République Algérienne Démocratique et Populaire
الجمهوريـــة الجـزائريـة الديمـقراطيـة الشـعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التـعليـم العـالى و البـحث العلـمى

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
Ecole nationale Supérieure d'Informatique

2ème année cycle supérieure(2CS)

Option: Systèmes Informatiques (SQ)

Thème:

Mise en place et administration d'un cluster Kubernetes pour la Gestion des TP au niveau de l'ESI

Equipe N° 10:
- BELKESSA Linda (CE)
- DJABELKHIR Sarah
- CHELLAT Hatem
- BENHAMADI Yasmine
- BOUZOUAD Meriem
- LAMDANI Wilem

Encadré par:
- Mr AMROUCHE Hakim
- Mr SEHAD Abdenour
- Mr HAMANI Nacer

Promotion : 2021-2022

Résumé

Au cours des dernières années, l'utilisation des technologies de virtualisation dans l'infonuagique a considérablement augmenté. Ceci est dû principalement aux avantages en termes d'efficacité d'utilisation des ressources et de robustesse que procure la virtualisation. La virtualisation par conteneurs Docker et la virtualisation par hyperviseurs sont les deux principales technologies apparues sur le marché. Parmi elles, la virtualisation par conteneurs Docker se distingue par sa capacité de fournir un environnement virtuel léger et efficace, mais qui nécessite la présence d 'un orchestrateur. Par conséquent, les fournisseurs de l'infonuagique adoptent la plateforme ouverte Kubernetes comme étant le gestionnaire standard des applications conteneurisées.

Notre équipe Barbaros présente les détails de l'implémentation et l'installation de notre système de gestion des TPs, à travers ce rapport aussi, nous nous intéressons à l'aspect technique de notre solution en détaillant l'architecture et la conception de l'application qui se voit divisée en deux parties majeures à savoir, une plateforme web d'inscription et authentification facilitant l'accès au différents services, et le corps lui-même de notre cluster distribué et orchestré par Kubernetes.

Nous avions déjà présenté au client le prototype final qui assurait ce qui suit :

- Plateforme web d'authentification et d'inscription qui automatise l'affectation d'un port ensuite le lancement du VNCviewer.

- Cluster distribué sur plusieurs machines physiques appartenant au même réseau local (nous avions prouvé le bon fonctionnement pour un cluster de 1 master node et 2 worker nodes distribués).

- Possibilité d'accéder aux pods depuis n'importe quelle machine cliente connectée au réseau local.

- Utilisation d'un serveur NFS sur une machine indépendante comme moyen de partage des fichiers nécessaires au déroulement des TPs, où les enseignants accèdent aux volumes dédiés aux modules afin de mettre ou récupérer des fichiers, pareil pour les étudiants qui peuvent facilement accéder aux ressources et envoyer leurs travaux.

- Démonstration du déploiement d'une image docker sur kubernetes.

- Déroulement complet pour l'application Packet Tracer dans le cas de 2 binomes.

Mots Clés : Kubernetes, cluster, MERN, orchestration

Table des matières

Table des figures

Liste des tableaux

1. Introduction

Docker est une plate-forme de conteneurisation largement connue utilisée pour développer, déployer et exécuter n'importe quelle application en tant que conteneur portable et autonome. Comme nous l'avons expliqué dans l'étape précédente de ce projet, Docker fournit une solution open source pour emballer et distribuer des applications conteneurisées (notamment DockerHub), les développeurs peuvent rechercher dans ces registres de conteneurs et extraire des images existantes ce qui leurs permet d'économiser beaucoup de temps et d'améliorer la productivité. Cependant, à mesure que le nombre de conteneurs augmente, la complexité de leur gestion augmente également. Avec cette solution, la tâche des enseignants responsables des séances TPs deviendra fastidieuse, qui implique d'assurer :

— La communication entre les utilisateurs et les conteneurs.

— La gestion de plusieurs utilisateurs simultanément.

— La scalabilité de nombreuses instances de conteneurs.

— La planification du déploiement des applications.

Kubernetes est une technologie d'orchestration développée essentiellement pour pour faire face à la complexité de garder un nombre conséquent de conteneurs actifs, au point d'automatiser la création et l'intégration d'un nouveau conteneur en cas de crash. De plus, Kubernetes inclut de nombreuses fonctionnalités avantageuses lors de la gestion et l'orchestration des conteneurs, telles que l'équilibrage de charge (load-balancing), la sécurité, la mise en réseau, un mécanisme d'isolation intégré, l'auto-réparation et la possibilité de la mise à l'échelle sur tous les nœuds qui s'exécutent sur les conteneurs construits. On peut déduire à ce stade que Docker et Kubernetes sont en effet des technologies complémentaires. Même si les deux ont des rôles similaires, ils sont en fait très différents et peuvent parfaitement fonctionner ensemble. Comme il est désormais clair, Docker est l'une des principales technologies sous-jacentes de Kubernetes, qui à son tour est capable de gérer intégralement les conteneurs instanciés par Docker via un plan de contrôle. Dans cette dernière étape du projet, nous expliquerons le processus complet de l'implémentation d'une architecture logique complète d'un cluster Kubernetes, proposée par notre équipe après avoir taclé les problèmes techniques les plus pertinents avec Docker, ensuite nous mettrons en évidence la manière dont les acteurs interagissent avec le système en effectuant une série de tests qui simulent un cas réel d'un déroulement d'une séance TP à l'ESI.

2. Problématique

Ayant réussi à conteneuriser les applications en utilisant Docker, cette technologie a permis de créer un environnement d'exécution léger pouvant être exécuté sur différents types de machines et de systèmes d'exploitation, tout en étant facilement duplicable et partageable.

La prochaine étape de l'implémentation consiste à concevoir un cluster Kubernetes offrant une grande flexibilité, de bonnes performances et évolutivité qui soit compatible avec les cas d'utilisation des applications choisies.

Afin d'être en mesure de trouver la solution la plus appropriée pour notre système, ces points seront à étudier :

— Les différents composants de Kubernetes

— Les différents types qu'ils peuvent avoir et la différence entre ces derniers.

— Le fonctionnement et la méthode pour implémenter ces composants.

Pour pouvoir décider :

— Quels composants feront partie de la solution ?

— Comment ces composants sont-ils utilisés ensemble ?

— Et comment vont-ils être utilisés, dans le contexte de la solution, pour résoudre les problèmes liés à nos systèmes tout en répondant aux exigences de performances et de sécurité ?

3. Solution de la problématique

3.1 Architecture logique

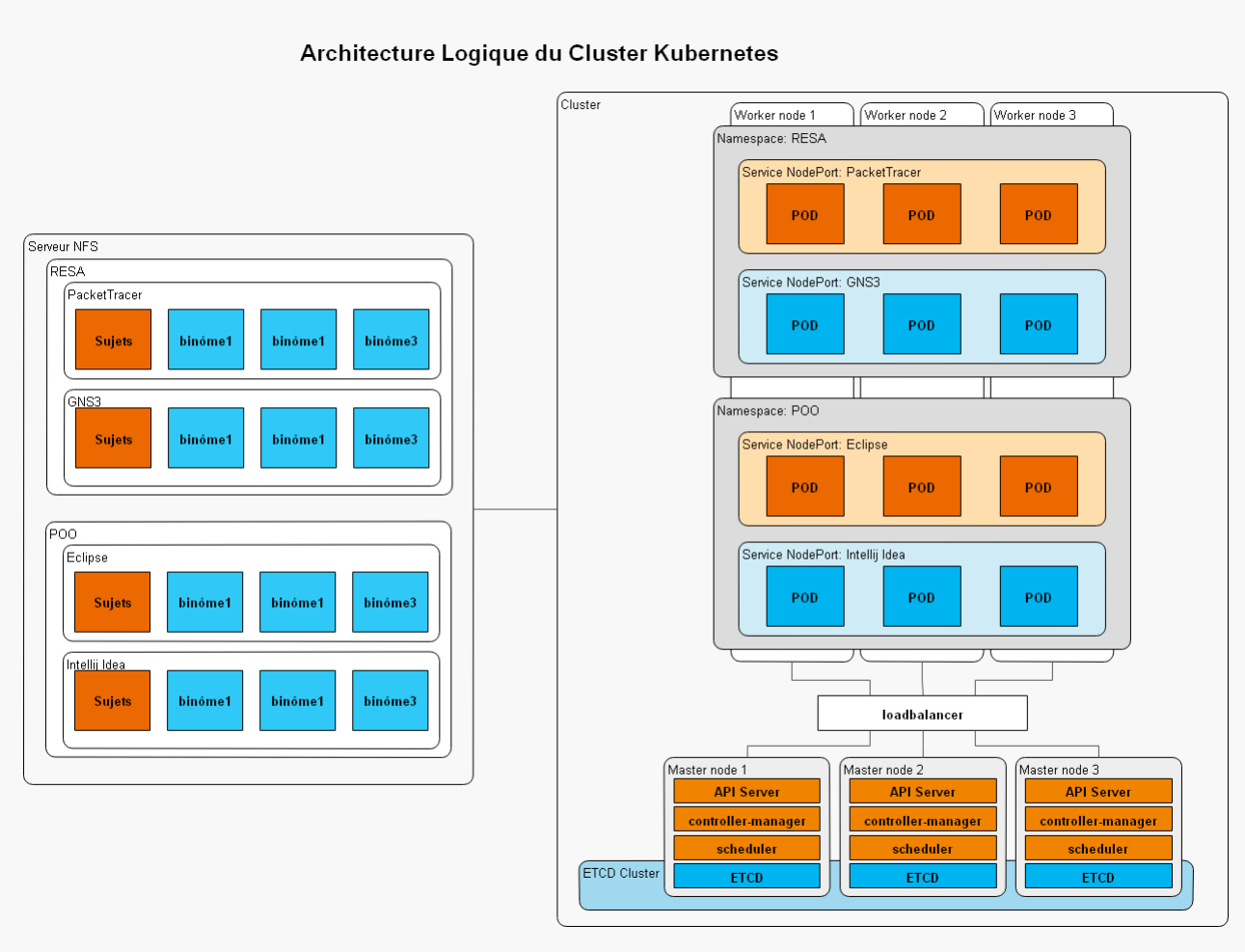


FIGURE 3.1 – Diagramme global de l"architecture logique

3.1.1 Explication de l'architecture

3.1.1.1 Pods et Déploiement

1. **Pods :** c'est une encapsulation d'un ou plusieurs conteneurs, qui partagent un contexte, un espace de stockage, et une identité réseau et qui sont ordonnancés et exécutés de la

même manière, et sur la même machine physique ou virtuelle.
Le Pod représente l'unité de déploiement la plus petite sur un cluster Kubernetes. Il est éphémère, c'est-à-dire qu'un pod n'est pas persistant, et qu'il a une courte durée de vie.

De point de vue Kubernetes, un Pod est un processus qui s'exécute sur le Cluster.

