Introduction à Kubernetes

Warning: ce qui est écrit en italique est ma propre interprétation, ce n'est peut-être pas toujours exact. ## 1. Concepts ### 1.1 Applications monolithiques et microservices ### 1.2 Les conteneurs ### 1.3 Orchestration des conteneurs ## 2 Architecture ### 2.1 Architecture de Kubernetes ### 2.2 Pods et Nodes #### 2.2.1 Pods #### 2.2.2 Nodes ## 3 Utilisation ### 3.1 Les différents types d'installation du cluster #### 3.1.1 Bare metal #### 3.1.2 Dans un environnement Cloud #### 3.1.3 Minikube ## 3.2 Les blocs de base de K8s (nodes, pods, label, services, volumes, namespaces, ...) #### 3.2.1 Nodes #### 3.2.2 Pods #### 3.2.3 Services #### 3.2. ### 3.3 Gestion du cluster Kubernetes #### 3.3.1 L'environnement local minikube #### 3.3.2 deployer une application #### 3.3.3 Explorer une application #### 3.3.4 Utiliser les services pour exposer son application #### 3.3.5 Mettre à l'échelle une application #### 3.3.6 Mettre à jour une application

Introduction à Kubernetes

Kubernetes (abrégé K8s) est un mot grec qui veut dire "gouvernail". *On peut faire l'analogie avec le conteneur de marchandise qui a révolutionné la logistique en créant un standard: on peut dire que Docker a également établi un standard dans le packaging d'application, et Kubernetes quand à lui pourrait être comparé à un port automatisé (par exemple via la 5G en Chine https://www.youtube.com/watch?v=GhIdUWpTKvE).*

Kubernetes permet de faire du calcul distribué sur un cluster. C'est une sorte de linux réparti sur plusieurs serveurs, dans lequel les conteneurs jouent le rôle de processus: tout comme l'OS gère la répartition des ressources entre les différents processus pour éxecuter des application en parallèle, Kubernetes gère la répartition des ressources entre les conteneurs. En plus de cela, des sondes sont intégrées dans k8s pour surveiller en permanence l'état de santé des conteneurs: si ces derniers disfonctionnent, ils sont tout simplement détruits et recrées pour aboutir à l'état souhaité par l'administrateur, état qu'il a défini dans des fichiers de configuration. Le rôle de l'administrateur kubernetes est différent de celui d'administrateur de vm, qui doit lui gérer lui-même l'état de santé des vm, le stockage et l'infrastructure réseau.

1. Concepts

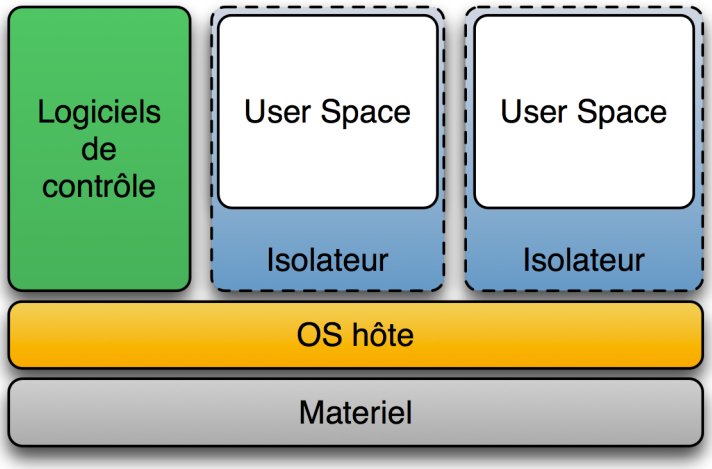
1.1 Applications monolithiques et microservices

On peut faire le parallèle entre la programmation orientée objet et les microservices: les deux concepts sont basés sur la **modularité** et le minimum de couplage entre les parties d'une application. Un système basé sur un mainframe "legacy" (typiquement dans le secteur bancaire) développé dans des langages plus anciens(tels que Cobol) est un exemple d'application monolithique. Les

applications modernes développées pour le cloud (typiquement les GAFAM, Netflix, Uber, toutes les grosses entreprises qui ont besoin de haute disponibilité et de résilience) sont des microservices. Pour simplifier, on pourrait prendre l'image d'un immense rocher versus des petits galets: avec le temps, les nouvelles fonctionnalités et améliorations ont ajouté de la compléxité au code du monolithe, rendant le développement plus difficile et augmentant les durées des mises à jour, alors qu'il est beaucoup plus facile de collecter des cailloux dans un sceau et de les transporter là où c'est nécessaire. Cependant, l'architecture distribuée des microservices est plus complexe.

1.2 Comparaison des diférents type de virtualisation

Un point de départ usuel pour expliquer les conteneurs est de les comparer aux machines virtuelles. Il y a différentes techniques de virtualisation mais qui peuvent être complémentaires: typiquement faire tourner des conteneurs dans un parc de machines virtuelles.



- Isolateurs
  Avantages des isolateurs:

une très forte densité: on peut déployer beaucoup plus de conteneurs que de machines virtuelles. Par exemple j'ai pû déployer 200 réplications d'un conteneur docker basé sur une image nginx, en 10 secondes, pourtant sur un laptop peu puissant (core i3 dualcore et 6 Go de RAM).

performances: très peu d'overhead (temps passé par le système à ne rien faire d'autre que de se gérer lui-même). empreinte réduite: un conteneur est vraiment comme une *enveloppe* autour de l'application. Par exemple une application de test hello-world de taille 1,8Mio peut être packagée dans un conteneur de taille 1,9Mio!

Outre la densité et les performances pendant l'utilisation, on améliore le temps

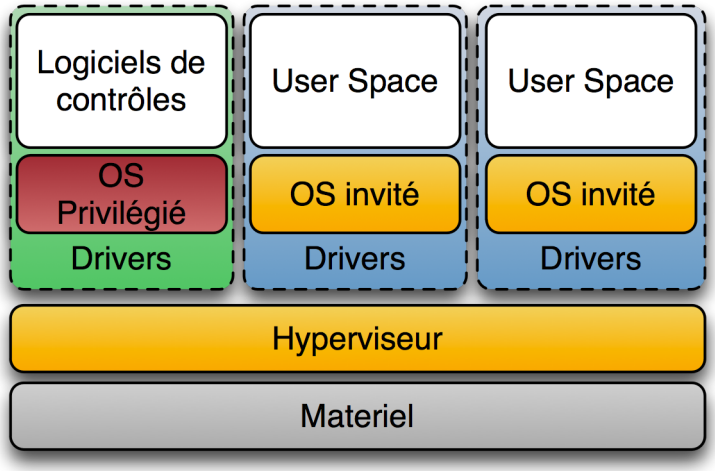
de *cold start*, et, plus important encore on réduit la surface d'attaque au strict minimum (il y a moins de vulénrabilités potentielles dans un binaire que dans un Linux, même un Linux minimaliste).

