République Tunisienne
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université de Tunis
École Nationale Supérieure d'Ingénieurs de Tunis
Tunisie

Réf :ING-GInfo-2017-23

Rapport de
Projet de Fin d'étude

Pour obtenir le
Diplôme d'Ingénieur en Génie Informatique

Option : Systèmes, Réseaux et Sécurité (SRS)

Présenté par :

Wiem BEN ABDALLAH

Ordonnancement des conteneurs

Du 15 février au 15 juillet 2017

Organisme d'accueil :
Laboratoire d'Informatique de Paris Nord (LIPN)
Institut Galilée, Université Paris 13

Encadré par :
M. Christophe CERIN
Professeur à Paris 13
M. Walid SAAD
Assistant en informatique à l'ENSIT

Année universitaire : 2016-2017

Abstract

In this report, we present a new scheduling strategy in Docker Swarm. Our goal is to resolve the problem of companies which have a private infrastructure of machines, and would like to optimize the scheduling of several requests submitted online by their users. Currently, Swarm has three scheduling strategies : spread, binpack and random. Our contribution is to add a fourth strategy of placement based on the notion of priority and service level agreement (SLA) classes. For each container, we calculate dynamically the number of computing resources which must be used according to an economic model and the load of machines in the infrastructure. We also present our approach concerning the implementation until the unit tests.

Keys words : scheduling strategies, Docker container and Swarm scheduler.

Résumé

Dans ce rapport, nous présentons une nouvelle stratégie d'ordonnancement dans Docker Swarm. Notre objectif est de résoudre le problème des entreprises qui possèdent une infrastructure privée de machines et souhaitant optimiser l'ordonnancement des demandes soumises en ligne par leurs utilisateurs. Actuellement, Docker Swarm a trois stratégies d'ordonnancement : spread, binpack et aléatoire. Notre contribution est d'ajouter une quatrième stratégie tout en introduisant la notion de priorité et les classes d'accord de niveau de service (SLA). Pour chaque conteneur, nous calculons dynamiquement le nombre de ressources informatiques qui doivent être utilisées selon un modèle économique et à la charge des machines dans l'infrastructure. Nous présentons également notre démarche qui va jusqu'aux tests unitaires.

Mots clés : stratégies d'ordonnancement, conteneur Docker et ordonnanceur Swarm.

ملخص

في هذَا التقرير، نقدم استرَاتيجية جدولة جديدة هدفنَا هو تقديم مفهوم الَأولوية وفءة من اتفَاقية مستوَى الخدمة لحل مشكلة الشركَات التي لديهَا بنية تحتية خَاصة وترغب في تحسين جدولة الطلبَات المبعوثة عبر الَأنترنيت من طرف مستخدميهَا. الجديد في تلك التي قدمنَاهَا هوَ الحسَاب الحيوي لّموَارد التي سيقع استعمَالهَا لتخفيض الطَاقة المستعملة وحسب طَاقة استيعَاب البنية التحتية.

الكلمَات المفَاتيح : استرَاتيجية، جدولة وَالحَاوية.

Remerciements

Ce travail est l'aboutissement d'un long cheminement au cours duquel j'ai bénéficié de l'encadrement et des encouragements de plusieurs personnes, à qui je tiens à dire profondément et sincèrement merci.

A cet égard, je tiens à exprimer ma profonde gratitude à mon tuteur de stage au sein du laboratoire Monsieur Christophe Cérin pour son accueil chaleureux, son suivi, son aide gracieuse, sa disponibilité et surtout l'ambiance agréable de travail qu'il m'avait fournie durant la période de mon stage.

J'adresse aussi mes remerciements les plus vifs à mon encadrant académique Monsieur Walid Saad qui m'a beaucoup aidé pour trouver ce stage. Je le remercie pour sa confiance, son orientation et sa patience qui ont constitué un apport considérable sans lequel ce travail n'aurait pas pu être mené au bon port.

Un grand merci s'adresse aux deux membres de l'équipe AOC dont je suis membre : Tarek Menouer et Leila Abidi pour leur aide et leurs encouragements. Je ne manquerai pas de remercier notre chef de département informatique Madame Meriem Riahi pour son assistance et ses encouragements.

Je tiens à saisir cette occasion et adresser mes profondes reconnaissances à mes parents, mes frères et ma sœur qui par leurs prières et leurs encouragements, j'ai pu surmonter tous les obstacles en espérant que j'ai répondu aux espoirs qu'ils ont fondés en moi.

Enfin, je remercie les membres de jury d'avoir accepté d'évaluer mon travail.

Table des matières

Table des figures

Liste des tableaux

Introduction générale

Le présent travail s'inscrit dans le cadre de la réalisation du projet de fin d'études des élèves ingénieurs à l'École Nationale Supérieure d'Ingénieurs de Tunis (ENSIT). Le stage scientifique de cinq mois débutant le 15 février 2017 et finissant le 15 juillet 2017 s'est déroulé au sein du Laboratoire d'Informatique de Paris Nord (LIPN) à l'institut Galilée qui fait partie du campus universitaire Paris 13.

Le stage intitulé « ordonnancement des conteneurs » est en raisonnance avec le projet Wolphin 2.0 [44] démarré en janvier 2017. Il cherche à permettre les industriels, fournisseurs de l'informatique en nuage, de proposer une facture exacte à leurs clients. Son objectif majeur est de fournir une solution d'hébergement open source conçue pour permettre la facturation précise de la consommation des ressources par le service demandé.

L'objectif de notre travail est de proposer une nouvelle stratégie de placement de conteneurs dans l'ordonnanceur Docker Swarm. Ceci en introduisant, dans le contexte conteneur, la notion de priorité et des classes d'accord de niveau de service (SLA).

Le rapport, qui récapitule le travail effectué, se compose de quatre chapitres qui sont :

- Chapitre 1 : nous présenterons dans ce chapitre le cadre général du projet, son contexte et ses objectifs.
- Chapitre 2 : nous y présenterons notre lecture bibliographique sur les machines virtuelles en ce qui concerne leur placement, redimensionnement et migration en cas de besoin.
- Chapitre 3 : ce chapitre donne une vision sur les notions nécessaires pour comprendre le travail effectué. Nous y présenterons notre lecture bibliographique sur les ordonnanceurs des conteneurs.
- Chapitre 4 : nous détaillerons dans ce chapitre nos contributions tout en introduisant les motivations et les démarches suivies.

Chapitre 1

Cadre général du projet

Introduction

Notre stage s'est déroulé au sein du Laboratoire d'Informatique de Paris Nord (LIPN [1]). Dans ce chapitre, nous faisons une brève description de LIPN. De plus, nous présentons l'informatique en nuage, le conteneur et la machine virtuelle. Après, nous mettons le stage dans son contexte et nous donnons ses objectifs.

1.1 Présentation du LIPN

Ce laboratoire est associé au CNRS depuis janvier 1992 et a le statut d'unité mixte de recherche depuis janvier 2001 (UMR 7030). Les recherches dans LIPN sont centrées autour de trois axes : l'automatisation du raisonnement, de l'informatique fondamentale et l'intelligence artificielle. Des tels axes s'arment des compétences en algorithmique, logique, langage naturel et apprentissage artificiel. Le laboratoire se compose de cinq équipes dont les thèmes convergent dans plusieurs domaines de la recherche. Ces dernières sont :

- Apprentissage Artificiel et Applications (A^3) : le travail de cette équipe est concentré sur les problèmes d'apprentissage artificiel numérique et symbolique. Elle utilise des méthodes supervisées, non supervisées ou hybrides. Ses recherches sont autour de trois axes qui sont :
  - Modèles logiques et algébriques de l'apprentissage.
  - Apprentissage non supervisé collaboratif et évolutif.
  - Apprentissage de structures à partir de données hétérogènes.

1. http ://lipn.univ-paris13.fr/fr/

- Algorithmes et Optimisation Combinatoire (AOC) : cette équipe a des compétences en optimisation combinatoire, en algorithmes et en logiciels et architectures distribués. Le travail de l'équipe AOC dont nous sommes membre est centré sur trois axes de recherche qui sont optimisations des graphes, programmation mathématique et algorithme et logiciels et architectures distribuées.
- Combinatoire, Algorithmique et Interactions (CALIN) : l'équipe CALIN a des compétences en différents aspects de la combinatoire. Elle étudie la complexité des algorithmes en privilégiant le comportement selon la distribution des données.
- Logique, Calcul et Raisonnement (LCR) : le travail de cette équipe est centré autour de deux axes qui sont :
  - Logique linéaire et diverses applications en informatique.
  - Spécifications des systèmes et l'aide à la modélisation avec des applications aux systèmes dynamiques, distribués et aux bases de données.
- Représentation des Connaissances et Langage Naturel (RCLN) : cette équipe s'intéresse à la langue et la représentation des connaissances en tant qu'outil mis au service pour le traitement du langage naturel. Ses recherches sont centrés autour de quatre axes :
  - Analyse et annaotation sémantique de corpus.
  - Découverte et structuration de connaissances pour le web sémantique.
  - Recherche d'information sémantique.
  - Intégration syntaxe-discours.

1.2 Informatique en nuage

C'est une technique qui permet de gérer des ressources et d'adapter très rapidement une infrastructure à des variations de charge de manière transparente pour l'administrateur et l'utilisateur [19]. On trouve trois types de service offerts par l'informatique en nuage qui sont :

- SaaS (Software as a Service) : permet d'utiliser l'application à distance qui est hébergée par l'éditeur.
- PaaS (Platform as a Service) : dispose d'environnements spécialisés au développement comprenant les langages de programmation, les outils et les modules nécessaires.
- IaaS (Infrastructure as a Service) : ce service permet la mise à la disposition, à la demande, des ressources d'infrastructures dont la plus grande partie est localisée à distance dans des centre de données. L'utilisateur peut louer des grappes, de la mémoire ou du stockage de données. Le coût est lié au taux d'occupation.

Dans les centres de données traditionnels, les applications sont liées à des serveurs physiques spécifiques qui sont souvent sur-provisionnés pour faire face à la charge de travail supérieure. Cette configuration rend les centres de données coûteux à entretenir.

Grâce à la technologie de virtualisation, les centres de données de l'informatique en nuage deviennent plus flexibles, sécurisés et fournissent un meilleur soutien pour l'allocation à la demande. Elle masque l'hétérogénéité et la consolidation du serveur et améliore l'utilisation de ce dernier. Un serveur devient capable d'héberger plusieurs machines virtuelles possédant des spécifications de ressources différentes et des types de charges de travail variables [45].

1.3 Conteneur

On peut définir les conteneurs comme un ensemble de processus visant à offrir des outils de gestion de ressources tout en assurant leur isolement au niveau du noyau et empêchant l'interférence entre eux.

Un conteneur contient une image plus les fichiers qui composent l'application qui va être exécutée à l'intérieur. Son principe de base est l'empaquetage qui rend le déploiement des applications plus facile. De plus, il consomme une quantité limitée de ressources afin de satisfaire les demandes des clients. Souvent, il est utilisé dans les micro-services et il peut être connecté à d'autres conteneurs ou services via le réseau.

Parmi les exemples de conteneurs qu'on peut citer : le conteneur Linux LXC [20]. Grâce aux conteneurs Linux ou bien la technologie Docker, le code d'application conteneurisée est totalement isolé du SE (Système d'Exploitation) de la machine hôte et des autres conteneurs, ce qui donne au développeur plus de souplesse de développement et déploiement au niveau de l'application [38].

1.4 Machines virtuelles vs conteneurs

Avant d'entamer la comparaison entre la machine virtuelle et le conteneur, il faut tout d'abord comprendre le mode de fonctionnement de chacune de ces notions, prises séparément. La machine virtuelle représente un ordinateur complet contenant tout ce qui justifie cette imitation : un système d'exploitation(SE) complet, des pilotes, des systèmes de fichiers binaires ou bibliothèques ainsi que l'application elle-même.

Pour son exécution, une machine virtuelle nécessite la présence d'un hyperviseur qui s'exécute à son tour sur un système d'exploitation hôte. Ce dernier est le responsable du fonctionnement du matériel du serveur physique. Par contre, le conteneur contenant notamment une application est libéré d'un SE. Il fonctionne comme une

série de couches : une image de base composée d'un SE puis une application [24].

La Figure 1.1 [24] représente une comparaison d'architecture entre ces deux mécanismes.

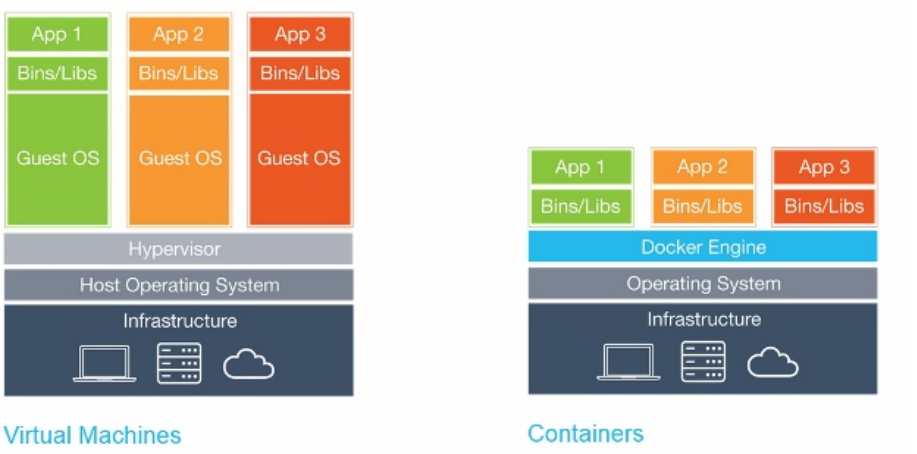

FIGURE 1.1 – Machines virtuelles vs conteneurs : comparaison d'architecture

Dans ce qui suit, le tableau 1.1 présente une comparaison entre une machine virtuelle et un conteneur [12, 24].

Machine virtuelle(VM)	Conteneur (isolateur)
-Une VM s'exécute dans un environnement isolé de virtualisation matérielle grâce à un hyperviseur.	-Ne nécessite pas d'hyperviseur pour assurer son isolation et n'y a pas recours.
-Tous les éléments exécutés dans une VM sont masqués du SE hôte.	-Il utilise les fonctionnalités d'isolation des processus et du système de fichier du noyau Linux pour exposer sur le conteneur certaines fonctionnalités d'un noyau uniquement et son propre système de fichiers isolés.
-Une VM apparaît comme un ordinateur physique autonome.	-Le conteneur paraît comme une instance unique du système d'exploitation.
-Très lourde lors de démarrage.	-L'espace disque nécessaire du conteneur ne contient pas un SE dans son intégralité.
	-Son temps de démarrage du conteneur et la surcharge d'espace disque requis sont réduits.

Table 1.1 – Machine virtuelle vs conteneur

Ce tableau montre les points de différences entre une machine virtuelle (VM) et un conteneur. En effet, il montre que le conteneur est très léger et consomme moins de ressources par rapport à une VM. C'est pour cette raison, il est très utilisé dans les infrastructures micro-services.

