More Than a Series of Tubes Networking in Kubernetes

Jeff Poole

Vivint SmartHome

What happens when you start with Kubernetes

- You deploy a Kubernetes cluster
- Everything is amazing! Jobs are getting scheduled and scaled!
- "Huh, that was odd..."
- "Why can pod A not talk to pod B?"
- "Why can pod A *sometimes* not talk to pod B?"
- "Why do I even work with computers in the first place?"

Network problems

When the network goes bad, sometimes it isn't clear why.

Sometimes it only goes a little bad.

Sometimes it is triggered by something we didn't expect or could directly see.

Too much to know

As of the time I am writing this, there are 19 CNI plugins listed in the Kubernetes documentation on networking

In general, each has a ton of settings

Most people running Kubernetes aren't networking experts

The most "mature" solutions are only a few years old

"A computer lets you make more mistakes faster than any other invention with the possible exceptions of handguns and tequila."
- *Mitch Ratcliffe*

"Learning"

But, fortunately, in the tech world, we have a culture of treating mistakes as a form of learning.

So we can learn really, really fast!

"Learning"

We've learned a lot about Kubernetes networking

My main focus here is where we have the most expertise: **Flannel** and **Calico**, running on bare metal hardware

That also happens to cover both a Layer 2 (VXLAN) and Layer 3 (IP-in-IP) overlay

Audience poll

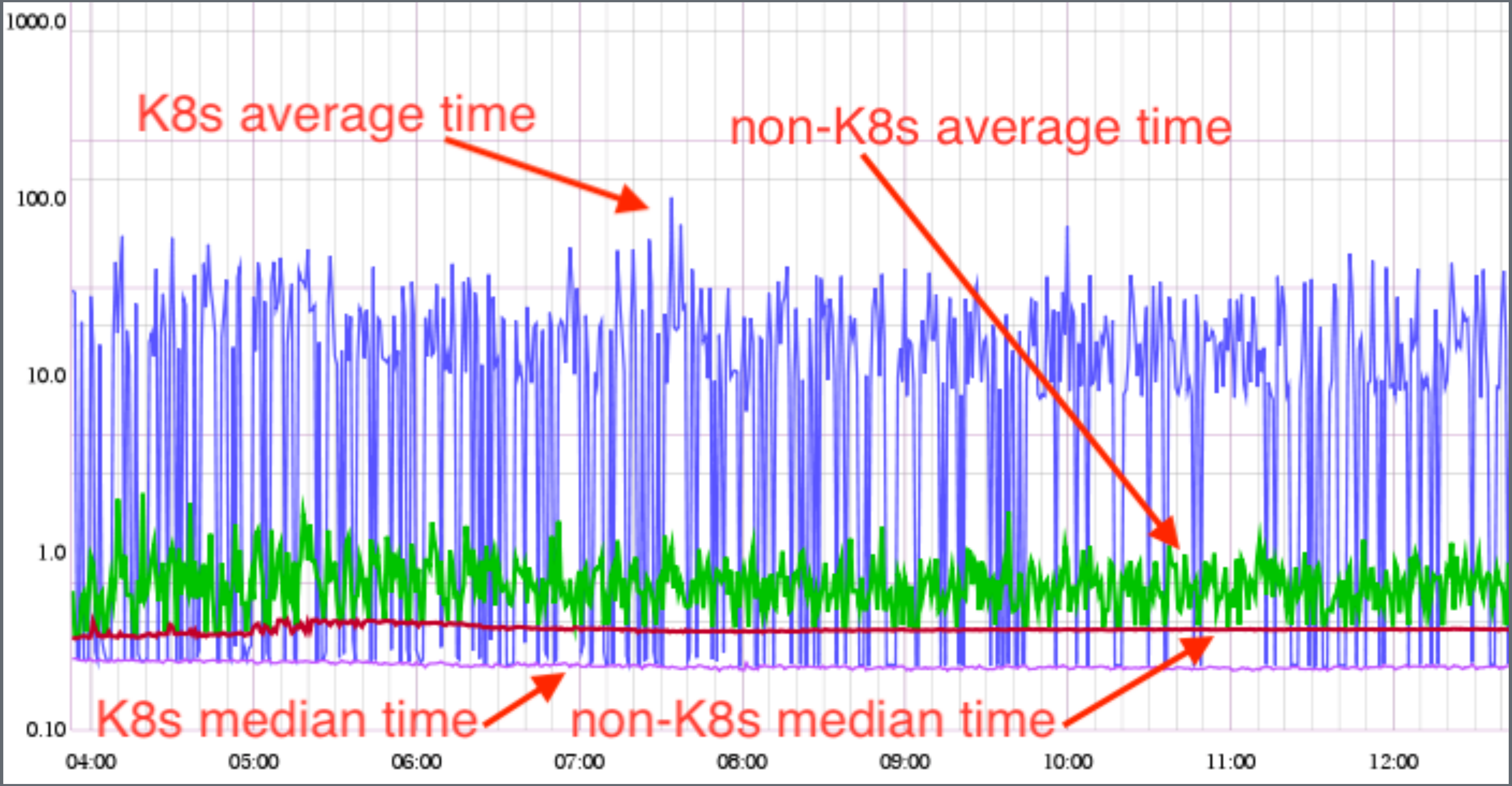
- Who has used Kubernetes at all?
- Who has used Kubernetes in *production*?
- Who has a *networking* background?
- … a *software development* background?
- … an *operations* background?
- Who here has had to debug a problem with Kubernetes networking?

Won't the network *just work*?

- As **operations**, we set up and run systems we don't understand
- As **developers**, we can't debug communication problems between our code and other parts of the system
- As **network engineers**, we may not know anything about Kubernetes or how to make sure the overlay network is playing nicely with the underlying network

Our first experience with Kubernetes networking issues

- It was ~~a dark and stormy night~~ early 2017
- We were running our first production Kubernetes cluster, using Weave as the network plugin
- Our CoreOS servers had recently moved to Linux Kernel 4.10


1000.0
100.0
10.0
1.0
0.10
K8s average time
non-K8s average time
K8s median time
non-K8s median time
04:00
05:00
06:00
07:00
08:00
09:00
10:00
11:00
12:00

Other signs

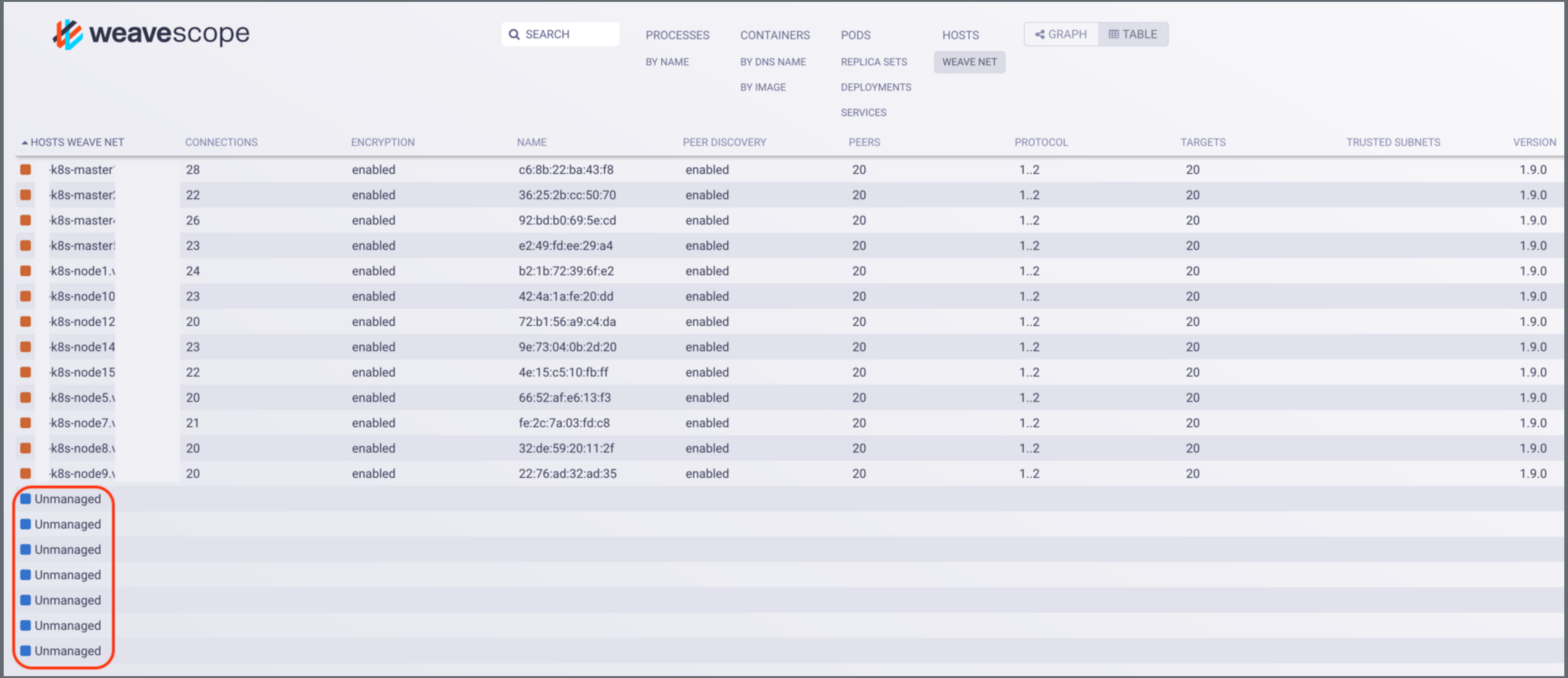
- Tons of these messages showing up in the Weave logs:

  ```
  ERRO: 2017/03/14 02:05:34.430803 Captured frame from MAC (a6:36:9b:8e:37:b6) to
  (0a:a9:1b:94:f0:e7) associated with another peer b2:1b:72:39:6f:e2(k8s-node1)
  ERRO: 2017/03/14 02:05:34.465847 Captured frame from MAC (d2:70:a3:f3:fd:19) to
  (0a:a9:1b:94:f0:e7) associated with another peer 32:de:59:20:11:2f(k8s-node8)
  ```

