Cluster Kubernetes sur Raspberry Pi

Table of Contents

- Cluster Kubernetes sur Raspberry Pi
  - Préparation des systèmes
    - Raspbian
    - ArchLinux
    - Configurer le système
  - Installation de l'écosystème des conteneurs
    - Containerd
    - LXE
    - rktlet
  - Configuration du LoadBalancer
  - Déploiement Kubernetes
    - Déploiement Manuel (ArchLinux)
    - Déploiement avec automatisé (Raspbian)

La conteneurisation démontre son efficacité dans le déploiement d'applications en Cloud Computing. Les conteneurs peuvent encapsuler des programmes complexes avec leurs dépendances dans des environnements isolés, ce qui rend les applications plus portables. Ils sont donc adoptés dans les clusters de **c**alcul **h**aute **p**erformance (**HPC**).

Cet article introduit l'orchestrateur de conteneurs (Kubernetes) afin de produire une architecture hybride qui intègre les clusters HPC .

Kubernetes est une solution de conteneurisation open-source, simplifiant le déploiement, la mise à l’échelle, et la gestion en général, d’applications conteneurisées.

Préparation des systèmes

Raspbian

Pour le noeud master, il est impératif d’utiliser une image 64b, sans quoi **Kubespray** ne sait pas installer **etcd**. Le problème étant que les mainteneurs du projet etcd ne publient pas de version 32b, il reste possible d’installer une version 32b d’etcd, fournie par Debian, quoi que **Kubespray** ne supporte pas cette combinaison

Pour les autres noeuds, on peut utiliser la dernière image 32b.

Une fois ces archives téléchargées et extraites, on peut flasher les cartes micro-sd de nos Raspberry Pi :

```
dd if=./2020-08-20-raspios-buster-arm64-lite.img | pv | dd of=/dev/mmcblk0
```

```
apt-get update --allow-releaseinfo-change
apt-get install python-apt cgroupfs-mount ceph-common rbd-nbd
apt-get upgrade
apt-get dist-upgrade
apt-get install haproxy hatop
sudo apt-get install python-pip git ca-certificates
```

ArchLinux

Installer les packages **python-apt**, **ceph**, **haproxy**, **hatop**, **python-pip**, **git**, **ca-certificates**

Pour installer les packages des dépôts officiels, exécuter :

```
pacman -S foobar
```

Pour installer les packages de l'Arch User Repository (AUR), télécharger l'archive PKGBUILD, l'extraire, vérifier le contenu et enfin exécuter, dans le même dossier :

```
makepkg -sri
```

Configurer le système

Démarrer le Raspberry, avec écran et clavier, afin d'y configurer un mot de passe root :

```
sudo -i
passwd
```

Idéalement, on ne devrait pas être root lors de l'installation ou de l'exécution du cluster comme ce serait le cas dans les environnements de production. On devrait donc créer un utilisateur avec le privilège sudo. Sur Arch linux en tant que root:
- installer sudo : `pacman -S sudo`
- éditer le fichier des sudoers: `visudo` - décommenter pour permettre aux membres du groupe wheel d'exécuter n'importe quelle commande : `%wheel ALL=(ALL) ALL`
- créer un utilisateur et l'ajouter au groupe wheel

Une addresse IP statique :

```
cat <<EOF >/etc/network/interfaces
```

```
auto eth0
iface eth0 inet static
    address x.y.z.a
    netmask 255.255.255.0
    gateway x.y.z.b
EOF
```

Serveur DNS :

```
cat <<EOF >/etc/resolv.conf
nameserver 10.255.255.255
domain friends.intra.example.com
search friends.intra.example.com
EOF
```

Désactiver le fichier d'échange swap[1)]:

```
sed -i 's|CONF_SWAPSIZE=.*|CONF_SWAPSIZE=0|' /etc/dphys-swapfile
systemctl disable dphys-swapfile
```

Configurer le serveur SSH :

```
sed -i -e 's|#PermitRootLog.*|PermitRootLogin yes|' -e
's|#PasswordAuth|PasswordAuthentication yes|' /etc/ssh/sshd_config
systemctl enable ssh
```

Désactiver les services réseaux qu'on n'utilisera pas :

```
systemctl disable dhcpcd
systemctl disable wpa_supplicant
systemctl disable bluetooth
```

Configurer le nom de la machine :

```
cat <<EOF >/etc/hostname
EOF
```

```
cat <<EOF >/etc/hosts
x.y.z.a <fqdn> <hostname>
127.0.0.1 <fqdn> <hostname>
127.0.0.1 localhost
::1 localhost6 localhost6.localdomain
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
EOF
```

Activer les cgroups[2] mémoire :

```
sed -i 's|rootwait$|rootwait cgroup_enable=cpuset cgroup_enable=memory|'
/boot/cmdline.txt
```

```
cat /boot/cmdline.txt
```

```
console=serial0,115200 console=tty1 root=PARTUUID=dcca89ee-02
rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait
cgroup_enable=cpuset cgroup_enable=memory
```

Redémarrer, et vérifier les mots de passe root, IP, gateway, groups memory:

```
cat /proc/cgroups
```

```
hostname -s
hostname -f
ip a
ip r
```

