Life of a Packet

KubeCon Europe
2017

Michael Rubin <mrubin@google.com>
TL/TLM in GKE/Kubernetes
github.com/matchstick

Kubernetes is about clusters

Because of that, networking is pretty important

Most of Kubernetes centers on network concepts

Our job is to make sure your applications can communicate:

- With each other
- With the world outside your cluster
- Only where you want

The IP-per-pod model

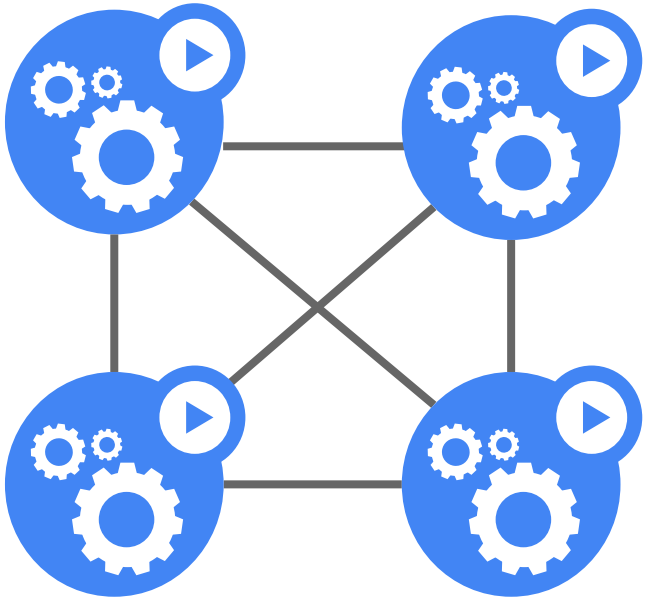
Every pod has a real IP address

This is different from the out-of-the-box model Docker offers

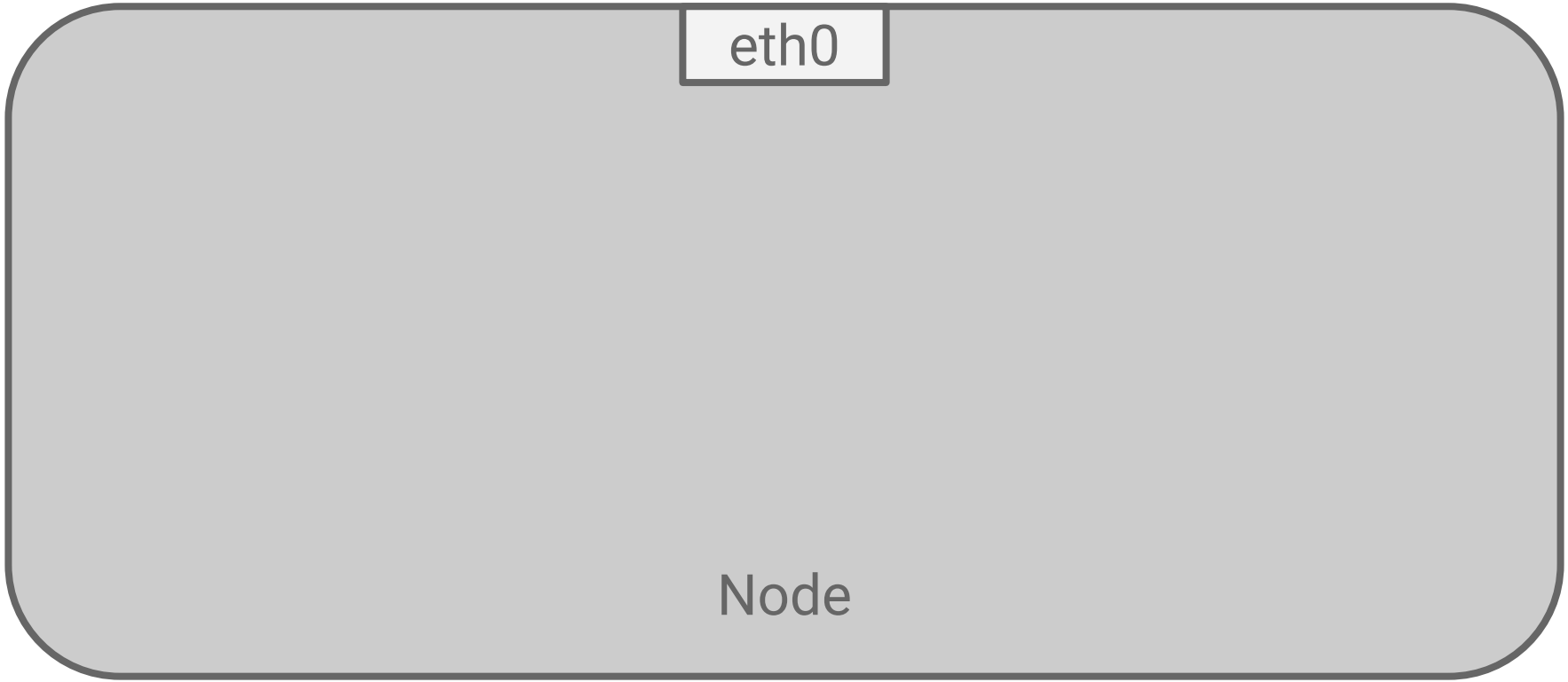
- No machine-private IPs
- No port-mapping

Pod IPs are accessible from other pods, regardless of which VM they are on

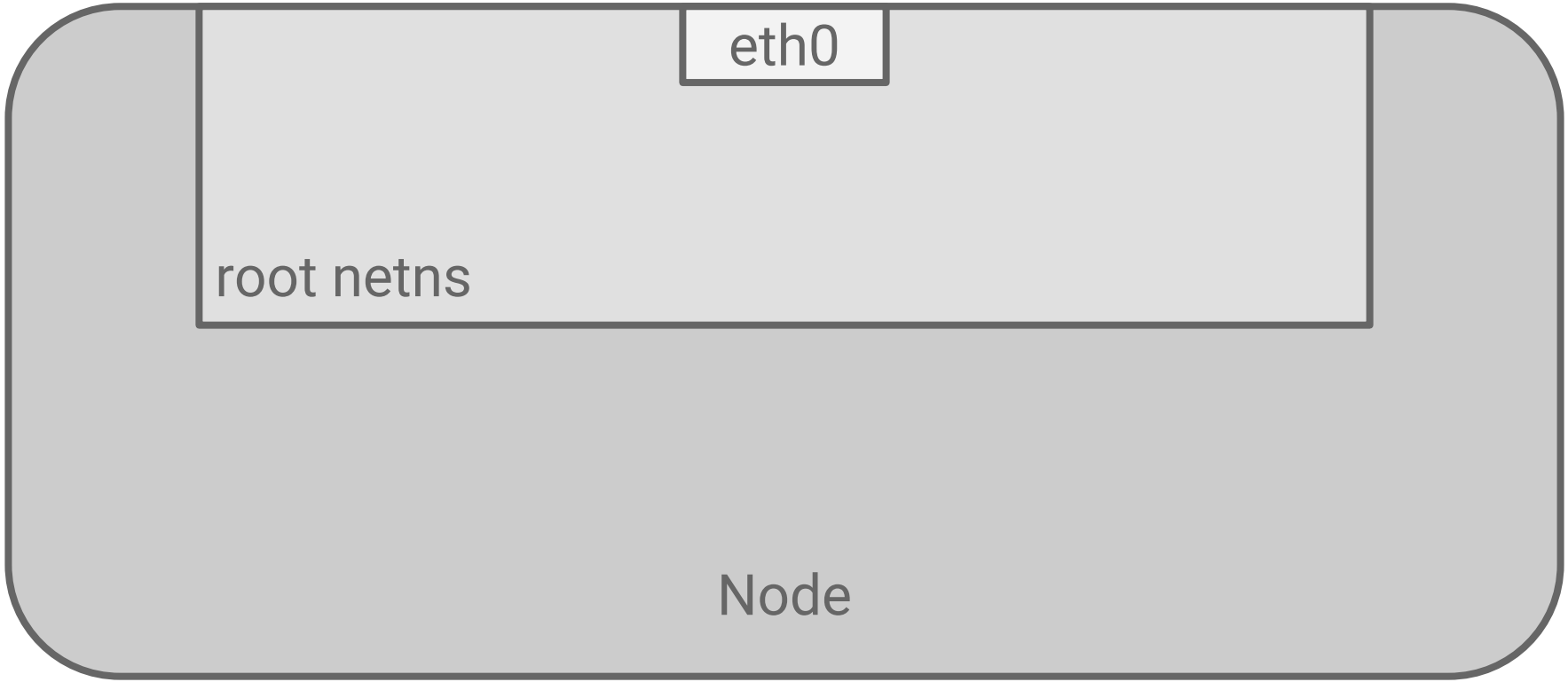
Linux “network namespaces” (aka “netns”) and virtual interfaces



