Workshop Kubernetes (bases et stateless)

Jean-Marc Pouchoulon

Septembre 2023

Les containers se sont révélés comme des entités informatiques nativement adaptées aux applications CLOUD. Très proches de la vision applicative du système d'information il sont devenus rapidement des "first class citizen" des infrastructures actuelles.

En tant que tels ils leur faut un cadre de fonctionnement solide et adaptable, piloté par un chef d'orchestre : Kubernetes. Issu de Google, Kubernetes est devenu rapidement la référence en termes d'orchestration de containers.

L'API de Kubernetes est identique quelque soit les infrastructures sous-jacentes. C'est un atout important pour délivrer du code évoluant pour des raisons de sécurité informatique ou d'amélioration continue. C'est aussi la promesse d'un couplage faible entre les infrastructures et le code permettant une organisation claire entre Dev && Ops.

Kubernetes ou "k8s" [1] évolue rapidement et offre des solutions pour piloter des containers mais aussi des machines virtuelles. [2]

1 Compétences à acquérir lors du TP.

Compétences pré-requises:

— Enseignements précédents en particulier sur les containers applicatifs : Construire un container avec un Dockerfile , utiliser un registry, les commandes Docker...

Compétences principales:

L'objectif de ce TP est d'orchestrer des containers pour améliorer la disponibilité et la scalabilité des applications. La haute disponibilité du "control plane" de k8s ne sera pas traitée ici.
Les compétences visées:

— travailler avec une configuration Kubernetes sur son propre poste;

— utiliser la commande "kubectl";

— transformer les commandes "imperative" en "declarative";

— utiliser Docker et Kubernetes;

1. il est appelé ainsi car entre le K et le S de Kubernetes il y a 8 lettres
2. voir par exemple `https://www.vmware.com/fr/products/vsphere/projectpacific.html` et `https://www.weave.works/blog/firekube-fast-and-secure-kubernetes-clusters-using-weave-ignite`

— créer et gérer un "pod";
— créer et gérer un "pod init";
— limiter la consommation de ressources d'un pod;
— utiliser un label;
— utiliser les concepts de "taint & toleration";
— mettre un noeud k8s en maintenance et le remettre en production;
— générer une définition en "Yaml" d'un objet k8s à partir d'un "dry-run";
— "kubernétiser" une image Docker;
— rendre disponible son application en dehors du Cluster (services dont "NodePort" et "LoadBalancer", "ingress controller", "Bare metal loadbalancer","Ingress Controller");
— gérer le cycle de vie d'une application dans Kubernetes;
— utiliser le manager d'applications "Helm" for "fun & profit";
— stocker des données de façon permanente dans un cluster;

Savoir:

— comprendre la philosophie de Kubernetes;
— savoir énoncer les rôles du "Control plane" et du "Data Plane" dans Kubernetes;
— savoir décrire et énoncer les fonctions de quelques objets essentiels de Kubernetes ("NameSpaces" ,"pods" , "deployments", "services", "replicatsets"...);

Vous travaillerez individuellement pour ce workshop.
Vous changerez le nom de votre hôte afin de faire apparaître votre nom via la commande suivante suivi d'un reboot:

```
hostnamectl set-hostname votrenom-vm
```

Sur le compte-rendu figureront les numéros de questions, les réponses mais aussi les résultats de vos commandes montrant la réussite de vos actions. Le prompt modifié ci-dessus apparaîtra sur vos "output" N'oubliez pas votre nom sur vos compte-rendus.

2 "Crash course" sur Kubernetes

2.1 Approches globales de Kubernetes

Kubernetes peut être vu sous plusieurs angles:

— comme un système d'exploitation réparti sur plus nœuds ou hôtes du réseau. Il va piloter des ressources ("containers" ou "virtual machine").
— comme un pilote [1] sous forme d'une "boucle informatique" qui sans cesse vérifie qu'il respecte les ordres de son amiral. On peut voir les hôtes machines physiques ou virtuelles qui participent au cluster Kubernetes comme des vaisseaux (participant au *"data plane"* avec d'autres navires) dirigés par une cellule de commandement (le *"control plane"*) et remplissant des tâches en fonction des ordres données par ce dernier. A l'intérieur de chaque vaisseau les groupes de processus sous forme de PODS (un ensemble d'un ou plusieurs containers) sont des marins qui partagent un même but logiciel (un micro-service utile à l'ensemble de la flotte) et des ressources (Network , IPC, voire process "NameSpace").

1. En grec ancien Kubernetes signifie le pilote. Le mot Cybernétique vient de là. C'était la minute culturelle de ce workshop

3 Installation du cluster Kubernetes avec Kind

Il existe de nombreuse façon de travailler avec Kubernetes sur son poste de travail. On peut travailler en local avec le client "kubectl" et se connecter à un cluster de développement distant. On peut aussi installer des noeuds Kubernetes sur son laptop sous forme de machine virtuelle ou de containers... Docker.

On citera:

- minikube;
- micro-k8s (Ubuntu);
- Docker Desktop;
- k3d;
- kind (Kubernetes in Docker);
- ...

Dans ce workshop nous allons utiliser "kind". Les noeuds du cluster Kubernetes sont des containers Docker et le "plugin" réseaux de Kind a été bâtie pour être compatible avec Docker.

- installer kubectl sur votre machine [1];
- installez en suivant la documentation issue de `https://github.com/kubernetes-sigs/kind`;
- créer votre cluster kind en clonant le repository
  `https://github.com/pushou/substrat-labs-k8s.git`
  et lancez le script create-kind-cluster-with-ec.sh;

Vous pouvez visualisez les nodes Kubernetes qui sont ici des containers Docker via la classique commande

```
docker ps
```

et vous rendre sur le noeud Kubernetes via

```
docker exec -it id-container-noeud bash
```

Les pages suivantes:

- `https://kubernetes.io/docs/tasks/`;
- `https://kubernetes.io/fr/docs/reference/kubectl/cheatsheet/`;
- `https://kubernetes.io/docs/tasks/debug-application-cluster/crictl/`;

vous seront d'une grande aide. La commande "k explain" suivi du sujet peut aussi vous aider:

```
k explain pod # afficher l'essentiel
k explain pod.apiVersion # détailler un sujet
k explain pod.apiVersion --recursive # lister tous les champs
```

Il est recommandé d'installer fzf afin de faciliter l'accès à l'historique de commandes:

```
~fzf/install
```

Le client kubernetes kubectl permet d'interagir avec "l'API server". Le format de communication des échanges est le format "json". Ce client est installé sur le poste de l'administrateur et se sert d'un fichier "config" situé dans /.kube qui contiendra l'IP et le port de "l'API server".

1. `https://kubernetes.io/fr/docs/tasks/tools/install-kubectl/`

kubectl supporte plusieurs formats de sortie dont les formats "json" et "yaml". Vous installerez jq via apt et jid [1] pour traiter du json. Passez la commande suivante afin de récupérer les capacités de votre cluster:

```
kubectl get nodes -o json |
jq ".items[] | {name:.metadata.name} + .status.capacity"
```

Créer cet alias essentiel dans votre .bashrc et activer la complétion pour kubectl:

```
alias k=kubectl
echo 'source <(kubectl completion bash)' >>~/.bashrc
kubectl completion bash >/etc/bash_completion.d/kubectl
echo 'alias k=kubectl' >>~/.bashrc
echo 'complete -F __start_kubectl k' >>~/.bashrc
source ~/.bashrc
```

kind permet de charger image depuis un registry pour l'ensemble des noeuds du cluster. Par exemple la commande suivante permet de charger l'image registry.iutbeziers.fr/pythonapp:latest.

```
# Chargement de l'image en local
docker pull registry.iutbeziers.fr/pythonapp:latest
# upload de l'image sur les containers Docker
kind --name tp1k8s load docker-image registry.iutbeziers.fr/pythonapp:latest
```

Vous pourrez utiliser cette fonctionnalité si le registry est privé.

4 Premiers pas avec Kubernetes

4.1 Configuration des objets en mode déclaratif et impératif

Quand on utilise [2] un mode déclaratif on déclare dans un "manifest yaml" les éléments de configurations. Ces éléments décrivent un état souhaité que l'on peut faire évoluer. [3] En "mode impératif" on donne des ordres via la ligne de commande.Ce mode ne garde pas la mémoire des configurations.