  (of course, we found a ticket that indicated this *sometimes* is not an actual error)

- Pods failing due to:

  ```
  Readiness probe failed: Get http://172.21.144.70:8081/readiness: dial tcp
  172.21.144.70:8081: connect: no buffer space available
  ```



weavescope

SEARCH | PROCESSES | CONTAINERS | PODS | HOSTS | GRAPH | TABLE
BY NAME | BY DNS NAME | REPLICA SETS | WEAVE NET
BY IMAGE | DEPLOYMENTS
SERVICES

HOSTS WEAVE NET	CONNECTIONS	ENCRYPTION	NAME	PEER DISCOVERY	PEERS	PROTOCOL	TARGETS	TRUSTED SUBNETS	VERSION
🟧 k8s-master	28	enabled	c6:8b:22:ba:43:f8	enabled	20	1..2	20		1.9.0
🟧 k8s-master	22	enabled	36:25:2b:cc:50:70	enabled	20	1..2	20		1.9.0
🟧 k8s-master	26	enabled	92:bd:b0:69:5e:cd	enabled	20	1..2	20		1.9.0
🟧 k8s-master	23	enabled	e2:49:fd:ee:29:a4	enabled	20	1..2	20		1.9.0
🟧 k8s-node1.	24	enabled	b2:1b:72:39:6f:e2	enabled	20	1..2	20		1.9.0
🟧 k8s-node10	23	enabled	42:4a:1a:fe:20:dd	enabled	20	1..2	20		1.9.0
🟧 k8s-node12	20	enabled	72:b1:56:a9:c4:da	enabled	20	1..2	20		1.9.0
🟧 k8s-node14	23	enabled	9e:73:04:0b:2d:20	enabled	20	1..2	20		1.9.0
🟧 k8s-node15	22	enabled	4e:15:c5:10:fb:ff	enabled	20	1..2	20		1.9.0
🟧 k8s-node5.	20	enabled	66:52:af:e6:13:f3	enabled	20	1..2	20		1.9.0
🟧 k8s-node7.	21	enabled	fe:2c:7a:03:fd:c8	enabled	20	1..2	20		1.9.0
🟧 k8s-node8.	20	enabled	32:de:59:20:11:2f	enabled	20	1..2	20		1.9.0
🟧 k8s-node9.	20	enabled	22:76:ad:32:ad:35	enabled	20	1..2	20		1.9.0
🟦 Unmanaged									
🟦 Unmanaged									
🟦 Unmanaged									
🟦 Unmanaged									
🟦 Unmanaged									
🟦 Unmanaged									
🟦 Unmanaged									

There were signs that our Weave nodes couldn't stay connected to each other...

What did we do?

- We gave up
- Moved the workloads off of Kubernetes, replaced the network layer with Flannel, and move the workload back
- Problem solved...?

Of course it wasn't solved

- While this worked much better, we eventually ran into further issues
- We moved a HTTP workload into Kubernetes and found that sometimes requests were not returning
- SYN from ingress controller to pod was getting eaten

Time for Googling

- Our search pointed to `tcp_tw_recycle` kernel setting
- Someone had thought it was wise to enable it on all of our machines
- Apparently this is a bad idea. From this link:

So in essence, use `tcp_tw_reuse` to free up sockets stuck in `TIME_WAIT`, but **stay away from `tcp_tw_recycle` as it will cause an unrelenting amount of connection problems for both administrators and end users**.

Time for Googling

- Also, that functionality is completely broken when interacting with machines on kernel 4.10 (which we had just updated to). From here:

Update (2017.09): Starting from Linux 4.10 (commit 95a22caee396), Linux will randomize timestamp offsets for each connection, **making this option completely broken**, with or without NAT.

- For that reason, the `tcp_tw_recycle` flag has been removed from the kernel as of Linux 4.12.

Reflection on problems

- We think this may have been our primary issue with Weave, but there were other open bugs that could have been the real issue
- Since Weave uses a full-mesh of TCP connections for coordination between nodes, any issues in the underlying TCP stack could have caused erratic behavior

Docker

Docker

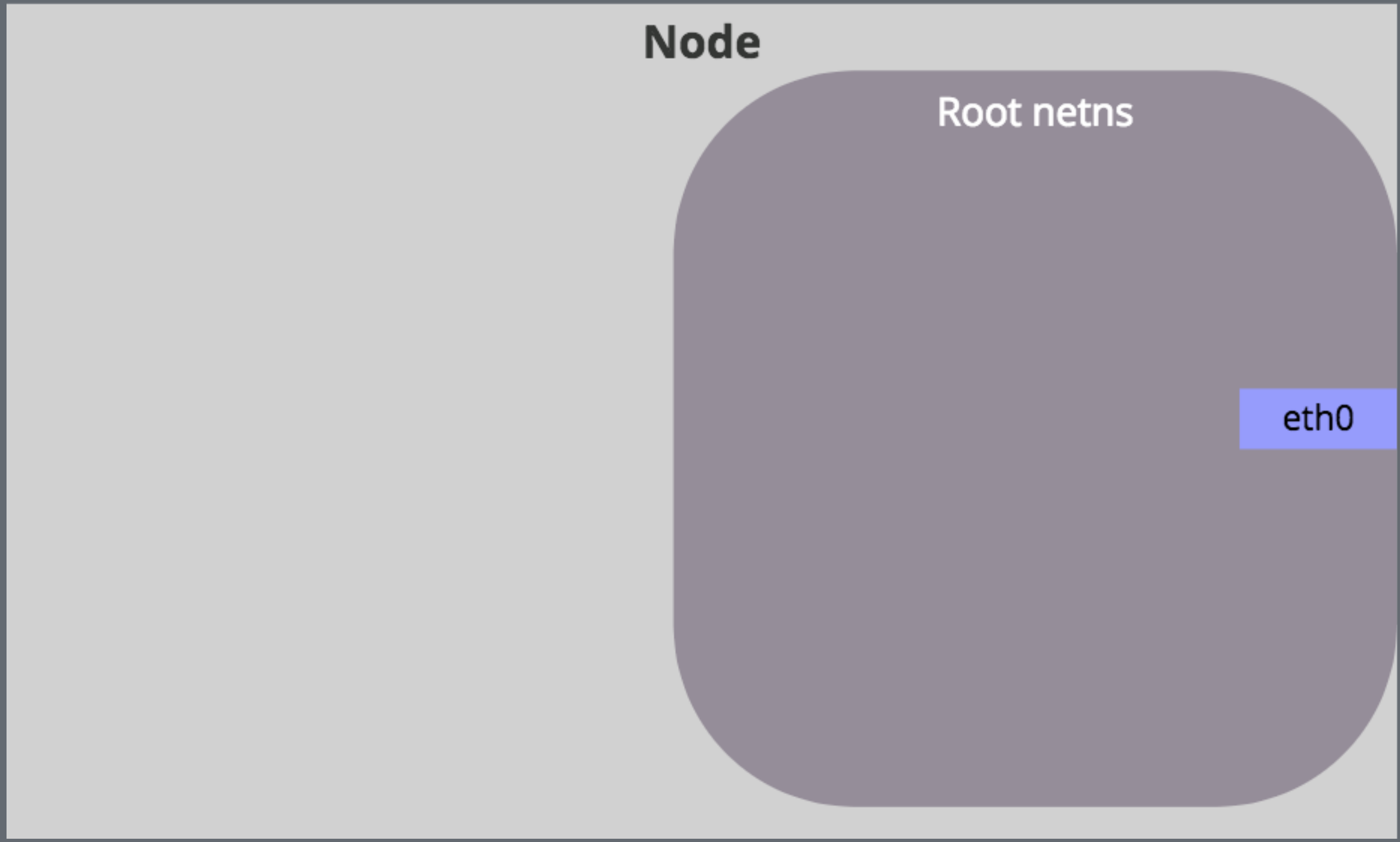
Docker allows you to run a process with various forms of isolation from the host operating system.

For our purposes, we only really care about network isolation.

Docker containers are run in a **network namespace** -- this means they have no access to the host network adapters by default.