Cependant, on ne peut pas vraiment parler de virtualisation de système d'exploitation.

Exemples: chroot, BSD Jail, OpenVZ, Docker (qui s'appuie sur LXC - Linux Container - et les namespaces, fonctionnalités du noyau Linux) La principale différence entre un conteneur et une machine virtuelle est que le conteneur **utilise le noyau de l'hôte**, ils sont donc très légers et très faciles et rapides à déployer/détruire/redéployer.

- Les hyperviseurs de type 2 emulateur Utile pour les développeurs et les testeurs, en formation ou pour découvrir de nouveaux OS. Cette solution isole bien les OS invités mais les performances sont bien moindres que dans le cas des hyperviseurs de type 1, car dans le type 2, l'hyperviseur est un programme lourd qui tourne sur l'OS de l'hôte.

Exemples: VMware Fusion/Workstation, VirtualBox, QEMU, ...



- Les hyperviseurs de type 1
  C'est la solution adoptée en entreprise. L'hyperviseur de type 1 est un noyau système (Linux) très léger (juste avec une busybox) et optimisé pour gérer les accès des noyaux d'OS invités à l'architecture sous-jacente. Inconvénient: plus onéreux.

Exemples: VMware ESXi/vSphere, Hyper-V, KVM, Xen, ...

1.3 Conteneurs

Qu'est ce qu'un conteneur? **********
Avant de plonger dans l'orchestration de conteneurs, examinons d'abord ce que sont les conteneurs.

Les conteneurs sont une méthode centrée sur les applications pour fournir des applications hautement performantes et évolutives sur toute infrastructure de votre choix. Les conteneurs sont les mieux adaptés pour fournir des microservices en fournissant des environnements virtuels portables et isolés pour que les applications s'exécutent sans interférence d'autres applications en cours d'exécution.

Les microservices sont des applications légères écrites dans divers langages de programmation modernes, avec des dépendances, des bibliothèques et des exigences environnementales spécifiques. Pour garantir qu'une application dispose de tout ce dont elle a besoin pour s'exécuter correctement, elle est empaquetée avec ses dépendances.

Les conteneurs encapsulent les microservices et leurs dépendances mais ne les exécutent pas directement. Les conteneurs exécutent des images de conteneur.

Une image de conteneur regroupe l'application avec son environnement d'exécution, ses bibliothèques et ses dépendances, et elle représente la source d'un conteneur déployé pour offrir un environnement exécutable isolé pour l'application. Les conteneurs peuvent être déployés à partir d'une image spécifique sur de nombreuses plates-formes, telles que les postes de travail, les machines virtuelles, le cloud public, etc.

1.4 Orchestration des conteneurs

La gestion d'un petit nombre de conteneurs (pour les développeurs et les testeurs) n'a pas besoin d'un orchestrateur, mais dès qu'on passe dans des environnements de production il faut une Infrastructure As A Service comme K8s, ou Plateform As A Service (OpenShift, Rancher, ...).

- Docker swarm: La solution d'orchestration de conteneurs de la société Docker est efficace et simple à mettre en place, mais elle a ses limites. Si un conteneur tombe en panne par exemple, c'est l'administrateur qui doit s'occuper manuellement de corriger le problème, et pendant ce temps, l'application est indisponible. Par contre Kubernetes garantit la disponibilité en s'occupant lui-même de corriger ce problème. Ce n'est qu'un exemple de ce que K8s fait de mieux que ses concurrents (Docker Swarm, Mesos,...), on pourrait citer aussi la gestion de l'infrastructure réseau.
- Kubernetes: À l'origine, Kubernetes vient du projet Borg développé par Google depuis 2005, qu'ils ont ensuite rendu opensource en 2015 sous le nom de Kubernetes, et qui est depuis développpé par de nombrexu acteurs de l'opensource (RedHat, VMWare, ...). K8s est écrit en langage Go, langage développé par Ken Thompson le créateur d'Unix. *Go est un langage efficace pour la programmation parallèle, et donc bien adapté à un projet tel que K8s.*

****

L'orchestration des conteneurs: **********

Avec les entreprises qui conteneurisent leurs applications et les déplacent vers le cloud, il existe une demande croissante de solutions d'orchestration de conteneurs. Bien qu'il existe de nombreuses solutions disponibles, certaines sont de simples redistributions d'outils d'orchestration de conteneurs bien établis, enrichis de fonctionnalités et, parfois, de certaines limitations de flexibilité.

Bien que non exhaustive, la liste ci-dessous fournit quelques outils et services d'orchestration de conteneurs disponibles aujourd'hui:

Amazon Elastic Container Service Amazon Elastic Container Service (ECS) est un service hébergé fourni par Amazon Web Services (AWS) pour exécuter des conteneurs Docker à grande échelle sur son infrastructure. Instances de conteneur Azure Azure Container Instance (ACI) est un service d'orchestration de conteneur de base fourni par Microsoft Azure. Azure Service Fabric Azure Service Fabric est un orchestrateur de conteneurs open source fourni par Microsoft Azure. Kubernetes Kubernetes est un outil d'orchestration open source, initialement lancé par Google, qui fait aujourd'hui partie du projet Cloud Native Computing Foundation (CNCF). Marathon Marathon est un framework pour exécuter des conteneurs à grande échelle sur Apache Mesos. Nomade Nomad est l'orchestrateur de conteneurs et de charges de travail fourni par HashiCorp. Essaim de dockers Docker Swarm est un orchestrateur de conteneurs fourni par Docker, Inc. Il fait partie de Docker Engine.Avec les entreprises qui conteneurisent leurs applications et les déplacent vers le cloud, il existe une demande croissante de solutions d'orchestration de conteneurs. Bien qu'il existe de nombreuses solutions disponibles, certaines sont de simples redistributions d'outils d'orchestration de conteneurs bien établis, enrichis de fonctionnalités et, parfois, de certaines limitations de flexibilité.

Bien que non exhaustive, la liste ci-dessous fournit quelques outils et services d'orchestration de conteneurs disponibles aujourd'hui:

Amazon Elastic Container Service Amazon Elastic Container Service (ECS) est un service hébergé fourni par Amazon Web Services (AWS) pour exécuter des conteneurs Docker à grande échelle sur son infrastructure. Instances de conteneur Azure Azure Container Instance (ACI) est un service d'orchestration de conteneur de base fourni par Microsoft Azure. Azure Service Fabric Azure Service Fabric est un orchestrateur de conteneurs open source fourni par Microsoft Azure. Kubernetes Kubernetes est un outil d'orchestration open source, initialement lancé par Google, qui fait aujourd'hui partie du projet Cloud Native Computing Foundation (CNCF). Marathon Marathon est un framework pour exécuter des conteneurs à grande échelle sur Apache Mesos. Nomade Nomad est l'orchestrateur de conteneurs et de charges de travail fourni par HashiCorp. Essaim de dockers Docker Swarm est un orchestrateur de conteneurs fourni par Docker, Inc. Il fait partie de Docker Engine.