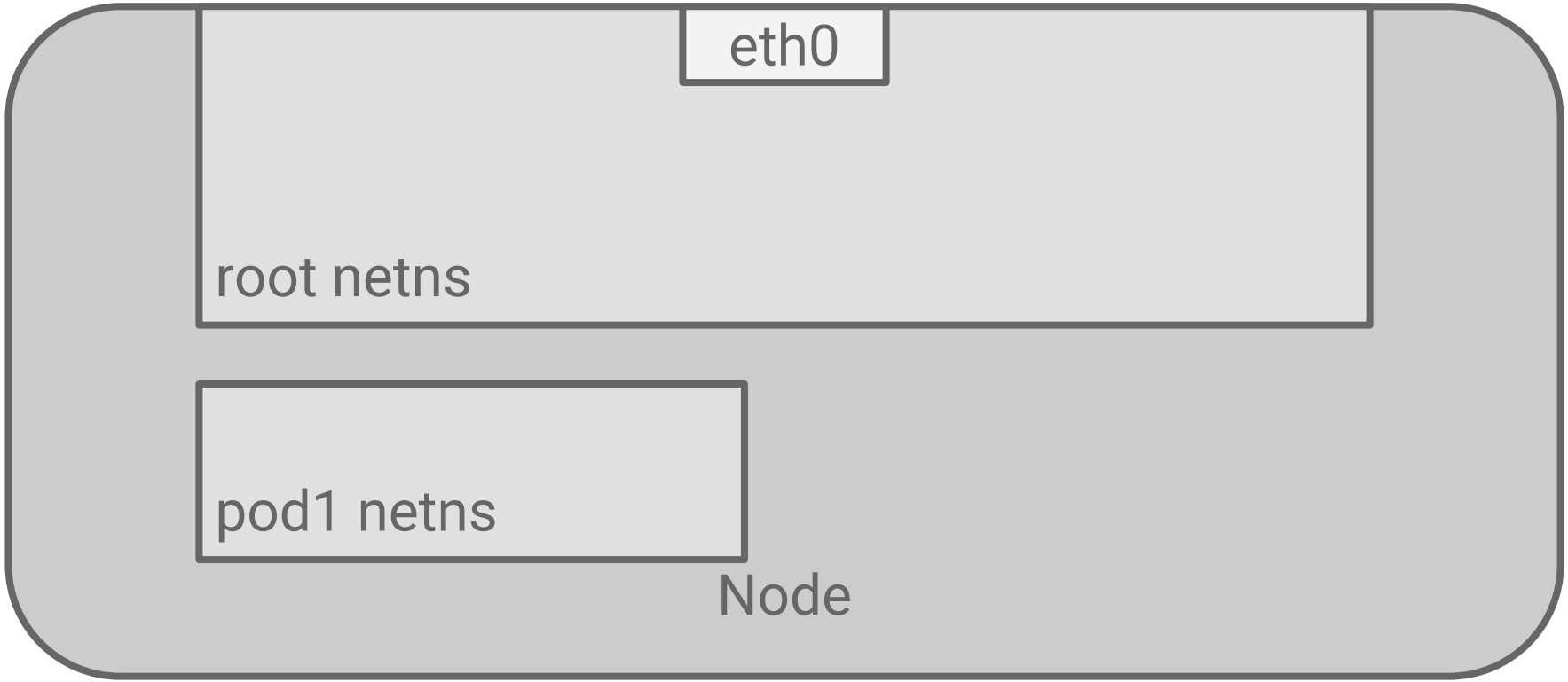
Network namespaces



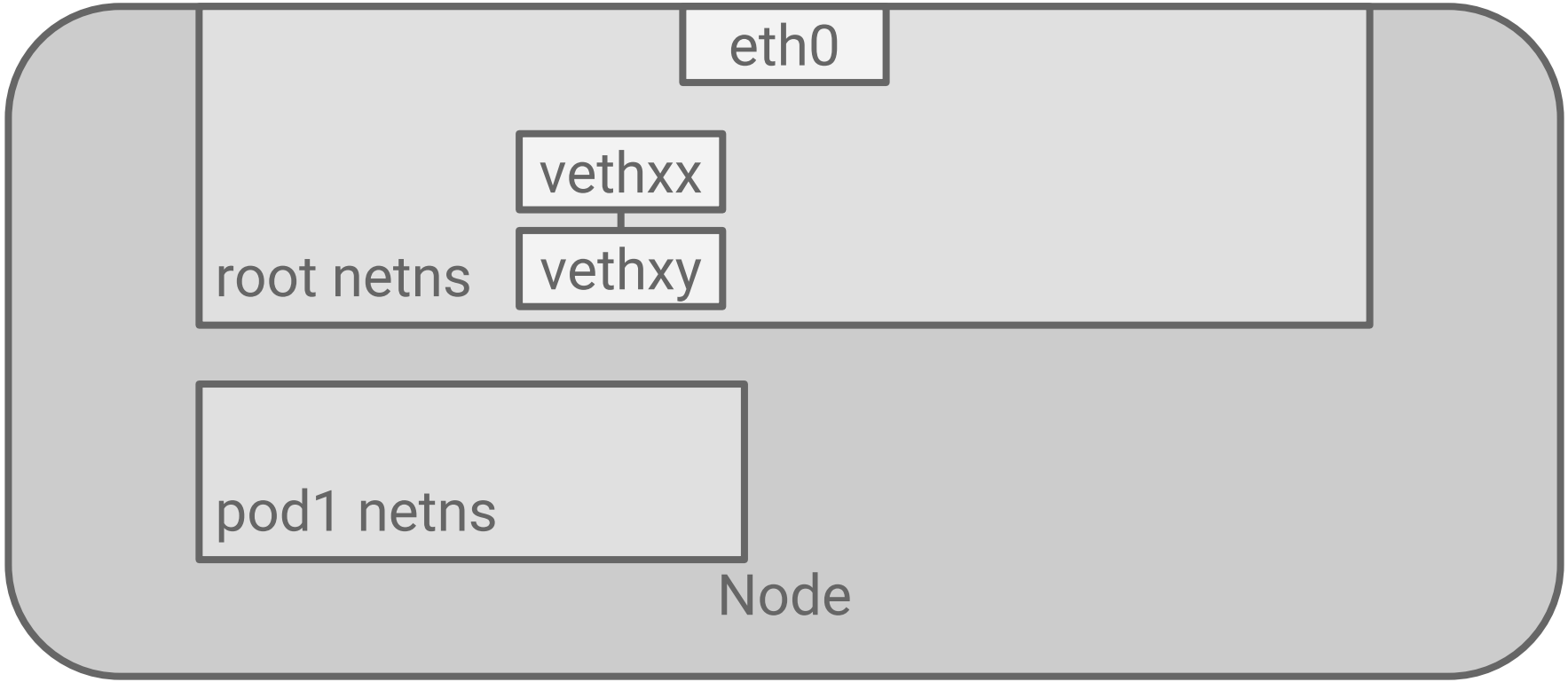
Network namespaces



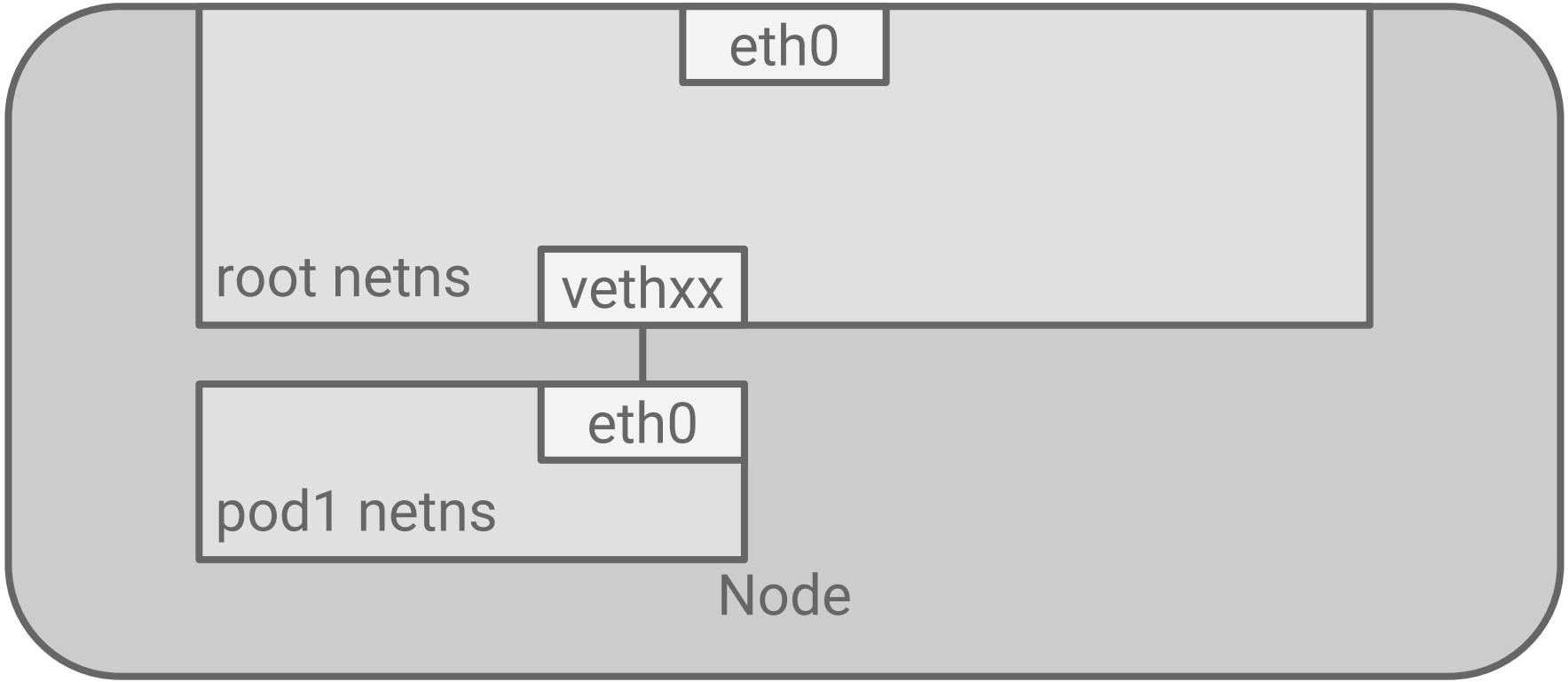
Network namespaces



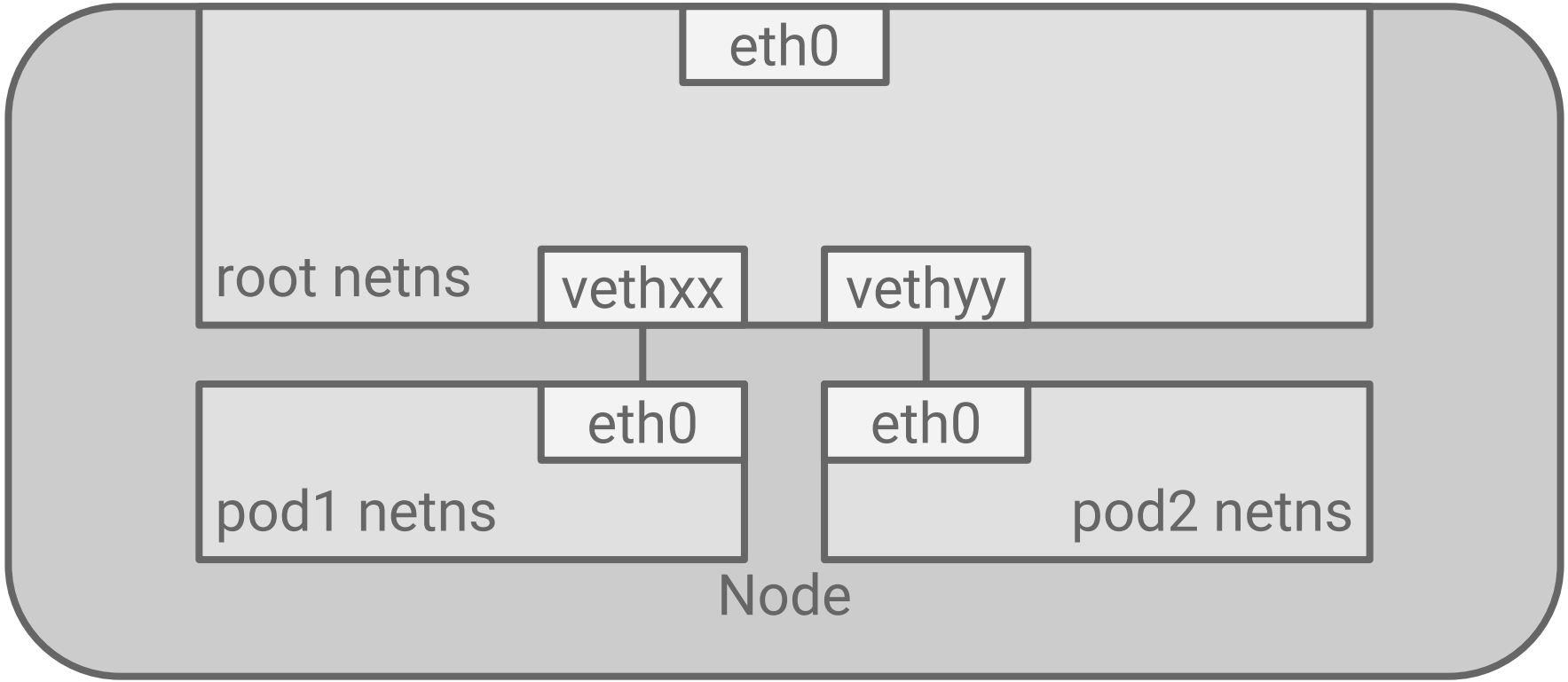
Network namespaces



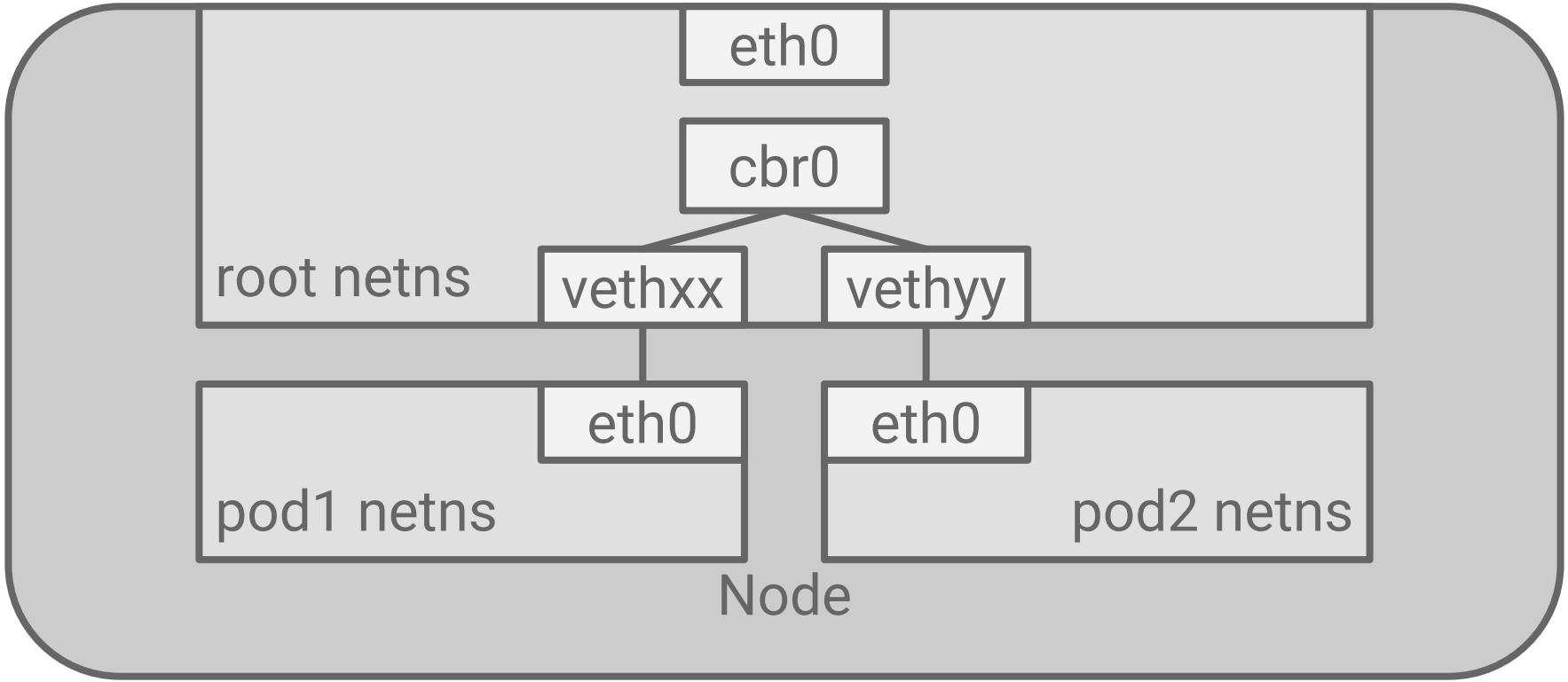
Network namespaces



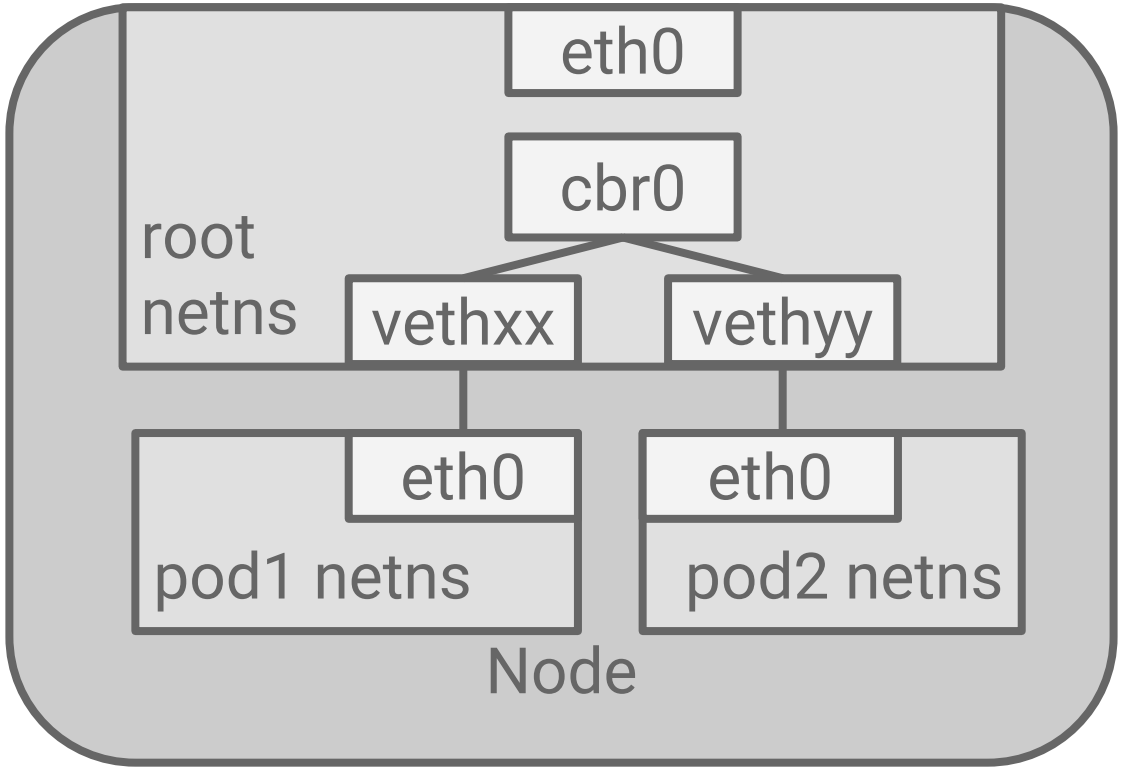
Network namespaces



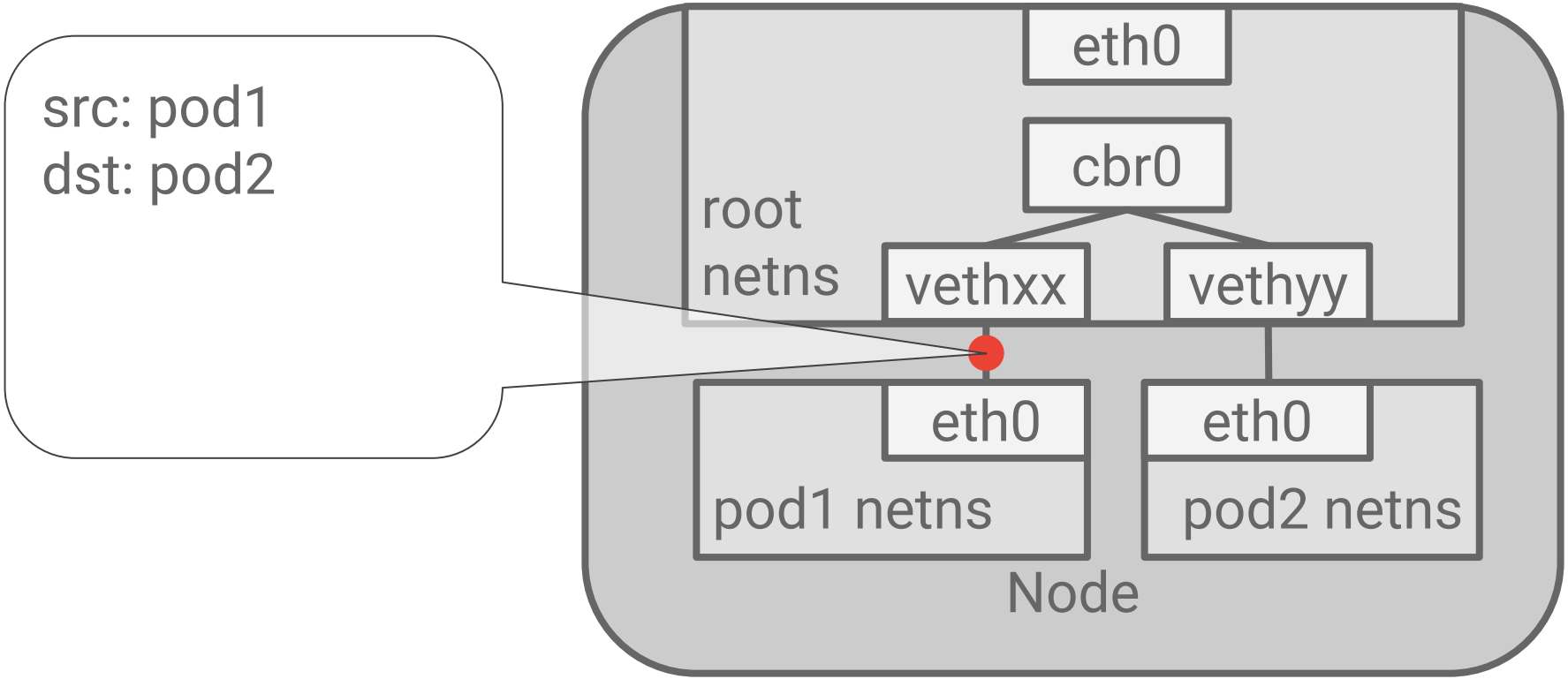
Network namespaces



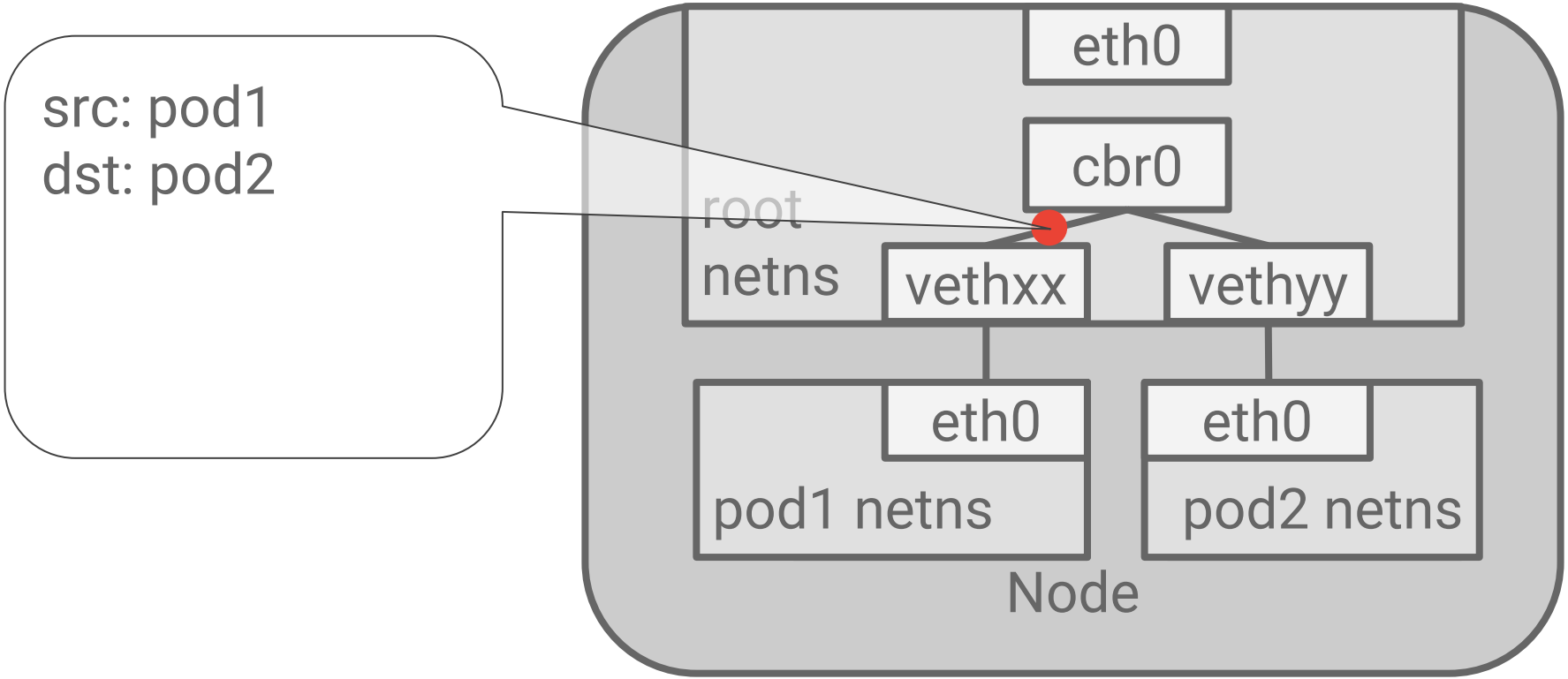
Life of a packet: pod-to-pod, same node



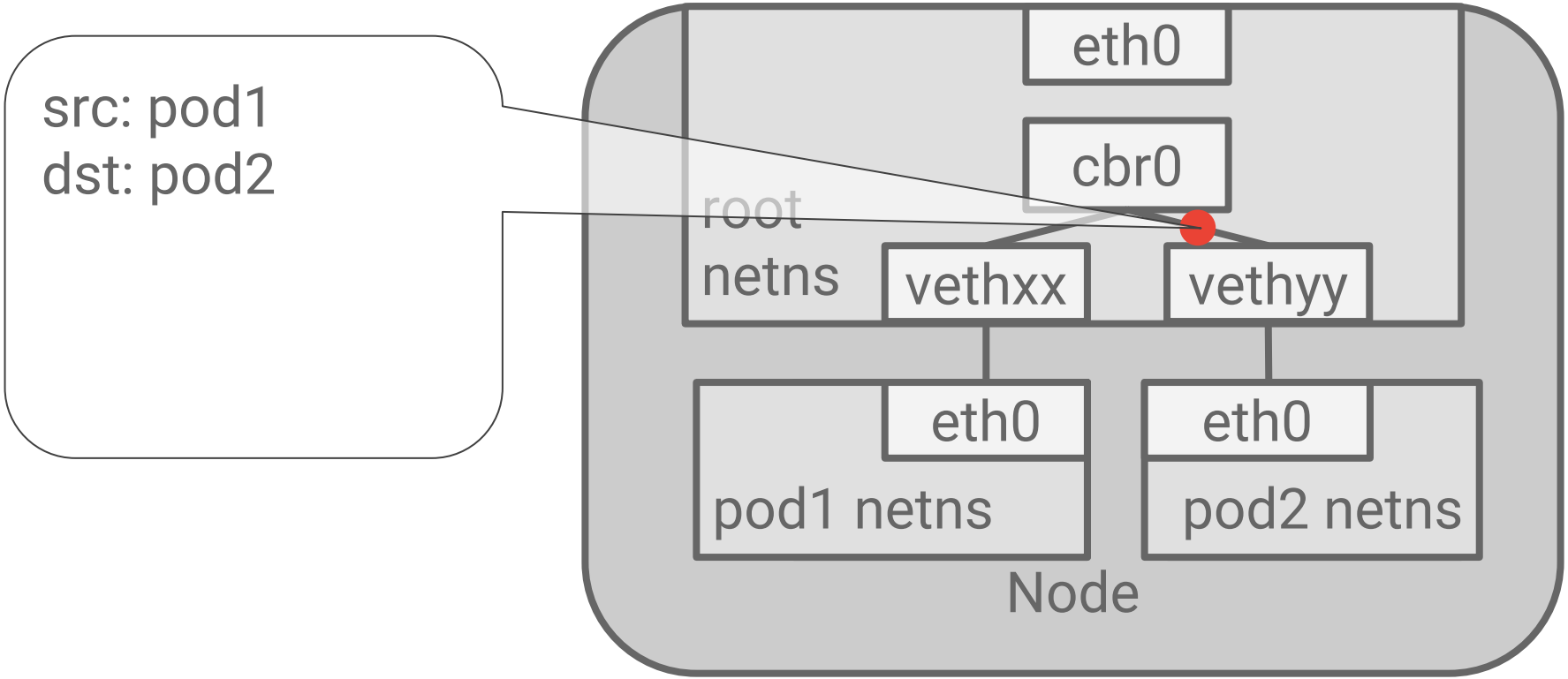
Life of a packet: pod-to-pod, same node



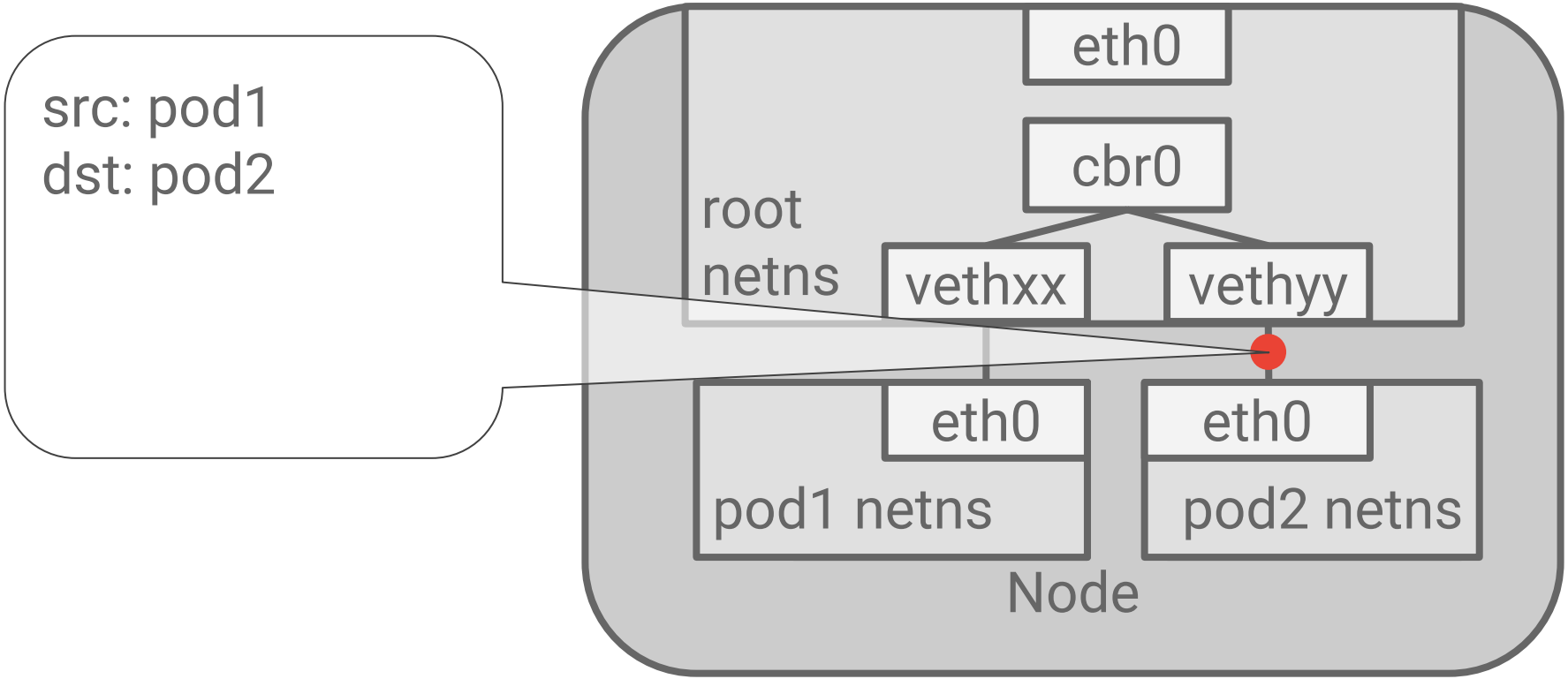
Life of a packet: pod-to-pod, same node



Life of a packet: pod-to-pod, same node



Life of a packet: pod-to-pod, same node



Flat network space

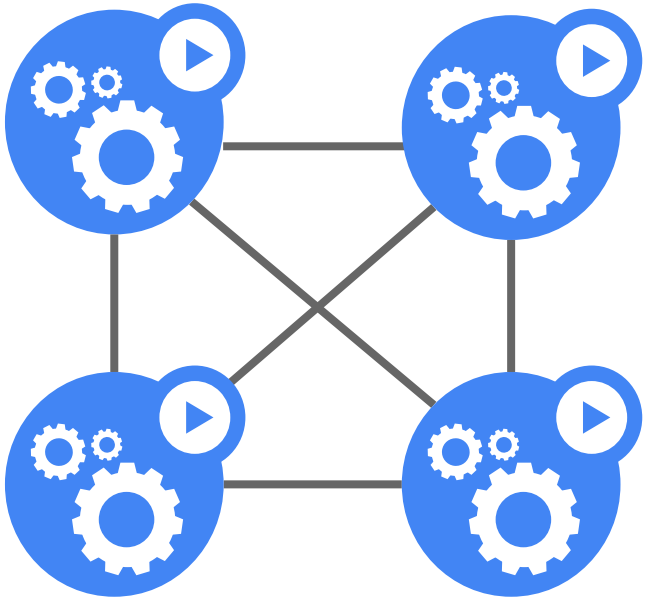
Pods must be reachable across Nodes, too

