Table des matières

Montée en puissance des microservices avec Kubernetes *1*

Déclaration ..... *i*

Remerciements ..... *ii*

Résumé ..... *iii*

Liste des tableaux ..... *vi*

Liste des figures ..... *vi*

1. Introduction *1*

1.1 DevOps **1**

1.2 Microservices **2**

1.3 Docker **2**

1.4 Kubernetes **3**

2. Problématique *5*

2.1 CQRS **5**

2.2 Publish and Subscribe **6**

3. Architecture Kubernetes *8*

3.1 Introduction **8**

3.2 Docker **8**

3.2.1 Docker deamon & docker-cli 8

3.2.2 Docker Image 8

3.2.3 Dockerfile 8

3.2.4 DockerHub 9

3.3 Pods **10**

3.4 Worker Node **11**

3.5 ReplicationController, ReplicaSet **12**

3.6 HorizontalPodAutoscalling **14**

3.7 Master Node **16**

3.8 Service **18**

3.8.1 NodePort : 19

3.8.2 LoadBalancer : 20

3.8.3 Ingress : 21

4. Accès aux services par une requête externe *22*

4.1 Introduction **22**

4.2 Networking **23**

4.3 Communication entre les ressources Kubernetes pour la montée en puissance ... **24**

4.3.1 Ajustement du nombre de pods vis-à-vis du trafic 24

4.3.2 Création des pods suite à une mise-à-jour de l'etcd 25

4.3.3 Sélection du worker node 26

5. Installation *28*

5.1 Docker 28

5.2 Minikube 30

5.2.1 Introduction 30
5.2.2 Prérequis 30
5.2.3 Virtualisation VT-x ou AMD-v 30
5.2.4 Hyperviseur 31
5.2.5 Kubectl 31
5.2.6 Minikube 32

6. Case study 33

6.1 Introduction 33

6.2 Création de l'image NBA avec Docker 33

6.2.1 Dockerfile 33
6.2.2 Docker CLI 35

6.3 Publier l'image sur DockerHub 36

6.4 Créer un container avec l'image NBA 36

6.5 Essai de montée en puissance 38

6.5.1 Introduction 38
6.5.2 Création minikube 38
6.5.3 Création ReplicaSet 41
6.5.4 Diminution manuelle du nombre de pods « nba » 42
6.5.5 NodePort Service 43
6.5.6 HPA 45

7. Conclusion 49

Bibliographie 50

Liste des tableaux

Aucune entrée de table d'illustration

Liste des figures

Figure 1 : Container vs VM 3
Figure 2 : CQRS 5
Figure 3 : Pod 10
Figure 4 : Worker node 11
Figure 5 : Schéma du rôle du Replication Controller 12
Figure 6 : Changement nombre de replicas 13
Figure 7 : HPA 14
Figure 8 : Cluster Kubernetes 16
Figure 9 : Service 18
Figure 10 : NodePort Service 19
Figure 11 : LoadBalancer 20
Figure 12 : Ingress 21
Figure 13 : Cluster NBA 22
Figure 14 : NodePort Networking 23
Figure 15 : Sequence diagram HPA 24
Figure 16 : Sequence diagram ReplicationController 25
Figure 17 : Worker nodes filter 26
Figure 18 : Sequence diagram Scheduler 27
Figure 19 : Téléchargement Docker MAC OS 28
Figure 20 : Drag & Drop Docer MAC OS 28
Figure 21 : Barre de statuts MAC OS 28
Figure 22 : Téléchargement Docker Microsoft Windows 29
Figure 23 : Barre de recherche Microsoft Windows 29
Figure 24 : Barre de status Microsoft Windows 29
Figure 25 : Message Docker Hyper-V 30
Figure 26 : Cluster NBA avec Minikube 33
Figure 27 : Dockerfile 34
Figure 28 : Construction de l'image dans le terminal 35
Figure 29 : Publication de l'image dans DockerHub 36
Figure 30 : Création du container 36
Figure 31 : Liste des containers 37
Figure 32 : Requête au container 37
Figure 33 : Programmes et fonctionnalités Windows 39
Figure 34 : Fonctionnalité Hyper-V 39
Figure 35 : Construction du Cluster 40
Figure 36 : Deployment descriptor 41
Figure 37 : Liste des ReplicaSets 41
Figure 38 : Liste des pods 42
Figure 39 : Diminution des pods "nba" .1 42
Figure 40 : Diminution des pods "nba" .2 42
Figure 41 : NodePort descriptor 43
Figure 42 : Liste des Services 43
Figure 43 : Informations du cluster 44
Figure 44 : Requête aux pods "nba" 44
Figure 45 : Démonstration de "LoadBalancing" 44
Figure 46 : HPA Descriptor 45
Figure 47 : Erreur terminal HPA 45

Figure 48 : Création HPA 46
Figure 49 : Liste HPA 46
Figure 50 : HPA après trafic sur le cluster 46
Figure 51 : Augmentation des pods nba avec HPA 47
Figure 52 : Disponibilité des nouveaux pods nba 47
Figure 53 : Diminution progressive des pods NBA 48
Figure 54 : Suppression des derniers pods nba 48

1. Introduction

1.1 DevOps

Il fut un temps, les applications fonctionnaient comme un seul processus pouvant être appelés monolithiques pour faire l'analogie entre l'application et un grand bloc de pierre. L'équipe de développement « *dev* » transmettait l'application à l'équipe opérationnelle « *ops* » qui se chargeait de la déployer sur les machines.

Celles-ci étaient de grandes tailles, ce qui rendait le cycle de release très lent et très peu fréquent. Avec les méthodologies de gestion de projet de type « waterfall », il fallait attendre 1 an ou plus avant de pouvoir déployer l'application sur une infrastructure.

La séparation entre ces deux équipes posait quelques problèmes. Les développeurs ne pensaient qu'à développer et ne se souciaient guère de comment fonctionnait leurs codes sur l'environnement de production. Tant que tout fonctionnait sur leurs machines, ils avaient fait leur part du boulot. Aujourd'hui, on voit les choses autrement:

> *They care not just about implementing user features, but also actively ensure their work flows smoothly and frequently through the entire value stream without causing chaos and disruption to IT Operations or any other internal or external customer.*[1]

L'équipe opérationnelle a pour but d'assurer la stabilité et la qualité de l'infrastructure. Elle préparait les machines à accueillir l'application avec toutes les dépendances nécessaires. Puis, une fois en production, elle assure que l'application ne tombe pas en panne et qu'elle respecte un certain niveau de performance.

Leurs objectifs respectifs créent alors un fossé entre les deux équipes et génèrent un conflit majeur. Les *devs* veulent coder le plus de fonctionnalités possibles dans un temps restreint, alors que les *ops* veulent un code stable qui respecte plusieurs critères de qualité qui ne concernent pas les fonctionnalités métier.

DevOps met en place une série d'étapes au développement du logiciel formant un cycle qui intègre les développeurs et les équipes opérationnelles. Des tâches sont automatisées : les tests unitaires, la gestion de la configuration des machines et le déploiement. La mise en production de nouvelles fonctionnalités est plus fréquente. Ce nouveau système permet d'améliorer la collaboration, la productivité de chaque équipe et permet d'être plus compétitif vis-à-vis de la concurrence.

Aujourd'hui, avec la livraison continue de valeur ajoutée que proposent les méthodes agiles, il devient primordial que ces deux équipes collaborent plus étroitement qu'auparavant.

Cependant, il est compliqué de s'adapter aussi rapidement aux changements si les composants des applications sont tous fortement couplés. Ces derniers peuvent nécessiter des modifications et être déployés toutes les 2 semaines, par exemple.

De plus, gérer la performance de grandes applications reste compliqué, car *scaler* (s'adapter à un changement d'ordre de grandeur des requêtes) horizontalement (dupliquer une ou plusieurs instances du produit) est impossible puisque les composants sont fortement couplés et les applications ne sont pas indépendantes. Scaler verticalement en ajoutant du CPU ou autre sur un serveur ne nécessite pas de changement au niveau du code mais est très couteux à l'inverse de l'horizontale.

1.2 Microservices

> *"Domain-driven design, Continuous delivery. On demand virtualization, infrastructure automation. Small autonomous teams. Systems at scale. Microservices have emerged from this world."* [2]

Les micros services répondent à cette problématique car ils sont tous autonomes, de petites tailles, sont plus simples à modifier, déployés indépendamment et sont par conséquent scalables ! Ils sont déployés dans des containers virtuels, communiquent entre eux avec un protocole REST.

Cela permet de réagir plus rapidement aux changements métier, d'être plus rapide contre la concurrence, aux DevOps d'établir une meilleure collaboration et de scaler horizontalement et donc d'économiser sur l'infrastructure en n'ayant pas à augmenter la capacité du serveur en CPU.

1.3 Docker

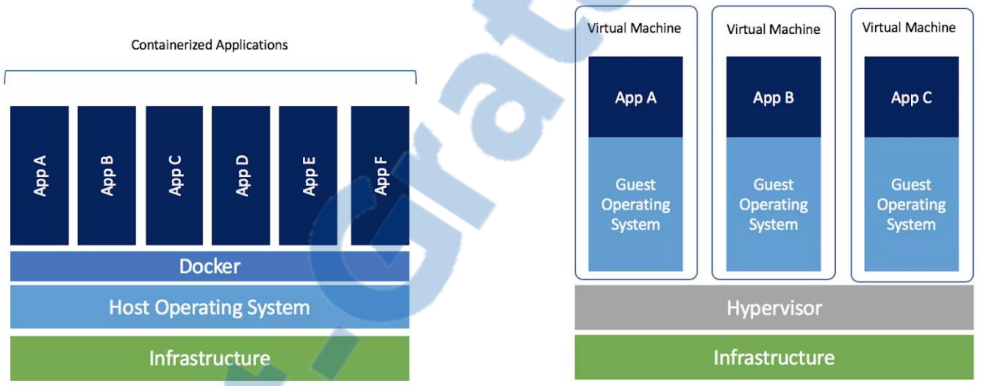
Docker est une technologie utilisée pour manager des containers virtuels et y insérer des services. [3]

Le but d'un container est de fournir un environnement isolé, léger, pour faire fonctionner une application. Cela permet de déployer plus facilement cette application sur n'importe quel serveur.

Un container peut être représenté comme une boîte qui fonctionne sur une base UNIX. Un container est beaucoup moins volumineux qu'une machine virtuelle (VM) car il n'a plus de couche de système d'exploitation. À contrario des machines virtuelles, les containers ne virtualisent pas l'hardware comme les VM. Ils sont directement en lien avec le Kernel VM du système d'exploitation, car il n'y a pas de couche qui les séparent. Cela permet aux containers de partager leurs librairies. (Voir Figure 1 : Container vs VM)

> *"Docker don't use hardware virtualization. Programs running inside Docker containers interface directly with the host Linux Kernel."* [4]

Figure 1 : Container vs VM



[5]

L'isolation de ces containers est permise grâce au Linux namespace.

1.4 Kubernetes

Les microservices peuvent être déployés de manière très rapide grâce à docker et peuvent être répliqués puisqu'ils sont tous indépendants. Toutefois, comment gérer tout cela manuellement si des containers sont dispersés dans un grand nombre de machines ?

Dans le cas où il y a un container qui pose un problème, comment le détecter ? Lequel aurait besoin de plus de réplications pour subvenir à une subite augmentation de la charge ? L'équipe opérationnelle monitore les serveurs et si un de ces derniers tombe en panne, il devra créer de nouveaux containers avec les services déchus dans un autre serveur.

Kubernetes permet justement d'automatiser le déploiement, la gestion de demande de puissance et la gestion des applications containerisées.[6]

Ainsi, l'équipe *ops* pourra déléguer la responsabilité de gérer les applications déployées à l'équipe dev et le système peut être programmé pour pouvoir s'autogérer sans la supervision d'un humain.

Kubernetes s'occupera de déployer un container avec l'application qui était présente dans la machine qui est tombée en panne, dans une machine qui a la capacité de l'accueillir.

Dans le cas où une machine est très fortement sollicitée et ne parvient plus à suivre à la sollicitation d'un ou plusieurs services, Kubernetes pourra être programmé pour réagir et déployer des containers offrant les services nécessaires sur d'autres machines et via un LoadBalancing, dispatcher les requêtes pour répondre à cette demande.

Puis bien évidemment, enlever des containers quand ils ne seront plus nécessaires et utilisent inutilement du temps computationnel.

Ce travail de Bachelor va décrire comment Kubernetes met en place cette fonctionnalité de gestion de la charge et en faire un *case study*.

Pour résumer, tous ces changements tant au niveau du développement d'une application que de son maintien, ont engendré de nouveaux concepts avec leurs nouvelles problématiques.

Les premières architectures d'applications étaient monolitques et leurs composants étaient fortement couplés. Dorénavant, celles-ci sont distribuées et basées sur des microservices indépendants, dispatchés sur plusieurs machines et gérés par un orchestrateur de conteneurs, Kubernetes.