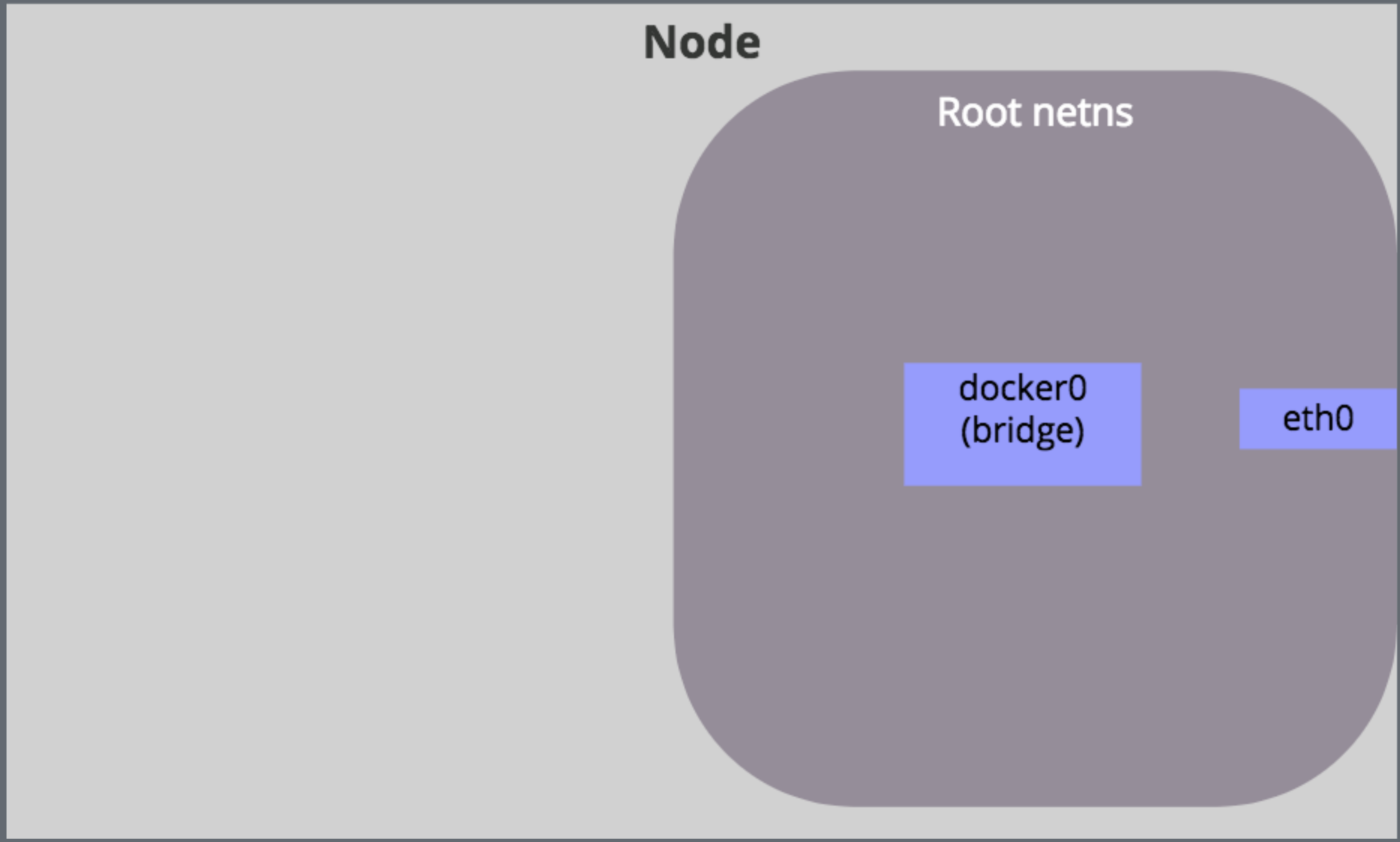
Docker bridge mode

- The standard Docker networking mode is "bridge mode"



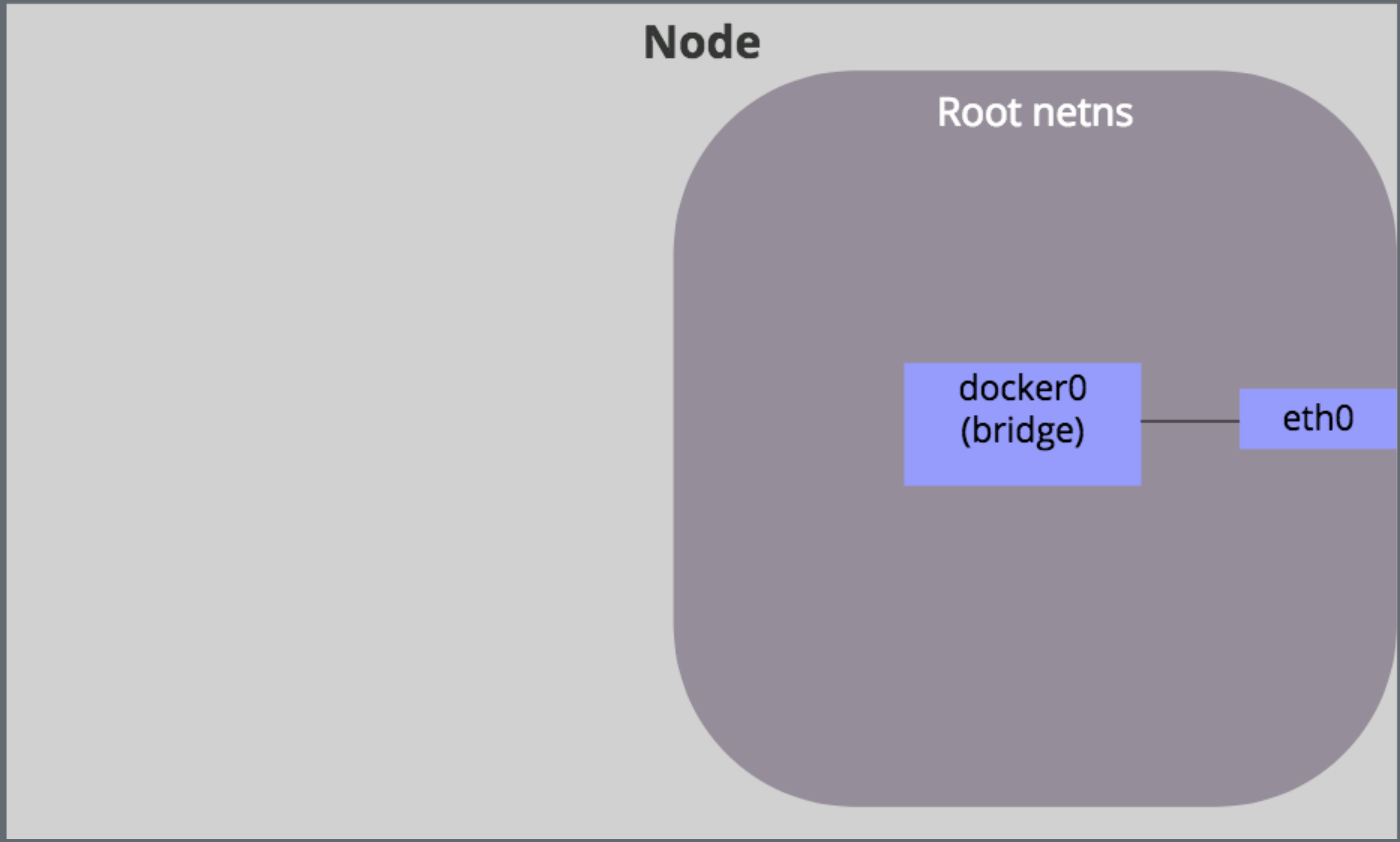
Docker bridge mode

- Docker creates a bridge device (`docker0` unless you specify otherwise) and allocates a block of IPs (172.17.0.0/16 by default)
- This bridge device operates like an Ethernet switch, running in software



