September 25 - 27, 2018
Amsterdam, The Netherlands

ons
EUROPE

OPEN NETWORKING //
Integrate, Automate, Accelerate

ONS EUROPE

OPEN NETWORKING // Integrate, Automate, Accelerate

September 25 - 27, 2018
Amsterdam, The Netherlands

Kubernetes Networking Made Easy with Open vSwitch and OpenFlow

Péter Megyesi
Co-founder @ LeanNet ltd.

Who Am I?

PhD in Telecommunications @ Budapest University of Technology

- Measurement and monitoring in Software Defined Networks
- Participating in 5G-PPP EU projects
- Graduated in the EIT Digital Doctoral School

Co-founder & CTO @ LeanNet Ltd.

- Evangelist of open networking solutions
- Currently focusing on SDN in cloud native environments

megyesi@leannet.eu

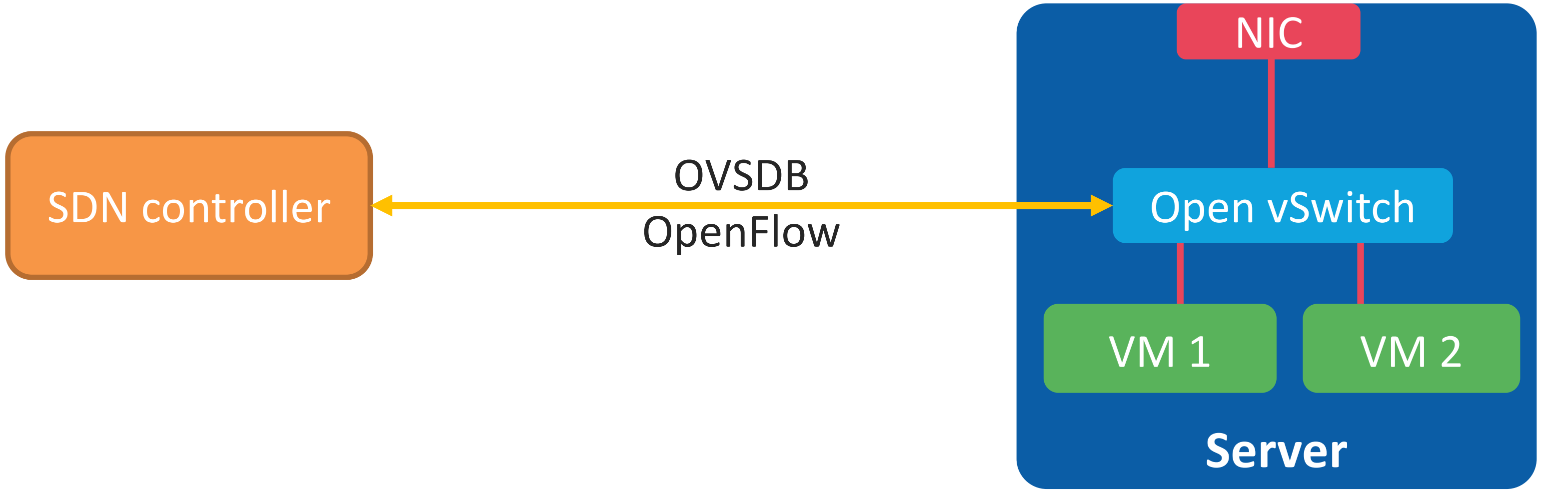
twitter.com/M3gy0

linkedin.com/in/M3gy0

What is Open vSwitch?

The de facto production quality, multilayer virtual switch

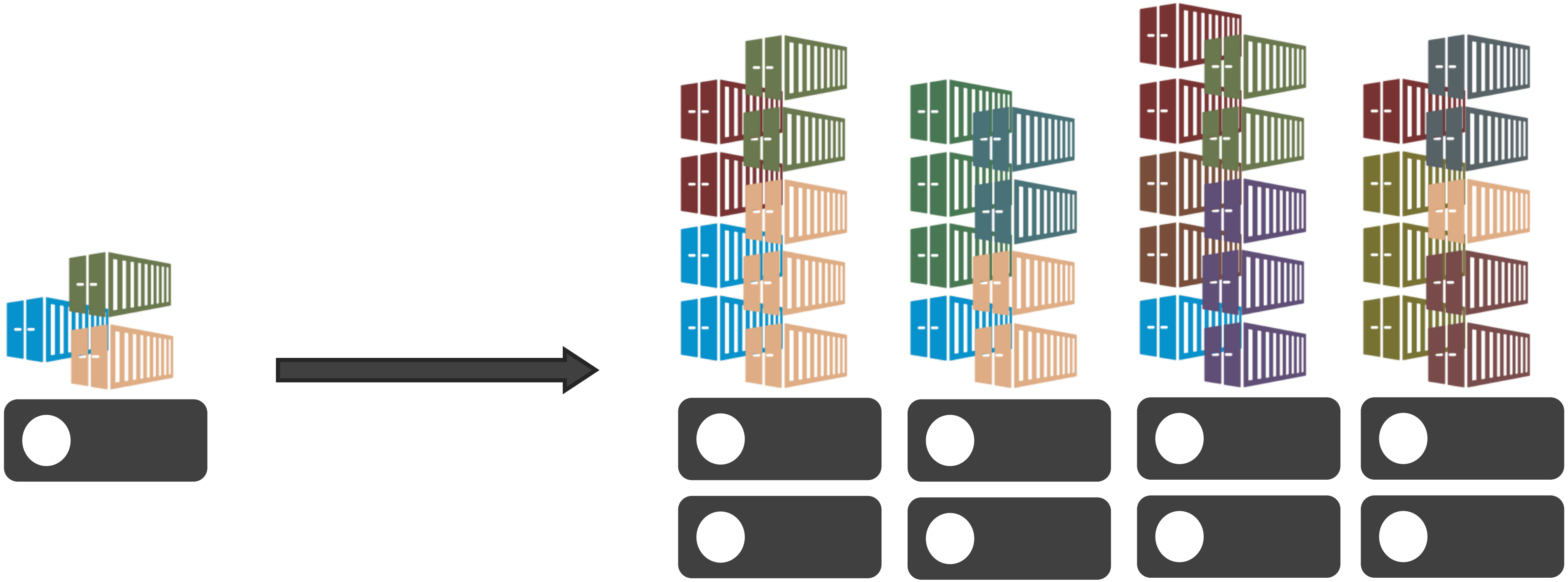
- Originally developed by Nicira (the inventors of SDN and OpenFlow)
- Now it's developed under the Linux Foundation
- Designed to be programmable by OVSDB and OpenFlow
- Compatible with standard management interfaces (NetFlow, sFlow, IPFIX, RSPAN, LACP)
- The basis of VMware NSX-T, OpenStack and many other public clouds…
- Able to run in user-space mode via DPDK, thus can provide speed up to ~80 Gbps



What is Kubernetes?

The de facto production quality, container-orchestration framework

- Originally developed by Google (Borg project)
- Now maintained by the Cloud Native Compute Foundation
- Automating deployment, scaling, and management of containerized applications



Basic Kubernetes Terminology

Kubernetes Master

- Controller of a Kubernetes cluster

Kubernetes Node (Worker / Minion)

- Hosts (server or VM) that run Kubernetes applications

Container

- Unit of packaging

Pod

- Unit of deployment

Labels and Selectors

- Key-Value pairs for identification

Replication Set

- Ensures availability and scalability

Services

- Collection of pods exposed as an endpoint

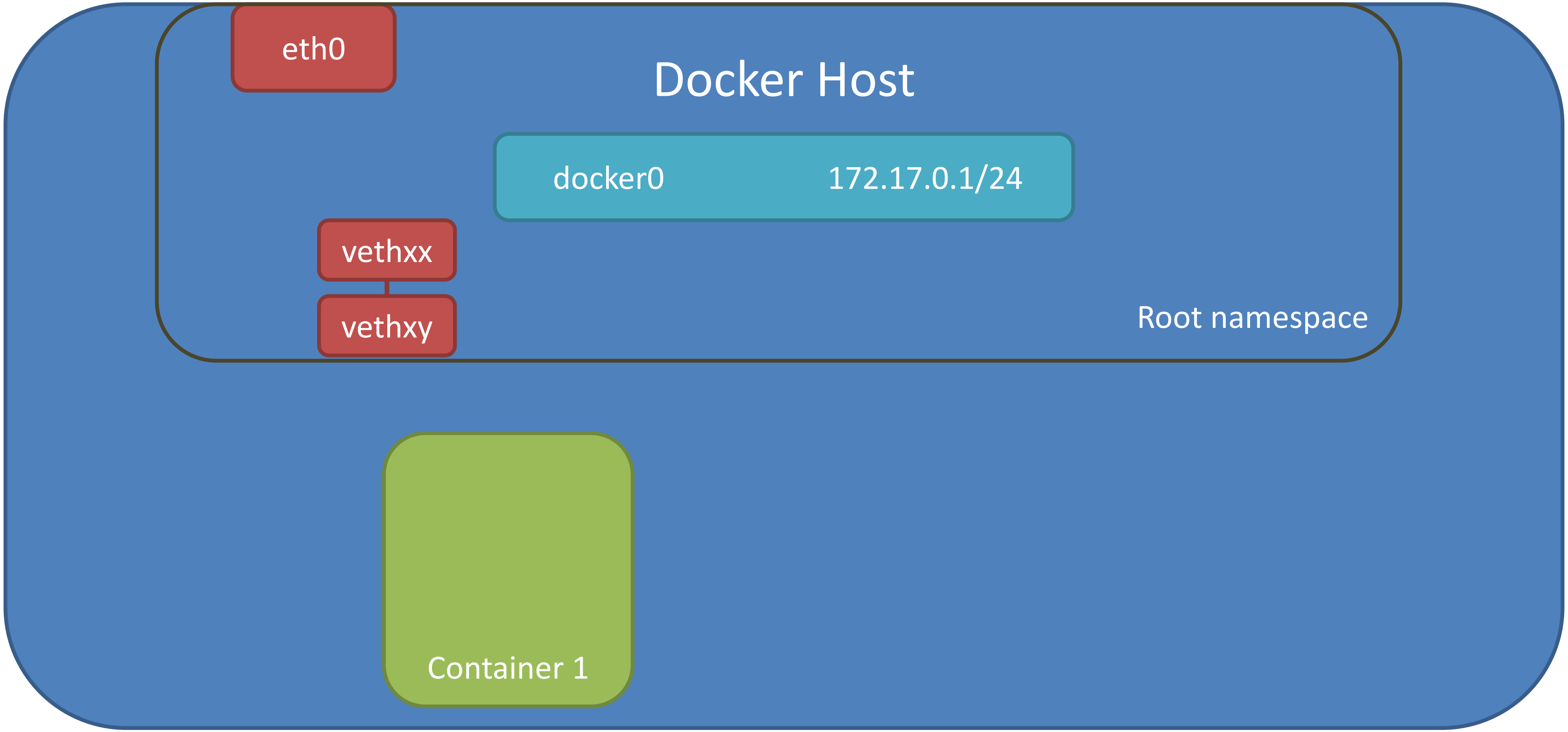
Node Port

- Expose services internally

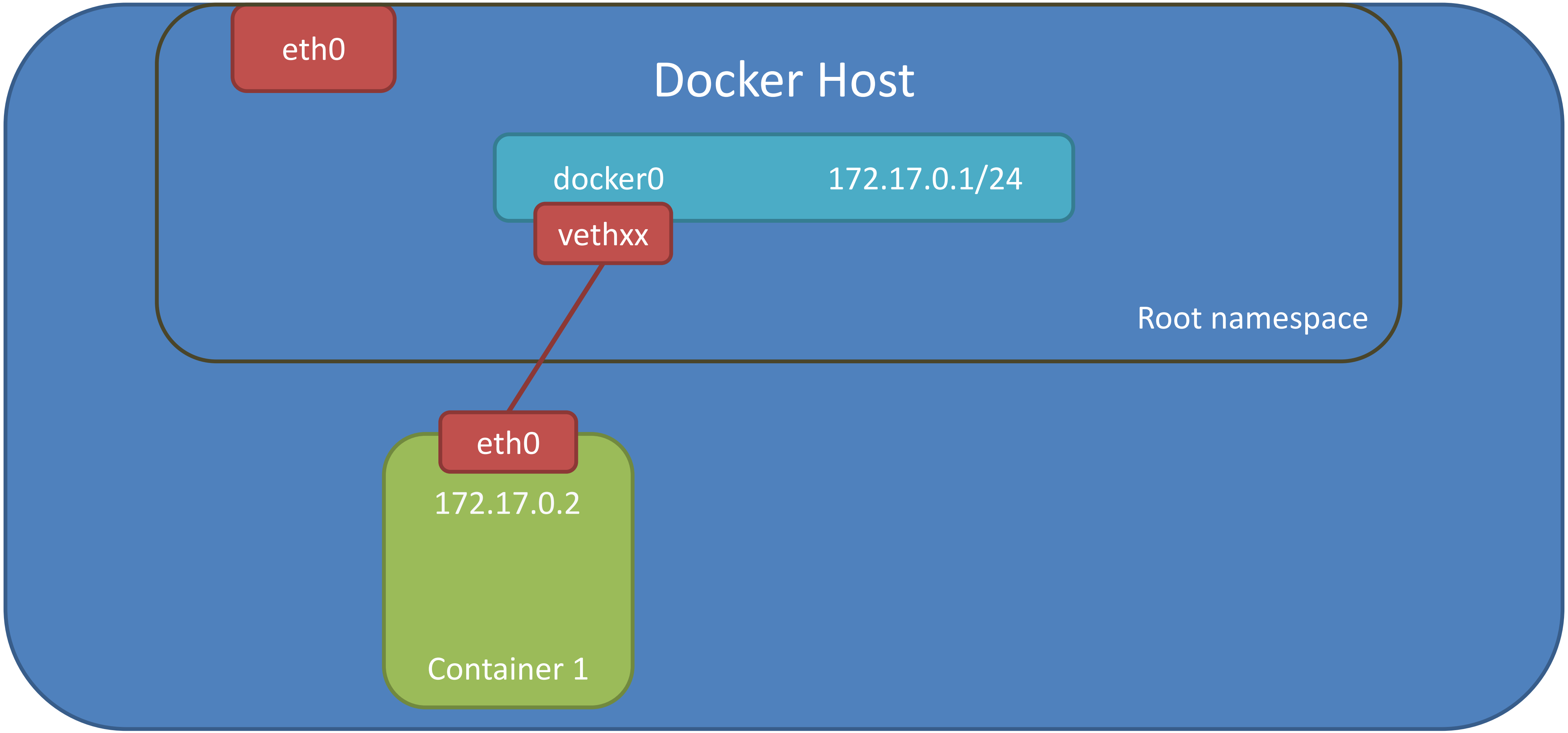
Load Balancer

- The way for external access

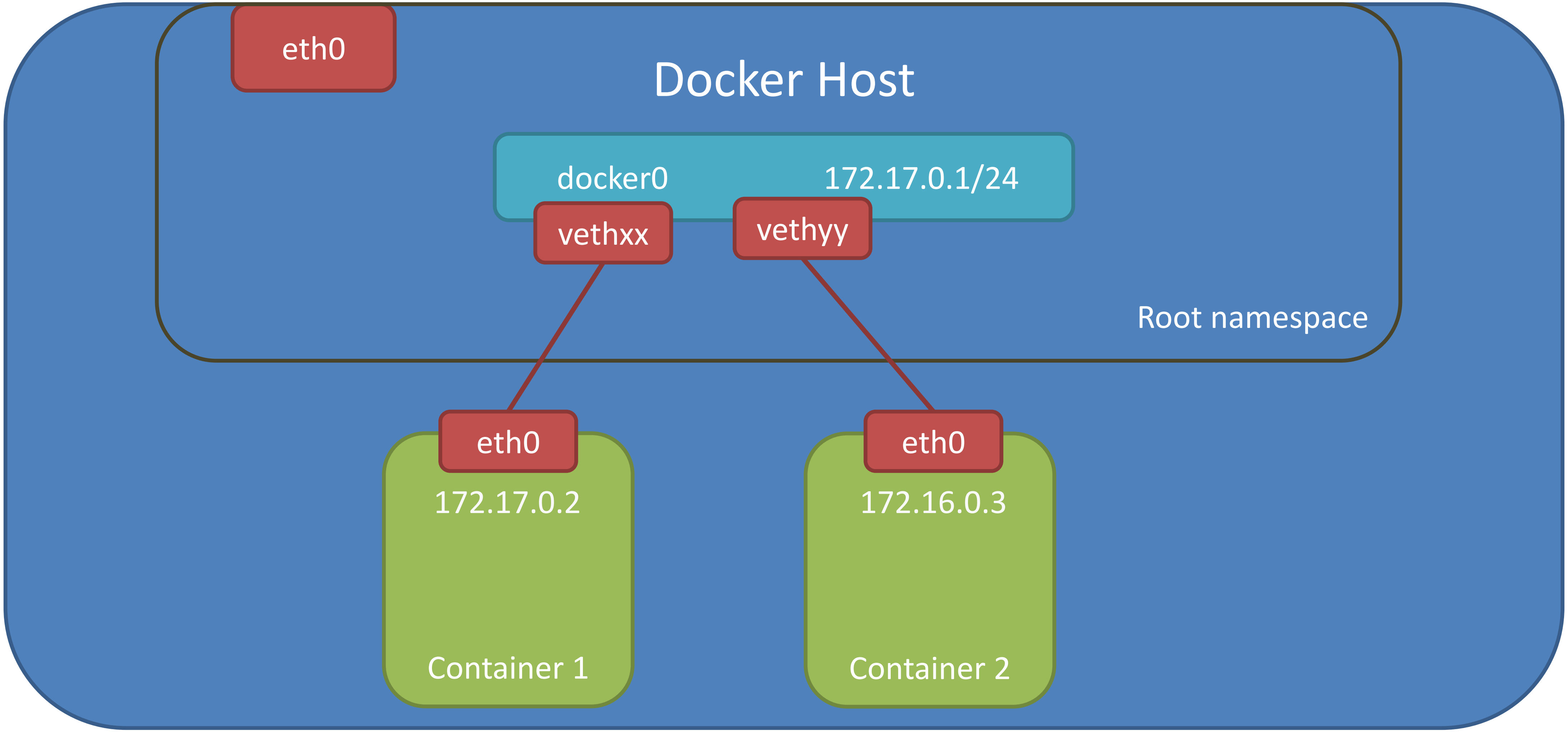
The Docker Model



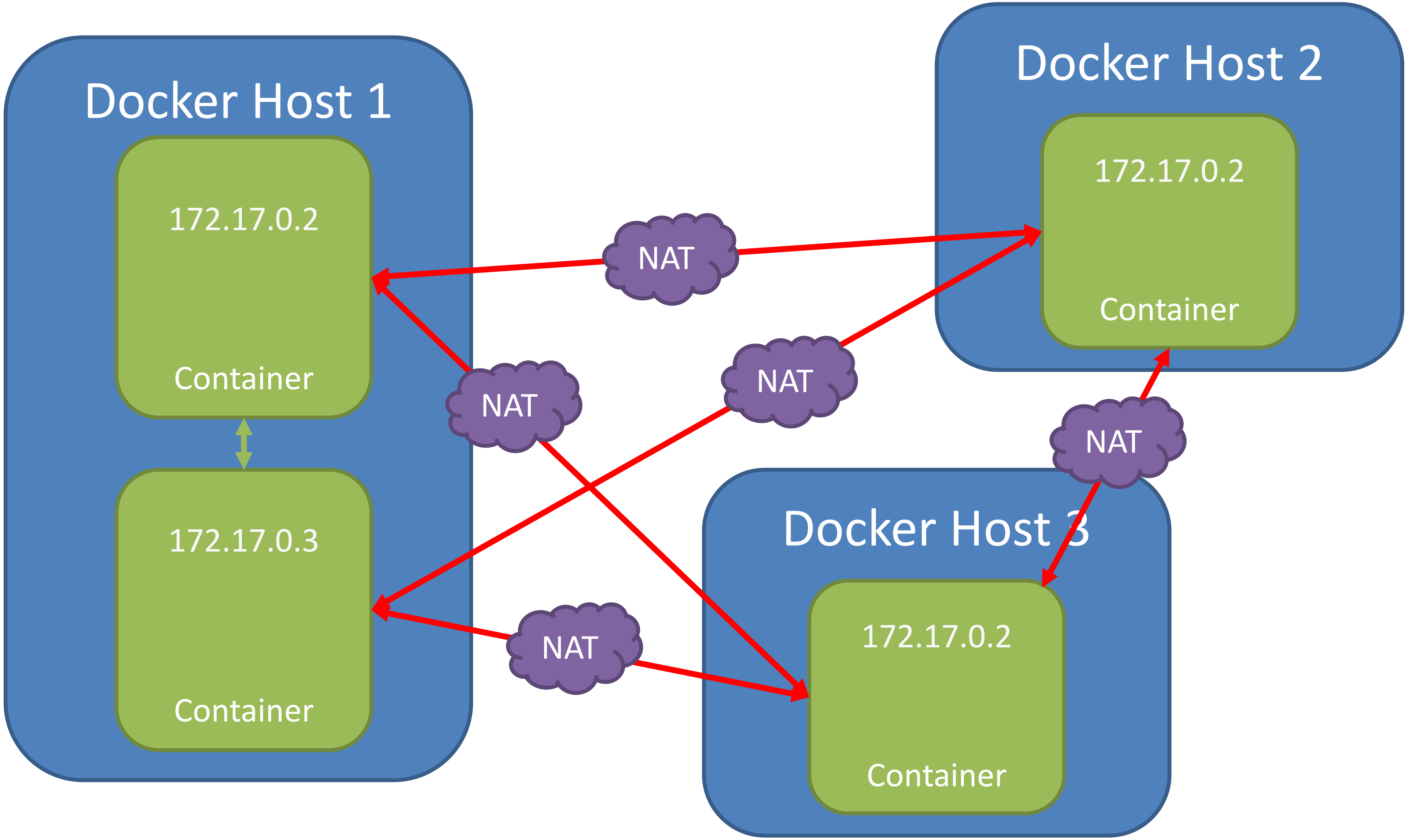
The Docker Model



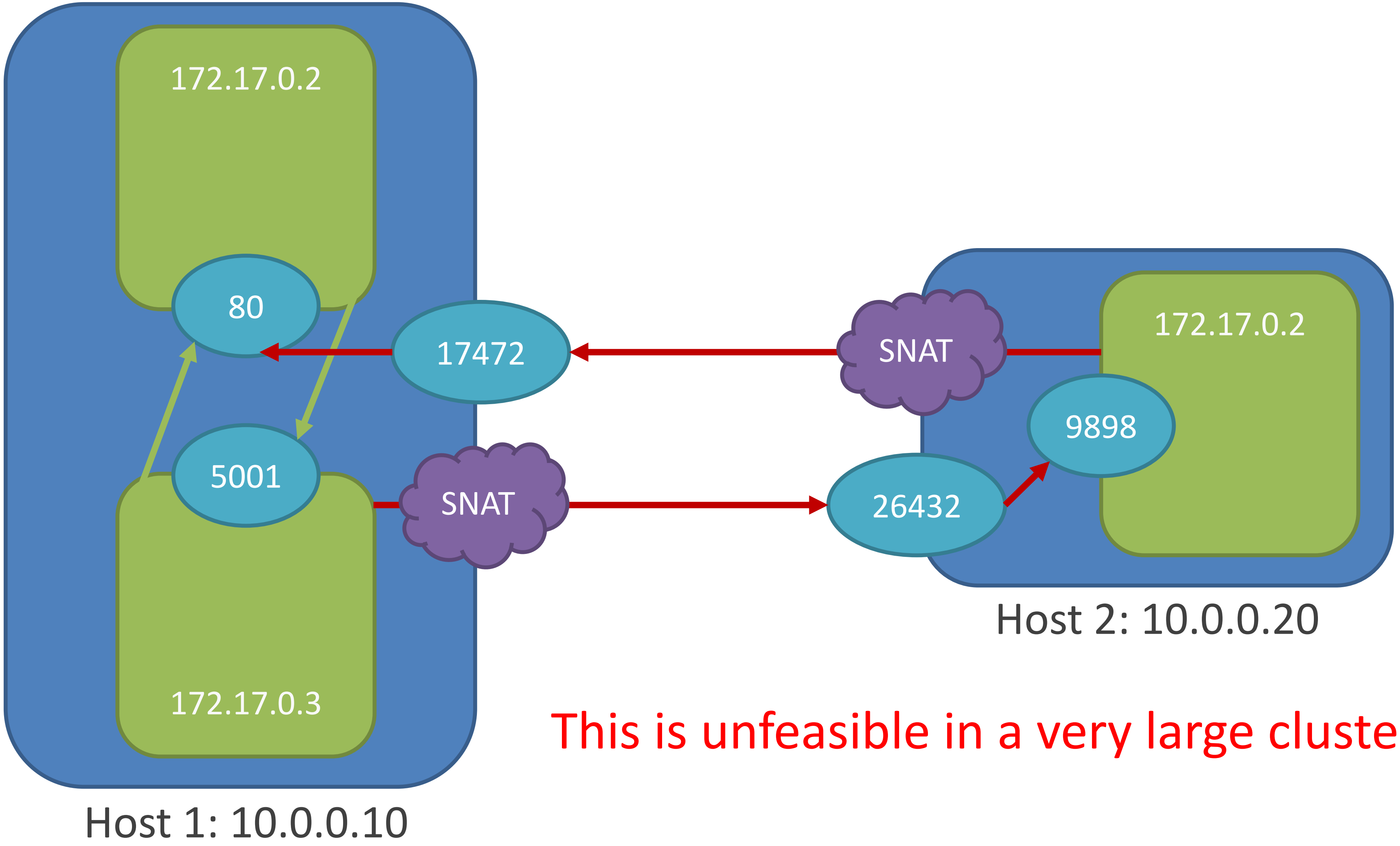
The Docker Model



The Docker Model



Docker Host Ports



This is unfeasible in a very large cluster!