- Ce processus possède une identité réseau : adresse IP et numéro de Port afin de communiquer avec les autres Pods et services.
- Ce processus est ordonnancé en l'associant à un nœud de cluster selon la disponibilité des ressources. C'est-à-dire qu'un Pod peut se retrouver sur n'importe quel nœud travailleur dans le cluster.
- Ce processus n'est pas persistant, et n'a, par défaut, pas d'espace de stockage persistant qui lui est associé.

Utilisation des Pods dans notre solution : Chaque image créée dans la première partie de ce projet sera déployée dans un Pod, et chaque Pod sera attribué à un binôme. Le pod, contiendra donc, une instance d'une application conteneurisé déployée au cours d'exécution qui sera exposé afin de permettre son utilisation par un binôme.

2. **Déploiement :** Le déploiement est une unité logique persistante considérée comme un blueprint ou un template d'un ou plusieurs pods. C'est une description déclarative [1] utilisée par kubernetes pour créer des pods qui répondent à un état désiré, ou remplacer des Pods existants par d'autres Pod a description différente.
Le déploiement est un objet kubernetes contenant des informations utilisées par Kubernetes pour la création, le remplacement ou la mise à jour des Poids, ces informations sont décrites par un fichier YAML [2]

Pourquoi utiliser un déploiement ? Les Pods sont des unités éphémères et dynamiques, et sur un cluster Kubernetes il peut y avoir un très grand nombre de Pods. La gestion des Pods d'une manière manuelle devient très difficile et impratique. Le déploiement est une abstraction des Pods qui permet de modifier un ensemble des Pods identiques d'une manière simple et automatisée.
Contenu de fichier YAML de déploiement : Un fichier de déploiement débute par la spécification de la version API de kubernetes utilisée, suivi par le type de fichier (un fichier de déploiement), s'en suit deux parties importantes :

- **Partie "Metadata" :** c'est la partie qui contient les champs de données permettant d'identifier un déploiement de manière unique. Elle contient le nom de déploiement, le namespace auquel il appartient, et les labels donnés au déploiement.

1. Kubernetes est un système déclaratif, où un état désiré est décrit et c'est à la responsabilité du système de le réaliser. Contrairement au système impératif où l'on fournit un algorithme explicite censé nous amener à l'état désiré .
2. YAML : abréviation de Yet Another Marketing Language est un langage de sérialisation de données lisible et convivial, souvent utilisée pour les fichiers de configuration

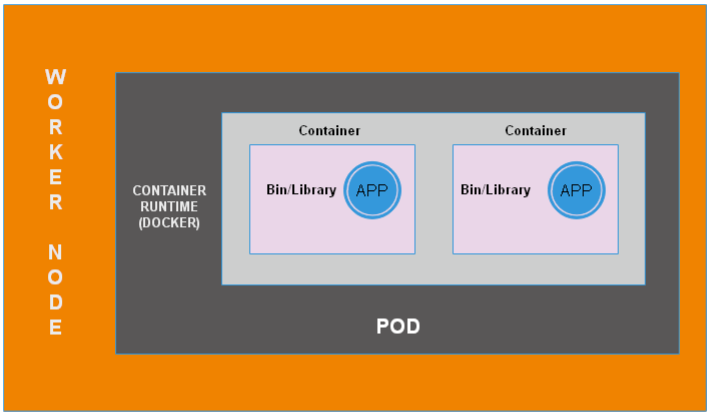
Label : dans kubernetes un label est un pair clé-valeur attachée à des objets Kubernetes, il sont utilisés pour donner des attributs "personnalisés" indépendant de sémantique de Kubernetes et définis par l'utilisateur de Cluster au ces objets)

Label selector : dans kubernetes un Label Select est utilisé pour identifier et regrouper les objets Kubernetes ayant un label particulier.

— **Partie "spec" ou spécifications :** c'est la partie utilisée pour décrire l'état désiré du déploiement.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: <NOM-DU-DEPLOYMENT>
  namespace: <NAMESPACE-DU-DEPLOYMENT>
  labels:
    app: <VALEUR-LABEL>
spec:
  replicas: <NOMBRE-DE-PODS>
  selector:
    matchLabels:
      app: <VALEUR-LABEL>
  template:
    metadata:
      labels:
        app: <VALEUR-LABEL>
    spec:
      volumes:
      - name: <NOM-DE-VOLUME>
        persistentVolumeClaim:
          claimName: <NOM-DU-PERSISTANTVOLUMECLAIM>
      containers:
      - name: <NOME-DE-CONTENEUR>
        image: <hub-user>/<repo-name>:<tag>
        command: ["<COMMANDE>"]
        args: ["<ARGUMENTS>"]
        ports:
        - containerPort: <NUMÉRO-DE-PORT>
        volumeMounts:
        - mountPath: "<CHEMIN-DU-MONTAGE-DE-VOLUME>"
          name: <NOM-DU-VOLUME>
```

FIGURE 3.2 – Fichier YAML



Déploiement d'un Worker Node

FIGURE 3.3 – diagramme de déploiement

3.1.1.2 Services

1- Qu'est-ce qu'un service : C'est une unité abstraite statique ayant une identité réseau invariante, qui permet d'exposer une application s'exécutant sur un ensemble des Pods sur le cluster. Un service définit un ensemble logique de Pods et la façon d'y accéder. Elle utilise les sélecteurs de labels pour identifier et localiser les Pod visés.
Il existe 4 types de service dans Kubernetes :

- ClusterIP : Pour exposer les services à l'intérieur du Cluster.
- NodePort : Pour exposer les services à l'extérieur via un port statique sur l'IP de chaque nœud.
- LoadBalancer : Pour exposer les services à travers un load balancer extérieur d'un fournisseur de cloud.
- ExternalName : Pour exposer les services en associant un enregistrement CNAME à chaque service.
  Enregistrement CNAME (nom canonique) : est un type d'enregistrement ressource dans le DNS(Domain Name System) qui permet d'associer un alias à un nom de domaine.

Pourquoi l'utiliser ? Les pods sont dynamiques et éphémères, un pod peut tomber en panne et être remplacé par un autre et dans ce cas l'adresse IP de nouveau ne sera, dans la plupart des cas, pas la même. Pour pouvoir exposer et accéder à une application d'une manière sûre et consistante, une adresse IP statique est nécessaire.

Le service NodePort : Comme mentionné précédemment, les services NodePort exposent les Pod à l'extérieur via des port statiques. Le choix s'est porté pour ce type de service dans ce projet car il permet d'exposer un certain Pod à un binôme, afin qu'il ne soit accessible que par eux.

Extensibilité : la plage de ports à utiliser pour l'allocation NodePort dans Kubernetes va de **30000 à 32767** ce qui donne 2767 différents numéros de Port exploitables pour exposer les Pods.

Dans notre école il y a en moyenne 250 étudiants par promotion ce qui donne environ **125 binômes** par module et par promotion. Et **500 binômes** pour les 4 promos confondues. Ce qui permet à chaque binôme d'accéder **à cinq applications.**

Fichier YAML : Comme pour les autres objets Kubernetes, un service est décrit via un fichier YAML.
On présente ci-dessous, le contenue de fichier YAML d'un service NodePort :

```
apiVersion: v1
kind: Service
metadata:
  name: <NOM-DE-SERVICE>
  namespace: <NAMESPACE-DE-SERVICE>
spec:
  selector:
    app: <VALEUR-LABEL>
  type: NodePort
  ports:
    - protocol: <PROTOCOL>
      port: <PORT-DE-CONTENEUR>
      targetPort: <PORT-DE-CONTENEUR>
      nodePort: <NUMERO-DE-PORT-STATIQUE>
```

FIGURE 3.4 – Service NodePot dans un fichier YAML

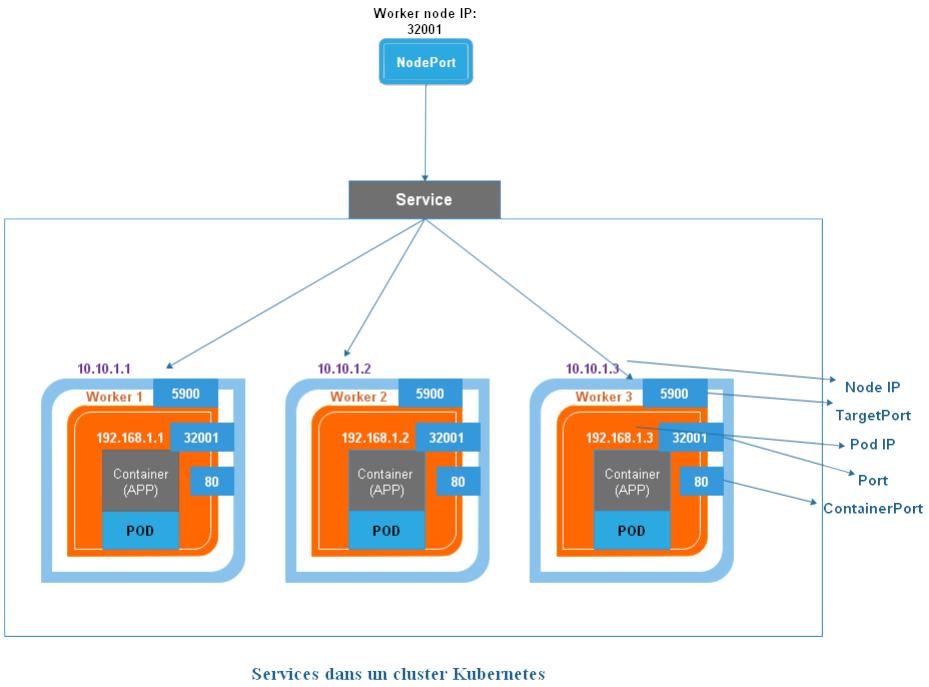


FIGURE 3.5 – diagramme de Service

3.1.1.3 Namespaces

Qu'est-ce qu'un namespace : Les namespace (aussi appelés groupes de noms) sont des unités logiques isolées qui simulent des sous-clusters virtuels à l'intérieur du cluster Kubernetes.

Ils fournissent un modèle qui permet :D'isoler et d'organiser les ressources utilisées dans des contextes différents, ainsi les noms des ressources doivent être uniques dans un namespace, mais pas dans l'ensemble des namespaces. Contrôler l'accès aux ressources et limiter les ressources utilisées par un namespace particulier.

Les nœuds et les volumes persistants sont les seules ressources qui ne peuvent être associées à des namespaces particuliers.

Pourquoi l'utiliser ? Les namespace sont utilisés pour organiser les services au sein du cluster en les regroupant selon des équipes ou des projets, et répartir les ressources d'un cluster entre plusieurs utilisateurs. L'utilisation de namespaces :

— Permet à différentes équipes d'utiliser le même Cluster physique sans impacter le travail des autres.

— Offre une meilleure sécurité et performances quant à l'accès aux namespaces par les utilisateurs.

— Facilite la gestion et la maintenance du Cluster parce qu'elle organise l'ensemble des objets Kubernetes en des sous-ensemble isolés.

Utilisation des namespaces dans le cadre du projet : Dans ce cas, les namespaces serviront à séparer les objets kubernetes relatifs aux applications utilisées dans les différents modules, par exemple : le module réseaux aura un namespace "RESA "et le module système d'exploitation et Linux "SYS1"

Comment créer les namespaces : Il y a par défaut 4 namespaces créés par Kubernetes by default :

— default : utilisé par défaut par kubernetes pour tous les objets créés sans spécifier un namespace personnalisé.

— kube-public : utilisé pour les services et les ressources publiques qui ne nécessite pas d'authentification

— kube-system : utilisé pour les objets créés par le système de Kubernetes

— kube-node-lease : utilisé pour les objets appelé "lease objects" qui sont utilisés pour surveiller la santé des noeuds travailleur du cluster.

Pour créer notre propre namespace en peut : 1- Utiliser la commande :
kubectl create namespace <NOM-DU-NAMESPACE>
2- Créer un fichier YAML qui decrit le namespace :

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: <NOM-DU-NAMESPACE>
```

