Mise à l'échelle hybride des microservices conteneurisés

par

Aziz BEN CHEIKH LARBI

MÉMOIRE PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DE LA MAÎTRISE
AVEC MÉMOIRE EN GÉNIE DES TECHNOLOGIES DE L'INFORMATION
M. Sc. A.

MONTRÉAL, LE 6 MAI 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE:

M. Abdelouahed Gherbi, Directeur de mémoire
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

Mme. Nadjia Kara, co-Directrice
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Kaiwen Zhang, Président du jury
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

M. Aris Leivadeas, Membre du jury
Département de génie logiciel et des technologies de l'information à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE "22 AVRIL 2021"

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je profite de l'occasion qui m'est offerte pour exprimer ma gratitude et ma reconnaissance envers tous ceux qui ont permis à ce projet de voir le jour.

Je souhaite remercier mes directeurs de recherche M. Abdelouahed GHERBI et Mme. Nadjia KARA pour leurs encadrements, leurs disponibilités et leurs conseils qu'ils ont pu me prodiguer au cours de mon mémoire.

Par la même occasion, j'adresse ma reconnaissance à tous les enseignants qui ont assuré ma formation au sein de l'ETS.

Finalement, je tiens à remercier les membres du jury qui m'ont fait l'honneur de juger mon présent travail.

Mise à l'échelle hybride des microservices conteneurisés

Aziz BEN CHEIKH LARBI

RÉSUMÉ

La mise à l'échelle automatique est primordiale pour les fournisseurs de services infonuagiques. La précision et la réactivité sont des critères recherchés afin d'éviter des coûts financiers supplémentaires. D'une part, la sur-allocation de ressources engendre un coût d'achat et d'utilisation de plus de machines. D'autre part, la sous-allocation engendre une dégradation du QoS qui se traduit en violations des SLAs.

Dans le domaine des conteneurs, la solution la plus populaire est la mise à l'échelle horizontale de Kubernetes (HPA). Dans ce travail, nous proposant deux algorithmes de mise à l'échelle hybride bénéficiant des avantages du scaling horizontal et vertical à la fois. En outre, notre deuxième algorithme utilise un profileur, capable d'estimer le besoin en CPU pour répondre à un certain nombre de requêtes reçues. Nos algorithmes ont été testés dans différentes situations et leurs performances sont comparées à ceux de HPA de Kubernetes. Les résultats obtenus démontrent une amélioration importante de QoS, d'utilisation et d'allocation de ressources, ainsi que dans la consommation d'énergie.

Mots-clés: Docker, conteneur, mise à l'échelle hybride, microservice

Hybrid scaling of containerized microservices

Aziz BEN CHEIKH LARBI

ABSTRACT

Autoscaling is critical for cloud service providers. Precision and reactivity are criteria sought in order to avoid extra financial costs. Indeed, the over-allocation of resources generates a cost of purchasing and using more machines. On the other hand, the under-allocation leads to a degradation of the QoS which results in violations of the SLAs.

In the containers domain, the most popular solution is horizontal scaling of Kubernetes (HPA). In this work, we propose two hybrid scaling algorithms benefiting from the advantages of both horizontal and vertical scaling. In addition, our second algorithm uses a profiler able to estimate the CPU requirement to respond to a certain number of received requests. Our algorithms have been tested in different situations, and their performances are compared to those of Kubernetes HPA. The results obtained demonstrate a significant improvement in QoS, resources usage and allocation, as well as in energy consumption.

Keywords: Docker, container, hybrid scaling, microservice

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

API	Application programming interface
AS	Autoscaler
BNF	Besoin non-fonctionnel
CPU	Central processing unit
HPA	Horizontal pod autoscaler
LB	Load balancer
LG	Load generator
MAE	Mean Absolute Error
MGR	Méthode de génération de requêtes
OS	Operating System
QoS	Quality of service
RMSE	Root mean squared error
SLA	Service Level Agreement
VM	Virtual Machine

LISTE DES SYMBOLES ET UNITÉS DE MESURE

c	Cœur de CPU
GB	GigaByte
j	Joule
s	Seconde

INTRODUCTION

Nous assistons ces dernières années à une expansion considérable de la technologie des conteneurs dans différents domaines. En effet, elle a atteint une certaine maturité lui permettant d'être une meilleure option face aux machines virtuelles (VM), et ce dans divers cas d'utilisation. Étant donné qu'elle repose sur une virtualisation au niveau système d'exploitation, il en résulte des tailles réduites des conteneurs, une rapidité des déploiements et une augmentation des performances.

L'utilisation des conteneurs est fortement liée à l'implémentation de l'architecture microservices. Cette dernière est devenue essentielle dans le développement des applications, spécialement celles de grande taille. En effet, cette architecture permet de séparer l'application en plusieurs composants indépendants ou faiblement couplés, permettant ainsi de paralléliser le développement et réduire les responsabilités de chaque module. De surcroît, la mise à jour d'un microservice peut se faire sans affecter les autres composants, ce qui rend le processus de développement plus agile.

Les propriétaires des applications conteneurisées cherchent la meilleure qualité de service au meilleur prix. En conséquence, les fournisseurs de services sont obligés d'allouer suffisamment de ressources aux applications afin d'éviter des violations de SLA (Service Level Agreement) et des pénalités financières. En contrepartie, il est nécessaire d'éviter une sur-allocation de ressources, car elle engendre un coût d'achat et d'utilisation de plus de machines. Conséquemment, une mise à l'échelle automatique en temps réel s'impose, afin de traiter le compromis qui existe.

En termes de mise à l'échelle, il existe d'une part les algorithmes horizontaux, qui consistent à créer ou supprimer des réplicas. D'autre part, il y a les algorithmes verticaux, qui consistent à modifier la quantité des ressources allouées à une instance donnée. Étant donné que chacune de ces deux catégories comporte des qualités différentes, l'objectif principal de ce projet est de

développer une solution de mise à l'échelle hybride, profitant ainsi des avantages des scalabilités horizontale et verticale.

0.1 Problématique

Dans le cadre d'allocation de ressources aux microservices, les fournisseurs de services sont confrontés aux contraintes suivantes :

- une sous-allocation des ressources engendre une dégradation de la qualité du service, des violations des ententes de niveau de service et une pénalité financière ;
- une sur-allocation des ressources engendre un coût d'achat de plus de machines et une plus grande consommation d'énergie.

Dans ces conditions, une mauvaise allocation des ressources se traduit forcément par des coûts financiers supplémentaires. De ce fait, notre projet de recherche mettra l'accent sur l'amélioration de la qualité de service, l'efficacité d'utilisation de ressources et la consommation d'énergie en appliquant la mise à l'échelle hybride.

0.2 Limites

Dans le but de bien définir notre projet de recherche, nous posons les limites suivantes :

1. Nous nous limitons à la mise à l'échelle des microservices conteneurisés. Ceci nous permettra d'investiguer sur l'estimation des ressources nécessaires pour le bon fonctionnement des microservices face aux requêtes reçues ;
2. Nous nous limitons à l'allocation du CPU, puisque cette métrique est compressible, en opposition à d'autres comme la mémoire ;
3. Nous nous limitons à une approche réactive de mise à l'échelle, excluant la prédiction du besoin en ressources dans le futur.

0.3 Contributions

Il est fondamental de traiter notre projet de façon à apporter des contributions et de l'originalité. Cette originalité doit s'appuyer tout d'abord sur une bonne connaissance de l'état de l'art.
À cet effet, notre travail présente plusieurs contributions se résumant aux points suivants :

- la conception et l'implémentation d'une plateforme d'orchestration des conteneurs, qui est capable de tester et de comparer des algorithmes de mise à l'échelle et d'ordonnancement ;
- le développement d'un profileur de microservices, capable d'estimer la quantité de CPU nécessaire pour répondre aux requêtes reçues ;
- la conception et l'implémentation des deux algorithmes hybrides de mise à l'échelle ;
- l'analyse des performances des algorithmes implémentés à travers des expériences dans un environnement contrôlé.

0.4 Plan du mémoire

Le présent rapport synthétise le travail réalisé durant notre projet de recherche, il est subdivisé en trois chapitres :

- tout d'abord l'introduction, où nous avons présenté le cadre du projet et ses motivations, la problématique et les limites ainsi que les contributions ;
- le premier chapitre est consacré à l'état de l'art, il introduit les concepts de base nécessaires à la compréhension du projet et présente la revue de littérature ;
- le deuxième chapitre présente la méthodologie utilisée ; il détaille l'architecture de la plateforme de test et ses différents composants. Ensuite, il explique le fonctionnement des algorithmes implémentés et développe la phase de création du profileur ;
- le troisième chapitre traite les résultats expérimentaux. Dans ce contexte, nous présentons la configuration matérielle et logicielle utilisée lors des expériences. Ensuite, nous révélons les résultats des expériences et les interprétations de ces données ;

- en dernier lieu, nous élaborons la conclusion générale et les recommandations. Cette partie résume le travail réalisé et présente les perspectives de notre projet.

CHAPITRE 1

ÉTAT DE L'ART

L'état de l'art est une étape cruciale dans la réalisation de notre projet. Dans ce premier chapitre, nous nous intéressons à l'étude des concepts clés sur lesquels se focalisent nos travaux. Ensuite, nous présentons une revue de la littérature dévoilant les solutions existantes, leurs points forts et leurs faiblesses.

1.1 Concepts de base

Dans cette section, nous présentons les terminologies utilisées au cours de notre travail. Nous définissions l'architecture microservices suivie de la technologie des conteneurs.

1.1.1 Architecture microservices

D'après IBM Cloud Education (2019), l'architecture en microservices est en opposition à une architecture monolithique. En effet, elle décompose une application en plusieurs composants (microservices) faiblement couplés, déployés indépendamment sur une ou plusieurs machines. Ces différents modules communiquent entre eux à travers des agents de messages et des APIs.

Cette architecture apporte plusieurs avantages, notamment la précision de la mise à l'échelle ; elle permet d'ajouter des ressources aux composants les plus sollicités seulement, au lieu de la totalité de l'application. De surcroît, les microservices peuvent être développés parallèlement par différentes équipes, en utilisant différents langages de programmation.

1.1.2 Technologie des conteneurs

Un conteneur est une enveloppe applicative isolée, permettant de contenir une application (généralement un microservice) et toutes ses dépendances, telles que les librairies. Les conteneurs utilisent et partagent le noyau du système d'exploitation de la machine hôte. En conséquence, cela leur donne une taille réduite par rapport au VMs et engendre une rapidité de déploiement.

