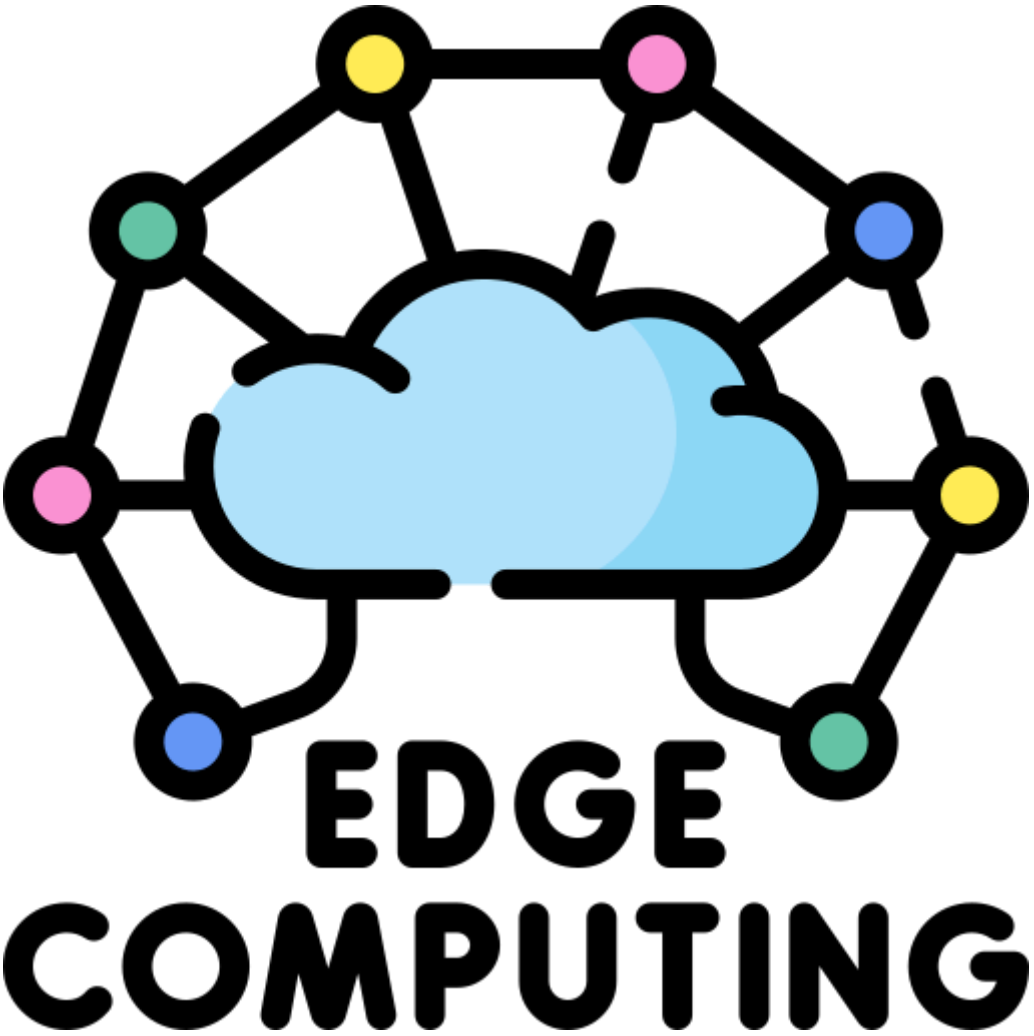
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

INNOVATIVE SMART SYSTEMS

Rapport de TP :
Cloud & Edge Computing



Florent MIRANVILLE
Tom LASALLE

1. Introduction aux hyperviseurs cloud

1.1. Introduction

Ce TP a pour but d'approfondir les connaissances sur les notions et les technologies utilisées pour les techniques de virtualisation, et ainsi mieux comprendre le fonctionnement et la structure d'un cloud numérique, notamment pour l'approvisionnement de nouveaux systèmes logiciels en prenant en compte les potentielles contraintes telles que la qualité de service et la sécurité, dans un environnement généralement dynamique et distribué.

Nous allons dans cette première partie de compte-rendu comparer les deux principaux types d'hôtes de virtualisation à savoir les VM (machines virtuelles) et les CT (les conteneurs) pour souligner leurs similarités et différences, et également étudier la mise en oeuvre des configurations réseau (NAT ou Bridge) pour permettre les communications des hôtes à travers Internet. Ensuite nous utiliserons la plateforme OpenStack et verrons comment créer et gérer un environnement Cloud pour enfin déployer notre propre topologie réseau.

1.2. Partie théorique

1.2.1. Similitudes et différences entre les principaux hôtes de virtualisation (VM et CT)

La comparaison se base sur 5 contraintes :

- Coût de virtualisation (en terme de mémoire et de CPU)
- Utilisation mémoire / CPU / réseau pour une application donnée
- Sécurité de l'application
- Performance en terme de response time
- Outil d'intégration continue (CI/CD)

Il y a également une différence entre les besoins d'un développeur d'application et d'un administrateur d'infrastructure, nous allons donc faire la comparaison selon ces deux points de vue.

Pour un développeur :

	VM	CT
Coût de virtualisation	Comme chaque machine virtuelle possède un système d'exploitation entier, et que l'architecture d'une VM possède un hyperviseur, le coût de virtualisation est élevé et le démarrage est plus lent .	Comme les conteneurs partagent le noyau du système hôte, et qu'il n'y pas d'hyperviseur, le coût de virtualisation est moindre, le démarrage est plus léger et rapide.
Utilisation ressources	La présence d'un hyperviseur assure l'allocation correcte des ressources. Par contre, chaque VM exécute son propre OS et donc les librairies associées donc plus de ressources sont utilisées.	Les ressources sont partagées entre les applications, cela permet une utilisation plus optimale car pas besoin d'exécuter plusieurs fois les services systèmes.
Sécurité	L'isolation des VM assure un accès à la mémoire et aux ressources.	L'isolation est plus faible, il y a une moins bonne sécurité car l'accès aux données est en concurrence directe avec l'OS de la machine, les librairies sont partagées donc problème sur une librairie affecte toutes les applications qui les utilisent, une faille du noyau affecte tous les conteneurs.
Performances	La présence de l'hyperviseur et du système d'exploitation pour une VM rend les performances moindres.	Il y a de meilleures performances que pour la VM car les CT ont directement accès au noyau et il n'y a pas d'hyperviseur.
Outil d'intégration continue	Pas pratique pour de l'intégration continue car les images à cloner sont grandes, les déploiements en sont donc plus lourds.	Les CT étant plus légères, plus simples et rapides à créer, ils sont adaptés à l'intégration continue.

Pour un administrateur :

	VM	CT
Coût de virtualisation	La présence de l'hyperviseur augmente le coût de virtualisation et chaque VM doit allouer ses ressources et son propre OS.	Pas d'hyperviseur donc coût moindre, et comme les conteneurs partagent le même OS.
Utilisation ressources	L'avantage est qu'on est certain d'avoir accès à la quantité allouée de ressources. Cependant, il est possible d'allouer trop de ressources que nécessaire et donc de rendre l'utilisation moins efficace.	Les conteneurs offrent un démarrage rapide et une faible consommation, et le partage de l'OS offre une meilleure utilisation des ressources.
Sécurité	L'isolation forte des VM assure que s'il y a un problème sur l'OS de l'hôte, les VM ne seront pas affectées.	Malgré l'isolation, comme les conteneurs utilisent directement une partie de l'OS de l'hôte, un problème sur celui-ci affecte tous les conteneurs.
Performances	Les performances allouées lors de la création de la VM sont uniformes.	Les performances sont proches de celles de l'hôte natif.
Outil d'intégration continue	Malgré les difficultés matérielles, les VM peuvent être efficaces pour l'intégration continue d'une application composée de différents services ayant besoin d'un développement spécifique. On peut alors déployer un CI/CD pipeline approprié sur chaque VM.	idéal pour les microservices, l'intégration continue et déploiement continue CI/CD.

Il existe plusieurs types de conteneurs sur le marché, comme par exemple LXC/LXD, Docker, Rocket ou encore OpenVZ. Il n'est pas simple de vérifier laquelle de ces technologies est la meilleure, nous allons donc comparer certaines d'entre elles, à savoir Linux LXC et Docker, selon quelques critères :

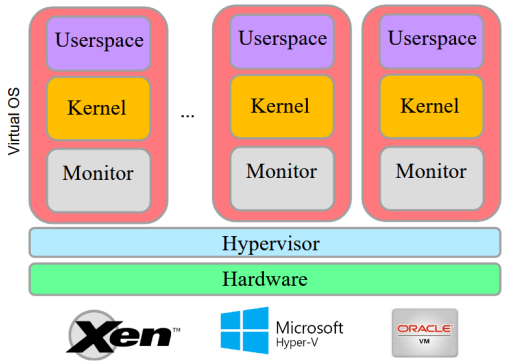
- Isolation au niveau applicatif et ressources (Quels sont les mécanismes utilisés pour l'isolation et la gestion des ressources)
- Niveau de containerisation (Partage du même noyau ou via des images applicatives ou encore un hyperviseur)
- Outils utilisés (API, Intégration CI/CD..)

