rx-m cloud native & ai training & consulting

Kubernetes

Services

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?

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 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 applications integrated with the Kubernetes control plane, Kubernetes offers a simple Endpoints API that is updated whenever the set of pods in a Service changes. For Kubernetes hosted applications, Kubernetes offers a virtual-IP-based façade which redirects connections to the backend pods. For applications outside of the Kubernetes cluster, Kubernetes offers a cluster wide port forwarding feature that provides a way for external traffic to enter the pod network and reach a service's pods.

In Kubernetes, a Service is just a JSON object in etcd, similar to a Pod. Like all of the "REST" objects, a Service definition can be POSTed to the kube-apiserver to create a new service instance. The service controller then acts on service specifications reserving Cluster IPs and Node Ports as needed. This in turn causes the KubeProxy agents and/or SDN implementations to take local action on each node (modifying iptables/ipvs/etc.).

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, pods or other resources 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 “`app=testweb`”.

Create the service using `kubectl create`:

```
~$ mkdir ~/svc && cd ~/svc

~/svc$ kubectl create svc clusterip testweb --tcp 80 --dry-run=client -o
yaml > svc.yaml
```

```
~/svc$
```

Examine the Service; you can remove the `status` and `creationTimestamp` keys from the svc.yaml you generated but leaving them in place will not be harmful:

```
~/svc$ cat svc.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: null
  labels:
    app: testweb
  name: testweb
spec:
  ports:
  - name: "80"
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: testweb
  type: ClusterIP
status:
  loadBalancer: {}
```

```
~/svc$
```

Now create the service:

```
~/svc$ kubectl apply -f svc.yaml

service/testweb created

~/svc$
```

> N.B. Your host, service and pod ip addresses will be different from those shown in the lab examples. Be sure to substitute the correct addresses from you lab system as you work through the lab!

List the services in your namespace:

```
~/svc$ kubectl get svc

NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   85m
testweb      ClusterIP   10.111.174.45   <none>        80/TCP    5s

~/svc$
```

Like other manifests, we can extract individual values from Services; extract the clusterIP value and register it as an environment variable:

```
~/svc$ SVC=$(kubectl get service testweb --template={{.spec.clusterIP}})
&& echo $SVC

10.111.174.45

~/svc$
```

Great we have a service running. Now what?

Let's try to use it. To make use of services we need to be in the pod network. Some SDN solutions provide an onramp (route) on the nodes to the pod network but the ClusterIP is virtual so it is often not available outside of running pods. Services are always available inside pods though so we'll run a pod to use as a test client.

Run a pod for testing the service, setting the `SVC` environment variable in the pod to the same variable on our host:

```
~/svc$ kubectl run -it testclient --image=busybox:1.27 --env SVC=$SVC

If you don't see a command prompt, try pressing enter.

/ #
```

Try to `wget` your service's Cluster IP from inside the pod:

```
/ # wget --timeout 5 -O - $SVC

Connecting to 10.105.23.143 (10.105.23.143:80)
wget: download timed out

/ #
```

> N.B. without the --timeout flag, wget will eventually return the message: "wget: can't connect to remote host (...): Connection refused"

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.

Exit your testclient pod:

```
/ # exit

Session ended, resume using 'kubectl attach testclient -c testclient -i -
t' command when the pod is running

~/svc$
```

As before, we can re-attach to the testclient pod with the command mentioned after we exited.

Dry-run the following pod:

```
~/svc$ kubectl run bigwebstuff --image nginx -l app=testweb --port 80 --
dry-run=client -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    app: testweb
  name: bigwebstuff
spec:
  containers:
  - image: nginx
    name: bigwebstuff
    ports:
    - containerPort: 80
    resources: {}
  dnsPolicy: ClusterFirst
  restartPolicy: Always
status: {}
```

```
~/svc$
```

Using the `-l app=testweb` argument, we label the pod with the same key-value pair the Service is using in its selector.