La figure 1.1 illustre les différences entre une machine virtuelle (VM) et un conteneur.

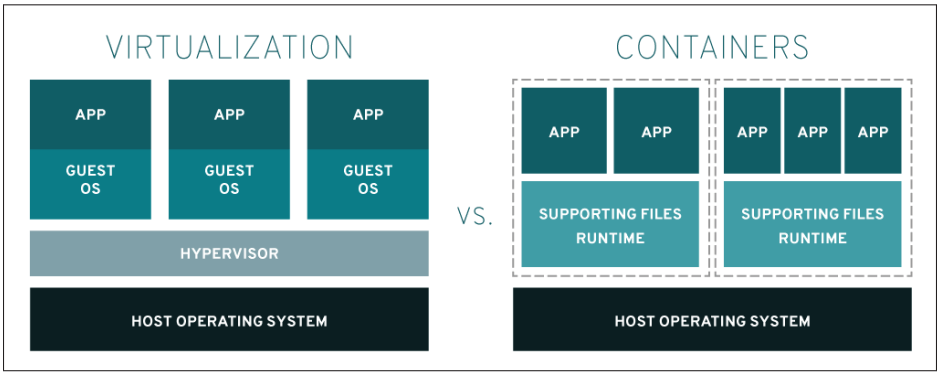

Figure 1.1 Différences entre VMs et conteneurs, tirée de RedHat (2020)

Comme illustré dans la figure 1.1, en opposition aux VMs, les conteneurs exploitent la virtualisation au niveau système d'exploitation. De ce fait, ils n'ont pas besoin de système d'exploitation invité ni d'hyperviseur.

En termes d'efficacité par rapport aux VMs, les conteneurs présentent de meilleures performances dans la plupart des cas. Ceci est validé par les analyses de Felter et al. (2015) ainsi que celles de Potdar et al.(2020).

De plus, selon les travaux de Ferreira et al. (2017), différents conteneurs déployés sur la même machine peuvent partager l'utilisation de la même librairie (inclus dans leurs images) en utilisant un point de montage en union AUFS. Conséquemment, cela augmente davantage le gain en termes d'utilisation de mémoire.

Parmi les types de conteneurs les plus populaires, nous pouvons citer ceux de Docker. En effet, ce dernier a permis de démocratiser l'utilisation des conteneurs grâce à sa probabilité et sa flexibilité. En terme de processeur, un conteneur Docker peut avoir un quota qui spécifie la quantité maximale utilisable. Il peut aussi avoir une priorité, lui garantissant en tout temps une proportion en cycles de CPU.

1.2 Revue de la littérature

Dans cette section, nous présentons les travaux de recherches et les solutions existantes dans le domaine de mise à l'échelle. Nous présentons ainsi les travaux réalisés sur les VMs et les conteneurs, en vue de leur similarité. Ceci nous permettra de prendre connaissance des progrès atteints dans ce domaine, de trouver les faiblesses dans ces solutions afin de les éviter dans notre projet, ainsi que de savoir leurs points forts dans le but de s'en inspirer.

1.2.1 Mise à l'échelle horizontale

Dans notre contexte, la mise à l'échelle horizontale consiste à ajouter ou diminuer le nombre d'instances d'un microservice suivant le besoin en ressources. Même si la création de nouveaux réplicas requière généralement un répartiteur de charge afin de redistribuer les requêtes, elle permet de renforcer la résilience, diminuer la nécessité d'achat de machines puissantes et faciliter la mise à jour graduelle des microservices. En revanche, elle comporte certains désavantages. D'une part, elle nécessite que les applications soient décomposables et réplicables, sinon des interruptions de sessions ou des incohérences de données pourraient se produire. D'autre part, elle n'est pas précise, puisque la plus petite unité gérée est l'instance, au lieu d'unité de ressources réelles telles que des cœurs de CPU ou des mégaoctets de mémoire. De plus, elle est lente à cause du temps de démarrage des nouveaux réplicas, toutefois, ce coût est relativement faible pour les conteneurs, ce qui rend la mise à l'échelle horizontale très répondue lorsque ces derniers sont utilisés.

En ce qui concerne les VMs, plusieurs plateformes proposent le scaling horizontal. Nous pouvons citer Azure de Microsoft, Amazon EC2, Openstack avec Heat et Cloudstack.

Azure de Microsoft (2020) propose le concept de *scale sets*, celui-ci permet de définir un comportement élastique aux VMs. En effet, il offre une multitude de règles comportant chacune des seuils sur différentes métriques (CPU, mémoire, trafic réseau, etc.). L'utilisateur choisit ensuite l'action à faire lors du dépassement de ces seuils, comme l'ajout ou le retrait d'un nombre donné de réplicas.

Ceil est une solution proposée par Kohei & Kourai (2020) qui traite l'autoscaling de VMs contenant des services conteneurisés. Tout d'abord, le système surveille les besoins des services dans chaque VM. Ensuite, lorsque certains services requièrent plus de ressources, le système crée une nouvelle VM contenant seulement les services en question. D'après les auteurs, cette méthode augmente la précision de mise à l'échelle et diminue le coût des VMs ainsi que le temps de leur création. Bien que cette solution apporte un certain nombre d'avantages, elle augmente la complexité et génère des coûts supplémentaires par rapport à l'utilisation des conteneurs sans VMs.

Fernandez et al. (2014) ont élaboré un autoscaler de VMs capable de gérer différents niveaux de QoS pour chaque utilisateur. Le système repose principalement sur trois parties. Premièrement, le profileur qui est responsable de gérer l'hétérogénéité du matériel et augmente la précision du scaling. Deuxièmement, le prédicteur qui permet d'estimer la demande future sur les services. Troisièmement, le répartiteur de charge qui permet d'augmenter les poids des serveurs les plus puissants et de répartir les requêtes suivant ces poids.

Concernant les conteneurs, la solution d'orchestration la plus populaire est celle de Kubernetes. En effet, ce dernier permet d'automatiser le déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Le scaleur principal de Kubernetes (2020a) est l'autoscaler de pod horizontal (HPA). Il est capable de surveiller plusieurs métriques (CPU, mémoire, etc.) et d'agir en conséquence. Son rôle est d'ajouter ou d'enlever des réplicas afin de maintenir une moyenne d'utilisation de ressources proche d'une valeur désirée. Nous présenterons son fonctionnement plus en détail dans le chapitre suivant.

1.2.2 Mise à l'échelle verticale

Dans le cadre de notre projet, la mise à l'échelle verticale consiste à augmenter ou diminuer les ressources allouées à une instance d'un microservice. Malgré que cette méthode est limitée aux ressources de la machine physique, elle reste plus appropriée lorsqu'assez de ressources y sont disponibles. Elle apporte ainsi différents avantages, notamment le support des applications

non-décomposables ou non-réplicables d'une part, et elle permet d'éliminer les attentes de démarrage des nouvelles instances et l'utilisation d'un répartiteur de charge d'autre part. De surcroît, elle réduit l'effort administratif et augmente la précision d'allocation de ressources. En revanche, elle engendre un point de défaillance unique vu l'absence de réplicas et rend les mises à jour des microservices plus difficiles.

Kubernetes (2020b) a intégré récemment un autoscaler de pod vertical (VPA). Il permet ainsi de modifier verticalement les ressources des conteneurs, tout en gardant le même ratio entre *limit* et *requested*. Toutefois, la mise à jour des ressources allouées à un conteneur est réalisée en récréant ce conteneur, possiblement sur un autre nœud. De plus, il ne tient pas compte de la capacité des nœuds, rendant le pod non ordonnançable à cause de sa grande taille. Bien que Kubernetes propose les deux types de scaling, ces derniers ne peuvent pas gérer la même métrique conjointement puisqu'ils entreraient en conflit.

ElasticDocker est une autre solution de scaling vertical de conteneurs élaborée par Al-Dhuraibi et al. (2017). Entre autres, elle implémente une boucle MAPE-K (Monitor, Analyze, Plan, Execute and Knowledge) pour surveiller et gérer le CPU et la mémoire des conteneurs. Lorsqu'une insuffisance de ressources est détectée sur un nœud, une migration en direct vers une autre machine est réalisée. En somme, les auteurs de ce système prétendent une réduction du temps d'exécution des conteneurs (de 38%) par rapport à l'autoscaler horizontal de kubernetes. En revanche, la période de la boucle de contrôle de kubernetes est de 30 secondes et celle de ElasticDocker est de 16 secondes, ce qui avantage injustement ElasticDocker.

Rajamani et al. (2015) ont proposé une autre solution de mise à l'échelle verticale appelée Spyre. Le framework fonctionne en isolant les conteneurs de chaque tenant en un groupe appelé *Slice*. De ce fait, chaque groupe de conteneurs partage un ensemble de ressources (cœurs du CPU, mémoire et accès aux réseaux). Ces ressources peuvent s'ajuster verticalement selon le besoins des conteneurs. Bien que l'élasticité verticale apporte une meilleure précision, Spyre utilise une allocation de CPU en termes de cœurs physiques dédiés, ce qui augmente l'isolation, mais

diminue considérablement la granularité. De plus, l'élasticité des ressources est appliquée sur les slices et non les conteneurs individuellement, ce qui diminue davantage la précision.

1.2.3 Mise à l'échelle hybride

Dans notre contexte, la mise à l'échelle hybride consiste d'une part, à modifier verticalement les ressources allouées aux conteneurs, d'autre part, ajouter ou supprimer des réplicas à un microservice. De ce fait, le scaling hybrid profite plus ou moins des avantages combinés des mises à l'échelle verticale et horizontale. Toutefois, il existe peu de solutions utilisant ce type de scaling, d'où nous est venue l'idée d'investiguer sur le sujet et proposer notre solution.

Ye et al. (2017) ont proposé une approche proactive de mise à l'échelle hybride. Tout d'abord, lors des montées de charge, le système utilise l'approvisionnement vertical en vu de sa rapidité. Ensuite, lors des baisses de charge, il décroît les ressources horizontalement puisqu'il n'engendre pas des violations de SLA malgré sa lenteur. En outre, le système prédit s'il y a une montée de charge future et augmente le nombre de réplicas et leurs tailles si besoin.

Baresi & Quattrocchi (2020) proposent COCOS, une extension de Kubernetes capable de gérer les conteneurs dans des VMs. Le projet est basé sur trois niveaux. Premièrement, le niveau des conteneurs qui est responsable de fournir ou retirer verticalement les ressources aux conteneurs. Deuxièmement, le niveau des VMs qui est responsable de résoudre les conflits d'accès aux ressources partagées dans le premier niveau, modifier sa période de boucle de contrôle, ainsi que de communiquer avec le niveau suivant pour créer ou supprimer des conteneurs. Troisièmement, le niveau des clusters qui est responsable de la mise à l'échelle horizontale des VM et des conteneurs.