1.5 Contexte et problématique

L'informatique en nuage est essentiellement une offre commerciale d'abonnement à des services externes. Son principe de base est la facturation à l'usage qui touche différents éléments comme l'application demandée, la capacité de stockage de données, la capacité de traitement de mémoire et le nombre d'utilisateurs. Mais, les solutions d'hébergement actuelles ne permettent pas aux industriels qui sont des fournisseurs de cette technologie de proposer à leurs clients une facture exacte et précise vis-à-vis de la consommation des ressources par le service demandé.

En d'autres termes, les hébergeurs vendent à leurs clients des ressources dépassant leurs besoins, freinant ceux-ci à adopter les offres IaaS (Infrastructure as a service) classiques représentant l'un des services fournis par l'informatique en nuage. L'origine de ce problème revient aux offres forfaitaires que les hébergeurs proposent à leurs clients provoquant par la suite un gaspillage de la mémoire, de l'énergie et des ressources de stockage.

Suite à ce problème, le projet open source Wolphin 2.0, financé par Fonds Unique Ministériel (FUM), a démarré en janvier 2017. Son objectif est de fournir une solution performante d'hypervision et de facturation d'infrastructures micro-services. Ce tye d'infrastructure constitue aujourd'hui une méthode de développement. Son but est de concevoir une application sous la forme d'une série de services modulaires. Chaque module s'occupe d'une fonction spécifique en utilisant une interface simple et bien définie afin de faciliter la communication avec d'autres modules.

L'équipe AOC (Algorithmes et Optimisation Combinatoire) du laboratoire LIPN est membre de ce projet avec d'autres partenaires qui sont [3] :

- AlterWay (coordinateur) [2] ;
- Objectif Libre ;
- Laboratoire d'Informatique de Paris 6 (LIP6), université Pierre et Marie Curie ;
- Gandi.

Les objectifs du projet Wolphin 2.0 sont [44] :

- Facturer en fonction de la consommation des ressources. C'est à dire de chercher à atteindre l'efficacité commerciale.
- Éviter le gaspillage, ne consommer que ce qui est nécessaire et profitable aux applications afin de réaliser l'efficacité énergétique.
- Faciliter l'introduction des usagers à la technologie d'hébergement des conteneurs pour rendre ce marché Open Source.

Ce projet doit répondre aux cas d'utilisation suivants vis à vis des services déployés :

- ***Service long*** : c'est un service persistant connaissant des pics de charge (cpu, ram, disk, réseau) comme des périodes d'inactivité.
  Exemple de site métier : site e-commerce.
  À ce niveau de service, le projet cherche à atteindre deux objectifs qui sont :
  - Objectif 1 : en cas de pics de charge (global ou par ressource), optimiser le placement du service afin d'assurer sa disponibilité.
  - Objectif 2 : en cas d'inactivité de service, optimiser la consommation des serveurs (machines virtuelles ou physiques) en déplaçant / arrêtant le service, et en éteignant / décomissionnant le serveur.

2. https ://www.alterway.fr/

- ***Service court*** : il représente un service provoquant une surcharge (cpu, ram, disk, réseau) ou en attente d'un service distant.
  Exemple : conversion en pdf.
  Pour ce cas d'utilisation, il faut réussir à quantifier sa consommation fine.

- ***Micro-service*** : c'est le service dont la durée de vie est inférieure à la fréquence de collecte de métriques.
  Exemple : la collecte s'effectue toutes les minutes et la durée de vie du conteneur était de quelques millisecondes.
  Le projet Wolphin veut réussir à présenter ce service éphémère et un indice de consommation à la facturation.

Le travail effectué dans le cadre de ce stage fait partie du projet Wolphin 2.0 en ce concentrant sur la partie de la gestion des conteneurs. Le but de cette gestion est de démarrer les conteneurs sur l'hôte le plus approprié et être capable d'équilibrer les conteneurs sur l'ensemble de l'infrastructure matérielle lorsqu'il y a trop de données à calculer ou à traiter sur un seul nœud physique.

1.6 Objectifs de stage

D'une manière générale, le placement des conteneurs change les problématiques par rapport au placement des machines virtuelles sur les deux plans :

- La combinatoire est plus grande : on peut faire tourner facilement 20 à 50 fois plus de conteneurs sur un nœud par rapport aux machines virtuelles.

- La dépendance des services à placer : les conteneurs liés partagent les données d'une même application.

Les liens que nous pouvons faire dès à présent entre le projet Wolphin et l'expertise du laboratoire LIPN sont les suivants. Nous devons proposer des stratégies de placement des tâches (c.f. compétences du LIPN en ordonnancement, calcul haute performance), en contexte conteneur (compétences Docker chez AlterWay), tout en veillant à des bonnes performances vis à vis de la mémoire. C'est à dire de proposer une stratégie qui aide à la décision de placement de groupes de conteneurs liés.

Comme première étape, nous avons proposé un algorithme générique qui offre un nouveau modèle d'ordonnancement des requêtes soumises en ligne. Le but de cet algorithme est de satisfaire le maximum possible des demandes soumises en réduisant la consommation énergétique de l'infrastructure. La deuxième étape consiste à adapter ce système dans le contexte de conteneurs Docker. C'est à dire, nous devons l'implémenter dans l'ordonnanceur de conteneurs Docker Swarm sans changer ni son architecture ni son principe de fonctionnement.

Conclusion

Ce chapitre a été réservé à la mise du travail dans son cadre général, à la présentation du projet Wolphin dont le stage fait partie ainsi qu'à la présentation des objectifs de ce dernier. Le chapitre qui suit présente une étude de l'art des principales thématiques traitées par notre travail.

Chapitre 2

Placement, redimensionnement et migration des machines virtuelles

Introduction

Nous présentons dans ce chapitre un état de l'art sur les machines virtuelles : leur placement, leur dimensionnment, les techniques de consolidation pour minimiser leur consommation énergétique et leur migration en cas de détection d'une surcharge.

2.1 Placement des machines virtuelles

2.1.1 Présentation du problème

Dans les centres de données, la sélection de la machine virtuelle à placer et la machine physique la plus appropriée à l'exécuter représente un problème un peu délicat. Plusieurs études scientifiques ont été faites dont leurs objectifs majeurs étaient [32] :

- Développement d'algorithmes rapides qui résolvent le problème de placement des machines virtuelles dans un temps le plus court possible.
- Optimisation énergétique des topologies de réseaux virtuels entre les machines virtuelles pour un placement optimal afin de réduire l'énergie consommée par l'infrastructure du réseau.
- Développement de nouveaux algorithmes de gestion thermique pour contrôler la température et la consommation d'énergie.
- Décentralisation et approches distribuées pour l'évolutivité.

2.1.2 Approches d'optimisation

Ils existent trois approches d'optimisation pour le problème de placement des machines virtuelles [32] :

1. Approche mono-objectif : elle considère l'optimisation d'un seul objectif ou l'optimisation individuelle d'une fonction objective au plus. La majorité des articles étudiés proposent ce type d'approche pour trouver une solution au problème de placement des machines virtuelles.

2. Approche multi-objectifs résolue comme approche mono-objectif : cette approche consiste à combiner toutes les fonctions objectives en une seule fonction objective c'est pour ce qui justifie son appellation. Mais l'inconvénient de cette approche est l'exigence de la connaissance du domaine du problème afin d'allouer la combinaison correcte des fonctions objectives, ce qui n'est pas toujours possible.

3. Approche multi-objectifs pure : le problème général d'optimisation multi-objectifs pure inclut un nombre p de variables de décision, q fonctions objectives et r contraintes. Les fonctions objectives et les contraintes sont des fonctions de variables de décision.

2.1.3 Solutions techniques

Pour résoudre ce problème, une grande variété des techniques ont été proposées. En totalité, dans les lectures que nous avons effectuées il y avaient quatre solutions qui sont :

- Algorithme déterministe : un algorithme qui, pour une entrée particulière, produira toujours la même sortie, avec la machine sous-jacente passant toujours par la même séquence d'états.

- Heuristique : une méthode de calcul qui fournit rapidement une solution réalisable qui n'est pas nécessairement optimale, pour un problème d'optimisation NP-difficile.

- Méta-heuristique : est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficiles pour lesquels on ne connaît pas de méthode classique plus efficace.

- Algorithme d'optimisation : cherche à déterminer le jeu de paramètres d'entrée d'une fonction donnant à cette fonction la valeur maximale ou minimale.

2.2 Dimensionnement efficace des machines virtuelles

2.2.1 Présentation de problème

En plus des services traditionnels offerts par l'informatique en nuage qui sont SaaS (Software as a Service), PaaS (Platform as a Service) et l'IaaS (Infrastructure as a Service), un nouveau service vient d'être ajouté à cette nomenclature qui est le CaaS (Container as a Service). Le problème avec ce nouveau service c'est que l'utilisateur surestime les ressources requises pour le déploiement de son service.

Ce qui en résulte l'utilisation inefficace de l'infrastructure de centre de données. Donc, il faut trouver le dimensionnement efficace des machines virtuelles pour héberger les conteneurs de telle sorte que la charge de travail est exécutée avec un minimum de gaspillage d'énergie [31].

2.2.2 Approche proposée

La technique qui a été proposée pour résoudre le problème cité précédemment se compose de deux phases. La première, c'est la phase de pré-exécution dans laquelle le système doit déterminer la taille des machines virtuelles en fonction des informations historiques sur l'utilisation des ressources des conteneurs. La deuxième est la phase d'exécution. Durant cette phase, chaque conteneur va être affecté à la machine virtuelle avec un type spécifique [31]. Pour cette approche, une tâche est définie grâce à ces variables :

- Longueur de la tâche : c'est le temps pendant lequel la tâche était exécutée sur une machine.
- Taux de soumission : le nombre de fois qu'une tâche est soumise au centre de données.
- Classe d'ordonnancement : le degré de sensibilité de la tâche/job à la latence. Elle est représentée par un entier compris entre 0 et 3. Plus la classe est élevée plus la tâche est sensible à la latence.
- Priorité : elle montre l'importance d'une tâche. C'est un entier dans la plage de 0 à 10.
- Utilisation des ressources : l'utilisation moyenne des ressources en terme de CPU, de mémoire et de disque pendant la période observée.

Afin d'attribuer un type de machine virtuelle qui peut exécuter les tâches qui appartiennent au cluster, un nouveau paramètre a été ajouté nommé "capacité de la tâche". Ce dernier représente le nombre maximum de tâches pouvant s'exécuter dans une machine virtuelle sans entraîner le rejet de la tâche. Cette solution essaye de tenir en compte l'utilisation réelle des ressources par les conteneurs plutôt que des estimations faites par les utilisateurs [31].

2.2.3 Résultats

L'évaluation de l'approche proposée était sur deux critères : l'exécution de la tâche et la consommation de l'énergie. L'ensemble de données utilisées a été dérivé de la seconde version du journal de trace des nuages Google collectée pendant 29 jours. Le journal se compose des tableaux de données décrivant les machines (hétérogènes en terme de CPU, mémoire et capacité du disque), travaux et tâches. Dans le journal de suivi, chaque travail consiste à un certain nombre de tâches avec des contraintes spécifiques. En considérant ces contraintes, l'ordonnanceur détermine le placement des tâches sur les machines appropriées.

Pour étudier l'efficacité de la technique de dimensionnement des machines virtuelles(VM), des scénarios de base dans lesquels les tailles de VMs sont fixes ont été considérés. L'approche basée sur l'utilisation, où les tailles des VMs sont choisies en fonction des exigences réelles des applications plutôt que du montant demandé par les utilisateurs, surpasse l'approche basée sur la demande de près de 50% en terme d'énergie moyenne consommée par centre de données. De plus, l'approche proposée surestime ces scénarios basés sur l'utilisation de près de 7,55% en terme de consommation d'énergie des centres de données. En dehors de la perspective énergétique, l'approche proposée aboutit à moins de nombre d'instanciations de VMs par rapport à toutes les autres expériences [31].

2.3 Consolidation dynamique de machines virtuelles

2.3.1 Présentation de problème

La consolidation dynamique des machines virtuelles (VMs) dont le but est de minimiser le nombre d'hôtes physiques actifs, est le moyen le plus efficace pour améliorer l'utilisation des ressources et l'efficacité énergétique dans les centres de données. Le fait de déterminer quand est-ce que les VMs doivent être déplacées de la machine surchargée vers une autre représente l'aspect de la consolidation dynamique des VMs ayant un impact sur l'utilisation des ressources et la qualité de service fournie par le système.

Des solutions actuelles existent mais elles conduisent à des résultats sous-optimaux et elles ne permettent pas de spécifier explicitement la qualité des services. Par conséquent, le problème devient la minimisation de la consommation de l'énergie sous contrainte de la qualité de service (QoS : Quality of Service). Ce dernier peut être divisé en plusieurs sous problèmes dont la détection d'hôte surchargé va être le problème essentiel à résoudre dans l'article [7].

2.3.2 Approche proposée

L'approche proposée par Beloglazov et Buyya [7] cherche à fournir un modèle qui permet de déterminer qu'un hôte est surchargé d'où la nécessité de déplacer certaines machines virtuelles vers d'autres hôtes tout en donnant la possibilité de spécifier explicitement les exigences de la qualité de service qui doit être fournie par le système.

D'abord, pour une charge de travail stationnaire connue et une configuration d'état donnée, la politique de contrôle dérivé du modèle Markov résout de manière optimale le problème de détection de surcharge d'hôte dans le paramètre en ligne en maximisant le temps moyen entre les migrations tout en respectant l'objectif de qualité de service. Puis, pour une charge de travail non stationnaire inconnue, la solution était d'adapter, en utilisant une approche d'estimation de cette charge à l'aide d'une fenêtre coulissante à plusieurs vitesses, le modèle Markov pour gérer ce type de charge de travail.

De plus, un algorithme hors ligne optimal a été proposé pour le problème de la détection d'hôte surchargée. Cet algorithme incrémente le temps et recalcule à chaque itération la valeur de l'OTF (Overload Time Fraction) qui représente le seuil d'utilisation de CPU au niveau requis inférieurs à 100% [7]. Un hôte est considéré surchargé si l'utilisation de CPU a dépassé le seuil spécifié.

2.3.3 Résultats

Afin d'évaluer la performance du modèle proposé pour résoudre le problème de la détection de surcharge d'hôte, des simulations de traces de charges de travail à partir des milliers de machines virtuelles de PlanetLab hébergés dans des serveurs localisés dans plus que 500 endroits dans le monde entier ont montré que l'approche proposée superforme le meilleur algorithme de référence et fournit approximativement 88% de la performance de l'algorithme hors ligne optimal [7].

2.4 Migration à chaud des machines virtuelles

2.4.1 Présentation de problème

Dans un centre de données, les machines virtuelles migrent (se déplacent d'un serveur à un autre à travers le réseau) afin de répartir les charges, réduire la consommation de l'énergie ou effectuer une opération de maintenance des serveurs en production [22]. Cette opération de migration est coûteuse en terme de bande passante, d'énergie et de CPU.

