Lesson Plan
Pod Connectivity Fundamentals

Summary

1. Lab Setup
2. How Pods See the Network
3. How Hosts See Connections to Pods
4. How Hosts Route Pod Traffic

Lab Setup

This lab assumes you already have the "Yet Another Online Bank" (yaobank) installed, as created in the “Installing the Sample Application” module in Week 1. If you don’t already have it installed then go back and do so now.

For reference, this is architecture of YAOBank:

If you are not already on host1, you can enter host1 by using the multipass shell command.

```
multipass shell host1
```

How Pods see the Network

We'll start by examining what the network looks like from the pod's point of view. Each pod gets its own Linux network namespace, which you can think of as giving it an isolated copy of the Linux networking stack.

Before we start, from host1, get the details the customer pod:

```
kubectl get pods -n yaobank -l app=customer -o wide
```

Example output:

```
NAME                        READY   STATUS    RESTARTS   AGE    IP
NODE    NOMINATED NODE   READINESS GATES
customer-574bd6cc75-2blnv   1/1     Running   0          9m1s
198.19.22.132   node1   <none>           <none>
```

Note the pod's IP address.

Exec into the pod

Let's exec into the pod so we can see its local view of the network (the IP addresses, network interfaces, and the routing table within the pod).

```
CUSTOMER_POD=$(kubectl get pods -n yaobank -l app=customer -o name)
kubectl exec -ti -n yaobank $CUSTOMER_POD -- /bin/bash
```

Execing into the pod means our command line is now in the context of the pod's network namespace.

Interfaces

To begin, let's take a look at the interfaces within the pod by using ip addr.

```
ip addr
```

Example output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1000
    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@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue state
UP group default
    link/ether 3a:0c:14:d0:92:89 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 198.19.22.132/32 brd 198.19.22.132 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::380c:14ff:fed0:9289/64 scope link
       valid_lft forever preferred_lft forever
```

The key things to note in this output are:

- There is a lo loopback interface with an IP address of 127.0.0.1. This is the standard loopback interface that every network namespace has by default. You can think of it as localhost for the pod itself.
- There is an eth0 interface which has the pods actual IP address, 198.19.22.132. Notice this matches the IP address that kubectl get pods returned earlier.

Next let's look more closely at the interfaces using ip link. We will use the -c option, which colors the output to make it easier to read.

```
ip -c link show up
```

Example output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc noqueue state
UP mode DEFAULT group default
    link/ether 3a:0c:14:d0:92:89 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

The key things to note are:

- eth0 is a link to the host network namespace (indicated by link-netnsid 0). This is the pod's side of the virtual ethernet pair (veth pair) that connects the pod to the node's host network namespace.
- The @if9 at the end of the interface name (on eth0) is the interface number for the other end of the veth pair, which is located within the host's network namespace itself. In this example, interface number 9. Remember this number for later. You might want to write it down, because we will need to know this number when we take a look at the other end of the veth pair shortly.

Routing Table

Finally, let's look at the routes the pod sees.

```
ip route
```

Example output:

```
default via 169.254.1.1 dev eth0
169.254.1.1 dev eth0  scope link
```

This shows that the pod's default route is out over the eth0 interface. i.e. Anytime it wants to send traffic to anywhere other than itself, it will send the traffic over eth0. (Note that the next hop address of 169.254.1.1 is a dummy address used by Calico. Every Calico networked pod sees this as its next hop.)

Exit from the customer pod

We've finished our tour of the pod's view of the network, so we'll exit out of the exec to return to host1.

```
exit
```

How Hosts See Connections to Pods

From the host1 instance we can use kubectl to find which node is running the customer pod. We can do this by running the following command:

```
kubectl get pods -n yaobank -l app=customer -o wide
```

Example output:

```
NAME                        READY   STATUS    RESTARTS   AGE   IP
NODE    NOMINATED NODE   READINESS GATES
customer-574bd6cc75-2blnv   1/1     Running   0          11m
198.19.22.132   node1   <none>           <none>
```

In this case, we can see that the node where the pod is running is node1. Your environment may vary.

SSH into whichever node is hosting the customer pod.

```
ssh node1
```

After we reach the node prompt, we will invoke the same command we used to view the active network interfaces inside the pod.

