Cours sur Kubernetes

M2 PLS

Julien David

Références

- Cours de DevOpsSec
- Playlist Kubernetes de Xavki
- Les pages wikipédia de toutes les notions abordées dans le cours
- Le site officiel de Kubernetes

Contexte

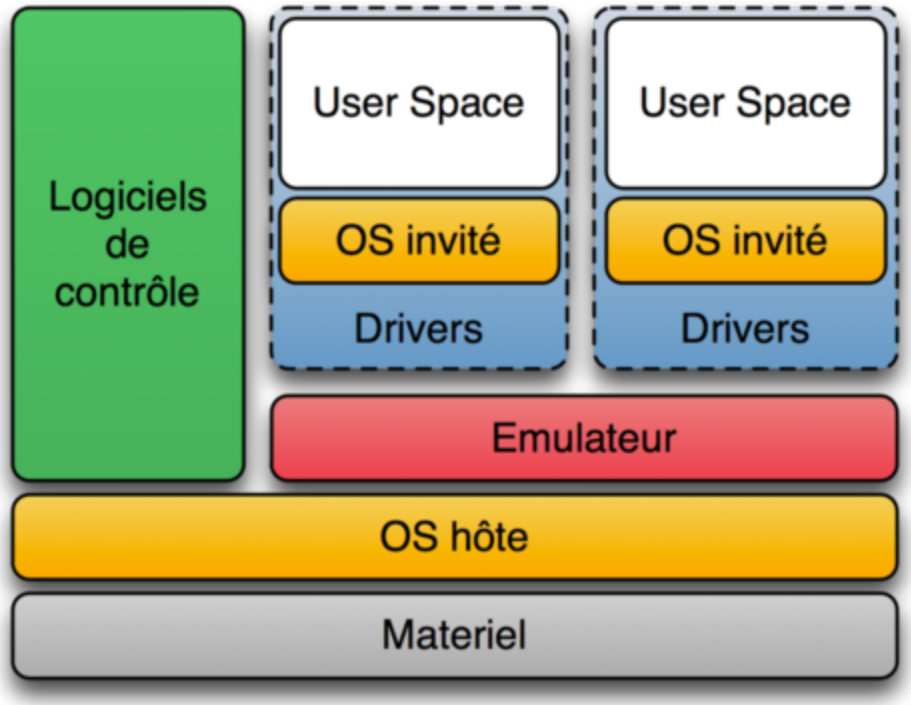
- Virtualisation
- Conteneurs
- Data Center
- Histoire de Kubernetes

Once upon a time...

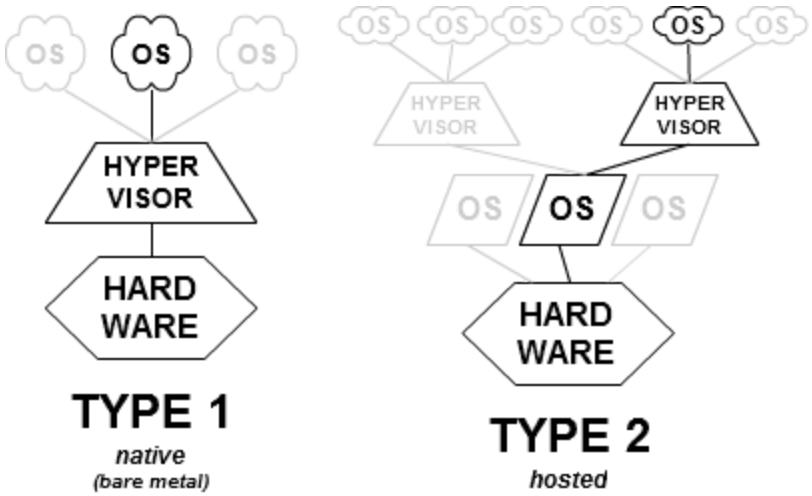
L'époque des serveurs dédiés.

- Un programme = un serveur (ou plus)
- Le système d'exploitation devait être le même sur le serveur de développement et le serveur de production.

Virtualisation



Virtualisation



Virtualisation : avantages

- possibilité d'émuler/limiter les ressources matérielles de chaque machine virtuelle $\Longrightarrow$ adaptation des ressources pour chaque application
- possibilité de reproduire l'environnement de développement en production.
- possibilité de cloner, sauvegarder, déplacer les machines virtuelles,
- permet de réduire les coûts (économiques) en regroupant des applications sur une même machine.

Virtualisation : inconvénients

- La consommation élevée de ressources:
  - l'accès aux resssources passe par une couche d'abstraction matérielle, qui ralentit les performances.
  - chaque machine virtuelle contient un OS qui consomme lui aussi des ressources.

Conteneurs

Objectif: garder les avatanges de la virtualisation en évitant ses inconvénients.

Conteneurs : Isolation par espace de nommage

- Le **namespace PID** : une structure hiérarchique qui fournit l'isolation pour l'allocation des identifiants de processus (PIDs), la liste des processus et de leurs détails.