Networking in Kubernetes

Pod-to-Pod communication

- Each Pod in a Kubernetes cluster is assigned an IP in a flat shared networking namespace
- All PODs can communicate with all other PODs without NAT
- The IP that a PODs sees itself as is the same IP that others see it as

Pod-to-Service communication

- Requests to the Service IPs are intercepted by a Kube-proxy process running on all hosts
- Kube-proxy is then responsible for routing to the correct POD

External-to-Internal communication

- All nodes can communicate with all PODs (and vice-versa) without NAT
- Node ports are can be assigned to a service on every Kuberentes host
- Public IPs can be implemented by configuring external Load Balancers which target all nodes in the cluster
- Once traffic arrives at a node, it is routed to the correct Service backends by Kube-proxy

The Container Network Interface

CNI in Kubernetes

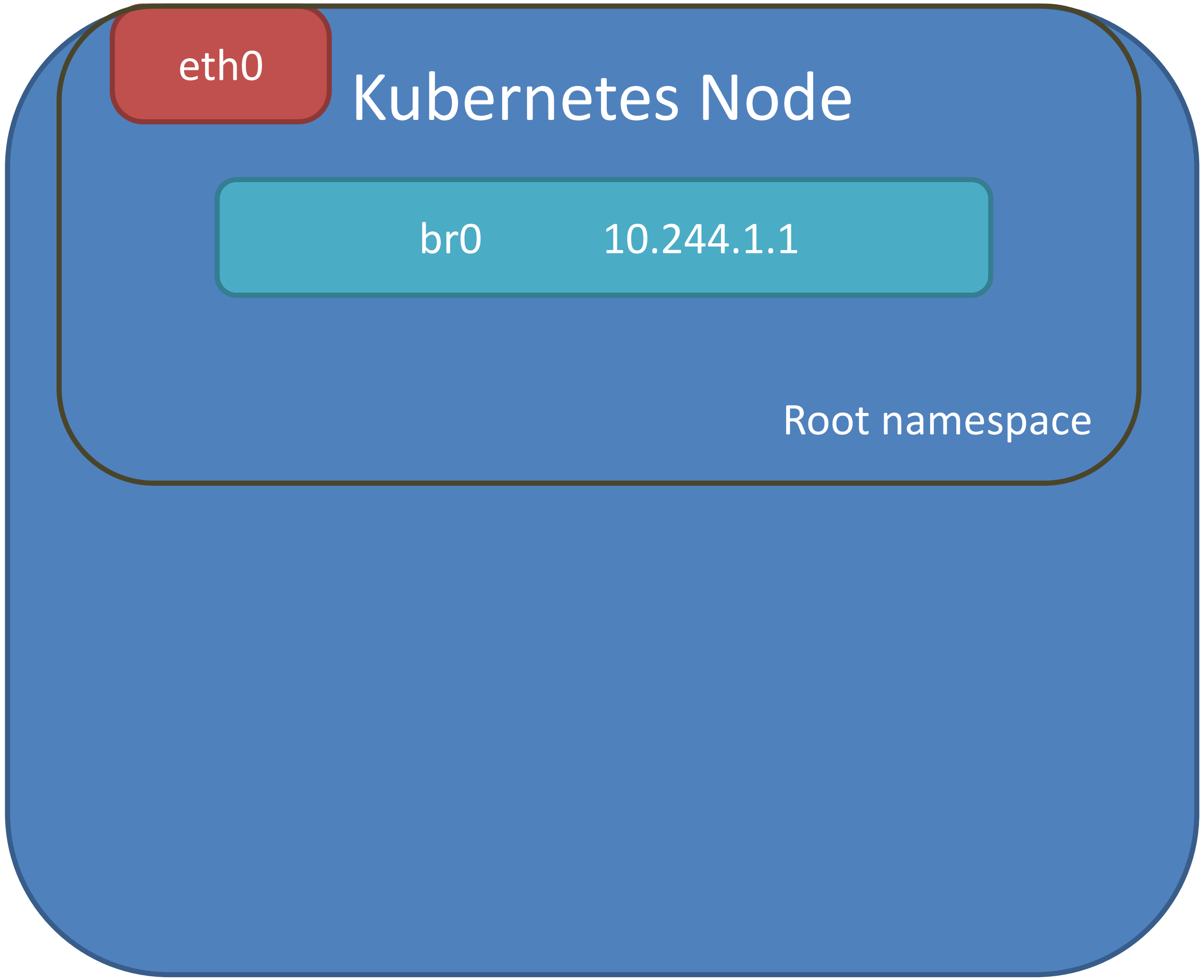
Script / binary placed on every host

- Kubelet calls it with the right environmental variables and STDIN parameters

Example for configuration

- /etc/cni/net.d/01-dunlin.conf

```
{
    "cniVersion": "0.2.0",
    "name": "dunlin",
    "type": "ovs_cni",
    "bridge": "br0",
    "isGateway": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.244.1.0/24",
        "gateway": "10.244.1.1"
    }
}
```



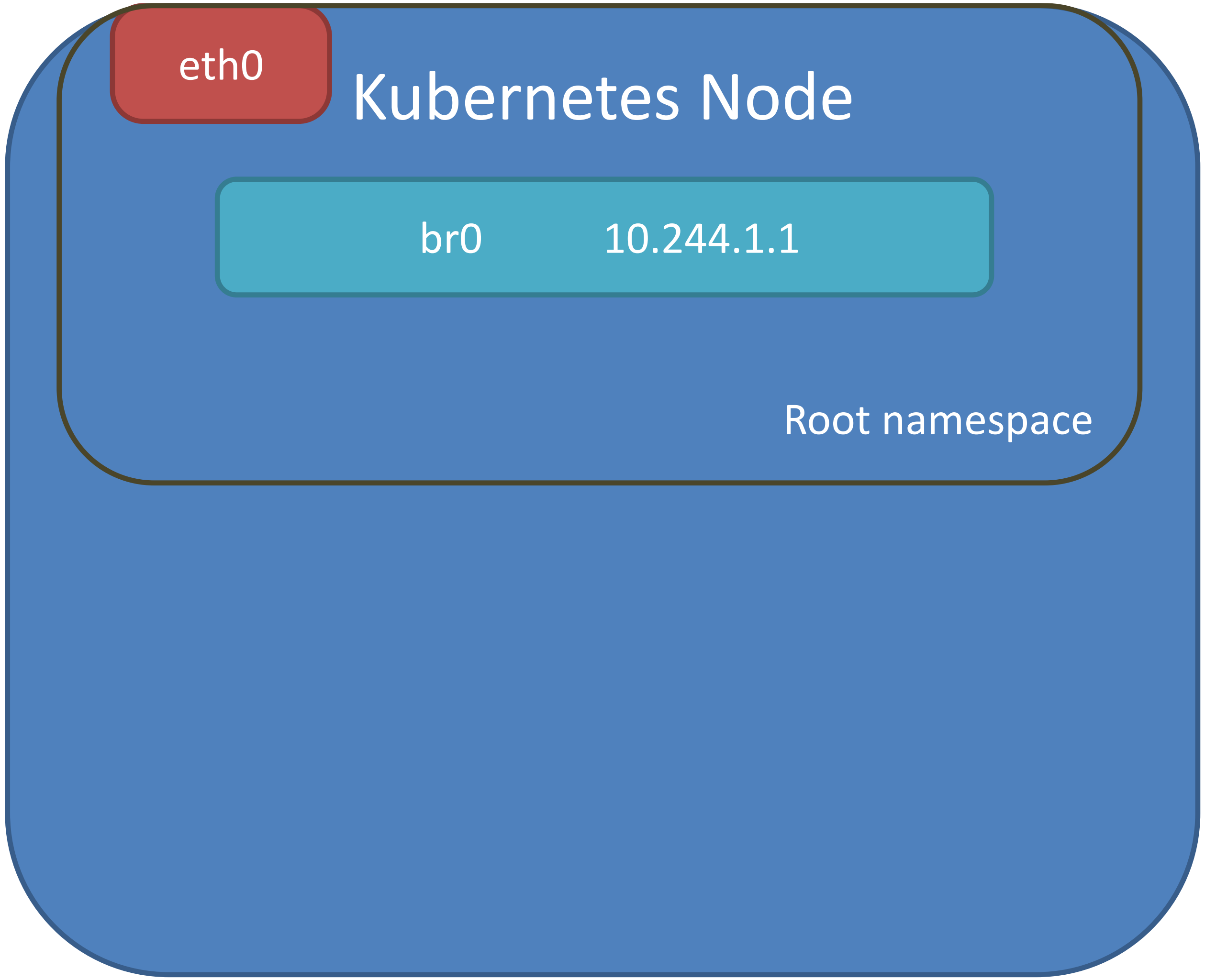
CNI in Kubernetes

Script / binary placed on every host

- Kubelet calls it with the right environmental variables and STDIN parameters

Example environment variables

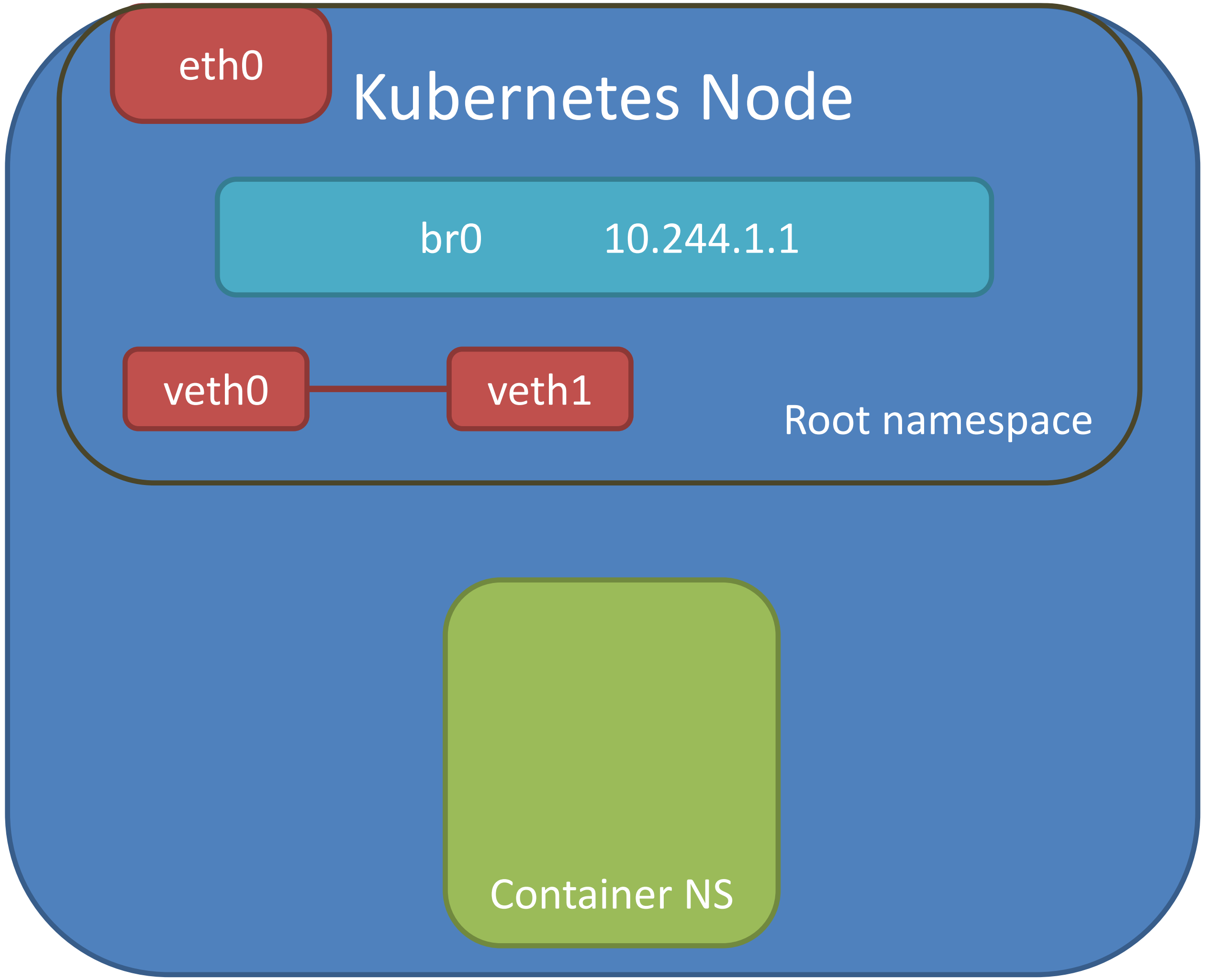
- CNI_command: add or delete
- CNI_netns: /proc/<PID>/ns/net
- CNI_ifname: eth0
- CNI_path: /opt/bin/cni
- CNI_containerid
- K8S_pod_name
- K8S_pod_namespace



CNI With Open vSwitch

Create virtual ethernet port pair

- ip link add *veth0* type veth peer name *veth1*



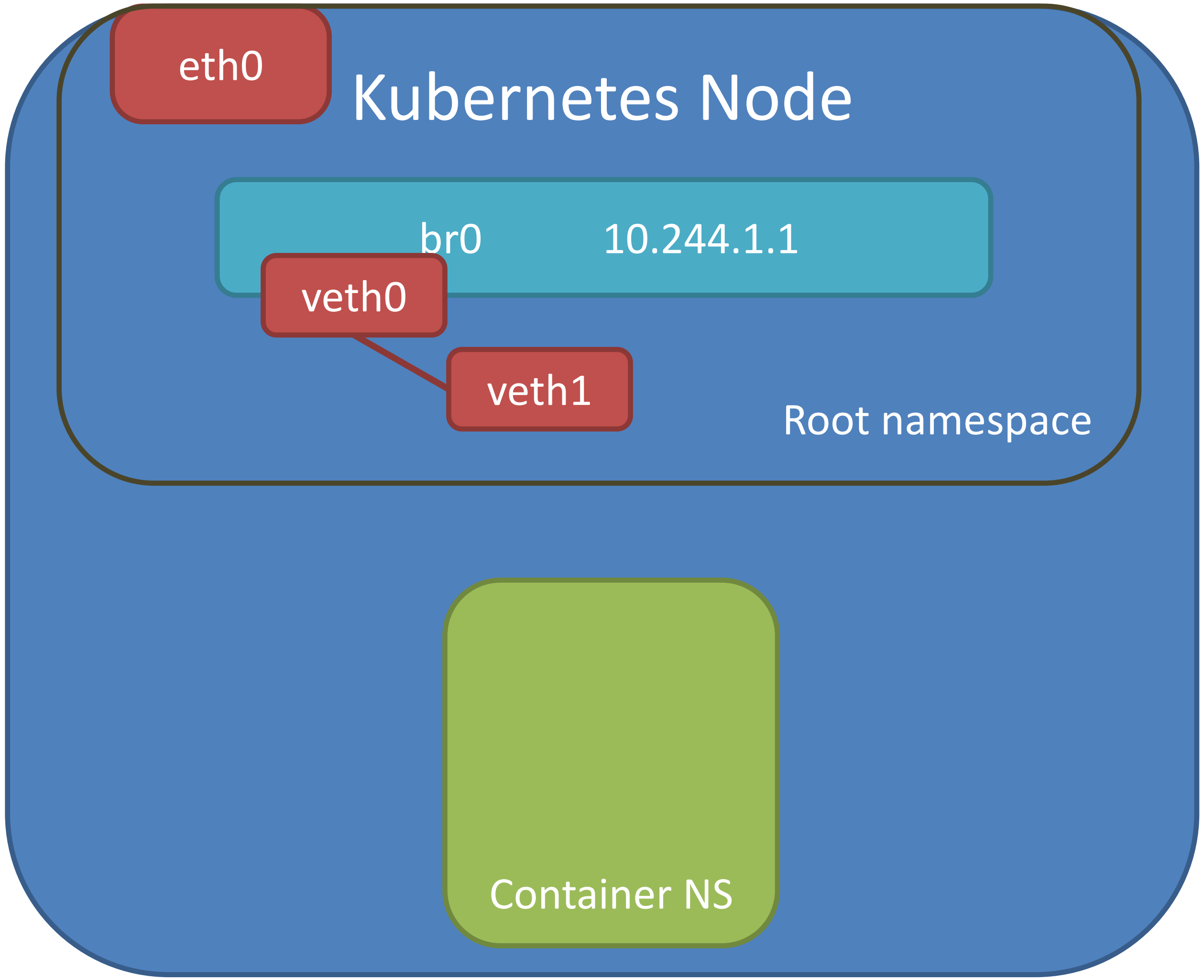
CNI With Open vSwitch

Create virtual ethernet port pair

- ip link add *veth0* type veth peer name *veth1*

Add interface to OVS bridge

- ovs-vsctl add-port *br0 veth0*



CNI With Open vSwitch

Create virtual ethernet port pair

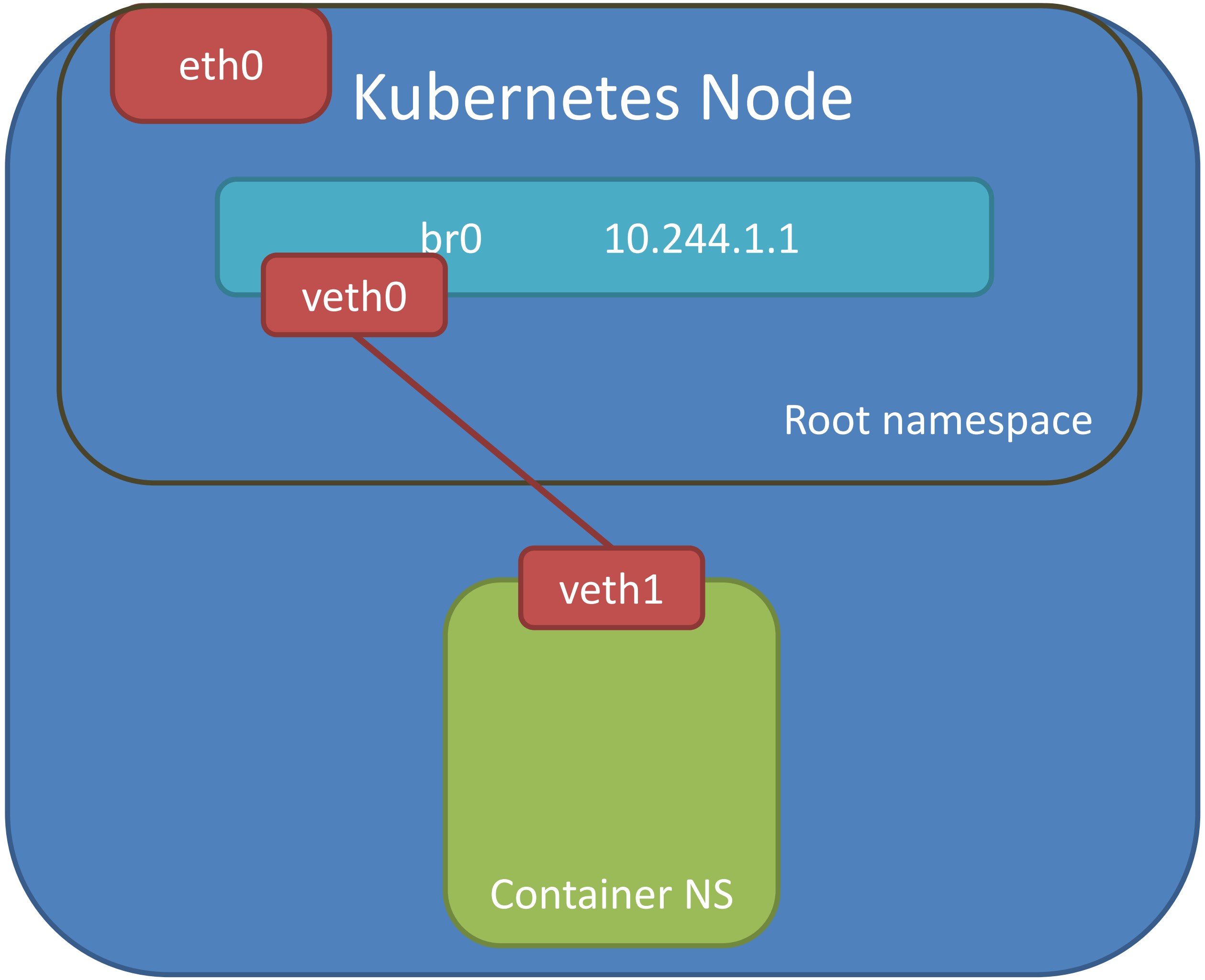
- ip link add *veth0* type veth peer name *veth1*

Add interface to OVS bridge

- ovs-vsctl add-port *br0 veth0*

Add the other interface to namespace

- ip set link *veth1* netns *$CNI_netns*



CNI With Open vSwitch

Create virtual ethernet port pair

- ip link add *veth0* type veth peer name *veth1*

Add interface to OVS bridge

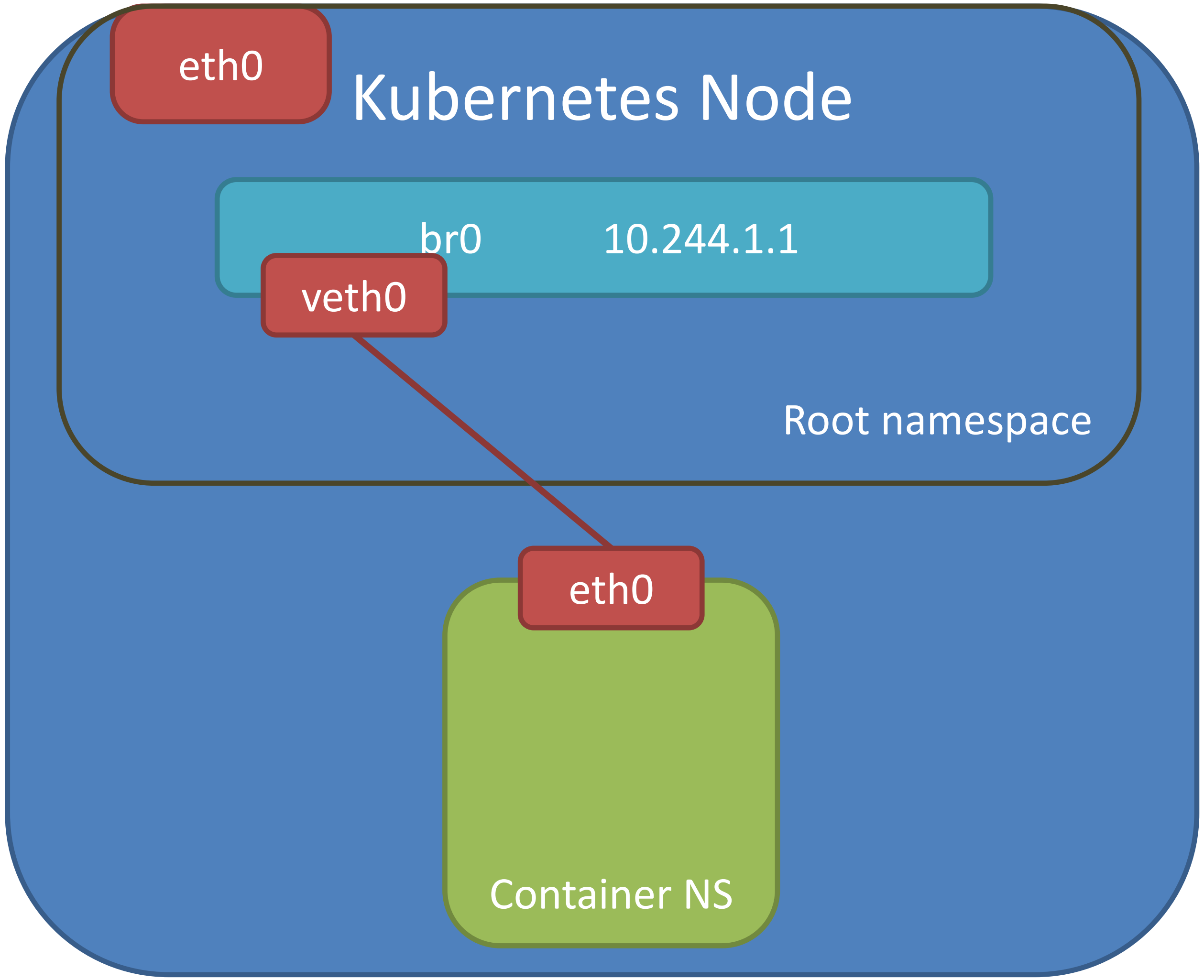
- ovs-vsctl add-port *br0* *veth0*

Add the other interface to namespace

- ip set link *veth1* netns *$CNI_netns*

Rename and setup interface

- ip netns exec *$CNI_netns*
  - ip link set dev *veth1* name *eth0*
  - ip link set dev *eth0* address *10.244.1.2*
  - ip link set dev *eth0* mtu 1450
  - ip route add default via *10.244.1.1*



CNI With Open vSwitch

Create virtual ethernet port pair

- ip link add *veth0* type veth peer name *veth1*

Add interface to OVS bridge

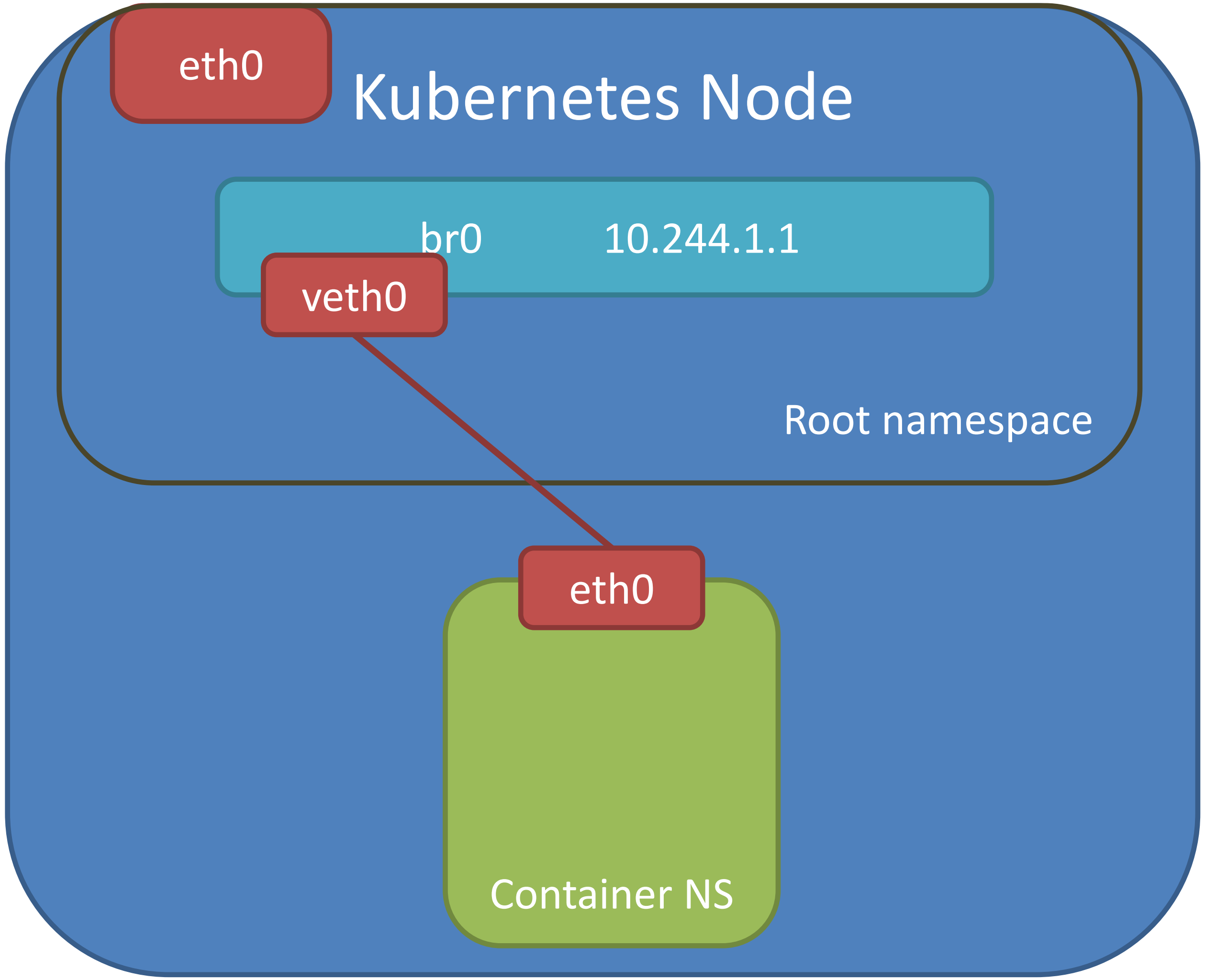
- ovs-vsctl add-port *br0 veth0*

Add the other interface to namespace

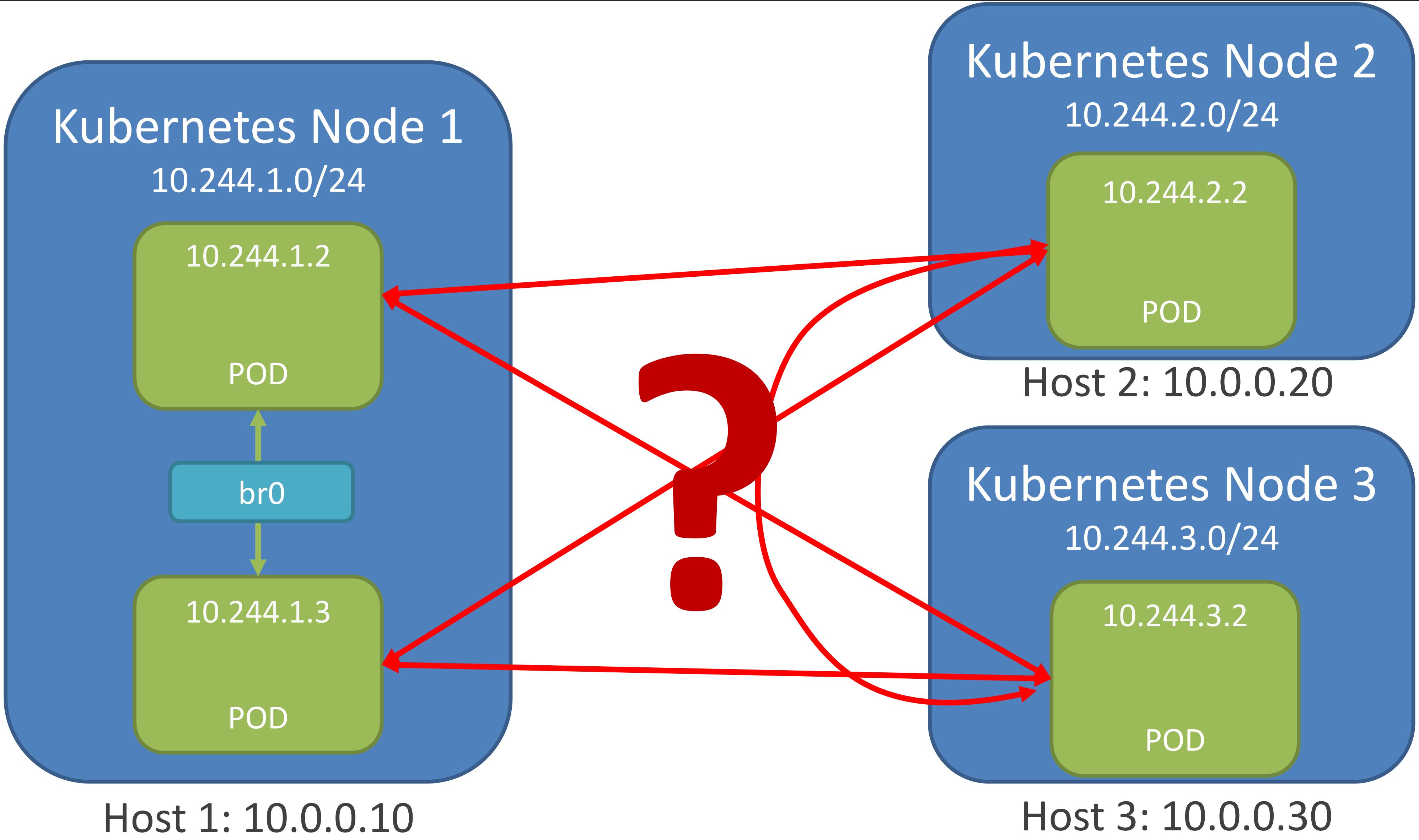
- ip set link *veth1* netns *$CNI_netns*

Rename and setup interface

- ip netns exec *$CNI_netns*
  - ip link set dev *veth1* name *eth0*
  - ip link set dev *eth0* address *10.244.1.2*
  - ip link set dev *eth0* mtu 1450
  - ip route add default via *10.244.1.1*

