TP Kubernetes

Au cours de ce TP, vous allez installer, configurer et administrer un cluster K8S. Puis, vous allez manipuler différents objets K8S (Workloads, Pods, Volumes etc).

Creation de l'infrastructure

Dans cette section, vous devez créer trois machines virtuelles dans OpenStack avec les caractéristiques suivantes:

- Image Ubuntu 20.04.3 LTS - Docker Ready
- 2 vCPU
- 4GB RAM
- 10GB d'espace disque

Une machine sera le Master Node (Control Plane) et deux autres seront des Worker Nodes.

Ces machines doivent avoir des hostnames suivants:

- [num_etu]-master-node
- [num_etu]-worker1
- [num_etu]-worker2

Installation et Validation

Dans cette section, vous allez installer et valider le fonctionnement du cluster Kubernetes. Nous allons utiliser Docker comme Container Runtime.

Installation

Sur tous les VMs

- Ajoutez la clé GPG et le repository Kubernetes

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -
$ sudo add-apt-repository "deb https://apt.kubernetes.io/ kubernetes-xenial main"
```

- Installez les packages Kubernetes

```
$ sudo apt-get install -y kubelet=1.21.8-00 kubeadm=1.21.8-00
kubectl=1.21.8-00
```

  - Dans cette section vous installez la version 1.21.8 du Kubernetes. Dans la section suivante, vous verrez comment mettre à jour K8S.

- Bloquez la mise à jour automatique des packages installés précédemment

```
$ sudo apt-mark hold kubelet kubeadm kubectl
```

  - La mise à jour automatique de ces packages peut casser votre cluster.

Configuration du proxy

- Ajoutez la variable NO_PROXY dans les variables d'environnement
  - Ajoutez la ligne dans /etc/environment

```
NO_PROXY=univ-lyon1.fr,127.0.0.1,localhost,10.244.0.0/16,10.96.0.0
/12,192.168.0.0/16
```

    - 10.244.0.0/16 - la plage des adresses des PODS dans votre cluster
    - 10.96.0.0/12 - la plage des adresses systeme de Kubernetes

- Testez si Docker peut télécharger et lancer un conteneur

```
$ sudo docker run hello-world
```

- Redémarrez tous les machines

Initialization du Cluster

Une fois les packages Kubernetes installés, vous pouvez initialiser le cluster et installer un CNI (Container Network Interface).
Le processus d'initialisation du cluster est très simple. Vous allez utiliser **kubeadm** pour initialiser votre cluster et **flannel** comme le CNI.

Sur le noeud master

- Initialisez votre cluster

```
$ sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

  - **Attention!** Mémorisez bien le token donné par cette commande, ce token sera utilisé par vos nœuds workers pour rejoindre le cluster.
  - Qu'est-ce que l'option "--pod-network-cidr" permet de faire?
  - Essayez de comprendre les étapes d'initialisation du cluster Kubernetes.

- Configurez l'outil d'administration **kubectl**

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

  - Qu'est-ce que cet outil permet de faire?

Sur les noeuds worker

- Ajoutez les Workers dans cluster

```
$ sudo kubeadm join [join_token]
```

  - Le token a été donné par la commande **kubeadm init**, lors de l'initialisation du cluster.

Sur le noeud Master

- Verifiez l'etat des nodes

```
$ kubectl get nodes
$ kubectl describe nodes
```

  - Vérifiez l'état des nœuds.
  - Pourquoi l'état des nœuds est "NotReady"?

- Installez CNI (Container network interface)

```
$ kubectl apply -f
https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
$ kubectl get pods -n kube-system
```

  - Pour initialiser Flannel, Kubernetes crée un objet de type "DaemonSet". Pourquoi un objet de type "DaemonSet" est-il créé?
  - Attendez que les Pods Flannel soient en état "Running".
- Re-vérifiez l'état des nœuds
  - Qu'avez-vous remarqué?

- Decrivez les pods du namespace kube-system

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

Validation de l’installation

- Creez un deployment nginx

```
$ kubectl create deployment --image=nginx nginx
```

- Vérifiez que le pod est bien lancé et que le déploiement a été bien créé

```
$ kubectl get pods
$ kubectl get deployments
```

- Créez un port forward et vérifiez son fonctionnement

```
$ kubectl port-forward PODNAME 8081:80 &
$ curl 127.0.0.1:8081
```

  - Que permet de faire un port-forward?