En outre, elle réduit temporairement la disponibilité des machines virtuelles [25]. Les algorithmes de décision actuels se basent sur des hypothèses irréalistes (charge nulle pour les VMs et un réseau non bloquant) ce qui en résulte des migrations très

longues, inutiles et incontrôlables. En d'autres termes, il n'y a pas un contrôle des migrations.

2.4.2 Approche proposée

La solution proposée a consisté de donner un ordonnanceur des migrations mVM qui prend en compte la topologie réseau, la charge du travail des machines virtuelles et l'expectation du client et de l'administrateur du centre de données pour calculer la bonne séquence des migrations avec toutes les actions nécessaires pour effectuer une reconfiguration du centre de données tout en satisfaisant continuellement les contraintes et réduire au maximum la durée de cette opération[22, 25].

Le mVM, en fait, est un plugin (un paquet qui complète un logiciel hôte pour lui apporter de nouvelles fonctionnalités) de BtrPlace ; un ordonnanceur des machines virtuelles qui calcule le nouveau placement de ces dernières, le prochain état des serveurs et les actions à exécuter pour arriver au stade souhaité [25] ; qui peut être personnalisé avec des contraintes d'ordonnancement supplémentaires afin de bien gérer les actions de migrations.

L'ordonnanceur BtrPlace utilise la programmation par contraintes plus la librairie Java Choco qui est une solution open source dédiée à ce genre de programmation offrant une résolution du problème modélisé par l'utilisateur d'une manière déclarative et en indiquant l'ensemble des contraintes qui doivent être satisfaites dans chaque solution en alternant des algorithmes de filtrage de contraintes à l'aide des mécanismes de recherche [34]. Pour simplifier les choses, la seule relation entre BtrPlace et mVM est l'ordonnanceur de migration [22].

2.4.3 Résultats

Une série d'expérimentations ont été faites sur la plateforme Grid'5000 afin d'évaluer le gain apporté par mVM par rapport à BtrPlace ainsi que l'intérêt pratique de mVM à intégrer des contraintes énergétiques. En premier lieu, une expérience qui consiste à migrer les machines virtuelles (VMs) de quatre serveurs source ayant chacun deux VMs vers un seul serveur destination montre que les migrations ont été 3.5 fois plus rapides avec mVM grâce à sa parallélisation optimale des migrations [22]. En deuxième lieu, une expérience de simulation de décommissionnement de deux racks de douze serveurs, hébergeant chacun quatre machines virtuelles, vers un rack de serveurs plus récents a été effectuée. Initialement, les nouveaux serveurs sont éteints et les anciens serveurs seront éteints une fois vidés de leur VMs.

Dans l'expérience, le processus de reconfiguration est borné par une contrainte qui limite la puissance instantanée de l'infrastructure. Les résultats ont montré un gain énergétique de 5.86% dû la synchronisation de la fin de démarrage des serveurs avec le début des migrations les remplissant [22]. Enfin, à très grande échelle, la durée de résolution de mVM devient plus importante par rapport à BtrPlace. Une solution

pour surmonter cette limitation consiste à partager l'opération en plusieurs étapes successives [22, 25]. C'est à dire, lorsque la bande passante utilisée pour migrer les VMs dépasse la capacité des commutateurs, mVM prend la décision de faire migrer les VMs par groupes [22, 25].

Conclusion

Ce chapitre est un aperçu sur les problèmes traités concernant le placement, le redimensionnment, la consolidation et la migration des machines virtuelles. Notre stage s'intéresse non seulement aux machines virtuelles mais aussi aux ordonnanceurs des conteneurs qui seront présentés dans le chapitre qui suit.

Chapitre 3

Systèmes d'ordonnancement des conteneurs

Introduction

Dans ce chapitre, nous parlerons des systèmes d'ordonnancement qui existent dans la littérature tout en détaillant leurs principes de fonctionnement. Nous parlerons de Docker en grande partie autour duquel notre travail est centré.

3.1 Docker

3.1.1 Définition

Docker est une plateforme de virtualisation par conteneur qui, contrairement à la virtualisation sur hyperviseur où des nouvelles machines complètes doivent être créées pour isoler celles-ci les unes des autres et s'assurer de leur indépendance, permet à ses utilisateurs de créer des conteneurs qui vont uniquement contenir une ou plusieurs applications.

Grâce à l'empaquetage sous forme de conteneurs indépendants l'un de l'autre, ces applications sont facilement déployées sur tout hôte exécutant Docker. Cela permet d'étendre la flexibilité et la portabilité d'exécution d'une application. La première version de Docker est sortie en mars 2013 et la dernière version 1.13 en janvier 2017.

Docker est basé sur des images représentant des instances des systèmes d'exploitation qui peuvent être personnalisées et un système de fichiers en couches afin de partager le noyau du système d'exploitation hôte [20]. Cette technologie permettant de créer et de gérer des conteneurs peut simplifier la mise en œuvre des systèmes

distribués en permettant à de multiples applications, tâches de fond et autres processus de s'exécuter de façon autonome sur une seule machine physique ou à travers un ensemble de machines isolées.

3.1.2 Fonctionnement

Tout d'abord, il convient d'installer le Docker Engine, élément de base de la plateforme de conteneurisation, et créer après des conteneurs à travers les images Docker disponibles. Ces images peuvent être soit importées de Docker Hub contenant une grande variété des images développées par Docker ou par des utilisateurs en utilisant la commande **docker pull**, soit créées par le client lui même en partant d'une image de base comme debian, fedora ou ubuntu puis faire les modifications souhaitées en utilisant la commande **docker build**.

Cette opération ne nécessite qu'un seul fichier nommé **Dockerfile** (voir Annexe A) contenant une description de ce que comporte l'image à créer et établie ses dépendances d'une façon simple à l'aide de quelques commandes. Après la fabrication de l'image, l'utilisateur peut lancer le nombre de conteneurs qu'il veut en tenant compte de la limite des ressources disponibles à travers la commande **docker run**. L'image créée peut être récupérée du conteneur et aussi sauvegardée sous forme d'archive ce qui facilite sa portabilité.

D'un point de vue architecture, représentée à la Figure 3.1 [10], toute instruction lancée par le client sur la machine hôte va être communiquée par le client Docker au démon Docker qui se charge de son exécution.

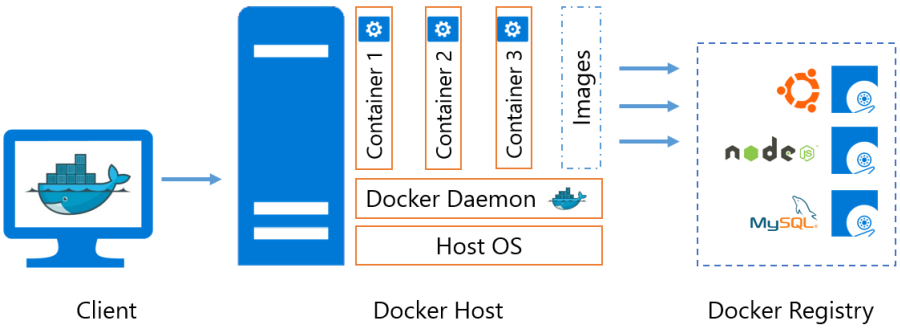

FIGURE 3.1 – Architecture de Docker

Comme les conteneurs sont basés sur des images et puisqu'ils s'exécutent indépendamment les uns des autres, il faut noter que ceci rend le transfert de données entre l'hôte et le conteneur ou entre les conteneurs eux même, coûteux par rapport aux machines virtuelles. C'est pour cette raison que Docker offre la possibilité d'avoir un volume partagé entre les conteneurs ou entre les conteneurs et la machine hôte ce qui réduit le temps de transfert de données. La gestion des volumes se fait en ajoutant l'option **-v** à la commande **docker run** ou **docker create**.

3.1.3 Mode Swarm du Docker Engine

À partir de la version 1.12 du Docker Engine, Swarm devient une partie native du démon Docker et qui offre la possibilité à l'utilisateur de faire fonctionner Docker Engine en "mode swarm". Ceci permet de configurer facilement et d'une manière simple une grappe de démons Docker.

Lors de la création d'une grappe (voir Annexe A), on a besoin de désigner au moins un nœud gestionnaire car le mode Swarm supporte plusieurs nœuds gestionnaires. Dans le cas où on a plus qu'un nœud gestionnaire, il y aura une élection afin de choisir celui qui sera le gestionnaire primaire responsable de l'orchestration.

Le reste des nœuds vont être des nœuds travailleurs ayant comme rôle l'exécution des conteneurs. Dans le mode Swarm, le client lance une demande de création du service en utilisant la commande "**docker service create - -name nom_service**". Ce dernier est un conteneur Docker de longue durée qui peut être déployé sur n'importe quel nœud travailleur.

Il y en a deux types : répliqué (c'est le mode par défaut) et global. Pour les services répliqués, le client spécifie le nombre de tâches de réplication que le gestionnaire Swarm doit planifier sur les nœuds disponibles en utilisant l'option "- -**replicas nombre_de_replicas**". Ces répliques qui seront sous forme de tâches se composent de plusieurs instances de conteneurs Docker spécifiées.

Afin d'afficher la liste de ces tâches et par suite les conteneurs qui tournent dans la grappe pour un service bien précis, on peut utiliser la commande "**docker service ps nom_service**". Par contre, la commande "**docker service ls**" liste les services. Pour les services globaux, le nœud gestionnaire élu comme primaire place une tâche sur chaque nœud disponible. Ce mode est spécifié lors de la création d'un service grâce à l'option "- -**mode global**".

3.1.4 Outils logiciels de Docker

Docker offre plusieurs outils pour gérer les conteneurs en groupe : Docker Machine, Docker Compose et Docker Swarm. Lorsqu'on utilise une approche d'architecture micro-services pour développer des applications, une application sera faite de petits services indépendants avec des canaux de communication pertinents entre certains serveurs.

L'outil Docker Machine par exemple permet à l'utilisateur d'installer Docker Engine sur des machines virtuelles puis de les gérer grâce à la commande **docker-machine**. Quant à Docker Compose, il permet au client de créer et de configurer à l'aide d'un seul fichier les services qu'il veut déployer. Par conséquent, l'exécution de l'application dans son ensemble nécessite moins d'effort car tous les services sont gérés à partir d'un seul outil. Ce dernier est utilisé pour gérer de multiples conteneurs sur la même machine.

Par contre, avec Docker Swarm, l'ordonnanceur de conteneurs Docker, l'utilisateur n'a pas besoin de connaître sur quelle machine son conteneur s'exécute car le gestionnaire swarm s'en charge [21].

3.1.5 Swarmkit

SwarmKit est un projet très récent de Docker qui représente une nouvelle structure logicielle pour la construction des systèmes d'orchestration. Avec SwarmKit, le service est utilisé comme modèle de base et non pas le conteneur. La grappe du SwarmKit peut se composer d'un ou plusieurs nœud(s) dont il peut être soit un travailleur qui exécute les conteneurs soit un gestionnaire actif qui fait partie du protocole RAFT.

Afin de communiquer avec le gestionnaire, le client peut utiliser l'API SwarmKit pour demander la création de d'un service dont il spécifie son état par le nombre de replicas qu'il lui faut, l'image, le mode déploiement, etc. Comme le montre la figure 3.2, la demande sera reçue par le nœud gestionnaire qui utilise des sous systèmes afin de diviser le service en une ou plusieurs tâche(s). La tâche va correspondre à un conteneur bien spécifique qui sera assigné après à un nœud travailleur.

Pour déployer une application dans SwarmKit, le client soumet la définition du service au nœud gestionnaire. Ce dernier s'occupe de la distribution des unités de travail appelées tâches aux nœuds travailleurs de l'infrastructure qui vont finalement faire exécuter le conteneur. La Figure 3.2 [15] représente un schéma récapitulatif du fonctionnement du Docker SwarmKit.

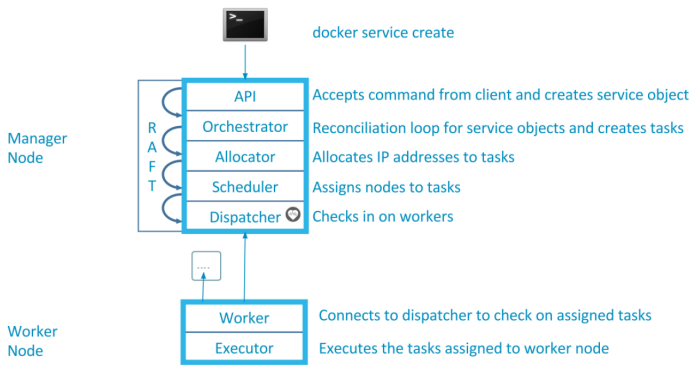

Figure 3.2 – Fonctionnement du SwarmKit

Le gestionnaire et le travailleur sont les deux éléments de base dans cette chaîne [9]. En prenant en premier lieu le gestionnaire, ce nœud se compose de cinq sous éléments dont chacun a un rôle bien précis :

- API (Application Programming Interface) : elle accepte les commandes de création du service et crée un objet service.

- Orchestrateur (orchestrator) : boucle de réconciliation qui crée des tâches pour les objets services.
- Allocateur (allocator) : alloue des adresses IP aux tâches.
- Ordonnanceur (scheduler) : attribue les tâches aux nœuds.
- Répartiteur (dispatcher) : charge le nœud travailleur d'exécuter une tâche.

En deuxième lieu le travailleur, cet élément assure son rôle grâce à deux sous éléments qui sont :

- Travailleur (worker) : il est connecté au répartiteur afin de connaître s'il y a des tâches qui lui ont été affectées.
- Exécuteur (executor) : il exécute la tâche affectée au nœud travailleur.

De plus, la Figure 3.2 montre que les cinq sous éléments du nœud gestionnaire forment ensemble un protocole appelé RAFT. En fait, SwarmKit implémente le concept de réconciliation d'état souhaité en utilisant ce protocole. Cela signifie que chaque fois qu'il y a un problème avec le service créé dans n'importe quel nœud, le nœud gestionnaire reconnaîtra une telle déviation par rapport à l'état souhaité et résoudra ce problème en créant une nouvelle instance de conteneur compensé.

Actuellement, Swarmkit s'appuie sur la stratégie "spread" qui favorise le nœud le moins chargé tout en respectant les contraintes et les besoins en ressources pour affecter le conteneur à un nœud travailleur.

3.2 Ordonnanceurs de conteneurs

3.2.1 Docker Swarm

Swarm est un ordonnanceur de conteneurs qui a été développé par Docker. Son architecture est donné par la figure 3.3 [43].

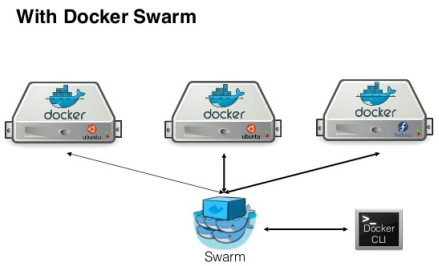

Figure 3.3 – Architecture de Swarm

Comme cette figure le montre, cet ordonnanceur se base sur deux éléments :

- Gestionnaire Swarm (manager) : c'est un nœud responsable de l'ordonnancement des conteneurs sur chaque agent. Dans une grappe Swarm, il est possible d'avoir plus qu'un seul gestionnaire Swarm mais un et un seul de ces gestionnaires va être élu comme primaire et les autres seront secondaires.