Créons notre premier conteneur:

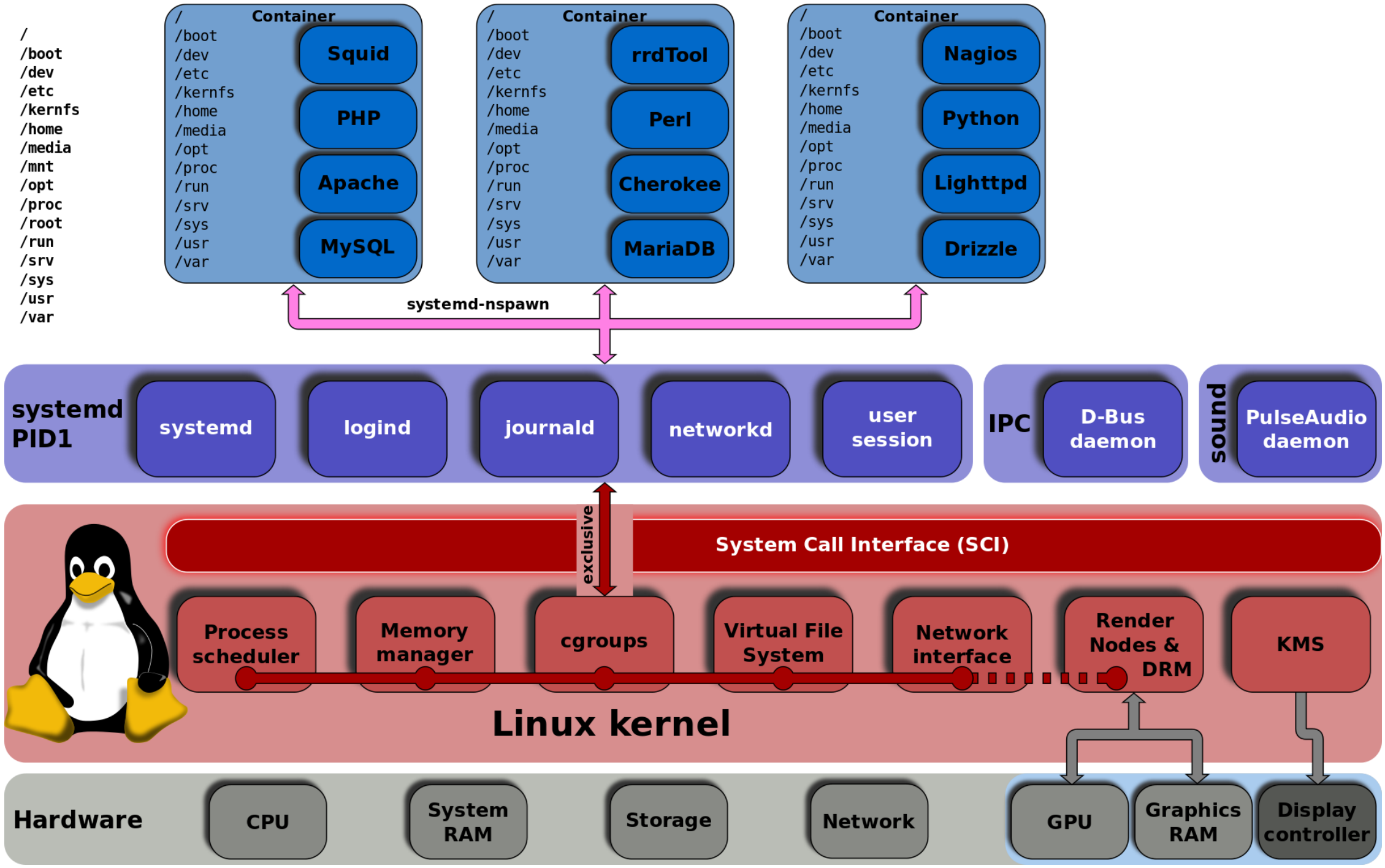
```
sudo unshare --fork --pid --mount-proc bash
```

Conteneurs : Isolation par espace de nommage

- Le **namespace IPC** : empêche la communication avec les autres processus, plus simplement il interdit l'échange d'informations avec les autres processus.
- Le **namespace NET** : crée une pile réseau entièrement nouvelle, y compris : un ensemble privé d'adresses IP, sa propre table de routage, liste de socket, table de suivi des connexions, pare-feu et autres ressources liées au réseau.
- Le **namespace MOUNT** : monte un système de fichier propre différent du système de fichier de la machine hôte.

Conteneurs : Isolation par espace de nommage

- Le **namespace USER** : fournit à la fois l'isolation des privilèges et la séparation des identifications d'utilisateurs entre plusieurs ensembles. Permet par exemple de donner un accès root dans le conteneur sans qu'il soit root sur la machine hôte.
- Le **namespace UTS** : associe un nom d'hôte et de domaine au processus pour avoir son propre hostname.


/
/boot
/dev
/etc
/kernfs
/home
/media
/mnt
/opt
/proc
/root
/run
/srv
/sys
/usr
/var
Container
Squid
PHP
Apache
MySQL
rrdTool
Perl
Cherokee
MariaDB
Nagios
Python
Lighttpd
Drizzle
systemd-nspawn
systemd PID1
systemd
logind
journald
networkd
user session
IPC
D-Bus daemon
sound
PulseAudio daemon
exclusive
System Call Interface (SCI)
Process scheduler
Memory manager
cgroups
Virtual File System
Network interface
Render Nodes & DRM
KMS
Linux kernel
Hardware
CPU
System RAM
Storage
Network
GPU
Graphics RAM
Display controller

Control Groups (CGroups)

Fonctionnalité du noyau Linux permettant de limiter, surveiller et isoler l'utilisation des ressources.

