This lab is not possible without a multi-host cluster. Minikube is not suffiecent for testing this lab. Please consult your instructor if you have any questions.

Kubernetes Services Management

Objective: Expose a simple application through various types of Services, understand how each Service type handles source IP NAT, and understand the tradeoffs involved in preserving source IP.

Preparation: You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a cluster, you can create one by using Minikube.

Outcome: Particapant will have exposed a simple app thorugh Kubernetes Services and gain a better overall perspective of how to work with applications using Kubernetes.

Data Files: Ask Instructor

Applications running in a Kubernetes cluster find and communicate with each other, and the outside world, through the Service

abstraction. This document explains what happens to the source IP of packets sent to different types of Services, and how you can toggle this behavior according to your needs.

This document makes use of the following terms:

- **NAT:** network address translation
- **Source NAT:** replacing the source IP on a packet, usually with a node's IP
- **Destination NAT:** replacing the destination IP on a packet, usually with a pod IP
- **VIP:** a virtual IP, such as the one assigned to every Kubernetes Service
- **Kube-proxy:** a network daemon that orchestrates Service VIP management on every node

Step 1. Prep Check-up

You must have a working Kubernetes 1.5 cluster to run the examples in this document. The examples use a small nginx webserver that echoes back the source IP of requests it receives through an HTTP header.

1. You can create it as follows:

```
$ kubectl run source-ip-app --image=gcr.io/google_contain
ers/echoserver:1.4
```

Output:

```
deployment "source-ip-app" created
```

Step 2. IP Info

1. Packets sent to ClusterIP from within the cluster are never source NAT’d if you’re running kube-proxy in iptables mode, which is the default since Kubernetes 1.2. Kube-proxy exposes its mode through a proxyMode endpoint:

```
$ kubectl get nodes
```

Output:

```
NAME                           STATUS
 AGE
kubernetes-minion-group-6jst   Ready
 2h
kubernetes-minion-group-cx31   Ready
 2h
kubernetes-minion-group-jj1t   Ready
 2h
```

2. Let’s check our progess using `curl`, like this:

```
$ curl localhost:10249/proxyMode
```

Output:

```
iptables
```

3. You can test source IP preservation by creating a Service over the source IP app:

```
$ kubectl expose deployment source-ip-app --name=clusteri
p --port=80 --target-port=8080
```

Output:

```
service "clusterip" exposed
```

4. Check to see if service is initiated and get the info:

```
$ kubectl get svc clusterip
```

Output:

```
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
clusterip    10.0.170.92   <none>        80/TCP    51s
```

5. And hitting the ClusterIP from a pod in the same cluster:

```
$ kubectl run busybox -it --image=busybox --restart=Never
 --rm
```

Output:

```
Waiting for pod default/busybox to be running, status is
Pending, pod ready: false
```

> *NOTE: If you don't see a command prompt, try pressing enter.*

6. Inside, that's pull up ou ip info, like this:

```
# ip addr
```

Output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc
 noqueue
    link/ether 0a:58:0a:f4:03:08 brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.8/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::188a:84ff:feb0:26a5/64 scope link
       valid_lft forever preferred_lft forever
```

7. Using `wget`, let's do this:

```
# wget -qO - 10.0.170.92
```

Output:

```
CLIENT VALUES:
client_address=10.244.3.8
command=GET
...
```

Step 3. Source IP for Services with Type=NodePort

1. As of Kubernetes 1.5, packets sent to Services with Type=NodePort are source NAT'd by default. You can test this by creating a `NodePort` Service:

```
$ kubectl expose deployment source-ip-app --name=nodeport
 --port=80 --target-port=8080 --type=NodePort
```

```
service "nodeport" exposed
```

```
$ NODEPORT=$(kubectl get -o jsonpath="{.spec.ports[0].nod
ePort}" services nodeport)
```

```
$ NODES=$(kubectl get nodes -o jsonpath='{ $.items[*].sta
tus.addresses[?(@.type=="ExternalIP")].address }')
```

2. If you're running on a cloudprovider, you may need to open up a firewall-rule for the `nodes:nodeport` reported above. Now you can try reaching the Service from outside the cluster through the node port allocated above.

```
$ for node in $NODES; do curl -s $node:$NODEPORT | grep -
i client_address; done
```

Output:

```
client_address=10.180.1.1
client_address=10.240.0.5
client_address=10.240.0.3
```

> *NOTE: That these are not the correct client IPs, they're cluster internal IPs. This is what happens:*

- Client sends packet to `node2:nodePort`
- `node2` replaces the source IP address (SNAT) in the packet with its own IP address
- `node2` replaces the destination IP on the packet with the pod IP
- packet is routed to node 1, and then to the endpoint

- the pod's reply is routed back to node2
- the pod's reply is sent back to the client

Visually:

```
           client
              \ ^
               \ \
                v \
   node 1 <--- node 2
    | ^   SNAT
    | |   --->
    v |
 endpoint
```

To avoid this, Kubernetes 1.5 has a feature triggered by the `service.beta.kubernetes.io/external-traffic` annotation. Setting it to the value `OnlyLocal` will only proxy requests to local endpoints, never forwarding traffic to other nodes and thereby preserving the original source IP address. If there are no local endpoints, packets sent to the node are dropped, so you can rely on the correct source-ip in any packet processing rules you might apply a packet that make it through to the endpoint.

3. Set the annotation as follows:

```
$ kubectl annotate service nodeport service.beta.kubernet
es.io/external-traffic=OnlyLocal service "nodeport" annot
```

```
ated
```

4. Now, re-run the test:

```
$ for node in $NODES; do curl --connect-timeout 1 -s $nod
e:$NODEPORT | grep -i client_address; done
```

Output:

```
client_address=104.132.1.79
```

> *NOTE: That you only got one reply, with the right client IP, from the one node on which the endpoint pod is running on. This is what happens:*

- client sends packet to node2:nodePort, which doesn't have any endpoints
- packet is dropped
- client sends packet to node1:nodePort, which does have endpoints
- node1 routes packet to endpoint with the correct source IP

Visually:

```
        client
      ^ /   \
     / /     \
```

```
     / v       X
   node 1     node 2
    ^ |
    | |
    | v
 endpoint
```

Step 4. Source IP for Services with Type=LoadBalancer

As of Kubernetes 1.5, packets sent to Services with Type=LoadBalancer are source NAT'd by default, because all schedulable Kubernetes nodes in the `Ready` state are eligible for loadbalanced traffic. So if packets arrive at a node without an endpoint, the system proxies it to a node with an endpoint, replacing the source IP on the packet with the IP of the node (as described in the previous section).

1. You can test this by exposing the source-ip-app through a loadbalancer, as follows:

```
$ kubectl expose deployment source-ip-app --name=loadbala
ncer --port=80 --target-port=8080 --type=LoadBalancer
```

Output:

```
service "loadbalancer" exposed
```

```
$ kubectl get svc loadbalancer
```

Output:

```
NAME           CLUSTER-IP    EXTERNAL-IP       PORT(S)
AGE
loadbalancer   10.0.65.118   104.198.149.140   80/TCP
5m
```

```
$ curl 104.198.149.140
```

Output:

```
CLIENT VALUES:
client_address=10.240.0.5
...
```

However, if you're running on GKE/GCE, setting the same `service.beta.kubernetes.io/external-traffic` annotation to `OnlyLocal` forces nodes without Service endpoints to remove themselves from the list of nodes eligible for loadbalanced traffic by deliberately failing health checks. We expect to roll this feature out across a wider range of providers before GA (see next section).

Visually:

```
                      client
                        |
                      lb VIP
                     / ^
                    v /
health check --->   node 1   node 2 <--- health check
        200  <---   ^ |             ---> 500
                    | V
                  endpoint
```

2. You can test this by setting the annotation:

```
$ kubectl annotate service loadbalancer service.beta.kube
rnetes.io/external-traffic=OnlyLocal
```

3. You should immediately see a second annotation allocated by Kubernetes:

```
$ kubectl get svc loadbalancer -o yaml | grep -i annotati
ons -A 2
```

Output:

```
annotations:
    service.beta.kubernetes.io/external-traffic: OnlyLoca
l
    service.beta.kubernetes.io/healthcheck-nodeport: "321
```

```
22"
```

4. The `service.beta.kubernetes.io/healthcheck-nodeport` annotation points to a port on every node serving the health check at `/healthz`. You can test this:

```
$ kubectl get po -o wide -l run=source-ip-app
```

Output:

```
NAME                              READY     STATUS    RESTA
RTS   AGE       IP              NODE
source-ip-app-826191075-qehz4   1/1       Running   0
      20h       10.180.1.136    kubernetes-minion-group-6j
st
```

```
kubernetes-minion-group-6jst $ curl localhost:32122/healt
hz
```

Output:

```
1 Service Endpoints found
```

```
kubernetes-minion-group-jj1t $ curl localhost:32122/healt
hz
```

Output:

```
No Service Endpoints Found
```

5. A service controller running on the master is responsible for allocating the cloud loadbalancer, and when it does so, it also allocates HTTP health checks pointing to this port/path on each node. Wait about 10 seconds for the 2 nodes without endpoints to fail health checks, then curl the lb ip:

```
$ curl 104.198.149.140
```

Output:

```
CLIENT VALUES:
client_address=104.132.1.79
...
```

Step 5. Cross Platform Support

As of Kubernetes 1.5 support for source IP preservation through Services with Type=LoadBalancer is only implemented in a subset of cloudproviders (GCP and Azure). The cloudprovider you're running on might fulfill the request for a loadbalancer in a few different ways:

- With a proxy that terminates the client connection and opens a new connection to your nodes/endpoints. In such cases the

source IP will always be that of the cloud LB, not that of the client.

- With a packet forwarder, such that requests from the client sent to the loadbalancer VIP end up at the node with the source IP of the client, not an intermediate proxy.

Loadbalancers in the first category must use an agreed upon protocol between the loadbalancer and backend to communicate the true client IP such as the HTTP X-FORWARDED-FOR header, or the proxy protocol. Loadbalancers in the second category can leverage the feature described above by simply creating a HTTP health check pointing at the port stored in the `service.beta.kubernetes.io/healthcheck-nodeport` annotation on the Service.

Step 6. Clean Up

1 .Delete the Services:

```
$ kubectl delete svc -l run=source-ip-app
```

2. Delete the Deployment, ReplicaSet and Pod:

```
$ kubectl delete deployment source-ip-app
```

Conclusion

In this lesson, we learned how to expose application through Kubernete’s Services, and we learned how those Services handle

source IP and NAT.