Kubernetes doesn't care HOW, but this is a requirement

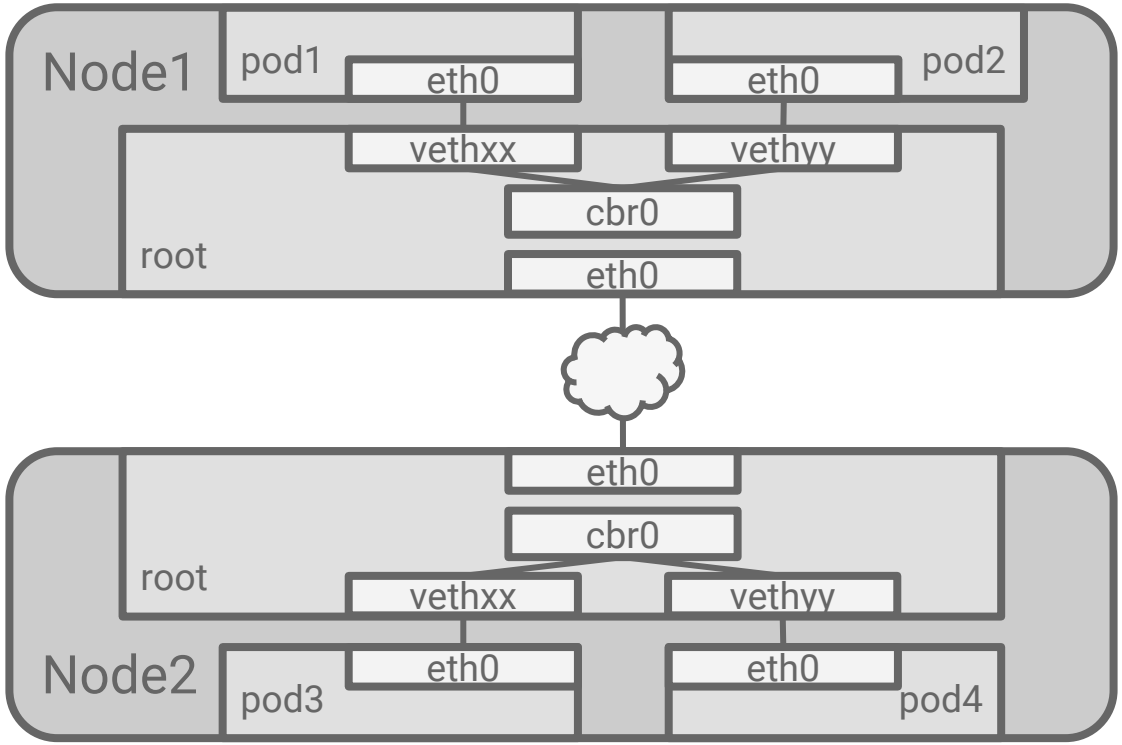
- L2, L3, or overlay

Assign a CIDR (IP block) to each Node

GCP: Teach the network how to route packets

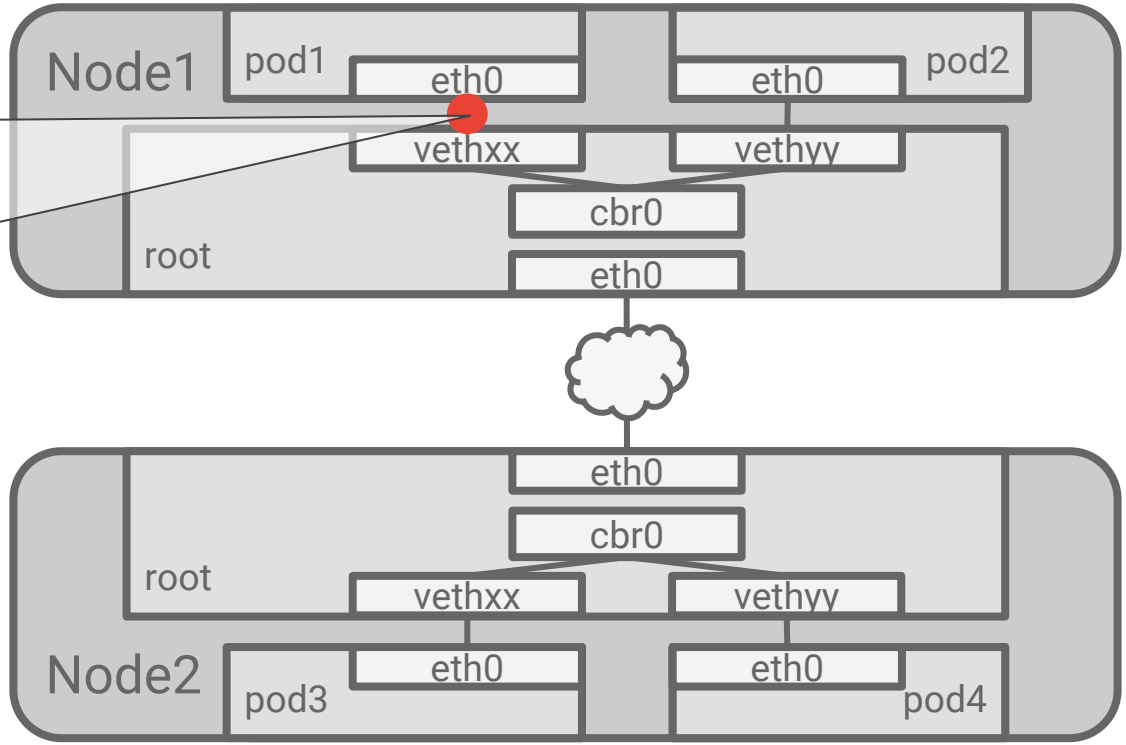


Life of a packet: pod-to-pod, across nodes



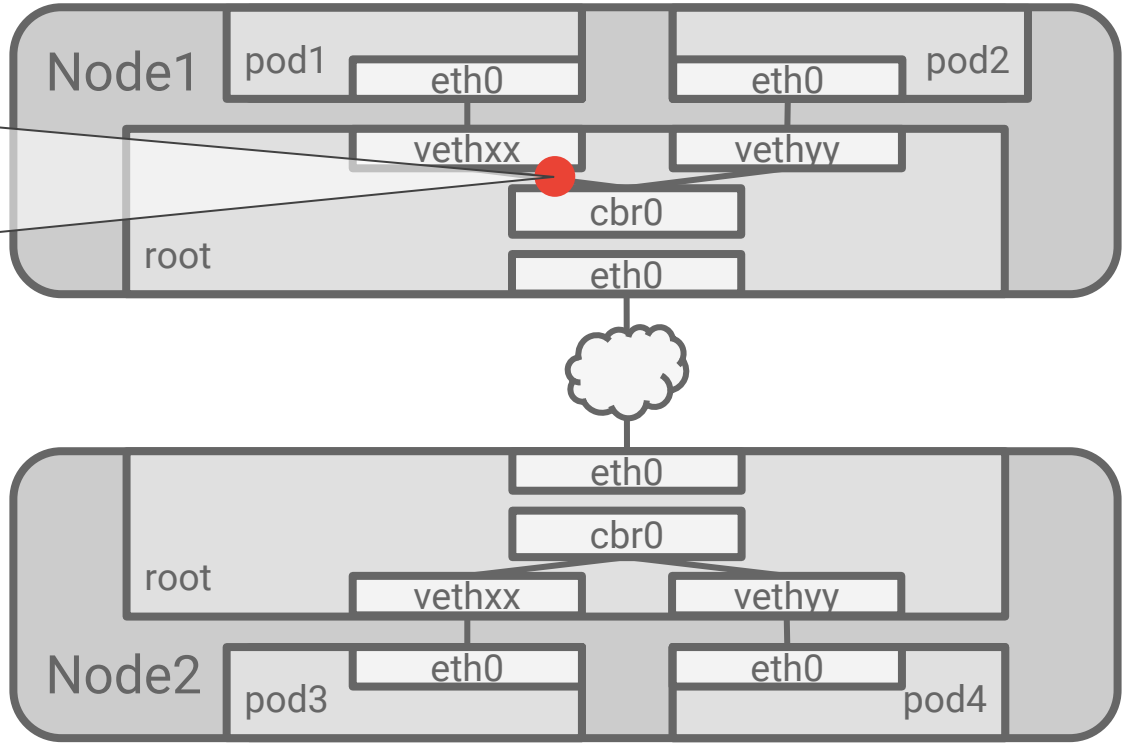
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



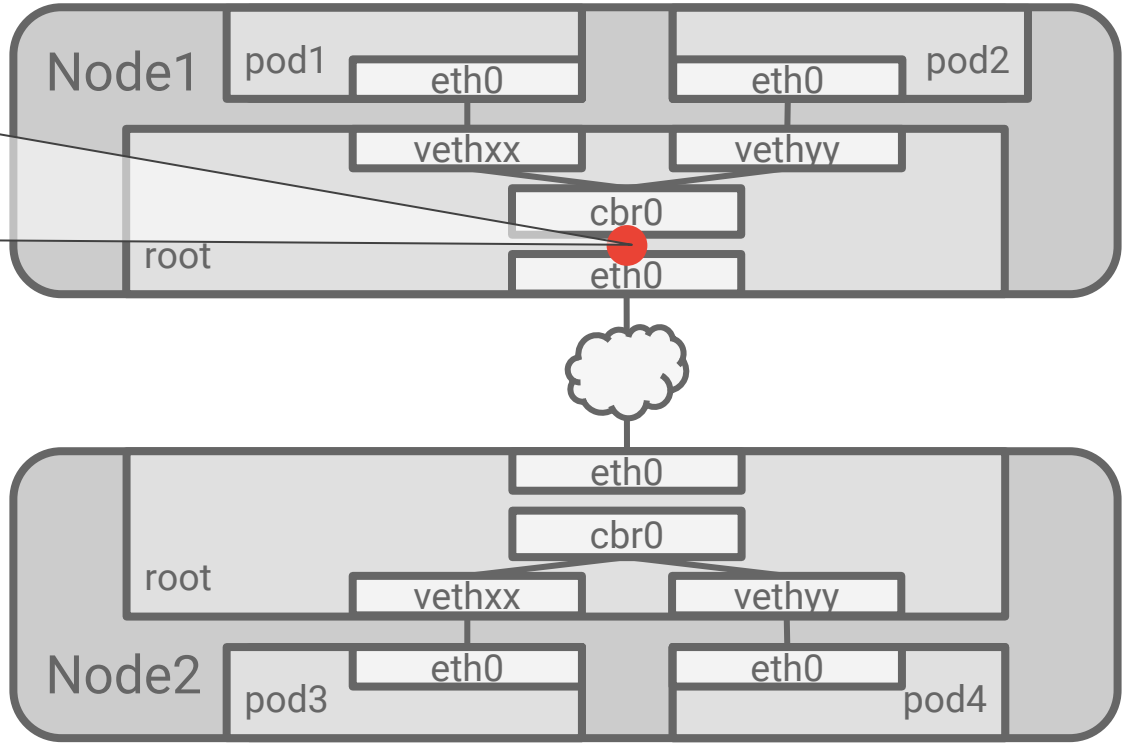
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



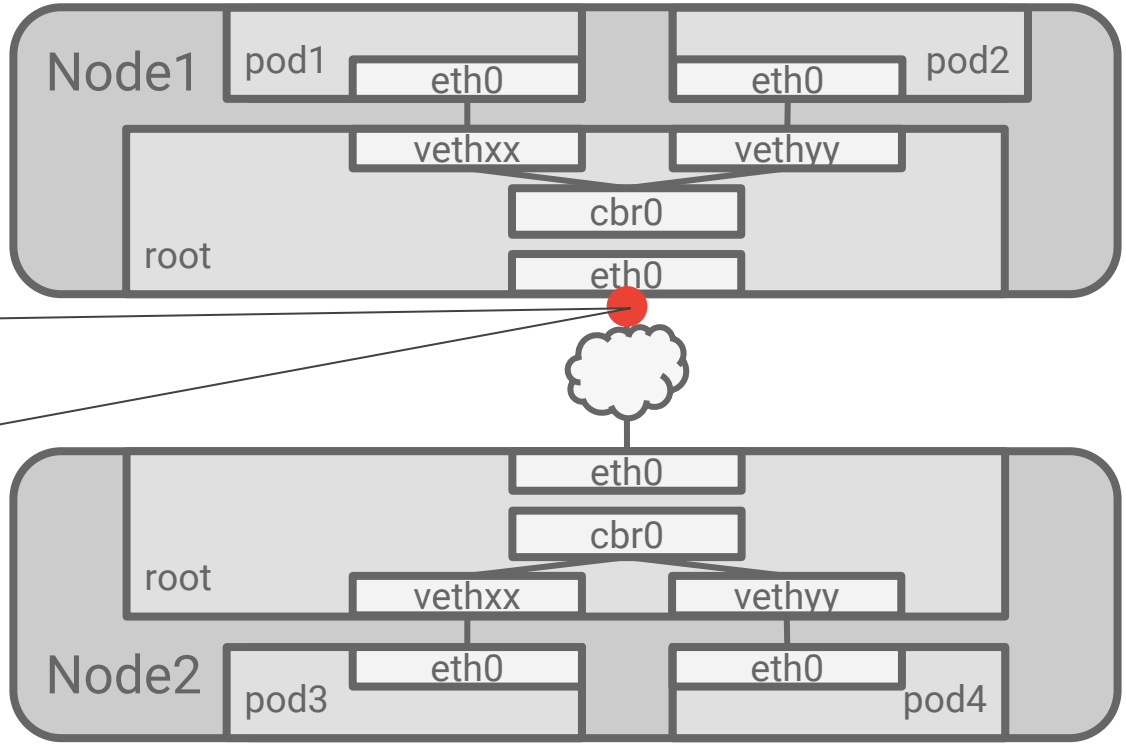
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



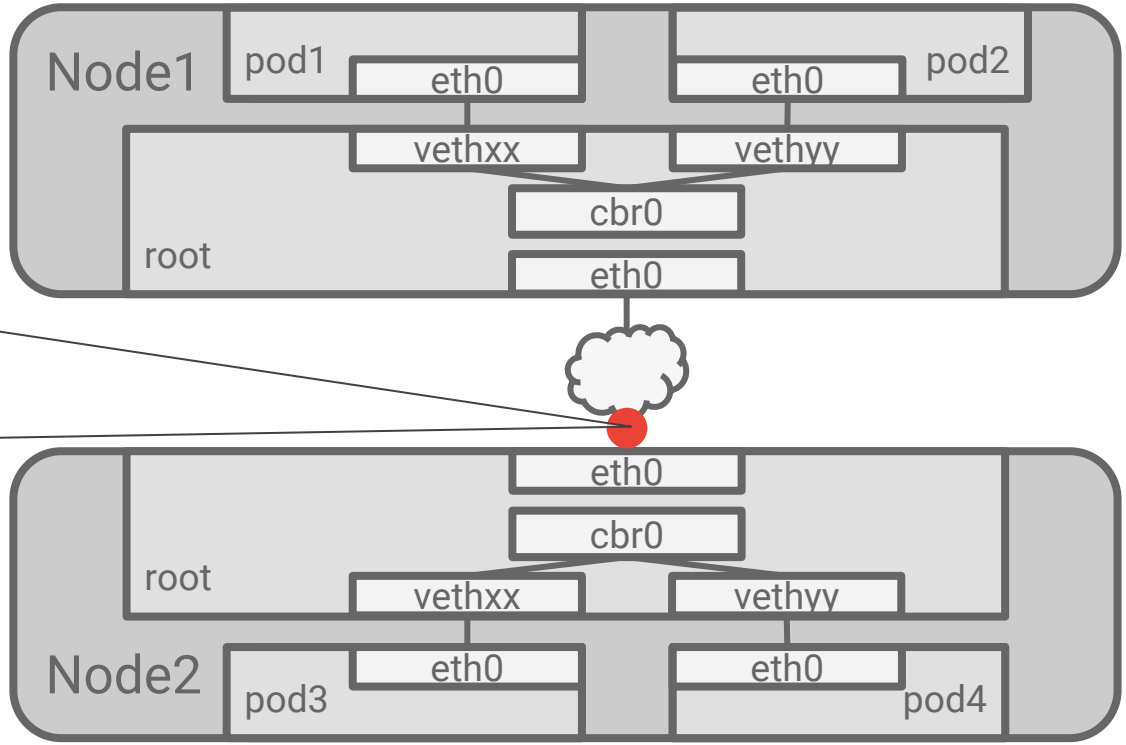
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



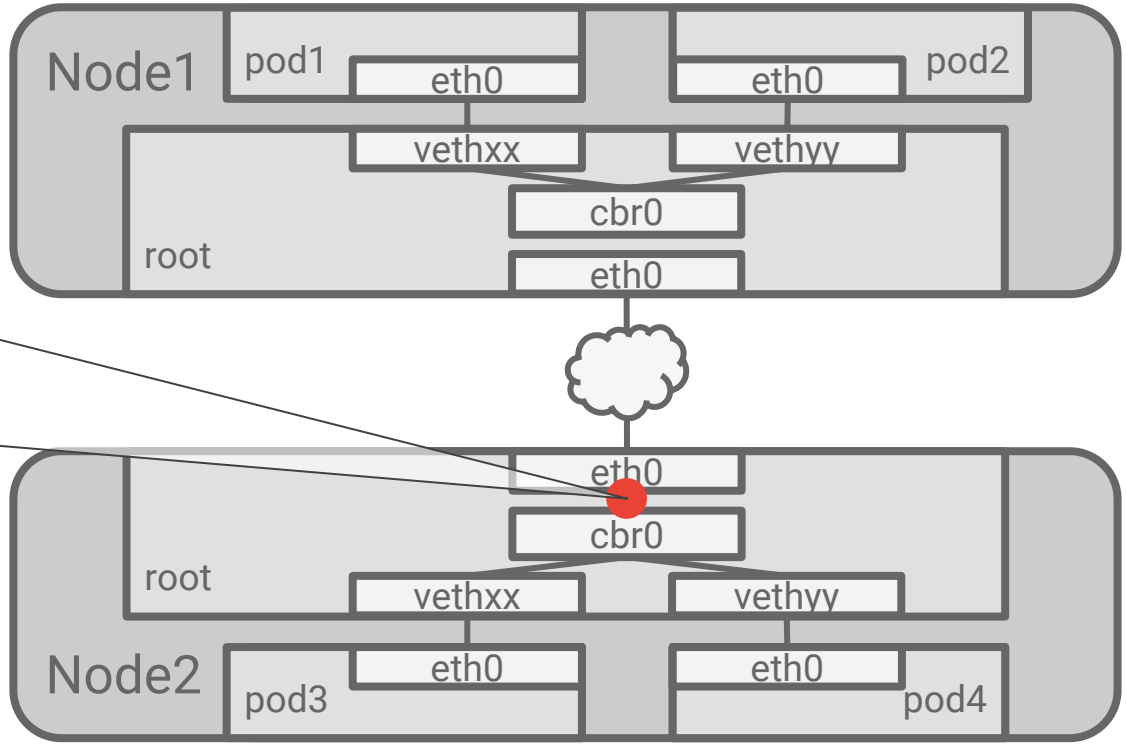
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