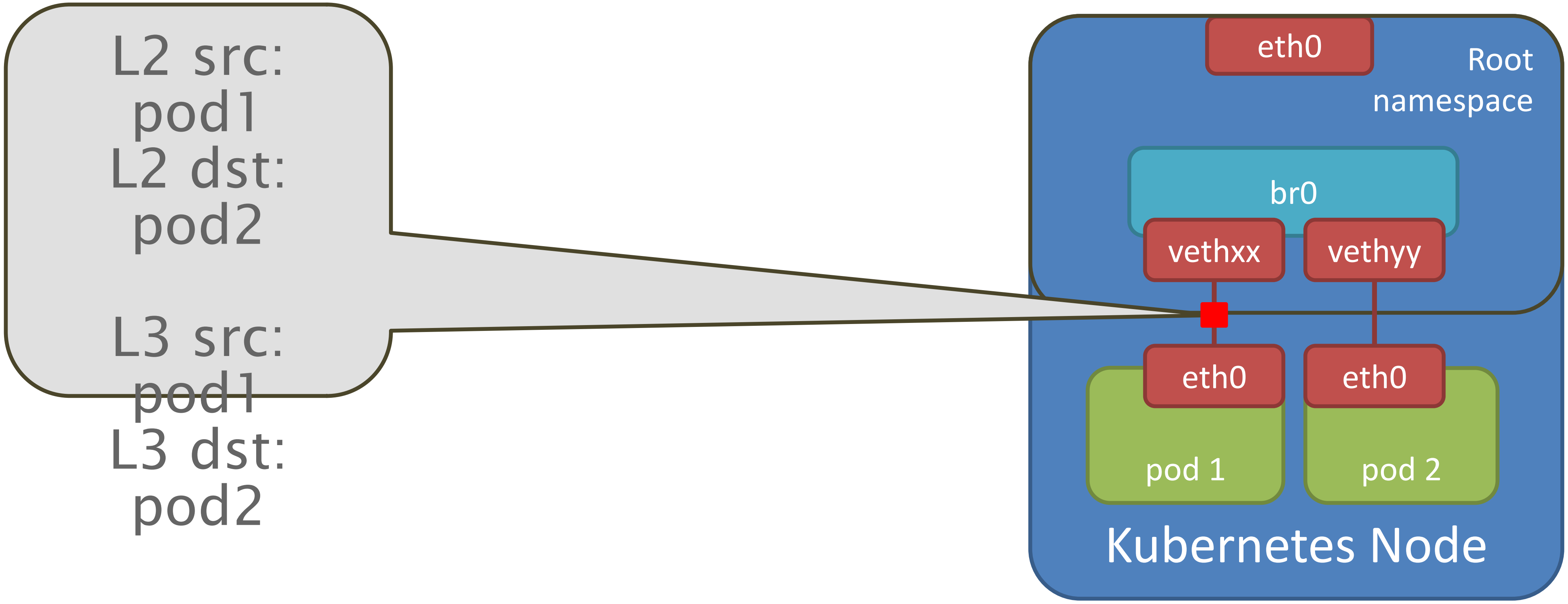


The Kubernetes Model – The IP per POD Model



Cluster Networking in Kubernetes

Life of a Packet: POD-to-POD, Same Node



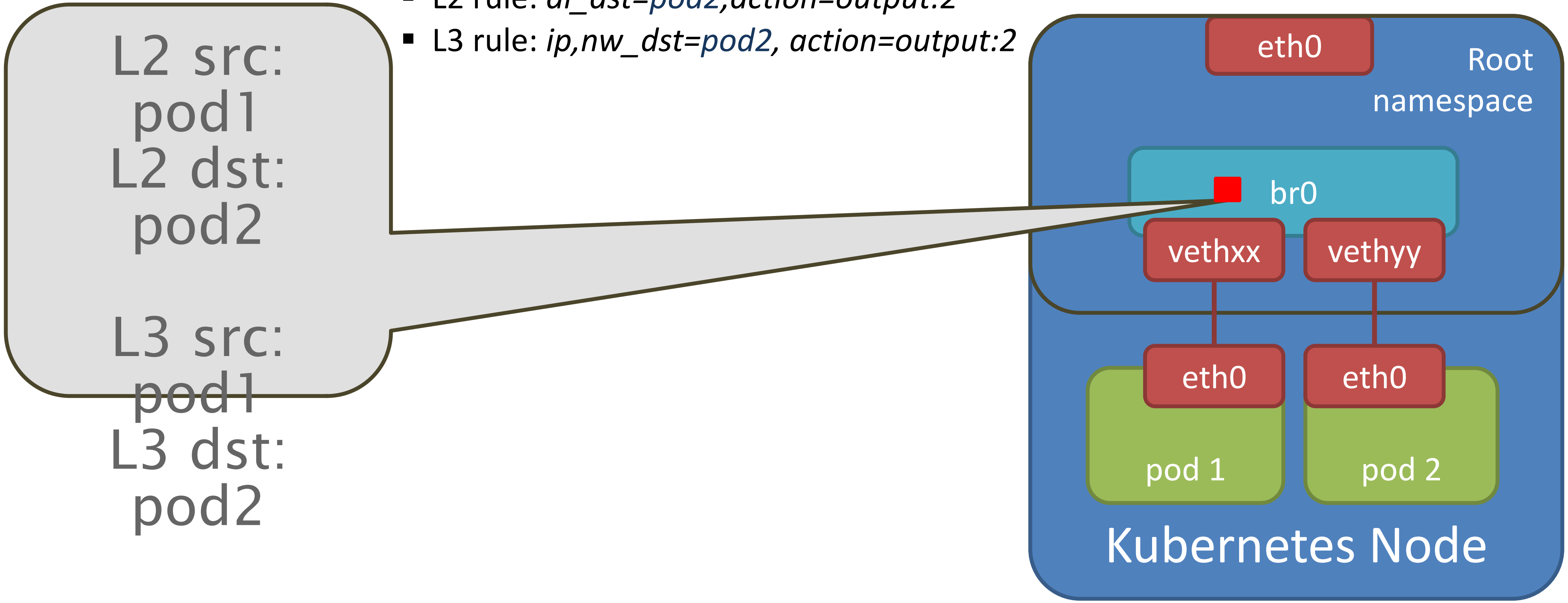
Life of a Packet: POD-to-POD, Same Node

Linux Bridge

- MAC learning

Open vSwitch

- MAC learning: *action=normal*
- L2 rule: *dl_dst=pod2,action=output:2*
- L3 rule: *ip,nw_dst=pod2, action=output:2*



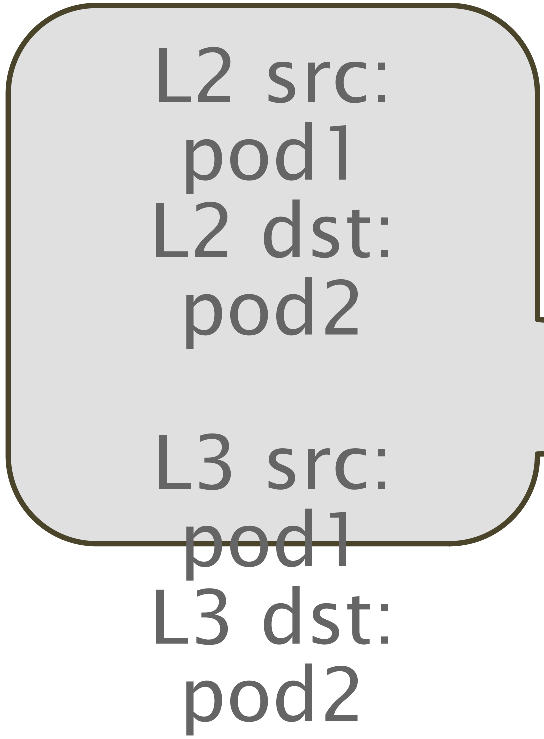
Life of a Packet: POD-to-POD, Same Node

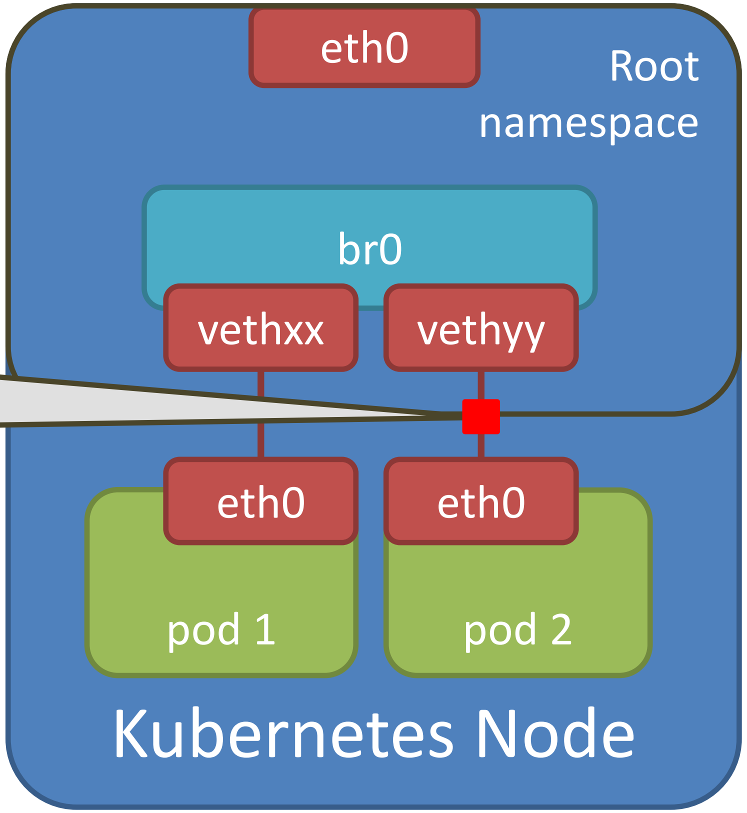
Linux Bridge

- MAC learning

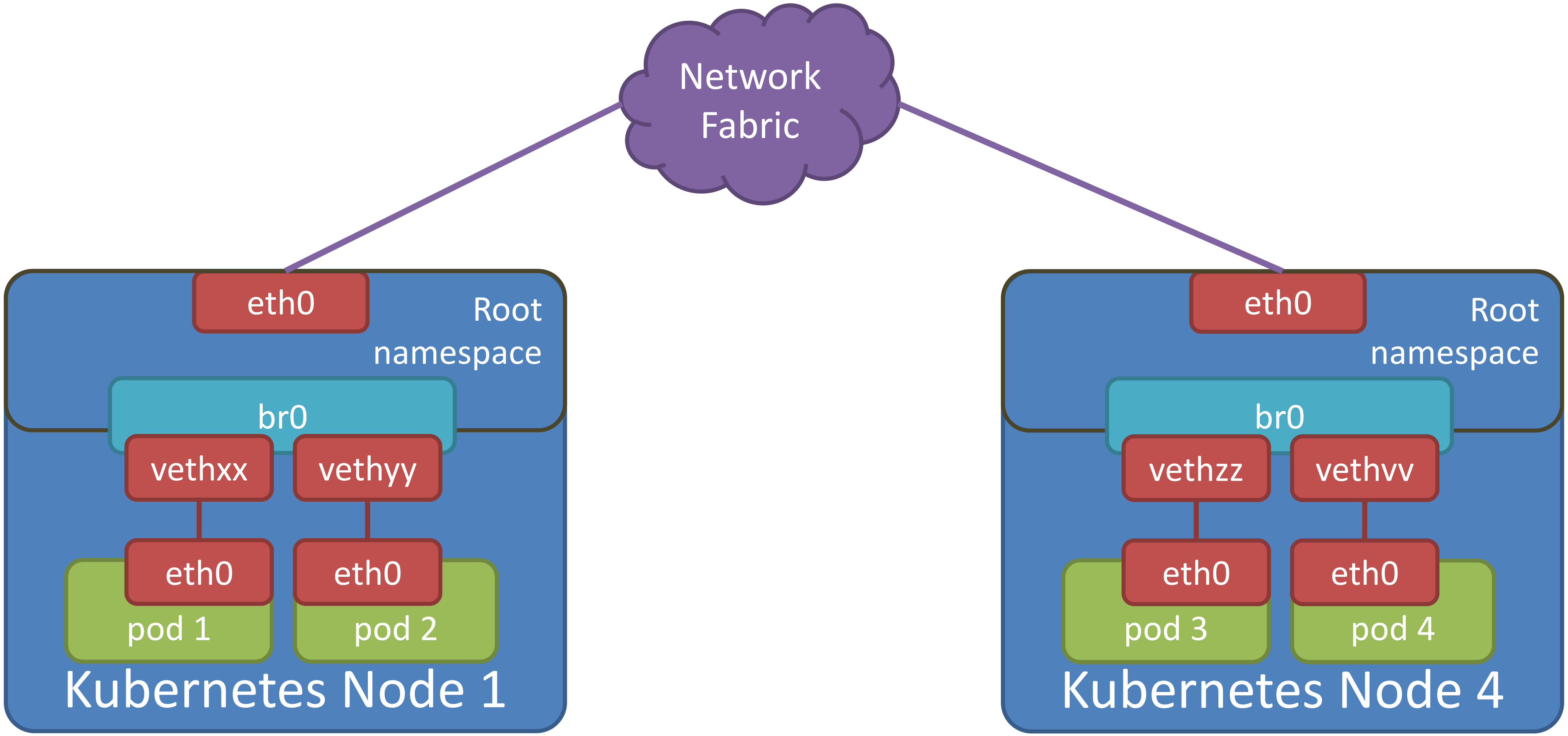
Open vSwitch

- MAC learning: *action=normal*
- L2 rule: *dl_dst=pod2,action=output:2*
- L3 rule: *ip,nw_dst=pod2, action=output:2*

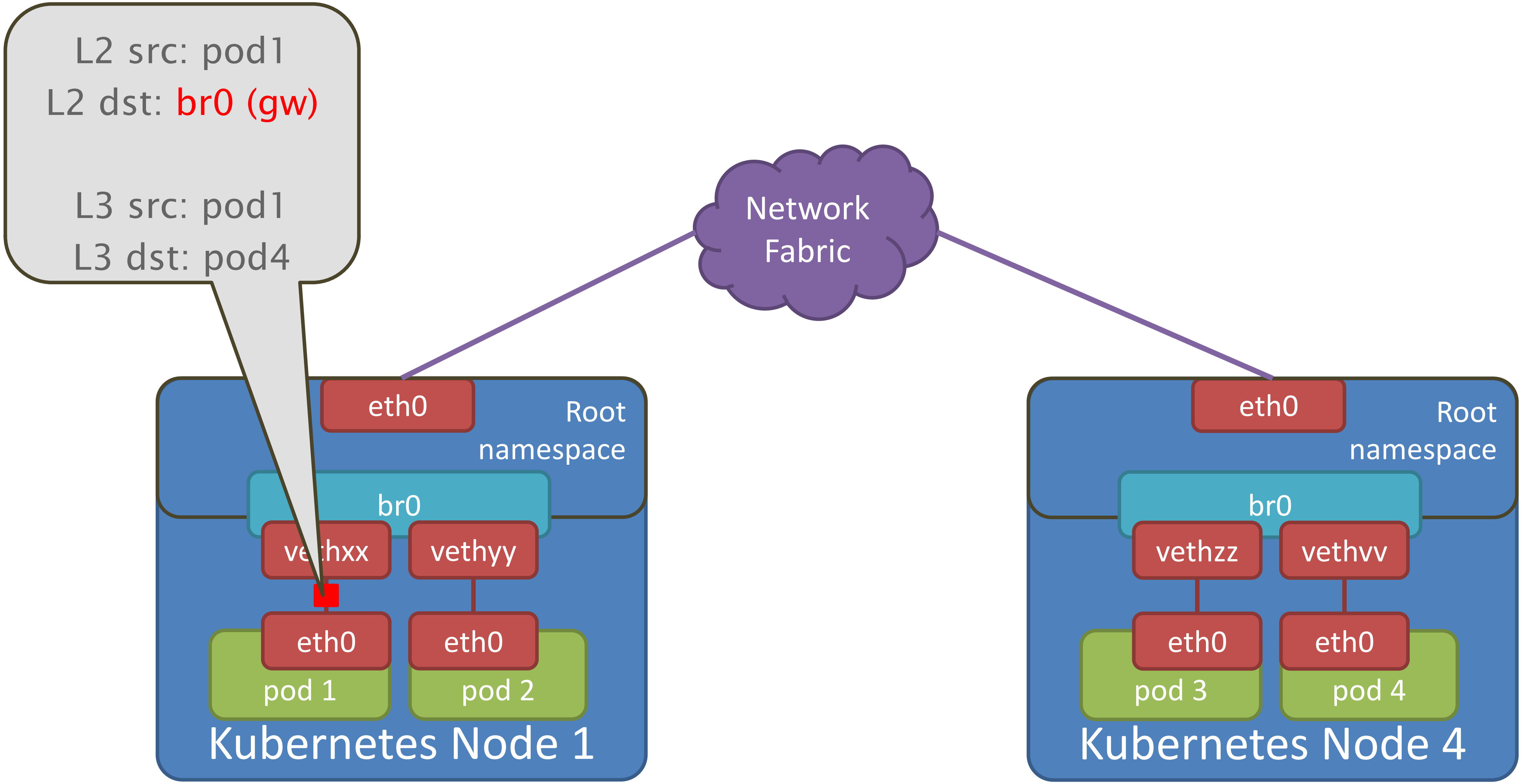




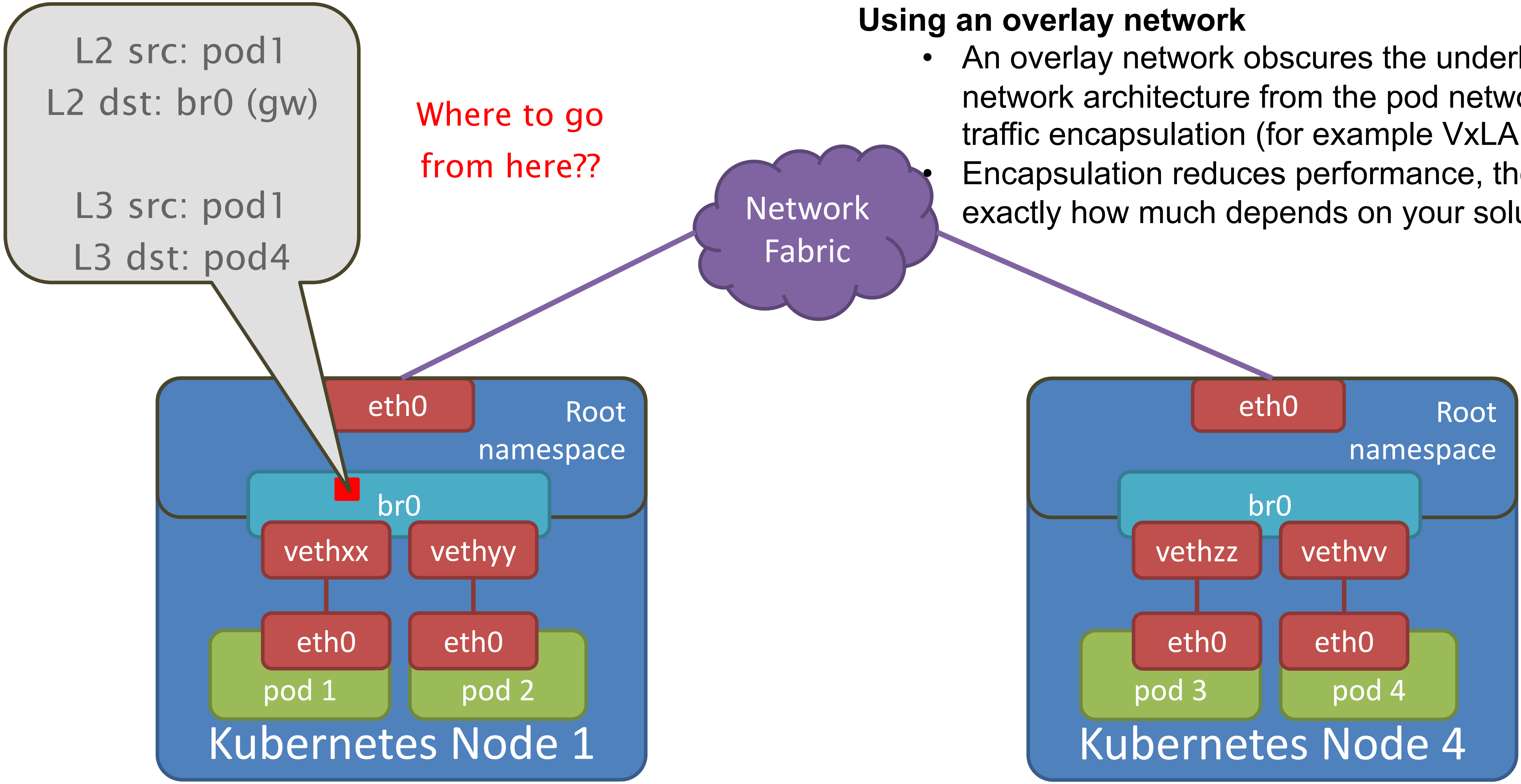
Life of a Packet: POD-to-POD, Between Nodes



Life of a Packet: POD-to-POD, Between Nodes



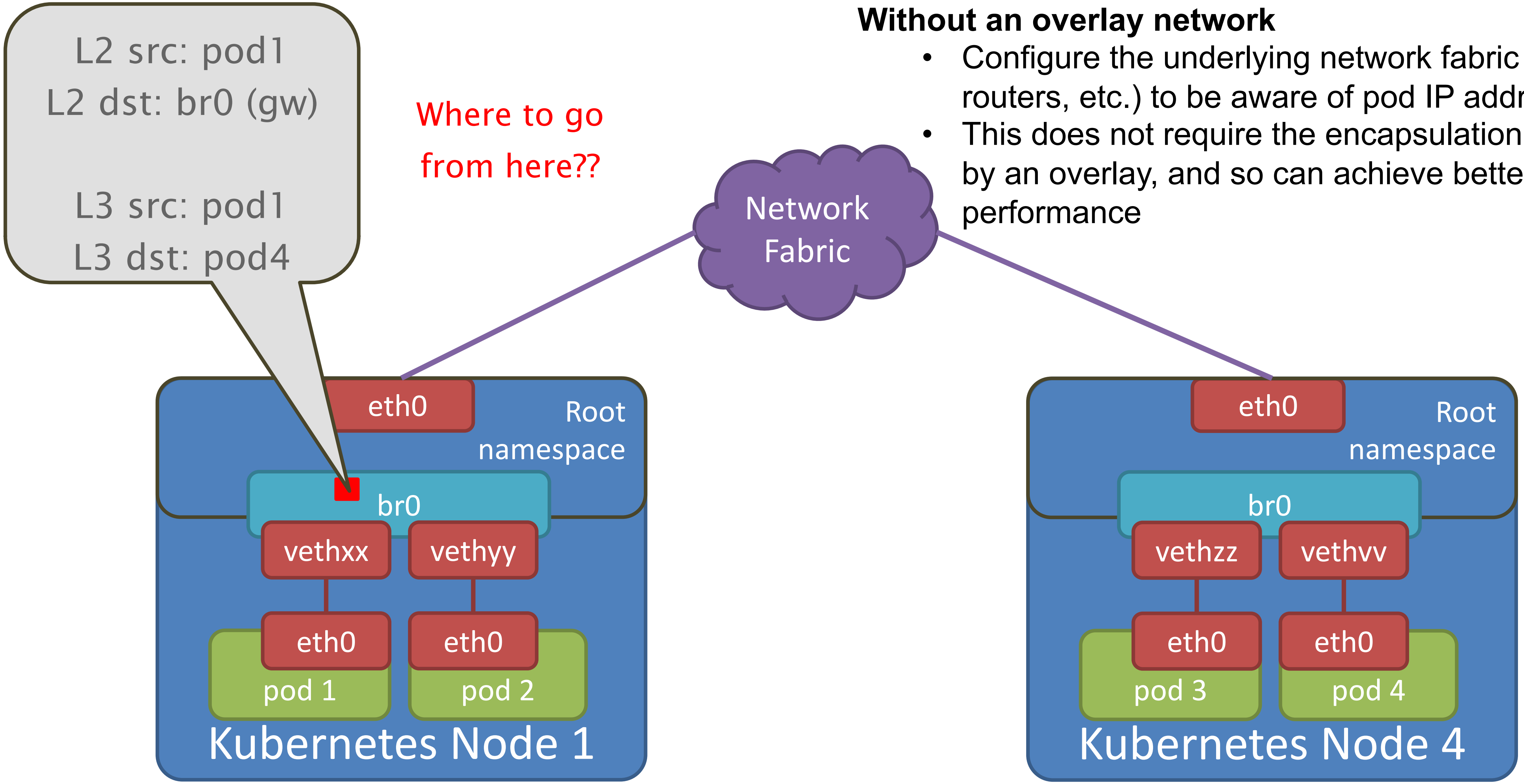
Life of a Packet: POD-to-POD, Between Nodes



Using an overlay network

- An overlay network obscures the underlying network architecture from the pod network through traffic encapsulation (for example VxLAN, GRE)
- Encapsulation reduces performance, though exactly how much depends on your solution

Life of a Packet: POD-to-POD, Between Nodes



Without an overlay network

- Configure the underlying network fabric (switches, routers, etc.) to be aware of pod IP addresses
- This does not require the encapsulation provided by an overlay, and so can achieve better performance

Kubernetes Cluster Networking Plugins

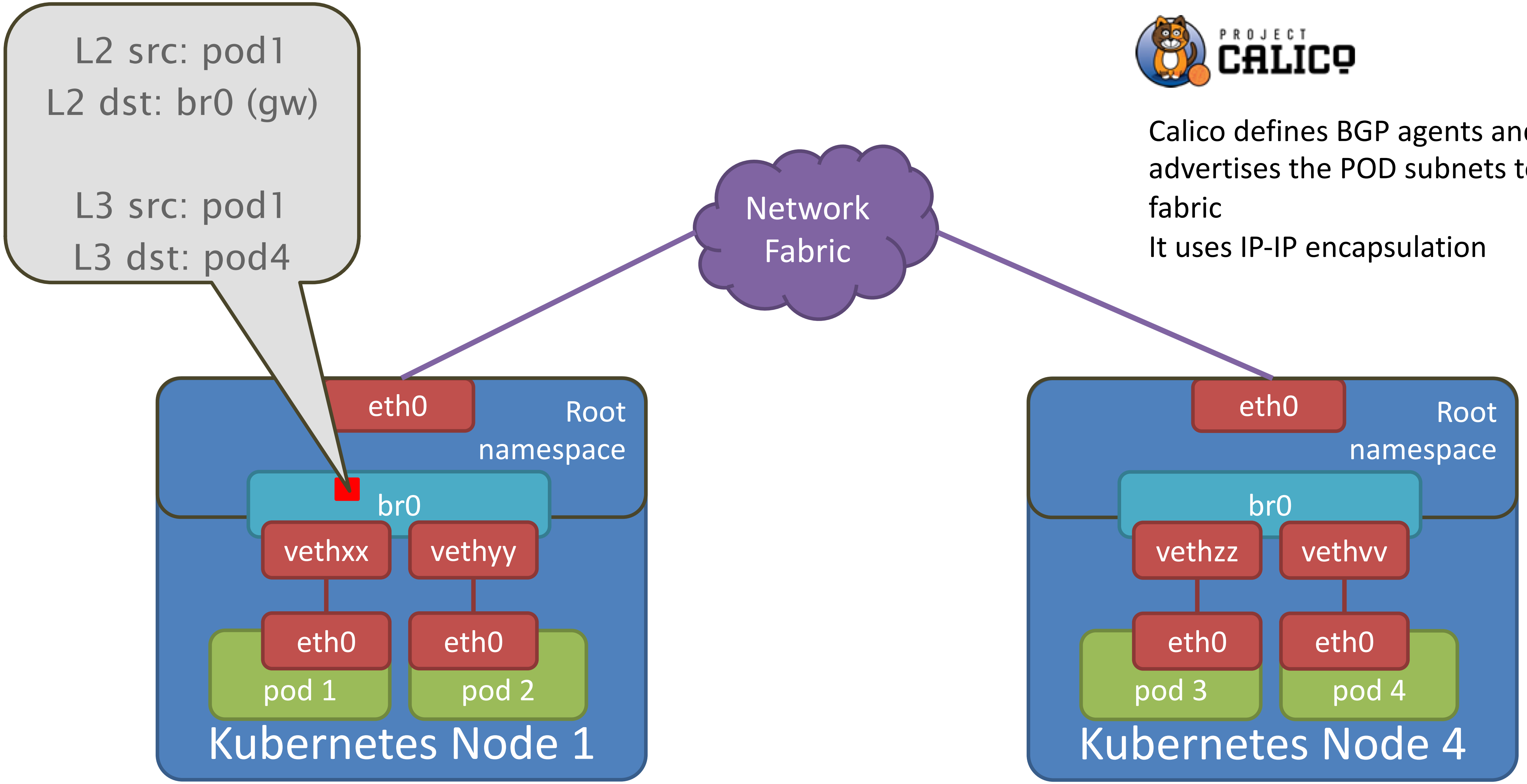
Public clouds which supports Kuberentes program this into the fabric

- E.g. in Google Container Engine: “everything to 10.1.1.0/24, send to this VM”

In other cases we need to use an external plugin

- Flannel
- Calico
- Canal
- Romana
- Weave
- Cisco Contiv
- Huawei CNI-Genie
- Nuage Networks VCS (by Nokia)
- Open Virtual Network

Life of a Packet: POD-to-POD, Between Nodes



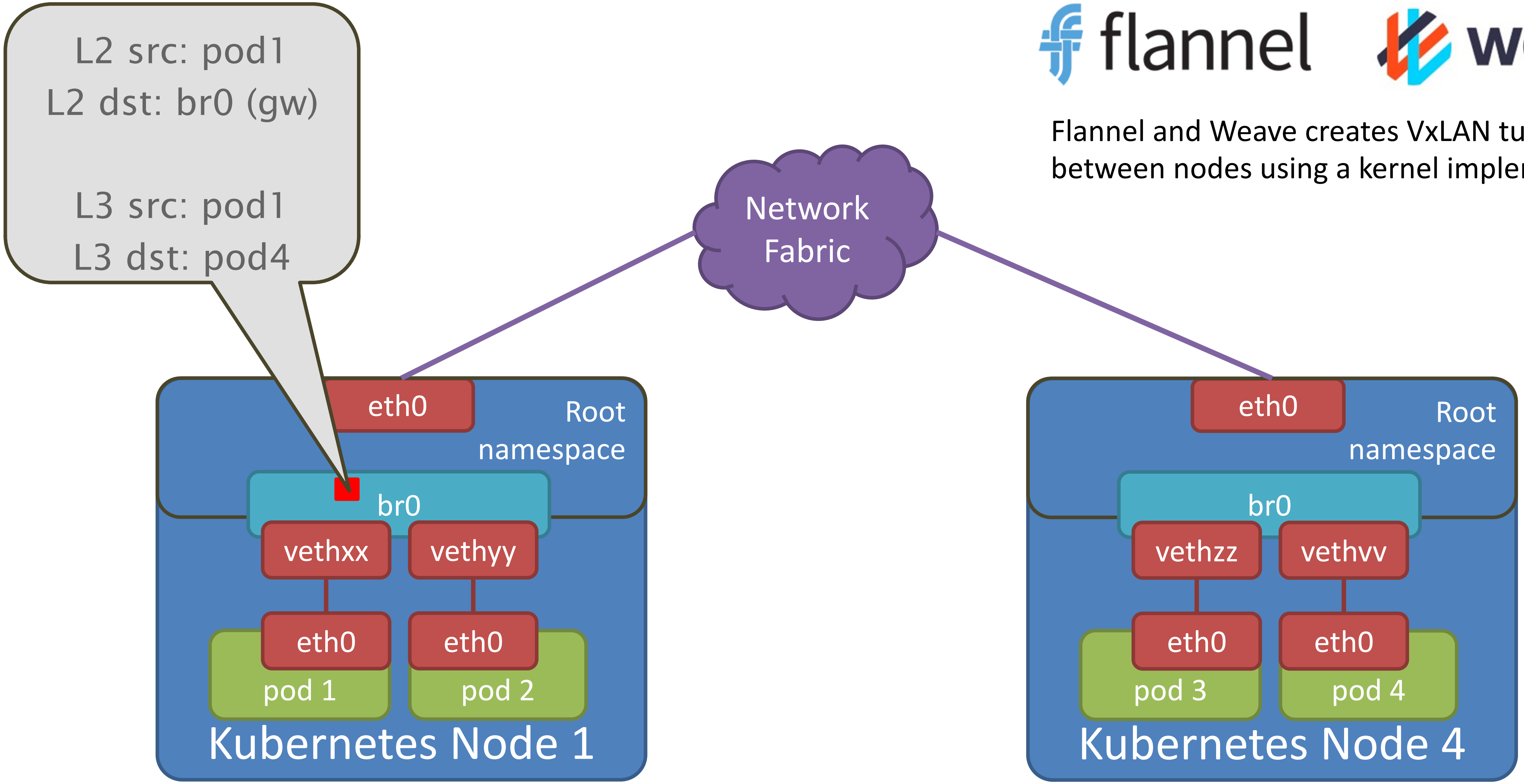
PROJECT CALICO

Calico defines BGP agents and advertises the POD subnets to the fabric

It uses IP-IP encapsulation

Life of a Packet: POD-to-POD, Between Nodes

Flannel and Weave creates VxLAN tunnels between nodes using a kernel implementation