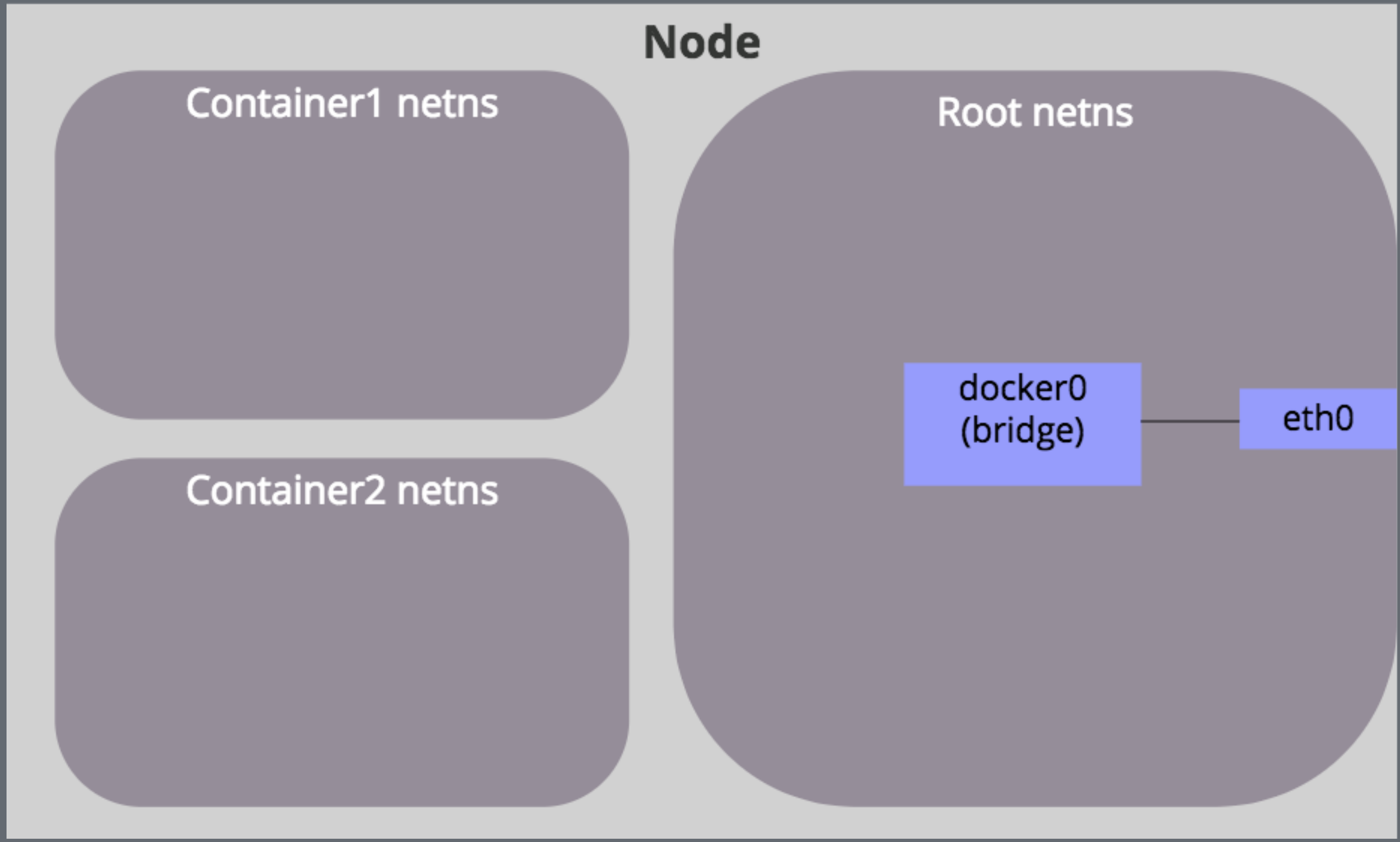
Docker bridge mode

- The bridge gets attached to the host network interface



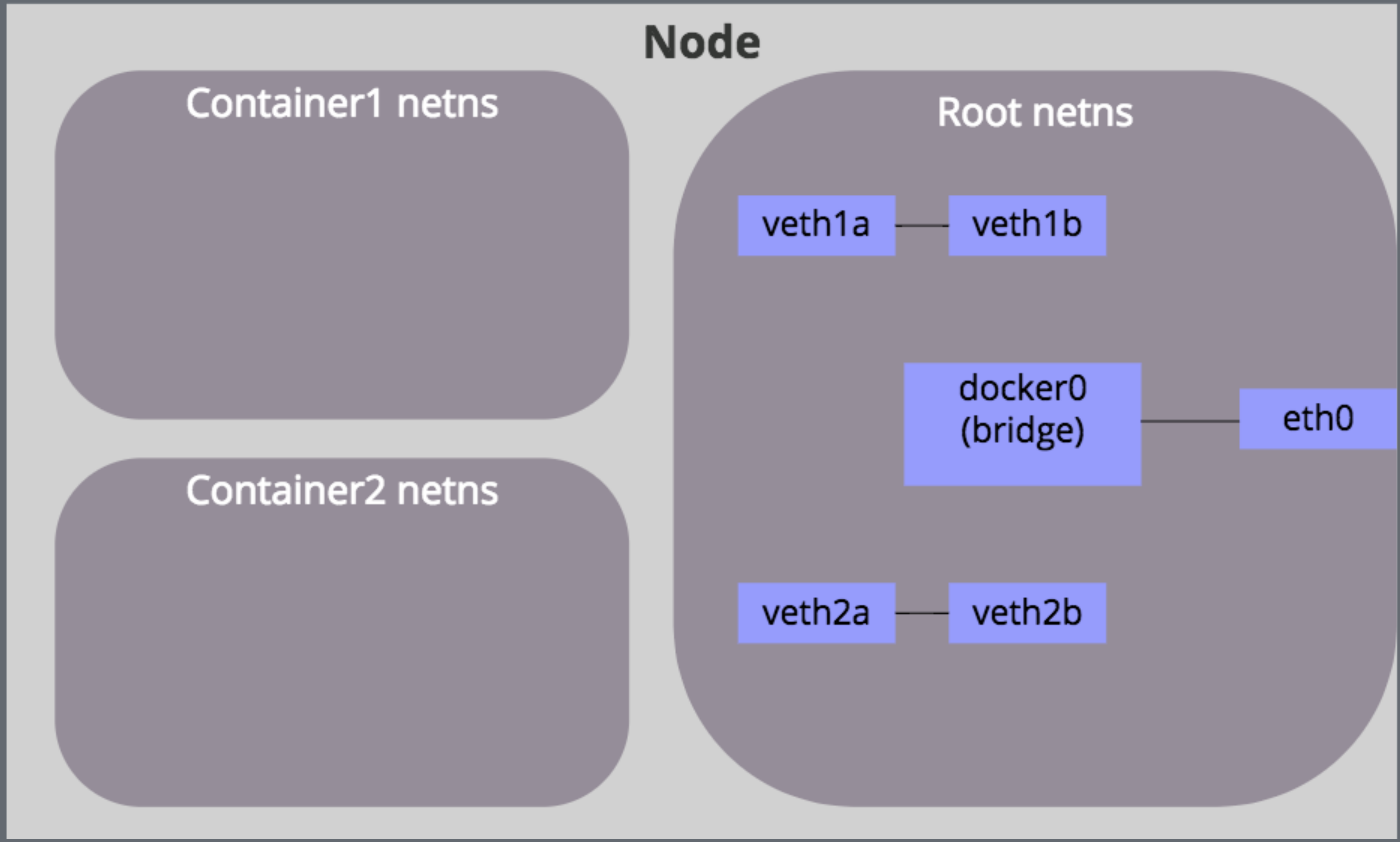
Docker bridge mode

- Docker creates a new network namespace for each container



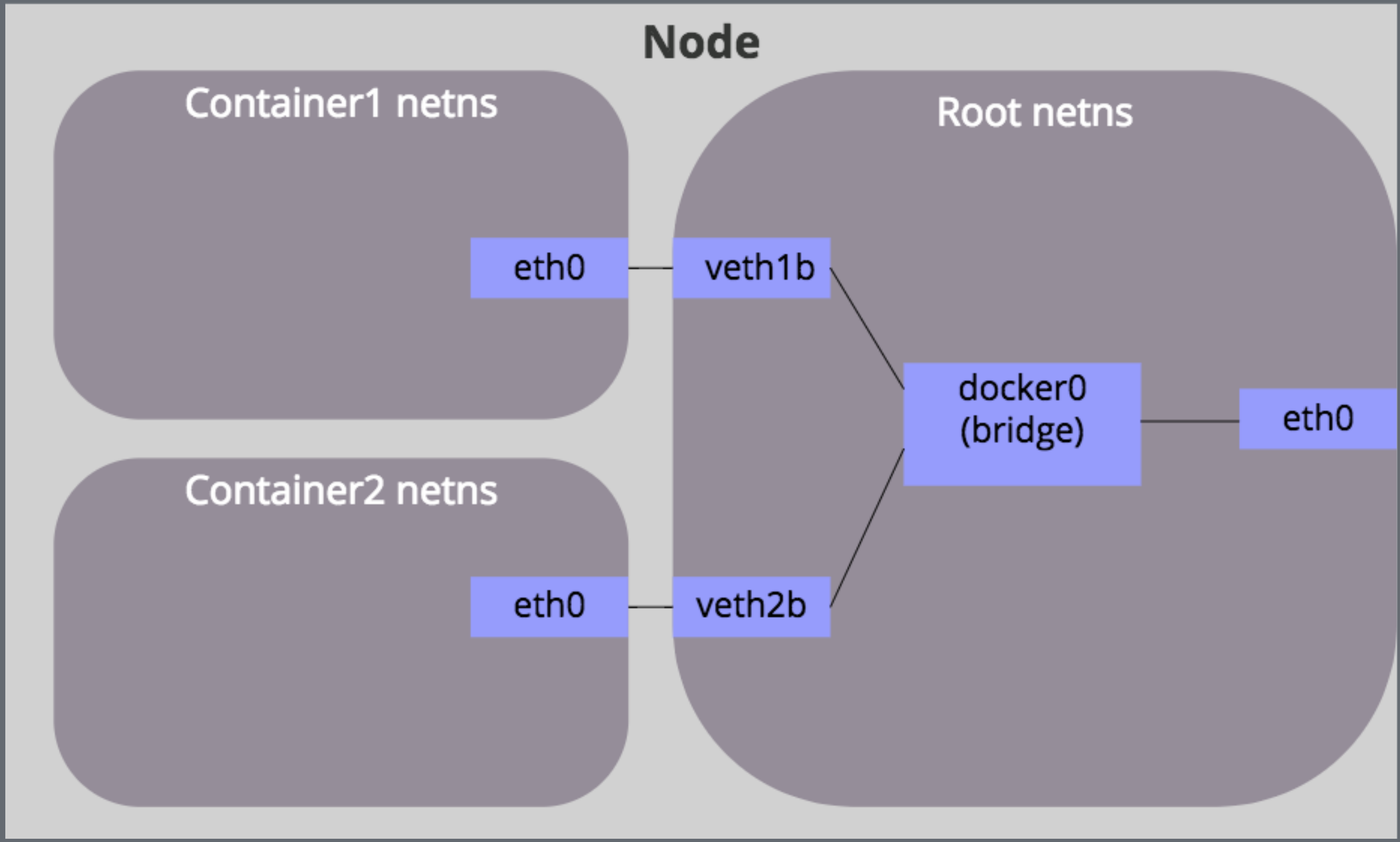
Docker bridge mode

- Docker creates a VETH pair (virtual ethernet device). This is like two network devices with a pipe between them. It attaches one of the devices to the `docker0` bridge and the other it moves into the container's network namespace and names it `eth0` within that namespace.