Une autre solution de mise à l'échelle hybride est celle de Kwan et al. (2019). Ils ont utilisé le scaling vertical suivi de l'horizontal pour satisfaire l'évolution du besoin en CPU et en mémoire. Tout d'abord, pour chaque microservice, le système calcule la différence totale entre les ressources allouées et celles utilisées. Ensuite, il fournit ou rétracte verticalement les ressources à chaque réplica. De ce fait, une instance est enlevée si elle atteint le seuil minimal et

une nouvelle instance est créée si l'élasticité verticale ne suffit plus. Dans les grandes lignes, les auteurs prétendent une amélioration (de 35%) de l'efficacité d'utilisation de ressources par rapport au HPA de Kubernetes. Cependant, nous pouvons constater quelques lacunes dans leur approche. Tout d'abord, le scaling vertical fournit à chaque réplica les ressources qu'il en a besoin seulement, sans prendre compte du besoin total du microservice. Ceci engendre la création inutile d'un nouveau réplica dès qu'un seul conteneur est limité par la capacité du nœud, omettant d'agrandir davantage les autres réplicas. De plus, la suppression d'un conteneur se fait que lorsqu'il atteint une taille minimale. Ceci engendre le maintien d'un grand nombre de petites instances, même après une baisse de charge.

CHAPITRE 2

MÉTHODOLOGIE

Dans ce chapitre, nous présentons la plateforme de test et ses différents composants. Ensuite, nous détaillons les algorithmes implémentés dans ce projet. Enfin, nous expliquons la phase de développement du profileur.

2.1 Plateforme de test

Dans cette section, nous commençons par présenter les besoins non-fonctionnels de notre plateforme. Ensuite, nous détaillons l'architecture et chacun de ses composants.

2.1.1 Expression des besoins non-fonctionnels

Le but principal de cette application est d'être un environnement contrôlé pour le test et la comparaison des algorithmes de mise à l'échelle et d'ordonnancement. Ainsi, elle doit respecter les contraintes suivantes :

- BNF 1 **(Probabilité)** : tous les composants de la plateforme doivent être conteneurisés ;
- BNF 2 **(Maintenabilité)** : chaque méthode ou fonction doit avoir un commentaire comme description pour son rôle et les variables doivent avoir des noms significatifs ;
- BNF 3 **(Scalabilité)** : l'ajout de nouveau nœud ouvrier doit pouvoir se faire facilement et sans interruption de service ;
- BNF 4 **(Réutilisabilité)** : chaque composant doit implémenter une API REST ;
- BNF 5 (**Sécurité)** : le système doit obliger l'authentification pour accéder à l'application.

2.1.2 Architecture globale

Dans cette section, nous présentons l'architecture globale de la plateforme de test. La figure 2.1 illustre le diagramme de déploiement de l'application.

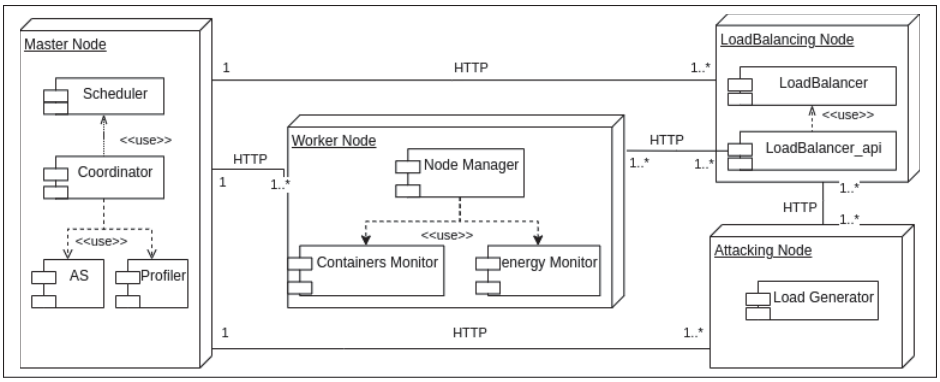

Figure 2.1 Architecture globale de la plateforme de test

Comme le montre le diagramme de la figure 2.1, la plateforme est constituée de plusieurs composants repartis sur plusieurs nœuds. Étant donné que chaque composant du système est un service web proposant une API REST, la configuration physique est flexible et le deploiment de la plateforme peut se faire sur un nombre quelconque de machines.

Afin de comprendre le fonctionnement de la plateforme, nous expliquons le déroulement de la mise à l'échelle à travers le diagramme de séquence disposée dans la figure 2.2 :

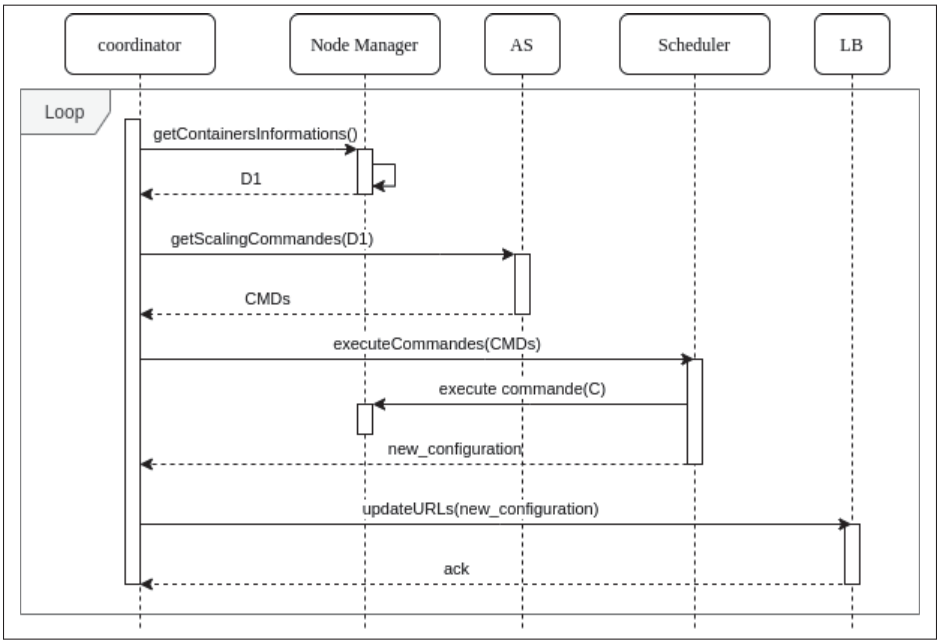

Figure 2.2 Scénario de mise à l'échelle d'un microservice

Dans des intervalles de temps réguliers, le coordinateur collecte la consommation de ressources et d'énergies à partir des gestionnaires de nœuds. Ensuite, il fournit les informations collectées à l'autoscaler (AS) qui lui retourne les commandes de mise à l'échelle à effectuer. Ces dernières sont transmises à l'ordonnanceur qui décide sur quels nœuds les exécuter. Enfin, le coordinateur informe le répartiteur de charge (LB) de la nouvelle disposition des conteneurs ainsi que de leurs poids respectifs.

2.1.2.1 Coordinateur

Le coordinateur est le point central de la plateforme, il s'exécute sur le nœud maître et communique périodiquement avec les modules suivants :

- **gestionnaire de nœud** : afin de récolter les informations d'utilisation des ressources ainsi que la consommation d'énergie ;
- **profileur** : afin d'estimer les ressources nécessaires pour satisfaire les requêtes reçues pour chaque conteneur ;
- **autoscaler** : afin d'invoquer un algorithme de mise à l'échelle ;
- **ordonnanceur** : afin d'exécuter les commandes de mise à l'échelle ;
- **API du répartiteur de charge** : afin de mettre à jours la liste des réplicas et leurs poids ;
- **générateur de requêtes** : afin de commencer ou arrêter la génération de requêtes, ainsi que de sélectionner le type d'attaque à effectuer.

Le coordinateur est l'interface de contrôle de l'application. Il est utilisé afin de choisir les algorithmes d'ordonnancement et de mise à l'échelle, les microservices à déployer ainsi que tous les autres paramètres. C'est le seul composant équipé d'une interface graphique.

2.1.2.2 Moniteur des conteneurs

Le moniteur des conteneurs (CM) s'exécute sur le nœud ouvrier. Il est responsable de fournir les informations de consommation de ressources par conteneur et par machine physique. Étant

donné que nous utilisons Docker comme gestionnaire de conteneurs, nous avons décidé de réutiliser cAdvisor.

CAdvisor est un outil open-source déployable dans un conteneur Docker. Il est utilisé pour fournir différentes métriques de performances telles que la consommation du CPU, l'utilisation de la mémoire et le nombre de paquets reçus par conteneur. Il est généralement utilisé conjointement avec d'autres outils comme Prometheus et InfluxDB. Cependant, afin de réduire la complexité et la consommation de ressources, nous avons décidé d'utiliser cAdvisor sans outils supplémentaires. En conséquence, le moniteur de conteneurs lit des données brutes à partir de l'API de cAdvisor, les traite, puis les retourne au gestionnaire de nœud.

2.1.2.3 Moniteur d'énergie

Le moniteur d'énergie (EM) s'exécute sur le nœud ouvrier, notre implémentation utilise l'interface de RAPL (Running Average Power Limit) de Intel pour estimer la consommation d'énergie du CPU et de la mémoire.

D'après la documentation du *The Linux kernel* (2020), la technologie RAPL fait partie du *Power Capping Framework*, qui est intégré dans le noyau de Linux. Son interface est constituée de plusieurs compteurs incrémentés périodiquement. Chacun de ces compteurs est disposé dans un fichier différent et cible un composant physique donné (CPU, mémoire, etc.). Comme le montre l'analyse de Khan et al. (2018), l'estimation de la consommation d'énergie fournie par l'interface de RAPL se rapproche de celle fournie par un compteur électrique (power meter).

Ce service est accessible par une API REST permettant l'accès aux données, la définition des différents paramètres comme la périodicité des lectures des compteurs et le choix des composants cibles.

2.1.2.4 Gestionnaire de nœud

Le gestionnaire de nœud (NM) s'exécute sur le nœud ouvrier. Il est le responsable de :

- créer et supprimer des conteneurs ;
- modifier la quantité de ressources allouées aux conteneurs ;
- fournir les informations sur la consommation de ressources par conteneurs en faisant appel au moniteur des conteneurs ;
- fournir les informations sur la consommation d'énergie en faisant appel au moniteur d'énergie.