Now run the pod by removing the `--dry-run` and `-o yaml` arguments:

```
~/svc$ kubectl run bigwebstuff --image nginx -l app=testweb --port 80

pod/bigwebstuff created

~/svc$
```

With the pod up, retry the service IP from the test pod:

```
~/svc$ kubectl exec testclient -- wget --timeout 5 -O - $SVC

Connecting to 10.111.174.45 (10.111.174.45:80)
-                    100% |*******************************|   615
0:00:00 ETA

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
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>

~/svc$
```

On the host, describe your service to verify the wiring between the ClusterIP and the container in the pod:

```
~/svc$ kubectl describe service testweb
```

```
Name:               testweb
Namespace:          default
Labels:             app=testweb
Annotations:        <none>
Selector:           app=testweb
Type:               ClusterIP
IP Family Policy:   SingleStack
IP Families:        IPv4
IP:                 10.111.174.45
IPs:                10.111.174.45
Port:               80  80/TCP
TargetPort:         80/TCP
Endpoints:          10.32.0.5:80
Session Affinity:   None
Events:             <none>
```

```
~/svc$
```

So as you can see in the example, our nginx container must be listening on IP and Port combination listed next to `Endpoints`.

Use kubectl to display the pod and host IPs:

```
~/svc$ sudo apt install jq -y

...

~/svc$ kubectl get po bigwebstuff -o json | jq -r '.status | .podIP,
.hostIP'

10.32.0.5
172.31.4.161

~/svc$
```

Try hitting the pod by its pod IP from the test container:

```
~/svc$ kubectl exec testclient -- wget -O - 10.32.0.5    # Make sure to
use the pod IP from the command above

Connecting to 10.32.0.5 (10.32.0.5:80)
-                    100% |*******************************|   615
```

```
0:00:00 ETA

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
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>

~/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.

```
~/svc$ kubectl create deploy bigwebstuff --image nginx:latest --replicas 3
--port 80 --dry-run=client -o yaml > webd.yaml

~/svc$
```

Edit the manifest:

- Change the value of all `app:` keys in the spec to `testweb`:

```
~/svc$ nano webd.yaml && cat $_
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: testweb              # Change this
  name: bigwebstuff
spec:
  replicas: 3
  selector:
    matchLabels:
      app: testweb            # Change this
  strategy: {}
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: testweb          # Change this
    spec:
      containers:
      - image: nginx:latest
        name: nginx
        ports:
        - containerPort: 80
        resources: {}
status: {}
```

```
~/svc$
```

Now create the deployment:

```
~/svc$ kubectl apply -f webd.yaml

deployment.apps/bigwebstuff created

~/svc$
```

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

```
~/svc$ kubectl describe service testweb
```

```
Name:                   testweb
Namespace:              default
Labels:                 app=testweb
```

```
Annotations:       <none>
Selector:          app=testweb
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.111.174.45
IPs:               10.111.174.45
Port:              80  80/TCP
TargetPort:        80/TCP
Endpoints:         10.32.0.5:80,10.32.0.6:80,10.32.0.7:80 + 1 more...
Session Affinity:  None
Events:            <none>
```

```
~/svc$
```

List the running Pods:

```
~/svc$ kubectl get po

NAME                          READY   STATUS    RESTARTS       AGE
bigwebstuff                   1/1     Running   0              106s
bigwebstuff-cb98f5c6c-4tjqt   1/1     Running   0              19s
bigwebstuff-cb98f5c6c-blh7m   1/1     Running   0              19s
bigwebstuff-cb98f5c6c-vdcpr   1/1     Running   0              19s
testclient                    1/1     Running   1 (119s ago)   2m13s

~/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 with the suffixes from?

Service selector and ReplicaSet selector behavior differ in subtle ways. You will notice the Service routes traffic to 4 pods, while the ReplicaSet controls only 3 pods.

3. CHALLENGE - Service selectors

Without changing any of the running resources, create a new deployment that runs the `httpd` image with 2 replicas such that the service will send traffic to it as well.

Test your service to verify proper operation using curl in the test container.

