Networking the Netty

A (slightly more than ankle) deep dive into packets in K8s

Anthony Critelli

Kubernetes concept overview

- Node - A host running the K8s stack. Very roughly analogous to a hypervisor in traditional virtualization
- Pod - The smallest schedulable unit of work in K8s.
  - Often, but not necessarily, a single container* in simple environments.
  - A pod is scheduled on a single node
- Service - A group of pods that expose a network service for others to consume.

* Excluding the pause container, in Docker

What problems are we trying to solve?

- It might sound like a stupid question, but: what are we really trying to do when we talk about networking in Kubernetes?
- Well, a few things:
  - We want workload (pods) within our cluster to be able to talk to each other
  - We want workload within our cluster to talk to the *services* that we have in our cluster
  - We want users outside of our cluster to talk with some of our services
- Let’s see how all this works

Networking Within the Cluster

Internal Cluster Networking

- Pods need a way to talk to each other
- There needs to be a way to expose services for other pods in the cluster to access

Example:

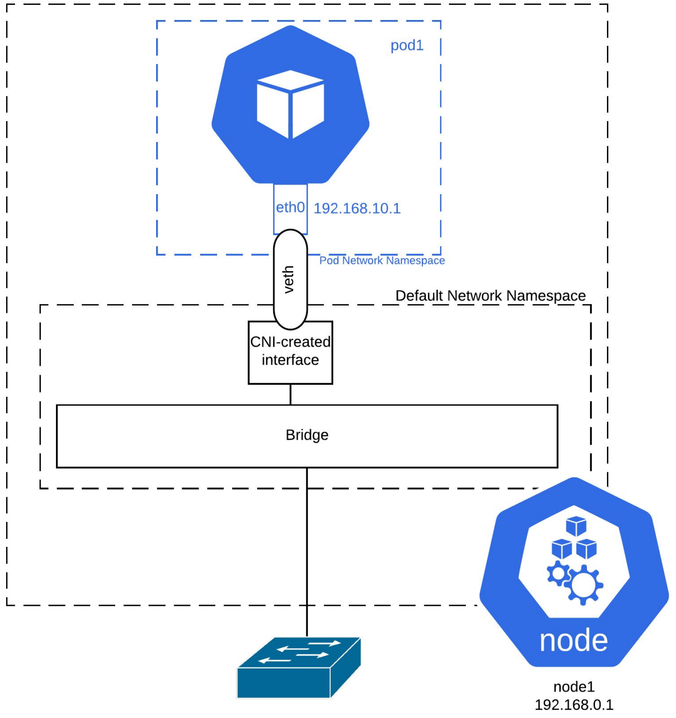
- An etcd *service* is composed of a set of *pods*. It exposes a service endpoint (192.168.100.1) for other pods to access.
- A web service, also composed of multiple pods, needs to be able to hit the etcd service on 192.168.100.1
- They all want to do this without caring about the underlying physical network

The Container Network Interface (CNI)

- The CNI is concerned with (wait for it…) networking for individual containers
- Kubernetes wants each pod to have a IP address
  - The CNI plugin handles allocation and deletion (of interfaces and IPs) as pods are created and destroyed
- Typically, you don't just deploy a CNI plugin. You deploy an entire *network plugin* that also implements a CNI
  - Calico is a popular one: https://www.projectcalico.org/
  - https://kubernetes.io/docs/concepts/cluster-administration/networking/
- Why are there so many options for network plugins? Because different plugins offer different features.
  - More on that soon

How does the CNI work?

- Pods run in different *network namespaces* and we want to connect them to the default system namespace
  - They only see interfaces in their namespace
- To accomplish this Virtual Ethernet (veth) pairs are set up
  - One interface exists in the pod namespace
  - The other interface exists in the default namespace
  - A veth connects them together across the namespace



Step-by-step Example

First, we'll find a pod to look at:

```
fsh$ kubectl get pod -o wide -n demo-ns demo-pod6
NAME        READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
demo-pod6   1/1     Running   8          23h   10.205.133.129   raspi02   <none>           <none>
```

We see this pod is on **raspi02** with an IP of 10.205.133.129. Next, let's find the container ID:

```
root@raspi02:/home/ansible# docker ps | grep demo
7d92754f28ee        alpine                 "sleep 10000"            About an hour ago   Up About an hour
k8s_demo-container_demo-pod6_demo-ns_ec09cb4e-9ec6-4916-8822-01427a8e11df_8
6e2cd91f32b8        k8s.gcr.io/pause:3.1   "/pause"                 24 hours ago        Up 24 hours
k8s_POD_demo-pod6_demo-ns_ec09cb4e-9ec6-4916-8822-01427a8e11df_0
```

For this example, it doesn't matter if you use the app container or the pause container. They share the same network namespace. We'll go with the app (7d92754f28ee)

Step-by-step Example Continued

Then we figure out what the PID of the container is:

```
root@raspi02:/home/ansible# docker inspect 7d92754f28ee --format '{{ .State.Pid }}'
2547
```

Let’s enter the namespace for that PID and examine the interface:

```
root@raspi02:/home/ansible# nsenter -t 2547 -n ip -br a
lo               UNKNOWN        127.0.0.1/8
tunl0@NONE       DOWN
eth0@if50        UP             10.205.133.129/32
```

Hey look! That’s the pod IP from the earlier kubectl output (10.205.133.129).

Step-by-step Example Continued

Now, how does this talk to the “host”? Let’s see what the other end of the veth is:

```
root@raspi02:/home/ansible# nsenter -t 2547 -n ethtool -S eth0
NIC statistics:
     peer_ifindex: 50
     rx_queue_0_xdp_packets: 0
     rx_queue_0_xdp_bytes: 0
     rx_queue_0_xdp_drops: 0
```

The other end of this veth tunnel is interface index 50. So now we have to find the interface with index 50 in the default namespace:

```
root@raspi02:/home/ansible# ip link sh | grep '^50'
50: caliae4a3a51b92@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1440 qdisc noqueue state UP mode DEFAULT group default
```

The host-side interface here is caliae4a3a51b92, which appears to be bridged to interface index 4.