FIGURE 3.6 – Fichier YAML qui décrit les namespaces

Et créer le namespace en utilisant la commande :
kubectl create -f CHEMIN-VERS-LE-FICHIER-YAML

Après avoir créé le namespace, la création des objets Kubernetes y appartienant se fait par deux méthodes :
1 - En spécifiant le namespace lors de la création d'objets en utilisant kubectl :
kubectl apply -f FICHIER-YAML-DE-OBJECT –namespace=test
2- Ou en spécifiant le namespace dans le fichier YAML sous le champs "metadata" :

```yaml
metadata:
  name: <NOM-DE-OBJECT>
  namespace: <NOM-DE-NAMESPACE>
```

FIGURE 3.7 – Spécification des métadata

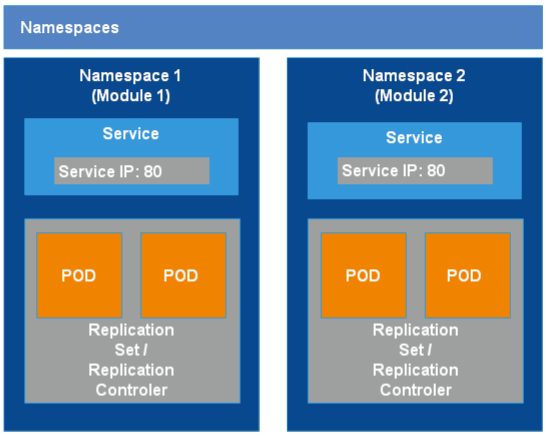


FIGURE 3.8 – diagramme des Namespaces

3.1.1.4 Volumes

Le stockage dans kubernetes est basé sur le concept de Volumes . Comme les pods ont besoin de données afin d'assurer l'exécution, l'idée est de séparer les volumes des pod afin de pouvoir restaurer leur état en cas de panne, la seconde raison d'opter pour les volumes est de permettre le partage des données inter-pods.
Un volume kubernetes est un répertoire qui peut être accessible aux conteneurs d'un même pod, c'est également un moyen de connecter les pods -qui sont par définition éphémères- à des données persistantes.
Nous distinguons deux types de volumes :

— Les volumes éphémères qui sont communément appelé "volumes" ayant la même durée de vie qu'un pod, soit éphémères.

— Les volumes persistants (Persistant volumes) qui sont sauvegardés même si le pod lié est détruit.

Les volumes persistants(PV) sont des ressources du cluster, tout comme la RAM ou le CPU provisionnées par l'administration du cluster. Ces PV sont consommés par les utilisateurs à travers ce que l'on appelle persistent volume claim (PVC), ce qui leur permet de réserver un espace mémoire en spécifiant la taille et le mode d'accès vers celui-ci.

La Création de Volumes dans Kubernetes : Comme toute ressource kubernetes, un PV est créé à partir d'un fichier yaml, où l'on peut spécifier l'espace utilisé et le mode d'accès au volume. Étant donné que le PV est un composant abstrait du cluster Kubernetes, l'associer à un espace physique tel que le disque local ou bien un serveur NFS ou en cloud est une pratique nécessaire. Le PV est comparable à une extension externe du Cluster, Les applications peuvent utiliser des stockages différents de plusieurs types selon la nécessité.

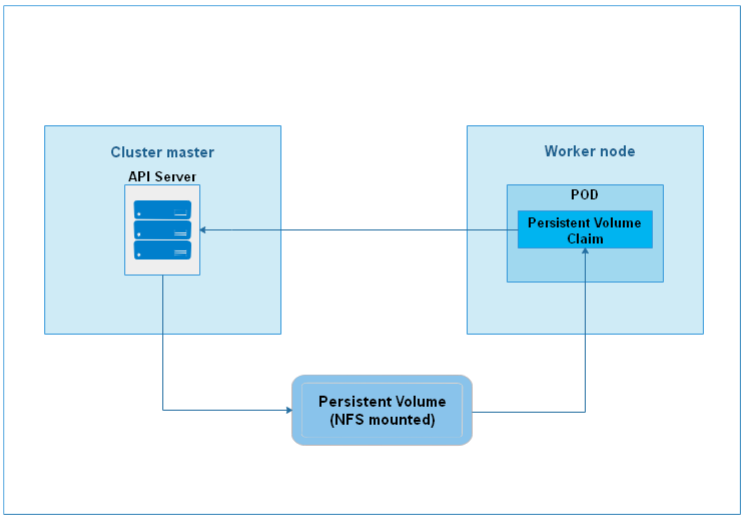
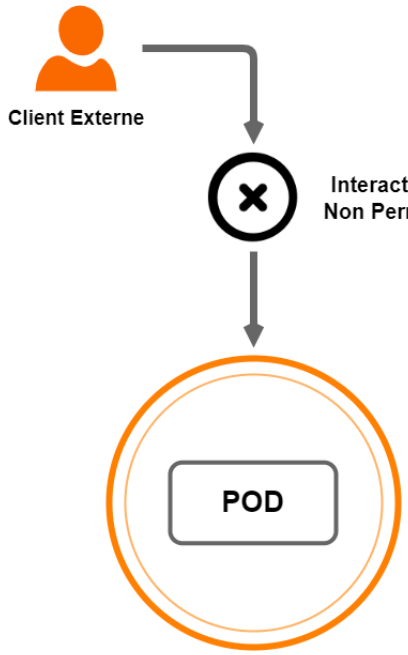
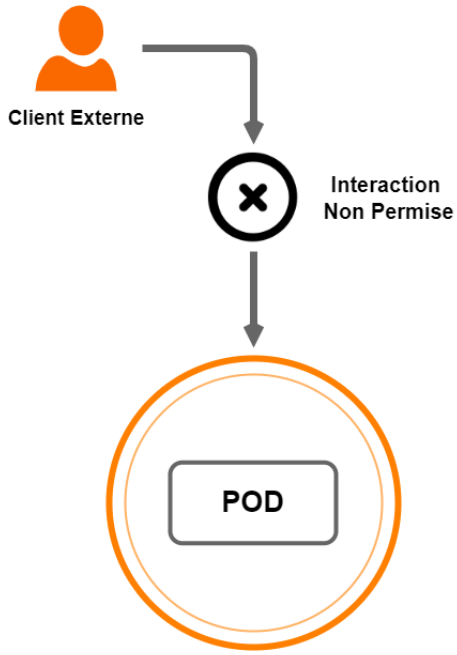


FIGURE 3.9 – diagramme des volumes

3.1.1.5 Accès



(a) Intéraction non-permise



(b) Permission d'accès

Le logiciel Virtual Network Computing (VNC) permet de réduire la charge de travail de X11 sur les réseaux à forte latence tels qu'Internet. En termes pratiques, une fois qu'une session VNC est en cours, les latences sont de l'ordre de quelques secondes plutôt que de quelques minutes. VNC peut rendre les applications X11 distantes utiles au lieu d'être fastidieuses et non productives.
Le principe de fonctionnement implique un processus de serveur hôte (par exemple, Xvnc) qui communique avec les applications X11 exécutées dans les conteneurs. Le processus du serveur hôte transmet des images et des mises à jour d'images au client de visualisation du système distant en utilisant un protocole à faible coût.
Pourquoi opter pour une solution avec VNC au lieu de Xserver : Les commandes X11 sont redirigées au Xserver local. C'est donc comme si on exécutait le programme localement, alors qu'il est en réalité exécuté sur l'ordinateur à l'autre bout. C'est très lent car cela utilise une grande quantité de bande passante. (On dira que X11 est "transparent au réseau").
Contrairement à cela, VNC et d'autres applications de bureau à distance laissent l'ordinateur client traiter tous les dessins graphiques, etc. et transmettent, en fait, une capture d'écran à votre ordinateur. Cela peut sembler beaucoup plus rapide, car beaucoup moins d'informations sont nécessaires pour tout afficher. Cependant, il envoie également le bureau entier, plutôt qu'une seule application.
X11 a besoin d'envoyer plusieurs petits paquets mais en plusieurs aller-retour sur le réseau, tandis que VNC fait tout l'inverse, il envoie moins de paquets dont la taille est plus importante.
Étapes pour configurer VNC et visualiser l'affichage du conteneur :

Pour notre démonstration, nous utiliserons l'image docker de base de l'application Packet tracer et Bash et le modèle d'objet de page publique Packet tracer et Bash dont nous avons parlé dans le cadre de notre précédent post. Voici les étapes nécessaires à la mise en place de VNC.

Configurer et activer le serveur VNC sur le conteneur.

Configurez le visualiseur VNC sur votre système local.

Connectez-vous au serveur VNC en utilisant l'adresse IP et le port.

Pour configurer et activer le serveur VNC, nous devons On ajoute ce qui suit au niveau du dockerfile
a) Installer le serveur VNC : `RUN pip install --upgrade pip && apt-get update` apt-get install -y git x11vnc
b) Exposer le port pour se connecter et partager les données d'affichage avec un VNC Viewer. Expose port 5920 to view display using VNC Viewer EXPOSE 5920
c) Activer le serveur VNC en indiquant le mot de passe et les arguments d'affichage.

Pour ce projet, nous allons utiliser un fichier Docker et un fichier script shell. Cela nous permettra de cloner notre modèle d'objet de page publique Packet tracer / bash dans le conteneur en utilisant git, installer le serveur VNC, exposer le port, activer VNC et exécuter le test.
NOTE : Dans notre cas, nous utiliserons git pour obtenir un code source ouvert dans le conteneur. Idéalement, nous préférons utiliser la fonction de volume de Docker pour obtenir le code à l'intérieur du conteneur. il faudra du moins éviter git si vous optez pour un dépôt privé pour des raisons évidentes de sécurité.

Configurer le visualiseur VNC sur votre machine locale

Le serveur VNC envoie/transmet des informations sur les données d'affichage sur un port spécifié. Pour capturer et lire ces données, nous devons installer le visualiseur VNC qui agit comme un client. VNC viewer est disponible sur plusieurs plateformes, on peut télécharger n'importe quel client VNC pour se connecter au serveur VNC en utilisant l'adresse IP et le port correspondants.
Pour connecter VNC viewer au serveur VNC, vous devez connaître l'adresse IP et l'adresse du port. docker-machine ip docker run -it -p 5920 :5920 image-name-which-provided-in-step-1

La commande ci-dessus connecte/mappe le port 5920 de l'hôte local au port 5920 du conteneur et pour voir les tests en cours sur l'écran, vous devez vous connecter rapidement au serveur VNC car une fois le test terminé, il ne vous montrera qu'un écran noir.

