Kubernetes / Raspberry Pi workshop Quintor 17 november 2016

Environment config

Server	Adres
WLAN	Raspberry Pi Network / quintor2016
RPI SSH Login	host:rpi-node-* usr: root pwd: hypriot

1. Kubernetes worker node

Op iedere Raspberry Pi is Kubernetes voorgeinstalleerd om als worker node toegevoegd te worden aan het kubernetes cluster.

Let’s check if everything is working correctly. Two docker daemon processes must be running.

```
$ ps -ef|grep docker

root       318     1  0 12:00 ?        00:00:24 /usr/bin/docker daemon -H
unix:///var/run/docker-bootstrap.sock -p /var/run/docker-bootstrap.pid
--storage-driver=overlay --storage-opt dm.basesize=10G --iptables=false
--ip-masq=false --bridge=none --graph=/var/lib/docker-bootstrap

root       697     1  8 12:01 ?        00:19:24 /usr/bin/docker daemon -H
fd:// --insecure-registry buildserver:5000 --storage-driver=overlay -D
--mtu=1472 --bip=10.1.62.1/24

root     30240 30106  0 15:48 pts/0    00:00:00 grep docker
```

The flannel container must be up.

```
$ docker -H unix:///var/run/docker-bootstrap.sock ps

CONTAINER ID        IMAGE                             COMMAND
CREATED             STATUS              PORTS               NAMES

2cceeaa7a06a        quay.io/coreos/flannel:v0.6.1-arm   "/opt/bin/flanneld
--"   About an hour ago   Up About an hour
kube_flannel_d1509
```

The flannel network segment assigned to the node (see flannel0 => 10.1.xxx.0) must be used by the docker0 network bridge (10.1.xxx.1).

```
$ ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:2a:e1:bc:f2
          inet addr:10.1.62.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:2aff:fee1:bcf2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:45 errors:0 dropped:0 overruns:0 frame:0
          TX packets:43 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2800 (2.7 KiB)  TX bytes:6171 (6.0 KiB)


flannel0  Link encap:UNSPEC  HWaddr
00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          inet addr:10.1.62.0  P-t-P:10.1.62.0  Mask:255.255.0.0
          UP POINTOPOINT RUNNING NOARP MULTICAST  MTU:1472  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:500
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

The hyperkube kubelet and proxy must be up.

```
$ docker ps

CONTAINER ID        IMAGE
COMMAND                  CREATED             STATUS              PORTS
NAMES

cc27ddfd55a0        gcr.io/google_containers/hyperkube-arm:v1.3.6
"/hyperkube proxy --m"   About an hour ago   Up About an hour
kube_proxy_5e947

ef4ecc0c46da        gcr.io/google_containers/hyperkube-arm:v1.3.6
"/hyperkube kubelet -"   About an hour ago   Up About an hour
kube_kubelet_17c52
```

Once all the services on the worker node are up and running we can check that the node is added to the cluster on the master node. That can be done with `kubectl`, a tool for managing kubernetes. `Kubectl` for ARM can be downloaded from googleapis storage. The default cluster server (master node) is configured, so that we don't have to specify it every time we run a command with kubectl. You can specify multiple clusters and use contexts to switch between them.

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    server: http://rpi-master-1:8080
  name: rpi-cluster
contexts:
- context:
    cluster: rpi-cluster
    namespace: rpi-node-61
    user: ""
  name: rpi-cluster
current-context: rpi-cluster
kind: Config
preferences: {}
users: []
```

`kubectl get nodes` shows which cluster nodes are registered along with its status.

```
$ kubectl get nodes
NAME            STATUS    AGE
10.150.42.100   Ready     2h
10.150.42.101   Ready     2h
```