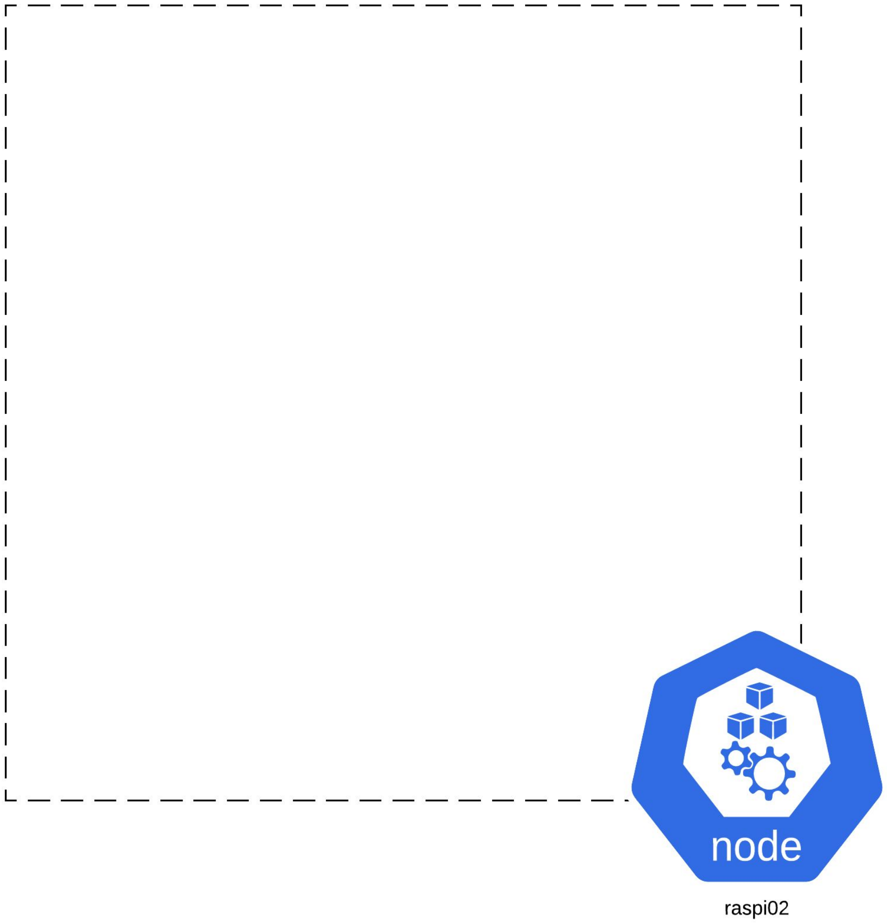
Step-by-step Example Continued

So what's interface index 4?

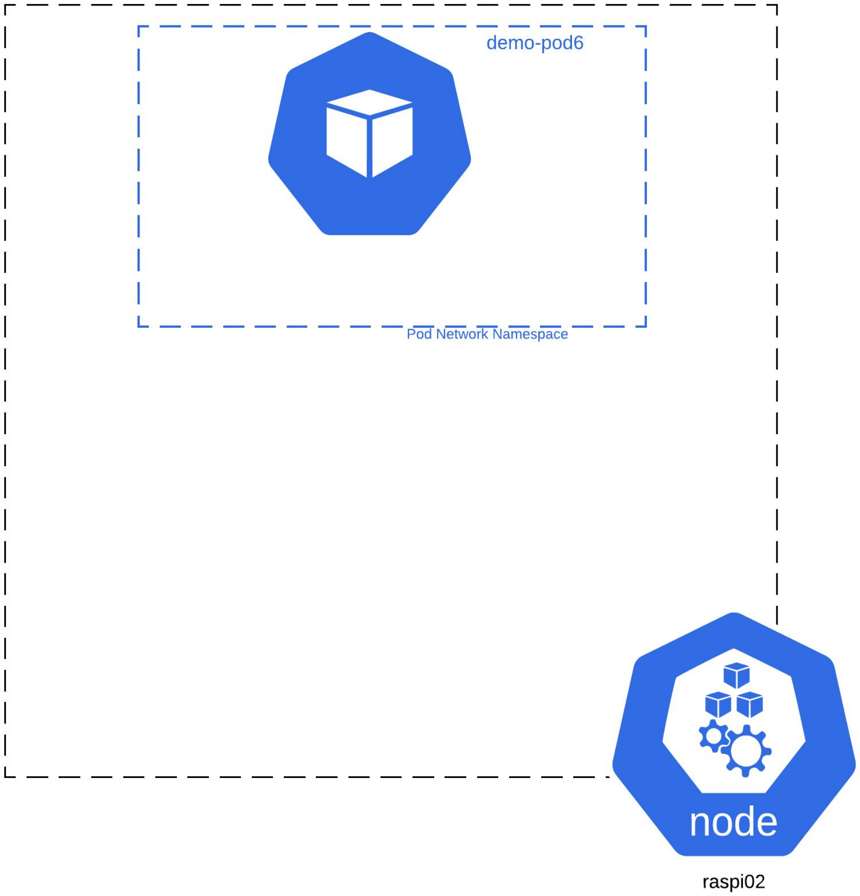
```
root@raspi02:/home/ansible# ip link sh | grep '^4:'
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
```

The docker bridge! Now we're "bridged" to an interface (technically a bridge) on the host. We'll stop here, but at this point: traffic would just use the normal host routing table.

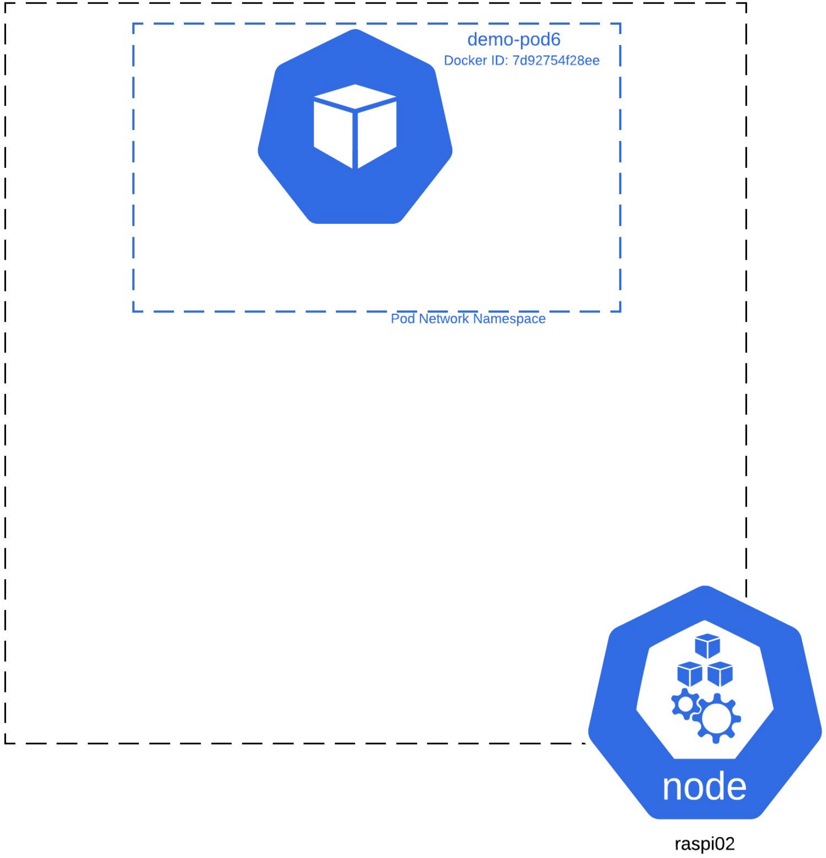
The same sample, but overlaid on a diagram