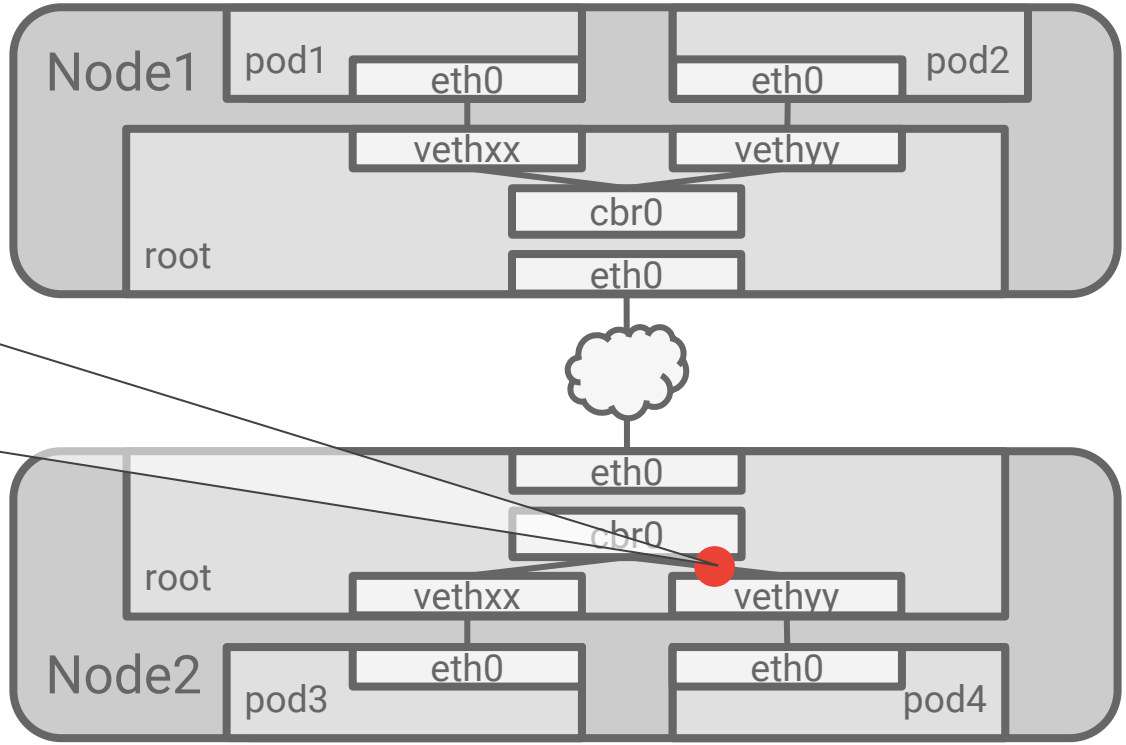
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



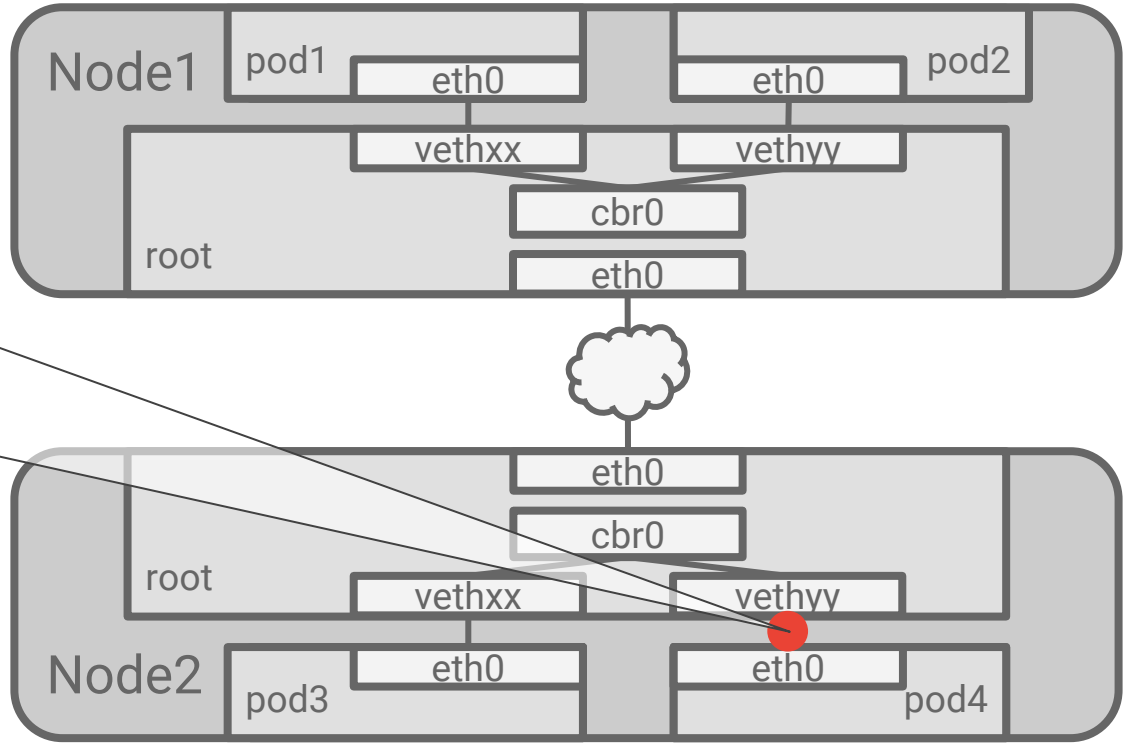
Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



Life of a packet: pod-to-pod, across nodes

src: pod1
dst: pod4



Overlays

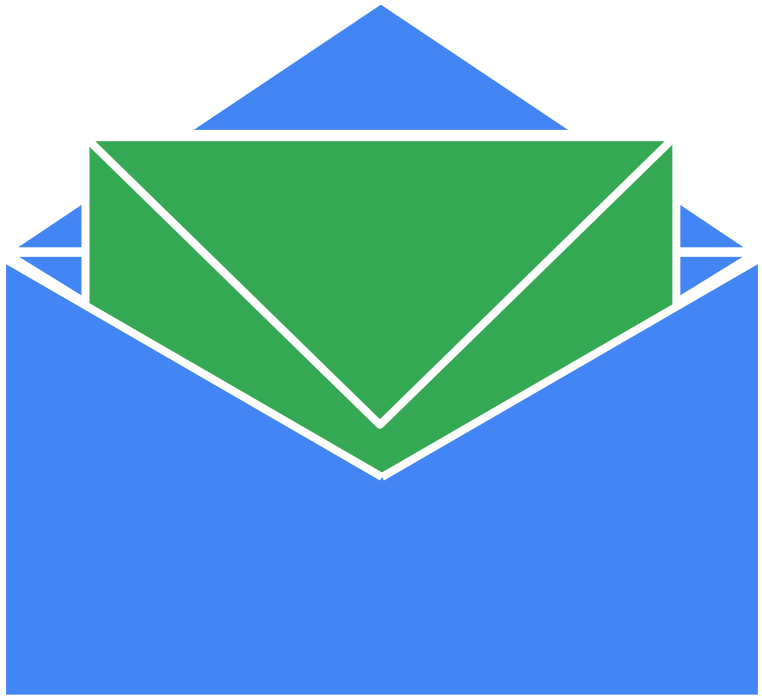
Overlay networks

Why?

- Can’t get enough IP space
- Network can’t handle extra routes
- Want management features

Encapsulate packet-in-packet

Traverse the native network between Nodes

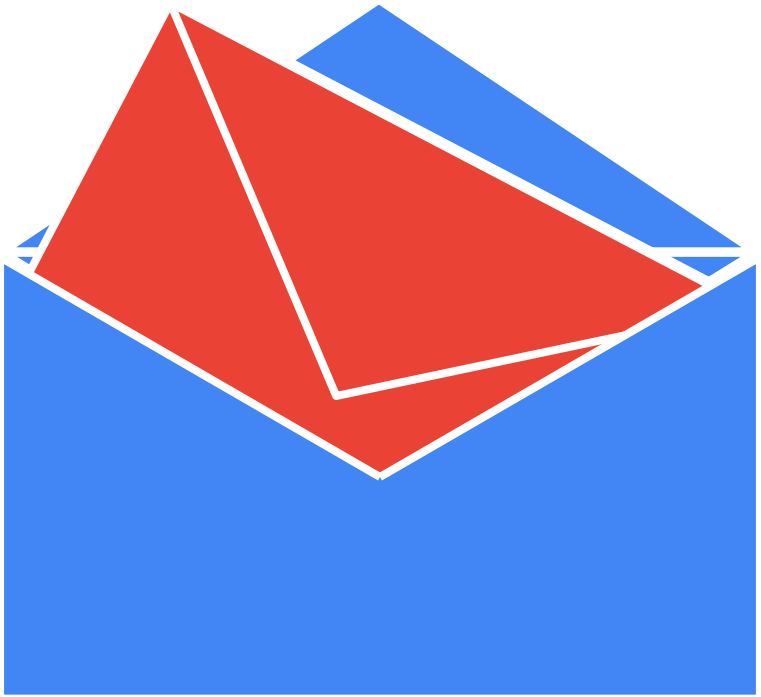


Overlay networks

Why Not?

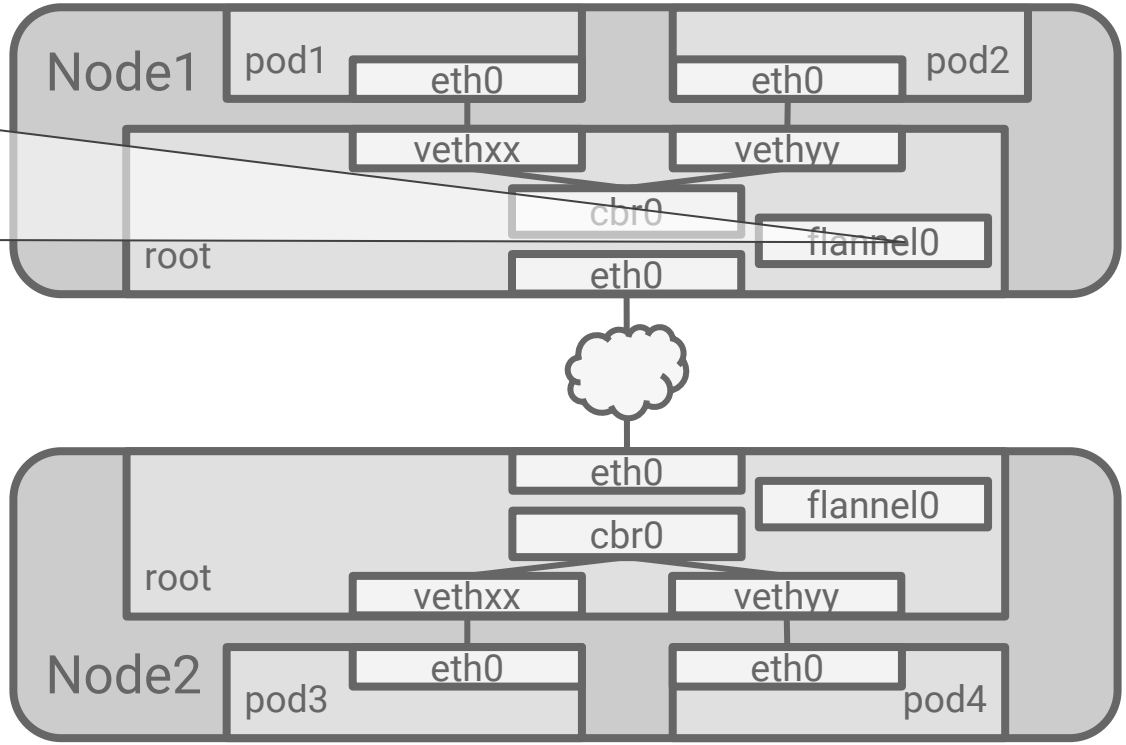
- Latency overhead in some cloud providers
- Complexity overhead
- Often not required

Use it when you know you need it



Overlay example: Flannel (vxlan)

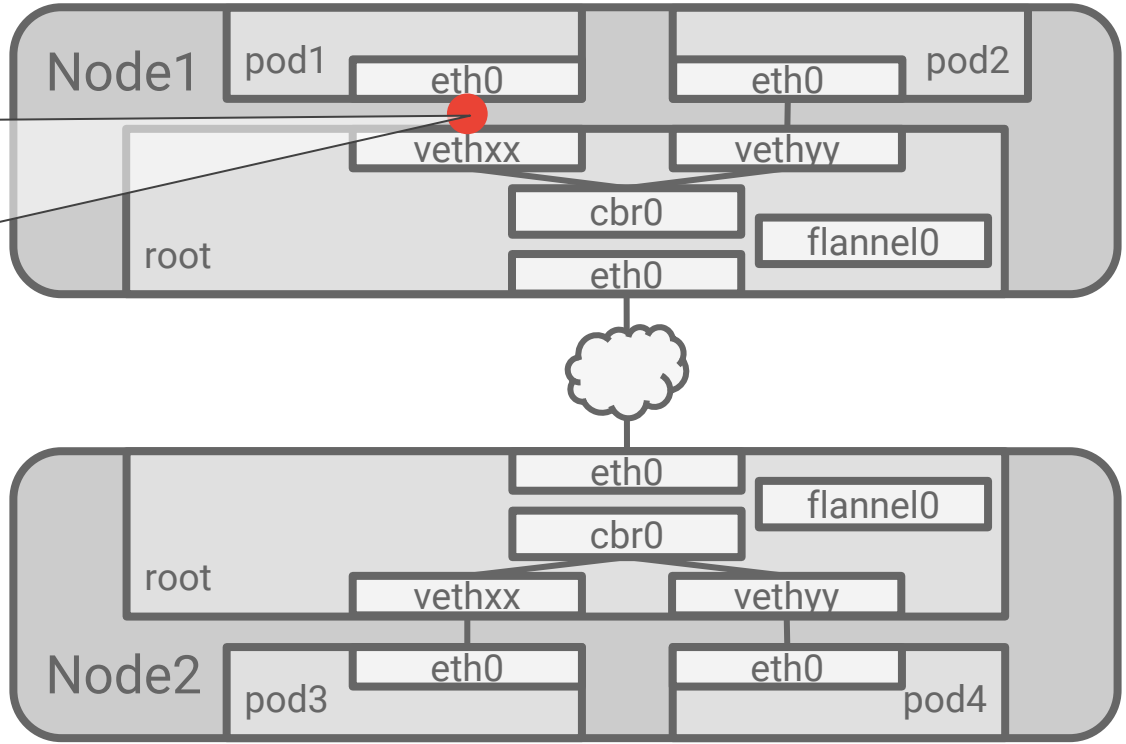
VXLAN interface acts like any other vNIC