Le gestionnaire de nœuds est l'interface de la machine sur laquelle il s'exécute. Il applique les ordres de l'ordonnanceur et fournit des informations au coordinateur.

2.1.2.5 Répartiteur de charge

Le répartiteur de charge (LB) est responsable de rediriger les requêtes reçues vers des URLs de sorties et la répartition des requêtes se fait selon les poids des réplicas.

Le répartiteur de charge et son API sont séparés en deux conteneurs. l'API modifie le fichier de configuration (URLs et leurs poids) à la demande du coordinateur. Ensuite, le LB lit ce fichier afin d'appliquer les modifications. Cette configuration en «side car» permet de mettre à jour un composant sans avoir à modifier l'autre.

Notre solution utilise HAProxy Community et lui associe une interface REST sous forme de conteneur comme décrit précédemment.

2.1.2.6 Générateur de requêtes

Le générateur de requête s'exécute sur le nœud d'attaque. Il simule l'utilisation des microservices conteneurisés déployés sur les nœuds ouvriers. Il est capable de générer des requêtes avec un débit variable et de fournir des statistiques sur les requêtes de chaque intervalle de temps. Deux fonctions périodiques de génération de requêtes sont intégrées :

- **première méthode de génération de requêtes (MGR-1)** : sous forme d'une fonction sinus, elle permet des transitions lisses et graduelles ;

- **deuxième méthode de génération de requêtes (MGR-2)** : elle contient des changements brusques et de grandes amplitudes permettant de simuler des charges instables.

Dans notre implémentation, nous avons réutilisé le générateur de requêtes Vegeta. Il est utilisé pour tester les services web et permet de mesurer les latences moyennes, le 95e centile ainsi que le taux de requêtes échouées.

2.1.2.7 Profileur

Le profileur s'exécute sur le nœud maître et intègre plusieurs modèles de régression. En premier lieu, son rôle est d'établir une relation entre une variable et d'autres qui lui sont corrélées. Dans notre cas, c'est la quantité de CPU nécessaire pour répondre à un nombre donné de requêtes reçues, et ceci pour chaque microservice.

Dans notre implémentation, nous avons utilisé la librairie Smile. Elle facilite l'utilisation et l'entraînement des modèles avec le langage Java.

Nous détaillons la collecte des données, l'entraînement et la sélection du meilleur modèle dans la section 2.3 «Développement du profileur».

2.1.2.8 Ordonnanceur

L'ordonnanceur (scheduler) s'exécute sur le nœud maître. Il est responsable d'appliquer les ordres de l'autoscaler. Les décisions qu'il effectue sont les suivantes :

- déterminer sur quel nœud créer un réplica. Par défaut, il utilise l'algorithme Round-Robin ;
- déterminer les réplicas à supprimer.

Concernant la mise à l'échelle verticale, l'ordonnanceur exécute les ordres de l'autoscaler sans prendre de décisions, puisqu'ils sont spécifiques à des conteneurs donnés.

Une fois les décisions prises, l'ordonnanceur ordonne au gestionnaire de nœud d'exécuter ses commandes. Ensuite, il informe le coordinateur des changements effectués.

2.1.2.9 Autoscaler

Ce composant s'exécute sur le nœud maître. Il est responsable de prendre des décisions d'ajout ou de retrait de ressources pour chaque microservice et il intègre plusieurs algorithmes de mise à l'échelle, notamment :

- l'algorithme horizontal de Kubernetes (Kube-AS) ;
- notre premier algorithme hybride (Hybrid-AS), qui utilise l'utilisation du CPU pour prendre des décisions ;
- notre deuxième algorithme hybride (Hybrid-P-AS), qui utilise l'estimation du besoin en CPU pour prendre des décisions.

Concernant la mise à l'échelle horizontale, l'autoscaler se contente de donner des ordres d'ajout ou de retrait de conteneurs. Il ne précise pas sur quel nœud la création sera exécutée, ni quel conteneur sera supprimé. Concernant la mise à l'échelle verticale, il spécifie quel conteneur est concerné par la modification de ressources.

2.2 Les algorithmes de mise à l'échelle implémentés

Dans cette section, nous présentons d'abord le fonctionnement de l'algorithme de kubernetes et expliquons ensuite les algorithmes que nous avons conçus.

Tous les algorithmes présentés dans cette section sont implémentés dans une boucle de contrôle d'une période de 5 secondes. De plus, la seule ressource allouée est le CPU.

Chaque microservice a les valeurs suivantes :

- **minReplicas** : c'est le nombre minimum de réplicas attribuables à ce service (1 par défaut) ;
- **maxReplicas** : c'est le nombre maximum de réplicas attribuables à ce service (30 par défaut) ;
- **defaultRequested** : c'est la valeur attribuée à un réplica pour sa valeur *requested*. Elle est spécifique aux algorithmes horizontaux ;
- **maxRequested** : c'est la valeur maximale que peut accepter un réplica pour sa valeur *requested*. Elle est spécifique aux algorithmes hybrides ;

- **minRequested** : c'est la valeur minimale que peut prendre un réplica pour *requested*. Elle est spécifique aux algorithmes hybrides ;
- **target** : c'est un pourcentage par rapport à la valeur *requested* (75% par défaut). Il est utilisé par l'algorithme de la mise à l'échelle pour décider s'il doit ajouter ou retirer des ressources.

Chaque réplica d'un microservice a les variables suivantes :

- **requested** : c'est la quantité de CPU allouée au réplica et qui lui est garantie en tout temps. Elle est variable pour les algorithmes hybrides. Concernant les algorithmes horizontaux, *requested* reçoit la valeur *defaultRequested*. Concrètement, elle est implémentée par le concept de *shares* de Docker ;
- **limit** : c'est la quantité de CPU maximale que peut utiliser un réplica. Cette valeur est fixe pour les algorithmes horizontaux et variable pour les algorithmes hybrides. Elle reçoit par défaut deux fois la valeur du *requested*. Pratiquement, la limite est implémentée en utilisant «*quota*» et «*period*» de Docker ;
- **desiredValue** : c'est le produit des valeurs *requested* et *target*. Si l'utilisation du CPU dépasse cette quantité, le conteneur est considéré en manque de ressources ;
- **usage** : c'est la moyenne du CPU utilisé par le réplica pendant l'intervalle précédent.

2.2.1 HPA de kubernetes

Le premier algorithme que nous détaillons est l'algorithme horizontal de mise à l'échelle de Kubernetes. Il consiste à changer automatiquement le nombre de réplicas suivant l'utilisation du CPU observée. Ainsi, pour un microservice *m*, il calcule le nouveau nombre de réplicas avec la formule suivante :

$$desiredReplicas_m = \lceil \frac{\sum_{r \in m}(usage_r)}{desiredValue_m} \rceil \tag{2.1}$$

Dans cette implémentation, l'algorithme ajoute des réplicas si l'utilisation moyenne de CPU est supérieure à la valeur d'utilisation désirée (desiredValue_m). Par exemple, soit un microservice ayant deux réplicas existants et *desiredValue*=0,5 cœur. Si la valeur moyenne d'utilisation de CPU est de 1 c, alors le nouveau nombre de réplicas sera quatre.

Afin de réduire les situations d'instabilité, où le nombre de réplicas fluctue d'un intervalle à l'autre. L'algorithme ne changera la configuration que si l'inéquation suivante est vraie :

$$0.9 < \frac{\sum_{r \in m}(usage_r)}{currentReplicas_m * desiredValue_m} < 1.1 \tag{2.2}$$

De plus, l'algorithme implémente une fenêtre de stabilisation (5 minutes par défaut) qui permet de lisser la baisse du nombre de réplicas. En effet, l'algorithme choisit le nombre de réplicas désiré le plus élevé dans cet intervalle de temps. Dans notre implémentation, cette fenêtre est fixée à une minute.

Il est à noter que HPA de Kubernetes gère des pods et non des conteneurs. Un pod est le plus petit objet géré qui contient un ou plusieurs conteneurs. Toutefois, pour des fins de simplicité, nous considérons qu'un pod est équivalent à un conteneur.

2.2.2 Hybrid-AS

Notre premier algorithme hybride se sert de l'utilisation globale du CPU afin de décider des changements à faire. Tout d'abord, pour chaque microservice m, il calcule la quantité manquante ou excédante de CPU avec la formule suivante :

$$delta_m = \sum_{r \in m}(\frac{usage_r}{target_m} - requested_r) \tag{2.3}$$

Le delta peut s'interpréter comme la différence entre l'utilisation et l'allocation de ressources. Quand l'algorithme termine de calculer les deltas, il procède à la libération de ressource pour

chaque microservice ayant un delta négatif. Ensuite, il commence l'allocation de ressource pour ceux ayant un delta positif.

2.2.2.1 Phase de libération de ressources

Dans cette phase, les microservices traités sont ceux ayant un delta négatif.

L'algorithme 2.1 est le pseudo-code détaillant cette phase :

Algorithme 2.1 algorithme de libération de ressources

Input : microservice m with a negative delta and its replicas sorted by descendant requested values
Output : The microservice with an updated delta

```
1  foreach replica r ∈ m do
2      if ( requested_r + delta_m ) > 0 then
3          requested_r ← max(minRequested_m, requested_r + delta_m ) ;
4          delta_m ← 0 ;
5      else if (currentReplicas_m > Min_Replicas_m) then
6          delta_m ← delta_m + requested_r ;
7          remove(r) ;
8      else
9          delta_m ← delta_m + (requested_r - minRequested_m) ;
10         requested_r ← minRequested_m ;
11     end if
12 end foreach
```

Comme entrée, l'algorithme 2.1 accepte un microservice m et ses réplicas triés par taille descendante. Il consiste à parcourir les instances une par une et décide soit de les rétrécir ou de les supprimer, jusqu'à ce que le delta devient nul. Ainsi, pour chacune d'elles, trois cas de figure peuvent se présenter. Premièrement, si le conteneur peut absorber le delta en entier, alors il continue d'exister et sa valeur $requested_{\mathrm{r}}$ est diminuée (scaling verticale). Deuxièmement, si le delta est plus large que le réplica et le microservice tolère la suppression d'un conteneur. Alors le réplica est éliminé (scaling horizontal) et sa valeur $requested_{\mathrm{r}}$ est absorbée par le delta. Troisièmement, si le delta est plus large que l'instance et le microservice ne tolère plus la

suppression d'un conteneur, alors la valeur $requested_{\mathrm{r}}$ est réduite au minimum autorisé et le delta absorbe la différence.

2.2.2.2 Phase d'allocation de ressources

Dans cette phase, les microservices traités sont ceux avec un delta positif.