Pourquoi utiliser un orchestrateur de conteneurs? **********

Bien que nous puissions gérer manuellement quelques conteneurs ou écrire des scripts pour gérer le cycle de vie de dizaines de conteneurs, les orchestrateurs facilitent grandement les choses pour les opérateurs, en particulier lorsqu'il s'agit de gérer des centaines et des milliers de conteneurs s'exécutant sur une infrastructure mondiale.

La plupart des orchestrateurs de conteneurs peuvent:

- Regrouper les hôtes lors de la création d'un cluster
- Planifier les conteneurs à exécuter sur les hôtes du cluster en fonction de la disponibilité des ressources
- Permettre aux conteneurs d'un cluster de communiquer entre eux quel que soit l'hôte sur lequel ils sont déployés dans le cluster
- Lier des conteneurs et des ressources de stockage
- Regroupez des ensembles de conteneurs similaires et liez-les à des constructions d'équilibrage de charge pour simplifier l'accès aux applications conteneurisées en créant un niveau d'abstraction entre les conteneurs et l'utilisateur
- Gérez et optimisez l'utilisation des ressources Permettre la mise en œuvre de stratégies pour sécuriser l'accès aux applications exécutées à l'intérieur des conteneurs.

Avec toutes ces fonctionnalités configurables mais flexibles, les orchestrateurs de conteneurs sont un choix évident lorsqu'il s'agit de gérer des applications conteneurisées à grande échelle. Dans ce cours, nous explorerons Kubernetes, l'un des outils d'orchestration de conteneurs les plus demandés actuellement.

2 Architecture

2.1 Architecture de Kubernetes

Le noeud **Master** est aussi appelé **Control Plane**.

2.2 Les différents types d'installation du cluster

2.2 Pods et Nodes

Kubernetes Pods **********

Lorsqu-on a créé un déploiement dans le module 2, Kubernetes a créé un pod pour héberger votre instance d'application. Un pod est une abstraction Kubernetes qui représente un groupe d'un ou plusieurs conteneurs d'application (tels que Docker) et certaines ressources partagées pour ces conteneurs. Ces ressources comprennent:

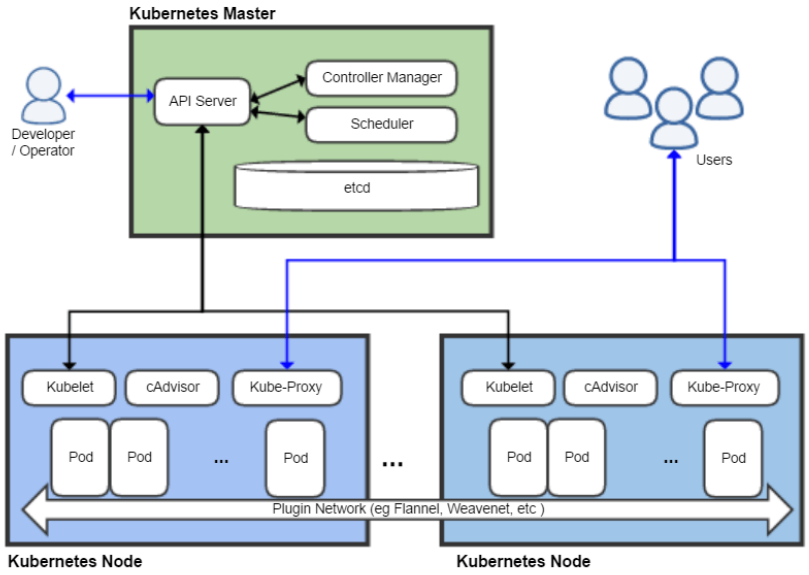


Figure 1: kubernetes

Stockage partagé, sous forme de volumes Mise en réseau, en tant qu'adresse IP de cluster unique Informations sur la façon d'exécuter chaque conteneur, telles que la version de l'image du conteneur ou des ports spécifiques à utiliser

Un pod modélise un «hôte logique» spécifique à une application et peut contenir différents conteneurs d'application qui sont relativement étroitement couplés. Par exemple, un pod peut inclure à la fois le conteneur avec votre application Node.js et un autre conteneur qui alimente les données à publier par le serveur Web Node.js. Les conteneurs d'un pod partagent une adresse IP et un espace de port, sont toujours colocalisés et co-programmés, et s'exécutent dans un contexte partagé sur le même noeud.

Les pods sont l'unité atomique sur la plate-forme Kubernetes. Lorsque nous créons un déploiement sur Kubernetes, ce déploiement crée des pods avec des conteneurs à l'intérieur (par opposition à la création directe de conteneurs). Chaque pod est lié au noeud où il est planifié et y reste jusqu'à la fin (conformément à la politique de redémarrage) ou la suppression. En cas de défaillance d'un noeud, des pods identiques sont planifiés sur d'autres noeuds disponibles dans le cluster. Résumé:

Un pod est un groupe d'un ou plusieurs conteneurs d'applications (tels que Docker) et comprend un stockage partagé (volumes), une adresse IP et des informations sur la façon de les exécuter.

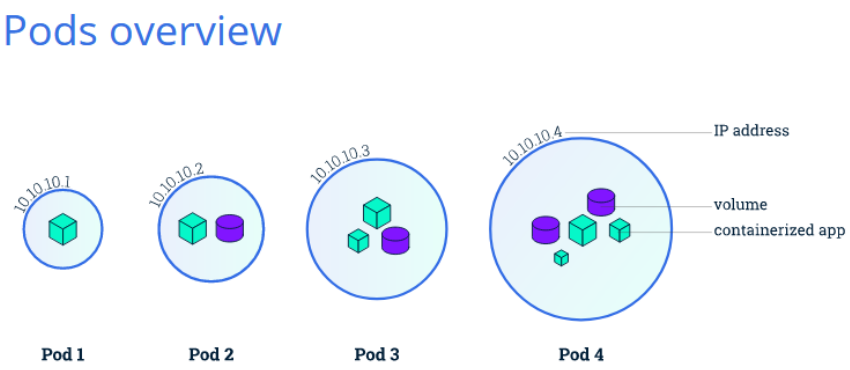


Figure 2: pod

Kubernetes Nodes **********

Un pod fonctionne toujours sur un noeud. Un noeud est une machine de travail dans Kubernetes et peut être une machine virtuelle ou physique, selon le cluster. Chaque noeud est géré par le maître. Un noeud peut avoir plusieurs pods et le maître Kubernetes gère automatiquement la planification des pods sur les noeuds du cluster. La planification automatique du Master prend en compte les ressources disponibles sur chaque Noeud.