> N.B. Apache httpd returns a different response than that returned by nginx

- Clean up the challenge resources once finished, and then clean up the rest

```
~/svc$ kubectl delete deploy/bigwebstuff po/bigwebstuff svc/testweb

deployment.apps "bigwebstuff" deleted
pod "bigwebstuff" deleted
service "testweb" deleted

~/svc$
```

4. Blue-Green Deployments

Blue-Green deployments are useful when you want to replace one version of a microservice with a new or different one, but you want to check that it's working correctly before you turn traffic from the old (blue) one to the new (green) one, or if you want the entire deployment to swap over at once instead of performing a rolling update.

Since we *do not* want to use the rolling update feature, we will use ReplicaSets instead of Deployments. First, make an RS to serve as your blue deployment:

```
~/svc$ nano blue.yaml && cat $_
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: blue-rs
  labels:
    demo: blue-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      demo: blue-rs
  template:
    metadata:
      labels:
        demo: blue-rs
    spec:
      containers:
      - name: blue-containers
        image: rxmllc/hostinfo:alpine
        ports:
        - containerPort: 9898
```

```
~/svc$ kubectl apply -f blue.yaml

replicaset.apps/blue-rs created
```

```
~/svc$
```

Now make a service which exposes pods on the cluster using one of the labels: `method=blue-green`:

```
~/svc$ nano bgsvc.yaml && cat $_
```

```
apiVersion: v1
kind: Service
metadata:
  name: blue-green-svc
spec:
  ports:
  - port: 80
    targetPort: 9898
  selector:
    method: blue-green
```

```
~/svc$ kubectl apply -f bgsvc.yaml

service/blue-green-svc created

~/svc$ kubectl get svc

NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
blue-green-svc   ClusterIP   10.103.212.79   <none>        80/TCP    2s
kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP   90m
testweb          ClusterIP   10.111.174.45   <none>        80/TCP    5m16s

~/svc$
```

List your service's endpoints:

```
~/svc$ kubectl get ep blue-green-svc

NAME             ENDPOINTS   AGE
blue-green-svc   <none>      6s

~/svc$
```

None of our pods currently fulfill our service's selector.

To add our blue pods to the service's load balancing we can imperatively update pod labels using `kubectl label`. This is why we use RSes! If we used a Deployment, a config change to the pods could trigger a rolling update--which we *do not* want; using RSes eliminates any potential for a rolling update to trigger since only the Deployment controller has the rolling update feature.

The following command finds all pods with the `demo=blue-rs` label and feeds their names to the `label` command, adding the service selector's label: "`method=blue-green`":

```
~/svc$ kubectl label $(kubectl get po -l demo=blue-rs -o name)
method=blue-green

pod/blue-rs-656j5 labeled
pod/blue-rs-ntn57 labeled
pod/blue-rs-z5zth labeled

~/svc$
```

List your service's endpoints; *there should be 3*:

```
~/svc$ kubectl get ep blue-green-svc

NAME             ENDPOINTS                                         AGE
blue-green-svc   10.32.0.10:9898,10.32.0.11:9898,10.32.0.9:9898   31s

~/svc$
```

Perfect! Store the service's clusterIP in an environment variable `BGSVC`:

```
~/svc$ BGSVC=$(kubectl get service blue-green-svc --template=
{{.spec.clusterIP}}) && echo $BGSVC

10.108.63.65

~/svc$
```

Run a pod for testing the service, setting the `BGSVC` environment variable in the pod to the same variable on our host:

```
~/svc$ kubectl run -it bgtest --image=busybox:1.27 --env BGSVC=$BGSVC

If you don't see a command prompt, try pressing enter.

/ #
```

Try to `wget` your service's Cluster IP from inside the pod:

```
/ # wget -qO - $BGSVC

blue-rs-z5zth 10.34.0.2

/ #
```

We can see from the hostname that one of our blue pods answered (as expected).

```
/ # exit

Session ended, resume using 'kubectl attach bgtest -c bgtest -i -t'
command when the pod is running

~/svc$
```

Now create the green RS, using the "`latest`" tag for the `rxmllc/hostinfo` image:

```
~/svc$ nano green.yaml && cat $_
```

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: green-rs
  labels:
    demo: green-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      demo: green-rs
  template:
    metadata:
      labels:
        demo: green-rs
    spec:
      containers:
      - name: green-containers
        image: rxmllc/hostinfo:latest
        ports:
        - containerPort: 9898
```