La commande `cat /proc/cgroups` doit retourner quelque chose comme cela:

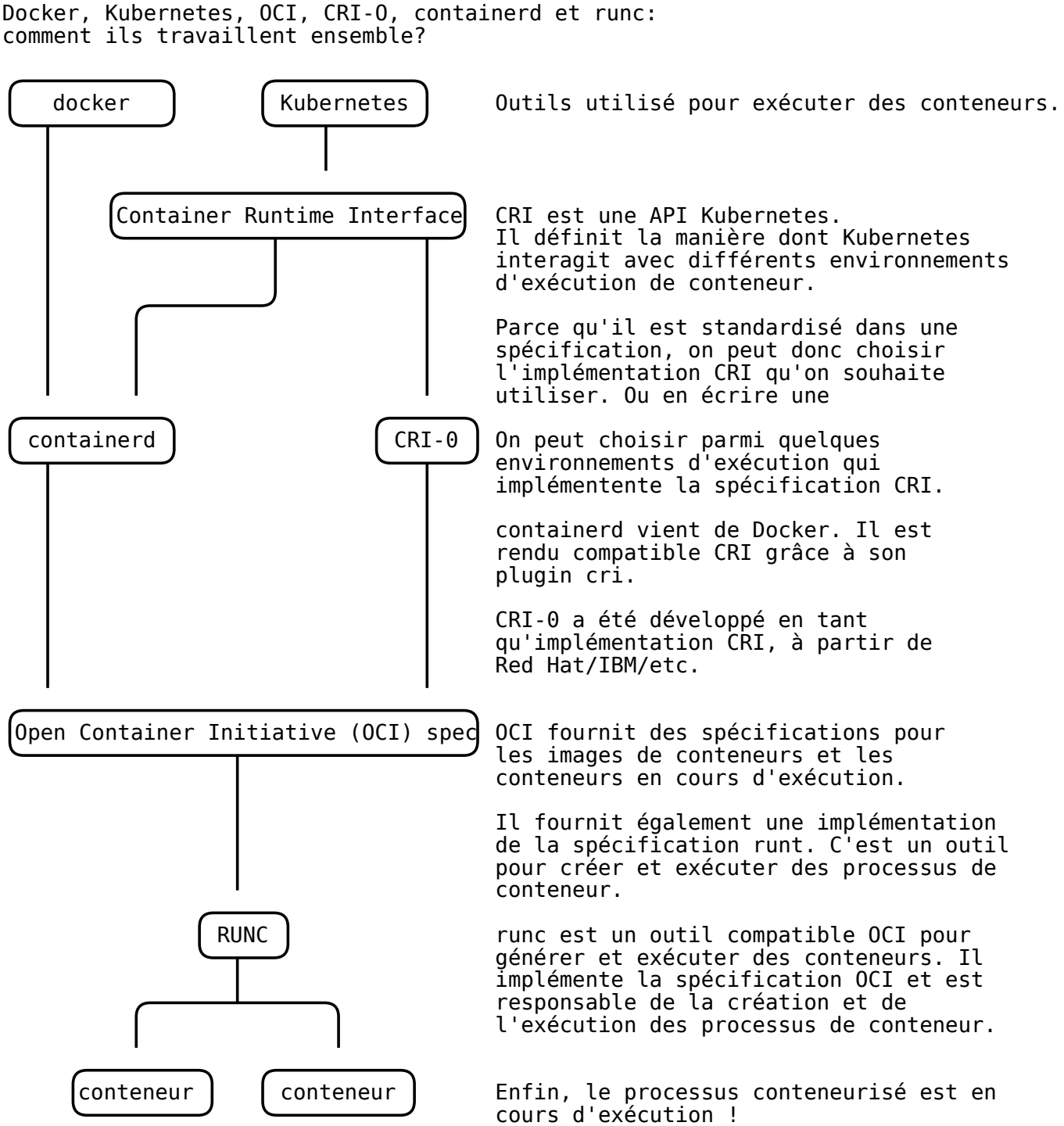
```
subsys_name hierarchy    num_cgroups    enabled
cpuset  9    3    1
cpu 3    60    1
cpuacct 3    60    1
blkio   2    60    1
memory  4    88    1
devices 8    96    1
freezer 6    3    1
net_cls 7    3    1
perf_event  5    3    1
net_prio    7    3    1
pids    10    110    1
```

Mettre le système à jour.

Installation de l'écosystème des conteneurs

Il faut installer un environnement d'exécution de conteneur dans chaque nœud du cluster afin que les pods puissent s'y exécuter. Cette section décrit ce qui est impliqué et les tâches associées pour la configuration des nœuds.

Cette illustration montre comment Docker, Kubernetes, CRI, OCI, containerd et runc s'intègrent dans cet écosystème :



Containerd

Installer le package containerd.io à partir des référentiels Docker officiels.