	Linux LXC	Docker
C1	L'isolation des containers LXC se fait à l'aide de technologies telles que les Control groups (cgroups) utilisant les systemd permettant l'initialisation des espaces utilisateurs et la gestions de leurs processus Chaque conteneur LXC possède son propre système de fichiers, son propre réseau et son propre espace de processus isolé.	Par défaut, Docker va isoler des applications plutôt qu'un système entier comme Linux LXC. Docker exploite également les namespaces et les cgroups pour offrir un environnement et un réseau isolé.
C2	OS-level : LXC se concentre sur la création de conteneurs qui sont essentiellement des machines virtuelles légères. Chaque conteneur peut exécuter une distribution Linux complète avec son propre système de fichiers, sa propre configuration réseau	Application-level : Docker se concentre sur des conteneurs ciblés pour une application ou un service en particulier.
C3	Possède une API C stable. Possède également une option CICD LXC afin de configurer la CI/CD pour tous les types de serveurs (dev, test , staging, prod)	Ensemble d'outils pour la gestion d'images, de registre et d'intégration avec des pipelines CI/CD

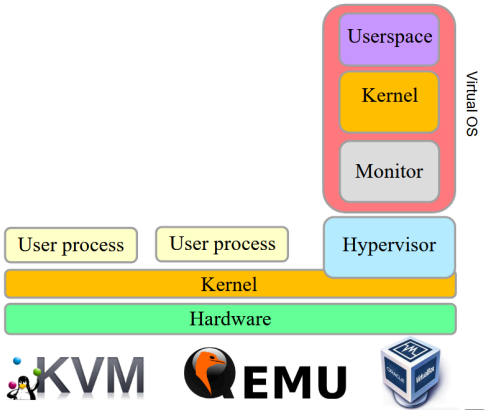
Docker est idéal pour les architectures à microservices, où les applications sont décomposées en éléments plus petits et indépendants (microservices) pouvant être déployés et gérés de manière dynamique.

1.2.2. Similitudes et différences entre les architectures d'hyperviseur de type 1 et de type 2

L'hyperviseur de type 1 fonctionne directement sur le matériel physique, sans nécessiter de système d'exploitation hôte, et bénéficie ainsi d'un accès direct aux ressources matérielles. C'est pourquoi il est également appelé *hyperviseur bare-metal* (ou "hyperviseur matériel nu").



À l'inverse, l'hyperviseur de type 2 est une application qui s'exécute sur un système d'exploitation hôte. Il est souvent désigné sous les termes *hyperviseur hébergé* ou *hyperviseur embarqué*.

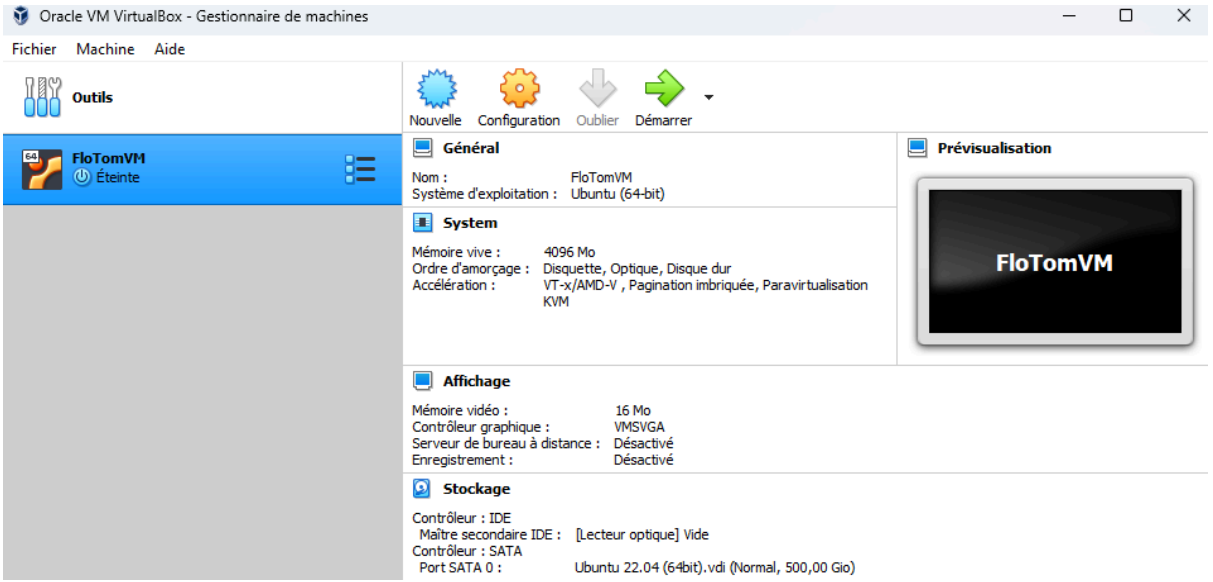


Dans le cas de VirtualBox, il s'agit d'un hyperviseur de type 2. Celui-ci s'exécute au-dessus d'un système d'exploitation hôte, tel que Windows, Linux ou macOS, et est donc qualifié d'hyperviseur hébergé.
Dans le cas d'OpenStack, il s'agit d'une plateforme de cloud computing qui orchestre un /plusieurs hyperviseurs de type 1 pour gérer et déployer des machines virtuelles à grande échelle.

1.3. Partie pratique

1.3.1. Créer et configurer une machine virtuelle

La première étape de cette partie pratique consiste à installer une machine virtuelle sur VirtualBox en mode NAT, puis à configurer un réseau permettant une communication bidirectionnelle avec cette machine. Cette VM sera basée sur un système d'exploitation Ubuntu, avec 4 Go de RAM pour garantir un confort d'utilisation et un processeur monocœur (single-core).

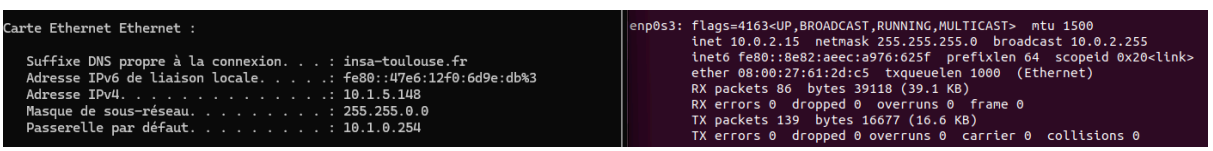


Après le lancement de la VM, on s'authentifie avec les identifiants :

- username : **osboxes**
- password: **osboxes.org**

1.3.2. Test de la connectivité de la machine virtuelle

Et on vérifie les paramètres réseaux à l'aide des commandes ipconfig et ifconfig respectivement sur la machine hôte sous Windows et la VM sous Linux.



On observe que les deux adresses sont dans des réseaux différents. On veut tester la connectivité dans différents scénarios :

VM vers l'Internet, en testant un ping vers l'adresse 8.8.8.8 :

```
osboxes@osboxes:~/Desktop$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
64 bytes from 8.8.8.8: icmp_seq=1 ttl=113 time=7.09 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=113 time=6.84 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=113 time=6.72 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=113 time=6.94 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=113 time=6.86 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=113 time=6.63 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=113 time=6.77 ms
```

La machine virtuelle étant connectée à Internet, elle peut donc envoyer des données vers des adresses publiques.

L'hôte vers la VM :

```
C:\Users\miranvil>ping 10.0.2.15

Envoi d'une requête 'Ping'  10.0.2.15 avec 32 octets de données :
Réponse de 195.83.11.254 : Durée de vie TTL expirée lors du transit.
Réponse de 195.83.11.254 : Durée de vie TTL expirée lors du transit.
Réponse de 195.83.11.254 : Durée de vie TTL expirée lors du transit.

Statistiques Ping pour 10.0.2.15:
    Paquets : envoyés = 3, reçus = 3, perdus = 0 (perte 0%),
```