1. Adaptez à votre contexte et passez les commandes suivantes pour vous familiariser avec Kubernetes et son mode impératif:

   ```
   kubectl run nginx-pod --image nginx
   kubectl exec -it nginx-pod -- sh
   kubectl create deployment hello-nginx --image nginx
   kubectl scale deployment hello-nginx --replicas 2
   kubectl expose deployment hello-nginx --type=LoadBalancer --port 80 --target-port 80
   ```

2. Visualisez les objets Kubernetes générés avec les commandes suivantes:

   ```
   # quelques commandes à tester
   k get nodes -o wide --show-labels
   k describe "ressources=node,pods, deploy,service..."
   k logs "pods"
   ```

   Ce mode est direct mais limité dans ses options. Il n'est pas orienté DevOps et "infrastructure as code". On peut lui préférer le mode "déclaratif" qui passe par l'édition d'un fichier yaml. Ces fichiers, cartes perforées des temps modernes sont envoyés à l'API de Kubernetes qui va les comprendre et s'efforcer de mettre en œuvre l'état demandé.

1. `https://github.com/fiatjaf/jiq`
2. voir aussi un article intéressant sur `https://atix.de/en/gitops-kubernetes-the-easy-way-part1/`
3. config objet=un objet K8S

Il existe aussi un mode impératif avec fichiers de commandes. Ce mode permet d'avoir de la persistence mais il n'est pas idempotent [1]. Deux personnes peuvent appliquées des changements et écraser les modifications de l'autre. Les ordres suivants sont caractéristiques de ce mode. (config objet=une configuration yaml):

```
# Ne passez pas ces commandes c'est juste une information sur le mode impératif qui ne fonctionnera que quand
# vous aurez un fichier yaml valide (voir question suivante)
kubectl create -f "config-objet.yaml"
kubectl replace -f "config-objet.yaml"
kubectl delete -f "config-objet.yaml"
```

3. Utilisez la commande 'kubectl create –dry-run=client -o yaml "objet"' afin de générer les manifests des objets créés en mode impératif précédemment.
4. Créer via la commande Kubectl et l'option "–from-literal" une "configmap" nommée maconfigmap avec les couples clefs/valeurs suivants:
   — k8s=leprésent
   — virt=legacy

   Affichez ensuite les valeurs.

   Solution: kubectl create configmap maconfigmap –from-literal=k8s=leprésent –from-literal=virt=legacy kubectl get cm -o yaml maconfigmap kubectl describe cm
5. Créer de même un secret nommé monsecret avec la valeur mdp=torototo et affichez le "décodé" (il est en base 64).

   Solution:
   kubectl create secret generic monsecret –from-literal=mdp=torototo kubectl get secret monsecret -o yaml echo 'dG9yb3RvdG8=' |base64 -d

   Solution:

4.2 Les "PODS" l'unité atomique de Kubernetes

Le pod est le plus petit objet déployable d'un cluster Kubernetes. Il est éphémère ("cattle" et pas "pet" !). Il est bâti à partir des namespaces, des cgroups et des capabilities.

Il est constitué par un ou plusieurs containers partageant les mêmes NameSpaces réseaux et IPC [2]. Les containers d'un POD sont aussi capables de partager entre eux des données au travers d'un partage sur l'hôte sur lequel ils s'exécutent. On distingue communément 3 patterns possibles [3]:

— Le plus connu est le pattern "side-car". Un container en "side-car" améliore ou étend les fonctionnalités d'un autre container. Par exemple les autres containers utilisent un partage recueillant les "logs" et qui va être traité par le container en "side-car".
— Le pattern "ambassadeur" va "proxyfier" les communications pour et à destination du POD. On peut imaginer un programme qui s'adresse à une IP et un port du POD (les containers dans un POD partage la même couche réseau) et qui en fait se connecte à un container jouant le role de proxy vers d'autres POD. Il est communément implémenté en "side-car". On peut lui déléguer les terminaisons SSL, l'authentification vers le micro-service du container...
— Le pattern "adapter" qui lui va normaliser le flux en sortie d'un POD par exemple en modifiant le format des logs exportées par un serveur web.

1. https://fr.wikipedia.org/wiki/Idempotence
2. Le "process namespace du container" peut aussi être partagé voir `https://kubernetes.io/docs/tasks/configure-pod-container/share-process-namespace/#understanding-process-namespace-sharing`
3. Voir Design patterns for container-based distributed systems de Brendan Burns et David Oppenheimer (Google)

Un pod peut servir à initialiser un autre pod (récupération d'un fichier et partage).

Un pod est dit "statique" quand il est piloté par le daemon Kubelet de l'hôte. On ne peut pas rajouter dynamiquement un container dans un "pod" existant ce qui est parfois limitant. [1]

Les Pods sont constitués de containers qui vont être mis en œuvre au travers d'un "Container Runtime" qui implémentent tous une interface(C.R.I.). "dockerd" n'est plus le runtime de Kubernetes, les acteurs du marché l'ayant trouvé trop général pour être efficient. Son sous-ensemble: containerd est par contre largement utilisé avec celui de RedHat : CRI-O.

En conséquence l'interface de commande de Docker ne peut plus être utilisé sur chaque noeud Kubernetes. La commmande "ctr" permet d'interagir avec les pods sur les nœuds K8S mais il y en a d'autres comme crictl et nerdctl.

4.2.1 Opérations basiques sur les PODS

1. Générez la configuration d'un pod debian à l'aide de la commande suivante:

   ```
   k run --dry-run=client  debianpod --image=registry.iutbeziers.fr/debianiut:latest -o yaml > monpremierpod.yml
   ```

   Créez ce manifest sur votre cluster afin de créer votre premier pod avec "kubectl create".
2. Vérifiez l'état de votre pod ?
3. Sur quel nœud votre pod s'exécute-t-il ?
4. Dans quel NameSpace votre Pod s'exécute-t-il ?
5. Accédez au container en utilisant "kubectl exec". Démarrez un server apache dans le container.
6. Pouvez-vous accéder au serveur web Apache qui s'exécute dans ce Pod ? Accéder au serveur Apache depuis votre client Kubernetes en utilisant "kubectl port-forward". Quel est le principe de cette commande ?

4.2.2 Manipulation des Pods

1. Supprimez le pod et recréez-le cette fois-ci en utilisant un "kubectl apply -f votre-manifeste". Quelle est la différence entre "apply" et "create" ?

   Solution:

1. En fait c'est possible au travers d'un container éphémère dans les dernières versions de Kubernetes voir `https://github.com/kubernetes-sigs/kind/issues/1210` pour la mise en œuvre en version alpha

+13 votes

answered May 29, 2019 by Yashica Sharma (10.6k points)

S. No.	Kubectl apply	Kubectl create
1.	It directly updates in the current live source, only the attributes which are given in the file.	It first deletes the resources and then creates it from the file provided.
2.	The file used in apply can be an incomplete spec	The file used in create should be complete
3.	Apply works only on some properties of the resources	Create works on every property of the resources
4.	You can apply a file that changes only an annotation, without specifying any other properties of the resource.	If you will use the same file with a replace command, the command would fail, due to the missing information.

FIGURE 1 – apply versus create

```
k create -f monpremierpod.yml
k get pod debianpod-o wide # voir le noeud
k describe pod/debianiut # dumper la configuration du pod
k exec -it pod-1 -- bash # Lancez un processus bash dans le pod
k port-forward pod/debianiut  8002:80 & # nattez le port pour y accéder depuis l'hôte.
k delete -f debianpod
k apply -f monpremierpod.yml


apiVersion: v1
kind: Pod
metadata:
creationTimestamp: null
labels:
run: debianiut
name: debianiut
spec:
containers:
- image: registry.iutbeziers.fr/debianiut:latest
name: debianiut
dnsPolicy: ClusterFirst
restartPolicy: Always


cat  ./monpremierpod.yml|k diff -f -
```

2. Recréer le POD avec en sus un container issu d'une image busybox. Rattachez-vous au pod via

"kubectl exec" en précisant avec l'option -c le container nom donné au container busybox.