Configurer containerd :

```
sudo mkdir -p /etc/containerd

containerd config default | sudo tee /etc/containerd/config.toml
```

Redémarrer containerd :

```
sudo systemctl restart containerd
```

Activer les dépendances d'exécution requises:

```
cat <<EOF | sudo tee /etc/modules-load.d/containerd.conf
overlay
br_netfilter
EOF
```

```
sudo modprobe overlay
sudo modprobe br_netfilter
```

Configurer les paramètres sysctl requis, ceux-ci persistent lors des redémarrages.

```
cat <<EOF | sudo tee /etc/sysctl.d/99-kubernetes-cri.conf
net.bridge.bridge-nf-call-iptables  = 1
net.ipv4.ip_forward                 = 1
net.bridge.bridge-nf-call-ip6tables = 1
EOF
```

Appliquer les paramètres sans reboot

```
sudo sysctl --system
```

LXE

LXE fournit l'interface d'exécution de conteneur Kubernetes pour LXD. Il faut avoir installé LXD >= 3.3, un LXD construit par source est également pris en charge.

Actuellement, LXE nécessite golang 1.13 ou plus récent pour être compilé et utilise des modules Go.

Pour Archlinux: activer le dépôt de la communauté dans /etc/pacman.conf:

```
[community]
Include = /etc/pacman.d/mirrorlist
```

Installer le paquet lxd et go:

```
pacman -Syu go
pacman -Syu lxd
```

Cloner le référentiel https://github.com/automaticserver/lxe à l'emplacement souhaité.

Construire le projet en utilisant la commande suivante, qui donnera le binaire dans ./bin/

```
make build
```

On peut également exécuter le programme directement en utilisant :

```
make run lxe -- --log-level info --network-plugin cni
```

Mne fois que LXE est exécuté sur lee système, on peut définir le socket LXE comme point de terminaison CRI dans **kubelet**. Il faut définir les options suivantes pour que **kubelet** puisse se connecter au socket LXE:

```
--container-runtime=remote\
--container-runtime-endpoint=unix:///run/lxe.sock
```

rktlet

rktlet est une implémentation de l'interface d'exécution de conteneur Kubernetes utilisant **rkt** comme environnement d'exécution de conteneur principal.

rkt est un outil CLI écrit en go pour exécuter un conteneur sous Linux. Il utilise une conception multicouche permettant de modifier le ou les temps d'exécution en fonction des souhaits de l'implémenteur. Par défaut, **rkt** utilise **systemd** et **systemd-nspawn** en combinaison pour créer des conteneurs. systemd-nspawn est utilisé pour gérer l'espace de noms dans lequel **systemd** est exécuté pour gérer les groupes de contrôle. Les applications de conteneur sont exécutées en tant qu'unités **systemd**. En utilisant différentes images “étape 1”, les outils utilisés pour exécuter l'application peuvent être modifiés des outils **systemd** à presque tout le reste.

On peut pouvoir construire rkt sur n'importe quel système Linux moderne avec Go (1.5+) installé. Pour la plupart, la base de code est autonome, mais l'assemblage du stage1 nécessite néessite des dépendances suivantes glibc, libdl, libacl, C compiler. Une fois les dépendances satisfaites, on peut créer rkt.

Par défaut, rkt obtient systemd à partir d'une image Linux CoreOS Container pour générer stage1. Il est également possible de construire systemd à partir des sources. Pour ce faire, utilisez les paramètres de configuration suivants après avoir exécuté ./autogen.sh :

```
git clone https://github.com/rkt/rkt.git
cd rkt
./autogen.sh && ./configure --with-stage1-flavors=src --with-stage1-systemd-
version=v231 --with-stage1-systemd-revision=master --with-stage1-systemd-
src= $HOME/src/systemd && make
```

Pour pouvoir exécuter rkt, activer les dépendances d'exécution requises:

```
sudo modprobe overlay
```

Cloner le référentiel https://github.com/kubernetes-retired/rktlet à l'emplacement souhaité.

Construire le projet en utilisant la commande suivante:

```
make go build -o bin/rktlet ./cmd/server/main.go
```

Configurer Kubernetes pour utiliser **rktlet**, en supposant que le processus **rktlet** s'exécute avec la configuration par défaut, définir les options suivantes suivantes:

```
--cgroup-driver=systemd \
--container-runtime=remote \
--container-runtime-endpoint=/var/run/rktlet.sock \
--image-service-endpoint=/var/run/rktlet.sock
```

Configuration du LoadBalancer

Raspberry 3 et 4 conviendront parfaitement au déploiement de **Kubernetes**. En revanche, les modèles plus anciens seront incapables de lancer certaines composantes **Kubernetes**, faute d'images adaptées à leur ARM v6.