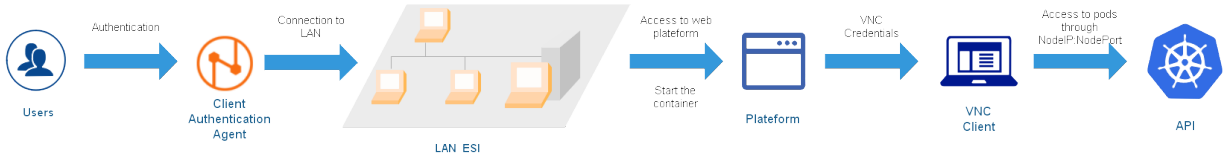


FIGURE 3.11 – Diagramme d'authentification et accès

3.2 Architecture physique

3.2.1 Master Nodes

Un nœud est une machine de travail dans un cluster Kubernetes, connue auparavant sous le nom de minion. Un nœud peut être une machine virtuelle ou une machine physique, selon le cluster. Chaque nœud contient les services nécessaires à l'exécution de pods et est géré par les composants du nœud master. Les services dans un nœud incluent le container runtime, kubelet et kube-proxy.

Un cluster Kubernetes se compose généralement de deux types de nœuds, chacun responsable de différents aspects niveau fonctionnalités :

— **Nœuds maîtres (Master Nodes)**– Ces nœuds hébergent les aspects du plan de contrôle du cluster et sont responsables, entre autres, du point de terminaison API avec lequel les utilisateurs interagissent et fournissent la planification des pods à travers les ressources. Généralement, ces nœuds ne sont pas utilisés pour planifier les charges de travail des applications.

— **Nœuds de calcul (Worker Nodes)**– Nœuds responsables de l'exécution des charges de travail pour les utilisateurs du cluster.

Les nœuds maîtres sont essentiels au fonctionnement du cluster. Si aucun maître n'est en cours d'exécution ou si les nœuds maîtres ne parviennent pas à atteindre un quorum, le cluster est incapable de planifier et d'exécuter des applications. Les nœuds maîtres constituent le plan de contrôle du cluster et, par conséquent, une attention particulière doit être accordée à leur dimensionnement et à leur quantité.

Architecture avec trois noeuds maîtres dans le cluster : Le fait d'avoir plusieurs nœuds maîtres garantit la disponibilité des services en cas de défaillance d'un nœud maître dont dépend tout le reste du cluster (éviter le phénomène du **Single Spot Of Failure**). Afin de faciliter la disponibilité des services maîtres, ils doivent être déployés avec des nombres impairs (par exemple 3) afin que le quorum (majorité du nœud maître) puisse être maintenu en cas de défaillance d'un ou plusieurs maîtres. Dans ce scénario, Kubernetes conservera une copie des bases de données **etcd** sur chaque maître, mais organisera des élections pour les responsables de la fonction de plan de contrôle **kube-controller-manager** et **kube-scheduler** afin d'éviter les conflits. Les nœuds de travail peuvent communiquer avec n'importe quel serveur d'API maître via un équilibreur de charge (Load Balancer).

Le déploiement de plusieurs maîtres sur des serveurs physiques différents est la configuration minimale recommandée pour la plupart des clusters de production.

Pourtant cette stratégie implique une certaine complexité dans la mise en place initiale, on a opté pour cette solution puisqu'elle fournit des services maîtres hautement disponibles, garantissant que la perte de jusqu'à $\mathbf{(n/2) - 1}$ nœuds maîtres n'affectera pas les opérations du cluster ce qui reste notre priorité absolue.

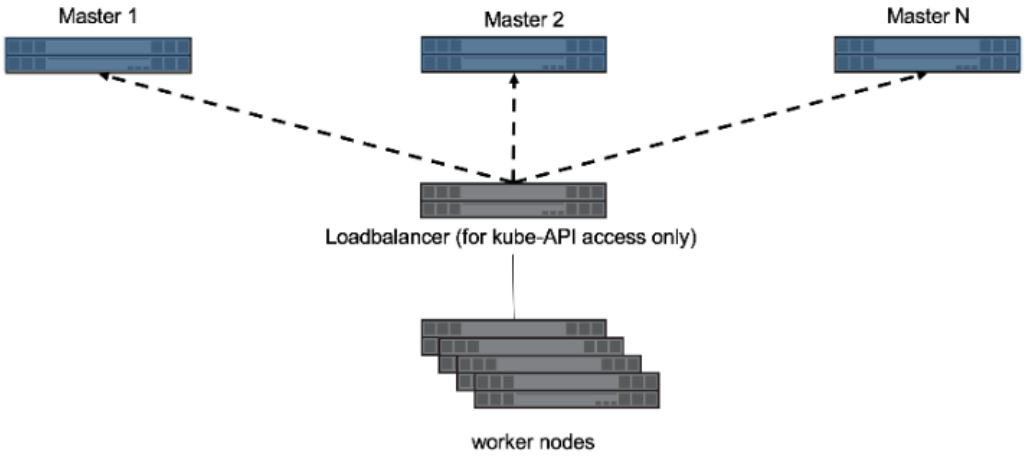


FIGURE 3.12 – Architecture Kubernetes à plusieurs noeuds maîtres

3.2.2 Worker Nodes

Un nœud est une machine de travail dans Kubernetes et peut être une machine virtuelle ou physique, selon le cluster. Chaque nœud est géré par le plan de contrôle. Un nœud peut avoir plusieurs pods, et le plan de contrôle Kubernetes gère automatiquement la planification des pods sur les nœuds du cluster. La planification automatique tient compte des ressources disponibles sur chaque nœud.

Chaque nœud Kubernetes exécute au moins :

— **Kubelet**, un processus responsable de la communication entre le plan de contrôle Kubernetes et le nœud ; il gère les Pods et les conteneurs s'exécutant sur la machine.

— **Un environnement d'exécution de conteneur** (comme Docker) chargé d'extraire l'image du conteneur d'un registre, de décompresser le conteneur et d'exécuter l'application.

Architecture avec un nombre minimal de noeuds de calcul dans le cluster : Il n'y a pas de solution unique quant au nombre de nœuds devant constituer un cluster Kubernetes. En effet, ce nombre varie en fonction des exigences spécifiques de la charge de travail. Cependant, il est important d'équilibrer le nombre de nœuds avec les objectifs de coût. Le tout est de déterminer le nombre de nœuds nécessaires pour répondre aux exigences de performances et de disponibilité, sans configurer de nœuds superflus qui engendrent des coûts inutiles.

C'est pour cela qu'on propose une mise en place initiale avec un nombre minimal de nœuds de calcul (3 pas exemple). Exécuter la même charge de travail sur moins de nœuds signifie naturellement que plus de pods s'exécutent sur chaque nœud. Cela pourrait devenir un problème étant donné que chaque pod introduit une surcharge sur les agents Kubernetes s'exécutant sur le nœud, tels que l'environnement d'exécution de conteneurs (Docker dans notre cas), le kubelet et cAdvisor. Néanmoins, on a opté pour une telle stratégie comme configuration initiale pour ne pas surcharger les nœuds maîtres du cluster en réduisant le nombre de communications entre les nœuds qui croient au carré du nombre de nœuds et donc permettre une meilleure performance. Selon le type d'applications déployées dans le cluster (gourmandes en ressources ou pas) ainsi que la montée en charge, les responsables techniques peuvent décider d'ajouter ou pas des nœuds de calcul en tenant compte des besoins relatifs aux TPs.

Pour déterminer si les nœuds sont basés sur des serveurs physiques dédiés, des machines virtuelles ou une combinaison des deux, il faut tenir compte du fait que :

— Les nœuds basés sur des machines virtuelles pourraient laisser le cluster plus à risque de tomber en panne. Par exemple, si plusieurs nœuds s'exécutent en tant que VM, mais que toutes ces VM sont hébergées sur le même serveur physique, la défaillance de ce dernier rendrait tous les premiers indisponibles et réduirait considérablement le nombre total de nœuds d'un seul coup.

— En revanche, un serveur physique dédié pour chaque nœud réduit la probabilité que plusieurs nœuds échouent simultanément. Mais il est également plus coûteux, dans la plupart des cas, d'exécuter tous les nœuds en bare metal.

Par conséquent, la meilleure approche consiste souvent à utiliser **une combinaison de machines physiques et virtuelles** pour les nœuds de calcul Kubernetes.

3.2.3 Serveur NFS :

L'un des types de volumes les plus utiles dans Kubernetes est NFS(network file system). Un serveur NFS doit exister avec les capacités de stockage suivants :

— un minimum de 50Go (pour une classe de 15 binôme qui auront au moyen 5 module avec des TPs et 10 TPs par semestre, en supposant que la taille d'un TP est 10Mo)

— il est recommandé que les disque utilisés dans le serveur NFS soient des disque SSD pour que le temps d'accès au fichier des TP soit minimal

il est recommandé que les disque utilisés dans le serveur NFS soient des disque SSD pour que le temps d'accès au fichier des TP soit minimal Nous devons créer un dossier à partager pour chaque application et dans ce dossier nous créons un sous dossier pour mettre les énoncés de TPs ou les étudiants n'ont pas le droit d'écriture pour ne pas modifier l'énoncé.

Un NFS est utile pour deux raisons :

1. Premièrement, ce qui est déjà stocké dans le NFS n'est pas supprimé lorsqu'un pod est détruit. Les données sont persistantes.
2. Deuxièmement, un NFS est accessible à partir de plusieurs pods en même temps. Un NFS peut être utilisé pour partager des données entre les pods .

4. Implémentation de la solution

4.1 La création du cluster :

Le principe de Kubernetes est de coordonner un cluster hautement disponible d'ordinateurs connectés pour travailler comme une seule unité.

les étapes à suivre pour la création de celui-ci :

— Installer la plateforme de conteneurisation sur tous les nœuds "Docker".

— Installer les outils Kubeadm, Kubelet et kubectl sur tous les nœuds.

— Lancer la configuration du plan de contrôle Kubeadm sur les nœuds maîtres.

— Enregistrer la commande node join avec le jeton.

— Installer le plugin réseau.

— Joindre le nœud de travail au nœud maître (plan de contrôle) à l'aide de la commande join.

— Déployer un serveur NFS pour le stockage des fichiers de TPs.