Solution: k exec -it pod/debianpod -c shellc -it

3. Modifiez le manifeste du pod afin de limiter le cpu et la mémoire consommer par ce pod via des "limits".

Solution:

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    run: debianpod
    name: debianpod
spec:
  containers:
  - image: registry.iutbeziers.fr/debianiut:latest
    name: debianpod
    resources: {}
      limits:
        memory: "128Mi"
        cpu: "500m"
  - image: busybox
    name: shellc
    stdin: true
    tty: true
  dnsPolicy: ClusterFirst
  restartPolicy: Never
status: {}
```

4. Générer un autre pod sur un autre "node" K8s et effectuez un traceroute entre les deux pods. Quelles différences constatez-vous avec la gestion réseaux des containers Docker ?

Solution:

1. Tous les pods peuvent communiquer entre eux sans NAT
2. Les nœuds peuvent communiquer avec les pods (et inversement) sans NAT
3. L'IP vue par le conteneur est la même que l'IP vue par les composants externes
4. Des containers dans un pods partagent le même NetworkNameSpace

```
ubuntu@master:~$ k exec -it pods/debian1 bash
root@debian1:/# traceroute 10.42.3.4
traceroute to 10.42.3.4 (10.42.3.4), 30 hops max, 60 byte packets
1  10.42.2.1 (10.42.2.1)  0.053 ms  0.012 ms  0.009 ms
2  10.42.3.0 (10.42.3.0)  0.618 ms  0.555 ms  0.520 ms
3  10.42.3.4 (10.42.3.4)  0.724 ms  0.693 ms  0.658 ms
root@debian1:/# mtr
root@debian1:/# mtr 10.42.3.4
root@debian1:/# ip r
default via 10.42.2.1 dev eth0
10.42.0.0/16 via 10.42.2.1 dev eth0
10.42.2.0/24 dev eth0 proto kernel scope link src 10.42.2.4
```

4.2.3 Utilisation des utilitaires des "Container runtime" et de kubectl pour manipuler des containers

1. Lancez un process bash dans le pod créé avec kubectl.

Solution:

```
root@tp1k8s-worker2:/# ctr --namespace k8s.io c ls|grep -i 658117e89928b
658117e89928bcdebacdc25c9a974dfed9da579a0d585b3354bc122c8f0142de    sha256:c849a36f50593a69b2da055c28d474b2dcf0058124ae36f
root@tp1k8s-worker2:/# crictl exec -it 658117e89928bcdebacdc25c9a974dfed9da579a0d585b3354bc122c8f0142de sh
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
3: eth0@if17: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether b2:97:bf:d0:8a:3a brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 10.244.3.35/24 brd 10.244.3.255 scope global eth0
valid_lft forever preferred_lft forever
inet6 fe80::b097:bfff:fed0:8a3a/64 scope link
valid_lft forever preferred_lft forever
```

2. Lister les images des containers utilisées par votre cluster Kubernetes à l'aide de kubectl [1].

Solution:

1. voir `https://kubernetes.io/fr/docs/tasks/access-application-cluster/list-all-running-container-images/`

```
kubectl get pods --all-namespaces -o jsonpath="{..image}" |\
tr -s '[[:space:]]' '\n' |\
sort |\
uniq -c

      8 docker.io/kindest/kindnetd:v20210326-1e038dc5
      2 docker.io/rancher/local-path-provisioner:v0.0.14
      4 k8s.gcr.io/coredns/coredns:v1.8.0
      2 k8s.gcr.io/etcd:3.4.13-0
      2 k8s.gcr.io/kube-apiserver:v1.21.1
      2 k8s.gcr.io/kube-controller-manager:v1.21.1
      8 k8s.gcr.io/kube-proxy:v1.21.1
      2 k8s.gcr.io/kube-scheduler:v1.21.1
      2 k8s.gcr.io/metrics-server/metrics-server:v0.4.4

#ou
kubectl get pods --all-namespaces -o jsonpath="{.items[*].spec.containers[*].image}"|sed  's/ /\n/g'

#ou
kubectl get po -o 'custom-columns=name:metadata.name,images:spec.containers[*].image'
name        images
debianpod   registry.iutbeziers.fr/debianiut:latest,busybox



kubectl get po -o jsonpath='
   {range .items[*]}
   {"pod: "}
   {.metadata.name}
   {"\n"}{range .spec.containers[*]}
   {"\tname: "}
   {.name}
   {"\n\timage: "}
   {.image}
   {"\n"}
   {end}'

kubectl get pods debianpod -o jsonpath='{.spec.containers[*].name}'
kubectl get pods  -o jsonpath='{.spec.containers[*].name}'
```

3. Utilisez les utilitaires nerdctl [1] et crictl [2] pour récupérer la liste des images sur le noeud "worker" qui exécute le container. [1]

Solution:

1. `https://github.com/containerd/nerdctl`
2. Passez en root pour lancer cette commande
1. Il faut être root sur le noeud et utiliser l'option –namespace=k8s.io

```
#avec crictl

root@tp1k8s-control-plane:/# crictl images
IMAGE                                TAG                 IMAGE ID            SIZE
docker.io/kindest/kindnetd           v20210326-1e038dc5  6de166512aa22       54MB
docker.io/rancher/local-path-provisioner  v0.0.14             e422121c9c5f9       13.4MB
k8s.gcr.io/build-image/debian-base   v2.1.0              c7c6c86897b63       21.1MB
k8s.gcr.io/coredns/coredns           v1.8.0              296a6d5035e2d       12.9MB
k8s.gcr.io/etcd                      3.4.13-0            0369cf4303ffd       86.7MB
k8s.gcr.io/kube-apiserver            v1.21.1             6401e478dcc01       127MB
k8s.gcr.io/kube-controller-manager   v1.21.1             d0d10a483067a       121MB
k8s.gcr.io/kube-proxy                v1.21.1             ebd41ad8710f9       133MB
k8s.gcr.io/kube-scheduler            v1.21.1             7813cf876a0d4       51.9MB
k8s.gcr.io/pause                     3.4.1               0f8457a4c2eca       301kB


#avec ctr

root@tp1k8s-worker:/# ctr --debug   --namespace=k8s.io  task ls
TASK                                                    PID     STATUS
a2fa063cdb69e5dd73431fa140206b47eb6dc08d8eb3d4d0c029de5b47dbd4aa    436     RUNNING
62e7661cd9f91cf48bb0b1ef999a8c14739eb3d0e99965432c516429ff7f8b17    490     RUNNING
1fc05fbcb8c25731ddf2afe9b02a60914f5ccf4c9bdbc578fe458cb83307ff09    653     RUNNING
247248398b849c2e0c172d060827a77f7bbeef01ccabc502236a5b93407ec625    **36772**    RUNNING
b89414c9c880627f560b8092958bcdf80ccae556deb0006fec670d77c98951ab    3000    RUNNING
fd7f75288967a0f72b2f34552daf58fee8e0041a4b39998a50586941866b0f93    395     RUNNING
bf6ee24025681766c87c9c1fb16d717681226b604727e8647b6bf7c30afaa78b    795     RUNNING
fb1df54e05b0feca67e64987ed5741bd71912f0d098610a937b323d686847f55    836     RUNNING
ad3cc57a07ef68091fd03684cd5541eb6655480d194bbca0c89c6dc67dfba2f6    2534    RUNNING
root@tp1k8s-worker:/# ps -ef|grep bash
root       32874       0  0 15:26 pts/1    00:00:00 bash
root       **36772**    2514  0 15:54 ?        00:00:00 /bin/bash -c trap : TERM INT; sleep infinity & wait
root       41214   32874  0 16:25 pts/1    00:00:00 grep --color=auto bash
On fait le lien PID pour retrouver la "task" et on lance
root@tp1k8s-worker:/# ctr  --namespace=k8s.io  task exec --exec-id 3672 -t  247248398b849c2e0c172d060827a77f7bbeef01ccabc502236a5
l'exec-id peut être changé:

\footnote{Attention il faut utiliser un ID de container qui ne soit pas tronqué. Vous pouvez le récupérer via "ctr --namespace k8s.io c ls".L
ctr  --namespace=k8s.io  task exec --exec-id 36722 -t  247248398b849c2e0c172d060827a77f7bbeef01ccabc502236a5b93407ec625 /bin/bas

