RX-M Cloud Native Consulting

Advanced Kubernetes

Lab 4 – Services and kube-proxy

In the last lab our cluster became much more complete as we added the scheduler and controller manager to the contingent of master services. The `kube-apiserver` backed by `etcd` gave us the semblance of a cluster. The `kubelet` gave us actual nodes that we could run workloads (pods) on. The `kube-scheduler` allowed us to let Kubernetes determine the best place to run the workloads and the `kube-controller-manager` gave us the ability to scale the number of containers implementing our workloads.

In this lab we are going to add the final piece of core Kubernetes functionality: support for services. In Kubernetes, the `kube-proxy` performs the functions necessary to create services.

Before starting make sure that your Controller Manager, Scheduler, API Server and etcd are running on the master and that the kubelets are running on both nodea and nodeb. Also make sure that your cluster has no resources in the default namespace (delete all prior pods).

If you need to restart any/all of your services, the commands to do so have been put below for convenience.

etcd

```
sudo rm -rf /var/lib/etcd

rm -rf ~/default.etcd/

etcd
```

api-server

```
sudo $HOME/k8s/_output/bin/kube-apiserver \
--etcd-servers=http://localhost:2379 \
--service-cluster-ip-range=10.0.0.0/16 \
--insecure-bind-address=0.0.0.0  \
--disable-admission-plugins=ServiceAccount
```

controller manager

```
$HOME/k8s/_output/bin/kube-controller-manager --kubeconfig=nodea.conf
```

scheduler

```
$HOME/k8s/_output/bin/kube-scheduler --kubeconfig=nodea.conf
```

kubelet nodea

```
sudo rm -rf /var/lib/kubelet

sudo $HOME/k8s/_output/bin/kubelet \
--kubeconfig=nodea.conf \
--config=nodea.yaml \
--allow-privileged=true \
--runtime-cgroups=/systemd/machine.slice \
--kubelet-cgroups=/systemd/machine.slice \
--pod-infra-container-image=k8s.gcr.io/pause:3.1
```

kubelet nodeb

```
sudo rm -rf /var/lib/kubelet

sudo $HOME/kube-bin/kubelet \
--kubeconfig=nodeb.conf \
--config=nodeb.yaml \
--allow-privileged=true \
--runtime-cgroups=/systemd/machine.slice \
--kubelet-cgroups=/systemd/machine.slice \
--pod-infra-container-image=k8s.gcr.io/pause:3.1
```

1. Services and networking

To begin we'll re-run our deployment from lab 3; as a reminder, it looks like this:

```
ubuntu@nodea:~$ cat testdep.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl create -f testdep.yaml

deployment.apps/nginx-deployment created
user@nodea:~$
```

What if we want to retrieve some web pages from one of the nginx Pods in our Deployment? Some questions:

- Do we care which one we get the pages from?
- Do we want to be wired to a single Pod, what if it crashes?

The answers to these questions are typically "No" and "No". Deployments create ReplicaSets and ReplicaSets create replicas. The reason we have replicas is for scale and HA (i.e. to ensure that failure of one replica does not cause failure of the whole). In essence we want access to the "service" without being tied to the Pod that implements it.

In Kubernetes, "Services" provide a layer of abstraction on top of a set of Pod replicas implementing the service. Services identify the pods that implement them using a label selector. Identify the labels assigned to your Pods:

```
ubuntu@nodea:~$  kubectl get pods --show-labels

NAME                               READY    STATUS     RESTARTS    AGE    LABELS
nginx-deployment-6dd86d77d-h9ffz   1/1      Running    0           20m    app=nginx,pod-template-hash=6dd86d77d
nginx-deployment-6dd86d77d-ll595   1/1      Running    0           20m    app=nginx,pod-template-hash=6dd86d77d
ubuntu@nodea:~$
```

Our template defines the label "`app=nginx`". Note that the pod also contains a template hash. This allows you to identify the template that was used to create the pod and to detect pods that are not implementing the current template.

Create a Kubernetes Service that selects the two Pods to back a service called "nsvc":

```
ubuntu@nodea:~$ vim nsvc.yaml
ubuntu@nodea:~$ cat nsvc.yaml

apiVersion: v1
kind: Service
metadata:
  name: nsvc
spec:
  ports:
    - port: 2000
      targetPort: 80
  selector:
    app: nginx
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl create -f nsvc.yaml

service/nsvc created
ubuntu@nodea:~$
```

Verify the creation of the service:

```
ubuntu@nodea:~$ kubectl get svc

NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.0.0.1      <none>        443/TCP    1h
nsvc         ClusterIP   10.0.51.206   <none>        2000/TCP   21s
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl describe svc nsvc

Name:              nsvc
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          app=nginx
Type:              ClusterIP
IP:                10.0.51.206
Port:              <unset>  2000/TCP
TargetPort:        80/TCP
Endpoints:         172.17.0.2:80,172.17.0.2:80
Session Affinity:  None
Events:            <none>
ubuntu@nodea:~$
```

The API Server has created our service and given it an IP (in the example above) of 10.0.51.206 and a port of 2000 (as we requested in the spec).

You may have noticed in earlier labs the kube-apiserver flag `--service-cluster-ip-range=10.0.0.0/16`. This range is the pool of IPs that the ClusterIP pulls from for service IPs (10.0.51.206 in our example.) Often, this IP is called the VIP (virtual IP), ClusterIP, or just IP. This range *must not overlap* with your nodes subnet (192.168.225.0/24) or your container network(S) (172.17.0.0/16).

Try curling this end point:

```
ubuntu@nodea:~$ curl -I 10.0.51.206:2000

curl: (7) Failed to connect to 10.0.0.253 port 2000: Connection refused
ubuntu@nodea:~$
```

No luck. This is a Virtual IP (VIP). Virtual IPs are, well, virtual. They are not connected with real listening endpoints, rather they are hardware/software table entries that redirect traffic somewhere else. In Kubernetes the process responsible for creating the rules on every node to redirect VIP traffic is the Kube-Proxy and we have not started it yet.

The service description also reports endpoints associated with each of the Pods running the service.