- Visualisez des logs du Pod

```
$ kubectl logs PODNAME
```

- Créez et vérifiez un service de type NodePort

```
$ kubectl expose deployment nginx --port 80 --type NodePort
$ kubectl get services
$ curl -I 127.0.0.1:NODE_PORT
```

  - Que permet de faire un service?
  - Que fait un service de type “NodePort”?

La mise à jour du cluster

Dans cette section vous allez mettre à jour votre cluster K8s.
Pour effectuer cela, vous allez utiliser l'outil **kubeadm**.

- Que permet de faire l'outil **kubeadm**?

Préparation de la mise à jour

Avant de commencer la mise à jour du cluster, il faut vérifier la version actuelle des éléments de votre cluster. Ensuite, vous devez trouver quelle est la dernière version stable de K8S.

- Vérifiez la version de **kubelet** sur les noeuds

```
$ kubectl get nodes
```

- Vérifiez la version d'API du client et serveur

```
$ kubectl version --short
```

- Vérifiez la version de **kubeadm**

```
$ kubeadm version
```

Au moment de la rédaction de ce TP, la dernière version stable de Kubernetes est la **v1.22.5**. Vous allez donc mettre à jour le cluster vers cette version.

La mise à jour du cluster

Maintenant, vous pouvez commencer la mise à jour de votre cluster. Nous allons commencer par le noeud Master.

- Exportez les variables d'environnement suivantes

```
$ export VERSION=v1.22.5
$ export ARCH=amd64
```

- Récupérez et installez de la nouvelle version de **kubeadm**

```
$ curl -sSL
https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > kubeadm
$ sudo install -o root -g root -m 0755 ./kubeadm /usr/bin/kubeadm
$ sudo kubeadm version
```

- Exécutez la commande de planification de la mise à jour

```
$ sudo kubeadm upgrade plan
```

  - Que fait cette commande?
  - Si toute l'information affiché vous semble correcte, vous pouvez effectuer la mise à jour du cluster

- Mettez à jour le cluster

```
$ sudo kubeadm upgrade apply v1.22.5
```

- Vérifiez les versions de **kubelet** sur les noeuds

```
$ kubectl get nodes
```

  - Que pouvez-vous constater?

- Mettez à jour le **kubelet**
  - Exportez les variables d'environnement suivantes

```
$ export VERSION=v1.22.5
$ export ARCH=amd64
```

  - Installez la nouvelle version de **kubelet**

```
$ curl -sSL
https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubel
et > kubelet
$ sudo install -o root -g root -m 0755 ./kubelet
/usr/bin/kubelet
$ sudo systemctl restart kubelet.service
```

  - Vérifiez la mise à jour de **kubelet**

```
$ kubectl get nodes
```

- Vérifiez la version du **kubectl**

```
$ kubectl version
```

  - Que pouvez-vous constater?

- Mettez à jour le **kubectl**

```
$ curl -sSL
https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubectl >
kubectl
$ sudo install -o root -g root -m 0755 ./kubectl /usr/bin/kubectl
$ kubectl version
```

- Mettez à jour tous les noeuds du cluster
  - Quel composant doit être mis à jour sur les nœuds workers afin de finaliser la mise à jour de votre cluster ?

Bravo! Vous avez mis à jour votre cluster sans aucune interruption de service!

Utilisation du cluster

Dans cette section, vous allez déployer quelques objets Kubernetes sur votre cluster. Les objets K8S peuvent être créés directement en utilisant l'outil **kubectl**. Vous avez déjà créé vos premiers objets K8S dans la section validation de votre installation. Dans cette section, vous allez décrire les objets K8S dans les fichiers yml, puis les créer dans K8s avec la commande

```
$ kubectl create -f nom_du_fichier.yaml
```

Creation d'un pod

Vous commencerez par créer un Pod qui est un objet de base de K8S.