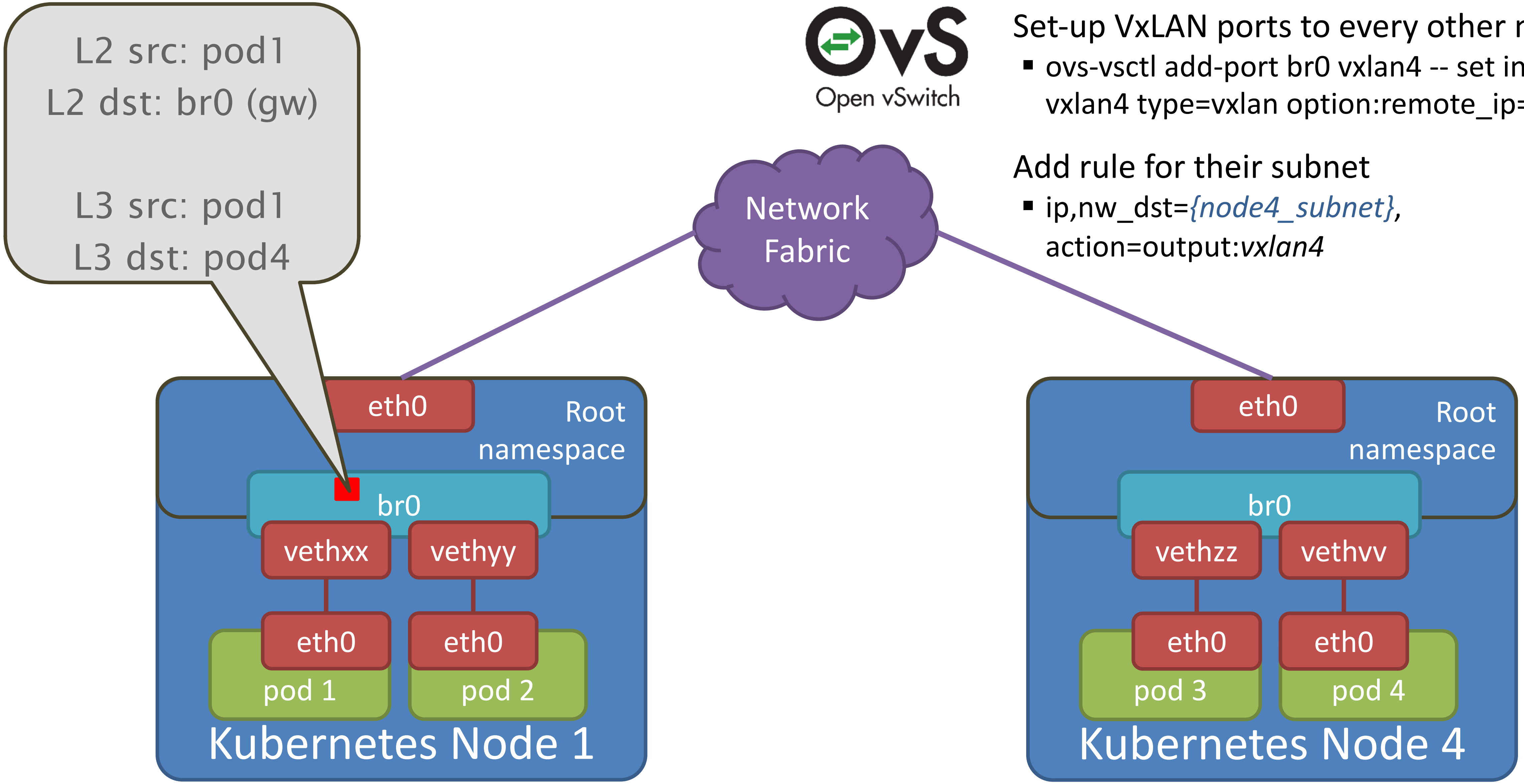
Life of a Packet: POD-to-POD, Between Nodes

Set-up VxLAN ports to every other node

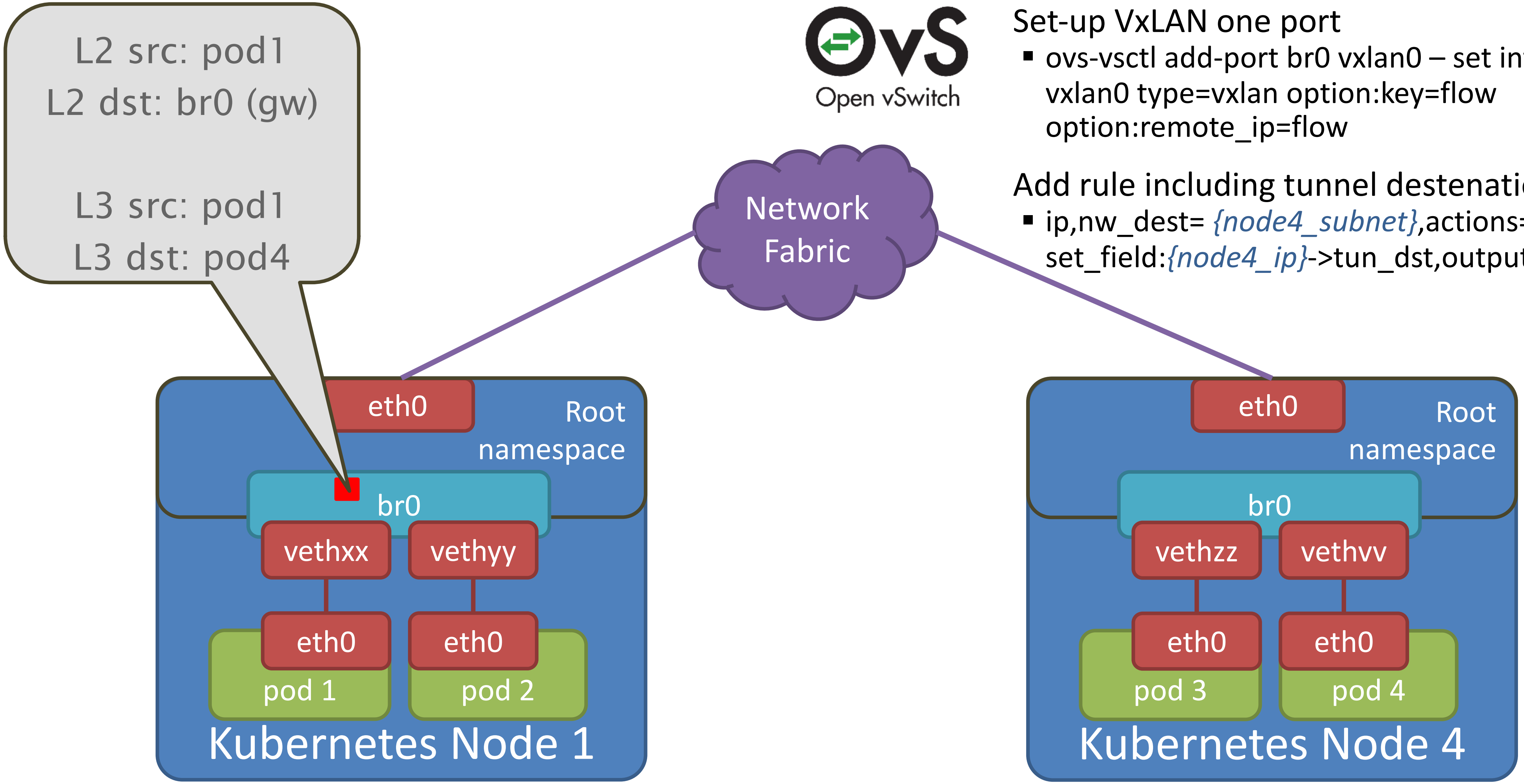
- ovs-vsctl add-port br0 vxlan4 -- set interface vxlan4 type=vxlan option:remote_ip=*{node4_ip}*

Add rule for their subnet

- ip,nw_dst=*{node4_subnet}*, action=output:*vxlan4*



Life of a Packet: POD-to-POD, Between Nodes



Set-up VxLAN one port

- ovs-vsctl add-port br0 vxlan0 – set interface vxlan0 type=vxlan option:key=flow option:remote_ip=flow

Add rule including tunnel destenation

- ip,nw_dest= *{node4_subnet}*,actions= set_field:*{node4_ip}*->tun_dst,output:*vxlan0*

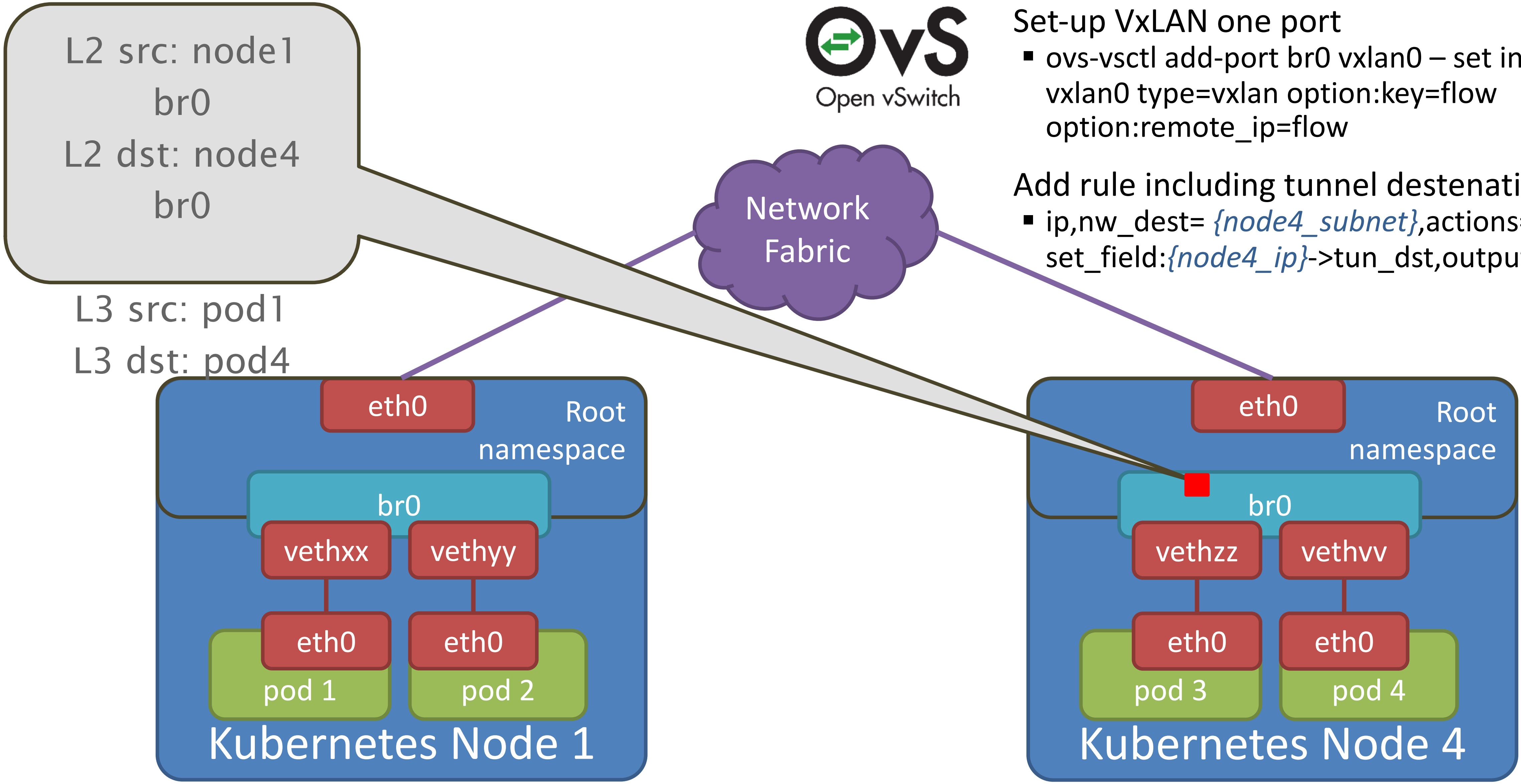
Life of a Packet: POD-to-POD, Between Nodes

Set-up VxLAN one port

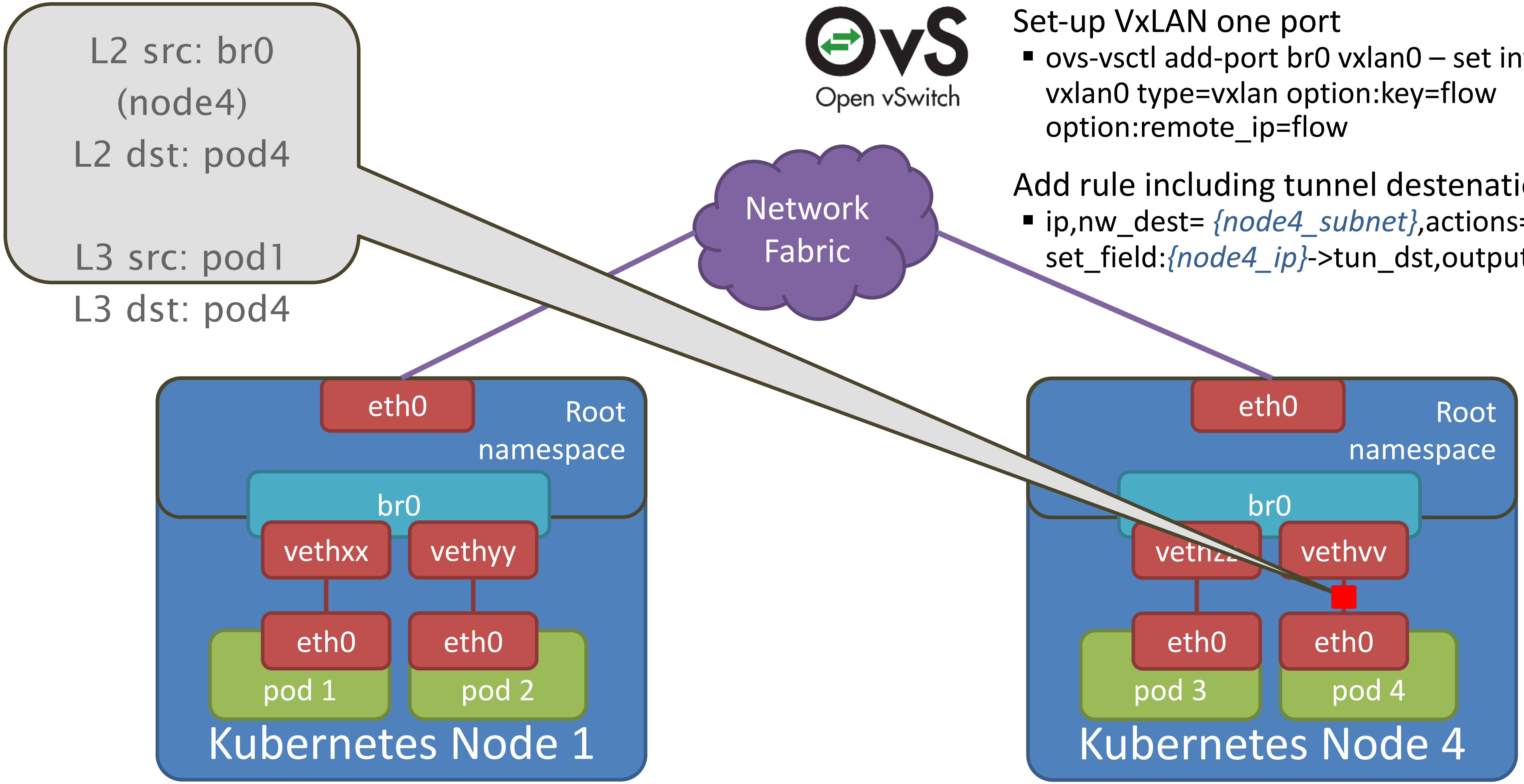
- ovs-vsctl add-port br0 vxlan0 – set interface vxlan0 type=vxlan option:key=flow option:remote_ip=flow

Add rule including tunnel destenation

- ip,nw_dest= *{node4_subnet}*,actions= set_field:*{node4_ip}*->tun_dst,output:*vxlan0*



Life of a Packet: POD-to-POD, Between Nodes



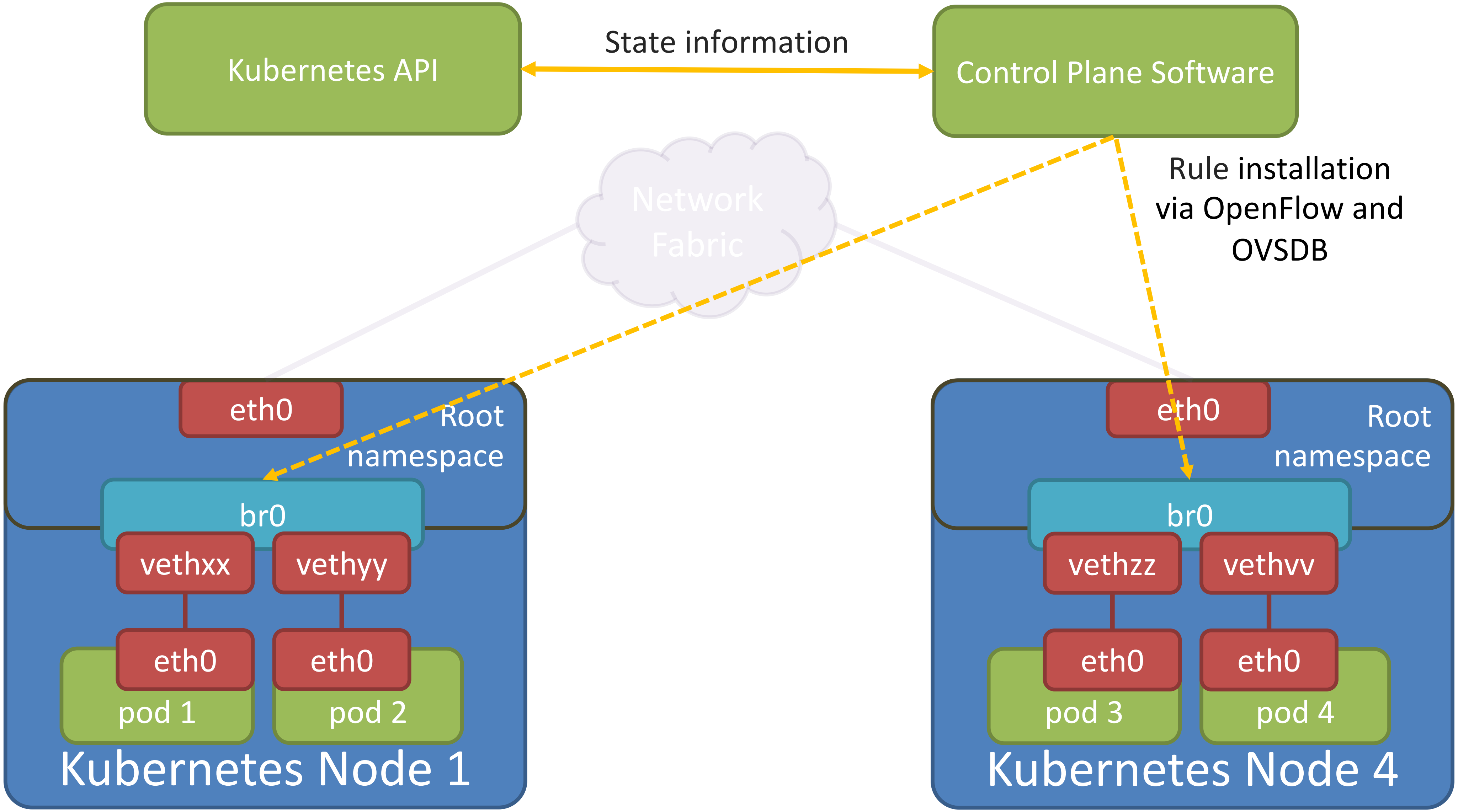
Set-up VxLAN one port

- ovs-vsctl add-port br0 vxlan0 – set interface vxlan0 type=vxlan option:key=flow option:remote_ip=flow

Add rule including tunnel destenation

- ip,nw_dest= *{node4_subnet}*,actions= set_field:*{node4_ip}*->tun_dst,output:*vxlan0*

For This, You Will Need a Control Plane



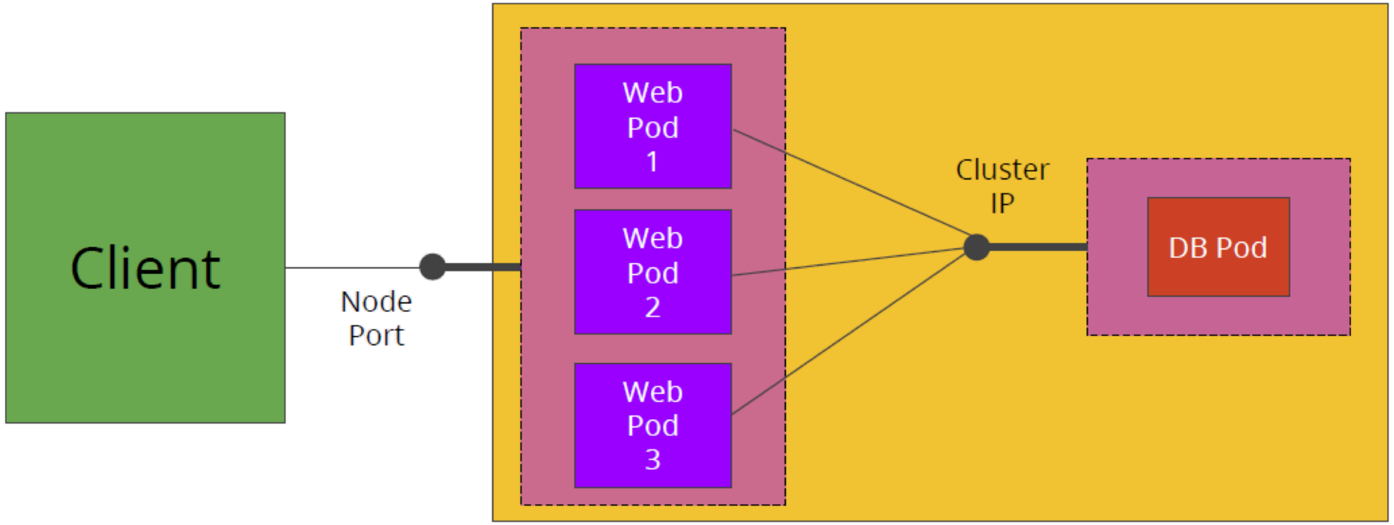
Pod to Service Communication in Kubernetes

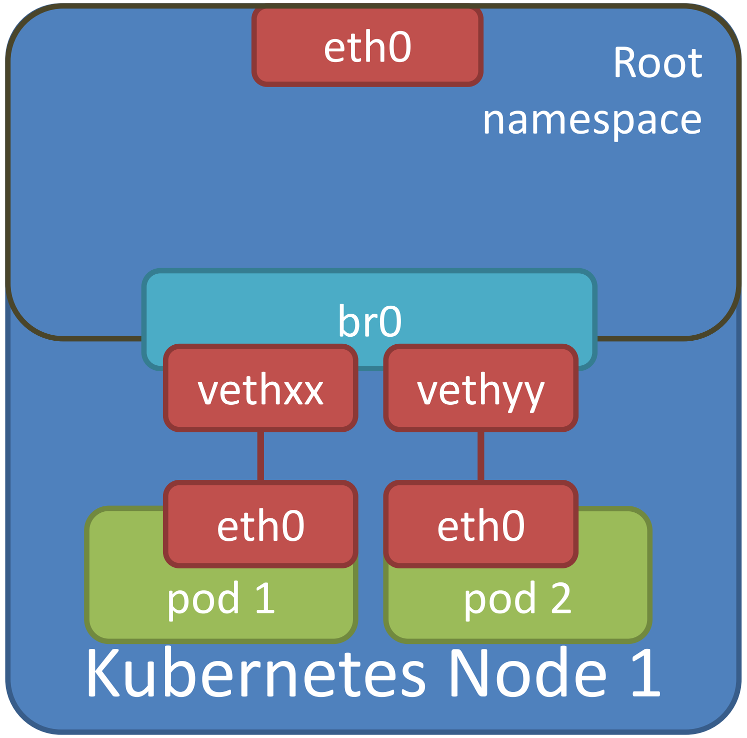
Services in Kubernetes

Definition:

- Service is an abstraction to define a logical set of Pods bound by a policy by to access them
- Defined by labels and selectors
- Supports TCP and UDP
- Interfaces with Kube-Proxy to manipulate IPtables
- Service can be exposed internally by cluster/service IP

Remember:
PODs are Mortal!!!





