MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE ABDELHAMID IBN BADIS - MOSTAGANEM

Faculté des Sciences Exactes et d'Informatique

Département de Mathématiques et informatique

Filière : Informatique

MEMOIRE DE FIN D'ETUDES

Pour l'Obtention du Diplôme de Master en Informatique

Option : **Réseaux et Systèmes**

Présenté par :

Zoudji Younes

Mechenguel Mohamed Hakim

THÈME :

Validation et mesure de performances des stratégies d'ordonnancement pour Kubernetes

Soutenu le : 09/10/2023

Devant le jury composé de :

Mme.MAGHNI-SANDID Zoulikha	Université de Mostaganem	Présidente
Mr.MECHAOUI Moulay Driss	Université de Mostaganem	Examinateur
Mme.FILALI Fatima Zohra	Université de Mostaganem	Encadrante

Année Universitaire 2022-2023

Résumé

Kubernetes étant une plateforme d’orchestration des conteneurs, elle automatise de nombreux processus manuels impliqués dans le déploiement, la gestion et la mise à l’échelle des applications conteneurisées. Toutes ces taches en appuyant sur son ordonnanceur qui utilise sa stratégie par défaut, cette dernière montre leurs limites dans des conditions dynamiques et changeantes. Donc plusieurs solutions ont été publiées faisant l’ordonnancement d’une façon plus complexe.

Le but de notre étude consiste à tester une nouvelle stratégie d’ordonnancement pour kubernetes et la comparer avec les autres travaux existants en validant leurs performances.

Mot clés : kubernetes, conteneur, ordonnancement.

Abstract

As Kubernetes is a container orchestration platform, it automates many manual processes involved in deploying, managing, and scaling containerized applications. All these tasks by pressing its scheduler which uses its default strategy, the latter shows their limits in dynamic and changing conditions. So several solutions have been published making the scheduling in a more complex way.

The goal of our study is to test a new scheduling strategy for Kubernetes and compare it with other existing works by validating their performance.

Keywords: kubernetes, container, scheduling.

ملخص

نظرًا لأن Kubernetes عبارة عن نظام أساسي لتنسيق الحاويات، فإنه يعمل على أتمتة العديد من العمليات اليدوية المرتبطة بنشر التطبيقات الموجودة في حاويات وإدارتها وتوسيع نطاقها وذلك عن طريق استعمال المجدول الخاص به الذي يستخدم استراتيجيته الافتراضية، هذه الأخيرة تظهر حدودها في الظروف الديناميكية والمتغيرة. لذلك تم نشر العديد من الحلول للقيام بالجدولة بطريقة أكثر تعقيدًا.

الهدف من دراستنا هو اختبار استراتيجية جدولة جديدة لـ Kubernetes ومقارنتها بالأعمال الأخرى الموجودة من خلال التحقق من صحة أدائها.

كلمات مفتاحية: حاوية ،جدولة ، kubernetes

Remerciements

La réalisation de ce mémoire n'aurait pas été possible sans le secours de certaines personnes à qui je voudrais adresser des remerciements particuliers.

Dans un premier temps, J'adresse mes sincères remerciements à mon directeur de mémoire, Mme FILALI Fatima Zohra. Son orientation, son aide, ses conseils et son encadrement ont permis la réalisation de ce mémoire.

Je remercie, ensuite, tous les intervenants professionnels et l'équipe pédagogique de la faculté des sciences exactes et informatique. Ils m'ont aidé dans les démarches administratives et m'ont assuré une formation de qualité.

Enfin, ma gratitude va à mes proches et je les remercie de m'avoir soutenu et encouragé financièrement et mentalement.

Mohamed Hakim Mechenguel

Remerciements

Avant toute personne, Je remercie ALLAH tout puissant de m'avoir accordé la puissance et la volonté au cours de ce travail.

J'adresse mes sincères remerciements à notre encadrante Madame Filali Fatima Zohra pour ses conseils et son aide tout au long de ce projet.

Je tiens à exprimer toute ma gratitude à tous les membres de jury, pour avoir bien voulu juger notre travail et à tous les enseignants du département informatique de la faculté des sciences exactes et informatique qui ont contribué à notre formation.

À ma chère mère qui m'a toujours soutenu et encouragé tout au long de mon parcours académique et à tous ceux qui m'ont renforcé.

Younes Zoudji

Liste des figures

Figure N°	Titre de la figure	Page
Figure 1.1	Différence de fonctionnement entre un conteneur et une machine virtuelle	7
Figure 2.1	Architecture de docker	12
Figure 2.2	Swarm Cluster	15
Figure 2.3	Architecture de Kubernetes	18
Figure 3.1	Figure indiquant les symboles et leurs descriptions	33
Figure 3.2	Algorithme du clustering hiérarchique	38
Figure 3.3	Algorithme de colonies de fourmi	43
Figure 3.4	Architecture de la solution d’ordonnancement	44
Figure 4.1	Bibliothèques clientes officielles supportées par Kubernetes	49
Figure 4.2	Résultat d’ordonnancement avec la stratégie par défaut	50
Figure 4.3	L'étiquette du nœud 2	49
Figure 4.4	Configuration des pods avec NodeAffinity	51
Figure 4.5	Résultat d’ordonnancement de NodeAffinity	51
Figure 4.6	Configuration des pods avec Pod AntiAffinity	52
Figure 4.7	Résultat d’ordonnancement de Pod AntiAffinity	52
Figure 4.8	Le dossier d’ordonnanceur aléatoire	53
Figure 4.9	Le Dockerfile d’ordonnanceur aléatoire	53
Figure 4.10	Le répertoire dockerhub de l’image docker	54
Figure 4.11	Construction de l’image docker	54
Figure 4.12	Le partage d’image docker	54
Figure 4.13	Téléchargement de l’image docker	55
Figure 4.14	Fichier de configuration de l’ordonnanceur aléatoire	56

Figure 4.15	Fichier de configuration de pod	57
Figure 4.16	Résultat d’ordonnancement aléatoire	57
Figure 4.17	Le dossier d’ordonnanceur K8s-BWA	57
Figure 4.18	Le Dockerfile d’ordonnanceur K8s-BWA	58
Figure 4.19	Le répertoire dockerhub de l’image d’ordonnanceur	58
Figure 4.20	Les paquets échangés entre les pods de l’espace de nom resys	59
Figure 4.21	Les valeurs cpu des pods	59
Figure 4.22	Les valeurs du stockage des pods	60
Figure 4.23	Les valeurs de la ram des pods	60
Figure 4.24	Les valeurs de la bande passante des nœuds	61
Figure 4.25	Les valeurs cpu des nœuds	61
Figure 4.26	Les valeurs du stockage des nœuds	61
Figure 4.27	Les valeurs de la ram des nœuds	61
Figure 4.28	Identifiants des clusters avec leurs pods	62
Figure 4.29	Identifiants des clusters avec le meilleur placement	62
Figure 4.30	Fichier de configuration de l’ordonnanceur K8s-BWA	63
Figure 4.31	Fichier de configuration de pod	64
Figure 4.32	Résultat d’ordonnancement avec la stratégie K8s-BWA	64
Figure 4.33	L’usage de la bande passante avec la stratégie par défaut	65
Figure 4.34	L’usage de la bande passante avec la stratégie aléatoire	66
Figure 4.35	L’usage de la bande passante avec la stratégie K8s-BWA	66
Figure 4.36	L’usage du CPU avec la stratégie par défaut	67
Figure 4.37	L’usage du CPU avec la stratégie aléatoire	68
Figure 4.38	L’usage du CPU avec la stratégie K8s-BWA	68

Liste des tableaux

Tableau N°	**Titre du tableau**	**Page**
Tableau 1	Comparaison entre les travaux d'ordonnancement	28
Tableau 2	Tableau présentant la fiche technique de l'environnement physique	46

Liste des abréviations

Abréviation	Expression Complète	Page
AWS	Amazon Web Services	1
CNCF	Cloud Native Computing Foundation	1
OS	Operating System (Système d'exploitation)	1
IAAS	Infrastructure as a Service	3
PAAS	Platform as a Service	3
SAAS	Software as a Service	3
NIST	National Institute of Standards and Technology	4
VM	Virtual Machine (Machine Virtuelle)	7
CPU	Central Processing Unit	7
RAM	Random Acces Memory	7
POO	programmation orientée objet	9
API	Application Programming Interface	9
HTTP	Hypertext Transfer Protocol	9
Cgroups	Control groups	14
COE	Container Orchestration Engine	14
K8s	Kubernetes	16
IP	Internet Protocol	16
POD	Point Of Delivery	16
DNS	Domain Name System	17
YAML	YAML Ain't Markup Language	19
JSON	JavaScript Object Notation	19
ACO	Ant Colony Optimization	23

PSO	Particle Swarm Optimization	23
ProCon	Progres-Based Container Scheduling	24
ARTU	average response time of user	25
ER	error rate	25
CSML	Container Scheduling based on Machine Learning	25
PM	Machines physiques	31
NVMe	non-volatile memory express	47
RAID1	redundant array of independent disks	47

Table des matières

Introduction générale 1
Chapitre 1 Coud et Virtualisation 3
I Introduction 3
II Informatique en nuage (Cloud Computing) 3
1. Les services de l'informatique en nuage 4
III La virtualisation 5
1. Définition 5
2. Types de virtualisation 5
3. La virtualisation et le cloud 6
IV La Conteneurisation 7
1. La différence entre la conteneurisation et la virtualisation 7
2. Les Conteneurs 8
3. Les Microservices 9
V Conclusion 10
Chapitre 2 Docker, kubernetes et Ordonnancement 11
I Introduction 11
II Docker 11
1. Définition 11
2. Architecture de Docker 12
3. Fonctionnement 14
III Orchestration des conteneurs 14
1. Ordonnanceur des conteneurs 14
2. Exemple d'ordonnanceurs 15
IV Kubernetes 16
1. Définition 16
2. Historique 16
3. Les concepts de K8s 16
4. Architecture 18
5. Les composants de K8s 19
V Les travaux d'ordonnancement existants 20
1. Techniques d'ordonnancement de kubernetes 20
2. Technique ACO-PSO 23

3. ProCon 24
4. Stratégie d'ordonnancement basé sur l'apprentissage machine dans une architecture microservice (CSML) 25
VI Conclusion 26
Chapitre 3 Conception et développement 27
I Introduction 27
II Comparaison des travaux 27
III Réalisation de la solution 29
1. Une stratégie d'ordonnancement méta-heuristique basée sur l'hybridation entre ACO et PSO 29
2. Placement économe de machine virtuelle dans les centres de données cloud 31
IV Problématique 35
1. Les métriques de performance 35
2. Formule de calcul des métriques 36
V Description de la solution 36
1. Phase1 37
2. Phase 2 37
3. Phase 3 39
VI Architecture de la solution 44
VII Conclusion 44
Chapitre 4 Implémentation et tests 45
I Introduction 45
II Environnement de travail 45
1. Environnement matériel 45
2. Environnement logiciel 46
3. Langage de programmation 48
III Implémentation 49
1. Implémentation de la stratégie par défaut 49
2. Implémentation de la stratégie aléatoire 53
3. Implémentation de la stratégie proposée 57
IV Evaluation 65
1. Analyse des résultats de la bande passante 65
2. Analyse des résultats du CPU 67
V Conclusion 70

Conclusion générale 71
Bibliographie.. 73

Introduction générale

Depuis les années 2000, l'émergence de solutions de type cloud computing engendre une profonde transformation de l'industrie informatique et des pratiques organisationnelles, comme individuelles. Il s'agit d'un service à la demande pour n'importe quelle ressource et à n'importe quel instant avec une grande agilité et à moindre coût, les entreprises de toute taille utilisent le cloud pour une variété de cas d'utilisation tel que la sauvegarde des données et la messagerie. Le cloud computing rend facilement l'expansion vers de nouvelles régions et garantit un déploiement international, notamment Amazon Web Services (AWS) possède des infrastructures partout dans le monde.

Avec l'augmentation du nombre de serveurs et le gaspillage d'énergie, la virtualisation était une solution dans l'intérêt d'améliorer l'efficacité et la disponibilité des ressources et des applications dans une couche abstraite du matériel physique, elle facilite la prise en charge de nombreuses plateformes cloud, en particulier celles conçues pour un public plus large. La technologie des conteneurs offrit la capacité d'isoler virtuellement les applications à déployer en précisément de types microservice d'une façon indépendante au système d'exploitation, ces applications sont plus faciles à déplacer d'un environnement à un autre pour cela ils peuvent fonctionner sur des architectures cloud hautement distribuées.

Kubernetes est l'orchestrateur le plus populaire pour la gestion des conteneurs, il a été créé par Google qui l'ont ensuite donnée à la Cloud Native Computing Foundation (CNCF). Il prend en charge plusieurs kernel (OS) autrement dit son ordonnanceur gère les conteneurs sur différents serveurs en planifiant leurs exécution avec des stratégies d'ordonnancement mais elles sont statiques et ne peuvent pas répondre aux besoins dynamiques en ressources des applications car elles ne tiennent pas compte des caractéristiques des environnements et des applications microservices.

Récemment, l'ordonnancement des conteneurs dans les environnements cloud est devenu un sujet de recherche très important afin de proposer de nouvelles stratégies plus adaptables aux différents environnements.

Alors nous avons fait ce travail dans le but d'étudier les multiples stratégies d'ordonnancement qui suivent la stratégie par défaut de kubernetes et proposer une solution qui corrige les limites de cette dernière en prenant en compte la consommation d'énergie et de la bande passante.

Ce mémoire est composé de quatre chapitres divisés comme suit :

- Chapitre 1 : Nous parlons dans ce chapitre sur les concepts de cloud et la virtualisation, ainsi que la conteneurisation et l'architecture microservice.
- Chapitre 2 : Nous discutons dans ce chapitre sur Docker, son architecture et l'ordonnancement des conteneurs, ensuite nous présentons Kubernetes, ses concepts et son architecture. Enfin nous introduisons les différents travaux d'ordonnancement existants.
- Chapitre 3 : Dans ce chapitre nous décrivons notre solution, ses phases et les travaux dont nous sommes inspirés pour sa réalisation ainsi que les métriques de performances prises en compte pour évaluer l'ensemble des stratégies.
- Chapitre 4 : Dans ce chapitre nous parlons sur les détails de l'implémentation, l'environnement matériel et logiciels utilisés ainsi que les résultats des tests.

Chapitre 1
Cloud et Virtualisation

I Introduction

Le cloud computing a évolué en tant que paradigme informatique clé parallèlement avec le progrès rapide des entreprises dans leur parcours de transformation numérique, les entreprises recherchent des moyens d'accroître l'agilité, la continuité des activités et l'évolutivité. La technologie du cloud computing sera au cœur de toute stratégie visant à atteindre ces objectifs dans la nouvelle normalité. La virtualisation est considérée comme un backbone pour l'informatique en nuage qui est capable de découpler les applications de l'infrastructure essentiel en permettant le partage des ressources pour exécuter diverses applications de manière isolée. Ce chapitre présentera les notions essentielles sur ces différents concepts.

II Informatique en nuage (Cloud Computing)

Le Cloud Computing est un concept où les ressources sont faciles à obtenir, configurables à distance. L'accès doit être à la demande des clients et via internet ce qui permet aux périphériques (ordinateurs, téléphones mobiles…) de devenir des points d'accès pour exécuter des applications ou consulter des données qui sont hébergées sur les serveurs distants [1]. Il se caractérise par une flexibilité rapide permettant de gérer les ressources d'une manière illimitée à n'importe quel moment ou emplacement. Le service cloud peut être mesuré par son utilisation grâce au payement comme le service de stockage Google drive qui est payant pour plus de 10 Go. Le Cloud Computing se forme de plusieurs niveaux de services : IAAS (Infrastructure As A Service) : Seule l'infrastructure matérielle est externalisée. PAAS (Platform As A Service) : L'externalisation concerne l'infrastructure matérielle, les données et les applications. SAAS (Software As A Service) : inclut l'externalisation complète, la mise en fonctionnement et la maintenance.

1. Les services de l'informatique en nuage

Le NIST (National Institute of Standards and Technology) a proposé trois catégories principales des services cloud computing :

1.1. Infrastructure As A Service (IAAS)

Les services d'infrastructure cloud proposent généralement des plates-formes de virtualisation où les clients peuvent déployer leur propre logiciel sur les machines virtuelles, le contrôler et le gérer. Cela permet aux entreprises de louer les ressources plutôt, que de dépenser de l'argent pour acheter des serveurs dédiés et du matériel réseau. [2]

Des exemples de produits IAAS : Amazon1 propose S3 pour le stockage, EC2 pour la puissance de calcul et SQS pour la communication réseau à des petites entreprises.

1.2. Platform As A Service (PAAS)

Dans ce modèle, le fournisseur fournit, exécute et maintient à la fois le système d'exploitation et les ressources informatiques, le client gère et exécute le logiciel d'application sous le système d'exploitation et sur les ressources virtuelles fournies. Contrairement au SaaS qui fournit au client des applications complètes (prêtes à l'emploi), le PaaS lui donne la possibilité de concevoir, développer et tester directement des applications sur le nuage ; par conséquent, il peut contrôler le cycle de vie du logiciel. [3]

Exemples de fournisseurs PaaS : Windows Azure, Google Apps Engine et Aptana cloud.

1.3. Software As A Service (SAAS)

Le SaaS est un modèle dans lequel un fournisseur héberge des logiciels d'application, du système d'exploitation et des ressources informatiques, les clients peuvent accéder à des applications hébergées telles que Gmail et Google Docs via différents appareils. Contrairement aux logiciels traditionnels, le SaaS présente l'avantage que le client n'a pas besoin d'acheter des licences ou d'exécuter des logiciels sur son propre ordinateur. [4]

Un SaaS très connu est le courrier électronique basé sur le Web, tel que Gmail, Yaho etc.

III La virtualisation

1. Définition

La technologie de virtualisation est l'une des composantes fondamentales du cloud computing. Elle s'agit d'un ensemble de techniques qui offrent des services informatiques pour une utilisation à la demande d'une façon indépendante à l'infrastructure matérielle sous-jacente. [5]

La virtualisation repose sur trois éléments :

- L'abstraction des ressources informatiques : l'ajout d'une couche qui dissocie le système d'exploitation du matériel afin de garantir une exploitation flexible.
- La création d'environnements virtuels : chaque machine virtuelle dispose de ses propres ressources virtuelles qui sont similaires aux composants matériels, à savoir le processeur, la mémoire, le disque dur etc.
- La répartition des ressources par l'intermédiaire de différents outils par conséquent les machines virtuelles seront complètement indépendantes et pourront fonctionner sur la même machine physique.

2. Types de virtualisation

Les types de virtualisation peuvent être :

2.1. Virtualisation de serveur

Dans la virtualisation de serveur, un serveur unique exécute la tâche de plusieurs serveurs en répartissant les ressources d'un serveur dans plusieurs environnements, ce qui permet de réduire le cout et une utilisation efficace du processeur.

