RX-M Cloud Native Consulting

Kubernetes

Lab 5 – Services & Network Policy

In the first part of this lab we will explore the nature of Kubernetes Services and how to work with them.

Kubernetes pods are mortal. They are born and they die, and they are not resurrected. ReplicaSets create and destroy Pods dynamically (e.g. when scaling up or down or when doing rolling updates). While each pod gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem; if some set of backend Pods provides functionality to other frontend Pods inside the Kubernetes cluster, how do the frontends find out and keep track of the backends?

Services

A Kubernetes Service is an abstraction which defines a logical set of pods and a policy by which to access them. The set of pods targeted by a Service is usually determined by a label selector. As an example, consider an image-processing backend which is running with 3 replicas. Those replicas are fungible, frontends do not care which backend they use. While the actual pods that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of backends themselves. The Service abstraction enables this decoupling.

For Kubernetes-native applications, Kubernetes offers a simple Endpoints API that is updated whenever the set of pods in a Service changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to Services which redirects to the backend pods.

In Kubernetes, a Service is a REST object, similar to a Pod. Like all of the REST objects, a Service definition can be POSTed to the kube-apiserver to create a new instance.

1. A Simple Service

Let’s begin by creating a simple service using a service config file. Before you begin delete any services (except the kubernetes service), resource controllers and pods you may have running.

Now create a simple service called “testweb” with its own *ClusterIP* passing traffic on port 80 and configure the service to use the selector “run=testweb”.

Something like this:

```
ubuntu@ip-10-0-2-200:~/jobs$ cd ~

ubuntu@ip-10-0-2-200:~$ mkdir svc

ubuntu@ip-10-0-2-200:~$ cd svc

ubuntu@ip-10-0-2-200:~/svc$ vim svc.yaml
```

```
ubuntu@ip-10-0-2-200:~/svc$ cat svc.yaml

apiVersion: v1
kind: Service
metadata:
  name: testweb
  labels:
    name: testweb
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    run: testweb
ubuntu@ip-10-0-2-200:~/svc$
```

Now create the service:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl create -f svc.yaml

service/testweb created
ubuntu@ip-10-0-2-200:~/svc$
```

List the services in your namespace:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl get services

NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   12h
testweb      ClusterIP   10.97.132.225   <none>        80/TCP    13s

ubuntu@ip-10-0-2-200:~/svc$ SVC=$(kubectl get service testweb -o template --
template={{.spec.clusterIP}}) && echo $SVC

10.97.132.225
ubuntu@ip-10-0-2-200:~/svc$
```

Great we have a service running. Now what?

Try to `curl` your service.

```
ubuntu@ip-10-0-2-200:~/svc$ curl $SVC

curl: (7) Failed to connect to 10.97.132.225 port 80: Connection refused
ubuntu@ip-10-0-2-200:~/svc$
```

We have created a service and it has an IP but the IP is virtual and there’s no pod(s) for the proxy to send the traffic to. Services truly can outlive their implementations.

To fix this lack of implementation we can create a pod with a label that matches the service selector.

Create a simple nginx pod to support your service and run it.

```
ubuntu@ip-10-0-2-200:~/svc$ vim web.yaml

ubuntu@ip-10-0-2-200:~/svc$ cat web.yaml

apiVersion: v1
kind: Pod
metadata:
  name: bigwebstuff
  labels:
    run: testweb
spec:
  containers:
  - name: web-container
    image: nginx
    ports:
    - containerPort: 80
ubuntu@ip-10-0-2-200:~/svc$
```

Now run the pod:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl create -f web.yaml

pod/bigwebstuff created
ubuntu@ip-10-0-2-200:~/svc$
```

With the pod up, retry curling the service IP:

```
ubuntu@ip-10-0-2-200:~/svc$ curl -I $SVC

HTTP/1.1 200 OK
Server: nginx/1.13.10
Date: Fri, 30 Mar 2018 00:19:19 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 20 Mar 2018 10:03:54 GMT
Connection: keep-alive
ETag: "5ab0dc8a-264"
Accept-Ranges: bytes
ubuntu@ip-10-0-2-200:~/svc$
```

Describe your service to verify the wiring between the ClusterIP and the container in the pod.

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl describe service testweb

Name:                   testweb
Namespace:              default
Labels:                 name=testweb
Annotations:            <none>
```

```
Selector:           run=testweb
Type:               ClusterIP
IP:                 10.97.132.225
Port:               <unset>  80/TCP
TargetPort:         80/TCP
Endpoints:          10.32.0.4:80
Session Affinity:   None
Events:             <none>
ubuntu@ip-10-0-2-200:~/svc$
```

So as you can see our nginx container must be listening on 10.32.0.4.