node
raspi02

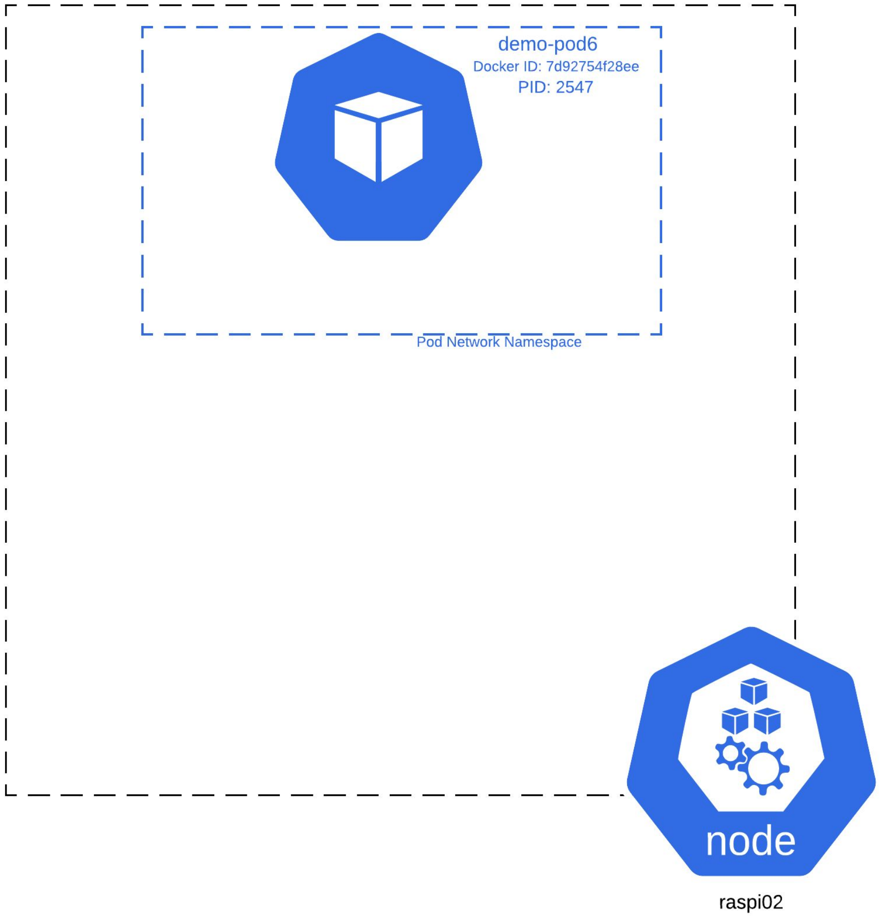
```
fsh$ kubectl get pod -o wide -n demo-ns demo-pod6
NAME        READY   STATUS    RESTARTS   AGE   IP               NODE      NOMINATED NODE   READINESS GATES
demo-pod6   1/1     Running   8          23h   10.205.133.129   raspi02   <none>           <none>
```



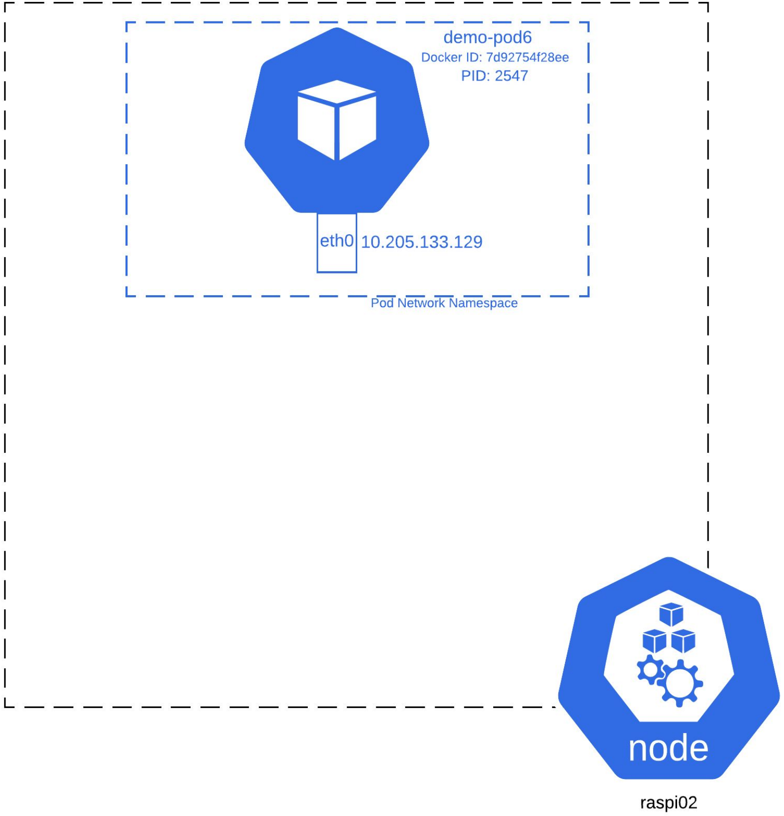
```
root@raspi02:/home/ansible# docker ps | grep demo
7d92754f28ee        alpine                  "sleep 10000"            About an hour ago   Up About an hour
k8s_demo-container_demo-pod6_demo-ns_ec09cb4e-9ec6-4916-8822-01427a8e11df_8
6e2cd91f32b8        k8s.gcr.io/pause:3.1    "/pause"                 24 hours ago        Up 24 hours
k8s_POD_demo-pod6_demo-ns_ec09cb4e-9ec6-4916-8822-01427a8e11df_0
```



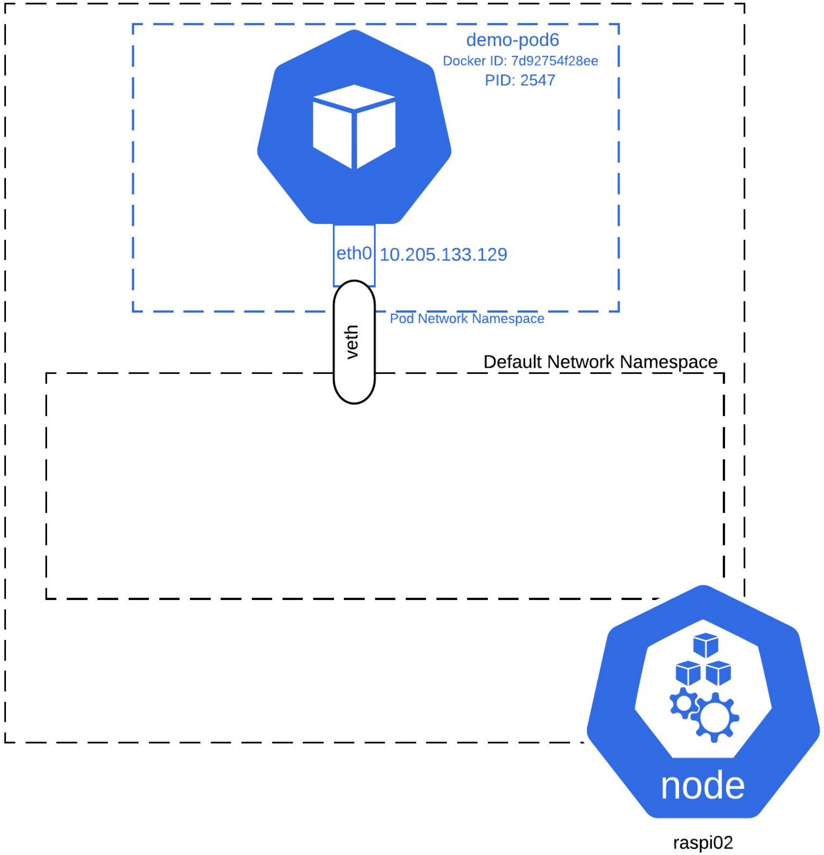
```
root@raspi02:/home/ansible# docker inspect 7d92754f28ee --format '{{ .State.Pid }}'
2547
```