Docker bridge mode

- The `eth0` interface inside the container is assigned an IP from the internal IP range Docker has assigned to that bridge (`172.17.0.2-.254`).
- This allows containers to talk to each other via their private IP range.
- Docker adds an `iptables` MASQUERADE rule that allows containers to make outbound connections from the host.



Docker bridge mode

- If you want containers to be able to accept connections, there are Docker flags that will set up specific port forwards from the host to a port on a container.
- Containers on different machines can only connect to each other via these mapped ports or via manual route configuration.

Kubernetes

Kubernetes networking model

Kubernetes has the concept of **pods**

Each pod includes one or more containers, **all in the same network namespace**

This means containers within a pod must use unique ports, and can reach each other via `localhost`

Kubernetes networking model

Unlike Docker, Kubernetes expects a flat networking model.

As told in the Kubernetes docs:

- all containers can communicate with all other containers without NAT
- all nodes can communicate with all containers (and vice-versa) without NAT
- the IP that a container sees itself as is the same IP that others see it as

Kubernetes networking model

This is great for ease of use -- if you need to talk to a service, just use it's IP and port like a VM.

But it can be much more challenging to set up.

Size of available IP range, ability to assign multiple IPs to a machine, network hardware config...

Kubernetes Services

Kubernetes Services

Thanks to the flat network, if one pod knows the IP of another, they can talk directly, just like any other machines on a network.

Assuming one pod needs to find another pod, Kubernetes has the concept of **Services**.

Kubernetes Services

A service has a DNS name inside the cluster DNS:

`<myservice>.<namespace>.svc.cluster.local`

This name resolves to a `ClusterIP`, and comes from a different range than pod IPs.

These IPs don't map to anything specific, but a process on each machine (`kube-proxy`) keeps a set of `iptables` rules maintained that redirect requests sent to the `ClusterIP` randomly to IPs for the backing services.

The `ClusterIP` is used to avoid issues with DNS caching.

The `iptables` `nat` table for a service with 3 backing pods, `ClusterIP` `172.30.30.30`:

First, we have a rule that maps requests sent to the cluster IP to another chain (`KUBE-SVC-ZLJA` here).

```
Chain KUBE-SERVICES (2 references)
target         prot opt in out source          destination
KUBE-MARK-MASQ tcp  --  *  *  !172.28.0.0/16  172.30.30.30  /* ns/svc:80 cluster IP */ tcp dpt:80
KUBE-SVC-ZLJA  tcp  --  *  *   0.0.0.0/0      172.30.30.30  /* ns/svc:80 cluster IP */ tcp dpt:80
```

That chain has three rules where the only condition is a random probability.

The first rule has a 33% chance of running, the second has a 50% chance of running, and the third always runs (if you get there)

```
Chain KUBE-SVC-ZLJA (2 references)
target         prot opt in out source     destination
KUBE-SEP-6A7J all  --  *  *   0.0.0.0/0  0.0.0.0/0  /* ns/svc:80 */ statistic mode random probability 0.333
KUBE-SEP-7NAF all  --  *  *   0.0.0.0/0  0.0.0.0/0  /* ns/svc:80 */ statistic mode random probability 0.500
KUBE-SEP-QGVR all  --  *  *   0.0.0.0/0  0.0.0.0/0  /* ns/svc:80 */
```

Finally, those chains just forward the request to a destination pod, causing each pod to have a 1/3 chance of getting any request.

```
Chain KUBE-SEP-6A7J (1 references)
target         prot opt in out source         destination
KUBE-MARK-MASQ all  --  *  *   172.28.236.61  0.0.0.0/0  /* ns/svc:80 */
DNAT           tcp  --  *  *   0.0.0.0/0      0.0.0.0/0  /* ns/svc:80 */ tcp to:172.28.236.61:80
```

Debugging Kubernetes Services

Services are just a DNS name and a set of `iptables` rules

Debugging Services generally involves verifying the `iptables` rules to ensure they are getting created correctly, then treating it as a general connectivity issue

CNI

CNI

Since Kubernetes isn't opinionated on how you accomplish the goals of a flat networking model, it needs a plugable interface between Kubernetes processes and the networking layer.

Kubernetes adopted CNI (Container Network Interface), originally from CoreOS, now a CNCF[cncf] project.

[cncf] CNCF - Cloud Native Computing Foundation -- the foundation that houses Kubernetes and several related projects

CNI

The interface provided by CNI consists of two components: the IPAM plugin and the NetPlugin

The *IPAM plugin* handles IP address allocation.

The *NetPlugin* handles the network plumbing -- how communication happens in and out of the container. Some NetPlugins want to handle their own IPAM, but sometimes you can mix and match.

Underlays

Despite the common use of overlay networks, it is generally preferred to use underlay approaches since they have less overhead in terms of both:

- processing time (less for the kernel to do)
- and packet size (which can reduce the effective MTU on your network)

Using an underlay approach does require that your underlying network support the functionality you need.

Both the technologies I am going to mention generally require that all nodes be on the same layer 2 network.

MACvlan

This gives the container a virtual network adapter that has a **unique MAC address** that appears to be directly on the host network.

Some networks limit the number of distinct MAC addresses on each port (port security), and cloud provider networks usually limit the number of MAC addresses on one VM.

IPvlan

This gives the container a virtual network adapter that has a unique **IP address** that appears to be directly on the host network, **sharing a host MAC address**.

Note: Kubernetes Services can't rely on iptables if you are using a pass-through driver like MACvlan or IPvlan.

Encapsulation

Encapsulation

To get us all on the same page, let's talk about how encapsulation works in networking.

I'm going to assume we are talking about **IPv4** over **Ethernet**

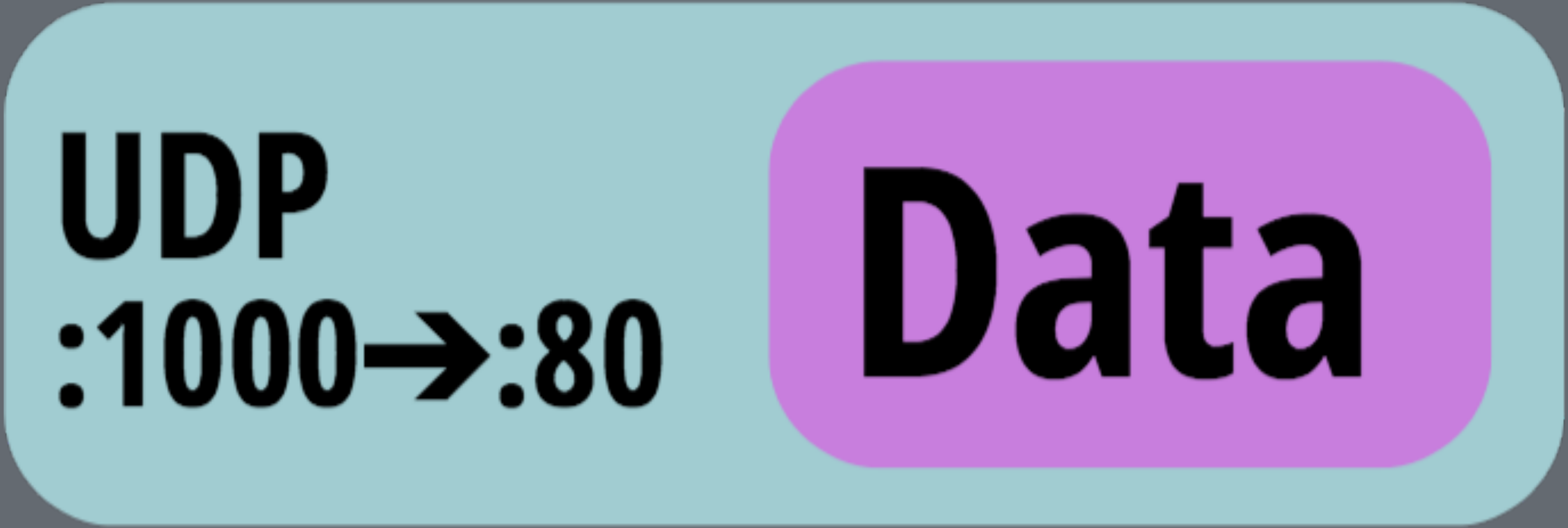
IPv6-only support was added as an *alpha* feature in Kubernetes 1.9 -- Dual-stack support is *alpha* in Kubernetes 1.11.