- Agent(s)/nœud (s) travailleur(s) (worker) : ce sont des machines ayant l'API Docker disponible par la machine Swarm lors du démarrage du démon Docker. Ils sont responsables de l'exécution des conteneurs.

L'outil Docker Swarm offre plusieurs commandes qui facilitent pour l'utilisateur de gérer sa grappe dont les principales sont les suivantes [14] :

- **docker run - -rm swarm create** ; cette commande utilise le backend de découverte hébergé de Docker Hub pour créer un jeton de découverte unique pour la grappe à créer.

- **docker run swarm join [OPTIONS] <discovery>** ; cette commande est utilisée pour créer un nœud swarm ayant comme rôle l'exécution des conteneurs.

- **docker run swarm manage [OPTIONS] <discovery>** ; cette dernière permet de créer un gestionnaire swarm.

- **docker run swarm list [OPTIONS] <discovery>** ; elle permet d'afficher la liste des nœuds dans la grappe.

Une fois la grappe a été créée et configurée, le gestionnaire Swarm doit décider sur quel nœud le conteneur doit être exécuté lors de la réception d'une demande. En premier lieu, il doit sélectionner grâce à des filtres qui seront détaillés ultérieurement les nœuds éligibles à exécuter le conteneur selon certains critères. En deuxième lieu, il faut choisir un seul nœud à chaque fois qu'un conteneur est lancé. Pour sélectionner le nœud sur lequel le conteneur va s'exécuter, Swarm utilise l'une de ces trois stratégies pour faire le classement des hôtes :

- **Spread** : favorise le nœud qui a moins de conteneurs par rapport aux autres sans tenir compte de leurs états. Si aucune stratégie n'est spécifiée au démarrage du Swarm, cette stratégie sera la stratégie adoptée par défaut.

- **Binpack** (voir Annexe A) : favorise le nœud le plus chargé. C'est à dire celui qui a le moins de ressources disponibles et qui contient le plus grand nombre de conteneurs.

- **Aléatoire** : choisir aléatoirement un nœud.

Mais comme décrit précédemment, il faut préparer, tout d'abord, la liste des nœuds parmi les nœuds disponibles de la grappe, éligibles à exécuter le conteneur. Pour faire cette sélection, Swarm dispose de cinq filtres disponibles et qui peuvent être utilisés simultanément :

- **Filtre de contrainte** : au début, il associe à un nœud une paire de clé/valeur dès que le démon Docker a démarré sur ce nœud. Lorsqu'il s'agit d'un environnement de production, on peut spécifier les contraintes exigées. Enfin, le conteneur ne va pas être exécuté que sur un nœud étiqueté par une clé/valeur. Si aucun nœud ne répond aux exigences, ce conteneur sera en attente et il ne sera pas démarré.
- **Filtre de santé** : empêche l'ordonnancement sur des nœuds défaillants.
- **Filtre d'affinité** (voir Annexe A) : crée des attractions lors de l'exécution d'un nouveau conteneur. Plus précisément, il ordonnance les conteneurs à exécuter sur des nœuds où une image spécifique est déjà instanciée.
- **Filtre de dépendance** : il est utilisé lorsque le conteneur à exécuter dépend d'un autre conteneur.
- **Filtre de port** : il sert à faire exécuter un conteneur sur un nœud ayant un port spécifique libre.

Afin de mesurer la différence de performance entre les trois stratégies d'ordonnancement : aléatoire, binpack et spread, un test a été fait sur deux grappes l'une avec deux nœuds et l'autre avec cinq nœuds [21]. A part la différence au niveau du nombre de nœuds, les images dans la première grappe sont installées seulement lorsqu'un nœud a été sélectionné comme prochain nœud alors que dans la deuxième elles sont installées dès la création de la grappe. Le nombre de conteneurs a été progressivement augmenté, ce qui a donné lieu à des unités de temps de fonctionnement indiquées dans le tableau 2.2 pour le cluster à cinq nœuds [21] :

Nombre de conteneurs	50	100	500	1000	2000
spread	1.0	2.3	10.8	22.2	43.1
random	1.2	3.4	11.6	27.5	44.1
binpack	5.1	8.3	15.9	28.0	44.6

TABLE 3.1 – Délai d'exécution en fonction du nombre de conteneurs

Comme le tableau ci-dessus le montre, plus le nombre de conteneurs augmente plus la différence de performance entre les trois stratégies devient légère.

3.2.2 Apache Mesos avec Marathon

Cet ordonnanceur offre beaucoup de fonctionnalités en terme d'ordonnancement. Nous pouvons citer par exemple les contraintes, des contrôles d'état (health checks), le service de découverte et l'équilibrage de charges. Son architecture est représentée

par la Figure 3.4 [20].

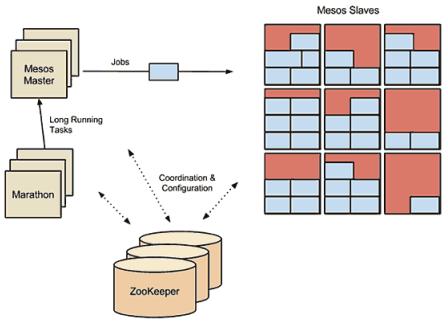

Figure 3.4 – Architecture d'Apache Mesos avec Marathon

Mesos se base dans son fonctionnement sur quatre éléments principaux qui sont :

» Zookeeper : il aide le Marathon à rechercher l'adresse du maître Mesos.

» Marathon : c'est celui qui démarre, surveille, et déplace les conteneurs.

» Maître Mesos : il envoie la tâche à exécuter au nœud approprié et il fait des offres au Marathon lorsqu'un nœud a peu de CPU/RAM libre.

» Esclaves de Mesos : ils ont pour rôle l'exécution du conteneur et ils envoient une liste de leurs ressources disponibles au Zookeeper.

Les contraintes donnent aux opérateurs le contrôle de l'endroit où les applications doivent s'exécuter. Elles se composent de trois parties : un nom de champ (peut être le nom d'hôte esclave, ou tout attribut Mesos esclave), un opérateur et une valeur facultative. Les opérateurs disponibles sont :

□ UNIQUE : il force le caractère unique de caractéristiques. Par exemple, si la contrainte ["hostname","UNIQUE"] assure qu'une seule tâche d'application s'exécute sur chaque hôte.

□ CLUSTER : il exécute l'application sur les esclaves qui partagent un attribut. Prenant l'exemple suivant, la contrainte ["rack id", "CLUSTER", "rack-1"] oblige l'application à s'exécuter sur rack-1 ou à être dans un état en attente pour attendre quelques CPU / RAM libre sur rack-1.

□ GROUP BY : il sert à répartir l'application uniformément entre les nœuds avec un attribut particulier.

□ LIKE : il garantit que l'application ne fonctionne que sur les esclaves ayant un certain attribut. Lorsqu'il n'y a qu'une seule valeur, il fonctionne comme l'opérateur CLUSTER expliqué déjà dans les items précédents, mais beaucoup de valeurs peuvent être adaptées parce que l'argument est une expression régulière.

- UNLIKE : il fait le contraire de LIKE.

De plus, il utilise les contrôles d'état qui représentent les dépendances et les exigences d'une application qu'il faut implémenter manuellement. Quant au service de découverte, il est utilisé en cas de besoin, et il permet d'envoyer des données à des applications en cours d'exécution.

3.2.3 Google Kubernetes

On peut définir Kubernetes [48] comme un outil d'orchestration pour les conteneurs Docker utilisant le concept des étiquettes et des pods pour grouper les conteneurs en unités logiques. Un pod est un groupe de conteneurs se trouvant sur la même machine et partageant le même ensemble d'espaces de noms Linux [29]. Ceci facilite le management du cluster en comparaison avec un co-ordonnancement par affinité de récipients (comme dans le cas de Swarm et Mesos) [20]. L'architecture de Kubernetes est montrée par la Figure 3.5 [20].

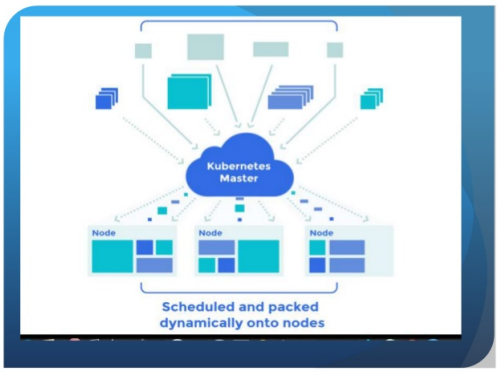

Figure 3.5 – Architecture de Kubernetes

Le fonctionnement de Kubernetes est assuré par les composants suivants [35] :

- Maître : exécute l'interface de programmation d'application (API, Application Programming Interface) de Kubernetes, et contrôle le cluster.
- Label : il s'agit d'une paire clé/valeur utilisée pour la découverte de services. Un label marque les conteneurs et les fédère au sein de groupes.
- Contrôleur de réplication : Il garantit que les nombres de pods demandés s'exécutent selon les exigences de l'utilisateur. Ce composant gère l'évolution horizontale des conteneurs, en régulant leur nombre pour répondre aux besoins de traitement de l'application en général.
- Service : intégrateur et équilibreur de charge configuré automatiquement qui s'exécute à l'échelle du cluster.

Afin de sélectionner le nœud sur lequel le pod va être exécuté, il y a deux étapes. La première consiste à filtrer tous les nœuds et la seconde est de classer les nœuds restants pour trouver le meilleur ajustement pour le Pod. Le filtrage des nœuds a pour but de filtrer les nœuds qui ne répondent pas à certaines exigences du pod. Actuellement, Kubernetes utilise plusieurs prédicats comme outil pour assurer ce filtrage, notamment :

- PodFitPorts : le nœud doit être en mesure d'accueillir le pod sans conflits de port.
- PodFitsRessources : le nœud a suffisamment de ressources pour accueillir le pod.
- NoDiskConflict : le nœud a suffisamment de place pour le pod et les volumes liés.
- NoVolumeZoneConflict : évaluer si les volumes d'une demande de pod sont disponibles sue le nœud en tenant compte des restrictions de zone.
- MatchNodeSelector : le nœud correspond au paramètre de la requête du sélecteur définie dans la description du pod.
- HostName : le nœud a le nom du paramètre hôte dans la description du pod.

Tous ces prédicats peuvent être utilisés en combinaison afin d'effectuer une politique de filtrage sophistiquée. Une fois le filtrage est terminé, dans la plupart des cas il reste plus qu'un nœud, Kubernetes considère que la liste des nœuds filtrés comme aptes à héberger le pod. Alors, il donne des priorités aux nœuds restants pour sélectionner le meilleur pour le pod en utilisant comme outil des fonctions de priorités.

Pour chaque nœud restant, la fonction de priorité lui donne un score qui varie entre 0 et 10 avec 10 désigne le plus préféré et 0 le moins préféré. Chaque fonction de priorité est pondérée par un nombre positif et le score final de chaque nœud sera la somme de tous les scores pondérés.

Pour mieux comprendre ce principe, on va prendre comme exemple le cas où deux fonctions de priorités qui vont être utilisées priorityFunc1 et priorityFunc2 avec des facteurs de pondération Weight1 et Weight2 respectivement. Le score final pour un nœud nommé NodeA sera calculé comme suit :

FinalScoreNodeA=(Weight1*priorityFunc1)+(Weight2*priorityFunc2)

Parmi les fonctions de priorités utilisées par Kubernetes, on peut trouver :

- LeastRequestedPriority : le nœud est prioritaire en fonction de la fraction du nœud qui serait libre si le nouveau Pod était planifié sur le nœud. Le CPU et la mémoire sont également pondérés. Le nœud avec la fraction libre la plus élevée est le plus préféré. Cette fonction de priorité a pour effet de répartir les pods entre les nœuds en ce qui concerne la consommation de ressources.
- BalancedResourceAllocation : cette fonction de priorité essaie de mettre le pod sur un nœud de telle sorte que le taux d'utilisation de la CPU et de la mémoire est équilibré après le déploiement du pod.

- ServiceSepreadingPriority : elle favorise les nœuds qui sont utilisés par différents pods.
- EqualPriority : cette fonction est utilisée uniquement pour les tests. Elle donne une priorité égale à tous les nœuds du cluster.

Une fois que les scores de tous les nœuds sont calculés, le nœud ayant le score le plus élevé sera choisi comme l'hôte qui va héberger le pod. S'il y en plus qu'un nœud avec le même score représentant le score le plus élevé, Kubernetes choisit aléatoirement le nœud final.

3.2.4 Quelques résultats des comparaisons entre ces ordonnanceurs

Afin de faire la comparaison entre les trois outils cités précédemment, deux exemples différents sont utilisés. Le premier est le déploiement d'un site web de camions alimentaires et le deuxième exemple est l'exécution d'une application de vote [20]. Cette comparaison nous guide à dégager les points faibles ou fort d'un ordonnanceur par rapport à l'autre :

- Swarm crée une image comme un seul hôte ; c'est à dire qu'elle ne vivra que sur un seul nœud et elle ne sera pas distribuée sur les autres nœuds du cluster.
- L'ordonnancement assuré par Swarm se dégrade s'il y en a beaucoup de conteneurs à utiliser. C'est pour cette raison qu'il faut vérifier régulièrement le nombre de conteneurs utilisés afin d'échapper à une dégradation de performance.
- Swarm ne conserve pas une trace du nombre d'instances d'un service. Si un conteneur échoue, il ne crée pas un nouveau.
- Swarm est incapable de faire des mises à jour pour certains conteneurs dans la grappe.
- Kubernetes facilite beaucoup l'acheminement du trafic vers un pod à l'aide d'une adresse IP élastique. Une adresse IP élastique est une adresse qui pointe vers une autre adresse dans l'informatique en nuage. Les adresses IP élastiques permettent à l'utilisateur de disposer d'une adresse IP bien connue, qui au fil du temps, peut pointer vers un autre emplacement. C'est à dire, le client dispose d'une adresse IP unique qu'il peut associer à différentes instances.
- Kubernetes est le meilleur ordonnanceur pour les micro-services.
- Kubernetes possède une architecture plus complexe que Swarm.
- Kubernetes offre la possibilité de faire des mises à jour aux conteneurs dans la grappe.
- Swarm offre un degré élevé de configuration pour les développeurs qui souhaitent des flux de travail très spécifiques.

- Swarm est open source. Il n'est pas attaché à aucun fournisseur de l'informatique en nuage.
- Mesos offre un fichier de définition plus puissant pour que le conteneur soit bien décrit ainsi que ses exigences dans la grappe.

De plus, d'autres expériences ont été faites en jouant sur le critère d'affinité aux nœuds (certains nœuds ont un positionnement réseau particulier qui les rend éligibles à porter des conteneurs particuliers et d'autres ont des caractéristiques techniques), d'anti-affinité (exiger d'exécuter certain conteneurs sur des machines différentes) entre les conteneurs et de la labellisation des nœuds à chaud.