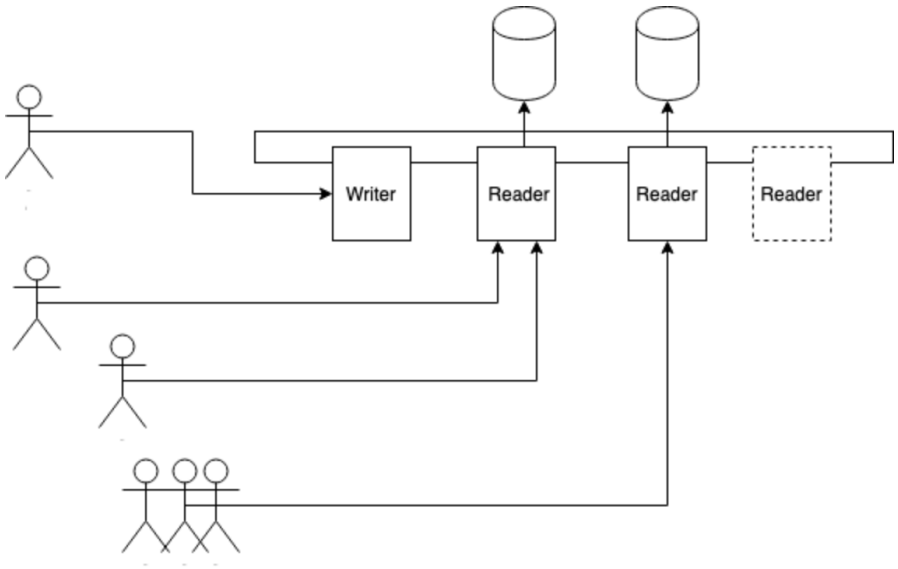
2. Problématique

Pour permettre de mieux comprendre certains aspects de la problématique de ce travail de Bachelor avec Kubernetes, les patterns ci-dessous seront utilisés à titre d'exemple afin de faire l'analogie de concept connu et des mécanismes utilisés par Kubernetes.

2.1 CQRS

Le pattern « Command and Query Responsability Segregation » repose sur la séparation entre des services d'écriture et de lecture. [7]

Figure 2 : CQRS



Sur le schéma ci-dessus, des clients font de multiples requêtes sur les services readers du pattern CQRS. (Voir Figure 2 : CQRS)

Généralement, les services sont plus souvent sollicités par des requêtes de lecture que des requêtes d'écriture.

Le pattern CQRS utilise un bus afin que les requêtes d'écriture propagent les données aux services de lecture intéressés. Cela permet la dissociation des deux différents services.

Il permet ainsi de créer autant de service reader (de lecture) que besoin. Il suffit de l'enregistrer dans le bus avec un certain sujet « topic » pour que le reader maintienne ses données à jour.

2.2 Publish and Subscribe

Pour éviter que les composants communiquent entre eux directement, les composants utilisent un bus de communication asynchrone pour y publier des informations selon un topic ou s'abonner à un topic.

Par exemple, un reader de données liées au cinéma veut s'abonner à un topic « film ». Ce reader s'enregistre sur le bus qui maintient une table des abonnés avec le topic correspondant.

Lorsqu'un writer publie une nouvelle donnée avec le topic « film », cette donnée est communiquée au bus. Ce dernier va retrouver dans sa table les services abonnés à ce topic et leur distribue l'information.

Dans le cas où les services readers subissent une forte sollicitation de la part des clients externes, on peut répondre à cette augmentation de charge en ajoutant un service reader (voir chapitre 2.1). C'est une action de montée en puissance. A l'inverse, avec une faible sollicitation des services on peut supprimer un service reader.

[8]

Avant l'existence de Kubernetes, l'équipe *ops* devrait se charger d'installer manuellement une nouvelle instance du reader sur un serveur dès qu'elle reçoit l'information que les services déjà déployés sont surchargés ou en panne.

La prise de conscience de cette augmentation de charge et le déploiement du service fait manuellement par l'équipe prend un certain temps, ce qui a pour conséquence l'indisponibilité des services pendant ce temps.

Kubernetes automatise ce processus, il se charge de créer ou supprimer des containers avec le service reader automatiquement.

Il sera expliqué dans ce manuscrit :

- Où les containers seront installés au sein de Kubernetes
    - A quel moment cela sera fait
    - Par quel moyen
- Comment fonctionnent et communiquent les composants de cette technologie
- Comment Kubernetes permet de monter en puissance automatiquement les services en surexploitation et comment il obtient les informations des containers
- Comment les clients pourront accéder à ces services, une fois que ces containers seront dispatchés dans plusieurs machines

3. Architecture Kubernetes

3.1 Introduction

Kubernetes est un gestionnaire de disponibilité de microcontainers. Sa responsabilité est de surveiller les différentes instances de microcontainers qu'il supervise et d'en réguler la réplication sur différentes machines en fonction de la montée en charge des requêtes. Toutefois, la manipulation des microcontainers eux-mêmes est de la responsabilité d'un gestionnaire de microcontainers avec lequel Kubernetes commnunique, Dans le cadre de ce mémoire, le gestionnaire choisi est Docker.

Le système de Kubernetes est constitué d'un master node lié à un ou plusieurs worker nodes. L'ensemble de ces nodes est appelé cluster.

3.2 Docker

Docker permet de rendre plus portable nos microservices en évitant de se soucier de l'infrastructure. A titre d'exemple, pour exécuter une application JAVA dans n'importe quel serveur, on utilise une Java Virtual Machine (JVM). Docker exécutera une image de notre application dans un container.

3.2.1 Docker deamon & docker-cli

Les utilisateurs de docker interagissent avec la machine docker (deamon) qui s'occupe de gérer les containers via des commandes exécutées dans l'interface de commande client, nommée docker CLI. Le docker CLI et le docker deamon communiquent grâce à des APIs REST.[9]

3.2.2 Docker Image

Une image est un ensemble de processus logiciels regroupant tous les fichiers nécessaires à l'exécution d'une application. [10]

3.2.3 Dockerfile

Le Dockerfile est un fichier qui permet de créer l'image de l'application. Il contient une série d'instructions que le deamon exécute pour la créer.[11]

3.2.4 DockerHub

DockerHub est un repository en ligne contenant des images d'applications. [12]

Les utilisateurs de ce service en ligne peuvent télécharger des images publiques que d'autres utilisateurs auront créées.

Ils peuvent les utiliser dans des nouvelles instances de containers ou alors mettre en ligne leurs images pour les réutiliser ou les partager avec la communauté de Docker.

Pour avoir recours à ce service, il faut au préalable créer un accès pour s'authentifier à la plateforme. Ce service est gratuit. Il propose un service premium afin de rendre privées les images de l'utilisateur.

Kubernetes utilise donc cet outil afin de pouvoir déployer, très rapidement et une multitude de fois, les services que les développeurs lui auront demandé de gérer.

Plus besoin d'avoir le code localement, s'il est disponible sur Dockerhub. Il lui suffira de télécharger l'image pour l'utiliser dans un container dans n'importe quel serveur.

Dans le chapitre 6.2, il sera expliqué comment créer une image.

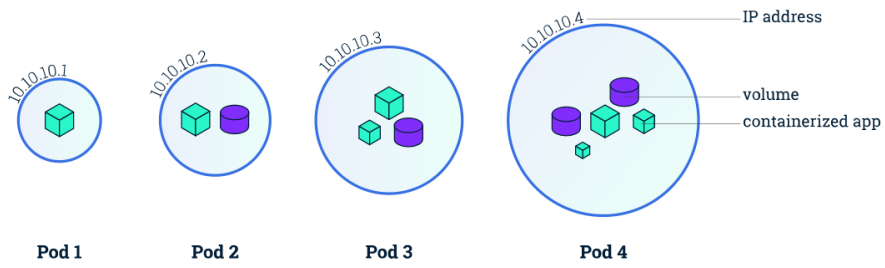
3.3 Pods

Les containers sont installés dans une ressource de Kubernetes appelé pod.

Un pod est un ensemble d'un ou plusieurs containers. (voir Figure 3 : Pod). En effet, Kubernetes ne fonctionne pas en interagissant directement avec les containers mais avec des pods. Ces pods peuvent être considérés comme des machines avec des adresses IP uniques.

> *A pod is a group of one or more tightly related containers that will always run together on the same worker node and in the same Linux namespace(s). Each pod is like a separate logical machine with its own IP, hostname, processes, and so on, running a single application.* [13]

Figure 3 : Pod



[14]

Sur la Figure 3 : Pod, les composants « volume » se trouvant dans les pods représentent un espace de stockage de données. Il en existe différents types.

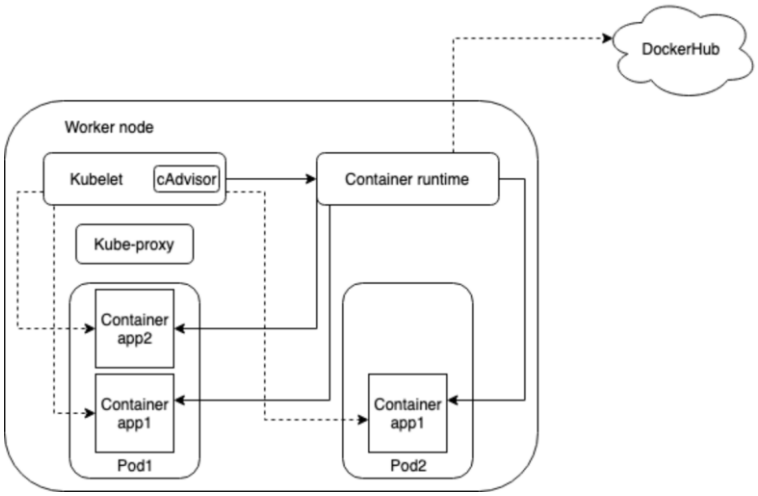
Chaque pod contient un fichier « descriptor » qui indique les paramètres d'installation du pod

Le pod est la ressource avec la plus petite granularité du système Kubernetes. Ils seront toujours hébergés au sein d'un worker node.

3.4 Worker Node

Un worker node (WN) est une machine physique ou une VM qui détient toutes les ressources nécessaires afin de garantir l'exécution d'un ou plusieurs pods. [15] Cette entité va héberger tous les services qu'un développeur aura décidé de déployer (Figure 4).

Figure 4 : Worker node



Chaque WN est composé d'un Kubelet, d'un kube-proxy et d'un Container runtime.

Le Kubelet a comme responsabilité de gérer tous les pods ainsi que leurs containers contenus dans le WN. Lui-même contient un composant appelé « cAdvisor » qui lui s'occupe de récupérer des métriques sur chaque container de chaque pods.

Le Kube-proxy a comme responsabilité de rediriger le trafic réseau vers les pods contenant les services requis. Pour cela, il contient une table « IPTable » qui détient les adresses des pods.

Le Container runtime est extérieur à Kubernetes. C'est le composant qui gère les containers dans un WN, Docker dans le cadre de notre mémoire.

Pour créer un container, le Kubelet inspecte le descriptor d'un pod du WN et communique les images qui y sont mentionnées au Container runtime via le protocole REST. Ce dernier télécharge l'image à partir de DockerHub si l'image n'est pas disponible localement et instancie un container avec cette image.

Si un pod est supprimé ou dysfonctionne, le Kubelet a la responsabilité d'arrêter le ou les containers qui y étaient hébergés via le container runtime.

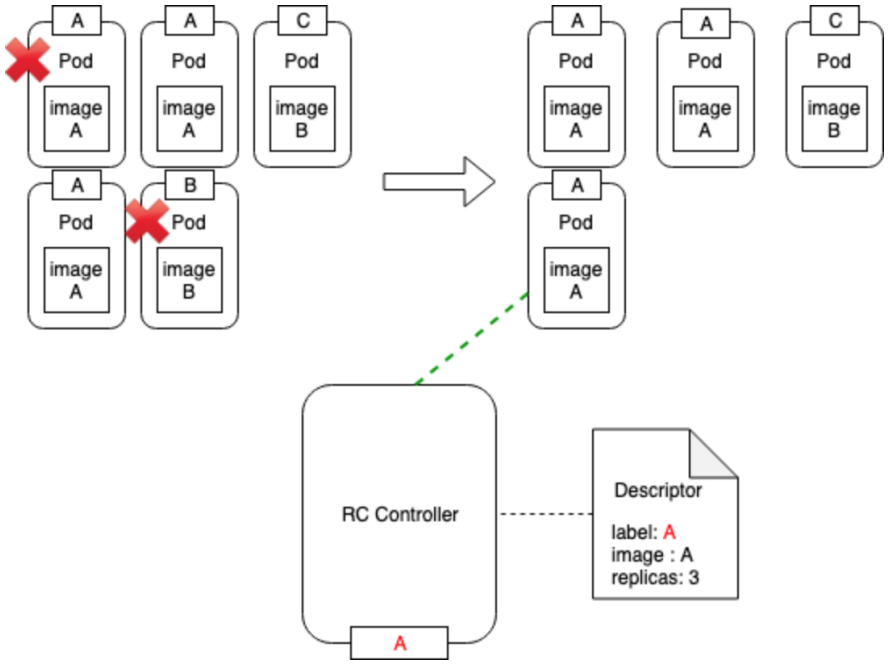
3.5 ReplicationController, ReplicaSet

Le ReplicationController (RC) a la charge de créer, supprimer et de maintenir une ou plusieurs instances d'un pod selon le nombre de réplicas mentionné dans le descriptor du RC. Il est situé au sein du master node (voir chapitre 3.7 Master Node)

Un RC s'occupe uniquement des pods qui contiennent un certain type d'image à qui on va assigner un label dans son descriptor. Si le descriptor mentionne uniquement le label A, il ne se préoccupera pas des pods ayant un label B. [16]

Dans la Figure 5 : Schéma du rôle du Replication Controller, nous pouvons voir que le descriptor du RC ne mentionne que le label « A » avec une image « A ».