Try curling one of them:

```
ubuntu@nodea:~$ curl -I 172.17.0.2:80

HTTP/1.1 200 OK
Server: nginx/1.7.9
Date: Thu, 18 Jan 2019 03:32:11 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 23 Dec 2014 16:25:09 GMT
Connection: keep-alive
ETag: "54999765-264"
Accept-Ranges: bytes
ubuntu@nodea:~$
```

This works but only if you try it on the machine that the Pod is running on. Why? Because our two nodes are using default Docker installations and all current Docker installations create containers on the docker0 bridge and the docker0 bridge has the subnet 172.17.0.0/16 by default, *on every node*! You may, for example see your two service pods having the exact same IP address!

Examine the docker0 network on your two nodes:

nodea:

```
ubuntu@nodea:~$ ip a show dev docker0

3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
```

```
    link/ether 02:42:10:a3:5a:ce brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:10ff:fea3:5ace/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@nodea:~$
```

nodeb:

```
ubuntu@nodeb:~$ ip a show dev docker0

3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:2e:20:db:74 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2eff:fe20:db74/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@nodeb:~$
```

There are many many ways to configure networking in a Kubernetes cluster. The simplest way is to statically configure each node in the cluster with a unique docker0 subnet and then to set routes in every node to all of the other node docker0 subnets. This will ensure that each node assigns unique IPs to its Pods and that all Pods can reach each other directly via the static routes.

Docker automatically configures a route on the Docker host to the docker0 bridge by default. Display the route table on nodea for example:

```
ubuntu@nodea:~$ ip route

default via 172.31.16.1 dev eth0
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1 linkdown
172.31.16.0/20 dev eth0  proto kernel  scope link  src 172.31.28.198
ubuntu@nodea:~$
```

All 172.17/16 traffic will be placed on the docker0 Linux Bridge (which acts like an L2 switch).

2. Configuring a flat network on nodeb

While static network configuration is straightforward and does not involve SDN, tunnels, or other slow downs, it is static. This means that it requires a static infrastructure to be reliable and changes require work. We'll try SDN in a later lab, for now we'll configure static routes and docker0 bridges with non-overlapping subnets.

To begin delete all of the resources on your cluster.

Deployments:

```
ubuntu@nodea:~$ kubectl get deploy

NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2/2     2            2           8s
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl delete deploy nginx-deployment

deployment.extensions "nginx-deployment" deleted
ubuntu@nodea:~$
```

Services:

```
ubuntu@nodea:~$ kubectl get svc

NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes   10.0.0.1     <none>        443/TCP    24m
nsvc         10.0.0.253   <none>        2000/TCP   6m
ubuntu@nodea:~$
```

The kubernetes service is created by the API server and is the VIP for the API server, allowing any pod in the cluster to easily lookup and call the API Server. Delete the nsvc service you created but **do not** delete the kubernetes service.

```
ubuntu@nodea:~$ kubectl delete svc nsvc

service "nsvc" deleted
ubuntu@nodea:~$
```

Verify that all pods are terminated:

```
ubuntu@nodea:~$ kubectl get po

No resources found.
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ docker container ls -a

CONTAINER ID        IMAGE        COMMAND          CREATED             STATUS              PORTS
NAMES
ubuntu@nodea:~$
```

Change to nodeb and verify that no containers are running under Docker:

```
ubuntu@nodeb:~$ docker container ls -a

CONTAINER ID        IMAGE         COMMAND          CREATED          STATUS          PORTS           NAMES
ubuntu@nodeb:~$
```

Now we can change *nodeb's* docker0 subnet.

On Ubuntu 16.04 Docker runs as a systemd service. We can augment the service configuration by editing the docker daemon configuration file. Locate the systemd Docker service file (press 'q' to exit the log listing):

```
ubuntu@nodeb:~$ sudo systemctl status docker.service --full

● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Sat 2019-03-30 07:42:43 UTC; 17h ago
     Docs: https://docs.docker.com
 Main PID: 4243 (dockerd)
    Tasks: 0
   Memory: 33.8M
      CPU: 838ms
   CGroup: /system.slice/docker.service
           ‣ 4243 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Mar 30 09:02:32 nodeb dockerd[4243]: time="2019-03-30T09:02:32.953518958Z" level=info msg="ignoring event"
module=libcontainerd namespace=moby topic=/tasks/delet
Mar 30 09:02:33 nodeb dockerd[4243]: time="2019-03-30T09:02:33.000393918Z" level=info msg="ignoring event"
module=libcontainerd namespace=moby topic=/tasks/delet
Mar 30 09:02:33 nodeb dockerd[4243]: time="2019-03-30T09:02:33.077128619Z" level=info msg="ignoring event"
module=libcontainerd namespace=moby topic=/tasks/delet
Mar 30 09:02:33 nodeb dockerd[4243]: time="2019-03-30T09:02:33.115933550Z" level=info msg="ignoring event"
module=libcontainerd namespace=moby topic=/tasks/delet
Mar 30 09:02:33 nodeb dockerd[4243]: time="2019-03-30T09:02:33.194496263Z" level=info msg="ignoring event"
module=libcontainerd namespace=moby topic=/tasks/delet
ubuntu@nodeb:~$
```

Now inspect the Docker service file:

```
ubuntu@nodeb:~$ cat /lib/systemd/system/docker.service

[Unit]
Description=Docker Application Container Engine
Documentation=https://docs.docker.com
BindsTo=containerd.service
After=network-online.target firewalld.service containerd.service
Wants=network-online.target
Requires=docker.socket

[Service]
Type=notify
# the default is not to use systemd for cgroups because the delegate issues still
# exists and systemd currently does not support the cgroup feature set required
# for containers run by docker
ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always

# Note that StartLimit* options were moved from "Service" to "Unit" in systemd 229.
# Both the old, and new location are accepted by systemd 229 and up, so using the old location
# to make them work for either version of systemd.
StartLimitBurst=3

# Note that StartLimitInterval was renamed to StartLimitIntervalSec in systemd 230.
# Both the old, and new name are accepted by systemd 230 and up, so using the old name to make
# this option work for either version of systemd.
StartLimitInterval=60s

# Having non-zero Limit*s causes performance problems due to accounting overhead
# in the kernel. We recommend using cgroups to do container-local accounting.
LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity

# Comment TasksMax if your systemd version does not supports it.
# Only systemd 226 and above support this option.
TasksMax=infinity

# set delegate yes so that systemd does not reset the cgroups of docker containers
Delegate=yes

# kill only the docker process, not all processes in the cgroup
KillMode=process

[Install]
WantedBy=multi-user.target
ubuntu@nodeb:~$
```

We will make changes to Docker's configuration in subsequent steps.

