOpLight permet d'adresser ce problème.

2.7 APPLICABILITÉ DANS LE NOYAU

Dans cet état de l'art, nous avons vu de nombreux travaux de recherche s'intéressant au dimensionnement. Ceux-ci s'appuient sur plusieurs domaines de recherche qui permettent d'adapter les ressources à la charge. Nous synthétisons les solutions étudiées dans le Tableau 1. Pour chaque solution, nous avons indiqué si elle utilise une approche réactive ou proactive, l'espace d'exécution de la solution, le dimensionnement proposé et la théorie sur laquelle elle se base.

Nous allons maintenant commenter l'applicabilité des solutions évoquées dans le noyau. Nous procédons selon deux critères :

1. La complexité : Les solutions présentées sont souvent trop complexes pour être incluses dans le noyau.
2. L'espace d'exécution : Une solution s'exécutant en espace utilisateur permet d'avoir des métriques plus proches du ressenti de l'application, mais empêche d'avoir une vision globale du système.

Nous détaillons donc ces deux points dans les sous-sections suivantes.

2.7.1 *Complexité des solutions*

La plupart des approches présentées siègent en espace utilisateur. Il est compliqué de les transposer dans le noyau, car celui-ci limite la complexité des algorithmes. En effet, le noyau s'exécute, par exemple lors d'un appel système, mais pas seulement, en ayant interrompu un

TABLE 1 – Tableau récapitulatif des différentes solutions évoquées dans cet état de l'art

Auteurs	Approche	Espace d'exécution	Dimensionnement horizontal	Dimensionnement vertical	Théorie utilisée
Kubernetes [93]	réactive	utilisateur	oui	non	Seuil
Google Cloud Scaling [66]	réactive	?	oui	non	Seuil
Microsoft Azure Scaling [118]	réactive	?	oui	non	Seuil
Kriushanth *et al.* [90]	réactive	utilisateur	oui	non	Seuil
Dupont *et al.* [57]	réactive	utilisateur	oui	logiciel	Autonomique avec seuils
Kouki *et al.* [89]	réactive	utilisateur	oui	non	Autonomique
Maurer *et al.* [109]	proactive	utilisateur	oui	non	Autonomique
Malrait *et al.* [108, 107]	réactive	utilisateur	non	logiciel	Théorie du contrôle
Berekmeri *et al.* [26]	proactive	utilisateur	oui	non	Théorie du contrôle
De Cicco *et al.* [48]	réactive	utilisateur	oui	non	Théorie du contrôle
Rao *et al.* [135]	réactive	hyperviseur	non	oui	Apprentissage par renforcement
AWS Auto Scaling [23, 14]	proactive	?	oui	non	Apprentissage automatique
Li *et al.* [102]	proactive	utilisateur	oui	non	Série temporelle
Jin-Gang *et al.* [82]	proactive	utilisateur	oui	non	Série temporelle
Ge *et al.* [63]	réactive	utilisateur	oui	non	Approche originale
Auto ballooning [94, 32]	proactive	noyau	non	mémoire	Approche originale
Kim *et al.* [88]	proactive	noyau et hyperviseur	non	mémoire	Approche originale
Carver *et al.* [34]	proactive	noyau	non	mémoire	Approche originale

code utilisateur [29, Chapitre 7]. La durée de l'interruption doit donc être minimisée, il n'est par conséquent pas possible de bénéficier de toute la richesse proposée par ces approches.

Le noyau possède aussi d'autres limites. Par exemple, l'utilisation d'opérations flottantes dans celui-ci est déconseillée. En effet, sur une architecture x86_64, il faut sauvegarder les registres spécifiques à l'unité de calcul flottant pour ne pas corrompre son état. Cette sauvegarde a malheureusement un coût [149, fichier `arch/x86/kernel/fpu/core.c`, ligne 127]. Or, l'apprentissage par renforcement est fortement basé sur l'utilisation de calcul flottant [145, page 38]. Un mécanisme noyau d'apprentissage par renforcement verrait donc ses performances amoindries par ce coût.

2.7.2 *Espace d'exécution*

Les solutions que nous avons présentées agissent dans différents espaces d'exécutions. Pour nous, il existe deux espaces d'exécution qui sont :

ESPACE UTILISATEUR : Les solutions s'exécutent en espace utilisateur.

ESPACE NOYAU/HYPERVISEUR : Les mécanismes prennent place dans le noyau ou l'hyperviseur.

Dans la suite de cette section, nous pesons le pour et le contre de chacun de ces espaces.

Espace utilisateur

Comme le montre le Tableau 1, la plupart des techniques de cet état de l'art s'exécutent en espace utilisateur. L'avantage de cet espace est qu'il permet l'exécution de solutions complexes. De plus les métriques collectées sont représentatives des performances des applications puisqu'elles sont générées par les applications.

Néanmoins, l'espace utilisateur manque de recul. En effet, une solution s'exécutant dans cet espace n'a pas connaissance de l'utilisation de toutes les ressources de la machine. De plus, il ne peut pas non plus savoir si la machine entière est en pression mémoire. Ainsi, il n'est pas possible de savoir s'il faut rendre certaines ressources qui ont été allouées.

Espace noyau/hyperviseur

Comparé à l'espace utilisateur, celui-ci permet d'avoir une vision d'ensemble de l'utilisation des ressources de la machine. Le noyau, ou hyperviseur de type 1, est aussi capable de savoir s'il y a pression mémoire puisque c'est lui qui est responsable de sa gestion.

Par contre, les métriques collectées sont bas niveau (nombre de cycles CPU, utilisation de la mémoire, bande passante disque, etc.). Il est donc complexe d'inférer l'état de performance des applications à partir de celles-ci.

Espace d'exécution \ Caractéristiques	Vision	Métriques
Utilisateur	locale	**proches des performances de l'application**
Noyau/Hyperviseur	**globale**	éloignées des performances de l'application
Double	**globale**	**proches des performances de l'application**

TABLE 2 – Tableau récapitulatif des avantages et inconvénients des espaces d'exécution évoqués

2.8 CONCLUSION

Dans cet état de l'art, nous avons étudié différentes solutions dont les caractéristiques principales sont compilées dans le Tableau 1. La plupart de celles-ci offrent un dimensionnement horizontal prenant place en espace utilisateur.

Nous avons aussi synthétisé les avantages et inconvénients offerts par les espaces utilisateur et noyau/hyperviseur. Pour pallier aux problèmes de ces deux espaces tout en profitant de leurs avantages, nous proposons une solution s'exécutant dans les deux espaces. L'espace utilisateur est utilisé pour profiter des métriques proches des applications tandis que l'espace noyau offre une vision globale du système. Le Tableau 2 donne les avantages et inconvénients de ces différents espaces.

Avant de présenter MemOpLight, il nous faut présenter les bases techniques sur lesquelles ce mécanisme s'appuie. Ainsi, le Chapitre 3 est dédié à l'explication des prérequis techniques permettant d'appréhender notre solution.

3 LA MÉMOIRE

3.1 INTRODUCTION

Dans ce chapitre, nous allons étudier la mémoire telle qu'utilisée en informatique. La mémoire est définie ainsi par le wiktionnaire [154] :

> Capacité à retenir, conserver et rappeler de nombreuses informations antérieures.

Un ordinateur a notamment besoin d'une telle capacité afin de se rappeler l'état courant des données manipulées ainsi que les instructions à exécuter.

Les différents composants d'un ordinateur possèdent presque tous un dispositif permettant la mémorisation. Ceux-ci divergent de par leurs réalisations et utilisations. Nous verrons donc, dans un premier temps, comment les dispositifs de mémorisation d'un ordinateur s'articulent et quels sont les compromis entre ceux-ci.

Les systèmes d'exploitation permettent à plusieurs applications de s'exécuter simultanément. Pour ce faire, ils garantissent un certain niveau d'isolation entre ces dernières. Nous regardons alors en détail comment cette isolation est mise en place par Linux [91].

Malgré l'isolation des applications, les systèmes d'exploitation offrent des mécanismes permettant de consolider la mémoire afin d'offrir de meilleures performances aux applications et une équité de celles-ci dans la réclamation de leurs mémoires. Nous étudions donc sous quelle forme la consolidation est offerte par Linux.

3.2 LA MÉMOIRE D'UN POINT DE VUE PHYSIQUE

Dans cette section, nous étudions les différents dispositifs sur lesquels s'appuient les composants d'un ordinateur pour mémoriser des informations. Nous présentons d'abord l'architecture d'un ordinateur moderne avant de nous intéresser plus particulièrement à la mémoire dynamique.

3.2.1 *Architecture d'un ordinateur moderne*

L'architecture des ordinateurs modernes, désormais quasiment commune aux mobiles multifonctions, hérite de l'architecture dite de von Neumann [122]. Une représentation de cette architecture est donnée dans la Figure 6.

Cette architecture repose sur les éléments suivants :

UNITÉ ARITHMÉTIQUE ET LOGIQUE : Cette unité est en charge des opérations de calcul comme les additions ou multiplications.

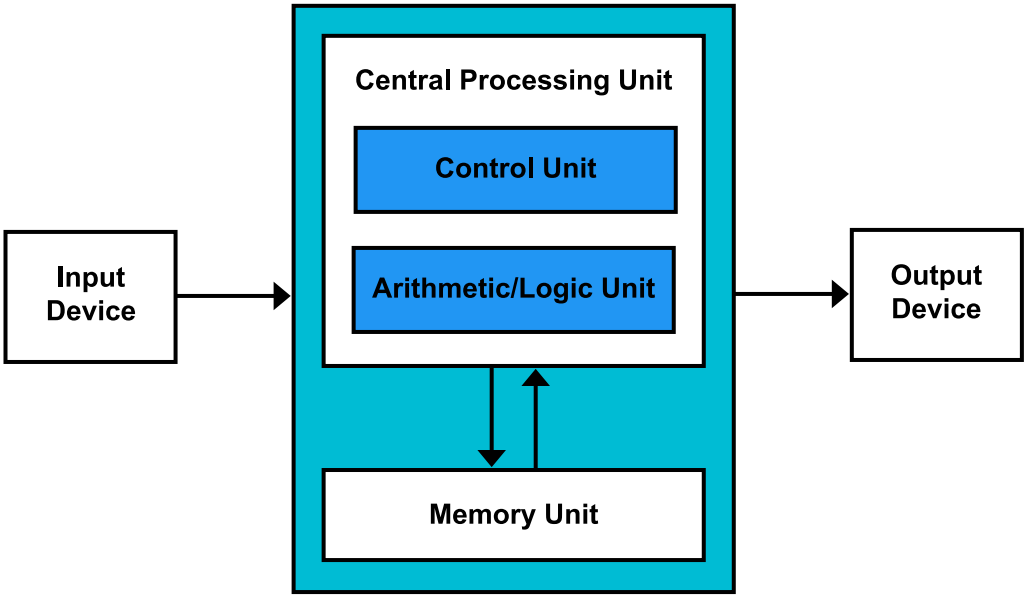


FIGURE 6 – Architecture dite de von Neumann [37]

UNITÉ DE CONTRÔLE : Après l'exécution d'une instruction, c'est cette unité qui est chargée de faire passer le programme à l'instruction suivante.

UNITÉ DE MÉMOIRE : Les instructions et les données manipulées par l'unité arithmétique et logique sont stockées dans cette mémoire. L'architecture dite de von Neumann tient son originalité du fait que les données et les instructions sont stockées dans la même mémoire ; contrairement à l'architecture Harvard où ces informations sont conservées dans des mémoires séparées [21].

PÉRIPHÉRIQUE D'ENTRÉE : Ce type de périphérique envoie des informations à l'unité arithmétique et logique. Une souris ou un clavier sont de parfaits exemples de tels périphériques.

PÉRIPHÉRIQUE DE SORTIE : Ces périphériques permettent à l'unité arithmétique et logique de présenter des informations. Un écran est un périphérique de sortie.

Notez toutefois que certains composants, tels que les disques durs et les cartes réseau, sont à la fois des périphériques d'entrée et de sortie.

Dans un ordinateur, l'unité de contrôle et l'unité arithmétique et logique font partie du même composant : le processeur [77, Section 2.2.2]. C'est donc ce composant qui exécute les instructions permettant la réalisation des applications. Pour ce faire, le processeur s'appuie sur des registres lui permettant de stocker le résultat des instructions et de les réutiliser [77, Section 3.4.1]. Ces registres sont néanmoins petits, 64 bit pour une architecture moderne, et en nombre limité.

L'unité de mémoire est organisée de manière hiérarchique. Au plus proches du processeur, les mémoires sont rapides, mais petites. Plus ces mémoires sont distantes du processeur, plus elles sont volumineuses, mais lentes. Ce compromis, connu comme la pyramide mémoire, est dépeint dans la Figure 7

Les processeurs embarquent, en plus des registres, plusieurs niveaux de cache contenant des copies des dernières données et instructions

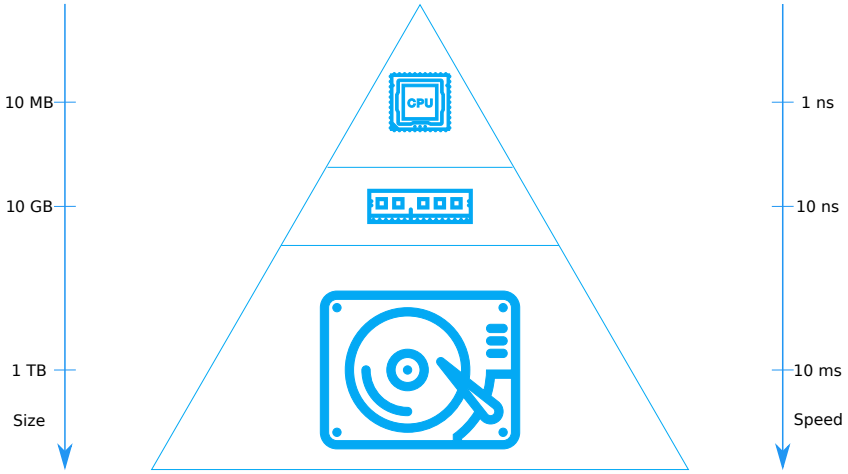


FIGURE 7 – La pyramide mémoire

utilisées [77, Figure 2-1][80, Section 5.2.1]. Ces caches ont une taille finie, de l'ordre du MB [19]. Au-dessus de ces caches se trouve la mémoire dynamique, extérieure au processeur, se présentant sous forme de barrettes. Ces barrettes de mémoire comportent des puces composées de plusieurs milliards de cellules permettant, chacune, de stocker un bit. L'organisation interne de ces puces est très différente de celle des registres et des caches du processeur [80, Section 8.2].

Toutes les mémoires que nous avons présentées jusqu'ici sont des mémoires vives. C'est-à-dire que les données stockées sont perdues lorsque ces mémoires ne sont plus sous tension. Ainsi, et pour stocker des données sur le long terme, l'ordinateur peut s'appuyer sur des disques, que ce soit des disques durs ou des disques statiques à semi-conducteurs. Dans les premiers, les données sont stockées sous la forme d'un champ magnétique [80, Section 17.1.1]. Tandis que les seconds stockent les bits dans la grille flottante des transistors [28, Section 2.1.2].

Toutes ces mémoires s'articulent entre elles selon deux axes : la capacité de stockage et la vitesse d'accès. Comme nous l'avons évoqué, les registres et caches du processeur sont très rapides, un accès ne prend que quelques ns, mais aussi très petits, quelques MB [19]. De l'autre côté, les disques peuvent atteindre plusieurs TB mais sont beaucoup plus lents, l'accès se fait dans l'ordre de la ms pour les plus lents [58].

3.2.2 *La mémoire dynamique*

Dans cette thèse, nous nous intéressons plus particulièrement à la mémoire dynamique. Dans les différents composants que nous avons présentés dans la Section 3.2.1, la mémoire dynamique se trouve dans les puces situées sur les barrettes de mémoire. Aujourd'hui, il est plus courant de parler de mémoire dynamique à accès aléatoire, ou *Dynamic Random Access Memory* (DRAM).

Cette mémoire est dite dynamique, car son contenu nécessite d'être périodiquement rafraîchi sous peine d'être perdu [80, Section 8.2.1].

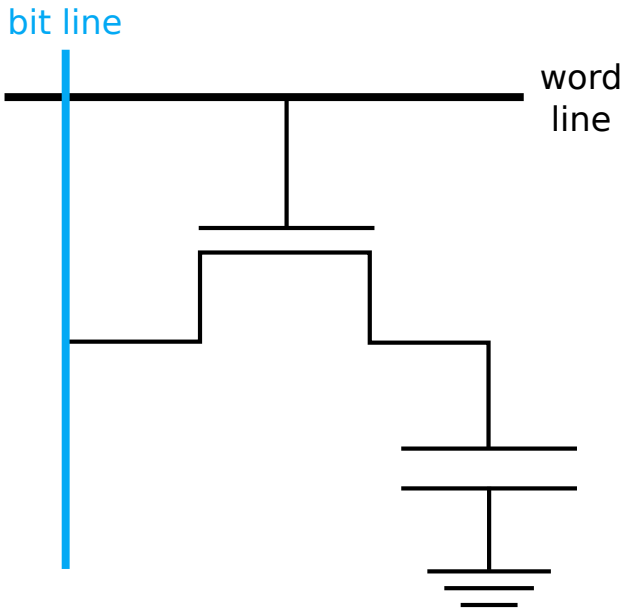


FIGURE 8 – Une cellule de DRAM [80, Figure 8.2]

En effet, une cellule, l'élément permettant de stocker un bit, s'appuie sur un condensateur dont la charge fuit au cours du temps [80, Figure 8.2]. Contrairement aux cellules des caches et registres processeurs qui s'appuient uniquement sur des transistors et qui sont donc dites statiques [80, Figure 5.2]. La Figure 8 montre une cellule de mémoire dynamique composée d'un condensateur et d'un transistor d'accès.

Ces cellules sont organisées en matrices qui sont ensuite regroupées en plusieurs blocs. Dans ce manuscrit, nous n'entrerons pas dans le détail de l'organisation interne d'une barrette de DRAM. Néanmoins, la Figure 9 présente l'agencement des différents composants tel que standardisé par le *Joint Electron Device Engineering Council* (JEDEC) pour la DDR4 [81].

De plus, la DRAM n'a pas été sujette aux mêmes progrès que les processeurs [159]. Elle peut donc être un frein aux performances des applications, même si des progrès ont été faits pour contrer sa faible fréquence et augmenter sa bande passante [5]. Ainsi, une gestion efficace de la mémoire par le système d'exploitation s'avère cruciale pour les performances des applications. Nous allons donc voir comment les systèmes d'exploitation modernes, Linux plus particulièrement, gèrent la mémoire.

3.3 L'ISOLATION GARANTIE PAR LE SYSTÈME D'EXPLOITATION

Aujourd'hui, les systèmes d'exploitation permettent l'exécution simultanée de plusieurs applications. Néanmoins, les données manipulées par celles-ci doivent être strictement privées pour éviter bogues et indiscrétions. Le mécanisme de mémoire virtuelle, basé sur des possibilités offertes par le matériel, permet d'isoler les applications entre elles.

Dans cette section, nous commençons par présenter le concept de mémoire virtuelle ainsi que celui de mémoire paginée. Puis, nous exa-

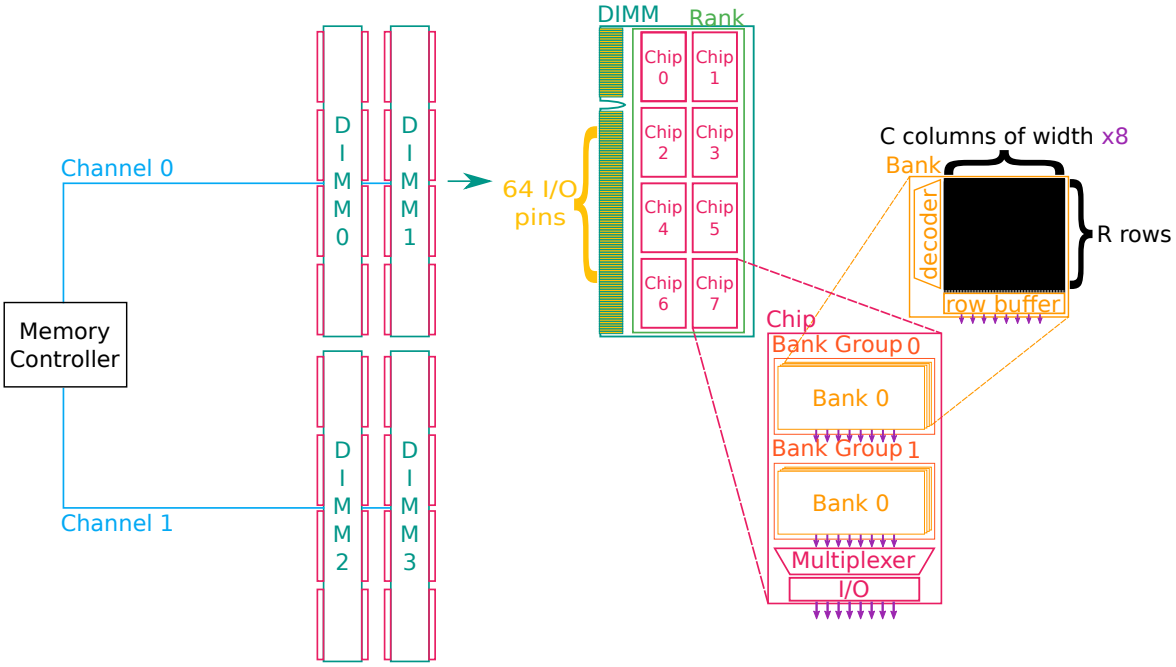


FIGURE 9 – L'organisation de la DDR4 telle que standardisée par le JEDEC [81]

minons, sans entrer dans le détail, les structures de données utilisées par Linux pour représenter différents concepts propres à la mémoire virtuelle paginée. Ensuite, nous étudions brièvement les capacités d'isolation offertes par les hyperviseurs. Pour terminer, nous nous attardons sur les conteneurs en les étudiant au niveau utilisateur et noyau.

3.3.1 *La mémoire virtuelle paginée*

Linux est un système d'exploitation basé sur le concept de mémoire virtuelle [51]. Dans ce concept, les processus ne manipulent pas des adresses physiques, mais des adresses virtuelles. Un processus a donc l'illusion qu'il dispose de toute la mémoire, en plus d'être isolé de ses congénères.

La mémoire virtuelle peut être divisée en unités insécables appelées «pages» [77, Section 3.3.2]. Dans Linux, la mémoire virtuelle est paginée, une page est donc la plus petite unité mémoire qu'il puisse allouer à une tâche[5]. Une page virtuelle est associée à une page physique. Les droits d'accès d'une page physique, par exemple, lecture seule ou lecture/écriture, sont gérés via des drapeaux propres au matériel [78, Table 4-19][67, Chapter 3].

Une association entre une page virtuelle et physique est rangée dans une structure, propre à chaque tâche, nommée la table des pages. Le format de cette table est imposé par le matériel puisque c'est la *Memory Management Unit* (MMU) qui est chargée d'effectuer la traduction d'une adresse virtuelle en une adresse physique. Cette table n'est pas contiguë et est découpée en plusieurs niveaux, ce pour limiter sa taille. Chacun des niveaux intermédiaires est une page. Lors

[5] *Linux ne fait pas la différence entre un processus et un thread [105, Section "The Linux Implementation of Threads"]. Pour lui, des threads sont des tâches partageant des données.*

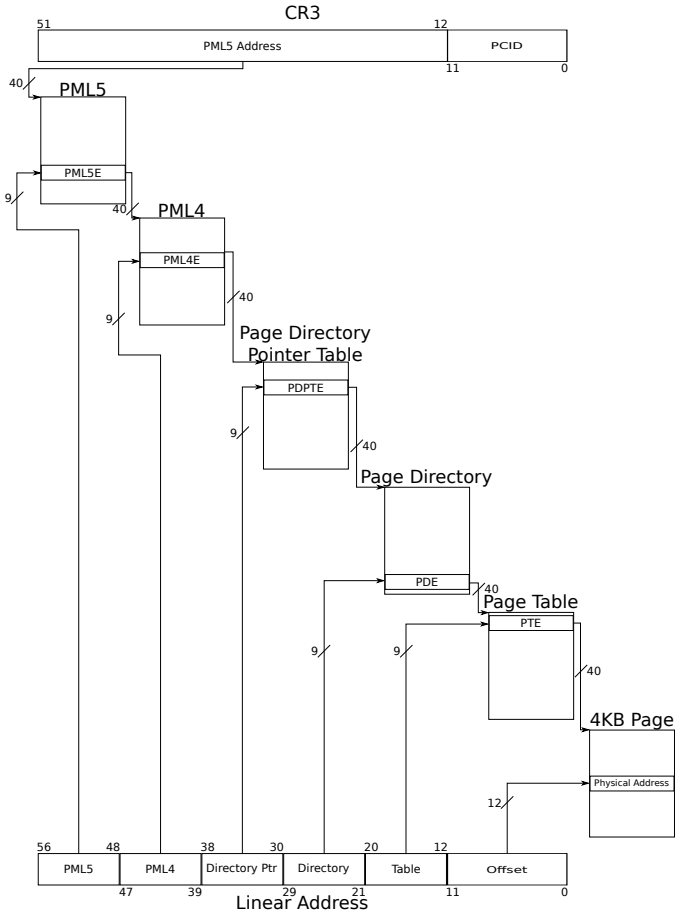


FIGURE 10 – Table des pages à 5 niveaux [74, Figure 2-1]

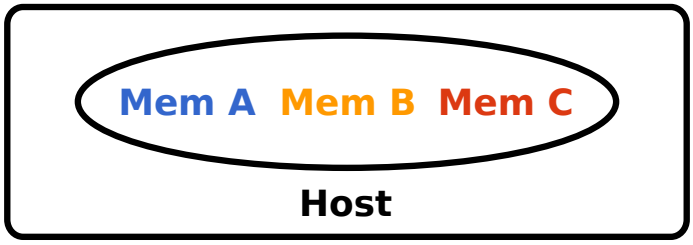
d'une traduction, l'adresse virtuelle est découpée en plusieurs parties. Les bits de chacune de ces parties sont utilisés pour accéder au niveau correspondant dans la table des pages.

La Figure 10 montre l'organisation d'une table des pages à 5 niveaux pour l'architecture x86_64 [74, Figure 2-1]. L'adresse du plus haut niveau de la table des pages, PML5, de la tâche courante est stockée dans le registre `cr3`. L'adresse virtuelle[6] est découpée en 5 parties de 9 bits. Par exemple, si les bits de la partie PML5 ont pour valeur décimale 484, l'entrée 484 du PML5 sera accédée pour passer au PML4 correspondant. Les bits de poids faible de l'adresse virtuelle ne sont pas utilisés pour la traduction et correspondent au déplacement à effectuer dans la page physique pour obtenir la donnée demandée. Ces bits sont donc communs à une adresse virtuelle et une adresse physique.