2.2. Virtualisation du réseau

La virtualisation de réseau peut combiner plusieurs réseaux physiques en un réseau logiciel virtuel, ou encore diviser un réseau physique en plusieurs réseaux virtuels indépendants**,** elle permet de fournir des fonctions réseau, des ressources matérielles et des ressources logicielles sous la forme d'un réseau virtuel. [6]

2.3. Virtualisation de stockage

La virtualisation de stockage permet de fournir du stockage virtuel. Elle consiste à extraire et à couvrir les fonctions internes d'un périphérique de stockage à partir de l'application hôte, des serveurs hôtes ou d'un réseau afin de faciliter la gestion du stockage indépendamment de l'application et du réseau. [7]

2.4. Virtualisation d’application

La virtualisation d’application sépare l’utilisation d’une application des environnements matériels et logiciels nécessaires à son exécution alors qu'en fait, elle s'exécute sur une machine virtuelle et est accessible par la machine locale. Ce type de virtualisation peut s’exécuter à distance ou ce qu’on appelle le streaming d’application où le serveur envoie au client tous les fichiers dont l’application a besoin pour s’exécuter et celle-ci est exécutée sur l’ordinateur client avec ses propres ressources. Exemple : Microsoft App-V. [8]

2.5. Virtualisation de poste de travail

La virtualisation de poste de travail est une technique de simulation d’un poste utilisateur pour accéder à un poste depuis un appareil connecté, à distance ou en local. Cette virtualisation est intéressante en raison de la sécurité contre les menaces et l'utilisateur peut accéder à son environnement de travail à partir de n'importe quel poste relié au serveur. [9]

2.6. Virtualisation de données

La virtualisation de données permet à une application de manipuler les données sans savoir où se trouve leur emplacement ou comment est-il leur format. Elle sert à réduire le cout de stockage car il n’est pas essentiel de répliquer ou transformer les données dans plusieurs formats et également l’intégration de données qui est utilisée par les entreprises récemment. [10]

3. La virtualisation et le cloud

La technologie de virtualisation s’appuie sur un hyperviseur afin de créer des machines virtuelles à partir des serveurs physiques rendant possibles le cloud computing en attribuant des ressources virtuelles, ainsi le cloud computing n’existerait pas sans la virtualisation.

IV La Conteneurisation

La conteneurisation est un outil nécessaire pour déployer les applications et systèmes existants en créant de nouvelles applications conçues dans le cloud, c'est une forme évolutive de la virtualisation des systèmes d'exploitation où les applications sont exécutées d'une manière isolée dans leur propre conteneur en utilisant le même système d'exploitation partagé. Elle sert à créer et déployer l'application de façon plus sécurisée et garantir la portabilité des conteneurs.

Après son émergence en 2013, Docker Engine devient la norme de l'industrie en ce qui concerne le processus de conteneurisation grâce à une approche de packaging universelle. [11]

1. La différence entre la conteneurisation et la virtualisation

Dans le cas de la virtualisation, l'isolation des VMs se fait au niveau matériel (CPU/RAM/Disque) avec un accès virtuel aux ressources de l'hôte via un hyperviseur. De plus, généralement les ordinateurs virtuels fournissent un environnement avec plus de ressources que la plupart des applications n'en ont besoin. Par contre dans le cas de la conteneurisation, l'isolation se fait au niveau du système d'exploitation. Un conteneur va s'exécuter sous Linux de manière native et va partager le noyau de la machine hôte avec d'autres conteneurs, ne prenant pas plus de mémoire que tout autre exécutable, ce qui le rend léger.

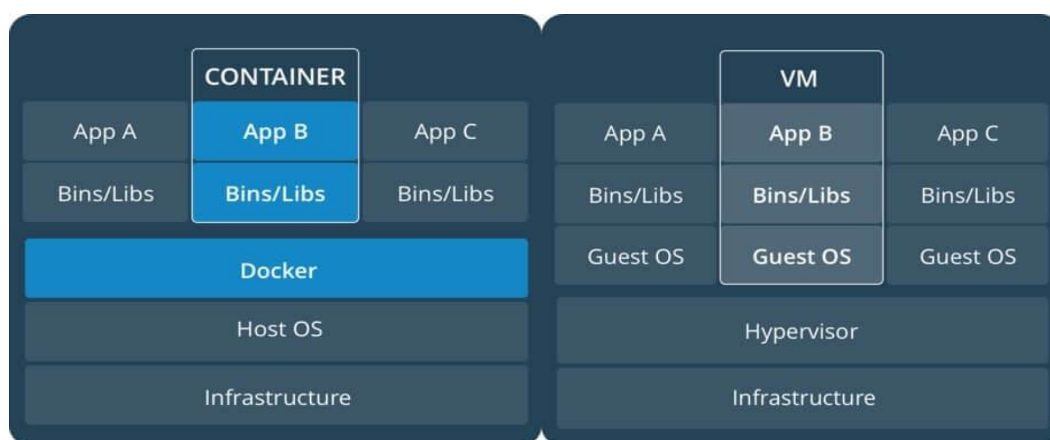

Figure 1.1 – Différence de fonctionnement entre un conteneur et une machine virtuelle [12]

1.1. Avantages de la conteneurisation par rapport à la virtualisation traditionnelle

- **Volume exploité :** la machine virtuelle elle-même intègre un système d'exploitation de la taille de gigaoctet. Ce n'est pas le cas des conteneurs. Le conteneur appelle directement le système d'exploitation pour exécuter ses appels système et exécuter ses applications. C'est beaucoup moins gourmand en ressources.
- **Le déploiement :** est l'un des points clés à considérer aujourd'hui. On peut déplacer des conteneurs d'un environnement à un autre très rapidement (en fait, c'est plus facile et plus rapide avec Docker puisqu'il suffit de partager des fichiers de configuration qui sont souvent très légers). On pourrait bien sûr faire de même pour les machines virtuelles en les déplaçant complètement d'un serveur à l'autre, mais la couche d'OS ralentit le déploiement, et en même temps le processus d'émulation des ressources physiques prendra un certain temps d'exécution, d'où le délai supplémentaire.

2. Les Conteneurs

Les conteneurs représentent une abstraction au niveau de la couche application qui regroupe le code et les dépendances et fournissent une virtualisation au niveau du système d'exploitation, afin d'isoler les processus et définir les limites d'utilisation du système pour les ressources telles que le processeur et la mémoire. Plusieurs conteneurs peuvent s'exécuter sur la même machine et partager le noyau du système d'exploitation avec d'autres conteneurs et chacun s'exécute indépendamment dans son propre environnement. [13]

2.1. Avantages et inconvénients des conteneurs [14]

Les conteneurs sont plus avantageux en termes de :

- **Portabilité** : possibilité de déplacement entre tous les systèmes qui partagent le type de système d'exploitation hôte.
- **Espace de stockage** : ils occupent moins d'espace disque et de mémoire que les machines virtuelles.
- **Disponibilité** : quel que soit l'environnement, le logiciel conteneurisé fonctionnera toujours de la même manière et il est disponible pour les applications Linux et Windows.

En dépit des avantages mentionnés, il existe des inconvénients pour les conteneurs tels que :

- Le manque d'isolement des conteneurs avec le noyau du système d'exploitation hôte.
- La menace de sécurité et flexibilité à cause de partage du système d'exploitation hôte.
- Avoir un grand nombre de conteneurs, exige des difficultés à les surveiller.

3. Les Microservices

Dans les années quatre-vingt-dix, pour créer une application, on devait posséder une architecture modulaire conforme aux principes de programmation orientée objet (POO), dans laquelle les différents modules sont combinés en un seul programme. Malgré les avantages de cette architecture comme le fait de pouvoir copier l'application empaquetée sur un serveur, elle a fait face à des défis : les applications sont très large et complexe, les modules sont plus emmêlées les uns dans les autres, on ne peut mettre à l'‘échelle que l'application entière au lieu d'un service spécifique.

Les microservices permettent de résoudre ces différents défis. Les microservices désignent à la fois une architecture et une approche de développement logiciel qui consiste à décomposer les applications en éléments simples, indépendants les uns des autres. Ils communiquent avec des mécanismes qui ne consomment pas beaucoup de bande passante telle que l'utilisation des API RESTful via http. [15]

3.1. Caractéristiques des microservices [16]

- **L'autonomie :** Chaque service est développé par des équipes différentes avec des outils adaptées et possède son propre stockage de données. Il peut fonctionner et s'exécuter de manière autonome.
- **Conçu pour les entreprises :** Le style des microservices est généralement organisé autour des capacités et des priorités de l'entreprise.
- **Intégration facile :** les microservices peuvent être intégrer en utilisant des protocoles ouverts bien connus. Il offre une hétérogénéité de technologies.
- **Évolutionniste :** L'architecture à microservices permet également de faire évoluer chaque service indépendamment. On peut déployer le nombre exact d'instances nécessaire pour respecter les contraintes de capacité et de disponibilité du service. On peut également utiliser le matériel qui correspond le mieux aux besoins en ressources de chaque service.

3.2. Les microservices et les conteneurs

Une application basée sur des microservices dispose d'une unité de déploiement et d'un environnement d'exécution parfaitement adapté. Lorsque les microservices sont stockés dans des conteneurs, il est plus facile d'exploiter les services indépendamment du matériel et les orchestrer, y compris les services de stockage, de mise en réseau et de sécurité.

C'est pour cette raison que la CNCF affirme qu'ensemble, les microservices et les conteneurs constituent la base du développement d'applications cloud-native. [15]

Une application cloud-native se compose de services plus petits, indépendants et faiblement couplés. Elle est conçue de façon à apporter une valeur métier incontestée, comme la capacité à prendre en compte rapidement l'avis des utilisateurs dans un effort d'amélioration continue. Lorsque l'on dit d'une application qu'elle est « native pour le cloud », cela signifie qu'elle a été conçue spécialement pour offrir une expérience cohérente de développement et de gestion automatisée dans les clouds privés, publics et hybrides.

V Conclusion

Le cloud computing a connu ces dernières années une évolution exponentielle. Ce développement a poussé les fournisseurs à trouver des solutions pour le bon fonctionnement de leurs services en temps et en ressources. Par conséquence plusieurs technologies sont apparues dans le cloud computing.

Au cours de ce chapitre nous avons discuté les principes du cloud computing et les notions gravitant autour de ce paradégime.

Dans le prochain chapitre, nous allons présenter en détails le gestionnaire des conteneurs le plus utilisé Docker et le gestionnaire de groupes de conteneurs le plus célèbre Kubernetes et les différentes stratégies d’ordonnancements existantes.

Chapitre 2
Docker, kubernetes et Ordonnancement

I Introduction

Les entreprises optent pour la technologie des conteneurs afin de gérer leurs infrastructures. De plus, les conteneurs permettent aux développeurs de déployer des applications dans tous les environnements avec peu ou pas de modification.

Mais pour créer et déployer ces conteneurs, les développeurs ont besoin d'un environnement d'exécution des conteneurs, dans le cadre de la résolution de ce problème, de nombreuses plateformes ont vu le jour et ont joué le rôle de gérer les cycles de vie des conteneurs et permettent aussi de gérer un grand nombre de conteneur à la fois.

II Docker

1. Définition

La conteneurisation est une technologie qui combine l'application, les dépendances associées et les bibliothèques système organisé sous la forme d'un conteneur. Docker est une plateforme qui garantit que l'application fonctionne dans tous les environnements .Il automatise également les applications qui seront déployées dans des conteneurs. Docker aide aussi à fournir un environnement rapide et léger pour exécuter un code efficacement. [17]

2. Architecture de Docker

L'architecture docker est une architecture client-serveur où l'on trouve : le client docker, l'hôte (host) docker, les réseaux et les composants de stockage.

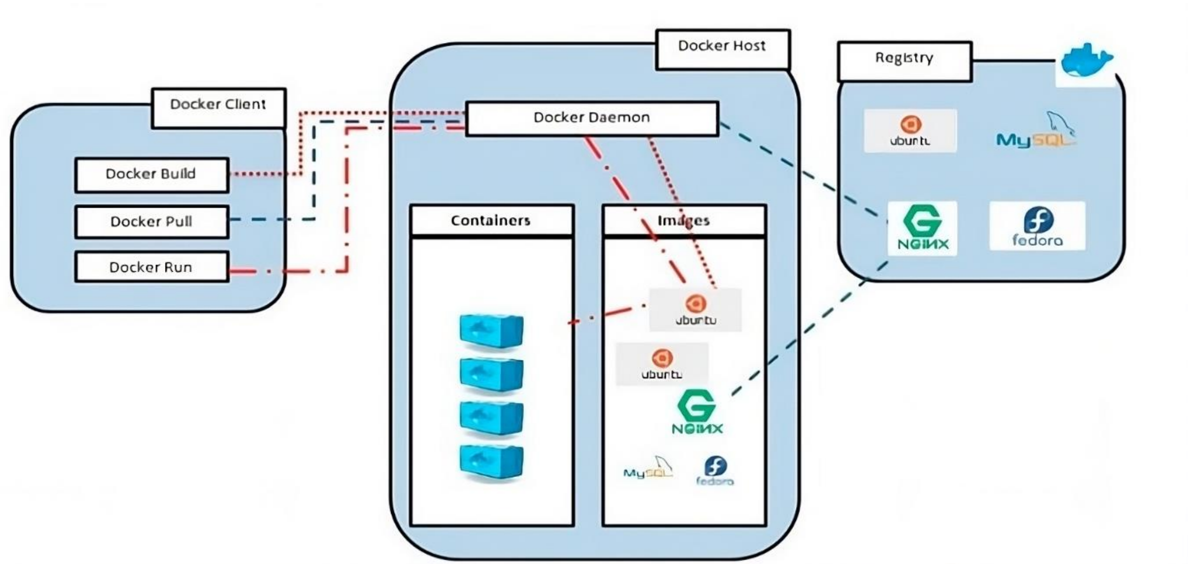

Figure 2.1 – Architecture de Docker [17]

2.1. Client Docker

Le client Docker permet aux utilisateurs d'interagir avec docker. Un client peut communiquer avec plusieurs démons (daemons) et il offre une interface de ligne de commande pour construire, exécuter et arrêter une application.

2.2. Docker host

L'hôte Docker fournit un environnement complet pour exécuter des applications. Il comprend :

2.2.1. Docker Engine

Est une technologie de conteneurisation open source pour la création et la conteneurisation d'applications. Le moteur Docker s'exécute comme une application client-serveur. Un serveur exécutant un processus démon Docker de longue durée, les programmes communiquent avec et former le démon Docker d'après l'usage d'un API qui spécifient les interfaces de liaison. [18]

2.2.2. Démon Docker

Il est responsable de toutes les opérations liées au conteneur et reçoit les commandes via la CLI ou l'API REST. Il peut également communiquer avec d'autres démons pour gérer ses services. Le démon Docker extrait et crée des images de conteneur en fonction des demandes des clients. Une fois qu'il extrait l'image demandée, il crée un modèle de travail pour le conteneur à l'aide d'un ensemble d'instructions appelé fichier de construction (builtfile). [19]

2.2.3. Image Docker

Les images Docker sont des instantanés ou des modèles à partir desquels de nouveaux conteneurs peuvent être démarrés. C'est une représentation du système de fichiers et des bibliothèques pour un système d'exploitation donné. De nouvelles images peuvent être créées en exécutant un ensemble de commandes contenues dans un Dockerfile qui représente un document texte qui contient toutes les commandes qu'un utilisateur peut appeler sur la ligne de commande pour assembler une image.

2.2.4. Conteneur (container) Docker

Les conteneurs sont des environnements encapsulés dans lesquels on exécute des applications. Un conteneur est défini par une image et des options de configuration supplémentaires fournies au démarrage du conteneur, y compris, les connexions réseau et les options de stockage.

2.3. Réseau (Network)

Docker implémente la mise en réseau d'une manière axée sur les applications. Il existe essentiellement deux types de réseaux : le réseau Docker par défaut et les réseaux définis par l'utilisateur. Par défaut, il existe trois réseaux différents lors de l'installation de Docker (aucun, pont et hôte).

2.4. Registre (Registry)

Le registre Docker est un système de stockage et de distribution pour les images Docker nommées. Une même image peut avoir plusieurs versions différentes, identifiées par leurs balises.

L'historique Docker est organisé en référentiels Docker, où le référentiel contient toutes les versions d'une image donnée. Le registre permet aux utilisateurs de Docker d'extraire des images localement, ainsi que de pousser de nouvelles images vers le registre (avec les autorisations d'accès appropriées accordées, le cas échéant).

Par défaut, le moteur Docker interagit avec DockerHub, l'instance de registre public de Docker. Cependant, il est possible d'exécuter localement un registre/distribution Docker open source, ainsi qu'une version prise en charge commercialement appelée Docker TrustedRegistry. Il existe d'autres registres publics disponibles sur Internet. [20]

3. Fonctionnement

La technologie Docker utilise le noyau Linux ainsi que ses fonctionnalités (comme les Cgroups et les espaces de noms) pour séparer des processus afin qu'ils s'exécutent de manière indépendante. Cette indépendance reflète l'objectif des conteneurs : exécuter plusieurs processus et applications séparément les uns des autres afin d'optimiser l'utilisation d’une infrastructure .Les outils de conteneurisation, dont Docker, fournissent un modèle de déploiement basé sur les images pour faciliter le partage d'une application ou d'un ensemble de services ainsi que toutes leurs dépendances dans plusieurs environnements. [21]

III Orchestration des conteneurs

Avec le passage à l’échelle des applications et services, le nombre des conteneurs a augmenté ce qui rend la gestion considérablement plus complexe, par conséquent, l'évolutivité a conduit à la création de COE (Container Orchestration Engine), une variété d'outils qui gèrent un grand nombre de conteneurs et de services et garantissent un déploiement et une maintenance sans effort. Il fournit également une planification efficace des conteneurs à travers les ressources physiques.

1. Ordonnanceur des conteneurs

Les outils d’orchestration de conteneurs permettent aux utilisateurs de guider le déploiement des conteneurs et d’automatiser les mises à jour, la surveillance d’état et les procédures de basculement. Les applications sont généralement constituées de composants mis en conteneur individuellement, qui doivent fonctionner ensemble pour permettre à l’application de fonctionner correctement. L’ordonnanceur des conteneurs désigne le processus d’organisation du travail du composant individuel et des niveaux d’application.

2. Exemple d'ordonnanceurs

Il existe plusieurs ordonnanceurs de conteneurs, nous citerons trois ordonnanceurs qui sont Docker Swarm, Mesos et Kubernetes. Nous allons présenter brièvement les deux premiers et parlerons en détails du dernier.

2.1. Docker Swarm

Un essaim est un ensemble de machines fonctionnant avec Docker et fusionnées en un groupe. Une fois le cluster formé, on continue à exécuter les instructions Docker habituelles, mais à ce stade, elles sont exécutées sur un tas par un manager d'essaim. Les machines d'un essaim peuvent être physiques ou virtuelles. Après avoir rejoint un essaim, ils sont appelés worker. Les managers d'essaim peuvent utiliser quelques méthodologies pour exécuter les conteneurs, par exemple, le "hub le plus vide" qui remplit les machines les moins utilisées avec des conteneurs. Ou encore "dans l'essaim entier", qui garantit que chaque machine obtient précisément une partie prédéterminée du conteneur. Les managers d'essaim sont les principales machines qui peuvent lancer les instructions ou approuver différentes machines pour rejoindre l'essaim en tant qu'ouvriers (workers).