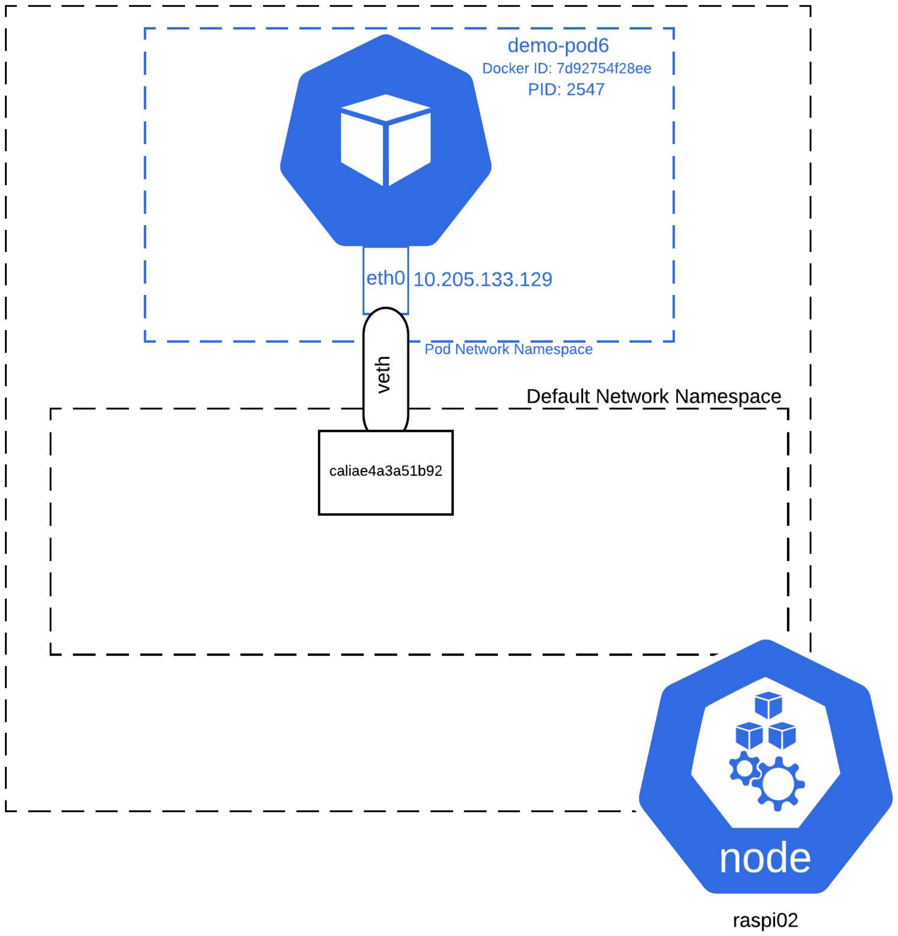
```
root@raspi02:/home/ansible# nsenter -t 2547 -n ip -br a
lo               UNKNOWN        127.0.0.1/8
tunl0@NONE       DOWN
eth0@if50        UP             10.205.133.129/32
```



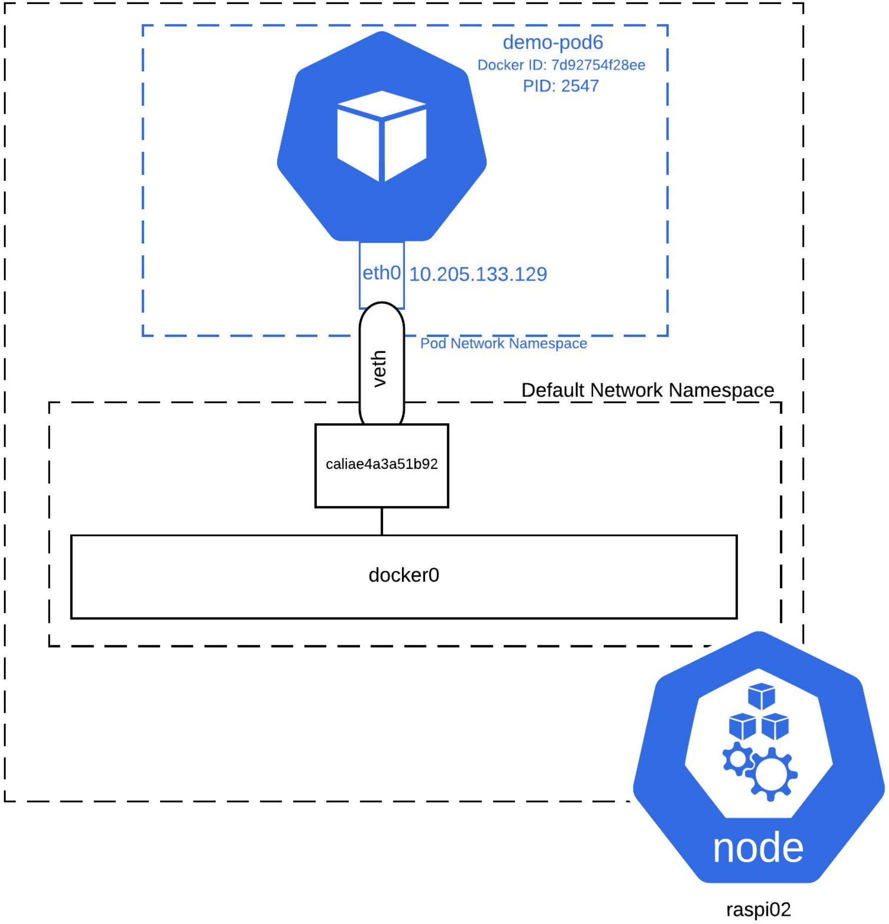
```
root@raspi02:/home/ansible# nsenter -t 2547 -n ethtool -S eth0
NIC statistics:
     peer_ifindex: 50
     rx_queue_0_xdp_packets: 0
     rx_queue_0_xdp_bytes: 0
     rx_queue_0_xdp_drops: 0
```



```
root@raspi02:/home/ansible# ip link sh | grep '^50'
50: caliae4a3a51b92@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1440 qdisc noqueue state UP mode DEFAULT group default
```