L'algorithme 2.2 est le pseudo-code détaillant cette phase :

Algorithme 2.2 algorithme d'acquisition de ressources

Input : microservice m with positive delta and its replicas sorted by ascendent requested values, the worker nodes $\mathcal{N} = \{\mathbf{n}_1, ..., \mathbf{n}_I\}$
Output : The microservice with an updated delta

foreach *replica* $r \in m$ **do**
  $acquired_r \leftarrow$ min(availableCPU($node_r$), $delta_m$, $maxRequested_m$ - $requested_r$) ;
  $requested_r \leftarrow requested_r + acquired_r$;
  $delta_m \leftarrow delta_m$ - $acquired_r$;
end foreach
foreach *node* $n \in N$ **do**
  $acquired_r \leftarrow$ min(availableCPU(n), $delta_m$, $maxRequested_m$);
  if ($acquired_r \geq minRequested_m$ *and* $currentReplicas_m < maxReplicas_m$) **then**
    $delta_m \leftarrow delta_m$ - $acquired_r$;
    createContainer(m, n , $acquired_r$) ;
  end if
end foreach

Comme entrée, l'algorithme 2.2 accepte un microservice m, ses réplicas triés par taille ascendante et les nœuds ouvrier. Il consiste en premier temps à agrandir verticalement les réplicas un par un, jusqu'à ce que le delta devient nul. Si ce dernier demeure positif après la fin du scaling vertical, l'algorithme entame la création de nouvelles instances (scaling horizontal).

2.2.3 Hybrid-P-AS

Notre deuxième algorithme hybride est similaire au premier dans son fonctionnement. En revanche, il utilise des valeurs estimées par le profileur au lieu de l'utilisation observée du CPU. Tout d'abord, pour chaque microservice *m*, il calcule la différence entre les ressources allouées et celles nécessaires pour une meilleure exécution avec les formules suivantes :

$$estimated\_usage_r = estimateCPU(NumberRequests_r, id_r) \tag{2.4}$$

$$estimated\_delta_m = \sum_{r \in m} (\frac{estimated\_usage_r}{target_m} - requested_r) \tag{2.5}$$

Une fois le calcul des deltas terminé, l'algorithme passe à l'étape suivante. Pour chaque microservice ayant un delta négatif, une libération de ressources excédantes est effectuée. Ensuite, il entame l'allocation de ressources pour ceux ayant un delta positif.

Ces deux phases sont exactement similaires à celles du premier algorithme hybride, expliquées par les algorithmes 2.1 et 2.2.

L'idée principale d'utiliser le profileur, vient du fait que les réplicas n'ont pas toujours les ressources nécessaires à disposition. De ce fait, l'acquisition de ressources va se faire sur plusieurs intervalles de temps et une dégradation de la QoS aura lieu sur une longue durée. Ceci est vrai notamment dans les montées de charge de grande amplitude sur une courte période. Nous expliquons en détail le rôle du profileur et la phase de son développement dans la section suivante.

2.3 Développement du profileur

Dans cette section, nous nous intéressons à concevoir le profileur de notre plateforme de test. Tout d'abord, nous expliquons l'idée initiale derrière la conception du profileur. Ensuite, nous

présentons les microservices utilisés et les modèles d'apprentissage implémentés, puis nous expliquons la collecte de données. Enfin, nous évaluons les performances de chaque modèle.

2.3.1 Idée initiale

Les algorithmes de mise à l'échelle gérant le CPU se basent sur son utilisation observée par rapport à «*desiredValue*» pour prendre une décision. En conséquence, si un conteneur consomme plus de processeur que cette valeur désirée, il sera considéré comme manquant de ressources. Ainsi, la quantité de ressources manquante est la différence entre l'utilisation observée et la valeur désirée.

L'utilisation du CPU d'un conteneur est limitée par une valeur maximale «*limit*» ainsi que par la capacité du nœud. De ce fait, les ressources utilisées pourraient être largement inférieures à celles nécessaires pour le bon fonctionnement du microservice. Ce phénomène se produit notamment lorsqu'il y a une augmentation rapide et de grandes amplitudes du besoin en ressources. Ainsi, la mise à l'échelle se fera lentement sur plusieurs intervalles de temps, de sorte qu'une dégradation du QoS peut se produire sur une longue durée.

Par exemple, soit un microservice *m* ayant *limit*=1 c et *desiredValue*=0,5 c. À l'état initial (t=0), il a un seul réplica qui reçoit 10 requêtes/s et consomme 0,5 c en moyenne.
Supposons que dans l'intervalle de temps suivant (t=1), le débit augmente à 100 requêtes/s et le besoin de *m* en CPU devient 5 c. Dans ce cas, *m* ne pourra consommer que 1 cœur car son réplica est plafonné par la limite autorisée. Puisque le besoin en CPU est inconnu de l'autoscaler, il va utiliser la formule 2.1 pour créer un seul réplica. Conséquemment, la création des conteneurs se fera sur plusieurs itérations et *m* ne pourra consommer assez de ressources (5 c) qu'après trois intervalles de temps (à t=4).
Enfin, à t=5, il y aura 10 conteneurs de disponibles, consommant chacun autant que la valeur désirée (0,5 c). Cet état est considéré comme stable puisqu'aucune décision de mise à l'échelle n'est requise.

Le tableau 2.1 illustre l'exemple précèdent en montrant l'évolution du nombre de réplicas jusqu'à l'obtention de la stabilité :

Tableau 2.1 Évolution du nombre de réplicas sans le profileur

t	‖réplicas‖	somme des limites (c)	usage total (c)	besoin total (c)	débit (req/s)
0	1	1	0,5	0,5	10
1	1	1	1	5	100
2	2	2	2	5	100
3	4	4	4	5	100
4	8	8	5	5	100
5	10	10	5	5	100

À partir du tableau 2.1, nous pouvons constater qu'il a fallu attendre trois intervalles de temps pour créer 8 conteneurs et consommer une quantité suffisante de CPU (5 c). De surcroît, la stabilité a été atteinte avec 10 conteneurs qu'après quatre périodes (t=5). D'autre part, si nous incluons le délai de démarrage des réplicas, la création d'un nombre suffisant d'instances se ferait sur une plus longue durée.

Étant donné que le besoin en CPU est inconnu de l'autoscaler, nous pouvons utiliser le profileur afin de lui fournir cette information. En effet, le rôle principal du profileur est d'estimer le besoin en CPU à partir du nombre de requêtes reçues.

Nous pouvons reprendre l'exemple précèdent en intégrant le profileur dans la chaîne de prise de décision. Dans ce cas, à t=1, l'autoscaler sait immédiatement qu'il faut créer 9 nouveaux réplicas pour répondre à l'augmentation de débit de requêtes. Ceci vient du fait que le profileur a été entraîné pour reconnaître que 100 requêtes/s requière 5 cœurs de CPU. En conséquence, à t=2, chaque conteneur consommera autant que *desiredValue* et aucune mise à l'échelle ne sera nécessaire (état stable).

Le tableau 2.2 illustre l'évolution du nombre de réplicas jusqu'à l'obtention de la stabilité en utilisant le profileur :

Tableau 2.2 Évolution du nombre de réplicas avec le profileur

t	‖réplicas‖	somme des limites (c)	usage total (c)	besoin total (c)	débit (req/s)
0	1	1	0,5	0,5	10
1	1	1	1	5	100
2	10	10	5	5	100

À partir du tableau 2.2, nous pouvons constater que *m* a attendu un seul intervalle de temps pour consommer une quantité suffisante de CPU et obtenir de la stabilité. L'avantage du profileur est plus prononcé quand nous prenons en compte le délai de démarrage des conteneurs.

2.3.2 Microservices utilisés

Nos différentes expériences requièrent de déployer différents types de microservices. Pour cette raison, nous avons développé quatre microservices qui ont la tâche de consommer une quantité de CPU pour chaque requête reçue.

Les différents microservices sont développés avec le langage java, en utilisant Spring boot. Ils implémentent une version itérative de la fonction de Lucas avec respectivement les entrées suivantes : 10^2, 10^3, 10^4 et 10^5. Ainsi, pour chaque requête reçue, chaque microservice consommera une quantité différente de CPU.

D'autre part, dans le but de réduire la consommation de mémoire et nous concentrer sur le processeur, nous avons choisi d'implémenter la version itérative de Lucas au lieu de la version récursive.

2.3.3 Modèles d'apprentissage

Dans un contexte d'apprentissage supervisé, nous avons utilisé différents modèles de régression afin de comparer leurs performances. Les différents modèles implémentés sont les suivants :

- **régression linéaire** : ce modèle a pour but de trouver une relation linéaire entre la primitive et l'étiquette ;
- **arbre de régression** : ce modèle appartient aux arbres de décision. Il commence sa décision à partir de la racine, et descend d'un nœud à l'autre jusqu'à l'arrivée à une feuille (résultat) ;
- **random forest (RF)** : ce modèle utilise plusieurs arbres de régression. En effet, il calcule la moyenne des prédictions de tous les arbres afin de prendre sa décision ;
- **gradient boosted trees (GBT)** : ce modèle utilise plusieurs arbres de décision. Il améliore ses prédictions en ajoutant des sous-modèles un à la fois. La technique utilisée est la descente du gradient.

2.3.4 Collecte de données

Dans la démarche de développement du profileur, il nous fallait entraîner nos modèles de régression. Pour ce faire, nous avons collecté les différents données en réutilisant des composants de notre plateforme de test.

La figure 2.3 présente les configurations matérielle et logicielle utilisées lors de la collecte de données.

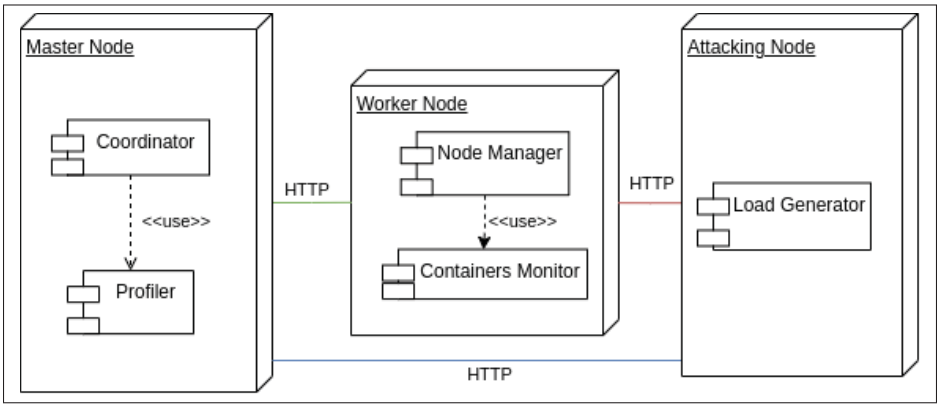

Figure 2.3 Configuration de la collecte de données pour la création du profileur

Concernant les caractéristiques des nœuds utilisés dans cette phase, elles sont présentées dans le tableau 2.3.