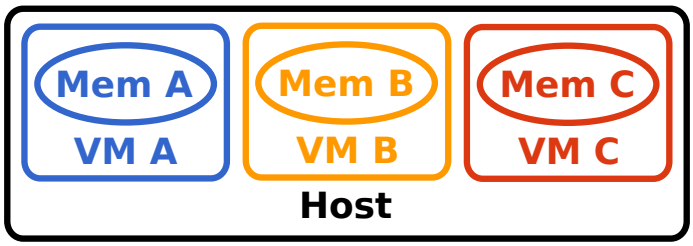
[6] *Dans la Figure 10, le terme «Linear Address» correspond à «adresse virtuelle».*

Linux, dans son implémentation, distingue deux types de pages :

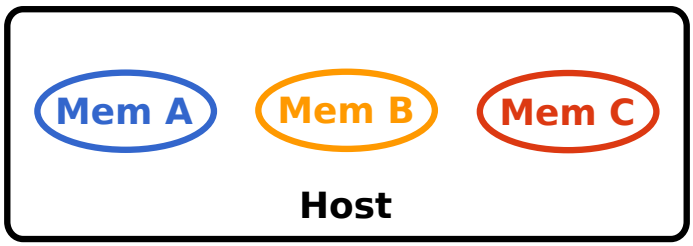
PAGES FICHIERS : Ces pages contiennent des données qui ont été récupérées depuis le disque. En effet, quand une donnée sur le disque est demandée, Linux récupère plusieurs données à la fois et les stocke en mémoire physique à l'aide d'une page. Ainsi, si la donnée demandée est à nouveau accédée, ou si une donnée proche l'est, elle pourra être lue depuis la mémoire physique. Ce qui permet alors un gain de temps considérable par rapport à une lecture depuis le disque [139].



(a) La mémoire des tâches n'est pas limitée



(b) La quantité maximale de mémoire qu'une machine virtuelle peut utiliser est fixée



(c) L'empreinte mémoire d'un conteneur ne peut pas dépasser une certaine valeur

FIGURE 11 – Organisation de la mémoire pour différentes solutions

PAGES ANONYMES : Ces pages contiennent des données qui ont été créées lors de l'exécution d'un programme. Par exemple, les pages utilisées pour la pile ou le tas sont des pages anonymes.

Malgré cette distinction, Linux s'efforce d'appliquer des algorithmes similaires à toutes les pages.

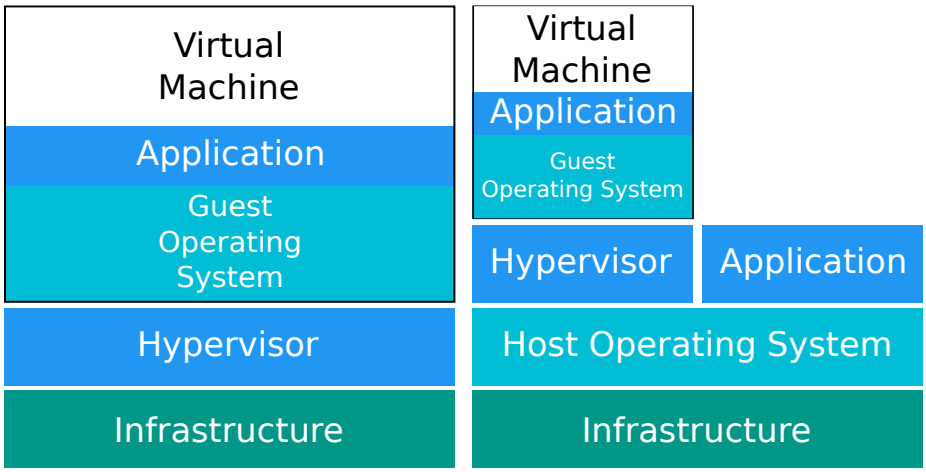
3.3.2 *Espace d'adressage virtuel d'une tâche*

Dans l'espace d'adressage virtuel d'une tâche, les pages sont regroupées en région. Une région est une partie contiguë de l'espace d'adressage virtuel. Elle regroupe des pages de même type ayant des droits d'accès communs. Par exemple, il y aura une région pour la pile, une pour le tas et une autre pour un fichier projeté en mémoire suite à l'invocation de l'appel système `mmap`.

Une région peut être créée suite à un appel à `mmap`[7]. Suite à la création d'une nouvelle tâche, via un appel à `fork`, la tâche enfant hérite des régions de son parent [67, Section "Copy On Write"].

[7] *`mmap` est appelé par `malloc` [143].*

Les différentes régions d'une même tâche sont regroupées dans l'espace d'adressage virtuel de celle-ci. Il existe un espace d'adressage virtuel pour chaque tâche. Cet espace est associé à la table des pages de la tâche.



(a) Hyperviseur de type 1 (b) Hyperviseur de type 2

FIGURE 12 – Schémas de l'organisation des deux types d'hyperviseurs

Comme le montre la Figure 11a, les différentes tâches partagent la mémoire physique de la machine et ne sont limitées que par celle-ci.

3.3.3 *L'isolation poussée à l'extrême : les hyperviseurs*

Un hyperviseur est un logiciel permettant d'exécuter des VM, il en existe deux types [128][31, Section 1.4] :

HYPERVISEUR DE TYPE 1 : Il s'exécute à même le matériel, il remplace donc le système d'exploitation. `xen` est un bon exemple d'hyperviseur de ce type [160].

HYPERVISEUR DE TYPE 2 : Ce type d'hyperviseur s'exécute par-dessus un système d'exploitation, c'est donc un processus appartenant à un système d'exploitation. `qemu` peut être cité comme hyperviseur de type 2 [132].

La Figure 12 dépeint la différence entre les deux types d'hyperviseurs.

Un hyperviseur permet, en plus d'exécuter un système de manière isolée, de restreindre les ressources qui lui sont allouées. Ainsi, au démarrage du système invité, il est possible de l'exécuter sur tel nombre de cœurs ou avec telle quantité de mémoire. Cette fonctionnalité est donc à la base du concept d'instance [18]. La Figure 11b montre bien que, contrairement à la Figure 11a, la mémoire de chaque VM est bornée. Ainsi, une VM ne peut s'accaparer toute la mémoire.

Un hyperviseur doit aussi gérer la mémoire du système invité. En effet, le système invité possède ses propres tables des pages, or celui-ci n'a pas accès directement à la mémoire. Ses tables des pages associent donc une adresse virtuelle invitée à une adresse physique invitée [141, Section 8.3.1]. La traduction d'une adresse virtuelle invitée en une adresse physique de l'hôte est alors réalisée par l'hyperviseur au moyen des *shadow pages tables* [141, Section 8.3.2].

3.3.4 *Sous les conteneurs : les `cgroups`*

Comme nous l'avons vu dans l'introduction, les conteneurs présentent une alternative plus légère aux VM. Comme le montre la Figure 1, les conteneurs, contrairement aux VM, ne s'appuient pas sur un système d'exploitation invité : c'est l'application uniquement qui est conteneurisée [55, 163].

Les conteneurs proposent principalement trois services :

SÉCURITÉ : Les données d'un conteneur ne peuvent être accédées par un autre conteneur sauf partage explicite.

CONTRÔLE DES RESSOURCES : Il est possible d'allouer finement des ressources à un conteneur, comme un cœur de processeur ou un octet de la mémoire physique. Contrairement aux tâches conventionnelles, l'empreinte mémoire d'un conteneur pourra être bornée par une valeur fixée par l'utilisateur.

FACILITÉ DE DÉPLOIEMENT : Lancer un conteneur est aussi simple qu'exécuter une commande `shell`.

Dans les sous-sections qui suivent, nous présentons, dans un premier temps, l'organisation logicielle des conteneurs `docker` [52]. Il n'existe pas, dans Linux, de structure de données correspondant à un conteneur. Ainsi, les conteneurs forment une abstraction basée sur plusieurs fonctionnalités offertes par les systèmes d'exploitation. Nous détaillons donc, dans un deuxième temps, deux de ces fonctionnalités : les `namespaces` et les `cgroups`.

Les conteneurs `docker`

`Docker` est, comme `lxc`, `kata` et `podman`, un des nombreux logiciels existants permettant d'exécuter et gérer des conteneurs [52, 106, 84, 127].

Un conteneur est une application exécutée avec toutes ses dépendances et ses données. Celles-ci sont stockées dans une image de l'application. Ainsi, au démarrage du conteneur, `docker` va extraire ces données de l'image, notamment le code de l'application afin de l'exécuter.

Ces images peuvent être récupérées en ligne [54]. Il existe des images de base, comme `debian`, pouvant être utilisées pour construire de nouvelles images. En effet, l'utilisateur peut installer de nouveaux logiciels dans un conteneur et le sauvegarder comme une nouvelle image. Celle-ci peut ensuite être utilisée par l'utilisateur pour exécuter son service personnalisé. Une image peut être créée sans avoir à exécuter de conteneur auparavant en utilisant un `dockerfile` [110, Section "Introducing the Dockerfile"].

Originellement `docker` était un système monolithique. Il est désormais, dans un système GNU/Linux, composé des briques logicielles suivantes [129] :

DOCKER-CLI : Cette brique contient les outils en ligne de commande permettant, entre autres, la création, l'exécution, l'arrêt ou la sauvegarde d'un conteneur [53] .

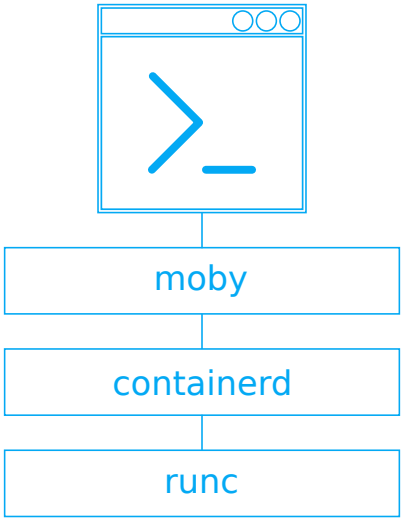


FIGURE 13 – Organisation logicielle des différentes briques composant `docker`

MOBY : `moby` gère les différents volumes et réseaux utilisés par les conteneurs. Un volume est un système de fichier pouvant être attaché à un conteneur. Ainsi, deux conteneurs différents peuvent partager des fichiers si un même volume leur est attaché. Ce démon, un processus s'exécutant en arrière-plan, est aussi en charge de l'orchestration des conteneurs. L'orchestration consiste à exécuter des conteneurs sur différentes machines tout en leur permettant de communiquer [119, 47].

CONTAINERD : Ce démon propose l'abstraction des conteneurs. Il gère aussi les images que ce soit leurs créations ou leurs sauvegardes [38, 47] .

RUNC : Cette brique dialogue avec le système d'exploitation pour créer, lancer ou détruire des conteneurs. Elle contient des outils en ligne de commande permettant de manipuler des conteneurs. Pour ce faire, `runc` s'appuie sur la `libcontainer` qui exécute ce qui a été demandé par la ligne de commande. C'est au travers de cette bibliothèque que `containerd` communique avec `runc` [123, 73]. Lors du lancement d'un nouveau conteneur, `runc` instancie un nouveau `cgroup`, pour chaque sous-système, et un nouveau `namespace`, pour chaque ressource à abstraire.

Ces différentes briques sont organisées comme décrit dans la Figure 13.

Les `namespaces`

Pour la sécurité, les conteneurs s'appuient sur les `namespaces` [134, 86, 87]. Un `namespace` abstrait une ressource système globale. La finalité étant de faire croire aux tâches appartenant à une instance de ce `namespace` qu'elles possèdent leur propre exemplaire, isolé, d'une ressource donnée.

Il existe de nombreux `namespaces` dont le `mount namespace` et le `PID namespace`. Le premier permet qu'un système de fichier ne soit monté que pour les tâches appartenant au `mount namespace` donné. Il

devient donc possible d'imaginer deux instances du `mount namespace` avec des tâches inscrites dans chacun et accédant à deux systèmes de fichiers complètement différents. Les processus du premier `mount namespace` ne voyant pas le second système de fichier, ils ne pourront donc pas accéder à ses fichiers.

Quant au second *namespace*, il offre la possibilité d'avoir des tâches ayant le même «Process ID» (PID), l'identifiant unique d'un processus, dans plusieurs `PID namespaces`. Avec ce `namespace`, il peut y avoir plusieurs tâches de PID 1, c'est-à-dire plusieurs `init` [43].

Les `cgroups`

Le contrôle fin des ressources des conteneurs est rendu possible par les *control groups* (`cgroups`) [134, 71]. Comme le montre la Figure 11c, les `cgroups` peuvent, entre autres, restreindre la quantité de mémoire qu'un conteneur peut utiliser. Le mécanisme des `cgroups` a fait ses premiers pas dans le noyau en 2006 [40]. Néanmoins, ce n'est qu'en 2007, suite à son inclusion dans la version 2.26.24, qu'il gagne son nom de `cgroup`, le nom *containers* ayant été jugé trop générique [44, 43].

Néanmoins, l'implémentation des `cgroups` a été la cible de plusieurs critiques. En 2012, Jonathan Corbet écrivait ceci à leurs propos [41] :

> Control groups are one of those features that kernel developers love to hate.

L'un des principaux reproches adressés aux `cgroups` visait alors l'organisation hiérarchique de ceux-ci. L'implémentation des `cgroups` a donc été modifiée pour donner naissance à `cgroup v2` [137, 70]. Avec cette nouvelle version, l'organisation des différents `cgroups` est simplifiée, il n'est, entre autres, plus possible d'avoir des tâches appartenant à un `cgroup` nœud.

Cette nouvelle version n'est par contre pas encore utilisée par tous les logiciels permettant de lancer des conteneurs [144, 146]. En effet, `docker`, que nous utilisons dans cette thèse, n'est pas encore compatible avec `cgroup v2` tandis que `podman` l'est.

Les `cgroups` permettent de grouper plusieurs tâches avec un ensemble de paramètres pour un ou plusieurs sous-systèmes [39]. Un sous-système étant défini comme un module souhaitant appliquer un certain traitement à un ensemble de tâches. Dans la pratique, un sous-système est un contrôleur de ressource permettant d'ordonnancer une ressource ou d'appliquer des limites à l'utilisation de celle-ci à un `cgroup` donné. Les `cgroups` permettent aussi de comptabiliser l'utilisation des ressources faites par un ensemble de tâches.

De base, une tâche appartient à un `cgroup` particulier, le `cgroup root`, pour chaque sous-système. Le `cgroup root` a la particularité de ne pas être limité dans son usage des ressources. Pour limiter l'usage d'une tâche existante, il faut inscrire son identifiant dans le `cgroup` correspondant.

Tout comme les `namespaces`, plusieurs sous-systèmes existent pour les `cgroups`. Le `cgroup cpuset` permet de restreindre l'exécution d'une instance d'un `cgroup` sur un nombre de cœurs donné [46]. Le `cgroup process number` offre la possibilité de limiter le nombre de tâches

d'un `cgroup` [131]. Ainsi, il est possible d'obtenir à tout moment le nombre de tâches d'un `cgroup` et de limiter l'explosion d'une `fork bomb` [153].

L'existence d'un sous-système pour chaque ressource n'implique pas que toutes ces ressources sont aussi faciles à gérer les unes que les autres. En effet, la part de temps CPU est assez simple à gérer, soit un `cgroup` utilise sa part soit il ne l'utilise pas. La gestion de la mémoire est par contre plus complexe. En effet, un `cgroup` peut utiliser, ou non, la totalité de sa mémoire, mais il n'est pas possible de savoir si les données stockées dans celle-ci lui sont utiles ou non.

3.4 LES MÉCANISMES DE CONSOLIDATION OFFERTS PAR LES SYSTÈMES D'EXPLOITATION

Malgré l'existence de mécanisme d'isolation que nous venons de présenter, Linux a fait le choix d'une gestion commune de la mémoire dans trois situations :

1. Lors de l'allocation mémoire, Linux utilise un ensemble de pages libres commun à toutes les applications. Ainsi, aucune application n'est prioritaire pour obtenir de nouvelles pages.
2. La seconde est celle où des blocs lus depuis le disque sont ramenés dans la mémoire principale. Ce cache est partagé par toutes les applications.
3. Lors d'une réclamation mémoire, quand celle-ci se fait rare, les applications sont toutes réclamées de la même manière. Linux garantit donc une équité totale entre ces dernières.

Dans la suite de cette section, nous étudions comment Linux alloue la mémoire aux différentes applications. Puis, nous analysons le fonctionnement du *page cache* de Linux ainsi que l'algorithme exécuté lors d'une phase de réclamation mémoire. Nous continuons par une brève étude du mécanisme de *memory ballooning* offert par les hyperviseurs pour consolider la mémoire entre plusieurs VM. Nous finissons par détailler les spécificités apportées par le `cgroup` mémoire dans la gestion mémoire.

3.4.1 *Allocation mémoire*

Lors d'un appel à `malloc`, aucune page physique n'est associée à une page virtuelle, mais elles sont seulement marquées comme pouvant être utilisées. En effet, Linux alloue les pages physiques de manière paresseuse, c'est-à-dire qu'une page physique est allouée uniquement quand elle est touchée par la tâche qui souhaite l'utiliser [124].

Le premier accès à une page non allouée déclenche une exception, plus communément appelée «défaut de page» [78, Section 4.7]. Si l'accès est licite, entre autres quand l'adresse virtuelle ayant déclenché le défaut de page appartient bien à une des régions de l'espace d'adressage virtuel de la tâche, une page est allouée à la tâche. Sa table des pages est aussi modifiée afin que l'adresse virtuelle soit

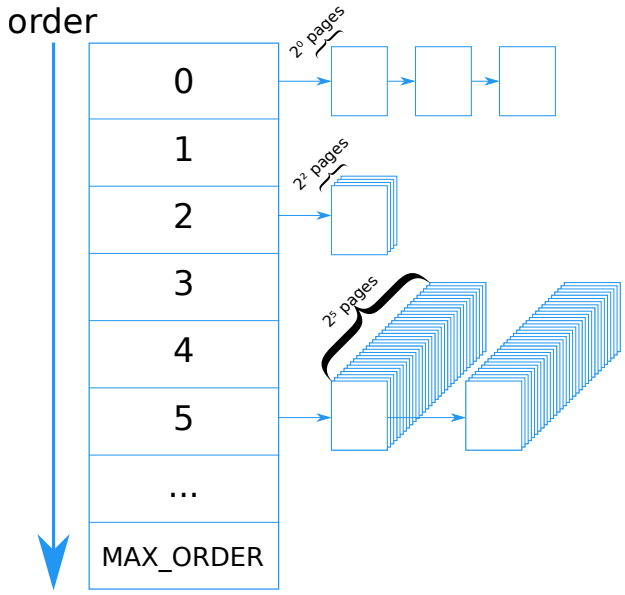


FIGURE 14 – Organisation du *buddy allocator* [67, Figure 6.1]

désormais associée à une adresse physique. Le prochain accès à cette même adresse virtuelle ne causera donc plus de défaut de page.

Pour allouer une page, Linux s'appuie sur un allocateur nommé le *buddy allocator* [67, Figure 6.1]. Cet allocateur permet de répondre à des allocations de plusieurs pages contiguës tout en réduisant la fragmentation externe [67, Section 6.6]. Il repose sur un tableau où chaque case est une liste de blocs de pages. D'une manière générale, la case n pointe sur une liste de blocs de 2^n pages contiguës. Par exemple, la liste stockée dans la case 5 contient des blocs de $2^5 = 32$ pages. La Figure 14 montre l'organisation du *buddy allocator*.

Pour allouer un bloc de, par exemple, 7 pages, Linux va devoir allouer un bloc de 8 pages. Si, malheureusement, aucun bloc de 8 pages contiguës n'existe actuellement, il sélectionne un bloc de 16 pages qu'il casse en 2 blocs de 8 pages. Le premier est rangé dans la liste de la case 3. Tandis que le second est utilisé pour satisfaire l'allocation. La page restante est bien entendu ajoutée à la liste pointée par la case 0.

Une fois alloué, ce bloc est ajouté au *page cache* que nous détaillons dans la Section 3.4.2.

3.4.2 *Le page cache*

Comme nous l'avons vu dans la Section 3.2.1, le stockage sur disque est beaucoup plus lent que la DRAM ou les caches processeurs. Ainsi, plutôt que de lire une donnée sur le disque à chaque accès, les systèmes d'exploitation ramènent plusieurs données, formant un bloc, depuis le disque qu'ils stockent dans la mémoire physique.

Pour Linux, ce mécanisme s'appelle le *page cache* [67, Figure 10.1]. Celui-ci peut occuper toute la mémoire qui n'a pas encore été allouée [105, Chapter 16].

Dans la section 3.3.1, nous avons indiqué que Linux distingue les pages fichiers des pages anonymes. Ainsi, pour ces deux types de pages, le cache contient deux listes de pages, il y a donc quatre listes au total :

LISTE ACTIVE : Une page qui a été accédée plusieurs fois se situe en liste active.

LISTE INACTIVE : Une page qui n'a été accédée qu'une fois est d'abord placée en liste inactive. Si elle est à nouveau accédée, elle est promue en liste active.

3.4.3 *Réclamation mémoire*

Malheureusement, la mémoire est une ressource finie. Quand les pages physiques commencent à manquer, il est courant de dire que le système est en pression mémoire. Dans ce cas, le noyau doit réclamer des pages avant de pouvoir satisfaire une nouvelle allocation.

Pour ce faire, Linux s'appuie sur deux mécanismes [29, Figure 17-3] :

KSWAPD : Ce démon est réveillé quand le nombre de pages libres passe en dessous d'un certain seuil. Il a pour but de réclamer des pages afin d'éviter, de façon préventive, que la pression mémoire n'empire [67, Section 10.6].

DIRECT RECLAIM : Lors d'une allocation de page, si le nombre de pages libres est inférieur à un seuil, lui-même inférieur au seuil de réveil de `kswapd`, Linux procède à un *direct reclaim*. Ce mécanisme consiste simplement à réclamer des pages avant de répondre à la demande d'allocation [29, Section "Low On Memory Reclaiming"].

Ces mécanismes agissent sur le *page cache* pour récupérer de la mémoire [149, fichier `mm/vmscan.c`, ligne 2482]. Linux essaye de gérer les deux catégories de pages de façon identique. Une page fichier peut être réclamée sans effort, si son contenu n'a pas été modifié par rapport à celui sur le disque, car elle peut facilement être relue depuis le disque. Cependant, la réclamation des pages anonymes est plus complexe, car les données contenues ne peuvent être lues depuis le disque. Linux s'appuie donc sur un espace d'échange, une zone particulière de la mémoire de stockage où peuvent être conservées, temporairement, les pages anonymes.

Lorsqu'il y a pression mémoire, Linux va essayer de réclamer, à chaque liste, une certaine quantité de pages. Une liste inactive est toujours réclamée tandis qu'une liste active l'est uniquement si la liste inactive correspondante comporte peu de pages. Lorsqu'une liste active est réclamée, les pages qui ont été récupérées ne sont pas marquées immédiatement comme libres, mais sont d'abord transférées dans la liste inactive analogue. Une page qui appartenait à une liste active se voit donc offrir une seconde chance. *A contrario*, les pages réclamées depuis une liste inactive peuvent être immédiatement réutilisées.

La Figure 15 donne une représentation de l'action de la réclamation mémoire sur le *page cache*.

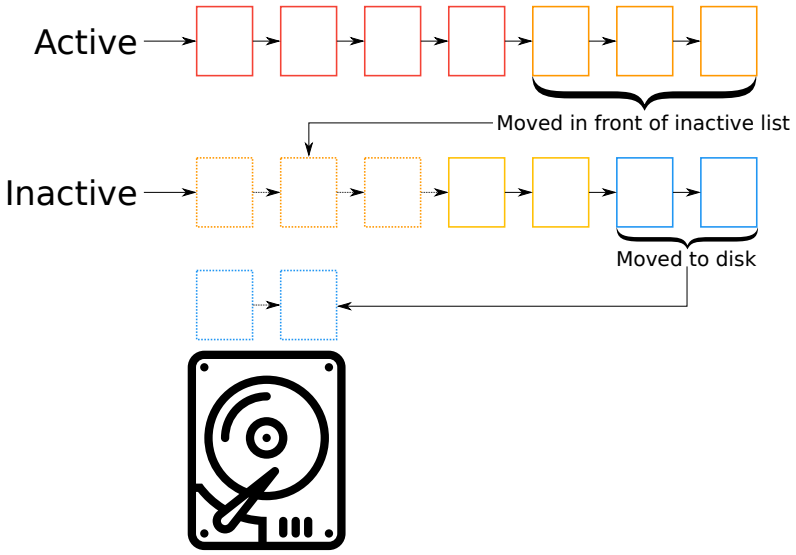


FIGURE 15 – Le déplacement des pages dans le *page cache* lors d'une pression mémoire

3.4.4 *Le ballooning des hyperviseurs*

Pour transférer la mémoire entre plusieurs VM, les hyperviseurs s'appuient sur le *memory ballooning* [133]. Nous avons décrit ce mécanisme dans la Section 2.6 et conclu qu'il était difficile à utiliser. Nous verrons, dans la prochaine sous-section, que le `cgroup` mémoire facilite le transfert de mémoire entre plusieurs conteneurs.

3.4.5 *La consolidation pour le `cgroup` mémoire*

Lorsqu'un conteneur est lancé via `docker`, un nouveau `cgroup` mémoire est instancié pour celui-ci. Ce `cgroup` mémoire dispose de son propre *page cache* [42, 136]. Ces différents *page caches* sont exclusifs, c'est-à-dire qu'une page dans le *page cache* d'un `cgroup` n'appartient pas au *page cache* d'un autre `cgroup`, `root` par exemple. Ce design permet donc de récupérer de la mémoire à un `cgroup` en particulier plutôt qu'à toutes les tâches du système. La Figure 16 schématise cette organisation.

Quand une page est ajoutée au *page cache* d'un `cgroup`, il est dit que cette page est chargée dans ce `cgroup`. En effet, la fonction responsable de cela est `mem_cgroup_try_charge` [149, fichier `mm/memcontrol.c`, ligne 5874].

Pour prendre certaines décisions, notamment liées à la réclamation mémoire, le `cgroup` mémoire se base sur des métriques système, exposées à l'utilisateur via le fichier `memory.stat` [113, Section 5.2]. Le Tableau 3 compile celles-ci.