- Développé chez Google en 2006.
- Intégré au noyau Linux fin 2007.

Control Groups (CGroups) : Fonctionnalités

- **Limitation/Priorisation** des resssources: RAM, CPU, bande passante.
- **Mesures** des quantités de ressources consommées.
- **Isolation** par **espace de nommage**, ou *namespace*:
  restriction de la visibilité des processus, des connexions réseaux et des fichiers.
- **Contrôle**: possibilité de mettre des groupes en pause, de sauvegarder, de redémarrer.

Control Groups (CGroups) : exemple

- Créons un `cgroup` dont le but est de limiter la mémoire:

```
sudo cgcreate -a <nom_d_utilisateur> -g memory:tp_k8s
```

- Regardons tous les fichiers crées

```
ls -l /sys/fs/cgroup/memory/tp_k8s
```

```
more /sys/fs/cgroup/memory/tp_k8s/memory.kmem.limit_in_bytes
```

- On limite l'espace mémoire utilisable par ce groupe à 20 Mo

```
sudo echo 20000000 >
```

```
/sys/fs/cgroup/memory/tp_k8s/memory.kmem.limit_in_bytes
```

```
sudo cgexec -g memory:tp_k8s emacs
```

Pourquoi est-ce que je vous parle de ça dans un cours sur Kubernetes

- Les **CGroups** sont à la base du fonctionnement de **Docker** et **Kubernetes**
- On retrouve le concept d'**espace de nommage** dans **Kubernetes**.

Docker

Les fonctionnalités de **Docker** reposent beaucoup sur celles des **CGroups**.

- Un conteneur Docker est une exécution d'une image, isolée à l'aide des de **cgroups** *préconfigurés*

Ce que **Docker** apporte :

- la simplification de l'utilisation de ces conteneurs,
- la possibilité de sauvegarder/charger une configuration dans un **Dockerfile**.

kubernetes

Kubernetes

Crée par Google et devenu un projet open source depuis 2014.

- Cette année là Google déclaraît lancer 2 milliards de conteneurs par semaine.
- le nombre de leur serveurs approchaît alors le million.
- l'entreprise avait donc eu besoin d'automatiser le déploiement des conteneurs sur leurs serveurs.

Surnom **K8S**: K+longeur(ubernete)+S

Ce que Kubernetes apporte (1)

- automatisation des **déploiements** de conteneurs,
- automatisation des **mises à jour**,
- la **scalabilité** (ou mise à l'échelle) automatique.

Ce que Kubernetes apporte (2)

- la possibilité de rendre visible plusieurs conteneurs par une seule et même interface.
- le **load-balancing**: permet de distribuer les calculs sur les différents conteneurs.
- la gestion automatique du stockage.

Ce que Kubernetes apporte (3)

- l'optimisation de la gestion des ressources: on précise pour chaque conteneur la quantité de RAM et de CPU souhaité et Kubernetes optimise la répartition des conteneurs sur les serveurs.

K8S utilise des heuristiques permettant de résoudre le problème du sac à dos (NP-complet)

Ce que Kubernetes apporte (4)

- la réparation automatique: relance les conteneurs qui ont échoués, termine ceux qui ne vérifient pas des conditions définies par l'utilisateur, empêche l'accès des conteneurs aux clients tant que ceux ci ne sont pas prêts.
- gestion des secrets (mots de passe, clés SSH, ...)