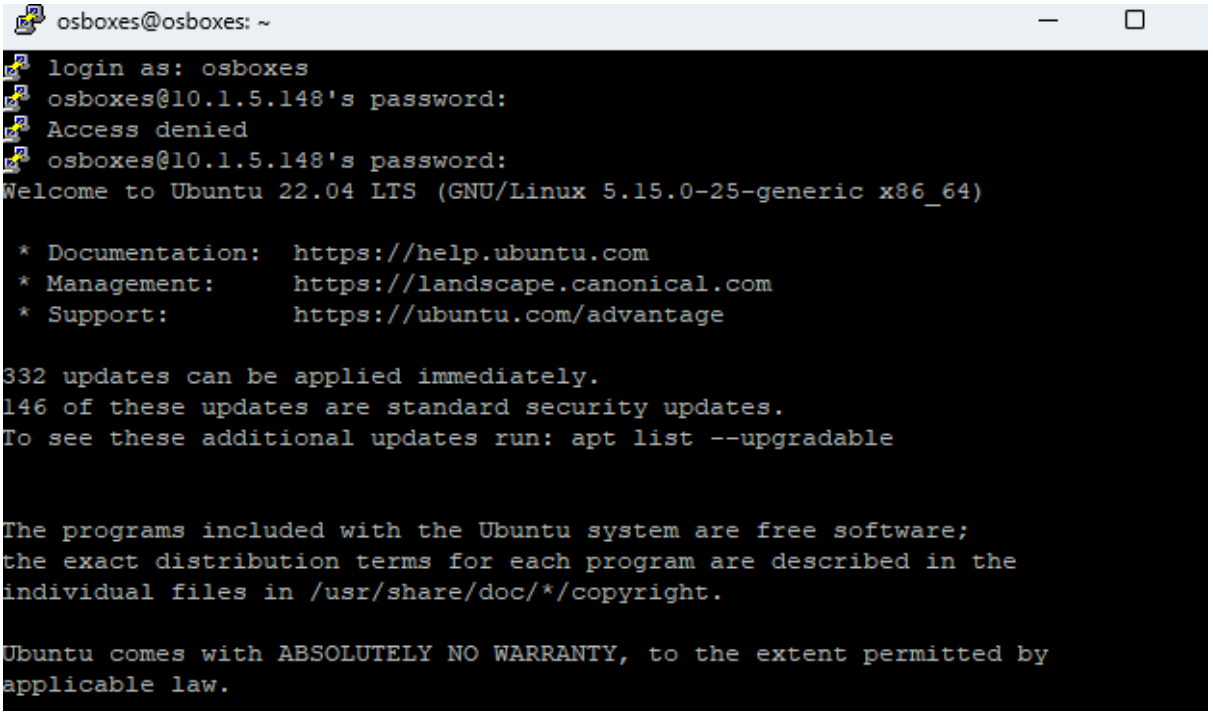
L'adresse IP de la VM n'étant pas routable, on ne peut pas communiquer directement avec la VM depuis la machine hôte, il faut donc du port forwarding.

1.3.3. Configurer la connectivité « manquante ».

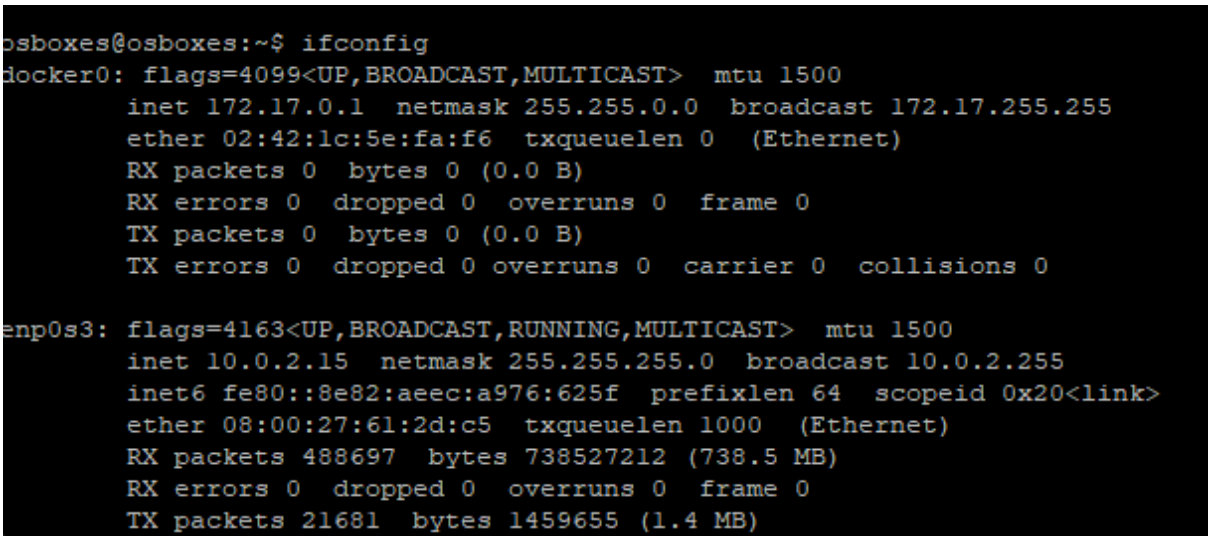
La machine voisine à l'hôte vers la VM :

Le ping échoue car l'adresse IP de la machine virtuelle (VM) se trouve sur un réseau différent, ce qui la rend non routable depuis l'extérieur. Pour garantir la connectivité, il est nécessaire de configurer une règle de redirection de port (port forwarding) dans l'interface de VirtualBox. Il faut y spécifier l'adresse IP de la machine hôte (10.1.5.89), choisir un port disponible, comme le port 1234, et rediriger ce port vers le port 22 (SSH) de la machine virtuelle, qui a pour adresse IP 10.0.2.15.

Ainsi, pour se connecter en SSH à la machine virtuelle, il suffira de se connecter sur le port 1234 de la machine hôte. La règle de NAT (Network Address Translation) transférera automatiquement la demande vers le port 22 de l'IP de la VM, soit 10.0.2.15.



on vérifie l'adresse IP avec ifconfig



1.3.4. Clonage d'une machine virtuelle

Dans notre processus de gestion d'infrastructures virtualisées, il peut être nécessaire de créer de nouvelles machines virtuelles à partir de disques existants.

La première étape consiste à cloner le fichier de disque virtuel qui contient toutes les données de la machine virtuelle. Le clonage de ce disque nous permet de créer une copie

exacte sans avoir à reconfigurer la VM de zéro. Pour ce faire, nous pouvons utiliser la commande suivante :

```
VBoxManage clonemedium "chemin\vers\disk.vdi" "chemin\vers\disk-copy.vdi"
```

Une fois le disque cloné, il est nécessaire de créer une nouvelle machine virtuelle et d'attacher le disque cloné à celle-ci. Nous réalisons cette action à l'aide de VirtualBox.

1.3.5. Approvisionnement de conteneurs Docker

Nous allons mettre en place cette fois un environnement Docker, qui sera déployé sur une VM VirtualBox pour qu'on puisse avoir les privilèges administrateurs. Il faut tout d'abord installer Docker Engine.

Après avoir mis à jour les listes de paquets avec `$ sudo apt update`, les commandes suivantes vont permettre de récupérer le dossier Docker pour pouvoir utiliser ses outils.

```
$ sudo apt install apt-transport-https ca-certificates curl
software-properties-common
```

Nous ajoutons la clé GPG officielle de Docker pour garantir la sécurité des téléchargements :

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg
--dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

Nous ajoutons ensuite le dépôt Docker à notre liste de sources afin de pouvoir récupérer les paquets depuis Docker et non depuis les dépôts d'Ubuntu par défaut :

```
$ echo "deb [arch=$(dpkg --print-architecture)
signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo
tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Avant d'installer Docker, nous pouvons vérifier que les paquets proviennent bien du dépôt Docker et non d'Ubuntu en exécutant la commande suivante :

```
$ apt-cache policy docker-ce
```

Enfin, nous installons Docker avec la commande suivante :

```
$ sudo apt install docker-ce
```

et on vérifie que le daemon tourne bien.

```
osboxes@osboxes:~/Desktop$ sudo systemctl status docker
[sudo] password for osboxes:
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset
     Active: active (running) since Thu 2025-11-06 06:22:59 EST; 1min 48s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 976 (dockerd)
      Tasks: 10
     Memory: 102.8M
        CPU: 1.009s
```

```
$ sudo systemctl status docker
```

Nous allons maintenant créer des conteneurs et vérifier les connectivités. On commence par récupérer une image ubuntu :

```
$ sudo docker pull ubuntu
```

```
osboxes@osboxes:~/Desktop$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
4b3ffd8ccb52: Pull complete
Digest: sha256:66460d557b25769b102175144d538d88219c077c678a49af4afca6fbfc1b5252
Status: Downloaded newer image for ubuntu:latest
docker.io/library/ubuntu:latest
```

Et on exécute le 1er conteneur (CT1) depuis cette image :

```
$ sudo docker run --name ct1 -it ubuntu
```

Après avoir installé les outils de test de connectivité, on peut analyser différents paramètres.

1. **Quel est l'adresse IP du Docker?** 172.17.0.2
2. **Test de ping d'une ressource internet depuis le Docker?** ça fonctionne.
3. **Test de ping vers la machine virtuelle depuis le Docker?** ça fonctionne.
4. **Test de ping vers le Docker depuis la machine virtuelle?** ça fonctionne.

Les communications marchent bien car le Docker Engine va créer un VLAN regroupant les conteneurs et il sera accessible par la machine hôte grâce à au NAT du Docker Engine.

Il est également possible de créer des instances depuis un snapshot. Pour cela on commence d'abord par créer un conteneur CT2, sur lequel on va installer l'outil d'édition de texte "nano" :

```
$ sudo docker run --name ct2 -p 2223:22 -it ubuntu
```

```
[CT2] $ apt-get -y update && apt install nano
```

On fait un snapshot de CT2 :

```
$ sudo docker commit %ID_de_CT2  %REPO:%TAG
```

Et on crée une nouvelle instance CT3 grâce au snapshot précédent :