Déployer le LoadBalancer, qui se trouvera devant le service API des masters **Kubernetes**:

```
cat <<EOF >/etc/haproxy/haproxy.cfg
```

```
global
    log /dev/log local0
    log /dev/log local1 notice
    chroot /var/lib/haproxy
    stats socket /run/haproxy/admin.sock mode 660 level admin expose-fd
listeners
    stats timeout 30s
    user haproxy
    group haproxy
    daemon
    ca-base /etc/ssl/certs
    crt-base /etc/ssl/private
    ssl-default-bind-ciphers
ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:ECDH+AES128:DH+AES:RSA+AESGCM:RS
A+AES:!aNULL:!MD5:!DSS
    ssl-default-bind-options no-sslv3
```

```
defaults
    log global
    option dontlognull
    timeout connect 5000
    timeout client 50000
    timeout server 50000
    errorfile 400 /etc/haproxy/errors/400.http
    errorfile 403 /etc/haproxy/errors/403.http
    errorfile 408 /etc/haproxy/errors/408.http
    errorfile 500 /etc/haproxy/errors/500.http
    errorfile 502 /etc/haproxy/errors/502.http
    errorfile 503 /etc/haproxy/errors/503.http
    errorfile 504 /etc/haproxy/errors/504.http

listen kubernetes-apiserver-https
    bind 0.0.0.0:6443
    mode tcp
    option log-health-checks
    timeout client 3h
    timeout server 3h
    server master1 10.42.253.40:6443 check check-ssl verify none inter 10s
    server master2 10.42.253.41:6443 check check-ssl verify none inter 10s
    server master3 10.42.253.42:6443 check check-ssl verify none inter 10s
    balance roundrobin
EOF
```

```
cat <<EOF >/etc/profile.d/haproxy.sh
alias hatop='hatop -s /run/haproxy/admin.sock'
EOF
```

```
systemctl start haproxy
systemctl enable haproxy
. /etc/profile.d/haproxy.sh
hatop
```

Déploiement Kubernetes

Déploiement Manuel (ArchLinux)

Cette section présente l'installation rapide sur Arm64 & Arch de kubeadm pour déployer un cluster **Kubernetes** bare metal non HA un maître deux travailleurs.

```
sudo pacman -S ethtool ebtables socat cni-plugins conntrack-tools iproute2
iptables util-linux
```

Installer aur/kubelet-bin et

https://aur.archlinux.org/cgit/aur.git/tree/PKGBUILD?h=kubeadm-binaur/kubeadm-bin

Sur tous les hôtes :

```
cat > /etc/sysctl.d/bridge.conf <<EOF
net.bridge.bridge-nf-call-iptables=1
EOF
```

```
cat > /etc/docker/daemon.json <<EOF
{
  "insecure-registries" : ["mymasternode:5000"],
  "exec-opts": ["native.cgroupdriver=systemd"]
}
EOF
```

Démarrer kubeadm sur le maître

```
sudo kubeadm init --pod-network-cidr 10.244.0.0/16 --apiserver-advertise-
address 192.168.40.10 --apiserver-cert-extra-sans extrahostname.node  --
node-name mymasternode
```

Pour que **kubectl** communique avec le nouveau cluster, il faut copier le fichier `admin.conf` dans le `$HOME/.kube` de chacun des nœuds du cluster:

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

Modifier la configuration de **kubeadm** pour la faire pointer vers `/usr/lib/cni` qui est le chemin utilisé par le package Arch éditer `/var/lib/kubelet/kubeadm-flags.env`:

```
KUBELET_KUBEADM_ARGS="--cgroup-driver=systemd --hostname-
override=mymasternode --network-plugin=cni --pod-infra-container-
image=k8s.gcr.io/pause:3.1 --resolv-conf=/run/systemd/resolve/resolv.conf --
cni-bin-dir=/usr/lib/cni"
sudo systemctl restart kubelet
```

Sur les nœuds de calcul :

```
kubeadm join  192.168.40.10:6443 --token q3l12s.r811b5pbibi9mjy \
    --discovery-token-ca-cert-hash sha256:b67aaaaaaaaaaaaaaabbbbbbbccccccc
--node-name myworker1
```

Modifier `/var/lib/kubelet/kubeadm-flags.env` pour ajouter `--cni-bin-dir=/usr/lib/cni` sur les workers puis redémarrer **kubelet**.

Initialiser le plan de contrôle

```
sudo kubeadm init --apiserver-advertise-address=192.168.0.26 --pod-network-cidr=192.168.0.0/16 --upload-certs --ignore-preflight-errors=NumCPU,Mem
```

192.168.0.26 est l'adresse IP du nœud maître qui est transmise au plan de contrôle. Ce sera le serveur sur lequel le service APi sera disponible et le cidr utilisé par le cluster s'étendra sur l'ensemble du réseau domestique:

1. Le cidr du réseau domestique est 192.168.0.0/16.
2. Le nœud maître est sur IP 192.168.0.26

Les cidrs ne doivent pas se chevaucher car le maître, les noeuds de calcul et chaque élément présent sur le cluster auront le même réseau virtuel superposé aux sous-réseaux physiques réels. Cela impose en quelque sorte une restriction.

En cas de succès, on doit voir le message «Your Kubernetes control-plane has initialized successfully! » avec le jeton et la commande pour rejoindre le travailleur. Noter le jeton retourné par kubeadm join.