Notions élémentaires

- Les noeuds ou *nodes*
- Les *Pods*
- Services

Notions élémentaires

Les noeuds/*nodes*

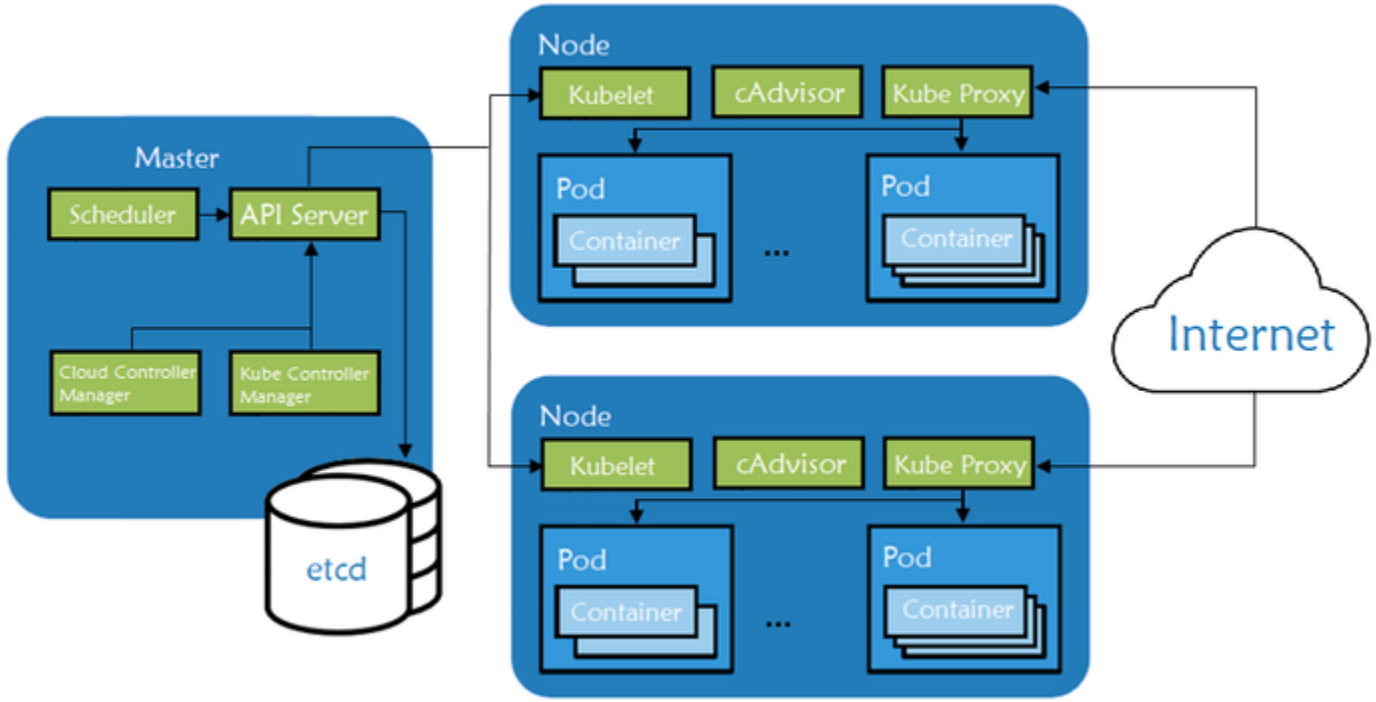
Les noeuds représentent des serveurs qui peuvent être:

- physique ou virtuel,
- maître (`master`) ou noeud d'exécution.

Les noeuds maîtres permettent de:

- gérer l'infrastructure de kubernetes
- déployer les conteneurs sur les noeuds d'exécution.

Architecture



Notions élémentaires

Les *pods*

Un pod est un ensemble de conteneurs qui ont vocation à fonctionner simultanément.

- les conteneurs seront localisés sur la même machine hôte,
- ils en partagent donc les ressources (CPUs, RAM, ...),
- ils partagent également des namespaces.

Pour chaque pod:

- on a une adresse IP unique dans le cluster de Kubernetes,
- on peut définir des volumes partagés,
- kubernetes garantit le maintient de son exécution jusqu'à son expiration ou sa suppression.

Notions élémentaires

Les *pods*, vocabulaire

- Replique, ou *replicas*, est le nombre d'instance d'un pod.
- ReplicaSet permet de maintenir un ensemble de répliques d'un pod dans un état *running*.
- Déploiement, ou *Deployment*, état désiré du pod, gère le ReplicaSet.

Notions

Les services

- Les services permettent de donner un accès aux pods.
  - attribuent une adresse IP unique pour un ensemble de pods.
  - le nom du service devient le nom DNS de cette adresse IP.

Notions

Les services: connection au service

- Lorsque l'on souhaite accéder à un service, on peut se connecter via le nom *DNS*, et non par l'adresse IP.

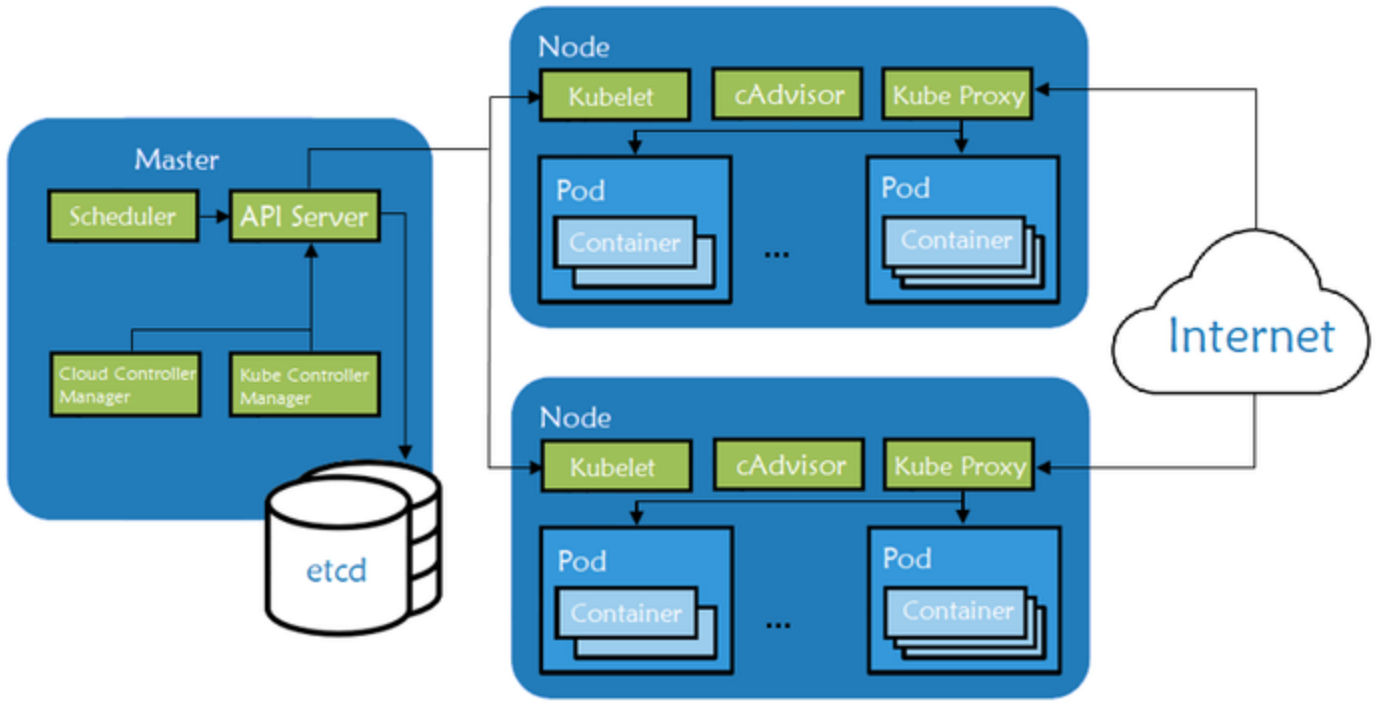
Le service est ainsi toujours joignable, même si les pods sont déplacés sur d'autres noeuds.