# avec nerdctl
for cont in $(docker ps |grep -v registry|awk '{print $1}'|grep -v CONTAINER); do docker exec -it $cont  bash -c "nerdctl --namespace k8
for cont in $(docker ps |grep -v registry|awk '{print $1}'|grep -v CONTAINER); do docker exec -it $cont  bash -c "nerdctl --namespace k8
root@tp1k8s-worker:/# nerdctl --namespace=k8s.io  exec -it 6C2407826480 bash
root@tp1k8s-worker:/# crictl exec -it 6b42407826480 bash
```

4. Lancez un process bash dans le pod avec crictl[1]. Faite de même avec l'utilitaire nerdctl.
5. Pour information il est possible de lancer un processus dans un container à l'aide de ctr l'utilitaire standard de containerd

   Les étapes sont les suivantes:

```
# On retrouve l'ID du container (attention contrairement à la commande docker on ne peut pas le tronquer pour sélectionner un container)
ctr   --namespace=k8s.io  c ls
```

1. voir `https://asciinema.org/a/179047` pour son usage

```
# On lance un shell dans notre container avec l'ID non tronqué du container debian
ctr --namespace=k8s.io task exec --exec-id 1223 247248398b849c2e0c172d060827a77f7bbeef01ccabc502236a5b93407ec625 \
/bin/bash -c 'ps -ef'
```

Pour nerdctl vous ultiliserez aussi le namespace k8s.io

4.2.4 Implémentation d'un pattern commun: l'init pod

1. installation d'un init pod Inspirez-vous de :
   `https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-initialization/`
   afin de créer un init pod qui récupère un fichier html sur un autre serveur web pour l'afficher.
   Vous utiliserez l'image apache [1] standard [2].

Solution:

```
apiVersion: v1
kind: Pod
metadata:
creationTimestamp: null
labels:
   run: webpod
   name: webpod
spec:
  containers:
    - name: apache
  image: httpd
  ports:
    - containerPort: 80
  volumeMounts:
  - name: workdir
      mountPath: /usr/local/apache2/htdocs
  initContainers:
  - name: install
  image: busybox
  command:
  - wget
  - "-O"
- "/tmp/mapage/README.html"
- https://raw.githubusercontent.com/pushou/multipass-k3sup/master/README.html
volumeMounts:
- name: workdir
mountPath: "/tmp/mapage"
dnsPolicy: Default
volumes:
- name: workdir
emptyDir: {}
```

```
# récupérer les containers par POD
kubectl get pods -o=custom-columns=NameSpace:.metadata.namespace,NAME:.metadata.name,CONTAINERS:.spec.containers[*].name
```

1. `https://hub.docker.com/_/httpd`
2. afin de simplifier l'utilisation de wget utilisez un site en clair comme `http://store.iutbeziers.fr`

5 Gestion du cycle de vie des applications dans Kubernetes

Le POD est un objet Kubernetes qui a des limitations:

— il ne permet pas le passage à l'échelle. La scalabilité va être assurée par l'objet "Deployment".

— Le POD est éphémère et il n'est donc pas d'adresse IP persistente sur le réseau. L'objet "service" créé une adresse IP persistante pour les PODS et leur "deployment".

5.1 Déploiement de son application Kubernetes

5.1.1 Les "NameSpaces"

Les "NamesSpaces" sont une brique essentielle de la construction des containers et donc aussi pour les "pods".Il est fortement recommandé de s'en servir pour isoler les applicatifs entre eux afin de regrouper les "pods" créant des micro-service pour une même application. Ils permettent aussi de séparer ce qui relève de l'infrastructure des containers ("namespace kube-system") de ce qui relève de l'applicatif.

1. Créez un namespace "applicatif".
2. Editez le "manifest" de ce namespace avec kubectl et modifier son nom en "applicatifs".

   Solution:

   ```
   k edit ns applicatifs
   ```

3. Listez l'ensemble des namespaces de ce cluster kubernetes puis l'ensemble des objets kubernetes de votre cluster.

   Solution:

   ```
   k get ns --all-namespaces
   ```

4. Créez un "pod" nginx dans ce "namespace" applicatifs et affichez-le avec kubectl.

5. A quoi sert le namespace kube-system ? le name-space default ?

   Solution: Kubernetes est constitué d'objets qui sont sensibles et qu'il convient de ne pas laisser entre les mains de n'importe qui d'ou ce namespace dédié qui n'est pas accessible à tous.

5.2 Déployer une application Python dans Kubernetes

5.2.1 "Deployment" de l'application

On va ré-utiliser l'application Python générée avec le TP Docker. L'U.R.I. /env permet de récupérer les variables d'environnement du contrôleur. Vous utiliserez la commande kubectl.

1. Créez un "deployment" avec "kubectl create" utilisant l'image registry.iutbeziers.fr/pythonapp:latest dans le namespace applicatifs.

   Solution: La séparation des namespaces est un fondamental des containers.

2. Dumper la configuration de ce "deployment" dans un fichier yaml. Editez ce fichier et expliquez les différents objets qui le compose.

   Solution:

```
k create ns applicatifs
k config set-context --current --namespace=applicatifs
k create deploy pythonapp --image registry.iutbeziers.fr/pythonapp:latest -n applicatifs --save-config

multipass@master:~/PythonAppK8s$ k get deploy pythonapp -o wide
NAME        READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES                                SELECTOR
pythonapp   1/1     1            1           65m   pythonapp    registry.iutbeziers.fr/pythonapp:latest   app=pythonapp
```

```
multipass@master:~/PythonAppK8s$ k get deploy pythonapp -o yaml > deployapppython.yml
apiVersion: extensions/v1beta1
kind: Deployment # Le genre
metadata:
annotations:
deployment.kubernetes.io/revision: "1"
creationTimestamp: "2019-08-24T19:58:11Z"
generation: 1
labels:
app: pythonapp
name: pythonapp
namespace: applicatifs
resourceVersion: "49374"
selfLink: /apis/extensions/v1beta1/namespaces/applicatifs/deployments/pythonapp
uid: 8052a021-c6a9-11e9-a0b1-5254006b839a
spec:
progressDeadlineSeconds: 600
replicas: 1
revisionHistoryLimit: 10
selector:
matchLabels:
app: pythonapp
strategy:
rollingUpdate:
maxSurge: 25%
maxUnavailable: 25%
type: RollingUpdate
template:
metadata:
creationTimestamp: null
labels:
app: pythonapp
spec:
containers:
- image: registry.iutbeziers.fr/pythonapp:latest
imagePullPolicy: Always
name: pythonapp
resources: {}
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
dnsPolicy: ClusterFirst
restartPolicy: Always
schedulerName: default-scheduler
securityContext: {}
terminationGracePeriodSeconds: 30
status:
availableReplicas: 1
conditions:
- lastTransitionTime: "2019-08-24T19:58:14Z"
lastUpdateTime: "2019-08-24T19:58:14Z"
message: Deployment has minimum availability.
reason: MinimumReplicasAvailable
status: "True"
type: Available
- lastTransitionTime: "2019-08-24T19:58:11Z"
lastUpdateTime: "2019-08-24T19:58:14Z"
message: ReplicaSet "pythonapp-7769c55546" has successfully progressed.
reason: NewReplicaSetAvailable
status: "True"
type: Progressing
observedGeneration: 1
readyReplicas: 1
replicas: 1
updatedReplicas: 1
```

Solution:

```
multipass@master:~/PythonA READY   STATUS    RESTARTS   AGE    IP          NODE     NOMINATED NODE   READINESS G
pythonapp-c864db666-zpcg8   1/1     Running   0          2m37s   10.42.0.18   master   <none>           <none>
multipass@master:~/PythonAppK8sppK8s$ k get   pods pythonapp-c864db666-zpcg8  -o wide
NAME                  $ curl 10.42.0.18:5000/env
{
  "FLASK_APP": "app.py",
  "FLASK_ENV": "development",
  "FLASK_RUN_FROM_CLI": "true",
  "GPG_KEY": "0D96DF4D4110E5C43FBFB17F2D347EA6AA65421D",
  "HOME": "/root",
  "HOSTNAME": "pythonapp-c864db666-zpcg8",
  "KUBERNETES_PORT": "tcp://10.43.0.1:443",
  "KUBERNETES_PORT_443_TCP": "tcp://10.43.0.1:443",
  "KUBERNETES_PORT_443_TCP_ADDR": "10.43.0.1",
  "KUBERNETES_PORT_443_TCP_PORT": "443",
  "KUBERNETES_PORT_443_TCP_PROTO": "tcp",
  "KUBERNETES_SERVICE_HOST": "10.43.0.1",
  "KUBERNETES_SERVICE_PORT": "443",
  "KUBERNETES_SERVICE_PORT_HTTPS": "443",
  "LANG": "C.UTF-8",
  "PATH": "/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
  "PYTHON_PIP_VERSION": "19.0.3",
  "PYTHON_VERSION": "3.7.2",
  "WERKZEUG_RUN_MAIN": "true",
  "WERKZEUG_SERVER_FD": "3"
}
multipass@master:~/PythonAppK8s$ curl 10.42.0.18:5000/whoami
{
  "ip_hote": "10.42.0.18"
}
```

3. En modifiant le "deployment" passer à trois "pods" Python.

5.2.2 Utilisation des labels

1. Affichez les pods dont le label est "app=pythonapp". Rajoutez un label "env=dev". Vérifiez la prise en compte de votre action. Enlevez le label app. Détruisez les pods labellisés "app=pythonapp". Supprimez au final le déploiement.

Solution:

```
multipass@master:~$ k label pod --selector app=pythonapp env=dev
pod/pythonapp-7769c55546-9gxt8 labeled
pod/pythonapp-7769c55546-crxct labeled
pod/pythonapp-7769c55546-ld2zg labeled
multipass@master:~$ k get pods --selector env=dev
NAME                         READY   STATUS    RESTARTS   AGE
pythonapp-7769c55546-9gxt8   1/1     Running   0          2d11h
pythonapp-7769c55546-crxct   1/1     Running   0          2d11h
pythonapp-7769c55546-ld2zg   1/1     Running   0          2d11h

multipass@master:~$ k label pod --selector env=dev  app-

multipass@master:~$ k delete pod --selector env=dev
pod "pythonapp-7769c55546-9gxt8" deleted
pod "pythonapp-7769c55546-crxct" deleted
pod "pythonapp-7769c55546-ld2zg" deleted

On peut tout labelliser y compris les nodes.
multipass@master:~$ k label  nodes node1 statutnode=prod
node/node1 labeled
multipass@master:~$ k label  nodes node2 statutnode=prod
node/node2 labeled
multipass@master:~$ k label  nodes node3 statutnode=maintenance
node/node3 labeled
multipass@master:~$ k get nodes --show-labels
NAME     STATUS   ROLES    AGE     VERSION        LABELS
master   Ready    master   5d17h   v1.14.6-k3s.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd6
node1    Ready    worker   5d16h   v1.14.6-k3s.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd6
node2    Ready    worker   5d16h   v1.14.6-k3s.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd6
node3    Ready    worker   5d16h   v1.14.6-k3s.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd6
```

2. Labellisez deux nodes "statusnode=maintenance". Drainer les nodes en maintenance en utilisant le label. [1]

Solution:

```
multipass@master:~$ k drain --force --ignore-daemonsets --selector statutnode=maintenance
node/node2 cordoned
node/node3 cordoned
```

3. En utilisant l'adresse obtenue lors de l'affichage du POD essayer d'accédez à l'application en CLI via l'utilitaire curl en dehors du cluster? que constatez-vous? essayez depuis un noeud du cluster.

Solution: L'adresse IP est celle du POD. Elle n'est accessible que depuis les noeuds du cluster.

1. utilisez –force pour forcer le kill des pods et –ignore-daemonsets.

```
multipass@master:~$ k get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
pythonapp-7769c55546-4thbf   1/1     Running   0          65m   10.42.3.6   node3    <none>           <none>
pythonapp-7769c55546-dq7s6   1/1     Running   0          65m   10.42.0.24  master   <none>           <none>
pythonapp-7769c55546-v96td   1/1     Running   0          65m   10.42.0.23  master   <none>           <none>
multipass@master:~$ curl 10.42.3.6:5000
Le Python c'est bon mangez en
multipass@master:~$
```

4. Modifiez le déployment pythonapp afin que le pod s'execute sur node3.

Solution:

```
multipass@master:~$ k get nodes node3  -o wide --show-labels
NAME    STATUS   ROLES    AGE    VERSION        INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION
node3   Ready    worker   3h35m  v1.14.6-k3s.1  10.162.140.57   <none>        Ubuntu 18.04.3 LTS   4.15.0-58-generic   containerd:

    spec:
containers:
- image: registry.iutbeziers.fr/pythonapp:latest
imagePullPolicy: Always
name: pythonapp
resources: {}
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
dnsPolicy: ClusterFirst
nodeSelector:
kubernetes.io/hostname: node3


metadata:
annotations:
deployment.kubernetes.io/revision: "1"
kubectl.kubernetes.io/last-applied-configuration: |
{"kind":"Deployment","apiVersion":"apps/v1","metadata":{"name":"debianiut","creationTimestamp":null,"labels":{"app":"debianiut"
```

5. Taintez le node "master" afin que ne soit plus schedulé à l'avenir de pods dessus.

Solution:

```
kubectl taint nodes master node-role.kubernetes.io/master=true:NoSchedule

node/master tainted
kubectl taint nodes master node-role.kubernetes.io/master=true:NoSchedule-
node/master untainted
```

6. Utilisez les fonctionnalités de taint & toleration pour qu'un pod soit schedulé sur node1 - (Taintez différemment tout vos noeuds et lancez un pod avec une toleration pour node1).Expliquez quel est le principe de "taint & toleration".
Pour lancer le pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-1
  labels:
    security: s1
spec:
 containers:
 - image: registry.iutbeziers.fr/debianiut:latest
   name: debianiut
   command: [ "/bin/bash", "-c", "--" ]
   args: [ "while true; do sleep infinity; done;" ]
 tolerations:
 - key: "node"
    operator: "Equal"
    value: "1"
    effect: "NoSchedule"
```

Solution:

```
      # No schedule Pod1
      kubectl taint nodes node1 cenoeud=jaimepaslesPods:NoSchedule
      node/node1 tainted
      kubectl taint nodes master cenoeud=jaimepaslesPodssurmaster:NoSchedule
      node/master tainted

      kubectl taint nodes node2 cenoeud=jaimepaslesPods2:NoSchedule
      node/node2 tainted
      kubectl taint nodes node3 cenoeud=jaimepaslesPods3:NoSchedule
      node/node3 tainted
      k apply -f ./tolerationpod.yaml

tolerations:
- key: "cenoeud"
  operator:  "Equal"
  value: "jaimepaslesPods"
  effect: "NoSchedule"
```

5.3 Réaliser une migration de version de l'image Python

Créez un nouvelle image Docker de l'application Python (en générant un nouveau tag pour aller vite) ou en modifiant l'application que vous pouvez récupérer sur: Vous pouvez charger l'image avec "kind load" sur l'ensemble des noeuds. Vous devrez indiquer comme "pull policy" de l'image "ifnotpresent" [1]

```
git clone https://registry.iutbeziers.fr:11443/pouchou/PythonAppK8s.git
```

1. Migrez vers cette version en déployant cette nouvelle version.
2. Suivez cette migration à l'aide de la commande rollout.

1. NOTE: The Kubernetes default pull policy is IfNotPresent unless the image tag is :latest or omitted (and implicitly :latest) in which case the default policy is Always. IfNotPresent causes the Kubelet to skip pulling an image if it already exists. If you want those images loaded into node to work as expected, please: don't use a :latest tag and / or: specify imagePullPolicy: IfNotPresent or imagePullPolicy: Never on your container(s).

Solution:

```
docker push luca6799/tp_k8s:pythonv2
kubectl set image deployment/pythonapp pythonapp=luca6799/tp_k8s:pythonv2 -n applicatifs
kubectl rollout status -w deployment/pythonapp
-n applicatifs
Waiting for deployment "pythonapp" rollout to finish: 1 out of 3 new
replicas have been updated...
Waiting for deployment "pythonapp" rollout to finish: 1 out of 3 new
replicas have been updated...
Waiting for deployment "pythonapp" rollout to finish: 1 out of 3 new
replicas have been updated...
Waiting for deployment "pythonapp" rollout to finish: 2 out of 3 new
replicas have been updated...
Waiting for deployment "pythonapp" rollout to finish: 2 out of 3 new
replicas have been updated...
```

6 Services

Un "deployment" permet de scaler une application mais ne rend pas l'application accessible même dans le cluster.