Figure 5 : Schéma du rôle du Replication Controller



Un pod ayant le label « A » à gauche de la Figure 5 : Schéma du rôle du Replication Controller ne fonctionne plus.

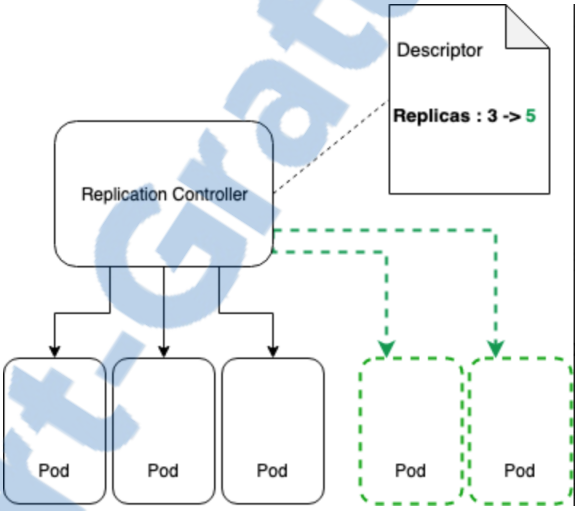
Le descriptor mentionne qu'il faut trois réplicas actifs alors qu'il n'y en a plus que deux. Par conséquent, le RC en crée un nouveau pour s'ajuster au descriptor. Mais ce n'est pas lui qui les assigne aux WN. L'assignation du pod à un WN est effectuée par le Scheduler, cette étape sera expliquée dans un prochain chapitre.

Le pod ayant le label « B » est supprimé lui aussi. Cependant, aucun RC ne s'en occupe. Il n'y aura plus aucun pod ayant le label B.

Répliquer un pod veut dire en créer un nouveau avec les mêmes caractéristiques. Dans le schéma de la figure 5 à droite, le pod lié au trait-tillé vert est un pod ayant les mêmes caractéristiques que les pods contenant le label « A ».

Le descriptor du RC indique combien de pods d'un certain type il doit maintenir en permanence. Ainsi, en augmentant manuellement ce nombre, le RC crée immédiatement de nouveaux pods de ce type. C'est une manière d'augmenter manuellement la disponibilité d'un service.

Figure 6 : Changement nombre de réplicas



Chaque type de pod étant géré par des RC distincts, il faudrait changer chacun de leurs descriptors respectifs pour les « scaler » ensemble.

Par exemple, admettons que le nombre de pod de type « FrontEnd » a été augmenté. Sachant qu'il communique toujours avec un pod « Backend », il pourrait être judicieux d'augmenter le nombre de ce dernier aussi.

Kubernetes utilise un autre concept pour gérer plusieurs types de pods simultanément : le ReplicaSet. Ce dernier fonctionne de la même manière qu'un RC, mais il est possible de sélectionner plusieurs types de pods avec des labels différents dans son descriptor.[1]

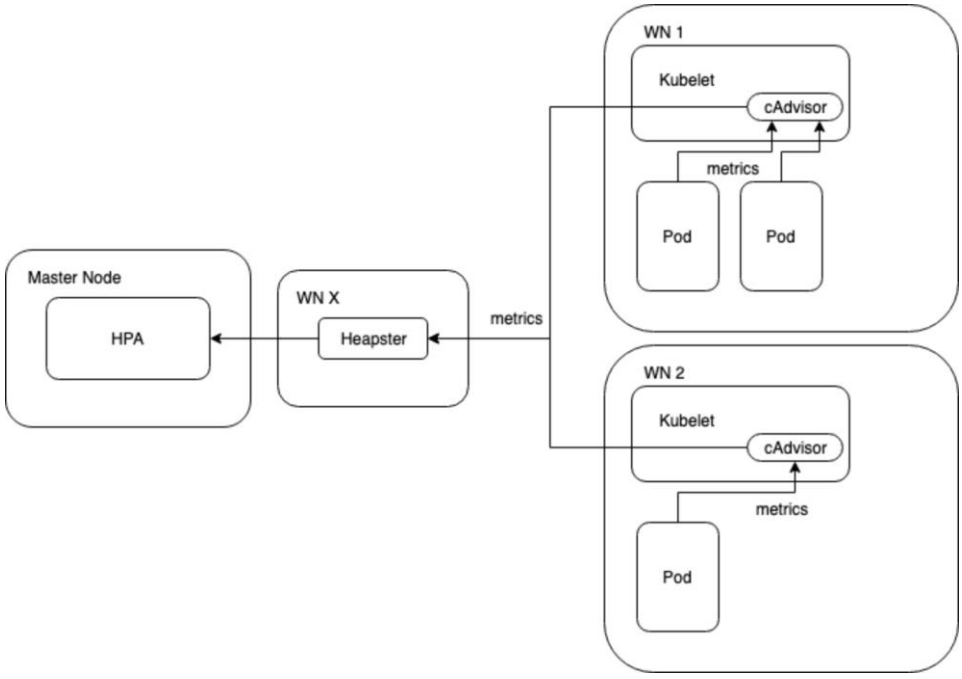
[1] Il semblerait que le ReplicaSet sera le seul concept pour gérer les pods dans les prochaines versions de Kubernetes [16]

Si la montée en charge d'un service est prévisible, l'équipe ops modifie le descriptor du RC en augmentant le nombre de pods contenant l'image correspondante. Par exemple si un site de vente en ligne publie ses nouveaux articles à minuit, il est prévisible que les clients se connectent majoritairement entre minuit et une heure du matin. Par conséquent, il est possible de scaler les pods avec les images correspondant au site de vente en ligne pour s'adapter à cette hausse du nombre de requêtes manuellement en changeant le nombre de réplicas dans le descriptor.

3.6 HorizontalPodAutoscalling

Le composant HorizontalPodAutoscaler (HPA) permet de gérer le besoin de montée en puissance lors d'augmentation de charge imprévisible.[17] Il sait ajuster le nombre de pod nécessaire quand un type de pod est surchargé en détectant une suractivité du CPU, ou via d'autres métriques du pod telles que le nombre de requêtes par seconde « QPS ».

Figure 7 : HPA



Ci-avant, nous avons vu que le composant « cAdvisor » contenu dans le Kubelet collecte les métriques provenant des containers. Le composant Heapster a pour responsabilité de collecter les métriques de tous les Kubelets de chaque worker node afin de les transmettre à l'HPA. Il est localisé au sein d'un WN, c'est pourquoi dans la Figure 7 : HPA, il se trouve dans un WN X mais il pourrait se situer dans le WN 1 ou WN 2.

A partir de ces métriques l'HPA calcule le nombre adéquat de pods pour s'ajuster aux métriques souhaitées qui ont été paramétrées dans le descriptor de l'HPA. Il communique ensuite avec l'API Server pour exploitation des autres composants de Kubernetes. [13] (chapitre 3.7 Master Node)

Le HPA peut ajuster le nombre de pods aussi bien vers le haut, quand il faut pouvoir monter en puissance, que vers le bas si les services disponibles occupent plus de ressources que nécessaire par rapport à la charge actuelle.

3.7 Master Node

Le master node a la responsabilité d'administrer le cluster. Il coordonne les activités telles que la mise en échelle des applications, la maintenance des applications à l'état désiré et la propagation des mises à jour.

> ***The Master is responsible for managing the cluster.*** *The master coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.* [18]

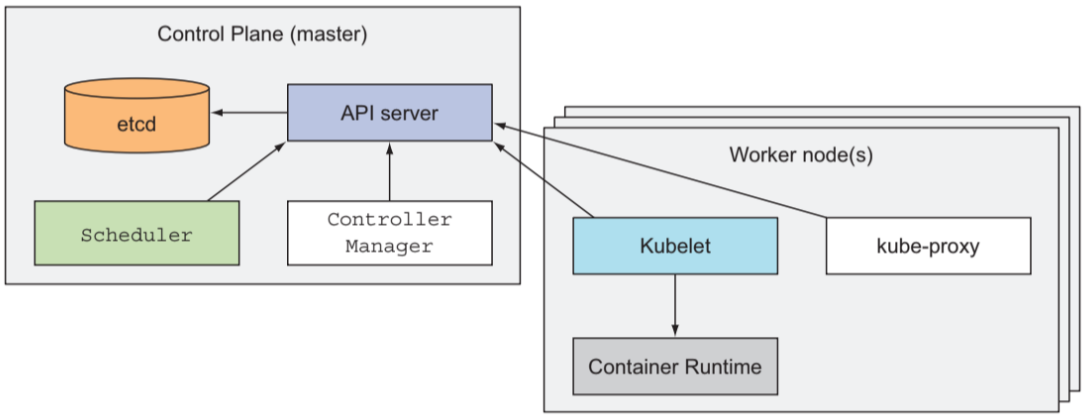
Composants du master node :

- Scheduler est chargé d'assigner les pods non-assignés aux worker nodes (voir chapitre 3.5 ReplicationController, ReplicaSet).
- Controller manager s'occupe de contrôler les worker nodes et de gérer les erreurs. Il en existe plusieurs types comme HPA, ReplicationController, etc.
- Etcd est une base de données qui stocke la configuration du cluster. Ce composant enregistre l'état actuel de l'ensemble des composants du cluster.
- API Server est un composant de communication utilisé pour mettre en relation le master avec les WNs. Il est aussi le seul à communiquer avec l'etcd et communique via le protocole REST.

[13]

Les composants du cluster communiquent avec l'API Server, car il est le seul à pouvoir communiquer avec l'etcd et par conséquent est le seul à pouvoir mettre à jour l'état du cluster.

Figure 8 : Cluster Kubernetes



[13]

Les composants de Kubernetes écoutent les évènements qui les intéressent selon leurs responsabilités au sein du système et vont réagir quand ils auront reçu un changement de l'état du cluster.

On appelle ce phénomène **watch** et il est très semblable au pattern Publish and Subscribe (voir chapitre 2.2 Publish and Subscribe). Les composants du cluster peuvent publier ou s'abonner à ces évènements.

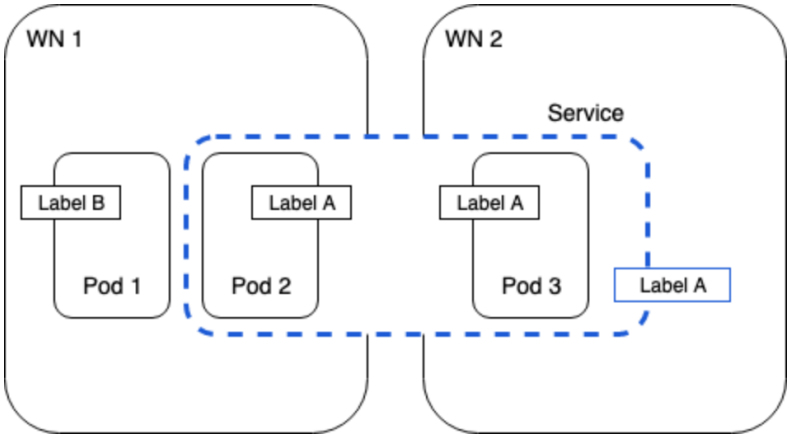
C'est avec ces évènements de publish et subscribe que les ressources de Kubernetes comme l'HPA, le RC et le Scheduler se coordonnent afin de réaliser une activité comme la montée en charge.

3.8 Service

Comme les ressources sont gérées dynamiquement par Kubernetes, il faut pouvoir disposer d'un point d'entrée sur les microservices car on ne connait pas à priori leurs adresses.

C'est le rôle de la notion abstraite nommée « Service » [19] qui est un regroupement virtuel de pods ayant un certain label.

Figure 9 : Service



Un « Service » c'est le point d'accès aux microservices contenus dans les images des containers contenus dans les pods. Ces derniers possèdent un label afin de pouvoir les identifier. La Figure 9 : Service montre que les pods avec le Label A sont regroupés dans un même « Service », même s'ils sont localisés sur différents worker nodes.

En s'adressant à ce « Service », un client externe peut effectuer une requête et atteindre un microservice dans le pod de label A sans connaître sa localisation exacte. Comme il y a plusieurs Pods par « Service », Kubernetes réalise ainsi une forme de load balancing : quand une requête arrive sur un « Service », Kubernetes lui assigne un Pod disponible.

Il existe plusieurs façons d'atteindre un Service, selon les techniques appelées :

- NodePort
- Loadbalancer
- Ingress

Ces méthodes d'accès se configurent via un deployment descriptor de « Service ».