Les résultats ont montré une bonne performance de Kubernetes (version 1.1.7) par rapport à tous les autres ordonnanceurs. Cet ordonnanceur s'est distingué par une anti-affinité souple avec priorité et la labellisation des nœuds à chaud [26, 27].

Conclusion

Grâce à ce chapitre, on peut conclure qu'on ne peut pas dire qu'une solution d'ordonnancement de conteneurs est meilleure que les autres car selon les besoins de notre grappe on peut choisir la solution apte à être la meilleure. Dans le chapitre qui suit, nous expliquons la nouvelle stratégie de placement que nous avons proposée dans Docker Swarm et sa nouveauté par rapport aux stratégies qu'il utilise actuellement.

Chapitre 4

Contributions

Introduction

Dans ce chapitre, nous présentons notre contribution tout en détaillant les motivations, les démarches suivies et les expériences qui ont été faites pour la valider.

4.1 Motivations

Malgré les avantages que Docker Swarm présente, comme la haute disponibilité, la réplication d'état, l'équilibrage de la charge et la gestion de la grappe, cet ordonnanceur possède des limites sur lesquelles nous nous sommes basés pour décider d'implémenter une nouvelle stratégie d'ordonnancement :

- Il ne gère pas plusieurs demandes de création de conteneurs au même instant. Donc les clients ne peuvent pas soumettre plusieurs demandes de lancement de conteneurs au nœud gestionnaire au même instant.
- Swarm ne fait pas la différenciation des clients en classes. Cela veut dire qu'il n'y a pas un client plus prioritaire que l'autre.
- Chaque conteneur possède un nombre de ressources fixes exigés pour l'exécuter. C'est à dire qu'actuellement il n'y a pas de calcul dynamique des ressources à allouer au conteneur pour l'exécuter dans la grappe selon les ressources disponibles et sur les nœuds de l'infrastructure.
- Lorsqu'il n'y a pas de ressources disponibles suffisantes pour l'exécution du conteneur, ce dernier sera en attente. Donc il y aura la possibilité qu'un conteneur reste tout le temps en attente (problème de famine).

4.2 Algorithme générique proposé

Le but principal de l'algorithme générique qui a été proposé est la résolution du problème suivant : pour une entreprise fournisseur de services pour l'informatique en nuage, qui reçoit les demandes de ses clients en ligne dans une infrastructure privée et qui utilise un modèle économique, quelles seront les ressources qui vont être allouées pour chaque requête selon la classe de l'utilisateur et quelle est la machine la plus optimale qui doit être sélectionnée pour exécuter ce travail et comment satisfaire le maximum de requêtes soumises tout en réduisant la consommation énergétique de l'infrastructure.

L'idée est d'offrir au client une interface qui lui permette de choisir une classe SLA (Service Level Agreement) selon ses besoins et son budget monétaire. Puis, il soumet sa requête en ligne. A chaque fois que le système reçoit une nouvelle requête, il va passer par ces trois phases :

- **Ordonnancement des requêtes** : le système choisit parmi la liste des requêtes dont il dispose celle qui doit être exécutée ;
- **Allocation des ressources** : dans cette étape le système va décider combien de cœurs doivent être alloués à la requête sélectionnée ;
- **Affectation des ressources** : le système doit décider laquelle des machines qui doit exécuter cette requête.

Le modèle économique proposé offre trois classes de SLA (Service Level Agreement) pour répondre aux besoins des clients :

- Classe "Premium" : c'est la classe la plus prioritaire. Elle représente l'utilisateur qui a un service de longue durée et qui veut avoir la solution le plus tôt possible sans tenir compte du prix de l'opération ;
- Classe "Advanced" : cette classe est conçue pour l'utilisateur qui a un service court et il veut avoir un temps d'exécution le plus court avec un budget limité ;
- Classe "Best Effort" : cette classe désigne les utilisateurs qui ont un micro-service et ils sont intéressés à trouver une solution avec la meilleure offre (solution à faible coût).

Chaque classe a une priorité et le nombre de cœurs est limité par un nombre maximum et un nombre minimum. En effet, la priorité sert à accélérer l'exécution de la requête par rapport à une autre alors que la limitation du nombre de cœurs est mise pour les raisons suivantes :

- S'assurer qu'il n'y aura pas la possibilité qu'une requête moins prioritaire prenne plus de cœurs qu'une requête plus prioritaire ;
- Si à un instant donné, le système a une seule requête et si tous les cœurs de l'infrastructure sont libres, la limitation du nombre de cœurs va permettre de ne pas donner tous les cœurs disponibles de la même machine à une seule requête.

Ceci permet d'exécuter d'autres requêtes qui arrivent plus tard et qui sont plus prioritaire ;

- Donner une idée au client sur le nombre de cœurs qui peut être consommé par chaque classe SLA selon l'infrastructure utilisée.

Pour calculer le nombre maximum et le nombre minimum de cœurs pour chaque classe, il faut tout d'abord fixer k comme le nombre minimum de cœurs libres sur toutes les machines de l'infrastructure utilisée. Après, on peut calculer les deux bornes de la façon suivante :

- Nombre minimum des cœurs = facteur_minimum * k
- Nombre maximum des cœurs = facteur_maximum * k

Le facteur minimum et maximum est une variable fixée pour chaque classe SLA.

4.2.1 Ordonnancement des requêtes

L'algorithme 1 présente le principe de cette phase. En effet, le système proposé va utiliser une file d'attente pour stocker les requêtes soumises en ligne. Dans le contexte utilisé, la requête qui sera sélectionnée est celle qui a la priorité la plus élevée. A chaque fois qu'une nouvelle requête est soumise, toute la file des requêtes est triée de nouveau tout en prenant compte la priorité comme critère de tri.

Algorithm : Choix de requête à exécuter
Data: Q, requests' queue
Data: R, a new submitted request
Result: selected query
while *true* **do**
  if *R is submitted to the system* **then**
    Update the priority of all requests saved in the queue Q;
    Add R in Q;
    Order all the requests saved in Q according to their priority;
  end
  if *Q is not empty* **then**
    R_*= request with the highest priority in Q;
    Compute the number of cores associated to R_* ;
    if R_* *can be executed* **then**
      Remove R_* from Q;
      Execute R_* ;
    end
  end
end

Algorithm 1: Ordonnancement des requêtes

De plus, les priorités des requêtes qui sont déjà dans la file vont être mises à jour en multipliant leurs priorités actuelles par un facteur qui dépend de la classe dont

elles font parties. Ce principe est appliqué afin de favoriser l'exécution d'une requête moins prioritaire mais qui a été en attente pour une longue durée par rapport à une requête plus prioritaire qui vient d'être soumise récemment.
Pour mieux éclaircir le focntionnement de cette phase, nous avons utilisé le logiciel open source draw.io version 7.3.7 pour dessiner un organigramme représenté par la figure 4.1.

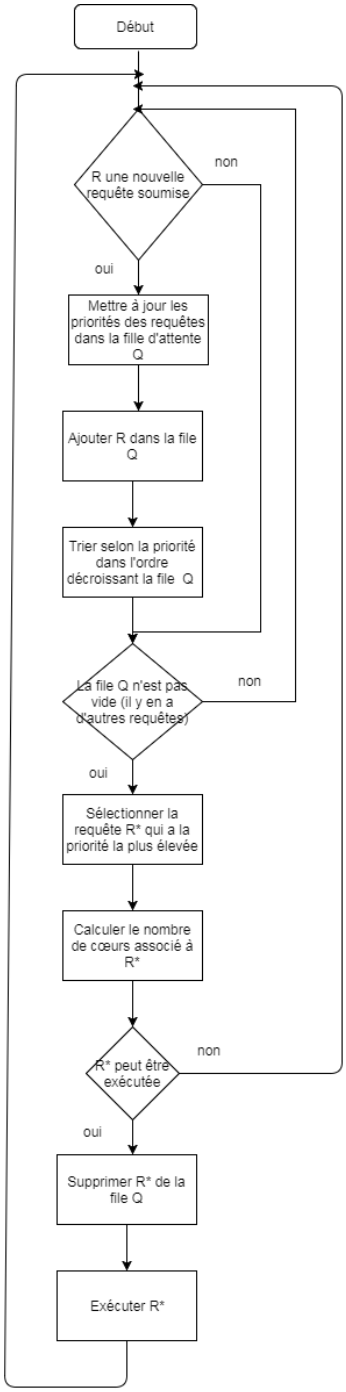

FIGURE 4.1 – Organigramme ordonnancement des requêtes

4.2.2 Allocation des ressources

Une fois le système a choisi la requête à exécuter ayant comme priorité P_i, il doit décider combien de cœurs va être alloués pour l'exécution de cette dernière. Le calcul du nombre de cœurs C_i pour cette requête prend en compte les requêtes dans la file d'attente (req_1,req_2,..,req_n) et leurs priorités (P_1,P_2,...,P_n) avec le nombre de tous les cœurs disponibles dans l'infrastructure(T). Et par conséquent, C_i est donné par la formule suivante :

$$\boxed{C_i = \frac{P_i * T}{\sum_{j=0}^{n} P_j}}$$

Après avoir fixé le nombre de cœurs à allouer C_i, on compare la valeur trouvée aux deux bornes qui limitent le nombre de cœurs pour chaque classe SLA. En effet, si C_i > nombre_max_coeurs fixé pour la classe SLA dont la requête fait partie alors C_i= nombre_max_coeurs. Dans le cas où C_i < nombre_min_coeurs de la classe SLA, C_i=nombre_min_coeurs. Afin que la requête soit exécutée, le système doit trouver une machine qui a un nombre de ressources libres C_{max} supérieur à la valeur fixée pour C_i.

Donc chaque requête a deux possibilités : soit elle va être exécutée lorsqu'il y aura au moins une machine dans l'infrastructure qui a des ressources libres plus grandes que la valeur C_i soit elle va être en attente dans la file. Dans le système proposé, on favorise l'exécution de la requête plutôt que de la mettre dans la file d'attente.

4.2.3 Affectation des ressources

Après avoir choisi la requête qui va être exécutée, parmi plusieurs requêtes dans la file d'attente, et après avoir fixé le nombre de cœurs qui va être alloué à cette dernière, le système doit choisir la machine la plus "optimale" qui peut l'exécuter. Pour faire ce choix, nous utilisons l'heuristique binpacking. En effet, elle favorise la machine la plus chargée pour maximiser l'utilisation des ressources et minimiser au même temps le nombre de machines actives. D'une manière plus précise, la machine qui va être choisie est celle qui a le grand nombre de cœurs utilisés par rapport aux autres machines.

4.3 Implémentation dans Docker Swarm

Notre tâche dans ce stage consiste à adapter et implémenter l'algorithme générique proposé dans le contexte des conteneurs Docker. Pour faire ceci, nous nous mettons au niveau du code source de Docker Swarm écrit en langage Go [1], langage qui a été développé par Google. Nous proposons d'y implémenter cet algorithme comme une nouvelle stratégie de placement.

1. https ://golang.org/

4.3.1 Présentation du langage Go

Le langage Go a été créé par une petite équipe chez Google qui a sorti une première version en novembre 2009, à laquelle se sont adjoints d'autres contributeurs du monde entier.

Au début, il a été conçu pour la programmation système puis il a été étendu aux applications. C'est un langage impératif et concurrent car il intègre directement les traitements de code en concurrence nommé **goroutine**. Plusieurs projets ont été écrits en langage Go, et nous pouvons citer, pour notre contexte Système, par exemple Docker, Docker Swarm, Kubernetes et consul qui représente un système de découverte de service et de gestion de configuration.

4.3.2 Interventions

Nous avons essayé de garder au maximum possible l'algorithme générique tel qu'il a été proposé puis l'adapter dans le contexte des conteneurs sans modifier l'architecture de Swarm ou son principe de fonctionnement. Dans ce contexte, on a adopté qu'une requête sera une demande de lancement de conteneur et le nombre de cœurs sera le nombre de CPUs à allouer à cette dernière. Notre intervention dans le code source de Docker Swarm se résume en deux phases montrées par la figure 4.2.

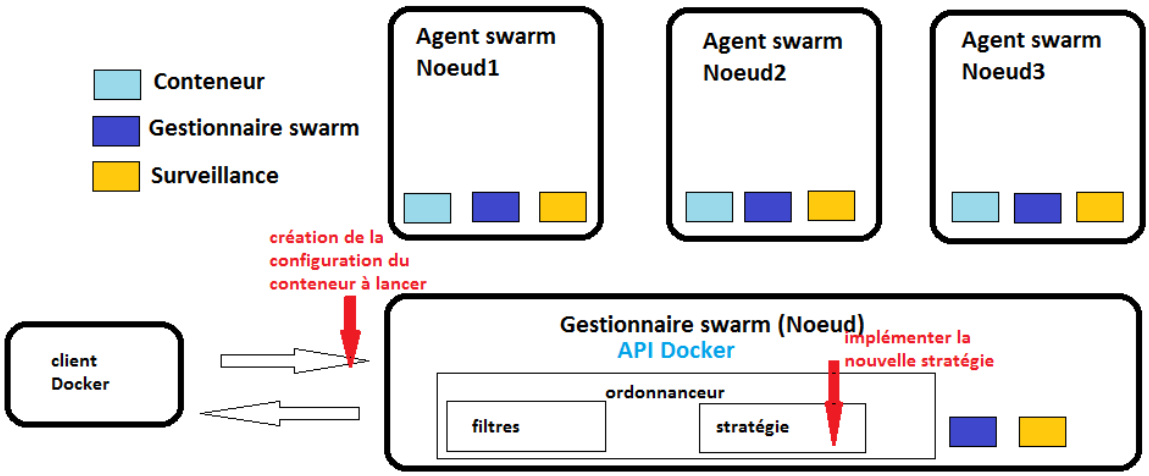

FIGURE 4.2 – Intervention au niveau du code source de Swarm

La première intervention est au niveau de la structure de configuration du conteneur. Pour que le conteneur ait une priorité faisant référence à sa classe SLA, nous

avons créé une nouvelle structure nommée "ClassSLA" qui est définie par la liste des attributs montrés par le tableau ci-dessous :

Nom Attribut	Type	Signification
Type	entier	il représente la classe SLA. S'il a la valuer 0, ça sera la classe "premium". S'il prend la valeur 1, la classe SLA sera "advanced". Et s'il a la valeur 2, cela veut dire que la classe est "best effort".
Priority	réel	il désigne la priorité du conteneur à exécuter.
MinCpus	réel	il représente le nombre de CPUS minimum à allouer au conteneur.
MaxCpus	réel	il représente le nombre maximum à allouer au conteneur.
FactorMin	réel	il représente le facteur de la classe utilisée pour calculer la borne inférieure du nombre de CPUS à allouer.
FactorMax	réel	il représente le facteur de la classe utilisée pour calculer la borne supérieure du nombre de CPUS.

Table 4.1 – Attributs définissant la classe SLA

Puis, nous avons ajouté cette structure comme nouvel attribut nommé "NewAttribut" dans la structure "Config" du paquet conteneur dans le fichier ayant comme chemin d'accès dans le code source "github.com/docker/ swarm/vendor/github.com/docker/ docker/api/types/container/config.go". Afin d'affecter des valeurs à ce dernier, nous avons utilisé un fichier JSON au niveau du paquet "strategy" dont le chemin d'accès est "github.com/docker/ swarm/scheduler/ strategy/config.json".