```
ip -c link show up
```

Example output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode
DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
DEFAULT group default qlen 1000
    link/ether 00:15:5d:60:a5:a3 brd ff:ff:ff:ff:ff:ff
5: cali1eaab2bfc77@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc
noqueue state UP mode DEFAULT group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns
cni-8174e7bb-f2a6-0b61-1282-2c425f949ab5
6: cali9c9ee09e807@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc
noqueue state UP mode DEFAULT group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns
cni-e269d0db-f258-6f12-8f31-064bbb4cf87c
7: calid35188eb0ba@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc
noqueue state UP mode DEFAULT group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns
cni-cebefddd-a569-08c2-29d7-7551967f7cf4
8: calif0a98285df9@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc
noqueue state UP mode DEFAULT group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns
cni-0a89d746-b16d-2299-9e6e-948dd7b2b512
9: caliea2aa288365@if3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1410 qdisc
noqueue state UP mode DEFAULT group default
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netns
cni-f7159375-bf04-e8d8-85b8-0da49bfd53d0
```

Look for the interface number that we noted when looking at the interfaces inside the pod. In our example it was interface number 9. Looking at interface 9 in the above output, we see caliea2aa288365 which links to @if3 in network namespace ID 3 (the customer pod's network namespace). You may recall that interface 9 in the pod's network namespace was eth0, so this looks exactly as expected for the veth pair that connects the customer pod to the host network namespace. The interface numbers in your environment may be different but you should be able to follow the same chain of reasoning.

You can also see the host end of the veth pairs to other pods running on this node, all beginning with cali.

We will look at how the node routes traffic to and from these calico interfaces to the rest of the network next.

Now we can leave node1 and return to host1 by using the exit command.

```
exit
```

How Hosts Route Pod Traffic

First let's remind ourselves of the customer pod's IP address:

```
kubectl get pods -n yaobank -l app=customer -o wide
```

Now let's enter the node hosting the customer pod again.

```
ssh node1
```

Now let's look at the routes on the node.

```
ip route
```

Example output:

```
default via 192.168.44.65 dev eth0 proto dhcp src 192.168.44.75 metric 100
192.168.44.64/28 dev eth0 proto kernel scope link src 192.168.44.75
192.168.44.65 dev eth0 proto dhcp scope link src 192.168.44.75 metric 100
198.19.0.0/20 dev eth0 proto kernel scope link src 198.19.0.2
198.19.21.0/26 via 198.19.0.1 dev eth0 proto bird
198.19.21.64/26 via 198.19.0.3 dev eth0 proto bird
198.19.22.128 dev cali1eaab2bfc77 scope link
blackhole 198.19.22.128/26 proto bird
198.19.22.129 dev cali9c9ee09e807 scope link
198.19.22.130 dev calid35188eb0ba scope link
198.19.22.131 dev calif0a98285df9 scope link
198.19.22.132 dev caliea2aa288365 scope link
```

In this example output, we can see the route to the customer pod's IP (198.19.22.132) is via the caliea2aa288365 interface, the host end of the veth pair for the customer pod. You can see

similar routes for each of the IPs of the other pods hosted on this node. It's these routes that tell Linux where to send traffic that is destined to a local pod on the node.

We can also see several routes labeled proto bird. These are routes to pods on other nodes that Calico has learned over BGP.

To understand these better, consider this route in the example output above 198.19.21.64/26 via 198.19.0.3 dev eth0 proto bird . It indicates pods with IP addresses falling within the 198.19.21.64/26 CIDR can be reached 198.19.0.3 (which is node2) through the eth0 network interface (the host's main interface to the rest of the network). You should see similar routes in your output for each node.

Calico uses route aggregation to reduce the number of routes when possible. (e.g. /26 in this example). The /26 corresponds to the default block size that Calico IPAM (IP Address Management) allocates on demand as nodes need pod IP addresses. (If desired, the block size can be configured in Calico IPAM settings.)

You can also see the blackhole 198.19.22.128/26 proto bird route. The 198.19.22.128/26 corresponds to the block of IPs that Calico IPAM allocated on demand for this node. This is the block from which each of the local pods got their IP addresses. The blackhole route tells Linux that if it can't find a more specific route for an individual IP in that block then it should discard the packet (rather than sending it out the default route to the network). You will only see traffic that hits this rule if something is trying to send traffic to a pod IP that doesn't exist, for example sending traffic to a recently deleted pod.

If Calico IPAM runs out of blocks to allocate to nodes, then it will use unused IPs from other nodes' blocks. These will be announced over BGP as more specific routes, so traffic to pods will always find its way to the right host.

Exit from the node

We've finished our tour of the Customer pod's host's view of the network. Remember to exit out of the exec to return to host1.

```
exit
```