Services in Kubernetes

```
kind: Service
apiVersion: v1
metadata:
  name: store-be
  namespace: default
  creationTimestamp: 2016-05-06T19:16:56Z
  resourceVersion: "7"
  selfLink:
/api/v1/namespaces/default/services/store-be
  uid: 196d5751-13bf-11e6-9353-42010a800fe3
Spec:
  type: ClusterIP
  selector:
    app: store
    role: be
  clusterIP: 10.9.3.76
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 80
  sessionAffinity: None
```

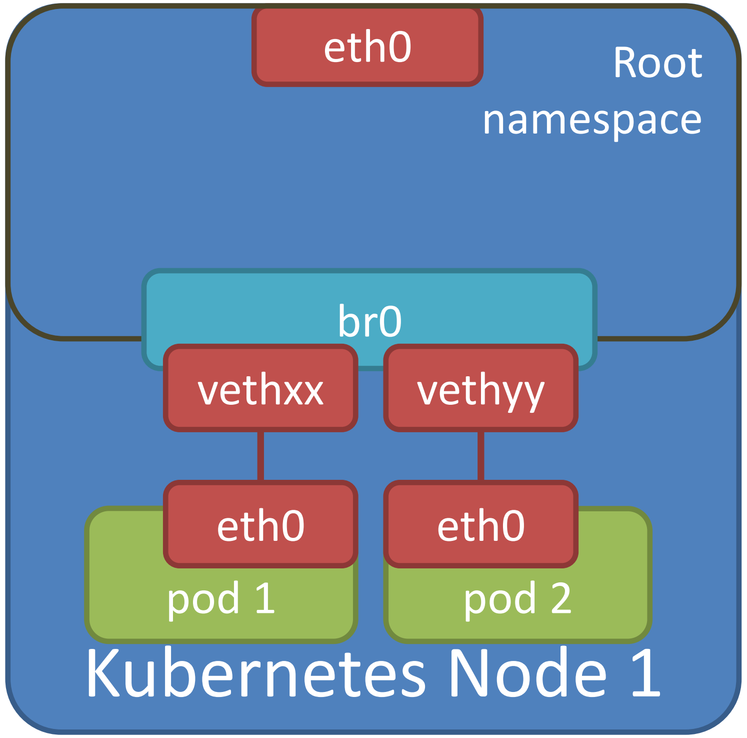
name: store-be → This will be the DNS name of the service

app: store / role: be → Selector for PODs

clusterIP: 10.9.3.76 → This will be the IP of the service

port: 80 → This will be the port of the service

targetPort: 80 → This is the POD port



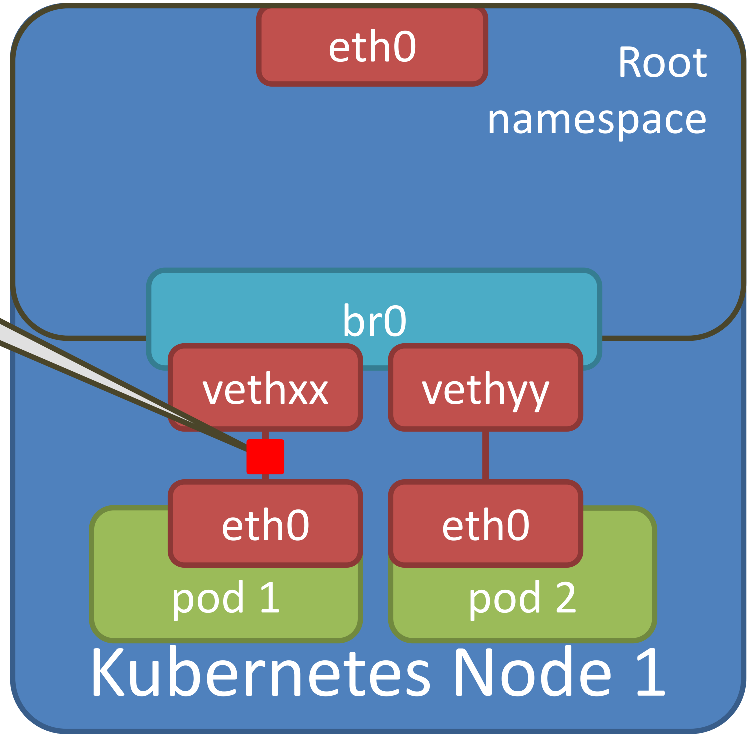
Life of a Packet: POD-to-Service

L2 src: pod1

L2 dst: br0

L3 src: pod1

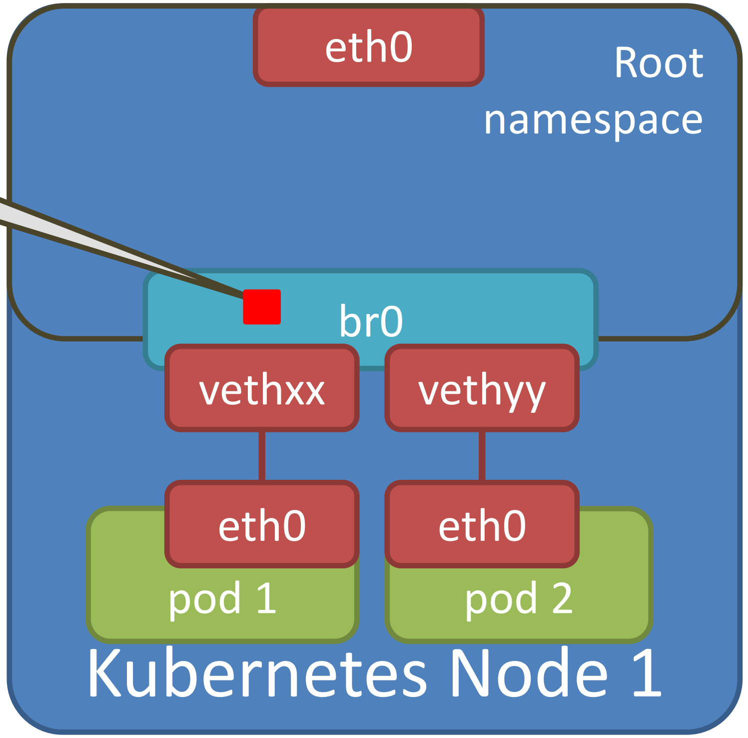
L3 dst: svc1



Life of a Packet: POD-to-Service

L2 src: pod1
L2 dst: br0

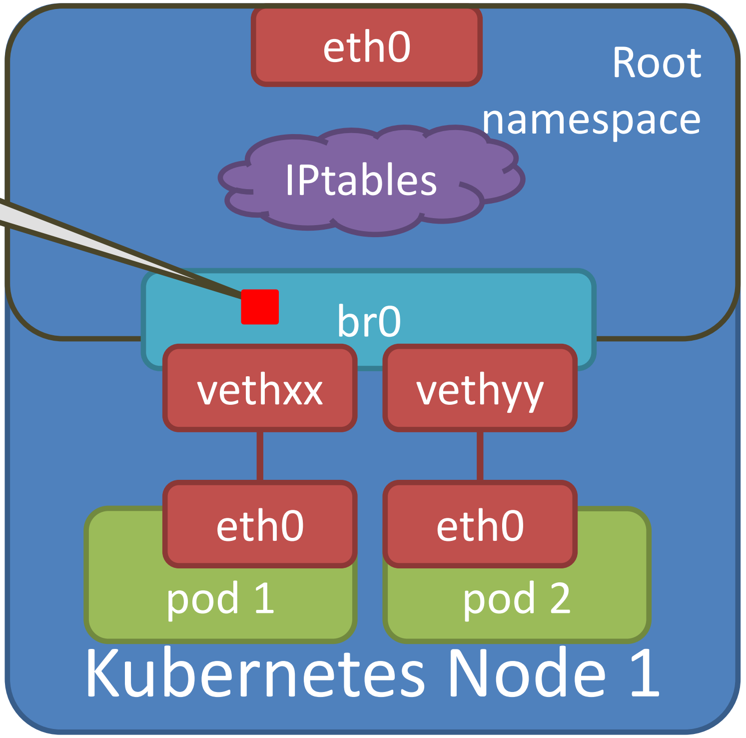
L3 src: pod1
L3 dst: svc1



Life of a Packet: POD-to-Service

L2 src: pod1
L2 dst: br0

L3 src: pod1
L3 dst: svc1



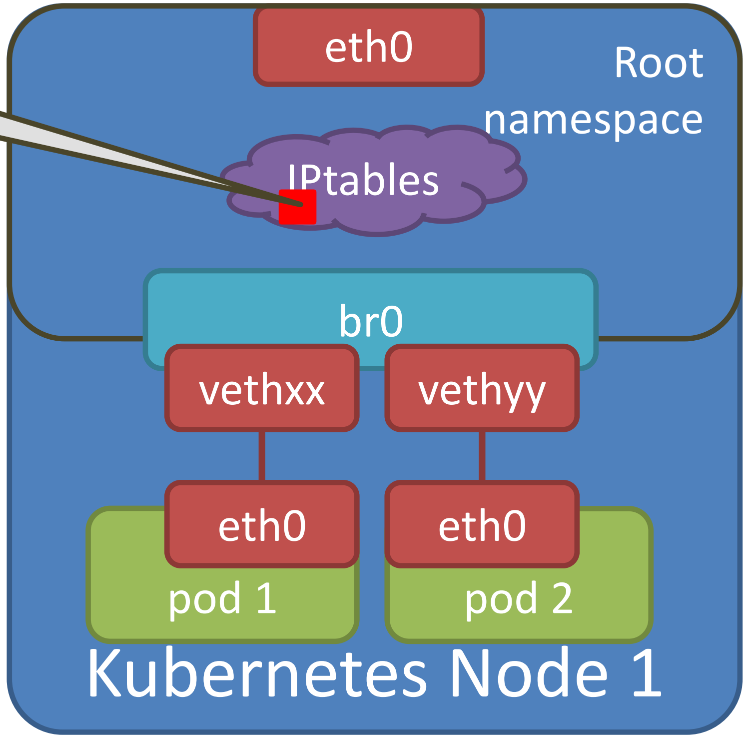
Life of a Packet: POD-to-Service

L3 src: pod1

~~L3 dst: svc1~~

L3 dst: pod88

DNAT, conntrack



Remember:

- Every node should reach every POD in the cluster
- *ip route add {global_pod_cidr} via br0*

  e.g. 10.244.0.0/16

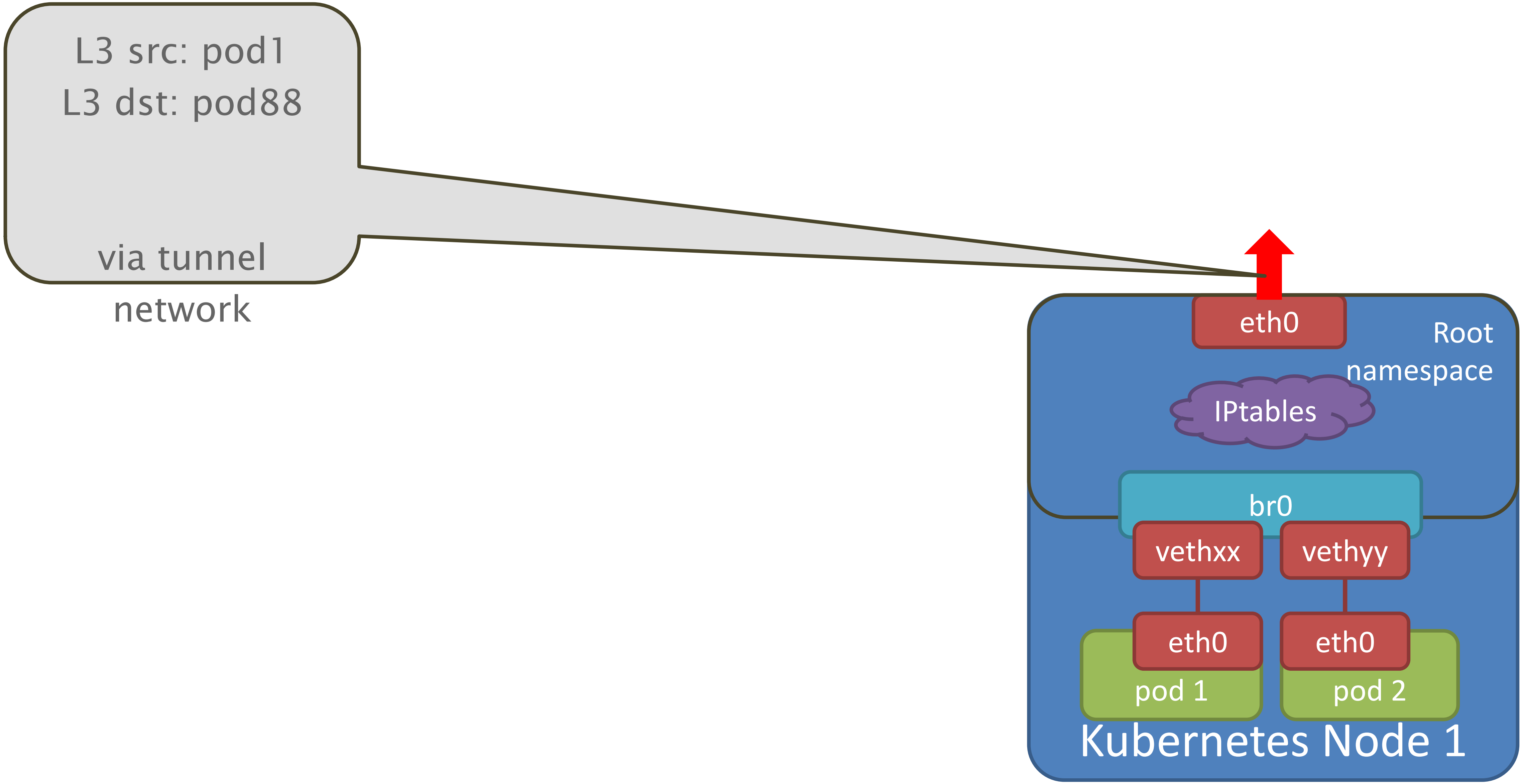
Example for IPtables Ruleset

```
Chain KUBE-SERVICES (2 references)
target     prot opt source               destination
KUBE-MARK-MASQ  tcp  -- !10.244.0.0/16        10.110.89.105        /* sock-shop/front-end: cluster IP */ tcp dpt:http
KUBE-SVC-LFMD53S3EZEAOUSJ  tcp  --  anywhere             10.110.89.105        /* sock-shop/front-end: cluster IP */ tcp dpt:http
KUBE-MARK-MASQ  tcp  -- !10.244.0.0/16        10.97.201.132        /* sock-shop/orders-db: cluster IP */ tcp dpt:27017
KUBE-SVC-K7W4GUVR3E4J4SGZ  tcp  --  anywhere             10.97.201.132        /* sock-shop/orders-db: cluster IP */ tcp dpt:27017
KUBE-MARK-MASQ  tcp  -- !10.244.0.0/16        10.97.121.97         /* sock-shop/rabbitmq: cluster IP */ tcp dpt:amqp
KUBE-SVC-HFJ6SIC3BWQ7VZIS  tcp  --  anywhere             10.97.121.97         /* sock-shop/rabbitmq: cluster IP */ tcp dpt:amqp
```

```
Chain KUBE-SVC-LFMD53S3EZEAOUSJ (2 references)
target     prot opt source               destination
KUBE-SEP-55SOKLSCEKVOUEYD  all  --  anywhere             anywhere             /* sock-shop/front-end: */ statistic mode random probability 0.25000000000
KUBE-SEP-VW6NJSN2QROYSWMQ  all  --  anywhere             anywhere             /* sock-shop/front-end: */ statistic mode random probability 0.33332999982
KUBE-SEP-KCT3UGP5JLP4PQYI  all  --  anywhere             anywhere             /* sock-shop/front-end: */ statistic mode random probability 0.50000000000
KUBE-SEP-NXIYBBHETPWGYE3W  all  --  anywhere             anywhere             /* sock-shop/front-end: */
```

```
Chain KUBE-SEP-55SOKLSCEKVOUEYD (1 references)
target     prot opt source               destination
KUBE-MARK-MASQ  all  --  10.244.1.7           anywhere             /* sock-shop/front-end: */
DNAT       tcp  --  anywhere             anywhere             /* sock-shop/front-end: */ tcp to:10.244.1.7:8079
```

Life of a Packet: POD-to-Service

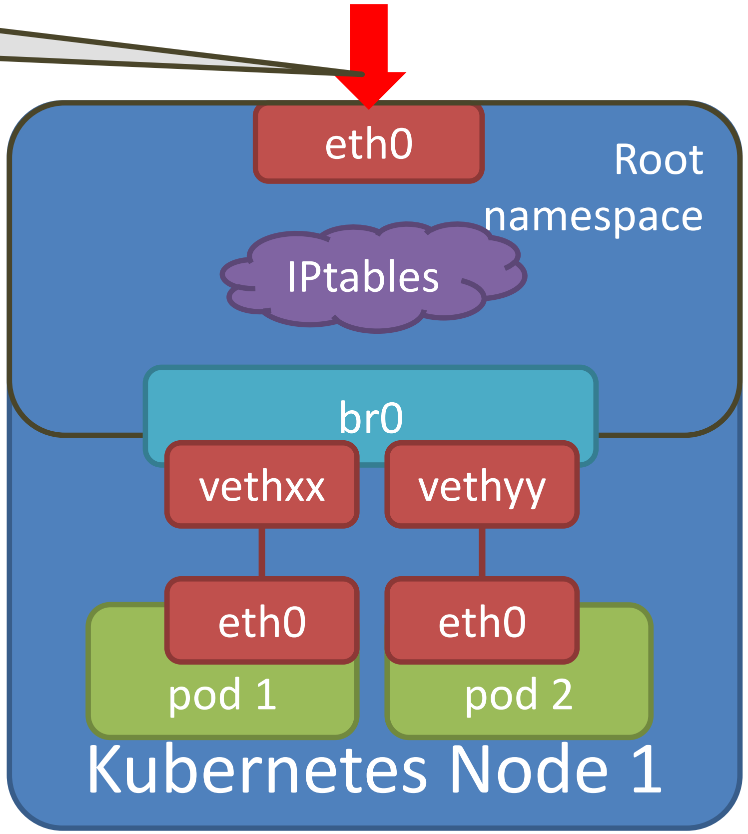


Life of a Packet: POD-to-Service

L3 src: pod88

L3 dst: pod1

via tunnel network



Life of a Packet: POD-to-Service

L3 src: pod88
L3 dst: pod1

via tunnel network

You can install only one pod network per cluster.

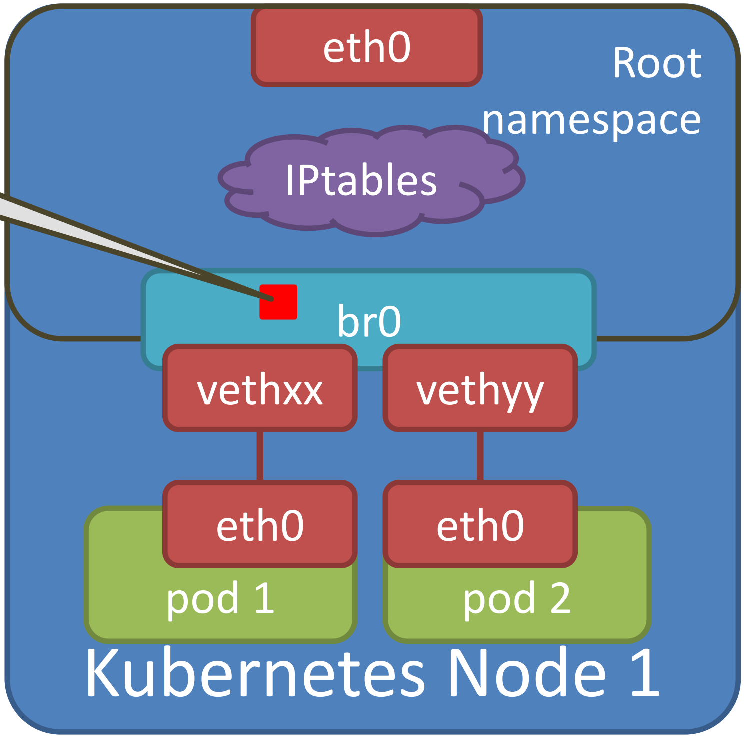
Choose one... | Calico | Canal | **Flannel** | Kube-router | Romana | Weave Net | JuniperContrail/TungstenFabric

For `flannel` to work correctly, you must pass `--pod-network-cidr=10.244.0.0/16` to `kubeadm init`.

Set `/proc/sys/net/bridge/bridge-nf-call-iptables` to `1` by running `sysctl net.bridge.bridge-nf-call-iptables=1` to pass bridged IPv4 traffic to iptables' chains. This is a requirement for some CNI plugins to work, for more information please see here.

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.10.0/Documentation/kube-flannel.yml
```

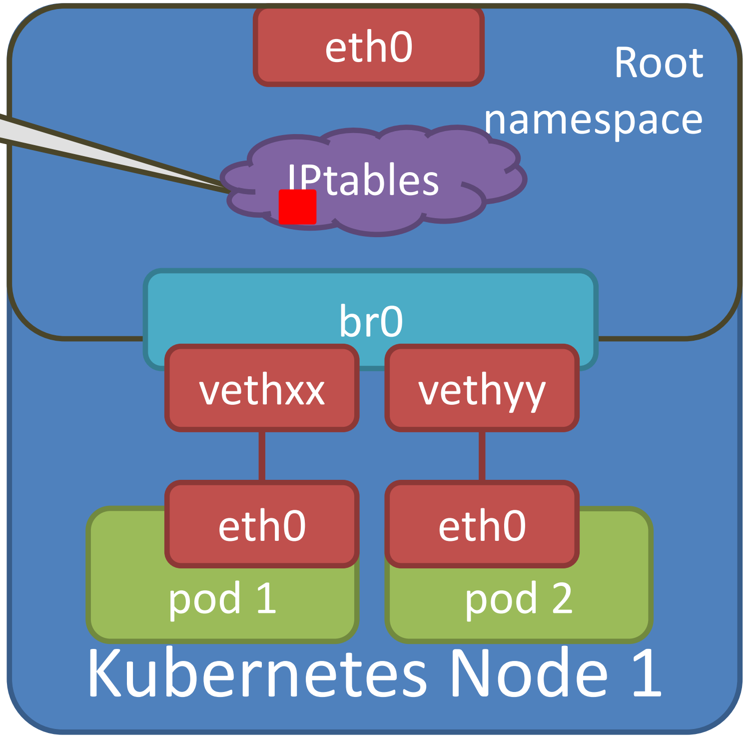
https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/



Life of a Packet: POD-to-Service

L3 ~~src: pod88~~
L3 src: svc1
L3 dst: pod1

un-DNAT



You can install only one pod network per cluster.

Choose one... | Calico | Canal | Flannel | Kube-router | Romana | Weave Net | JuniperContrail/TungstenFabric

For `flannel` to work correctly, you must pass `--pod-network-cidr=10.244.0.0/16` to `kubeadm init`.

Set `/proc/sys/net/bridge/bridge-nf-call-iptables` to `1` by running `sysctl net.bridge.bridge-nf-call-iptables=1` to pass bridged IPv4 traffic to iptables' chains. This is a requirement for some CNI plugins to work, for more information please see here.

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.10.0/Documentation/kube-flannel.yml
```

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/

Life of a Packet: POD-to-Service

L3 src: svc1
L3 dst: pod1

You can install only one pod network per cluster.

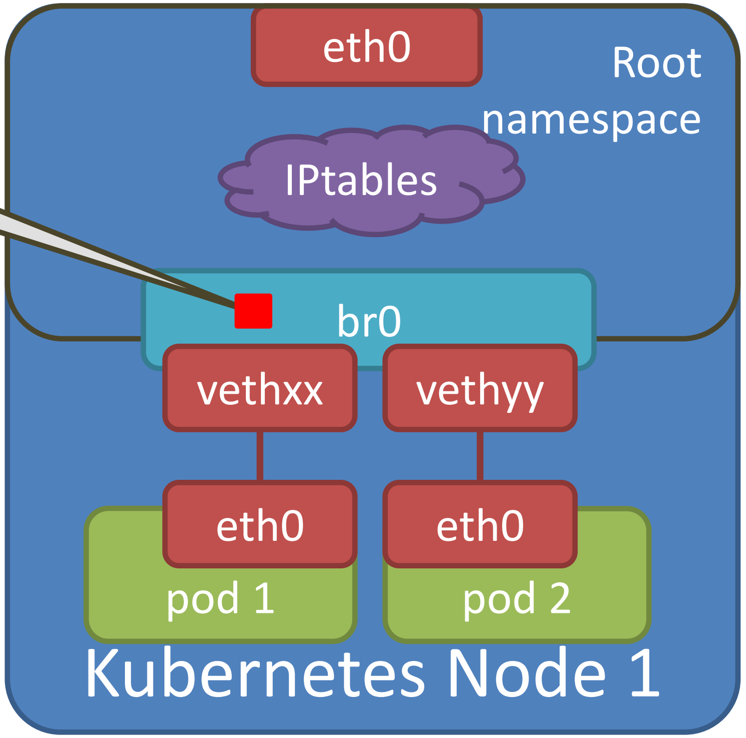
Choose one... | Calico | Canal | Flannel | Kube-router | Romana | Weave Net | JuniperContrail/TungstenFabric

For `flannel` to work correctly, you must pass `--pod-network-cidr=10.244.0.0/16` to `kubeadm init`.

Set `/proc/sys/net/bridge/bridge-nf-call-iptables` to `1` by running `sysctl net.bridge.bridge-nf-call-iptables=1` to pass bridged IPv4 traffic to iptables' chains. This is a requirement for some CNI plugins to work, for more information please see here.

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.10.0/Documentation/kube-flannel.yml
```

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/



Life of a Packet: POD-to-Service

L3 src: svc1
L3 dst: pod1

Unfortunately, you can't do the same with OVS

You can install only one pod network per cluster.

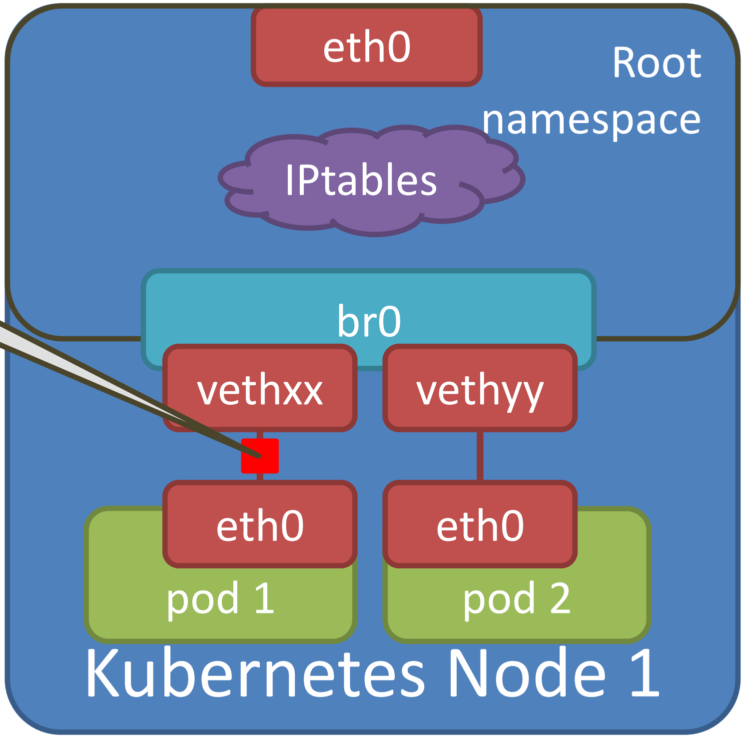
Choose one... | Calico | Canal | Flannel | Kube-router | Romana | Weave Net | JuniperContrail/TungstenFabric

For `flannel` to work correctly, you must pass `--pod-network-cidr=10.244.0.0/16` to `kubeadm init`.

Set `/proc/sys/net/bridge/bridge-nf-call-iptables` to `1` by running `sysctl net.bridge.bridge-nf-call-iptables=1` to pass bridged IPv4 traffic to iptables' chains. This is a requirement for some CNI plugins to work, for more information please see here.

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/v0.10.0/Documentation/kube-flannel.yml
```

https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/



Handling Service Communication with OVS: Option 1

Use *ct** flow rules:

- It uses the same *conntrack* kernel module as IPtables
- You can specify similar NAT rules than you would in IPtables
- For load balancing between POD backend, you can use group rules

```
table=0,ip,nw_src={pod_cidr},nw_dst={service_cidr},ct_state=-trk,action=ct(table=2)
table=0,ip,nw_src={pod_cidr},nw_dst={pod_cidr},ct_state=-trk,action=ct(table=4)

table=2,ip,nw_dst={svc1_ip},tp_dst={svc1_port},ct_state=+trk+new,action=group:1
table=2,ip,nw_dst={svc2_ip},tp_dst={svc2_port},ct_state=+trk+new,action=group:2
table=2,ct_state=+trk-new,action=table:4

table=4 contains the original switching / routing rules


group_id=1,type=select,       bucket=ct(commit,nat(dst={pod1_ip}:{pod_port}),table=4,
                              bucket=ct(commit,nat(dst={pod2_ip}:{pod_port}),table=4,
                              bucket=ct(commit,nat(dst={pod3_ip}:{pod_port}),table=4
```

* *ct* rules are actually not OpenFlow compatible

Handling Service Communication with OVS: Option 2

Use *stateless* NAT rules:

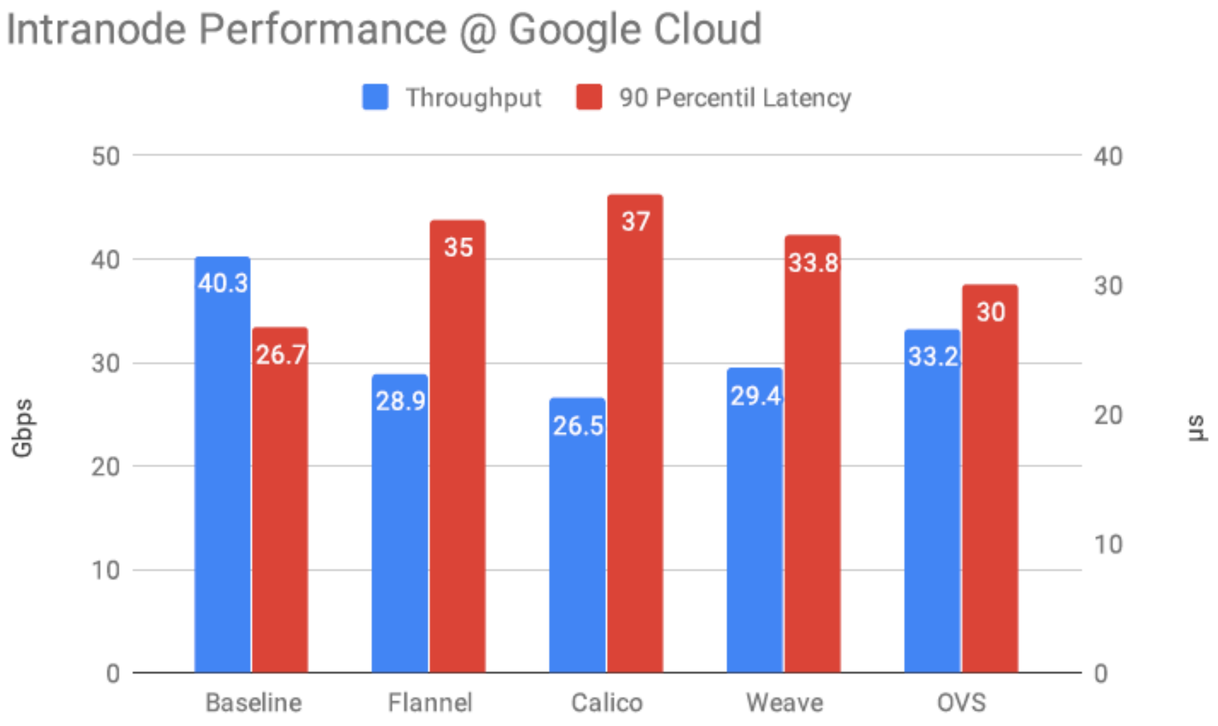
- If we see a Service IP we switch the destination IP to a POD backend
- But at the same time we modify the source IP to a shifted domain (e.g. 10.244.x.y → 172.24.x.y)
- This way we don't use any kernel specific rules which allows the integration into user-space (e.g. DPDK)

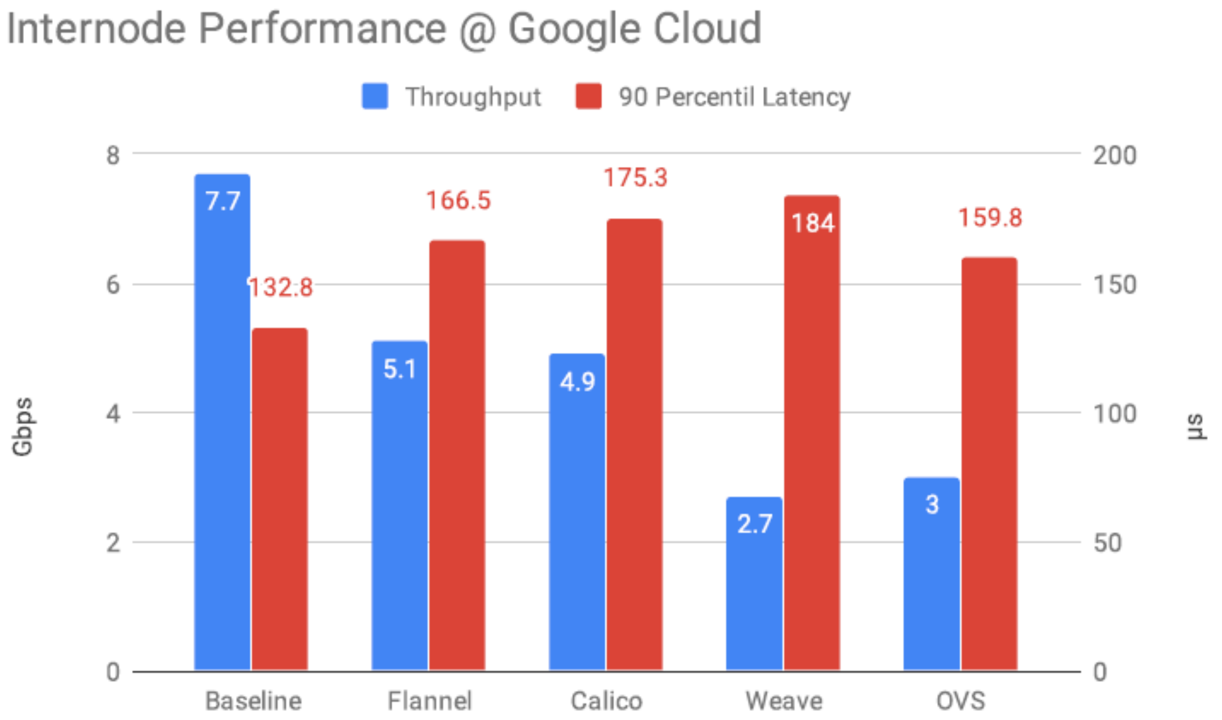
```
table=0,ip,nw_src={pod_cidr},nw_dst={service_cidr},action=table:2
table=0,ip,nw_src={pod_cidr},nw_dst={shifted_pod_cidr},action=table:3
table=0,ip,nw_src={pod_cidr},nw_dst={pod_cidr},action=table:4

table=2,ip,nw_dst={svc1_ip},tp_dst={svc1_port},actions=load:44056->NXM_OF_IP_SRC[16..31],group:1

table=3,ip,nw_src={pod1_ip},tp_src={pod_port},actions=mod_nw_src:{svc1_ip},mod_tp_src:{svc1_port}
                                                       ,load:2804->NXM_OF_IP_DST[16..31],resubmit:4

table=4 contains the original switching / routing rules


group_id=1,type=select,       bucket=mod_nw_dst:{pod1_ip},mod_tp_dst:{pod_port},resubmit=4,
                              bucket=mod_nw_dst:{pod2_ip},mod_tp_dst:{pod_port},resubmit=4,
                              bucket=mod_nw_dst:{pod3_ip},mod_tp_dst:{pod_port},resubmit=4
```