2.a. Assign docker0 a unique subnet

The default location of the docker daemon configuration file on Linux is `/etc/docker/daemon.json`. Add the `bip` option to the daemon.json with a value that will cause the docker0 bridge to use subnet 172.18/16. First **stop the kubelet** (with control+c) **and Docker** (using systemctl) on **nodeb**:

1. Press control+c in the kubelet terminal to exit the kubelet
2. Shutdown docker:

```
ubuntu@nodeb:~$ sudo systemctl stop docker
```

```
ubuntu@nodeb:~$
```

Next, remove the old bridge IP address (it has the old, now incorrect subnet):

```
ubuntu@nodeb:~$ ip a show docker0

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:22:9a:14:d0 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::42:22ff:fe9a:14d0/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@nodeb:~$
```

```
ubuntu@nodeb:~$ sudo ip addr del 172.17.0.1/16 dev docker0

ubuntu@nodeb:~$
```

Now update the Docker startup command so that docker assigns the docker0 bridge the 172.18 subnet with the bridge address of 0.1:

```
ubuntu@nodeb:~$ sudo vim  /etc/docker/daemon.json

ubuntu@nodeb:~$ cat /etc/docker/daemon.json

{
        "bip": "172.18.0.1/16"
}
ubuntu@nodeb:~$
```

Restart docker:

```
ubuntu@nodeb:~$ sudo systemctl start docker
```

```
ubuntu@nodeb:~$
```

Verify the configuration:

```
ubuntu@nodeb:~$ ip a show docker0

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:2e:20:db:74 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:2eff:fe20:db74/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@nodeb:~$
```

Perfect.

Now nodea has subnet 172.17 under its control and nodeb has subnet 172.18 under its control.

We have more work to do however. Test run an nginx container on nodeb:

```
ubuntu@nodeb:~$ docker container run -d nginx

5a8d5ee42d140010871186e287786e600f101b4f03f1c2550d9eacb708937817
ubuntu@nodeb:~$
```

Now try to curl the container on port 80:

```
ubuntu@nodeb:~$ docker container inspect $(docker container ls \
--filter=ancestor=nginx -q) -f "{{ .NetworkSettings.IPAddress }}"

172.18.0.2
ubuntu@nodeb:~$
```

```
ubuntu@nodeb:~$ curl -I 172.18.0.2
```

```
HTTP/1.1 200 OK
Server: nginx/1.15.10
Date: Sun, 31 Mar 2019 00:49:22 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 26 Mar 2019 14:04:38 GMT
Connection: keep-alive
ETag: "5c9a3176-264"
Accept-Ranges: bytes
ubuntu@nodeb:~$
```

Perfect, we can reach the container. This works because the host has a route to 172.18 (Docker creates it automatically):

```
ubuntu@nodeb:~$ ip route

default via 172.31.16.1 dev eth0
172.18.0.0/16 dev docker0  proto kernel  scope link  src 172.18.0.1
172.31.16.0/20 dev eth0  proto kernel  scope link  src 172.31.30.148
ubuntu@nodeb:~$
```

Now *change machines to nodea* and retry the curl experiment:

```
ubuntu@nodea:~$ curl 172.18.0.2

curl: (7) Failed to connect to 172.18.0.2 port 80: Connection refused
ubuntu@nodea:~$
```

> N.B. your terminal may hang for several moments before returning with the above message.

What is wrong?

nodea, of course, has no way to know where this new 172.18 subnet is. What we need is a route on nodea that forwards all 172.18 traffic to nodeb. nodeb already has a route to its docker0 bridge for all 172.18 traffic (as we have seen) and so it will forward the traffic to docker0 completing the route.

2.b. Create a route on nodea to docker0 on nodeb

First **identify the external IP of nodeb**, this is where we will need to route 172.18 traffic to from nodea:

```
ubuntu@nodeb:~$ ip a show eth0

2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default qlen 1000
    link/ether 02:d8:c0:66:a6:b8 brd ff:ff:ff:ff:ff:ff
    inet 172.31.30.148/20 brd 172.31.31.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::d8:c0ff:fe66:a6b8/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@nodeb:~$
```

Now add the route on *nodea* (**be sure to substitute the correct external IP for your nodeb system**). Remember we are creating a route on *nodea* that lets us direct traffic to the docker0 bridge on *nodeb*, so it will be in this pattern: `sudo ip route add <docker0_IP_on_nodeb> via <external_IP_of_nodeb>`, but using the actual IPs like the example below:

```
ubuntu@nodea:~$ sudo ip route add 172.18.0.0/16 via 172.31.30.148

ubuntu@nodea:~$
```

> N.B. if you make a mistake you can delete the route by replacing the "add" argument with "delete"

```
ubuntu@nodea:~$ ip route

default via 172.31.0.1 dev eth0
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1 linkdown
172.18.0.0/16 via 172.31.9.145 dev eth0
172.31.0.0/20 dev eth0  proto kernel  scope link  src 172.31.7.235
ubuntu@nodea:~$
```

Perfect, now try to curl the nginx container from nodea:

```
ubuntu@nodea:~$ curl -I 172.18.0.2

curl: (7) Failed to connect to 172.18.0.2 port 80: Connection timed out
ubuntu@nodea:~$
```

No luck. Let's see if nodeb is reachable:

```
ubuntu@nodea:~$ ping -c 2 nodeb

PING nodeb (172.31.30.148) 56(84) bytes of data.
64 bytes from nodeb (172.31.30.148): icmp_seq=1 ttl=64 time=0.395 ms
64 bytes from nodeb (172.31.30.148): icmp_seq=2 ttl=64 time=0.408 ms

--- nodeb ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.395/0.401/0.408/0.021 ms
ubuntu@nodea:~$
```

So we can reach nodeb.