4.1.1 installer docker container runtime :

— installation des packages de dépendances de docker :

```
$sudo apt-get update -y
$sudo apt-get install -y \
apt-transport-https \
ca-certificates \
curl \
gnupg \
lsb-release
```

— Ajout de la clé GPG de docker et du répertoire apt :

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

$echo \
"deb [arch=amd64 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
```

— Installation de docker :

```
$sudo apt-get update -y
$sudo apt-get install docker-ce docker-ce-cli containerd.io -y
```

— Configuration du docker daemon :

```
$cat <<EOF | sudo tee /etc/docker/daemon.json
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2"
}
EOF
```

— Lancement du service docker :

```
$sudo systemctl enable docker
$sudo systemctl daemon-reload
$sudo systemctl restart docker
```

4.1.2 Installation du kubeadm,kubelet et kubectl :

— Installation des dépendances :

```
$sudo apt-get update
$sudo apt-get install -y apt-transport-https ca-certificates curl
$sudo curl -fsSLo /usr/share/keyrings/kubernetes-archive-keyring.gpg
https://packages.cloud.google.com/apt/doc/apt-key.gpg
```

— Ajout de la clé GPG et du répertoire apt :

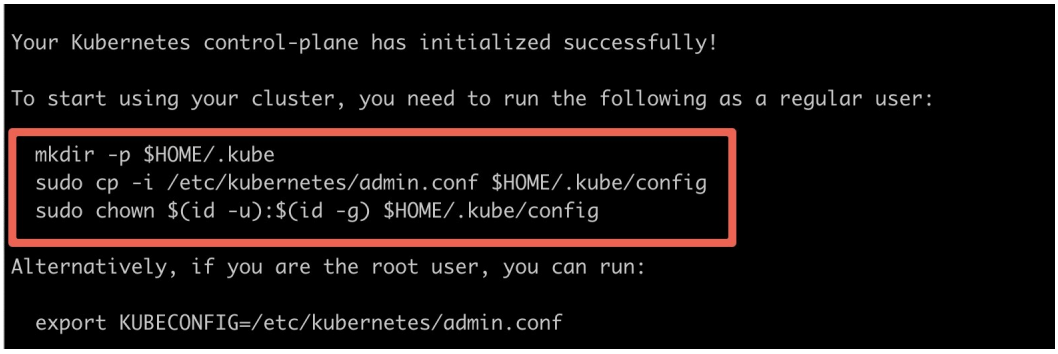
```
$echo "deb [signed-by=/usr/share/keyrings/kubernetes-archive-keyring.gpg]
https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee
/etc/apt/sources.list.d/kubernetes.list
```

— Mise à jour et installation du kubeadm,kubelet et kubectl :

```
$sudo apt-get update -y
$sudo apt-get install -y kubelet kubeadm kubectl
```

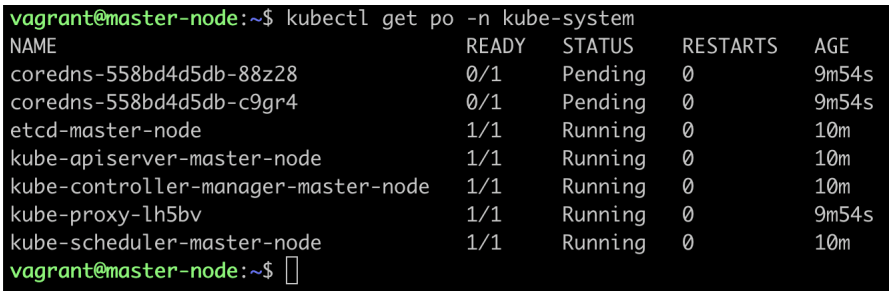
4.1.3 Configuration du plan de contrôle Kubeadm sur le noeud master :

```
$sudo kubeadm init --apiserver-advertise-address=$IPADDR
--apiserver-cert-extra-sans=$IPADDR  --pod-network-cidr=192.168.0.0/16
--node-name master
```

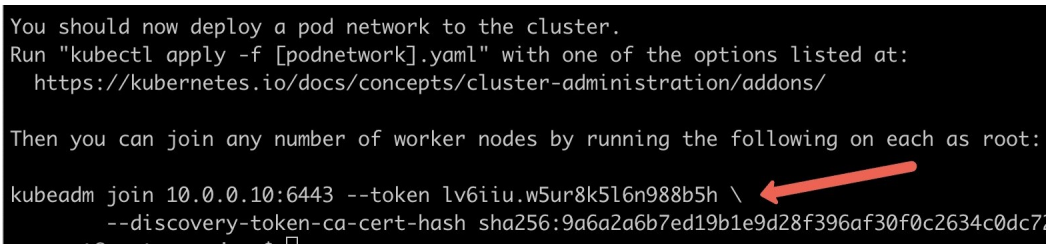


4.1.4 Vérification des pods du noeud master :

```
$kubectl get pods -n kube-system
```



4.1.5 Enregistrement de la commande permettant de joindre le cluster :

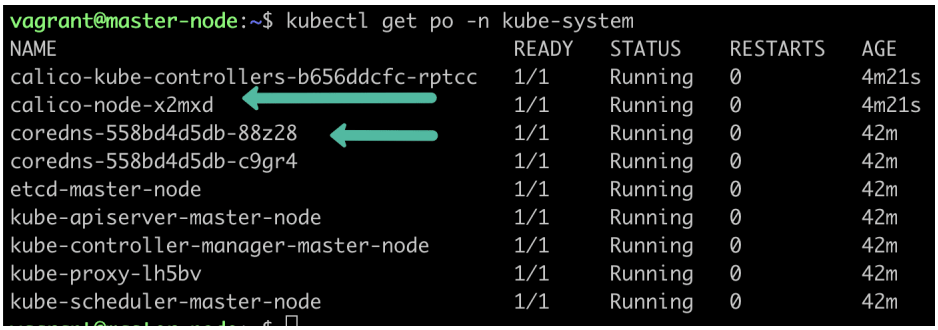


Nous devons enregistrer la commande kubeadm join avec le token et le certificat pour pouvoir joindre les nœuds workers au cluster.

4.1.6 Installation du plugin réseau :

Kubeadm ne dispose d'aucun plugin réseau intégré, nous devons donc l'installer séparément. Nous avons utilisé le plugin Calico car il fournit des solutions réseau sécurisées pour les conteneurs. Il est également connu pour ses bonnes performances, sa flexibilité et sa puissance.

```
$kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
```




```
$sudo kubeadm join 10.128.0.37:6443 --token j4eice.33vgvgyf5cxw4u8i
--discovery-token-ca-cert-hash
sha256:37f94469b58bcc8f26a4aa44441fb17196a585b37288f85e22475b00c36f1c61
```

4.1.7 Joindre les noeuds au cluster :

```
$sudo kubeadm join 10.128.0.37:6443 --token j4eice.33vgvgyf5cxw4u8i
--discovery-token-ca-cert-hash
sha256:37f94469b58bcc8f26a4aa44441fb17196a585b37288f85e22475b00c36f1c61
```

4.1.8 Déploiement et configuration du serveur NFS :

— Installation des packages nécessaire :

```
$sudo apt update && apt -y upgrade
$sudo apt install -y nfs-server
```

— Création du dossier partagé :

```
$mkdir /data
$cat << EOF >> /srv/TPs 10.10.1.0/24(rw,no_subtree_check,no_root_squash)
EOF
```

— Démarrer le serveur nfs et exportation du dossier partagé :

```
$systemctl enable --now nfs-server
$exportfs -ar
```

— Puis nous devons configuré les noeud workers pour pouvoir accéder au dossier partagé :

```
$sudo apt install -y nfs-common
```

4.2 Le déploiement des TPs :

les étapes à suivre :

— Créer l'image docker avec accès en utilisant VNC
— Créer le fichier de déploiement des PV et PVC (pour chaque module)
— Déployer les PV et PVC
— Créer le fichier de déploiement des pods (pour chaque binôme dans le module)
— Déployer les pods
— Créer le fichier de déploiement des service (pour chaque pod)
— Déployer les services
— Accéder au pods en utilisant l'adresse ip et le port spécifié pour chaque binôme

4.2.1 Création des images docker :

Les images utilisées disposent d'un serveur VNC intégré afin de pouvoir y accéder à travers un client VNC.

après la création des images avec succès nous devons les déposer dans docker hub afin de pouvoir les utiliser au niveau des déploiements.

4.2.2 Création des fichier de déploiement des PV et PVC :

4.2.2.1 Le déploiement des Persistent Volumes :

Les Persistent Volumes (PV) vont utiliser les répertoires exposés par le serveur NFS -dans notre implémentation le répertoire /srv/TPs-. Pour en créer un nous devons préciser : le nom du PV, la capacité de stockage, le mode d'accès, le nom de la classe de stockage qui sera "NFS" pour notre solution et on termine par le chemin du répertoire exposé et l'adresse IP du serveur NFS

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv-pt
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: nfs
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /srv/TPs
    server: 10.10.1.4
```

FIGURE 4.1 – Ficher YAML de PersistentVolume

Pour son déploiement nous devons utiliser la commande suivante :

```
$ kubectl apply -f ‘‘le chemin vers le fichier de déploiement"
```

4.2.2.2 Le déploiement du persistent volume claim (PVC) :

Les Persistent Volumes Claims (PVC) vont consommer les ressource PV. P leur création un nous devons préciser : le nom du PVC, la capacité de stockage, le mode d'accès, le nom de la classe de stockage qui sera "NFS" pour notre solution

Note : Elle doit être identique à celle utilisée dans le déploiement du Persistant Volume visé.

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pv-claim
spec:
  storageClassName: nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

FIGURE 4.2 – Ficher YAML de PersistentVolumeClaim

Pour le déployer nous devons utiliser la commande suivante :

```
$ kubectl apply -f ‘‘le chemin vers le fichier de déploiement"
```

4.2.2.3 Création des fichier de déploiement des pods(un pour chaque binôme) :

Pour notre solution un pod encapsule un seule conteneur, pour créer un nous devons spécifier : le nom du déploiement (un nom unique pour chaque binôme) , le nom de l'application, le nombre de réplicas, le nom du volume, le nom pvc associé au pod et on termine par les paramètre à passer pour créer le conteneur dans le pod qui sont : le nom du conteneur,l'image à utiliser, les commandes à exécuter dans le conteneur pour que chaque binôme ait son propre mot de passe pour accéder au pod en utilisant le client VNC, le port du conteneur et finalement le chemin du volume dans le conteneur avec son nom.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pt-deployment-b1
  namespace: tp-pt
  labels:
    app: packettracerb1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: packettracerb1
  template:
    metadata:
      labels:
        app: packettracerb1
    spec:
      volumes:
      - name: pt-pv-storage
        persistentVolumeClaim:
          claimName: nfs-pv-claim
      containers:
      - name: packettracerb1
        image: esialg/packettracer:1cs
        command: ["/bin/sh","-c"]
        args: ["x11vnc -storepasswd pwbinome1 ~/.vnc/passwd; x11vnc -create -usepw -forever"]
        ports:
        - containerPort: 5900
        volumeMounts:
        - mountPath: "/home/student/Tps"
          name: pt-pv-storage
```

FIGURE 4.3 – Ficher YAML de Deployment

Pour son déploiement nous devons utiliser la commande suivante :

```
$  kubectl apply -f ‘‘le chemin vers le fichier de déploiement"
```

4.2.2.4 Création des fichier de déploiement de services :

Chaque pod disposera de son propre service NodePort -et donc de sa propre valeur NodePort - pour l'exposer à l'extérieur du cluster. Pour en créer un, nous devons spécifier : le nom du déploiement (un nom unique pour chaque binôme), le nom de l'application associée à ce service -le nom donné dans le fichier de déploiement du pod-, le type de service -nous utilisons NodePort, finalement nous devons spécifier le protocole avec le targetPort (le port du conteneur) et la valeur du NodePort.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pt-service-b1
spec:
  selector:
    app: packettracerb1
  type: NodePort
  ports:
    - protocol: TCP
      port: 5900
      targetPort: 5900
      nodePort: 32000
```

FIGURE 4.4 – Ficher YAML de Service

4.2.2.5 Accès au conteneur après les phases d'authentification :

Pour accéder au conteneur, l'étudiant doit disposer d'identifiants VNC -qui lui seront communiqués au préalable- et les utiliser pour s'authentifier et ainsi pouvoir utiliser l'application du TP.