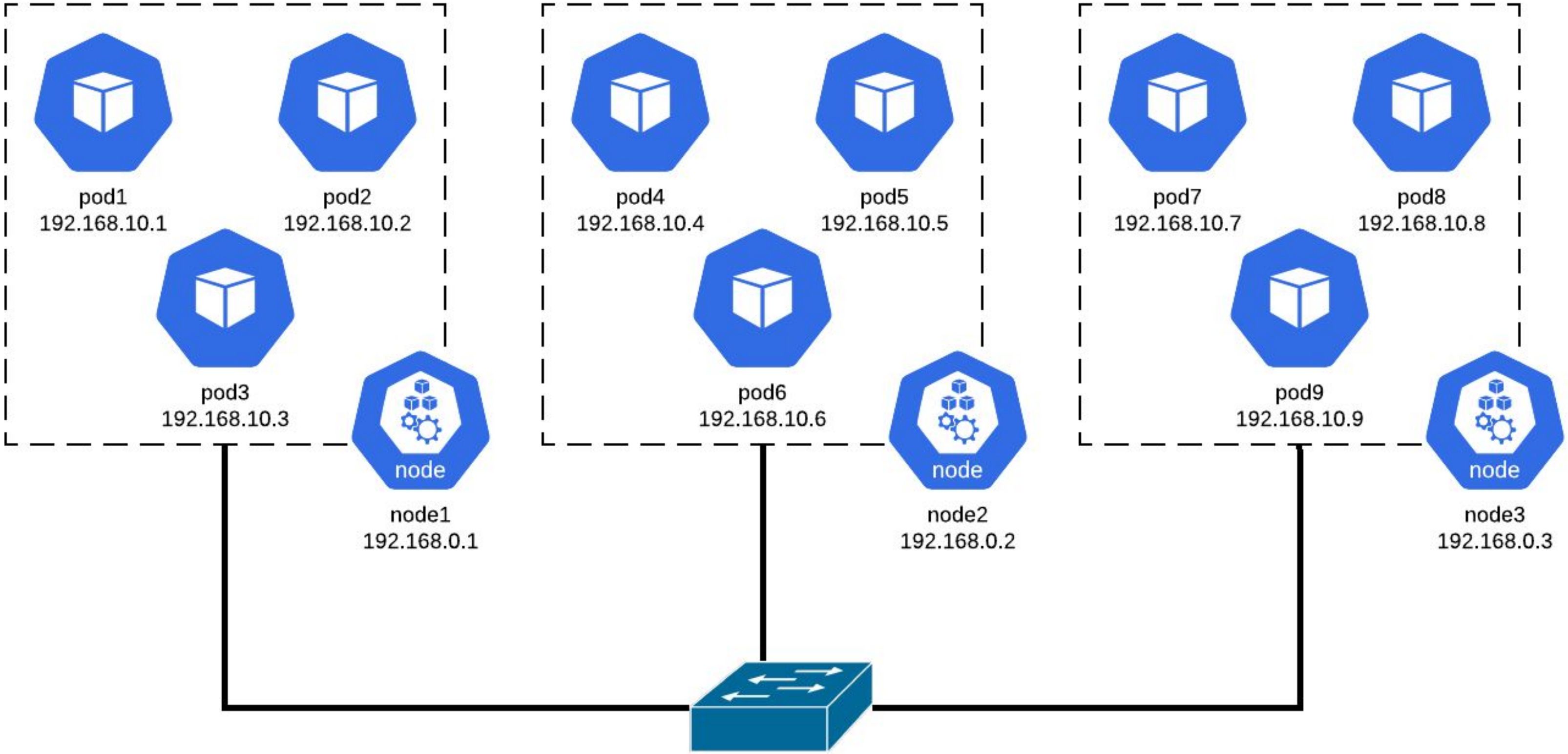
```
root@raspi02:/home/ansible# ip link sh | grep '^4:'
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
```



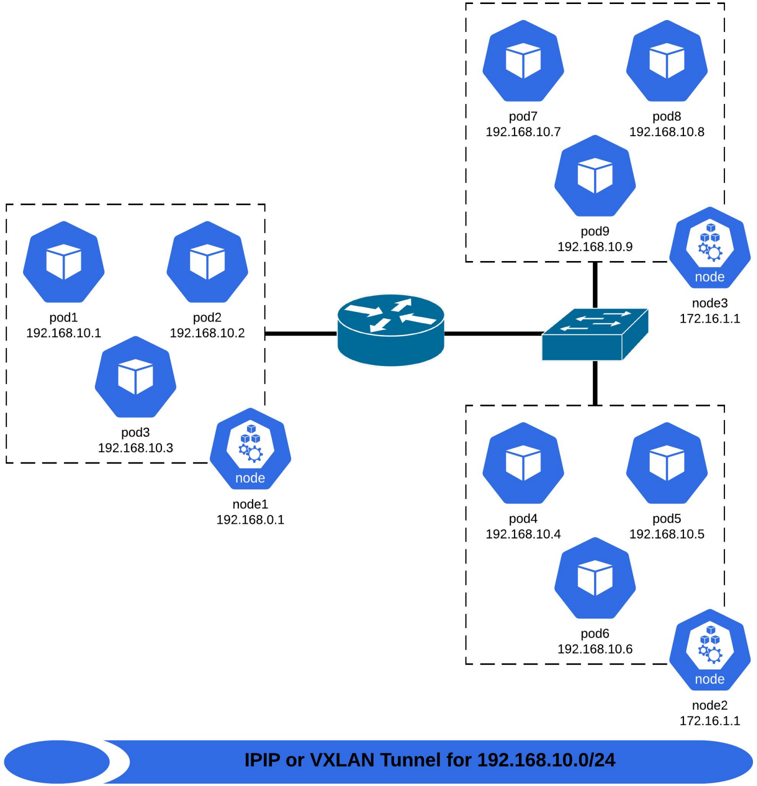
Network Plugins - Beyond just IPs

- In a really basic topology, all of your K8s nodes could be on one layer 2 subnet
  - In this case, the network plugin can (mostly) just hand out IP addresses and configure container NICs
- But maybe you want more advanced functionality
  - BGP advertisement of your pod IPs
  - Stretched layer 2 / the ability to put your nodes on different IP subnets
  - Firewalling capabilities and security controls to enforce policy for inter-pod communication
- Different network plugins have different advanced capabilities
- Let’s look at 2 basic examples of how network plugins might handle pod addressing and communication

Simple topology - single broadcast domain



More advanced topology - stretched L2



But what about service IPs?

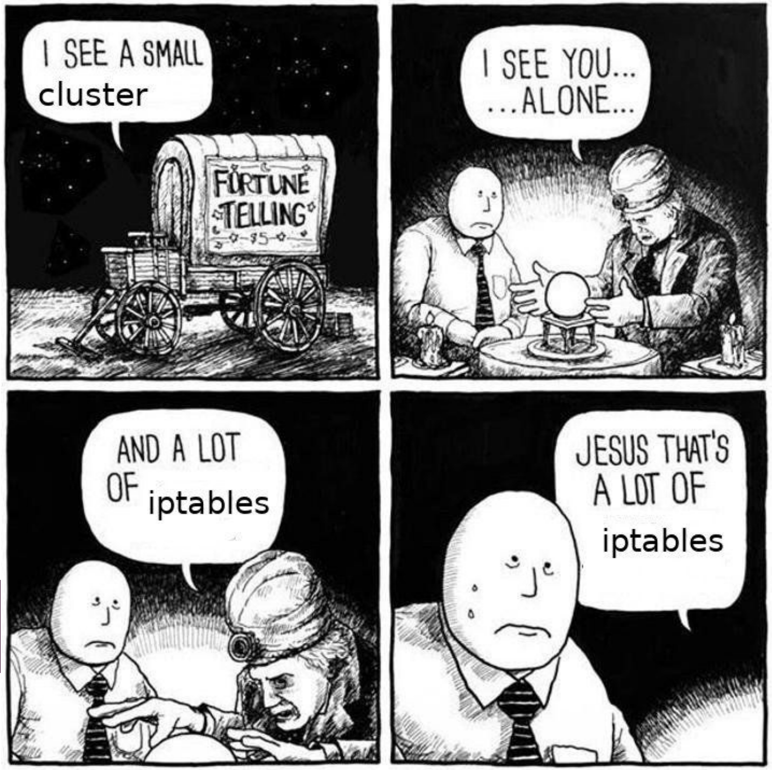
- Remember that a group of pods can expose a *service* that is reachable via a single IP within the cluster
- How does that work?
- How can we ensure that a single IP is reachable everywhere, and potentially served by multiple pods? What if a pod tries to access a service that doesn't have any pods on the same host?
    - E.g., what if our web service tried to access the etcd service, but there were no local pods?
- Enter the kube-proxy

kube-proxy

“What is the most morally depraved thing we can do with iptables?” - Someone building Kubernetes, probably