Verify this through `kubectl` and `curl`:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl get pod bigwebstuff -o json | jq -r '.status
| .podIP, .hostIP'

10.32.0.4
10.0.2.200

ubuntu@ip-10-0-2-200:~/svc$ IP=$(kubectl get pod bigwebstuff -o json | jq -r
'.status | .podIP') && echo $IP

10.32.0.4
ubuntu@ip-10-0-2-200:~/svc$
```

Try curling the pod:

```
ubuntu@ip-10-0-2-200:~/svc$ curl -I $IP

HTTP/1.1 200 OK
Server: nginx/1.13.10
Date: Fri, 30 Mar 2018 00:20:19 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 20 Mar 2018 10:03:54 GMT
Connection: keep-alive
ETag: "5ab0dc8a-264"
Accept-Ranges: bytes
ubuntu@ip-10-0-2-200:~/svc$
```

All we needed to do to enable our service was to create a pod with the right label. Note that if our pod container dies, no one will restart it as things now stand, but our service will carry on.

2. Add a Resource Controller to Your Service

To improve the robustness of our service implementation we can switch from a pod to a resource controller. Change your config to instantiate a deployment which creates 3 replicas and with a template just like the pod we launched in the last step.

```
ubuntu@ip-10-0-2-200:~/svc$ vim webd.yaml
```

```
ubuntu@ip-10-0-2-200:~/svc$ cat webd.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: bigwebstuff
  labels:
    name: bigwebstuff
spec:
  replicas: 3
  selector:
    matchLabels:
      run: testweb
  template:
    metadata:
      labels:
        run: testweb
    spec:
      containers:
      - name: podweb
        image: nginx:latest
        ports:
        - containerPort: 80
ubuntu@ip-10-0-2-200:~/svc$
```

Now create the deployment:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl create -f webd.yaml

deployment.apps/bigwebstuff created
ubuntu@ip-10-0-2-200:~/svc$
```

Now let’s see what happened to our pods and our service.

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl describe service testweb

Name:              testweb
Namespace:         default
Labels:            name=testweb
Annotations:       <none>
Selector:          run=testweb
Type:              ClusterIP
IP:                10.97.132.225
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.32.0.4:80,10.32.0.5:80,10.32.0.6:80 + 1 more...
Session Affinity:  None
Events:            <none>
ubuntu@ip-10-0-2-200:~/svc$
```

List the running Pods:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl get pods

NAME                          READY   STATUS    RESTARTS   AGE
bigwebstuff                   1/1     Running   0          115s
bigwebstuff-b5c589b5f-f822r   1/1     Running   0          23s
bigwebstuff-b5c589b5f-fk9bz   1/1     Running   0          23s
bigwebstuff-b5c589b5f-gh8bs   1/1     Running   0          23s
ubuntu@ip-10-0-2-200:~/svc$
```

- What happened to our old pod?
- How many pods did the RS create?
- What does the age output in the get pods command tell you?
- What are the pods names of a different form?

Service selector and Replica Set selector behavior differ in subtle ways. You will notice the Service has 4 pods, while the Replica Set has 3.

Delete all resources before moving on.

Network Policy

A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints. NetworkPolicy resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods. Network policies are implemented by a CNI network plugin, so you must use a CNI networking solution which supports NetworkPolicy.

By default, pods are non-isolated; they accept traffic from any source. Pods become isolated by having a NetworkPolicy that selects them. Adding a NetworkPolicy to a namespace selecting a particular pod, causes that pod to become "isolated", rejecting any connections that are not explicitly allowed by a NetworkPolicy. Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.