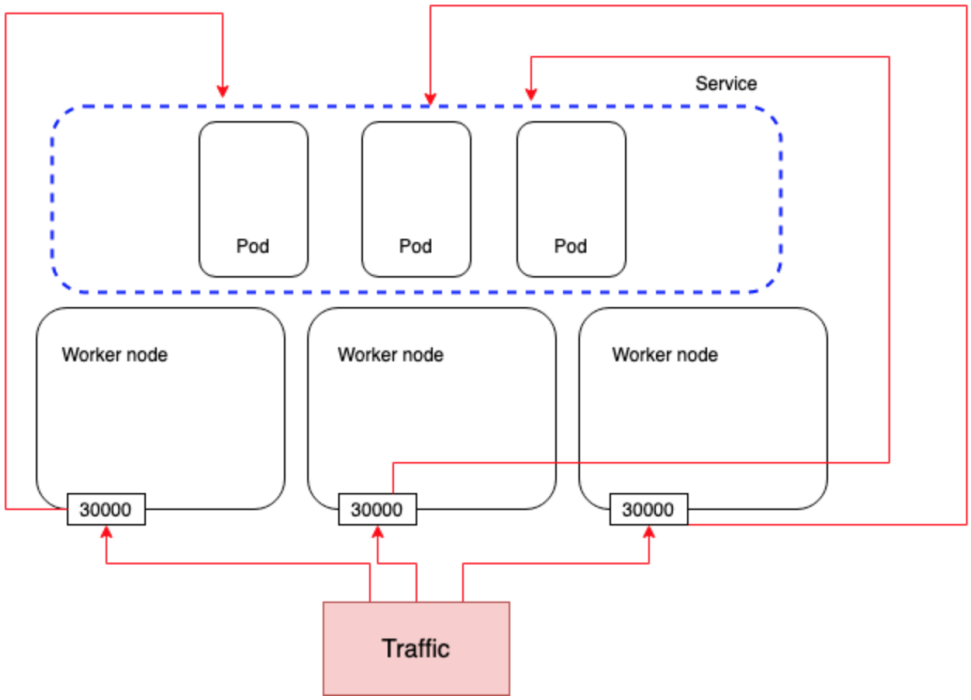
3.8.1 NodePort :

Dans ce mode, le Service est exposé via un port spécifique et commun sur chaque worker node. Sur la Figure 10 : NodePort Service nous voyons que le « Service » est atteignable via le port 30000 des WNs.

Ce port sera disponible sur tous les WNs, même ceux qui ne contiennent pas un pod qui est contenu par le « Service ».

Lorsqu'une requête d'un client externe arrive sur ce port, la requête est redirigée vers le « Service » qui la redirigera vers un des pods capable de répondre à la requête. C'est une redirection en deux étapes.

Figure 10 : NodePort Service



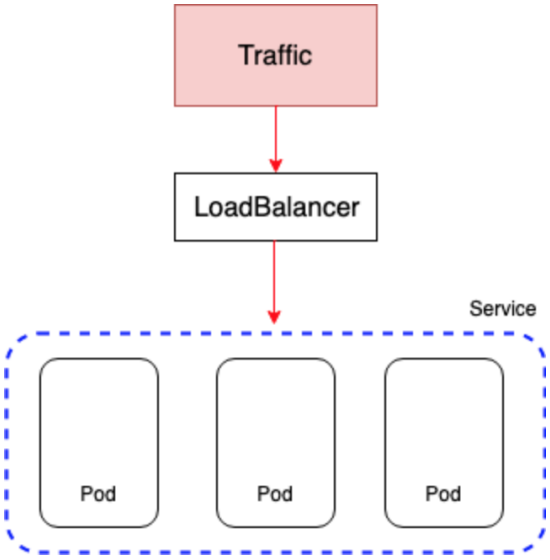
[20]

3.8.2 LoadBalancer :

Le LoadBalancer est un service externe à Kubernetes. Il permet de définir une adresse IP pour chaque « Service » et redirige toutes les requêtes vers le service. On doit donc se reposer sur un fournisseur externe pour pouvoir disposer d'un LoadBalancer. Les clusters provenant d'infrastructure cloud et payantes comme Google Kubernetes Engine (GKE) ou Azure Kubernetes Service, l'intègrent à leur offre. Avec ce mode d'accès, il faut un LoadBalancer par « Service », qui va être facturé par le provider de l'infrastructure cloud ce qui est coûteux si notre cluster offre beaucoup de services.

Le LoadBalancer peut supporter plusieurs protocoles, HTTP, TCP et UDP, par exemple.

Figure 11 : LoadBalancer



[20]

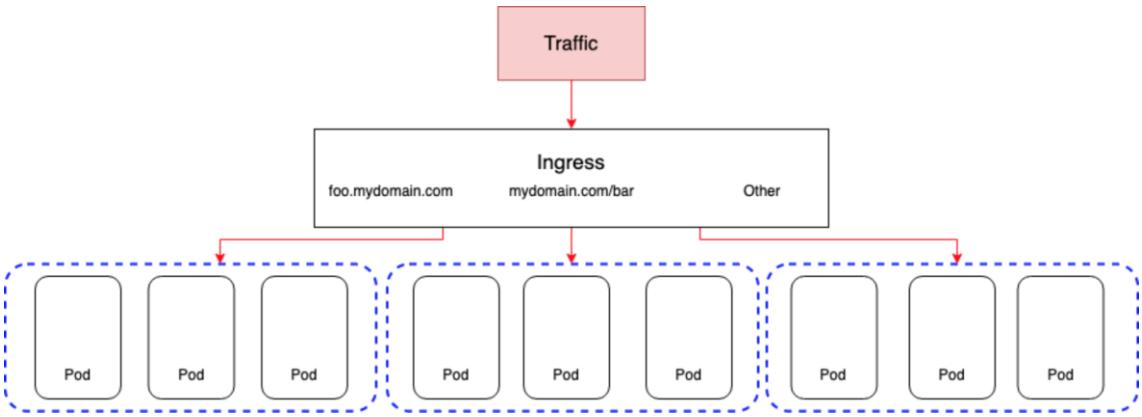
All traffic on the port you specify will be forwarded to the service. There is no filtering, no routing, etc. This means you can send almost any kind of traffic to it, like HTTP, TCP, UDP, Websockets, gRPC, or whatever. [20]

3.8.3 Ingress :

Dans ce mode interne à Kubernetes, il n'y a qu'un seul point d'entrée au trafic et donc une seule adresse IP pour l'ensemble des « Services ».

L'Ingress redirige les requêtes externes au bon « Service » en utilisant un chemin fourni dans la requête, car chaque service possède un nom de domaine. Par exemple : ***AdresseIPIngress****.foo.mydomain.com* redirigera la requête vers le « Service » tout à gauche de la Figure 12 : Ingress.

Figure 12 : Ingress



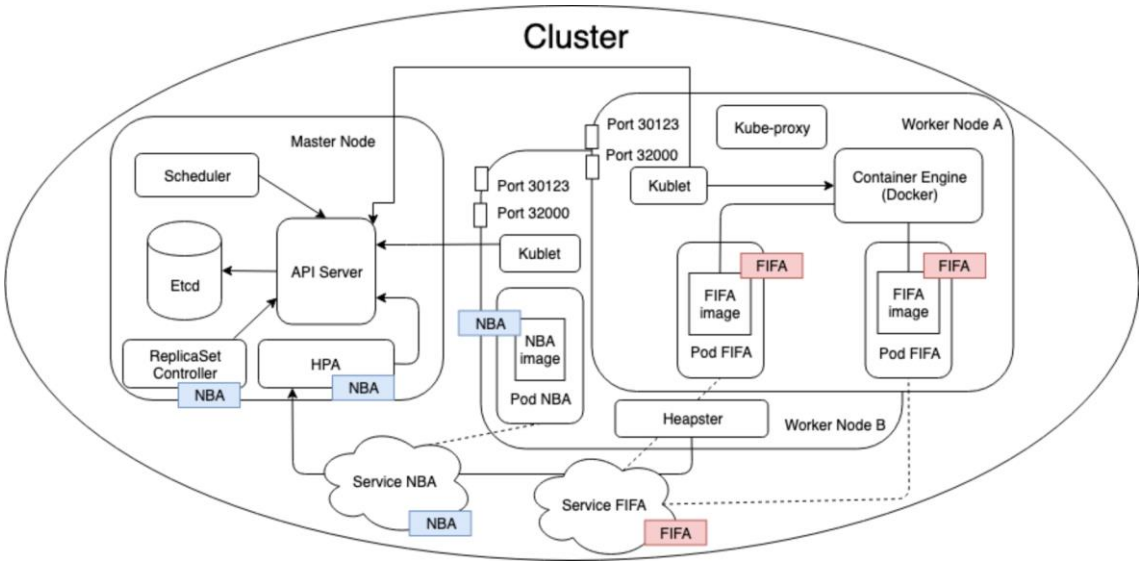
[20]

4. Accès aux services par une requête externe

4.1 Introduction

Nous présentons ci-après un exemple avec deux « Services » permettant d'obtenir des résultats de sport (FIFA et NBA). Nous définissons ainsi deux worker nodes, l'un pour les résultats FIFA et l'autre pour les résultats NBA (Figure 13 : Cluster NBA).

Figure 13 : Cluster NBA



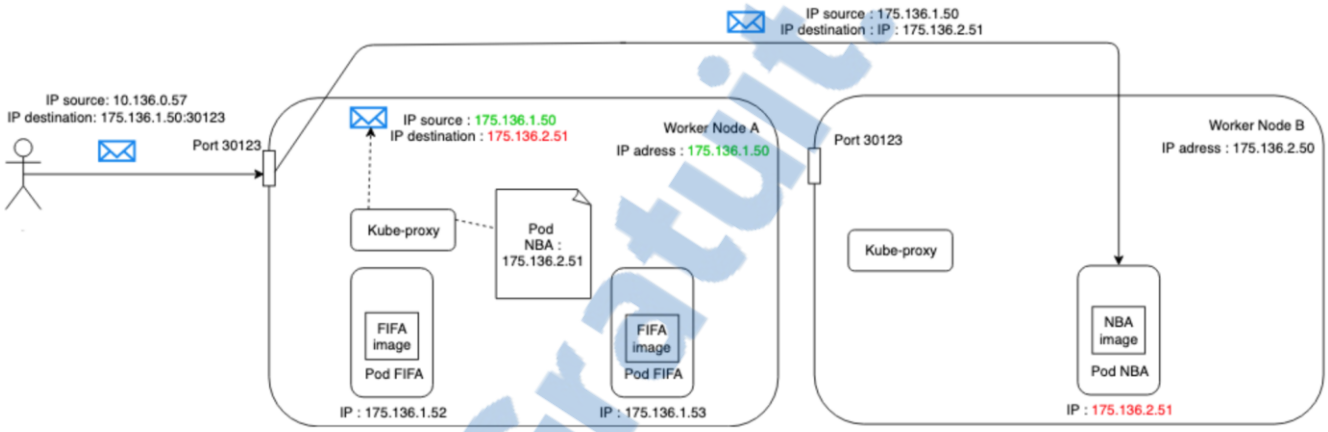
Cette architecture contient 1 master node et 2 worker nodes A et B. Le WN B contient un pod avec une image d'un service « NBA ». Ce service est capable de retourner les scores des derniers matchs de basket. Le WN A accueille deux instances de pod « FIFA » qui retournent les scores des derniers matchs de football. Nous avons choisi un accès de type NodePort pour le trafic externe : le port 30123 pour le service « NBA » et le port 32000 pour le service « FIFA ». Un HPA régule le nombre de réplicas des pods NBA vis-à-vis de la charge du trafic.

Lorsqu'un client veut atteindre un pod contenant l'image du service NBA afin d'avoir les scores, il émet une requête au cluster avec le port 30123.

4.2 Networking

Voyons maintenant en détail comment le trafic est géré :

Figure 14 : NodePort Networking



Le seul pod qui détient un container avec l'image NBA, se trouve dans le WN B (Figure 14 : NodePort Networking).

Quand la requête du navigateur web atteint le port 30123 du WN A, les iptables rules du Kube-Proxy (chapitre 3.4 Worker Node) changent l'adresse de destination par l'adresse IP du pod NBA, mais ils changent aussi l'adresse source du paquet réseau avec l'adresse IP du WN A.

De cette façon, malgré qu'une requête soit entrée dans un WN qui n'ait pas le pod contenant le service NBA, le client a tout de même pu accéder à celui présent dans le WN B. [21]

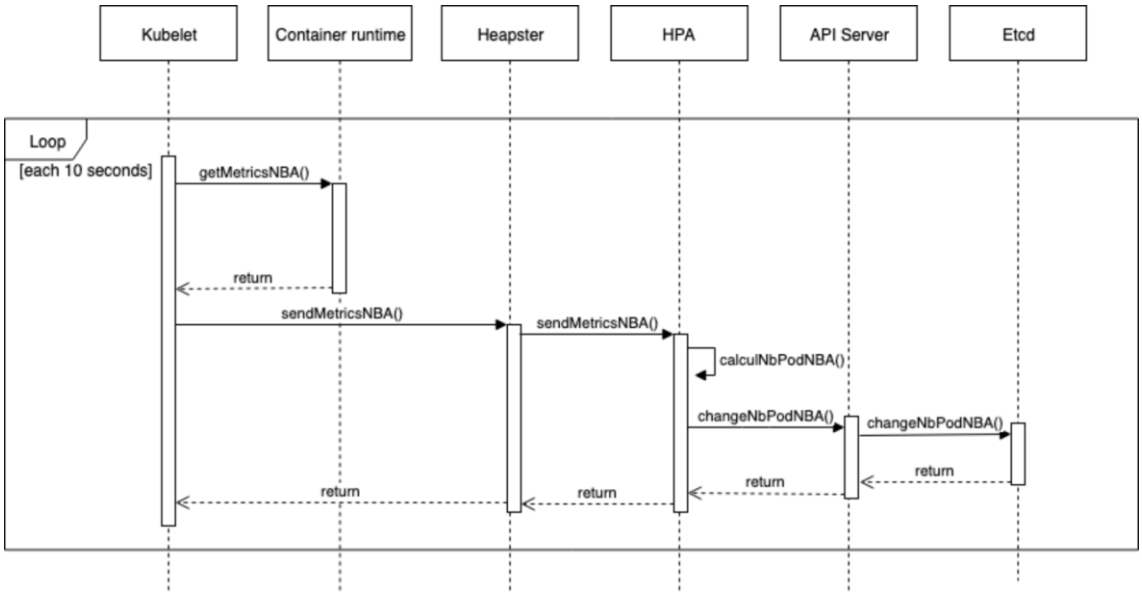
4.3 Communication entre les ressources Kubernetes pour la montée en puissance

Sur la base de notre exemple, nous allons maintenant voir en détail la mécanique de la montée en puissance en cas de surcharge d'un « Service »

4.3.1 Ajustement du nombre de pods vis-à-vis du trafic

Le composant Heapster récupère périodiquement les métriques du cAdvisor contenu dans le Kubelet du WN B. Ensuite, il informe l'HPA afin que ce dernier puisse s'assurer que la charge du trafic est correcte vis-à-vis du nombre de pods disponible (chapitre 3.6 HorizontalPodAutoscalling). Son calcul indique par exemple qu'il faudrait cinq et non pas trois instances de ce pod pour répondre à la charge actuelle. L'HPA transmet le résultat obtenu à l'API Server afin de changer l'état de l'etcd.