Limitations actuelles

Actuellement, pour un fonctionnement optimal, Kubernetes recommmande les limitations
suivantes:

- pas plus de 5000 noeuds,
- pas plus de 150 000 pods,
- pas plus de 300 000 conteneurs,
- pas plus de 100 pods par noeuds.

Architecture



Architecture

```
- Master
    - Le Serveur d'API
    - Etcd
    - L'ordonnanceur
    - Le gestionnaire de contrôleurs
```

Architecture: le serveur d'API

kube-apiserver : point d'entrée exposant l'API HTTP Rest de k8s depuis le maître du cluster Kubernetes.

- Conçu pour faire du scaling horizontal (on déploie de nouvelles instance plutôt que d'en
  chercher des plus puissantes).
- En cas de traffic important, plusieurs instances de kube-apiserver peuvent être lancées.

Architecture: etcd

etcd : base de données clé-valeur.

- stocke toutes les informations de configuration pouvant être utilisées par chacun des nœuds du cluster.
- lorsque l'on modifie l'état du cluster, l'information est enregistrée dans etcd

Architecture: ordonnanceur/scheduler

kube-scheduler : responsable de la répartition et l'utilisation de la charge de travail sur les nœuds du cluster selon les ressources nécessaires et celles disponibles.

- lorsqu'un pod est crée, le scheduler se charge de trouver un noeud adéquat où l'exécuter.
- plusieurs paramètres sont pris en compte:
  - les ressources requises par le(s) pod(s).
  - les contraintes hardware/software
  - les affinités et les anti-affinités (prochain cours)
  - la localisation des données

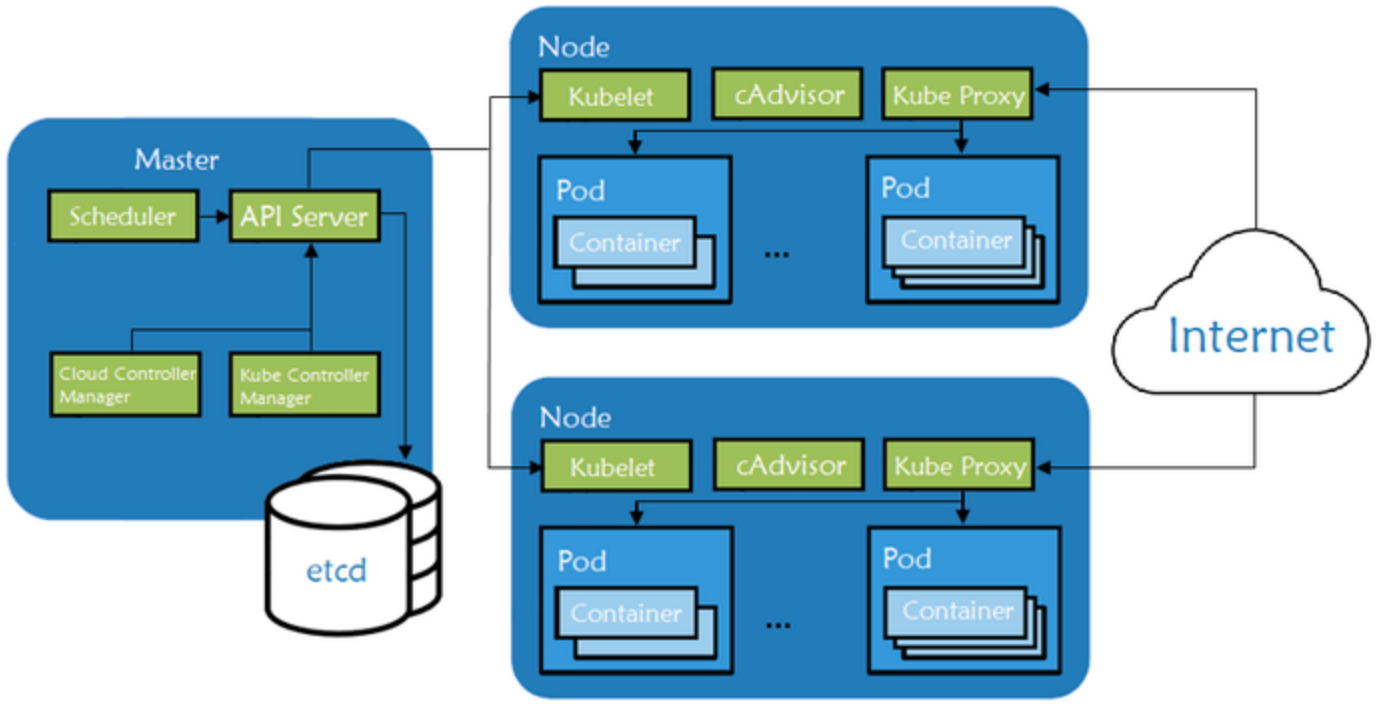
Architecture: le gestionnaire de contrôleurs.