4.3 Tests :

Déroulement avec PacketTracer :

1. Authentification à partir de la plateforme web
2. Déployer deux pods dans le cluster
3. Mettre le fichier du TP au niveau du serveur NFS
4. Accéder au pods depuis deux machines clientes
5. Appliquer une modification
6. Enregistrer Le fichier
7. Vérifier la sauvegarde de la modification

4.3.1 Authentification à partir de la plateforme web :

1. Mise en place de la base de données en MangoDB Atlas avec le schéma suivant :
   - (_id, username, password, password_confirmation, port) où le numéro de port et le id sont affecté automatiquement par le backend - L'étudiant peut s'inscrire lui-meme dans un module, la validation de cette opération se fait par le bakcned en vérifiant s'il appartient au module d'abord et s'il n'avait pas encore fait une inscription à son niveau.

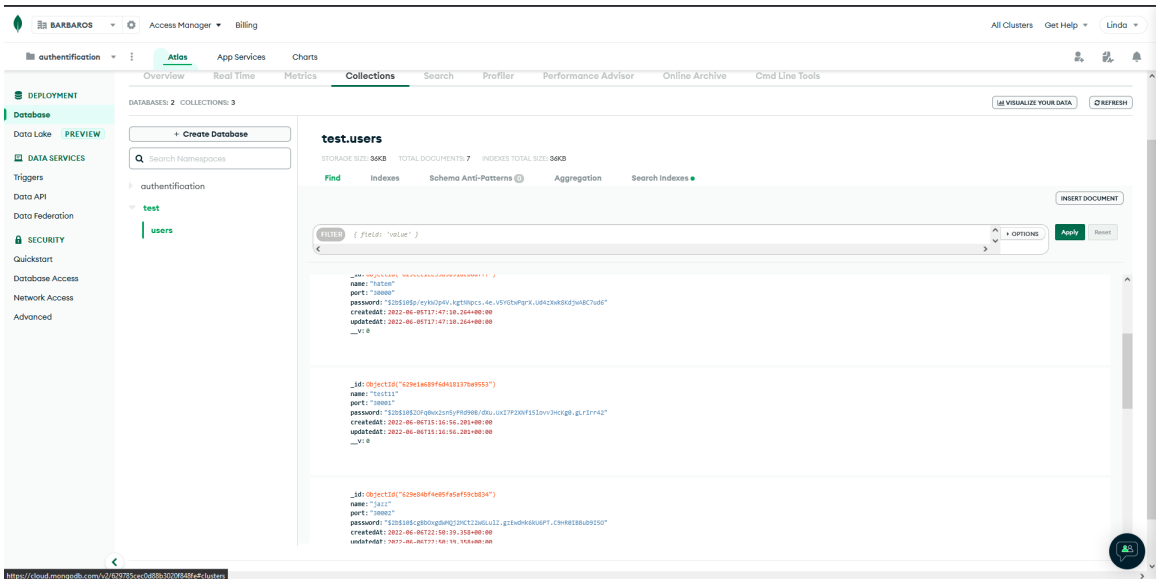


FIGURE 4.5 – Schéma de la base de données MangoDB

2. Lancer le backend avec npm start :

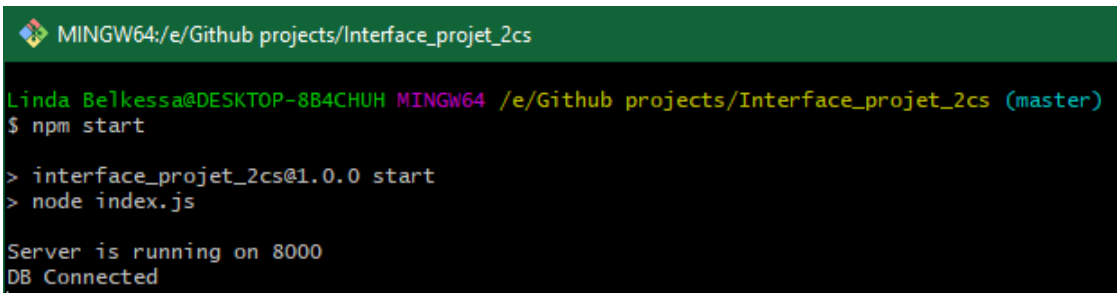


FIGURE 4.6 – npm start du serveur

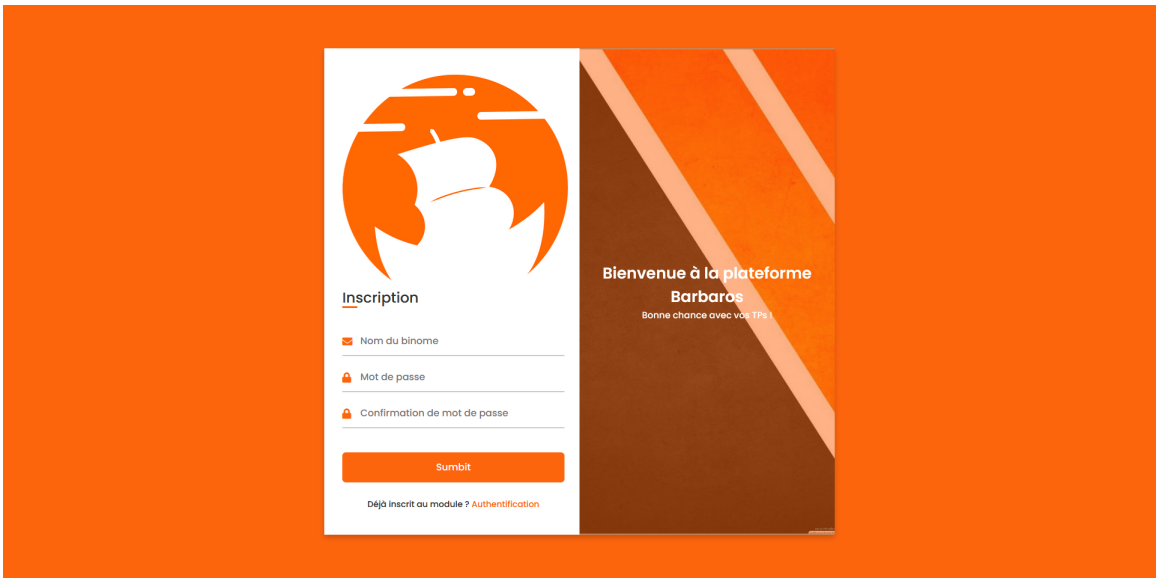


FIGURE 4.7 – Vue globale de la plateforme

3. Inscrire un binome :

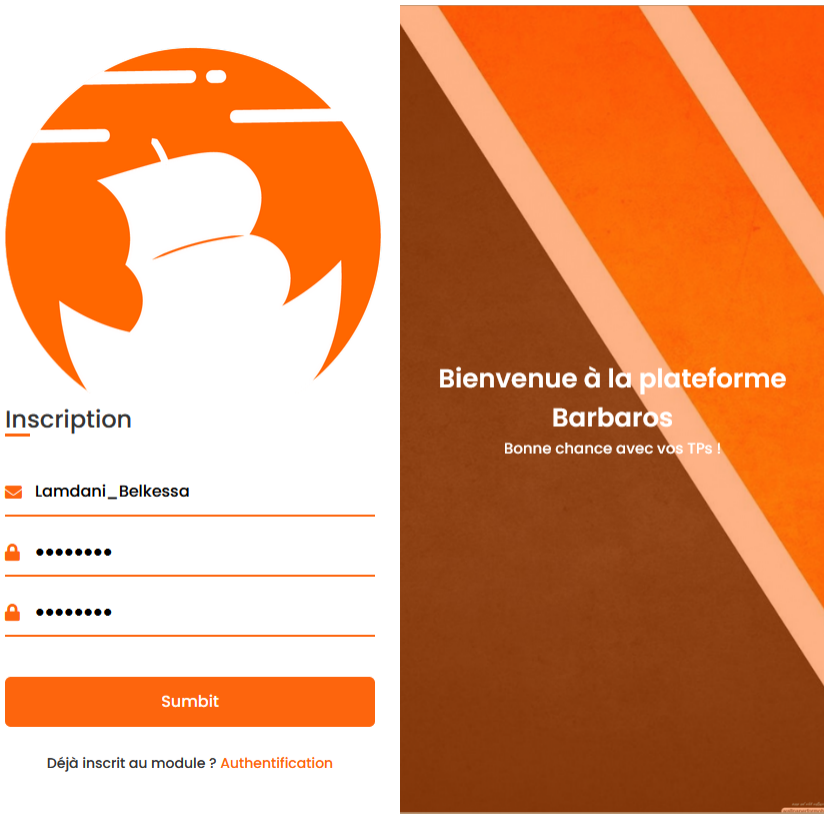


FIGURE 4.8 – Inscription d'un binome

4. Le backend renvoie automatiquement un numéro de port valide :

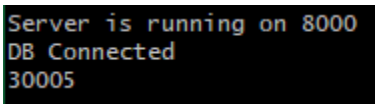


FIGURE 4.9 – Affectation automatique de numéro de port

5. Authentification avec le nouveau binome créé :

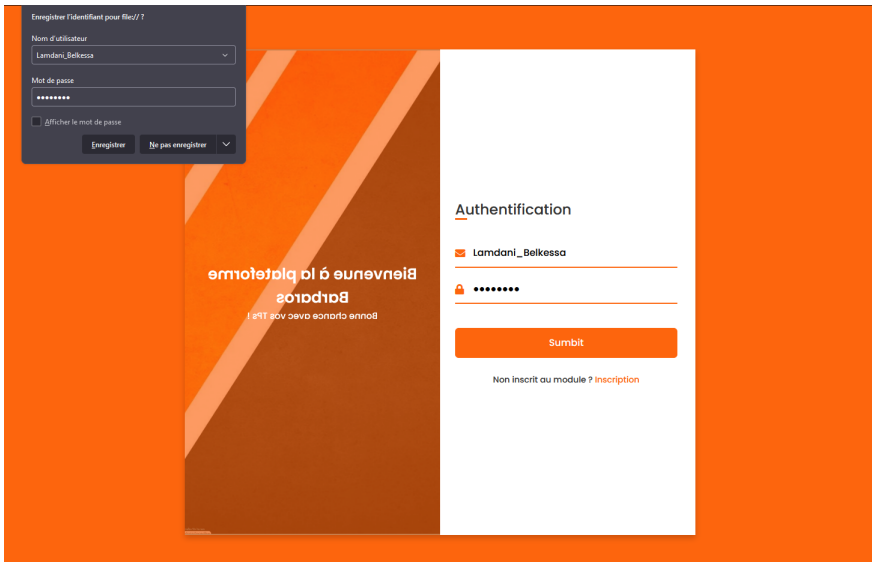


FIGURE 4.10 – Authentification depuis la plateforme

6. Lancement automtique de VNCviewer avec le numéro de port convenable :

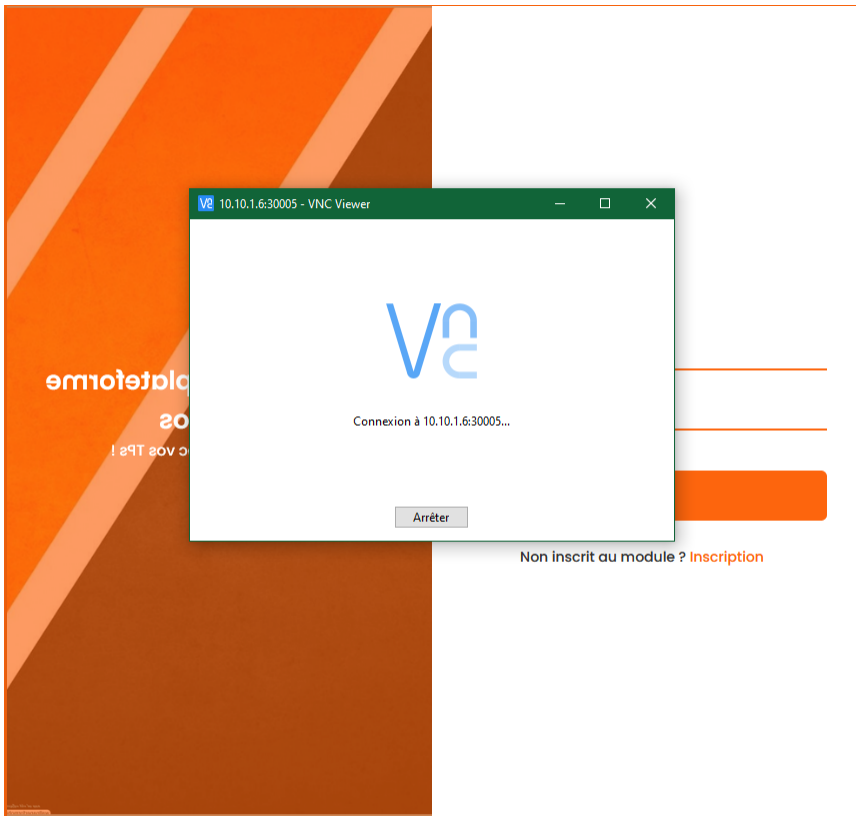
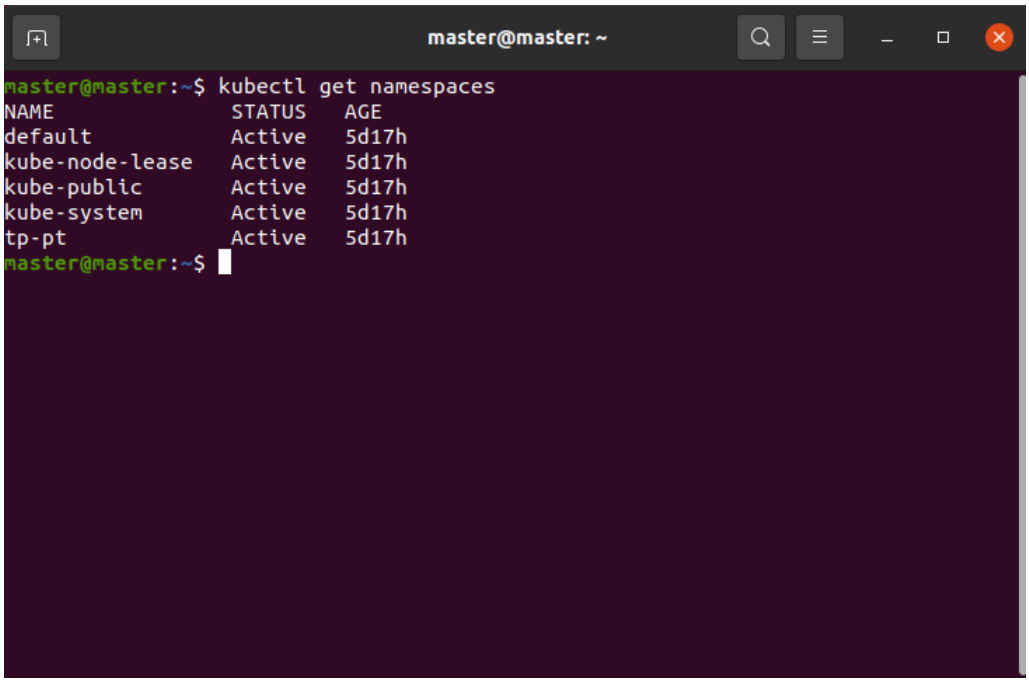


FIGURE 4.11 – Lancement de VNCviewer automatiquement

4.3.2 Déployer deux pods dans le cluster :

(a) créer le namespace pour le TP réseau :



(b) créer le Persistant Volume : Le fichier de déploiement du persistent volume :

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pv-pt
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: nfs
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /srv/TPs
    server: 10.10.1.4
```

Visualiser le persistent volume au niveau du cluster :

```
master@master:~/Desktop/pt deployment/binome2$ kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                STORAGECLASS   REASON   AGE
nfs-pv   10Gi       RWX            Recycle          Bound    tp-pt/nfs-pv-claim   nfs                     26h
```

(c) créer le persistant volume claim : Le fichier de déploiement du persistent volume claim :

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pv-claim
spec:
  storageClassName: nfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Visualiser le persistent volume claim au niveau du cluster :

```
master@master:~/Desktop/pt deployment/binome2$ kubectl get pvc -n tp-pt
NAME           STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nfs-pv-claim   Bound    nfs-pv   10Gi       RWX            nfs            26h
```