Now lets try to reach docker0 on nodeb:

```
ubuntu@nodea:~$ ping -c 2 172.18.0.1

PING 172.18.0.1 (172.18.0.1) 56(84) bytes of data.
64 bytes from 172.18.0.1: icmp_seq=1 ttl=64 time=0.389 ms
64 bytes from 172.18.0.1: icmp_seq=2 ttl=64 time=0.417 ms

--- 172.18.0.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.389/0.403/0.417/0.014 ms
ubuntu@nodea:~$
```

Also good!

> If the above ping fails, have your instructor check your EC2 instance "Networking->Source/Dest check", to be sure that it is disabled. Otherwise EC2

> VPCs will not allow traffic to be sent to a node using an alternate IP.

Let's try to reach the nginx container:

```
ubuntu@nodea:~$ ping -c 1 172.18.0.2

PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.

--- 172.18.0.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms
ubuntu@nodea:~$
```

No good. What could stop our packets from getting forwarded?

Change to a terminal on nodeb and display the IP Filter table:

```
ubuntu@nodeb:~$ sudo iptables -L -vn -t filter

Chain INPUT (policy ACCEPT 388 packets, 195K bytes)
 pkts bytes target     prot opt in     out     source               destination
31755   91M KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy DROP 5 packets, 324 bytes)
 pkts bytes target     prot opt in     out     source               destination
    5   324 DOCKER-USER  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    5   324 DOCKER-ISOLATION  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0            ctstate
RELATED,ESTABLISHED
    5   324 DOCKER     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 docker0  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT 374 packets, 60038 bytes)
 pkts bytes target     prot opt in     out     source               destination
28000 2713K KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination
```

```
Chain DOCKER-ISOLATION (1 references)
 pkts bytes target     prot opt in     out     source               destination
    5   324 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER-USER (1 references)
 pkts bytes target     prot opt in     out     source               destination
    5   324 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain KUBE-FIREWALL (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes firewall
for dropping marked packets */ mark match 0x8000/0x8000
ubuntu@nodeb:~$
```

A ha! The FORWARD chain is dropping packets. The default FORWARD policy is "DROP" and in the example above, 5 packets have been dropped. This is because there is no rule to ACCEPT packets headed to 172.18, so if the packet has to be FORWARDED it will be DROPped instead.

2.c. Create a rule in the filter table FORWARD chain that allows traffic to docker0

Let's add a rule on *nodeb* that ACCEPTs traffic from eth0 headed to 172.18:

```
ubuntu@nodeb:~$ sudo iptables -A FORWARD -i eth0 -d 172.18.0.0/16 -j ACCEPT

ubuntu@nodeb:~$
```

```
ubuntu@nodeb:~$ sudo iptables -L -vn -t filter

Chain INPUT (policy ACCEPT 10 packets, 5751 bytes)
 pkts bytes target     prot opt in     out     source               destination
31860   91M KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    5   324 DOCKER-USER  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    5   324 DOCKER-ISOLATION  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0            ctstate
RELATED,ESTABLISHED
    5   324 DOCKER     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  eth0  *       0.0.0.0/0            172.18.0.0/16

Chain OUTPUT (policy ACCEPT 9 packets, 1757 bytes)
 pkts bytes target     prot opt in     out     source               destination
28111 2730K KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination

Chain DOCKER-ISOLATION (1 references)
 pkts bytes target     prot opt in     out     source               destination
    5   324 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER-USER (1 references)
 pkts bytes target     prot opt in     out     source               destination
    5   324 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain KUBE-FIREWALL (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes firewall
for dropping marked packets */ mark match 0x8000/0x8000

ubuntu@nodeb:~$
```

Looks good. Now return to nodea and retry your ping and curl of the nginx container on nodeb:

```
ubuntu@nodea:~$ ping -c 2 172.18.0.2

PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
64 bytes from 172.18.0.2: icmp_seq=1 ttl=63 time=0.416 ms
64 bytes from 172.18.0.2: icmp_seq=2 ttl=63 time=0.404 ms

--- 172.18.0.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.404/0.410/0.416/0.006 ms
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ curl -I 172.18.0.2

HTTP/1.1 200 OK
Server: nginx/1.15.10
Date: Sun, 31 Mar 2019 00:57:00 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 26 Mar 2019 14:04:38 GMT
Connection: keep-alive
ETag: "5c9a3176-264"
Accept-Ranges: bytes
ubuntu@nodea:~$
```

Magic!

Return to nodeb and redisplay the filter table:

```
ubuntu@nodeb:~$ sudo iptables -L -vn -t filter

Chain INPUT (policy ACCEPT 23 packets, 1344 bytes)
 pkts bytes target     prot opt in     out     source               destination
23866  210M KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
   16  1354 DOCKER-USER  all  --  *      *       0.0.0.0/0            0.0.0.0/0
   16  1354 DOCKER-ISOLATION-STAGE-1  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    6   419 ACCEPT     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0            ctstate
RELATED,ESTABLISHED
    4   312 DOCKER     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
    6   623 ACCEPT     all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 docker0  0.0.0.0/0            0.0.0.0/0
    2   144 ACCEPT     all  --  eth0   *       0.0.0.0/0            172.18.0.0/16

Chain OUTPUT (policy ACCEPT 32 packets, 5988 bytes)
 pkts bytes target     prot opt in     out     source               destination
29007 4263K KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination
```

```
Chain DOCKER-ISOLATION-STAGE-1 (1 references)
 pkts bytes target     prot opt in     out     source               destination
    6   623 DOCKER-ISOLATION-STAGE-2  all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
   16  1354 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER-ISOLATION-STAGE-2 (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
    6   623 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER-USER (1 references)
 pkts bytes target     prot opt in     out     source               destination
   16  1354 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain KUBE-FIREWALL (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes firewall
for dropping marked packets */ mark match 0x8000/0x8000
ubuntu@nodeb:~$
```

Notice that our new rule has ACCEPTed packets (shown in the pkts column) allowing connectivity from nodea to docker0 on nodeb:

```
Chain FORWARD (policy DROP 0 packets, 0 bytes)
pkts   bytes target     prot opt in     out     source          destination

...

    2  144   ACCEPT     all  --  eth0   *       0.0.0.0/0       172.18.0.0/16
```

3. Configuring a flat network on nodea

We are only 1/2 way done. While we have configured things so that nodea can reach nodeb, we need to make the same changes in reverse so that nodeb can reach containers on nodea.