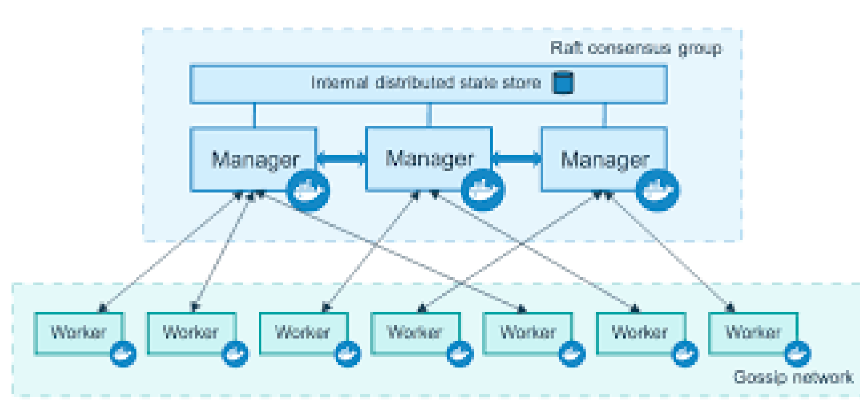

Figure 2.2 – Swarm Cluster [22]

2.2. Mesos

Apache Mesos est un projet Open Source de gestion de clusters pensé pour mettre en place et optimiser des systèmes distribués. Mesos permet la gestion et le partage des ressources de manière fine et dynamique entre différents nœuds et pour diverses applications. [23]

IV Kubernetes

1. Définition

L'un des COE les plus populaires est Kubernetes (k8s) qui représente un système open source facilitant l'automatisation de déploiement, la mise à l'échelle et la gestion des applications conteneurisées. Il a été initialement conçu par Google et maintenant il est géré par la CNCF. Les utilisateurs utilisent kubctl en tant qu'une ligne de commande permettant l'interaction avec le serveur API par exemple, en déployant un conteneur. [24]

2. Historique

Le nom Kubernetes vient du grec et signifie timonier ou pilote, K8s en tant qu'abréviation résulte du comptage des huit lettres entre le "K" et le "s". Google a ouvert le projet Kubernetes en 2014 qui a été créé par Joe Beda, Brendan Burns et Craig McLuckie. La conception et le développement de Kubernetes intégraient les meilleures idées du système de gestion de clusters interne orienté conteneur de Google nommé Borg, cependant Kubernetes a été écrit en langage Go.

Le nom de code original de Kubernetes au sein de Google était Project 7, la version 1.0 est sorti le 21 juillet 2015 en parallèle Google s'est associé à la Linux Foundation pour former la CNCF et a proposé Kubernetes comme technologie de départ. En février 2016 le gestionnaire de packages Helm pour Kubernetes a été publié. Le 6 mars 2018 le projet a atteint la neuvième place dans la liste des projets GitHub par le nombre de commits et la deuxième place pour les auteurs et les problèmes, après le noyau Linux. Plusieurs versions ont été délivrées à Kubernetes, la plus récente est la version v1.24 sortie en mai 2022. [25]

3. Les concepts de K8s

3.1. Pods

Les pods sont les plus petites unités informatiques déployables dans Kubernetes, il s'agit d'un ensemble de conteneurs sur un cluster qui partagent les mêmes processus et les espaces d'adressage réseau (interfaces réseau virtuelles, ports, adresses IP et mémoire partagée).

Le contenu d'un pod est exécuté dans un contexte partagé : Les processus à l'intérieur de pod peuvent communiquer entre eux utilisant des sockets ou avec des installations de

communication inter-processus. Les conteneurs du pod peuvent partager des données en accèdent aux volumes partagés. [26]

3.2. Services

Kubernetes fournit aux pods leurs propres adresses IP, un nommage basé sur DNS et peut équilibrer la charge entre eux à l'aide des services. L'adresse IP et le port du service sont fournis en tant que variables d'environnement lorsque le conteneur est démarré et le nom du service est résolu à l'aide du serveur DNS de Kubernetes.

Il existe des nombreux types de services par défaut c'est le ClusterIP où les pods communiquent entre eux à l'aide d'une adresse IP virtuelle statique, cependant avec les services NodePort, le pod peut avoir une adresse IP de l'hôte. [26]

3.3. ReplicaSets

Le ReplicaSet est chargé de maintenir un nombre spécifié de répliques pour un pod particulier. Il est utilisé pour garantir la disponibilité d'un nombre spécifié de pods identiques en surveillant l'état du cluster et planifie de nouveaux pods ou supprime ceux qui existent déjà. [26]

3.4. Déploiements

Un déploiement est l'objet API recommandé pour déployer un microservice sur Kubernetes permettant la gestion des ReplicaSet et les pods en fournissant des mises à jour. [27]

3.5. Espace de noms

Les espaces de noms représentent un mécanisme afin d'organiser le cluster en sous-clusters virtuels lorsque plusieurs projets partagent le même cluster Kubernetes de sorte que les ressources sont isolées [28]. Ils s'appliquent uniquement aux objets à espace de noms comme les services et les déploiements. [29]

3.6. Volume

Les volumes résolvent les problèmes de perte de fichiers lorsqu'un conteneur tombe en panne ou lors du partage de fichiers entre des conteneurs s'exécutant ensemble dans un fichier pod en fournissant un stockage persistant qui existe pour la durée de vie du pod lui-même. Les volumes sont montés à des points de montage spécifiques dans le conteneur. [30]

3.7. Load Balancer

Kubernetes considère les conteneurs comme un ensemble de services qu'ils exécutent ou fournissent. L'équilibrage de charge est l'élément responsable de la répartition de charge entre différents serveurs ou applications sur des infrastructures physiques et virtuelles, il distribue le trafic entrant sur un pool d'hôtes pour garantir des charges de travail optimales avec une haute disponibilité [31]. De plus, il analyse la requête demandée, détermine quelle machine est disponible pour y répondre, puis la transmet à la destination. [32]

4. Architecture

Une architecture Kubernetes se compose de plans de contrôle qui sont responsable de la gestion de l'ensemble du cluster, l'exposition de l'interface de programme d'application (API). Les nœuds de calcul qui ont pour rôle d'exécuter un environnement d'exécution de conteneur dans un pod avec un agent qui communique avec le plan de contrôle.

Dans les environnements de production, le plan de contrôle s'exécute généralement sur plusieurs stations et un cluster exécute généralement plusieurs nœuds en offrant une tolérance aux pannes et une grande disponibilité.

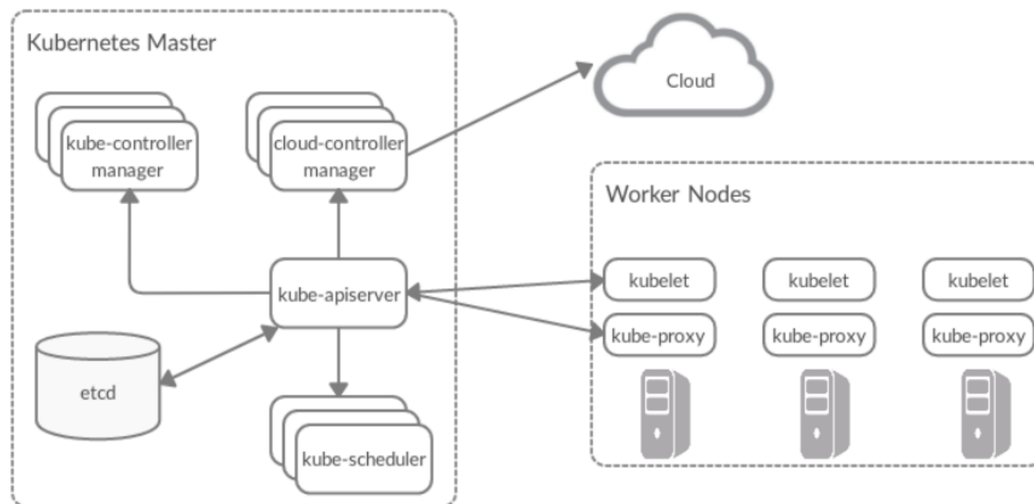

Figure 2.3 – Architecture de Kubernetes [26]

5. Les composants de K8s

5.1. Composants du plan de contrôle

Le plan de contrôle Kubernetes est le cerveau du cluster qui prend des décisions logiques. Ses composants répondent aux événements du cluster comme le démarrage d'un nouveau pod lorsque la valeur d'un champ de déploiement n'est pas satisfaite. De plus, il contrôle la planification, puisque les applications conteneurisées sont planifiées sur les nœuds workers en fonction de la mémoire allouée par déploiement. [33]

Ces composants sont les suivants : [34]

a. **kube-apiserver** : ou le serveur API kubernetes est l'interface du plan de contrôle avec laquelle les utilisateurs interagissent avec leur cluster, il détermine si une demande est valide, puis la traite. [35]

b. **Etcd** : représente la base de données pour toutes les données du cluster.

c. **kube-scheduler** : est un composant qui affecte les pods non planifiés au nœud concerné en fonction de son utilisation de la mémoire considérant les facteurs des ressources et les contraintes matérielles et logicielles.

d. **kube-control-manager** : ce composant exécute le processus de contrôle.

e. **cloud-controller-manager** : le plan de contrôle intègre une logique de contrôle spécialement pour le cloud, ce gestionnaire permet de lier le cluster à l'API de fournisseur cloud et exécute uniquement les contrôleurs qui interagissent avec lui.

5.2. Composants de nœuds de calcul

Ils s'exécutent sur chaque nœud, en maintenant les pods en exécution et en fournissant l'environnement d'exécution Kubernetes. [34]

a. **Kubelet** : Le kubelet est le principal agent qui s'exécute sur chaque nœud du cluster, il assure l'exécution des conteneurs dans le pod de sorte que les équipes informatiques connectent leurs K8s à d'autres API. Il fonctionne en termes de PodSpec qui est un objet YAML (YAML Ain't Markup Language) ou JSON (JavaScript Object Notation) pour décrire un pod. Le kubelet prend un ensemble de PodSpecs qui sont fournis via divers mécanismes (généralement apiserver) et garantit que les conteneurs décrits fonctionnent correctement. [36]

b. **Kube-proxy** : est un proxy réseau qui s'exécute sur chaque nœud (précisément dans les pods) et implémente la ressource Service dans Kubernetes suivant des règles réseau pour faire une communication réseau avec les pods.
c. **Container runtime** : c'est le logiciel responsable de l'exécution des conteneurs.
d. **Addons** : ils utilisent les ressources Kubernetes afin de fournir des fonctionnalités au niveau du cluster.
e. **DNS** : le serveur DNS est requis pour les services et les pods Kubernetes, il sert la création des enregistrements DNS. Kubelet configure le DNS des pods afin que les conteneurs en cours d'exécution fassent la recherche des services par nom.
f. **Dashboard** : est une interface utilisateur web pour gérer les applications s'exécutant dans les clusters Kubernetes.
g. **Surveillance des ressources de conteneur** : agréger les données collectées des métriques de séries chronologiques à propos des conteneurs dans une base de données centrale et fournir une interface utilisateur pour les consulter.
h. **Journalisation au niveau du cluster** : a pour rôle d'enregistrer les journaux de conteneur dans un magasin de journaux central avec une interface de recherche ou navigation.

V Les travaux d'ordonnancement existants

1. Techniques d'ordonnancement de kubernetes

Kubernetes possède un ordonnanceur par défaut qui s'appelle kube-sheduler dont son rôle est de trouver un nœud pour un pod nouvellement créé. Il travaille avec une file d'attente dans laquelle le pod principal sera placé en premier vers un nœud sélectionné.

kube-scheduler sélectionne un nœud pour le pod dans une opération en deux étapes :

1.1. Filtrage

Trouver l'ensemble de nœuds sur lesquels les pods peuvent être planifiés. Après cette étape, la liste des nœuds contient tous les nœuds associés ; généralement, il y en aura plus d'un. Si la liste est vide, ce pod n'est pas encore planifiable.

Actuellement cette stratégie englobe quatre règles de filtrages : [37]

A. General Predicates

Elles suppriment les nœuds qui ne peuvent pas contenir :

- **PodFitsResources**, ce pod vérifie que le nœud dispose des ressources libres nécessaires telles que CPU et la mémoire de l'hôte pour répondre aux exigences des pods. [38]
- **PodFitsHostPort** est utilisé pour vérifier la disponibilité des ports sur un nœud
- **PodFitsHost** est utilisé pour vérifier si un pod spécifie un nœud par son nom d'hôte

B. Volume FilteringRules

Ces règles sont responsables de la planification des politiques liées au volume persistant du conteneur.

- **NoDiskConflict** : évalue si les pods peuvent affecter à un nœud à cause d'un certain volume demandé et déjà monté, car s'il existe un volume monté sur ce nœud, un autre pod ne peut pas y planifier. [39]
- **MaxPDVolumeCountPredicate** : examine une condition de volume persistant sur un nœud, s'il dépasse un certain nombre alors le pod qui déclare l'utilisation de ce type de volume persistant ne peut plus être programmé sur ce nœud.
- **VolumeZonePredicate** : évalue si l'étiquette de zone (domaine de haute disponibilité) du volume persistant correspond à l'étiquette de zone du nœud à examiner.
- **VolumeBindingPredicate** : Il est chargé de vérifier si le champ NodeAffinity du PV correspondant au pod correspond à l'étiquette d'un certain nœud.

C. Host FilteringRules

Ils prennent comme rôle l'examination du pod à ordonnancer s'il répond aux conditions du nœud.

- **NodeMemoryPressurePredicate** vérifie si un pod peut être planifié sur un nœud selon une condition de pression de la mémoire. [40]
- **PodToleratesNodeTaints** vérifie si la souillure du nœud correspond à la tolérance du pod et s'il peut être programmé sur le nœud.

D. PodFilteringRules

Ce sont des règles qui ont une relation avec le pod lui-même.

- **PodAffinityPredicate** vérifie l'affinité et l'anti- affinité entre le pod à ordonnancer et le nœud en confirmant si le dernier peut attribuer plusieurs pods en même temps.

1.2. Notation (scoring)

Pendant la phase de notation, le planificateur classe les nœuds restants pour choisir le placement de pod le plus approprié. Le planificateur attribue à chaque nœud qui survit au filtre un score selon les règles de notation actives. Voici quelques règles et notions qui semblent important dans cette phase.

- **Pod affinity et Pod anti-affinity** permettent de spécifier des règles sur la manière dont les pods doivent être placés par rapport aux autres pods. Les règles sont définies à l'aide d'étiquettes personnalisées sur les nœuds et les sélecteurs d'étiquettes spécifiés dans les pods.

L'affinité de pod peut indiquer au planificateur de localiser un nouveau pod sur le même nœud que les autres pods si le sélecteur d'étiquette sur le nouveau pod correspond à l'étiquette sur le pod actuel. L'anti-affinité de pod peut empêcher le planificateur de localiser un nouveau pod sur le même nœud que des pods avec les mêmes étiquettes si le sélecteur d'étiquette sur le nouveau pod correspond à l'étiquette sur le pod actuel. [41]

- **Taints and Tolerations (Tache et Tolérance):** l'affinité de nœud est une propriété des pods qui les attire vers un ensemble de nœuds. Les taints sont à l'opposé, elles permettent à un nœud de repousser un ensemble de pods.

Les tolérances sont appliquées aux pods. Elles permettent au planificateur de planifier des pods avec des rejets correspondants mais ne garantissent pas l'ordonnancement : l'ordonnanceur évalue également d'autres paramètres dans le cadre de sa fonction.

Les taches et les tolérances fonctionnent ensemble pour garantir que les pods ne sont pas planifiés sur des nœuds inappropriés. Une ou plusieurs taches sont appliquées à un nœud ; cela indique que le nœud ne doit accepter aucun pod qui ne tolère pas les taches. [42]

Enfin, kube-scheduler attribue le pod au nœud le mieux classé. S'il y a plusieurs nœuds avec le même score, l'ordonnanceur choisira un au hasard.

2. Technique ACO-PSO

2.1. Algorithme de colonies de fourmis (ACO)

L'algorithme des colonies de fourmis suit une approche de type métaheuristique, il est proposé par le chercheur italien M.Dorigo afin de résoudre les problèmes d'optimisation combinatoire comme il peut améliorer l'ordonnancement des taches cloud computing.

Le principe de cet algorithme est d'affecter m fourmis à n villes tel que : n<m [43]. Il base sur un modèle probabiliste dans lequel un ensemble d'agents simples construit à plusieurs reprises des solutions candidates ce qui conduit les fourmis à trouver un schéma d'appariement efficace et une solution optimale en parallèle en fonction d'une phéromone initialisée sur la tâche avec le chemin de la machine virtuelle. La communication entre les fourmis se fait par le biais d'informations. [44]

2.2. Algorithme de colonies d'optimisation par essaim particulaire (PSO)

C'est une technique d'optimisation stochastique proposée par Eberhart et Kennedy en 1995, elle s'appuie sur un essaim qui lui permet de rechercher simultanément une grande région de particules dans l'espace de solution de la fonction objectif optimisée.

PSO utilise un vecteur de vitesse pour mettre à jour la position actuelle de chaque particule dans l'essaim. Dans l'article [43], le poids est amélioré pour qu'il puisse juger si la vitesse antécédente est raisonnable à travers la scène réelle. Si la vitesse est logique alors la valeur de proportion de la vitesse des particules précédentes dans l'itération actuelle est grande sinon elle est petite et dans ce cas le poids est appelé adaptatif.

2.3. Combinaison d'ACO et PSO

En tant qu'une solution initiale de l'algorithme PSO et dans l'objectif de résoudre le problème d'algorithme ACO, les auteurs de l'article [43] définir une hybridation entre les deux algorithmes pour l'ordonnanceur Kubernetes.

Voir que dans le modèle des colonies de fourmis, il produit une stagnation si chaque fourmi affecte une tache à un nœud avec la plus forte concentration de phéromones du coup l'algorithme converge prématurément vers une solution optimale locale et que la solution optimale globale ne peut pas être trouvée. Ainsi il doit que certaines fourmis suivre la stratégie d'allocation de phéromones la plus élevée, et d'autres suivent une stratégie aléatoire pour découvrir des nouvelles solutions optimales locales.

Dans PSO, la solution initiale est générée aléatoirement et la solution optimale de la particule est effectuée avec le transfert d'informations entre les particules. En comparant la valeur optimale historique de la particule avec la solution optimale globale de toutes les particules, le rapport pondéral de la vitesse de chaque particule itérative est trouvé. Si la valeur optimale de la particule itérative est inférieure à la valeur optimale de l'itération précédente de la particule et inférieure à la valeur optimale globale de la particule, la direction de vol de la particule est correcte. Le poids de la vitesse de vol des particules est important car il influence sur la direction et la vitesse de vol de la prochaine itération.

2.4. Avantages et inconvénients

Avantages

- Réduire le coût des ressources et la charge maximale des nœuds.
- Attribuer des taches équilibrées.
- Adaptabilité du l'essaim avec le changement d'environnement.
- Eviter les lacunes de la capacité de recherche locale d'ACO et le manque de capacité d'information de rétroaction.

Inconvénients

- Consommation d'énergie.
- Pas de mise à l'échelle.

3. ProCon

Dans l'article [45], les auteurs ont proposé une solution d'ordonnancement basée sur une technique heuristique nommée ProCon qui a fait un changement de critères pour le choix d'un nœud (l'état des nœuds travailleurs, disponibilité des ressources et la spécification des conteneurs). Par contre la solution par défaut de kubernetes, ProCon supervise la progression de l'exécution des conteneurs.

L'intégration du ProCon on Kubernetes permet de recevoir les tâches du nœud master et les transférer sur un nœud travailleur sélectionné dans l'objectif de réduire le temps d'exécution global des travailleurs dans le cluster en plaçant les conteneurs dans le nœud le plus approprié par une estimation du temps d'achèvement pour chaque conteneur puis un calcul du taux de contention, finalement il sélectionne le travailleur.

3.1. Avantages et inconvénients

Avantages

- Temps d'exécution des conteneurs optimisé.
- Utilisation efficace des ressources.
- Equilibrage de charge.

Inconvénients

- Besoin de supervision pour un bon fonctionnement.
- Consommation importante d'énergie.

4. Stratégie d'ordonnancement basé sur l'apprentissage machine dans une architecture microservice (CSML)

Les auteurs de l'article [46] ont proposé une solution qui prend en charge d'ajuster le nombre des conteneurs pour éviter la pression de charge actuelle des services.