2. Kubernetes

Running a container on kubernetes

An easy way to test the cluster is by running a simple docker image for ARM like the `rpi-nginx` one. `kubectl run` can be used to run the image as a container in a pod. `kubectl get pods` shows the pods that are registered along with its status. `kubectl describe` gives more detailed information on a specific resource. Each pod (container) get a unique ip-address assigned wihtin the cluster and is accessible on that througout the cluster, thanks to flannel overlay network. A pod can be deleted using kubectl delete pod/deployment/service <name>. Note that the run command creates a **deployment** (http://kubernetes.io/docs/user-guide/deployments/) which will ensure a crashed or deleted pod is restored. To remove your deployment, use kubectl delete deployment nginx. **(kubectl help is your friend!)** The --port flag exposes the pods to the internal network. The --labels flag ensures your pods are visible. Please use both flags.
Note that your nginx pod may seem to be stuck at "ContainerCreating" because it has to download the image first.

```
$ kubectl run nginx --image=buildserver:5000/rpi-nginx --port=80
--labels="run=nginx,visualize=true"
deployment "nginx" created

$ kubectl get pods -o wide
NAME                      READY     STATUS    RESTARTS   AGE       IP
NODE
nginx-1665122148-4amzl    1/1       Running   0          47s
10.1.87.2    10.150.42.100
$ kubectl describe pod nginx-1665122148-4amzl
Name:  nginx-1665122148-4amzl
Namespace: rpi-node-61
Node:  10.150.42.100/10.150.42.100
Start Time: Wed, 19 Oct 2016 02:18:38 +0200
Labels:  pod-template-hash=1665122148
  run=nginx
  visualize=true
Status:  Running
IP:  10.1.87.2
Controllers: ReplicaSet/nginx-1665122148
Containers:
$ kubectl get rs -o=wide
NAME               DESIRED   CURRENT   AGE       CONTAINER(S)   IMAGE(S)
SELECTOR
nginx-1665122148   1         1         3m        nginx
buildserver:5000/rpi-nginx
pod-template-hash=1665122148,run=nginx,visualize=true

$ kubectl get deployment -o=wide
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     1         1         1            1           4m
```

Now the container is running with kubernetes, the NGINX application is directly accessible via its IP address within the kubernetes cluster. Note that this is an IP address within the flannel overlaying network and is not accessible from outside the cluster. Also note that we do not have to specify any port mappings from the container to the host.

```
$ curl http://10.1.87.2/
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body bgcolor="white" text="black">
<center><h1>Welcome to nginx!</h1></center>
</body>
</html>
```

Exposing containers on kubernetes

Now the pod is running, but the application is not generally accessible. That can be achieved by creating a service in kubernetes. The service will have a cluster IP-address assigned, which is the IP-address the service is avalailable at within the cluster (`10.0.0.*`). Use the IP-address of your Raspberry Pi node as external IP and the service becomes available outside of the cluster (e.g.10.150.42.103 in my case). Check in your browser that `http://<ip-address-of-your-node>:90/` is available.

```
$ kubectl expose deployment nginx --port=90 --target-port=80
--external-ip=10.150.42.103
service "nginx" exposed

$ kubectl get svc
NAME      CLUSTER-IP   EXTERNAL-IP     PORT(S)   AGE
nginx     10.0.0.102   10.150.42.103   90/TCP    57s
$ curl http://10.150.42.103:90
AND/OR (but not accessible to the outside world)
$ curl http://10.0.0.102:90
<html>
<head>
<title>Welcome to nginx!</title>
</head>
<body bgcolor="white" text="black">
<center><h1>Welcome to nginx!</h1></center>
</body>
</html>
```

Scaling

The number of pod serving a service can easily be scaled via kubectl. Use `kubectl scale` to do so. Check out the visualizer the moment when you execute the scale command.

```
$ kubectl scale --replicas=3 deployment nginx
deployment "nginx" scaled

$ kubectl get pods -o=wide
NAME                     READY     STATUS              RESTARTS   AGE
IP            NODE
nginx-1665122148-4amzl   1/1       Running             0          13m
10.1.87.2     10.150.42.100
nginx-1665122148-peep7   0/1       ContainerCreating   0          1m
<none>        10.150.42.176
nginx-1665122148-shm28   0/1       ContainerCreating   0          1m
<none>        10.150.42.183
```

Doing a rolling update with Kubernetes (no service downtime)

In order to demonstrate a rolling update, we will use some prepared nginx containers which serve different static html depending on the version. Please remove your current deployment and deploy version 1 of this image, with 4 replicas, exposing port 80 on the pods (tip: if you don't remove the service exposing your raspi's port 90 to the world you can reuse it for this deployment). **This step can take some time since the image has to be downloaded**.

```
$ kubectl delete deployment nginx
deployment "nginx" deleted

$ kubectl run nginx --image=buildserver:5000/rpi-nginx-withcontent:3
--port=80 --replicas=4 --labels="run=nginx,visualize=true"
deployment "nginx" created

$ kubectl expose deployment nginx --port=90 --target-port=80
--external-ip=<my node's ip>
service "nginx" exposed
```