3.5 CONCLUSION

Dans ce chapitre, nous avons vu que la mémoire est sujette à un compromis entre sa taille et sa vitesse d'accès. Pour pallier à cela,

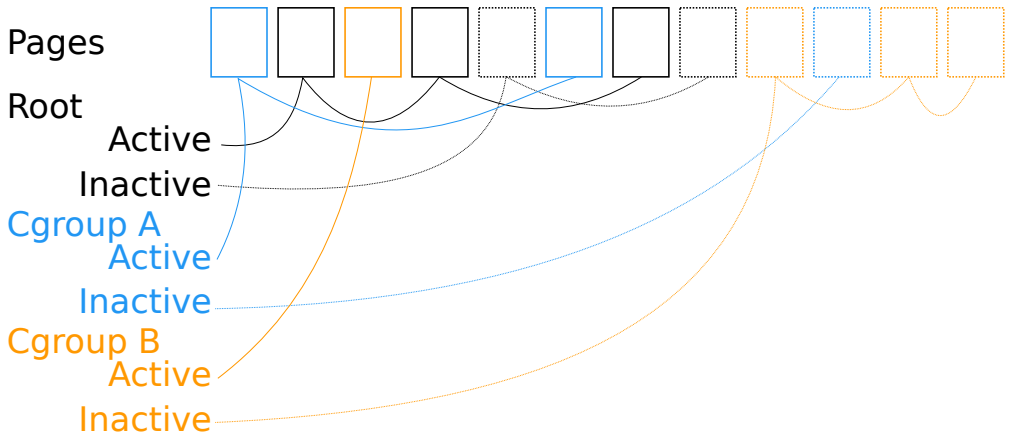


FIGURE 16 – Schéma de l'organisation du *page cache* entre les `cgroups`

TABLE 3 – Les différentes statistiques collectées par le `cgroup` mémoire [113, Section 5.2]

Nom	Description
`cache`	La taille du *page cache* en octet.
`rss`	La taille, en octet, de la mémoire utilisée pour des pages anonymes.
`rss_huge`	La taille, en octet, de la mémoire utilisée par des *huge pages* anonymes.
`mapped_file`	Nombre d'octets utilisés pour des fichiers projetés en mémoire via `mmap`.
`pgpgin`	Nombre de fois qu'une page a été chargée dans ce `cgroup`.
`pgpgout`	Nombre d'occurrences de déchargement de page pour ce `cgroup`.
`swap`	Taille actuelle, en octet, de l'espace d'échange.
`dirty`	Nombre d'octets en attente d'écriture sur le disque.
`writeback`	Nombre d'octets, en file d'attente pour être écrits sur le disque.
`inactive_anon`	Taille, en octet, de la liste anonyme inactive.
`active_anon`	Le nombre d'octets contenus dans la liste anonyme active.
`inactive_file`	Taille, en octet, de la liste fichier inactive.
`active_file`	Taille de la liste fichier active en octet.
`unevictable`	Taille, en octet, de la liste des pages ne pouvant être évincées.

les noyaux des systèmes d'exploitation proposent la mise en cache de blocs du disque dans la mémoire principale. D'un autre côté, les hyperviseurs et les conteneurs apportent des modifications à la gestion de la mémoire, comme la présence d'un *page cache* par instance du `cgroup` mémoire. Dans le chapitre suivant, nous mettons en exergue l'incapacité du `cgroup` mémoire à offrir et l'isolation et la consolidation.

4

DE LA GESTION MÉMOIRE PROBLÉMATIQUE DES CONTENEURS : L'ISOLATION SANS LA CONSOLIDATION

4.1 INTRODUCTION

Dans le Chapitre 3, nous avons montré que les conteneurs s'appuient sur les `namespaces` et les `cgroups`. Les premiers offrent une isolation orientée sécurité, par exemple, une tâche extérieure à un `namespace` ne peut envoyer de signaux à une tâche appartenant à ce `namespace`. Les `cgroups` sont, quant à eux, utilisés pour limiter l'utilisation des ressources d'un conteneur, assurant ainsi une isolation de performance. Ces deux fonctionnalités sont offertes par le système d'exploitation, qui dispose d'une vision globale des ressources, la question de la garantie de la consolidation se pose alors.

Nous présentons donc, dans ce chapitre, comment le `cgroup` mémoire effectue cette limitation. Nous montrons ensuite, par l'expérience, que ces mécanismes ne permettent pas de garantir en même temps l'isolation et la consolidation mémoire [100, 95].

4.2 LES MÉCANISMES DE LIMITES DU `cgroup` MÉMOIRE

Ce `cgroup` permet de contrôler l'empreinte mémoire d'un conteneur, c'est-à-dire la somme de la mémoire anonyme allouée et de l'ensemble des pages contenues dans le *page cache*. En pratique, ce contrôle opère sur les pages de la seconde catégorie. En effet, et comme nous l'avons discuté dans la Section 3.4.3, la réclamation de la mémoire anonyme s'appuie sur un espace d'échange situé sur le disque. Transférer une page depuis l'espace d'échange vers la mémoire est coûteux en temps et a donc un impact sur les performances des applications [161].

Pour contrôler la taille de l'empreinte mémoire des tâches qu'il contient, le `cgroup` mémoire s'appuie sur plusieurs limites. Ces limites peuvent être renseignées au lancement d'un conteneur ou lors de son exécution. Si elles ne sont pas renseignées, leurs valeurs sont équivalentes à $+\infty$, ce qui revient à ne pas activer le mécanisme [149, fichier `mm/memcontrol.c`, ligne 4485].

Lorsqu'une page est chargée dans un `cgroup`, Linux vérifie que l'allocation respecte les différentes limites mémoires fixées. Dans le cas contraire, les tâches du `cgroup` voient leurs mémoires réclamées pour satisfaire cette allocation.

Dans les sous-sections qui viennent, nous détaillons les max et soft limites. Puis nous expliquons brièvement les nouvelles limites introduites par cgroup v2 [1].

4.2.1 *La max limite*

La max limite indique le nombre maximal d'octets pouvant être dédié aux tâches d'un cgroup. Cette limite peut être dépassée mais la mémoire du cgroup sera immédiatement réclamée [113, Section 2.2]. Si la réclamation n'a pas abouti, l'*OOM killer* est invoqué pour tuer la tâche ayant la plus grosse empreinte mémoire du cgroup [113, Section 2.5][29, Section "The Out of Memory Killer"].

Le fait de pouvoir dépasser cette limite peut sembler étrange. Néanmoins, cette souplesse permet de réclamer des pages fichiers pour satisfaire une demande en pages anonymes. Sans celle-ci, un cgroup allouant et touchant beaucoup de mémoire anonyme serait tué tandis qu'ici le noyau le force à se débarrasser de ses pages fichiers pour les utiliser comme des pages anonymes.

4.2.2 *La soft limite*

Cette limite n'empêche pas le conteneur d'utiliser une quantité quelconque de mémoire, tant que les deux conditions suivantes sont remplies [113, Section 7] :

1. La mémoire du conteneur n'excède pas sa max limite.
2. Et il n'y a pas de pression mémoire.

C'est seulement quand il y a pression mémoire, c'est-à-dire quand la mémoire disponible devient rare, que la soft limite prend effet. Si les valeurs des soft limites sont trop hautes, par rapport à la mémoire dont dispose la machine, elles sont rognées. Ainsi, un conteneur ne peut affamer les autres. Par exemple, si la machine dispose de 3 GB de mémoire et que deux conteneurs sont créés avec chacun une soft limite fixée à 2 GB Linux ne pourra pas garantir ces valeurs.

La valeur idéale de cette limite est le WS de l'application [50]. Dans ce cas, l'application conserve de bonnes performances puisque son WS est en mémoire.

4.2.3 *Les limites de cgroup v2*

Comme nous l'avons vu dans la Section 3.3.4, la version 2 des cgroups amène son lot de nouveautés. Pour le cgroup mémoire, les limites suivantes remplacent les limites susnommées [70] :

MIN : Un cgroup ayant une empreinte mémoire inférieure à cette valeur ne sera pas réclamé. Néanmoins, si Linux n'arrive pas à

1. Dans la suite de ce manuscrit, nous n'utilisions pas ces nouvelles limites puisque nous lançons nos conteneurs via docker qui, lors de l'écriture de ces lignes, n'est compatible qu'avec cgroup v1 [144].

réclamer de la mémoire ailleurs il peut invoquer l'*OOM killer* sur ce `cgroup` afin de libérer de la mémoire.

LOW : Cette limite correspond à la `soft` limite de `cgroup v1`.

HIGH : Les tâches appartenant à un `cgroup` dont l'empreinte mémoire dépasse cette valeur subiront une forte réclamation mémoire.

MAX : Si l'empreinte mémoire du `cgroup` dépasse cette valeur et ne peut être réduite, l'*OOM killer* est invoqué.

Dans les faits, cette nouvelle mouture propose un renommage des deux limites existantes et introduit une nouvelle limite garantissant une empreinte mémoire minimale. De plus, les bases théoriques de ces limites sont conservées tout comme leur caractère statique. Nous allons, dans la suite de ce chapitre, démonter que ce dernier point est un problème.

4.3 ENVIRONNEMENT D'EXPÉRIMENTATION

Pour étudier l'efficacité des mécanismes de contrôle qu'offre le `cgroup` mémoire nous avons réalisé une expérience figurant une base de données `mysql` accédée par le *benchmark* `sysbench` [120, 1]. Nous renseignons d'abord les métriques auxquelles nous nous intéressons pour juger les mécanismes utilisés puis les caractéristiques matérielles et logicielles de l'environnement d'expérimentation.

4.3.1 *Métriques*

Dans nos expériences, nous souhaitons mesurer l'isolation ainsi que son impact sur les performances. Nous voulons aussi observer si la consolidation a lieu et établir une corrélation entre l'empreinte mémoire et les performances d'une application. En effet, si un conteneur a peu de mémoire ceci n'implique pas qu'il n'ait pas de bonnes performances si son WS est petit [50].

Ainsi, pour mesurer l'isolation et la consolidation nous nous intéressons aux métriques système suivantes :

L'EMPREINTE MÉMOIRE : L'empreinte mémoire d'un conteneur correspond à la quantité de mémoire qu'il utilise pour un instant donné. Les données de la base de données qui auront déjà été accédées seront stockées dans la mémoire du conteneur, plus particulièrement dans le *page cache*.

LE NOMBRE DE LECTURES DEPUIS LE DISQUE : Ce nombre représente les lectures effectuées par un conteneur pour un instant donné. Idéalement, ce nombre doit être le plus petit possible puisqu'une lecture vers le disque est lente et a un fort impact sur les performances du conteneur.

Pour mesurer les performances de l'application, nous observons son débit. Il s'agit du nombre de transactions que le *benchmark* est capable de traiter en une seconde. Une transaction est un ensemble de

requêtes adressées à une base de données. Plus le débit est élevé, plus le conteneur a de bonnes performances.

Le débit est obtenu depuis la sortie standard du *benchmark* tandis que les empreintes mémoires et lectures depuis le disque sont collectées par `docker`.

4.3.2 *Environnement matériel*

Nos premières expériences ont été lancées sur une station de travail Dell T3600 [49]. Cette machine possède un processeur Intel®Xeon®E5-1603 épaulé par 8 GB de mémoire DDR3 et un SSD [75].

Dans un second temps, nous avons mené nos expériences sur des machines appartenant au banc de test Grid'5000 [22, 96]. Elles sont équipées de deux processeurs Intel®Xeon®Gold 6130 cadencés à 2.1 GHz [76]. Les processeurs sont accompagnés de 192 GB de mémoire DDR4 et un SSD officie pour le stockage.

Les conclusions de ces deux séries d'expériences étant les mêmes, nous avons choisi de ne détailler ici que les chiffres obtenus suite au lancement de nos expériences sur la machine appartenant à Grid'5000. De plus, ces machines ont une architecture typée serveur convenant à une utilisation dans le *cloud*.

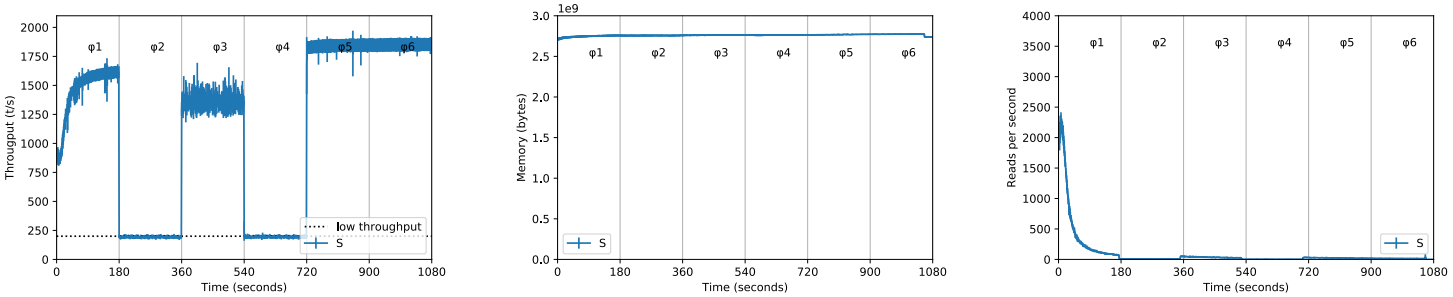
4.3.3 *Environnement logiciel*

Pour créer, artificiellement, de la pression mémoire afin que certaines limites s'activent, nous limitons la taille de la mémoire. Notre expérience est, pour cela, lancée dans une VM à laquelle 4 cœurs et 3 GB de mémoire ont été dédiés. Par ce fait, nous simulons la présence d'autres conteneurs générant de la pression mémoire. L'interprétation des résultats est aussi plus aisée. Le comportement de plusieurs conteneurs est toutefois étudié dans la Section 6.3.

Notre VM est gérée par l'hyperviseur `qemu 3.1.0` tandis que la gestion des conteneurs revient à `docker 19.03.2` et sa bibliothèque `python docker-py 4.0.2` [133, 52, 56, 98]. Le *benchmark* utilisé est `sysbench oltp` [1]. Nous avons modifié le code de ce dernier afin de faire varier le débit au cours du temps [99, 97]. Le *benchmark* interagit avec une base de donnée `mysql 5.7` [120]. La version du noyau Linux utilisée est la 4.19 [91].

4.4 EXPÉRIENCE DE RÉFÉRENCE

Afin de comparer efficacement les différents mécanismes, nous avons besoin de chiffres de référence. Ces chiffres vont nous permettre de comparer les mécanismes par rapport à des résultats idéaux, c'est-à-dire lorsque l'application s'exécute seule et sans être limitée. Pour les obtenir, nous avons mené une expérience de référence, ne figurant qu'un seul conteneur exécutant le scénario du conteneur B tel que décrit dans le Tableau 4. La Figure 17 montre les résultats obtenus pour cette expérience.



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 17 – Expérience de référence

Ainsi, la Figure 17a montre que le conteneur gère jusqu'à 2000 transactions par seconde (t/s), nous choisissons donc cette valeur comme référence. Comme le dépeint la Figure 17b, l'empreinte mémoire du conteneur est de 2.8 GB et ne varie pas beaucoup au cours du temps. Le nombre de lectures depuis le disque, visible sur la Figure 17c, est d'abord élevé au début de l'expérience, puisque le conteneur ramène des informations du disque vers la mémoire. Il diminue ensuite, puisque le conteneur a ramené les informations qui lui sont utiles en mémoire, pour atteindre une valeur d'environ 100 lectures par seconde.

4.5 SCÉNARIO D'EXPÉRIENCE AVEC DEUX CONTENEURS

Après avoir obtenu les chiffres de référence, nous pouvons désormais lancer les expériences visant à étudier les différents mécanismes offerts par le `cgroup` mémoire. Nous pourrons, en effet, comparer les résultats de chaque expérience à la référence. Notre expérience comporte deux conteneurs : A et B. Chacun exécute une charge de travail, dite *OnLine Transaction Processing* (OLTP), tout en expérimentant, au cours du temps, des variations dans celle-ci. Chacun de nos conteneurs, A et B, s'exécute avec deux cœurs et accède en lecture à une base de données de 4 GB.

Comme le ferait un administrateur fixant les limites d'un `cgroup`, nous fixons les SLO des deux conteneurs. Ainsi, le conteneur A possède un SLO fixé à 1700 t/s tandis que celui de B est défini à 1000 t/s, le conteneur A est donc plus prioritaire que le conteneur B. Un conteneur est dit satisfait, et ses performances sont dites satisfaisantes, si son SLO est respecté. Ces nombres sont représentés dans la Figure 18a à la Figure 20a par des lignes pointillées colorées. Les figures 18 à 20 comportent aussi les valeurs de référence, obtenues suite à l'expérience de la Section 4.4, présentées par des lignes noires horizontales discontinues.

Quand les deux conteneurs sont soumis à une forte charge de travail, nous nous attendons à observer l'isolation mémoire. C'est-à-dire qu'ils auront chacun de la mémoire et qu'aucun n'affamera l'autre. Dans le cas où un conteneur doit faire face à une forte activité pendant que

TABLE 4 – Charge de travail au cours des 6 phases de l'expérience pour les conteneurs A et B

Phases	Container A	Container B
$\varphi_1(h,h)$	forte charge	forte charge
$\varphi_2(h,l)$	forte charge	faible charge
$\varphi_3(h,i)$	forte charge	charge intermédiaire
$\varphi_4(l,l)$	faible charge	faible charge
$\varphi_5(i,h)$	charge intermédiaire	forte charge
$\varphi_6(s,h)$	conteneur arrêté	forte charge

son voisin reçoit une activité moindre, nous espérons observer de la consolidation mémoire. Le premier conteneur devrait donc récupérer la mémoire inutilisée du second afin d'obtenir les performances de référence.

Notre expérience est composée de 6 phases différentes, décrites dans le Tableau 4.

Avant de détailler chacune de ces phases, nous définissons la notation donnée dans la Formule 5 pour faciliter leurs descriptions.

$$\varphi_n(a,b) \tag{5}$$

Dans celle-ci, n correspond au numéro de la phase, allant de 1 à 6, a correspond à la charge reçue par le conteneur A et b à celle reçue par B. Les valeurs possibles pour a et b sont les suivantes :

— h : La charge de travail reçue est forte et correspond au débit de référence.
— l : Le conteneur est soumis à une faible activité. Nous avons fixé, arbitrairement, cette valeur comme valant 10% du débit de référence soit 200 t/s. Les lignes noires horizontales pointillées présentes dans les figures 18a à 20a dépeignent cette faible charge.
— i : Une charge de travail intermédiaire est envoyée au conteneur. Comme pour la faible charge, nous avons décidé de fixer la charge intermédiaire à 1500 t/s, soit 75% du débit de référence.
— s : Le conteneur est arrêté et ne reçoit aucune requête.

Chaque phase du scénario dure 180 s.

Dans un système idéal, les comportements suivants sont attendus pour chaque phase. Pendant la phase $\varphi_1(h,h)$, la mémoire physique disponible est inférieure à la somme des tailles des bases de données, nous sommes donc dans une situation de pression mémoire. Le *page cache* est forcé à rétrécir causant, ainsi, une augmentation du nombre de lectures depuis le disque et donc une diminution du débit. Dans $\varphi_2(h,l)$, si le `cgroup` mémoire permet la consolidation, A devrait être à même de prendre de la mémoire à B pour augmenter ses performances. D'une manière similaire à $\varphi_1(h,h)$, les performances devraient être faibles dans $\varphi_3(h,i)$. Dans $\varphi_4(l,l)$, les deux conteneurs devraient être capables de répondre à toutes les transactions qui leur sont adressées. $\varphi_5(i,h)$ est symétrique à $\varphi_3(h,i)$, comme dans celle-ci, les performances seront faibles. Enfin, pendant $\varphi_6(s,h)$, puisque A est arrêté B devrait obtenir les performances de référence.

Nous avons exécuté ce scénario 10 fois et calculé la moyenne et l'écart-type pour chaque métrique, ce pour chaque seconde. Nos résultats sont présentés dans les Figures 18 à 20.

4.6 DEUX CONTENEURS SANS LIMITE FIXÉE

Avec cette expérience nous étudions le comportement des conteneurs sans fixer de limite dans l'optique de respecter les SLO fixés ci-dessus.

La Figure 18 représente les résultats de cette expérience. Le débit au cours du temps est représenté sur la Figure 18a tandis que la Figure 18b et la Figure 18c montrent, respectivement, l'évolution des empreintes mémoires et les lectures depuis le disque des conteneurs au cours du temps.

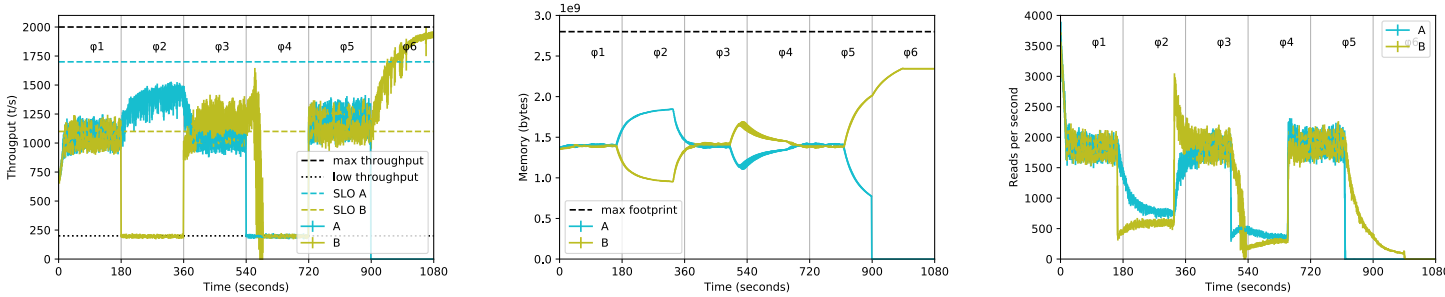
Dans $\varphi_1(h,h)$, les deux conteneurs reçoivent une forte charge. La Figure 18a montre que les deux conteneurs ont les mêmes performances. Ceci peut être expliqué par la Figure 18b, en effet, dans $\varphi_1(h,h)$, les deux conteneurs ont la même empreinte mémoire. Puisqu'il n'y a pas de mécanisme indiquant qu'un conteneur doit être plus réclamé que l'autre, ils ont donc les mêmes performances.

Dans $\varphi_2(h,l)$, le débit du conteneur recevant une forte charge de travail augmente aux dépens de celui soumis à une faible activité. Cette même phase, dans la Figure 18b, montre que l'empreinte mémoire du conteneur A augmente tandis que celle de B diminue. L'augmentation mémoire permet à A d'augmenter la taille de son *page cache* comme dépeint dans la Figure 18c. Néanmoins, même si B est soumis à une faible charge, il freine les performances de A, car son empreinte mémoire est toujours conséquente. Cette phase montre bien qu'il y a consolidation mais que celle-ci est imparfaite. En effet, le conteneur très actif n'atteint ni les performances de référence ni des performances satisfaisantes.

Dans $\varphi_3(h,i)$ et $\varphi_5(i,h)$, un conteneur reçoit une charge intermédiaire tandis que son voisin est soumis à une forte charge. Ces phases sont symétriques et présentent donc les mêmes résultats. Comme dans $\varphi_1(h,h)$, il n'y a pas de mécanisme indiquant qu'un conteneur doit être plus réclamé qu'un autre, les deux conteneurs ont donc des performances similaires. Il nous faut toutefois noter que le conteneur soumis à une charge intermédiaire présente de meilleures performances que l'autre conteneur. En effet, comme montré dans la Figure 18b, l'empreinte mémoire du premier est supérieure à celle du second.

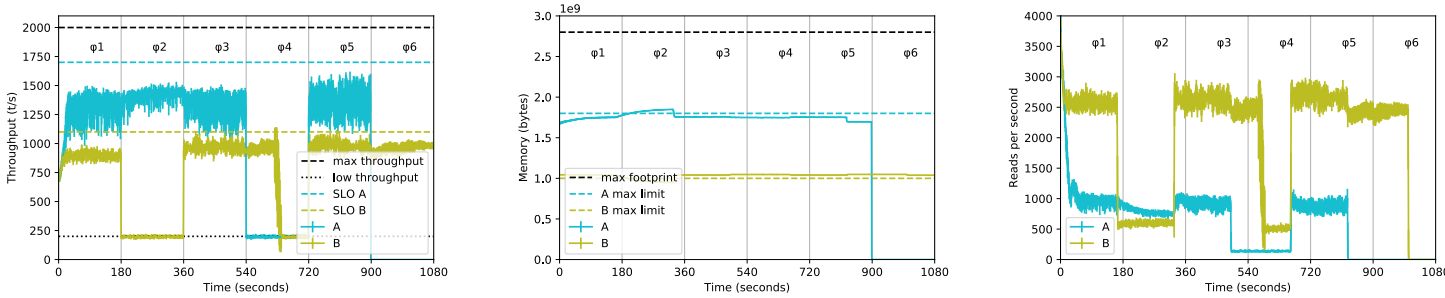
Pendant $\varphi_6(s,h)$, A est complètement arrêté, l'empreinte mémoire de B augmente donc et, par conséquent, il effectue moins de lectures depuis le disque. Il atteint donc les performances de référence. L'absence de limite permet, dans ce cas, d'atteindre les performances idéales, car B récupère la mémoire inutilisée par A.

Le lecteur averti aura remarqué que le débit du conteneur B subit un pic entre les phases $\varphi_3(h,i)$ et $\varphi_4(l,l)$. Puisque B est soumis à une forte charge dans $\varphi_3(h,i)$, les transactions qu'il n'a pas pu traiter dans cette



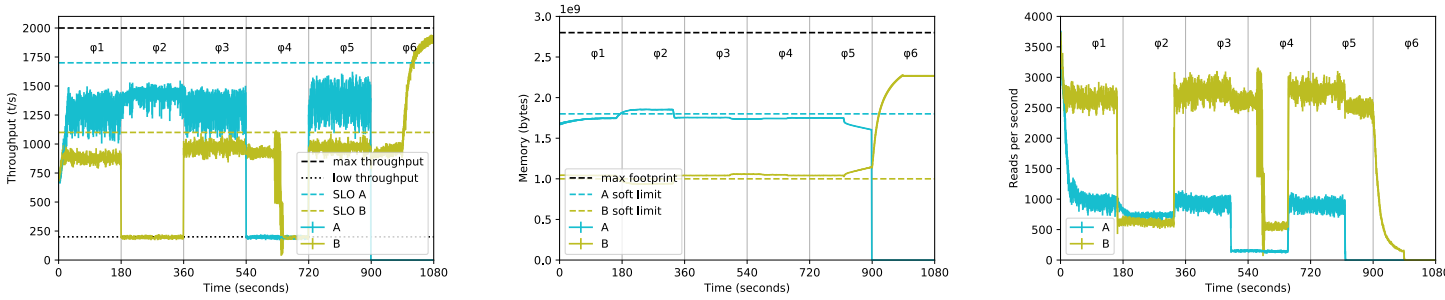
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 18 – Aucune limite fixée



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 19 – `Max` limites fixées à 1.8 GB (A) et 1 GB (B)



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 20 – `Soft` limites fixées à 1.8 GB (A) et 1 GB (B)

phase le sont dans $\varphi_4(l,l)$ puisque B dispose de plus de ressources. Ce comportement est visible dans les Figures 18a à 20a.

Ainsi, sans renseigner de limite, il n'y a pas d'isolation mémoire, il n'est donc pas possible d'assurer des performances satisfaisantes aux conteneurs. Néanmoins, il y a consolidation mémoire, mais celle-ci est imparfaite, car aucun conteneur n'atteint les performances de référence.

4.7 DEUX CONTENEURS AVEC max LIMITES

Dans la section précédente, nous avons vu que sans limite il n'y a pas d'isolation. Nous étudions donc la `max` limite pour régler ce problème. L'expérience est identique à celle lancée dans la section 4.6 mais nous avons, cette fois-ci, fixé les `max` limites des conteneurs à 1.8 GB et 1 GB pour A et B. Nous avons choisi ces valeurs de sorte que leur somme soit égale à l'empreinte mémoire de référence.