CoreOS

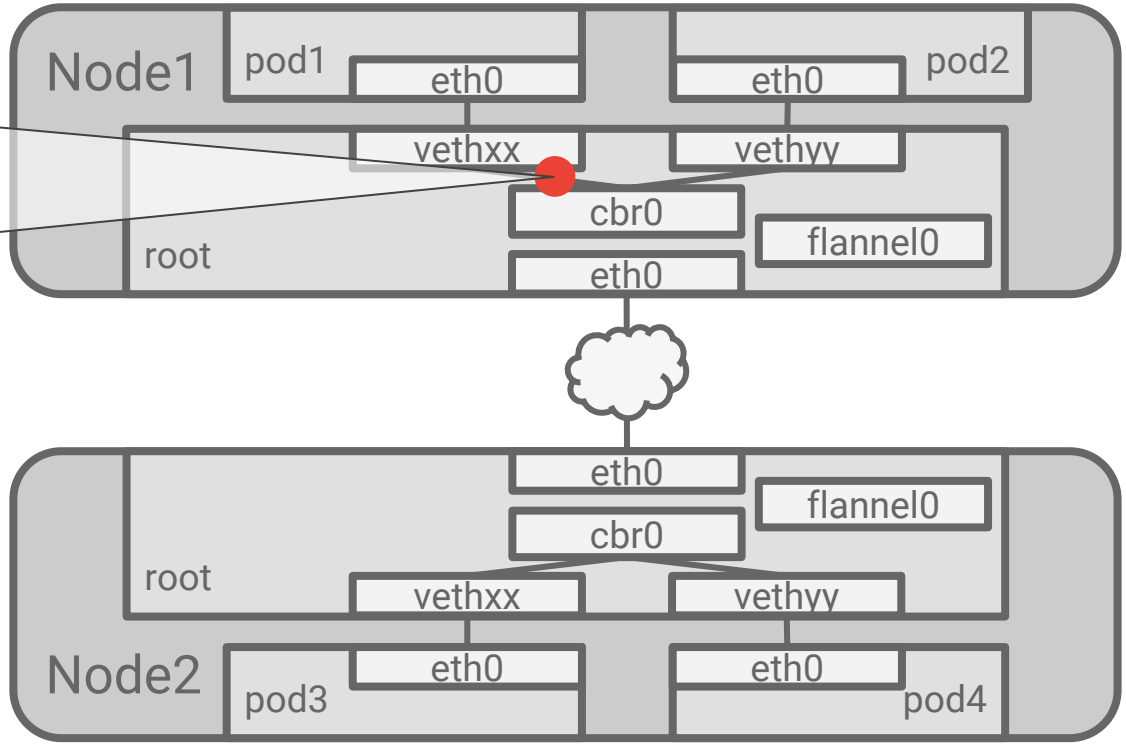
Overlay example: Flannel

src: pod1
dst: pod4



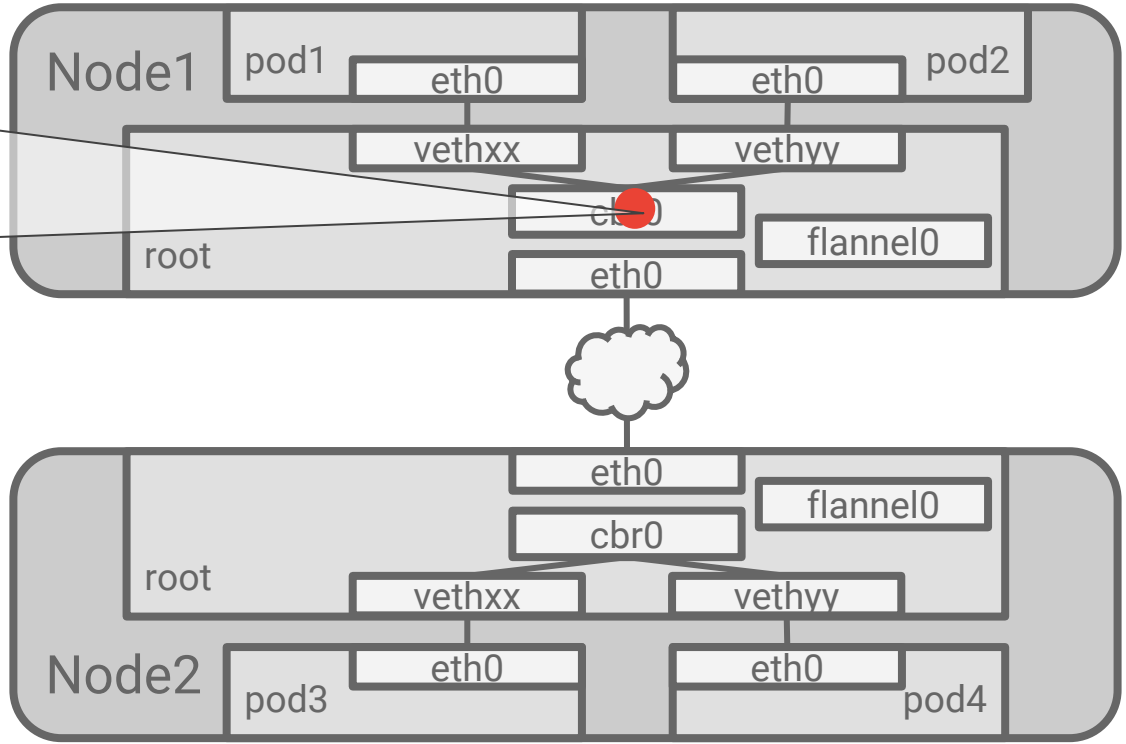
Overlay example: Flannel

src: pod1
dst: pod4

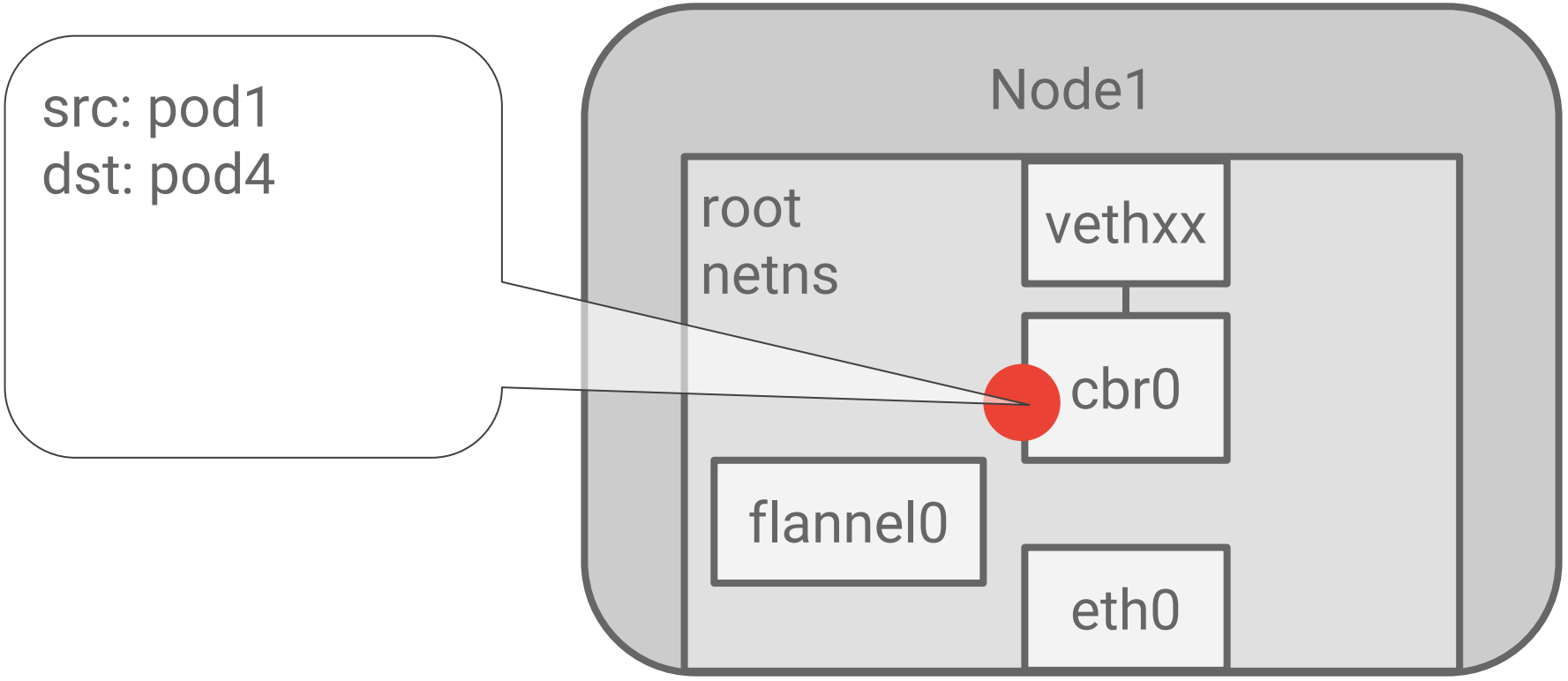


Overlay example: Flannel

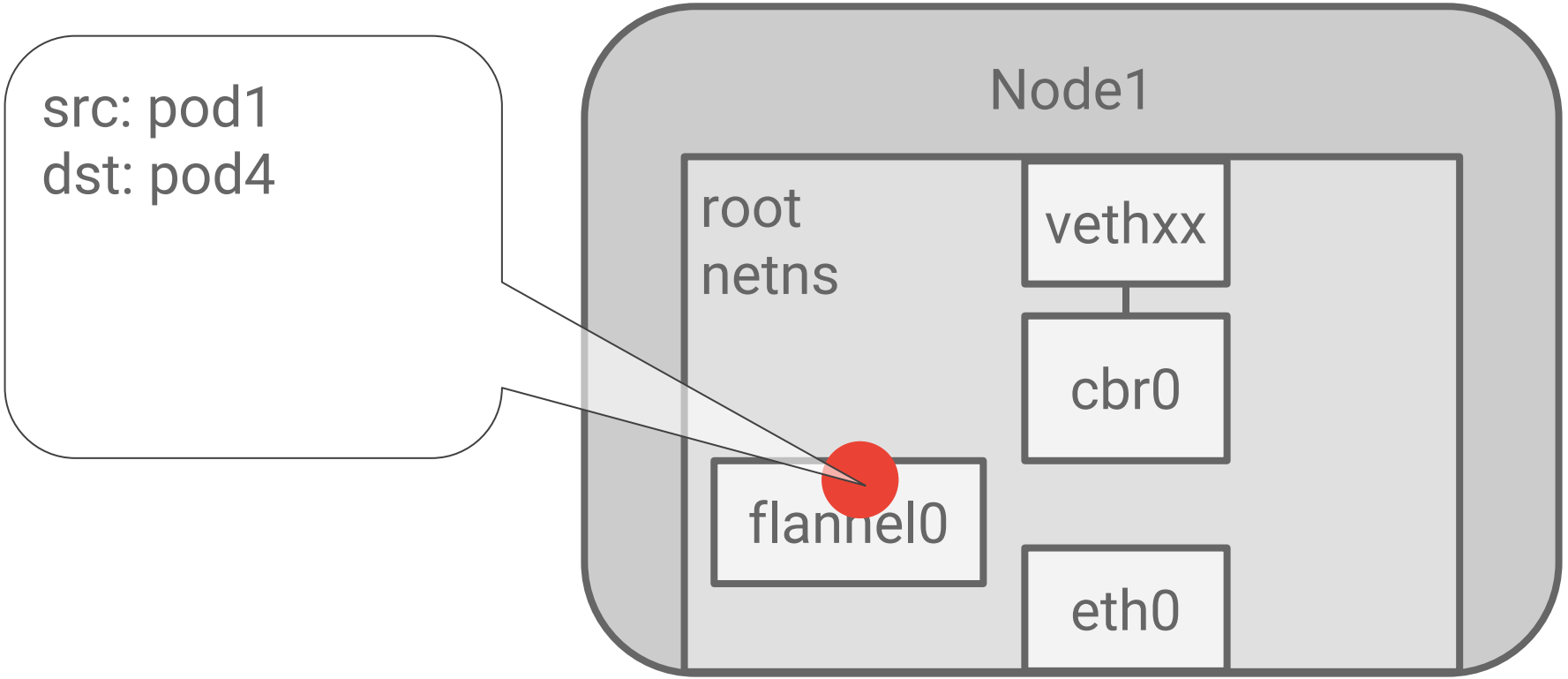
src: pod1
dst: pod4



Overlay example: Flannel



Overlay example: Flannel

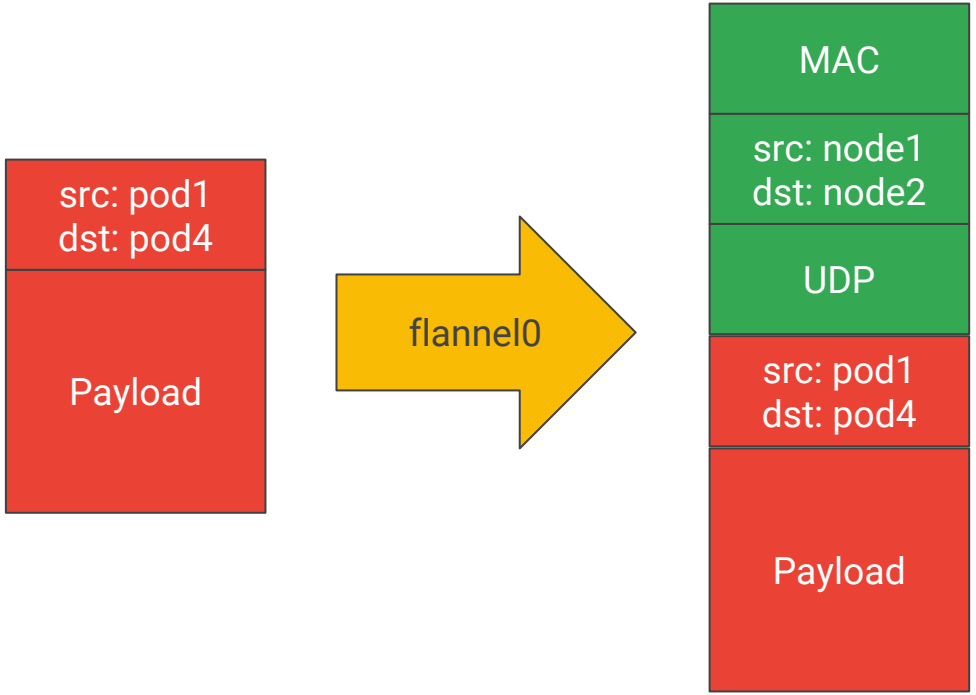


Overlay example: Flannel

Encapsulates the packet

Flannel device implementation:

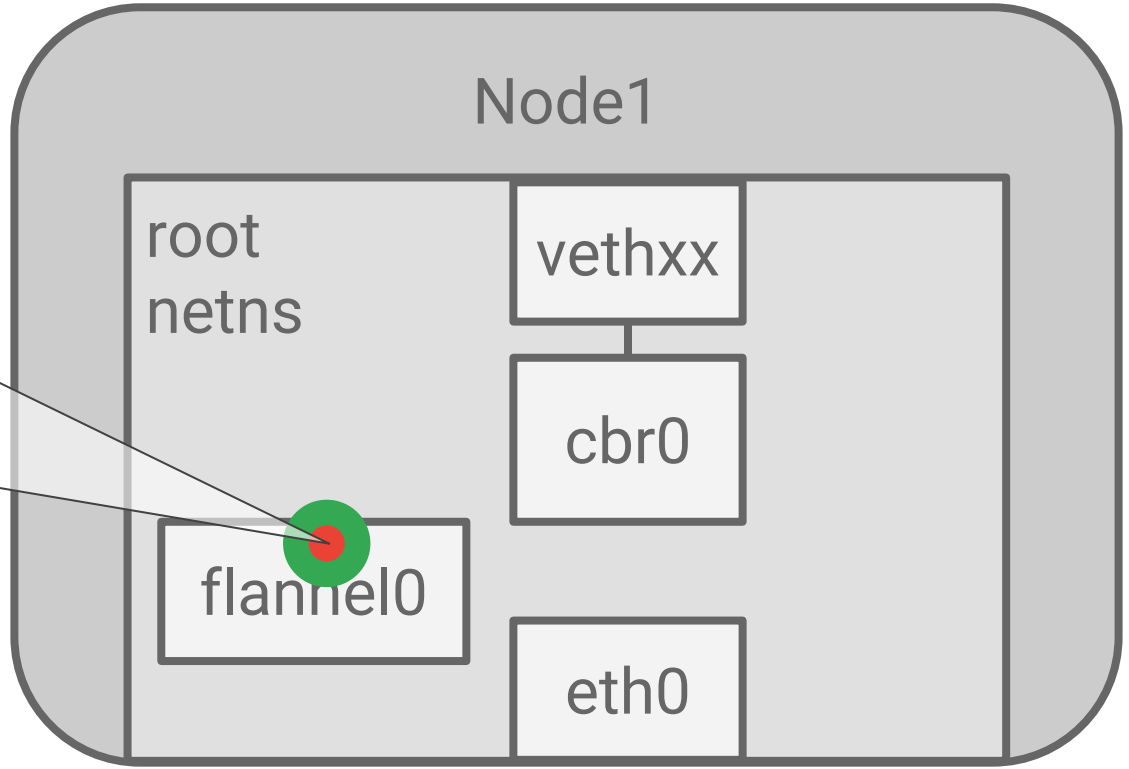
- Simple VXLAN, developed by CoreOS for containers and kubernetes
- Uses Linux native VXLAN devices
- A userspace agent for address resolution
- Data path is in-kernel (fast)