3.a. Assign docker0 a unique subnet

On nodea display the subnet for docker0:

```
ubuntu@nodea:~$ ip a show docker0

3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:10:a3:5a:ce brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:10ff:fea3:5ace/64 scope link
       valid_lft forever preferred_lft forever
ubuntu@nodea:~$
```

The nodea docker0 bridge is using 172.17. No one else is using this so we can leave it as is.

3.b. Create a route on nodeb to docker0 on nodea

The docker0 bridge on nodea uses the 172.17 subnet so we need to create a route to this subnet on *nodeb* (**be sure to substitute the correct external IP for your nodea system** in this case):

```
ubuntu@nodeb:~$ sudo ip route add 172.17.0.0/16 via 172.31.28.198

ubuntu@nodeb:~$
```

```
ubuntu@nodeb:~$ ip route

default via 172.31.16.1 dev eth0
172.17.0.0/16 via 172.31.28.198 dev eth0
172.18.0.0/16 dev docker0  proto kernel  scope link  src 172.18.0.1
172.31.16.0/20 dev eth0  proto kernel  scope link  src 172.31.30.148
ubuntu@nodeb:~$
```

3.c. Create a rule in the filter table FORWARD chain that allows traffic to docker0 on nodea

Finally, add the iptables rule on *nodea* that allows inbound traffic to 172.17:

```
ubuntu@nodea:~$ sudo iptables -A FORWARD -i eth0 -d 172.17.0.0/16 -j ACCEPT
```

```
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ sudo iptables -L -vn -t filter

Chain INPUT (policy ACCEPT 165 packets, 50749 bytes)
 pkts bytes target     prot opt in     out     source               destination
 886K  258M KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER-USER  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    0     0 DOCKER-ISOLATION  all  --  *      *       0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0            ctstate
RELATED,ESTABLISHED
    0     0 DOCKER     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  eth0   *       0.0.0.0/0            172.17.0.0/16

Chain OUTPUT (policy ACCEPT 162 packets, 51001 bytes)
 pkts bytes target     prot opt in     out     source               destination
1458K 2666M KUBE-FIREWALL  all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination

Chain DOCKER-ISOLATION (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER-USER (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain KUBE-FIREWALL (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes firewall
for dropping marked packets */ mark match 0x8000/0x8000
ubuntu@nodea:~$
```

3.d. Test the configuration

Now lets test our network setup by running a second container on nodea and then see if we can curl the nginx container on nodeb.

On **nodea** run a busybox container, then ping the IP and retrieve the nginx root doc:

```
ubuntu@nodea:~$ docker container run -it busybox

/ # ping -c 1 172.18.0.2

PING 172.18.0.2 (172.18.0.2): 56 data bytes
64 bytes from 172.18.0.2: seq=0 ttl=62 time=0.699 ms

--- 172.18.0.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.699/0.699/0.699 ms

/ # wget -qO - 172.18.0.2

Connecting to 172.18.0.2 (172.18.0.2:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
```

```
<p><em>Thank you for using nginx.</em></p>
</body>
</html>

/ # exit

ubuntu@nodea:~$
```

Perfect. We have setup cluster networking the hard way!

4. Services revisited

Now that we have Pod to Pod networking functioning in our cluster we can return to our initial goal. Setting up and running Kubernetes services.

To begin, terminate all containers running under Docker on both nodes:

nodea:

```
ubuntu@nodea:~$ docker container rm $(docker container stop $(docker container ls -qa))

...
ubuntu@nodea:~$
```

nodeb:

```
ubuntu@nodeb:~$ docker container rm $(docker container stop $(docker container ls -qa))

...
ubuntu@nodeb:~$
```

Verify that the API server, etcd, kubelet, controller manager, and scheduler are all running on nodea (if not restart the missing services):

```
ubuntu@nodea:~$ ps -aefo comm

COMMAND
```

```
bash
 \_ sudo
     \_ kubelet
bash
 \_ etcd
bash
 \_ sudo
     \_ kube-apiserver
bash
 \_ ps
bash
 \_ kube-scheduler
bash
 \_ kube-controller
agetty
agetty
ubuntu@nodea:~$
```

On nodeb, restart the kubelet:

```
ubuntu@nodeb:~$ sudo $HOME/kube-bin/kubelet \
--kubeconfig=nodeb.conf \
--config=nodeb.yaml \
--allow-privileged=true \
--runtime-cgroups=/systemd/machine.slice \
--kubelet-cgroups=/systemd/machine.slice \
--pod-infra-container-image=k8s.gcr.io/pause:3.1

...

I0331 01:04:58.182736   12904 kubelet_node_status.go:114] Node nodeb was previously registered
I0331 01:04:58.182761   12904 kubelet_node_status.go:75] Successfully registered node nodeb
I0331 01:04:58.362296   12904 reconciler.go:154] Reconciler: start to sync state
```

Now lets recreate our original deployment on nodea:

```
ubuntu@nodea:~$ kubectl create -f testdep.yaml
```

```
deployment.apps/nginx-deployment created
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl get deploy,rs,po

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/nginx-deployment   2/2     2            2           9s

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.extensions/nginx-deployment-6dd86d77d   2         2         2       9s

NAME                                   READY   STATUS    RESTARTS   AGE
pod/nginx-deployment-6dd86d77d-fg2xh   1/1     Running   0          9s
pod/nginx-deployment-6dd86d77d-wntbf   1/1     Running   0          9s
ubuntu@nodea:~$
```