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    service: web
spec:
   containers:
   - name: nginx
     image: nginx
     ports:
        - containerPort: 80
          hostPort: 8080
```

Ce fichier décrit un pod qui a les caractéristiques suivantes:

- **Nom**: nginx-pod
- **Label**: service = web
- **Image et nom du conteneur**: nginx

- **Le port du conteneur:** 80
- **Le port exposé sur le hôte:** 8080

- Créez cet objet dans le cluster

```
$ kubectl create -f pod.yaml
```

- Vérifiez si le pod a été bien créé, puis supprimez le

```
$ kubectl get pods
$ kubectl delete pod <nom_du_pod>
```

  - Utilisez l'option "-o wide" pour voir plus d'information sur les objets K8s
  - Sur quel nœud le Pod a-t-il été lancé?
  - Pouvez-vous accéder au Pod via le port 8080 ? Testez à partir de tous les nœuds du cluster.
  - Supprimez ce pod.

Creation d'un deployment

Dans la section précédente, vous avez créé un Pod avec l'application Nginx. Dans la vraie vie, vous ne manipulez jamais directement les Pods. Vous allez créer des objets contrôleurs (workload ressources), qui créent et gèrent plusieurs Pods pour vous. Ces objets assurent la réplication, le déploiement et le self-healing automatique des vos pods.

Dans cette section, vous allez déployer une application hautement disponible avec des mécanismes de mises à jour déclaratives et rollbacks. Pour commencer, vous allez créer un objet de type "Deployment".

deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: web
  replicas: 3
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
          - containerPort: 80
```

- Créez cet objet dans le cluster

```
$ kubectl create -f deployment.yaml
```

  - Quels rôles jouent les labels et les sélecteurs ?
  - Sur quelle image seront basés les conteneurs créés?

- Vous pouvez suivre le processus de déploiement et visualiser l'état des déploiements avec les commandes

```
$ kubectl rollout status deployments nginx-deployment
$ kubectl get deployments
```

  - Combien de réplicas ont été créés par le déploiement?

- Mettez à l'échelle votre déploiement pour avoir 6 replicas

```
$ kubectl scale deployment nginx-deployment --replicas=6
```

  - N'hésitez pas à utiliser la commande **kubectl describe** pour afficher une description détaillée d'un objet K8S

- Vérifiez que votre déploiement a lancé un bon nombre des replicas

```
$ kubectl get deployments
```

- Visualisez les pods lancées par le déploiement

```
$ kubectl get pods
```

  - Comment sont distribués les pods entre les nœuds workers? *(option **-o wide**)*

Nous avons créé un objet de type “Deployment” qui crée et maintient un nombre des Pods demandées. Ensuite, nous avons augmenté le nombre de Pods à la volée. “Deployment” peut être vu comme un regroupement des Pods dont le nombre est garanti par K8S.
Pour qu'un “Deployment” soit accessible, vous allez créer un objet de type “Service”.
Le service peut être considéré comme un équilibreur de charge qui distribue le trafic vers un ensemble des Pods.

service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-deployment
spec:
  type: NodePort
  selector:
    app: web
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

- Créez cet objet dans le cluster

```
$ kubectl create -f service.yaml
```

  - Quel est l'intérêt de la section “selector” dans le fichier yaml?

- Détectez quel port est exposé sur les noeuds pour atteindre le service

```
$ kubectl get services
```

- Affichez des endpoints du service

```
$ kubectl get endpoints
```

  - Quelles adresses sont affichées dans la liste des ENDPOINTS?

- Vérifiez que le déploiement est bien accessible depuis n’importe quel noeud

```
$ curl -I 127.0.0.1:[node_port]
```

- Vérifiez que le service est accessible depuis n'importe quel Pod créé précédemment

    - Le service doit être accessible en utilisant son nom comme nom de domaine depuis n'importe quel pod. Pour cela, vous devez effectuer une requête "curl" sur “http://nginx-deployment”. Avez-vous réussi à faire cela?

Rolling Updates

Imaginez que des développeurs ont publié une nouvelle version de l'application et que vous êtes responsable de la mise à jour.

Pour simuler ce scénario, nous allons changer la version de l’image nginx.