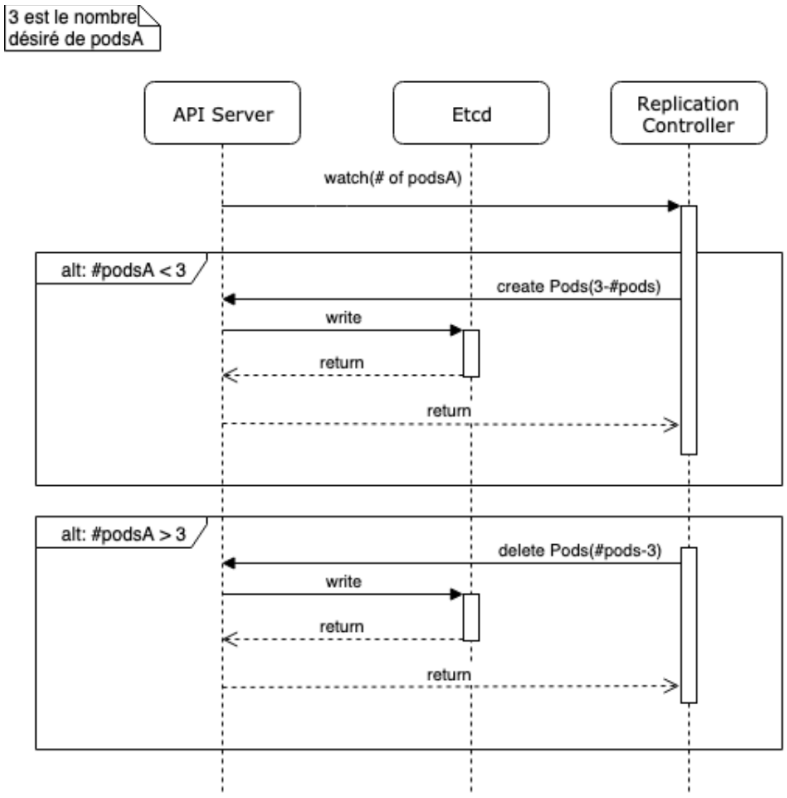
Figure 15 : Sequence diagram HPA



4.3.2 Création des pods suite à une mise-à-jour de l'etcd

Le ReplicaSet « watch » les évènements de changement du nombre de réplicas du Pod NBA émis par l'HPA. Après cela, il va générer deux instances de pods NBA puisqu'il y en a trois dans le cluster. En effet, l'etcd mentionne maintenant cinq réplicas après la mise à jour déclenchée par l'HPA. Dès que les pods sont créés, le ReplicaSet communique à l'API Server que deux pods ont été créés. Il faut maintenant les assigner à un Worker Node.

Figure 16 : Sequence diagram ReplicationController



4.3.3 Sélection du worker node

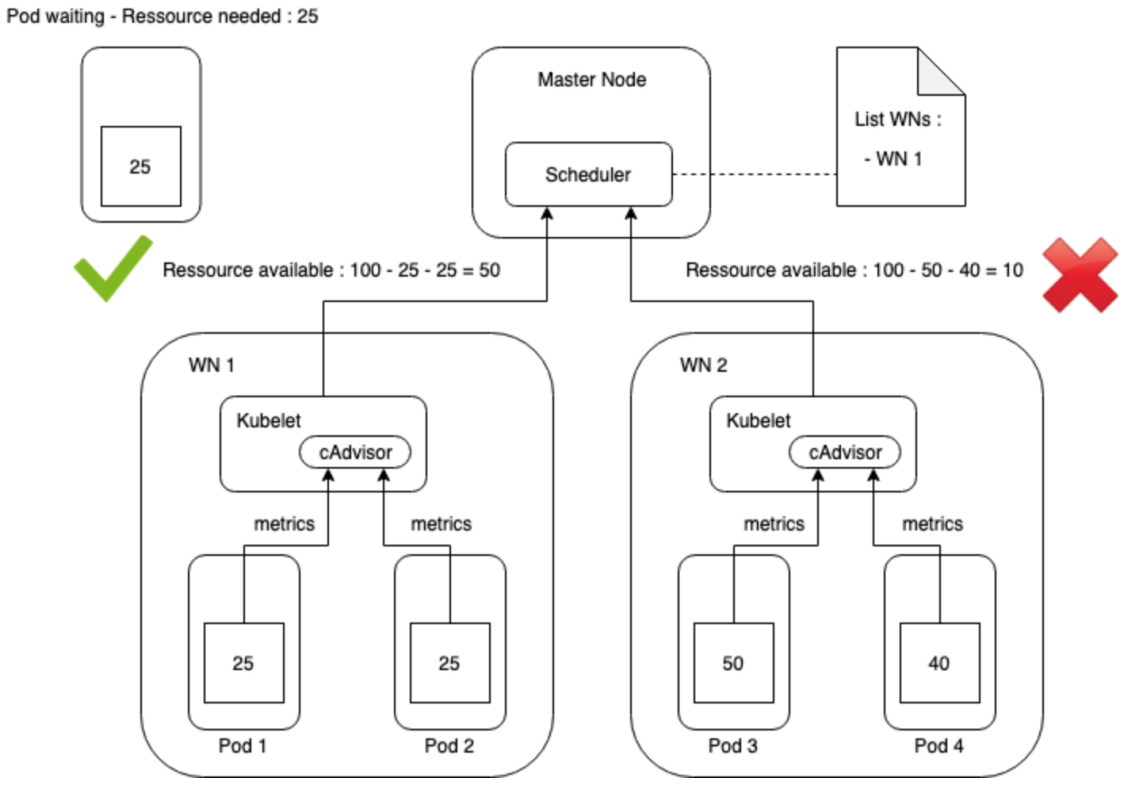
Le Scheduler, qui est abonné à l'évènement de création de pods du RC, sélectionne un WN parmi tous ceux présents dans le cluster pour accueillir les pods créés. [23] La sélection se fait en deux étapes :

1. Filtrer les WNs afin de n'avoir que ceux qui ont assez de ressources disponibles pour accueillir les pods.
2. Choisir aléatoirement dans la liste filtrée un WN. [22]

Pour réaliser l'étape 1, les cAdvisors contenus dans les Kubelets vont faire la somme des métriques obtenus des containers s'exécutant dans leur WN respectif. Ensuite, les Kubelets vont transmettre la différence entre les ressources disponibles et utilisées par les containers au Scheduler.

Dans la figure suivante admettons que nous devions assigner un pod de charge 25 (exemple de métrique récupérée par Kubelet). Chaque WN a une capacité de charge de 100. On voit que seul le WN 1 a les ressources nécessaires.

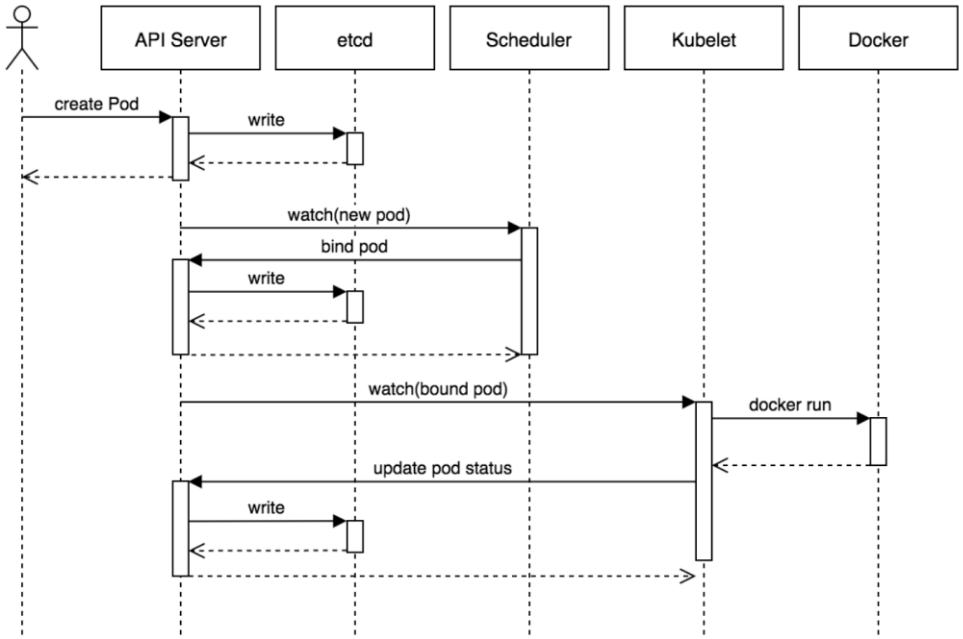
Figure 17 : Worker nodes filter



Le pod ayant été attribué, le Kubelet du WN sélectionné va à son tour recevoir l'information depuis l'API Server. Il prendra donc le descriptor du pod qui lui a été assigné pour regarder quelle image est nécessaire pour le faire fonctionner.

Ensuite, le Kubelet va communiquer l'image « NBA » via REST au Container Engine (Docker) qui se chargera de le mettre dans le container (commande « run » de docker). [24]

Figure 18 : Sequence diagram Scheduler



[24]

Cette étape est répétée pour chaque pod qui est en attente d'attribution à un WN.

Plus tard, si le traffic diminue suffisamment, l'Heapster recevra des métriques plus basses des pods NBA dispatchés dans le cluster et les communiquera à l'HPA. Les mêmes étapes s'enchaîneront et le ReplicaSet détruira des pods au lieu d'en créer.

5. Installation

5.1 Docker

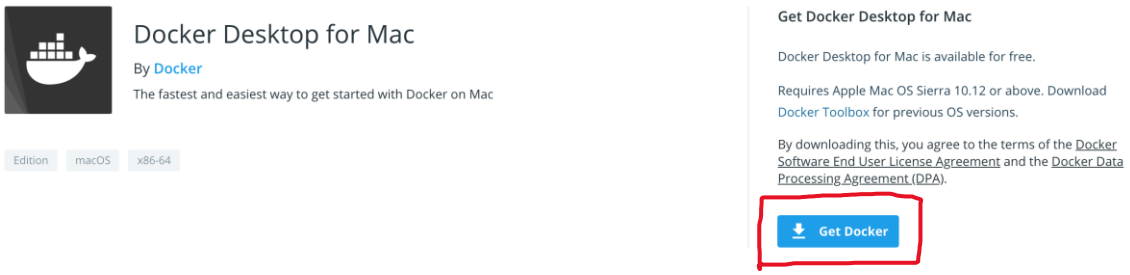
MAC OS :

Il faut se rendre sur :

https://hub.docker.com/editions/community/docker-ce-desktop-mac

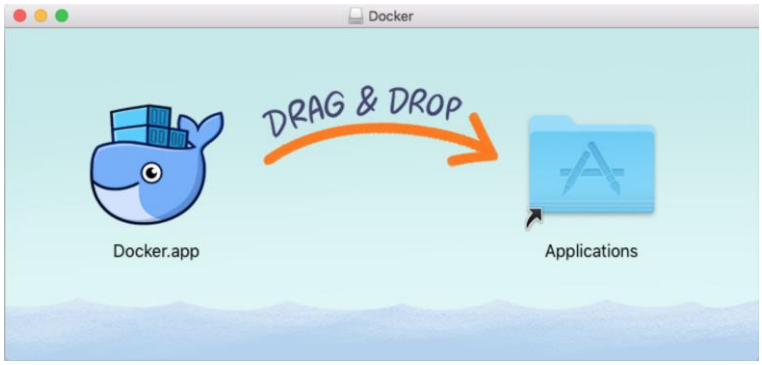
Une fois sur le site et un compte docker créé, cliquer sur le bouton « Get docker ».

Figure 19 : Téléchargement Docker MAC OS



Ensuite, double cliquer sur le fichier .dmg téléchargé et Drag & Drop l'application « Docker.app » dans le dossier « Applications ».

Figure 20 : Drag & Drop Docer MAC OS



Docker est maintenant installé, cliquer sur l'icône Docker pour mettre en marche l'application.

Si tout s'est bien passé, il est possible de voir une icône Docker dans la barre de statuts.