Ce fichier contient la structure "ClassSLA" dont chaque champ possède une valeur. Ensuite, la lecture des informations de ce fichier et l'affectation des valeurs seront effectuées au niveau de la fonction "BuildContainerConfig" qui se trouve dans le fichier "github.com/docker/swarm/cluster/ config.go" du paquet "cluster". En fait, cette fonction est celle qui crée la configuration du conteneur en récupérant les valeurs des options affectées par le client Docker lors de lancement de sa demande.

Nous avons proposé d'assigner les valeurs aux champs de l'attribut "NewAttribut" à ce niveau en récupérant les informations mises dans le fichier JSON afin de garantir

la conservation des données sur la structure "ClassSLA" de conteneur tout au long de son cycle de vie.

Dans la deuxième phase, nous avons implémenté la nouvelle stratégie en ajoutant un nouveau fichier dont le chemin d'accès est "github.com/docker/ swarm/scheduler/ strategy/hello.go" au niveau du paquet "strategy". La nouvelle stratégie proposée consiste à implémenter les deux phases de l'algorithme générique qui sont : allocation et affectation des ressources en faisant quelques modifications.

En effet, nous sommes obligés à respecter l'interface "PlacementStrategy" mise en place par Swarm et qui possède trois fonctions à implémenter. Sinon nous introduisons des modifications profondes au code Swarm.

Parmi ces trois fonctions, nous trouvons la fonction nommée "RankAndSort", dans laquelle nous avons implémenté le principe de notre stratégie. Elle prend en paramètres la liste des nœuds filtrés (qui ont été le résultat de l'opération de filtrage faite par l'ordonnanceur) et la configuration du conteneur et elle retournera un tableau des nœuds trié. Ce tableau va être utilisé par l'ordonnanceur qui choisira le nœud de la première case du tableau résultat pour exécuter le conteneur.

4.3.3 Nouvelle stratégie de placement

Comme première étape dans notre stratégie, nous commençons à chercher le nombre minimum de CPUS libres k sur toutes les machines déjà filtrées. L'étape suivante consiste à calculer les deux bornes du nombres de cpus que nous pouvons allouer au conteneur à exécuter en appliquant les mêmes formules utilisées dans l'algorithme générique.

Après, nous calculons la somme de tous les CPUs libres "*sommeCpus*" sur les nœuds filtrés. Puis, nous calculons "*sommePriority*" représentant la somme des priorités de tous les conteneurs en cours d'exécution sur les nœuds.

Ceci est l'une des simplifications que nous avons faites par rapport à l'algorithme proposé car Docker Swarm ne gère pas une file de demande de création simultanéee de conteneurs et par suite nous n'avons pas l'information sur les priorités des requêtes dans la file. A ce niveau, nous pouvons calculer le nombre de CPUS "*res*" qui sera alloué au conteneur ayant la priorité P_i en utilisant la formule ci-dessous :

$$\boxed{res = \frac{P_i * sommeCpus}{sommePriority}}$$

Ensuite, nous comparons cette valeur aux deux bornes de la classe dont le conteneur fait partie. Si la valeur de "res" dépasse la borne supérieure, nous lui affectons la valeur de cette borne et si la valeur de "res" est plus petite que la borne inférieure, nous lui donnons la valeur de cette dernière. Enfin, pour décider quelle machine va exécuter le conteneur nous parcourons la liste des nœuds et nous cherchons celle qui a le nombre

maximum de cpus utilisés.

Une fois que le nœud est trouvé, nous le plaçons à la première case (s'il n'était pas déjà à cette case) du tableau des nœuds en faisant une simple permutation entre le nœud trouvé et celui qui se trouve à la première case.

Le principe de notre stratégie est représenté par l'algorithme 2 :

Algorithm : Notre stratégie
Data: k, the minimum cpus' number
Data: FactorMin, factor used to calculate the maximum cpus' number which can be allocated
Data: FactorMax, factor used to calculate the minimum cpus' number which can be allocated
Data: max, max cpus of the $container_i$
Data: min, min cpus of the $container_i$
Data: maxRessources, the number max of used resources on the filtred nodes
Data: $UsedCpus_i$, the number of used Cpus on the $node_i$
Data: $UsedMemory_i$, the number of used Memory on the $node_i$
Data: $TotalCpus_i$, the total Cpus' number on the $node_i$
Data: $TotalMemory_i$, the total Memory on the $node_i$
Data: sommeCpus, the sum of unused cpus on the filtred nodes
Data: sommePriority, the sum of containers' priorities on the filtred nodes
Data: res, the number of Cpus which must be allocated to the container
Data: $containers_i$, number of containers on the $node_i$
Data: CPUShares, the number of Cpus which will be used by the container
Data: $priority_i$, the priority of the $container_i$
Data: node, the node having the highest number of used resources
Data: nodes, the list of filtred nodes
Result: output, sorted list of filtred nodes
k :=$TotalCpus_1$-$UsedCpus_1$;
sommeCpus :=0;
maxRessources :=$UsedCpus_1$+$UsedMemory_1$;
node :=nodes[0];
for *counter* $i \in nodes$ **do**
  if $TotalCpus_i$-$UsedCpus_i$<k **then**
    k :=$TotalCpus_i$-$UsedCpus_i$;
  end
  sommeCpus :=sommeCpus+($TotalCpus_i$-$UsedCpus_i$);
  for *counter* $j \in containers_i$ **do**
    sommePriority :=sommePriority+$priority_j$;
  end
end

max :=factorMax * k ;
min :=factorMin * k ;
if *sommePriority==0* **then**
  sommePriority :=1;
end
$res := \frac{priority_i * sommeCpus}{sommePriority}$;
if *res>max* **then**
  CPUShares :=max;
end
if *res<min* **then**
  CPUShares :=min;
end

for *counter* $i \in nodes$ **do**
  if $UsedCpus_i + UsedMemory_i > maxRessources$ **then**
    maxRessources :=$UsedCpus_i + UsedMemory_i$;
    node :=nodes[i];
  end
end
if *node<>nodes[0]* **then**
  permute(node,nodes[0]);
end
for *counter* $i \in nodes$ **do**
  output[i] :=nodes[i];
end

Algorithm 2: Allocation et affectation des ressources

La figure 4.3 réprésente une première partie d'un organigramme de l'algorithme 2 présenté précédemment pour faciliter la compréhension de déroulement de ce programme. Dans cette partie, le programme trouvera le nombre minimal de cpus libres comme première étape. Puis, il calculera la somme des cpus inutilisés dans l'infrastructure. Enfin, il calculera la somme des priorités de tous les conteneurs en cours d'exécution sur les nœuds.

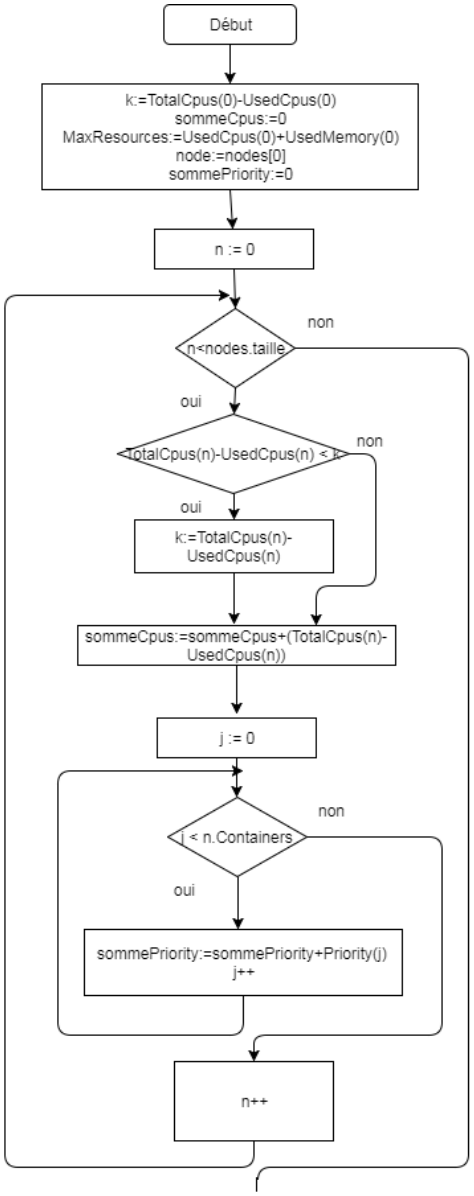

Figure 4.3 – Organigramme stratégie de placement-partie1

La figure 4.4 montre une deuxième partie de l'organigramme dans laquelle le pragramme calculera les valeurs de *min* et *max* représentant les deux bornes qui limitent le nombre de CPUs à allouer au conteneur à exécuter selon sa classe SLA.

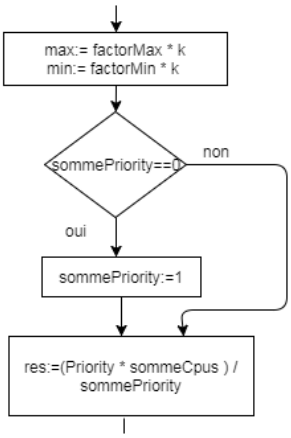

FIGURE 4.4 – Organigramme stratégie de placement- partie2

La figure 4.5 représente la suite de l'organigramme de la stratégie de placement. Dans cette partie, le programme décidera le nombre de CPUs qu'il allouera au conteneur.

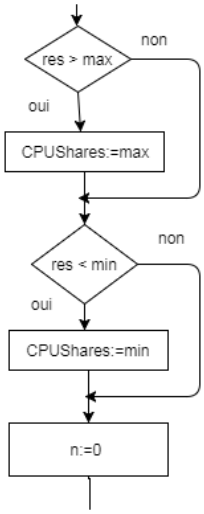

FIGURE 4.5 – Organigramme stratégie de placement- partie3

Enfin, la figure 4.6 montre la dernière phase de l'organigramme et par suite la fin du programme. Nous cherchons dans cette portion de l'organigramme la machine qui a le maximum de ressources (CPU et mémoire) utilisées. Puis, nous allons permuter le nœud choisi et celui qui est placé dans la première case du tableau dans le cas où les nœuds étaient différents. De cette façon, le nœud placé au début du tableau sera le nœud qui a été choisi pour exécuter le conteneur.

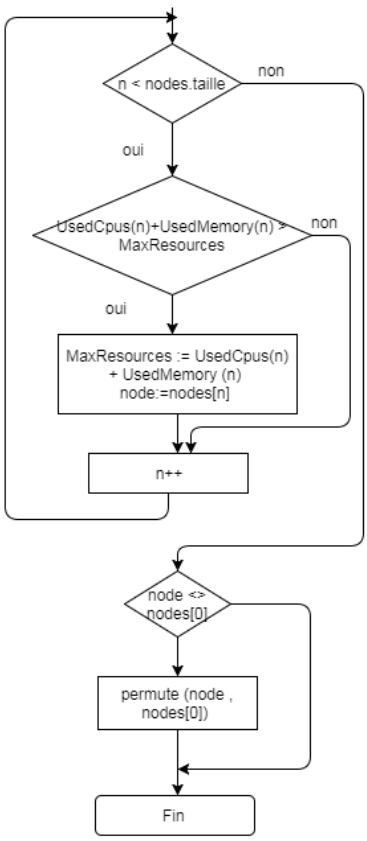

Figure 4.6 – Organigramme stratégie de placement-partie4

4.4 Expériences

4.4.1 Simulation de notre stratégie avec les traces Prezi

Nous avons implémenté une simulation pour la gestion des priorités ainsi que la sélection des requêtes et nous l'avons validé en utilisant les traces Prezi [33]. L'origine des données Prezi est liée à une compétition des ingénieurs pour appliquer leur connaissance en contrôle et en théories en files d'attente sur les problèmes réels. L'ensemble de données de Prezi est basé sur trois classes : générale (general), exportation (export) et URL. Chaque classe représente une catégorie d'application, et nous pouvons citer par exemple la classe URL qui représente les applications WEB. Notre simutaion considère l'exemple et la configuration représentés dans la Table 4.2 pour neuf requêtes.

Classe	Date	Durée	Priorité initiale
0	0	0.512	0.8
0	1	5.127	0.8
0	1	15.382	0.8
0	0	0.485	0.8
0	0	1.264	0.8
0	0	1.261	0.8
1	1	4.213	0.5
1	0	0.707	0.5
2	0	0.728	0.2

Table 4.2 – Extraction de 9 requêtes à partir de la trace Prezi

La première colonne représente la classe de la requête, la deuxième nous donne la "date" de la requête et la troisième représente la durée (temps d'exécution) de cette dernière. Toutes ces valeurs ont été obtenues après un pré-traitement de la trace Prezi originale. Après, la simulation suit l'algorithme 1. En fait, chaque requête active un conteneur Docker qui exécute un script Bash puis se met au repos pour la durée précisée dans la troisième colonne de la Table 4.2.
L'exécution de notre code nous donne les informations suivantes :

```
Soumission time 0 Type0 priority 0.8
Soumission time 0 Type1 priority 0.5
Soumission time 0 Type1 priority 0.5
Soumission time 0 Type0 priority 0.8
Soumission time 0 Type0 priority 0.8
Soumission time 0 Type0 priority 0.8
Soumission time 1 Type1 priority 0.5
Soumission time 1 Type0 priority 0.8
Soumission time 2 Type0 priority 0.8
Priority 2.447218 - fact min 0.300000 - fact max 0.400000
docker  run --rm -it ubuntu /bin/bash -c "sleep 0.512 ; exit; "
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
e0a742c2abfd: Pull complete
....
672363445ad2: Pull complete
Digest: sha256:84c334414e2bfdcae99509a6add166bbb4fa4041dc3fa6af08046a66fed3005f
Status: Downloaded newer image for ubuntu:latest
Priority 1.609086 - fact min 0.300000 - fact max 0.400000
docker  run --rm -it ubuntu /bin/bash -c "sleep 0.485 ; exit; "
Priority 1.399205 - fact min 0.300000 - fact max 0.400000
```

```
docker  run --rm -it ubuntu /bin/bash -c "sleep 1.264 ; exit; "
Priority 1.216700 - fact min 0.300000 - fact max 0.400000
docker  run --rm -it ubuntu /bin/bash -c "sleep 1.261 ; exit; "
Priority 0.974359 - fact min 0.200000 - fact max 0.300000
docker  run --rm -it ubuntu /bin/bash -c "sleep 5.127 ; exit; "
Priority 0.920000 - fact min 0.300000 - fact max 0.400000
docker  run --rm -it ubuntu /bin/bash -c "sleep 0.707 ; exit; "
Priority 0.885781 - fact min 0.200000 - fact max 0.300000
docker  run --rm -it ubuntu /bin/bash -c "sleep 15.382 ; exit; "
Priority 0.800000 - fact min 0.300000 - fact max 0.400000
docker  run --rm -it ubuntu /bin/bash -c "sleep 0.728 ; exit; "
Priority 0.605000 - fact min 0.200000 - fact max 0.300000
docker  run --rm -it ubuntu /bin/bash -c "sleep 4.213 ; exit; "
```

D'après ces informations, nous pouvons remarquer le tri des requêtes selon leurs temps d'arrivée, la mise à jour des requêtes une fois qu'une nouvelle requête est arrivée, la sélection de la requête ayant une durée égale à 0.512 unités de temps, le téléchargement d'une image Ubuntu, etc.