kube-controller-manager: Kubernetes inclut plusieurs contrôleurs, développés indépendamment mais compilés ensemble (un seul processus)

Exemple de contrôleurs:

- Contrôleur de noeuds: surveille les noeuds et réagit quand un noeud dysfonctionne.
- Contrôleur de EndPoints: join les services et les pods.

Architecture



Architecture

- Noeuds
  - kubelet
  - kube-proxy
  - cAdvisor

Architecture: kubelet

Agent présent sur chaque noeud, s'assurant que les containers exécutés dans les pods s'exécutent correctement.

Architecture: kube-proxy

Proxy réseau gérant une partie de la notion de Service de Kubernetes.

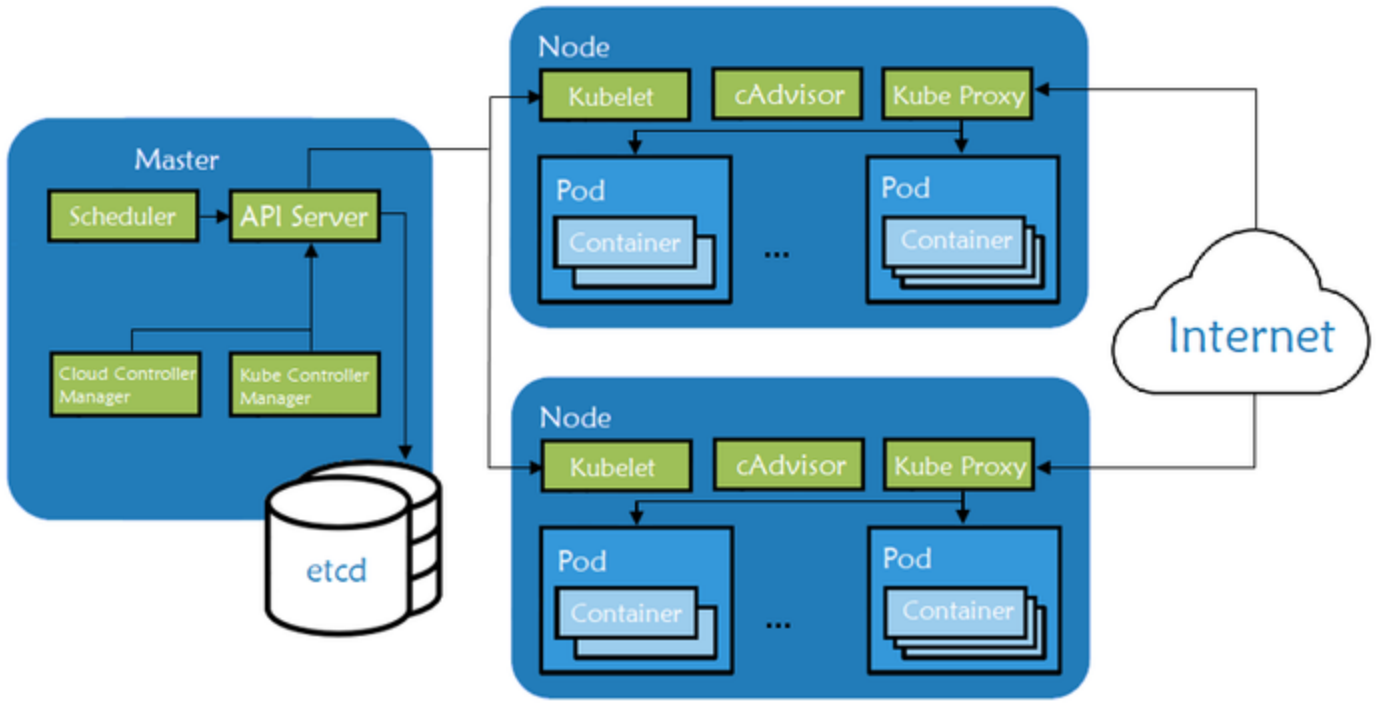
- Maintient les règles réseaux permettant la communication des pods avec les réseaux
  internes et externes du cluster.
- peut utiliser le système de filtrage des paquets du système d'exploitation.

Architecture: CAdvisor

Surveille l'utilisation des ressources faite par les conteneurs (*Monitoring*)

Ce *monitoring* permet de détecter quand un noeud est surchargé et permet à Kubernetes de faire de l'autoscaling.