```
$ sudo docker run --name ct3 -it %REPO:%TAG
```

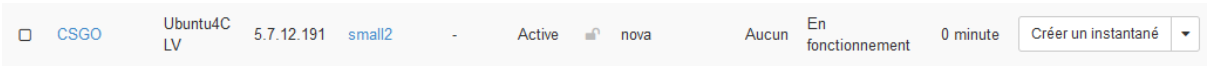
On remarque que sans même l'avoir installé, l'outil d'édition de texte "nano" est présent sur l'instance CT3. Comme le conteneur CT3 a été généré via le snapshot de CT2 où l'outil nano a été installé, il possède également toutes les données et installation de CT2.

1.3.6. Création de conteneurs et configuration sur OpenStack

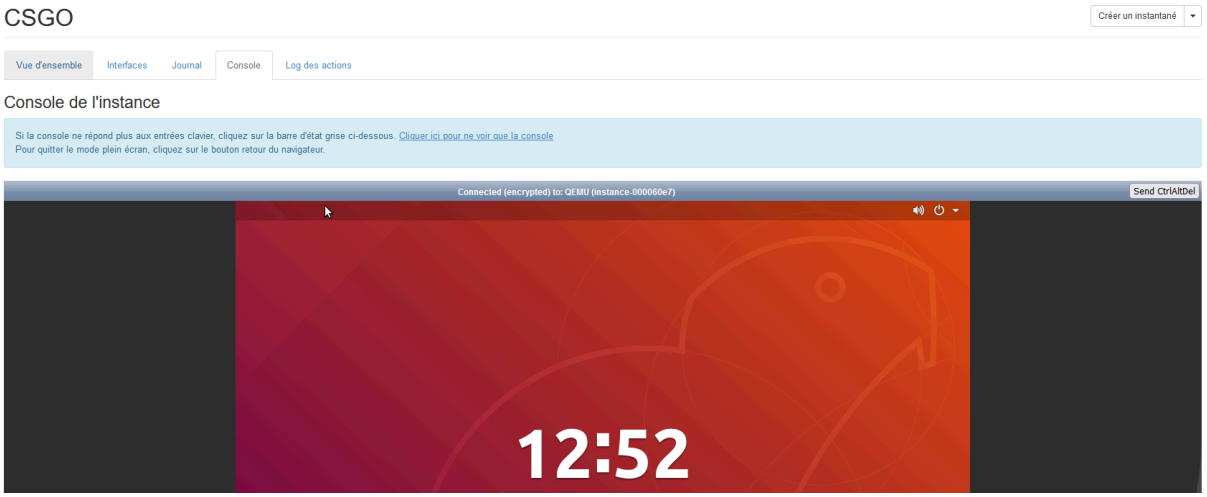
Nous allons maintenant prendre en main l'outil OpenStack pour gérer les machines virtuelles (VM) et les connexions réseau.

Commençons par nous connecter à l'interface web d'OpenStack et créer une machine virtuelle en utilisant les paramètres suivants :

- **Image :** "ubuntu4CLV"
- **Type d'instance :** "small2"



Une fois la VM créée, nous pouvons également nous y connecter en utilisant la console intégrée d'OpenStack.



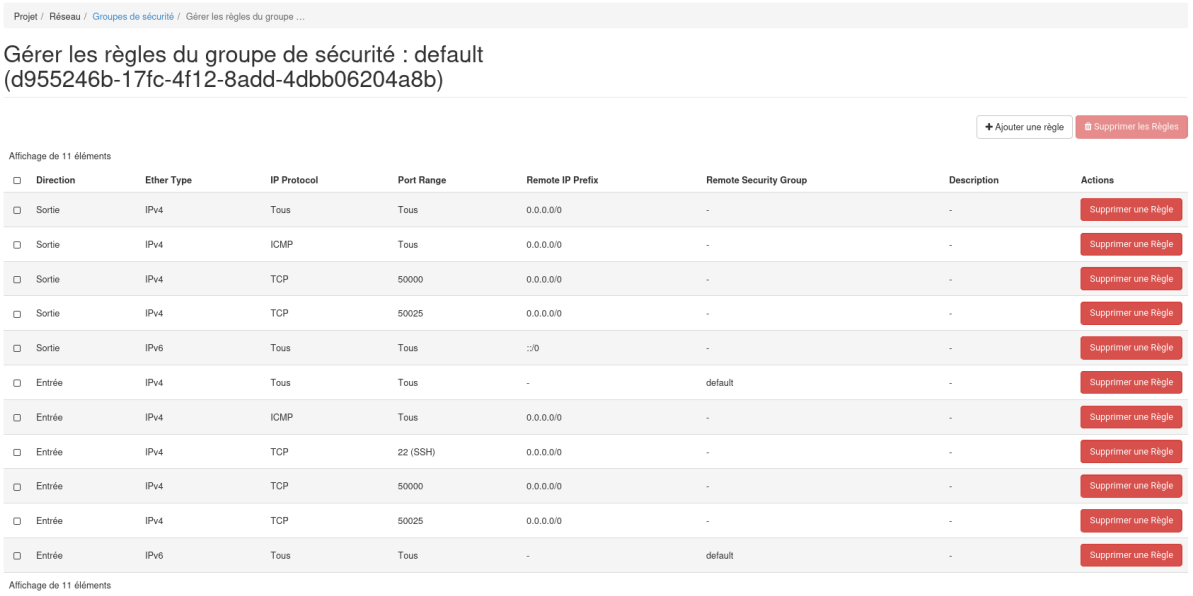
Cependant, la machine reste actuellement injoignable depuis l'extérieur, ce qui nécessite de configurer correctement les paramètres de réseau et de sécurité.

Configuration des règles de sécurité dans OpenStack

Par défaut, OpenStack bloque le trafic réseau sur le réseau privé virtualisé. Ce trafic peut être géré à l'aide de règles de sécurité spécifiques à chaque projet. Afin de pouvoir effectuer

un ping et se connecter en SSH à la machine distante, il est nécessaire d'autoriser le trafic ICMP et SSH.

Pour ce faire, il faut se rendre dans la section "Groupe de sécurité" sous l'onglet "Réseau". Et, modifiez le groupe de sécurité par défaut pour ajouter et autoriser les flux ICMP et SSH nécessaires.



1.3.7. Test de connectivité

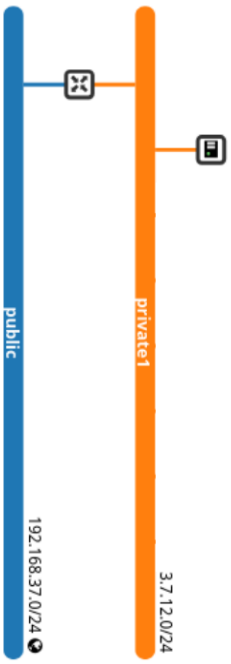
L'adresse IP de la machine virtuelle (VM), attribuée par l'hyperviseur, est affichée sur le tableau de bord (onglet Instances). Que pensez-vous de cette adresse ? Notez vos commentaires dans le document partagé.

L'adresse IP de la machine virtuelle est une adresse privée par conséquent non routable. La machine reste donc pour le moment injoignable depuis l'extérieur.

Utilisation d'un Ping (ou de toute autre commande appropriée) :

La VM, étant sur un réseau privé sans adresse publique, peut établir une connexion vers le poste de travail uniquement si les deux machines sont dans le même sous-réseau et que les règles de pare-feu le permettent. Le ping ne répond donc pas.

De la même façon, le poste de travail peut atteindre la VM via l'adresse IP privée si les deux machines sont dans le même réseau. Il n'y a donc pas de réponse au ping.



En adoptant cette configuration, en connectant le réseau privé à Internet via un routeur et en attribuant une adresse IP flottante à la machine virtuelle, celle-ci devient accessible depuis l'extérieur. Il est ainsi possible de réaliser des tests de connectivité, tels que des pings, entre la machine virtuelle et l'extérieur, dans les deux sens.

1.3.8. Instantané, restauration et redimensionnement d'une machine virtuelle.

Nous tentons dans un premier temps de redimensionner une machine virtuelle en fonctionnement directement depuis l'onglet Instances d'OpenStack, voici ce que nous pouvons observer :

Il n'est pas possible de redimensionner une VM lorsqu'elle est en fonctionnement. Nous avons dans un premier temps supposé qu'il fallait éteindre la VM pour pouvoir la redimensionner. En théorie, cela devrait être possible même en fonctionnement, mais cette option est désactivée pour les étudiants, probablement en raison des coûts associés à cette opération.

La fonctionnalité de redimensionnement à chaud présente de nombreux avantages. Elle permet une meilleure disponibilité, car elle n'entraîne aucune interruption de service lors du redimensionnement, garantissant ainsi une continuité des opérations. De plus, elle offre une amélioration des performances en temps réel, en ajustant dynamiquement les ressources en fonction des besoins. Cette fonctionnalité contribue ainsi à la flexibilité et à une agilité du cloud, permettant aux systèmes de s'adapter rapidement aux variations de la charge.

Une fois la VM arrêtée, nous réessayons de redimensionnement depuis l'onglet Instances. Nous constatons que :

Avec la VM éteinte, il n'est toujours pas possible de redimensionner la VM. : “Danger**:** Une erreur s'est produite. Veuillez réessayer ultérieurement.” Il est possible de redimensionner la VM en marche comme éteinte cependant comme vu plus haut, cette fonction est désactivée pour les étudiant car coûteuse.

La virtualisation nous offre une grande flexibilité pour ajuster les ressources allouées aux VMs, ce qui permet de réagir rapidement aux besoins changeants. Cependant, il existe certaines limitations techniques dans OpenStack.

À partir de l'onglet "Instances" d'OpenStack, nous créons un snapshot. Un snapshot est une image de la machine virtuelle à un moment précis, capturant son état, incluant le disque, le système d'exploitation et tous les processus en cours d'exécution.

L'image de base contient uniquement le système d'exploitation et des logiciels préinstallés. Elle ne capture pas les changements ou les données générées après la création de la VM (telles que les fichiers, les logiciels installés ou les configurations personnalisées). Un snapshot capture l'état complet de la VM à un moment donné, avec notamment :

- Les données du disque
- Les processus en cours

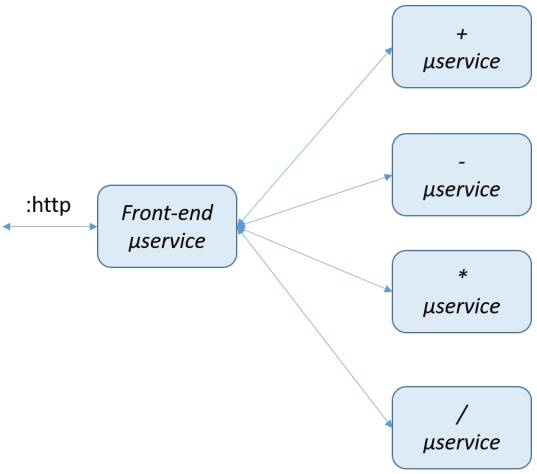
Lorsque nous utilisons un snapshot pour créer une nouvelle VM, il est important de noter

que cette opération peut prendre un certain temps supplémentaire par rapport à la création d'une machine virtuelle classique.

1.3.9. Topologie et spécifications d'une application Web à deux niveaux

Nous souhaitons désormais déployer une application web à deux niveaux sur OpenStack. Cette application consiste en une calculatrice capable de réaliser des opérations arithmétiques de base (addition, soustraction, multiplication et division) sur des nombres entiers.

Chaque opération est implémentée sous forme de microservice indépendant développé en Node.js. Un cinquième microservice gère le point d'entrée de l'application, en assurant l'écoute et la réception des requêtes HTTP lors du démarrage de l'application. Ces requêtes, spécifient l'opération arithmétique à effectuer, ces dernières sont ensuite traitées par les microservices correspondants. Les résultats des calculs sont ensuite affichés à la fois sur le front-end et renvoyés au client via l'interface HTTP.



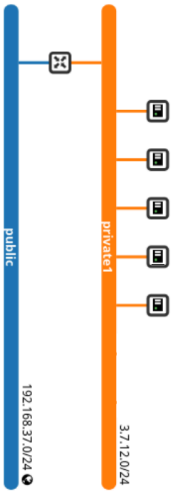
1.3.10. Déploiement de la calculatrice sur OpenStack

Pour déployer les microservices sur OpenStack, nous devons créer 5 machines virtuelles, chacune dédiée à un microservice. Nous utiliserons l'image Ubuntu4CLV.

Avant de démarrer un service, nous devons installer les outils nécessaires sur notre machine :

- **npm** pour la gestion des dépendances
- **Node.js** pour exécuter les services Node.js
- **curl** pour tester les services via des requêtes HTTP

Nous configurons la topologie du réseau de la manière suivante :



Une seule machine doit disposer d'une adresse IP flottante : la VM correspondant au **"CalculatorService"**. Cette VM doit être accessible depuis l'extérieur.

Pour tester le bon fonctionnement de chaque microservice, nous envoyons des requêtes cURL depuis notre machine (lors de l'installation du microservice, quand la VM dispose d'une IP flottante) ou depuis une autre VM située sur le même réseau privé.

Pour le microservice **"CalculatorService"**, nous devons également modifier son code afin que l'adresse IP de chaque microservice soit correctement configurée.