```
~/svc$ kubectl apply -f green.yaml

replicaset.apps/green-rs created
```

```
~/svc$ kubectl get po -l demo=green-rs

NAME             READY   STATUS    RESTARTS   AGE
green-rs-l2mgh   1/1     Running   0          12s
green-rs-ss4rc   1/1     Running   0          12s
green-rs-tl5rc   1/1     Running   0          12s

~/svc$
```

The new green pods are up and ready and there don't appear to be any errors.

List your service's endpoints; *there should still only be 3*:

```
~/svc$ kubectl get ep blue-green-svc

NAME             ENDPOINTS                                         AGE
blue-green-svc   10.32.0.10:9898,10.32.0.11:9898,10.32.0.9:9898   75s

~/svc$
```

Because Kubernetes has an eventual consistency model, we will need to add the green pods to the service while the blue pods are still present so that the endpoints list will never be empty. We will also want to wait *before* removing the blue pods to give time for kube-proxy to implement the new endpoints in iptables / ipvs tables. This means there will be a slight overlap when both blue and green pods answer as backends for our service.

To add our green pods to the service's load balancing we imperatively update pod labels using `kubectl label` once again:

```
~/svc$ kubectl label $(kubectl get po -l demo=green-rs -o name)
method=blue-green

pod/green-rs-l2mgh labeled
pod/green-rs-ss4rc labeled
pod/green-rs-tl5rc labeled

~/svc$
```

List your service's endpoints; *there should now be 6*:

```
~/svc$ kubectl get ep blue-green-svc

NAME             ENDPOINTS
AGE
blue-green-svc   10.32.0.10:9898,10.32.0.11:9898,10.32.0.13:9898 + 3
more...   92s
```

```
~/svc$
```

Now reattach to your test pod and `wget` the cluster IP again.

```
~/svc$ kubectl exec bgtest -- wget -qO - $BGSVC

green-rs-zzxlf 10.32.0.14

~/svc$
```

What happened?

Load balancing is random so in the above example we were load balanced to one of the blue pods (you may have gotten a green pod). Try the `wget` several times to view the load balancing in action:

```
~/svc$ kubectl exec bgtest -- wget -qO - $BGSVC

blue-rs-jjf2v 10.32.0.9

~/svc$ kubectl exec bgtest -- wget -qO - $BGSVC

green-rs-2mdg7 10.32.0.15

~/svc$
```

In its current state, we now have 6 pods serving as endpoints to the service, with an even split of probably of 50% of traffic going to each of the service. Each individual pod has a 16% probability of receiving the traffic from the service.

To remove the blue pods from the load balancing we can either delete the blue RS (this technique is also known as "highlander") or simply remove the service selector label from the blue pods. We will do the latter using the `label` command once more removing the `method` label key from the pods with the label `demo=blue-rs`:

```
~/svc$ kubectl label $(kubectl get po -l demo=blue-rs -o name) method-

pod/blue-rs-84d9r unlabeled
pod/blue-rs-j2lf6 unlabeled
pod/blue-rs-jjf2v unlabeled

~/svc$
```

List your service's endpoints; *there should be 3 again*:

```
~/svc$ kubectl get ep blue-green-svc

NAME             ENDPOINTS                                            AGE
blue-green-svc   10.32.0.13:9898,10.32.0.14:9898,10.32.0.15:9898   2m34s

~/svc$
```

You have successfully performed a blue-green deployment! 100% of all of the traffic the service is receiving is now being routed to your new (green) service.

5. Clean up

Delete the RSes, pods and service created in this lab using `kubectl delete`:

```
~/svc$ kubectl delete -f blue.yaml -f green.yaml -f bgsvc.yaml

replicaset.apps "blue-rs" deleted
replicaset.apps "green-rs" deleted
service "blue-green-svc" deleted

~/svc$ kubectl delete pod testclient bgtest

pod "testclient" deleted
pod "bgtest" deleted

~/svc$ cd ~

~$
```

Use `kubectl get all` to list all the remaining resources in the default namespace:

```
~$ kubectl get all

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   135m

~$
```

All that should remain is the kubernetes service.

Congratulations you have completed the lab!