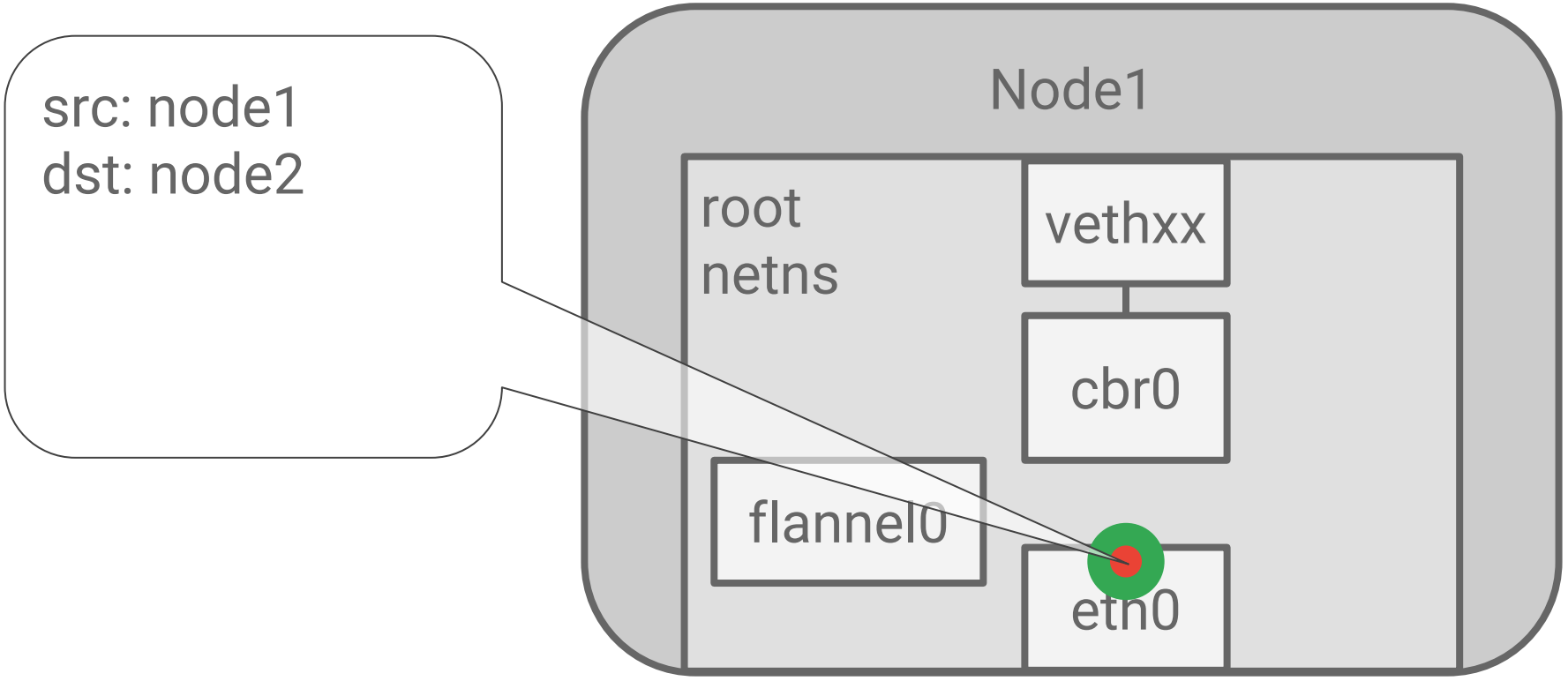
Overlay example: Flannel

~~src: pod1~~
src: node1
~~dst: pod4~~
dst: node2

encapsulated

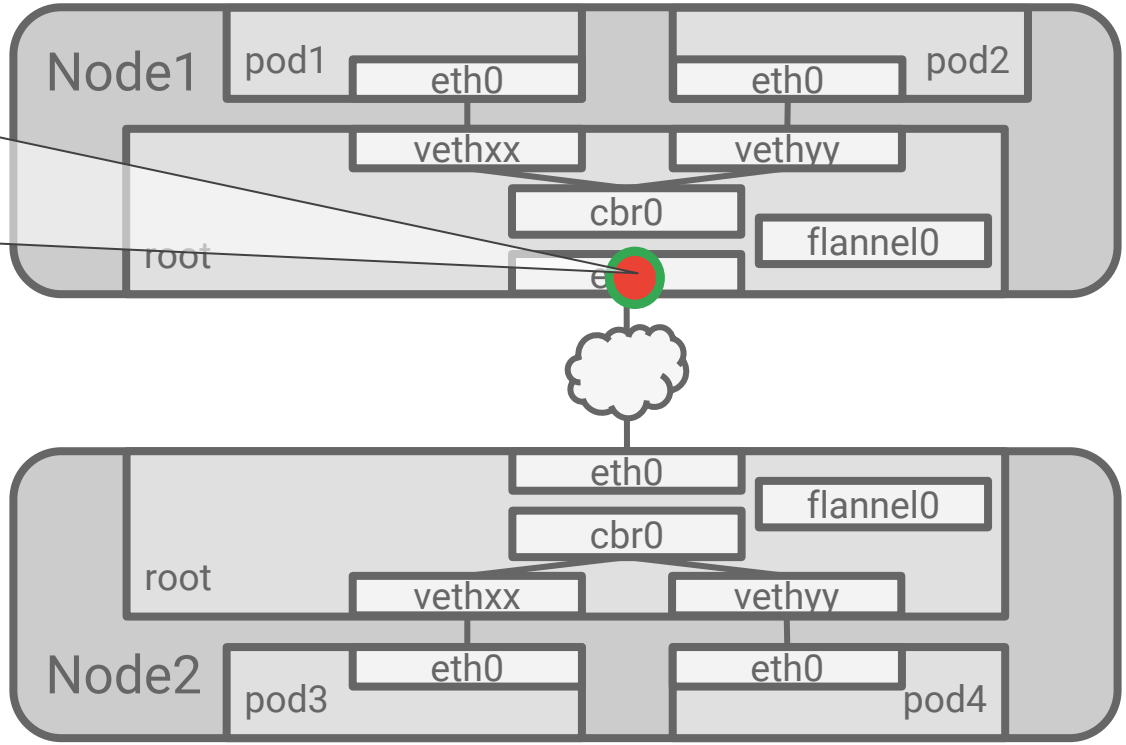


Overlay example: Flannel



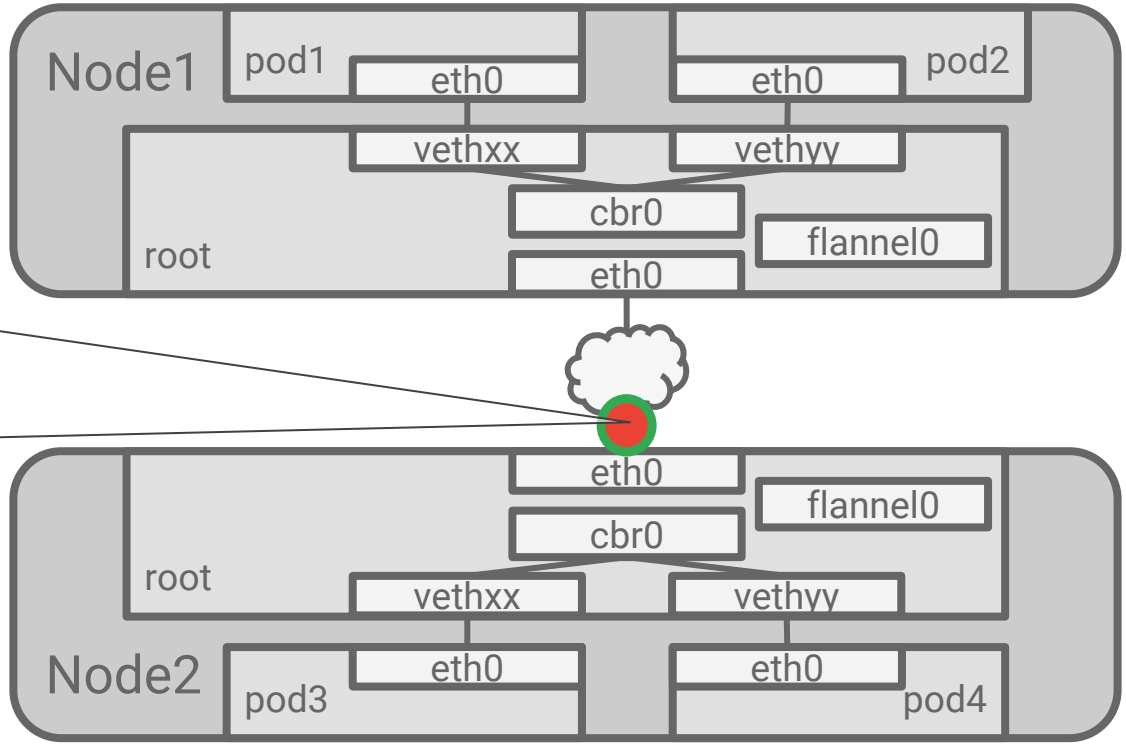
Overlay example: Flannel

src: node1
dst: node2

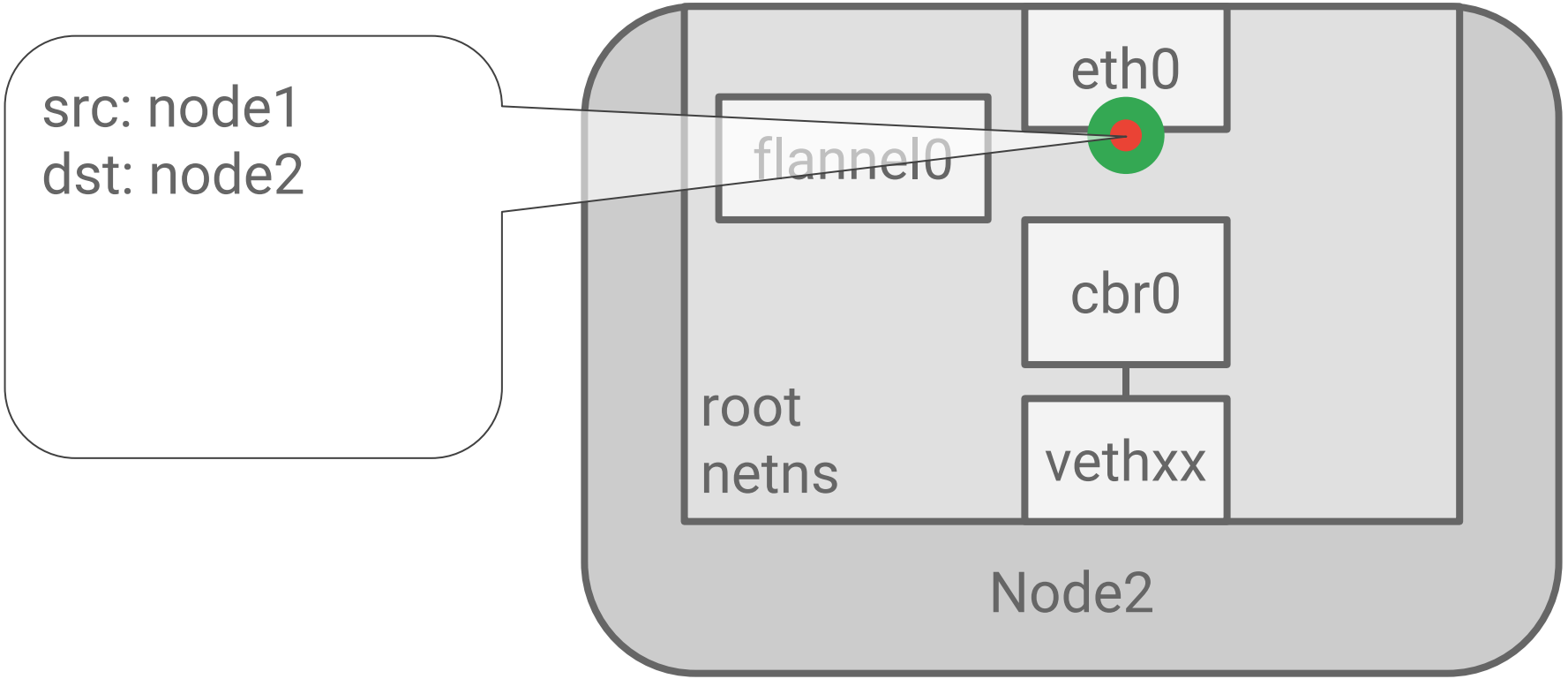


Overlay example: Flannel

src: node1
dst: node2



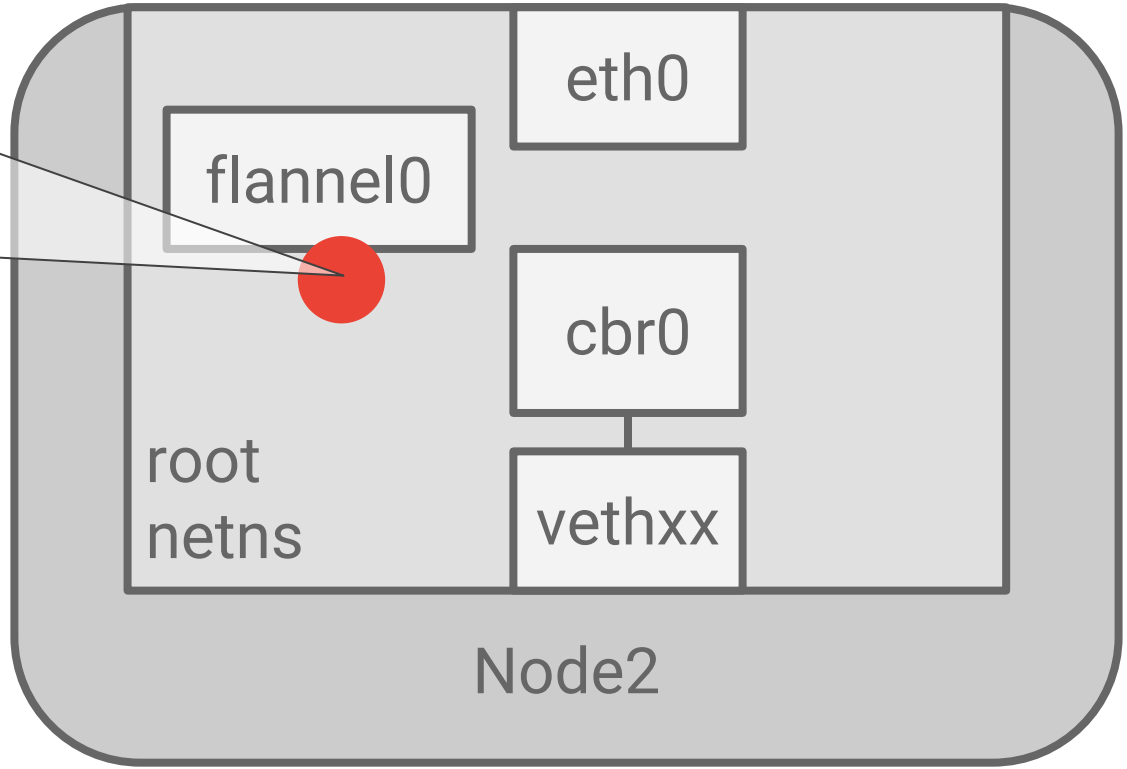
Overlay example: Flannel



Overlay example: Flannel

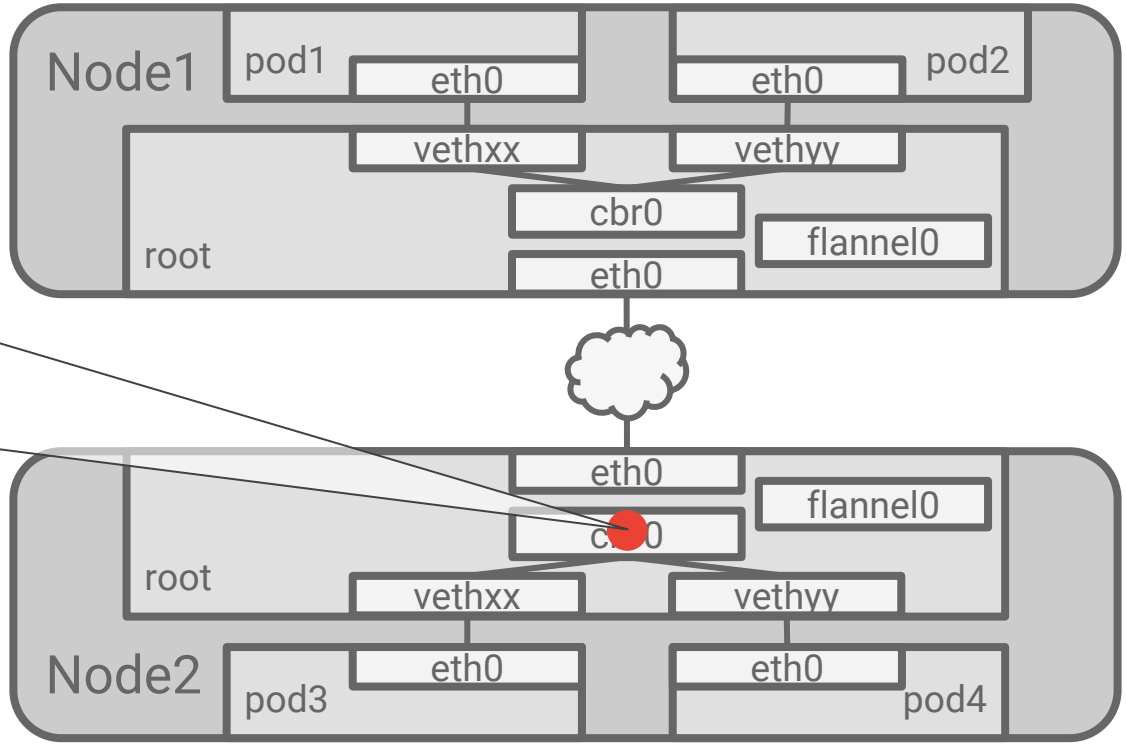
~~src: node1~~
src: pod1
~~dst: node2~~
dst: pod4

decapsulated



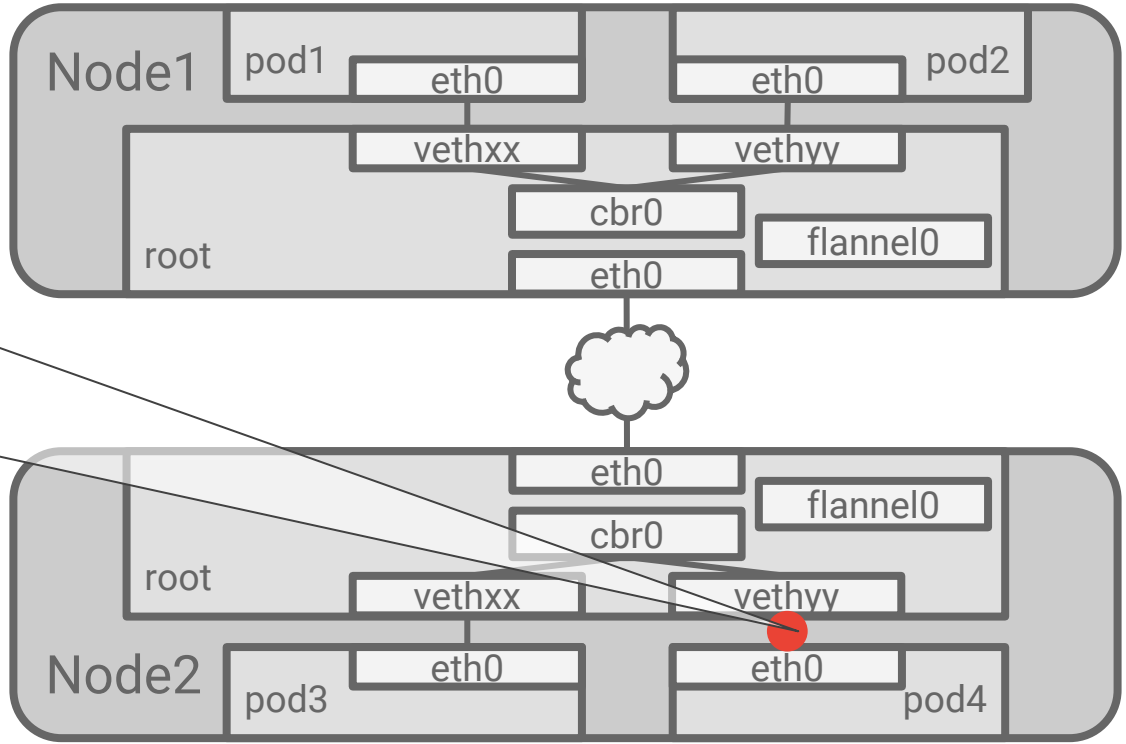
Overlay example: Flannel

src: pod1
dst: pod4



Overlay example: Flannel

src: pod1
dst: pod4



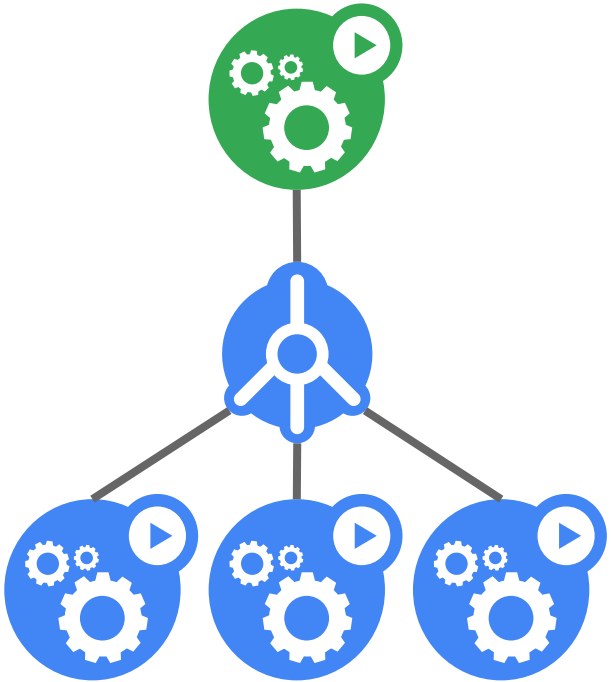
Dealing with change

A real cluster changes over time:

- Rolling updates
- Scale-up and scale-down events
- Pods crash or hang
- Nodes reboot

The pod addresses you need to reach can change without warning

You need something more durable than a pod IP



Services

The service abstraction

A service is a group of endpoints (usually pods)

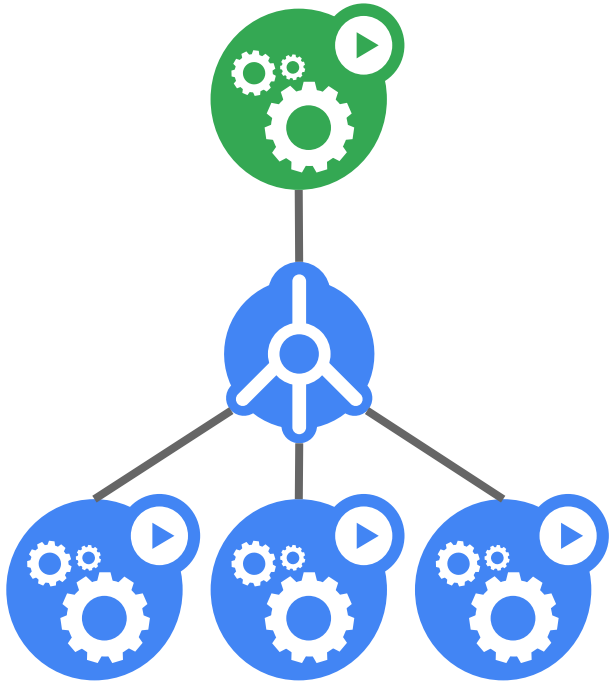
Services provide a stable VIP

VIP automatically routes to backend pods

- Implementations can vary
- We will examine the default implementation

The set of pods “behind” a service can change

Clients only need the VIP, which doesn’t change



Service

What you submit is simple

- Other fields will be defaulted or assigned

```
kind: Service
apiVersion: v1
metadata:
  name: store-be
spec:
 selector:
    app: store
    role: be
  ports:
  - name: http
    port: 80
```

Service

What you submit is simple

- Other fields will be defaulted or assigned

The ‘`selector`’ field chooses which pods to balance across

```
kind: Service
apiVersion: v1
metadata:
  name: store-be
spec:
  selector:
    app: store
    role: be
  ports:
  - name: http
    port: 80
```

Service

What you get back has more information

Automatically creates a distributed load balancer

```
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
```

Service

What you get back has more information

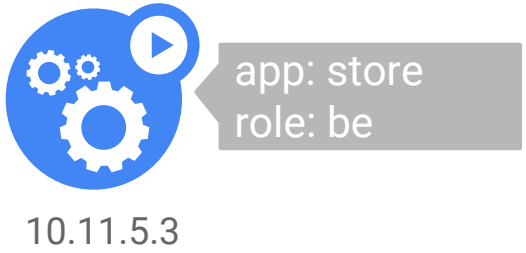
Automatically creates a distributed load balancer

The default is to allocate an in-cluster IP

```
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
```

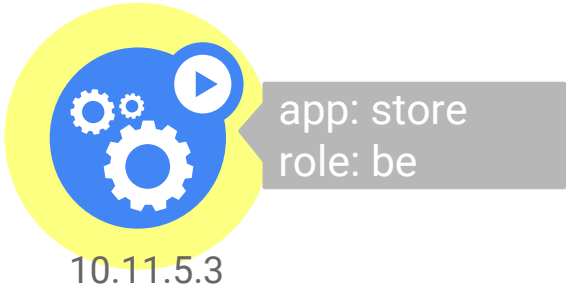
Endpoints

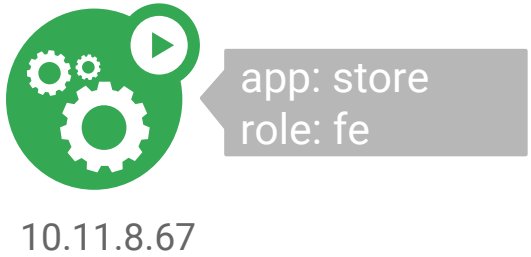
```
selector:
  app: store
  role: be
```

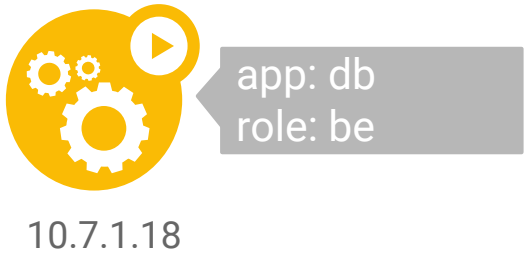


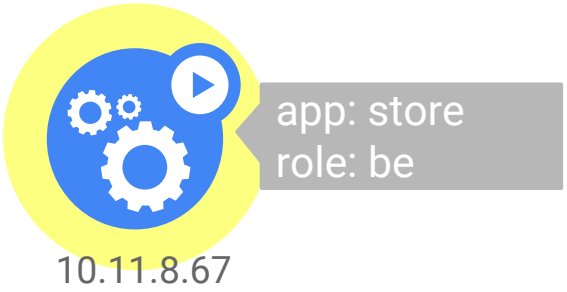
Endpoints

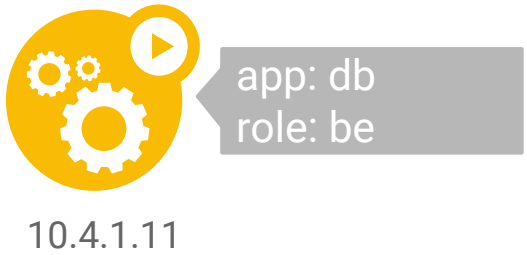
```
selector:
  app: store
  role: be
```

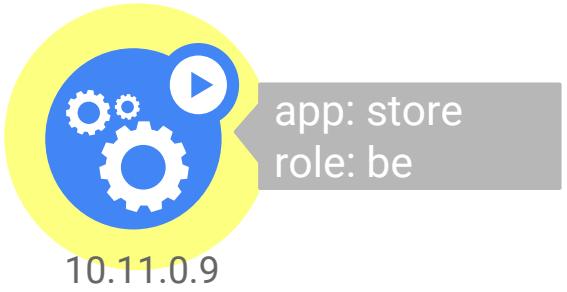












Endpoints

When you create a service, a controller wakes up

```
kind: Endpoints
apiVersion: v1
metadata:
  name: store-be
  namespace: default
subsets:
- addresses:
  - ip: 10.11.8.67
  - ip: 10.11.5.3
  - ip: 10.11.0.9
  ports:
  - name: http
    port: 80
    protocol: TCP
```