La solution est basée sur l'apprentissage machine plus précisément un algorithme de régression de foret appelé CSML (Container Scheduling Stratégie Based on Machine Learning in Microservice Architecture) pour prédire le nombre des conteneurs dans la prochaine fenêtre de temps sur la base des informations de la fenêtre de temps actuelle.

Pour les expérimentations, ils ont utilisé kubernetes pour gérer les charges de travail, les services conteneurisés et JMeter pour simuler les charges. Ainsi prometheus pour surveiller la qualité de service et la pression de charge à travers surveiller ARTU (average response time of user) et ER (error rate).

Les résultats des expérimentations ont été comparés avec l'effet de l'algorithme feedback. Lorsqu'il y a un changement lent de la pression de charge, les performances des deux algorithmes sont à peu près les mêmes en termes de temps de réponse moyen des utilisateurs et de taux d'erreur mais lorsque la pression de charge change radicalement, par rapport à l'algorithme de feedback, l'algorithme CSML peut réagir plus rapidement et plus précisément.

4.1. Avantages et inconvénients

Avantages

- Equilibrage de charge.
- Qualité de service.
- Expansion plus rapide.

Inconvénients

- Manque de tests en situation réelle.
- Consommation importante d’énergie.

VIConclusion

Dans ce chapitre, nous avons abordé Docker et Kubernetes qui sont des technologies basiquement différentes mais fonctionnent ensemble pour créer et déployer des applications conteneurisées. Docker devient le format du conteneur par défaut et son environnement d'exécution, alors que Kubernetes prend en charge de nombreux environnements d'exécution de conteneurs à mesure que les applications s'étendent sur plusieurs conteneurs déployés sur plusieurs serveurs.

Ainsi, nous avons vu les travaux d’ordonnancement existants avec ses avantages et ses inconvénients. Le prochain chapitre va décrire notre stratégie d’ordonnancement proposée.

Chapitre 3

Conception et développement

I Introduction

Après avoir définit les différentes stratégies d'ordonnancement pour les conteneurs, nous avons constaté que la stratégie par défaut de kubernetes ne répond pas aux besoins dynamiques et effectue l'ordonnancement des pods de la même manière quelle que soit la charge de travail.

Alors nous avons proposé une stratégie d'ordonnancement qui a pour but de réduire la consommation d'énergie et de la bande passante.

Dans ce chapitre, nous allons définir la problématique qui nous a incités à proposer cette stratégie en comparant les multiples travaux précédents et en donnant les métriques de performance qui seront considérées dans le test des travaux d'ordonnancement choisies. Ensuite nous allons décrire notre solution avec ses différentes phases. Enfin, nous allons clôturer ce chapitre par son architecture globale.

II Comparaison des travaux

Nous allons comparer dans ce qui suit les travaux présentés dans le deuxième chapitre. Le tableau suivant résume les différents objectifs de chaque stratégie d'ordonnancement ainsi que la technologie utilisée pour tester la stratégie. Nous présentons aussi l'approche de modélisation utilisée (mathématique, méta-heuristique, heuristique ou apprentissage automatique). Enfin, nous avons définis l'environnement dans lequel la stratégie a été testée.

Travail	Objective	Technologie	Approche	Environnements
K8S	• Sélectionne un hôte pour chaque pod non planifié	Kubernetes	Modélisation Mathématique	Environnement cloud
ACO-PSO	• Réduction du coût d'exécution • Meilleur équilibrage de charge	Kubernetes	Approche méta-heuristique	• Simulation de plateforme CloudSim • Modèle de planification des ressources de cloud computing
ProCon	• Utilisation efficace des ressources avec la prédiction des futurs besoins. • Equilibrage de charge • Temps d'exécution des conteneurs est optimisé	Kubernetes	Technique heuristique	Application de courte durée
CSML	• Equilibrage de charge et performance	Kubernetes	Technique d'apprentissage machine	Architecture Micro services

Tableau 1 – Comparaison entre les travaux d'ordonnancement

III Réalisation de la solution

Pour réaliser notre solution, nous allons nous focaliser sur deux travaux :

1. Une stratégie d'ordonnancement méta-heuristique basée sur l'hybridation entre ACO et PSO

Cette solution a été réalisée dans le but de résoudre le problème de coût des ressources en utilisant un algorithme méta-heuristique et en ajoutant la fonction objective de coût pour améliorer le modèle de l'ordonnanceur k8s.

1.1. Détails de la solution

Les auteurs de ce papier [43] ont combiné les algorithmes ACO (Ant Colony Optimization) et PSO (Particle Swarm Optimization) pour réaliser leur solution, mais dans notre solution nous retenons seulement l'algorithme ACO dans un souci de simplicité.

L'optimisation des colonies de fourmis (ACO) est un algorithme méta-heuristique basé sur une population importante de fourmis qui peut être utilisé pour trouver des solutions approximatives à des problèmes d'optimisation difficiles.

Dans ACO, un ensemble d'agents logiciels appelés fourmis artificielles recherchent de bonnes solutions à un problème d'optimisation donné. Pour appliquer ACO, le problème d'optimisation est transformé en un problème de recherche sur un graphe pondéré. Les fourmis artificielles construisent progressivement des solutions en se déplaçant sur le graphe. Le processus de construction de la solution est stochastique et il est biaisé par un modèle de phéromone, c'est à-dire un ensemble de paramètres associés à des composants de graphe (nœuds ou arêtes) dont les valeurs sont modifiées au moment de l'exécution par les fourmis.

La première phase consiste à initialiser un nombre de fourmis à n et un nombre de nœuds à m.

$\pi_{i\,j}(t)$ représente la concentration de phéromone sur le chemin entre le nœud i et le nœud j à l'instant t. $allow_k$ représente la liste des nœuds qui peuvent être visités par la fourmi k. $\mu_{i\,j}(t)$ est égal au score total assigné au nœud i par le pod j.

Le calcul de la probabilité qu'une fourmi *k* sélectionne un nœud *j* à partir du nœud *i* est :

$$P_{i,j}^{k} = \begin{cases} \dfrac{\left[\pi_{i,j}(t)\right]^{\alpha}\left[\mu_{i,j}(t)\right]^{\beta}}{\sum_{K \epsilon allow}\left[\pi_{i,j}(t)\right]^{\alpha}\left[\mu_{i,j}(t)\right]^{\beta}} & j \in allow_k \\ 0 & other \end{cases} \quad (1)$$

La mise à jour de la concentration de phéromone est calculée selon les équations suivantes :

$$\pi_{i,j}(t+1) = (1-\rho)\pi(t) + \Delta\pi_{i,j}(t) \quad (2)$$

$$\Delta\pi_{i,j}(t) = \frac{S_2}{f_k(m_{i,j})} \quad (3)$$

Tels que :

$\Delta_{i,j}$: La variation de l'équilibre des ressources du nœud *j* après le déploiement de la tâche *i*.

S_2: est une constante.

$f_k(m_{i,j})$: Représente le score d'égalisation de la ressource de nœud après l'affectation de Pod *j* au nœud *i*. Le calcul de score total se fait par l'équation suivante :

$$\mu = Score1 + Score2 \quad (4)$$

Tels que :

$\mu_{i,j}$: Score assigné au nœud *i* par le pod *j*.

Où le *Score1* est calculé à travers l'équation suivante :

$$Score1 = \frac{S_{mem} + S_{cpu}}{2} \quad (5)$$

Tels que :

$$S_{mem} = \frac{T_{mem} - S_{Reqmem} - P_{Reqmem}}{T_{mem}} \quad (6)$$

Où :

T_{mem}: La quantité de mémoire totale allouée au pod dans un nœud.

S_{Reqmem} : La quantité de mémoire consommée par le pod.

P_{Reqmem}: La quantité de mémoire nécessaire pour le prochain pod.

Et :

$$S_{cpu} = \frac{T_{cpu} - S_{Reqcpu} - P_{Reqcpu}}{T_{cpu}} \quad (7)$$

Où :

T_{cpu}: La quantité de cpu totale allouée au pod dans un nœud.

S_{Reqcpu}: La quantité de cpu consommée par le pod.

P_{Reqcpu} : La quantité de cpu nécessaire pour le prochain pod.

Le *Score2* est calculé avec l'équation suivante :

$$Score2 = \frac{1 - abs(S_{cpu} - S_{mem})}{2} \quad (8)$$

Nous avons considéré cette solution parce qu'elle offre un bon placement des conteneurs dans les nœuds et une meilleure disponibilité tout en réduisant le coût d'exécution. Par contre elle a certaines limites en termes de consommation d'énergie et n'offre pas d'évolutivité.

2. Placement économe de machine virtuelle dans les centres de données cloud

Dans ce papier [47] les auteurs se sont intéressés au placement des machines virtuelles (VMs) dans les machines physiques (PMs) dans le but d'améliorer l'utilisation des ressources et réduire la consommation d'énergie.

Ils ont proposé un schéma de placement de VMs répondant à de multiples contraintes de ressources telles que la taille du serveur physique (CPU, RAM,..) ,la capacité des liens réseaux pour améliorer l'utilisation des ressources et réduire le nombre de serveurs physiques actifs entrainant ainsi la réduction de la consommation d'énergie.

Afin d'optimiser les ressources réseaux, les auteurs ont pensé à minimiser le trafic réseaux dans les centres de données en convergeant le trafic entre les VMs dans une même PM ou Switch. Pour cela, ils ont utilisé deux matrices A et B tels que :

A est une matrice de trafic de communication entre les VMs.

B est une matrice de coût de trafic de communication équivalent au nombre de switch que traverse le trafic entre les VMs.

La fonction objective est formulée de la manière suivante :

$$\min cost_{net} \sum_{i,j=1}^{N} a_{i,j} b_{m,p}$$

$$s.t. \sum_{i=1}^{N} X_{i,m}.\overrightarrow{S_i} \leq \gamma_m.\overrightarrow{H_m}, \quad \forall m \tag{9}$$

$$\sum_{j=1}^{N} X_{j,p}.\overrightarrow{S_p} \leq \gamma_p.\overrightarrow{H_p}, \quad \forall p$$

$$\{X_{i,m}, X_{j,p} \in F\}$$

Les paramètres sont cités dans la figure 3.1

Symbol	Description
M	Number of PMs , indexed by $m = 1,..,M$
N	Number of VMs , indexed by $i = 1..N$
$\vec{H}_m$	d dimensional resource vector of PM m ,its value $\{H_{m,1}, H_{m,2}, \dots, H_{m,d}\}$, d is the number of resource types
$\vec{S}_i$	d dimensional resource vector of VM i its value $\{S_{i,1}, S_{i,2}, \dots, S_{i,d}\}$
Y_m	Binary variable , 1 indicates PM m is in the activation status ; 0 indicates that PM m is sleep
$X_{i,m}$	Binary variable , 1 indicates VM i is placed on the PM m , whereas is 0
A	Communication traffic matrix, $(a_{i,j})_{N\times N}$ is the traffic between VM i and VM j
B	Communication cost matrix B, The communication cost is equivalent to the number switch that the traffic between PMs traverse.

Figure 3.1 – Figure indiquant les symboles et leurs descriptions [47]

Pour les ressources serveurs, ils se sont focalisés sur la minimisation du nombre de PMs active.

La fonction objective est décrite dans l'équation suivante :

$$
\begin{gathered}
\min cost_{ser} \sum_{m=1}^{M} \\
\text{s.t.} \sum_{i=1}^{N} X_{i,m}.\overrightarrow{S_i} \leq \ \gamma_m.\overrightarrow{H_m}, \quad \forall m \\
\{X_{i,m}, X_{j,p} \in F\}
\end{gathered} \tag{10}
$$

Pour l'optimisation de l'énergie des ressources physiques, ils ont utilisé un problème d'optimisation multi-objectif qui est décrit dans l'équation suivante :

$$\min cost_{net} + r.\min cost_{ser} \qquad (11)$$

Dans cette solution, pour placer les VMs dans les PMs, les auteurs utilisent l'algorithme du Clusturing Hiérarchique suivi de l'algorithme Best Fit.

L'algorithme du Clusturing Hiérarchique consiste à réunir les VMs qui ont un grand trafic entre elles dans un même cluster pour assurer une meilleure performance des applications, et pour réduire le nombre d'éléments réseaux réduisant ainsi la consommation d'énergie.

Les VMs et leurs trafics sont présentés par un graphe unidirectionnel connecté *G = (V, E)* tels que :

V : Le nombre de VMs.

E : Le trafic entre les VMs.

Le clustering hiérarchique est réalisé en utilisant le minimum de coupes (une coupe se compose de toutes les arêtes qui ont une extrémité dans *Q* et l'autre extrémité dans *V \Q*, où *Q* est un ensemble de nœuds avec $Q \neq \emptyset$ et $Q \neq V$. Cet algorithme prend en entrée le graphe *G* et fournit en sortie le résultat *T(V)* qui est un arbre binaire représentant la coupe minimal de *G*. L'algorithme Best Fit prend en entrée l'arbre binaire *T(V)* puis parcours cet arbre pour générer un vecteur appelé VMlist (se compose des feuilles successives de l'arbre *T*).

Ce vecteur est utilisé pour placer les VMs.

L'algorithme place les différentes VMs présentes dans le vecteur dans les PMs correspondantes, pour chaque nouvelle VM il fait la recherche à partir de la première PM pour trouver celle qui peut accommoder cette nouvelle VM. Si aucune n'est trouvée alors une nouvelle PM est allouée.

Nous avons considéré cette solution car elle permet de réduire le trafic ainsi que le nombre de PMs nécessaire pour satisfaire les besoins de placement des VMs réalisant ainsi une économie considérable d'énergie. Par contre si on place un grand nombre de VM dans la même PM alors les ressources physiques seront surchargées, et il y'a aussi le risque de congestion du réseau.

IVProblématique

En vue des limites présentées sur les techniques d'ordonnancement de kubernetes en particulier dans l'efficacité énergétique et la surveillance des charges de travail, nous allons définir dans ce qui suit une solution basée sur des travaux existants afin de répondre à la problématique suivante : Comment placer les pods dans les nœuds appropriés tout en assurant une consommation minimale de l'énergie et de bande passante ?

1. Les métriques de performance

Les métriques de performance sont des critères qui permettent de déterminer la qualité et l'efficacité d'une stratégie d'ordonnancement parmi lesquelles on peut retrouver : [48]

- **Le coût :** il sert à surveiller la consommation des ressources et la surcharge dans le cluster, le coût des services aide à déterminer le coût total de l'application. Par exemple, le coût de calcul dépend du temps passé à exécuter une application sur les cœurs disponibles.
- **L'équilibrage de charge :** est l'un des problèmes importants pour la distribution d'un flux de traitement plus grand à des nœuds de traitement plus petits afin d'augmenter les performances totales du système. La charge de travail est répartie sur plusieurs ressources via les liaisons réseaux afin d'obtenir une utilisation optimale des ressources, un temps de réponse moyen minimal et éviter la surcharge des nœuds.
- **L'efficacité énergétique :** représente la quantité d'énergie consommée dans le déploiement des conteneurs. L'objective est de trouver des horaires qui minimisent la consommation électrique globale du cluster par conséquent le coût est réduit.
- **L'utilisation des ressources :** l'efficacité de l'utilisation des ressources disponibles sur un worker en termes de noyau, mémoire et la bande passante du réseau.
- **La bande passante :** est une métrique qui peut mesurer le nombre de bits transféré dans un lien de communication dans un laps de temps (généralement une seconde).
- **La latence :** elle indique le temps nécessaire pour exécuter une application du début à la fin. Elle est considérée comme cruciale pour les applications en temps réel qui nécessitent une faible latence ou qui ont des délais serrés.
- **L'empreinte carbone :** elle calcule la quantité de dioxyde de carbone rejetée dans l'atmosphère à la suite des activités d'une personne, une organisation ou une communauté en particulier. Par conséquent, il est très important de réduire la consommation d'énergie et de développer une énergie propre pour réduire l'empreinte carbone.

- **La sécurité :** une partie du travail d'orchestration consiste à protéger les données et les services à travers l'utilisation des outils et définir des stratégies pour assurer la protection des conteneurs contre les menaces.

2. Formule de calcul des métriques

En vue des objectifs des travaux existants et de la solution proposée dans la suite, nous avons choisi de nous focaliser sur les métriques suivantes : l'équilibrage de charge et la bande passante. Les formules de calcul de chaque métrique sont présentées dans ce qui suit :

- **Pour l'équilibrage de charge**

La puissance de traitement effectuée pour traiter les données (CPU) est divisée par le temps.

$$LB = \frac{CPU}{T}$$

- **Pour la bande passante**

La somme de la bande passante en Kbit/s est divisée par le nombre de serveurs.

$$BW = \frac{\sum_{i=0}^{n} B_i(\frac{Kbit}{s})}{S}$$

V Description de la solution

Selon les travaux que nous avons étudié, nous avons décidé de combiner les deux travaux cités ci-dessus en faisant une hybridation de l'algorithme de colonie de fourmis du premier pour le placement des pods dans les nœuds appropriés avec l'algorithme du clustering hiérarchique du second qui permet d'obtenir de meilleurs résultats en termes de consommation et d'économie énergie. De plus, nous allons utiliser l'API Prometheus pour récupérer et gérer les métriques (bande passante, cpu, stockage et la ram) des nœuds et des pods. Notre solution se fera en trois phases :

- La première phase consiste à récupérer les métriques des pods et nœuds à travers l'API Prometheus, et les fournir comme entrée à l'algorithme du clustering hiérarchique.
- La deuxième phase consistera à placer les pods de façon aléatoire dans les nœuds puis faire appel à l'algorithme du clustering hiérarchique pour regrouper les pods ayant un trafic réseaux important entre eux, et les réunir dans un même cluster pour optimiser la

consommation d'énergie. Cette phase produira la liste des clusters (groupe de pods) qui sera fournie en entrée à l'algorithme ACO pour effectuer le placement de ces derniers.

- La troisième phase consistera à choisir le meilleur nœud pour placer le groupe de pods résultant de la phase précédente en utilisant l'algorithme de colonie de fourmis (ACO).

1. Phase1

La première phase consiste à récupérer les métriques des pods et nœuds (bande passante, cpu, stockage et la ram) de notre cluster, pour cela nous utilisons l'API Prometheus qui va nous servir à collecter les différentes données relatives au cluster et aux pods présents dedans.

2. Phase 2

Pour la deuxième phase, nous allons appliquer l'algorithme du clustering hiérarchique à l'ensemble des pods, en nous focalisant sur l'optimisation des ressources réseaux afin de réduire la consommation d'énergie. Après avoir récupérer les valeurs des métriques, l'objectif principal est de regrouper l'ensemble les pods ayant un trafic réseau important et ceci afin de minimiser le nombre de nœuds nécessaire pour le déploiement des conteneurs et réduire le trafic réseau. Pour minimiser le trafic réseau entre les pods, nous utilisons deux matrices A et B :

A = $(a_{i,j})_{N,N}$

B = $(b_{m,p})_{M,M}$

Plus le coût de communication est grand plus la consommation d'énergie du réseau est élevée.

La fonction objective est formulée avec l'équation (10).

Soit un graphe *G = (V, E)* tels que :

V : Une collection de pods.

E : Trafic réseau entre ces pods.

L'algorithme de clustering hiérarchique est achevé en utilisant le minimum de coupe du graphe *G*, qui est décrit dans ce qui suit :

Soit un ensemble *Q* inclus dans *V*, δ (*Q*) désigne l'ensemble de toutes les arêtes avec une extrémité en *Q* et l'autre extrémité en *V* *Q*.