Encapsulation

To send a packet via, say, UDP, we get the data:

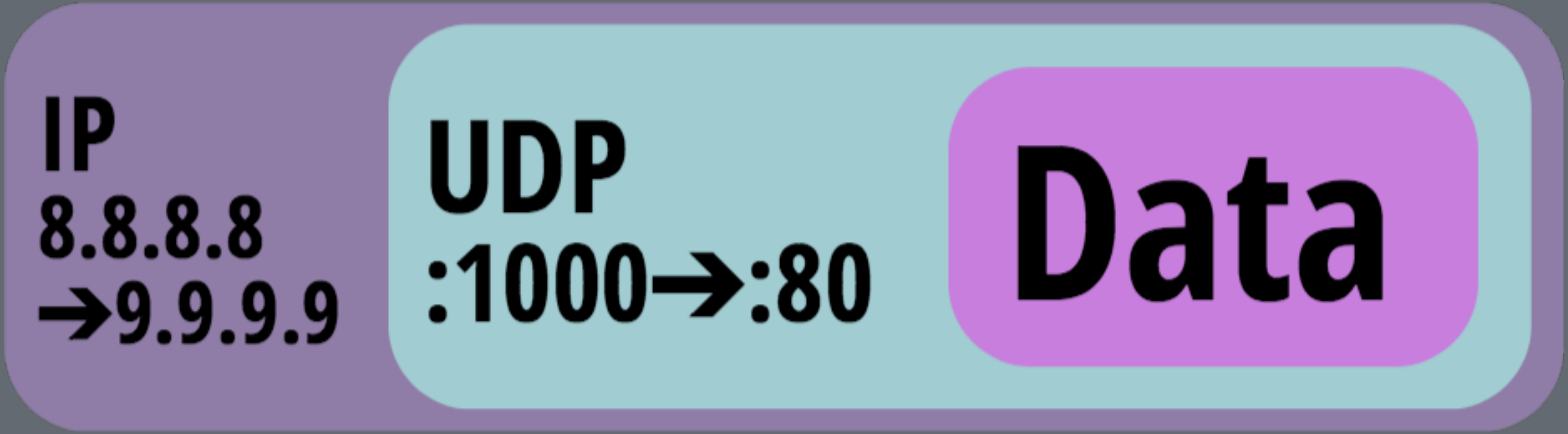
Encapsulation

We stick a (layer 4) UDP header on it:



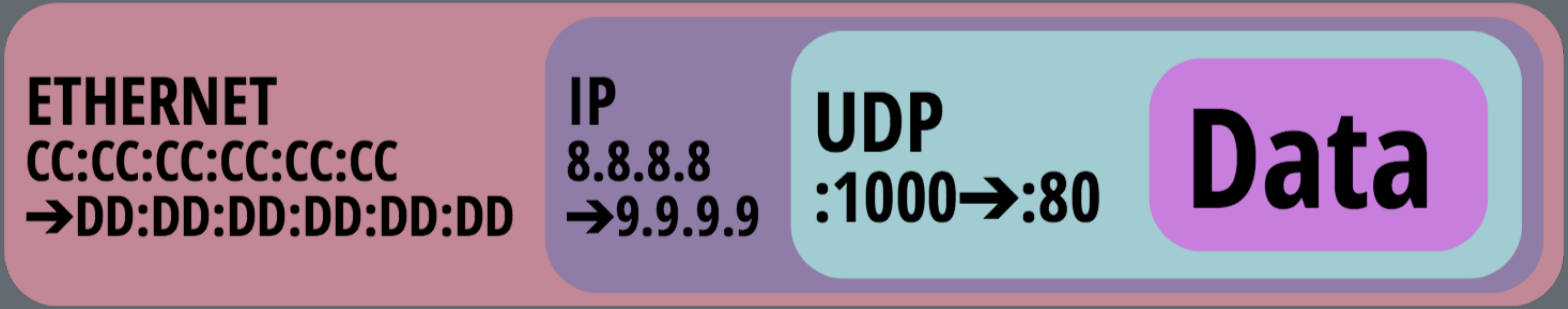
Encapsulation

We then stick a (layer 3) IP header on it:



Encapsulation

Then we stick a (layer 2) Ethernet header on top of that:



The next hop on the local network (addressed by the Ethernet header) will either:
- consume the IP packet
- replace the Ethernet header with one addressed to the next hop

Flannel

Flannel overlay

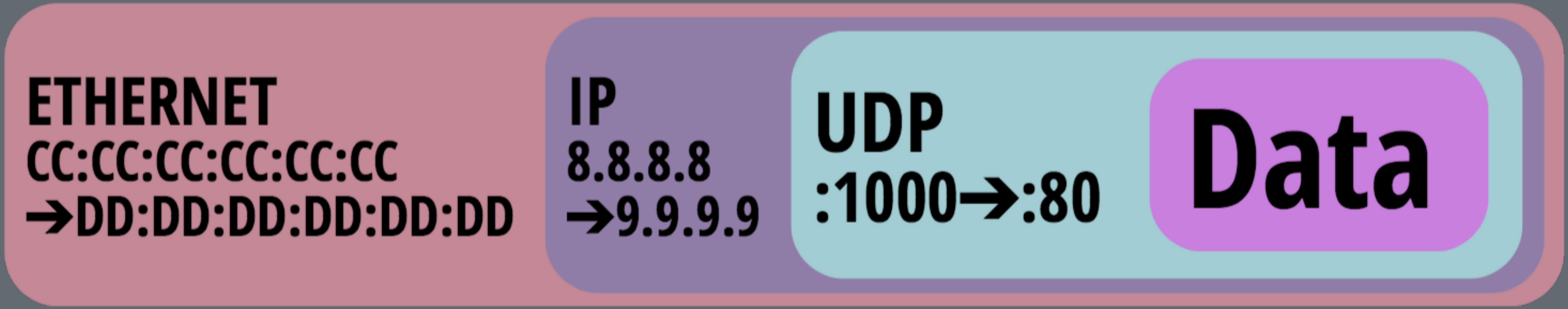
When Flannel is used with an overlay, VXLAN is the preferred technology.

It also has a `host-gw` mode where it just takes care of adding routes to hosts and doesn't provide a true overlay, and several experimental backends, though I won't cover those here.

VXLAN

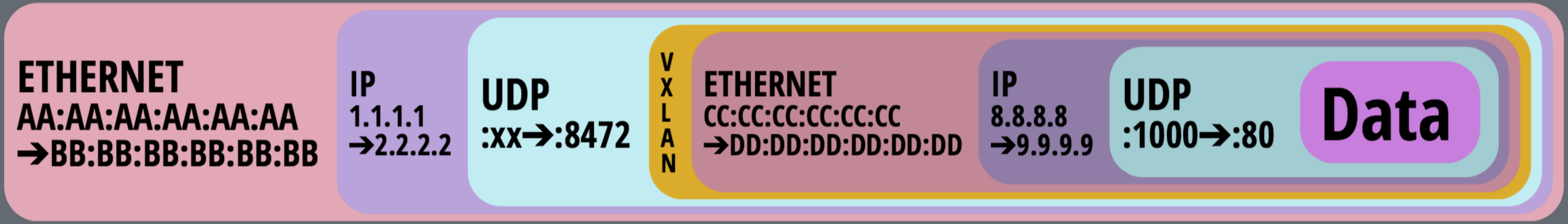
VXLAN was a technology developed for large cloud computing environments that may need more virtual L2 networks than 802.1Q VLANs can provide (VXLAN supports **16 million** unique networks), as well as providing L2 connectivity over larger IP networks.

VXLAN



Flannel's VXLAN mode encapsulates Ethernet packets in UDP packets sent to port 8472

VXLAN



Flannel's VXLAN mode encapsulates Ethernet packets in UDP packets sent to port 8472

VXLAN

But where does it address that UDP packet to?

VXLAN

Netlink is a socket family in Linux kernels that allows the transfer of network-related information between the kernel and userspace processes.

Userspace processes subscribe to a set of notifications and can respond to them. The Flannel daemon creates a Netlink socket and listens to two events -- `L2MISS` and `L3MISS`.

VXLAN

Say you are trying to reach a pod IP of `172.20.15.1`.

There will be a route on your node that says all packets destined for pod IPs go out the `flannel.1` device.

```
$ ip route
...
172.20.0.0/16 dev flannel.1
172.20.52.0/24 dev cni0 proto kernel scope link src 172.20.52.1
...
```

VXLAN

When a packet destined for `172.20.15.1` gets sent to the `flannel.1` device, if it doesn't know the Ethernet address for that IP, instead of sending an ARP request, it will send a `L3MISS` message to the Netlink socket, which will get to the `flanneld` daemon.

That daemon will look up that IP address in etcd, and respond with an `ADDL2` command to map that IP to a MAC address.

VXLAN

When the VXLAN portion of the code realizes it doesn't know where that MAC address is, it will send a `L2MISS` message to the Netlink socket, and the `flanneld` process can respond with the IP address of the node where that pod is located in an `ADDL3` message.

VXLAN

The `flanneld` logs may show something like this:

```
L3 miss: 172.20.15.1
calling NeighSet: 172.20.15.1, ee:c6:81:b1:41:4a
AddL3 succeeded

L2 miss: ee:c6:81:b1:41:4a
calling NeighAdd: 192.168.122.112, ee:c6:81:b1:41:4a
AddL2 succeeded
```

This indicates the pod IP **172.20.15.1** has a virtual MAC address of **ee:c6:81:b1:41:4a**, and that can be reached via a VXLAN tunnel to the node at IP **192.168.122.112**.