Tableau 2.3 Caractéristiques des nœuds

Nom du nœud	CPU	RAM	OS
master node	Intel i7-3630QM	8GB DDR3	Ubuntu 18.04.4
worker node	Intel i7-4770	16GB DDR3	Ubuntu 18.04.4
attacking node	Intel i7-4770	16GB DDR3	Ubuntu 18.04.4

L'idée principale est de déployer sur le nœud ouvrier, un seul conteneur sans lui imposer de limite de CPU. Ensuite, pendant 24 heures, le générateur de requêtes est responsable de le solliciter avec un débit de requêtes différent à chaque intervalle de temps (5 secondes). L'utilisation de CPU et le nombre de requêtes reçues sont enregistrés au niveau du coordinateur à chaque fin d'intervalle de temps. Une fois les 24 heures terminées, le conteneur est supprimé, un autre microservice est déployé et une nouvelle collecte de données commence.

Après avoir obtenu les données pour chacun des quatre microservices, elles sont nettoyées en omettant celles où il y a des paquets perdus. Ensuite, elles sont normalisées puis randomisées. Enfin, elles sont reparties en 3 ensembles :

- **ensemble d'entraînement** : il représente 60% des données et sert à entraîner les différents modèles ;
- **ensemble de validation** : il représente 20% des données et sert à valider les hyperparamètres d'un même modèle ;
- **ensemble de test** : il représente 20% des données et sert à tester la précision finale des modèles afin de les comparer.

2.3.5 Évaluation

Une fois les modèles entraînés et leurs hyperparamètres validés, nous avons évalué chacun d'eux avec l'ensemble de test.

Les métriques de comparaison sont les suivants :

- **mean absolute error (MAE)** : mesure de l'erreur absolue moyenne entre les prédictions et les valeurs réelles, elle est exprimée en cœurs de CPU ;
- **root mean squared error (RMSE)** : mesure l'erreur quadratique moyenne entre les prédictions et les valeurs réelles ;
- **temps** : constitue la durée d'entraînement et de validation des hyperparamètres, exprimée en secondes.

L'erreur quadratique moyenne pénalise davantage les grandes erreurs, c'est pourquoi nous lui avons affectée une plus grande importance lors du choix du meilleur modèle.

Les résultats obtenus sont présentés dans le tableau 2.4 :

Tableau 2.4 MAE et RMSE des différents modèles de régression

modèle	**MAE**	**RMSE**	**temps (s)**
régression linéaire	0,112	0,141	0
Arbre de régression	0,03	0,044	1
Random forest	0,029	0,043	104
Gradient boosted tree	0,028	0,045	1709

Les résultats présentés dans le tableau 2.4 sont les moyennes des performances des quatre microservices. Ainsi, nous avons décidé d'intégrer *Random forest* comme modèle principal du profileur, étant donné qu'il a la plus petite erreur quadratique moyenne. De surcroît, il est plus rapide que GBT dans la phase d'entraînement et de validation des hyperparamètres.

CHAPITRE 3

RÉSULTATS EXPÉRIMENTAUX

Dans ce chapitre, nous détaillons les expériences réalisées sur les algorithmes de mise à l'échelle afin de comparer leurs performances. Tout d'abord, nous présentons l'environnement logiciel et matériel de la collecte de données. Ensuite, nous élaborons une analyse détaillée des résultats.

3.1 Collecte des données

Les différentes expériences sont réalisées sur notre plateforme de test.
La figure 3.1 présente le diagramme de déploiement de la configuration de nos expériences.

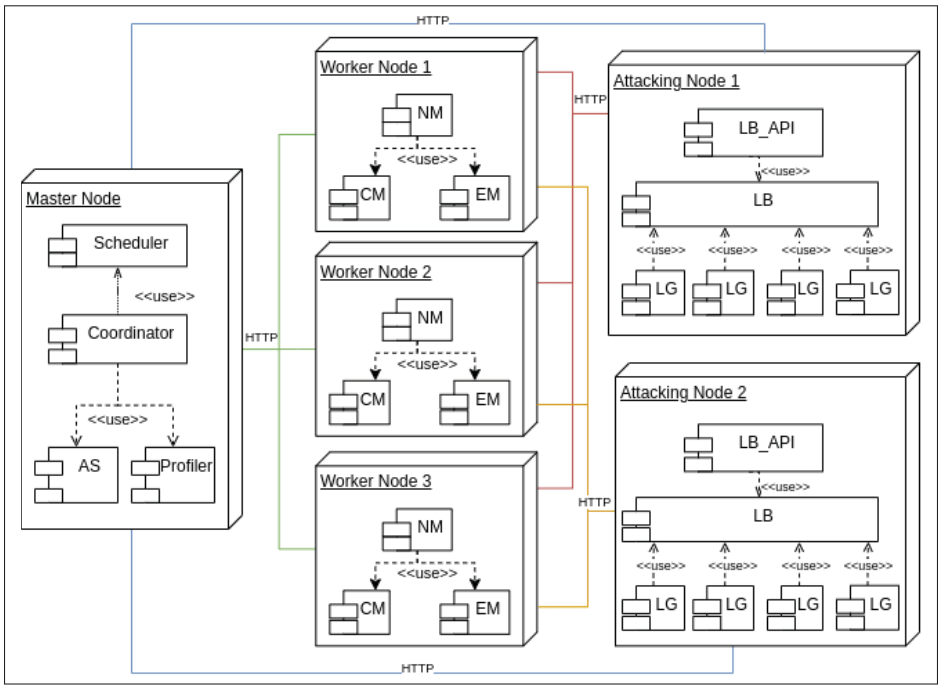

Figure 3.1 Configuration des expériences

Dans cette configuration, il existe deux nœuds d'attaque, chacun contient un répartiteur de charge (LB) et quatre générateurs de requêtes (LG), de sorte que chaque microservice est associé à un LG. D'autre part, nous utilisons trois nœuds ouvriers, chacun contient un gestionnaire de nœud (NM), un moniteur d'energie (EM), un moniteur de conteneurs (CM) et les différents réplicas

des microservices. Enfin, le nœud maître contient le coordinateur, l'ordonnanceur, l'autoscaler (AS) et le profileur entraîné.

Les caractéristiques des nœuds sont présentées dans le tableau 3.1 :

Tableau 3.1 Caractéristiques des nœuds

Nom du nœud	**CPU**	**RAM**	**OS**
master node	Intel i7-3630QM	8GB DDR3	Ubuntu 18.04.4
worker node (1)	Intel i7-4770	16GB DDR3	Ubuntu 18.04.4
worker node (2)	Intel i7-4770	16GB DDR3	Ubuntu 18.04.4
worker node (3)	Intel i7-4770	16GB DDR3	Ubuntu 18.04.4
attacking node (1)	Intel i7-4770	16GB DDR3	Ubuntu 18.04.4
attacking node (2)	Intel i7-3770	16GB DDR3	Ubuntu 18.04.4

Sur la configuration présentée précédemment, nous avons réalisé différents types d'expériences dans lesquelles nous avons testé les trois algorithmes de mise à l'échelle. Chaque expérience concerne un algorithme (Kube-AS, Hybrid-AS et Hybrid-P-AS) et une fonction de génération de requêtes (MGR-1 ou MGR-2). En résumé, nous avons effectué six expériences différentes. Dans chacune d'elles, huit microservices sont sollicités par les générateurs de requêtes avec un débit maximal de 4000 requêtes par seconde.

Les caractéristiques des microservices sont présentées dans le tableau 3.2 :

Tableau 3.2 Caractéristiques et paramétrages des microservices déployés

microservices	**N de Lucas**	**minRequested**	**defaultRequested**	**maxRequested**
m_1 et m_2	10^2	0,250 c	0,50 c	1,0 c
m_3 et m_4	10^3	0,275 c	0,55 c	1,1 c
m_5 et m_6	10^4	0,300 c	0,60 c	1,2 c
m_7 et m_8	10^5	0,325 c	0,65 c	1,3 c

Comme nous avons expliqué dans la section 2.2, les valeurs *minRequested* et *maxRequested* sont utilisées par les algorithmes hybrides. La valeur *defaultRequested* est utilisée seulement par Kube-AS. Concernant la valeur du *target*, elle est mise à 75% et la *limit* est maintenue à deux

fois la valeur du *requested*. Le temps de démarrage d'un conteneur est fixé à 15 secondes. Enfin, chaque expérience dure une heure et est répétée cinq fois afin de moyenner les résultats obtenus.

3.2 Analyse des résultats

Dans cette section, nous présentons les résultats des expériences relatives à la qualité de service, l'utilisation de ressources et enfin la consommation d'énergie.

3.2.1 Latences et disponibilité

Les métriques les plus perçues du point de vue de l'utilisateur sont les latences et la disponibilité. Concernant les latences, le 95e centile est plus significatif que la moyenne, car il présente les cas les plus désagréables. Les métriques traitées sont meilleures quand elles présentent une plus petite valeur. Cela signifie que les algorithmes fournissant assez de ressources aux microservices. De ce fait, de grandes latences impliquent que l'algorithme commet des sous-allocations de ressources.

Les figures 3.2a et 3.2b présentent les moyennes ainsi que les 95es centiles des latences de chaque algorithme :

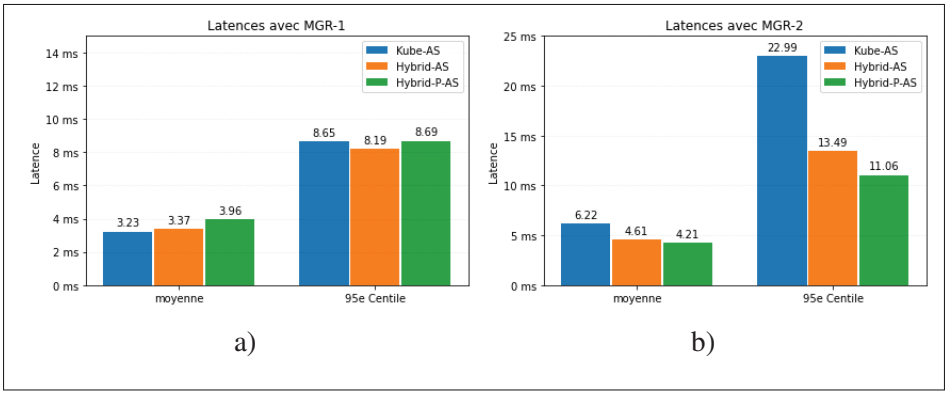

Figure 3.2 Moyenne et 95e centile des latences

Dans l'expérience avec la MGR-1, il n'y a pas une grande différence notable entre Hybrid-AS et Kube-AS. En revanche, Hybrid-P-AS présente une hausse de 23% dans la moyenne de latences par rapport à Kube-AS.