Une coupe désigne l'ensemble de toutes les arêtes avec une extrémité en Q et l'autre extrémité en $V \setminus Q$. Où Q est un ensemble tels que $Q \neq \emptyset$ et $Q \neq V$, la coupe est dénotée *(Q, V \Q)*.

On assigne à chaque arrête $ij \in E$ une capacité *c(ij)* non négative, cette capacité est définie comme étant la somme de toutes les arrêtes contenues dedans c'est à dire :

$c(Q, V \setminus Q) = \sum_{ij \in \delta(Q)} c(ij).$

Le problème de coupe minimum est de trouver une coupe en *G* avec la plus petite capacité.

La coupe minimale de *G* est exprimée avec un arbre binaire *T (V),* où le sous arbre gauche *TL* est dans *Q*, dont le poids est la somme des arrêtes qui sont dans *Q*, tels que :

$W(TL) = \sum_{ij \in \delta(Q)} c(ij)$, et le sous arbre droit *TR* est dans *V \Q* dont le poids est calculé avec W (TR) $= \sum_{ij \in V \setminus Q} c(ij)$.

La feuille dans *T(V)* représente un pod et la branche représente une collection de pods après le clustering. Le déroulement de l'algorithme est comme suit :

Algorithm 1 Algorithme du clustering hiérarchique

Entrée : Graphe $G = (V, E)$
Sortie : Arbre binaire $T(V)$

1: Coupe initiale :S
2: Arbre initial : $T(V)$
3: **while** G contient des noeuds **do**
4: Chosir 2 nouds distincts s et t
5: Calcul de la coupe $S2$ séparant s et t
6: **if** $c(S2, V \setminus S2) < C$ **then**
7: $C \leftarrow c(S2, V \setminus S2)$
8: $S \leftarrow S2$
9: **end if**
10: $TL \leftarrow G_s(V)$, Calcul $W(TL)$
11: $TR \leftarrow G_t(V)$, Calcul $W(TR)$
12: **if** $W(TL) < W(TR)$ **then**
13: $W(TL) \leftrightarrow W(TR)$
14: $G_s \leftrightarrow G_t$
15: **end if**
16: Remplacer G par G_s et G_t
17: **end while**
18: Retourner $T(V)$

Figure 3.2 – Algorithme du clustering hiérarchique

3. Phase 3

La troisième phase de notre solution, consiste à placer les groupes de pods résultants de la deuxième étape dans les nœuds les plus appropriés. Ce placement est réalisé avec l'algorithme de colonie de fourmis. Pour se faire, l'algorithme reçoit en entrée le résultat de l'algorithme du clustering hiérarchique qui est une collection de groupes de pods ayant un trafic réseau important entre eux.

Les équations relatives au calcul de la probabilité et de la mise à jour de la concentration de phéromone restent inchangées comme mentionné précédemment.

Afin de réaliser le placement des pods dans les nœuds adéquats nous devons calculer les scores (*Score1 et Score2*) sur la base des ressources utilisées par les groupes de pods qui sont le CPU, la RAM, stockage et la bande passante. Les valeurs seront calculées avec les formules suivantes :

- **Pour CPU**

En premier lieu nous récupérons la quantité de cpu utilisée par chaque pod $P_{cpuUsage}(i)$ appartenant au même cluster, puis nous calculons la quantité totale de cpu $T_{cpuUsage}(i)$ utilisée par tous les pods avec la formule suivante :

$$T_{cpuUsage} = \sum_{i=1}^{N} P_{cpuUsage}(i) \tag{12}$$

Tels que :

$P_{cpuUsage}$: La quantité de cpu utilisée par le *Pod (i).*

N : Le nombre de pods dans un cluster.

En deuxième lieu nous calculons le score du nœud par rapport à la quantité de cpu utilisée par l'ensemble des pods (S_{cpu}) qui sont dans le cluster avec la formule suivante :

$$S_{cpu} = \frac{T_{nodeCpu} - T_{cpuUsage}}{T_{nodeCpu}} \tag{13}$$

Tels que :

$T_{nodeCpu}$: La quantité de cpu d'un noeud.

- **Pour RAM**

En premier lieu nous récupérons la quantité de mémoire utilisée pour chaque pod $P_{memUsage}(i)$ appartenant au même cluster, ensuite nous calculons la quantité totale de mémoire utilisée par l'ensemble des pods $T_{memUsage}(i)$ avec la formule suivante :

$$T_{memUsage}(i) = \sum_{i=1}^{N} P_{memUsage}(i) \qquad (14)$$

Tels que :

$P_{memUsage}(i)$: La quantité de mémoire utilisée par le *Pod (i)*

N : Le nombre total des pods du cluster.

En deuxième lieu nous calculons le score du nœud par rapport à la quantité de mémoire utilisée par l'ensemble des pods (S_{mem}) qui sont dans le cluster avec la formule suivante :

$$S_{mem} = \frac{T_{nodeMem} - T_{memUsage}}{T_{nodeMem}} \qquad (15)$$

Tels que :

$T_{nodeMem}$: La quantité de mémoire d'un noeud.

- **Pour le stockage**

En premier lieu nous récupérons la quantité du stockage utilisée pour chaque pod $P_{storeUsage}(i)$ appartenant au même cluster, ensuite nous calculons la quantité totale du stockage utilisée par l'ensemble des pods $T_{storeUsage}(i)$ avec la formule suivante :

$$T_{storeUsage}(i) = \sum_{i=1}^{N} P_{storeUsage}(i) \qquad (16)$$

Tels que :

$P_{storeUsage}(i)$: La quantité du stockage utilisée par le *Pod (i)*

N : Le nombre total des pods du cluster.

En deuxième lieu nous calculons le score du nœud par rapport à la quantité du stockage utilisée par l'ensemble des pods S_{store} qui sont dans le cluster avec la formule suivante :

$$S_{store} = \frac{T_{nodeStore} - T_{storeUsage}}{T_{nodeStore}} \quad (17)$$

Tels que :

$T_{nodeStore}$: La quantité du stockage d'un nœud.

- **Pour la bande passante**

En premier lieu nous récupérons la quantité de la bande passante utilisée pour chaque pod $P_{BwUsage}(i)$ appartenant au même cluster, ensuite nous calculons la quantité totale de la bande passante utilisée par l'ensemble des pods $T_{BwUsage}(i)$ avec la formule suivante :

$$T_{BwUsage}(i) = \sum_{i=1}^{N} P_{BwUsage}(i) \quad (18)$$

Tels que :

$P_{BwUsage}(i)$: La quantité de la bande passante utilisée par le *Pod (i)*

N : Le nombre total des pods du cluster.

En deuxième lieu nous calculons le score du nœud par rapport à la quantité de la bande passante utilisée par l'ensemble des pods (S_{Bw}) qui sont dans le cluster avec la formule suivante :

$$S_{Bw} = \frac{T_{nodeBw} - T_{BwUsage}}{T_{nodeBw}} \quad (19)$$

Tels que :

T_{nodeBw} : La quantité de la bande passante d'un nœud.

L'équation du $Score1$ représentant le ratio d'inactivité des ressources du nœud est calculé comme suit :

$$Score1 = \frac{S_{mem} + S_{cpu} + S_{store} + S_{Bw}}{4} \quad (20)$$

Plus le score du nœud est élevé et plus il est susceptible d'être choisi pour exécuter le groupe de pods.

Le $Score2$ représente le degré d'équilibrage des ressources du nœud, est calculé comme suit :

$$Score2 = \frac{1 - abs((S_{cpu} + S_{Bw}) - (S_{mem} + S_{store}))}{4} \quad (21)$$

Plus la valeur est grande, meilleur est l'équilibre des ressources du nœud.

Le déroulement de l'algorithme se fait comme suit :

1. Initialiser un nombre de fourmis *(nbFourmis)* à *m* et initialiser les valeurs des scores des nœuds présents dans la liste résultante de l'algorithme du clustering hiérarchique. Le nombre maximal d'itérations est *maxIter* et le nombre d'itérations actuel est *iterI*. Initialiser aussi les valeurs du facteur heuristique des phéromones α, le facteur de volatilisation ρ, Le facteur heuristique β et les valeurs de la fonction objective.

2. Placer les fourmis aléatoirement dans les nœuds et traiter le $j^{ième}$ groupe de pods.

3. Chaque fourmi sélectionne le nœud approprié pour la prochaine tâche en se basant sur le score du nœud qui est calculé à l'aide de la formule (4), ainsi que la valeur de la fonction objective.

4. Après l'affectation de chaque tâche à un nœud avec la plus grande concentration de phéromone, la stagnation à lieu alors l'algorithme converge vers une solution locale.

5. Une fois que toutes les fourmis ont terminé toutes les allocations des pods, la mise à jour des phéromones est effectuée sur le schéma d'allocation global optimal par les formules (2) et (3).

6. Faire la comparaison entre le nombre actuel d'itérations et *maxIter*. Si le nombre actuel d'itérations est inférieur à *maxIter*, on va passer à l'étape 3, sinon on sort de la boucle, terminant ainsi l'algorithme.

Algorithm 2 Algorithme de colonis de fourmi

```
1: Initialiser le nombre de fourmi à m (le nombre de groupes de pods)
2: Initialiser les valeurs du facteur heuristique des phéromones α
3: Initialiser la valeur de facteur de volatilisation ρ
4: Initialiser le facteur heuristique β
5: Initialiser les valeurs de la fonction objective
6: while nb_iter != max_iter do
7:     Affecter aléatoirement chaque groupe de pods à un nœud
8:     while fourmis n'a pas encore construit la solution do
9:         for Chaque fourmis do
10:            Calculer score (équation 8) et valeur fonction objective.
11:            Sélectionner le nœud approprié
12:            Mise à jour local des phéromones
13:        end for
14:        Applique la règle de mise à jour globale
15:    end while
16: end while
```

Figure 3.3 – Algorithme de colonies de fourmi

VIArchitecture de la solution

Voici la figure 3.4 qui représente l'architecture globale de notre solution proposée.

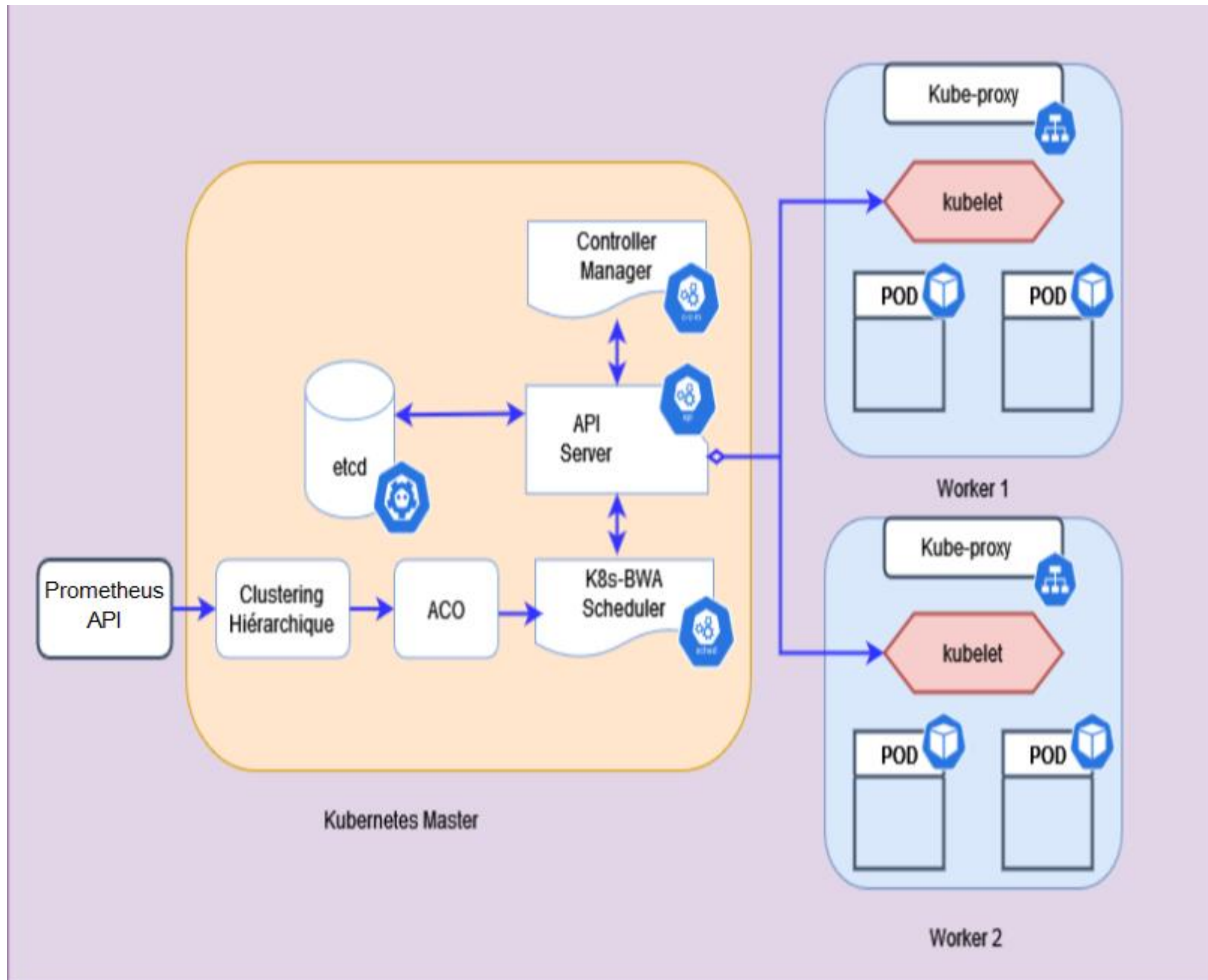

Figure 3.4 – Architecture de la solution d'ordonnancement

VII Conclusion

Dans ce chapitre, nous avons abordé la partie la plus importante de notre travail qui est la description et la conception de notre solution proposée. Cette solution a pour objectif la minimisation de la consommation de la bande passante et de l'énergie.

Nous avons vu le processus de sa réalisation ainsi que son architecture générale et les métriques de performance qui seront considérées dans l'évaluation des stratégies. Nous parlerons en détails dans le prochain chapitre de l'environnement matériel utilisé pour l'implémentation des travaux d'ordonnancement choisies, ainsi que les outils logiciels nécessaires.

Chapitre 4
Implémentation et tests

I Introduction

Nous avons abordé à travers les chapitres précédents divers concepts liés à l'ordonnancement des applications basées sur les conteneurs dans kubernetes. Ensuite, nous avons étudié certains travaux qui ont une relation avec notre projet en termes de contexte et d'objectifs. Nous sommes basé sur ces travaux afin de créer notre propre stratégie d'ordonnancement pour kubernetes dans le but de réduire la consommation des ressources ainsi que la bande passante, qui n'est pas prise en compte dans la stratégie d'ordonnancement par défaut, nous avons appelé notre proposition «K8s-BWA» (Kubernetes Bandwidth Aware Scheduler).

Dans ce chapitre, nous allons présenter l'implémentation de notre approche avec la stratégie de kubernetes et aléatoire en commençant par la description de l'environnement physique de développement, ainsi que les différents outils utilisés pour la réalisation. À la fin de ce chapitre, nous exposerons les différents résultats et évaluons notre solution dans le but de montrer son mérite en faisant une comparaison avec la stratégie par défaut de kubernetes et la stratégie aléatoire.

II Environnement de travail

1. Environnement matériel

Nous avons utilisé la plateforme CloudLab qui nous fournit les machines et les ressources nécessaire pour notre cluster kubernetes. Le tableau suivant montre la fiche technique de notre environnement physique.

Nom	Rôle	Adresses IP	Ressources
node0.kubernetes.labs1-pg0.utah.cloudlab.us	Nœud master	128.110.217.154	**OS** : Ubuntu 18.04.1 LTS **CPU** : Intel Xeon D-1548 8 Cœurs **RAM :** 4 GB
node1.kubernetes.labs1-pg0.utah.cloudlab.us	Nœud travailleur	128.110.217.149	**OS** : Ubuntu 18.04.1 LTS **CPU** : Intel Xeon D-1548 8 Cœurs **RAM :** 4 GB
node2.kubernetes.labs1-pg0.utah.cloudlab.us	Nœud travailleur	128.110.217.144	**OS** : Ubuntu 18.04.1 LTS **CPU** : Intel Xeon D-1548 8 Cœurs **RAM :** 4 GB
node3.kubernetes.labs1-pg0.utah.cloudlab.us	Nœud travailleur	128.110.217.147	**OS** : Ubuntu 18.04.1 LTS **CPU** : Intel Xeon D-1548 8 Cœurs **RAM :** 4 GB
node4.kubernetes.labs1-pg0.utah.cloudlab.us	Nœud travailleur	128.110.217.118	**OS** : Ubuntu 18.04.1 LTS **CPU** : Intel Xeon D-1548 8 Cœurs **RAM :** 4 GB

Tableau 2 – Tableau présentant la fiche technique de l'environnement physique

2. Environnement logiciel

L'environnement logiciel comprend les composants suivants :

2.1. CloudLab

Représente un méta-cloud pour construire des propres clouds sur ses ressources, il donne la main à exécuter des piles de logiciels cloud existantes comme OpenStack, Hadoop ou de les construire à partir de zéro.

La plupart des ressources CloudLab fournissent une isolation solide des autres utilisateurs. [49]

2.2. YAML

Il signifie un format de sérialisation de données afin de stocker les données de configuration ou échanger des données. Il est plus lisible pour l'utilisateur que XML ou JSON et devient une technologie utile pour créer des configurations complexes dans la conteneurisation et le déploiement dans le cloud etc. YAML dispose d'un modèle cohérent pour prendre en charge les outils génériques, ce qui permet la portabilité des données entre les langages de programmation. [50]

2.3. Ubuntu 18.04.1 LTS

C'est le premier incrément sur six versions prévues avant le plongeon 18.10. Cette version comprend également toutes les dernières applications, mises à jour de sécurité et mises à jour publiées après le 26 avril 2018. Parmi ses caractéristiques : [51]

- Accélération de la mise à niveau sans surveillance.
- Correction d'un problème de transparence sur le serveur d'affichage Wayland.
- Correction de la prise en charge de l'installation sur NVMe avec RAID1.
- Gestionnaire de mise à jour stable.

2.4. Docker

Dans notre stratégie la version de Docker utilisée est 24.0.2, les informations sur ce dernier sont mentionnées plus haut.

2.5. Kubernetes

Dans notre stratégie nous avons utilisé la version 1.18.5 de kubernetes, les informations sur kubernetes sont mentionnées plus haut

2.6. Prometheus

Prometheus est un outil open source de surveillance et d'alerte des systèmes créé en 2012, initialement conçue sur SoundCloud. Il a rejoint la CNCF en 2016 en tant que deuxième projet hébergé, après Kubernetes.

Prometheus collecte et stocke ses métriques sous forme de données de séries chronologiques, cela signifie que les changements sont enregistrés au fil du temps. Les

métriques jouent un rôle important pour comprendre que l'application fonctionne d'une certaine manière à travers des informations.

L'écosystème Prometheus se compose de plusieurs composants où les pluparts sont écrits en Go : [52]

- **Le serveur Prometheus** : il récupère et stocke les données de séries chronologiques.
- **Alertmanager** : utilisé pour la gestion des alertes.
- **API clients** : représentent des bibliothèques clientes qui surveillent les services, en ajoutant une instrumentation au code.
- **Pushgateway** : pour prendre en charge les emplois de courte durée.
- **Exportateurs spécialisés** : pour des services tels que HAProxy, StatsD, Graphite, etc.

3. Langage de programmation

3.1. Python

Pour implémenter notre stratégie et la stratégie aléatoire, nous avons utilisé python comme langage de programmation afin d'interagir avec l'API de l'orchestrateur des conteneurs Kubernetes, ce qui requiert l'utilisation d'une bibliothèque client qui s'appelle client-python.