Une fois le cluster initialisé, pour pouvoir joindre les nœuds de travail en exécutant ce qui suit en tant que root :

```
kubeadm join <your master node ip>:6443 --token <your-token \
--discovery-token-ca-cert-hash <your-hahs>
```

Actuellement `kubectl get nodes` indique NotReady, ce qui est dû au fait qu'aucun réseau n'est configuré sur le cluster.

```
kubectl get nodes
NAME        STATUS      ROLES    AGE   VERSION
archlinux   NotReady    master   18m   v1.20.0
```

On va nettoyer le nœud maître, ce qui permet de déployer des pods sur le maître, puis de configurer un réseau.

```
kubectl taint nodes --all node-role.kubernetes.io/master-
node/archlinux untainted
```

Il faut maintenant déployer un réseau sur les pods du cluster. Exécuter `kubectl apply -f [podnetwork].yaml` avec l'une des options répertoriées dans https://kubernetes.io/docs/concepts/cluster-administration/addons/

Sur un très petit cluster avec une connexion de couche 2 dédiée, il n'y a pas besoin de vxlan, on peut donc appliquer le déploiement multi-arch **Flannel**.

```
curl
https://raw.githubusercontent.com/coreos/flannel/62e44c867a2846fefb68bd5f178
daf4da3095ccb/Documentation/kube-flannel.yml |  sed "s/vxlan/host-gw/" >
kube-flannel.yaml
```

Le déploiement de **kube-router** est également assez simple:

```
kubectl apply -f
https://raw.githubusercontent.com/cloudnativelabs/kube-router/v1.1.0/daemons
et/kubeadm-kuberouter.yaml
```

Maintenant on peut voir l'état des pods:

```
kubectl get pods -A
```

```
NAMESPACE     NAME                                READY   STATUS    RESTARTS
kube-system   coredns-f9fd979d6-2wp2x             1/1     Running   0
kube-system   coredns-f9fd979d6-522nz             1/1     Running   0
kube-system   etcd-archlinux                      1/1     Running   0
kube-system   kube-apiserver-archlinux            1/1     Running   0
kube-system   kube-controller-manager-archlinux   1/1     Running   0
kube-system   kube-proxy-sjxvw                    1/1     Running   0
kube-system   kube-router-8fq2m                   1/1     Running   0
kube-system   kube-scheduler-archlinux            1/1     Running   0
# kubectl get nodes
NAME        STATUS  ROLES    AGE   VERSION
archlinux   Ready   master   20m   v1.20.0
```

Containerd fournit **ctr** pour interagir avec les conteneurs, il organise tous les pods liés à kubernetes dans un espace de noms k8s.io, où on peut répertorier les pods.

```
ctr -n k8s.io c list
```

```
CONTAINER  IMAGE                                              RUNTIME
0cf8a9c17  docker.io/cloudnativelabs/kube-router:latest
io.containerd.runc.v2
2bda5427a  k8s.gcr.io/pause:3.2
io.containerd.runc.v2
4dbdaea0d  k8s.gcr.io/pause:3.2
io.containerd.runc.v2
5430728fc  k8s.gcr.io/pause:3.2
io.containerd.runc.v2
546633029  k8s.gcr.io/pause:3.2
io.containerd.runc.v2
5593c7f09  k8s.gcr.io/kube-proxy:v1.20.0
io.containerd.runc.v2
7f4dd2ff4  k8s.gcr.io/pause:3.2
```

```
io.containerd.runc.v2
96dd370ec  k8s.gcr.io/pause:3.2
io.containerd.runc.v2
aab09f1e1  docker.io/cloudnativelabs/kube-router:latest
io.containerd.runc.v2
aba203048  k8s.gcr.io/pause:3.2
io.containerd.runc.v2
b1bd5082a  sha256:0369cf4303ffdb467dc219990960a9baa8...
io.containerd.runc.v2
dc02359d9  k8s.gcr.io/coredns:1.7.0
io.containerd.runc.v2
e42de4f65  k8s.gcr.io/kube-controller-manager:v1.20.0
io.containerd.runc.v2
fc943a990  k8s.gcr.io/kube-apiserver:v1.20.0
io.containerd.runc.v2
fe49fbf3f  k8s.gcr.io/kube-scheduler:v1.20.0
io.containerd.runc.v2
```

Étiqueter les nœuds comme souhaité :

```
kubectl label node myworker2 node-role.kubernetes.io/worker=worker
```

Déploiement avec automatisé (Raspbian)

Kubespray est un outil de déploiement faisant partie de l'écosystème **Kubernetes**, modulaire, fiable, et relativement exhaustif, il n'a rien à envier aux outils de déploiements d'OpenShift que sont openshift-ansible, ou openshift-install

Ansible est un moteur d'automatisation informatique Open Source qui automatise le provisionnement, la gestion des configurations, le déploiement des applications, l'orchestration et bien d'autres processus informatiques.

L'utilisation d'une machine dédié simplifiera les interventions futures sur le cluster, s'assurant que les mêmes versions de **kubespray**, **python**, **ansible**, … soient utilisées, si l'on veut, par exemple, rajouter un noeud.

Etape 1: installer les playbooks Kubespray, et Ansible

```
git clone https://github.com/kubernetes-sigs/kubespray
cd kubespray
sudo pip install -r requirements.txt
```

Etape 2: Construire les playbooks du cluster

Un playbooks **Ansible** est un fichier YAML dans lesquels sont mentionnés toutes les tâches qu'**Ansible** doit exécuter : de la configuration de l'environnement informatique de production jusqu'à

l'orchestration de toutes les étapes de déploiement d'une application ou d'une mise à jour sur celui-ci. La syntaxe quasi-naturelle utilisée pour décrire ces tâches est particulièrement simple. Le fichier YAML commence par décrire les hôtes (ou serveurs) ciblés puis les variables du processus avant de dérouler les jobs à accomplir.