Figure 21 : Barre de statuts MAC OS

[25]

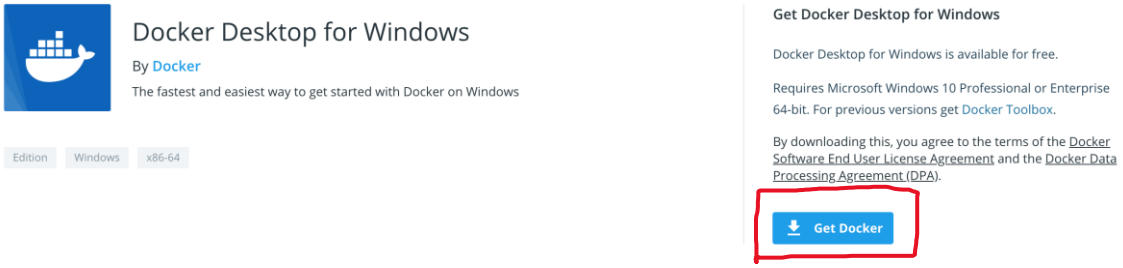
Microsoft Windows :

Il faut se rendre sur :

https://hub.docker.com/editions/community/docker-ce-desktop-windows

Une fois sur le site et votre compte docker créé, cliquez sur le bouton « Get Docker » :

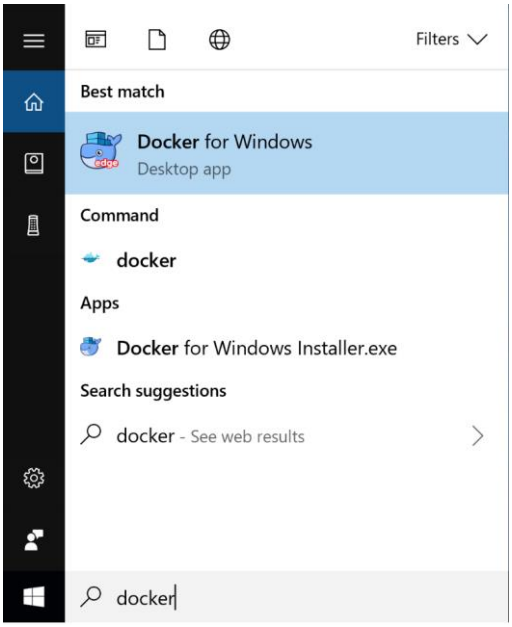
Figure 22 : Téléchargement Docker Microsoft Windows



Ensuite, double cliquer sur le fichier .exe téléchargé et suivre le guide d'installation.

Docker est maintenant installé, cherchez « Docker for Windows » dans la barre de recherche et cliquer sur l'application trouvée par la recherche.

Figure 23 : Barre de recherche Microsoft Windows



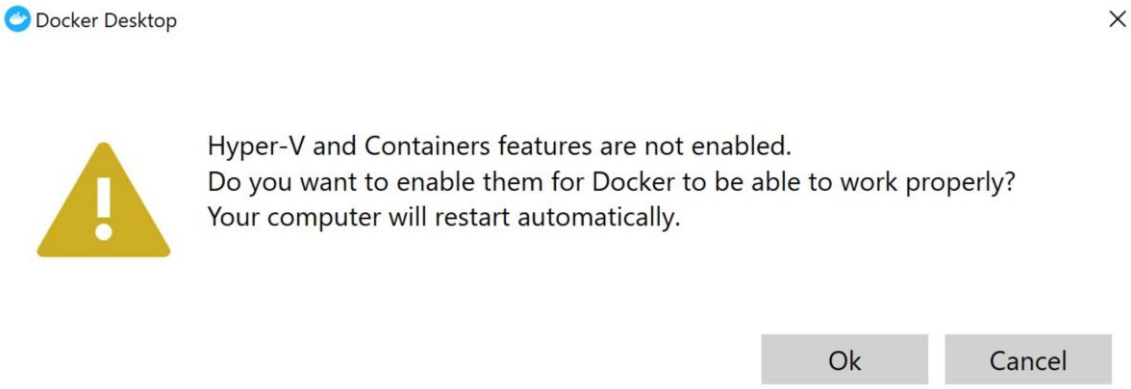
Si tout s'est bien passé, il est possible de voir une icône Docker dans la barre de statuts.

Figure 24 : Barre de status Microsoft Windows

[26]

Une fois que Docker est installé, un message peut apparaître et demander d'activer Hyper-V. Accepter la suggestion en cliquant sur « Ok ».

Figure 25 : Message Docker Hyper-V



5.2 Minikube

5.2.1 Introduction

Minikube permet de créer en local dans une machine un cluster Kubernetes. Ce n'est pas une technologie à installer pour un environnement de production mais à utiliser à des fins d'apprentissage. [27]

5.2.2 Prérequis

Il faut vérifier si la virtualisation est prise en charge par votre machine.

5.2.3 Virtualisation VT-x ou AMD-v

MAC OS :

Ouvrir un terminal et exécuter la commande :

```
sysctl -a | grep machdep.cpu.features
```

Si la sortie affiche « VMX » alors la machine prend en charge la virtualisation.

Microsoft Windows :

Ouvrir un terminal et exécuter la commande :

```
systeminfo
```

Si la sortie affiche :

```
Hyper-V Requirements:     VM Monitor Mode Extensions: Yes
                          Virtualization Enabled In Firmware: Yes
                          Second Level Address Translation: Yes
                          Data Execution Prevention Available: Yes
```

Alors la machine prend en charge la virtualisation.

5.2.4 Hyperviseur

Installer VirtualBox en cliquant sur le lien correspondant à votre système d'exploitation : https://www.virtualbox.org/wiki/Downloads

VirtualBox est un logiciel de virtualisation, il va allouer et gérer des machines virtuelles au sein de votre machine.

5.2.5 Kubectl

Kubectl est un client permettant d'exécuter des commandes dans les clusters Kubernetes.

MAC OS :

Avec le gestionnaire de paquet « Homebrew » télécharger le client Kubectl en exécutant la commande :

```
brew install kubernetes-cli
```

Microsoft Windows :

Si vous n'avez pas « curl », installer le avec la commande :

```
sudo apt-get install curl
```

Ensuite, installer Kubectl en exécutant la commande :

```
curl -LO https://storage.googleapis.com/kubernetes-
release/release/v1.15.0/bin/windows/amd64/kubectl.exe
```

Finalement, ajouter le binaire téléchargé par la commande précédente à votre PATH.

[28]

5.2.6 Minikube

MAC OS :

Exécuter la commande :

```
brew cask install minikube
```

Microsoft Windows :

Télécharger Chocolatey : https://chocolatey.org/

Quand Chocolatey est téléchargé et installé, exécuter la commande :

```
choco install minikube kubernetes-cli
```

[29]

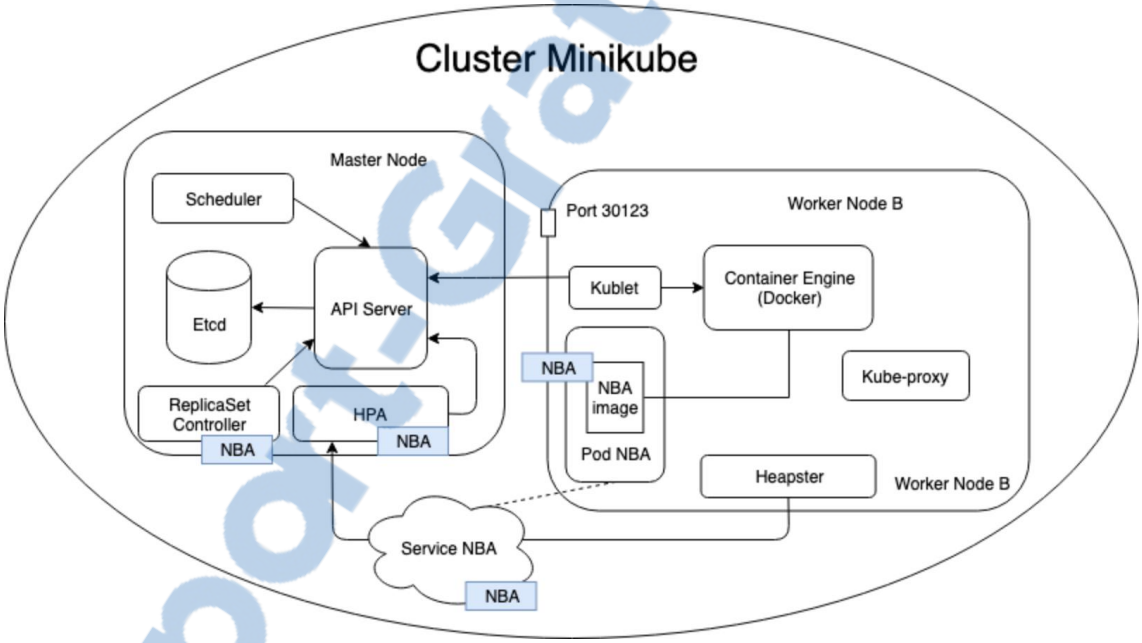
6. Case study

6.1 Introduction

Afin d'effectuer une démonstration de montée en puissance, nous allons déployer un service dans un cluster sur Minikube et accéder à ce service de multiples fois afin de le surcharger. Ensuite, nous allons observer comment Kubernetes gère cette surcharge avec les éléments que nous avons parcourus dans les chapitres précédents.

Nous allons reprendre le cluster vu dans le chapitre 4.1 et nous concentrer sur le service NBA. Cependant, Minikube ne met à disposition qu'un cluster composé uniquement d'un master node et d'un worker node :

Figure 26 : Cluster NBA avec Minikube



Les fichiers mentionnés dans les prochains chapitres seront accessibles sur ce git :

https://gitlab.com/HugoPF/kubernetestb

6.2 Création de l'image NBA avec Docker

6.2.1 Dockerfile

Premièrement, il faut s'assurer que l'application Docker est en marche grâce à l'icône de la baleine sur la barre de statuts. Deuxièmement, il faut ouvrir un terminal et naviguer sur un répertoire vide afin de réaliser la démonstration.

Télécharger tout le contenu du repository git, sauf les fichiers avec l'extension « .yaml » et un fichier nommé « Dockerfile », dans ce répertoire afin d'avoir le service NBA codé en NodeJS.

Ensuite, dans ce même répertoire, il faut créer un fichier nommé « Dockerfile » (voir 3.2.3 Dockerfile) et insérer les instructions suivantes :

Figure 27 : Dockerfile

Il faut toujours commencer un Dockerfile par une instruction « FROM ».

« FROM » signifie que l'image que nous allons créer va se construire à partir d'une autre image nommée node :8.11-alpine. Si elle n'est par présente en local, cette image va être téléchargée à partir de DockerHub.

« WORKDIR » permet d'initialiser un répertoire de travail (working directory) dans l'environnement du container où l'image sera installée. Dans notre cas, notre container aura un chemin /usr/src/app avec « app » comme working directory.

« COPY » permet de copier un fichier ou répertoire de notre machine et de l'ajouter à une destination dans le système de fichiers du container. En ligne 5, nous copions le fichier package.json dans le répertoire /usr/src/app.

« RUN » est une instruction qui démarre un terminal afin d'exécuter une commande au sein du container. « npm install » installe toutes les dépendances listées dans le fichier « package.json » [30].

La ligne 8 copie tout le répertoire où les fichiers liés au service NBA sont situés, vers le répertoire usr/src/app du container.

« CMD » est une instruction qui exécute une commande. Dans notre cas nous voulons démarrer le server nodeJS avec la commande « node app.js » [31].

[32]

6.2.2 Docker CLI

Lorsque le Dockerfile est généré, la commande docker permettant de créer une image est [38] :

```
docker build -t imageName .
```

Le « . » à la fin de la commande indique à Docker qu'il doit chercher le Dockerfile sur le répertoire courant.

Une fois la commande exécutée, docker affiche en sortie la construction de l'image et un message de succès.