* *NXM* stands for Nicira eXtended Match rules which are also not OpenFlow compatible

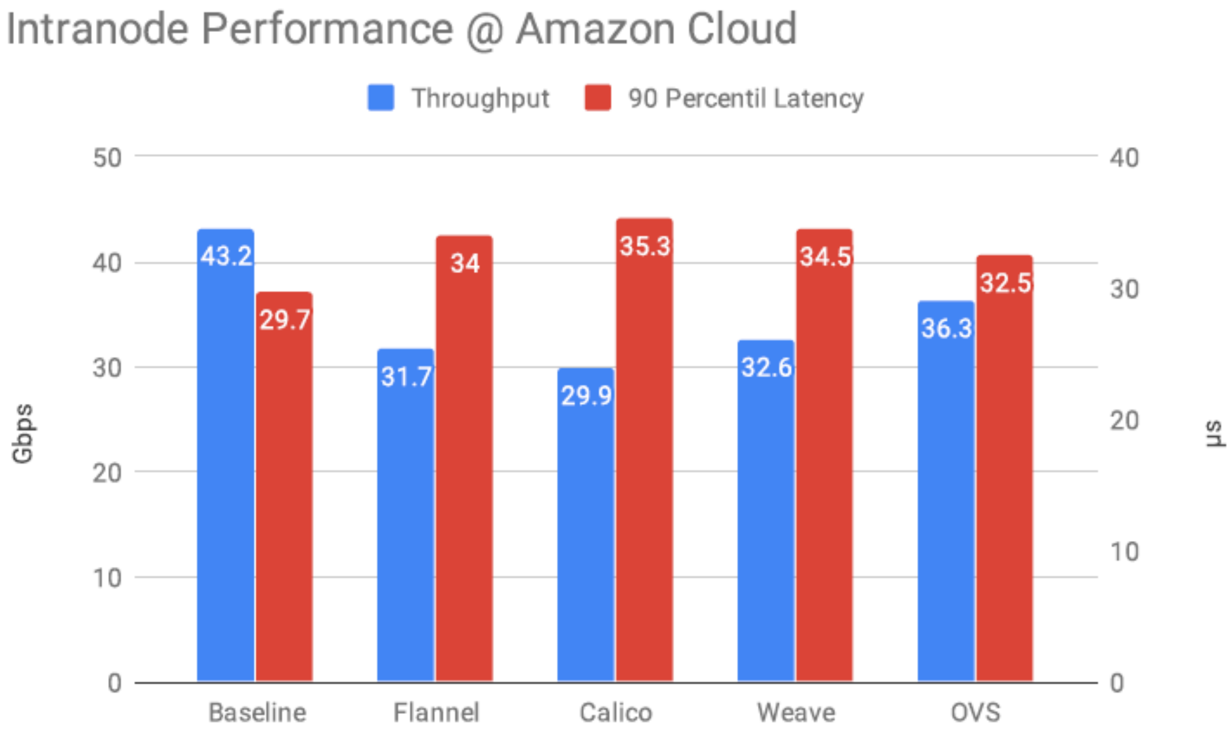
Finally, it’s demo time ☺

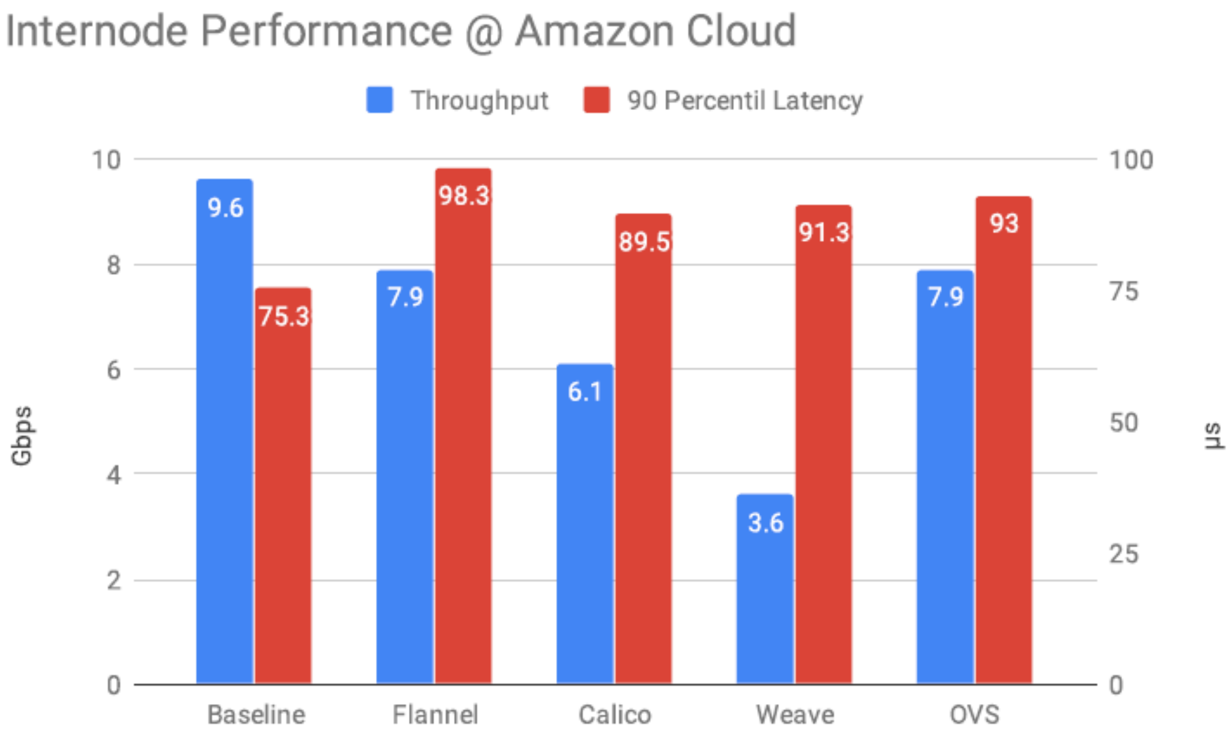
Performance Comparison: Google Cloud



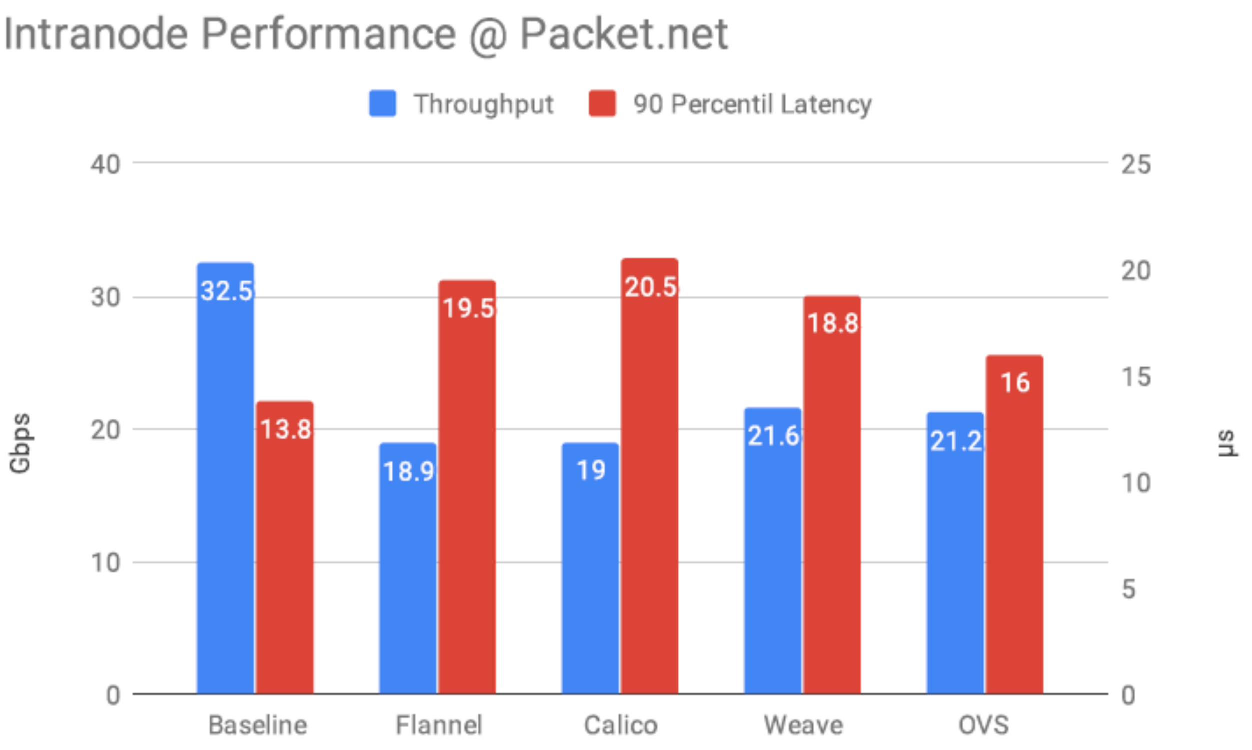


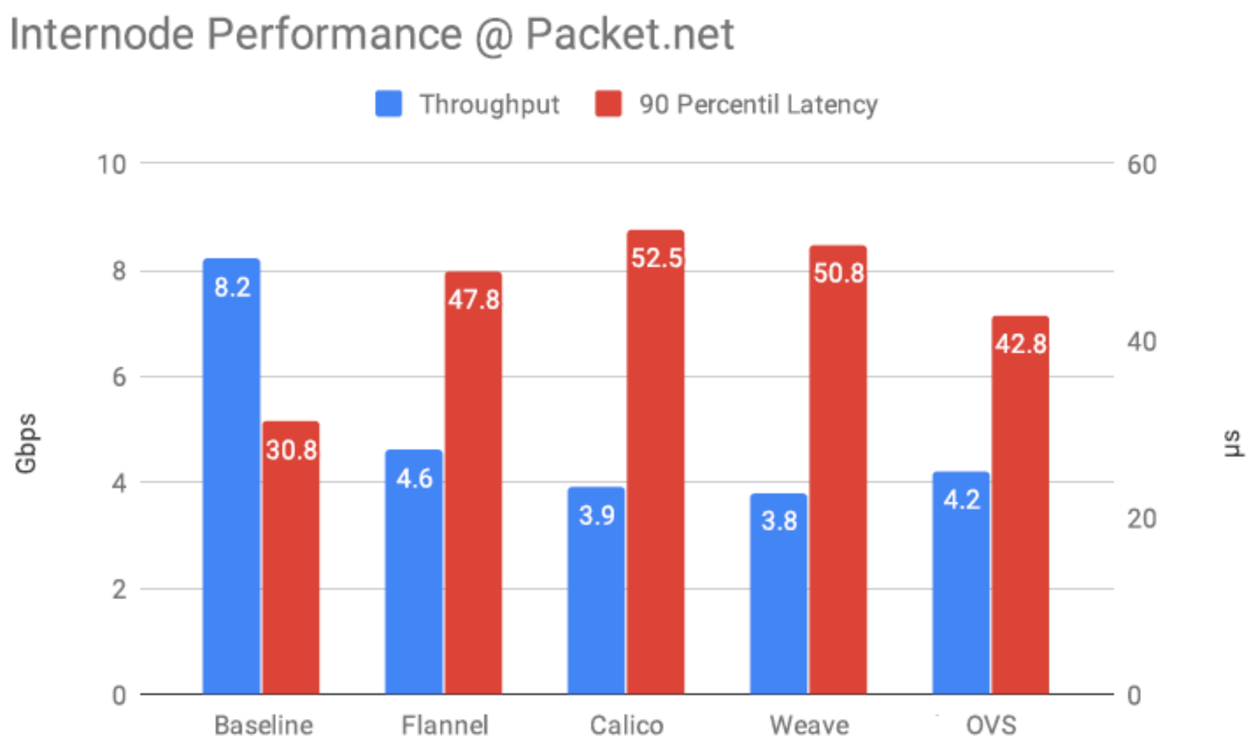
Performance Comparison: Amazon Cloud





Performance Comparison: Packet





Kubernetes Networking with Open vSwitch

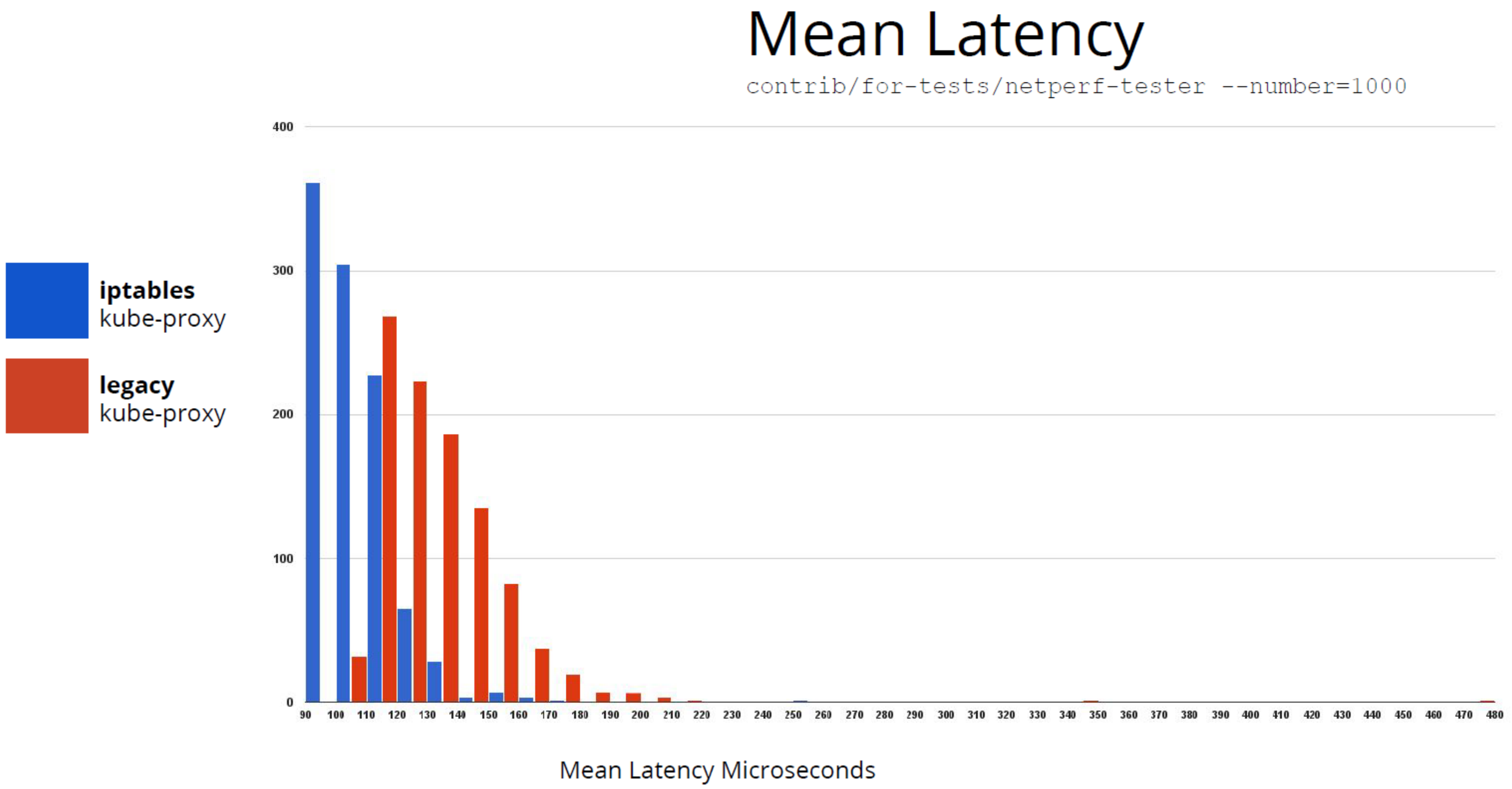
Pure OVS solution

- CNI binary attaching PODs to and OVS bridge
- POD-to-POD and POD-to-Service communication with OpenFlow rules
- Enhanced monitoring using Prometheus and OVS-exporter
- Speed and latency is comparable with leading plugins (Flannel, Calico, Weave)
- DPDK integration possibility
- 100% open source: https://github.com/dunlinplugin