VXLAN Debugging

Problem: Pod A can't talk to Pod B

- Make sure `flanneld` is running on all nodes
- Get Pod B's IP address and the node it is on from a `kubectl get po -o wide`
- Get Pod A's node from the same command
- On the node running Pod A, verify that `ip route` shows a path out device `flannel.1` for Pod B's IP

```
# ip route |grep flannel
172.28.0.0/16 dev flannel.1
```

VXLAN Debugging

- Make sure that IP address has a valid ARP table entry and that it is associated with the `flannel.1` device

```
# ip neigh |grep 172.28.50.175
172.28.50.175 dev flannel.1 lladdr a6:43:a3:b7:4b:7a REACHABLE
```

- Make sure that MAC address has a valid entry in the FDB (and that it points to the destination node):

```
# bridge fdb show dev flannel.1 |grep a6:43:a3:b7:4b:7a
a6:43:a3:b7:4b:7a dst 172.31.13.13 self permanent
```

VXLAN Debugging

- Watch that traffic leaves the source host and arrives at the destination host (`tshark` will decode the VXLAN packets and show the underlying addresses -- add `-V` if you want a TON of detail)

  ```
  # tshark -i any -d udp.port==8472,vxlan port
  8472
  ```

VXLAN Debugging

Problem: Pod A *sometimes* can't talk to Pod B (or arbitrary other addresses)

Remember VXLAN creates one large L2 network.

Check syslog for messages that look like:

```
kernel: Neighbour table overflow
```

VXLAN Debugging

If you see that, try increasing the ARP table limits. Defaults on most systems look like:

```
# sysctl -a |grep net.ipv4.neigh.default.gc_thresh
net.ipv4.neigh.default.gc_thresh1 = 128
net.ipv4.neigh.default.gc_thresh2 = 512
net.ipv4.neigh.default.gc_thresh3 = 1024
```

If you have anything close to 1000 pods, that's going to be a problem since you will have other hosts to put in there as well. Make sure your maximum number of pods is well below `gc_thresh2` (the soft maximum).

VXLAN Debugging

Not specific to VXLAN, but for intermittent connectivity problems (especially if there are signs of failed DNS lookups), consider if you have enough DNS pods running.

Kubernetes can put a lot of load on the internal DNS pods, possibly doing several lookups for each connection in the cluster

Calico

Calico

Calico's preferred overlay is IP-in-IP (RFC 2003).

IP-in-IP

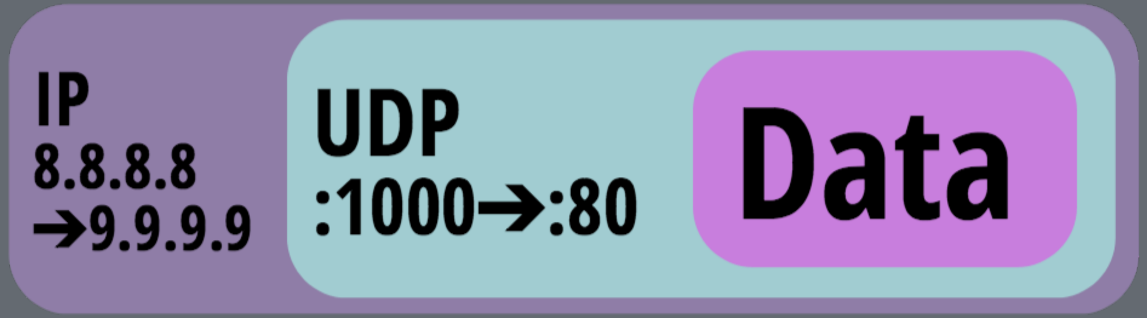
IP-in-IP is a simple encapsulation protocol at layer 3.

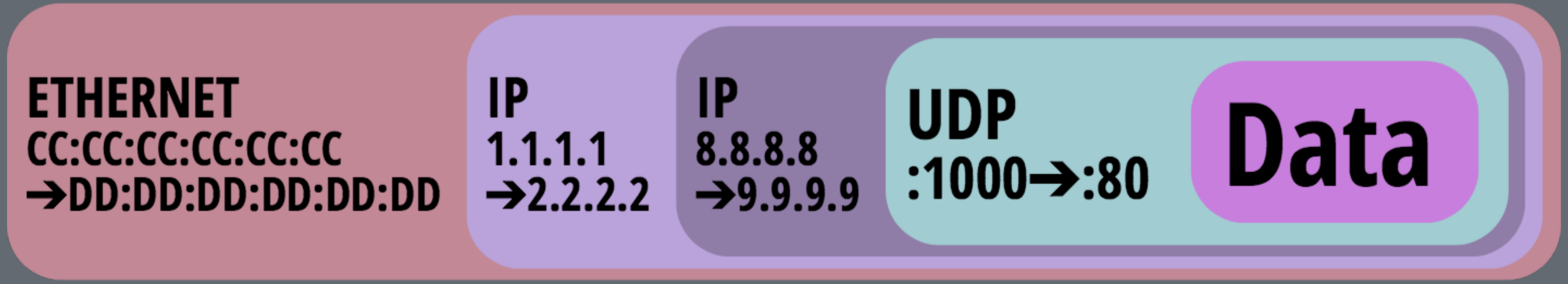
This means it encapsulates IP packets, not Ethernet packets, which means you can only run protocols over it that use unicast IP (HTTP - yes, DHCP - no, and nothing that uses multicast).

Only change to inner IP packet is to decrement the TTL by one, like a router would.

In the Linux kernel, this mode (`ipip`) only supports IPv4 over IPv4.

IP-in-IP


IP
8.8.8.8
➔9.9.9.9
UDP
:1000➔:80
Data


ETHERNET
CC:CC:CC:CC:CC:CC
➔DD:DD:DD:DD:DD:DD
IP
1.1.1.1
➔2.2.2.2
IP
8.8.8.8
➔9.9.9.9
UDP
:1000➔:80
Data

IP-in-IP Setup

Calico will create an IP-in-IP tunnel device in the root network namespace

```
tunl0@NONE: <NOARP,UP,LOWER_UP> mtu 1440 qdisc noqueue state UNKNOWN qlen 1000
link/ipip 0.0.0.0 brd 0.0.0.0
inet 172.28.69.0/32 scope global tunl0
   valid_lft forever preferred_lft forever
```

IP-in-IP Container Setup

When a pod is created, the kernel will insert a `tunl0` interface (not used) and Calico will insert a VETH interface called `eth0`, assigning it an IP from one of the `/26` networks that node has reserved:

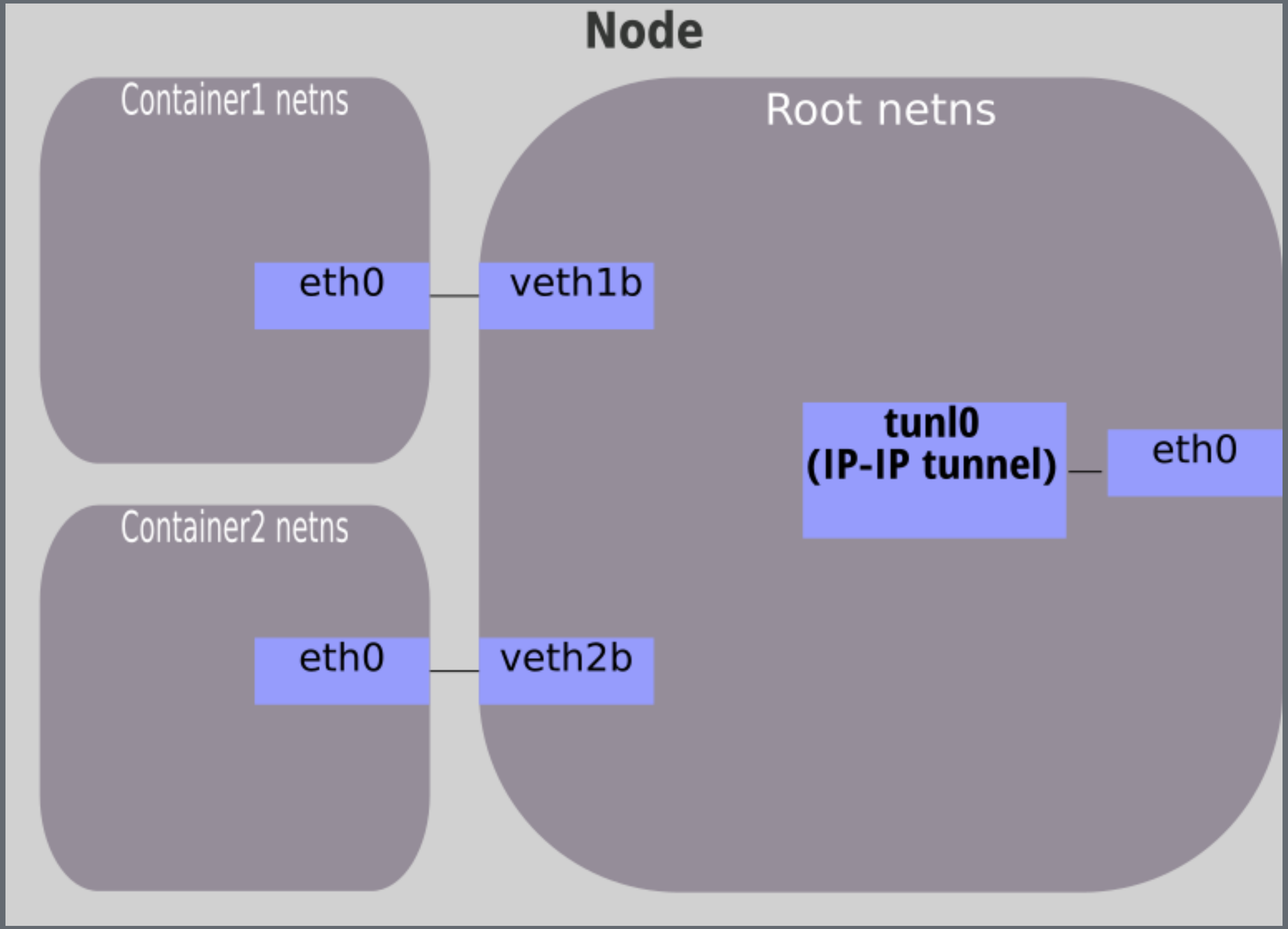
```
# PID=$(docker inspect -f '{{.State.Pid}}' $DOCKER_CONTAINER_ID)
# nsenter -t $PID -n ip addr show dev eth0
4: eth0@if1734: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether fa:5e:b3:e6:2f:ef brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.28.69.26/32 scope global eth0
       valid_lft forever preferred_lft forever
```