Chaque noeud Kubernetes exécute au moins:

Kubelet, un processus responsable de la communication entre le maître Kubernetes et le noeud; il gère les pods et les conteneurs fonctionnant sur une machine. Un environnement d'exécution de conteneur (comme Docker) chargé d'extraire l'image de conteneur d'un registre, de décompresser le conteneur et d'exécuter l'application.

Les conteneurs ne doivent être planifiés ensemble dans un seul pod que s'ils sont étroitement couplés et doivent partager des ressources telles qu'un disque.

3.3 Gestion du cluster Kubernetes

3.3.1 Créer un cluster avec l'environnement minikube

```
$ minikube start
* minikube v1.8.1 on Ubuntu 18.04
* Using the none driver based on user configuration
```

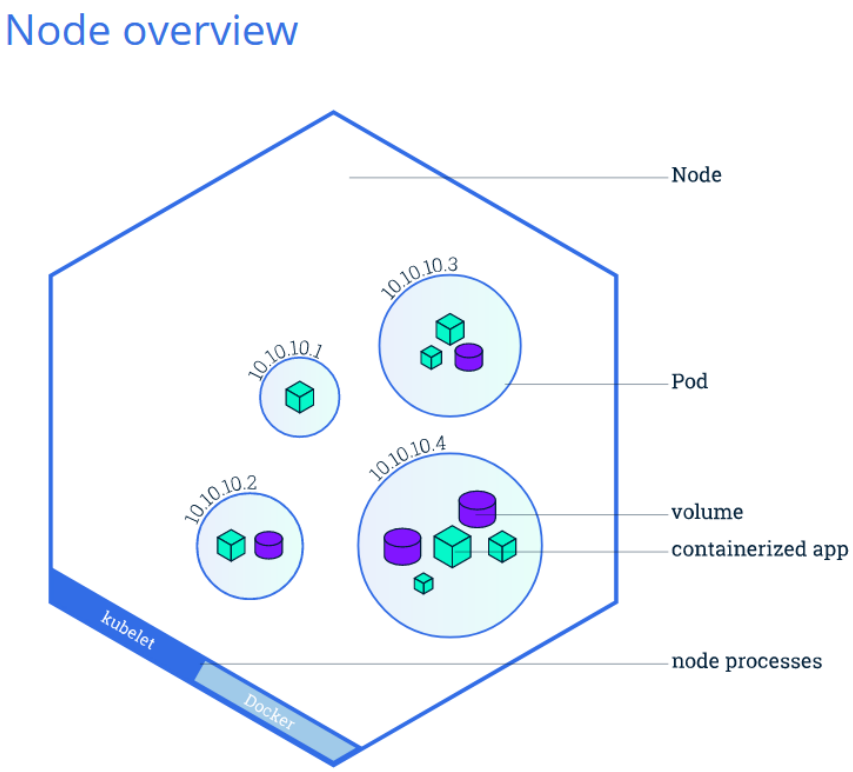


Figure 3: node

```
* Running on localhost (CPUs=2, Memory=2460MB, Disk=145651MB) ...
* OS release is Ubuntu 18.04.4 LTS
* Preparing Kubernetes v1.17.3 on Docker 19.03.6 ...
- kubelet.resolv-conf=/run/systemd/resolve/resolv.conf
* Launching Kubernetes ...
* Enabled addons: default-storageclass, storage-provisioner
* Configuring local host environment ...
* Waiting for cluster to come online ...
* Done! kubectl is now configured to use "minikube"
$
```

```
$ kubectl cluster-info
Kubernetes master is running at https://172.17.0.28:8443
KubeDNS is running at https://172.17.0.28:8443/api/v1/namespaces/kube-system/services/kube-d

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
$ kubectl get nodes
NAME       STATUS   ROLES    AGE    VERSION
minikube   Ready    master   102s   v1.17.3
$
```

3.3.2 deployer une application

---

vérification de la version de kubectl

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.0", GitCommit:"70132b0
Server Version: version.Info{Major:"1", Minor:"17", GitVersion:"v1.17.0", GitCommit:"70132b0
$ kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   31s   v1.17.0
$ kubectl create deployment kubernetes-bootcamp --image=gcr.io/google-samples/kubernetes-boo
deployment.apps/kubernetes-bootcamp created
$ kubectl get deployments.apps
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
kubernetes-bootcamp   1/1     1            1           38s
$
```

Visualiser notre application:

On va exécuter la commande `kubectl proxy` dans un autre terminal et laisser la commande tourner.

```
$ echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please
kubectl proxy
```

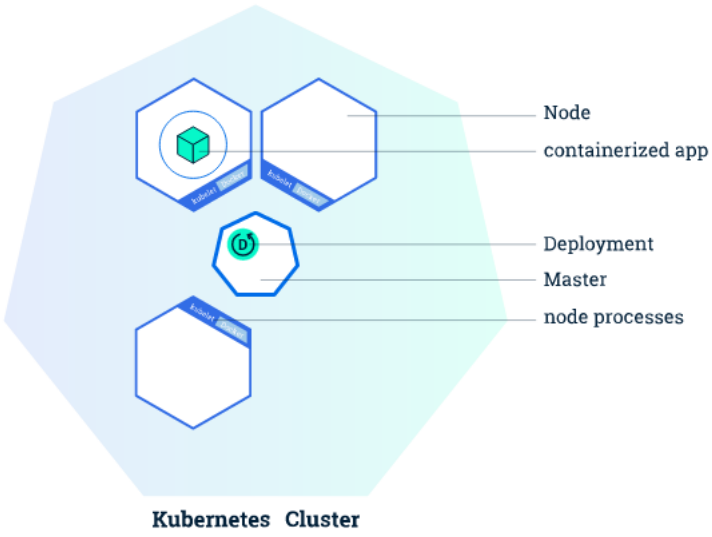


Figure 4: cluster

```
Starting Proxy. After starting it will not output a response. Please click the first Termina
```

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Nous avons maintenant une connexion entre notre hôte (le terminal en ligne) et le cluster Kubernetes. Le proxy permet un accès direct à l'API depuis ces terminaux.

On peut voir toutes ces API hébergées via le point de terminaison du proxy. Par exemple, nous pouvons interroger la version directement via l'API à l'aide de la commande curl (commande à éxecuter sur le premier terminal):

```
$ curl http://localhost:8001/version
{
"major": "1",
"minor": "17",
"gitVersion": "v1.17.0",
"gitCommit": "70132b0f130acc0bed193d9ba59dd186f0e634cf",
"gitTreeState": "clean",
"buildDate": "2019-12-07T21:12:17Z",
"goVersion": "go1.13.4",
"compiler": "gc",
"platform": "linux/amd64"
```

```
}$
```

Les pods qui s'exécutent dans Kubernetes s'exécutent sur un réseau privé et isolé. Par défaut, ils sont visibles depuis d'autres pods et services au sein du même cluster Kubernetes, mais pas en dehors de ce réseau. Lorsque nous utilisons kubectl, nous interagissons via un point de terminaison d'API pour communiquer avec notre application.