De plus, avec ces valeurs, le conteneur B ne freine pas les performances du conteneur A. En effet, le conteneur A a de meilleures performances dans les phases $\varphi_1(h,h)$, $\varphi_3(h,i)$ et $\varphi_5(i,h)$ dans la Figure 19a, comparée à la Figure 18a. Cette amélioration peut être expliquée en regardant la Figure 19b, celle-ci montre que les empreintes mémoires des conteneurs suivent leurs `max` limites. Puisque A a plus de mémoire que son voisin, il effectue moins de lectures depuis le disque, comme montré dans la Figure 19c.

Mais la `max` limite bloque la consolidation mémoire comme montrée dans la phase $\varphi_6(s,h)$ de la Figure 19b. Dans cette figure, l'empreinte mémoire de B est freinée par sa `max` limite. Il ne peut donc pas atteindre les performances de référence, alors qu'il n'y a qu'un seul conteneur actif, comme le montre la comparaison de la Figure 19a avec la Figure 18a.

De plus, dans $\varphi_2(h,l)$, la `max` limite présente le même problème que l'exécution sans limite. Le conteneur A, recevant une forte charge, ne peut pas réclamer la mémoire du conteneur B qui reçoit une faible charge. Ce comportement illustre le caractère statique de la `max` limite. Celle-ci ne prend pas en compte les variations dans la charge du travail des conteneurs.

La `max` limite permet l'isolation mémoire, puisque A possède plus de mémoire que B, mais bloque complètement la consolidation mémoire.

4.8 DEUX CONTENEURS AVEC soft LIMITES

Comme la `max` limite, la `soft` limite bloque la consolidation mémoire. Pour montrer ce fait, nous lançons une troisième expérience. Celle-ci est similaire à celle de la section 4.7 mais nous avons renseigné les `soft` limites en lieu et place des `max` limites.

Les résultats de la Figure 20 sont très similaires à ceux dépeints dans la Figure 19. En effet, de $\varphi_1(h,h)$ à $\varphi_5(i,h)$, notre système est sous pression mémoire. La mémoire des conteneurs est donc réclamée et leurs empreintes mémoires suivent leurs `soft` limites, qui ont les

TABLE 5 – L'isolation et la consolidation mémoire des différents mécanismes existants sous Linux

Mécanisme utilisé	Consolidation mémoire	Isolation mémoire
limite non renseignée	faible	non
`Max` limite	non	oui
`Soft` limite	non (sous pression mémoire)	oui

mêmes valeurs que les `max` limites renseignées précédemment. La `soft` limite garantit donc l'isolation sous pression mémoire.

Comme les deux autres mécanismes, dans $\varphi_2(h, l)$, la `soft` limite ne permet pas à A d'obtenir les performances de référence.

La `soft` limite diffère de la `max` limite dans la phase $\varphi_6(s, h)$ de la Figure 20b. Contrairement à cette même phase dans la Figure 19b, l'empreinte mémoire de B augmente. Puisque A est arrêté, il n'y a plus de pression mémoire et la `soft` limite n'est donc pas activée. L'empreinte mémoire du conteneur B peut donc augmenter, tout comme ses performances qui atteignent celles de référence.

La `soft` limite permet donc l'isolation mémoire, le conteneur B a moins de mémoire que son voisin, tant qu'il y a de la pression mémoire. Dans cette configuration, elle bloque la consolidation mémoire, mais celle-ci devient possible quand il n'y a plus de pression mémoire.

4.9 CONCLUSION

Dans ce chapitre, nous avons mené des expériences visant à mettre en évidence les problèmes du `cgroup` mémoire. En effet, les mécanismes existants du `cgroup` mémoire ne permettent pas de combiner de façon satisfaisante l'isolation et la consolidation mémoire.

Sans renseigner de limite, la consolidation mémoire prend place, imparfaitement, et il n'y a pas d'isolation mémoire. La `max` limite permet l'isolation mais bloque la consolidation. La `soft` limite est similaire lors d'une pression mémoire, mais la consolidation devient possible quand la mémoire est en quantité suffisante. Ces résultats sont résumés dans le Tableau 5.

De plus, les `max` et `soft` limites sont des mécanismes statiques. Elles ne peuvent donc pas s'adapter aux variations de charge de travail, particulièrement quand les applications font face à une faible charge de travail ou sont totalement arrêtées [24, 25, 111, 103].

Un mécanisme dynamique nécessiterait de corréler les performances de l'application à l'utilisation de ses ressources. Baser cette corrélation sur des métriques noyaux, comme la fréquence des défauts de page, les entrées-sorties, le nombre de cycles CPU consommés ou les empreintes mémoires, n'est pas chose aisée.

Dans le chapitre suivant, nous présentons notre contribution, MemOpLight, qui permet une meilleure répartition de la mémoire entre les conteneurs en corrigeant le problème du `cgroup` mémoire pour offrir à la fois l'isolation et la consolidation.

5 MEMOPLIGHT : PRINCIPE DE FONCTIONNEMENT

5.1 INTRODUCTION

Dans le Chapitre 4, nous avons montré qu'il est difficile de garantir à la fois l'isolation et la consolidation mémoire en utilisant les fonctionnalités offertes par le `cgroup` mémoire.

Il en résulte donc un problème pour le client et le fournisseur de *cloud*. En effet, l'absence d'isolation, dans une plateforme *cloud* signifie que le conteneur d'un client pourrait affamer ceux des autres. Ce comportement n'est pas souhaitable pour le client ni pour le fournisseur. En effet, il en résulterait une violation du SLA et ce dernier devrait donc payer des pénalités au premier [125, 6]. De plus, l'absence de consolidation empêche une utilisation optimale des ressources de la plateforme et donc un manque à gagner pour le fournisseur. De l'autre côté, si le conteneur d'un client n'est pas limité dans son utilisation des ressources, le coût de celui-ci sera onéreux, puisqu'un client est facturé pour ce qu'il utilise [10].

D'autre part, les mécanismes actuels offerts par le noyau Linux pour garantir l'isolation sont purement statiques et ne s'adaptent donc pas à des charges de travail dynamiques. Ainsi, fixer une `max` ou `soft` limite pour les requêtes reçues à 10 h du matin ne garantit absolument pas que le serveur conteneurisé pourra faire face au pic de requêtes arrivant à 18 h. Symétriquement, renseigner la valeur de ces limites par rapport aux besoins du précédent pic implique un surdimensionnement des ressources le reste du temps et donc une surfacturation pour le client.

De surcroît, les articles détaillés dans le Chapitre 2 ont montré qu'un dimensionnement basé sur les performances telles que perçues par une application permet d'être plus proche de ses besoins. Quoi de mieux que l'application elle-même pour juger de ses performances ? En effet, les métriques disponibles au niveau du système d'exploitation sont totalement déconnectées de la satisfaction comme la voit l'application.

Néanmoins, une approche basée uniquement sur l'application perd la vision globale des ressources qu'offre une vue bas niveau telle que celle du système ou de l'hyperviseur. En effet, avec le *page cache* une application a toujours besoin de mémoire, il faut savoir limiter la taille de celui-ci quand la ressource mémoire se fait rare.

C'est pourquoi, dans cette thèse, nous proposons MemOpLight[8]. Notre contribution peut être vue comme une double boucle MAPE. La première collecte des informations sur l'état de performance des applications vis-à-vis d'un SLO et en informe le noyau. Quant à la seconde, elle réclame la mémoire de certains conteneurs en se basant sur les informations fournies par la première.

[8] *Notre solution utilise les couleurs des feux de circulation pour indiquer l'état de performance d'un conteneur.*

MemOpLight vise à s'exécuter dans le *cloud*, mais cet environnement est régi par des accords fixant la qualité de service, notre mécanisme

doit donc prendre en compte ceux-ci. Ainsi, dans la suite de ce chapitre, nous commençons par définir les différents termes liés à la qualité de service pour terminer par une présentation de notre mécanisme.

5.2 QUALITÉ DE SERVICE

Dans l'introduction, nous avons indiqué que la location de ressources par un client à un fournisseur de cloud faisait l'objet d'une SLA. Nous revenons donc sur les différents termes utilisés dans ces ententes puis nous en donnons des exemples, fictifs ou réels.

5.2.1 *Termes liés à la qualité de service*

Les SLA se basent sur plusieurs termes, dont les sigles, se rapprochant l'un l'autre, ont des significations divergeant fortement. Dans les faits, une *Service Level Agreement* (SLA) se base sur des *Service Level Objectives* (SLO) [27, Chapter 4]. Ceux-ci sont définis comme une valeur cible mesurée par un *Service Level Indicator* (SLI). Un SLI est une mesure quantitative d'un aspect du service fourni comme la latence des requêtes ou la disponibilité[9].

Ainsi, un SLO garantit que la valeur mesurée, par exemple, de la disponibilité sera supérieure ou égale au SLI associé. Si cette valeur devient inférieure, le SLA n'est pas respecté.

Dans le cas de MemOpLight, un SLO[10] est passé en paramètre d'une boucle MAPE. Notre mécanisme fera tout son possible pour garantir le respect de celui-ci en informant le système d'exploitation qui fera alors varier l'allocation mémoire.

[9] *Comme nous l'avons indiqué dans l'introduction, la disponibilité est le SLI le plus utilisé par les fournisseurs de cloud.*

[10] *Pour MemOpLight, les termes SLO et «performance satisfaisante» sont interchangeables.*

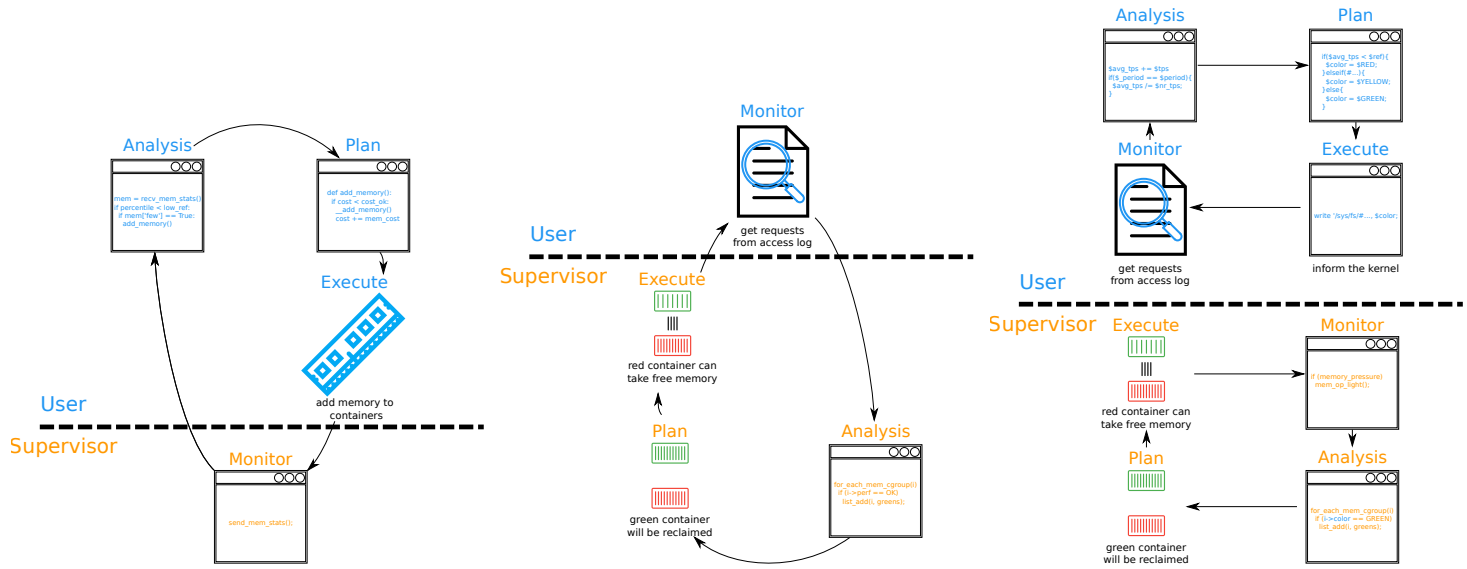
5.2.2 *Exemples de SLA*

Les SLA varient en fonction du service proposé. Ainsi, pour un IaaS, la SLA la plus utilisée se base sur la disponibilité. Par exemple, un fournisseur de *cloud* s'engage à ce que son service ait une disponibilité mensuelle d'environ 99.9% [117, 125, 6]. Tandis que pour un *Software as a Service* (SaaS) ou *Platform as a Service* (PaaS), il existe des SLA plus variés.

Le service de base de données Azure Cosmos DB est, à notre connaissance, le seul PaaS à garantir, en plus de la disponibilité, un temps de réponse borné. Celui-ci est de 10 ms, respectivement 15 ms, pour des opérations de lecture, respectivement d'écriture, ayant réussi en renvoyant moins de 1 kB de données [116].

En s'inspirant de ce que propose Azure Cosmos DB, il devient possible d'imaginer un SLA pour un serveur web où le SLO serait la latence des requêtes. Cet engagement pourrait spécifier que les requêtes ayant renvoyé un code 200 et moins de s octets aient une latence inférieure à t ms.

Pour un SaaS, il est possible d'imaginer une application de *streaming* faisant l'objet d'un SLA concernant la qualité de son flux et son débit d'informations par seconde. Le fournisseur d'un tel service orienté



(a) Boucle principalement en espace utilisateur (b) Boucle principalement en espace noyau (c) Double boucle

FIGURE 21 – Les différentes boucles possibles

vidéo pourrait s'engager sur l'un de ces critères, par exemple, un flux d'image définie en $1920 * 1080$ pixels, ou les deux, ce même flux avec un débit de 60 images par seconde. Ainsi, le client pourrait donc choisir le SLA correspondant le plus à son besoin.

5.3 MEMOPLIGHT

Afin d'allier les avantages de l'exécution en espace utilisateur à ceux procurés par l'espace noyau, plusieurs solutions sont possibles. En effet, les éléments de la boucle MAPE peuvent être répartis dans ces deux espaces.

Dans les sous-sections à venir, nous étudions les faiblesses d'une boucle prenant place principalement en espace utilisateur, respectivement noyau. Nous concluons par l'étude d'une double boucle permettant de pallier aux inconvénients des deux précédentes.

5.3.1 *Boucle principalement en espace utilisateur*

Comme représenté par la Figure 21a, il est possible d'imaginer une boucle dont les éléments siègent principalement en espace utilisateur. Ainsi, la décision est prise dans cet espace en se basant à la fois sur les performances de l'application et des informations transmises par le noyau.

Néanmoins, cette approche a deux principaux défauts. Premièrement, le noyau doit transmettre des informations aux applications ce qui peut résulter en des indiscrétions. Par exemple, l'application d'un client A pourrait inférer que l'application du client B est très active à partir de ces informations. Secondement, puisque chaque application prend une décision, celles-ci pourraient être en contradiction. Il est possible que ces décisions convergent, mais cela ne se ferait qu'au

bout d'un certain temps conduisant possiblement à un non-respect de la SLA.

Pour pallier ces problèmes, une boucle dont les éléments prennent principalement place en espace noyau semble être une réponse convenable.

5.3.2 *Boucle principalement en espace noyau*

Symétriquement à la boucle présentée dans la Figure 21a, celle montrée par la Figure 21b voit ses éléments occuper l'espace noyau. De cette manière, la décision est prise en espace noyau en se basant sur des informations transmises par les applications.

Néanmoins, cette approche doit être capable d'inférer l'état de performance de chaque application à partir de données hétérogènes. En effet, une application peut être un serveur web et une autre un logiciel de *streaming* vidéo. Il faut donc que l'algorithme de la phase *Analysis* soit capable d'inférer l'état de performance d'une application pour des métriques aussi variées qu'une latence ou un nombre d'images par seconde. Nous doutons qu'un algorithme aussi générique soit réalisable en espace noyau compte tenu des limitations de celui-ci évoquées dans la Section 2.7.1.

Cette approche a donc, comme l'approche précédente, des inconvénients. Une solution s'appuyant sur une double boucle, prenant place à la fois en espace utilisateur et noyau, nous semble être une solution de choix.

5.3.3 *MemOpLigt : une double boucle*

MemOpLight, la contribution principale de cette thèse, permet de remédier aux problèmes des deux approches susmentionnées. Ce mécanisme peut être représenté comme deux boucles MAPE chaînées entre elles. La Figure 21c montre cette représentation.

L'une de ces boucles correspond au contrôleur applicatif, et prend donc place en espace *user*, tandis que l'autre correspond au mécanisme noyau, et siège donc en espace *supervisor*. De cette manière, MemOpLight profite des avantages d'une double vue tels qu'énoncés dans le Tableau 2.

Le contrôleur applicatif, de la première boucle MAPE, permet de collecter les performances de l'application conteneurisées. Ces performances sont ensuite analysées et un compte-rendu de celles-ci est transmis au noyau Linux via un moyen de communication *user-supervisor*. Enfin, et à partir de ces données, un mécanisme d'arbitrage, résidant dans le noyau et représenté par la seconde boucle MAPE, permet de réclamer de la mémoire aux conteneurs présentant de bonnes performances. La mémoire réclamée pourra être utilisée par les conteneurs ayant des performances moindres dans l'optique d'augmenter celles-ci.

Dans le reste de chapitre, nous étudions, dans le détail, les composants de chacune de ces boucles.

5.4 LE CONTRÔLEUR APPLICATIF

Le contrôleur applicatif correspond à la boucle MAPE de l'espace *user* dans la Figure 21c. Il a pour but de vérifier que la SLA est respectée. Pour cela, il sonde les performances de l'application.

Comme tout instrument de mesure, il ne doit pas perturber les mesures effectuées [130]. De plus, ce mécanisme s'intègre à des applications existantes. Il doit donc, pour être couramment utilisé, accéder à des informations déjà disponibles pour minimiser les modifications à apporter aux applications . Dans les faits, un contrôleur ne fait que lire des métriques qui sont déjà calculées, il suffit alors de les exposer pour qu'il puisse rendre son verdict.

Dans cette section, nous décrivons, dans un premier temps, le but d'un contrôleur applicatif. Nous détaillons, dans un deuxième temps, son fonctionnement en détaillant chacune des étapes de la boucle MAPE. Dans un troisième et dernier temps, nous donnons des exemples de sondes pour différents types d'application.

5.4.1 *But d'un contrôleur applicatif*

Un contrôleur applicatif doit informer le noyau du respect de la SLA. Il utilise, pour ce faire, les métriques de l'application conteneurisée plutôt que des métriques noyaux qui sont trop éloignées des performances telles que vues par les applications. De plus, le contrôleur contextualise ces performances. En effet, celles-ci peuvent être incomplètes. En pratique, le contrôleur scrute plusieurs données :

L'ENTRÉE : C'est la charge du système en entrée. Par exemple, un serveur web a reçu 1000 requêtes cette seconde.

LA SORTIE : C'est ce qu'a pu traiter le système. Le serveur web a répondu à 900 requêtes cette seconde.

LA PERFORMANCE DE RÉFÉRENCE : Les performances du conteneur sont jugées satisfaisantes quand sa sortie est supérieure ou égale à cette donnée, fournie par l'utilisateur. Dans notre exemple, si l'utilisateur a fixé cette valeur à 850 requêtes par seconde, le serveur web conteneurisé a des performances satisfaisantes.

Le contrôleur doit absolument contextualiser les métriques de l'application. En effet, si dans notre exemple le serveur n'avait reçu, en entrée que 100 requêtes, et qu'il les avait toutes traitées, ses performances doivent alors être jugées comme satisfaisantes bien qu'elles soient inférieures au SLO donné par l'utilisateur.

5.4.2 *Étape Monitor*

Le contrôleur applicatif prend place dans un conteneur aux côtés de l'application. Il collecte les métriques mises à disposition par celle-ci. Ainsi, son intrusivité est limitée.

Dans l'exemple donné dans la Figure 21c, le contrôleur récupère les requêtes traitées par le serveur web depuis son fichier `access.log`.

5.4.3 *Étape Analysis*

Dans cette étape, les métriques collectées dans la phase *Monitor* sont transformées pour pouvoir être comparées au SLO fourni par l'utilisateur. Le calcul d'une moyenne ou d'un 95e centile sont de bons exemples des transformations que peut effectuer un contrôleur.

Dans la Figure 21c, le contrôleur calcule une moyenne du nombre de requêtes pour une période donnée, par exemple chaque seconde.

5.4.4 *Étape Plan*

Le contrôleur compare ici les performances de l'application au SLO. Selon le résultat de ces comparaisons, l'état de performance du conteneur est qualifié par 3 couleurs :

ROUGE : Les performances du conteneur sont inférieures aux performances de référence.

JAUNE : Le conteneur atteint les performances de référence, mais pourrait faire mieux si des ressources lui étaient allouées[11].

VERT : Les performances du conteneur sont les meilleures possibles.

Dans notre exemple dépeint par la Figure 21c, la couleur du conteneur est décidée suite à la comparaison de la moyenne du nombre de requêtes avec le SLO. L'exemple donné ici est volontairement simplifié puisqu'il ne prend pas en compte l'entrée du système contrôlé.

[11] *L'article R312-29 du code de la route stipule ceci [36] :*

> *Les feux de signalisation lumineux réglant la circulation des véhicules sont verts, jaunes ou rouges.*

5.4.5 *Étape Execute*

La couleur du conteneur, calculée dans l'étape antérieure, est ensuite communiquée au noyau via un moyen de communication *user-supervisor*.

Nous étudions d'abord les différents moyens de communication que nous aurions pu utiliser et pourquoi notre choix s'est porté sur le `sysfs`. Puis, nous montrons, dans les faits, comment cette communication s'effectue.

Interface user-supervisor

De multiples moyens de communication entre l'espace utilisateur et l'espace noyau existent comme les appels système, les *sockets* ou le `sysfs`. Un appel système peut sembler être la voie royale pour communiquer avec le noyau. Néanmoins, l'ajout d'un nouvel appel nécessite de modifier des domaines du noyau non relatifs à la mémoire. Son utilisation se fait via son invocation ce qui implique d'utiliser un langage bas niveau ou une interface depuis un langage haut niveau.

Les `ioctls` forment une alternative aux appels système [148]. En effet, l'ajout d'un nouvel `ioctl` est plus aisé que celui d'un appel système. Néanmoins, l'utilisation se fait via l'invocation de la fonction `ioctl` avec comme argument des drapeaux particuliers. Ainsi, le problème de l'appel système n'est pas solutionné.

Le `sysfs` est un système de fichier virtuel, les fichiers n'existent pas sur le disque, permettant d'interagir avec le noyau. Comparé aux deux autres mécanismes, il offre une interaction simplifiée basée sur des opérations de lecture/écriture dans des fichiers. Ce mécanisme est, entre autres, utilisé par les `cgroups` pour communiquer avec l'espace utilisateur.

Nous pensons que MemOpLight doit rester simple d'utilisation, surtout que le contrôleur est la seule partie devant être écrite par l'utilisateur. Une communication basée sur des entrées/sorties, comme l'offre le `sysfs`, nous semble donc être la meilleure solution.

Utilisation du `sysfs`

Dans notre cas, nous avons rajouté, pour chaque instance du `cgroup` mémoire, un fichier `memory.color`. Ce fichier est écrit par le contrôleur du conteneur pour y indiquer l'état de performance de ce dernier. Pour ce faire, et comme le montre le code suivant, un simple `echo`, en tant que superutilisateur, suffit :

```
GREEN=3
echo $GREEN > /sys/fs/cgroup/memory/docker/${container_id}/memory.color
```

La couleur est ensuite lue par notre mécanisme de réclamation qui prend place dans le noyau Linux.

5.4.6 *Exemples de sonde*

Il est assez facile d'imaginer des contrôleurs applicatifs pour différentes applications.

Les performances d'un conteneur exécutant un serveur web seraient jugées par rapport à la latence de ses requêtes. Ces latences sont récupérées depuis le fichier `access.log` du serveur. À partir de celles-ci, le contrôleur peut calculer le 99e centile pour les requêtes reçues la dernière seconde. Cette valeur est ensuite comparée à une latence de référence fournie par l'utilisateur. Si la première est inférieure à la seconde, le conteneur est rouge, si elle est égale, il est jaune, sinon il est vert. Cette couleur est ensuite transmise au noyau par le biais d'un moyen de communication, décrit dans la Section 5.4.5.

Pour une base de données, la latence des requêtes est aussi examinée. Celle-ci peut être lue en détournant le `slow_query_log` de `mysql` pour qu'il affiche les latences de toutes les requêtes [150]. Le reste du procédé est identique au contrôleur décrit ci-dessus.

Dans le cas d'une application de *streaming*, la définition de l'image et/ou le nombre d'images par seconde peuvent être mesurés pour statuer sur les performances du conteneur. Pour `OBS Studio`, le débit en termes d'images par seconde est disponible dans le code source [83]. Il faudrait donc afficher cette valeur chaque seconde pour qu'elle puisse être lue par le contrôleur. Similairement au serveur web, cette valeur est ensuite comparée à un débit de référence fourni par l'utilisateur ; la couleur du conteneur découle alors de cette comparaison.

5.5 MODIFICATIONS APPORTÉES AU NOYAU LINUX

Le noyau permet d'avoir une vue globale de l'utilisation des ressources. De plus, c'est lui qui est en charge de la gestion de la mémoire notamment quand celle-ci se fait rare. Il est donc tout naturel que MemOpLight siège dans le noyau [61].

Néanmoins, notre mécanisme doit avoir un temps de calcul limité, puisqu'il s'exécute en interrompant du code utilisateur, et avec une empreinte mémoire minimale, car il est invoqué lors d'une pression mémoire. De plus, et pour simplifier la maintenabilité de notre code, nous avons limité le nombre de sous-systèmes modifiés.

Dans la suite de cette section, nous analysons, les différentes étapes de la boucle MAPE, montrées dans la Figure 21c, siégeant dans le noyau. Nous finissons par détailler les modifications apportées au noyau pour introduire notre mécanisme.

5.5.1 *Étape Monitor*

Cette étape est double. Premièrement, notre mécanisme n'est appelé que lorsque le système est en pression mémoire. En effet, nous avons implémenté celui-ci dans la fonction `mem_cgroup_soft_limit_reclaim` invoquée par la `soft` limite [149, fichier `mm/memcontrol.c`, ligne 2954]. Secondement, cette étape va récupérer la couleur des `cgroups`, qui a été communiquée par le contrôleur applicatif, via le champ `color`.

5.5.2 *Étape Analysis*

Cette phase procède à une filtration des conteneurs en fonction de leurs couleurs. Comme le montre la boucle ligne 5 de l'Algorithme 1, les conteneurs verts et jaunes sont regroupés dans deux listes distinctes, `greens` et `yellows`, tandis que les conteneurs rouges sont dénombrés.

Noter que, puisque cet algorithme effectue un parcours de la liste de tous les `cgroups` existants, sa complexité est donc en $\mathcal{O}(n)$.

5.5.3 *Étape Plan*

En se basant sur le travail effectué dans la phase précédente, cette étape va décider quels conteneurs doivent être réclamés. Plusieurs cas de réclamations sont possibles, ceux-ci sont compilés dans le Tableau 6.