IP-in-IP Container Setup

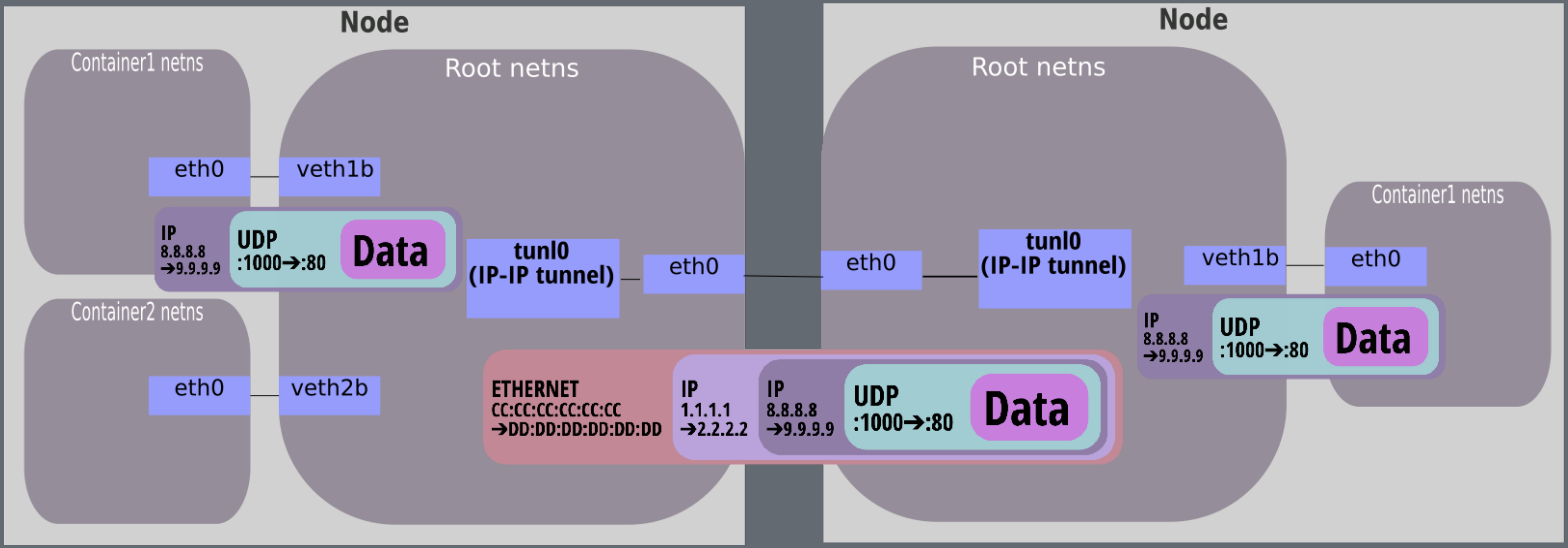
And the root namespace will have a route to that VETH interface

```
# ip addr
...
1734: calid90f1440737@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
    link/ether ee:ee:ee:ee:ee:ee brd ff:ff:ff:ff:ff:ff link-netnsid 14
...
# ip route |grep calid90f1440737
172.28.69.26 dev calid90f1440737 scope link
```

IP-in-IP Container Setup



IP-in-IP Encapsulation



IP-in-IP Routing

When a pod wants to send a message to a pod on a different node, there needs to be a route in the local route table for the node IP. It will look something like this:

```
172.28.52.192/26 via 10.1.32.47 dev tunl0 proto bird onlink
```

Those routes are advertised via BGP (Border Gateway Protocol), with every node running a BGP daemon (`bird`) that advertises routes to every other node.

Calico Debugging

Problem: Pods are having trouble getting IP addresses

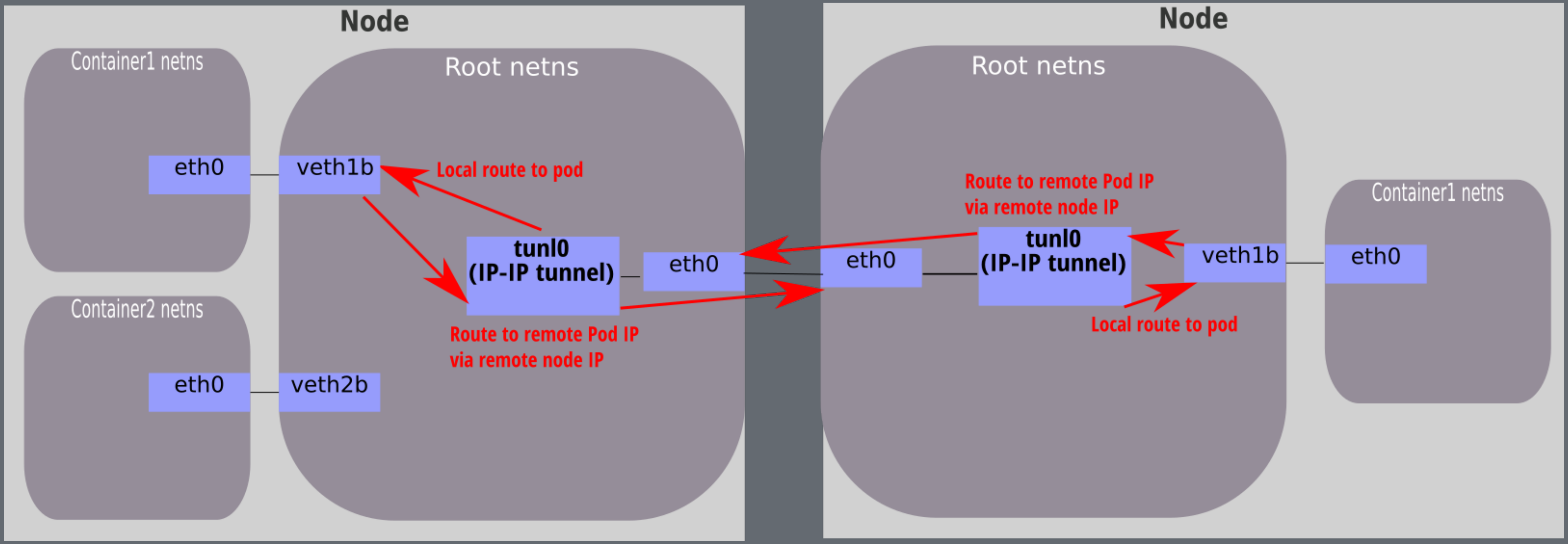
- Make sure the `CALICO_IPV4POOL_CIDR` environment variable is correct on the `calico-node` DaemonSet and verify it matches the `cidr` setting in the output of `calicoctl get ippool -o yaml`
- Make sure there are `calico-node` pods running on every Kubernetes node, and there is a `calico-kube-controllers` pod running (also check this if network policies aren't being applied properly)
- Check logs of both `kubelet` and `calico-node` processes when trying to create a pod
- Make sure `calico-node` can talk to the `etcd` cluster for Calico

Calico Debugging

Problem: PodA can't talk to PodB

- Check the routes in both directions

- Run `ip route` on the node with PodA.
- Is there a route to PodB's IP via it's node, pointing to `tunl0`? Is there a route to PodA's IP pointing to a `cali...` interface?
- Check the opposite routes on the node with PodB



Calico Debugging

- If the routes are missing, make sure `bird` is running on both nodes
- Run `tcpdump -i any port 179` to see if BGP traffic is flowing between the two nodes.
- If you run `ip addr` in the pod namespace, the `eth0` adapter will have a name like `eth0@if1714` -- look for a matching interface in the root netns:

```
1714: caliba7abdd03e5@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> ...
```

- Watch the pod traffic with `tcpdump -i caliba7abdd03e5@if4`

Summary

Summary

- Learn how the networks work
- Learn from others mistakes
- Be prepared for the networking layer to let you down

Thanks!

Contact:

Twitter: @_JeffPoole
Email: jeff@jeffpoole.net

Slides: http://jeffpoole.net/talks/2018/k8s-networking.pdf