Next recreate the service for the deployment:

```
ubuntu@nodea:~$ kubectl create -f nsvc.yaml

service/nsvc created
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl get svc

NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
kubernetes   10.0.0.1      <none>        443/TCP    43m
nsvc         10.0.34.209   <none>        2000/TCP   15s
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ kubectl get endpoints nsvc

NAME   ENDPOINTS                       AGE
nsvc   172.17.0.2:80,172.18.0.2:80   58s
```

```
ubuntu@nodea:~$
```

Try pinging each pod.

```
ubuntu@nodea:~$ ping -c 1 172.17.0.2

PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.097 ms

--- 172.17.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.097/0.097/0.097/0.000 ms
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ ping -c 1 172.18.0.2

PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
64 bytes from 172.18.0.2: icmp_seq=1 ttl=63 time=0.488 ms

--- 172.18.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.488/0.488/0.488/0.000 ms
ubuntu@nodea:~$
```

Now try to ping the pods from nodeb:

```
ubuntu@nodeb:~$ ping -c 1 172.17.0.2

PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.
64 bytes from 172.17.0.2: icmp_seq=1 ttl=63 time=0.416 ms

--- 172.17.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.416/0.416/0.416/0.000 ms
ubuntu@nodeb:~$
```

```
ubuntu@nodeb:~$ ping -c 1 172.18.0.2

PING 172.18.0.2 (172.18.0.2) 56(84) bytes of data.
64 bytes from 172.18.0.2: icmp_seq=1 ttl=64 time=0.045 ms

--- 172.18.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.045/0.045/0.045/0.000 ms
ubuntu@nodeb:~$
```

Excellent, we can reach both pods from anywhere in the cluster.

Now let's try to reach the pod using the nginx port (80):

```
ubuntu@nodea:~$ curl -I 172.17.0.2

HTTP/1.1 200 OK
Server: nginx/1.7.9
Date: Sun, 31 Mar 2019 01:07:58 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 23 Dec 2014 16:25:09 GMT
Connection: keep-alive
ETag: "54999765-264"
Accept-Ranges: bytes
ubuntu@nodea:~$
```

Good! Now try the Service VIP and port:

```
ubuntu@nodea:~$ curl 10.0.34.209:2000

curl: (7) Failed to connect to 10.0.44.224 port 2000: Connection refused
ubuntu@nodea:~$
```

What now!? We are missing the final piece of the service equation, kube-proxy. Remember, the API server simply records your wishes in etcd. It is up to other Kubernetes components to perform operations on the cluster that make those wishes real. In the case of services, it is the kube-proxy, which must run on every node, that creates the forwarding rules that bring service VIPs and Ports to life.

Display the NAT table rules on nodea:

```
ubuntu@nodea:~$ sudo iptables -L -vn -t nat

Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
  507 30496 DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type
LOCAL

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
  129  7740 DOCKER     all  --  *      *       0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type
LOCAL

Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
  936 70031 KUBE-POSTROUTING  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes
postrouting rules */
    2   144 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0

Chain KUBE-MARK-DROP (0 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MARK       all  --  *      *       0.0.0.0/0            0.0.0.0/0            MARK or 0x8000

Chain KUBE-MARK-MASQ (0 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MARK       all  --  *      *       0.0.0.0/0            0.0.0.0/0            MARK or 0x4000

Chain KUBE-POSTROUTING (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes service
traffic requiring SNAT */ mark match 0x4000/0x4000
ubuntu@nodea:~$
```

Note that there are no rules referencing our virtual IP, 10.0.34.209.

5. Setup kube-proxy

kube-proxy has a dependency on conntrack in order to interface with the kernel; the conntrack(8) the manual page has a solid description:

> conntrack provides a full featured userspace interface to the netfilter connection tracking system that is intended to replace the old /proc/net/ip_conntrack interface. This tool can be used to search, list, inspect and maintain the connection tracking subsystem of the Linux kernel. Using conntrack , you can dump a list of all (or a filtered selection of) currently tracked connections, delete connections from the state table, and even add new ones.

Installing the conntrack executable on workers is required because we are using kube-proxy as an executable; the containerized version includes all of its dependencies. Since we are also using our master as a worker, install conntrack on nodea:

```
ubuntu@nodea:~$ sudo apt-get install -y conntrack

...
```

Since Kubernetes 1.7, the kube-proxy component has been converted to use a configuration file. The `--write-config-to` flag has been provided to allow users to write the default kube-proxy configuration settings to a file.

Now **in a new tab or terminal** run kube-proxy on nodea, writing the default config to the file kube-dns-config:

```
ubuntu@nodea:~$ sudo $HOME/k8s/_output/bin/kube-proxy --write-config-to=kube-proxy-config

I0905 23:36:49.038759   20347 server.go:267] Wrote configuration to: kube-proxy-config
F0905 23:36:49.038804   20347 server.go:361] <nil>
ubuntu@nodea:~$
```

```
ubuntu@nodea:~$ cat kube-proxy-config

apiVersion: kubeproxy.config.k8s.io/v1alpha1
bindAddress: 0.0.0.0
```

```
clientConnection:
  acceptContentTypes: ""
  burst: 10
  contentType: application/vnd.kubernetes.protobuf
  kubeconfig: ""
  qps: 5
clusterCIDR: ""
configSyncPeriod: 15m0s
conntrack:
  max: 0
  maxPerCore: 32768
  min: 131072
  tcpCloseWaitTimeout: 1h0m0s
  tcpEstablishedTimeout: 24h0m0s
enableProfiling: false
healthzBindAddress: 0.0.0.0:10256
hostnameOverride: ""
iptables:
  masqueradeAll: false
  masqueradeBit: 14
  minSyncPeriod: 0s
  syncPeriod: 30s
ipvs:
  excludeCIDRs: null
  minSyncPeriod: 0s
  scheduler: ""
  syncPeriod: 30s
kind: KubeProxyConfiguration
metricsBindAddress: 127.0.0.1:10249
mode: ""
nodePortAddresses: null
oomScoreAdj: -999
portRange: ""
resourceContainer: /kube-proxy
udpIdleTimeout: 250ms
winkernel:
  enableDSR: false
  networkName: ""
  sourceVip: ""
ubuntu@nodea:~$
```