C'est la que la notion de **service** intervient: on expose les "deployment" pour les rendre accessibles.

Un service va fournir une adresse IP et un port, un enregistrement DNS et une liste de "endpoint" qui contiennent les IP des "pod". Ces pods sont sélectionnés grâce à un label déclaré dans le service.

On distingue 4 types de services:

- *ClusterIP* (type de service par défaut) utilisé pour communiquer en interne du cluster. (par exemple un service LDAP pour l'authentification).
- *NodePort*: il permet d'exposer le service sur un port aléatoire choisi entre 30000 et 32767 sur *chaque noeud* du cluster.(un port choisi pour tous les noeuds) Ce port sera joignable sur tous les noeuds du Cluster qu'il héberge un Pod éligible ou non.

  Les nœuds d'un cluster K8S dans le CLOUD n'ont pas toujours une IP publique: il faut un équilibreur de charge classique en frontal ou utiliser un VPN pour s'y connecter.
- *LoadBalancer*: Les Services de type LoadBalancer sont utilisés quand le cluster est hébergé chez un « Cloud Provider ».

  Le cluster va demander la création du load balancer en le faisant pointer sur tous les noeuds du cluster sur un port attribué aléatoirement sur chaque noeud (comme pour un NodePort).
  - Un Load Balancer par service est coûteux.
  - « On premise » si on veut de l'équilibrage de charge il faut "tricher" avec une solution comme metallb.
- *ExternalName*: C'est un type de service qui fait le lien vers l'extérieur du cluster. Les ressources sont par exemple hébergées en dehors d'un cluster.

6.1 Service de type NodePort

1. On va utiliser la commande impérative suivante afin de générer le manifeste du service de type node port.

   ```
   k expose deployment pythonapp --name pythonnp --target-port=5000 --port=9002 --type=NodePort --dry-run=client -o yaml > nodeport.ya
   ```

2. Modifiez le fichier en prenant comme nodeport 31234.

3. Quelle est la différence entre port/nodeport/target-port ? testez les nodes et le service.

Solution:

Le nouveau type d'objet créé est un service.

```
k expose deployment pythonapp  --name pythonnp --target-port=5000 --port=9002 --type=NodePort --dry-run > nodeport.yaml
```

```
apiVersion: v1
kind: Service
metadata:
creationTimestamp: null
labels:
app: pythonapp
name: pythonnp
spec:
ports:
- port: 9002 # port exposé dans le cluster sur l'IP du cluster
protocol: TCP
targetPort: 5000 # port exposé par le container
nodePort: 31234 # port exposé sur chaque node
selector:
app: pythonapp
type: NodePort
status:
loadBalancer: {}
```

```
# nodePort
Le nodePort est accessible sur chaque ip de chaque node
L'allocation est automatique si il n'est pas précisé
```

```
multipass@master:~$ k get nodes -o wide
NAME    STATUS   ROLES    AGE    VERSION        INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION
master  Ready    master   3d4h   v1.14.6-k3s.1  10.162.140.58   <none>        Ubuntu 18.04.3 LTS   4.15.0-58-generic   containerd:
node1   Ready    worker   3d3h   v1.14.6-k3s.1  10.162.140.144  <none>        Ubuntu 18.04.3 LTS   4.15.0-58-generic   containerd:
node2   Ready    worker   3d3h   v1.14.6-k3s.1  10.162.140.176  <none>        Ubuntu 18.04.3 LTS   4.15.0-58-generic   containerd:
node3   Ready    worker   3d3h   v1.14.6-k3s.1  10.162.140.34   <none>        Ubuntu 18.04.3 LTS   4.15.0-58-generic   containerd:
multipass@master:~$ curl http://10.162.140.58:31046
"Le Python c'est bon mangez en"
multipass@master:~$ curl http://10.162.140.144:31046
"Le Python c'est bon mangez en"
multipass@master:~$ curl http://10.162.140.176:31046
"Le Python c'est bon mangez en"
multipass@master:~$ curl http://10.162.140.34:31046
"Le Python c'est bon mangez en"
service.
k get svc -o wide -o json|jid |jq -r
```

```
# port
[]
{
    "nodePort": 31234,
    "port": 9002,
    "protocol": "TCP",
    "targetPort": 5000
}
]
```

```
multipass@master:~$ k get pods -o wide
NAME                      READY   STATUS    RESTARTS   AGE    IP          NODE     NOMINATED NODE   READINESS GATES
pythonapp-7769c55546-cd6hd   1/1     Running   0          3h51m   10.42.0.42   master   <none>           <none>

multipass@master:~$ curl http://10.42.0.42:5000
"Le Python c'est bon mangez en"
```

```
# targetPort
C'est le port sur lequel l'application écoute. L'équivalent du expose dans le DockerFile.
```

```
multipass@master:~$ curl http://10.43.181.245:9002
"Le Python c'est bon mangez en"
multipass@master:~$
```

4. Quels sont les inconvénients d'un service Node-Port ?Que manque-t-il dans cette architecture ?

Solution: Le node port est attribué sur le range 30000 - 32767 ce qui n'est pas souhaitable pour du grand public. Il faut donc en frontal un load balancer externe. Il est donc utile "on premise" ou combiné avec l'autre service de type lb.

5. Récupérez les "endpoints" reliés au service.

Solution:

```
k get endpoints
NAME         ENDPOINTS                                            AGE
kubernetes   172.17.0.4:6443                                      11h
pythonapp    10.244.1.24:5000,10.244.2.24:5000,10.244.3.22:5000   62m
```

6. Complétez l'architecture avec un équilibreur de charge externe (un haproxy à installer par package) ?
7. Dessinez un schéma de cette architecture sur papier.

Solution:

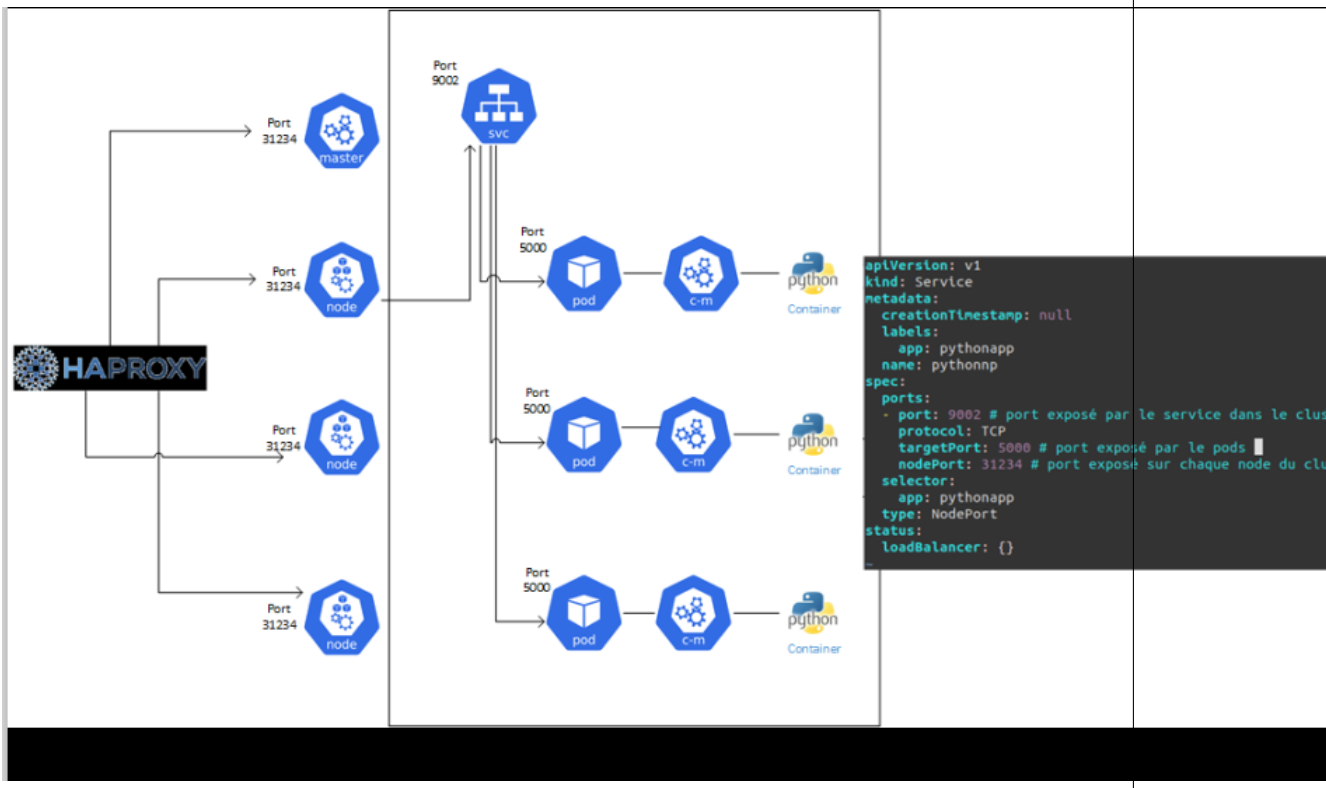

FIGURE 2 – Service NodePort

6.2 Service de type LoadBalancer

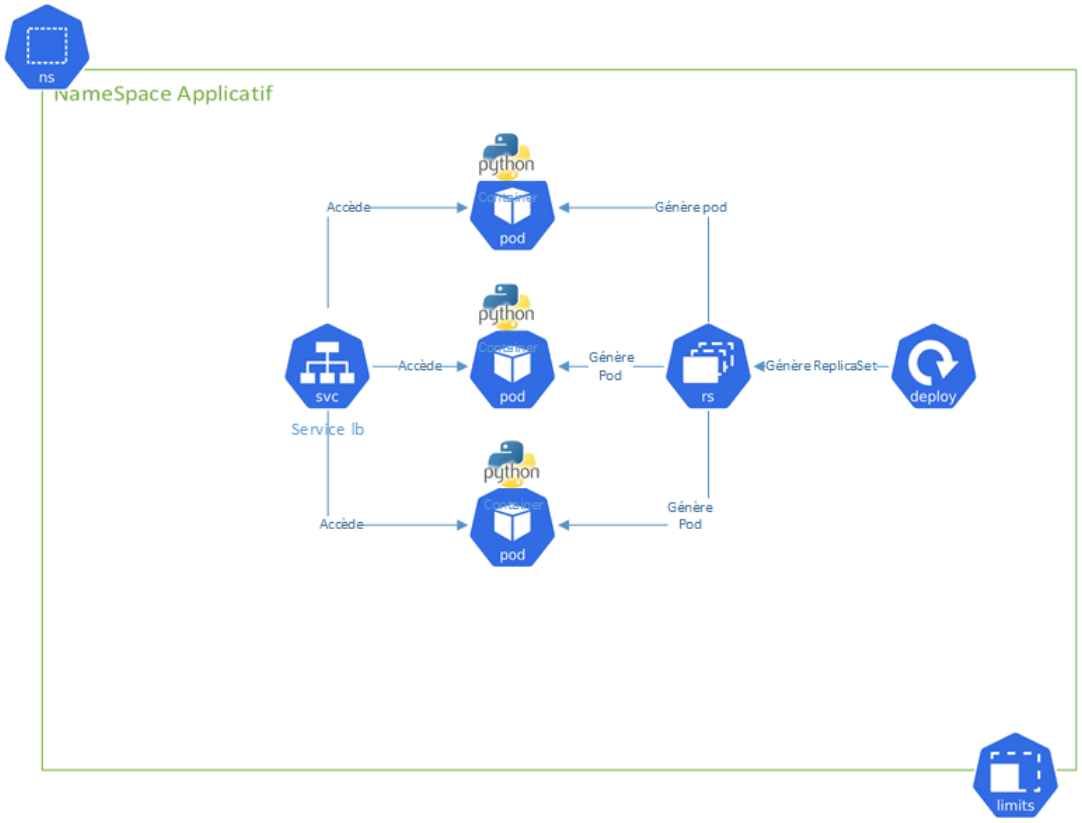

FIGURE 3 – Service

1. Créer un service de type loadbalancer. Pourquoi son EXTERNAL-IP est elle pending ?

Solution: Un loadbalancer se verra attribuer une adresse ip externe par le provider CLOUD. On Premise il faut un outil supplémentaire pour rendre accessible l'IP du service en loadbalancing ou il faut utiliser un INGRESS qui routera le trafic vers les PODS. Cet ingress est connecté à l'API Kubernetes et tiens compte de la vie et de la mort des pods.

```
multipass@master:~$ k get svc
NAME       TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)      AGE
pythonapp  LoadBalancer  10.43.52.5   <pending>     80:31503/TCP  3m14s

k expose deployment pythonapp --target-port=5000 --port=80  --type=LoadBalancer
multipass@master:~$ curl 10.43.52.5/env|grep HOSTNAME
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload   Total   Spent    Left  Speed
100  1136  100  1136    0     0   123k      0 --:--:-- --:--:-- --:--:--  138k
"HOSTNAME": "pythonapp-7769c55546-9gxt8",
multipass@master:~$ curl 10.43.52.5/env|grep HOSTNAME
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload   Total   Spent    Left  Speed
100  1136  100  1136    0     0   184k      0 --:--:-- --:--:-- --:--:--  221k
"HOSTNAME": "pythonapp-7769c55546-ld2zg",
multipass@master:~$ curl 10.43.52.5/env|grep HOSTNAME
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload   Total   Spent    Left  Speed
100  1136  100  1136    0     0   221k      0 --:--:-- --:--:-- --:--:--  221k
"HOSTNAME": "pythonapp-7769c55546-ld2zg",
multipass@master:~$ curl 10.43.52.5/env|grep HOSTNAME
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
Dload  Upload   Total   Spent    Left  Speed
100  1136  100  1136    0     0  75733      0 --:--:-- --:--:-- --:--:-- 81142
"HOSTNAME": "pythonapp-7769c55546-crxct",
```

2. Installer et utiliser Metallb [1] en mode L2 afin de rendre le service loadbalancer accessible depuis l'extérieur. [2]

   Rajouter le ConfigMap suivante en remplaçant la plage d'adresse par celle de vos noeuds k8s.

```
apiVersion: v1
kind: ConfigMap
metadata:
namespace: metallb-system
name: config
data:
config: |
address-pools:
- name: default
protocol: layer2
addresses:
-  172.18.0.250- 172.18.0.254
```

7 Ingress

Une autre façon d'accéder à une application c'est d'utiliser un "ingress controller". Ce n'est pas un service comme précédemment mais un reverse proxy. Ce contrôleur se base sur l'en-tête http "Host" afin d'aiguiller la requête vers le bon "backend". "*haproxy-ingress*" est une des solutions possibles pour un "ingress controller". "*Nginx*" ou "*Traeffik*" en sont des autres.

1. Installer helm en suivant `https://helm.sh/docs/intro/install/`

1. `https://metallb.universe.tf/`
2. degug de metallb via "kubetail -l component=speaker -n metallb-system" et " k logs -f controller-57f648cb96-khvk6 -n metallb-system"

2. Déployez le package de l' "ingress controller haproxy" en vous aidant de:
   `https://www.haproxy.com/fr/blog/use-helm-to-install-the-haproxy-kubernetes-ingress-controller/`
   et de
   `https://hub.helm.sh/charts/mirusresearch/haproxytech-haproxy-ingress`
   Le service du controller haproxy sera de type "LoadBalancer" afin de permettre à mettallb de publier l'adresse sur le réseau d'hôte.
3. Utilisez cet ingress afin de rendre accessible l'application Python.