Concernant MGR-2 et par rapport à Kube-AS, Hybrid-P-AS présente les meilleures latences avec une large baisse de 52% sur le 95e centile et 32% sur la moyenne. Quant à Hybrid-AS, il offre une baisse de 41% et 26% respectivement.

Dans un point de vue de la disponibilité, les figures 3.3a et 3.3b présentent les pourcentages de requêtes échouées de chaque algorithme :

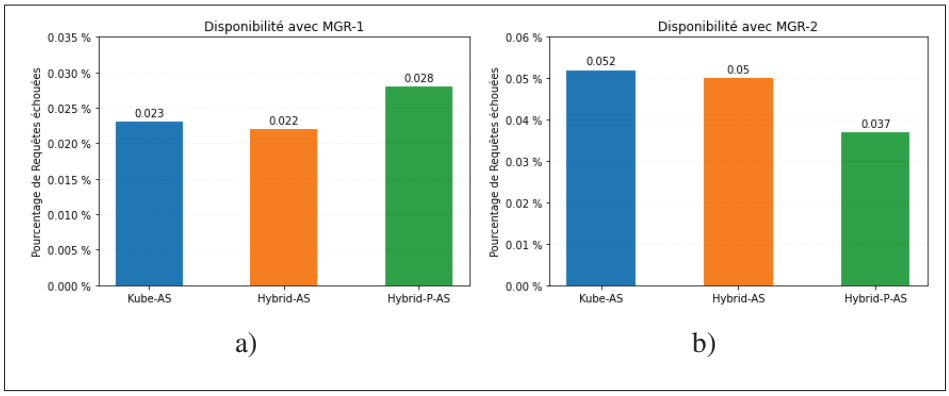

Figure 3.3 Pourcentages des requêtes échouées

Comme présenté dans les figures précédentes, il n'y a pas de grandes différences entre Hybrid-AS et Kube-AS dans les deux situations. En revanche, Hybrid-P-AS présente une augmentation de 22% par rapport à Kube-AS avec MGR-1 et une baisse de 29% avec MGR-2.

3.2.1.1 Discussion

D'après les résultats de latence et de disponibilité présentée ci-dessus, nous pouvons en déduire plusieurs conclusions.

Premièrement, lorsque le débit de requête change graduellement, la réactivité de mise à l'échelle hybride n'a pas d'impact sur les latences et la disponibilité. En revanche, l'utilisation du profileur introduit une dégradation de QoS, qui peut être dû aux erreurs d'estimation du besoin en CPU.

Deuxièmement, lorsqu'il y a des changements brusques dans le débit de requêtes (MGR-2), les algorithmes hybrides offrent de meilleures performances. Cela peut expliquer par la réactivité de la mise à l'échelle verticale. De surcroît, le profileur améliore davantage le QoS, notamment lorsque la limite du conteneur ou la capacité du nœud sont atteintes.

3.2.2 Utilisation des ressources

Dans le but de comparer les performances d'utilisation et d'allocation de ressources, nous avons collecté le nombre de cœurs utilisés et alloués dans chaque nœud. À niveau égal de QoS, l'algorithme le plus performant est celui qui utilise et alloue le moins de ressources.

Les figures 3.4a et 3.4b présentent les moyennes d'utilisation et d'allocation de CPU des nœuds ouvrier :

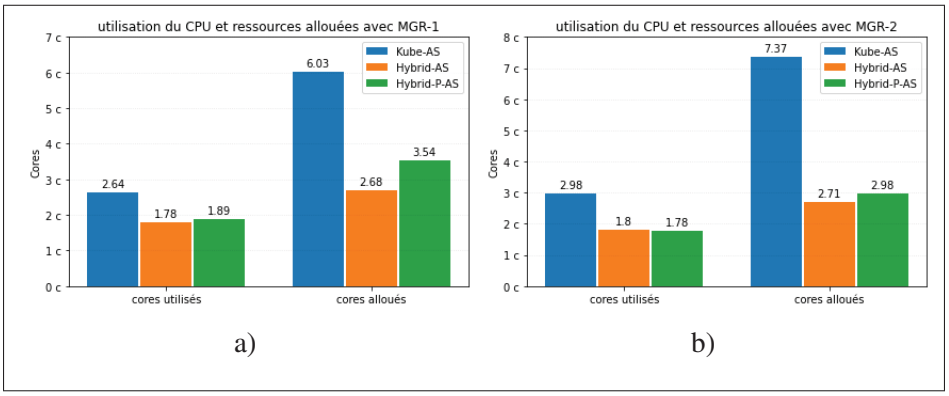

Figure 3.4 Moyennes d'utilisation et d'allocation de CPU

Nous pouvons remarquer qu'avec respectivement MGR-1 et MGR-2, Hybrid-AS montre une baisse de consommation de CPU de 33% et 40% par rapport à Kube-AS. Tandis qu'Hybrid-P-AS offre une baisse de 28% avec MGR-1 et 40% avec MGR-2.

Relativement à l'allocation de ressources, Hybrid-AS alloue moins de 56% et 63% que Kube-AS, avec respectivement MGR-1 et MGR-2. Quant à Hybrid-P-AS, il montre une baisse de 41% et 60%.

Concernant l'efficacité d'allocation de ressources, nous avons calculé la moyenne d'utilisation de CPU par rapport aux cœurs alloués. Étant donné que la valeur de *target* est de 75%, le meilleur algorithme est celui dont les résultats s'en rapprochent le plus.

les figures 3.5a et 3.5b montrent les moyennes d'utilisation par rapport au CPU alloué (nœuds ouvrier seulement) :

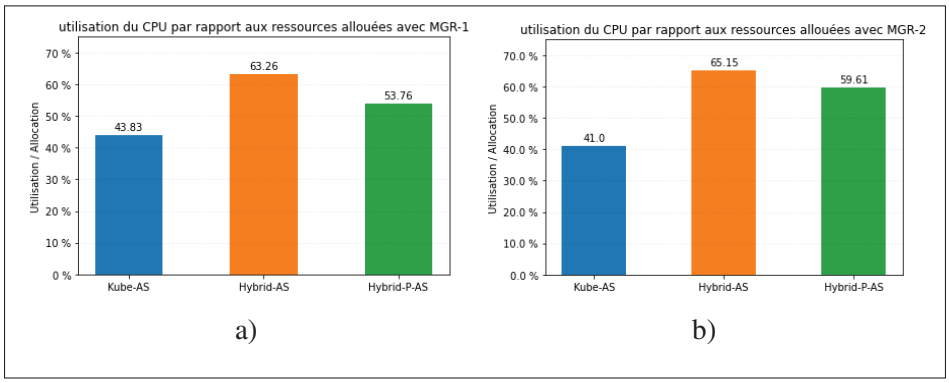

Figure 3.5 Moyenne d'utilisation du CPU par rapport aux ressources allouées

Comme illustré dans les figures 3.5a et 3.5b, Hybrid-AS détient la meilleure efficacité d'utilisation de ressources. En effet, il offre une hausse de 44% avec MGR-1 et 59% avec MGR-2 par rapport à Kube-AS. Quant à Hybrid-P-AS, les hausses d'efficacité sont de 23% et 45% avec MGR-1 et MGR-2 respectivement.

3.2.2.1 Discussion

À partir des résultats présentés, nous pouvons constater que nos algorithmes hybrides offrent de meilleures performances par rapport à Kube-AS.

Premièrement, la différence d'utilisation de CPU peut s'expliquer par le grand nombre de réplicas que déploie Kube-AS, puisque chaque réplica introduit un coût supplémentaire (overhead) d'utilisation de CPU. De plus, la fenêtre de stabilisation oblige Kube-AS à maintenir un grand nombre de réplicas même après une baisse de charges.

Deuxièmement, la différence d'allocation de ressources peut être expliquée par la rapidité et la précision de la mise à l'échelle hybride. En effet, Hybrid-AS et Hybrid-P-AS peuvent ajouter ou enlever rapidement des millicores de CPU aux conteneurs, tandis que Kube-AS est obligé de sur-allouer des ressources la plupart du temps, attendu que le réplica est la plus petite unité gérée. De surcroît, la fenêtre de stabilisation élargit davantage cette sur-allocation.

Troisièmement, les résultats d'efficacité d'utilisation de ressources reflètent l'avantage qu'apporte la mise à l'échelle hybride. En effet, elle diminue considérablement la sur-allocation par rapport à la mise à l'échelle horizontale.

Finalement, par rapport à Hybrid-AS, l'intégration du profileur dégrade légèrement les performances d'utilisation du CPU par rapport aux ressources allouées. Cela peut être dû aux erreurs d'estimation du besoin en CPU qui peuvent être diminuées en utilisant directement la consommation de CPU lorsque celle-ci est en dessous de la valeur «*requested*».

3.2.3 Énergie

La consommation d'énergie est un facteur primordial pour les fournisseurs de service, puisqu'elle se traduit directement par des coûts financiers. Dans le but de comparer les performances de chaque algorithme, nous avons collecté l'estimation de la consommation d'énergie dans chaque expérience. Les nœuds concernés par la mesure sont seulement les nœuds ouvrier où s'exécutent les réplicas des différents microservices.

Les figures 3.6a et 3.6b présentent la somme de consommation d'énergie du CPU (en kilojoules) des trois nœuds ouvriers.