- Kube-proxy is a container that runs on every node
- It handles programming of iptables rules and network config so that services are reachable

```
root@raspi03:/home/ansible# iptables-save | wc -l
559
```



Meme credit: https://itnext.io/kubernetes-networking-behind-the-scenes-39a1ab1792bb

kube-proxy continued

- A service is exposed internally via a ClusterIP
  - A pod can connect to this IP from any node
- This is accomplished via a bunch of iptables magic
  - You can also use IPVS or the kube-proxy can forward traffic for you
  - iptables mode seems to be the most common approach

In this example, any pod that tries to hit 10.103.220.213 will magically get sent to a pod for the “hass-service”:

```
fsh$ kubectl get services -o wide
NAME                  TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)                        AGE     SELECTOR
hass-service          LoadBalancer   10.103.220.213   10.200.1.42    80:31842/TCP                   16d     app=hass
kubernetes            ClusterIP      10.96.0.1        <none>         443/TCP                        17d     <none>
mqtt-service          LoadBalancer   10.107.180.209   10.200.1.43    1883:31503/TCP,9001:31925/TCP  7d22h   app=mqtt
pihole-tcp-service    LoadBalancer   10.100.51.69     10.200.1.41    80:30105/TCP,443:31676/TCP     17d     app=pihole
pihole-udp-service    LoadBalancer   10.108.185.152   10.200.1.40    53:31413/UDP,67:30280/UDP      17d     app=pihole
```

kube-proxy Step-by-step

First, let’s look at the ports involved in a service. There are 3 that we care about in the example below:

- Port: this is the port that the service is actually accessible on from within the cluster
- TargetPort: this is where Kubernetes will forward the traffic to in the container
- NodePort: the port on the node from which the service is accessible

```
fsh$ kubectl describe service hass-service | grep -i port
Port:                     <unset>  80/TCP
TargetPort:               8123/TCP
NodePort:                 <unset>  31842/TCP
HealthCheck NodePort:     32449
```

kube-proxy Step-by-step

Port: A different pod in the cluster can hit the ClusterIP (10.103.220.213) on the Port (80) and get a response:

```
fsh$ kubectl exec pihole-deployment-76bb945847-x4q9d -- curl -s http://10.103.220.213:80
<!DOCTYPE html><html lang="en"><head><link rel="preload" href="/frontend_latest/core.019f4c68.js" as="script" crossorigin="use-credentials"><link
rel="preload" href="/static/fonts/roboto/Roboto-Regular.woff2" as="font" crossorigin><link rel="preload" href="/static/fonts/roboto/Rob
```

TargetPort: The app in this pod listens on 8123.

```
fsh$ kubectl exec hass-deployment-5959fd979d-2bksh -- netstat -tl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:8123            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:8989            0.0.0.0:*               LISTEN
```

NodePort: I can hit the node port (31842) from the host and get a web response

```
root@raspi03:/home/ansible# curl localhost:31842
<!DOCTYPE html><html lang="en"><head><link rel="preload" href="/frontend_latest/core.019f4c68.js" as="script" crossorigin="use-credentials"><link
rel="preload" href="/static/fonts/roboto/Rob
```

kube-proxy Step-by-step

First, let’s look at the PREROUTING chain in the NAT table on a host:

```
root@raspi03:/home/ansible# iptables -t nat -nvL PREROUTING
Chain PREROUTING (policy ACCEPT 461 packets, 46004 bytes)
 pkts bytes target     prot opt in     out     source               destination
 197K   17M cali-PREROUTING  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* cali:6gwbT8clXdHdC1b1 */
 197K   17M KUBE-SERVICES  all  --  *      *       0.0.0.0/0            0.0.0.0/0            /* kubernetes service portals */
  233 14510 DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LOCAL
```

That KUBE-SERVICES chain looks like it might be relevant. Let’s look for our service IP (10.103.220.213):

```
root@raspi03:/home/ansible# iptables -t nat -nvL KUBE-SERVICES | grep '10.103.220.213'
    0     0 KUBE-MARK-MASQ  tcp  --  *      *      !10.201.0.0/16        10.103.220.213       /* default/hass-service: cluster IP */ tcp dpt:80
    0     0 KUBE-SVC-WCJSPVZVDFRKGG4S  tcp  --  *      *       0.0.0.0/0            10.103.220.213       /* default/hass-service: cluster IP */
    tcp dpt:80
```

KUBE-SVC sounds relevant. Let’s look at that.

kube-proxy Step-by-step

Looking at the relevant KUBE-SVC chain:

```
root@raspi03:/home/ansible# iptables -t nat -nvL KUBE-SVC-WCJSPVZVDFRKGG4S
Chain KUBE-SVC-WCJSPVZVDFRKGG4S (3 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-SEP-6NGDCEZCNCPTK3HH  all  --  *      *       0.0.0.0/0            0.0.0.0/0
```

Another layer of indirection! Let’s look at KUBE-SEP:

```
root@raspi03:/home/ansible# iptables -t nat -nvL KUBE-SEP-6NGDCEZCNCPTK3HH
Chain KUBE-SEP-6NGDCEZCNCPTK3HH (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-MARK-MASQ  all  --  *      *       10.201.133.152       0.0.0.0/0
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp DNAT [unsupported revision]
```

A destination NAT! I think the “[unsupported revision]” here is a bug/discrepancy between the iptables and nftables version. If this bug weren’t there, you’d see the container’s IP as the DNAT.

kube-proxy Step-by-step

Here’s what a DNAT should look like (taken from my minikube VM):

```
$ iptables -t nat -nvL KUBE-SEP-KLHDI3EBBS77E4EW
Chain KUBE-SEP-KLHDI3EBBS77E4EW (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 KUBE-MARK-MASQ  all  --  *      *       172.17.0.5           0.0.0.0/0
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp to:172.17.0.5:80
```

```
fsh$ kubectl get pods -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP           NODE       NOMINATED NODE   READINESS GATES
demo-deployment-6f76c696df-5t6r7  1/1     Running   0          20s   172.17.0.5   minikube   <none>           <none>
```