Figure 28 : Construction de l'image dans le terminal

6.3 Publier l'image sur DockerHub

Avant de pouvoir publier l'image sur DockerHub, il faut taguer cette dernière avec un identifiant créé au préalable sur la plateforme [33] :

```
Docker tag imageName your_id/imageName
```

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ docker tag nba hugopf/nba
MacBook-Pro-de-Hugo:kubernetestb hugomocho$
```

Ensuite, il est possible de publier l'image [33] :

```
docker push your_id/imageName
```

Le terminal affichera :

Figure 29 : Publication de l'image dans DockerHub

```
MBP-de-Hugo:kubernetestb hugomocho$ docker push hugopf/nba
The push refers to repository [docker.io/hugopf/nba]
76be9518d1f9: Pushed
055cb56b6289: Pushed
c1ead7136883: Pushed
3902e7ce1728: Pushed
d35df9c923f8: Pushed
a7d484df787a: Pushed
8dfad2055603: Pushed
latest: digest: sha256:707c5de4ef05fa957bb5312a9bf895a9161b0bd005ea0bfca2e6da
d909f1ba0f size: 1788
```

[2]

6.4 Créer un container avec l'image NBA

Une fois l'image créée et disponible, il faut démarrer un container avec cette image en associant le port 5000 de la machine avec le port 5000 du container.

Pour cela, il existe la commande [39] :

```
docker run -d -p5000:5000 imageName
```

Figure 30 : Création du container

```
MBP-de-Hugo:kubernetestb hugomocho$ docker run -d -p5000:5000 nba
f28e16495fa69f50f3f9c7428de2fe417b8e0bf3937c9e0b8c32714191a5ef76
```

[2] Si le retour du terminal ne montre pas un message de succès, il faut se connecter à Docker avec vos identifiants avec la commande « docker login ».

Afin de vérifier que le container a bien été créé, il est possible d'exécuter la commande [40] :

```
docker container ls
```

Cette commande affiche la liste de tous les containers en marche avec des informations à leurs sujet comme : l'image qu'ils contiennent, l'adresse IP, le port sur lequel le service est disponible, etc.

Figure 31 : Liste des containers

```
MBP-de-Hugo:kubernetestb hugomocho$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS                    NAMES
9e82d5f43b18        nba                 "node app.js"       28 hours ago        Up 28 hours         0.0.0.0:5000->5000/tcp   c-nba
```

Maintenant qu'il est certain que le container fonctionne, il est possible de tester une requête depuis un navigateur web avec l'url « localhost :5000 » afin d'atteindre le container.

Figure 32 : Requête au container

[3]

[3] Sur la Figure 32 : Requête au container, la première ligne correspond à l'identifiant du container et non d'un pod pour l'instant.

6.5 Essai de montée en puissance

6.5.1 Introduction

Pour cette démonstration nous allons tout d'abord créer minikube pour disposer de tous les composants requis de Kubernetes. Ensuite nous allons configurer un Replicaset pour pouvoir disposer de plusieurs pods pour exécuter notre image NBA. Ceci nous permettra en particulier de montrer l'augmentation manuelle du nombre de pods comme présenté au paragraphe 3.5.

Ensuite, nous allons configurer un Service du type NodePort pour montrer que le système fait du load balancing entre les pods à disposition en envoyant plusieurs requêtes sur la même adresse. Afin de pouvoir effectuer cette démonstration nous avons dû configurer notre application NBA pour que l'identifiant du pod soit affiché lors de la réponse aux requêtes.

La dernière étape est de configurer le HPA pour permettre la montée en puissance en cas de surcharge de requêtes. Nous verrons ainsi que le nombre de pods assignés à notre image NBA va augmenter automatiquement.

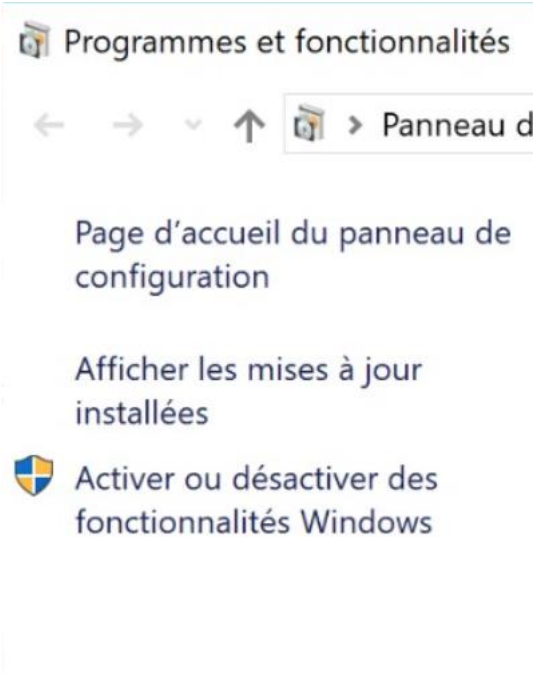
6.5.2 Création minikube

Pour lancer minikube, il faut exécuter depuis un terminal [35] :

Minikube start

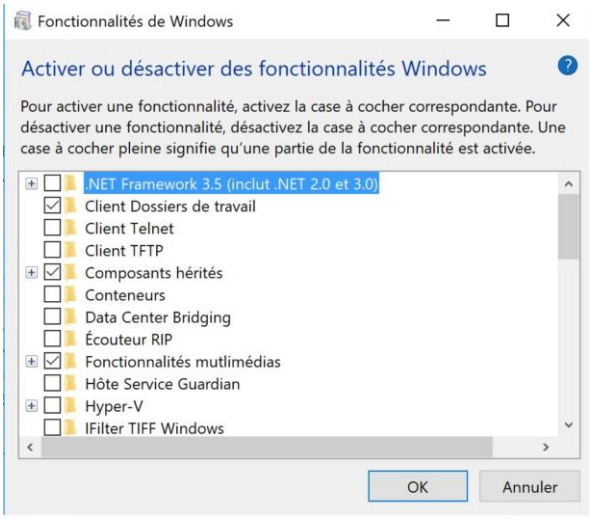
Si le système d'exploitation est Windows et qu'une erreur mentionne que l'hyper-V est actif, il faut aller sur « Programmes et fonctionnalités » et cliquer sur le lien « Activer ou désactiver des fonctionnalités Windows ».

Figure 33 : Programmes et fonctionnalités Windows



Ensuite, il faut décocher la case « Hyper-V ».

Figure 34 : Fonctionnalité Hyper-V



Après avoir exécuté cette commande avec succès, un cluster est créé et est composé :

- d'un master node avec :
    - un etcd
    - un API Server
    - un Scheduler
- et un worker node avec :
    - un Kubelet,
    - un Kube-proxy
    - un container-runtime docker

Ces composants sont mentionnés dans le message de retour du terminal.

Figure 35 : Construction du Cluster

```
MBP-de-Hugo:kubernetestb hugomocho$ minikube start
There is a newer version of minikube available (v1.2.0).  Download it here:
https://github.com/kubernetes/minikube/releases/tag/v1.2.0

To disable this notification, run the following:
minikube config set WantUpdateNotification false
😄  minikube v1.0.0 on darwin (amd64)
🤹  Downloading Kubernetes v1.14.0 images in the background ...
💡  Tip: Use 'minikube start -p <name>' to create a new cluster, or 'minikube delete' to delete this one.
🔄  Restarting existing virtualbox VM for "minikube" ...
⌛  Waiting for SSH access ...
📶  "minikube" IP address is 192.168.99.103
🐳  Configuring Docker as the container runtime ...
🐳  Version of container runtime is 18.06.2-ce
⌛  Waiting for image downloads to complete ...
✨  Preparing Kubernetes environment ...
🚜  Pulling images required by Kubernetes v1.14.0 ...
🔄  Relaunching Kubernetes v1.14.0 using kubeadm ...
⌛  Waiting for pods: apiserver proxy etcd scheduler controller dns
📯  Updating kube-proxy configuration ...
🤔  Verifying component health .....
💗  kubectl is now configured to use "minikube"
🏄  Done! Thank you for using minikube!
```

Il reste à y insérer : un ReplicaSet afin de créer et maintenir des pods contenant des containers NBA, un Service de type NodePort capable de les regrouper afin de les exposer en un seul point et finalement, un HPA capable de calculer le nombre optimal de pods pour répondre au trafic externe du cluster.

6.5.3 Création ReplicaSet

Afin de générer un ReplicaSet (3.5 ReplicationController, ReplicaSet), il est conseillé d'après la documentation de Kubernetes d'utiliser un Deployment descripteur [16] [34].

Il faut générer un fichier nommé : deployment-nba.yaml (ce nom n'est pas obligatoire, il s'agit d'une suggestion, idem pour les autres fichiers) :

Figure 36 : RS Deployment descriptor

```yaml
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: nba
5  spec:
6    replicas: 3
7    template:
8      metadata:
9        name: nba
10       labels:
11         app: nba
12     spec:
13       containers:
14       - image: hugopf/nba
15         name: nodejs
16         resources:
17           requests:
18             cpu: 10m
```

À partir de la ligne 5, il s'agit de la description du ReplicasSet. Avec cette configuration, il vérifiera que 3 réplicas d'un pod avec le label « nba » (ligne 6 et 10-11), composé d'un container avec l'image créée dans le chapitre 6.2 (ligne 13-14), seront toujours disponibles dans le cluster.

Ensuite, afin de créer le deployment [41] :

```
Kubectl create -f deployment-nba.yaml
```

Il est maintenant possible de retrouver le ReplicaSet ainsi que les pods dont il a la gestion. Pour lister tous les ReplicaSet du cluster, taper :

```
kubectl get rs
```

Figure 37 : Liste des ReplicaSets

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get rs
NAME              DESIRED   CURRENT   READY   AGE
nba-67db665d79    3         3         3       86s
```

Pour lister tous les pods du cluster, taper [42] :

```
Kubectl get pods
```

On voit dans la Figure 38 que les pods sont préfixés par le nom du ReplicaSet qui les a générés.

Figure 38 : Liste des pods

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
nba-67db665d79-5ldsk   1/1     Running   0          3m2s
nba-67db665d79-j9rhs   1/1     Running   0          3m2s
nba-67db665d79-khmkz   1/1     Running   0          3m2s
```

6.5.4 Diminution manuelle du nombre de pods « nba »

Dans le chapitre « ReplicationController, ReplicaSet », il est mentionné qu'il est possible de modifier le nombre de réplicas dans le descriptor d'un ReplicaSet afin d'augmenter ou diminuer la quantité des pods dans le cluster. Pour éviter de modifier le fichier et de créer à nouveau l'objet au sein du système, il existe la commande [43] :

```
kubectl scale deployment nba –-replicas=2
```

Après avoir exécuté cette dernière commande, il y a toujours 3 pods dans le cluster, mais un des statuts est passé à « Terminating ».

Figure 39 : Diminution des pods "nba" .1

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl scale deployment nba --replicas=2
deployment.extensions/nba scaled
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS        RESTARTS   AGE
nba-67db665d79-5ldsk   1/1     Running       0          17m
nba-67db665d79-j9rhs   1/1     Terminating   0          17m
nba-67db665d79-khmkz   1/1     Running       0          17m
```

Après quelques instants, il ne sera plus « prêt » à l'utilisation. Puis finalement, il sera supprimé du cluster.

Figure 40 : Diminution des pods "nba" .2

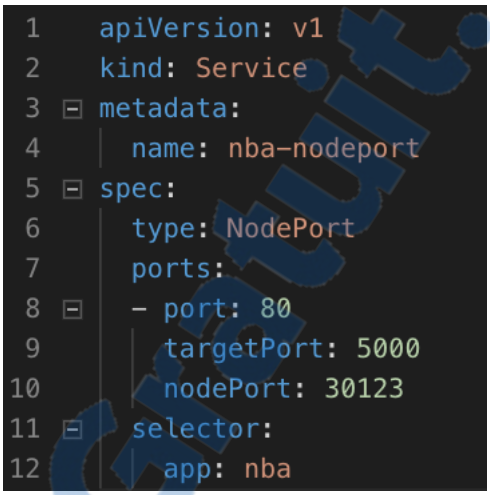
```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS        RESTARTS   AGE
nba-67db665d79-5ldsk   1/1     Running       0          18m
nba-67db665d79-j9rhs   0/1     Terminating   0          18m
nba-67db665d79-khmkz   1/1     Running       0          18m
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
nba-67db665d79-5ldsk   1/1     Running   0          18m
nba-67db665d79-khmkz   1/1     Running   0          18m
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
```

6.5.5 NodePort Service

Afin de rendre accessible les pods « NBA » via une adresse unique, il sera utilisé dans ce case study un Service de type Nodeport (voir chapitre 3.8.1 NodePort :).

Tout d'abord, il faut créer un descriptor de « Service » pour configurer le NodePort appelé « service-nodeport-nba.yaml »

Figure 41 : Service descriptor



Le Service de type NodePort va regrouper les pods ayant le label « NBA » (ligne 12). Il va cibler les ports 5000, port choisi au moment du lancement du container (voir chapitre 6.4), de tous les containers contenus au sein des pods qu'il aura regroupés afin de les rendre accessibles aux clients externes via le port 30123 du WN (ligne 9 et 10).

Pour lancer la création du NodePort via le descripteur, exécuter la commande [41] :

```
Kubectl create -f service-nodeport-nba.yaml
```

Afin de lister tous les services du cluster il faut entrer dans le terminal [42] :

```
Kubectl get svc
```

Figure 42 : Liste des Services

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get svc
NAME           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes     ClusterIP   10.96.0.1        <none>        443/TCP        16d
nba-nodeport   NodePort    10.103.201.180   <none>        80:30123/TCP   2d3h
```

Nous allons maintenant faire une requête sur l'adresse du worker node afin d'obtenir les scores de la ligue NBA. Premièrement, il faut connaître l'adresse IP du cluster (donc du worker node) grâce à la commande [44] :

Kubectl cluster-info

Figure 43 : Informations du cluster

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.103:8443
KubeDNS is running at https://192.168.99.103:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

Deuxièmement, reprendre l'adresse IP obtenue à la Figure 43 : Informations du cluster, utiliser le port 30123 et faire une requête avec un navigateur web.

On obtient bien le même résultat que lorsque l'image « NBA » avait été déployée sur docker. Cependant, l'identifiant (première ligne) est cette fois-ci le nom d'un pod de notre cluster. Noter que la requête a été faite sur le port 30123 qui a été déclaré dans le descripteur du NodePort.

Figure 44 : Requête aux pods "nba"

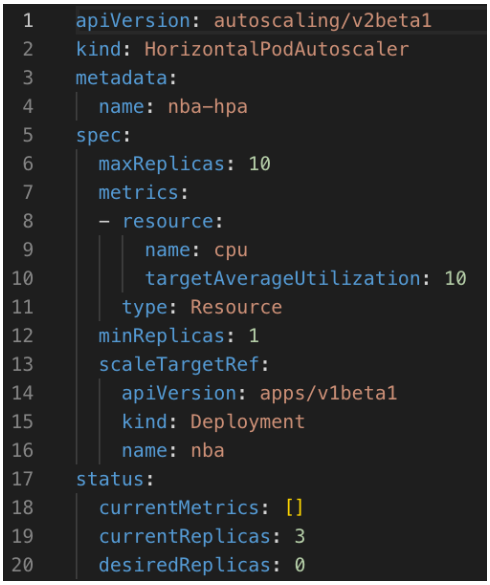
Pour vérifier que l'accès de type NodePort remplit son rôle de « loadbalancer », il suffit de rafraichir la page plusieurs fois sur le navigateur et d'observer que parfois l'identifiant du pod change.

Figure 45 : Démonstration de "LoadBalancing"

6.5.6 HPA

Afin d'augmenter automatiquement le nombre de pods selon le trafic, il faut générer un HPA. La Figure 46 présente le descripteur correspondant « nba-hpa.yaml » :

Figure 46 : HPA Descriptor