Créer le playbook inventoriant les machines composant le cluster:

```
mkdir -p inventory/rpi/group_vars/all inventory/rpi/group_vars/all/k8s-
cluster
cat <<EOF >inventory/rpi/hosts.yaml
all:
  hosts:
    pandore.friends.intra.example.com:
      etcd_member_name: pandore
      node_labels:
        my.topology/rack: rpi-rack1
        my.topology/row: rpi-row2
    hellenes.friends.intra.example.com:
      etcd_member_name: hellenes
      node_labels:
        my.topology/rack: rpi-rack2
        my.topology/row: rpi-row2
    epimethee.friends.intra.example.com:
      etcd_member_name: epimethee
      node_labels:
        my.topology/rack: rpi-rack3
        my.topology/row: rpi-row2
    pyrrha.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack1
        my.topology/row: rpi-row3
        node-role.kubernetes.io/infra: "true"
    calliope.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack2
        my.topology/row: rpi-row3
        node-role.kubernetes.io/infra: "true"
    clio.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack3
        my.topology/row: rpi-row3
        node-role.kubernetes.io/infra: "true"
    erato.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack1
        my.topology/row: rpi-row4
        node-role.kubernetes.io/worker: "true"
    euterpe.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack2
```

```
        my.topology/row: rpi-row4
        node-role.kubernetes.io/worker: "true"
    melpomene.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack3
        my.topology/row: rpi-row4
        node-role.kubernetes.io/worker: "true"
    polyhymnia.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack3
        my.topology/row: rpi-row1
        node-role.kubernetes.io/worker: "true"
    terpsichore.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack2
        my.topology/row: rpi-row1
        node-role.kubernetes.io/worker: "true"
    thalia.friends.intra.example.com:
      node_labels:
        my.topology/rack: rpi-rack1
        my.topology/row: rpi-row1
        node-role.kubernetes.io/worker: "true"
  children:
    calico-rr:
      hosts: {}
    etcd:
      hosts:
        pandore.friends.intra.example.com:
        hellenes.friends.intra.example.com:
        epimethee.friends.intra.example.com:
    kube-infra:
      hosts:
        pyrrha.friends.intra.example.com:
        calliope.friends.intra.example.com:
        clio.friends.intra.example.com:
    kube-master:
      hosts:
        pandore.friends.intra.example.com:
        hellenes.friends.intra.example.com:
        epimethee.friends.intra.example.com:
    kube-workers:
      hosts:
        erato.friends.intra.example.com:
        euterpe.friends.intra.example.com:
        melpomene.friends.intra.example.com:
        polyhymnia.friends.intra.example.com:
        terpsichore.friends.intra.example.com:
        thalia.friends.intra.example.com:
    kube-node:
      children:
        kube-master:
```

```
        kube-infra:
        kube-workers:
    k8s-cluster:
      children:
        calico-rr:
        kube-node:
EOF
```

Cet inventaire décrit un cluster composé de 3 noeuds masters+etcd, 3 noeuds “infra”, et 6 noeuds “workers”.

Créer un playbook déclarant les variables globales

```
cat <<EOF >inventory/rpi/group_vars/all/all.yaml
ansible_user: root
etcd_data_dir: /var/lib/etcd
etcd_kubeadm_enabled: false
bin_dir: /usr/local/bin
apiserver_loadbalancer_domain_name: api-k8s-arm.intra.example.com
loadbalancer_apiserver:
address: 10.42.253.52
port: 6443
loadbalancer_apiserver_localhost: false
loadbalancer_apiserver_port: 6443
upstream_dns_servers:
- 10.255.255.255
no_proxy_exclude_workers: false
cert_management: script
download_container: true
deploy_container_engine: true
EOF
```

On désigne ici le ou les serveurs DNS existants que Kubernetes devra interroger pour la résolution de noms hors clusters. Ainsi que la VIP d’un LoadBalancer, et le nom DNS correspondant, ceux-ci pointant sur l’**HAProxy** qu'on viens de déployer.

Créer un playbook déclarant les variables relatives au cluster etcd

```
cat <<EOF >inventory/rpi/group_vars/etcd.yaml
etcd_compaction_retention: 8
etcd_metrics: basic
etcd_memory_limit: 2GB
etcd_quota_backend_bytes: "2147483648"
etcd_peer_client_auth: true
etcd_deployment_type: host
EOF
```

On rajoute une limite mémoire et un quota de 2GB, pour etcd, qui doit rentrer sur les masters, disposant de 4GB de mémoire. Si l’on ne veut pas utiliser docker, comme container runtime, alors il

faudra changer le type de déploiement etcd. Par défaut conteneurisé, son déploiement dépend de commandes docker. Le déploiement de type host permettant l'utilisation d'un runtime crio, ou containerd.