Pour résumer, notre mécanisme vise d'abord à éliminer les `cgroups` rouges, c'est-à-dire ceux n'ayant pas des performances satisfaisantes. Puis, il améliore les performances des conteneurs jaunes, qui ont des performances satisfaisantes, mais pourraient faire mieux.

5.5.4 *Étape Execute*

C'est dans cette étape qu'a lieu la réclamation, si la phase *Plan* a décidé qu'elle devait avoir lieu. L'Algorithme 2 donne les étapes de la

```
1  Function mem_cgroup_soft_limit_special_reclaim()
2      reds = 0;
3      yellows = [];
4      greens = [];
       /* Cette boucle compte le nombre de cgroups rouges
          et ajoute les autres aux listes correspondantes.
          */
5      for iter ∈ mem_cgroups do /* Pour chaque cgroup dans
        le système.                                        */
6          if iter.color == RED then /* Si les performances du
            cgroup sont mauvaises.                         */
7              reds++;
8          else if iter.color == YELLOW then/* Si le cgroup a
            des performances satisfaisantes, mais pourrait
            faire mieux.                                   */
9              yellows.append(iter);
10         else if iter.color == GREEN then/* Si les
            performances du cgroup sont les meilleures
            possibles.                                     */
11             greens.append(iter);
12     end
13     reclaimed = 0;
14     expected = 0;
       /* La mémoire va être réclamée.                    */
15     if reds ∨ len(yellows) then /* S'il y a au moins un
        cgroup rouge ou un jaune, les cgroups verts sont
        réclamés.                                          */
16         green_reclaimed, green_expected =
            reclaim_cgroup_list(greens);
17         reclaimed += green_reclaimed;
18         expected += green_expected;
19     end
20     if reds then /* S'il y a au moins un cgroup rouge, les
        jaunes sont aussi réclamés.                        */
21         yellow_reclaimed, yellow_expected =
            reclaim_cgroup_list(yellows);
22         reclaimed += yellow_reclaimed;
23         expected += yellow_expected;
24     end
25     if expected ∧ reclaimed ≥ expected then /* Notre
        mécanisme s'arrête si suffisamment de mémoire a été
        réclamée.                                          */
26         return reclaimed
27     end
       /* Sinon, il s'arrête et indique l'échec de la
          réclamation.                                     */
28     return 0
```

Algorithme 1 : L'algorithme de MemOpLight

Cgroups rouges	Cgroups jaunes	Conséquence
0	0	Aucun cgroup n'est réclamé. Soit le système n'a pas de cgroup, soit «tout est au mieux».
0	≥ 1	Les cgroups verts sont réclamés.
≥ 1	0	Les cgroups verts sont réclamés.
≥ 1	≥ 1	Les cgroups verts sont d'abord réclamés puis les jaunes.

TABLE 6 – Les différents cas de MemOpLight

fonction `reclaim_cgroup_list`, en charge de récupérer de la mémoire aux `cgroups` d'une liste.

```
1  Function reclaim_cgroup_list(cgroups)
2      reclaimed = 0;
3      expected = 0;
4      for iter ∈ cgroups do
5          if ¬iter.reclaimed then /* Si le cgroup n'a pas été
               réclamé lors de la période courante.          */
               /* La quantité de mémoire à réclamer est
                  calculée.                                  */
6              to_take = mem_cgroup_usage(iter,false) / iter.divider;
               /* Puis le noyau essaye de libérer celle-ci.
                  */
7              local_reclaimed =
                 try_to_free_mem_cgroup_pages(iter, to_take);
8              reclaimed += local_reclaimed;
9              expected += to_take;
               /* Le cgroup est marqué comme ayant été
                  réclamé pendant cette période. La sonde se
                  chargera de mettre à false ce drapeau lors
                  de la prochaine période.                   */
10             iter.reclaimed = true;
11         end
12     end
13     return (reclaimed, expected)
```

Algorithme 2 : La fonction en charge de réclamer les `cgroups` d'une liste

Cette fonction réclame, pour chaque `cgroup` de la liste, une fraction de son empreinte mémoire, fraction pouvant être changée à la volée pour chaque `cgroup`. La réclamation s'effectue réellement dans la fonction `try_to_free_mem_cgroup_pages` [149, fichier `mm/vmscan.c`, `3298`, ligne .] Cette fonction, offerte par le noyau Linux, va agir sur le *page cache* du `cgroup`, comme résumé dans la Section 3.4.3, pour libérer des pages. L'utilisation de cette fonction nous permet de limiter l'intrusivité de notre mécanisme tout en rendant notre code résistant aux changements de l'implémentation du *page cache*.

Une fois que le `cgroup` est réclamé, il est marqué comme tel. Ainsi, il ne pourra pas être à nouveau victime du mécanisme pendant la période en cours. Nous revenons sur ce fait dans la sous-section suivante.

5.5.5 *Modifications apportées au noyau Linux*

Notre mécanisme prend place dans la fonction `mem_cgroup_soft_limit_reclaim` invoquée par la `soft` limite [149, fichier `mm/memcontrol.c`, ligne 2954]. L'Algorithme 3 montre les modifications que nous avons apportées à celle-ci.

```
1  mem_op_light_lock = init_lock();
2  Procedure timer()
3  |  unlock(mem_op_light_lock);
4  Function mem_cgroup_soft_limit_reclaim()
5  |  if lock(mem_op_light_lock) then /* Si le verrou est
   |   libre, soit MemOpLight n'est pas en cours
   |   d'exécution, soit sa dernière exécution remonte à
   |   la période précédente.                          */
6  |  |  reclaimed = mem_cgroup_soft_limit_special_reclaim();
7  |  |  if reclaimed then
   |  |  |  /* Un compte à rebours invoquera la fonction
   |  |  |     timer lors de la prochaine période.      */
8  |  |  |  set_timer(timer, 1) /* Si suffisamment de mémoire
   |  |  |     a été réclamée, la fonction est quittée.
   |  |  |     MemOpLight et la soft limite ne
   |  |  |     s'activeront pas jusqu'à la fin de cette
   |  |  |     période.                                 */
9  |  |  |  return reclaimed
10 |  |  end
11 |  |  unlock(mem_op_light_lock);
12 |  else
   |  |  /* Soit MemOpLight est en cours d'exécution soit
   |  |     il a déjà été invoqué cette période-ci.   */
13 |  |  return 0
   |  /* La fonction continue en appelant la soft limite.
   |     */
```

Algorithme 3 : Les modifications apportées à `mem_cgroup_soft_limit_reclaim`

Nous avons modifié cette fonction de sorte que MemOpLight soit appelé au plus une fois par période[12]. Ainsi, notre surcoût a, au pire, une complexité en $\mathcal{O}(n)$ chaque période. Si MemOpLight a réclamé suffisamment de mémoire, ni la `soft` limite ni lui ne seront rappelés pendant cette période. De cette façon, la `soft` limite ne vient pas détruire ce que MemOpLight a accompli. Si MemOpLight n'a, malheureusement, pu réclamer suffisamment de mémoire, la `soft` limite prend le relais.

[12] *Par défaut, une période dure une seconde.*

5.6 CONCLUSION

Nous venons, dans ce chapitre, de présenter MemOpLight. Comparée aux mécanismes existants, notre solution apporte au noyau Linux une réclamation basée sur les besoins des applications et non sur des limites statiques. Ainsi, MemOpLight garantit l'isolation et la consolidation mémoire.

De plus, nous avons restreint le caractère intrusif de notre mécanisme, car les modifications apportées au noyau Linux ne touchent que 2 fichiers dans un même sous-système. Nous avons aussi limité le surcoût de MemOpLight puisque sa complexité est en $\mathcal{O}(n)$ et qu'il s'exécute au plus une fois par période. Concernant les contrôleurs applicatifs, puisque ceux-ci accèdent à des informations déjà disponibles et informent le noyau au plus une fois par seconde nous avons la certitude que leur surcoût est faible.

Dans le prochain chapitre, nous mettons à rude épreuve MemOpLight d'abord en le comparant aux mécanismes existants. Puis nous faisons varier ses propres paramètres pour trouver une configuration optimale de ceux-ci.

6 MEMOPLIGHT : ÉVALUATION

6.1 INTRODUCTION

Après avoir mis en évidence, dans le Chapitre 4, les défauts de l'implémentation actuelle des `cgroups`, nous avons proposé, dans le Chapitre 5, MemOpLight. Ce mécanisme, basé sur une double boucle MAPE, assure et l'isolation et la consolidation.

Après avoir présenté notre solution, MemOpLight, nous allons évaluer celle-ci dans ce chapitre. Nous comparons donc MemOpLight aux différentes limites offertes par Linux. Pour cela, nous reprenons l'expérience, décrite dans la Section 4.3, qui a permis de montrer les faiblesses des mécanismes proposés par Linux. Nous mettons ensuite à l'épreuve MemOpLight, et les autres limites, dans un scénario plus réaliste. Nous terminons par étudier les différents paramètres de notre mécanisme.

6.2 MEMOPLIGHT AVEC DEUX CONTENEURS

Cette expérience reprend le scénario décrit dans la Section 4.5 tandis que l'environnement d'exécution est identique à celui détaillé dans la Section 4.3. La seule différence réside dans le fait que le mécanisme étudié ici est MemOpLight, les limites ayant les mêmes valeurs que dans les expériences analysées dans les Sections 4.7 et 4.8.

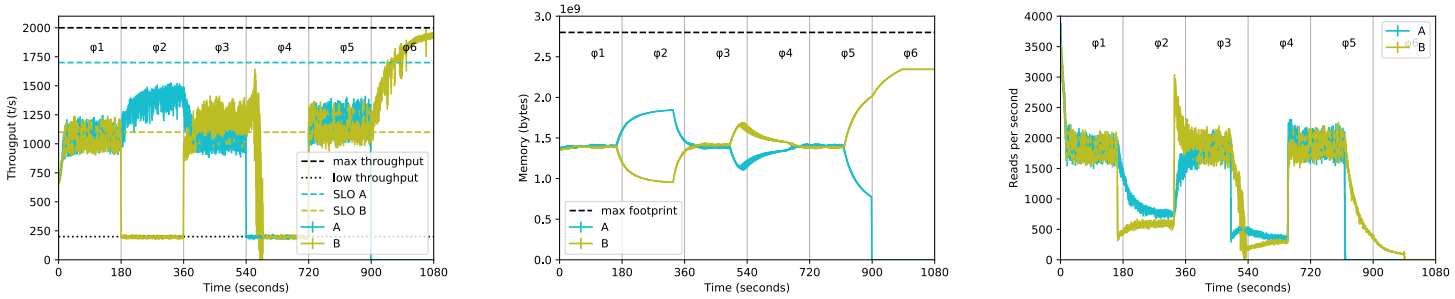
Pour faciliter la comparaison, les Figures 22, 23 et 24 sont rappelées. Les résultats de MemOpLight sont dépeints dans la Figure 25.

Nous étudions d'abord l'isolation proposée par MemOpLight puis la consolidation offerte par ce mécanisme.

6.2.1 *Isolation*

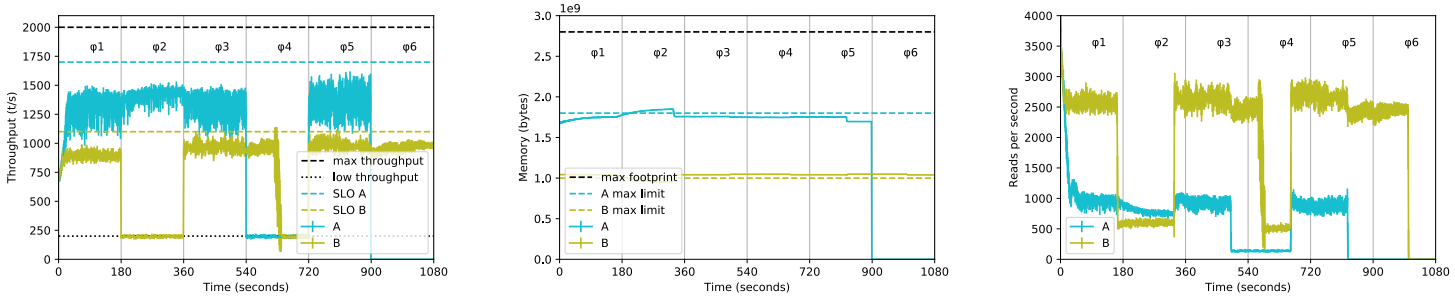
Les phases $\varphi_1(h,h)$, $\varphi_3(h,i)$ et $\varphi_5(i,h)$ de MemOpLight, montrées dans la Figure 25a, sont similaires à ces mêmes phases pour la `soft` limite, dépeintes dans la Figure 24a. Comme la Figure 24b, la Figure 25b montre que lorsque les deux conteneurs reçoivent une charge forte ou intermédiaire, dans les phases $\varphi_1(h,h)$, $\varphi_3(h,i)$ et $\varphi_5(i,h)$, leurs empreintes mémoires suivent leurs `soft` limites. Ainsi, le conteneur A a donc une empreinte mémoire supérieure à celle de B, MemOpLight permet donc l'isolation mémoire.

De plus, dans ces phases, les performances des conteneurs sont meilleures qu'avec les `max` et `soft` limites. En moyenne, le conteneur A répond à 1351, 1362 et 1370 t/s comparés à 1290, 1314 et 1358 t/s avec la `max` limite. Ceci représente une augmentation de, respectivement, 4.7%, 3.7% et 0.9%. Quant au conteneur B, avec MemOpLight, il gère 933, 999 et 1036 t/s et seulement 894, 968 et 978 t/s avec la `max`



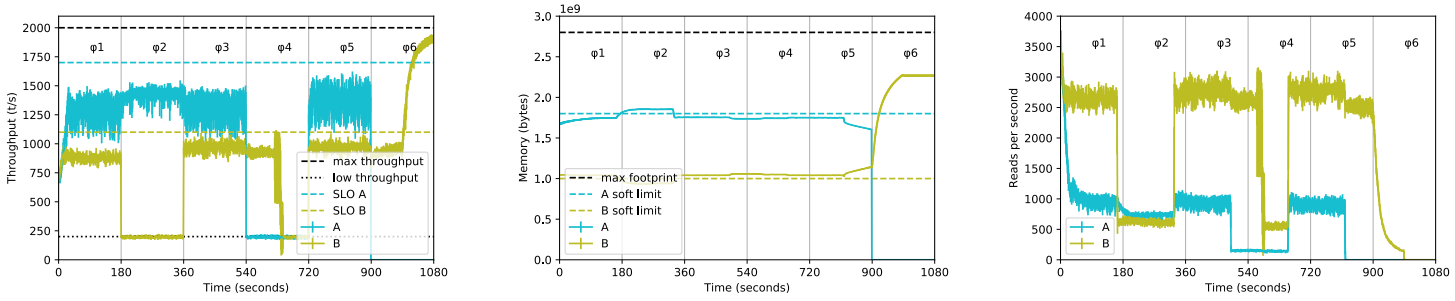
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 22 – Aucune limite fixée



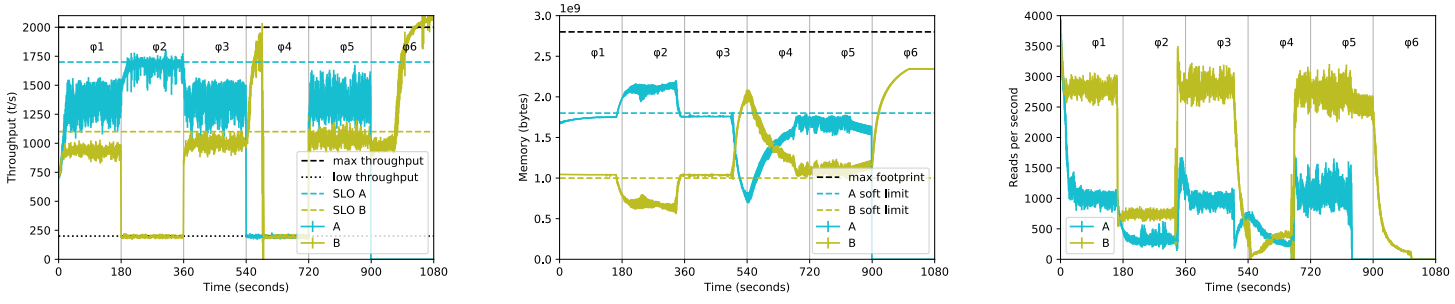
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 23 – `Max` limites fixées à 1.8 GB (A) et 1 GB (B)



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 24 – `Soft` limites fixées à 1.8 GB (A) et 1 GB (B)



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 25 – MemOpLight avec des `soft` limites fixées à 1.8 GB (A) et 1 GB (B)

limite. MemOpLight accroît donc les performances de 4.4%, 3.2% et 5.9%. Notre mécanisme permet de converger plus rapidement vers un équilibre où les conteneurs arrêtent de se voler de la mémoire ce qui explique ses performances légèrement meilleures.

6.2.2 *Consolidation lors d'une suspension d'activité*

Comme la phase $\varphi_6(s,h)$ dans la Figure 20b, quand A s'arrête complètement, l'empreinte mémoire de B augmente, au détriment de celle de A, et il atteint donc les performances de référence car il n'y a plus de pression mémoire.

Ce phénomène est similaire à cette même phase pour la `soft` limite et sans fixer de limite. En effet, dans ce cas de la `soft` limite ne s'active pas puisqu'il n'y a qu'un seul conteneur actif et donc plus de pression mémoire. Néanmoins, ce comportement semble peu réaliste, nous mettons à l'épreuve celui-ci dans l'expérience décrite dans la Section 6.3.

6.2.3 *Consolidation lors d'une baisse d'activité*

Le caractère dynamique de MemOpLight apporte une réelle plus-value dans la phase $\varphi_2(h,l)$. Tandis que dans cette phase, la `soft` limite montre ses faiblesses, notre mécanisme arrive à arbitrer. En effet, dans celle-ci, le conteneur B fait face à une faible charge tandis que A reçoit une forte charge. Puisque B n'a pas de requête en attente, sa mémoire est réclamée jusqu'au seuil lui permettant de répondre à 200 t/s. Le retour du contrôleur applicatif permet de trouver cette configuration mémoire sans exécuter d'analyse statique en amont. Ainsi, MemOpLight maximise la consolidation mémoire et le conteneur A atteint donc des performances satisfaisantes.

6.2.4 *Conclusion*

Le Tableau 7 compile toutes les mesures réalisées jusqu'à présent. Le conteneur B a des performances moindres avec MemOpLight comparée à l'exécution sans limite. Ce comportement est tout à fait normal puisque dans le second cas il n'y a pas d'isolation mémoire. Néanmoins, ce résultat n'est pas désiré dans une plateforme *cloud* puisqu'un client pourrait avoir payé plus qu'un autre pour avoir plus de ressources.

MemOpLight permet donc aux conteneurs d'obtenir de meilleures performances en redistribuant la mémoire selon les besoins de ces derniers.

6.3 MEMOPLIGHT AVEC HUIT CONTENEURS

Dans cette section, nous voulons consolider les résultats obtenus par MemOpLight. Pour ce faire, nous lançons une nouvelle expérience avec cette fois-ci huit conteneurs.

TABLE 7 – Valeur médiane du débit mesuré dans chaque phase pour chaque expérience (en t/s arrondi à l'entier le plus proche)

Phases / Transactions	$\varphi_1(h,h)$		$\varphi_2(h,l)$		$\varphi_3(h,i)$		$\varphi_4(l,l)$		$\varphi_5(i,h)$		$\varphi_6(s,h)$
Conteneur	A	B	A	B	A	B	A	B	A	B	B
Charge reçue	2000	2000	2000	200	2000	1500	200	200	1500	2000	2000
Aucune limite fixée (Section 4.6)	1053	**1043**	1374	196	1054	**1168**	195	197	1195	**1145**	**1751**
`Max` limite (Section 4.7)	1290	894	1411	196	1314	968	196	660	1358	978	962
`Soft` limite (Section 4.8)	1268	879	1423	197	1299	969	196	794	1360	970	974
MemOpLight (Section 6.2)	**1351**	933	**1670**	195	**1362**	999	196	197	**1370**	1036	1723

TABLE 8 – Débit dans chaque phase de l'expérience (en t/s)

Conteneur	Satisfaction (t/s)	Limite (si renseignée)	φ_1	φ_2	φ_3	φ_4	φ_5	φ_6	φ_7	φ_8	φ_9
A	1800	1400 MB	2500	1710	2500	200	1890	200	200	200	2500
B	1600	1000 MB	1520	2500	2500	200	2500	200	1520	2500	200
C	1400	800 MB	1330	1470	200	1470	2500	1330	1470	200	1470
D	1400	800 MB	2500	1470	200	200	200	1470	200	2500	2500
E	1200	600 MB	1140	1140	200	2500	1260	2500	2500	2500	200
F	1200	600 MB	2500	1260	200	2500	200	2500	1260	1260	2500
G	1000	400 MB	2500	200	2500	200	200	200	200	950	950
H	800	400 MB	840	2500	200	200	760	200	760	760	2500

Avant de détailler ces résultats nous allons d'abord décrire le scénario exécuté par chacun des huit conteneurs. Puis, nous décrivons l'environnement matériel et logiciel dans lequel cette expérience a été menée. Nous terminons ensuite par une analyse des résultats obtenus.

6.3.1 *Scénario*

Dans notre expérience, chacun des huit conteneurs reçoit une charge forte, intermédiaire ou faible. La charge de travail d'un conteneur change au cours des différentes phases de l'expérience. Celle-ci a été choisie aléatoirement afin de couvrir différents cas et mettre MemOpLight en difficulté.

Notre scénario et ses caractéristiques sont décrits dans le Tableau 8. La deuxième colonne contient la satisfaction d'un conteneur. Un conteneur est dit satisfait s'il atteint les performances renseignées dans la case correspondante. La troisième montre les valeurs des `max` et `soft` limites pour chaque conteneur. Les autres colonnes donnent les valeurs des charges de travail pour chaque phase de l'expérience. Les valeurs des charges de travail peuvent appartenir à trois catégories distinctes :

FORTE : Le conteneur fait face à 2500 t/s. Les phases où les conteneurs reçoivent une telle charge sont mises en exergue par les cellules noires du Tableau 8.

INTERMÉDIAIRE : Le conteneur reçoit le nombre de transactions définissant sa satisfaction ±5%. Dans le Tableau 8, les cellules grisées et légèrement grisées montrent les phases où les conteneurs sont soumis à une telle charge.

FAIBLE : Le conteneur ne reçoit que 200 t/s.

De plus, les différentes phases du scénario peuvent être groupées en quatre périodes différentes :

1. Dans φ_1 et φ_2, le système est fortement chargé.
2. La phase φ_3 représente une transition entre un système fortement chargé et un système peu chargé.
3. Les conteneurs reçoivent une faible charge de travail dans φ_4.
4. De φ_5 à φ_9, le système est modérément chargé, mais la charge de travail des conteneurs augmente au cours du temps.

6.3.2 *Environnement d'expérimentation*

La machine utilisée dans cette expérience est la même que celle de la Section 4.3. Néanmoins, la VM dispose maintenant de 16 cœurs et de 6 GB de mémoire. Chaque conteneur s'exécute donc sur 2 cœurs et accède, toujours en lecture seule, à une base de données de 2 GB. Les différentes quantités de mémoire ont été choisies de manière à conserver le même ratio entre la taille totale des bases de données et la mémoire de la VM que dans l'expérience précédente, à savoir :

$$\frac{db\_size \cdot nr\_containers}{vm\_size} = \frac{2 * 8}{6} = \frac{4 * 2}{3} \tag{6}$$

6.3.3 *Résultats*

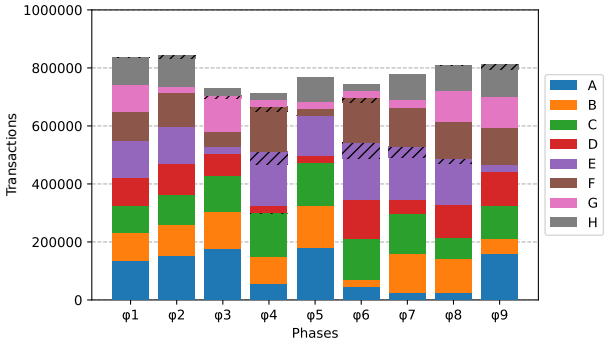
Dans cette expérience, nous nous focalisons principalement sur le respect de la satisfaction et les performances globales.

Performances brutes

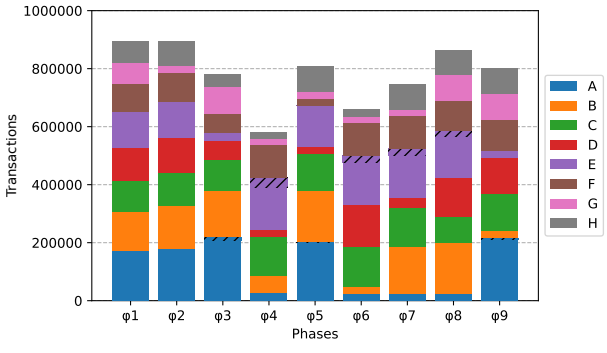
La Figure 26 montre le débit moyen des conteneurs généré sur 10 exécutions pour les différents mécanismes. La hauteur des barres correspond au débit global tandis que chaque couleur indique le débit du conteneur correspondant. Les zones hachurées montrent le surplus de transactions répondues par rapport aux performances satisfaisantes du conteneur.

Respect de la SLA

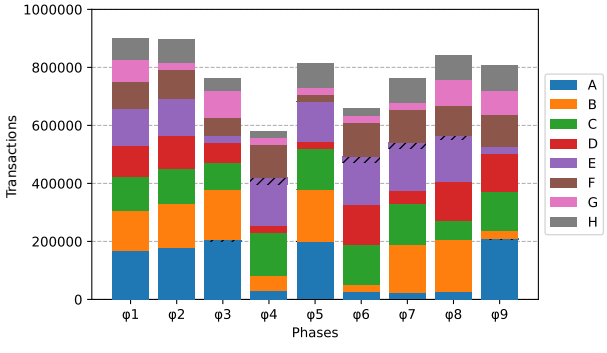
La Figure 27 met l'accent sur le respect de la satisfaction. Elle montre la couleur des conteneurs au cours du temps pendant la cinquième exécution de notre expérience. Pour dessiner ces courbes, nous avons activé le contrôleur pour chaque mécanisme. Celui-ci nous a permis d'obtenir la couleur des conteneurs pour chacun des mécanismes étudiés. Comme montré dans la Section 5.4.6, le surcoût du contrôleur est très faible.