Note the `kubeconfig: ""` and `clusterCIDR: ""` configs; in previous versions of K8s, we used the `--kubeconfig` to tell kube-proxy about our cluster. Now we will put the path to the kubeconfig file in kube-proxy's config along with the value for the `--service-cluster-ip-range` flag we gave to the api-server.

```
ubuntu@nodea:~$ sudo vim kube-proxy-config
ubuntu@nodea:~$ head kube-proxy-config

apiVersion: kubeproxy.config.k8s.io/v1alpha1
bindAddress: 0.0.0.0
clientConnection:
  acceptContentTypes: ""
  burst: 10
  contentType: application/vnd.kubernetes.protobuf
  kubeconfig: "nodea.conf"
  qps: 5
clusterCIDR: "10.0.0.0/16"
configSyncPeriod: 15m0s
ubuntu@nodea:~$
```

Now we can run kube-proxy **in a new terminal or tab** on nodea:

```
ubuntu@nodea:~$ sudo $HOME/k8s/_output/bin/kube-proxy --config=kube-proxy-config

W0331 01:14:02.387940   24682 server_others.go:295] Flag proxy-mode="" unknown, assuming iptables proxy
I0331 01:14:02.390790   24682 server_others.go:148] Using iptables Proxier.
I0331 01:14:02.390965   24682 server_others.go:178] Tearing down inactive rules.
I0331 01:14:02.399987   24682 server.go:555] Version: v1.14.0
I0331 01:14:02.416856   24682 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_max' to 131072
I0331 01:14:02.416907   24682 conntrack.go:52] Setting nf_conntrack_max to 131072
I0331 01:14:02.416981   24682 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_tcp_timeout_established' to
86400
I0331 01:14:02.417026   24682 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_tcp_timeout_close_wait' to
3600
I0331 01:14:02.417542   24682 config.go:202] Starting service config controller
I0331 01:14:02.417666   24682 controller_utils.go:1027] Waiting for caches to sync for service config controller
I0331 01:14:02.417977   24682 config.go:102] Starting endpoints config controller
I0331 01:14:02.418058   24682 controller_utils.go:1027] Waiting for caches to sync for endpoints config controller
I0331 01:14:02.518170   24682 controller_utils.go:1034] Caches are synced for endpoints config controller
I0331 01:14:02.518171   24682 controller_utils.go:1034] Caches are synced for service config controller
```

Rerun the iptables dump on the NAT table on nodea:

```
ubuntu@nodea:~$ sudo iptables -L -nv -t nat

Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    2   434 KUBE-SERVICES  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes service
portals */
   19  1164 DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type
LOCAL

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    1    76 KUBE-SERVICES  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes service
portals */
  312 18720 DOCKER     all  --  *      *       0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type
LOCAL

Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    1    76 KUBE-POSTROUTING  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes
postrouting rules */
    2   144 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0

Chain KUBE-MARK-DROP (0 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MARK       all  --  *      *       0.0.0.0/0            0.0.0.0/0            MARK or 0x8000

Chain KUBE-MARK-MASQ (5 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MARK       all  --  *      *       0.0.0.0/0            0.0.0.0/0            MARK or 0x4000

Chain KUBE-NODEPORTS (1 references)
 pkts bytes target     prot opt in     out     source               destination
```

```
Chain KUBE-POSTROUTING (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes service
traffic requiring SNAT */ mark match 0x4000/0x4000

Chain KUBE-SEP-2YXAMM7IVAYA72O7 (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-MARK-MASQ  all  --  *      *       172.18.0.2           0.0.0.0/0            /* default/nsvc: */
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            /* default/nsvc: */ tcp
to:172.18.0.2:80

Chain KUBE-SEP-7XLM5FY5OZDUTZPQ (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-MARK-MASQ  all  --  *      *       192.168.225.193      0.0.0.0/0            /*
default/kubernetes:https */
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            /*
default/kubernetes:https */ recent: SET name: KUBE-SEP-7XLM5FY5OZDUTZPQ side: source mask: 255.255.255.255 tcp
to:192.168.225.193:6443

Chain KUBE-SEP-XDPNDAH2CYMNR5MR (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-MARK-MASQ  all  --  *      *       172.17.0.2           0.0.0.0/0            /* default/nsvc: */
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            /* default/nsvc: */ tcp
to:172.17.0.2:80

Chain KUBE-SERVICES (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-MARK-MASQ  tcp  --  *      *      !10.0.0.0/16          10.0.0.1             /*
default/kubernetes:https cluster IP */ tcp dpt:443
    0     0 KUBE-SVC-NPX46M4PTMTKRN6Y  tcp  --  *      *       0.0.0.0/0            10.0.0.1             /*
default/kubernetes:https cluster IP */ tcp dpt:443
    0     0 KUBE-MARK-MASQ  tcp  --  *      *      !10.0.0.0/16          10.0.34.209          /* default/nsvc:
cluster IP */ tcp dpt:2000
    0     0 KUBE-SVC-254CYKZ73JNCZ5NW  tcp  --  *      *       0.0.0.0/0            10.0.34.209          /*
default/nsvc: cluster IP */ tcp dpt:2000
    0     0 KUBE-NODEPORTS  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes
service nodeports; NOTE: this must be the last rule in this chain */ ADDRTYPE match dst-type LOCAL

Chain KUBE-SVC-254CYKZ73JNCZ5NW (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-SEP-XDPNDAH2CYMNR5MR  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /*
default/nsvc: */ statistic mode random probability 0.50000000000
    0     0 KUBE-SEP-2YXAMM7IVAYA72O7  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /*
```

```
default/nsvc: */

Chain KUBE-SVC-NPX46M4PTMTKRN6Y (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-SEP-7XLM5FY5OZDUTZPQ  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /*
default/kubernetes:https */ recent: CHECK seconds: 10800 reap name: KUBE-SEP-7XLM5FY5OZDUTZPQ side: source mask:
255.255.255.255
    0     0 KUBE-SEP-7XLM5FY5OZDUTZPQ  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /*
default/kubernetes:https */
ubuntu@nodea:~$
```