Python est un langage de programmation interprété, orienté objet et de haut niveau avec une sémantique dynamique, ses structures de données intégrées de haut niveau, combinées au typage dynamique et à la liaison dynamique, le rendent très attrayant pour le développement rapide d'applications. La syntaxe simple et facile à apprendre dont il met l'accent sur la lisibilité et réduit donc le coût de maintenance du programme. Python prend en charge les modules et les packages, ce qui encourage la modularité du programme et la réutilisation du code. L'interpréteur Python et la vaste bibliothèque standard sont disponibles gratuitement sous forme source ou binaire pour toutes les principales plates-formes et peuvent être librement distribués. [53]

Dans notre implémentation nous avons utilisé la version 3.11.4.

3.2. Bibliothèques clientes

Les bibliothèques clientes gèrent souvent des tâches courantes telles que l'authentification. La plupart des bibliothèques clientes peuvent découvrir et utiliser le compte de service Kubernetes pour s'authentifier si le client API s'exécute dans le cluster Kubernetes, ou peuvent comprendre le format de fichier kubeconfig pour lire les informations d'identification et l'adresse du serveur API.

3.3. Bibliothèques clientes officielles supportées par Kubernetes

Il existe divers façons de personnaliser et d'étendre Kubernetes ; Comme l'utilisation des fichiers de configuration, l'intérêt d'étendre kubernetes, c'est la possibilité d'accéder à des bibliothèques clientes pour l'utilisation de son API à partir de divers langages de programmation tels qu'il est mentionné dans la figure 4.1.

Language	Client Library	Sample Programs
Go	github.com/kubernetes/client-go/	browse
Python	github.com/kubernetes-incubator/client-python/	browse

Figure 4.1 – Bibliothèques clientes officielles supportées par Kubernetes [54]

III Implémentation

Dans la section suivante, nous avons choisi trois travaux à implémenter.

1. Implémentation de la stratégie par défaut

Comme nous avons déjà discuté, l'ordonnanceur de kubernetes (kube-scheduler) utilise quelques mécanismes pour définir où les pods doit être planifiés. L'ordonnanceur peut déployer automatiquement des pods sur les nœuds sans autres règles spécifiées comme il est montré dans la figure 4.2.

Cependant, dans plusieurs cas, nous souhaitons définir que les pods doivent s'exécuter uniquement sur des nœuds spécifiques d'un cluster, ou éviter de s'exécuter sur certain nœuds.

Nous allons traiter deux exemples parmi les mécanismes de kube-scheduler.

```
root@node0:~# kubectl get pods -n resys -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP                NODE                                           NOMINATED NODE   READINESS GATES
apache-pod     1/1     Running   0          12s   192.168.1.41      node1.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
apache-pod2    1/1     Running   0          12s   192.168.212.105   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
apache-pod3    1/1     Running   0          12s   192.168.25.116    node4.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod    1/1     Running   0          12s   192.168.1.42      node1.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod2   1/1     Running   0          12s   192.168.1.44      node1.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod3   1/1     Running   0          12s   192.168.212.104   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod      1/1     Running   0          12s   192.168.25.115    node4.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod2     1/1     Running   0          12s   192.168.1.43      node1.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod3     1/1     Running   0          12s   192.168.25.114    node4.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
```

Figure 4.2 – Résultat d’ordonnancement avec la stratégie par défaut

1.1 NodeAffinity

Nous avons testé la règle NodeAffinity qui permet de limiter les nœuds sur lesquels notre pod peut être ordonnancé en fonction des étiquettes de nœud.

Dans la première étape, nous avons étiqueté le « node2 » comme la figure ci-dessous indique.

```
root@node0:~# kubectl label node node2.younescc-157245.labs1-pg0.utah.cloudlab.us rank=3
node/node2.younescc-157245.labs1-pg0.utah.cloudlab.us labeled
```

Figure 4.3 – L'étiquette du nœud 2

Ensuite, nous avons définis nos pods avec le type « nodeAffinity : required During Scheduling Ignored During Execution » et une clé « rank » inférieur à « 5 » cela indique que l’ordonnanceur ne peut pas planifier le pod à moins que la règle ne soit respectée.

Nous pouvons changer le type et les valeurs liés à l’affinité d’un nœud selon nos besoins.

```
  labels:
    app: httpd_app_nautilus
spec:
  replicas: 10
  selector:
    matchLabels:
      app: httpd_app_nautilus
  template:
    metadata:
      labels:
        app: httpd_app_nautilus
    spec:
      affinity:
          nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                     nodeSelectorTerms:
                        - matchExpressions:
                            - key: rank
                              operator: Lt
                              values:
                              - "5"
      containers:
        - name: httpd-container-nautilus
          image: httpd:latest
          ports:
            - containerPort: 80
```

Figure 4.4 – Configuration des pods avec NodeAffinity

Comme la figure 4.5 illustre, les pods « httpd » sont ordonnancés sur « node2 », car la valeur de leur clé est compatible avec la valeur mentionnée dans l'étiquette du « node2 ».

```
NAME                                        READY   STATUS    RESTARTS   AGE   IP               NODE
DINESS GATES
httpd-deployment-nautilus-865548c756-5zmss   1/1     Running   0          38s   192.168.186.18   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-hnbbb   1/1     Running   0          38s   192.168.186.26   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-hw2sk   1/1     Running   0          38s   192.168.186.20   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-19c49   1/1     Running   0          38s   192.168.186.17   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-n8dfj   1/1     Running   0          38s   192.168.186.24   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-ng2hd   1/1     Running   0          38s   192.168.186.23   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-rrwc6   1/1     Running   0          38s   192.168.186.22   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-tt6rq   1/1     Running   0          38s   192.168.186.25   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-wqwtp   1/1     Running   0          38s   192.168.186.19   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
httpd-deployment-nautilus-865548c756-xw9n7   1/1     Running   0          38s   192.168.186.21   node2.younescc-157245.labs1-pg0.utah.cloudlab.us
ne>
```

Figure 4.5 – Résultat d'ordonnancement de NodeAffinity

1.2 PodAntiAffinity

La règle d'anti-affinité indique que l'ordonnanceur doit éviter la planification d'un pod sur un nœud qui se trouve dans la même zone qu'un ou plusieurs pods portant une étiquette définies.

Dans Ce test, les pods « nginx » ne seront pas exécuté avec les pods « httpd ».

```
  labels:
    app: nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - "httpd_app_nautilus"
              topologyKey: kubernetes.io/hostname
```

Figure 4.6 – Configuration des pods avec Pod AntiAffinity

Le résultat ci-dessous montre le placement des pods « nginx » sur les différents nœuds du cluster à l'exception du « node2 » où les pods « httpd » sont exécutés.

```
nginx-deployment-b49b9cbb4-57spx          1/1     Running   0          5s    192.168.176.203   node4.younescc-157245.labs1-pg0.utah.cloudlab.us
one>
nginx-deployment-b49b9cbb4-hhqc9          1/1     Running   0          5s    192.168.160.84    node1.younescc-157245.labs1-pg0.utah.cloudlab.us
one>
nginx-deployment-b49b9cbb4-p54tw          1/1     Running   0          5s    192.168.176.204   node4.younescc-157245.labs1-pg0.utah.cloudlab.us
one>
nginx-deployment-b49b9cbb4-zcfhk          1/1     Running   0          5s    192.168.66.203    node3.younescc-157245.labs1-pg0.utah.cloudlab.us
one>
```

Figure 4.7 – Résultat d'ordonnancement de Pod AntiAffinity

2. Implémentation de la stratégie aléatoire

Dans l'intérêt de tester d'autres travaux d'ordonnancement, nous allons écrire un ordonnanceur à l'aide de python et l'exécuter en tant que pod autonome sur notre cluster et le faire servir de l'ordonnanceur pour tous les pods.

Cet ordonnanceur consiste dans la première phase de son travail à extraire tous les nœuds disponibles dans le cluster pour l'ordonnancement, ensuite il sélectionne au hasard l'un de ces nœuds et planifie le pod dessus.

2.1. Création de l'image docker

Premièrement, nous avons créé dans notre cluster un dossier nommé « Aleatoire » qui contient tous les fichiers nécessaires pour l'image docker de notre ordonnanceur.

```
root@node0:~/Aleatoire# ls
aleat.py  Dockerfile  requirements.txt
```

Figure 4.8 – Le dossier d'ordonnanceur aléatoire

aleat.py : représente le code de planificateur en python.

requirements.txt : le fichier qui contient la liste des packages nécessaires.

Le Dockerfile est un document nécessaire à ajouter car il contient des instructions pour créer l'image Docker de notre ordonnanceur.

```
FROM python:3.11.4
WORKDIR /root/Docker
COPY requirements.txt /root/Docker/
RUN pip install -r requirements.txt
COPY aleat.py /root/Docker/
CMD ["python", "aleat.py"]

~
~
~
```

Figure 4.9 – Le Dockerfile d'ordonnanceur aléatoire

En deuxième lieu, nous avons créé le répertoire de l'image sur dockerhub comme le montre la figure 4.10.

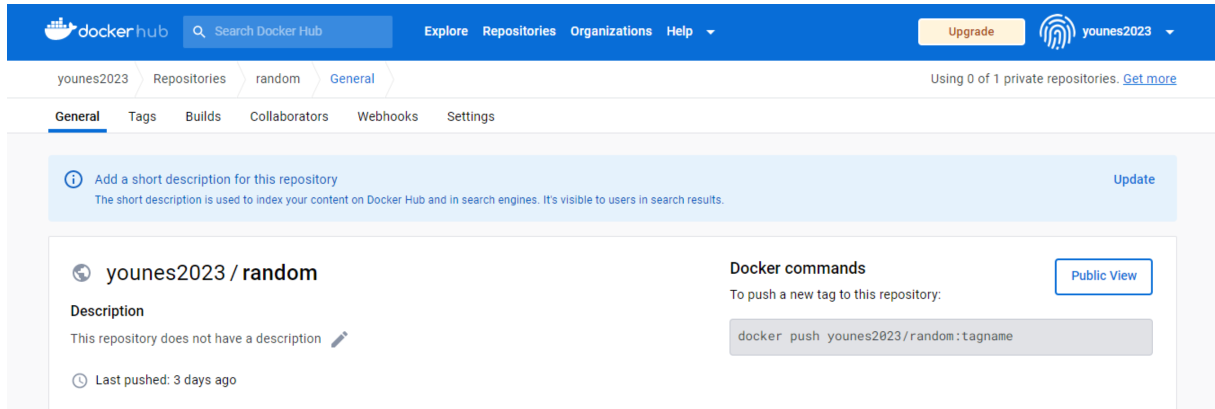

Figure 4.10 – Le répertoire dockerhub de l'image docker

Les étapes de création d'image docker de l'ordonnanceur aléatoire sont décrites dans les figures suivantes.

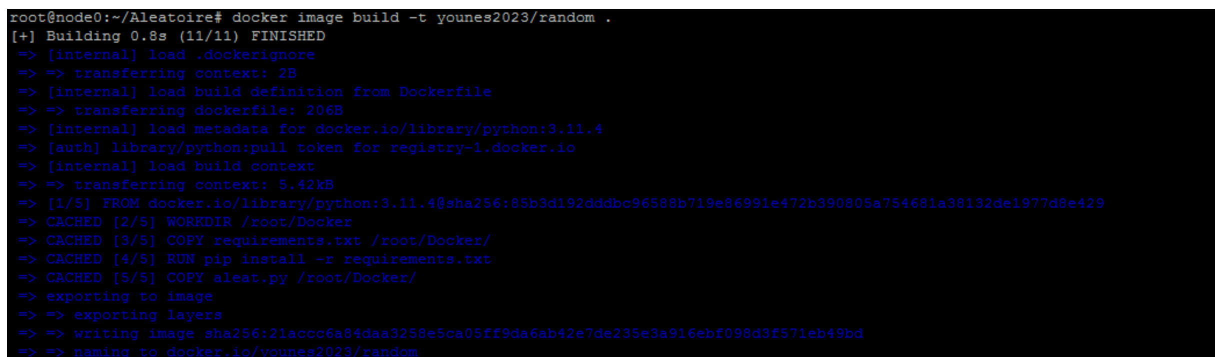

Figure 4.11 – Construction de l'image docker

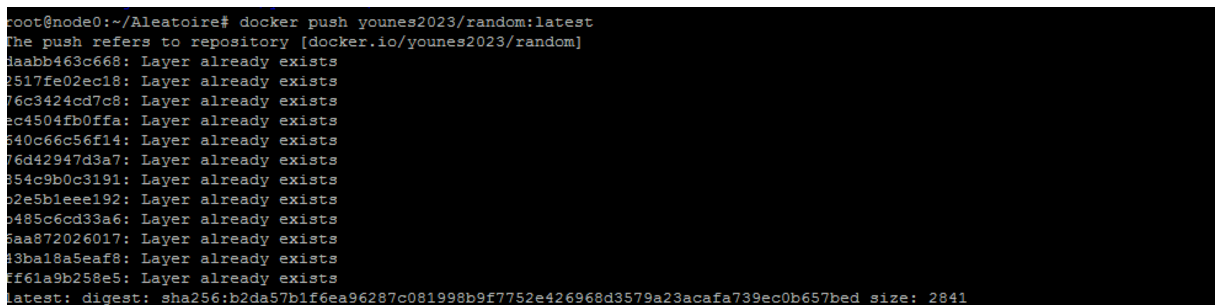

Figure 4.12 – Le partage d'image docker

Nous pouvons maintenant télécharger notre image à partir dockerhub

```
root@node0:~/Aleatoire# docker pull younes2023/random:latest
latest: Pulling from younes2023/random
Digest: sha256:b2da57b1f6ea96287c081998b9f7752e426968d3579a23acafa739ec0b657bed
Status: Image is up to date for younes2023/random:latest
docker.io/younes2023/random:latest
```

Figure 4.13 – Téléchargement de l’image docker

2.2. Configuration et exécution d’ordonnanceur

Nous exécutons l’ordonnanceur comme un pod dans l’espace de nom « kube-system », voici la figure 4.14 qui montre la configuration de notre ordonnanceur.

```yaml
metadata:
  name: aleat-scheduler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: aleat-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: aleat-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: aleat-scheduler
  namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      serviceAccountName: aleat-scheduler
      containers:
      - image: younes2023/random
        name: aleat-scheduler
```

Figure 4.14 – Fichier de configuration de l'ordonnanceur aléatoire

Afin de tester notre ordonnanceur, nous déployons un nombre de pods qui seront exécutés dans l'espace de nom « resys », voici la figure 4.15 qui montre la configuration des pods.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: resys
  labels:
    name: multischeduler-example
    schedulingStrategy: meetup
    serviceName: annotation-second-scheduler
spec:
  schedulerName: aleat-scheduler
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Figure 4.15 – Fichier de configuration de pod

En dernière étape, nous vérifions que les pods déployés sont correctement exécutés, comme le montre la figure 4.16.

```
root@node0:~# kubectl get pods -n resys -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP                NODE                                          NOMINATED NODE   READINESS GATES
apache-pod    1/1     Running   0          19s   192.168.212.102   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
apache-pod2   1/1     Running   0          19s   192.168.1.40      node1.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
apache-pod3   1/1     Running   0          19s   192.168.212.103   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod   1/1     Running   0          19s   192.168.1.39      node1.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod2  1/1     Running   0          19s   192.168.25.113    node4.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod3  1/1     Running   0          19s   192.168.51.88     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod     1/1     Running   0          19s   192.168.51.86     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod2    1/1     Running   0          19s   192.168.51.87     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod3    1/1     Running   0          19s   192.168.25.112    node4.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
```

Figure 4.16 – Résultat d'ordonnancement aléatoire

3. Implémentation de la stratégie proposée

3.1. Création de l'image docker

De la même façon que le planificateur aléatoire, nous créons un dossier nommé «Scheduler » qui contient tous les fichiers nécessaires pour l'image docker de notre ordonnanceur et également son répertoire sur dockerhub.

```
root@node0:~/Scheduler# ls
ant_colony.py  cpu.csv     fs.csv              nodebw.csv   nodefs.csv   ram.csv           test_ant_colony.py
bw.csv         Dockerfile  k8s-BWA-Scheduler.py  nodecpu.csv  noderam.csv  requirements.txt  test_cluster_scoring.py
root@node0:~/Scheduler#
```

Figure 4.17 – Le dossier d'ordonnanceur K8s-BWA

```
From python:3.11.4
WORKDIR /root/Scheduler
COPY requirements.txt /root/Scheduler
RUN pip install -r requirements.txt
COPY . /root/Scheduler
CMD ["python", "k8s-BWA-Scheduler.py"]
```

Figure 4.18 – Le Dockerfile d'ordonnanceur K8s-BWA

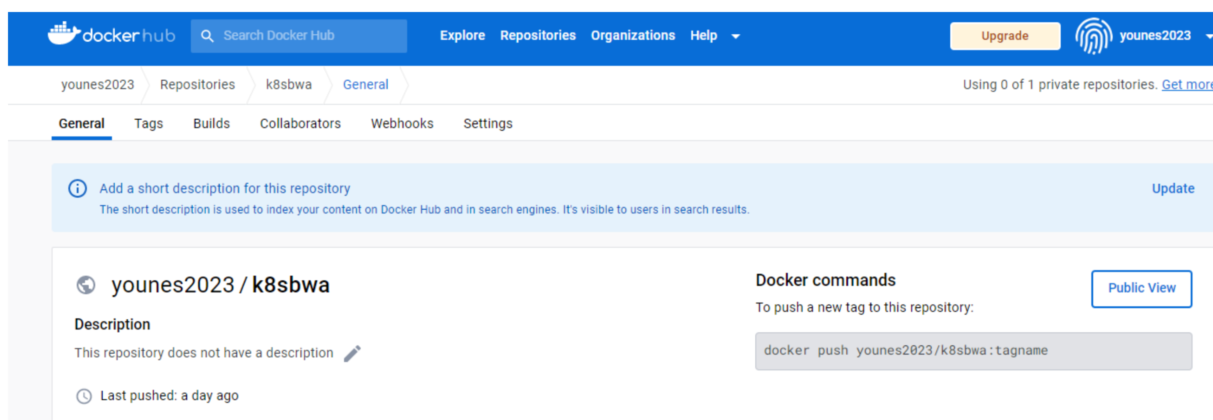

Figure 4.19 – Le répertoire dockerhub de l'image d'ordonnanceur

Les suites des étapes pour la création d'image docker sont identiques à l'ordonnanceur précédent sauf que nous allons travailler avec le nouveau répertoire dans le nouvel emplacement sur notre cluster.

3.2. Récupération des métriques des pods et nœuds

Après avoir créé l'image docker de notre planificateur, nous devons récupérer les valeurs de la bande passante, cpu, stockage et la ram des pods et nœuds. Pour cela nous avons écrit un code python (scraping.py) en utilisant l'Api Client Prometheus.

Prometheus est connecté à l'adresse IP du nœud master et affiche les résultats souhaités sous forme des fichiers csv.