Architecture



Architecture: le réseau

Le réseau d'un cluster Kubernetes doit résoudre quatre problématiques de communication:

- **entre conteneurs** à l'intérieur d'un même pod (se fait sur localhost)
- **entre Pods** : les Container Network Interface (CNI)
- **des Pods à un service**: géré par les Services
- **entre les services et l'extérieur**: géré par les Services

Architecture: le réseau

L'objectif de Kubernetes est de répartir des applications sur des machines.

- **Comment s'assurer que deux applications de tentent pas d'utiliser le même port**?
- il est trop compliqué de demander aux développeurs de gérer ces aspects.

Architecture: le modèle réseau

Chaque Pod (ou container) reçoit sa propre adresse IP sur le réseau interne du Kubernetes.

- lors de la création d'un pod, on s'assure que les conteneurs n'utilisent pas les même ports.
- pas de conflits entre pods puisque chacun sont sur des IPs différentes.
- les services vont exposer les pods sur des ports qui peuvent être déterminés automatiquement ou manuellement à la création du service.

Cela laisse toutefois de grandes libertés dans l'implantation du réseau interne.
De nombreuses solutions existent.

Architecture: exemple de CNI

Flannel:

- utilise un *overlay network* en IPv4.
- chaque noeud reçoit un intervalle d'adresse à attribuer.
- lorsqu'un pods est lancé sur un noeud, l'interface Docker Bridge attribue une IP à **chaque conteneur**
- les pods sur un même noeud communiquent via l'interface Docker Bridge.
- les pods sur différents noeuds communiquent via une encapsulation UDP, le routage
  étant géré par flannel.

Flannel est recommandé lorsque l'on débute avec Kubernetes.

Architecture: exemple de CNI

Calico:

Connu pour:

- ses performances et sa flexibilité
- la connectivité entre noeuds et pods.
- la sécurité réseau et l'administration

Fonctionnement:

- N'utilise pas un *overlay network*
- Configure un réseau IPv4 et utilise le protocole **BGP** pour acheminer les paquets entre les pods.
- BGP n'utilise pas d'encapsulation, ni de couche réseau supplémentaire.
- possibilité de contrôler le trafic réseau des pods via un système de règles.

Calico est un bon choix quand on désire avoir un meilleur contrôle sur le réseau.

Architecture: exemple de CNI

Weave Web:

- crée un *overlay network* de type **Mesh** entre les noeuds du cluster.
- permet un routage efficace et résiliant.
- installe un composant sur chaque noeud pour le routage. Ces composants échangent entre eux
  pour que chaque noeud connaisse l'état du réseau.
- comme Calico, permet de contrôle le trafic réseau via des règles.
- l'intégralité du trafic réseau est **encrypté**.

Facile à initialiser, donc très pratique pour ce TP!
Défaut: passe très mal à l'échelle.

Architecture: exemple de CNI

Chaque service de Cloud propose son CNI:

- AWS VPC CNI pour les Amazon Web Service
- Azure CNI pour le cloud de Microsoft
- GKE pour le cloud de Google.

Il en existe bien d'autres.
Le choix du CNI peut être déterminant et il peut-être coûteux d'en changer.

Marre de la théorie !

On s'ennuie ? Passons à la pratique!

Retour à la pratique : déployer une application

Prenons l'exemple d'un serveur nginx.

En ligne de commande (sur kmaster):

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

Puis observons ce qui a été crée par Kubernetes:

- Un pod `kubectl get pod`
- Un ReplicatSet `kubectl get replicatset`
- Un déploiement `kubectl get deployment`

Pour tout afficher d'un coup.

`kubectl get all`

Retour à la pratique : déployer une application

Que ce passe t-il sur le noeud?

Observons maintenant ce qui s'est passé sur le noeud *knode1*

```
sudo docker ps
```

Retour à la pratique : déployer une application

Que ce passe t-il sur le noeud?

Observons maintenant ce qui s'est passé sur le noeud *knode1*

```
sudo docker ps
```

Deux conteneurs ont été lancés:

- le serveur nginx,
- un conteneur portant presque le même nom, lancé avec le commande `pause`.

Retour à la théorie

Nous l'avons vu au premier cours, les conteneurs sont crées à l'aide des **namespaces**.

Afin de pouvoir partager les namespaces entre les conteneurs d'un même pod :

- Kubernetes commence à créer un conteneur ne faisant rien (d'où le `pause`), dans lequel des namespaces sont crées.
- Kubernetes lance ensuite chaque conteneur en précisant que les namespaces à utiliser
  sont ceux du "conteneur pause".

Pourquoi Faire? permet par exemple de placer les conteneurs sur une même interface réseau,
leur permettant de communiquer via `localhost`

(pour aller plus loin https://www.ianlewis.org/en/almighty-pause-container)

Fichiers de configuration YAML

On retrouve les champs suivants:

- apiVersion : version de l'API Kubernetes utilisée pour créer l'objet
- kind : genre de l'objet qu'on souhaite créer
  - Deployment
  - Service
  - ...
- metadata : permet d'identifier l'objet de façon unique
- spec : état désiré pour l'objet.

Retour à la pratique : déploiement via un fichier YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monnginx
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
```

Retour à la pratique: scaling de pods.

Une fois le déploiement effectué, il est possible de changer le nombre de répliques.