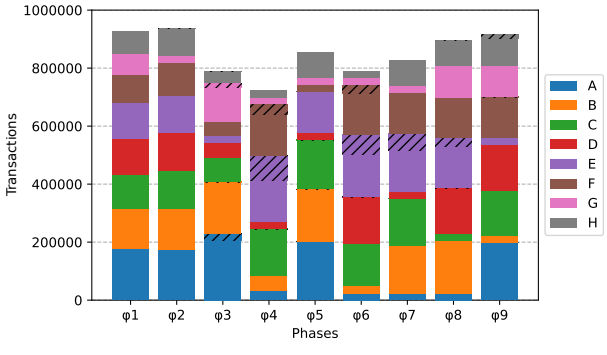
(a) Aucune limite fixée



(b) `Max` limites

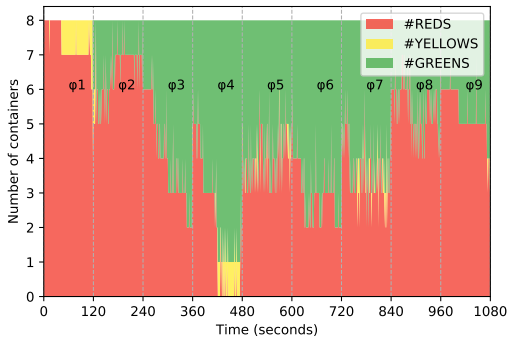


(c) `Soft` limites

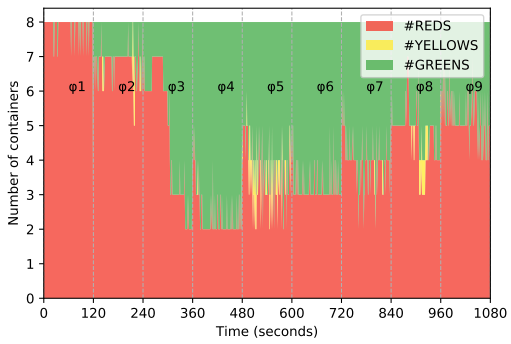


(d) MemOpLight

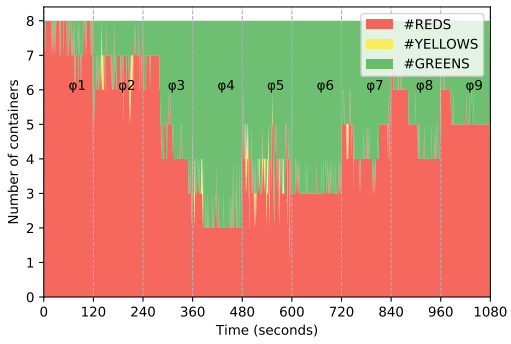
FIGURE 26 – Débit moyen des huit conteneurs pour chaque phase du scénario avec différents mécanismes



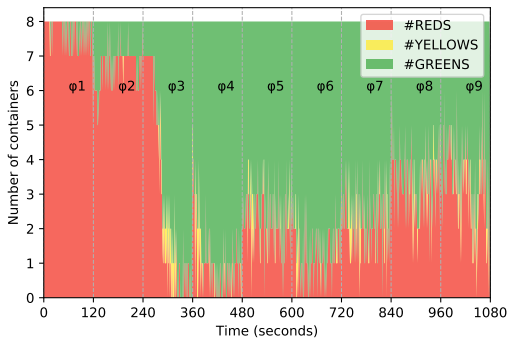
(a) Aucune limite fixée



(b) `Max` limites



(c) `Soft` limites



(d) MemOpLight

FIGURE 27 – Couleur des conteneurs pendant la cinquième exécution de notre expérience pour différents mécanismes

Analyse détaillée des résultats

Dans les paragraphes à venir, nous détaillons les résultats de notre expérience selon les différentes périodes décrites précédemment.

SYSTÈME CHARGÉ Dans φ_1 et φ_2, le système est surchargé puisque presque tous les conteneurs sont soumis à des charges fortes ou intermédiaires. Concernant la satisfaction, montrée dans la Figure 27, celle-ci est mauvaise et ceux pour n'importe quel mécanisme. En effet, le système est surchargé, cette situation ne permet pas des performances optimales et les mécanismes sont donc contraints de faire au mieux avec les ressources disponibles.

En termes de performance, le débit global est minimal sans avoir fixé de limite, 836453 transactions (t.) comparé à 895317.8 t. et 898728 t. pour les `max` et `soft` limites. Dans ces phases, les conteneurs se battent pour la mémoire et elle est donc mal utilisée. Concernant les `max` et `soft` limites, ces mécanismes assurent à chaque conteneur une empreinte mémoire minimale. Ainsi, leurs performances sont supérieures à celles obtenues sans fixer de limite.

MemOpLight permet, pour ces mêmes phases, d'améliorer les performances. En effet, il atteint 927651 t. et 937413 t. dans φ_1 et φ_2. Avec notre mécanisme, quand les conteneurs ont atteint un équilibre mémoire, ils arrêtent de se battre pour elle. Même les conteneurs qui ont moins de mémoire présentent des performances accrues car ils font un meilleur usage de cette ressource. La quantité de mémoire n'est donc pas la panacée. En effet, il vaut mieux avoir moins de mémoire, mais que son contenu soit stable. Ceci évite donc aux données de faire sans cesse des aller-retours entre le disque et la mémoire, ce qui a un impact désastreux sur les performances.

TRANSITION D'UN SYSTÈME CHARGÉ VERS UN SYSTÈME PEU CHARGÉ φ_3 est une transition entre les phases hautement et faiblement chargées. Cette phase permet donc de vérifier si les mécanismes sont capables de s'adapter au changement. De nouveau, la `max` et la `soft` limites permettent d'obtenir de meilleures performances que sans fixer de limite : 779570 t. et 762521 t. contre 729834 transactions. Comme montré dans la Figure 26d, MemOpLight accroît le débit global jusqu'à 788909 t. grâce à son caractère dynamique. Outre l'amélioration des performances, MemOpLight permet de respecter la SLA plus rapidement, comme montré dans la Figure 27d. Cette même figure montre aussi qu'une fois l'équilibre atteint la SLA est toujours respectée.

SYSTÈME PEU CHARGÉ Dans φ_4, les conteneurs continuent de recevoir de faibles charges de travail. Dans cette phase, les `max` et `soft` limites étant toujours actives, elles bloquent la consolidation mémoire. Celles-ci entravent donc les performances des conteneurs, 583156 t. et 580914 t. tandis qu'ils répondent à 714737 t. quand aucune limite n'est fixée. Dans ce cas, MemOpLight offre des performances similaires, 723084 t. *vs.* 714737 t., à la solution sans limite. Avec ce dernier, tous

TABLE 9 – Débit total des conteneurs (en millions de requêtes, pour les différents mécanismes, moyennés sur 10 exécutions)

Mécanisme \ Conteneur	A	B	C	D	E	F	G	H	Total
Aucune limite fixée	15.2	14.5	17.3	11.9	18.7	15.9	**8.8**	**10.3**	112.6
`Max` limite	13.6	13.8	13.8	8.5	14.9	9.9	4.6	6.8	85.9
`Soft` limite	17.1	**17.4**	17.6	12.6	17.5	13.4	7.3	9.5	112.4
MemOpLight	**17.5**	17.3	**18.4**	**13.7**	**20.2**	**16.8**	8.6	10.1	**122.6**

TABLE 10 – Part du temps (en pourcentage) où les conteneurs sont satisfaits (soit vert soit jaune) pour chaque mécanisme (moyenné sur 10 exécutions)

Mécanisme \ Conteneur	A	B	C	D	E	F	G	H	Total
Aucune limite fixée	42	23	8	35	69	39	62	**71**	44
`Max` limite	**60**	35	5	37	73	15	53	35	39
`Soft` limite	56	42	11	35	64	16	53	32	39
MemOpLight	58	**44**	**42**	**52**	**76**	**53**	**67**	67	**57**

les conteneurs sont satisfaits puisqu'il n'y a que des conteneurs verts comme le montre la phase φ_4 de la Figure 27d. Il permet aussi à ces derniers d'avoir des performances dépassant leurs satisfactions puisque les zones hachurées sont plus étendues, pour cette phase, dans la Figure 26d. *A contrario*, les mécanismes de Linux n'arrivent pas à satisfaire tous les conteneurs alors que le système est peu chargé. Pis, sans fixer de limite, les conteneurs respectent leurs SLA.

SYSTÈME MODÉRÉMENT CHARGÉ De φ_5 à φ_9, le système est modérément chargé, mais les charges reçues par les conteneurs s'intensifient au cours du temps. La Figure 27d montre que MemOpLight maximise la satisfaction des conteneurs. De plus, avec celui-ci, les conteneurs atteignent plus rapidement des performances satisfaisantes grâce au retour dynamique des contrôleurs applicatifs. Pendant ces phases, les pics de rouge peuvent s'expliquer par le fait que le *benchmark* offre une charge moyenne et non constante.

Conclusion

Cette expérience consolide notre résultat précédent comme le montrent le Tableau 9 et le Tableau 10.

Le Tableau 9[13] montre le débit pour chaque conteneur pour tous les mécanismes que nous avons testés. D'aucuns peuvent voir que MemOpLight permet d'obtenir de meilleures performances pour cinq des huit conteneurs. Dans les cas contraires, les performances sont inférieures d'environ seulement 0.6%, 2.3% et 1.9% pour les conteneurs B, G et H comparés à l'expérience sans limite fixée. D'une manière générale, avec MemOpLight, les conteneurs répondent à 122.6 millions de transactions ce qui représente une augmentation de 8.9% comparé au mécanisme sans limite.

[13] *Une transaction est composée de plusieurs requêtes.*

Le Tableau 10 montre la part du temps où les conteneurs sont satisfaits. Ce temps correspond à la somme du temps passé à être vert ou jaune. MemOpLight maximise le nombre de conteneurs satisfaits. Il augmente aussi le temps total passé à être satisfait au cours de l'expérience entière de 44% à 57%. Ainsi, MemOpLight permet un meilleur usage des ressources disponibles puisqu'il accroît les performances des conteneurs avec la même machine sous-jacente.

6.4 ÉTUDE DES PARAMÈTRES DE MEMOPLIGHT

MemOpLight, le mécanisme étudié dans les sections précédentes, repose sur plusieurs paramètres pouvant avoir un impact sur ses performances. Ses paramètres sont au nombre de deux :

1. Le premier consiste à faire varier le pourcentage de mémoire réclamée à chaque période. Jusqu'ici, nous avions fixé ce pourcentage à 2% de l'empreinte mémoire actuelle du conteneur.
2. Le second revient à modifier la période à laquelle MemOpLight s'active. Dans les expériences décrites précédemment, notre mécanisme s'activait au plus une fois par seconde.

Lors de l'implémentation de MemOpLight, nous avons fait le choix de réclamer la mémoire des conteneurs jaunes. Ce choix peut être discuté.

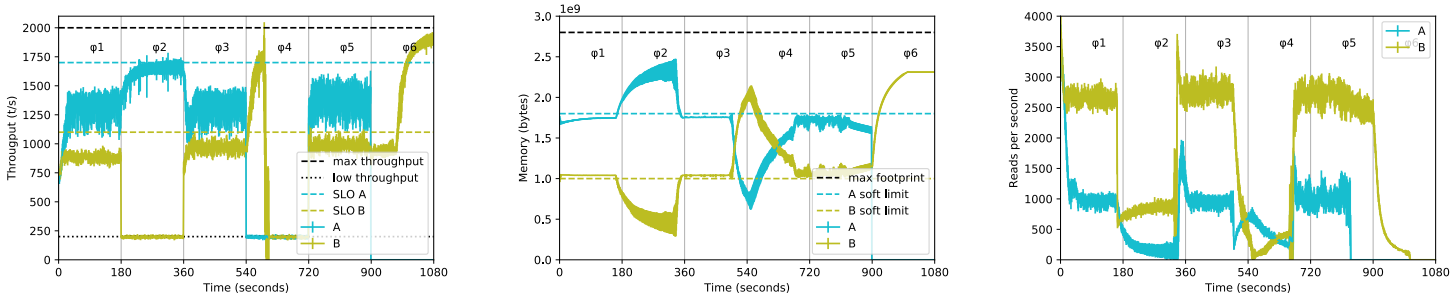
C'est pourquoi, dans les sous-sections qui suivent, nous étudions l'impact de ces paramètres sur les performances de MemOpLight. Nous procédons enfin à l'étude du choix d'implémentation décrit plus haut.

6.4.1 *Pourcentage de mémoire réclamé*

Nous commençons par faire varier le pourcentage de mémoire réclamée tout en fixant la période de réclamation à 1 s. Pour ce faire, nous exécutons le même scénario que celui de l'expérience avec deux conteneurs, décrit dans la Section 4.5. Les conditions de l'expérience sont similaires à celle décrite dans la Section 4.3, à savoir l'exécution sur une machine de Grid'5000. Les valeurs suivantes sont données au pourcentage : 1%, 2%, 5% et 10%.

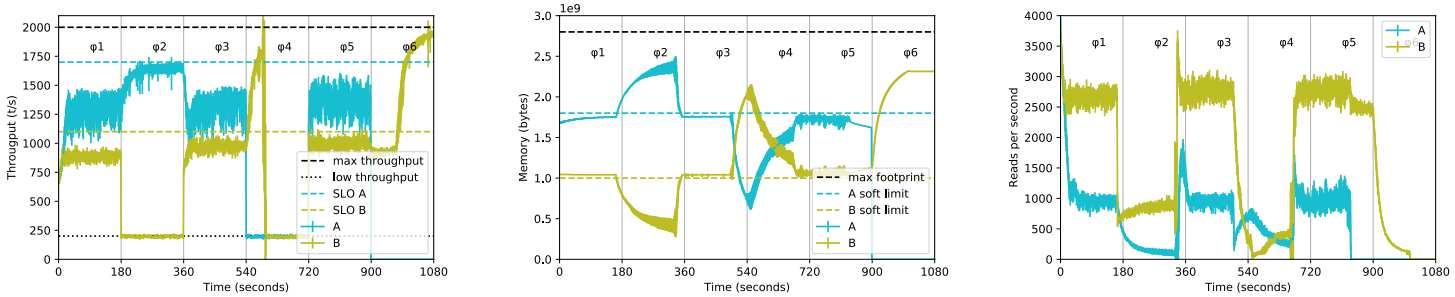
Les résultats obtenus avec un pourcentage fixé à 1% sont décrits dans la Figure 28. La Figure 29 rappelle les résultats obtenus pour un pourcentage de 2%. Les performances résultant d'une réclamation de 5% de l'empreinte mémoire sont montrées dans la Figure 30. Enfin, la Figure 31 dépeint les résultats pour une fraction égale à 10%. Toutes ces figures respectent le code couleur déjà présenté dans la Section 4.4.

Dans ces résultats, nous nous intéressons particulièrement à $\varphi_2(h, l)$, puisque c'est dans cette phase que MemOpLight a montré ses performances accrues. Tout d'abord, le débit est quasi identique dans toutes les figures. Mais celui-ci est plus stable avec 1% de mémoire réclamée qu'avec 10%, comme le montre la comparaison de la Figure 28 avec la Figure 31. Du point de vue du débit, il semble donc préférable de



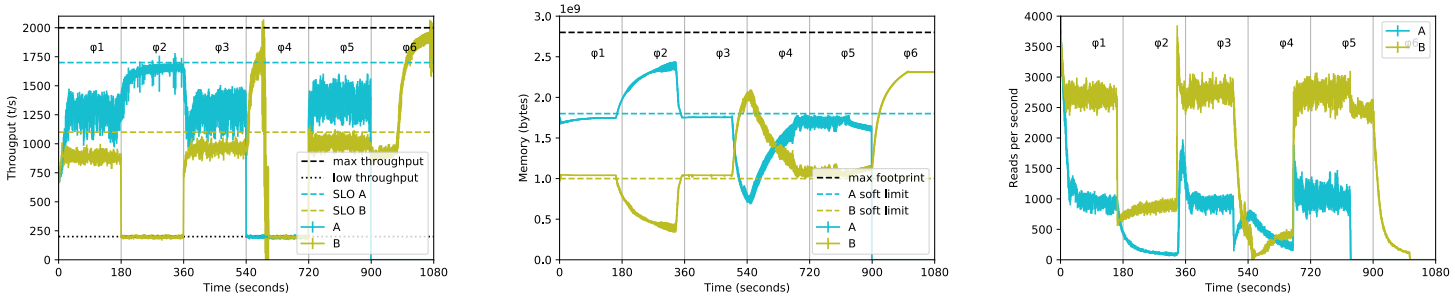
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 28 – MemOpLight récupérant 1% de l'empreinte mémoire chaque seconde



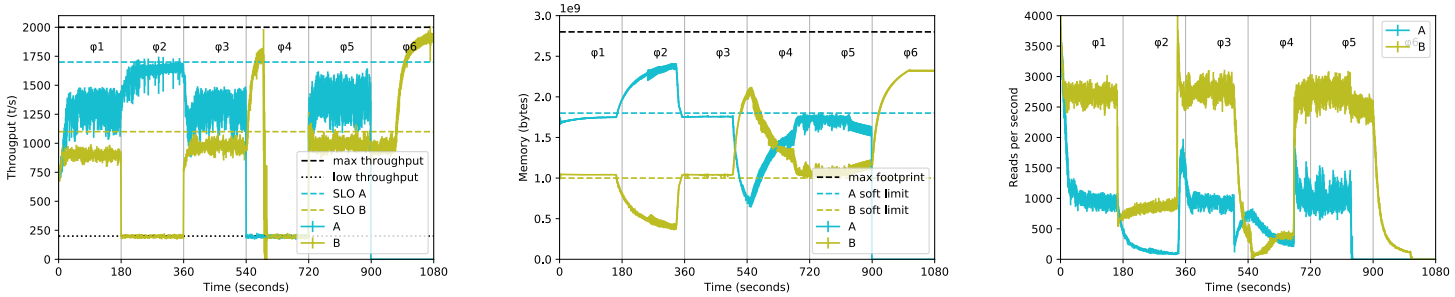
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 29 – MemOpLight récupérant 2% de l'empreinte mémoire chaque seconde



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 30 – MemOpLight récupérant 5% de l'empreinte mémoire chaque seconde



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 31 – MemOpLight récupérant 10% de l'empreinte mémoire chaque seconde

TABLE 11 – Valeur médiane du débit mesuré dans chaque phase pour les différents pourcentages de réclamation (en t/s arrondi à l'entier le plus proche)

Phases / Transactions	$\varphi_1(h,h)$		$\varphi_2(h,l)$		$\varphi_3(h,i)$		$\varphi_4(l,l)$		$\varphi_5(i,h)$		$\varphi_6(s,h)$
Conteneur	A	B	A	B	A	B	A	B	A	B	B
Charge reçue	2000	2000	2000	200	2000	1500	200	200	1500	2000	2000
1%	**1295**	881	**1645**	197	**1305**	965	196	200	1364	985	1481
2%	1280	883	1637	197	1305	971	196	199	1364	997	1471
5%	1272	889	1639	197	1288	962	196	199	1365	**1007**	1530
10%	1288	**897**	1629	197	1300	**973**	196	198	**1371**	995	**1600**
$\lvert 1\% - 10\% \rvert$	7	16	16	0	5	8	0	2	7	10	119

récupérer dix fois 1% de mémoire qu'une fois 10%. En effet, en prenant trop de mémoire, notre mécanisme brise l'ordre entre les pages actives et inactives du *page cache*.

Pour faciliter la comparaison, et ainsi nous rendre compte du faible impact de ce paramètre, nous avons compilé les résultats obtenus dans le Tableau 11. La dernière ligne contient la valeur absolue de la différence entre les résultats obtenus avec 1% et 10%. Cette différence est, au pire, de 16 transactions[14], soit 1.8% ce qui montre le faible impact de la variation du pourcentage réclamé.

D'aucuns pourraient donc se mettre à douter de la fiabilité de notre mécanisme et cette interrogation est justifiée. Néanmoins, MemOpLight s'appuie, pour réclamer la mémoire, sur des mécanismes existants du noyau Linux. En effet, il invoque la fonction `try_to_free_mem_cgroup_pages` pour réclamer des pages au `cgroup` considéré [149, fichier `mm/vmscan.c`, ligne 3298]. Il est certes possible de renseigner un nombre de pages à réclamer à cette fonction, mais celui-ci ne reste qu'une indication. En réalité, Linux va, pour récupérer de la mémoire, agir sur le *page cache*. Le mécanisme de réclamation du *page cache* étant très complexe il est donc ardu de deviner le nombre de pages effectivement réclamé et surtout si l'indication a été prise en compte.

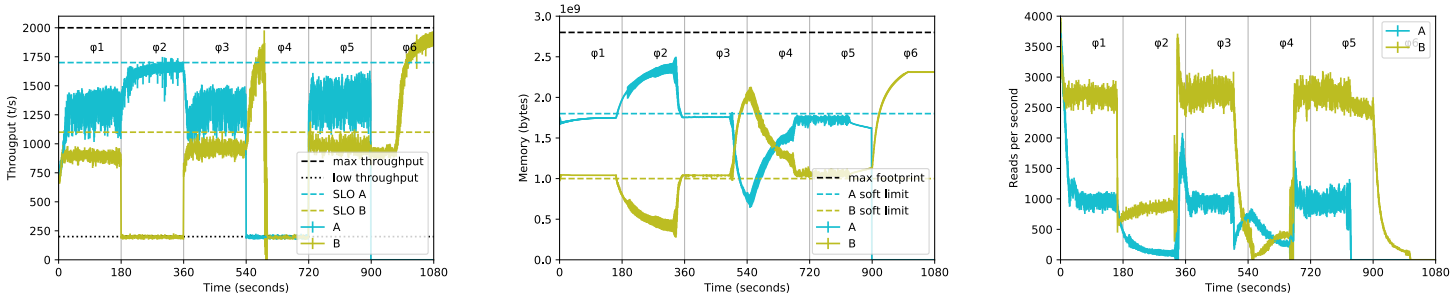
[14] *MemOpLight ne s'active pas dans $\varphi_6(s,h)$ puisqu'il n'y a plus de pression mémoire.*

6.4.2 *Période de réclamation*

Nous nous intéressons maintenant au second paramètre : la période. Tout en reprenant le même scénario et le même environnement d'expérimentation, nous faisons désormais varier la période de réclamation en lui donnant les valeurs suivantes : 1 s, 2 s, 5 s et 10 s. Le pourcentage de réclamation a été fixé à 2%.

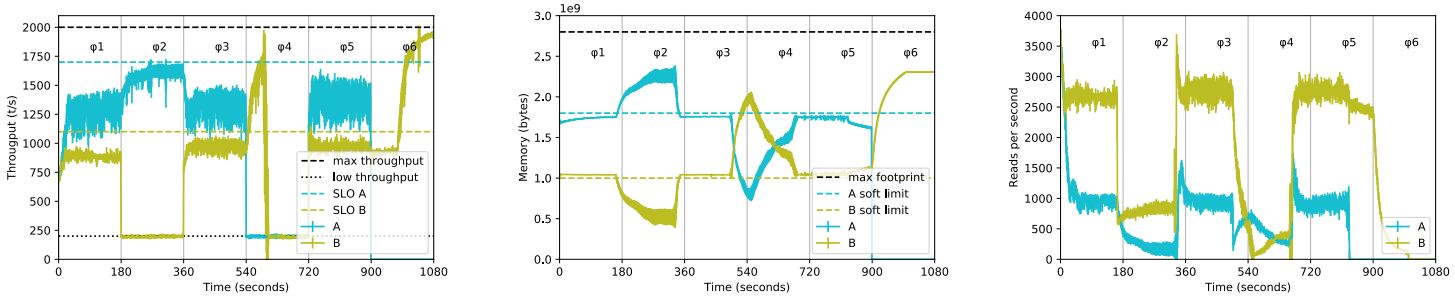
La Figure 32 rappelle les résultats précédemment obtenus. Les performances obtenues avec une période de réclamation de 2 s sont dépeintes dans la Figure 33. La Figure 34 montre les résultats collectés avec une période de réclamation fixée à 5 s. Finalement, les métriques correspondant à une période de 10 s sont rapportées dans la Figure 35.

Nous concentrons notre analyse sur $\varphi_2(h,l)$ pour les mêmes raisons qu'évoquées lors de l'étude du pourcentage de réclamation. D'une



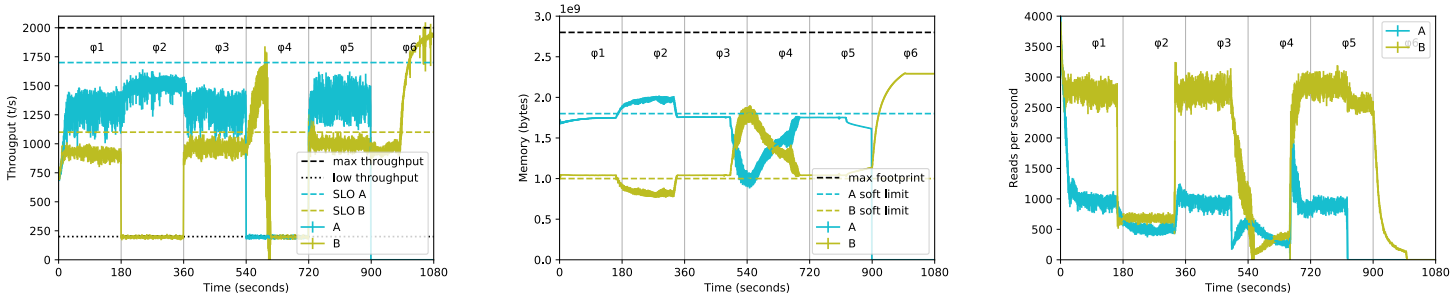
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 32 – MemOpLight récupérant 2% de l'empreinte mémoire chaque seconde



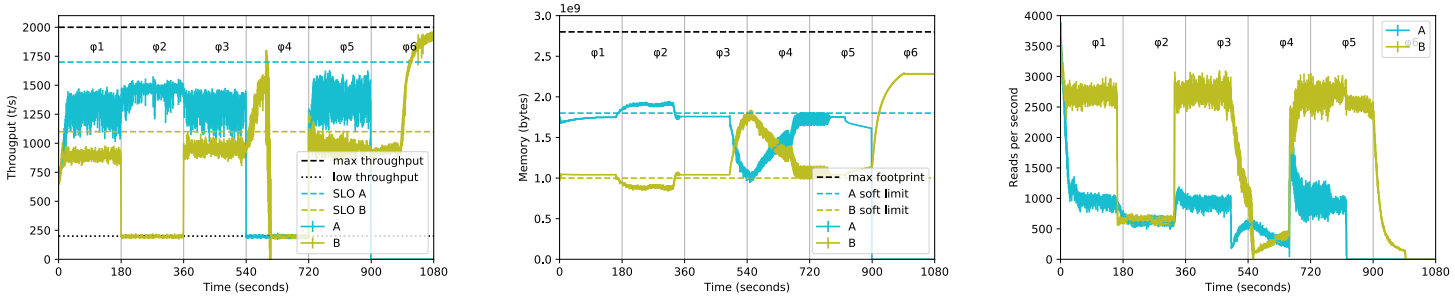
(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 33 – MemOpLight récupérant 2% de l'empreinte mémoire toutes les 2 secondes



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 34 – MemOpLight récupérant 2% de l'empreinte mémoire toutes les 5 secondes



(a) Débit (b) Empreinte mémoire (c) Lecture depuis le disque

FIGURE 35 – MemOpLight récupérant 2% de l'empreinte mémoire toutes les 10 secondes

TABLE 12 – Valeur médiane du débit mesuré dans chaque phase pour les différentes périodes de réclamation (en t/s arrondi à l'entier le plus proche)

Phases / Transactions	$\varphi_1(h,h)$		$\varphi_2(h,l)$		$\varphi_3(h,i)$		$\varphi_4(l,l)$		$\varphi_5(i,h)$		$\varphi_6(s,h)$
Conteneur	A	B	A	B	A	B	A	B	A	B	B
Charge reçue	2000	2000	2000	200	2000	1500	200	200	1500	2000	2000
1 s	**1299**	894	**1638**	**197**	1295	961	196	198	1362	982	**1476**
2 s	1269	884	1596	197	**1309**	**967**	196	198	1367	973	1434
5 s	1291	**914**	1498	196	1304	967	196	200	**1370**	**999**	1300
10 s	1286	894	1456	196	1303	953	196	**200**	1367	969	1115
$\lvert 1\,s - 10\,s \rvert$	13	0	182	1	8	8	0	2	5	13	361

manière générale, plus la période est longue plus son impact sur les performances est fort. En effet, avec 1 s, et comme le montre la Figure 32, le conteneur A atteint son SLO. Tandis qu'avec une période de 10 s, dépeint dans la Figure 35, ce conteneur ne dépasse pas les 1500 t/s.