```
1  apiVersion: autoscaling/v2beta1
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: nba-hpa
5  spec:
6    maxReplicas: 10
7    metrics:
8    - resource:
9        name: cpu
10       targetAverageUtilization: 10
11     type: Resource
12   minReplicas: 1
13   scaleTargetRef:
14     apiVersion: apps/v1beta1
15     kind: Deployment
16     name: nba
17 status:
18   currentMetrics: []
19   currentReplicas: 3
20   desiredReplicas: 0
```

Une limite à 1 pod minimum et 10 maximum est fixée (ligne 6 et 12). De la ligne 13 à 14 il s'agit de l'objet sur lesquels les métriques sont basées afin de le mettre à jour. Le type de métriques à analyser est mentionné entre les lignes 7 et 11.

A l'aide de ce descripteur nous pouvons maintenant créer un HPA avec la commande [41] :

Kubectl create- f nba-hpa.yaml

La figure 47 présente le résultat de cette commande. On remarque que le système demande que le champ « conditions » soit renseigné. Comme il n'est pas utile pour notre démonstration, on peut l'ignorer.

Figure 47 : Erreur terminal HPA

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl create -f nba-hpa-v0.yaml
error: error validating "nba-hpa-v0.yaml": error validating data: ValidationError(Horizonta
lPodAutoscaler.status): missing required field "conditions" in io.k8s.api.autoscaling.v2bet
a1.HorizontalPodAutoscalerStatus; if you choose to ignore these errors, turn validation off
 with --validate=false
```

Pour que le système exécute la commande sans être bloqué, on demande d'ignorer les validations en ajoutant un paramètre dans la commande comme présenté ci-après :

Kubectl create -f nba-hpa.yaml --validate=false

Figure 48 : Création HPA

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl create -f nba-hpa-v0.yaml --validate=fa
lse
horizontalpodautoscaler.autoscaling/nba-hpa created
```

Cette fois-ci tout a bien fonctionné et le HPA est configuré. On peut le vérifier avec la commande [42] :

Kubectl get hpa

Figure 49 : Liste HPA

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get hpa
NAME      REFERENCE         TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
nba-hpa   Deployment/nba    <unknown>/10%    1         10        2          3m2s
```

Le système indique que la valeur de la métrique cible (CPU avec une valeur à 10% déclarée dans les lignes 7 à 11 de la Figure 46) est inconnue. En effet, nous n'avons pas encore déployé le Heapster et configuré la transmission des métriques à l'HPA. Voici les commandes permettant de le faire :

Pour le premier cas de figure, il faut entrer les commandes [35] :

Minikube addons enable heapster
Minikube addons enable metrics-server

Afin de vérifier qu'il est bien activé, il faut exécuter la commande [35] :

Minikube addons list

Figure 50 : Minikube addons

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ minikube addons list
There is a newer version of minikube available (v1.3.1).  Download it here:
https://github.com/kubernetes/minikube/releases/tag/v1.3.1

To disable this notification, run the following:
minikube config set WantUpdateNotification false
- addon-manager: enabled
- dashboard: disabled
- default-storageclass: enabled
- efk: disabled
- freshpod: disabled
- gvisor: disabled
- heapster: enabled
- ingress: disabled
- logviewer: disabled
- metrics-server: enabled
- nvidia-driver-installer: disabled
- nvidia-gpu-device-plugin: disabled
- registry: disabled
- registry-creds: disabled
- storage-provisioner: enabled
- storage-provisioner-gluster: disabled
```

Ensuite on peut réémettre la requête [42] :

Kubectl get hpa

Si la valeur de la métrique cible est toujours manquante, cela veut dire qu'elles n'ont pas encore pu être récoltées car cela prend quelques secondes. Après quelques essais de cette même commande on obtient le résultat présenté à la Figure 50.

Figure 51 : HPA après trafic sur le cluster

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get hpa
NAME      REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nba-hpa   Deployment/nba    60%/10%   1         10        10         11m
```

On constate qu'il y a un pourcentage de consommation du CPU plus élevé que la cible et que les réplicas sont montés à 10 qui est le maximum mentionné dans le descriptor de l'HPA. Sept nouveaux pods ont été créés presque simultanément dû à une montée en charge subite.

La surcharge a été réalisée en exécutant des requêtes répétitives au NodePort.

Figure 52 : Augmentation des pods nba avec HPA

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
nba-67db665d79-49kml   1/1     Running   0          114s
nba-67db665d79-b5p2m   1/1     Running   0          99s
nba-67db665d79-bmn8d   1/1     Running   0          99s
nba-67db665d79-dxwd8   1/1     Running   1          12m
nba-67db665d79-f886b   1/1     Running   0          99s
nba-67db665d79-fmmzz   1/1     Running   0          114s
nba-67db665d79-gfdcm   1/1     Running   1          12m
nba-67db665d79-m8wsf   1/1     Running   0          99s
nba-67db665d79-s2s2c   1/1     Running   1          12m
nba-67db665d79-xhrfc   1/1     Running   0          114s
```

On constate qu'en émettant une nouvelle requête, un des pods de la liste, différent des deux pods des exemples précédents, est choisi pour exécuter la requête :

Figure 53 : Disponibilité des nouveaux pods nba

Après un certain temps, quand le trafic est à la baisse, le surplus de pods va être supprimé.

Figure 54 : Diminution progressive des pods NBA

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS        RESTARTS   AGE
nba-67db665d79-49kml   1/1     Running       0          7m23s
nba-67db665d79-b5p2m   0/1     Terminating   0          7m8s
nba-67db665d79-bmn8d   0/1     Terminating   0          7m8s
nba-67db665d79-dxwd8   1/1     Running       1          18m
nba-67db665d79-f886b   0/1     Terminating   0          7m8s
nba-67db665d79-fmmzz   0/1     Terminating   0          7m23s
nba-67db665d79-gfdcm   1/1     Running       1          18m
nba-67db665d79-s2s2c   1/1     Running       1          18m
nba-67db665d79-xhrfc   1/1     Running       0          7m23s
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
nba-67db665d79-49kml   1/1     Running   0          7m30s
nba-67db665d79-dxwd8   1/1     Running   1          18m
nba-67db665d79-gfdcm   1/1     Running   1          18m
nba-67db665d79-s2s2c   1/1     Running   1          18m
nba-67db665d79-xhrfc   1/1     Running   0          7m30s
MacBook-Pro-de-Hugo:kubernetestb hugomocho$
```

Quand le trafic est pratiquement nul, l'HPA applique le nombre minimum mentionné dans son descriptor et le ReplicaSet détruit le surplus.

Figure 55 : Suppression des derniers pods nba

```
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS        RESTARTS   AGE
nba-67db665d79-49kml   1/1     Terminating   0          10m
nba-67db665d79-dxwd8   1/1     Running       1          21m
nba-67db665d79-gfdcm   1/1     Terminating   1          21m
nba-67db665d79-s2s2c   1/1     Terminating   1          21m
nba-67db665d79-xhrfc   1/1     Terminating   0          10m
MacBook-Pro-de-Hugo:kubernetestb hugomocho$ kubectl get pods
NAME                   READY   STATUS    RESTARTS   AGE
nba-67db665d79-dxwd8   1/1     Running   1          22m
```

7. Conclusion

En conclusion, la solution proposée par Kubernetes répond parfaitement à la problématique de manière simple et efficace. On constate que l'accès de type Nodeport offre une solution simple de loadbalancer. De plus, en cas de forte fluctuation du trafic, le système s'ajuste pour répondre au besoin automatiquement grâce à l'HPA. De ce fait, les développeurs peuvent se concentrer sur les aspects métiers de leurs applications.

La phase d'étude de cette technologie était assez complexe car cette technologie est en constante évolution et m'était inconnue. J'ai dû passer par une étude de l'architecture de Kubernetes : ses composants et leurs interactions, afin de comprendre le fonctionnement du système global.

Cependant, la partie pratique de ce manuscrit est académique, le déploiement de services plus complexes sur des machines de production nécessiterait un approfondissement car la technologie est plus vaste que l'échantillon qui a été décrit.

Heureusement, Kubernetes possède une grande communauté d'utilisateurs, très active et une excellente documentation. Si le temps nécessaire à maitriser Kubernetes semble long, plusieurs alternatives payantes existent comme : Azure Kubernetes Service [36], Amazon EKS [37], etc. qui facilitent son usage.

Pour conclure, je suis très satisfait par les connaissances et compétences acquises lors de la rédaction de ce mémoire. J'espère pouvoir les approfondir lors de mon arrivée chez mon prochain employeur qui s'avère utiliser Kubernetes.

Bibliographie

[1] GENE, Kim, HUMBLE, Jez, DEBOIS,Patrick, WILLIS, John, 2015. *The DevOps Handbook*. IT Revolution Press. ISBN 1942788002.

[2] NEWMAN, Sam, 2015. *Building Microservices designing fined-grained systems*. O'Reilly Media. ISBN 1491950358.

[3] https://www.docker.com/why-docker, consulté le 20 juin 2019

[4] NICKOLOFF, Jeff, 2016. *Docker In Action*. Manning Publications. ISBN13 9781633430235

[5] https://www.docker.com/resources/what-container, consulté le 20 juin 2019

[6] https://kubernetes.io/, consulté le 26 juin 2019

[7] https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs, consulté le 15 juillet 2019

[8] DUGERDIL, Philippe, 2019. Cours à la Haute Ecole de Gestion.

[9] https://docs.docker.com/engine/docker-overview/, consulté le 28 juin 2019

[10] https://www.lebigdata.fr/docker-definition, consulté le 2 juillet 2019

[11] https://docs.docker.com/search/?q=dockerfile, consulté le 2 juillet 2019

[12] https://hub.docker.com/, consulté le 5 juillet 2019

[13] LUKSA, Marko, 2017. *Kubernetes in Action*. Manning Publications. ISBN 1617293725

[14] https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/, consulté le 15 juillet 2019

[15] https://kubernetes.io/docs/concepts/architecture/nodes/, consulté le 15 juillet 2019

[16] https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/, consulté le 20 juillet 2019

[17] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, consulté le 20 juillet 2019

[18] https://kubernetes.io/docs/tutorials/kubernetes-basics/create-cluster/cluster-intro/, consulté le 25 juillet 2019

[19] https://kubernetes.io/docs/concepts/services-networking/service/, consulté le 28 juillet 2019

[20] https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0, consulté le 28 juillet 2019

[21] Selon discussion privée avec LUKSA Marko, Software Engineer chez Red Hat et auteur du livre *Kubernetes in Action*.

[22] https://searchitoperations.techtarget.com/definition/Kubernetes-scheduler, consulté le 2 août 2019

[23] https://kubernetes.io/docs/concepts/scheduling/kube-scheduler/, consulté le 2 août 2019

[24] https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca, consulté le 2 août 2019

[25] https://docs.docker.com/docker-for-mac/install/, consulté le 4 août 2019

[26] https://docs.docker.com/docker-for-windows/install/, consulté le 4 août 2019

[27] https://medium.com/@eric.duquesnoy/kubernetes-lancer-un-cluster-mono-noeud-7ebace9f2a1a, consulté le 6 août 2019

[28] https://kubernetes.io/fr/docs/tasks/tools/install-kubectl/, consulté le 6 août 2019

[29] https://kubernetes.io/docs/tasks/tools/install-minikube/, consulté le 6 août 2019

[30] https://docs.npmjs.com/cli/install, consulté le 7 août 2019

[31] https//nodejs.dev/run-nodejs-scripts-from-the-command-line, consulté le 8 août 2019

[32] https://docs.docker.com/engine/reference/builder/, consulté le 8 août 2019

[33] https://docs.docker.com/v17.12/docker-cloud/builds/push-images/, consulté le 10 août 2019

[34] https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/, consulté le 10 août 2019

[35] https://kubernetes.io/fr/docs/tutorials/hello-minikube/, consulté le 13 août 2019

[36] https://azure.microsoft.com/fr-fr/free/kubernetes-service/search/?&OCID=AID2000121_SEM_gT3g3igv&MarinID=gT3g3igv_324571936554_%2Bkubernetes_b_c__67171911241_aud-395027706889:kwd-88228236663&lnkd=Google_Azure_Nonbrand&dclid=CN6d8tbDheQCFZY54AodLpgOoQ, consulté le 17 août 2019

[37] https://aws.amazon.com/fr/eks/, consulté le 17 août 2019

[38] https://docs.docker.com/engine/reference/commandline/build/, consulté le 20 août 2019

[39] https://docs.docker.com/engine/reference/run/, consulté le 20 août 2019

[40] https://docs.docker.com/engine/reference/commandline/container_ls/, consulté le 20 août 2019

[41] https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#create, consulté le 20 août 2019

[42] https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get, consulté le 20 août 2019

[43] https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#scale, consulté le 20 août 2019

[44] https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#cluster-info, consulté le 20 août 2019