Wow, the kube-proxy has been busy! Search for the VIP of our service:

```
ubuntu@nodea:~$ sudo iptables -L -nv -t nat | grep 10.0.34.209

    0     0 KUBE-MARK-MASQ  tcp  --  *      *                     !10.0.0.0/16    10.0.34.209      /* default/nsvc:
cluster IP */ tcp dpt:2000
    0     0 KUBE-SVC-254CYKZ73JNCZ5NW  tcp  --  *      *       0.0.0.0/0       10.0.34.209      /* default/nsvc:
cluster IP */ tcp dpt:2000
ubuntu@nodea:~$
```

The proxy has created a rule to intercept all traffic heading to our service VIP on port 2000. Examine the chain created for our service (the 2nd line of output above that starts with "`KUBE-SVC-`"):

```
ubuntu@nodea:~$ sudo iptables -L -nv -t nat | grep -A4 'Chain KUBE-SVC-254CYKZ73JNCZ5NW'

Chain KUBE-SVC-254CYKZ73JNCZ5NW (1 references)
 pkts bytes target                     prot opt in     out     source               destination
    0     0 KUBE-SEP-XDPNDAH2CYMNR5MR  all  --  *      *       0.0.0.0/0            0.0.0.0/0            statistic mode
random probability 0.50000000000
    0     0 KUBE-SEP-2YXAMM7IVAYA72O7  all  --  *      *       0.0.0.0/0            0.0.0.0/0
ubuntu@nodea:~$
```

The iptables chain listed applies a statistic probability to one of the two implementation pods backing our service. Display the chain for the first target pod:

```
ubuntu@nodea:~$ sudo iptables -L -nv -t nat | grep -A4 'Chain KUBE-SEP-XDPNDAH2CYMNR5MR'
```

```
Chain KUBE-SEP-XDPNDAH2CYMNR5MR (1 references)
 pkts bytes target     prot opt  in     out     source               destination
    0     0 KUBE-MARK-MASQ  all  --  *      *   172.17.0.2           0.0.0.0/0
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0       tcp to:172.17.0.2:80
ubuntu@nodea:~$
```

The DNAT rule takes all traffic and sends it to 172.17.0.2:80. Perfect!

Try curling your service using the service VIP and port:

```
ubuntu@nodea:~$ curl -I 10.0.34.209:2000

HTTP/1.1 200 OK
Server: nginx/1.7.9
Date: Sun, 31 Mar 2019 01:22:33 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 23 Dec 2014 16:25:09 GMT
Connection: keep-alive
ETag: "54999765-264"
Accept-Ranges: bytes
ubuntu@nodea:~$
```

Nice!

6. Completing the nodeb configuration

Lastly, to complete our compliment of cluster services let's start the kube-proxy on nodeb and test the service.

To begin, copy the `kube-proxy-config` from nodea to nodeb or run kube-proxy with the `--write-config-to` flag to generate the default config and edit it. In either case, remember to replace the *nodea.kubeconfig* so that it uses the nodeb.kubeconfig!

```
ubuntu@nodeb:~$ sudo apt-get install -y conntrack

ubuntu@nodeb:~$ sudo $HOME/kube-bin/kube-proxy --write-config-to=kube-proxy-config

I0905 23:39:45.167948   16518 server.go:267] Wrote configuration to: kube-proxy-config
```

```
F0905 23:39:45.168001   16518 server.go:361] <nil>

ubuntu@nodeb:~$ sudo vim kube-proxy-config
ubuntu@nodeb:~$ head kube-proxy-config

apiVersion: kubeproxy.config.k8s.io/v1alpha1
bindAddress: 0.0.0.0
clientConnection:
  acceptContentTypes: ""
  burst: 10
  contentType: application/vnd.kubernetes.protobuf
  kubeconfig: "nodeb.conf"
  qps: 5
clusterCIDR: "10.0.0.0/16"
configSyncPeriod: 15m0s
ubuntu@nodeb:~$
```

When the config file is ready, run the kube-proxy **in a new terminal or tab** on nodeb:

```
ubuntu@nodeb:~$ sudo $HOME/kube-bin/kube-proxy --config=kube-proxy-config

W0331 01:25:28.819722   13796 server_others.go:295] Flag proxy-mode="" unknown, assuming iptables proxy
I0331 01:25:28.822698   13796 server_others.go:148] Using iptables Proxier.
I0331 01:25:28.822856   13796 server_others.go:178] Tearing down inactive rules.
I0331 01:25:28.831733   13796 server.go:555] Version: v1.14.0
I0331 01:25:28.843330   13796 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_max' to 131072
I0331 01:25:28.843379   13796 conntrack.go:52] Setting nf_conntrack_max to 131072
I0331 01:25:28.849086   13796 conntrack.go:83] Setting conntrack hashsize to 32768
I0331 01:25:28.849400   13796 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_tcp_timeout_established' to
86400
I0331 01:25:28.849599   13796 conntrack.go:100] Set sysctl 'net/netfilter/nf_conntrack_tcp_timeout_close_wait' to
3600
I0331 01:25:28.850091   13796 config.go:202] Starting service config controller
I0331 01:25:28.850176   13796 controller_utils.go:1027] Waiting for caches to sync for service config controller
I0331 01:25:28.850236   13796 config.go:102] Starting endpoints config controller
I0331 01:25:28.850282   13796 controller_utils.go:1027] Waiting for caches to sync for endpoints config controller
I0331 01:25:28.950523   13796 controller_utils.go:1034] Caches are synced for endpoints config controller
I0331 01:25:28.950542   13796 controller_utils.go:1034] Caches are synced for service config controller
```

Now try curling the nsvc service from nodeb:

```
ubuntu@nodeb:~$ curl -I 10.0.34.209:2000

HTTP/1.1 200 OK
Server: nginx/1.7.9
Date: Sun, 31 Mar 2019 01:25:51 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 23 Dec 2014 16:25:09 GMT
Connection: keep-alive
ETag: "54999765-264"
Accept-Ranges: bytes
ubuntu@nodeb:~$
```

Mega.

You have now setup all of the core parts of a Kubernetes cluster the hard way (and hopefully learned something and had some fun in the process).

Remove your on-cluster resources (services, deployments, etc.)

Congratulations you have successfully completed the lab!