4.4.2 Tests unitaires

Parmi les étapes les plus importantes dans le développement informatique nous avons la partie des tests. Le fait d'écrire des tests pour le code que nous avons implémenté est l'une des méthodes qui peut assurer la qualité et améliorer la fiabilité.

Dans ce contexte, le langage Go offre un paquet nommé "testing" qui fournit un support pour les tests automatisés des paquets Go. Ce paquet de test est destiné pour être utilisé avec la commande "**go test**" qui sert à automatiser l'exécution de toute fonction de la forme :

func **TestXxx** (*testing.T)

Où Xxx peut être n'importe qu'elle chaîne alphanumérique dont la première lettre est en majuscule qui servira à identifier la routine de test. Afin de créer une suite de test, il suffit de créer un fichier dont le nom se termine par **_test.go** qui contiendra une ou plusieurs fonction(s) de tests décrite(s) précédemment. Après, il faut mettre ce fichier dans le même paquet désirant le tester. En outre, le fichier de test va être exclu des compilations régulières du paquet mais il sera inclus lorsque la commande "go test" sera exécutée.

Dans les tests que nous avons créés pour tester le bon fonctionnement de notre stratégie, nous nous sommes basés toujours, à titre d'exemple, sur un conteneur de classe Premium ayant les caractéristiques suivantes :

- Type =0
- Priority=0.8

- FactorMin=0.3
- FactorMax=0.4

Et les deux autres attributs : MinCpus et MaxCpus sont initialisés à zéro puis ils auront une valeur en appliquant les formules citées dans les parties précédentes. Les fonctions de tests qui ont été écrites simulent deux cas :

- Premier cas : les nœuds de la grappe ont un nombre différent de ressources (CPU et RAM)(voir Annexe B). L'exemple testé est représenté par la figure 4.7.

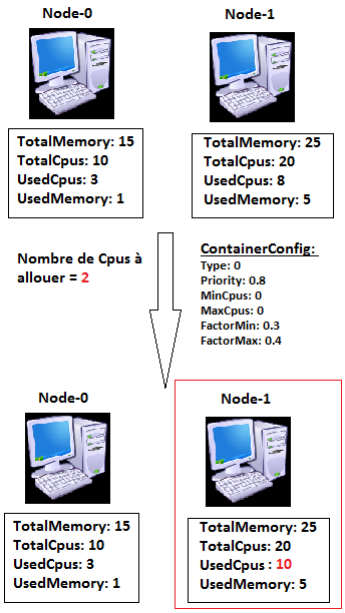

FIGURE 4.7 – Tests unitaires pour des nœuds de tailles différentes

- Deuxième cas : les nœuds de la grappe ont le même nombre de ressources. Un exemple de ce cas est représenté par la figure 4.8.

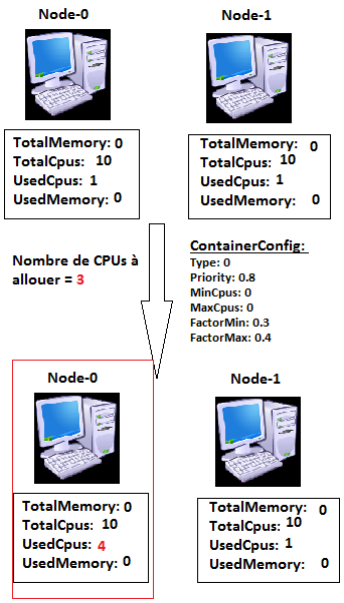

FIGURE 4.8 – Tests unitaires pour des nœuds de même taille

Ces tests nous permettent de nous assurer du bon fonctionnement de notre stratégie. C'est à dire de vérifier que l'ordonnanceur utilisé par le gestionnaire swarm choisisse la machine attendue pour exécuter le conteneur. Le principe est simple et peut être résumé comme suit :

1. Nous créons une liste des nœuds afin de simuler une grappe ;
2. Nous créons un conteneur dont la configuration est initialisée à zéro puis il aura la configuration de la classe "premium" sauvegardée dans le fichier JSON ;
3. Nous lançons l'ordonnanceur avec notre stratégie pour vérifier quel nœud a été sélectionné.

De plus nous avons mesuré le temps écoulé entre la création et l'exécution du conteneur. Ces mesures ont été effectuées sur un ordinateur dont les caractéristiques sont les suivantes :

- Système d'exploitation : ubuntu 16.04 LTS 64 bits
- Processeur : intel core i5 CPU 2.30 GHz x4
- Mémoire : 3.7 Gio
- Disque : 310,8 Go

Les résultats sont représentés dans le tableau ci-dessous :

	Nœuds de tailles différentes	Nœuds de même taille
Valeur moyenne (µs)	42.8158	42.44
Ecart type (µs)	8.324672	16.82904

Table 4.3 – Valeur moyenne et écart type du temps d'exécution

Nous avons calculé la valeur moyenne de dix valeurs de mesure de temps d'exécution parce que la valeur de cette dernière n'était pas fixe. Puis, nous avons calculé l'écart type pour prendre une idée sur la dispersion autour de la moyenne et il nous sera utile pour comparer la dispersion entre ces deux cas qui ont approximativement la même moyenne.

D'après ces résultats, nous remarquons une variation du temps d'exécution qui revient probablement à la précision du temps de mesure. De plus, la moyenne est pratiquement la même pour les deux situations. Donc, grâce à l'écart type, nous pouvons dire que la dispersion autour de la moyenne est plus étroite dans le cas des nœuds de tailles différentes.

Conclusion

Dans ce chapitre, nous avons présenté l'algorithme générique qui a été proposé ainsi que les démarches que nous avons adoptées pour réussir à l'adapter dans le contexte des conteneurs Docker, et ceci avec les moindres modifications possibles. Il en résulte une nouvelle version de Docker Swarm contenant une quatrième stratégie de placement de conteneurs.

Conclusion

Le problème des fournisseurs des services de l'informatique en nuage est de trouver comment satisfaire le maximum possible de requêtes soumises en ligne par leurs clients. De plus, comment sélectionner les machines les plus satisfaisantes à les exécuter tout en réduisant la consommation énergétique de l'infrastructure.

Dans ce contexte, nous avons proposé un système d'ordonnancement des requêtes basé sur trois étapes qui sont : ordonnancement des conteneurs, allocation des ressources et affectation de ces conteneurs. Dans notre stage, nous avons travaillé sur l'implémentation de cet algorithme dans le contexte de conteneurs Docker. Ce qui en résulte est une nouvelle version de l'ordonnanceur Docker Swarm.

La nouveauté de notre stratégie est le calcul dynamique du nombre de CPUs à allouer au conteneur à créer. Nous tenons compte des ressources libres au niveau de l'infrastructure, des paramètres de la classe SLA du conteneur et de sa priorité. De plus, nous utilisons une heuristique binpacking pour choisir la machine qui va l'exécuter.

Nous avons fait les expériences nécessaires pour valider notre approche et nous avons discuté avec AlterWay, le coordinateur du projet Wolphin, sur la possibilité d'implémenter notre stratégie dans Swarmkit, le projet le plus récent de Docker.

Enfin, nous pensons qu'on peut améliorer le système que nous avons proposé en ajoutant une méthode de consolidation. Cette dernière permettra notre système de décider le nombre de machines actives et par suite de réduire la consommation énergétique de l'infrastructure. De plus, nous avons planifié de faire des expérimentations sur le cluster OpenStack de AlterWay. De cette façon, nous pouvons mieux valider notre travail.

Bibliographie

[1] *IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016*. IEEE, 2016.

[2] abtechnologies. https ://abtec.fr/virtualisation-definition/, consulté le 02 mars 2017.

[3] Equipe AOC. http ://lipn.fr/fr/aoc-2/actualites/3145-projet-wolphin-2-0, consulté le 01 mars 2017.

[4] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[5] Kubernetes Authors. https ://kubernetes.io/docs/concepts/workloads/pods/pod/, consulté le 27 février 2017.

[6] Daniel Balouek-Thomert, Arya K. Bhattacharya, Eddy Caron, Karunakar Gadireddy, and Laurent Lefèvre. Parallel differential evolution approach for cloud workflow placements under simultaneous optimization of multiple objectives. In *IEEE Congress on Evolutionary Computation, CEC 2016, Vancouver, BC, Canada, July 24-29, 2016*, pages 822–829, 2016.

[7] Anton Beloglavoz and Rajkumur Buyya. Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints. *IEEE Transactions On Parallel And Distributed Systems*, 24, Juillet 2013.

[8] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Wiley Interscience*, pages 1–24, 2012.

[9] Laura Bernard-Reymond. Les nouveautés docker spécialement analysées et décryptées par nos experts. http ://treeptik.fr/language/fr/les-nouveautes-docker-specialement-analysees-et-decryptees-par-nos-experts-dockercon16-jour1/, Juin 2016, consulté le 04 avril 2017.

[10] Abhishek Chawla and Aneesh Devasthale. Build with aws. https ://www.slideshare.net/abhishekchawla12/buildwithaws-70400062, consulté le 01 mai 2017.

[11] Alain Clapaud. Docker vs vmware vs kvm : comparatif. http ://www.journaldunet.com/solutions/cloud-computing/vmware-vs-docker-vs-kvm-comparatif/, November 2014, consulté le 03 mars 2017.

[12] CloudMagazine. http ://www.cloudmagazine.fr/analyse/docker-casse-la-machine-virtuelle, consulté le 21 mars 2017.

[13] Hongchao Deng. Improving kubernetes scheduler performance. *Core OS*, 11, Février 2016.

[14] Docker. Swarm command-line reference. https ://docs.docker.com/swarm/reference/, consulté le 6 mars 2017.

[15] 2016 Docker Core Engineering July 28. Docker built-in orchestration ready for production : Docker 1.12 goes ga. https ://blog.docker.com/2016/07/docker-built-in-orchestration-ready-for-production-docker-1-12-goes-ga/, consulté le 4 mai 2017.

[16] Docker Documentation. Swarm filters. https ://docs.docker.com/v1.10/swarm/scheduler/filter/, consulté le 10 mars 2017.

[17] Kevin Fishner. Lessons learned from scheduling one million containers with hashicorp nomad. *infoq*, pages 5–11, Avril 2016.

[18] I Foster and C. Kesselman. The globus project : a status report. In *Heterogeneous Computing Workshop. (HCW 98) Proceedings. 1998 Seventh*, pages 4–18, Mar 1998.

[19] Noureddine GRASSA. http ://n.grassa.free.fr/cours/cours_virtualisation_et_Cloud, consulté le 20 mars 2017.

[20] Armand Grillet. Comparaison of containers schedulers. *medium*, février 2016.

[21] Anwar Hassen. A survey of docker swarm scheduling strategies. tps ://wiki.aalto.fi/download/attachments/116662239/slide_docker.pdf ?version=2&modifica December 2016, consulté le 22 mars 2017.

[22] Fabien Hermenier. Ordonnancement contrôlé de migrations à chaud. *INRIA*, Juillet 2015.

[23] Fabien Hermenier, Julia Lawall, Jean Marc-Menaud, and Gilles Muller. Dynamic consolidation of highly available web applications. *Inria*, 26, Février 2011.

[24] Stephan J.Bigelow. http ://www.lemagit.fr/conseil/Quelle-est-la-difference-entre-la-conteneurisation-et-la-virtualisation, consulté le 05 mars 2017.

[25] Vincent Kherbache, Éric Madelaine, and Fabien Hermenier. Scheduling live-migration for fast, adaptable and energy-efficient relocation operation. *IEEE*, 2015.

[26] Arnaud Mazin. Stratégies de placement de conteneurs docker (partie 1). *blog.octo.com*, Mars 2016.

[27] Arnaud Mazin. Stratégies de placement de conteneurs docker (partie 2). *blog.octo.com*, Octobre 2016.

[28] Francis MILLOT. Docker – on ne virtualise plus, on «containerise». http ://www.virtu-desk.fr/blog/le-cloud/docker-on-ne-virtualise-plus-on-containerise.html, April 2015, consulté le 13 avril 2017.

[29] Sivaram Mothiki. Schedulers, part2 : Kubernetes. https ://deis.com/blog/2016/schedulers-pt2-kubernetes/, Avril 2016, consulté le 03 avril 2017.

[30] B. Nicolae, P. Riteau, and K. Keahey. Transparent throughput elasticity for iaas cloud storage using guest-side block-level caching. In *7th International Conference on Utility and Cloud Computing (UCC)*, 2014.

[31] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo NĊalheiros, and Rajkumar Buyya. Efficient virtual machine sizing for hosting containers as a service. *IEEE*, 8, 2015.

[32] Fabio Lopez Pires and Benjamin Baran. Virtual machine placement literature review. *arXiv*, pages 1–11, 2015.

[33] Prezi. http ://prezi.com/scale/, consulté le 01 juin 2017.

[34] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco documentation. http ://www.choco-solver.org/, consulté le 23 mars 2017.

[35] Margaret Rouse. http ://www.lemagit.fr/definition/Kubernetes, consulté le 05 mars 2017.

[36] Jahanzeb Sherwani, Nosheen Ali, Nausheen Lotia, Zahra Hayat, and Rajkumar Buyya. Libra : An economy driven job scheduling system for clusters. In *6th Conference on High Performance Computing (HPC Asia)*, 2002.

[37] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. The easy - loadleveler api project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS '96, 1996.

[38] Ralph Squillace. https ://docs.microsoft.com/fr-fr/azure/virtual-machines/virtual-machines-linux-containers, 2016, consulté le 16 mars 2017.

[39] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Beowulf cluster computing with linux. In *Beowulf Cluster Computing with Linux*, chapter Condor : A Distributed Job Scheduler, pages 307–350. MIT Press, Cambridge, MA, USA, 2002.

[40] TiagoC.Ferreto, MarcoA.S.Netto, RodrigoN.Calheiros, and CésarA.F.DeRose. Server consolidation with migration control for virtualized data centers. *Elsevier*, pages 1–8, May 2011.

[41] J.D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3) :384 – 393, 1975.

[42] Steven J. Vaughan-Nichols. Containers vs. virtual machines : How to tell which is the right choice for your enterprise. https ://www.itworld.com/article/2915530/virtualization/containers-vs-virtual-machines-how-to-tell-which-is-the-right-choice-for-your-enterprise.html, April 2015, conusluté le 09 mars 2017.

[43] Rajasekharan Vengalil. Using docker swarm clusters on azure. http ://blogorama.nerdworks.in/using-docker-swarm-clusters-on-azure/, consulté le 28 avril 2017.

[44] Alter Way. https ://www.alterway.fr/wolphin-2-0-laureat-du-fui-22/, août 2016, consulté le 24 février 2017.