Créer un playbook déclarant les variables spécifiques du cluster:

```
cat <<EOF >inventory/rpi/group_vars/k8s-cluster/k8s-cluster.yaml
kube_config_dir: /etc/kubernetes
kube_script_dir: "{{ bin_dir }}/kubernetes-scripts"
kube_manifest_dir: "{{ kube_config_dir }}/manifests"
kube_cert_dir: "{{ kube_config_dir }}/ssl"
kube_token_dir: "{{ kube_config_dir }}/tokens"
kube_api_anonymous_auth: true
kube_version: v1.19.5
local_release_dir: /tmp/releases
retry_stagger: 5
kube_cert_group: kube-cert
kube_log_level: 2
credentials_dir: "{{ inventory_dir }}/credentials"
kube_oidc_auth: false
kube_token_auth: true
kube_network_plugin: flannel
kube_network_plugin_multus: false
kube_service_addresses: 10.233.128.0/18
kube_pods_subnet: 10.233.192.0/18
kube_network_node_prefix: 24
kube_apiserver_ip: "{{
kube_service_addresses|ipaddr('net')|ipaddr(1)|ipaddr('address') }}"
kube_apiserver_port: 6443
kube_apiserver_insecure_port: 0
kube_proxy_mode: ipvs
authorization_modes: ['Node', 'RBAC']
kube_proxy_strict_arp: false
kube_encrypt_secret_data: false
cluster_name: cluster.local
ndots: 2
dns_mode: coredns
enable_nodelocaldns: true
nodelocaldns_ip: 169.254.25.10
nodelocaldns_health_port: 9254
enable_coredns_k8s_external: false
enable_coredns_k8s_endpoint_pod_names: false
resolvconf_mode: none
deploy_netchecker: false
skydns_server: "{{
kube_service_addresses|ipaddr('net')|ipaddr(3)|ipaddr('address') }}"
skydns_server_secondary: "{{
kube_service_addresses|ipaddr('net')|ipaddr(4)|ipaddr('address') }}"
dns_domain: "{{ cluster_name }}"
container_manager: containerd
```

```
kata_containers_enabled: false
kubeadm_certificate_key: "{{ lookup('password', credentials_dir +
'/kubeadm_certificate_key.creds length=64 chars=hexdigits') | lower }}"
k8s_image_pull_policy: IfNotPresent
kubernetes_audit: false
dynamic_kubelet_configuration: false
default_kubelet_config_dir: "{{ kube_config_dir }}/dynamic_kubelet_dir"
dynamic_kubelet_configuration_dir: "{{ kubelet_config_dir |
default(default_kubelet_config_dir) }}"
podsecuritypolicy_enabled: true
kubeconfig_localhost: true
kubectl_localhost: false
system_reserved: true
system_memory_reserved: 128Mi
system_cpu_reserved: 250m
system_master_memory_reserved: 256Mi
system_master_cpu_reserved: 250m
volume_cross_zone_attachment: false
persistent_volumes_enabled: false
event_ttl_duration: 1h0m0s
force_certificate_regeneration: false
minimal_node_memory_mb: 896
kube_proxy_nodeport_addresses: >-
  {%- if kube_proxy_nodeport_addresses_cidr is defined -%}
  [{{ kube_proxy_nodeport_addresses_cidr }}]
  {%- else -%}
  []
  {%- endif -%}
EOF
```

Beaucoup de variables ci-dessus reprennent des défaults. Entre autre changements, on note:

1. le plugin réseau, “flannel“, et la définition des “kube*service*addresses” et “kube*pods*subnet” ou le “resolvconf*mode“. - Le runtime conteneur : “containerd“. - L’activation des PodSecurityPolicy. - Les “system*” vont réserver un peu de CPU et mémoire pour l’OS des noeuds.
2. Le “minimal_node_memory” sera indispensable, lorsque certains noeuds ont moins d’1Gi de mémoire – les RPI 3b+ remontent 924Mi.

Il sera possible de configurer le runtime conteneur du cluster, en créant un playbook tel que le suivant :

```
cat <<EOF >inventory/rpi/group_vars/all/containerd.yaml
containerd_config:
  grpc:
    max_recv_message_size: 16777216
    max_send_message_size: 16777216
  debug:
    level: ""
  registries:
```

```
    docker.io: "https://registry-1.docker.io"
    katello: "https://katello.vms.intra.example.com:5000"
    "registry.registry.svc.cluster.local:5000":
"http://registry.registry.svc.cluster.local:5000"
  max_container_log_line_size: -1
  metrics:
    address: ""
    grpc_histogram: false
EOF
```

Kubespray permet le déploiement de diverses applications optionnelles :

```
cat <<EOF >inventory/rpi/group_vars/k8s-cluster/addons.yaml
helm_enabled: false
registry_enabled: false
metrics_server_enabled: true
metrics_server_kubelet_insecure_tls: true
metrics_server_metric_resolution: 60s
metrics_server_kubelet_preferred_address_types: "InternalIP"
local_path_provisioner_enabled: false
local_volume_provisioner_enabled: false
cephfs_provisioner_enabled: false
rbd_provisioner_enabled: false
ingress_nginx_enabled: true
ingress_ambassador_enabled: false
ingress_alb_enabled: false
cert_manager_enabled: false
metallb_enabled: false
EOF
```

Dans l'exemple, on active **metrics_server**, et l'**Ingress Controller Nginx**.