Cette différence de performance s'explique en regardant les figures montrant l'empreinte mémoire des conteneurs. En effet, puisque notre mécanisme est moins invoqué, et que la fraction de réclamation est fixe, celui-ci ne permet pas aux conteneurs d'augmenter leurs empreintes mémoires pour faire face aux pics de charge reçus. Ce fait est particulièrement montré dans la Figure 34b et la Figure 35b où notre mécanisme n'arrive pas à récupérer la mémoire de B. Cependant, la différence entre les empreintes mémoires des figures 32b et 33b est moindre. Une période de 2 s permet donc d'atteindre un équilibre.

Comme pour le pourcentage de réclamation, les chiffres médians pour chaque phase et chaque conteneur sont synthétisés dans le Tableau 12.

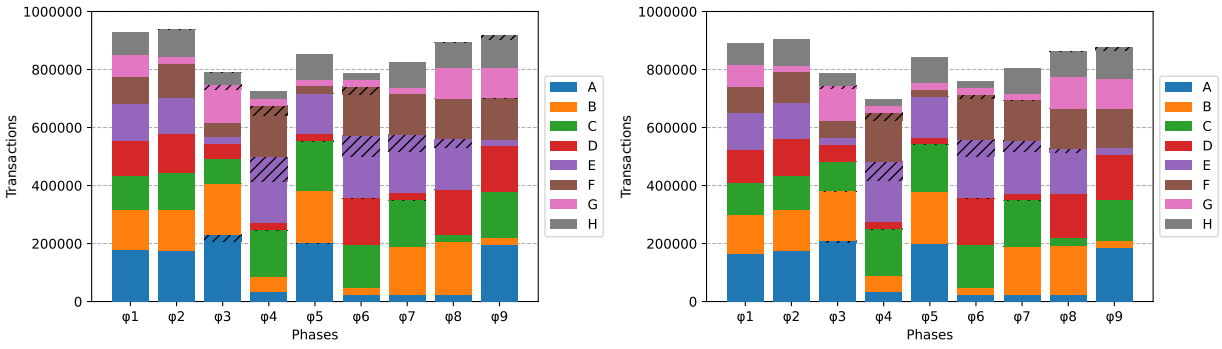
La différence est la plus importante dans $\varphi_2(h,l)$, où notre mécanisme apporte la consolidation. En effet, les transactions médianes chutent de 1638 à 1456 soit une diminution de 182 transactions ou 10.6%.

6.4.3 *Pas de réclamation des conteneurs jaunes*

Les paramètres que nous avons étudiés précédemment sont des paramètres numériques.

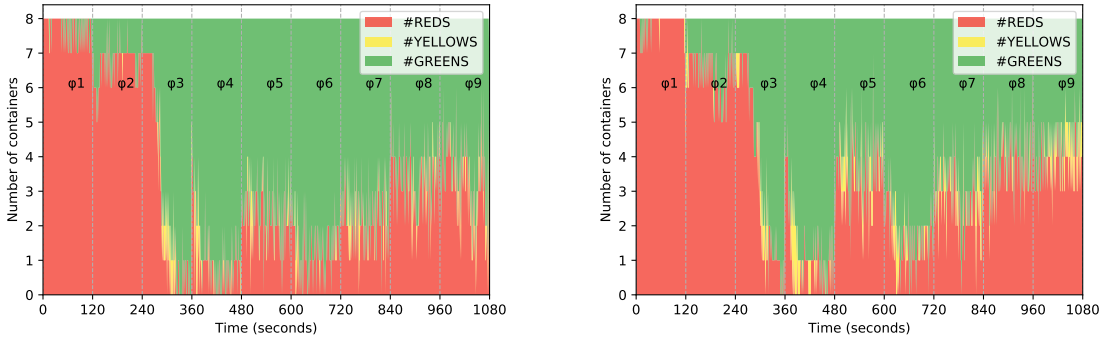
L'implémentation actuelle de MemOpLight, décrite dans l'Algorithme 1, réalise un arbitrage entre les conteneurs, notamment en réclamant la mémoire des conteneurs jaunes. Ce choix d'implémentation vise à donner plus de mémoire aux conteneurs rouges pour diminuer leur nombre. Nous avons donc modifié ce comportement pour que ceux-ci ne soient pas réclamés. Nous appelons cette variante MemOpLightNoYellowReclaim.

Par cette modification, nous souhaitons mesurer l'impact de l'état intermédiaire «jaune» qui, rappelons-le, correspond au fait que le conteneur a des performances satisfaisantes, mais qu'il pourrait faire mieux. Pour ce faire, nous relançons l'expérience avec huit conte-



(a) MemOpLight (b) MemOpLightNoYellowReclaim

FIGURE 36 – Débit moyen des huit conteneurs pour chaque phase du scénario avec différentes variantes de MemOpLight



(a) MemOpLight (b) MemOpLightNoYellowReclaim

FIGURE 37 – Couleur des conteneurs pendant la cinquième exécution de notre expérience pour chaque variante de MemOpLight

neurs, décrite dans la Section 6.3.1, dans le même environnement d'expérimentation.

La Figure 36 montre le débit moyen des conteneurs, pour chaque phase du scénario. Le temps passé à être satisfait lors de la sixième exécution de l'expérience est dépeint dans la Figure 37. Les Figures 26d et 27d sont rappelées pour faciliter la comparaison.

De prime abord, les résultats de cette expérience concordent avec ceux obtenus précédemment. En effet, les performances sont supérieures à celles qu'offrent les mécanismes existants. Toutefois, MemOpLight est meilleur que MemOpLightNoYellowReclaim comme le montrent particulièrement les phases φ_6, φ_7, φ_8 et φ_9 des figures 36a et 36b. Les zones hachurées, correspondant aux dépassements des SLO, sont aussi moins étendues pour la variante du mécanisme. De plus, les conteneurs sont plus souvent jaunes avec la variante de MemOpLight comme dépeint dans la Figure 37b

Pour faciliter la comparaison, nous avons compilé ces résultats dans trois tableaux :

1. Le premier, Tableau 13, contient le débit total des conteneurs.

TABLE 13 – Débit total des conteneurs (en millions de requêtes, pour chaque variante de MemOpLight, moyennés sur 10 exécutions)

Mécanisme \ Conteneur	A	B	C	D	E	F	G	H	Total
MemOpLight	**17.5**	**17.3**	**18.4**	**13.7**	**20.2**	**16.8**	**8.6**	**10.1**	**122.6**
MemOpLightNoYellowReclaim	16.7	17.0	18.1	13.4	19.0	16.3	8.4	10.0	118.9

TABLE 14 – Part du temps (en pourcentage) où les conteneurs sont jaunes pour chaque variante de MemOpLight (moyenné sur 10 exécutions)

Mécanisme \ Conteneur	A	B	C	D	E	F	G	H	Moyenne
MemOpLight	3.28	2.29	7.88	2.45	0.99	3.36	0.17	1.34	2.72
MemOpLightNoYellowReclaim	3.81	2.66	9.21	2.57	1.29	4.28	0.32	0.44	3.07

2. Le deuxième, Tableau 14, dépeint le pourcentage de temps qu'un conteneur a passé en étant jaune.
3. Le dernier, Tableau 15, montre, pour chaque conteneur, le pourcentage de temps qu'il passe en étant satisfait.

Ces tableaux rappellent aussi les chiffres obtenus pour MemOpLight.

En termes de débit, MemOpLightNoYellowReclaim présente de moins bonnes performances que MemOpLight. Celui-ci offre de meilleures performances à chacun des conteneurs, les performances globales chutent donc de 3% avec MemOpLightNoYellowReclaim.

Le Tableau 14 consolide le résultat de la Figure 37b. En effet, avec MemOpLightNoYellowReclaim, les conteneurs passent plus de temps à être jaunes. Plus précisément, ceux-ci passent, en moyenne, 3.07% du temps de l'expérience à être jaunes contre 2.72% avec MemOpLight.

Quant à la satisfaction globale, ici aussi, MemOpLightNoYellowReclaim montre des performances amoindries comparées à son homologue. En effet, la satisfaction globale passe de 57% à 52%.

Les moindres performances de MemOpLightNoYellowReclaim peuvent s'expliquer par le fait que les conteneurs passent plus de temps à être rouges comme le montre le Tableau 16. En effet, le temps total passé à ne pas être satisfait augmente de 12%.

Contrairement à la période de réclamation, l'important ici n'est pas l'ordre du *page cache* mais sa taille par rapport au WS. Un conteneur vert utilise toute sa mémoire pour son WS. Pour un conteneur jaune, il peut y avoir un effet de seuil. Pour traiter les requêtes en plus

TABLE 15 – Part du temps (en pourcentage) où les conteneurs sont satisfaits (soit vert soit jaune) pour chaque variante de MemOpLight (moyenné sur 10 exécutions)

Mécanisme \ Conteneur	A	B	C	D	E	F	G	H	Moyenne
MemOpLight	**58**	**44**	**42**	**52**	**76**	**53**	**67**	**67**	**57**
MemOpLightNoYellowReclaim	54	38	35	48	75	45	63	59	52

TABLE 16 – Part du temps (en pourcentage) où les conteneurs ne sont pas satisfaits (rouge) pour chaque variante de MemOpLight (moyenné sur 10 exécutions)

Mécanisme \ Conteneur	A	B	C	D	E	F	G	H	Moyenne
MemOpLight	**42**	**56**	**58**	**48**	**24**	**47**	**33**	**33**	**43**
MemOpLightNoYellowReclaim	46	62	65	52	25	55	37	41	48

de son SLO, il faudrait peut-être à un tel conteneur beaucoup plus de mémoire. Les protéger n'apporte donc pas forcément un gain. Pis, puisque les conteneurs jaunes ne sont pas réclamés, la mémoire n'est réclamée qu'aux conteneurs verts qui sont moins nombreux. Les conteneurs rouges ont donc moins de possibilités d'augmenter leurs empreintes mémoires pour accroître leurs performances.

L'état jaune peut sembler anecdotique, mais il prend tout son sens lorsqu'il n'y a que des conteneurs verts et jaunes. Dans cette configuration, la mémoire est récupérée aux conteneurs verts. Les conteneurs jaunes peuvent donc accroître leurs performances ce qui mène, à terme, à un équilibre des performances des conteneurs.

6.5 CONCLUSION

Dans ce chapitre, nous avons comparé MemOpLight aux mécanismes existants fournis par Linux. Notre mécanisme permet une redistribution dynamique de la mémoire en réclamant celle-ci aux conteneurs ayant de bonnes performances. Pour ce faire, il s'appuie sur des contrôleurs applicatifs indiquant si les performances, telles que perçues par l'application conteneurisée, sont bonnes ou non. Les conteneurs ayant de mauvaises performances pourront donc augmenter leurs empreintes mémoires dans l'espoir que celles-ci égalent leurs WS.

La réelle plus-value de MemOpLight est son caractère dynamique. Il s'adapte donc aux variations de la charge de travail d'une application. Il permet, comme les `max` et `soft` limites, l'isolation mémoire, un conteneur ne peut donc affamer ses congénères. Néanmoins, et contrairement à ces limites, son caractère dynamique offre la consolidation mémoire et permet donc aux conteneurs d'accroître leurs performances.

Les expériences faisant varier les paramètres de MemOpLight ont montré que, pour notre expérience, l'impact de la valeur du pourcentage de mémoire réclamée est plutôt faible. Ceci est dû à la complexité du mécanisme noyau invoqué par MemOpLight pour réclamer la mémoire.

La période de réclamation a par contre un impact non négligeable dans notre expérience. Néanmoins, si la charge de travail reste constante pendant une durée plus longue une réclamation ayant lieu moins souvent devrait fournir suffisamment de mémoire aux conteneurs pour leur offrir de bonnes performances.

MemOpLight offre aussi de meilleures performances lorsque les conteneurs jaunes sont réclamés. L'ajout d'états de performances intermédiaires à l'état optimal permet d'affiner la configuration de la répartition mémoire et donc de réclamer plus justement.

7 CONCLUSION

Le *cloud* aura marqué un changement, bienvenu ou délétère, dans l'informatique. Avant lui, la plupart des acteurs de ce domaine possédaient leurs propres centres de données. Dorénavant, l'infrastructure a été abstraite et n'est possédée que par quelques géants du numérique menant ainsi à une situation d'oligopole. Pour rendre possible son abstraction, le *cloud* s'est appuyé sur la virtualisation. Si, historiquement, les machines virtuelles sont les porte-étendards du concept du même nom, les conteneurs sont devenus, petit à petit, une alternative viable.

La virtualisation permet d'isoler les services de différents clients. Ainsi, le conteneur d'un client donné ne peut pas s'accaparer toutes les ressources au détriment des autres. Cette isolation permet donc de garantir la satisfaction liée aux SLA signées par le client et le fournisseur. Néanmoins, un fournisseur de *cloud* souhaite consolider les ressources de son système afin d'en faire un meilleur usage et de pouvoir accueillir plus de clients.

Linux propose des mécanismes garantissant l'isolation, mais, comme nous l'avons montré dans cette thèse, ils ne sont pas suffisants pour garantir la consolidation. Pour mettre en exergue ce problème, nous avons développé une plateforme expérimentale reproduisant le comportement de conteneurs exécutant des bases de données utilisées par des sites de commerce en ligne. Nos conteneurs étaient soumis à des charges de travail dynamiques représentant ainsi les différentes périodes d'activité d'une journée. Les mécanismes que propose Linux pour contrôler l'empreinte mémoire des conteneurs ont montré leurs limites. En effet, ils ne peuvent s'adapter à des charges de travail dynamiques à cause de leur caractère statique. Ce problème prend tout son sens dans le *cloud* où il mène à un surdimensionnement de la mémoire des clients et à une utilisation non optimale de celle-ci pour le fournisseur.

Les mécanismes de Linux sont certes souples, puisqu'il est permis, dans certaines conditions, à un conteneur de dépasser la valeur d'une limite mémoire. Mais ils ne peuvent s'adapter aux variations dans la charge de travail des conteneurs. De plus, le système d'exploitation a certes une vision globale de l'utilisation des ressources, mais les métriques dont il dispose sont déconnectées des performances des applications. En s'inspirant de l'état de l'art, notamment de la boucle autonomique, nous avons conclu qu'une double vue pouvait être la solution à ce problème. D'un côté, l'espace utilisateur permet la collecte de métriques proches des applications et donc représentatives de leurs performances. De l'autre, l'espace noyau offre une vision globale du système permettant de savoir si celui-ci est en pression mémoire.

Ainsi, nous avons proposé MemOpLight. Ce mécanisme s'appuie sur les informations haut niveaux offertes par l'espace utilisateur pour réclamer au mieux la mémoire en tant que mécanisme noyau. Cette réclamation cible prioritairement les conteneurs ayant de bonnes performances dans l'espoir de trouver un équilibre d'utilisation de la mémoire où le maximum de conteneurs est satisfait.

Nous avons éprouvé notre solution en la comparant aux mécanismes existants dans une expérience calquant le comportement d'un site de commerce en ligne. MemOpLight a ainsi montré de meilleures performances que les mécanismes offerts par Linux. Contrairement à ceux-ci, notre solution est dynamique et s'adapte donc aux besoins des applications. Nous avons ensuite consolidé nos résultats en lançant une expérience de plus grande envergure Ainsi, MemOpLight offre des performances supérieures de 8.9% tout en augmentant la satisfaction de 13%, ce grâce à un meilleur usage de la mémoire disponible.

Nous verrons, dans la suite de cet ultime chapitre, tout d'abord les améliorations pouvant être apportées à MemOpLight pour pallier à ses faiblesses ou augmenter ses performances. Puis, nous terminerons par une ouverture sur les horizons futurs brossés par cette thèse.

7.1 AMÉLIORATIONS DE MEMOPLIGHT

MemOpLight apporte la consolidation aux conteneurs tout en assurant une isolation entre ceux-ci. Toutefois, notre mécanisme peut être amélioré. Nous pouvons, premièrement, lancer des expériences plus conséquentes pour éprouver celui-ci. Deuxièmement, il nous faudrait traiter particulièrement le cas d'un cycle de réclamation. Enfin, le contrôleur applicatif est la seule partie qu'un utilisateur doit écrire pour utiliser MemOpLight, il faut donc faciliter au possible cette écriture.

Pour mettre à l'épreuve MemOpLight dans des expériences plus conséquentes, plusieurs solutions sont possibles. Tout d'abord, nous pouvons reprendre notre expérience mais en allouant plus de ressources aux conteneurs et en augmentant, encore, le nombre de conteneurs. Il est aussi possible de changer le *benchmark* exécuté par les différents conteneurs voir de mixer les *benchmarks* utilisés. Notre solution peut aussi être testée avec une suite logicielle utilisée dans l'industrie. Nous avons commencé à réaliser une telle expérience où des conteneurs sollicitent, via `gatling`, un site web hébergeant `woocommerce`, une extension transformant `wordpress` en site de commerce en ligne, et s'appuyant donc sur `apache` et `mysql` [62, 156, 157, 20, 120, 60]. Néanmoins, pour que cette expérience soit représentative de la réalité il nous faudrait posséder une copie d'un fichier `access.log` utilisé en production. En effet, le fichier dont nous disposons correspond à un site de commerce en ligne iranien, mais le nombre de requêtes est limité [162]. Nous pourrions extrapoler celui-ci, mais nous introduirions alors des biais statistiques qui fausseraient la réalité. D'autres traces sont disponibles, comme celles de différents fournisseurs de *cloud* ou du site web de la coupe du monde de football 98, mais

elles ne sont pas exemptes de défauts [155, 45, 3, 158]. En effet, les premières sont trop abstraites puisqu'elles ne contiennent que des informations temporelles sur l'utilisation de ressources [155, 45, 3]. Quant à la dernière, elle date de 1998 et n'est donc plus représentative du web d'aujourd'hui [158].

Une autre amélioration aurait pour but de prévenir le cas pathologique d'un cycle de réclamation où MemOpLight empêche un conteneur d'avoir des performances stables. Nous pensons donc ajouter une connaissance à celui-ci. Cette connaissance, au sens *Knowledge* de la boucle MAPE-K, pourrait prendre la forme d'un historique d'états de satisfaction pour chaque conteneur. Outre la connaissance, un cycle d'hystérésis forçant l'espacement des réclamations mémoire successives visant un conteneur devrait briser ce cercle vicieux.

Enfin, et pour faciliter l'utilisation de MemOpLight dans l'industrie, la formalisation des contrôleurs nous semble importante. En effet, le contrôleur applicatif est la seule partie qu'un utilisateur doit écrire pour pouvoir se servir de notre mécanisme. Nous pensons donc que le développement d'un cadriciel de contrôleurs applicatifs faciliterait l'écriture de ces derniers et démocratiserait l'utilisation de notre solution.

7.2 HORIZONS FUTURS

Dans cette thèse, nous nous sommes concentrés sur la mémoire présente dans les barrettes mémoires. Néanmoins, les caches L3 des processeurs sont désormais de tailles conséquentes. En effet, des processeurs ayant un cache L3 de 256 MB sont disponibles à l'achat [19]. Nous pourrions donc utiliser MemOpLight pour partager ce niveau de cache entre les différentes applications s'exécutant sur ce processeur. En pratique, une application n'étant pas satisfaite se verrait allouer une plus grande part du cache qu'une application satisfaite.

À un autre niveau, les conteneurs ne prennent pas en compte, à l'écriture de ces lignes, la topologie mémoire. Par exemple, il n'est pas possible d'allouer à un conteneur X GB sur un nœud A et Y GB sur un nœud B. MemOpLight pourrait donc être utilisé pour effectuer un arbitrage de la mémoire selon les différents nœuds NUMA du système.

La mémoire est entourée d'autres ressources et elle n'est pas forcément limitante. La bande passante disque est intimement liée à la mémoire puisque la seconde est utilisée comme un cache lorsque des données sont lues depuis le disque. Ainsi, si la mémoire n'est pas suffisamment grande pour accueillir ces données, la bande passante disque sera saturée. Contrairement à la bande passante disque, le CPU et la bande passante réseau ne sont pas forcément liés à la mémoire. Néanmoins, si la bande passante réseau est limitée, la charge reçue sera faible. MemOpLight indiquera donc de bonnes performances alors qu'il n'en est en fait rien.

Ce problème se résume donc à de l'optimisation multicritère qui est certes compliqué à résoudre mais qui, de l'autre côté, ne doit être

négligé. Dans le *cloud*, les clients payent pour obtenir des ressources, trouver un équilibre entre celles-ci est donc important. En effet, il est inutile d'avoir une trop grande quantité d'une ressource donnée si une autre fait défaut et mine les performances.

BIBLIOGRAPHIE

[1] Alexey Kopytov. *sysbench*.

[2] Alibaba. *Alibaba Cloud*. 2009. URL : https://us.alibabacloud.com/.

[3] Alibaba. *Alibaba Cluster Trace Program*. 2017. URL : https://github.com/alibaba/clusterdata.

[4] Alibaba. *Alibaba Container Service*. URL : https://www.alibabacloud.com/product/container-service?spm=a2c5t.10695662.1996646101.searchclickresult.55a3212bnXyv1v.

[5] Graham Allan. *DDR4 Bank Groups in Embedded Applications*. URL : https://www.synopsys.com/designware-ip/technical-bulletin/ddr4-bank-groups.html.

[6] Amazon. *Amazon Compute Service Level Agreement*. URL : https://aws.amazon.com/compute/sla/?nc1=h_ls.

[7] Amazon. *Amazon DynamoDB*. URL : https://aws.amazon.com/fr/dynamodb/.

[8] Amazon. *Amazon EC2 Container Service is Now Generally Available*. Avr. 2015. URL : https://aws.amazon.com/fr/about-aws/whats-new/2015/04/amazon-ec2-container-service-is-now-generally-available/.

[9] Amazon. *Amazon EC2 Exits Beta and Now Offers a Service Level Agreement*. Oct. 2008. URL : https://aws.amazon.com/fr/about-aws/whats-new/2008/10/23/amazon-ec2-exits-beta-and-now-offers-a-service-level-agreement/.

[10] Amazon. *Amazon EC2 Pricing*. Rapp. tech. URL : https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls.

[11] Amazon. *Amazon EC2 Spot Instances*. URL : https://aws.amazon.com/ec2/spot/?nc1=h_ls.

[12] Amazon. *Amazon Elastic Compute Cloud*. 2008. URL : https://aws.amazon.com/ec2/?nc1=h_ls.

[13] Amazon. *Amazon Elastic Container Service*. URL : https://aws.amazon.com/ecs/?nc1=h_ls.

[14] Amazon. *AWS Auto Scaling Guide de l'utilisateur*. Rapp. tech. Nov. 2018, p. 23. URL : https://docs.aws.amazon.com/fr_fr/autoscaling/plans/userguide/as-plans-ug.pdf#auto-scaling-getting-started.

[15] Amazon. *Qu'est-ce que le cloud computing ?* URL : https://aws.amazon.com/fr/what-is-cloud-computing/.

[16] Amazon. *Success-stories de clients AWS*. URL : https://aws.amazon.com/fr/solutions/case-studies/?customer-references-cards.sort-by=item.additionalFields.publishedDate&customer-references-cards.sort-order=desc.

[17] AMAZON. *Tableau des régions*. Avr. 2020. URL : https://aws.amazon.com/fr/about-aws/global-infrastructure/regional-product-services/.

[18] AMAZON. *Types d'instances Amazon EC2*. URL : https://aws.amazon.com/fr/ec2/instance-types/.

[19] AMD. *AMD EPYC™ 7702P*. Août 2019. URL : https://www.amd.com/en/products/cpu/amd-epyc-7702p.

[20] *Apache HTTP Server Project*. URL : https://httpd.apache.org/.

[21] ARM. *GENERAL : HARVARD VS VON NEUMANN*. Jan. 2009. URL : https://www.keil.com/support/docs/3445.htm.

[22] Daniel BALOUEK et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In : *Cloud Computing and Services Science*. Sous la dir. d'Ivan I. IVANOV et al. T. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, p. 3-20. ISBN : 978-3-319-04518-4. DOI : 10.1007/978-3-319-04519-1_1.

[23] Jeff BARR. *New AWS Auto Scaling – Unified Scaling For Your Cloud Applications*. Jan. 2018. URL : https://aws.amazon.com/fr/blogs/aws/aws-auto-scaling-unified-scaling-for-your-cloud-applications/.

[24] Luiz André BARROSO, Jimmy CLIDARAS et Urs HÖLZLE. "The Datacenter as a Computer : An Introduction to the Design of Warehouse-Scale Machines, Second edition". en. In : *Synthesis Lectures on Computer Architecture* 8.3 (juil. 2013), p. 1-154. ISSN : 1935-3235, 1935-3243. DOI : 10.2200/S00516ED2V01Y201306CAC024. URL : http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024 (visité le 29/06/2018).

[25] Luiz André BARROSO et Urs HÖLZLE. "The Case for Energy-Proportional Computing". In : *Computer* 40.12 (déc. 2007), p. 33-37. ISSN : 0018-9162. DOI : 10.1109/MC.2007.443. URL : http://ieeexplore.ieee.org/document/4404806/ (visité le 03/04/2018).

[26] Mihaly BEREKMERI et al. "Feedback Autonomic Provisioning for Guaranteeing Performance in MapReduce Systems". In : *IEEE Transactions on Cloud Computing* 6.4 (oct. 2018), p. 1004-1016. ISSN : 2168-7161, 2372-0018. DOI : 10.1109/TCC.2016.2550047. URL : https://ieeexplore.ieee.org/document/7446289/ (visité le 27/02/2019).

[27] Betsy BEYER et al. *Site reliability engineering : how Google runs production systems*. English. OCLC : 930683030. 2016. ISBN : 978-1-4919-2912-4.

[28] Jalil BOUKHOBZA et Pierre OLIVIER. *Flash memory integration : performance and energy considerations*. Energy management in embedded systems set. OCLC : ocn960894929. London, UK : Kidlington, Oxford : ISTE Press ; Elsevier, 2017. ISBN : 978-1-78548-124-6.