Pour pouvoir suivre la mise à jour, nous allons ralentir ce processus. Le déploiement sera suspendu pendant 10 secondes après le déploiement de chaque nouveau Pod.

```
$ kubectl patch deployment nginx-deployment -p '{"spec":
{"minReadySeconds": 10}}'
```

Mettre à jour un déploiement

Vous avez deux moyens de mettre à jour votre déploiement:

- En modifiant le fichier yaml de l’objet

  deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: web
  replicas: 3
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
```

```
      - name: nginx
        image: nginx:1.16.0
        ports:
          - containerPort: 80
```

  - Pour appliquer les changements

```
$ kubectl apply -f deployment.yaml
```

- Inline (en utilisant la ligne de commande)

```
$ kubectl set image deployments/nginx-deployment nginx=nginx:1.16.0 --v
6
```

- Mettez-à-jour le déploiement et vérifiez que l'application a été bien mise-à-jour

```
$ kubectl get services
$ curl -I 127.0.0.1:[node_port]
```

  - Exécutez la commande plusieurs fois
  - Comme nous avons ralenti le processus de déploiement au début de cette section, vous pouvez suivre en temps réel le déploiement de la nouvelle version

```
$ watch -n 1 curl -I 127.0.0.1:[node_port]
```

Comme vous voyez la mise à jour est passée sans aucune interruption de service.

Rollbacks

Imaginez que vous avez mis à jour une application en production et que cette application ne fonctionne plus ou que la nouvelle version n'est plus compatible avec d'autres éléments de la stack applicative. Kubernetes vous offre la possibilité d'effectuer un Rollback.

- Effectuez le rollback de déploiement nginx-deployment

```
$ kubectl rollout undo deployments nginx-deployment
```

- Suivez le processus de Rollback

```
$ watch -n 1 curl -I 127.0.0.1:[node_port]
```

Vous pouvez spécifier la version vers laquelle vous souhaitez revenir. Avant cela, vous devez récupérer l'historique des déploiements et choisir la révision du déploiement.

- Récupérez l'historique des déploiements

```
$ kubectl rollout history deployment nginx-deployment
```

  - Affichez les détails de la version 2 du Deployment. Quelle commande utiliserez-vous?

Vous pouvez revenir à une version particulière du déploiement en utilisant l'option "--to-revision".

- Revenez à la révision 2 du déploiement
  - Quelle commande utiliserez-vous?

Volumes

Parfois, vos applications auront besoin d'un stockage persistant. Dans cette section vous allez créer et manipuler des volumes persistants.

La création d'un volume et son attribution à un pod se font en plusieurs étapes.

Premièrement, vous devez créer un objet "Persistent Volume".
Dans le cadre de ce TP, nous allons créer un volume "local" (un répertoire sur le host). Cette tâche est généralement effectuée par l'administrateur du cluster.

pv.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

- Créez cet objet dans le cluster

```
$ kubectl create -f pv.yaml
```

  - Que signifie l'accès mode “ReadWriteOnce”?

- Quel est son statut après la création?

```
$ kubectl get pv
```

  - Quelle est la stratégie de rétention de volume persistant créée et que signifie-t-elle ?

Vous ne pouvez pas attacher directement un volume à votre Pod.

Kubernetes ajoute une couche d’abstraction - l’objet “PersistentVolumeClaim”.

Cette abstraction permet de découpler les volumes configurés et mis à disposition par les administrateurs K8S et les demandes d’espace de stockage des développeurs pour leurs applications.

Pour créer une demande d’un volume, nous allons créer l’objet “PersistentVolumeClaim”.

Nous allons demander un volume qui a au moins 3 Giga de stockage et qui peut fournir un accès en lecture-écriture à au moins un nœud.

pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

- Créez cet objet dans le cluster

```
$ kubectl create -f pvc.yaml
```

- Quel est le statut du volume persistant et du PVC après la création de la claim?

```
$ kubectl get pv
$ kubectl get pvc
```

- Est-ce que deux claims peuvent utiliser le même volume persistant?
  - Vous pouvez tester cela en créant une autre Persistent Volume Claim avec un nom différent et voir si cette claim sera Bound.

Maintenant, vous pouvez attribuer le “PersistantVolumeClaim” à un Pod.

pvc-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
    - image: mongo
      name: mongodb
      volumeMounts:
        - name: mongodb-data
          mountPath: /data/db
      ports:
        - containerPort: 27017
          protocol: TCP
  volumes:
    - name: mongodb-data
      persistentVolumeClaim:
        claimName: task-pv-claim
```

- Créez cet objet dans le cluster

  ```
  $ kubectl create -f pvc-pod.yaml
  ```

- Vérifiez que votre pod est lancé

  ```
  $ kubectl get pods
  ```

- Trouvez un moyen de vérifier que le volume persistent fonctionne correctement

  - Comment l'avez-vous vérifié ?