(d) créer le déploiement de chaque binôme : Le fichier de déploiement de l'application pour le binôme 1 :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pt-deployment-b1
  namespace: tp-pt
  labels:
    app: packettracerb1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: packettracerb1
  template:
    metadata:
      labels:
        app: packettracerb1
    spec:
      volumes:
      - name: pt-pv-storage
        persistentVolumeClaim:
          claimName: nfs-pv-claim
      containers:
      - name: packettracerb1
        image: esialg/packettracer:1cs
        command: ["/bin/sh","-c"]
        args: ["x11vnc -storepasswd pwbinome1 ~/.vnc/passwd; x11vnc -create -usepw -forever"]
        ports:
        - containerPort: 5900
        volumeMounts:
        - mountPath: "/home/student/Tps"
          name: pt-pv-storage
```

Le fichier de déploiement de l'application pour le binôme 2 :

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pt-deployment-b2
  namespace: tp-pt
  labels:
    app: packettracerb2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: packettracerb2
  template:
    metadata:
      labels:
        app: packettracerb2
    spec:
      volumes:
      - name: pt-pv-storage
        persistentVolumeClaim:
          claimName: nfs-pv-claim
      containers:
      - name: packettracerb2
        image: esialg/packettracer:1cs
        command: ["/bin/sh","-c"]
        args: ["x11vnc -storepasswd pwbinome2 ~/.vnc/passwd; x11vnc -create -usepw -forever -geometry 1920x1080"]
        ports:
        - containerPort: 5900
        volumeMounts:
        - mountPath: "/home/student/Tps"
          name: pt-pv-storage
```

Visualiser les déploiements au niveau du cluster virtuel tp-pt :

```
master@master:~/Desktop/pt deployment/binome2$ kubectl get deployment -n tp-pt
NAME               READY   UP-TO-DATE   AVAILABLE   AGE
pt-deployment-b1   1/1     1            1           25h
pt-deployment-b2   1/1     1            1           9h
```

Visualiser les pods au niveau du cluster virtuel tp-pt :

```
master@master:~/Desktop/pt deployment/binome2$ kubectl get pods -n tp-pt
NAME                               READY   STATUS        RESTARTS   AGE
pt-deployment-b1-79497986b9-k9jc5   1/1     Running       0          31m
pt-deployment-b2-9f8798f4-tz6qp     1/1     Running       0          8m12s
```

(e) créer le service du déploiement de chaque binôme : Le fichier de déploiement du service NodePort pour le binome 1 :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pt-service-b1
  namespace: tp-pt
spec:
  selector:
    app: packettracerb1
  type: NodePort
  ports:
    - protocol: TCP
      port: 5900
      targetPort: 5900
      nodePort: 30000
```

Le fichier de déploiement du service NodePort pour le binome 2 :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: pt-service-b2
  namespace: tp-pt
spec:
  selector:
    app: packettracerb2
  type: NodePort
  ports:
    - protocol: TCP
      port: 5900
      targetPort: 5900
      nodePort: 30001
```

Visualiser les services au niveau du cluster virtuel tp-pt :