```
var http = require ('http');
var request = require('sync-request');

const PORT = process.env.PORT || 50000;

const SUM_SERVICE_IP_PORT = 'http://3.7.12.210:50001';
const SUB_SERVICE_IP_PORT = 'http://3.7.12.28:50002';
const MUL_SERVICE_IP_PORT = 'http://3.7.12.201:50003';
const DIV_SERVICE_IP_PORT = 'http://3.7.12.212:50004';
```

Après modification du code, le service doit être redémarré pour que les changements prennent effet. Pendant cette phase de redémarrage, le service devient temporairement indisponible, ce qui peut entraîner l'échec des requêtes envoyées à ce moment-là.

Cependant, les autres microservices, hébergés sur des machines virtuelles distinctes, ne sont pas affectés, car ils fonctionnent de manière indépendante. Cela démontre l'indépendance et la modularité des microservices : il est possible de modifier ou redéployer un service sans impacter les autres.

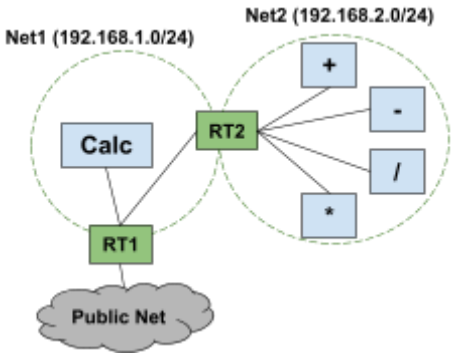
Il est important de noter qu'il n'y a pas de rechargement automatique après modification du code : toute mise à jour nécessite un redémarrage manuel. Pour automatiser ce processus et éviter les interruptions, des outils comme Docker peuvent être utilisés pour faciliter le redéploiement.

Pour que les VMs puissent recevoir des requêtes HTTP nous avons également dû ouvrir un port pour pouvoir communiquer en TCP, nous avons choisi le 50000, correspondant au premier port autorisé par le CSN défini entre 50000 et 50050.

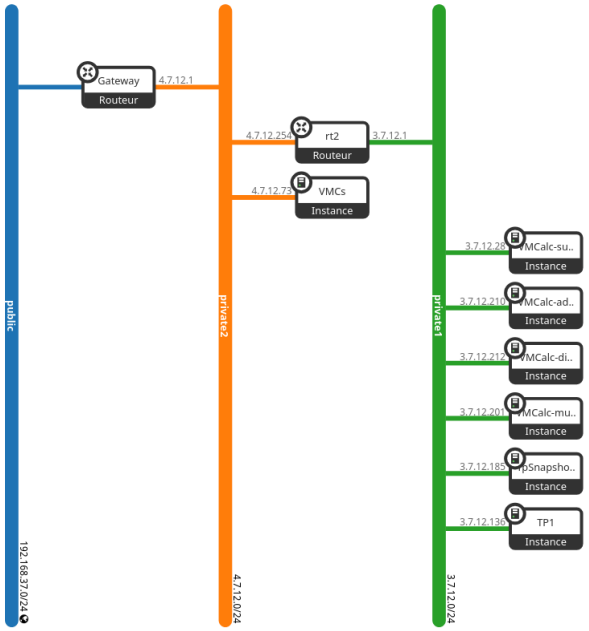
Entrée	IPv4	TCP	50000	0.0.0.0/0	-	-	Supprimer une Règle
Sortie	IPv4	TCP	50000	0.0.0.0/0	-	-	Supprimer une Règle

1.3.11. Exigences du client et la topologie du réseau cible

Afin de garantir un accès sécurisé aux services intermédiaires nous déployons la topologie réseau suivante :



Ce déploiement comprend deux sous-réseaux distincts hébergeant des machines virtuelles (VM) qui exécutent les fonctionnalités de l'application.



Nous souhaitons vérifier la connectivité entre la machine virtuelle VMCs situées sur le réseau 4.7.12.0/24 et les services arithmétiques disponibles sur le réseau 3.7.12.0/24.

Lors de l'exécution des commandes de diagnostic réseau (ping et curl) depuis la VM VMCs vers les services sur 3.7.12.0/24, l'erreur suivante apparaît : "Network is unreachable"

Cela indique une absence de route réseau vers le sous-réseau 3.7.12.0/24 depuis la VM VMCs.

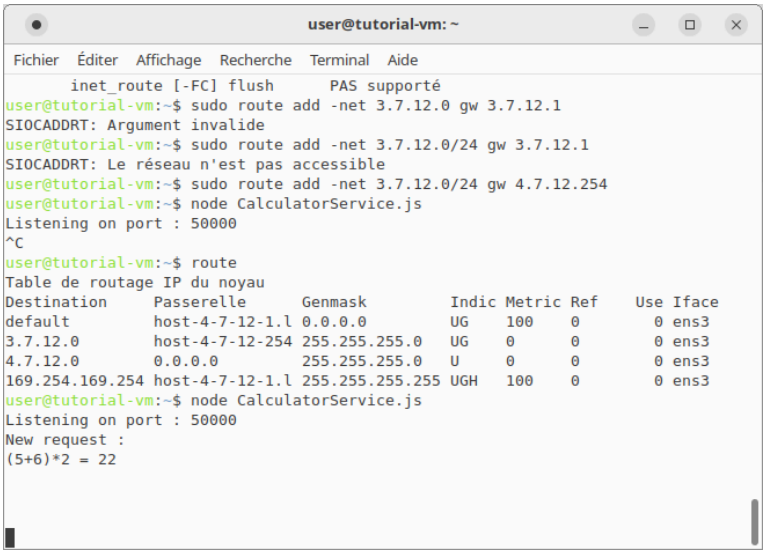
La raison de cet échec de communication est l'absence de route configurée pour acheminer le trafic entre les sous-réseaux 3.7.12.0/24 et 4.7.12.0/24, l'enssemble du trafic est dirigé vers la passerelle par défaut.

Corrections :

Il est possible de configurer une route manuellement sur la VM VMCs pour accéder au sous-réseau 3.7.12.0/24 via la passerelle (gateway) du réseau 4.7.12.0/24. Cela peut être effectué en exécutant la commande suivante sur la VM VMCs :

```
$ sudo route add -net @3.7.12.0/24 gw 4.7.12.254
```

Après modification les tests sont réussis (ping et curl), la connectivité entre la VM VMCs et les services arithmétiques est rétablie.



1.3.12. Automatisation/Orchestration du déploiement de l'application (facultatif)

Nous souhaitons désormais automatiser le déploiement de la calculatrice distribuée (5 microservices NodeJS). Pour ce faire nous utilisons des scripts Bash et openrc, afin d'automatiser sans intervention manuelle.

Pour automatiser le déploiement des applications sur un ensemble de machines virtuelles créées dans l'environnement OpenStack. Nous utilisons un script Bash pour :

Créer et configurer les ressources dans OpenStack (réseaux, sous-réseaux, routeurs et VMs).