Solution:

```
helm install moncontrollerhap  haproxytech/kubernetes-ingress \
--set controller.kind=DaemonSet \
--set controller.daemonset.useHostPort=true \
--set-string "controller.config.ssl-redirect=false" \
--set controller.service.type=LoadBalancer
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
creationTimestamp: null
labels:
run: pythonapp
name: pythonapp
spec:
replicas: 3
selector:
matchLabels:
run: pythonapp
template:
metadata:
creationTimestamp: null
labels:
run: pythonapp
spec:
containers:
- image: registry.iutbeziers.fr/pythonapp:latest
name: pythonapp
ports:
- containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
labels:
run: pythonapp
name: pythonapp
annotations:
haproxy.org/check: "enabled"
haproxy.org/forwarded-for: "enabled"
haproxy.org/load-balance: "leastconn"
haproxy.org/pod-maxconn: "50"
spec:
selector:
run: pythonapp
ports:
- name: port-1
port: 80
protocol: TCP
targetPort: 5000
---
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
name: web-ingress
namespace: default
spec:
rules:
- host: pythonapp.172.18.0.5.nip.io
http:
paths:
- path: /
backend:
serviceName: pythonapp
servicePort: 80
```

Les annotations permettent de développer des fonctionnalités rapidement mais sont susceptibles d'évoluer avec les versions des objet k8s.

4. Vérifiez que l'"ingress controller" fonctionne:

```
http  pythonapp.172.18.0.250.nip.io/env
```

   NB: le domaine nip.io permet de résoudre tous les FQDN en adresse IP.

5. Visionnez dans votre navigateur la page de statistiques de l'"HAPROXY ingress". Faite un test de charge avec la commande ab (apachebench);

   Solution: Soit sur le node maitre, soit sur l'hôte (dnat sur le port 1936) http://172.18.0.5:1936/

6. Vérifiez que l'"Ingress Controller" s'adapte si des pods disparaissent.
7. En suivant `https://github.com/jcmoraisjr/haproxy-ingress/tree/master/examples/deployment` générez un certificat auto-signé pour l'application Python et vérifiez en le bon fonctionnement.

   Solution:
   https://pythonapp.172.18.0.5.nip.io

8. Dessinez un schéma de l'architecture Ingress.

   Solution:

```
http get http://pythonapp.172.18.0.5.nip.io/env
http get  172.18.0.5/env  Host:pythonapp.172.18.0.5.nip.io
curl -H 'Host:python.cbon' 172.18.0.5/env

VIP metallb port 80 --> service ingress controller --> service Python ---> pod 5000
---> pod 5000
---> pod 5000
```

8 Stockage persistent et non réparti

La persistence des données est une difficulté commune des containers.

Les "ConfigMaps" et les "Secrets" permettent d'assurer la persistence des données mais pour des configurations et mots de passes. "HostPath" [1] permet de partager des fichiers ou des répertoires de l'hôte. "local-persistent-volumes" [2] permet de partager des disques.

Kind installe le "Rancher Localpath provisionner" [3] qui permet de provisionner du stockage avec un volume local dynamiquement réservé.

Bien sur l'utilisation d'un stockage réparti et persistent (NFS, CEPH, GLUSTERFS, ...) est possible. Il existe des solutions comme LongHorn, Rook , OpenEBS... qui installent du stockage persistent et réparti pour le cluster.

L'administateur du cluster va définir des classes de stockages (ssd, capacitif, nfs..). Le développeur lors de la création de ses pods fait une demande de volumes qui va s'appuyer sur un "persistent volume claim" qui pourra "piocher" de l'espace dans la classe de stockage défini précédemment.

Ainsi le développeur ne fait que demander de l'espace et ne s'occupe pas de sa gestion.

Dans votre environnement kind partage avec les noeuds Docker un espace dans votre directory "creation-cluster-kind" qui s'appelle shared-storage. Sa création est gérée par Kind et vous n'avez pas à la créer (c'est du Docker pas du Kubernetes).

Par défaut dans chaque noeud Docker, le répertoire /var/local-path-provisioner est utilisé pour stocker les volumes des PODS.

1. `https://kubernetes.io/docs/concepts/storage/volumes/#hostpath`
2. `https://kubernetes.io/blog/2018/04/13/local-persistent-volumes-beta/`
3. `https://github.com/rancher/local-path-provisioner`

1. Quel est l'intérêt de faire du stockage local ? du stockage réparti.

   Solution: La performance pour le local. La disponibilité pour le réparti

2. Retrouvez la classe de stockage via kubectl ? y a t il des PV ou des PVC ?

   Solution:

   ```
   k get pv -A
   k get pvc -A
   k get storageclass -A
   ```

3. En vous inspirant de la documentation Rancher Créez un PVC qui pointera sur la StorageClass "standard".

   Solution:

   ```yaml
   kind: PersistentVolumeClaim
   apiVersion: v1
   metadata:
     name: pvc-localpath
   spec:
     storageClassName: standard
     accessModes:
       - ReadWriteOnce
     resources:
       requests:
         storage: 2Gi
   ```

4. Créez un "deployment" nginx et donner le chemin /usr/share/nginx/html comme volume persistent. Vérifiez que le volume est conservé si vous supprimez le "deployment" ?pourquoi ne peut-on pas supprimer le pvc avant le pv ?

   NB: quelques commandes utiles pour vérifier le fonctionnement de la demande de stockage:

   ```
   k -n local-path-storage logs -f -l app=local-path-provisioner
   k get persistentvolumeclaim
   k get persistentvolume
   ```

   Solution:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    run: pv-deploy
  name: pv-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      run: pv-deploy
  template:
    metadata:
      creationTimestamp: null
      labels:
        run: pv-deploy
    spec:
      volumes:
        - name: pv-storage-localpath
          persistentVolumeClaim:
            claimName: pvc-localpath
      containers:
      - image: nginx:stable
        name: pv-nginx-deploy
        ports:
        - containerPort: 80
        volumeMounts:
          - mountPath: "/usr/share/nginx/html"
            name: pv-storage-localpath
```

9 Utiliser Helm pour installer un dns unbound dans votre cluster.

1. Recherchez la charte pour unbound/stable et installez là:
2. Configurez un fichier yaml ([1]) qui permettra une résolution recursive depuis les hôtes Docker.

Solution:

```
helm search hub unbound
helm install stable/unbound  -f ./configmapunbound.yaml --namespace default --generate-name
```

1. voir `https://hub.helm.sh/charts/stable/unbound`

```
allowedIpRanges:
- "10.0.0.0/8"
- "172.18.0.0/16"
- "192.168.0.0/16"
localRecords:
- name: "kind-control-plane.local"
ip: "172.18.0.5"
- name: "kind-worker.local"
ip: "172.18.0.4"
- name: "kind-worker2.local"
ip: "172.18.0.2"
- name: "kind-worker3.local"
ip: "172.18.0.3"

forwardZones:
- name: "."
forwardIps:
- "8.8.8.8"
- "1.1.1.1"
```

10 Containers éphémères pour débugguer

"Ephemeral container" [1] est une fonctionnalité qui permet de lancer un container dynamiquement dans le même POD que le container qui pose problème. Votre cluster kind dispose de cette "feature". [2] Testons la:

```
kubectl run mon-app --image=k8s.gcr.io/pause:3.1 --restart=Never
kubectl debug -it mon-app --image=busybox --target=mon-app
```

1. Expliquez le mode de fonctionnement de cette feature et les options ci-dessus.
2. Lancez un container nginx et débugguer le de la même façon.
3. Quel est le pid du process nginx ? pouvez-vous lister la configuration du container nginx sous cat /proc/pid-trouvé-par-ps/root/ ?
4. Rajoutez l'option –share-processes et –copy-to en remplacement de –target avec kubectl debug pour accéder totalement au container nginx. Affichez la configuration du fichier nginx.con.

   Solution: https://martinheinz.dev/blog/49 As you probably noticed, in addition to –share-processes we also included –copy-to=new-pod-name because - as was mentioned - we need to create a new pod whose name is specified by this flag. If we then list running pods from another terminal we will see the following:

1. `https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/`
2. voir `https://martinheinz.dev/blog/49`

```
kubectl run mon-app --image=nginx --restart=Never
kubectl debug -it mon-app --image=busybox --share-processes --copy-to=mon-app
cat /proc/7/root/etc/nginx/nginx.conf
user  nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

echo "liste des containers du pod debianpod"
kubectl get pods debianpod -o jsonpath='{.spec.containers[*].name}'
echo "on s'attache"
kubectl attach -it debianpod -c debugger
....
```

11 Installation d'utilitaires..utiles

1. Installez et testez Krew.
2. Installez et testez kubetail.
3. Installez et testez stern.
4. Installez et testez kubectx.
5. Installez et testez contena-lentz (installation via snap).
6. Installez et testez le tableau de bord de Kubernetes.