```
master@master:~/Desktop/pt deployment/binome2$ kubectl get service -n tp-pt
NAME            TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
pt-service-b1   NodePort   10.100.28.2     <none>        5900:30000/TCP   26h
pt-service-b2   NodePort   10.98.208.45    <none>        5900:30001/TCP   26h
```

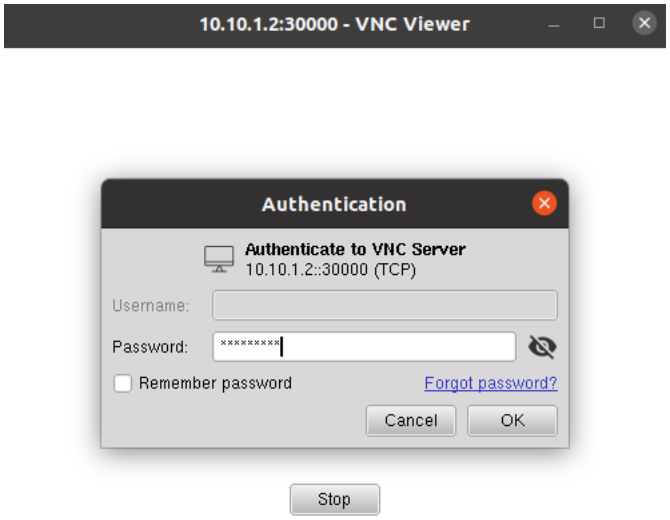
4.3.3 Mettre le fichier du TP au niveau du serveur NFS :

Visualiser le fichier du tp mis dans le serveur nfs :

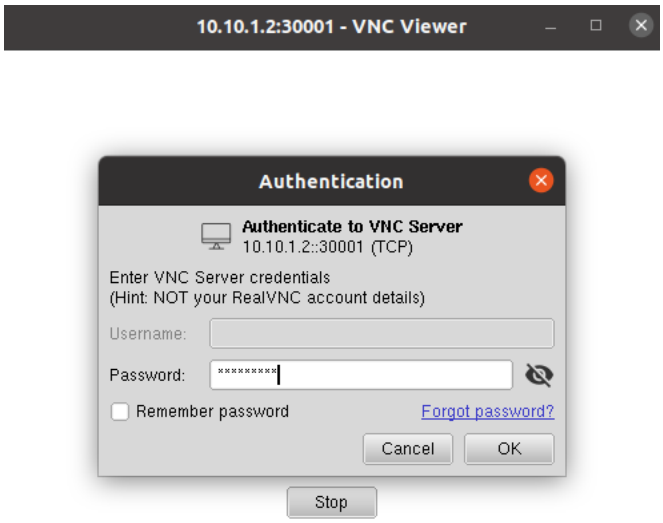
```
nfs@ubuntu:~/Desktop$ ls /srv/TPs/Sujet
TP1.pkt
```

4.3.4 Accéder au pods depuis deux machines clientes :

— de la machine du binôme 1 :

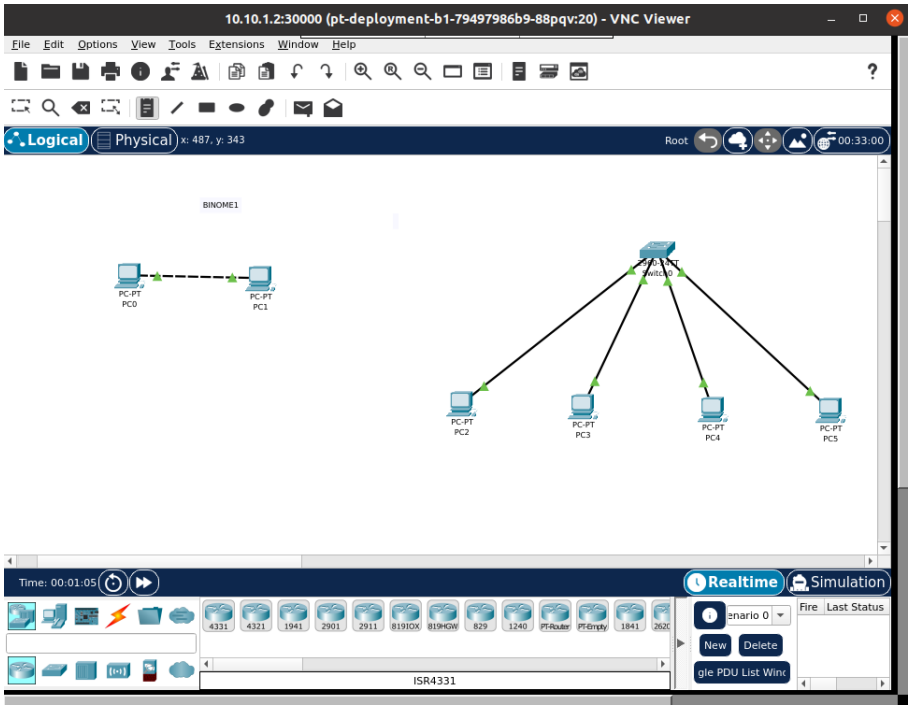


— de la machine du binôme 2 :

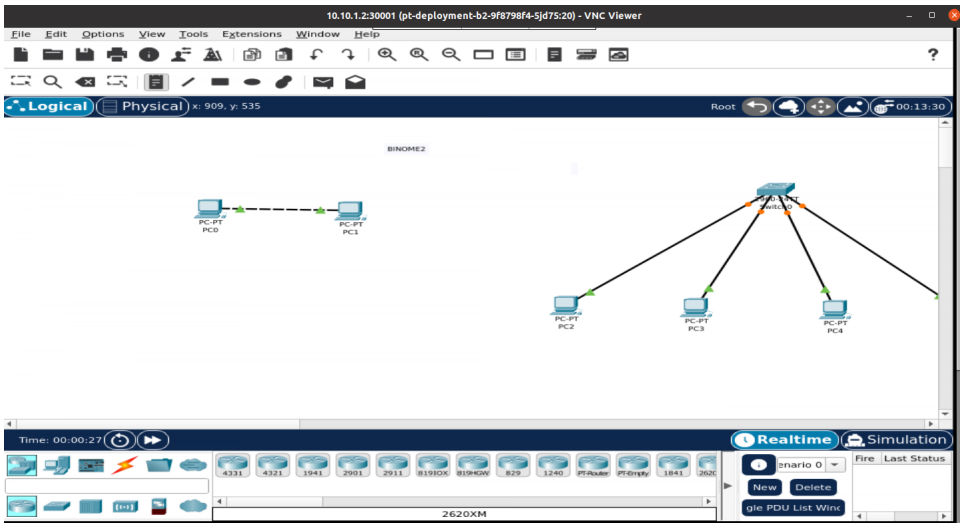


4.3.5 Appliquer une modification

— binôme 1 :

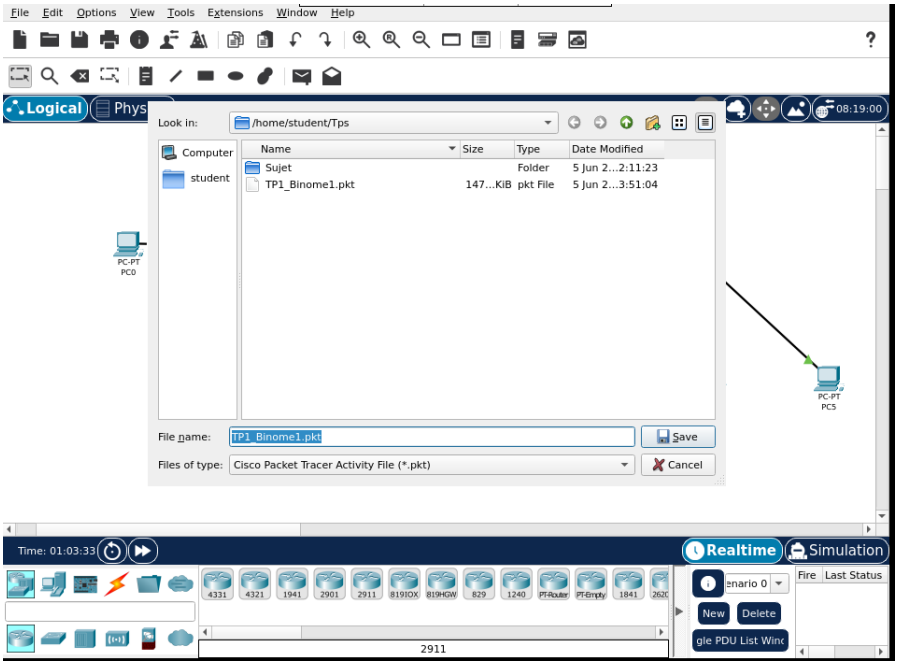


— binôme 2 :

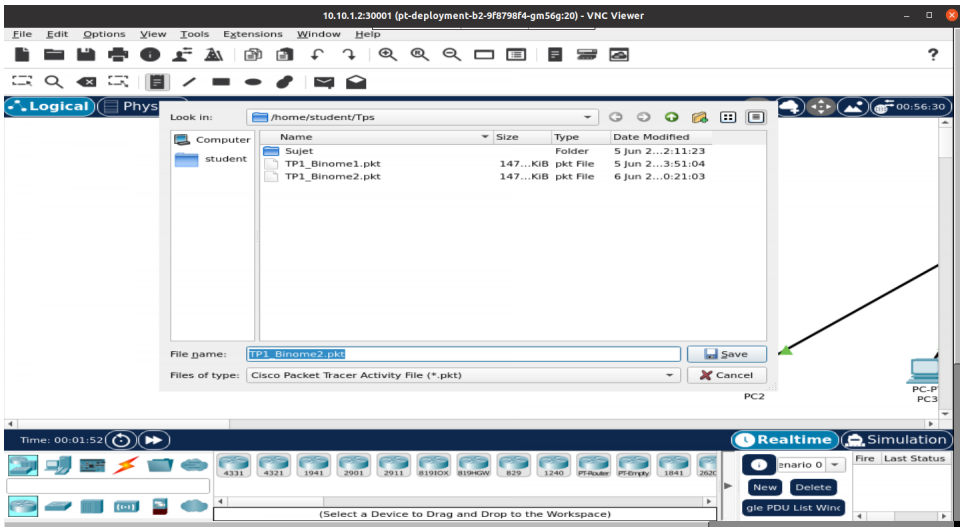


4.3.6 Enregistrer Le fichier :

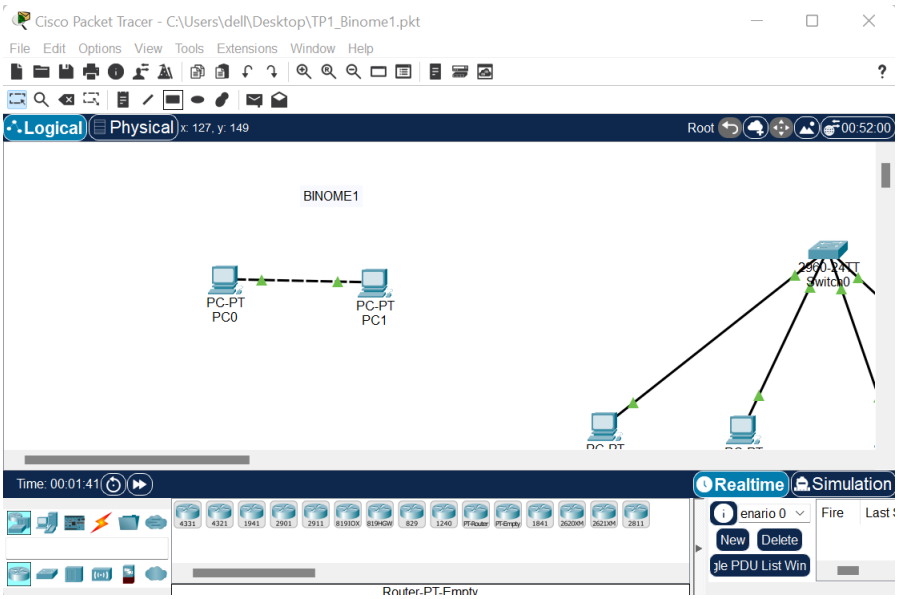
— binôme 1 :



— binôme 2 :



4.3.7 Vérifier la sauvegarde de la modification :



5. Conclusion

A la fin de ce travail, nous réitérons une autre fois sur l'importance de la conteneurisation des ressources et outils pour unifier les environnements de travail.
Notre système facilite l'utilisation des outils des TPs au niveau de l'Ecole Nationale Supérieure d'Informatique d'Alger tout en assurant une expérience d'apprentissage agréable et moins de soucis techniques pour les enseignants leur permettant ainsi d'investir le temps consacré à chaque module au contenu lui-meme au lieu que ce soit aux problèmes d'installation et configuration.

La mise en production de ce système est aussi considéré lors de la conception ce qui permet le passage direct du prototype vers le produit final.