```
  GNU nano 6.2                                    create.sh
set -e #arrete le script en cas d'erreur
#Variables
nom_machine=("ADDGO" "SUBGO" "MULGO" "DIVGO")
flavor="small2"
image="Ubuntu4CLV"

#Commands
source Downloads/5ISS-A2-4-openrc.sh
openstack network create private3
echo "reseau 3 cree"
openstack network create private4
echo "reseau 4 cree"
openstack subnet create subnet3 --network private3 --subnet-range 5.7.12.0/24
echo "subnet3 cree"
openstack subnet create subnet4 --network private4 --subnet-range 6.7.12.0/24
echo "subnet4 cree"
openstack router create router3
openstack router set router3 --external-gateway public
openstack router add subnet router3 subnet3
echo "router 3 ready"
openstack router create router4
openstack port create PORT4TO3 --network private3 --fixed-ip subnet=subnet3,ip-address=5.7.12.254
openstack router add subnet router4 subnet4
openstack router add port router4 PORT4TO3
echo "router 4 ready"
openstack server create CSGO --flavor "$flavor" --image "$image" --network private3 --security-group default
for i in {0..3}; do
    openstack server create "${nom_machine[$i]}" --flavor "$flavor" --image "$image" --network private4 --security-group "default"
    echo "machine cree"
done

echo "tout est cree"
```

2. Services orchestrateurs dans un environnement hybride cloud/edge

2.1. Introduction

L'objectif principal de ce TP est de comprendre comment le cloud computing et l'edge computing collaborent pour répondre aux exigences en temps réel, telles que celles des voitures autonomes. Le cloud computing permet d'accéder à des ressources puissantes, mais éloignées des utilisateurs, ce qui engendre une latence élevée et des coûts potentiellement plus importants. En revanche, l'edge computing offre un accès à des ressources plus proches de l'utilisateur, ce qui permet de réduire la latence et constitue une solution idéale pour des applications nécessitant des réponses quasi instantanées, comme c'est le cas pour les voitures autonomes.

Ainsi, les services associés à la voiture autonome doivent pouvoir migrer d'un nœud à un autre au sein du réseau edge, afin de suivre en permanence le véhicule et de maintenir une connexion à faible latence.

Dans le cadre de ce TP, l'infrastructure suivante est considérée :

- **1 Master node** (gère le cluster Kubernetes)

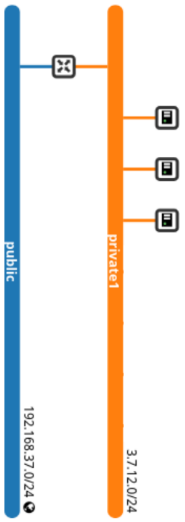
- **2 Worker nodes** (hébergent les services edge)

Nous utiliserons le système d'exploitation Ubuntu 20.04, ainsi que Docker pour l'exécution des conteneurs. Enfin, Kubernetes sera utilisé pour l'orchestration des conteneurs et la gestion du déploiement.

2.2. Configuration de l'infrastructure cloud

Nous commençons dans un premier temps par créer les machines virtuelles sur OpenStack :

Nous avons ainsi 3 VMs : Master, Worker1, Worker2 avec la topologie réseau suivante :



Nous associons également une adresse IP flottante à chaque machine, elles sont donc joignables depuis l'extérieur.

Nous configurons les machines virtuelles afin de les préparer pour le déploiement d'un cluster Kubernetes.

Nous effectuons les actions suivantes pour chaque VMs:

1. Création et configuration
2. Création d'adresses IP flottantes
3. Création d'un nouvel utilisateur
    a. Attributions des privilèges sudo à l'utilisateur
4. Configurer SSH
    a. Modification de la configuration SSH pour autoriser l'authentification par mot de passe
    b. Redémarrer le service SSH pour appliquer les changements
5. Désactivation du swap et modification du nom d'hôte
    a. Désactiver le swap sur chaque machine

b. Changer le nom d'hôte des machines

Un fois avoir réalisé l'instantiation et la configuration de nos machines nous passons l'installation de Kubernetes et de la configuration des nœuds dans un cluster, avec Docker comme moteur de conteneurs, sur toutes les machines (le nœud master et les workers).

1. Mise à jour des packages et installation de curl pour récupérer le dépôt Kubernetes
2. Obtentions de la clé du dépôt Kubernetes
3. Ajout du dépôt Kubernetes
4. Mise à jours des paquets
5. Installation des composants Kuberentes :
    - kubeadm : outil utilisé pour initialiser le cluster Kubernetes.
    - kubelet : il fonctionne sur chaque nœud du cluster est permet de gérer les conteneurs et les pods.
    - kubectl : permet d'interagir avec le cluster Kubernetes.
6. Installation de Docker
7. Activation de Docker au démarrage du système
8. Modification du pilote Cgroup de Docker pour éviter les erreurs
9. Redémarrage de Docker pour appliquer les changement
10. Vérifier que le pilote Cgroup est bien défini sur systemd
11. Activation de l'IP Forwarding et les modules nécessaires pour Kubernetesv
12. Rechargement la configuration des services systemd
13. Redémarrage du service kubelet

Après avoir correctement configuré notre système pour exécuter Kubernetes, nous avons vérifié que Docker, kubelet et les autres composants essentiels fonctionnent. Nous souhaitons désormais initialiser notre cluster Kubernetes. Nous commençons par initialiser la master en exécutant sur le master node la commande suivante :

```
sudo kubeadm init --apiserver-advertise-address <master_private_ip>
--control-plane-endpoint <master_private_ip>
```

Nous définissons ainsi l'adresse IP utilisée par les noeuds pour communiquer avec le master.

Nous effectuons ensuite ces 3 commandes :

```
mkdir -p $HOME/.kube
```

```
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

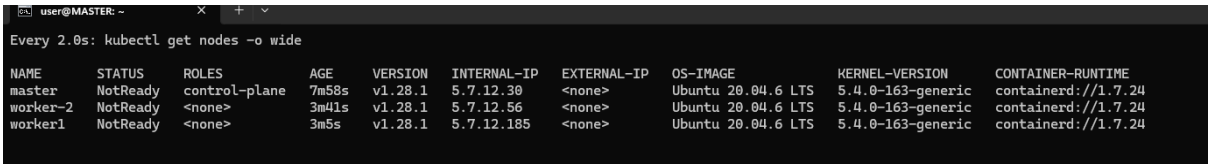
Permettent de configurer l'accès kubectl pour le user courant.

Une fois avoir créé le token d'adhésion sur le master, nous connectons chacun des deux workers à ce dernier.

Vérification du cluster :

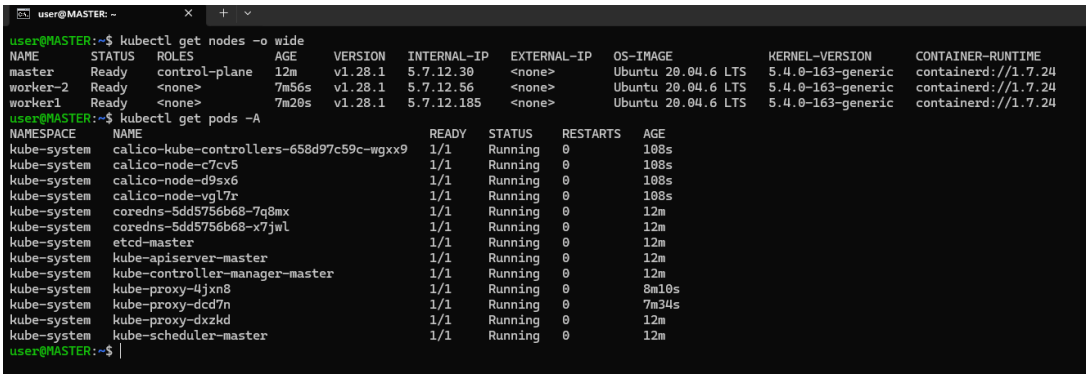
Une fois tous les noeuds ajoutés, nous effectuons la commande suivante avec un watch afin d'observer l'évolution :

```
kubectl get nodes -o wide
```



Nous remarquons que le statut des nœuds est NotReady. Les nœuds ne sont pas encore opérationnels et ne peuvent pas encore exécuter de pods.