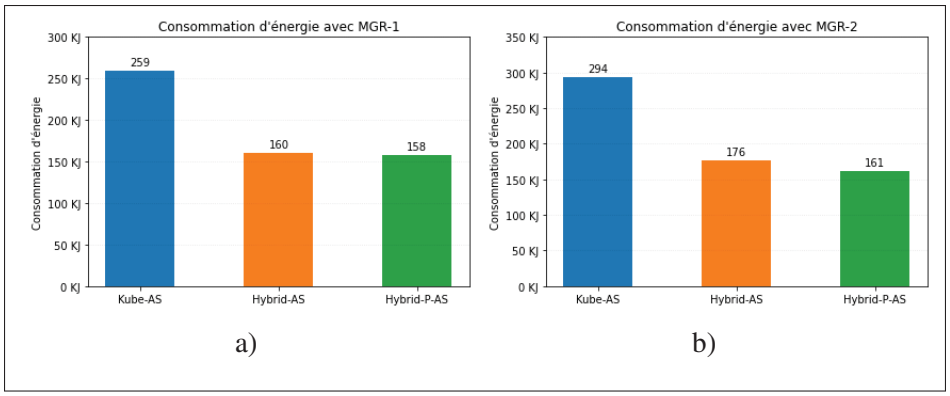

Figure 3.6 Consommation d'énergie totale

Dans les deux situations (MGR-1 et MGR-2), il n'y a pas de grandes différences entre les résultats des algorithmes hybrides. En revanche, par rapport à Kube-AS, Hybrid-AS et Hybrid-P-AS offrent respectivement des baisses de 38% et 39% avec MGR-1. Quant à MGR-2, ils présentent respectivement des gains de 40% et 45%.

3.2.3.1 Discussion

À partir des résultats de consommation d'énergie présentés, nous pouvons constater que nos algorithmes hybrides offrent de meilleures performances par rapport à Kube-AS. Ceci peut s'expliquer par la différence de consommation de CPU constatée dans la section 3.2.3. Malgré que ces données sont des estimations, nous pouvons en conclure que nos algorithmes hybrides consomment moins d'énergie, vu le grand écart entre les résultats.

3.2.4 Synthèse des résultats

Hybrid-AS s'est révélé performant dans la majorité des métriques étudiées et dans les différentes situations. En effet, il améliore considérablement l'utilisation et l'allocation de CPU ainsi que la consommation d'énergie. De plus, il apporte un large gain en termes de latences quand le besoin en ressources évolue brusquement. Tout ceci est dû grâce à la réactivité de la mise à l'échelle hybride et sa précision.

L'intégration du profileur a montré une efficacité dans l'amélioration du QoS, lorsque le besoin en ressource évolue brusquement. Cependant, il montre une légère dégradation des latences lorsque le débit de requêtes évolue graduellement. Ceci peut s'expliquer par les erreurs commises lors d'estimation du besoin en CPU. En conclusion, il est préférable d'activer le profileur que lorsque les ressources des conteneurs sont compressées par la valeur «*limit*» ou par la capacité de la machine. De ce fait, nous pouvons profiter des avantages de ce module tout en évitant des erreurs inutiles d'estimation du besoin en CPU.

CONCLUSION ET RECOMMANDATIONS

Dans le domaine du cloud computing, une meilleure allocation de ressources est de plus en plus recherchée. En effet, dans le cas de sous-allocation de ressources, une dégradation du QoS peut se produire, des violations des SLAs et des pénalités financières en résultent. En contrepartie, une sur-allocation de ressources engendre un coût d'achat de plus de machines et une plus grande consommation d'énergie.

En se basant sur cette constatation et en se focalisant sur les microservices conteneurisés, nous nous sommes proposé de développer une plateforme d'orchestration de conteneurs. Le rôle principal de cette dernière est de tester différents algorithmes de mise à l'échelle et d'ordonnancement. De surcroît, nous avons développé deux algorithmes de mise à l'échelle hybride, capables d'améliorer la QoS, de diminuer la consommation d'énergie et d'augmenter l'efficacité d'allocation du CPU.

À travers le présent document, nous avons détaillé les différentes étapes conduisant à la réalisation de ce travail. Tout d'abord, nous avons présenté le contexte général du projet, la problématique et les limites. Ensuite, nous avons consacré le premier chapitre à l'état de l'art, où nous avons présenté les concepts de base et la revue de la littérature. Dans le deuxième chapitre, nous avons détaillé les spécifications et l'architecture de notre plateforme d'orchestration, puis nous avons expliqué les algorithmes implémentés et élaborer le développement du profileur. Dans le troisième chapitre, nous avons présenté l'environnement logiciel et matériel utilisé, suivi d'une analyse des données obtenues lors des expériences. L'analyse effectuée comprend une comparaison des résultats de nos algorithmes avec ceux de l'algorithme horizontal de Kubernetes. Dans cette étape nous avons montré la supériorité des performances de nos algorithmes hybrides, notamment dans l'amélioration de la qualité de service, l'utilisation et l'allocation du CPU, ainsi qu'au niveau de la consommation d'énergie.

En termes de perspectives, notre plateforme peut être aisément améliorée afin qu'elle héberge des microservices en phase de production. Concernant nos algorithmes de mise à l'échelle, nous prévoyons de les améliorer afin d'inclure plus de métriques telles que la mémoire. Enfin, nous recommandant l'amélioration de Hybrid-P-AS afin d'utiliser le profileur qu'en cas de besoin, ce qui aura probablement un effet positif sur les performances.

BIBLIOGRAPHIE

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. & Merle, P. (2017). Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pp. 472-479.

Apache CloudStack. (2020). Configuring AutoScale without using NetScaler. Repéré le 2021-02-15 à https://docs.cloudstack.apache.org/projects/archived-cloudstack-getting-started/en/4.3/networking/autoscale_without_netscaler.html.

AWS. (s.d.). Amazon EC2 Auto Scaling. Repéré le 2020-12-25 à https://aws.amazon.com/ec2/autoscaling.

Baresi, L. & Quattrocchi, G. (2020). COCOS : A Scalable Architecture for Containerized Heterogeneous Systems. *IEEE International Conference on Software Architecture*, pp. 103-113.

Docker. (2017). Runtime options with Memory, CPUs, and GPUs. Repéré le 2021-01-02 à https://docs.docker.com/config/containers/resource_constraints.

Docker. (s.d.). Swarm mode overview. Repéré le 2020-09-02 à https://docs.docker.com/engine/swarm/.

Dutta, S., Gera, S., Verma, A. & Viswanathan, B. (2012). SmartScale : Automatic Application Scaling in Enterprise Clouds. *2012 IEEE Fifth International Conference on Cloud Computing*, pp. 221-228.

Felter, W., Ferreira, A., Rajamony, R. & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171-172.

Fernandez, H., Pierre, G. & Kielmann, T. (2014). Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. *2014 IEEE International Conference on Cloud Engineering*, 8, 195-204.

Ferreira, J., Cello, M. & Iglesias, J. (2017, 08). More Sharing, More Benefits ? A Study of Library Sharing in Container-Based Infrastructures. pp. 358-371.

Google Cloud. (2020). Using autoscaling for highly scalable applications. Repéré le 2021-02-02 à https://cloud.google.com/compute/docs/tutorials/high-scalability-autoscaling.

Han, R., Guo, L., Ghanem, M. M. & Guo, Y. (2012). Lightweight Resource Scaling for Cloud Applications. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pp. 644-651.

Han, R., Ghanem, M. M., Guo, L., Guo, Y. & Osmond, M. (2014). Enabling Cost-Aware and Adaptive Elasticity of Multi-Tier Cloud Applications. *Future Gener. Comput. Syst.*, 32(C), 82-98.

Hoenisch, P., Weber, I., Schulte, S., Zhu, L. & Fekete, A. (2015). Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers. Dans Barros, A., Grigori, D., Narendra, N. & Dam, H. (Éds.), *International Conference on Service-Oriented Computing* (pp. 316-323). Goa, India : Springer.

Huber, N., Brosig, F. & Kounev, S. (2011). Model-Based Self-Adaptive Resource Allocation in Virtualized Environments. *6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 90–99.

IBM Cloud Education. (2019). What are Microservices ? Repéré le 2021-01-12 à https://www.ibm.com/cloud/learn/microservices.

Kan, C. (2016). DoCloud : An elastic cloud platform for Web applications based on Docker. *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pp. 478-483.

Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K. & Ou, Z. (2018). RAPL in Action : Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), 1-26.

Kubernetes. (2020a). Horizontal Pod Autoscaler. Repéré le 2021-01-03 à https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale.

Kubernetes. (2020b). Vertical Pod Autoscaler. Repéré le 2021-01-02 à https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler.

Kubernetes. (2020c). Managing Resources for Containers. Repéré le 2021-02-25 à https://kubernetes.io/docs/concepts/configuration/manage-resources-containers.

Kwan, A., Wong, J., Jacobsen, H. & Muthusamy, V. (2019). HyScale : Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 80-90.

Lu, L., Zhu, X., Griffith, R., Padala, P., Parikh, A., Shah, P. & Smirni, E. (2014). Application-driven dynamic vertical scaling of virtual machines in resource pools. *IEEE Symposium on Network Operations and Management*, pp. 1-9.

Luksa, M. (2018). *Kubernetes in Action* (éd. 1). New York, États-Unis : Manning.

Microsoft. (2020). Overview of autoscale with Azure virtual machine scale sets. Repéré le 2021-01-14 à https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview.

Moltó, G., Caballer, M. & de Alfonso, C. (2016). Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems*, 56(C), 1-10.

Potdar, A. M., D G, N., Kengond, S. & Mulla, M. M. (2020). Performance Evaluation of Docker Container and Virtual Machine. *Procedia Computer Science*, 171, 1419-1428.

Rajamani, K., Felter, W., Ferreira, A. & Rubio, J. (2015, November). *Spyre : A Resource Management Framework for Container-Based Clouds*. communication présentée à USENIX Container Management Summit, Washington, D.C. Repéré à https://www.usenix.org/conference/ucms15/summit-program/presentation/rajamani.

Red Hat. (2020). What's a Linux container ? Repéré le 2021-01-13 à https://www.redhat.com/en/topics/containers/whats-a-linux-container.

The Linux Kernel. (2020). Power Capping Framework. Repéré le 2021-01-02 à https://www.kernel.org/doc/html/latest/power/powercap/powercap.html.

Ueki, K. & Kourai, K. (2020). Fine-grained Autoscaling with In-VM Containers and VM Introspection. *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pp. 155-164.

Vigneshwaran, L. (2016). *Preservation of low latency service request processing in dockerized microservice architectures*. (Mémoire de maîtrise, Colorado State University, Fort Collins, Colorado). Repéré à http://hdl.handle.net/10217/178889.

Yang, J., Liu, C., Shang, Y., Cheng, B., Mao, Z., Liu, C., Niu, L., & Chen, J. (2014). A cost-aware auto-scaling approach using the workload prediction in service clouds. *Information Systems Frontiers*, 16(1), 7–18.

Ye, T., Guangtao, X., Shiyou, Q. & Minglu, L. (2017). An Auto-Scaling Framework for Containerized Elastic Applications. *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, pp. 422-430.