[29] Daniel P. Bovet et Marco Cesati. *Understanding the Linux kernel : from I/O ports to process management*. eng. 3. ed., covers version 2.6. OCLC : 255008440. Beijing : O'Reilly, 2006. ISBN : 978-0-596-00565-8.

[30] George E. P. Box et al. *Time series analysis : forecasting and control*. Fifth edition. Wiley series in probabilit and statistics. Hoboken, New Jersey : John Wiley & Sons, Inc, 2016. ISBN : 978-1-118-67491-8 978-1-118-67492-5.

[31] Edouard Bugnion, Jason Nieh et Dan Tsafrir. "Hardware and Software Support for Virtualization". en. In : *Synthesis Lectures on Computer Architecture* 12.1 (fév. 2017), p. 1-206. ISSN : 1935-3235, 1935-3243. DOI : 10.2200/S00754ED1V01Y201701CAC038. URL : http://www.morganclaypool.com/doi/10.2200/S00754ED1V01Y201701CAC038 (visité le 15/04/2020).

[32] Luiz Capitulino. *Automatic ballooning*. Oct. 2013. URL : https://www.linux-kvm.org/images/5/58/Kvm-forum-2013-automatic-ballooning.pdf.

[33] Damien Carver, Julien Sopena et Sebastien Monnet. "ACDC : Advanced consolidation for dynamic containers". In : *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA : IEEE, oct. 2017, p. 1-8. ISBN : 978-1-5386-1465-5. DOI : 10.1109/NCA.2017.8171363. URL : http://ieeexplore.ieee.org/document/8171363/ (visité le 22/05/2019).

[34] Damien Carver, Julien Sopena et Sebastien Monnet. "ACDC : Advanced consolidation for dynamic containers". In : *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA : IEEE, oct. 2017, p. 1-8. ISBN : 978-1-5386-1465-5. DOI : 10.1109/NCA.2017.8171363. URL : http://ieeexplore.ieee.org/document/8171363/ (visité le 22/05/2019).

[35] CloudCheckr. *Revealed : The 7 Hidden Costs Every Public Cloud User Needs to Avoid*. Mai 2017. URL : https://cloudcheckr.com/cloud-cost-management/revealed-7-hidden-costs-every-public-cloud-user-needs-avoid/.

[36] *Code de la route*. Mar. 2001. URL : https://www.legifrance.gouv.fr/affichCodeArticle.do;jsessionid=8D4439635D4B08FEB2B955D9E3854E70.tplgfr42s_1?idArticle=LEGIARTI000006842150&cidTexte=LEGITEXT000006074228&dateTexte=20170301.

[37] Wikimedia Commons. *File :Von Neumann Architecture.svg — Wikimedia Commons, the free media repository*. [Online ; accessed 29-May-2020]. 2020. URL : https://commons.wikimedia.org/w/index.php?title=File:Von_Neumann_Architecture.svg&oldid=420303556.

[38] containerd. *containerd*. URL : https://github.com/containerd/containerd.

[39] *Control Groups*. URL : https://www.static.linuxfound.org/jp_uploads/seminar20081119/CgroupMemcgMaster.pdf.

[40] Jonathan CORBET. *Another container implementation*. Sept. 2006. URL : https://lwn.net/Articles/200073/.

[41] Jonathan CORBET. *Fixing control groups*. Fév. 2012. URL : https://lwn.net/Articles/484251/.

[42] Jonathan CORBET. *Integrating memory control groups*. Mai 2011. URL : https://lwn.net/Articles/443241/.

[43] Jonathan CORBET. *Notes from a container*. Oct. 2007. URL : https://lwn.net/Articles/256389/.

[44] Jonathan CORBET. *Process containers*. Mai 2007. URL : https://lwn.net/Articles/236038/.

[45] Eli CORTEZ et al. "Resource Central : Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". en. In : *Proceedings of the 26th Symposium on Operating Systems Principles*. Shanghai China : ACM, oct. 2017, p. 153-167. ISBN : 978-1-4503-5085-3. DOI : 10.1145/3132747.3132772. URL : https://dl.acm.org/doi/10.1145/3132747.3132772 (visité le 25/08/2020).

[46] *CPUSETS*. URL : https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cpusets.html.

[47] Michael CROSBY. *What is containerd ?* Août 2017. URL : https://blog.docker.com/2017/08/what-is-containerd-runtime/.

[48] Luca DE CICCO, Saverio MASCOLO et Dario CALAMITA. "A resource allocation controller for cloud-based adaptive video streaming". In : *2013 IEEE International Conference on Communications Workshops (ICC)*. Budapest, Hungary : IEEE, juin 2013, p. 723-727. ISBN : 978-1-4673-5753-1. DOI : 10.1109/ICCW.2013.6649328. URL : http://ieeexplore.ieee.org/document/6649328/ (visité le 05/03/2019).

[49] DELL. *Spécifications matérielles pour la station de travail Precision T3600*. 2012. URL : https://www.dell.com/support/article/fr-fr/sln290703/the-hardware-specifications-for-the-precision-t3600-desktop-workstation.

[50] Peter J. DENNING. "The working set model for program behavior". en. In : *Proceedings of the ACM symposium on Operating System Principles - SOSP '67*. Not Known : ACM Press, 1967, p. 15.1-15.12. DOI : 10.1145/800001.811670. URL : http://portal.acm.org/citation.cfm?doid=800001.811670 (visité le 13/03/2019).

[51] Peter J. DENNING. "Virtual Memory". In : *ACM Computing Surveys* 2.3 (sept. 1970), p. 153-189. ISSN : 03600300. DOI : 10.1145/356571.356573. URL : http://portal.acm.org/citation.cfm?doid=356571.356573 (visité le 06/10/2017).

[52] DOCKER. *Docker CE*. URL : https://github.com/docker/docker-ce.

[53] DOCKER. *Docker CLI*. URL : https://github.com/docker/cli.

[54] DOCKER. *Docker Hub*. URL : https://hub.docker.com/.

[55] DOCKER. *What is a Container ?* URL : https://www.docker.com/resources/what-container.

[56] *docker-py*. URL : https://github.com/docker/docker-py.

[57] Simon DUPONT et al. "Experimental Analysis on Autonomic Strategies for Cloud Elasticity". In : *2015 International Conference on Cloud and Autonomic Computing*. Boston, MA, USA : IEEE, sept. 2015, p. 81-92. ISBN : 978-1-4673-9566-3. DOI : 10.1109/ICCAC.2015.22. URL : http://ieeexplore.ieee.org/document/7312143/ (visité le 19/02/2020).

[58] Tim FISHER. *What Does a Hard Drive's Seek Time Mean ?* Déc. 2019. URL : https://www.lifewire.com/what-does-seek-time-mean-2626007.

[59] Agence Régionale de Santé Île-de FRANCE. *Habitat indigne*. Sept. 2020. URL : https://www.iledefrance.ars.sante.fr/habitat-indigne.

[60] FRANCIS LANIEL. *Thesis_gatling*. 2020. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_gatling.

[61] FRANCIS LANIEL. *Thesis_linux*. 2020. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_linux.

[62] *Gatling*. URL : https://gatling.io/.

[63] Yaozhong GE et al. "Resource Provisioning for MapReduce Computation in Cloud Container Environment". In : *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA, USA : IEEE, sept. 2019, p. 1-4. ISBN : 978-1-72812-522-0. DOI : 10.1109/NCA.2019.8935023. URL : https://ieeexplore.ieee.org/document/8935023/ (visité le 21/04/2020).

[64] GOOGLE. *Google Cloud*. Oct. 2011. URL : https://cloud.google.com/?hl=fr.

[65] GOOGLE. *Preemptible VM Instances*. URL : https://cloud.google.com/compute/docs/instances/preemptible.

[66] *Google Cloud : Équilibrage de charge et scaling*. Déc. 2018. URL : https://cloud.google.com/compute/docs/load-balancing-and-autoscaling#policies.

[67] Mel GORMAN. *Understanding The Linux Virtual Memory Manager*. eng. Bruce Perens' open source series. OCLC : 834670086. Upper Saddle River, NJ : Prentice Hall, 2004. ISBN : 978-0-13-145348-7.

[68] Brendan GREGG. *Working Set Size Estimation*. Fév. 2018. URL : http://www.brendangregg.com/wss.html.

[69] Joseph L HELLERSTEIN. *Feedback control of computing systems*. English. OCLC : 1120403456. New York : IEEE Press : Wiley, 2004. ISBN : 978-0-471-66881-7. URL : https://ebookcentral.proquest.com/lib/upcatalunya-ebooks/detail.action?docID=214288 (visité le 24/04/2020).

[70] Tejun HEO. *Control Group v2*. Oct. 2015. URL : https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[71] Kamezawa HIROYU. *Cgroup And Memory Resource Controller*. Nov. 2008. URL : https://www.static.linuxfound.org/jp_uploads/seminar20081119/CgroupMemcgMaster.pdf.

[72] Petr Jan HORN. "Autonomic Computing : IBM's Perspective on the State of Information Technology". In : 2001. URL : https://pdfs.semanticscholar.org/1ad1/c619a9b3ba5a3ac597f51c8d15011a83423b.pdf.

[73] Solomon HYKES. *Introducing runC : a lightweight universal container runtime*. Juin 2015. URL : https://www.docker.com/blog/runc/.

[74] INTEL. *5-Level Paging and 5-Level EPT*. Rapp. tech. Intel, mai 2017. URL : https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.

[75] INTEL. *Intel Xeon Processor E5-1603*. 2012. URL : https://ark.intel.com/content/www/fr/fr/ark/products/64600/intel-xeon-processor-e5-1603-10m-cache-2-80-ghz-0-0-gt-s-intel-qpi.html.

[76] INTEL. *Intel Xeon Processor Gold 6130*. 2017. URL : https://ark.intel.com/content/www/us/en/ark/products/120492/intel-xeon-gold-6130-processor-22m-cache-2-10-ghz.html.

[77] INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1 : Basic Architecture*. Rapp. tech. 253665. Intel, oct. 2019. URL : https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[78] INTEL. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D) : System Programming Guide*. Rapp. tech. 325384. Intel, oct. 2019. URL : https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[79] Yury IZRAILEVSKY, Stevan VLAOVIC et Ruslan MESHENBERG. *Fin de la migration de Netflix vers le cloud*. Fév. 2016. URL : https://media.netflix.com/fr/company-blog/completing-the-netflix-cloud-migration.

[80] Bruce JACOB, Spencer NG et David WANG. *Memory Systems : Cache, DRAM, Disk*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2007. ISBN : 978-0-12-379751-3.

[81] JEDEC. *DDR4 SDRAM*. Rapp. tech. JESD79-4B. Juin 2017.

[82] Yu JIN-GANG et al. "Research and Application of Auto-Scaling Unified Communication Server Based on Docker". In : *2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA)*. Changsha : IEEE, oct. 2017, p. 152-156. ISBN : 978-1-5386-1230-9. DOI : 10.1109/ICICTA.2017.41. URL : http://ieeexplore.ieee.org/document/8089924/ (visité le 26/02/2019).

[83] JP9000 et DMITRY-ME. *OBS Studio*. 2018. URL : https://github.com/obsproject/obs-studio/blob/c938ea712bce0e9d8e0cf348fd8f77725122b9a5/libobs/media-io/frame-rate.h#L21.

[84] *Kata Containers*. 2017. URL : https://katacontainers.io/.

[85] J.O. KEPHART et D.M. CHESS. "The vision of autonomic computing". en. In : *Computer* 36.1 (jan. 2003), p. 41-50. ISSN : 0018-9162. DOI : 10.1109/MC.2003.1160055. URL : http://ieeexplore.ieee.org/document/1160055/ (visité le 19/02/2020).

[86] Michael KERRISK. *LCE : The failure of operating systems and how we can fix it*. Nov. 2012. URL : https://lwn.net/Articles/524952/.

[87] Michael KERRISK. *Namespaces in operation, part 1 : namespaces overview*. Jan. 2013. URL : https://lwn.net/Articles/531114/.

[88] Jinchun KIM et al. "Dynamic Memory Pressure Aware Ballooning". en. In : *Proceedings of the 2015 International Symposium on Memory Systems - MEMSYS '15*. Washington DC, DC, USA : ACM Press, 2015, p. 103-112. ISBN : 978-1-4503-3604-8. DOI : 10.1145/2818950.2818967. URL : http://dl.acm.org/citation.cfm?doid=2818950.2818967 (visité le 12/03/2019).

[89] Yousri KOUKI et Thomas LEDOUX. "SLA-driven capacity planning for Cloud applications". In : *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. Taipei, Taiwan : IEEE, déc. 2012, p. 135-140. ISBN : 978-1-4673-4510-1 978-1-4673-4511-8 978-1-4673-4509-5. DOI : 10.1109/CloudCom.2012.6427519. URL : http://ieeexplore.ieee.org/document/6427519/ (visité le 28/02/2019).

[90] M. KRIUSHANTH et L. AROCKIAM. "Load balancer behavior identifier (LoBBI) for dynamic threshold based auto-scaling in cloud". In : *2015 International Conference on Computer Communication and Informatics (ICCCI)*. Coimbatore, India : IEEE, jan. 2015, p. 1-5. ISBN : 978-1-4799-6804-6 978-1-4799-6805-3. DOI : 10.1109/ICCCI.2015.7218115. URL : http://ieeexplore.ieee.org/document/7218115/ (visité le 26/02/2019).

[91] Greg KROAH-HARTMAN. *Linux 4.19*. Oct. 2018. URL : https://lkml.org/lkml/2018/10/22/184.

[92] KUBERNETES. *Kubernetes*. URL : https://kubernetes.io/.

[93] *kubernetes/autoscaler*. Fév. 2019. URL : https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md.

[94] KVM. *Automatic Ballooning*. 2013. URL : https://www.linux-kvm.org/page/Projects/auto-ballooning.

[95] Francis LANIEL. *Thesis_experiments*. 2020. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_experiments.

[96] Francis LANIEL. *Thesis_g5kscripts*. 2019. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_g5kscripts.

[97] Francis LANIEL. *Thesis_images*. 2020. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_images.

[98] Francis LANIEL. *Thesis_scripts*. 2020. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_scripts.

[99] Francis LANIEL. *Thesis_sysbench*. 2019. URL : https://gitlab.com/eiffel_thesis/thesis_software/thesis_sysbench.

[100] Francis LANIEL et al. "Highlighting the Container Memory Consolidation Problems in Linux". In : *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA, USA : IEEE, sept. 2019, p. 1-4. ISBN : 978-1-72812-522-0. DOI : 10.1109/NCA.2019.8935034. URL : https://ieeexplore.ieee.org/document/8935034/ (visité le 18/02/2020).

[101] Andrew LARKIN. *Disadvantages of Cloud Computing*. Août 2019. URL : https://cloudacademy.com/blog/disadvantages-of-cloud-computing/.

[102] Yunchun LI et Yumeng XIA. "Auto-scaling web applications in hybrid cloud based on docker". In : *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*. Changchun, China : IEEE, déc. 2016, p. 75-79. ISBN : 978-1-5090-2129-1. DOI : 10.1109/ICCSNT.2016.8070122. URL : http://ieeexplore.ieee.org/document/8070122/ (visité le 26/02/2019).

[103] Huan LIU. "A Measurement Study of Server Utilization in Public Clouds". In : *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*. Sydney, Australia : IEEE, déc. 2011, p. 435-442. ISBN : 978-1-4673-0006-3 978-0-7695-4612-4. DOI : 10.1109/DASC.2011.87. URL : http://ieeexplore.ieee.org/document/6118751/ (visité le 20/06/2019).

[104] Tania LORIDO-BOTRAN, Jose MIGUEL-ALONSO et Jose A. LOZANO. "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments". en. In : *Journal of Grid Computing* 12.4 (déc. 2014), p. 559-592. ISSN : 1570-7873, 1572-9184. DOI : 10.1007/s10723-014-9314-7. URL : http://link.springer.com/10.1007/s10723-014-9314-7 (visité le 26/02/2019).

[105] Robert LOVE. *Linux kernel development*. 3rd ed. Developer's library : essential references for programming professionals. OCLC : ocn268788260. Upper Saddle River, NJ : Addison-Wesley, 2010. ISBN : 978-0-672-32946-3.

[106] *LXC*. 2008. URL : https://linuxcontainers.org/lxc/.

[107] Luc MALRAIT, Sara BOUCHENAK et Nicolas MARCHAND. "Experience with CONSER : A System for Server Control through Fluid Modeling". In : *IEEE Transactions on Computers* 60.7 (juil. 2011), p. 951-963. ISSN : 0018-9340. DOI : 10.1109/TC.2010.164. URL : http://ieeexplore.ieee.org/document/5499462/ (visité le 27/02/2019).

[108] Luc MALRAIT, Sara BOUCHENAK et Nicolas MARCHAND. "Fluid modeling and control for server system performance and availability". In : *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. Lisbon, Portugal : IEEE, juin 2009, p. 389-398. ISBN : 978-1-4244-4422-9. DOI : 10.1109/DSN.2009.5270311. URL : http://ieeexplore.ieee.org/document/5270311/ (visité le 27/02/2019).

[109] Michael MAURER et al. "Revealing the MAPE loop for the autonomic management of Cloud infrastructures". In : *2011 IEEE Symposium on Computers and Communications (ISCC)*. Corfu, Greece : IEEE, juin 2011, p. 147-152. ISBN : 978-1-4577-0680-6. DOI : 10.1109/ISCC.2011.5984008. URL : http://ieeexplore.ieee.org/document/5984008/ (visité le 26/02/2019).

[110] Russ MCKENDRICK et Scott GALLAGHER. *Mastering Docker : master this widely used containerization tool*. eng. Second edition. OCLC : ocn995131069. Birmingham, Mumbai : Packt, 2017. ISBN : 978-1-78728-024-3.

[111] David MEISNER, Brian T. GOLD et Thomas F. WENISCH. "PowerNap : eliminating server idle power". en. In : *ACM SIGARCH Computer Architecture News* 37.1 (mar. 2009), p. 205. ISSN : 01635964. DOI : 10.1145/2528521.1508269. URL : http://dl.acm.org/citation.cfm?doid=2528521.1508269 (visité le 27/03/2018).

[112] P M MELL et T GRANCE. *The NIST definition of cloud computing*. en. Rapp. tech. NIST SP 800-145. Edition : 0. Gaithersburg, MD : National Institute of Standards et Technology, 2011, NIST SP 800-145. DOI : 10.6028/NIST.SP.800-145. URL : https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visité le 09/07/2020).

[113] *Memory Resource Controller*. URL : https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/memory.html.

[114] MICROSOFT. *Microsoft Azure*. 2010. URL : https://azure.microsoft.com/en-us/.

[115] MICROSOFT. *Microsoft Web App for Containers*. URL : https://azure.microsoft.com/en-us/services/app-service/containers/.

[116] MICROSOFT. *SLA for Azure Cosmos DB*. Sept. 2018. URL : https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/.

[117] MICROSOFT. *SLA summary for Azure services*. URL : https://azure.microsoft.com/en-us/support/legal/sla/summary/.

[118] *Microsoft Azure : Autoscaling*. Mai 2017. URL : https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling.

[119] *Moby*. URL : https://github.com/moby/moby.

[120] *mysql*. URL : https://dev.mysql.com/doc/refman/5.7/en/.

[121] Assemblée Nationale. *Projet de Loi de programmation de la recherche pour les années 2021 à 2030 et portant diverses dispositions relatives à la recherche et à l'enseignement supérieur*. Nov. 2020. URL : http://www.assemblee-nationale.fr/dyn/15/textes/l15t0501_texte-adopte-provisoire.pdf.

[122] J. von Neumann. "First draft of a report on the EDVAC". In : *IEEE Annals of the History of Computing* 15.4 (1993), p. 27-75. ISSN : 1058-6180. DOI : 10.1109/85.238389. URL : http://ieeexplore.ieee.org/document/238389/ (visité le 28/05/2020).

[123] opencontainers. *runc*. URL : https://github.com/opencontainers/runc.

[124] Alan Ott. *Virtual Memory and Linux*. Avr. 2016. URL : https://elinux.org/images/4/4c/Ott.pdf.

[125] OVH. *CONDITIONS PARTICULIÈRES DU SERVICE OVH PUBLIC CLOUD*. URL : https://www.ovh.com/fr/support/documents_legaux/contractPartOVHCloudFr.pdf.

[126] OVH. *OVH Public Cloud*. 2010. URL : https://us.ovhcloud.com/products/public-cloud.

[127] *Podman*. URL : https://podman.io/.

[128] Gerald J. Popek et Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures". In : *Communications of the ACM* 17.7 (juil. 1974), p. 412-421. ISSN : 00010782. DOI : 10.1145/361011.361073. URL : http://portal.acm.org/citation.cfm?doid=361011.361073 (visité le 30/10/2018).

[129] Arnaud Porterie. *Docker 1.11 : The first runtime built on containerd and based on OCI technology*. Avr. 2016. URL : https://blog.docker.com/2016/04/docker-engine-1-11-runc/.

[130] *Probe effect*. URL : https://www.semanticscholar.org/topic/Probe-effect/902040.

[131] *Process Number Controller*. URL : https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/pids.html.

[132] *Qemu*. URL : https://github.com/qemu/qemu.

[133] Qemu. *Qemu*. English. Mai 2019. URL : https://wiki.archlinux.org/index.php/QEMU.

[134] Rami Rosen. *Namespace and cgroups, the basis of Linux containers*. Seville, Spain, fév. 2016. URL : https://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf.

[135] Jia RAO et al. "A Distributed Self-Learning Approach for Elastic Provisioning of Virtualized Cloud Resources". In : *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. Singapore, Singapore : IEEE, juil. 2011, p. 45-54. ISBN : 978-1-4577-0468-0. DOI : 10.1109/MASCOTS.2011.47. URL : http://ieeexplore.ieee.org/document/6005367/ (visité le 26/02/2019).

[136] RIK VAN RIEL. *[PATCH -mm 06/24] vmscan : split LRU lists into anon & file share*. Anglais. Juin 2008. URL : https://lkml.org/lkml/2008/6/11/276.

[137] Rami ROSEN. *Understanding the new control groups API*. Mar. 2016. URL : https://lwn.net/Articles/679786/.

[138] SANCGARG. *Documentation/9psetup*. Sept. 2010. URL : https://wiki.qemu.org/Documentation/9psetup.

[139] Colin SCOTT. *Latency Numbers Every Programmer Should Know*. 2020. URL : https://colin-scott.github.io/personal_website/research/interactive_latency.html.

[140] SHURIK'N et AKHENATON. *Rêvolution*. Mar. 2017. URL : https://genius.com/Iam-revolution-lyrics.

[141] James E. SMITH et Ravi NAIR. *Virtual machines : versatile platforms for systems and processes*. Amsterdam ; Boston : Morgan Kaufmann Publishers, 2005. ISBN : 978-1-55860-910-5.

[142] Androski SPICER. *Deep Dive on Amazon EC2*. Jan. 2017. URL : https://www.slideshare.net/AmazonWebServices/deep-dive-on-amazon-ec2.

[143] SPLOITFUN. *Syscalls used by malloc*. Fév. 2015. URL : https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc/.

[144] Akihiro SUDA. *The current adoption status of cgroup v2 in containers*. Oct. 2019. URL : https://medium.com/nttlabs/cgroup-v2-596d035be4d7.

[145] Richard S. SUTTON et Andrew G. BARTO. *Reinforcement learning : an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts : The MIT Press, 2018. ISBN : 978-0-262-03924-6.

[146] Tom SWEENEY. *First Look : Rootless Containers and cgroup v2 on Fedora 31*. Oct. 2019. URL : https://podman.io/blogs/2019/10/29/podman-crun-f31.html.

[147] Fernanda TAFOYA. *10 Cloud Computing Success Stories by Leveraging AWS and Azure*. Juil. 2019. URL : https://www.itexico.com/blog/10-cloud-computing-success-stories-by-leveraging-aws-and-azure.

[148] THE KERNEL DEVELOPMENT COMMUNITY. *ioctl based interfaces*. URL : https://www.kernel.org/doc/html/latest/driver-api/ioctl.html.

[149] THE KERNEL DEVELOPMENT COMMUNITY. *Linux 4.19*. Oct. 2018. URL : https://elixir.bootlin.com/linux/v4.19/source.

[150] *The Slow Query Log*. URL : https://dev.mysql.com/doc/refman/5.7/en/slow-query-log.html.

[151] Sorbonne UNIVERSITÉ. *Charte du doctorat de Sorbonne Université*. 2019. URL : http://ifd.sorbonne-universite.fr/_resources/1-doctorat/textes-reference/CharteValid%25C3%25A9eCA_2_7_19.pdf.

[152] Kim WEINS. *Where Is the $10B in Waste in Public Cloud Costs?* Nov. 2017. URL : https://blogs.flexera.com/cloud/cloud-cost-analysis/where-is-the-10b-in-waste-in-public-cloud-costs/.

[153] WIKIPEDIA CONTRIBUTORS. *Fork bomb — Wikipedia, The Free Encyclopedia*. [Online ; accessed 2-June-2020]. 2020. URL : https://en.wikipedia.org/w/index.php?title=Fork_bomb&oldid=958857438.

[154] WIKTIONNAIRE. *mémoire — Wiktionnaire*. [En ligne ; accédé le 17-avril-2020]. 2020. URL : https://fr.wiktionary.org/w/index.php?title=m%C3%A9moire&oldid=27782952.

[155] John WILKES. *Yet more Google compute cluster trace data*. Published : Google research blog. Mountain View, CA, USA, avr. 2020. URL : https://ai.googleblog.com/2020/04/yet-more-google-compute-cluster-trace.html.

[156] *WooCommerce*. URL : https://woocommerce.com/.

[157] *WordPress*. URL : https://wordpress.org/.

[158] *WorldCup98*. 1998. URL : ftp://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[159] Wm. A. WULF et Sally A. MCKEE. "Hitting the memory wall : implications of the obvious". en. In : *ACM SIGARCH Computer Architecture News* 23.1 (mar. 1995), p. 20-24. ISSN : 0163-5964. DOI : 10.1145/216585.216588. URL : https://dl.acm.org/doi/10.1145/216585.216588 (visité le 29/05/2020).

[160] *Xen*. URL : https://xenbits.xen.org/gitweb/?p=xen.git;a=summary.

[161] Peter ZAITSEV. *The Impact of Swapping on MySQL Performance*. Jan. 2017. URL : https://www.percona.com/blog/2017/01/13/impact-of-swapping-on-mysql-performance/.

[162] Farzin ZAKER. *Online Shopping Store - Web Server Logs*. Version Number : V1. Harvard Dataverse, 2019. DOI : 10.7910/DVN/3QBYB5. URL : https://doi.org/10.7910/DVN/3QBYB5.

[163] Qi ZHANG et al. "A Comparative Study of Containers and Virtual Machines in Big Data Environment". In : *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. San Francisco, CA, USA : IEEE, juil. 2018, p. 178-185. ISBN : 978-1-5386-7235-8. DOI : 10.1109/CLOUD.2018.00030. URL : https://ieeexplore.ieee.org/document/8457798/ (visité le 15/04/2020).

Certaines figures de cette thèse utilisent des icônes provenant des auteurs suivants :

— Smashicons de Flaticon
— Nikita Golubev de Flaticon
— Freepik de Flaticon