External Networking

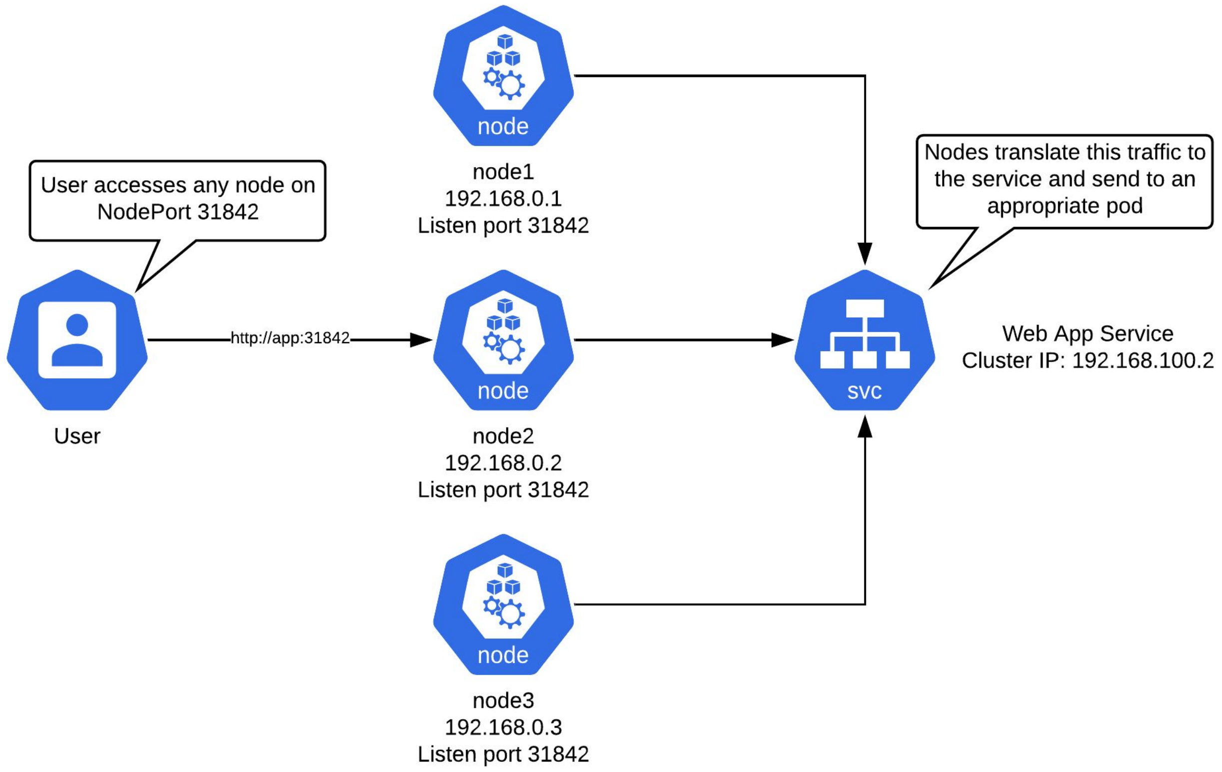
Handling External Traffic

- So you’ve built a Service
    - It has multiple pods
    - Each pod has received an IP from the CNI
    - All the pods can talk to each other because you’ve used a network plugin
    - Internally, your service is accessible via a ClusterIP. All pods can hit it.
- But how do your end user’s talk to these services in your cluster?
    - NodePort
    - LoadBalancer
    - ExternalName (DNS based, not discussed here)

NodePort

- A very simple way of exposing your services outside the cluster without any additional configuration
- A NodePort opens up a random port (if you don’t specify) on **all** hosts in the cluster
  - Traffic to this port is automatically forwarded to a pod hosting your service
- You basically need to do your own load balancing to make this usable
  - E.g., you could put NGINX or HAProxy in front of the cluster to route traffic based on web addresses
- And then you’d need a way to keep track of all the ports and update the upstream load balancer.
  - Do-able, as many LBs have APIs, but a pain (maybe less so with an Operator).

NodePort Diagram



NodePort Example

I have a service that has been given a NodePort of 31018 (which will send traffic to port 80 in the container:

```
fsh$ kubectl get services
NAME                 TYPE          CLUSTER-IP      EXTERNAL-IP   PORT(S)                              AGE
demo-nginx-service   NodePort      10.98.229.128   <none>        80:31018/TCP                         2m41s
```

There’s only one pod, and it’s on raspi03:

```
fsh$ kubectl get pods --selector=app=demo-nginx -o custom-columns=NAME:.metadata.name,NODE:.spec.nodeName
NAME                                NODE
demo-deployment-6fc546dc5-v4n6x     raspi03
```

NodePort Example

I can hit the service from any of the three nodes in my cluster on port 31018:

```
fsh$ http raspi03:31018 --headers
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: keep-alive
Content-Length: 612
Content-Type: text/html
Date: Sun, 29 Mar 2020 18:28:32 GMT
ETag: "5e5e6a8f-264"
Last-Modified: Tue, 03 Mar 2020 14:32:47 GMT
Server: nginx/1.17.9
```

```
fsh$ http raspi02:31018 --headers
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: keep-alive
Content-Length: 612
Content-Type: text/html
Date: Sun, 29 Mar 2020 18:28:36 GMT
ETag: "5e5e6a8f-264"
Last-Modified: Tue, 03 Mar 2020 14:32:47 GMT
Server: nginx/1.17.9
```

```
fsh$ http raspi01:31018 --headers
HTTP/1.1 200 OK
Accept-Ranges: bytes
Connection: keep-alive
Content-Length: 612
Content-Type: text/html
Date: Sun, 29 Mar 2020 18:36:33 GMT
ETag: "5e5e6a8f-264"
Last-Modified: Tue, 03 Mar 2020 14:32:47 GMT
Server: nginx/1.17.9
```