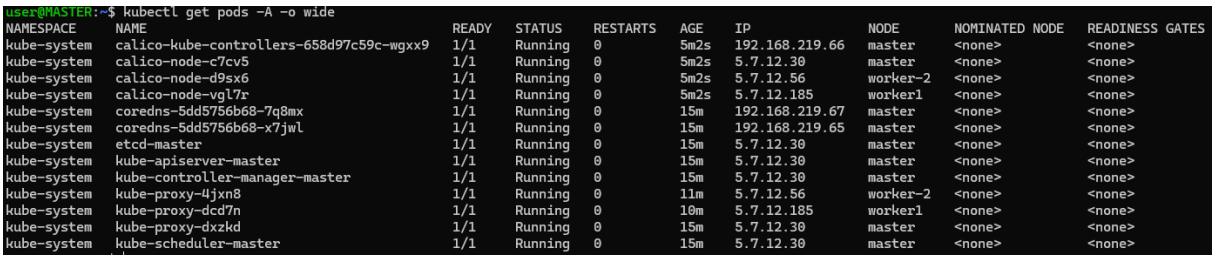
Nous utilisons le plugin réseau Calico pour que les pods puissent communiquer à travers les nœuds du cluster. Calico implémente le Container Networking Interface (CNI), qui permet de connecter les pods entre eux sur plusieurs nœuds.



Après avoir installé Calico, nous remarquons que tous les nœuds sont passés à l'état Ready. Cela signifie que les nœuds sont maintenant prêts à exécuter des pods et à assurer la communication réseau entre eux.

Pour afficher sur quel nœud un pod est exécuté, nous utilisons la commande :

```
kubectl get pods -o wide
```

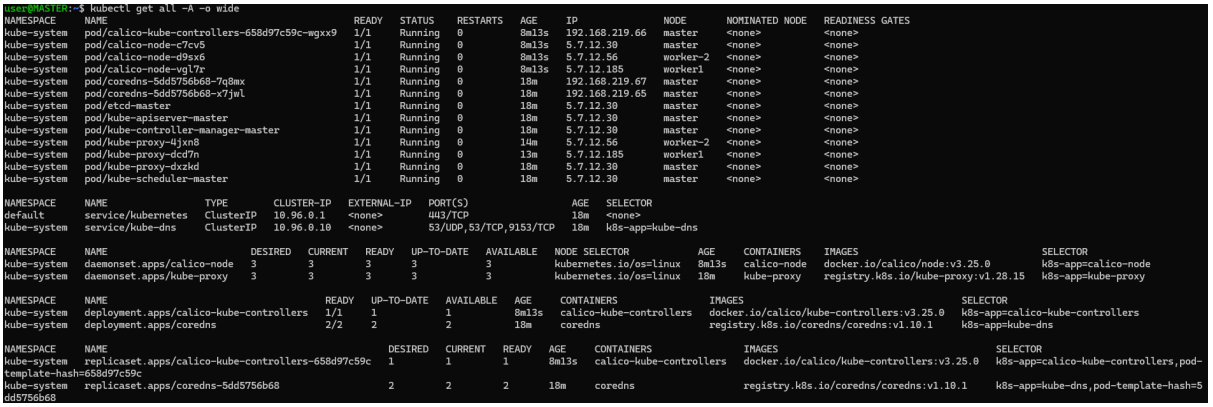


Cela nous permet de savoir précisément sur quel nœud un pod est déployé.

Nous utilisons ensuite la commande :

```
kubectl get all -A -o wide
```

Pour afficher une vue d'ensemble de tous les objets de notre cluster Kubernetes. Cela nous permet de pouvoir injecter l'état des objets sur les différents nœuds.



```
user@MASTER:~$ kubectl get all -A -o wide
NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE     IP               NODE       NOMINATED NODE   READINESS GATES
kube-system   pod/calico-kube-controllers-658d97c59c-mgxx9   1/1     Running   0          8m13s   192.168.219.66   master     <none>           <none>
kube-system   pod/calico-node-c7cv5                      1/1     Running   0          8m13s   5.7.12.30        master     <none>           <none>
kube-system   pod/calico-node-d9sx6                      1/1     Running   0          8m13s   5.7.12.56        worker-2   <none>           <none>
kube-system   pod/calico-node-vgl7r                      1/1     Running   0          8m13s   5.7.12.185       worker1    <none>           <none>
kube-system   pod/coredns-5dd5756b68-7q8mx               1/1     Running   0          18m     192.168.219.67   master     <none>           <none>
kube-system   pod/coredns-5dd5756b68-x7jwl               1/1     Running   0          18m     192.168.219.65   master     <none>           <none>
kube-system   pod/etcd-master                            1/1     Running   0          18m     5.7.12.30        master     <none>           <none>
kube-system   pod/kube-apiserver-master                  1/1     Running   0          18m     5.7.12.30        master     <none>           <none>
kube-system   pod/kube-controller-manager-master         1/1     Running   0          18m     5.7.12.30        master     <none>           <none>
kube-system   pod/kube-proxy-4jxn8                       1/1     Running   0          14m     5.7.12.56        worker-2   <none>           <none>
kube-system   pod/kube-proxy-dcd7n                       1/1     Running   0          13m     5.7.12.185       worker1    <none>           <none>
kube-system   pod/kube-proxy-dxzkd                       1/1     Running   0          18m     5.7.12.30        master     <none>           <none>
kube-system   pod/kube-scheduler-master                  1/1     Running   0          18m     5.7.12.30        master     <none>           <none>

NAMESPACE     NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE   SELECTOR
default       service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP                  18m   <none>
kube-system   service/kube-dns     ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   18m   k8s-app=kube-dns

NAMESPACE     NAME                         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE     CONTAINERS    IMAGES                                SELECTOR
kube-system   daemonset.apps/calico-node   3         3         3       3            3           kubernetes.io/os=linux   8m13s   calico-node   docker.io/calico/node:v3.25.0         k8s-app=calico-node
kube-system   daemonset.apps/kube-proxy    3         3         3       3            3           kubernetes.io/os=linux   18m     kube-proxy    registry.k8s.io/kube-proxy:v1.28.15   k8s-app=kube-proxy

NAMESPACE     NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS                IMAGES                                        SELECTOR
kube-system   deployment.apps/calico-kube-controllers   1/1     1            1           8m13s   calico-kube-controllers   docker.io/calico/kube-controllers:v3.25.0     k8s-app=calico-kube-controllers
kube-system   deployment.apps/coredns               2/2     2            2           18m     coredns                   registry.k8s.io/coredns/coredns:v1.10.1       k8s-app=kube-dns

NAMESPACE     NAME                                             DESIRED   CURRENT   READY   AGE     CONTAINERS                IMAGES                                        SELECTOR
kube-system   replicaset.apps/calico-kube-controllers-658d97c59c   1         1         1       8m13s   calico-kube-controllers   docker.io/calico/kube-controllers:v3.25.0     k8s-app=calico-kube-controllers,pod-
template-hash=658d97c59c
kube-system   replicaset.apps/coredns-5dd5756b68               2         2         2       18m     coredns                   registry.k8s.io/coredns/coredns:v1.10.1       k8s-app=kube-dns,pod-template-hash=5
dd5756b68
```

Il est possible de déployer des services sur des nœuds spécifiques (comme Deployments kube) ou bien de façon automatique lors de l'ajout de nouveaux nœuds grâce à des DaemonSets permettant que que chaque nœud du cluster exécute une copie d'un pod spécifique.

2.3. Compréhension de Kubernetes et de ses composants

Kubernetes n'exécute pas directement les conteneurs, mais les encapsule dans des pods.

Les principaux composants sont :

Interface	Composant utilisé
CRI (Container Runtime Interface)	Containerd
CNI (Container Networking Interface)	Calico
CSI (Container Storage Interface)	

Lors de la configuration du cluster, nous avons intégré ces composants pour permettre l'exécution des conteneurs, mais nous n'avons pas encore tout mis en place. Kubernetes ne travaille pas directement avec les conteneurs, mais avec des "pods", qui sont des objets encapsulant le concept de conteneur. Ces pods peuvent fonctionner seuls ou être associés à d'autres objets.

Il existe trois types de services dans Kubernetes :

Service	Utilisation
ClusterIP	Permet aux pods de communiquer entre eux à travers un réseau interne sans être accessible depuis l'extérieur du cluster. service par défaut dans Kubernetes)
NodePort	Expose un service à l'extérieur du cluster en attribuant un port fixe sur chaque nœud du cluster.
LoadBalancer	Répartit le trafic entrant sur plusieurs nœuds ou pods

Une fois un compte Docker créé, nous procédons à l'authentification à l'aide de la commande "docker login". Afin d'organiser les ressources nous avons ensuite ajouté des labels aux nœuds du cluster à l'aide de la commande"kubectl label node" pour identifier les nœuds selon leur rôle.