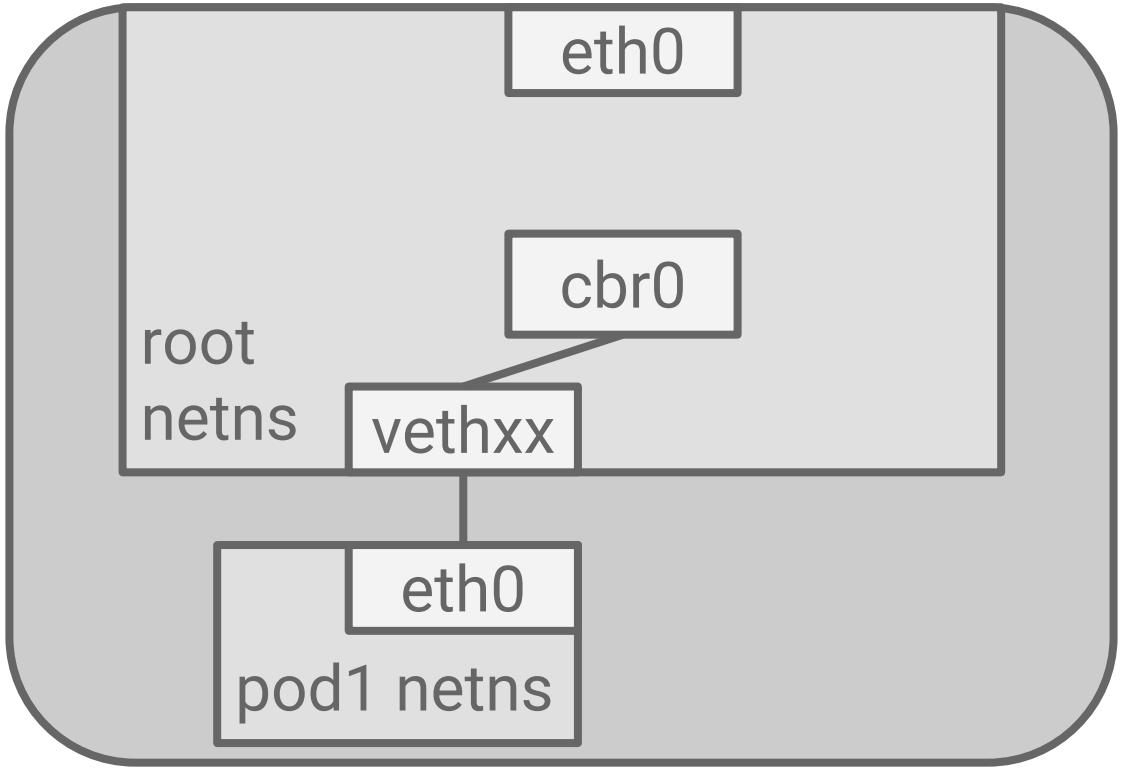
Endpoints

When you create a service, a controller wakes up

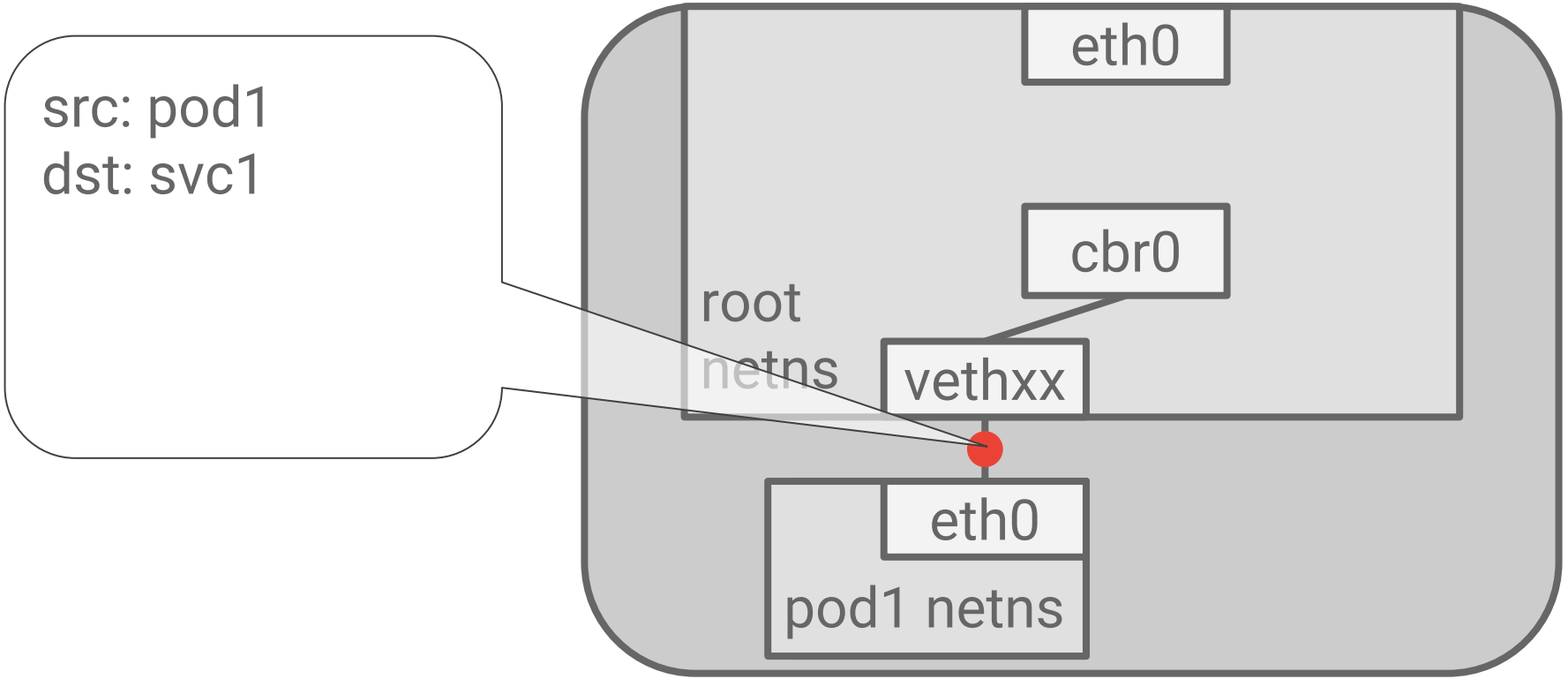
Holds the IPs of the pod backends

```
kind: Endpoints
apiVersion: v1
metadata:
  name: store-be
  namespace: default
subsets:
- addresses:
  - ip: 10.11.8.67
  - ip: 10.11.5.3
  - ip: 10.11.0.9
  ports:
  - name: http
    port: 80
    protocol: TCP
```

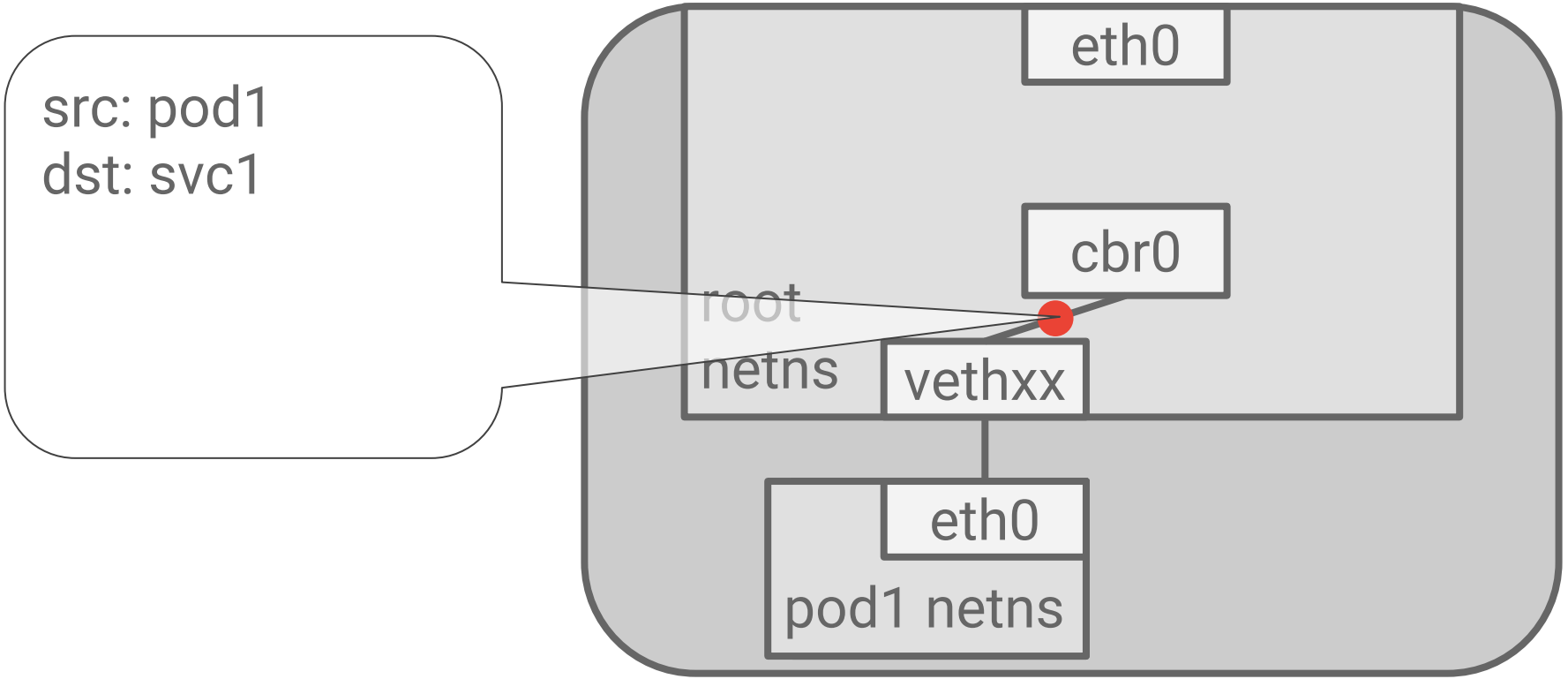
Life of a packet: pod-to-service



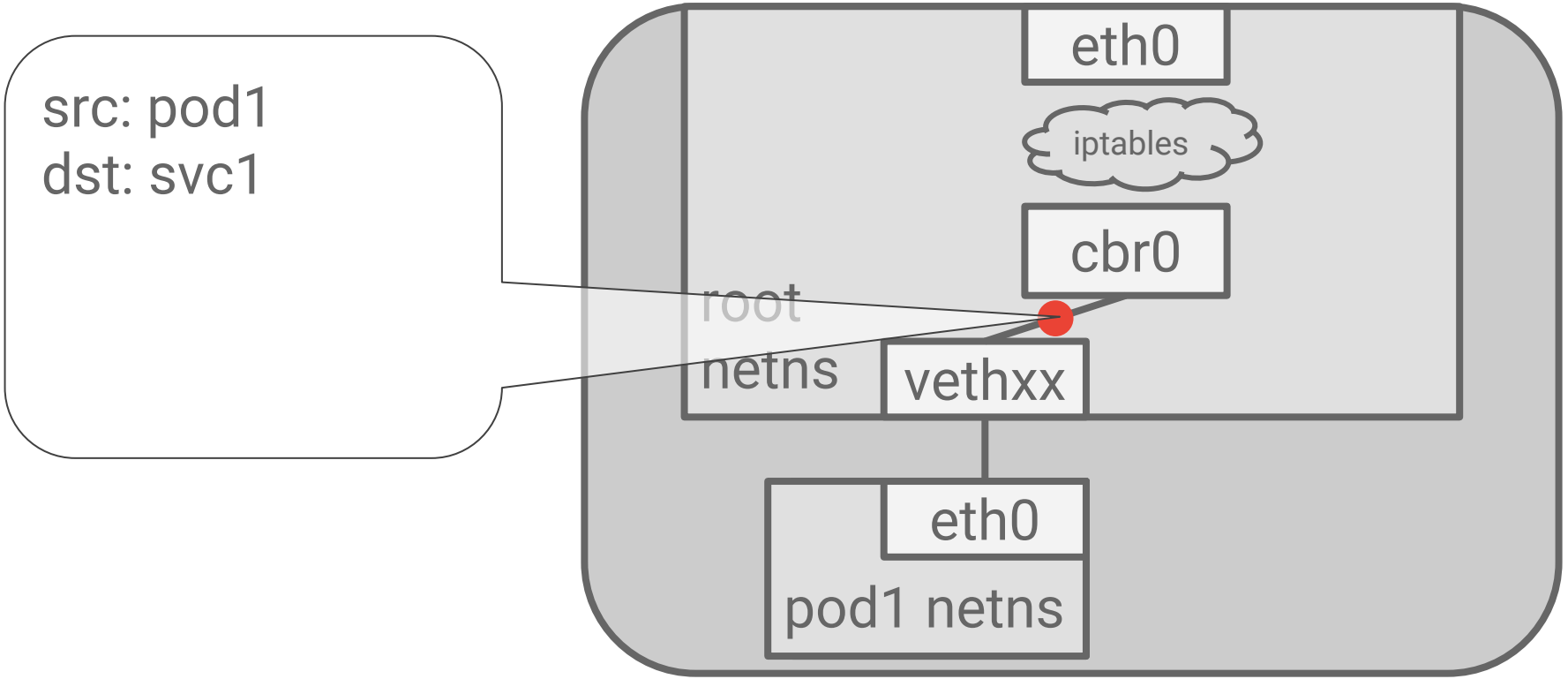
Life of a packet: pod-to-service



Life of a packet: pod-to-service



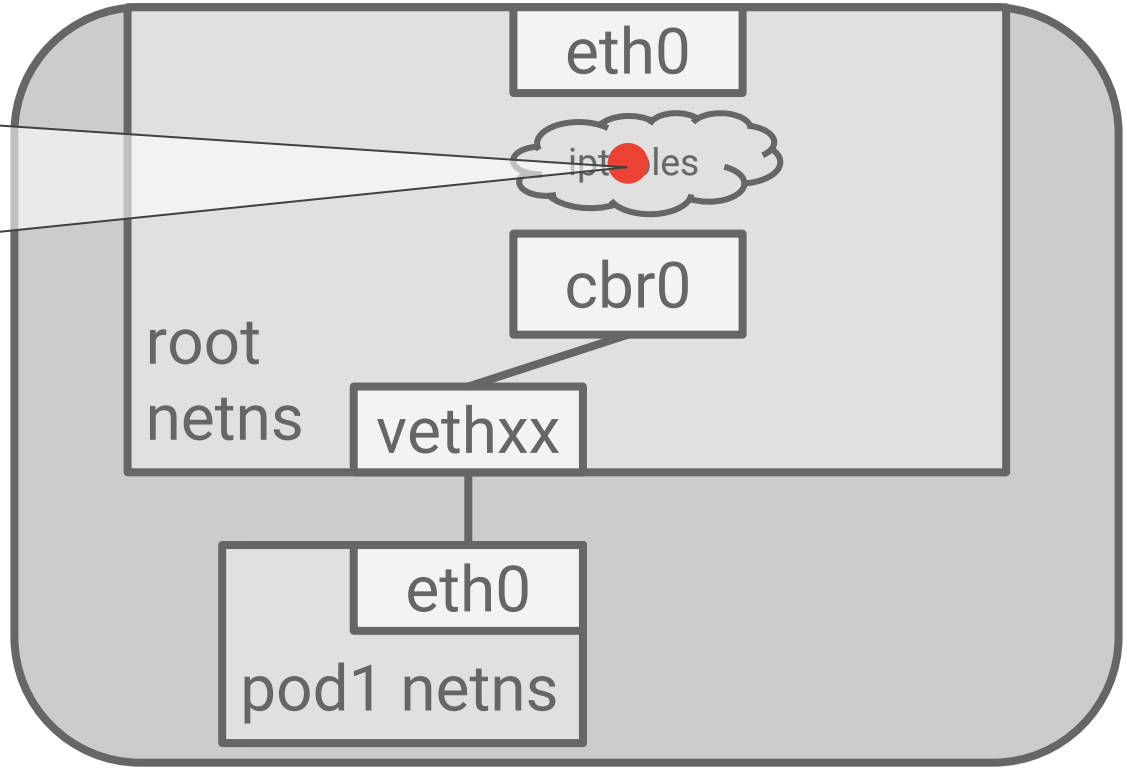
Life of a packet: pod-to-service



Life of a packet: pod-to-service

src: pod1
~~dst: svc1~~
dst: pod99

DNAT, conntrack



Conntrack

Linux kernel connection-tracking

Remembers address translations

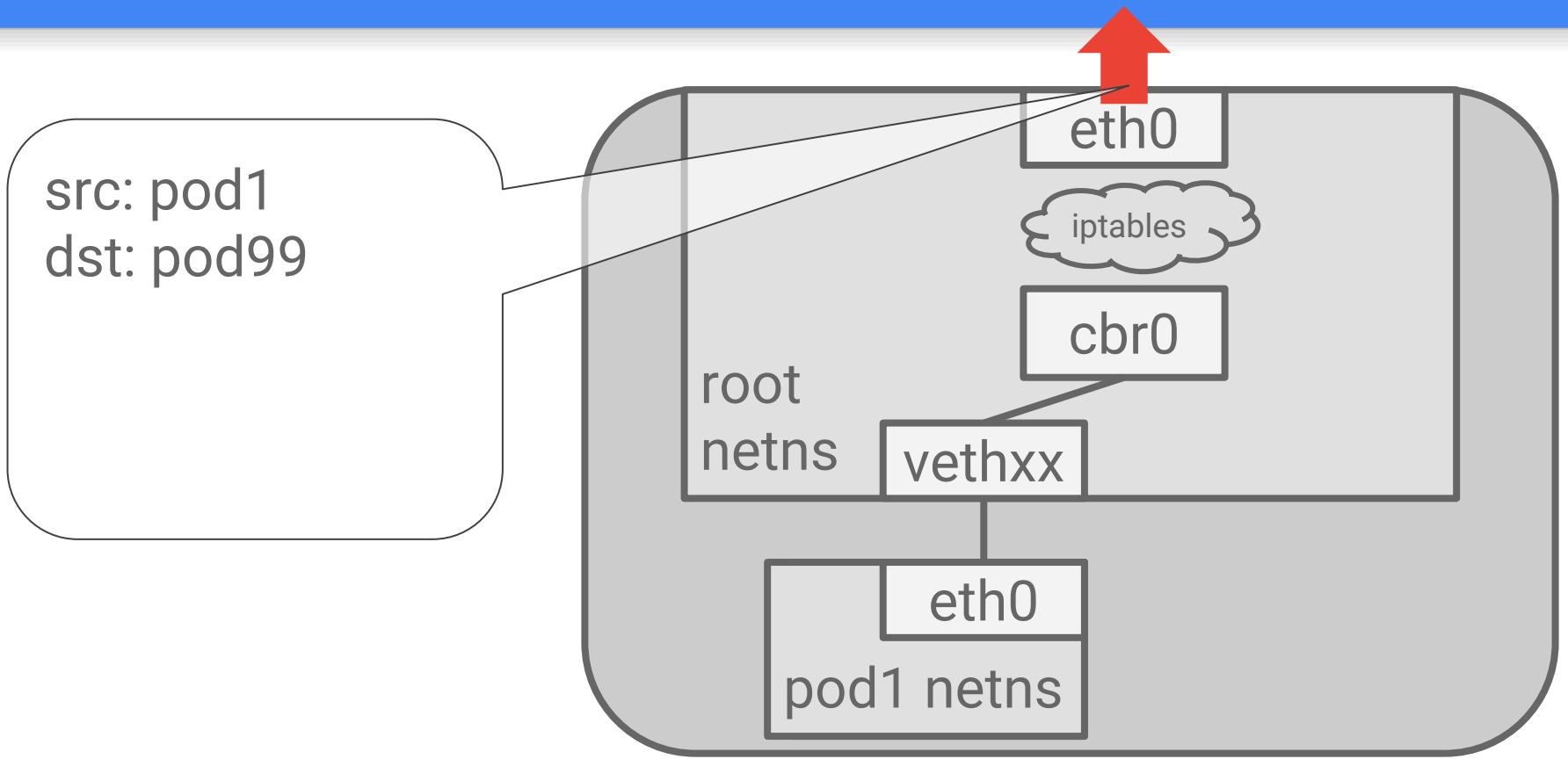
- Based on the 5-tuple

Does a lot more, but not very relevant here

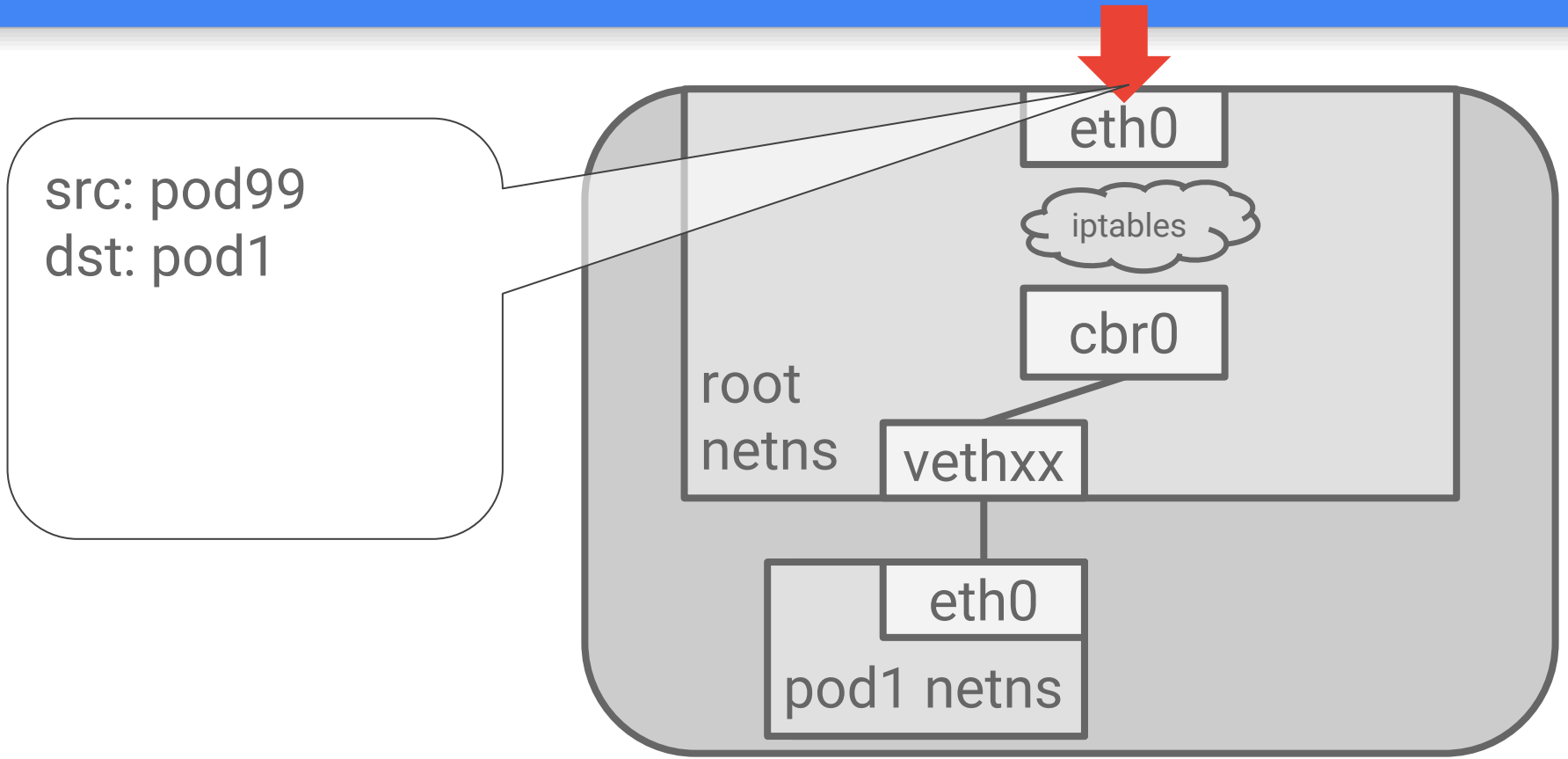
Reversed on the return path

```
{
    protocol = TCP
    src_ip = pod1
    src_port = 1234
    dst_ip = svc1
    dst_port = 80
} => {
    protocol = TCP
    src_ip = pod1
    src_port = 1234
    dst_ip = pod99
    dst_port = 80
}
```