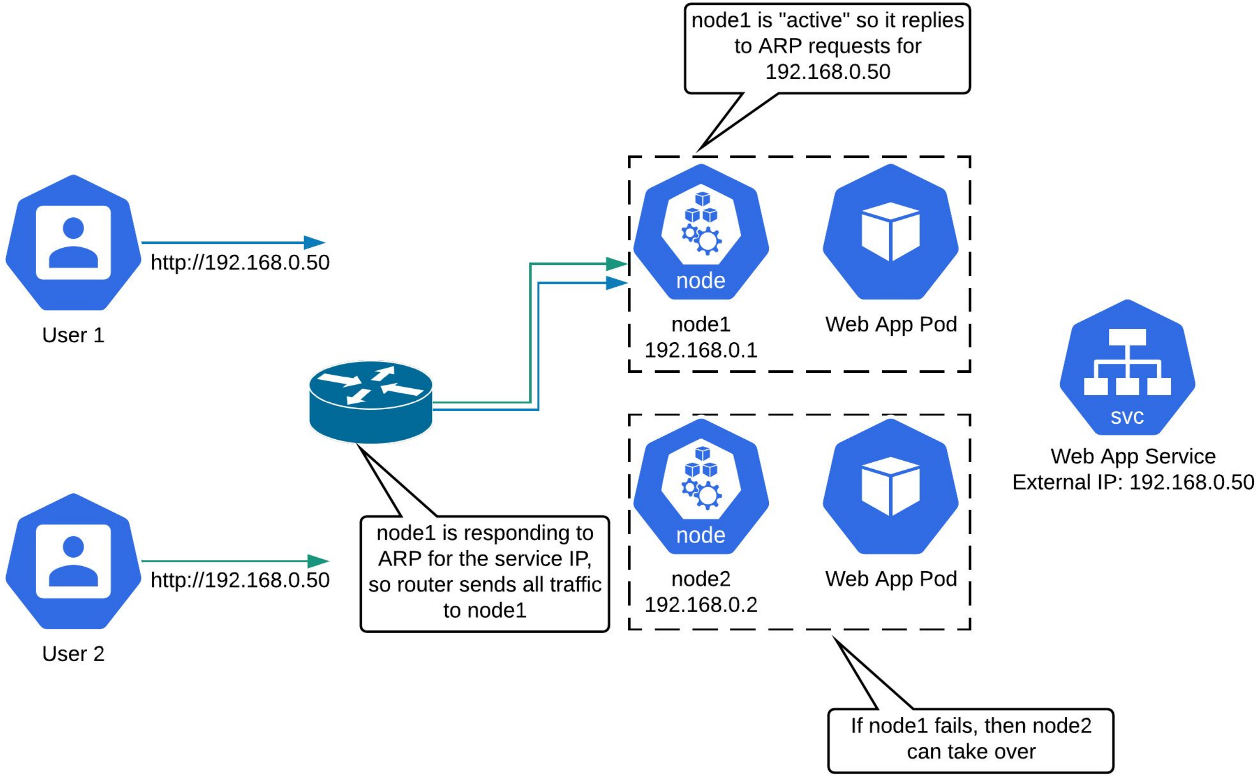
LoadBalancer

- LoadBalancers assign an externally reachable IP to your service and then figure out how to get that traffic into your cluster.
  - LoadBalancers are effectively just API calls to external cloud provider services, such as an AWS load balancer
  - There is **no reference implementation for on-premises deployments** (this is totally insane to me)
- MetalLB: Basically the only accessible way to do this on-prem
  - So we better all hope this project stays around for a long time
- Let's take a look at MetalLB
  - It's probably what you'll use if you're doing an on-prem deployment
  - It'll illustrate why networking knowledge is important for operating K8s

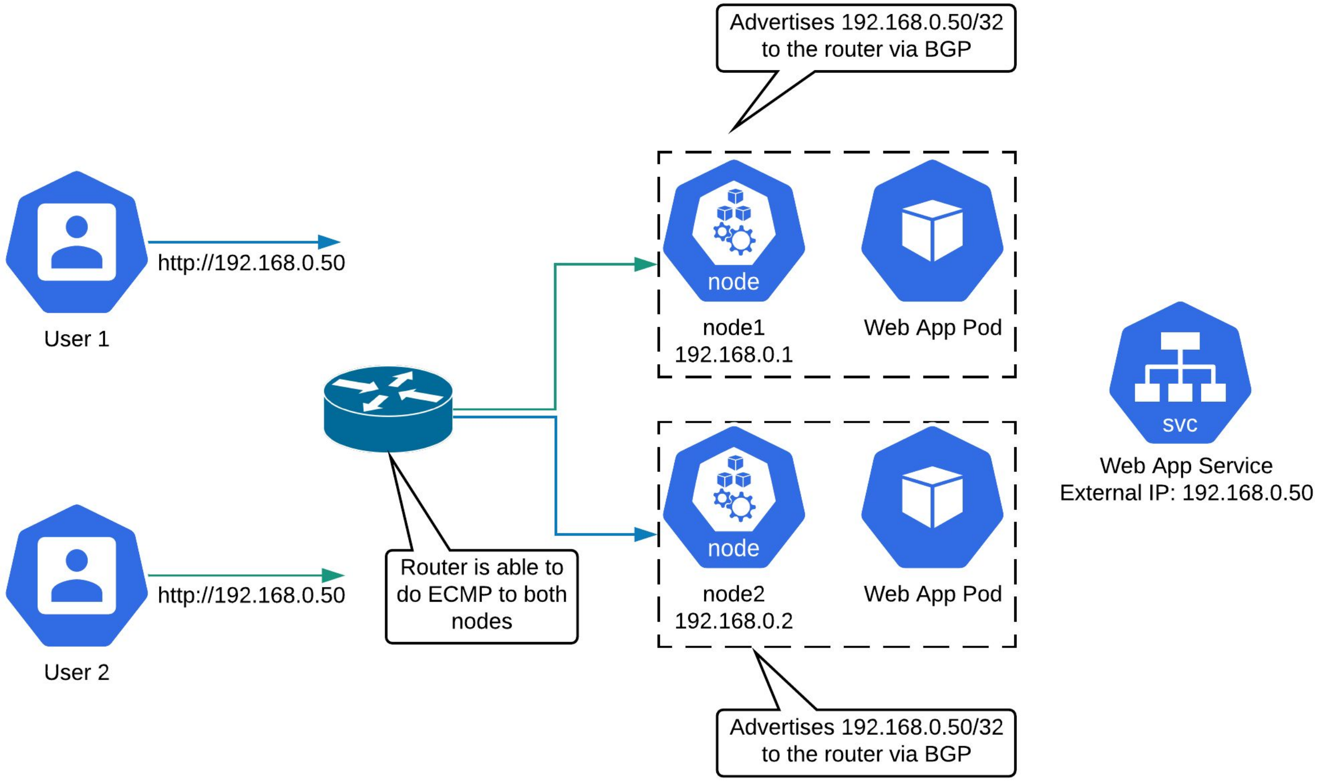
MetalLB Fundamentals

- You configure MetalLB with one or more *pools* of IP addresses, and it hands them out to your services
- MetalLB has two modes of operation: ARP Mode and BGP Mode
- ARP mode - the nodes that are hosting pods for the particular service will respond to ARP requests for the service's external IP address
  - There's an obvious problem with this: it means that only one node can be active at a time
    - Otherwise you can't maintain a user's session: each packet might end up going to a different node
  - This is very much like Keepalived
- BGP mode - each node with a pod for a service will advertise that service's IP to an upstream router using Border Gateway Protocol
  - Allows for true ECMP performed by upstream router
  - Need to be cognizant of the impact of topology changes on reachability

MetalLB - ARP Mode



MetalLB - BGP Mode



Wrapping up

Please stop talking

Finally, why should you care about any of this?

- I think networking, especially in “hashtagCloudNative” workloads, is very important
- Too often, people handwave over the networking configuration
  - This is fine, until it breaks and you have no idea how to even begin troubleshooting
  - “kubectl -f github.com/yolo-network-project/my-plugin.yml” isn’t “how the network works”
- Luckily, as you’ve seen: it’s not too hard.
  - It’s just a bunch of iptables and strung-together network fundamentals
  - Network protocols are still network protocols
- A real example:
  - I’ve got MetalLB configured to hand out IPs on a wireless subnet of mine
  - Every ~5 minutes, without fail, I’d stop being able to reach a service
  - If I fired up a packet capture on a host, the problem would immediately disappear.
  - Anyone know what the issue was?

The issue in my example

- ...the symptom (losing connectivity every ~5 minutes) was related to ARP
  - For some reason, MetalLB on the nodes would stop responding to ARPs for the service IP
  - This was probably obvious to anyone who has worked in network engineering
- But why would it go away when trying to observe the problem with tcpdump?
  - Well, what does tcpdump do to an interface? It sets it into promiscuous mode.
  - This causes lower layer traffic that isn’t destined for a real interface on the node to filter up through the network stack (normally this would be filtered out)
- Solution: apparently, raspis need their NICs manually set to promiscuous mode for MetalLB to work

Additional Resources

- [Kubernetes Networking: Behind the scenes]
    - A really outstanding guide that helped me when I got stuck trying to reverse engineer the CNI
- Kubernetes Docs: Publishing Services
    - More info on how to publish services outside the cluster
- MetalLB Concepts
    - The MetalLB docs are quite good
- A Deep Dive into Iptables and Netfilter Architecture
    - A great discussion of how to follow iptables tables and chains
- A really awesome network flow chart
    - Specifically for kube-proxy in iptables mode

Thanks!