dunlin.io

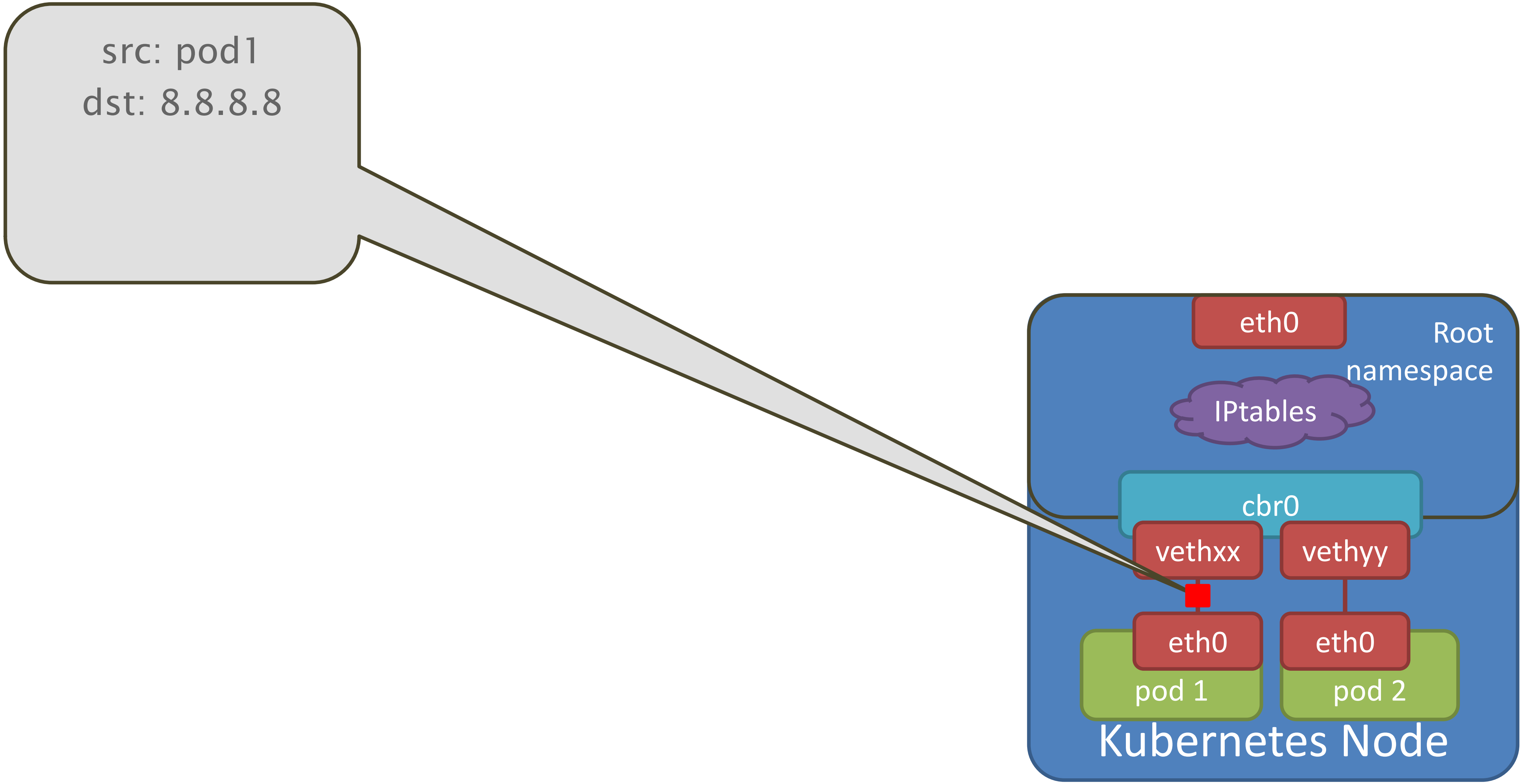
Backup Slides

IPtables Latency by Google

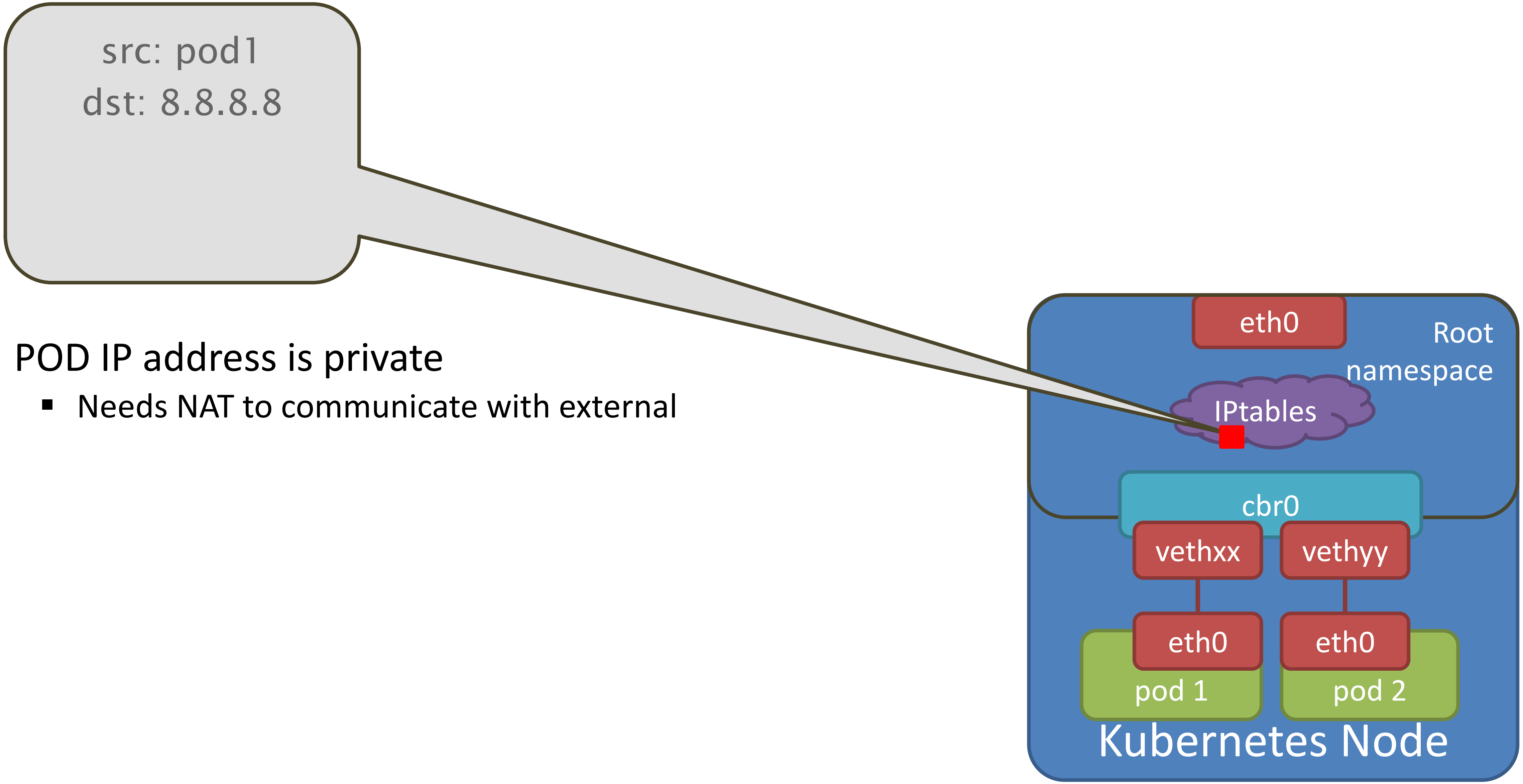


Pod to External Communication in Kubernetes

Life of a packet: pod-to-external



Life of a packet: pod-to-external



POD IP address is private

- Needs NAT to communicate with external

Life of a packet: pod-to-external

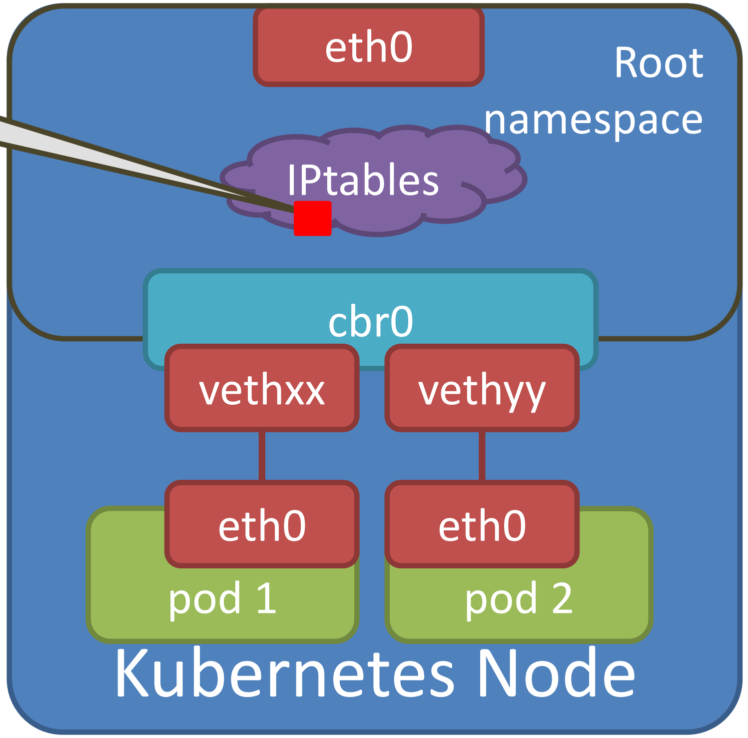
~~src: pod1~~
src: NodeIP
dst: 8.8.8.8

MASQUERADE

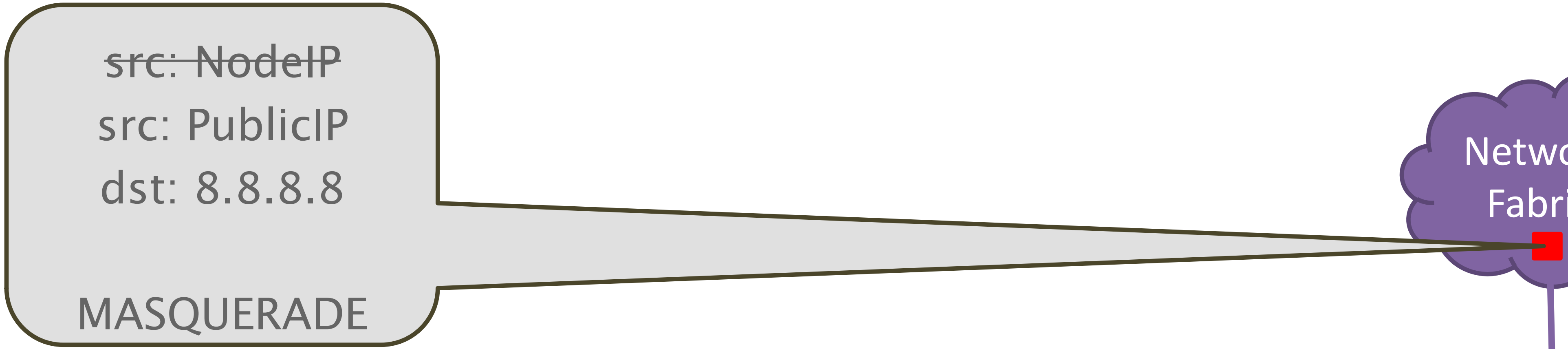
POD IP address is private

- Needs NAT to communicate with external

Node IPs are usually also private



Life of a packet: pod-to-external

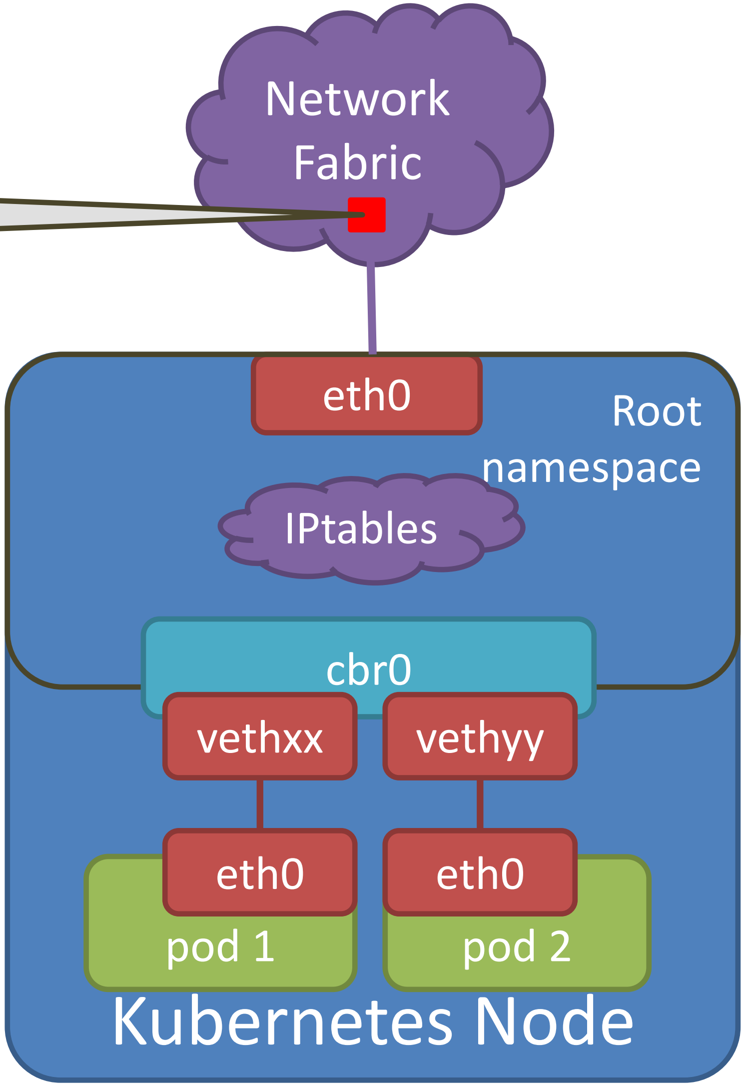


POD IP address is private

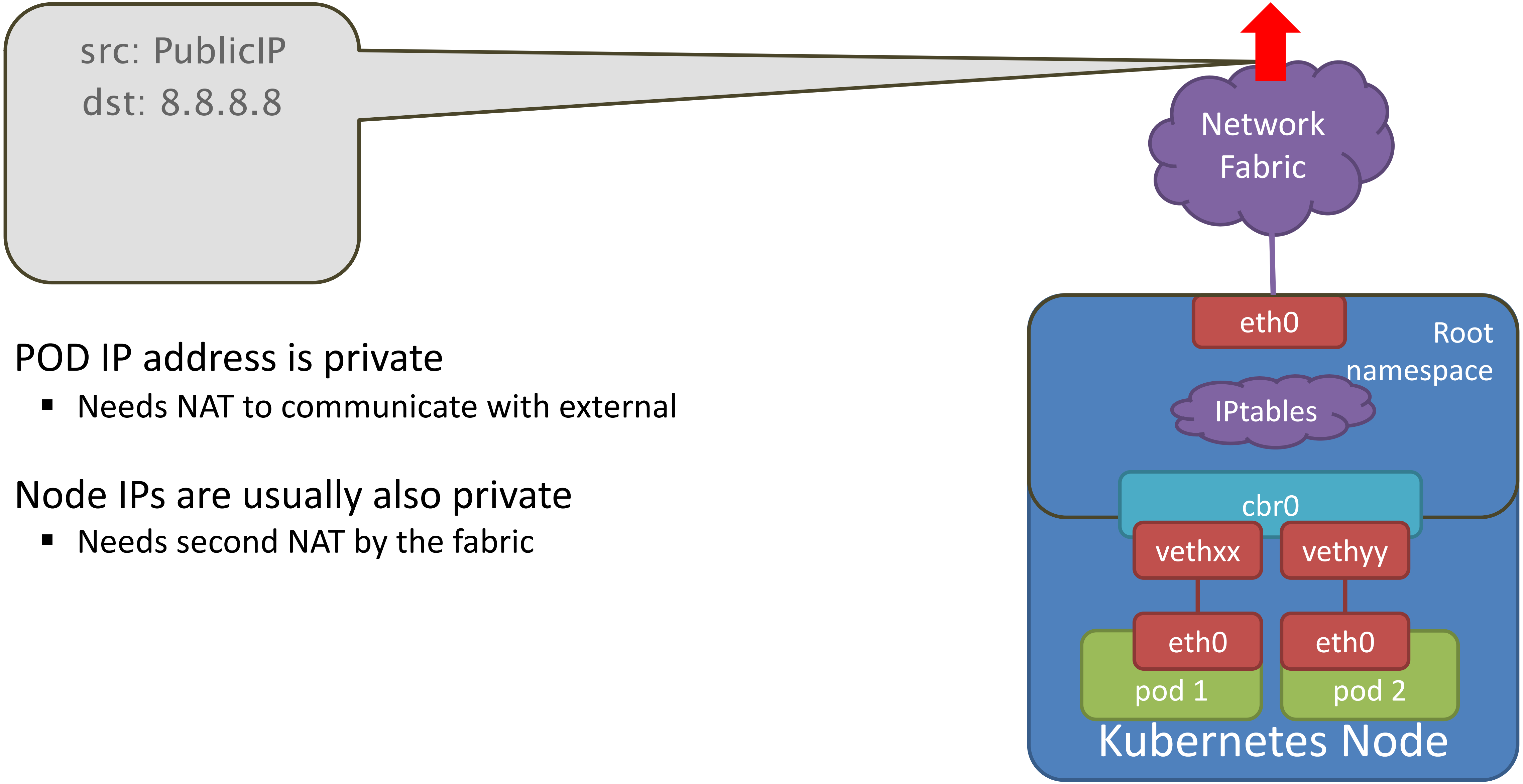
- Needs NAT to communicate with external

Node IPs are usually also private

- Needs second NAT by the fabric



Life of a packet: pod-to-external



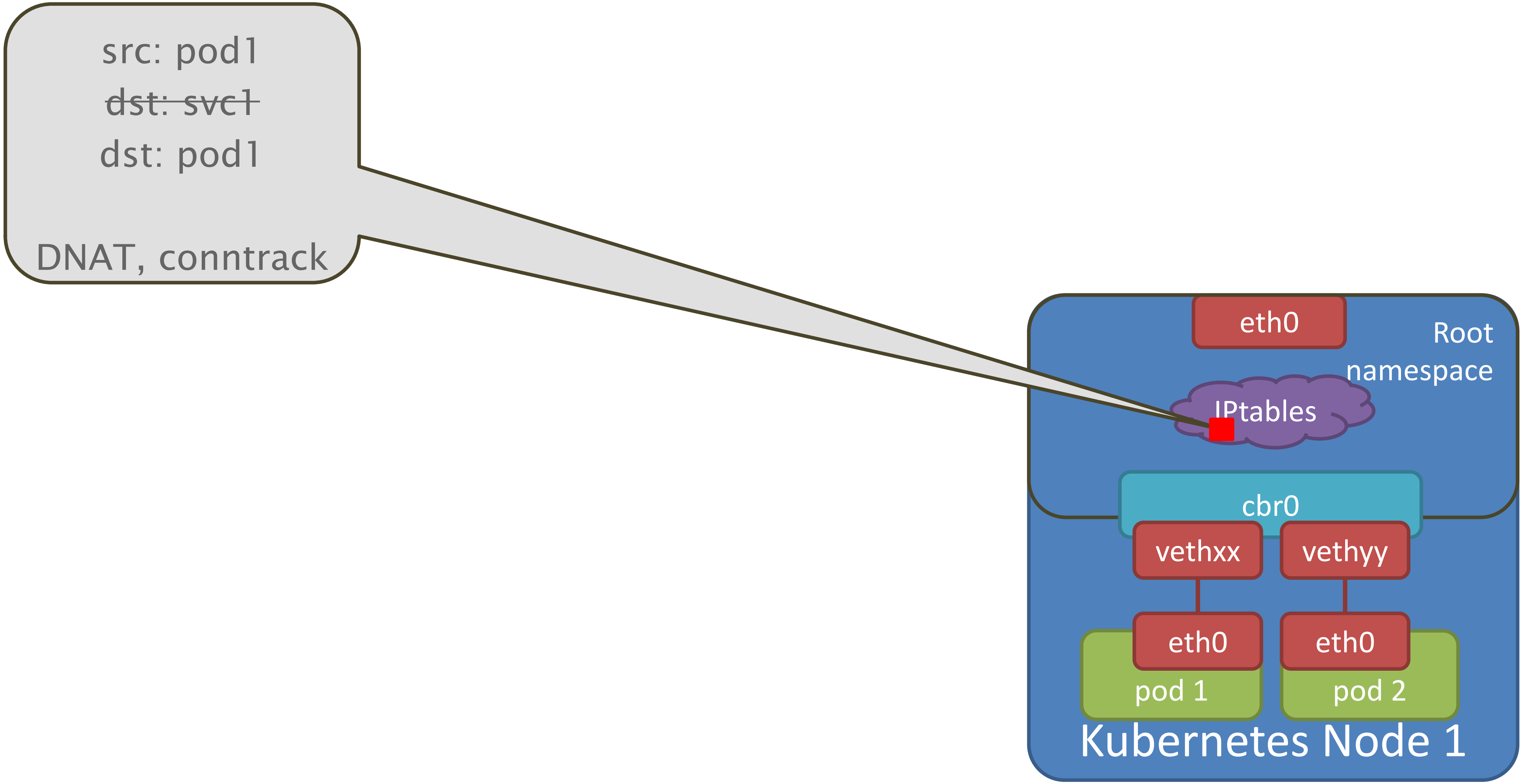
POD IP address is private

- Needs NAT to communicate with external

Node IPs are usually also private

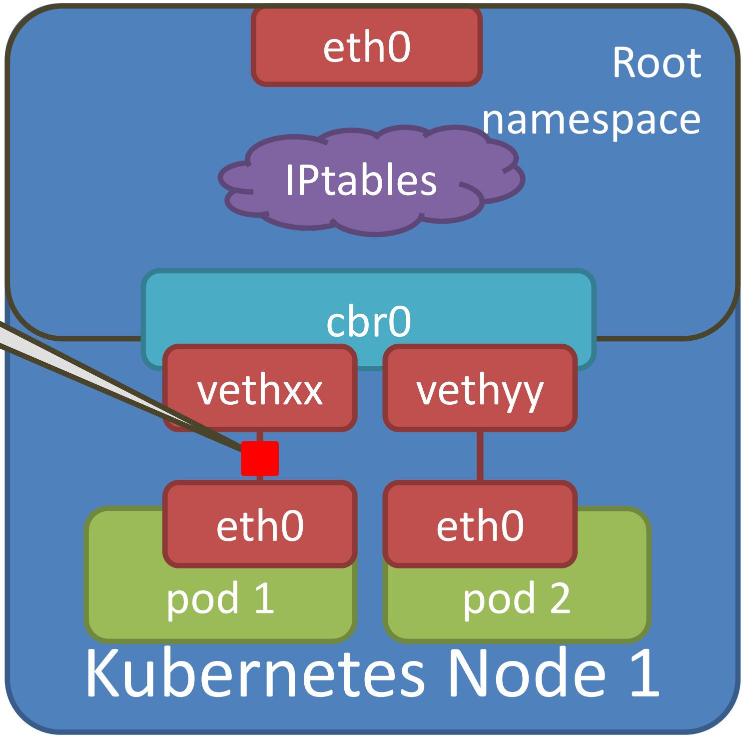
- Needs second NAT by the fabric

The Hairpin Problem

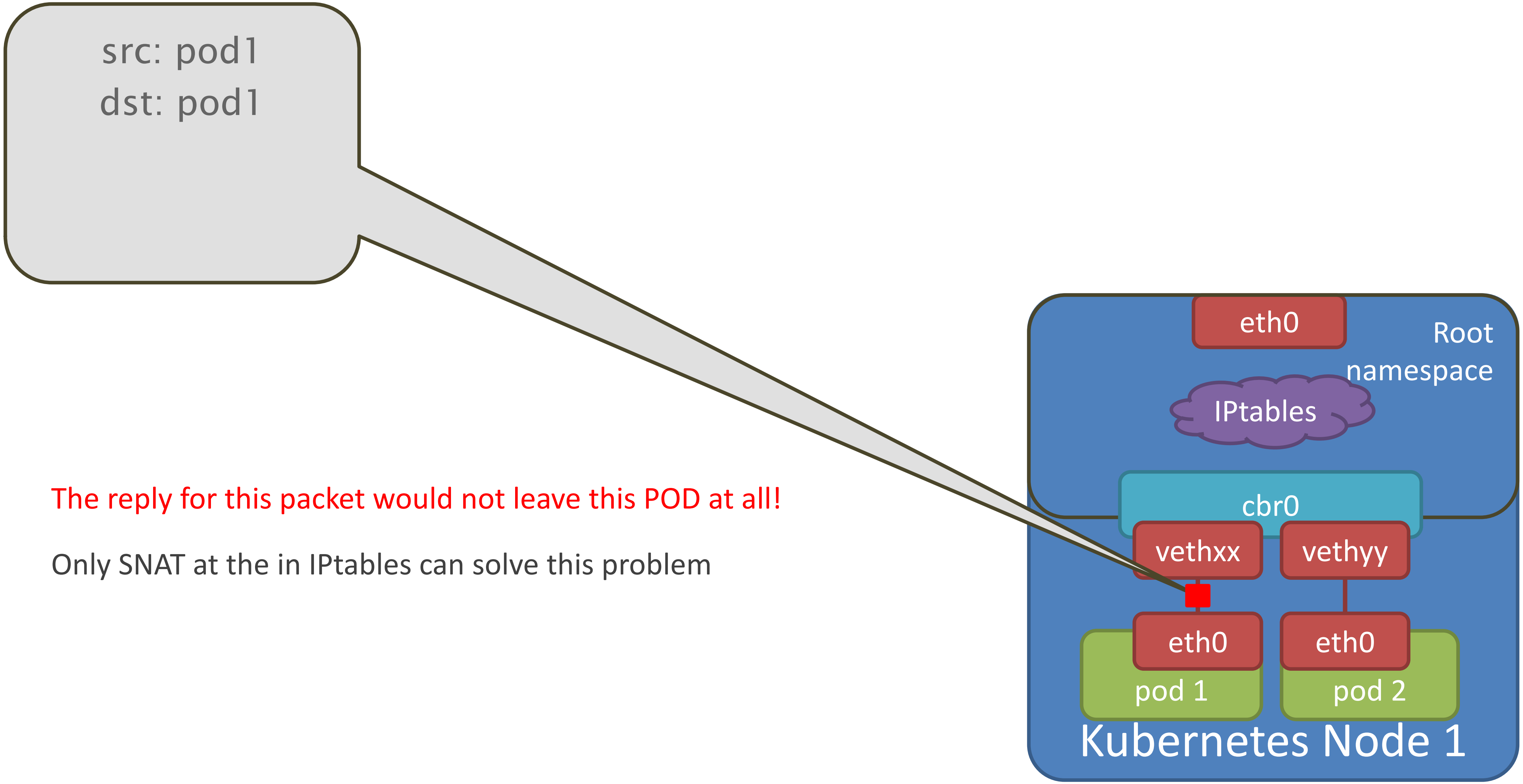


The Hairpin Problem

src: pod1
dst: pod1



The Hairpin Problem



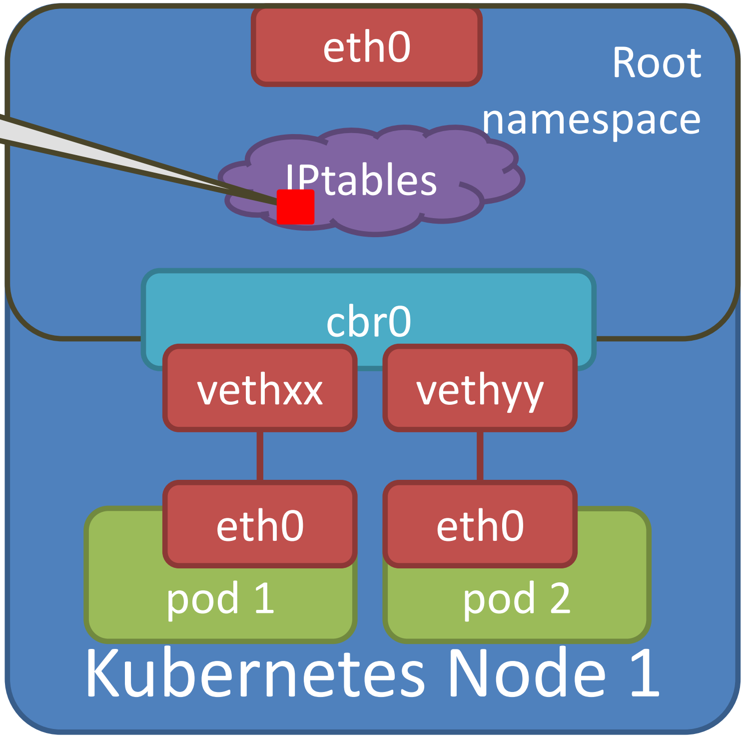
The reply for this packet would not leave this POD at all!

Only SNAT at the in IPtables can solve this problem

The Hairpin Problem

~~src: pod1~~
src: cbr0
~~dst: svc1~~
dst: pod1
DNAT, conntrack

weaveworks



External to Internal Communication in Kubernetes

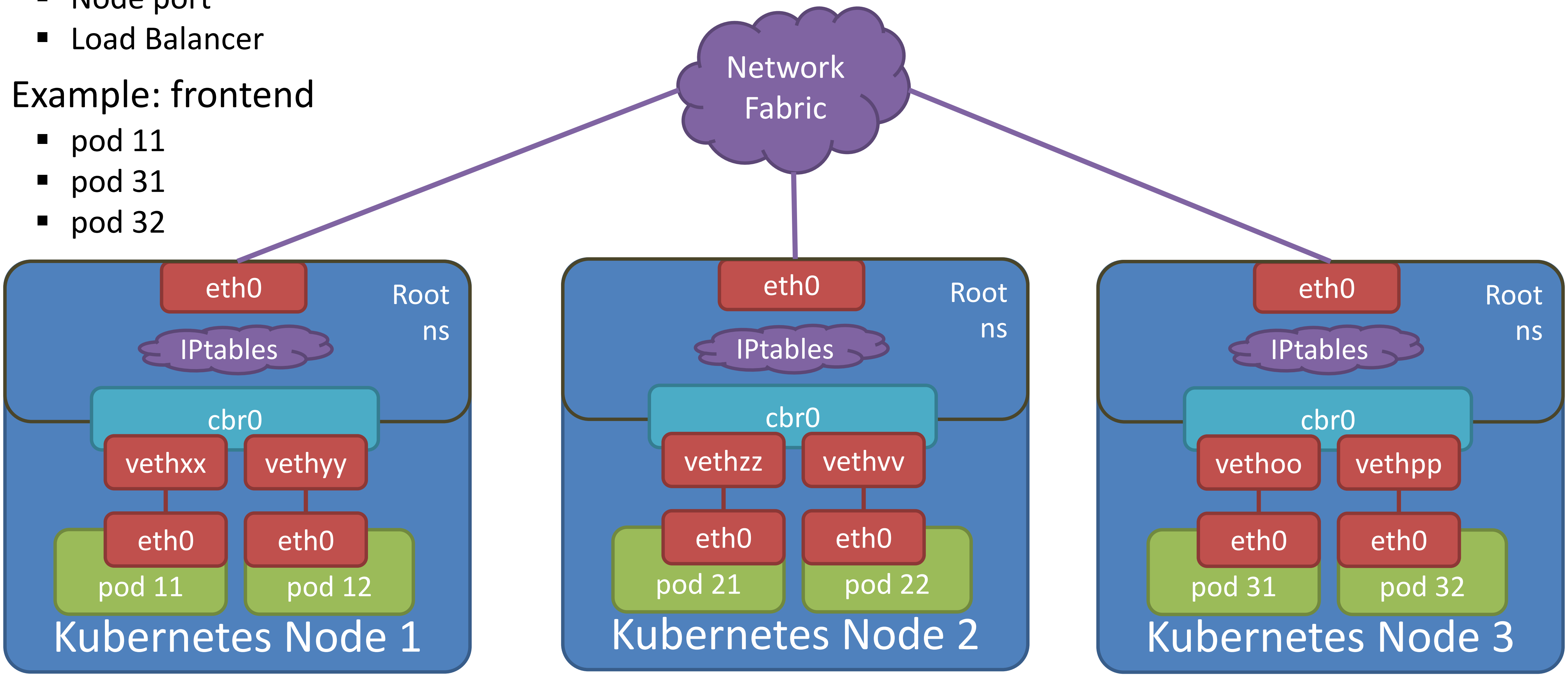
External-to-Internal Traffic

Serveices can be exposed to the outside by

- Node port
- Load Balancer

Example: frontend

- pod 11
- pod 31
- pod 32



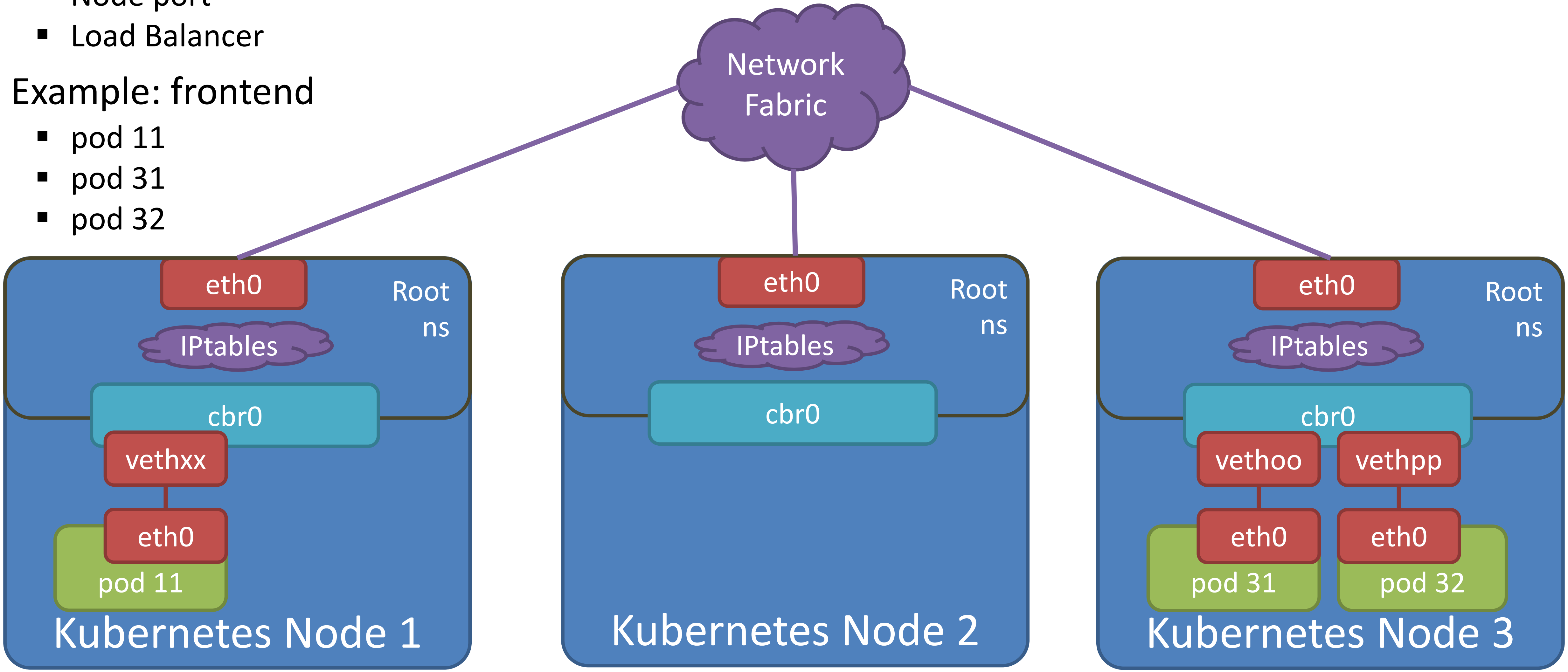
External-to-Internal Traffic

Serveices can be exposed to the outside by

- Node port
- Load Balancer

Example: frontend

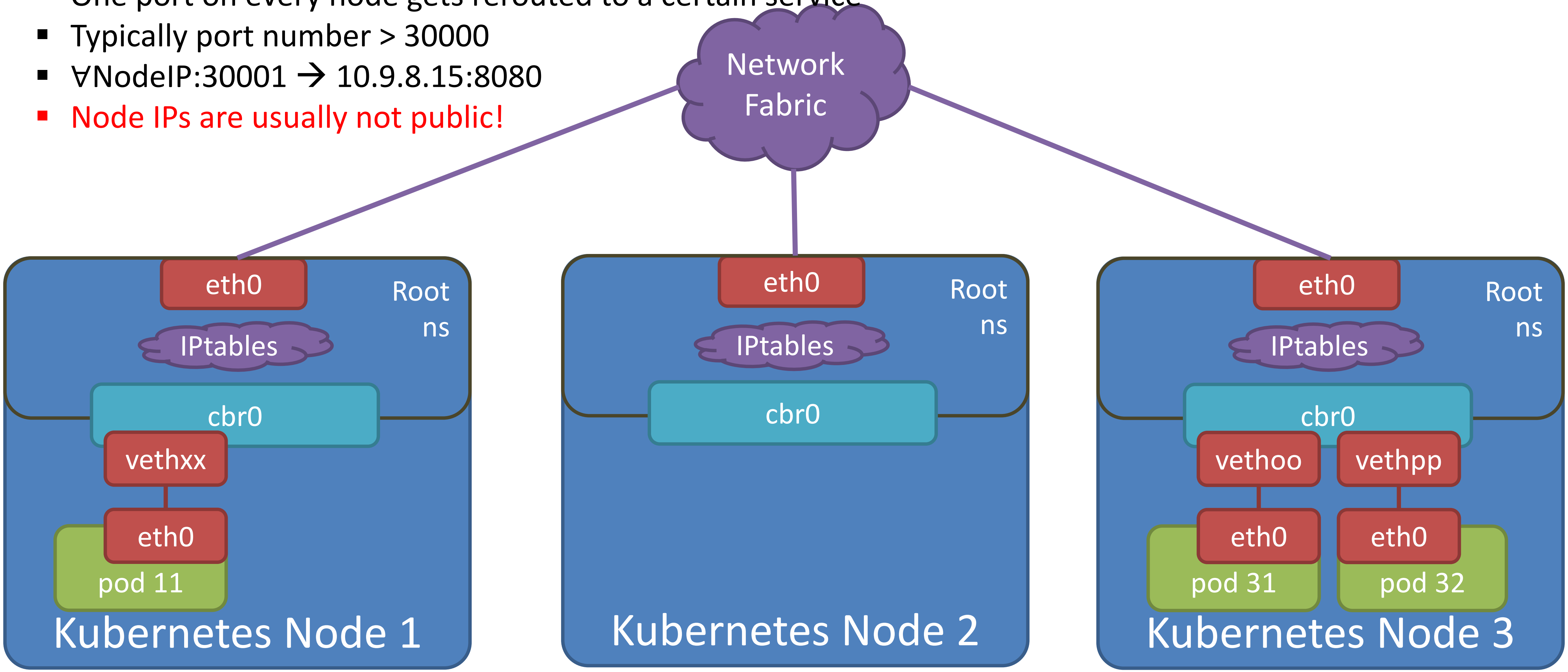
- pod 11
- pod 31
- pod 32



External-to-Internal Traffic

Node port

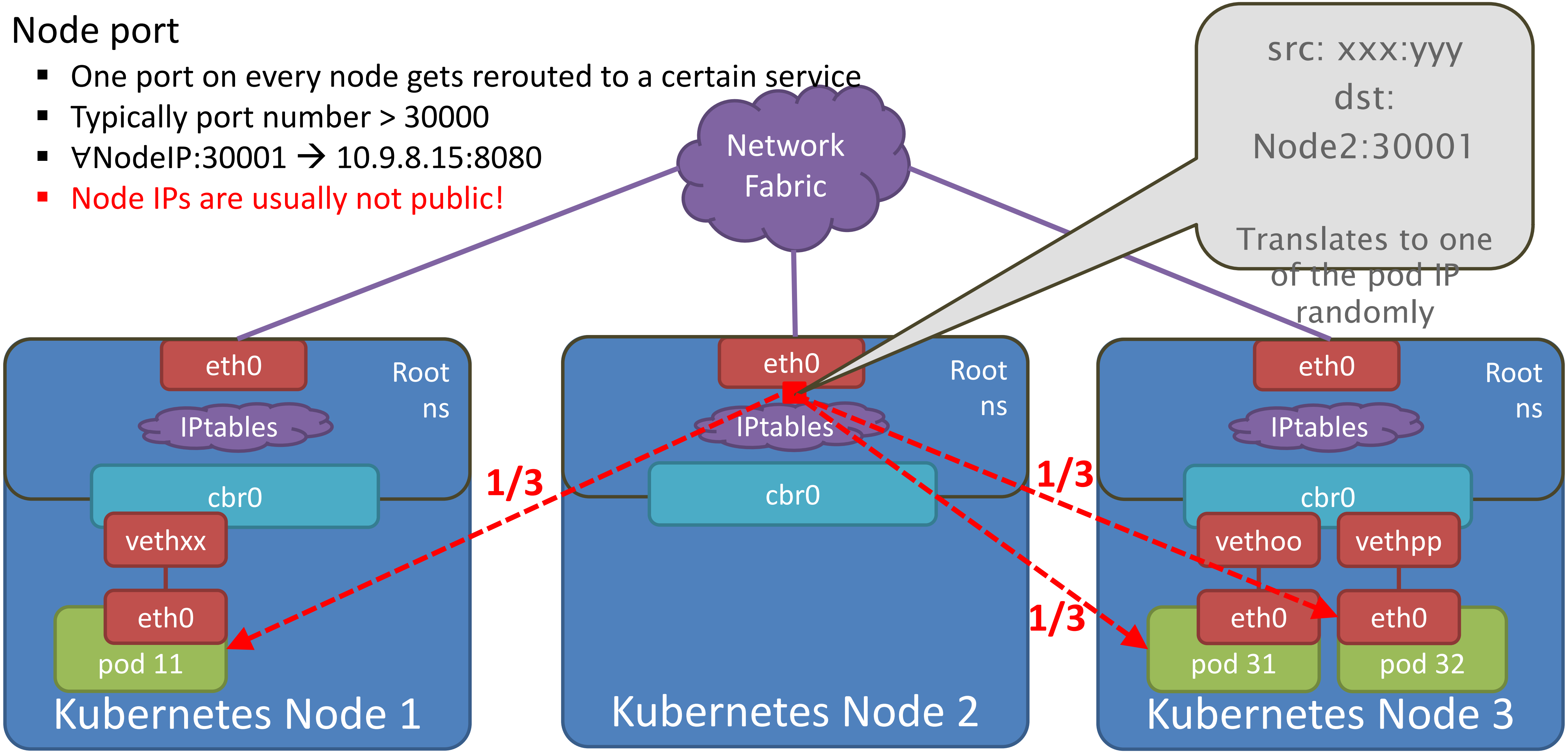
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!