```
kubectl scale --replicas=2 deployment.apps/monnginx
```

Retour à la pratique : scaling via un fichier YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monnginx
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
```

Retour à la pratique: les services

Afin de rendre nginx accessible depuis l'extérieur,
on va exposer les ports 80 des différents pods à l'aide d'un service

```
kubectl create service nodeport monnginx --tcp=8080:80
```

Observons le service crée

```
kubectl get service
```

Les différents types de services

- nodeport : rendre public les pods d'un déploiment via un port (30000-32767)
- Clusterip : minimale car exposition interne
- externalname : accès à une ressource via une url
- loadBalancerIP : pour exposer via controler ingress ou dans le cloud (hors cours)

Retour à la pratique : services via un fichier YAML

```
apiVersion: apps/v1
kind: Service
metadata:
  name: monnginx
spec:
  type : Nodeport
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      # Optionnel
      nodePort : 31871
```

Les volumes

- emptyDir : pas de persistence mais partage entre pods/conteneurs
- hostPath : répertoire partagé avec le host qui héberge le pod (attention déplacement)
- persistent volume claim : volume persistant partageable entre pods.
- externe : par exemple NFS mais de nombreux autres (volumes claim, glusterfs, vsphere...)

Volume : emptyDir

- permet de partager un volume éphémère entre conteneurs d'un même pod.

Volume : emptyDir

```
apiVersion: v1
kind: Pod
metadata:
 name: podemptydir
spec:
 containers:
  - name: monnginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: monvolume
  - name: monalpine
    image: alpine
    command: ['sh', '-c', 'echo "TP de K8S" >/commejeveux/index.html' ]
    volumeMounts:
    - mountPath: /commejeveux/
      name: monvolume
 volumes:
   - name: monvolume
     emptyDir: {}
```

Volume : emptyDir

L'exemple précédent fonctionne mais n'est pas *propre*:

- nous avons lancé un conteneur qui se ferme une fois sa tâche exécutée, alors que l'autre tourne en continu.
- Kubernetes considère qu'il y a eu une erreur et écrit le statut *CrashLoopBackOff*
- De plus, nous avons lancé un pod, sans le lancer dans un déploiement, ce qui n'est pas recommandé.

Volume : emptyDir

```
apiVersion: v1
kind: Pod
metadata:
 name: podemptydir
spec:
 containers:
  - name: monnginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: monvolume
  - name: monalpine
    image: alpine
    command: ['sh', '-c', 'while true; do date >/commejeveux/index.html; sleep 1; done' ]
    volumeMounts:
    - mountPath: /commejeveux/
      name: monvolume
 volumes:
   - name: monvolume
     emptyDir: {}
```

Volume : hostPath

- volume du host monté dans le pod (et dans le conteneur)

Attention : uniquement sur le noeud

- en cas de déplacement du pod sur un autre noeud, perte des données.

Volume : hostPath

```
apiVersion: v1
kind: Pod
metadata:
  name: monpod
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: monvolume
  volumes:
  - name: monvolume
    hostPath:
      path: /srv/data
      type: Directory
```

Attention le répertoire indiqué dans *path* doit déjà exister sur le noeud!

Volume : volume persistant

Un volume persistant, ou *PersistentVolume (PV)*, est une zone de stockage sur le cluster.

- il s'agit d'une ressource,
- dont le cycle de vie est indépendant des pods qui l'utilisent.
- l'api qui gère cette ressource s'occupe des détail de son implantation.

On associe les *persistant volume* avec des supports de stockage existant afin de les rendre disponible sur le cluster.

En pratique, il est plus conseillé d'utiliser du stockage distribué (NFS, iSCI, API fournie par le fournisseur d'accès au cloud, ...)

Volume : volume persistant

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: monpv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/pvdata"
```

Volume : Persistant Volume Claim

Un *Persistant Volume Claim* (PVC), est une demande d'accès à une zone de stockage par un utilisateur.

- cette demande peut spécifier une taille spécifique et des droits d'accès.
- Kubernetes se charge d'associer cette demande à un volume disponible.
- si aucun volume persistant ne possède les ressources nécessaire, la demande reste
  dans le statut *pending*

Volume : Persistant Volume Claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: monpvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 15Mi
```

Les secrets

Les secrets (notion que vous devez déjà connaître dans Docker), permettent de stocker des informations sensibles (mots de passe, clés ssh, ...).

S'il est tout à fait possible de gérer ces secrets à l'aide de Docker, Kubernetes le permet également.

Les Jobs

Nous l'avons vu dans un des exemples précédents, par défaut les pods et les conteneurs
sont supposés s'exécuter jusqu'à ce que leur arrêt soit explicitement de demandé via `kubectl`.

Comment faire lorsque nous souhaitons simplement exécuter des tâches non-récurrente?

Les Jobs

Un **Job** permet de créer un ou plusieurs **Pods*, dont on va spécifier un nombre d'exécutions qui se terminent avec succès.

C'est donc l'équivalent d'un déploiement pour des tâches non-récurrentes.

Lorsque le nombre d'exécutions réussies est atteint, le **Job** s'arrête de lui-même.

Les Jobs

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  completions: 2
  parallelism: 1
  backoffLimit: 4
```

Dans cet exemple, on calcule les 2000 premières décimales de π.

On effectue ce calcul 2 fois, l'un après l'autre.

On autorise au plus 4 échecs pour l'exécutions de la tâche. Après cela on abandonne.