Secrets

Les secrets sont utilisés pour sécuriser les données sensibles auxquelles vous pouvez accéder à partir de vos pods.

Dans cette section, vous allez créer un secret. Puis, vous allez l’utiliser dans un Pod.

Les secrets peuvent être fournis à vos Pods en tant que variables d'environnement ou en tant que volume.

secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: 42-secret
data:
  username: NDI=
  password: NDI=
```

- Modifiez le secret pour que les champs *username* et *password* soient *SRIV.* Pour cela, vous devez convertir la chaîne en base64 (commande ***base64***)
  - Quel est le contenu du fichier secret.yaml après les modifications?
  - N'hésitez pas à utiliser la documentation officielle.
- Créez le secret dans le cluster

```
$ kubectl create -f secret.yaml
```

pod-with-secret.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-secret
spec:
   containers:
   - name: busybox
     image: busybox
     command: [‘sh’, ‘-c’, ‘echo Hello $SECRET_USERNAME!
&& ls -al /secret && sleep 99999’]
```

- Modifiez la description du pod afin que la variable d'environnement “$SECRET_USERNAME” contient la partie “username” du secret et que le répertoire “/secret” soit monté comme un volume du secret.
  - N'hésitez pas à utiliser la documentation officielle.
- Créez le Pod dans le cluster

```
$ kubectl create -f pod-with-secret.yaml
```

- Visualisez les logs du pod

```
$ kubectl get pods
$ kubectl logs NOM_DU_POD
```

  - Que voyez-vous dans les logs du Pod?

Liveness et Readiness

Kubernetes peut vérifier automatiquement si vos applications répondent aux demandes des utilisateurs avec des sondes Liveness. Si votre application ne répond pas, K8S le détecte et redémarre ou recrée le conteneur.

Kubernetes peut également retenir le trafic entrant jusqu'à ce que votre service soit en mesure de recevoir les demandes des utilisateurs avec des sondes Readiness.

Dans cette section, vous allez déployer des Pods avec les sondes Liveness et Readiness.

Liveness probes

Pour rappel, les sondes Liveness permettent de vérifier si un conteneur est en vie et répond bien aux requêtes.

liveness-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness
spec:
  containers:
  - image: nginx
    name: nginx
    livenessProbe:
       httpGet:
         path: /
         port: 80
```

- Créez cet objet dans le cluster

```
$ kubectl create -f liveness-pod.yaml
```

- Déployez un Pod nginx avec la liveness probe qui échoue
  - Par exemple: qui requête un port qui n’est pas écouté par nginx

- Surveillez les events et le comportement du Pod

```
$ kubectl get pods
```

```
$ kubectl describe pod POD_NAME
```

- Que fait Kubernetes en cas d'échec de la liveness probe?
- Combien de fois Kubernetes essaye de redémarrer le Pod avant de conclure qu'il est défaillant?

Readiness probe

Pour rappel, les sondes Readiness permettent de vérifier si un conteneur peut recevoir les demandes des utilisateurs. Si la vérification échoue, le trafic ne sera pas dirigé vers ce Pod.

readiness.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-readiness
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: nginx-readiness
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-good
  labels:
      app: nginx-readiness
spec:
  containers:
  - name: nginx
    image: nginx
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-nogood
  labels:
```

```
      app: nginx-readiness
spec:
  containers:
  - name: nginx
    image: nginx:1.222
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
```

- Créez cet objet dans le cluster

```
$ kubectl create -f readiness.yaml
```

- Etudiez le comportement des pods avec readiness check
  - Surveillez les pods

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

  - Surveillez la liste des endpoints du service

```
$ kubectl get endpoints
```

  - Que remarquez-vous?

- Est-ce que le service répond aux requêtes?
  - Comment pouvez-vous expliquer un tel comportement?
- Trouvez et corrigez l’erreur
  - Utilisez la commande

```
$ kubectl edit pod nginx-nogood
```

- Surveillez l'état des pods et la liste des endpoints du service
  - Que remarquez-vous?

Bravo! Vous avez fini le TP!