External-to-Internal Traffic

Node port

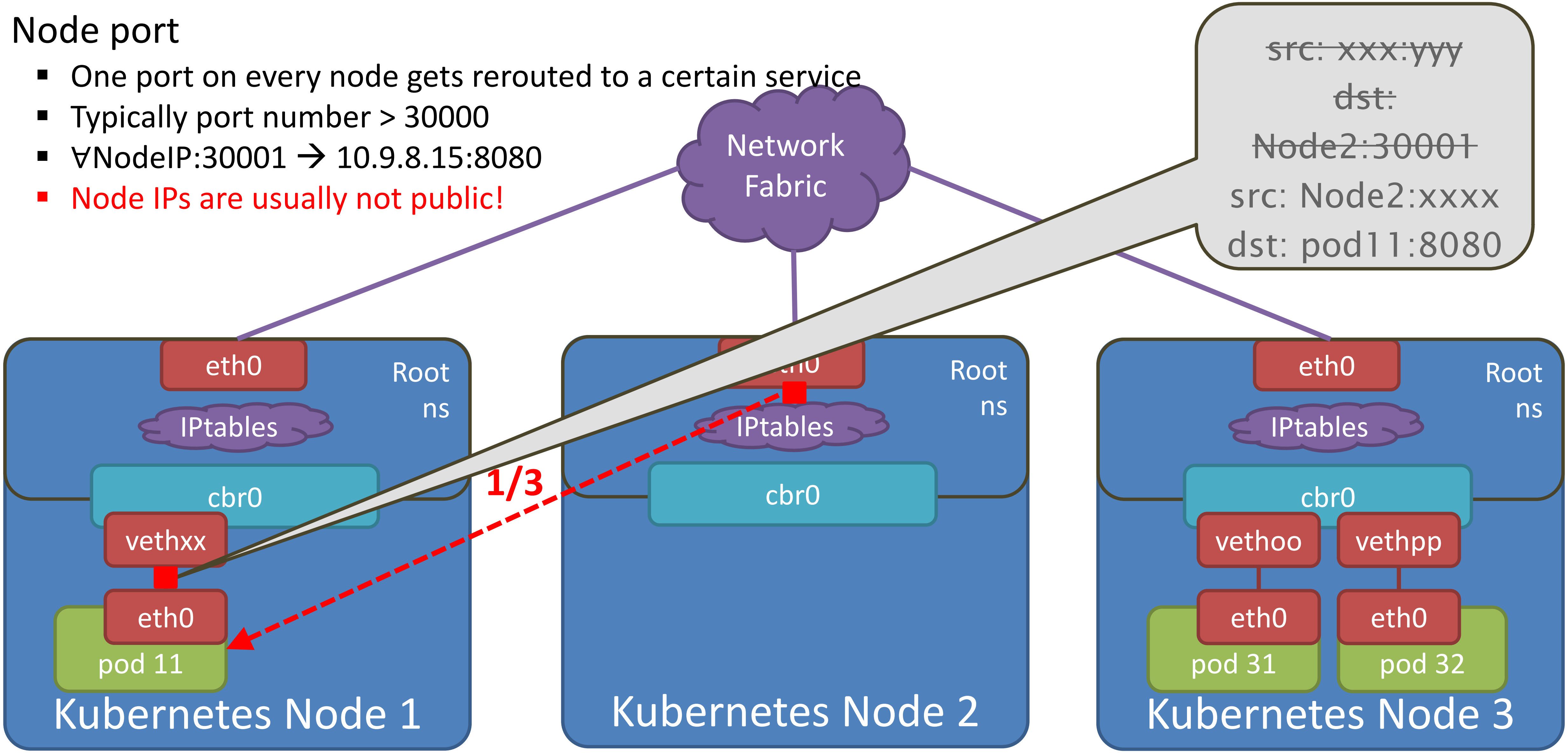
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!



External-to-Internal Traffic

Node port

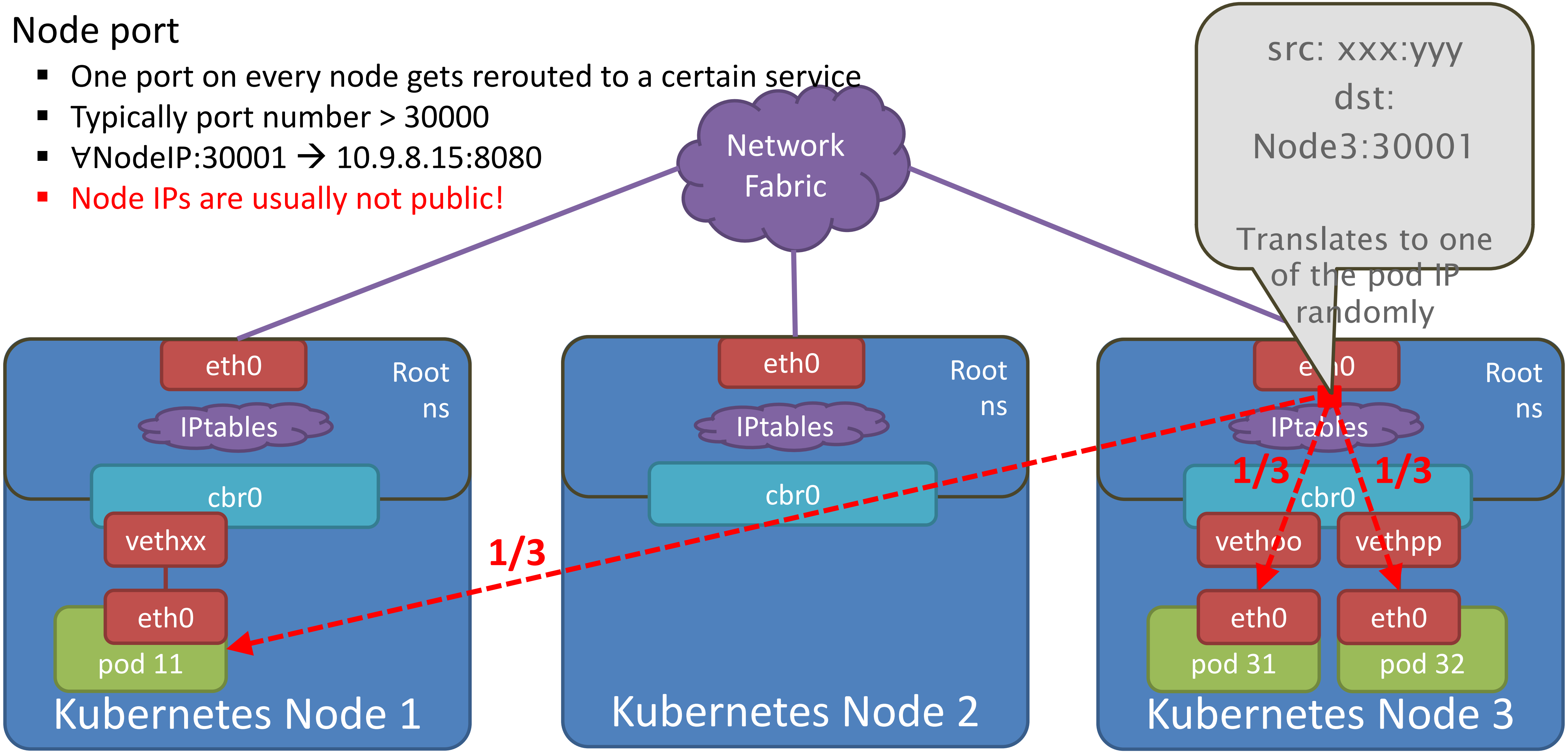
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!



External-to-Internal Traffic

Node port

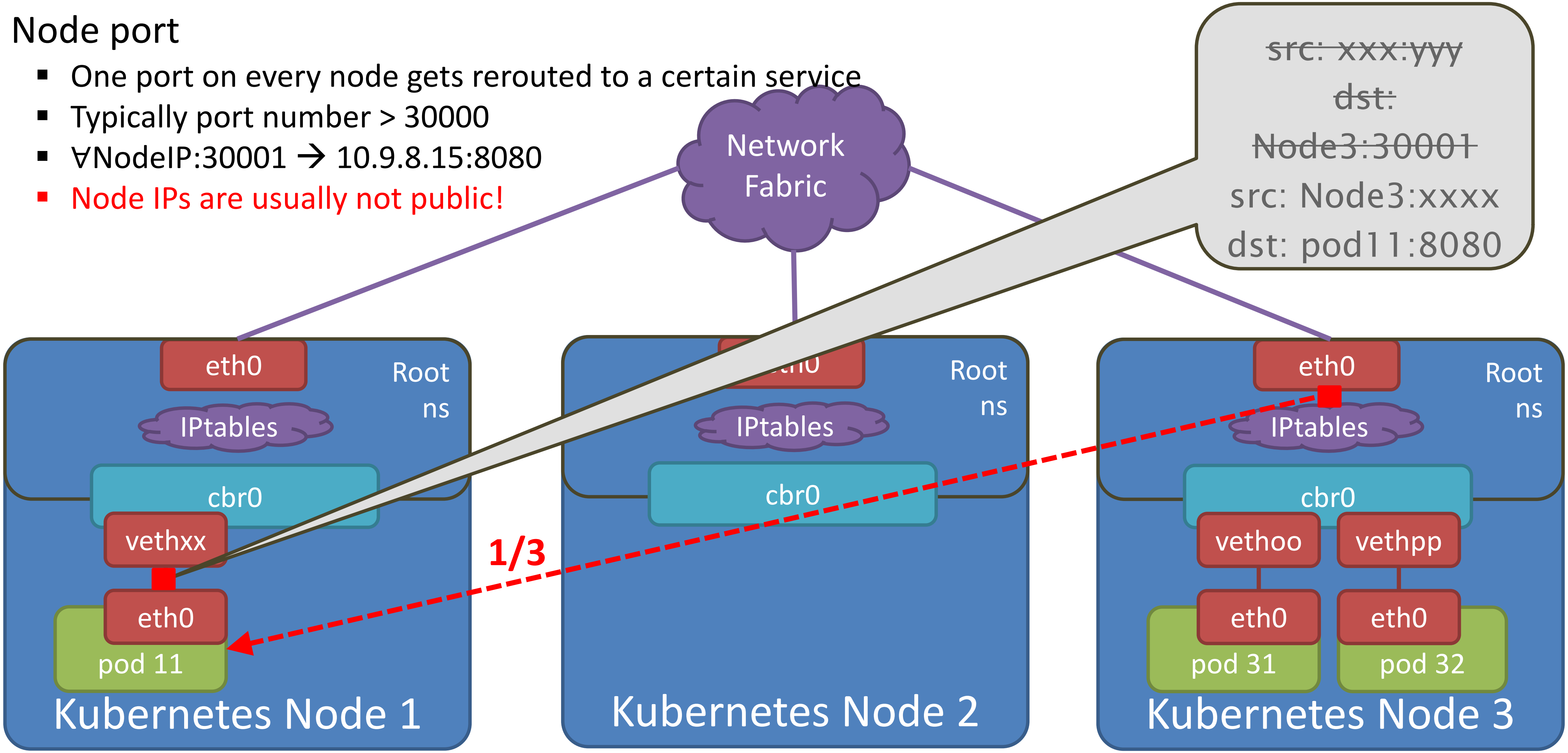
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!



External-to-Internal Traffic

Node port

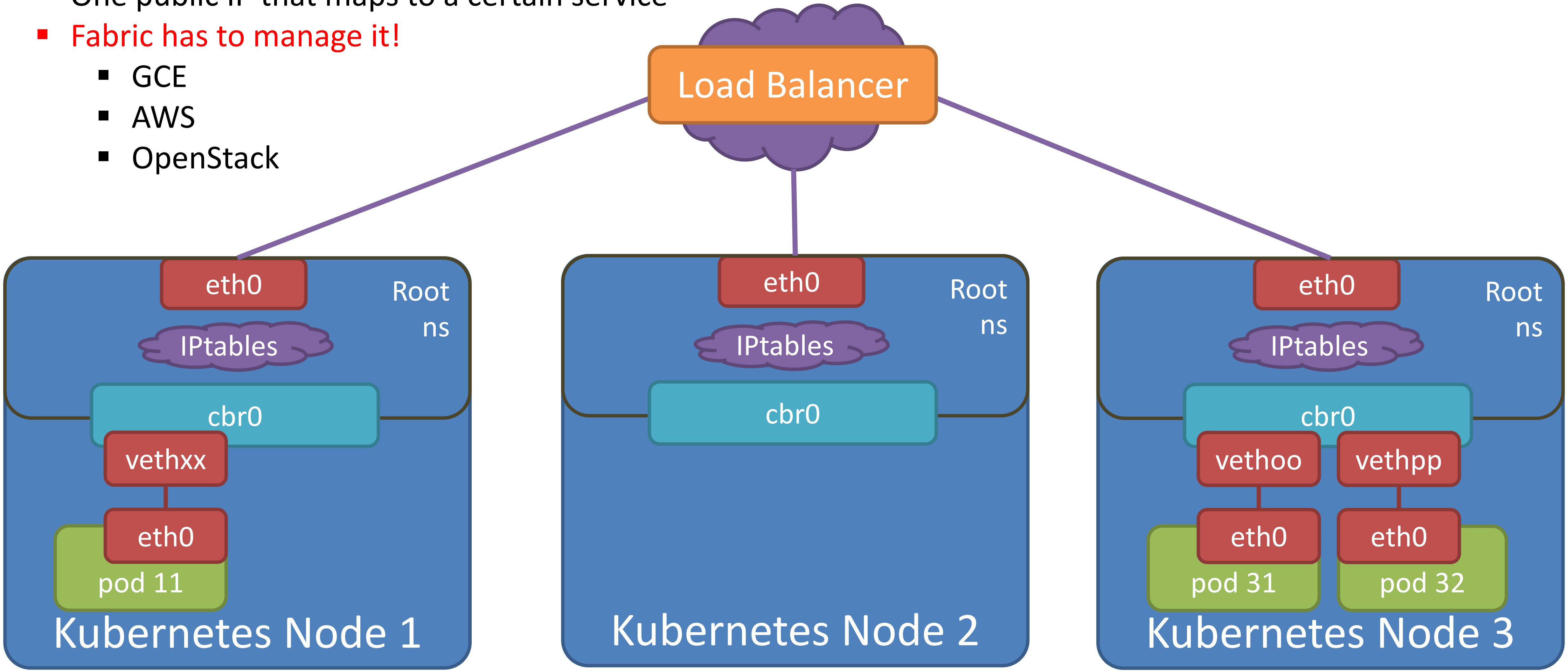
- One port on every node gets rerouted to a certain service
- Typically port number > 30000
- ∀NodeIP:30001 → 10.9.8.15:8080
- Node IPs are usually not public!



External-to-Internal Traffic

Load Balancer

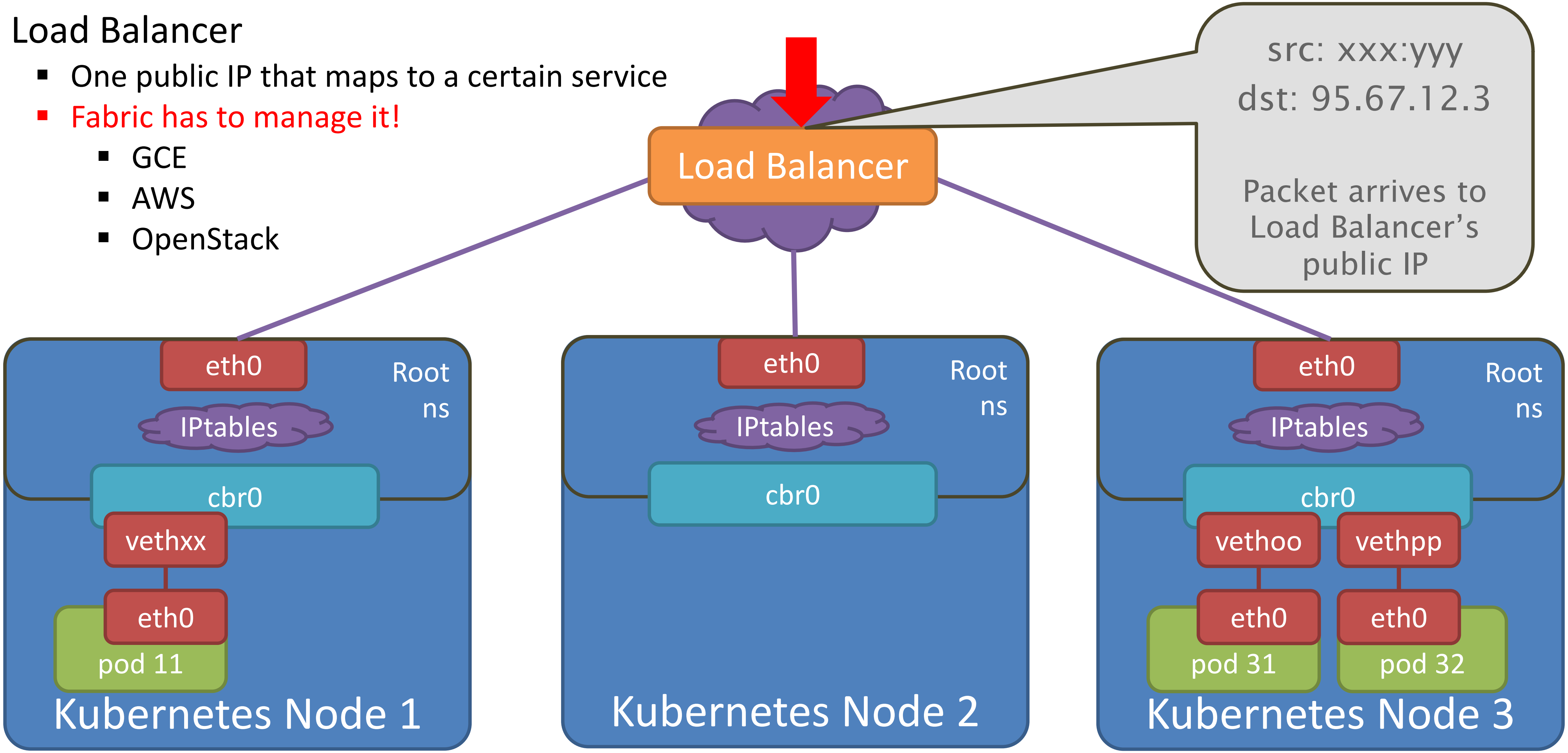
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack



External-to-Internal Traffic

Load Balancer

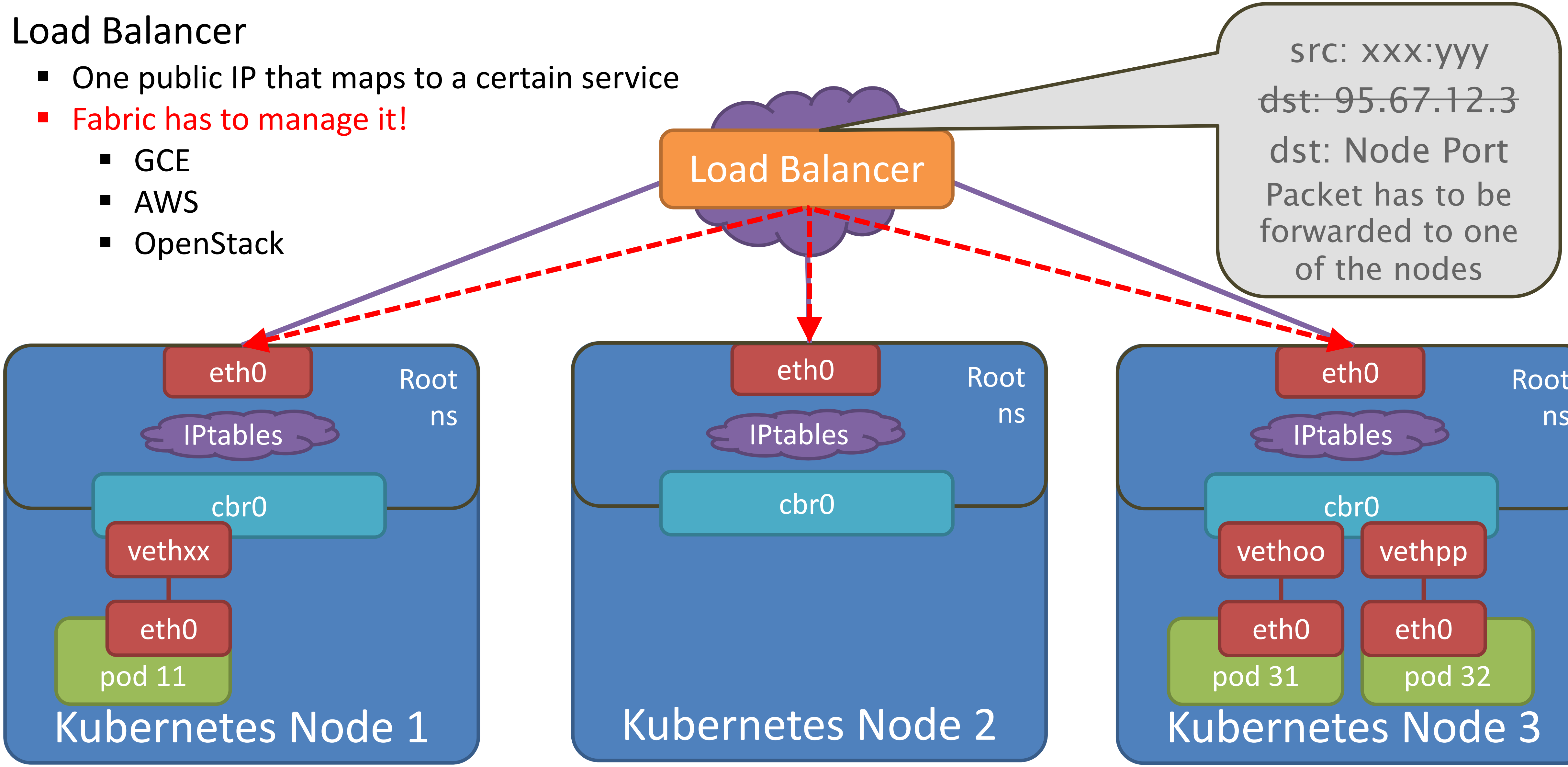
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack



External-to-Internal Traffic

Load Balancer

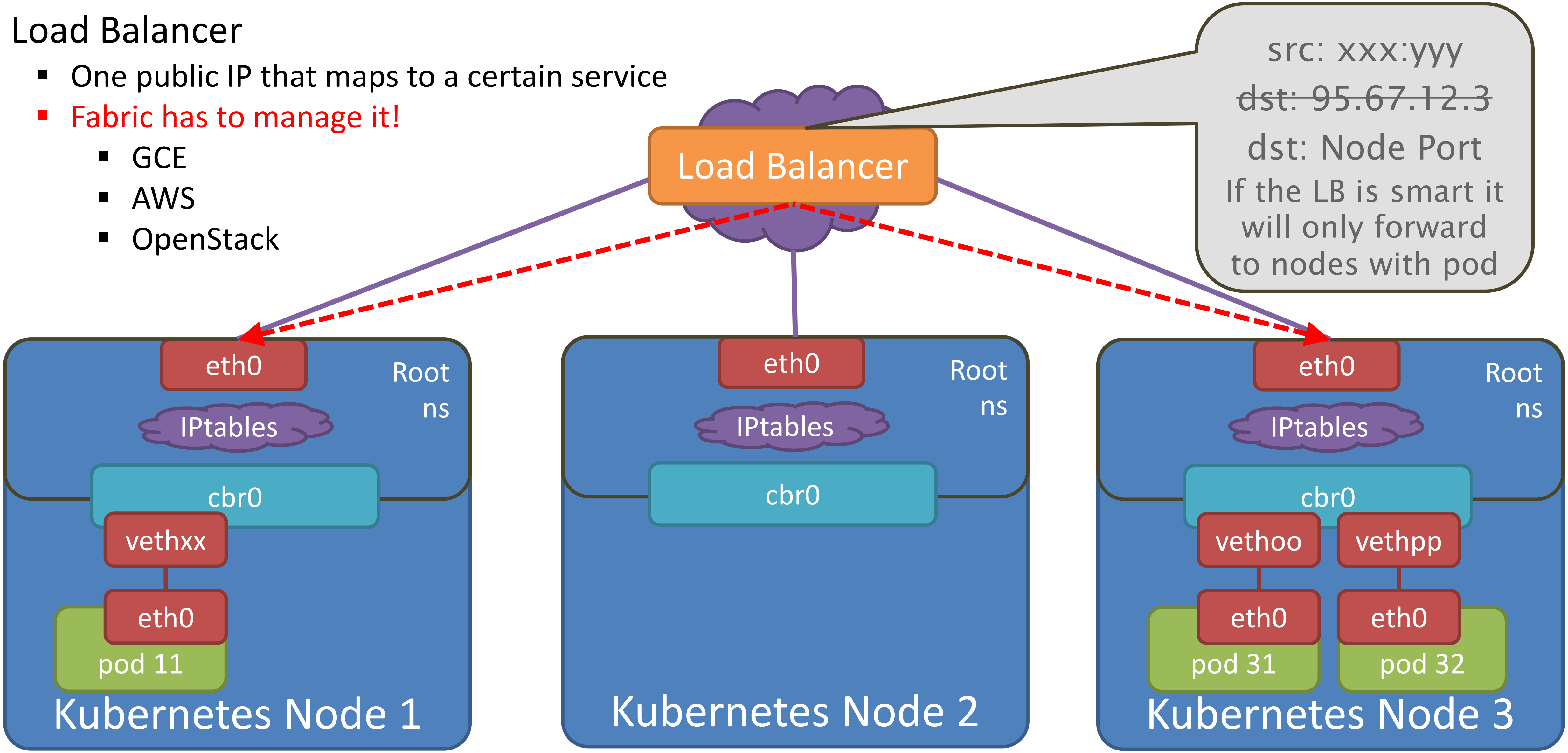
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack



External-to-Internal Traffic

Load Balancer

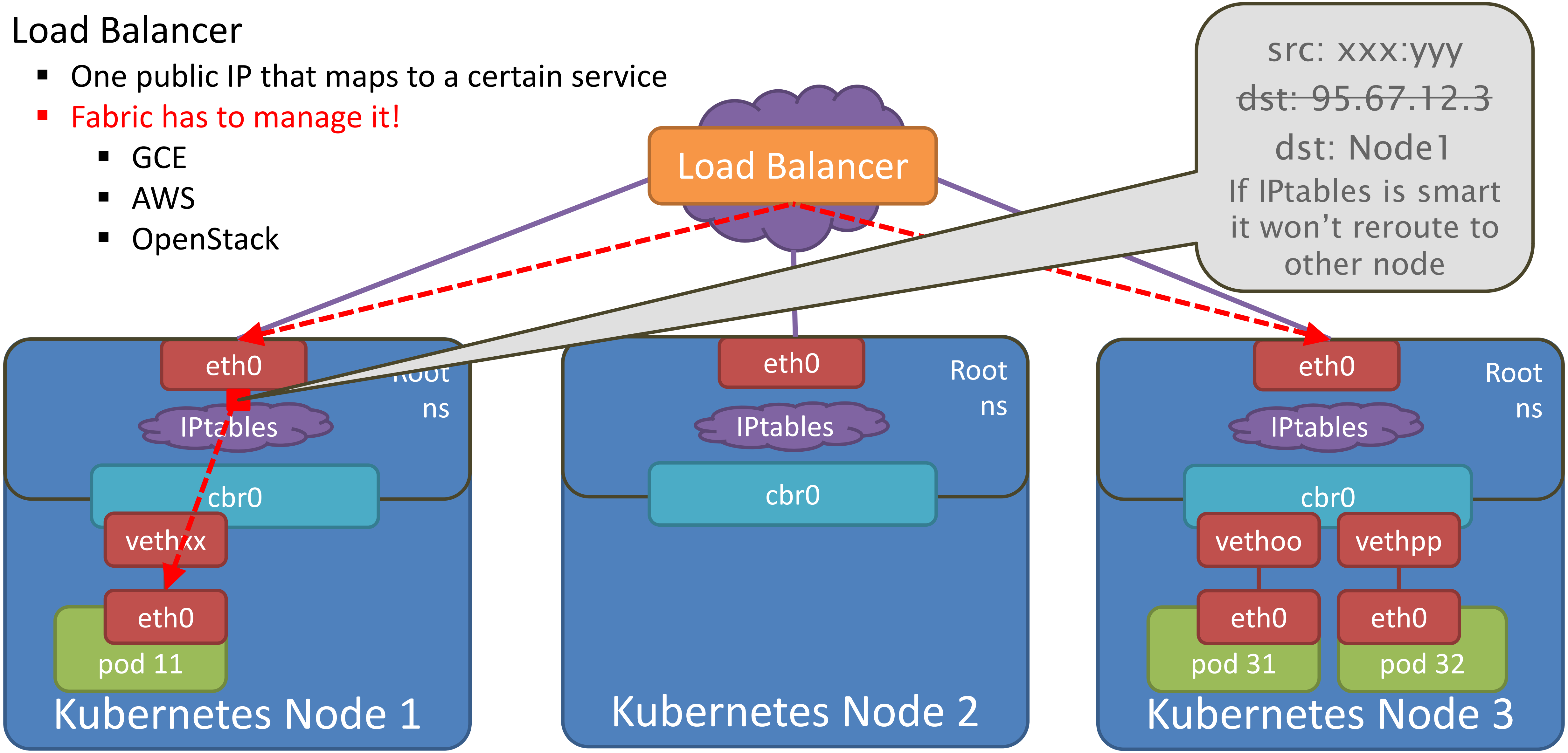
- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack



External-to-Internal Traffic

Load Balancer

- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack



External-to-Internal Traffic

Load Balancer

- One public IP that maps to a certain service
- Fabric has to manage it!
  - GCE
  - AWS
  - OpenStack