Voici les figures qui illustrent les valeurs des métriques récupérées.

bw.csv

```
pod,transmis,recieve,bw

[' nginx-pod'],[' 0'],[' 0'],[0.0]

[' apache-pod2'],[' 193.39379982529854'],[' 97.63945904701401'],[291.03325887231256]

[' nginx-pod2'],[' 98.58306216965377'],[' 98.58306216965377'],[197.16612433930754]

[' apache-pod'],[' 0'],[' 0'],[0.0]

[' busybox-pod'],[' 0'],[' 0'],[0.0]

[' apache-pod3'],[' 0'],[' 0'],[0.0]

[' busybox-pod3'],[' 0'],[' 0'],[0.0]

[' nginx-pod3'],[' 100.39563522651305'],[' 100.39563522651305'],[200.7912704530261]

[' busybox-pod2'],[' 100.97092865140165'],[' 100.97092865140165'],[201.9418573028033]
```

Figure 4.20 – Les paquets échangés entre les pods de l'espace de nom resys

cpu.csv

```
pod,value

[' nginx-pod'],[' 0']

[' apache-pod2'],[' 0.0006678964794302725']

[' nginx-pod2'],[' 0']

[' apache-pod'],[' 0.00008405425047075811']

[' busybox-pod'],[' 0']

[' apache-pod3'],[' 0.00008361131107492798']

[' busybox-pod3'],[' 0']

[' nginx-pod3'],[' 0']

[' busybox-pod2'],[' 0.00036154642153866197']
```

Figure 4.21 – Les valeurs cpu des pods

```
fs.csv
1   pod,value
2
3   [' busybox-pod3'],[' 0']
4
5   [' busybox-pod'],[' 0']
6
7   [' apache-pod'],[' 0']
8
9   [' nginx-pod2'],[' 0']
10
11  [' apache-pod2'],[' 0']
12
13  [' nginx-pod3'],[' 0']
14
15  [' apache-pod3'],[' 0']
16
17  [' busybox-pod2'],[' 0']
18
19  [' nginx-pod'],[' 75425.4305882353']
```

Figure 4.22 – Les valeurs du stockage des pods

```
ram.csv
1   pod,value
2
3   [' nginx-pod'],[' 85704704']
4
5   [' apache-pod2'],[' 74874880']
6
7   [' nginx-pod2'],[' 86966272']
8
9   [' apache-pod'],[' 71368704']
10
11  [' busybox-pod'],[' 5832704']
12
13  [' apache-pod3'],[' 72556544']
14
15  [' busybox-pod3'],[' 6373376']
16
17  [' nginx-pod3'],[' 86843392']
18
19  [' busybox-pod2'],[' 7684096']
```

Figure 4.23 – Les valeurs de la ram des pods

```
nodebw.csv
1   node,transmis,recieve,bw
2
3   [' 192.168.1.8'],[' 12553866962'],[' 507896104'],[13061763066.0]
4
5   [' 192.168.212.68'],[' 12632545417'],[' 510294459'],[13142839876.0]
6
7   [' 192.168.25.71'],[' 12715570711'],[' 511340414'],[13226911125.0]
8
9   [' 192.168.51.66'],[' 12671880774'],[' 506661474'],[13178542248.0]
```

Figure 4.24 – Les valeurs de la bande passante des nœuds

```
nodecpu.csv
1   node,value
2
3   [' 192.168.1.8'],[' 13135351710.80678']
4
5   [' 192.168.212.68'],[' 14028444550.508472']
6
7   [' 192.168.25.71'],[' 12852287022.861015']
8
9   [' 192.168.51.66'],[' 13451880815.945763']
```

Figure 4.25 – Les valeurs cpu des nœuds

```
nodefs.csv
1   node,value
2
3   [' 192.168.1.8'],[' 40732972146688']
4
5   [' 192.168.212.68'],[' 40734108643328']
6
7   [' 192.168.25.71'],[' 40722350358528']
8
9   [' 192.168.51.66'],[' 40727598936064']
```

Figure 4.26 – Les valeurs du stockage des nœuds

```
noderam.csv
1   node,value
2
3   [' 192.168.1.8'],[' 13020198314.47715']
4
5   [' 192.168.212.68'],[' 14028444550.508472']
6
7   [' 192.168.25.71'],[' 12852287022.861015']
8
9   [' 192.168.51.66'],[' 13451880815.945763']
```

Figure 4.27 – Les valeurs de la ram des nœuds

3.3. Regroupement des pods

Après la récupération des valeurs mentionnées dans la phase précédente, les pods ayant un trafic de communication important entre eux sont regroupés dans le même cluster, réduisant ainsi le nombre des clusters, ceci sera fait à travers l'algorithme de clustering hiérarchique.

La liste résultante de l'algorithme de clustering hiérarchique contenant les identifiants des clusters avec les pods contenus dans ceux-ci est affichée dans la figure suivante.

```
Cluster with Pod Names
 {1: [' nginx-pod', ' apache-pod', ' busybox-pod', ' apache-pod3', ' busybox-pod3'], 2: [' apache-pod2'], 0: [' nginx-pod2', ' nginx-pod3', ' busybox-pod2']}
```

Figure 4.28 – Identifiants des clusters avec leurs pods

3.4. Placement des groupes de pods dans les nœuds avec l'algorithme ACO

La liste des clusters ainsi que les scores des nœuds par rapport aux clusters sont donnés comme paramètres à l'algorithme de colonie de fourmis pour faire leur placement dans les nœuds adéquats en se basant sur les scores comme le montre la figure 4.29.

```
 Clusters With Scheduling Nodes
{1: "[' 192.168.212.68']", 2: "[' 192.168.51.66']", 0: "[' 192.168.51.66']"}
```

Figure 4.29 – Identifiants des clusters avec le meilleur placement

3.5. Configuration et exécution

Nous exécutons notre ordonnanceur comme un pod dans l'espace de nom kube-system, voici la figure 4.30 qui montre la configuration de notre ordonnanceur.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: my-scheduler
  namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
    spec:
      serviceAccountName: my-scheduler
      containers:
      - image: younes2023/k8sbwa
        name: my-scheduler
```

Figure 4.30 – Fichier de configuration de l'ordonnanceur K8s-BWA

Afin de tester notre ordonnanceur, nous déployons un nombre de pods qui seront exécutés dans l'espace de nom « resys », voici la figure 4.31 qui montre la configuration des pods.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  namespace: resys
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
  - name: nginx-container
    image: nginx:latest
    ports:
    - containerPort: 80
```

Figure 4.31 – Fichier de configuration de pod

En dernière étape, nous vérifions que les pods déployés sont correctement exécutés.

```
root@node0:~# kubectl get pods -n resys -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP                NODE                                         NOMINATED NODE   READINESS GATES
apache-pod    1/1     Running   0          8s    192.168.212.108   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
apache-pod2   1/1     Running   0          8s    192.168.51.90     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
apache-pod3   1/1     Running   0          8s    192.168.212.107   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod   1/1     Running   0          8s    192.168.212.106   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod2  1/1     Running   0          8s    192.168.51.89     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
busybox-pod3  1/1     Running   0          8s    192.168.212.110   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod     1/1     Running   0          8s    192.168.212.109   node2.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod2    1/1     Running   0          8s    192.168.51.92     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
nginx-pod3    1/1     Running   0          8s    192.168.51.91     node3.kubernetes.labs1-pg0.utah.cloudlab.us   <none>           <none>
```

Figure 4.32 – Résultat d'ordonnancement avec la stratégie k8s-BWA

IVEvaluation

Dans l'intention de mesurer les performences des stratégies implémentées, nous avons écrit un code python (test.py) faisant la récupération des valeurs liées aux métriques de performence que nous avons choisi précédemment pour chaque nœud travailleur pendant une durée de dix minutes.

1. Analyse des résultats de la bande passante

Les figures 4.33, 4.34 et 4.35 montrent le changement de la bande passante pour chaque nœud en fonction du temps sous forme des courbes après une simulation de trafic entre les pods ordonnancés.

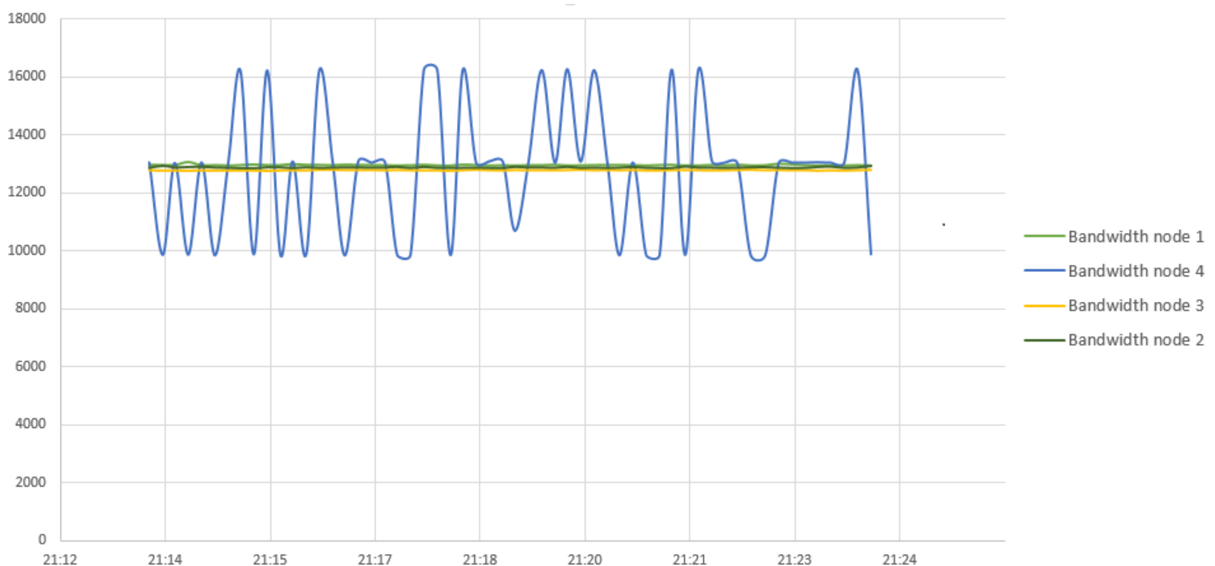

Figure 4.33 – L'usage de la bande passante avec la stratégie par défaut

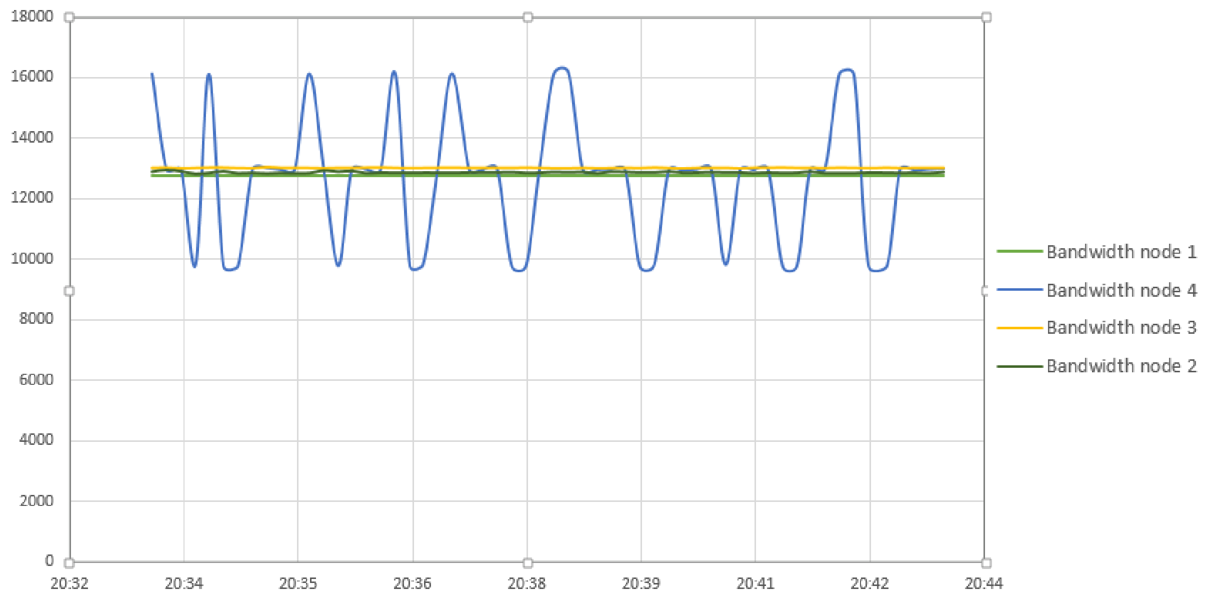

Figure 4.34 – L’usage de la bande passante avec la stratégie aléatoire

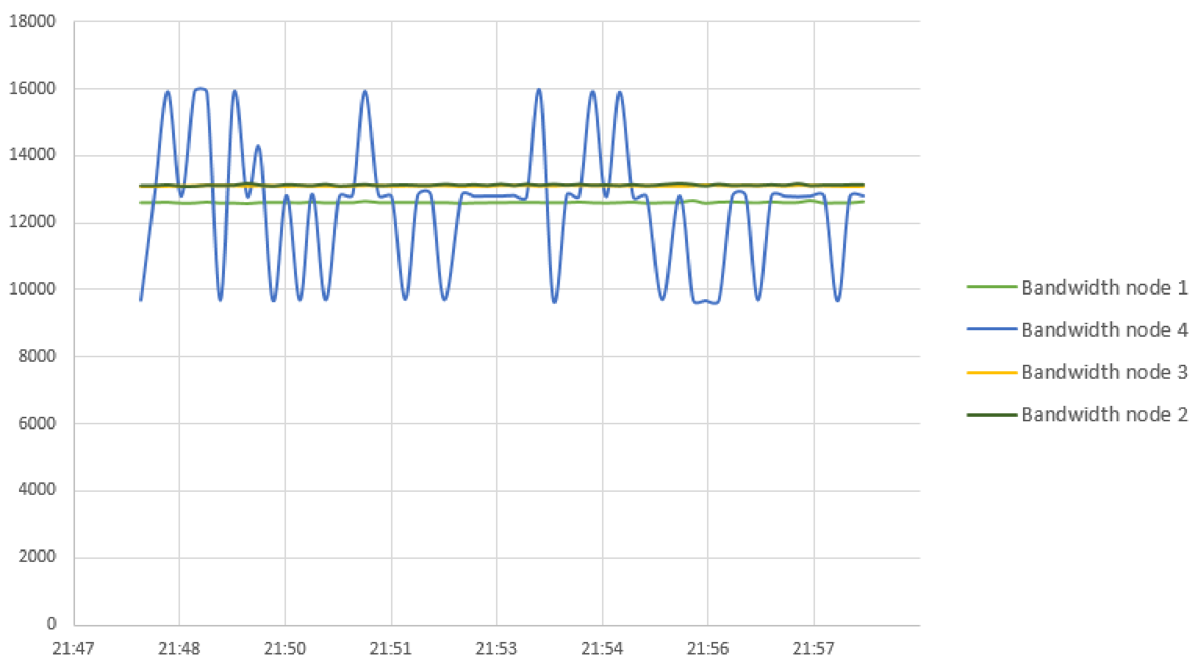

Figure 4.35 – L’usage de la bande passante avec la stratégie K8s-BWA

Dans la stratégie K8s-BWA représenté dans la figure 4.35, l'affectation des pods est faite dans les nœuds 2 et 3. La bande passante de ces deux nœuds est élevée par rapport aux autres nœuds, en voyant que pour les nœuds 2 et 3 la valeur de la bande passante est entre 13200 octets et pour les autres nœuds la valeur est moins de 13000 octets. Cette différence est due à la communication entre les nœuds travailleurs et le nœud master.

Comparant les résultats de notre stratégie avec la stratégie de kubernetes et aléatoire représentées dans les figures 4.33 et 4.34, nous notions que la bande passante est élevée pour tous les nœuds dans la stratégie aléatoire et kubernetes, les valeurs de la bande passante est environ 13000 octets pour chaque nœud renvoyées à l'affectation des pods dans tous les nœuds travailleurs.

2. Analyse des résultats du CPU

Les figures 4.36, 4.37 et 4.38 montrent l'usage du CPU pour chaque nœud en fonction du temps sous forme des courbes après une simulation de trafic entre les pods ordonnancés.