Etape 3: Installer la clé SSH du maître

Enfin, générer une clé SSH sur le noeud Ansible, et l'installer sur tous les noeuds composant le cluster :

```
ssh-keygen -t rsa -b 4096 -N '' -f ~/.ssh/id_rsa
for i in pandore hellenes .... terpsichore thalia
do ssh-copy-id -i ~/.ssh/id_rsa.pub root@$i.friends.intra.example.com; done
...
ansible -m ping -i inventory/rpi/hosts.yaml all
```

Etape 4: Procéder au déploiement

```
ansible-playbook -i inventory/rpi/hosts.yaml cluster.yml 2>&&1 | tee -a
```

```
deploy.$(date +%s).log
```

Si le déploiement échoue, on pourra corriger l'inventaire et relancer cette même commande, à moins que l'erreur ne se soit produite lorsqu'ansible lance l'initialisation du cluster Kubernetes (kubeadm init), auquel cas on devrait d'abord lancer le playbook de reset, pour réinitialiser les noeuds :

```
ansible-playbook -i inventory/rpi/hosts.yaml reset.yml
```

Si tout se passe bien, le déploiement ne prendra pas plus d'une trentaine de minutes.

```
...
PLAY RECAP ********************************************
calliope.friends.intra.example.com : ok=285 changed=29 unreachable=0
failed=0 skipped=514 rescued=0 ignored=1
clio.friends.intra.example.com : ok=285 changed=29 unreachable=0 failed=0
skipped=514 rescued=0 ignored=1
epimethee.friends.intra.example.com : ok=440 changed=67 unreachable=0
failed=0 skipped=864 rescued=0 ignored=1
erato.friends.intra.example.com : ok=285 changed=29 unreachable=0 failed=0
skipped=514 rescued=0 ignored=1
euterpe.friends.intra.example.com : ok=285 changed=29 unreachable=0 failed=0
skipped=514 rescued=0 ignored=1
hellenes.friends.intra.example.com : ok=442 changed=68 unreachable=0
failed=0 skipped=862 rescued=0 ignored=1
localhost : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
melpomene.friends.intra.example.com : ok=285 changed=29 unreachable=0
failed=0 skipped=514 rescued=0 ignored=1
pandore.friends.intra.example.com : ok=497 changed=86 unreachable=0 failed=0
skipped=924 rescued=0 ignored=1
pyrrha.friends.intra.example.com : ok=308 changed=30 unreachable=0 failed=0
skipped=604 rescued=0 ignored=1
polyhymnia.friends.intra.example.com : ok=285 changed=29 unreachable=0
failed=0 skipped=514 rescued=0 ignored=1
terpsichore.friends.intra.example.com : ok=285 changed=29 unreachable=0
failed=0 skipped=514 rescued=0 ignored=1
thalia.friends.intra.example.com : ok=285 changed=29 unreachable=0 failed=0
skipped=514 rescued=0 ignored=1
```

```
kubectl get nodes
NAME                                  STATUS   ROLES    AGE     VERSION
calliope.friends.intra.example.com    Ready    infra    7m14s   v1.19.5
clio.friends.intra.example.com        Ready    infra    7m14s   v1.19.5
epimethee.friends.intra.example.com   Ready    master   9m26s   v1.19.5
erato.friends.intra.example.com       Ready    worker   7m14s   v1.19.5
euterpe.friends.intra.example.com     Ready    worker   7m1s    v1.19.5
hellenes.friends.intra.example.com    Ready    master   9m24s   v1.19.5
melpomene.friends.intra.example.com   Ready    worker   7m1s    v1.19.5
```

```
pandore.friends.intra.example.com         Ready    master   9m59s   v1.19.5
pyrrha.friends.intra.example.com          Ready    infra    7m14s   v1.19.5
polyhimnia.friends.intra.example.com      Ready    worker   7m1s    v1.19.5
terpsichore.friends.intra.example.com     Ready    worker   7m13s   v1.19.5
thalia.friends.intra.example.com          Ready    worker   7m      v1.19.5
```

A ce stade, on peut intégrer le cluster avec une solution de stockage externe – au plus simple, un serveur NFS.

On reste limité par les modules kernels fournis avec Raspbian : pour s'interfacer avec avec un cluster Ceph, il faudra recompiler le kernel, ou s'intéresser à rbd-ndb.

1)

Lors de l'amorçage du cluster **Kubernetes**, l'un des contrôles en amont effectués par **Kubernetes** consiste à voir si on a un swap actif. S'il trouve une partition de swap active, il échouera et s'arrêtera là. Afin de réussir les contrôles en amont, il faut désactiver le swap sur chaque nœud du cluster Kubernetes.

2)

Les groupes de contrôle sont une fonctionnalité fournie par le noyau Linux pour gérer, restreindre et auditer des groupes de processus. Par rapport à d'autres approches les groupes de contrôle sont plus flexibles car ils peuvent opérer sur des sous-ensembles de processus et éventuellement avec différents utilisateurs du système