Now we can see Kubernetes' full magic at work. We will edit the deployment (http://kubernetes.io/docs/user-guide/deployments/#updating-a-deployment) to start using the second version of the image, which will be rolled out by the system, replacing one pod at a time. The service will never go down, during the update a user simply gets served either the old or the new version. To kick the update off, you must edit the deployment. Take note of the different parts of this deployment file. You can write such a file yourself to deploy your applications, which is often more practical than having a bloke or gall hammer commands into a cluster with kubectl. For now, change the container image to version 4. For those unfamiliar with this editor, start editing with insert, stop editing with esc, save the result with :w and quit with :q.

```
$ kubectl edit deployment nginx
deployment nginx edited

$ kubectl get deployments
```

Creating services, replicationcontrolers and pods from configuration files

Kubernetes resources can also be created from configuration files instead of via the command line. This makes it easy to put this kubernetes configuration in version control and maintain it from there.

A nice thing is that kubectl lets you

```
$ kubectl delete svc nginx
service "nginx" deleted

$ kubectl delete deployment nginx
deployment "nginx" deleted

$ kubectl create -f nginx-deployment.yaml
replicationcontroller "nginx" created

$ kubectl create -f nginx-svc.yaml
service "nginx" created

Now edit the deployment yaml file to use a different image version (4->3)
$ kubectl replace -f nginx-deployment.yaml
```

3. Deploying an three tier application

Creating and claiming persisted volumes

The buildserver also hosts NFS service providing multiple volumes for mounting. In Kubernetes you can make a volume available for usage by creating a Persisted Volume.

Edit the nfs-pvc.yaml file so that the nfs share path matches your node. Also change the PV name to a unique value.

```
$ kubectl create -f nfs-pv.yaml
persistentvolume "nfs-share-61" created
$ kubectl get pv
NAME           CAPACITY   ACCESSMODES   STATUS      CLAIM
REASON    AGE
nfs-share-61   1Gi        RWO           Available
28s
```

Before the volume can be used it needs to be claimed for a certain application. This is done by creating a Persisted Volume Claim.

```
$ kubectl create -f nfs-pvc.yaml
persistentvolumeclaim "mysql-pv-claim" created

$ kubectl get pvc
NAME             STATUS    VOLUME         CAPACITY   ACCESSMODES   AGE
mysql-pv-claim   Bound     nfs-share-61   1Gi        RWO           12s
$ kubectl get pv
NAME           CAPACITY   ACCESSMODES   STATUS    CLAIM
REASON    AGE
nfs-share-61   1Gi        RWO           Bound
rpi-node-61/mysql-pv-claim                        5m
```

More information can be found here: http://kubernetes.io/docs/user-guide/persistent-volumes/

The deployment and service yaml files in the "assignment-3" folder are incomplete. Open the files and lookup the missing values in the Kubernetes documentation http://kubernetes.io/docs/

Deploy the Mysql container and use the PVC for the data storage and create a service for mysql. **Edit the** cddb-mysql-service.yaml **file so it uses the IP of your node.**

```
$ kubectl create -f cddb-mysql-deployment.yaml
deployment "cddb-mysql" created

$ kubectl create -f cddb-mysql-service.yaml
service "cddb-mysql" created
```

Deploy the backend container and create a service for the backend. **Edit the** cddb-backend-service.yaml **file so it uses the IP of your node.**

```
$ kubectl create -f cddb-backend-deployment.yaml
deployment "cddb-backend" created

$ kubectl create -f cddb-backend-service.yaml
service "cddb-backend" created
```

Deploy the frontend container and create a service for the fronend. **Edit the** cddb-frontend-service.yaml **file so it uses the IP of your node.**

```
$ kubectl create -f cddb-frontend-deployment.yaml
deployment "cddb-frontend" created

$ kubectl create -f cddb-backend-frontend.yaml
service "cddb-frontend" created
```

Test that the application is working using a browser and that it stores the data in the database.

You can scale up the frontend and backend layer. But the mysql layer cannot be scaled. Though Kubernetes manages the persisted volumes and remounts them on a different node when needed. To test this find out on which node the mysql pod is running and pull the network cable from the node and look what happens.