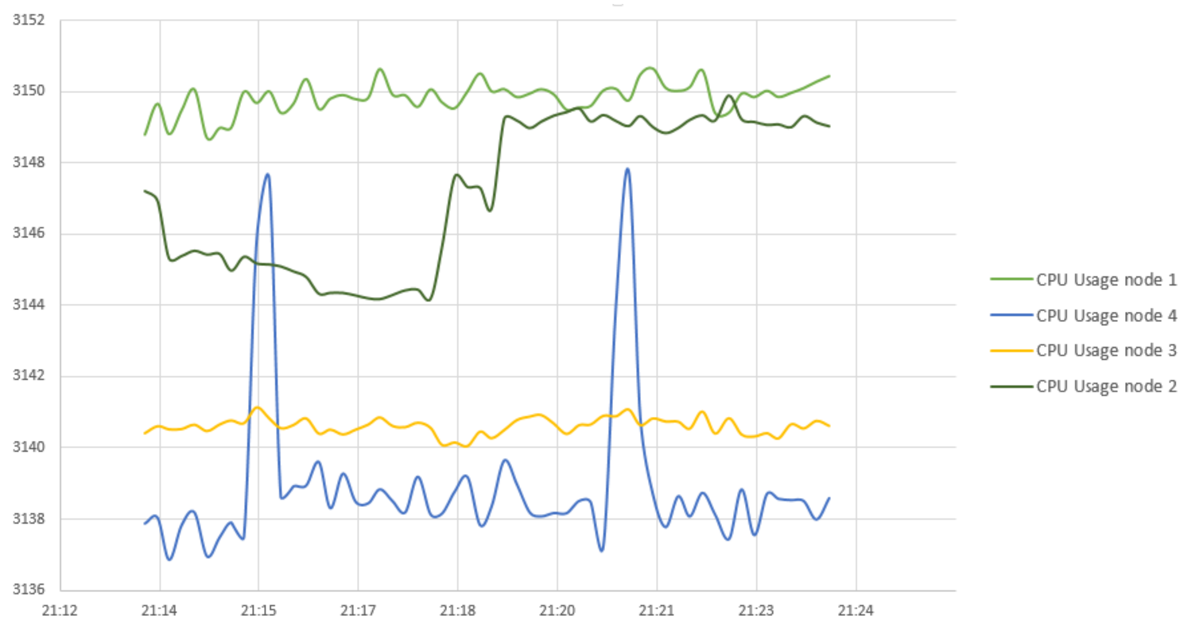

Figure 4.36 – L'usage du CPU avec la stratégie par défaut

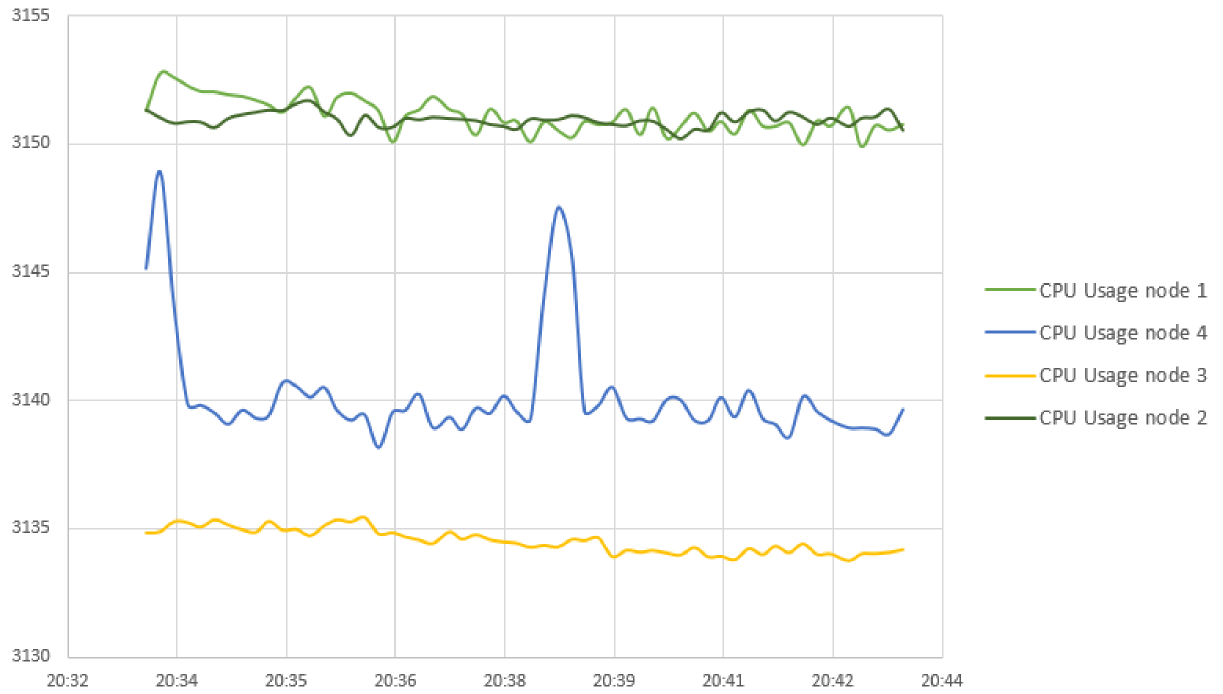

Figure 4.37 – L'usage du CPU avec la stratégie aléatoire

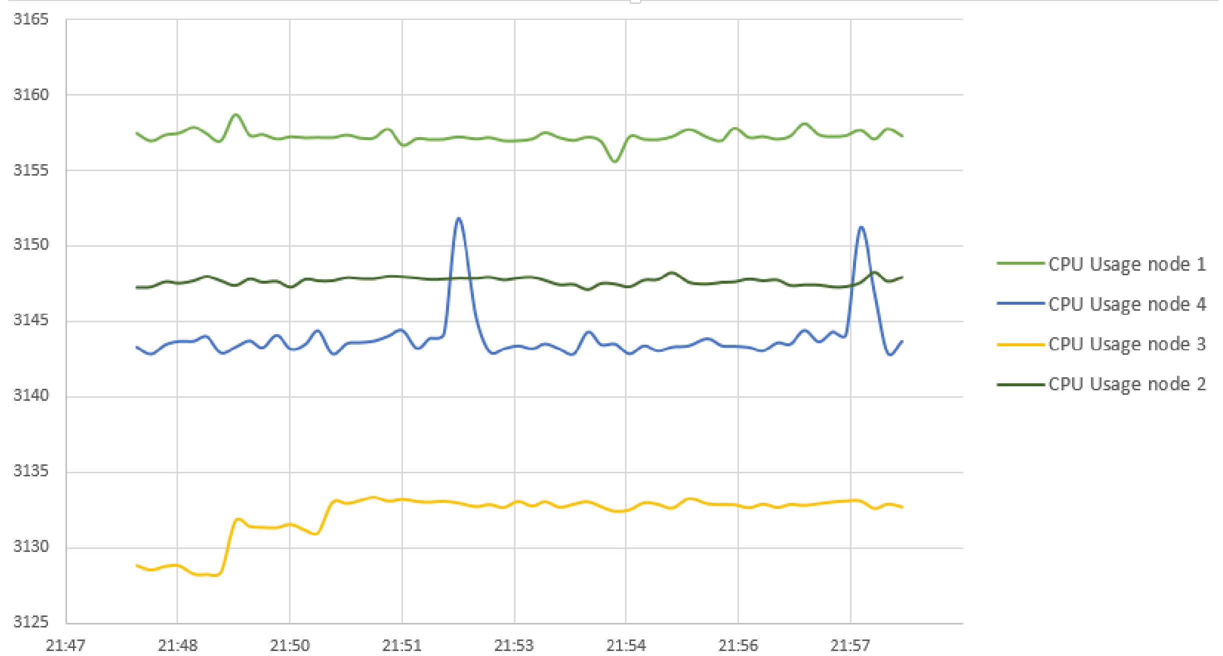

Figure 4.38 – L'usage du CPU avec la stratégie K8s-BWA

Dans la stratégie K8s-BWA représenté dans la figure 4.38, l'affectation des pods est faite dans les nœuds 2 et 3. Nous remarquons que l'usage du CPU par le nœud 3 est le minimum parmi les nœuds du cluster et l'usage du nœud 2 est stable dans une valeur moyenne, la différence avec les autres nœuds dû à la communication entre les nœuds travailleurs et le nœud master.

En comparant les valeurs du CPU entre les trois approches, nous constatons qu'avec notre solution, les valeurs dans le nœud 3 sont variées entre 3128 ms et 3132 ms tandis que ses valeurs dans la stratégie aléatoire sont 3134 ms et 3140 ms dans la stratégie par défaut. Cela causé par le trafic qui est effectué entre les pods du même nœud, ainsi il donne un équilibrage de charge. L'usage du CPU dans le nœud 2 est environ 3147 ms par rapport son usage dans la stratégie aléatoire qui utilise jusqu'à 3151 ms et 3149 ms dans l'approche kubernetes.

La différence des valeurs de la bande passante et CPU des nœuds qui contiennent les pods ordonnancés entre les trois stratégies est négligeable même si nous nous attendions une différence visible, mais ce résultat est dû à la simulation du trafic entre les pods car le volume et la charge du trafic ne sont pas assez élevés.

D'après les résultats de placement des pods ainsi l'usage de la bande passante avec le CPU, nous voyons que notre stratégie utilise moins de nœuds pour déployer les même pods faisant ainsi une consommation d'énergie, de plus vu que les pods sont regroupés dans des clusters en fonction de leur communication, on économise ainsi de la bande passante avec un équilibrage de charge par rapport aux autres stratégies.

V Conclusion

Dans ce dernier chapitre, nous avons abordé les différentes phases de l'implémentation de notre solution proposée, en commençant par la phase de récupération des valeurs de la bande passante des différents pods appartenant au même espace de nom, ensuite nous utilisons ces valeurs pour initier la seconde phase qui consiste à regrouper les pods qui ont un trafic important entre eux dans un même cluster en utilisant l'algorithme de clustering hiérarchique, et en dernier nous procédons au placement des groupes de pods résultant dans les nœuds appropriés qui est fait à travers l'algorithme de colonie de fourmis.

Nous avons montré l'implémentation de notre stratégie dans un environnement physique ainsi que l'ensemble des outils utilisés pour réaliser cette tâche. En dernier, nous avons illustré les résultats des expérimentations de notre ordonnanceur proposé, dans le but d'analyser son comportement. Et nous avons comparé notre ordonnanceur K8s-BWA-Scheduler avec la stratégie d'ordonnancement par défaut de kubernetes et la stratégie aléatoire.

Conclusion générale

Dans le cadre de ce travail, nous avons parlé du cloud computing et les différentes technologies qui assurent le bon fonctionnement des services cloud comme la virtualisation et la conteneurisation. L'évolution dans l'utilisation des conteneurs voyant ses avantages par rapport à la virtualisation nécessite l'usage des orchestrateurs qui répondent aux besoins des utilisateurs en lui offrant une haute qualité de service. Kubernetes prend en charge les services d'orchestration et devient plus vite une tendance dans les environnements de travail.

Kubernetes représente une solution open source efficace pour les applications à forte demande avec une configuration complexe. L'élément majeur dans le travail d'orchestration est l'ordonnanceur, ce dernier aide les utilisateurs à guider le déploiement des conteneurs et d'automatiser les mises à jour. L'ordonnanceur de kubernetes utilise des stratégies d'ordonnancement ne parviennent pas à satisfaire tous les besoins et elles ne s'adaptant pas à des charges dynamiques, c'est ce qui a poussé plusieurs travaux et études à élaborer des nouvelles stratégies d'ordonnancements de conteneurs essayant de combler les manques dans la stratégie par défaut de kubernetes.

Nous avons vu un ensemble de propositions pour l'ordonnancement en citant leurs points forts ainsi que leurs faiblesses. L'étude de ces travaux nous a aidés à implémenter notre propre solution et la comparer avec la stratégie par défaut et aléatoire selon des métriques adéquates dans le but réduire la consommation d'énergie ainsi que la bande passante. L'ordonnanceur proposé, nommé «K8s-BWA-Scheduler» (Kubernetes BandWidth Aware Scheduler) est réalisé en combinant deux algorithmes qui sont l'algorithme de clustering hiérarchique et l'algorithme de colonie de fourmi.

Nous avons montré à travers la validation et le mesure de performance des expérimentations la valeur ajoutée de notre ordonnanceur en termes de consommation d'énergie et de bande passante.

Ce travail, nous a permis de découvrir l'environnement de «Kubernetes» et les multiples mécanismes qui entrent en jeux dans le placement des pods dans les nœuds adéquats et d'acquérir de nouvelles connaissances dans le domaine de l'orchestration des conteneurs.

Comme perspectives, nous proposons l’amélioration de ce travail par :

- L’utilisation d’autres algorithmes autre que l’algorithme de colonie de fourmis pour l’affectation des pods dans les nœuds adéquats.
- Trouver un mécanisme pour réduire le nombre de groupes de pods exécutés sur un nœud dans le but d’éviter la surcharge de ce dernier ainsi que la congestion du réseau.

Bibliographie

Livres, Documents Web et Articles

[1] B. BENMAMMAR. : *Réseaux et communications - Gestion et contrôle des réseaux, ISTE Editions*, 2020.

[2] C.N. Hoefer, G. Karagiannis. : *Taxonomy of cloud computing services*, *4th IEEE Workshop on Enabling the Future Service-Oriented Internet* ,2010.

[3] Vania Goncalves, Pieter Ballon. : *Adding value to the network : Mobile operators,experiments with Software-as-a-Service and Patform-as-a-Service models, Telematics and Informatics*, 28 (1), 2011, pp. 12-21.

[4] Andrew Joint, Edwin Baker. *: Knowing the past to understand the present- issues in the contracting for cloud based services*, *Computer Law and Security Review*, 27, 2011, pp. 407-415.

[5] D.Hilly *: Cloud Computing: A Taxonomy of Platform and Infrastructure-level Offerings, Rapport technique CERCS, College of Computing Georgia Institute of Technology,* Avril 2009.

[6] What is network virtualization?. vmware.com. Disponible à l'adresse: https://www.vmware.com/topics/glossary/content/network-virtualization.html.Consulté le 14 décembre 2022.

[7] Qu'est-ce que la virtualisation du stockage? - définition de techopedia. theastrologypage.com. Disponible à l'adresse: https://fr.theastrologypage.com/storage-virtualization. Consulté le 14 décembre 2022.

[8] Qu'est-ce que la virtualisation d'applications? - définition de techopedia. theastrologypage.com. Disponible à l'adresse: https://fr.theastrologypage.com/application-virtualization. Consulté le 14 décembre 2022.

[9] Pourquoi virtualiser vos postes de travail ? . compufirst.com. Disponible à l'adresse: https://www.compufirst.com/rubrique-conseils/pourquoi-virtualiser-vos-postes-de-travail/main.do?appTreeId=43197. Consulté le 14 décembre 2022.

[10] Data Virtualization : qu’est-ce que la virtualisation de données ?. lebigdata.fr . Disponible à l'adresse: https://www.lebigdata.fr/data-virtualization. Consulté le 14 décembre 2022.

[11] What is Containerization? What are the Benefits?. veritas.com. Disponible à l’adresse: https://www.veritas.com/information-center/containerization. Consulté le 15 décembre 2022.

[12] Les différences entre la virtualisation et la conteneurisation. devopssec.fr. Disponible à l'adresse: https://devopssec.fr/article/differences-virtualisation-et-conteneurisation. Consulté le 16 décembre 2022.

[13] Vitor Goncalves da Silva, Marite Kirikova, Gundars Alksnis. : *Containers for Virtualization: An Overview, Applied Computer Systems,* 23(1), Mai 2018, pp. 21–27

[14] What are containers (container-based virtualization or containerization)?. techtarget.com. Disponible à l'adresse : https://www.techtarget.com/searchitoperations/definition/container-containerization-or-container-based-virtualization. Consulté le 16 décembre 2022.

[15] Understanding microservices . redhat.com. Disponible à l'adresse: https://www.redhat.com/en/topics/microservices. Consulté le 16 décembre 2022.

[16] Présentation des microservices. cloud.google.com. Disponible à l'adresse : https://cloud.google.com/architecture/microservices-architecture-introduction?hl=fr. Consulté le 16 décembre 2022.

[17] Amit M potdar, Narayan D G ,Shivaraj Kengond , Mohammed Moin Mulla. : *Performance evaluation of docker container and virtual machine,* t*hird international conférence on computing and network commuications (CoCoNet'19)*, *Procedia Computer Science* , 171, 2020, pp. 1419-1428.

[18] Docker Engine overview. docs.docker.com. Disponible à l'adresse: https://docs.docker.com/engine/. Consulté le 28 janvier 2023.

[19] Docker Architecture. aquasec.com. Disponible à l'adresse: https://www.aquasec.com/cloud-native-academy/docker-container/docker-architecture/. Consulté le 28 janvier 2023.

[20] Docker Registry. aquasec.com. Disponible à l'adresse : https://www.aquasec.com/cloud-native-academy/docker-container/docker-registry/. Consulté le 28 janvier 2023.

[21] What is Docker?. redhat.com. Disponible à l'adresse: https://www.redhat.com/en/topics/containers/what-is-docker. Consulté le 20 janvier 2023.

[22] N. Marathe, A. Gandhi and J. M. Shah. *: Docker Swarm and Kubernetes in Cloud Environement*, *third international conference on trends in electronics and inormatics (ICOEI) , Tirunelveli, India* , 2019, pp. 179-184.

[23] Introduction à MESOS. adaltas.com. Disponible à l'adresse: https://www.adaltas.com/fr/2017/11/15/introduction-mesos/. Consulté le 28 janvier 2023.

[24] overview. kubernetes.io. Disponible à l'adresse: https://kubernetes.io/docs/concepts/overview/. Consulté le 21 janvier 2023.

[25] Kubernetes. Disponible à l'adresse : https://en.wikipedia.org/wiki/Kubernetes. Consulté le 21 janvier 2023.

[26] K. Manaouil, A. Lebre. : *Kubernetes and the Edge? [Rapport de recherche] RR-9370, Inria Rennes - Bretagne Atlantique,* 2020, pp. 19.

[27] deployment . kubernetes.io. Disponible à l'adresse: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Consulté le : 23 janvier 2023.

[28] Kubernetes Namespace . vmware.com. Disponible à l'adresse: https://www.vmware.com/topics/glossary/content/kubernetes-namespace.html. Consulté le 23 janvier 2023.

[29] Namespaces. kubernetes.io. Disponible à l'adresse: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. Consulté le 23 janvier 2023.

[30] volumes. kubernetes.io. Disponible à l'adresse: https://kubernetes.io/docs/concepts/storage/volumes/. Consulté le 22 janvier 2023.

[31] Kubernetes Load Balancer. avinetworks.com. Disponible à l'adresse: https://avinetworks.com/glossary/kubernetes-load-balancer/. Consulté le 22 janvier 2023.

[32] Load balancer with Kubernetes. ovhcloud.com. Disponible à l'adresse: https://www.ovhcloud.com/en/public-cloud/kubernetes/kubernetes-load-balancer/ . Consulté le 23 janvier 2023.

[33] A sysadmin's guide to basic Kubernetes components. redhat.com. Disponible à l'adresse: https://www.redhat.com/sysadmin/kubernetes-components. Consulté le 25 janvier 2023.

[34] Kubernetes Components. kubernetes.io. Disponible à l'adresse: https://kubernetes.io/docs/concepts/overview/components/. Consulté le 22 janvier 2023.

[35] What is the Kubernetes API? .redhat.com.Disponible à l'adresse: https://www.redhat.com/en/topics/containers/what-is-the-kubernetes-API. Consulté le 25 janvier 2023.

[36] kubelet.jamesdefabia.github.io.Disponible à l'adresse: https://jamesdefabia.github.io/docs/admin/kubelet/. Consulté le 27 janvier 2023

[37] promacanthus.netlify.app.Scheduling strategy. Disponible à l'adresse :https://promacanthus.netlify.app/cloud-native/kubernetes/06-作业调度与资源管理/02-scheduler/. Consulté le 4 février 2023.

[38] github.com. Disponible à l’adresse: https://github.com/kubernetes/kubernetes/blob/e20c15174e96957c4e1faf7822125c7d295316a4/pkg/scheduler/algorithm/predicates/predicates.go#L1071. Consulté le 3 février 2023.

[39]github.com.Disponible à l’adresse: https://github.com/kubernetes/kubernetes/issues/8478. Consulté le 3 février 2023.

[40] github.com.Disponible à l’adresse: https://github.com/kubernetes/community/issues/530. Consulté le 4 février 2023.

[41] Advanced Scheduling and Pod Affinity and Anti-affinity.docs.openshift.com. Disponible à l'adresse: https://docs.openshift.com/container-platform/3.11/admin_guide/scheduling/pod_affinity.html. Consulté le 20 février 2023.

[42] taint and toleration .kubernetes.io. Disponible à l'adresse: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/. Consulté le 20 février 2023..

[43] Zhang Wei-guo, Ma Xi-lin,Zhang Jin-zhong. *: Research on Kubernetes' Resource Scheduling Scheme, Proceedings of the 8th International Conference on Communication and Network Security,* 2018, pp. 144-148.

[44] Qiang Guo. : *Task scheduling based on ant colony optimization in cloud environment*, *AIP Conference Proceedings*,1834 (1), 2017.

[45] Y. Fu et al. : *Progress-based Container Scheduling for Short-lived Applications in a Kubernetes Cluster*, *IEEE IEEE International Conference on Big Data (Big Data)*, Los Angeles, CA, USA, 2019, pp.278-287.

[46] J. Lv, M. Wei and Y. Yu. : *A Container Scheduling Strategy Based on Machine Learning in Microservice Architecture. 2019 IEEE International Conference on Services Computing (SCC)*, Milan, Italy, 2019, pp. 65-71 .

[47] J. Dong, X. Jin, H. Wang, Y. Li, P. Zhang and S. Cheng. *: Energy-Saving Virtual Machine Placement in Cloud Data Centers, 2013 13th IEEE/ACM International Symposium on Cluster,* Cloud, and Grid Computing, 2013, pp. 618-624.

[48] I.Ahmad, Mohammad Gh.AlFailakawi, A.AlMutawa et al. : *Container scheduling techniques: A survey and assessment*. Journal of king Saud University - Computer and Information Sciences, 34 (7), 2022, pp. 3887-4686

[49] cloudlab-overview.cloudlab.us. Disponible à l'adresse: https://www.cloudlab.us/files/cloudlab-overview.pdf. Consulté le 11 Juin 2023.

[50] Tarun Telang. : *Introduction to YAML.* Amazon Digital Services, 2020, p. 50.

[51] Ubuntu 18.04.1 LTS released, here is how to upgrade now. fosslinux.com. Disponible à l'adresse:https://www.fosslinux.com/4022/ubuntu-18-04-1-lts-released-here-is-how-to-upgrade-now.htm. Consulté le 11 Juin 2023.

[52] OVERVIEW.prometheus.io. Disponible à l'adresse: https://prometheus.io/docs/introduction/overview/. Consulté le 12 Juin 2023.

[53] What is Python? Executive Summary. python.org. Disponible à l'adresse: https://www.python.org/doc/essays/blurb/. Consulté le 12 Juin 2023.

[54] Client Libraries. pwittrock.github.io. Disponible à l'adresse: http://pwittrock.github.io/docs/reference/client-libraries/. Consulté le 12 Juin 2023.