3. Run a service and access it from a client

To begin let's run a pod and try connecting to it from another pod. The rxmllc/hostinfo image listens on port 9898 and responds to requests with the pod hostname. Run a pod with a hostinfo pod having the label "app=hi":

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl run hi --generator=run-pod/v1 --image
rxmllc/hostinfo --port 80 -l app=hi

pod/hi created
ubuntu@ip-10-0-2-200:~/svc$
```

Now run a client busybox pod and try curing the hostinfo pod from within the busybox pod:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl run client --generator=run-pod/v1 --image
busybox --command -- tail -f /dev/null

pod/client created
```

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl get pod

NAME      READY   STATUS    RESTARTS   AGE
client    1/1     Running   0          12s
hi        1/1     Running   0          34s

ubuntu@ip-10-0-2-200:~/svc$ kubectl get pod hi -o yaml | grep -i ip

  nodeName: ip-10-0-2-200
  hostIP: 10.0.2.200
  podIP: 10.32.0.4

ubuntu@ip-10-0-2-200:~/svc$ kubectl exec -it client sh

/ # wget -qO - http://10.32.0.4:9898

hi 10.32.0.4

/ # exit

ubuntu@ip-10-0-2-200:~/svc$
```

So with no network policy we can access one pod from another.

4. Create a blocking network policy

For our first network policy we'll create a policy that denies all inbound connections to pods in the default namespace. Create the following policy:

```
ubuntu@ip-10-0-2-200:~/svc$ vim np.yaml

ubuntu@ip-10-0-2-200:~/svc$ cat np.yaml
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl create -f np.yaml

networkpolicy.networking.k8s.io/default-deny created

ubuntu@ip-10-0-2-200:~/svc$ kubectl get networkpolicy

NAME           POD-SELECTOR   AGE
```

```
default-deny   <none>         1m
ubuntu@ip-10-0-2-200:~/svc$
```

This policy selects all pods (`{}`) and has no ingress policies (`Ingress`) for them. By creating any network policy however, we automatically isolate all pods.

Retry connecting to the hostinfo pod:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl exec -it client -- wget -qO -
http://10.32.0.4:9898

^C

command terminated with exit code 130
ubuntu@ip-10-0-2-200:~/svc$
```

The command should hang and you will have to press CTRL+C to cancel it. The presence of a network policy shuts down our ability to reach the other pod.

5. Create a permissive network policy

To enable our busybox pod to access our hostinfo pod we will need to create a network policy that selects the hostinfo pods and allows ingress from the busybox pod. The hostinfo pod has the label "app=hi" so we can use that to select the target pod and our busybox pod has the label "run=client" so we'll use that to identify the ingress pods allowed (anytime you create a pod with the `kubectl run` command, all of the resources created get assigned the label `run= <NAME>`, where NAME is the name of the pod).

Create the new policy:

```
ubuntu@ip-10-0-2-200:~/svc$ vim np-hi.yaml

ubuntu@ip-10-0-2-200:~/svc$ cat np-hi.yaml
```

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: access-hostinfo
spec:
  podSelector:
    matchLabels:
      app: hi
  ingress:
  - from:
    - podSelector:
        matchLabels:
          run: "client"
```

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl apply -f np-hi.yaml

networkpolicy.networking.k8s.io/access-hostinfo created

ubuntu@ip-10-0-2-200:~/svc$ kubectl get networkpolicy

NAME               POD-SELECTOR   AGE
access-hostinfo    app=hi         12s
default-deny       <none>         9m
ubuntu@ip-10-0-2-200:~/svc$
```

Now retry access the hostinfo pod from teh busybox pod:

```
ubuntu@ip-10-0-2-200:~/svc$ kubectl exec -it client -- wget -qO -
http://10.32.0.4:9898

hi 10.32.0.4
ubuntu@ip-10-0-2-200:~/svc$
```

It works!

For more information on network policy take a look at the resource reference:

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.12/#networkpolicy-v1-networking-k8s-io

Congratulations you have completed the lab.