```
$ export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.

$ echo Name of the Pod: $POD_NAME
Name of the Pod: kubernetes-bootcamp-69fbc6f4cf-8nmw9
$
```

3.3.3 Explorer une application

Les opérations les plus courantes peuvent être effectuées avec les commandes kubectl suivantes:

kubectl get - liste des ressources kubectl describe - affiche des informations détaillées sur une ressource kubectl logs - imprimer les journaux d'un conteneur dans un pod kubectl exec - exécute une commande sur un conteneur dans un pod

On peut utiliser ces commandes pour voir quand les applications ont été déployées, quel est leur état actuel, où elles s'exécutent et quelles sont leurs configurations.

Maintenant que nous en savons plus sur nos composants de cluster et la ligne de commande, explorons notre application.

Un noeud est une machine de travail dans Kubernetes et peut être une VM ou une machine physique, selon le cluster. Plusieurs pods peuvent s'exécuter sur un seul noeud.

```
$ kubectl describe pods
Name:         kubernetes-bootcamp-765bf4c7b4-qnsrl
Namespace:    default
Priority:     0
Node:         minikube/172.17.0.15
Start Time:   Mon, 15 Feb 2021 14:19:05 +0000
Labels:       pod-template-hash=765bf4c7b4
run=kubernetes-bootcamp
Annotations:  <none>
Status:       Running
IP:           172.18.0.4
IPs:
```

```
IP:           172.18.0.4
Controlled By:  ReplicaSet/kubernetes-bootcamp-765bf4c7b4
Containers:
kubernetes-bootcamp:
Container ID:   docker://c9f8d949af256cf489917a1019a6b551b165a0437cfa3d1949ce32d83579037c
Image:          gcr.io/google-samples/kubernetes-bootcamp:v1
Image ID:       docker-pullable://jocatalin/kubernetes-bootcamp@sha256:0d6b8ee63bb57c5f5b615
Port:           8080/TCP
Host Port:      0/TCP
State:          Running
Started:      Mon, 15 Feb 2021 14:19:08 +0000
Ready:          True
Restart Count:  0
Environment:    <none>
Mounts:
/var/run/secrets/kubernetes.io/serviceaccount from default-token-xm9zk (ro)
Conditions:
Type              Status
Initialized       True
Ready             True
ContainersReady   True
PodScheduled      True
Volumes:
default-token-xm9zk:
Type:        Secret (a volume populated by a Secret)
SecretName:  default-token-xm9zk
Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
node.kubernetes.io/unreachable:NoExecute for 300s
Events:
Type     Reason            Age                From               Message
----     ------            ----               ----               -------
Warning  FailedScheduling  14s (x3 over 22s)  default-scheduler  0/1 nodes are available: 1
Normal   Scheduled         8s                 default-scheduler  Successfully assigned defau
Normal   Pulled            5s                 kubelet, minikube  Container image "gcr.io/goo
Normal   Created           5s                 kubelet, minikube  Created container kubernete
Normal   Started           4s                 kubelet, minikube  Started container kubernete
$
```

Rappelez-vous que les pods fonctionnent dans un réseau privé isolé - nous devons donc leur accéder par proxy afin de pouvoir les déboguer et interagir avec eux. Pour ce faire, nous utiliserons la commande proxy kubectl pour exécuter un proxy dans une deuxième fenêtre de terminal. Cliquez sur la commande ci-dessous pour ouvrir automatiquement un nouveau terminal et exécuter le proxy:

```
$ echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output e first Terminal Ta


Starting Proxy. After starting it will not output a response. Please click the first Termina

Starting to serve on 127.0.0.1:8001
```

Encore une fois, nous allons obtenir le nom du pod et interroger ce pod directement via le proxy. Pour obtenir le nom du pod et le stocker dans la variable d'environnement POD_NAME:

```
$ echo -e "\n\n\n\e[92mStarting Proxy. After starting it will not output a response. Please
$ kubectl proxy
```

Encore une fois, on peut avoir le nom du Pod et faire des requêtes sur ce pod directement à travers le proxy. Pour avoir le nom du Pod et le stocker dans la variable d'environnement POD_NAME, exécuter la commande suivante:

```
$ export POD_NAME=$(kubectl get pods -o go-template --template '{{range .items}}{{.metadata.
$ echo Name of the Pod: $POD_NAME
```

- Container logs

Tout ce que l'application enverrait normalement à STDOUT devient des journaux pour le conteneur dans le pod. Nous pouvons récupérer ces journaux à l'aide de la commande kubectl logs:

```
$ kubectl logs kubernetes-bootcamp-765bf4c7b4-qnsrl
Kubernetes Bootcamp App Started At: 2021-02-15T14:19:08.285Z | Running On:  kubernetes-bootc

Running On: kubernetes-bootcamp-765bf4c7b4-qnsrl | Total Requests: 1 | App Uptime: 1785.875
$
```

- éxécuter une ommande dans un conteneur

Nous pouvons exécuter des commandes directement sur le conteneur une fois que le Pod est opérationnel. Pour cela, nous utilisons la commande exec et utilisons le nom du Pod comme paramètre.

Usage: kubectl exec (POD | TYPE/NAME) [-c CONTAINER] [flags] – COMMAND [args...] [options]

Listons par exemple les variables d'environnement:

```
$ kubectl exec $POD_NAME -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=kubernetes-bootcamp-765bf4c7b4-qnsrl
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
```

```
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
NPM_CONFIG_LOGLEVEL=info
NODE_VERSION=6.3.1
HOME=/root
$
```

Encore une fois, il convient de mentionner que le nom du conteneur lui-même peut être omis car nous n'avons qu'un seul conteneur dans le pod.

Commençons ensuite une session bash dans le conteneur du pod:

```
$ kubectl exec -ti $POD_NAME bash
root@kubernetes-bootcamp-765bf4c7b4-qnsrl:/#
```

Nous avons maintenant une console ouverte sur le conteneur où nous exécutons notre application NodeJS. Le code source de l'application se trouve dans le fichier server.js:

```
root@kubernetes-bootcamp-765bf4c7b4-qnsrl:/# cat server.js
var http = require('http');
var requests=0;
var podname= process.env.HOSTNAME;
var startTime;
var host;
var handleRequest = function(request, response) {
response.setHeader('Content-Type', 'text/plain');
response.writeHead(200);
response.write("Hello Kubernetes bootcamp! | Running on: ");
response.write(host);
response.end(" | v=1\n");
console.log("Running On:" ,host, "| Total Requests:", ++requests,"| App Uptime:", (new Date(
}
var www = http.createServer(handleRequest);
www.listen(8080,function () {
startTime = new Date();;
host = process.env.HOSTNAME;
console.log ("Kubernetes Bootcamp App Started At:",startTime, "| Running On: " ,host, "\n" )
});
$
```

```
root@kubernetes-bootcamp-765bf4c7b4-qnsrl:/# curl localhost:8080
Hello Kubernetes bootcamp! | Running on: kubernetes-bootcamp-765bf4c7b4-qnsrl | v=1
```

3.3.4 Utiliser les services pour exposer son application

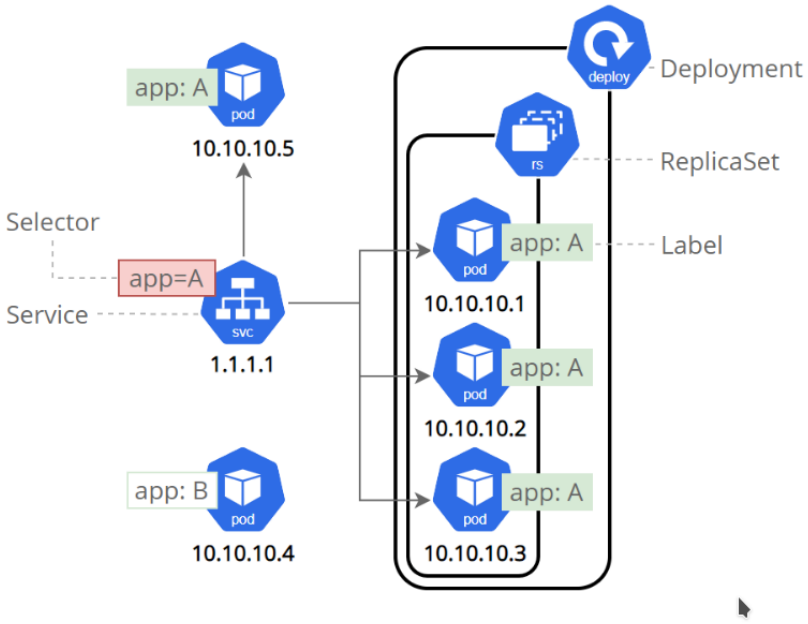
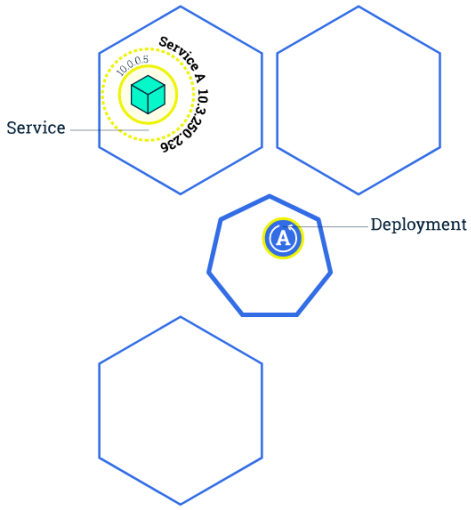


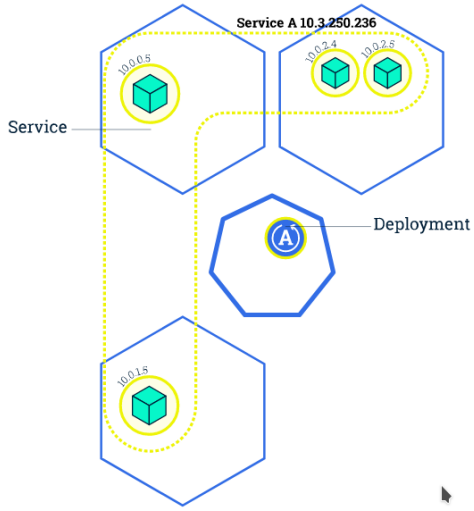
Figure 5: service

3.3.5 Mettre à l'échelle une application

Scaling overview



Scaling overview



****

3.3.6 Mettre à jour une application **********

4. Gestion du réseau dans Kubernetes

On aborde la partie réseau de kubernetes: comment les pods communiquent entre eux et avec l'extérieur. Je pense qu'une manière d'aborder le sujet est de parler de l'architecture réseau des conteneurs via la notion fondamentale de namespace.

4.1 Linux Network Namespace

4.1.1 La notion de namespace réseau

La suite iproute2 est une collection d'utilitaires réseaux et de contrôle de trafic. Ces outils communiquent avec le noyau Linux via l'interface (rt)netlink, fournissant des fonctionnalités inaccessibles par les commandes « ifconfig » et « route » héritées de net-tools.

Les options `addr` `route` ou `link` pour respectivement afficher les informations IP, gérer les routes et les interfaces sont généralement les plus utilisées. Ce qu'on va voir est l'option `netns` qui permet de créer et d'utiliser des *network namespaces*. Quel est le rapport avec les conteneurs et Kubernetes? Regardons ce que dit la page de manuel de ip-netns:

** network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices.

By default a process inherits its network namespace from its parent. Initially all the processes share the same default network namespace from the init process.

By convention a named network namespace is an object at /var/run/netns/NAME that can be opened. The file descriptor resulting from opening /var/run/netns/NAME refers to the specified network namespace. Holding that file descriptor open keeps the network namespace alive. The file descriptor can be used with the setns(2) system call to change the network namespace associated with a task.

For applications that are aware of network namespaces, the convention is to look for global network configuration files first in /etc/netns/NAME/ then in /etc/. For example, if you want a different version of /etc/resolv.conf for a network namespace used to isolate your vpn you would name it /etc/netns/myvpn/resolv.conf.

ip netns exec automates handling of this configuration, file convention for network namespace unaware applications, by creating a mount namespace and bind mounting all of the per network namespace configure files into their traditional location in /etc. **

D'autre part, la page wikipedia de Linux Namespace nous dit:

" Namespaces are a feature of the Linux kernel that partitions kernel resources such that **one set of processes sees one set of resources while another**

set of processes sees a different set of resources. The feature works by having the same namespace for a set of resources and processes, but those namespaces refer to distinct resources. Resources may exist in multiple spaces. Examples of such resources are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication. "

C'est exactement le principe des conteneurs.

" Namespaces are a fundamental aspect of containers on Linux. "

L'idée des Network Namespace est d'isoler de façon virtuelle et au niveau du noyau des interfaces virtuelles ayant leur propre routes, règles de firewall, etc, et de les faire communiquer entre eux. C'est la base du réseau de conteneur, et c'est ce que réalise Kubernetes à grande echelle.

4.2 Exemple le plus simple: faire communiquer deux namespaces via un commutateur virtuel.

La situation est la suivante: on a un linux usuel avec ses interfaces réseau, et on va créer deux namespaces (qu'on appelera "red" et "green"), des interfaces, adresses ip et route pour chacun de ces namespaces et les faire communiquer entre eux. On verra que la commande `ip netns exec green <cmd>`permet d'éxecuter `<cmd>` dans le namespace green, de façon analogue à un docker exec.

tutoriel vidéo disponible: https://www.youtube.com/watch?v=_WgUwUf1d34

4.2.1 état du réseau actuel

---

ip addr, ip link et ip route pour vérifier la configuration des couches 2 et 3:

```
root@debian101:~# ip addr
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
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default
link/ether 00:0c:29:93:ce:33 brd ff:ff:ff:ff:ff:ff
inet 192.168.43.99/24 brd 192.168.43.255 scope global ens33
valid_lft forever preferred_lft forever
inet6 fe80::20c:29ff:fe93:ce33/64 scope link
valid_lft forever preferred_lft forever
3: br-e6e3c03edb14: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN gr
link/ether 02:42:27:f6:30:1d brd ff:ff:ff:ff:ff:ff
```

```
inet 172.18.0.1/16 brd 172.18.255.255 scope global br-e6e3c03edb14
valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
link/ether 02:42:57:99:bf:bd brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
valid_lft forever preferred_lft forever
inet6 fe80::42:57ff:fe99:bfbd/64 scope link
valid_lft forever preferred_lft forever
5: br-164d5b19c7b9: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN gr
link/ether 02:42:78:72:53:18 brd ff:ff:ff:ff:ff:ff
inet 192.168.49.1/24 brd 192.168.49.255 scope global br-164d5b19c7b9
valid_lft forever preferred_lft forever
6: br-6be593cea93c: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN gr
link/ether 02:42:a8:c9:3a:3e brd ff:ff:ff:ff:ff:ff
inet 172.19.0.1/16 brd 172.19.255.255 scope global br-6be593cea93c
valid_lft forever preferred_lft forever
7: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
link/ether 02:42:fd:37:4a:88 brd ff:ff:ff:ff:ff:ff
inet 172.20.0.1/16 brd 172.20.255.255 scope global docker_gwbridge
valid_lft forever preferred_lft forever
inet6 fe80::42:fdff:fe37:4a88/64 scope link
valid_lft forever preferred_lft forever
9: veth201ba9f@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
link/ether 16:77:c7:83:d2:af brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet6 fe80::1477:c7ff:fe83:d2af/64 scope link
valid_lft forever preferred_lft forever
15: veth05815bb@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker
link/ether ca:7d:98:3f:68:f8 brd ff:ff:ff:ff:ff:ff link-netnsid 2
inet6 fe80::c87d:98ff:fe3f:68f8/64 scope link
valid_lft forever preferred_lft forever
root@debian101:~# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group defau
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT
link/ether 00:0c:29:93:ce:33 brd ff:ff:ff:ff:ff:ff
3: br-e6e3c03edb14: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mo
link/ether 02:42:27:f6:30:1d brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT g
link/ether 02:42:57:99:bf:bd brd ff:ff:ff:ff:ff:ff
5: br-164d5b19c7b9: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mo
link/ether 02:42:78:72:53:18 brd ff:ff:ff:ff:ff:ff
6: br-6be593cea93c: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mo
link/ether 02:42:a8:c9:3a:3e brd ff:ff:ff:ff:ff:ff
7: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode D
link/ether 02:42:fd:37:4a:88 brd ff:ff:ff:ff:ff:ff
9: veth201ba9f@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0
```

```
link/ether 16:77:c7:83:d2:af brd ff:ff:ff:ff:ff:ff link-netnsid 0
15: veth05815bb@if14: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker
link/ether ca:7d:98:3f:68:f8 brd ff:ff:ff:ff:ff:ff link-netnsid 2
root@debian101:~# ip route
default via 192.168.43.1 dev ens33 onlink
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
172.18.0.0/16 dev br-e6e3c03edb14 proto kernel scope link src 172.18.0.1 linkdown
172.19.0.0/16 dev br-6be593cea93c proto kernel scope link src 172.19.0.1 linkdown
172.20.0.0/16 dev docker_gwbridge proto kernel scope link src 172.20.0.1
192.168.43.0/24 dev ens33 proto kernel scope link src 192.168.43.99
192.168.49.0/24 dev br-164d5b19c7b9 proto kernel scope link src 192.168.49.1 linkdown
root@debian101:~#
```

4.2.2 Création de deux namespaces

On crée les deux namespace “red” et “green”:

```
root@debian101:~# ip netns
root@debian101:~# ip netns add red
root@debian101:~# ip netns add green
root@debian101:~# ip netns
green
red
```

root@debian101:~# ls /var/run/netns/ green red root@debian101:~#

On peut **éxécuter** la commande ip link **à l’intérieur des namespaces**:

```
root@debian101:~# ip netns exec red ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Pareil pour “green”:

```
root@debian101:~# ip netns exec green ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

4.2.3 Openvswitch

Installation d’openvswitch:

```
root@debian101:~# apt install openvswitch-{common,switch}
```

Création d'un pont qu'on nomme ovs1:

```
root@debian101:~# ovs-vsctl add-br ovs1
root@debian101:~# ovs-vsctl show
139260bd-d442-4341-a2bb-12f85cdfe54e
Bridge "ovs1"
Port "ovs1"
Interface "ovs1"
type: internal
ovs_version: "2.10.7"
```

4.2.4 Plomberie **********

On voit que les deux namespace ont chacun une interface de loopback. Rajoutons leur une interface virtuelle:

```
root@debian101:~# ip link add eth0-r type veth peer name veth-r
root@debian101:~# ip link show eth0-r
17: eth0-r@veth-r: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group d
    link/ether 8a:53:95:cd:3c:91 brd ff:ff:ff:ff:ff:ff
    root@debian101:~# ip link show veth-r
    16: veth-r@eth0-r: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT gro
        link/ether 0a:60:c7:af:4c:ad brd ff:ff:ff:ff:ff:ff
```

Ensuite on attache l'interface eth0-r au namespace red:

```
root@debian101:~# ip link set eth0-r netns red
```

On voit que l'interface eth0-r a disparu du root namespace:

```
root@debian101:~# ip link|grep eth0-r
root@debian101:~#
```

pour arriver comme voulu dans le namespace "red", réseau isolé du root namespace: shell root@debian101:~# ip netns exec red ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
group default qlen 1000 link/loopback 00:00:00:00:00:00 brd
00:00:00:00:00:00 17: eth0-r@if16: <BROADCAST,MULTICAST> mtu
1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
link/ether 8a:53:95:cd:3c:91 brd ff:ff:ff:ff:ff:ff link-netnsid 0
Maintenant connectons l'autre bout du pipe à ovs:

```
root@debian101:~# ovs-vsctl add-port ovs1 veth-r
```

On vérifie que veth-r est bien connectée à ovs1:

```
root@debian101:~# ovs-vsctl show
139260bd-d442-4341-a2bb-12f85cdfe54e
Bridge "ovs1"
```

```
Port veth-r
Interface veth-r
Port "ovs1"
Interface "ovs1"
type: internal
ovs_version: "2.10.7"
```

C'est bon, on a un pipe de communication entre le namespace red et le commutateur virtuel.

On fait de même pour le namespace "green":

```
root@debian101:~# ip link add eth0-g type veth peer name veth-g
```

Vérification:

```
root@debian101:~# ip link|grep ".*-g"
20: veth-g@eth0-g: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT
21: eth0-g@veth-g: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT
```

Comme pour "red", on attache eth0-g au namespace "green":

```
root@debian101:~# ip link set eth0-g netns green
```

On connecte l'autre bout du pipe à ovs1:

```
root@debian101:~# ovs-vsctl add-port ovs1 veth-g
```

Vérification: les deux pipes sont bien connectés:

```
root@debian101:~# ovs-vsctl show
139260bd-d442-4341-a2bb-12f85cdfe54e
Bridge "ovs1"
Port veth-r
Interface veth-r
Port veth-g
Interface veth-g
Port "ovs1"
Interface "ovs1"
type: internal
ovs_version: "2.10.7"
```

Donc nous sommes dans la configuration suivante, il ne manque plus qu'à attribuer des ip, masques de sous-réseau et routes pour avoir un réseau fonctionnel entre les deux namespaces

---

4.2.5 Ajout d'addresse ip et de route pour les interfaces eth0-r et eth0-g

****

Tout d'abord on active l'interface (veth-r pour l'instant, ce sera pareil pour veth-g) sur le switch virtuel:

```
root@debian101:~# ip link set veth-r up
```

Vérification: shell root@debian101:~# ip link show veth-r 16:
veth-r@if17: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc
noqueue master ovs-system state LOWERLAYERDOWN mode DEFAULT
group default qlen 1000 link/ether 0a:60:c7:af:4c:ad brd
ff:ff:ff:ff:ff:ff link-netns red

et maintenant, il nous faut exécuter les commandes **dans** le namespace "red" pour activer l'interface de loopback et eth0-r, puis créer l'IP et la route:

```
root@debian101:~# ip netns exec red ip link set lo up
root@debian101:~# ip netns exec red ip link set eth0-r up
root@debian101:~# ip netns exec red ip addr
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
17: eth0-r@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group def
link/ether 8a:53:95:cd:3c:91 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 10.0.0.1/24 scope global eth0-r
valid_lft forever preferred_lft forever
inet6 fe80::8853:95ff:fecd:3c91/64 scope link
valid_lft forever preferred_lft forever
```

Si on regarde la route dans le namespace red, on se rend compte que le réseau du root namespace n'a aucune conscience du réseau 10.0.0.0: on a bien une **isolation** des namespaces:

```
root@debian101:~# ip netns exec red ip route
10.0.0.0/24 dev eth0-r proto kernel scope link src 10.0.0.1
root@debian101:~# ip route
default via 192.168.43.1 dev ens33 onlink
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
172.18.0.0/16 dev br-e6e3c03edb14 proto kernel scope link src 172.18.0.1 linkdown
172.19.0.0/16 dev br-6be593cea93c proto kernel scope link src 172.19.0.1 linkdown
172.20.0.0/16 dev docker_gwbridge proto kernel scope link src 172.20.0.1
192.168.43.0/24 dev ens33 proto kernel scope link src 192.168.43.99
192.168.49.0/24 dev br-164d5b19c7b9 proto kernel scope link src 192.168.49.1 linkdown
```

Il ne reste plus qu'à effectuer la même chose avec "green":

```
root@debian101:~# ip link set veth-g up
```

sauf que contrairement à "red", on va procéder autrement: on va prendre un raccourci. Au lieu d'écrire ip netns exec green , on va directement *entrer* dans le namespace green en éxécutant ip netns exec green bash, et on pourra faire toutes les commandes précédentes depuis l'intérieur du namespace, pas de l'extérieur

comme on l'a fait pour "red": Tout d'abord, pour bien détailler ce qu'il se passe, on teste

```
root@debian101:~# echo $SHLVL
1
root@debian101:~# ip netns exec green bash
root@debian101:~# echo $SHLVL
2
root@debian101:~# exit
exit
root@debian101:~# echo $SHLVL
1
```

On peut bien entrer dans le namespace (on voit l'analogie avec les conteneurs)

à l'intérieur du namespace "green", on peut utiliser les commandes ip classiques:

```
root@debian101:~# ip link set eth0-g up
root@debian101:~# ip addr add 10.0.0.2/24 dev eth0-g
root@debian101:~# ip addr
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN group default qlen 1000
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
21: eth0-g@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group def
link/ether 52:39:3c:9a:7c:c9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet 10.0.0.2/24 scope global eth0-g
valid_lft forever preferred_lft forever
inet6 fe80::5039:3cff:fe9a:7cc9/64 scope link
valid_lft forever preferred_lft forever
root@debian101:~# exit
exit
```

4.2.6 Test de connectivité

Maintenant que les réseaux namespace red et green sont en place, on peut tester la connectivité entre les deux avec ping:

```
root@debian101:~# ip netns exec red bash
root@debian101:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.553 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.072 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.058 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.074 ms
^C
--- 10.0.0.2 ping statistics ---
```

```
4 packets transmitted, 4 received, 0% packet loss, time 12ms
rtt min/avg/max/mdev = 0.058/0.189/0.553/0.210 ms
root@debian101:~# exit
exit
root@debian101:~# ip netns exec green bash
root@debian101:~# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.423 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.063 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.060 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.088 ms
^C
--- 10.0.0.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 38ms
rtt min/avg/max/mdev = 0.060/0.158/0.423/0.153 ms
```

ça marche!

Deux vidéos pour aller plus loin:

https://www.youtube.com/watch?v=6v_BDHIgOY8 Container Networking From Scratch - Kristen Jacobs, Oracle

https://www.youtube.com/watch?v=Utf-A4rODH8 Building a container from scratch in Go - Liz Rice (Microscaling Systems)