[45] Minxian Xu, Wenhong Tian, and Rajkumar Buyya1. A survey on load balancing algorithms for virtual machines placement in cloud computing. *arXiv*, pages 1–22, 2017.

[46] Weiping Zhu and C.F. Steketee. An experimental study of load balancing on amoeba. In *First Aizu International Symposium Parallel Algorithms/Architecture Synthesis*, pages 220–226, Mar 1995.

[47] The apache software foundation. mesos, apache `http://mesos.apache.org/`, consulté le 05 avril 2017.

[48] Kubernetes scheduler `https://kubernetes.io/`, consulté le 22 mai 2017.

Annexe A

Docker

A.1 Dockerfile

L'exemple qui suit permet de lancer une image personnalisée de Whalesay qui est basé sur une image Ubuntu.

FROM docker/whalesay :latest

RUN apt-get -y update && apt-get install -y fortunes

=> Rafraîchir la liste des paquets disponible sur l'image et installer les "fortunes".

CMD /usr/games/fortune -a | cowsay

=> CMD indique à l'image quelle commande doit exécuter après la fin de l'installation de ses environnements. Ici, elle exécute "fortune -a" et redirige la sortie vers la commande "cowsay".

Il suffit après d'enregistrer le fichier Dockerfile et exécuter la commande "docker build".

A.2 Docker Swarm

A.2.1 Stratégie Binpack

Dans cette section, nous donnons comme exemple le code de l'une des stratégies de classement utilisées par Docker Swarm. C'est la stratégie Binpack qui favorise le nœud ayant le plus grand nombre de conteneurs afin de minimiser le nombre de machines actives et maximiser l'utilisation des ressources disponibles.

```
package strategy
import (
"sort"
"github.com/docker/swarm/cluster"
"github.com/docker/swarm/scheduler/node"
)

// BinpackPlacementStrategy exécute le conteneur
sur le nœud contenant le grand nombre de conteneurs dans le cluster.
BinpackPlacementStrategy struct {
}

// Initialiser une BinpackPlacementStrategy.
func (p *BinpackPlacementStrategy) Initialize() error {
return nil
}

// Name retourne le nom de la stratégie.
func (p *BinpackPlacementStrategy) Name() string {
return "binpack"
}
// RankAndSort ordonne les nœuds basée sur la stratégie binpack
strategy appliquée à la configuration du conteneur.
func (p *BinpackPlacementStrategy) RankAndSort
(config *cluster.ContainerConfig, nodes []*node.Node)
([]*node.Node, error) {
// Pour binpack, un  nœud "healthy" doit augmenter son poids
//pour augmenter sa chance d'être sélectionnée.
// mettre "healthFactor" à 10 pour avoir "health degree" dans [0, 100]
overpower cpu + memory
const healthFactor int64 = 10
weightedNodes, err := weighNodes(config, nodes, healthFactor)
if err != nil {
return nil, err
}
sort.Sort(sort.Reverse(weightedNodes))
output := make([]*node.Node, len(weightedNodes))
for i, n := range weightedNodes {
output[i] = n.Node
}
return output, nil
}
```

A.2.2 Filtre d'affinité

Les filtres sont utilisés par Docker Swarm afin de filtrer les nœuds du cluster donnant comme résultat une liste des nœuds aptes à exécuter le conteneur. Cette liste sera l'entrée de la stratégie utilisée par l'ordonnanceur. Swarm fournit plusieurs filtres y compris : "AffinityFilter" qu'on donne dans cette section son implémentation avec le langage Go du Google.

```
package filter
import (
"fmt"
"strings"
log "github.com/Sirupsen/logrus"
"github.com/docker/swarm/cluster"
"github.com/docker/swarm/scheduler/node"
)
// AffinityFilter sélectionne seulement les nœuds basés sur les autres conteneurs sur le nœud.
type AffinityFilter struct {
}
// Name retourne le nom du filtre
func (f *AffinityFilter) Name() string {
return "affinity"
}
// Le filtre est exporté
func (f *AffinityFilter) Filter (config *cluster.ContainerConfig,
 nodes []*node.Node, soft bool) ([]*node.Node, error) {
affinities, err := parseExprs(config.Affinities())
if err != nil {
return nil, err
}
for _, affinity := range affinities {
if !soft && affinity.isSoft {
continue
}
log.Debugf("matching affinity: \%s\%s\%s (soft=\%t)",
affinity.key, OPERATORS[affinity.operator],
affinity.value, affinity.isSoft)
candidates := []*node.Node{}
for _, node := range nodes {
switch affinity.key {
case "container":
containers := []string{}
for _, container := range node.Containers {
if len(container.Names) > 0 {
containers = append(containers, container.ID,
strings.TrimPrefix(container.Names[0], "/"))
}
}
```

```
if affinity.Match(containers...) {
candidates = append(candidates,node)
}
case "image":
images := []string {}
for _,image := range node.Images {
images = append(images, image.ID)
images = append(images, image.RepoTags...)
for _, tag := range image.RepoTags {
repo, _ := cluster.ParseRepositoryTag(tag)
images = append(images, repo)
}
}
if affinity.Match(images...) {
candidates = append(candidates, node)
}
default:
labels := []string{}
for _, container := range node.Containers {
labels = append(labels, container.Labels[affinity.key])
}
if affinity.Match(labels...) {
candidates = append(candidates,node)
}
}
}
if len(candidates)}== 0 {
return nil, fmt.Errorf("unable to find a node that satisfies
the affinity \%s\%s\%s", affinity.key,
OPERATORS[
affinity.operator
],
affinity.value)
}

nodes = candidates
}

return nodes, nil
}

// GetFilters retourne la liste des affinités trouvées dans la configuration du conteneur.
func (f *AffinityFilter) GetFilters(
config *cluster.ContainerConfig
) ([]string, error) {
allAffinities := []string{}
affinities, err := parseExprs(config.Affinities())
```

```
if err != nil {
return nil, err
}

for _,affinity := range affinities {
allAffinities = append(allAffinities, fmt.Sprintf("\%s\%s\%s
(soft=\%t)", affinity.key, OPERATORS[affinity.operator],
affinity.value, affinity.isSoft))
}
return allAffinities, nil
}
```

A.3 Manipuler un Cluster avec le mode swarm de Docker

Avant de commencer la manipulation d'un cluster Swarm, il faut tout d'abord :

- Installer Virtualbox sur la machine locale.
- Activer Virtualization technology Vt-x et Virtualization Technology Directed I/O Vt-d dans les paramètres du BIOS parce qu'elles sont désactivées par défaut.
- Enregistrer les modifications et redémarrer la machine.

Maintenant, voici les instructions utiles pour créer le cluster :

1. Créer les machines :
   $docker-machine create - -driver virtualbox swarm-master
   $docker-machine create - -driver virtualbox node1
   $docker-machine create - -driver virtualbox node2
2. Vérifier que les machines ont été bien créées :
   $docker-machine ls
3. Récupérer les variables d'environnement d'une machine :
   $docker-machine env swarm-master
   $eval $(docker-machine env manager)

4. Se connecter à la machine :
   $docker-machine ssh swarm-master

5. Initialiser Swarm sur cette machine :
   $docker swarm init - -advertise-addr 192.168.99.100

6. Ajouter un nœud au Swarm :
   $docker-machine ssh node1
   $ docker swarm join
   - -token SWMTKN-1-0q6bggjftx6dklgld6f92ech9wty48newo7rfjdni8hr0bes2h
   -dpr4vh1xykxmb8jut4b2vamgb
   192.168.99.100 :2377
   de même pour le 2 ème nœud.

7. Vérifier que les nœuds sont bien ajoutés au Swarm :
   $docker-machine ssh swarm-master
   $docker node ls

8. Pour obtenir des infos sur un nœud, il suffit de se connecter à ce dernier et taper :
   $docker info
   Enfin, on peut créer des services en tapant la commande suivante :
   $ docker service create - -name app alpine ping docker.com
   => Ce service se base sur l'image alpine et il exécute la commande "ping docker.com".
   Et pour afficher la liste des services en cours, il suffit de taper :
   $ docker service ls

A.4 Manipuler un Cluster Docker Swarm

Pour créer un cluster Docker Swarm, on va utiliser des machines virtuelles(VMs) sur virtualbox. Afin de provisionner ces machines sur virtualbox, on va utiliser docker machine. Pour l'installer, il faut exécuter les commandes suivantes dans l'ordre :

- $sudo apt-get remove docker docker-engine

- $sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-properties-common
- $curl -fsSL https ://download.docker.com/linux/debian/gpg | sudo apt-key add -
- $sudo add-apt-repository "deb [arch=amd64] https ://download.docker.com/linux/debian $(lsb_release -cs) stable"
- $sudo apt-get update
- $sudo apt-get install docker-ce
- $wget -q https ://github.com/docker/machine/releases/download/v0.10.0/docker-machine-‘uname -s‘-‘uname -m‘

Pour vérifier que Docker machine a été bien installé, on peut taper la commande : "$sudo docker-machine version".
De plus de l'installation de Docker CE et Docker machine, les VMs qui vont être créées utilisent comme driver virtualbox mais il faut bien noter qu'il n'est pas possible de créer ces Vms dans une machine virtuelle(il faut obligatoirement une machine physique sur laquelle il faut activer les paramètres de virtualisation dans son Bios). Pour installer ce driver, on doit exécuter les commandes ci-dessous :

- $wget -q -O- http ://download.virtualbox.org/virtualbox/debian/ oracle_vbox_2016.asc | sudo apt-key add -
- $echo "deb http ://download.virtualbox.org/virtualbox/debian $(lsb_release -sc) contrib" | sudo tee /etc/apt/sources.list.d/virtualbox.list
- $sudo apt-get update
- $echo "deb http ://download.virtualbox.org/virtualbox/debian $(lsb_release -sc) contrib" | sudo tee /etc/apt/sources.list.d/virtualbox.list && wget -q http ://download.virtualbox.org/virtualbox/debian/oracle_vbox.asc -O- | sudo apt-key add - && sudo apt-get update && sudo apt-get install virtualbox-5.1
- $sudo usermod -G vboxusers -a $USER

- $sudo /etc/init.d/vboxdrv setup

Docker machine va instancier des VMs dans virtualbox à lesquelles il peut se connecter à travers un ssh. Dans le cas où les paramètres du firewall de la machine physique sont strictes, le client docker ne pourra pas communiquer au démon docker via le port 2376 sur la machine virtuelle. Donc il faut s'assurer qu'il est ouvert. Pour faire ceci, il suffit de taper la commande suivante :

- $sudo iptables -A INPUT -p tcp - -dport 2376 -j ACCEPT

Dans l'étape qui suit on va créer notre cluster de deux nœuds (un nœud gestionnaire et un nœud travailleur) :

- $sudo docker run swarm create //générer un jeton (service de découverte pour le cluster)
  a5ff7146c2e31f520147a838d1d659d6 //ceci est le jeton qui a été généré
- $export discotoken=a5ff7146c2e31f520147a838d1d659d6 // mettre le jeton dans une variable discotoken
- $sudo docker-machine create -d virtualbox - -swarm - -swarm-master - -swarm-strategy "binpack" - -swarm-discovery token ://$discotoken swarm-manager //crétaion du nœud gestionnaire
- $sudo docker-machine create -d virtualbox - -swarm - -swarm-discovery token ://$discotoken node-01
- $sudo docker-machine ls //afficher la liste des nœuds dans le cluster
- $sudo docker info //pour afficher les caractéristiques du cluster

Maintenant, on va identifier l'adresse ip du nœud gestionnaire qui sera notre interface de communication avec le cluster :

- $sudo docker-machine ip swarm-manager
  192.168.99.100 //celle là est l'adresse ip du nœud swarm-manager

- $export managerip=192.168.99.100 //utiliser cette adresse ip comme variable d'environnement

Il reste maintenant à tester qu'on peut se connecter au docker api du notre cluster via TLS en utilisant cert :

- $sudo curl "https ://$managerip :3376/images/json" - -cert /home/bejaoui/ .docker/machine/machines/swarm-manager/cert.pem - -key /home/bejaoui/ .docker/machine/machines/swarm-manager/key.pem - -cacert /home/bejaoui/ .docker/machine/machines/swarm-manager/ca.pem

Puis, il faut copier les certificats et les fichiers associés dans le nœud gestionnaire :

- $sudo docker-machine ssh swarm-manager
- $rm -fr /home/docker/client-certs
- $mkdir /home/docker/client-certs
- $exit
- $sudo docker-machine scp /home/bejaoui/.docker/machine/machines/swarm-manager/cert.pem swarm-manager :client-certs/cert.pem
- $sudo docker-machine scp /home/bejaoui/.docker/machine/machines/swarm-manager/key.pem swarm-manager :client-certs/key.pem
- $sudo docker-machine scp /home/bejaoui/.docker/machine/machines/swarm-manager/ca.pem swarm-manager :client-certs/ca.pem

Dans ce qui suit on va vérifier si on peut se connecter au cluster en dehors du nœud gestionnaire à travers le TLS :

- $sudo docker-machine ssh swarm-manager docker - -tlsverify - -tlscacert= /home/docker/client-certs/ca.pem - -tlscert=/home/docker/client-certs/cert.pem - -tlskey=/home/docker/client-certs/key.pem -H "tcp ://$managerip :3376" info //on aura comme affichage toutes les informations sur le cluster créé comme ses nœuds,

le nombre de conteneurs et leurs états, la stratégie d'ordonnancement de conteneurs utilisé par le gestionnaire swarm, les filtres utilisés, etc.

Finalement, on peut lancer un conteneur dans le cluster en exécutant la commande :

- $sudo docker-machine ssh swarm-manager docker - -tlsverify - -tlscacert=/home/docker/client-certs/ca.pem - -tlscert=/home/docker/client-certs/cert.pem - -tlskey=/home/docker/client-certs/key.pem -H "tcp ://$managerip :3376" run hello-world

Annexe B

Expériences

B.1 Exemple des tests unitaires

L'exemple dont le code est mis ci-dessous représente un test unitaire pour le cas des nœuds de tailles différentes :

```
//Choisir la stratégie à utiliser ici HelloPlacementStrategy est notre stratégie.
s:=&HelloPlacementStrategy{}
//Simuler un cluster avec deux nœuds.
nodes := []*node.Node{
create_Node(
fmt.Sprintf("node-0"), 15, 10, 3, 1), create_Node(
fmt.Sprintf("node-1"), 25, 20, 8, 5
    ),}
start := time.Now()
//Créer un conteneur
config:=createConf(0,0)
//'node' est la machine sélectionnée pour exécuter le conteneur.
node:=selectFirstNode(
t, s, config, nodes
)
//Simuler le lancement du conteneur sur le nœud choisi.
assert.NoError(t,node.AddContainer
(
create_Container(
fmt.Sprintf("c%d",0),config)
)
)
fin:=time.Since(start)
//Afficher le temps écoulé entre la création jusqu'à l'exécution du conteneur.
fmt.Println("elapsedtime:",fin)
//Vérifier que le coneteneur a été bien ajouté sur le nœud choisi.
```

```
//Vérifier que l'ordonnanceur a choisi la machine susceptible d'être sélectionnée.
assert.Equal(t, len(nodes[1].Containers),1
)
```