Depuis le master nous effectuons les commandes suivantes :

```
git clone https://github.com/ced-yxos/kube_service.git
```

La commande git clone permet de télécharger sur le dépôt git ci-dessus plusieurs fichiers YAML de configuration Kubernetes pour des services NodePort et ClusterIP.

```
$ kubectl apply -f ./kube_service/ClusterIP
```

On vient ensuite appliquer la configuration décrites dans le fichiers YAML contenus dans le dossier CluserIP qui sont :

- app_deployment_1.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastapi-app

spec:
  replicas: 3
  selector:
    matchLabels:
      app: fastapi-app
  template:
    metadata:
      labels:
        app: fastapi-app

    spec:
      imagePullSecrets:
        - name: repo-key
      containers:
        - name: fastapi-app-container
          image: yxos/fastapi-app
          imagePullPolicy: Always
          ports:
            - containerPort: 5000
              protocol: TCP
      nodeSelector:
          PoP: space_1
```

- app_ingress.yaml

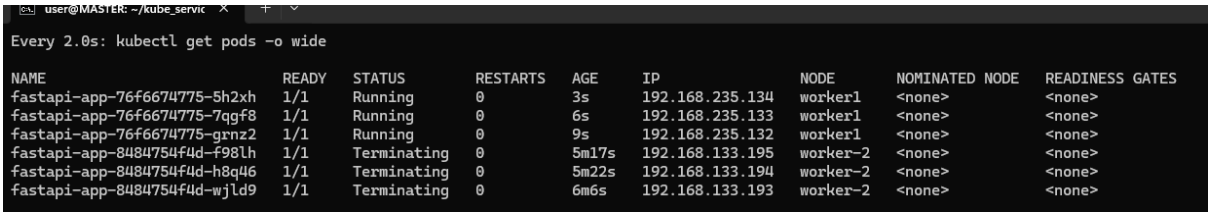
```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: fastapi-app-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:
  rules:
    - http:
        paths:
          - backend:
              service:
                name: fastapi-app-service
                port:
                  number: 5000
            path: /
            pathType: Prefix
```

- app_service_cluster_ip.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: fastapi-app-clusterip-service
spec:
  selector:
    app: fastapi-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: ClusterIP
```

Avec la commande kubectl get pods -o wide on voir apparaître plusieurs pods FastAPI en état de running. On voit également sur quel nœud chaque pod est déployé.



On remarque que 3 pods FastAPI ont bien été créés (car définit replicas: 3 dans spec dans le fichier Deployment).

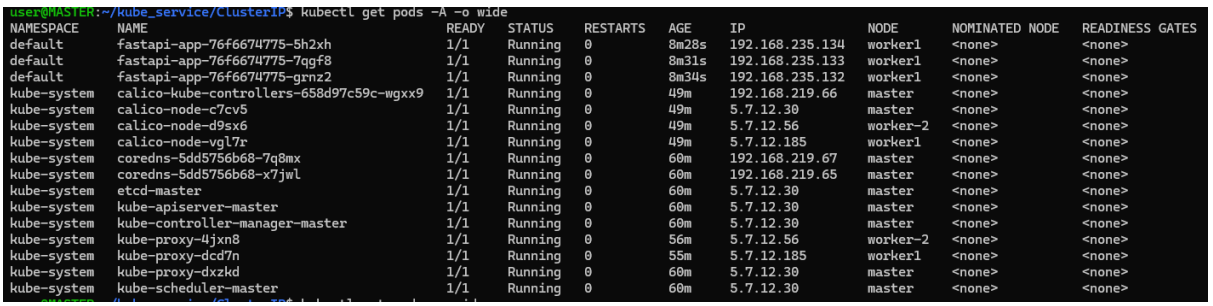
Les 3 pods tournent sur le même node (worker 1 dans le screenshot), car le fichier YAML utilise nodeSlector pour désigner les noeuds possédant le label “space_1” préalablement déterminé.

En modifiant la valeur du PoP dans le fichier app_deployment_1.yaml pour qu’elle corresponde à l’un des nœuds du cluster, il est possible de déplacer les services d’un nœud à l’autre.

Ainsi après avoir modifier le fichier app_deployment_1.yaml avec le label PoP: space_1 correspondant au noeud worker1, les pods du service ClusterIP précédemment déployés sur worker2 (dont le label est PoP: space_2) , on peut réexécuter la commande : $ kubectl apply -f ./kube_service/ClusterIP

Kubernetes met alors fin à ces pods sur worker2 et en recrée automatiquement de nouveaux sur worker1 afin de respecter la nouvelle contrainte de placement Kubernetes met alors fin à ces pods sur worker2 et en recrée automatiquement de nouveaux sur worker1 afin de respecter la nouvelle contrainte de nodeSelector. C’est ce mécanisme que l’on peut observer sur le screenshot ci-dessus.

Après la migration, de nouveaux pods apparaissent dans le namespace default, correspondant aux services ClusterIP. Ces pods sont désormais en état Running et s’exécutent sur le nœud worker1, confirmant que la relocalisation des services a bien été effectuée conformément au label PoP: space_1.



Ce résultat montre que Kubernetes a correctement réaffecté les pods en fonction de la nouvelle configuration du nodeSelector, garantissant ainsi la continuité du service tout en maintenant la faible latence.

On utilise ensuite la commande :

```
$ kubectl get services -o wide
```

Cette dernière nous permet d'obtenir la liste des services déployés dans le cluster, aced des informations comme le type de service (ClusterIP dans notre cas), l'adresse IP internet, le port et le sélecteur de label.

Puis :

```
$ kubectl describe services name_of_service
```

Qui nous permet d'obtenir une description plus détaillée du service, notamment la section Endpoints, qui indique les adresses IP / port des pods sur lesquels l'application FastAPI est réellement en cours d'exécution.

```
user@MASTER:~/kube_service/ClusterIP$ kubectl describe services fastapi-app-clusterip-service
Name:              fastapi-app-clusterip-service
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=fastapi-app
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.101.173.132
IPs:               10.101.173.132
Port:              <unset>  80/TCP
TargetPort:        5000/TCP
Endpoints:         192.168.235.132:5000,192.168.235.133:5000,192.168.235.134:5000
Session Affinity:  None
Events:            <none>
```

Nous avons les Endpoints suivants :
192.168.235.132:5000,192.168.235.133:5000,192.168.235.134:5000

Les adresses IP correspondent aux pods FastAPI créés par le Deployment appliqué précédemment. Chacune de ces adresses IP représente une instance de l'application, exécutée sur le nœud, dans ce cas, worker1. Le port 5000 est celui sur lequel le conteneur FastAPI écoute, comme défini précédemment dans le fichier de configuration YAML du Deployment : containerPort: 5000.

```
user@MASTER:~/kube_service/ClusterIP$ curl http://192.168.37.194:5000/
^C
```

Lorsque l'on tente d'accéder au service depuis l'extérieur du cluster avec la commande curl , la requête échoue. Car un service de type ClusterIP n'est accessible qu'à l'intérieur du cluster Kubernets, donc d'un pod à un autre. Ainsi les endpoits sont joignable uniquement via le réseau interne et non depuis une machine externe comme le master.

Avant de passer au service NodePort, on supprime toutes les ressources du service ClusterIP avec la commande :

```
$ kubectl delete -f ./kube_service/ClusterIP
```

Cela permet de nettoyer l'environnement et d'éviter tout conflit entre services portant le même nom ou utilisant les mêmes ports.

Ensuite depuis le master node, on déploie les nouvelles configurations lié au service NodePort :

```
$ kubectl apply -f ./kube_service/NodePort
```

kube_service/NodePort/ contient les fichiers suivants :

- app_deployment_1.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: fastapi-app

spec:
  replicas: 3
  selector:
    matchLabels:
      app: fastapi-app
  template:
    metadata:
      labels:
        app: fastapi-app

    spec:
      imagePullSecrets:
        - name: repo-key
      containers:
        - name: fastapi-app-container
          image: yxos/fastapi-app
          imagePullPolicy: Always
          ports:
            - containerPort: 5000
              protocol: TCP
      nodeSelector:
          PoP: space_1
```

- app_ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: fastapi-app-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"

spec:
  rules:
    - http:
        paths:
          - backend:
              service:
                name: fastapi-app-service
                port:
                  number: 5000
            path: /
            pathType: Prefix
```

- app_service_node_port.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: fastapi-app-service
spec:
  selector:
    app: fastapi-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: NodePort
```

Une fois appliqué, on exécute afin de vérifier le déploiement :

```
$ kubectl get pods -o wide
```

On remarque que les pods FastAPI sont à nouveau créés sur le nœud worker1, car le nodeSelector spécifie le label space_1 correspondant au worker1.

Nous observons ainsi 3 pods fastapi-app en état Running tous attachés au node worker1.

Nous utilisons ensuite :

```
$ kubectl get services -o wide
```

pour afficher les informations du service déployé.

On peut maintenant tester l'application depuis le master node avec la commande :

```
$ curl http://node_1_ip:NodePort_value/
```

Nous avons cette fois-ci une réponse, ce qui confirme que le service NodePort fonctionne correctement et que le trafic externe est bien redirigé vers les pods FastAPI internes.

Le passage de ClusterIP à NodePort montre bien la différence entre un service purement interne et un service accessible depuis le réseau externe.

Ainsi ClusterIP le service est accessible uniquement depuis à l'intérieur via un pod interne contrairement à NodePort qui est accessible via l'IP du nœud.