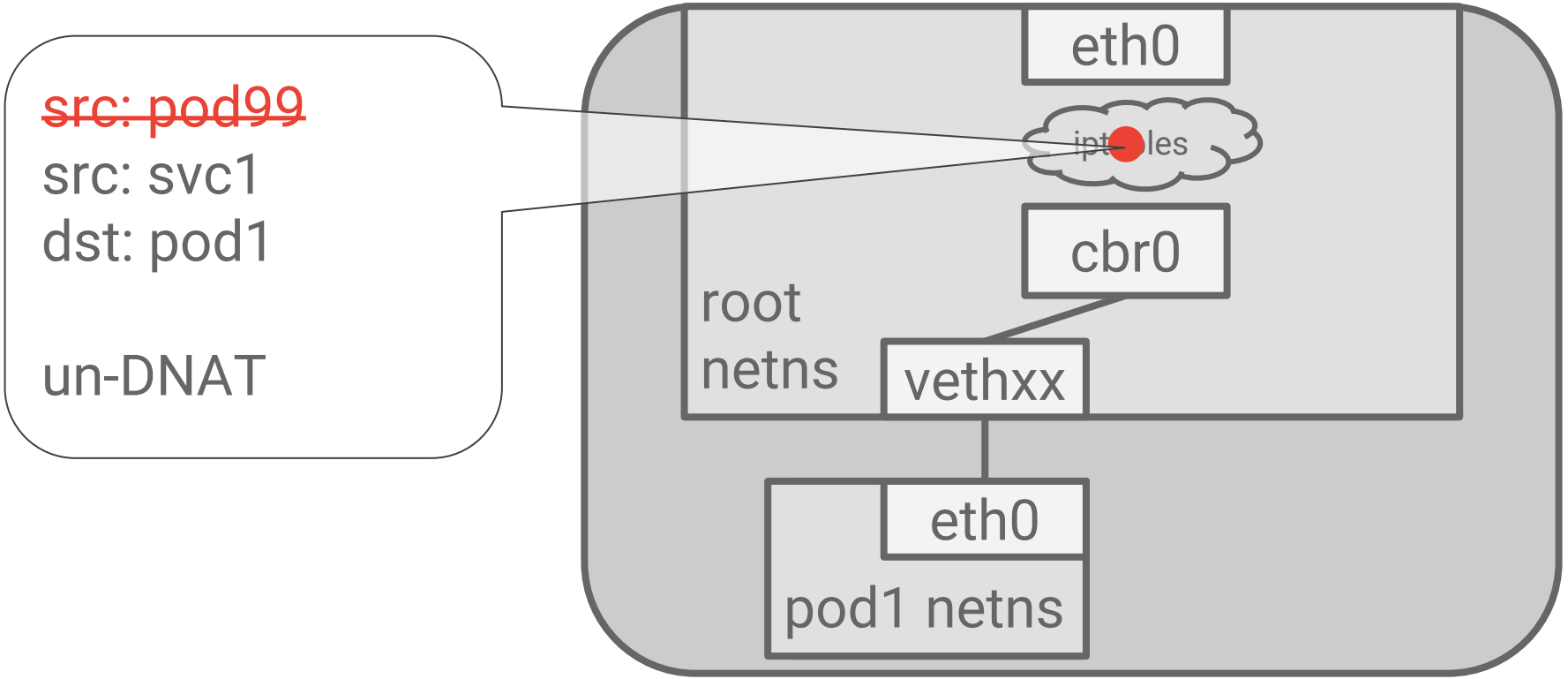
Life of a packet: pod-to-service



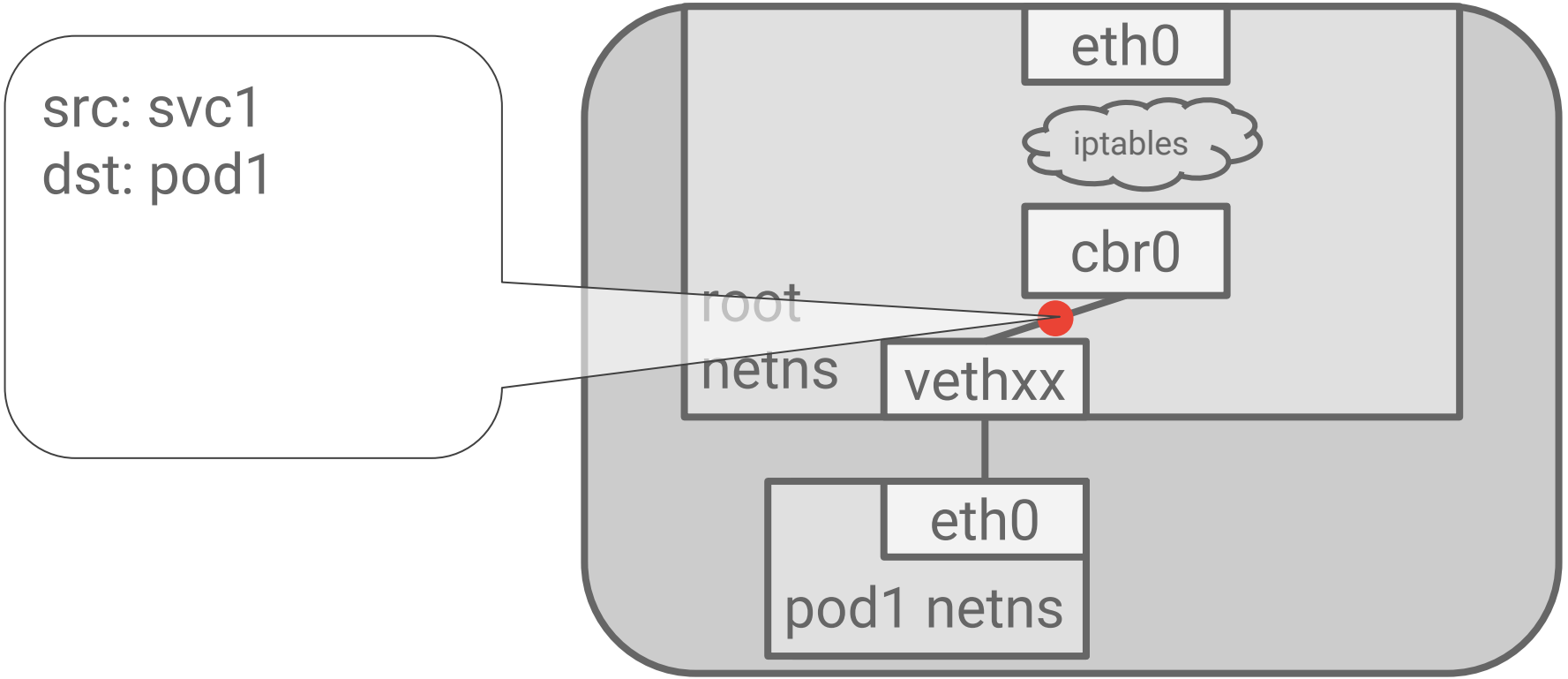
Life of a packet: pod-to-service



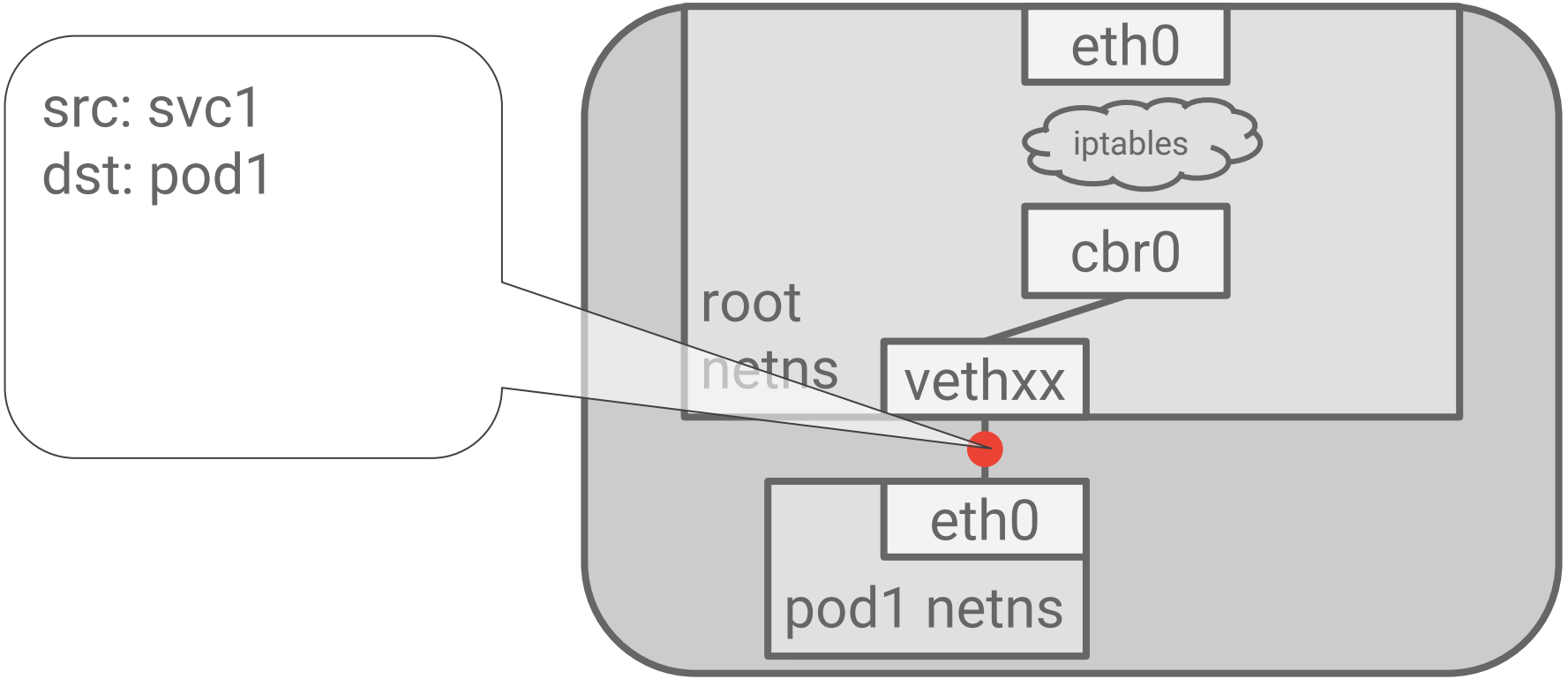
Life of a packet: pod-to-service



Life of a packet: pod-to-service



Life of a packet: pod-to-service



A bit more on iptables

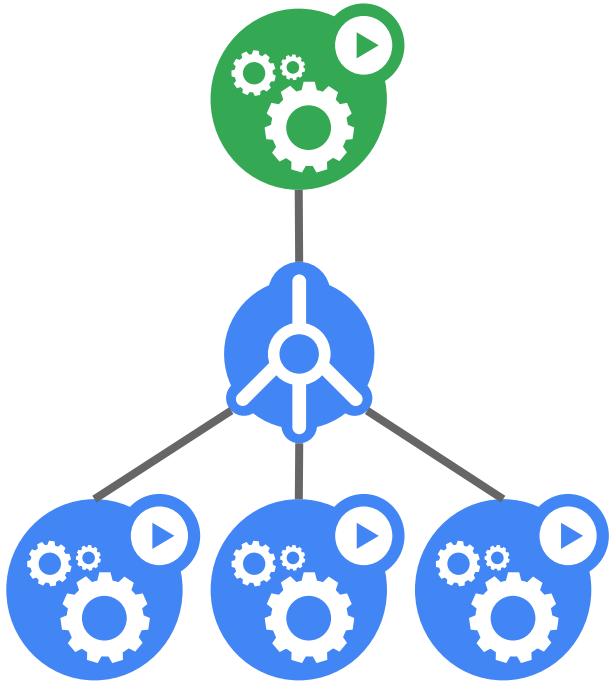
The iptables rules look scary, but are actually simple:

```
if dest.ip == svc1.ip && dest.port == svc1.port {
  pick one of the backends at random
  rewrite destination IP
}
```

Configured by ‘kube-proxy’ - a pod running on each Node

- Not actually a proxy
- Not in the data path

Kube-proxy is a controller - it watches the API for services



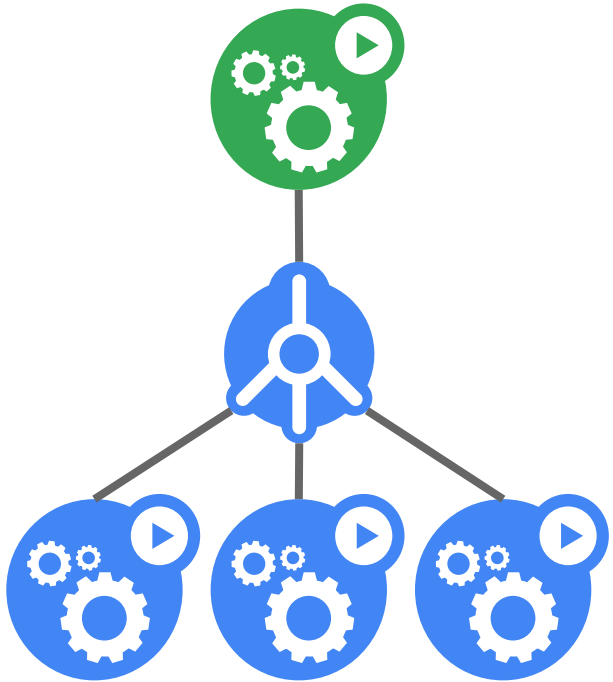
DNS

Even easier: services are added to an in-cluster DNS server

You would never hardcode an IP, but you might hardcode a hostname and port

Serves “A” and “SRV” records

DNS itself runs as pods and a service



DNS Service

Requests a particular cluster IP

Pods are auto-scaled with the cluster size

Service VIP is stable

```
kind: Service
apiVersion: v1
metadata:
  name: kube-dns
  namespace: kube-system
spec:
  clusterIP: 10.0.0.10
  selector:
    k8s-app: kube-dns
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

DNS Service

Requests a particular cluster IP

Pods are auto-scaled with the cluster size

Service VIP is stable

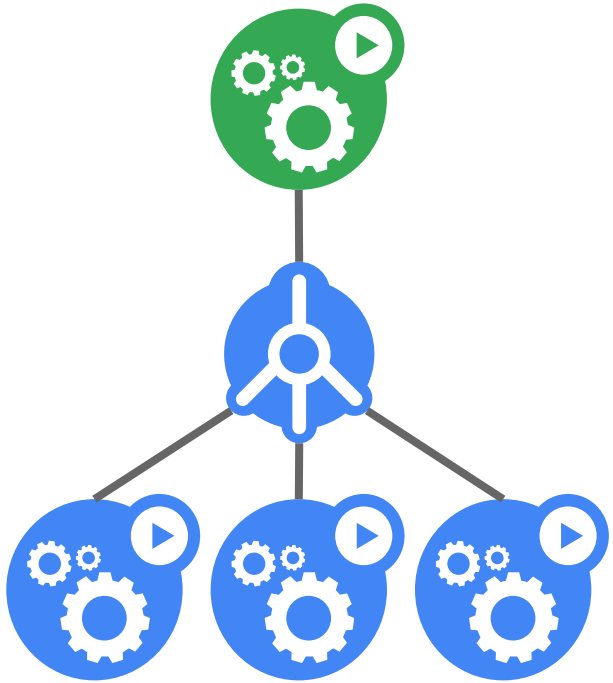
```
kind: Service
apiVersion: v1
metadata:
  name: kube-dns
  namespace: kube-system
spec:
  clusterIP: 10.0.0.10
  selector:
    k8s-app: kube-dns
  ports:
  - name: dns
    port: 53
    protocol: UDP
  - name: dns-tcp
    port: 53
    protocol: TCP
```

Simple and powerful

Can use any port you want, no conflicts

Can request a particular ‘clusterIP’

Can remap ports

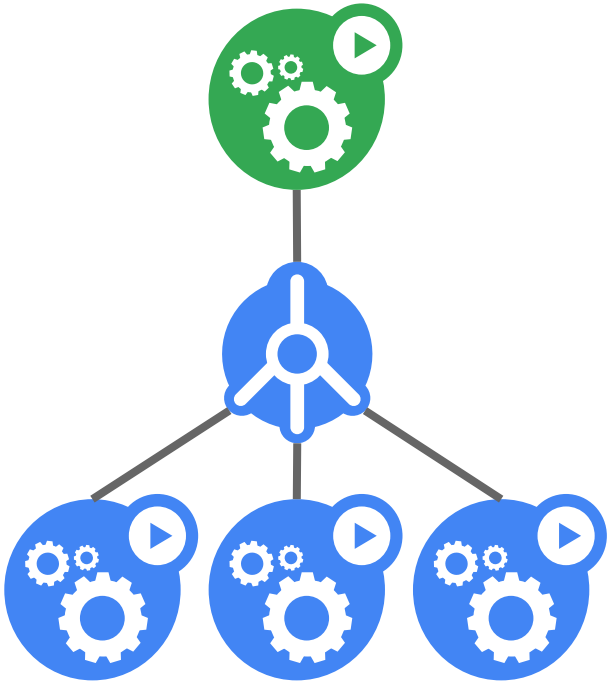


That’s all there is to it

Services are an abstraction - the API is a VIP

No running process or intercepting the data-path

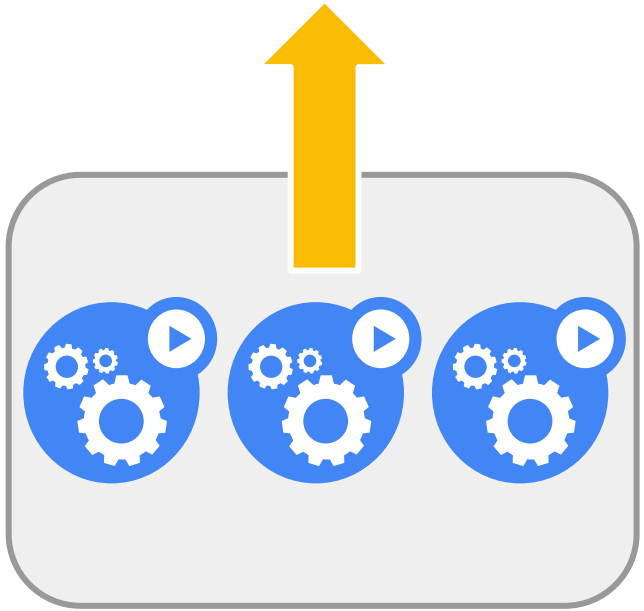
All a client needs to do is hit the service IP:port



Sending external traffic

Services are within a cluster

What happens if you want your pod to reach google.com?



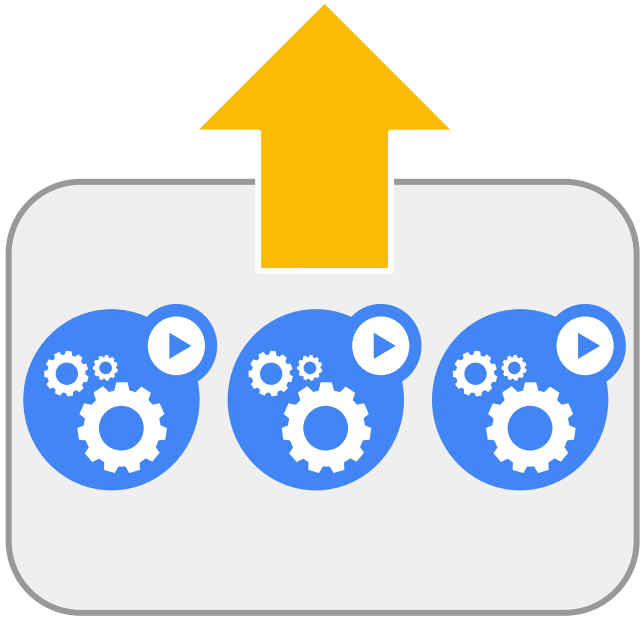
Egress

Leaving the GCP project

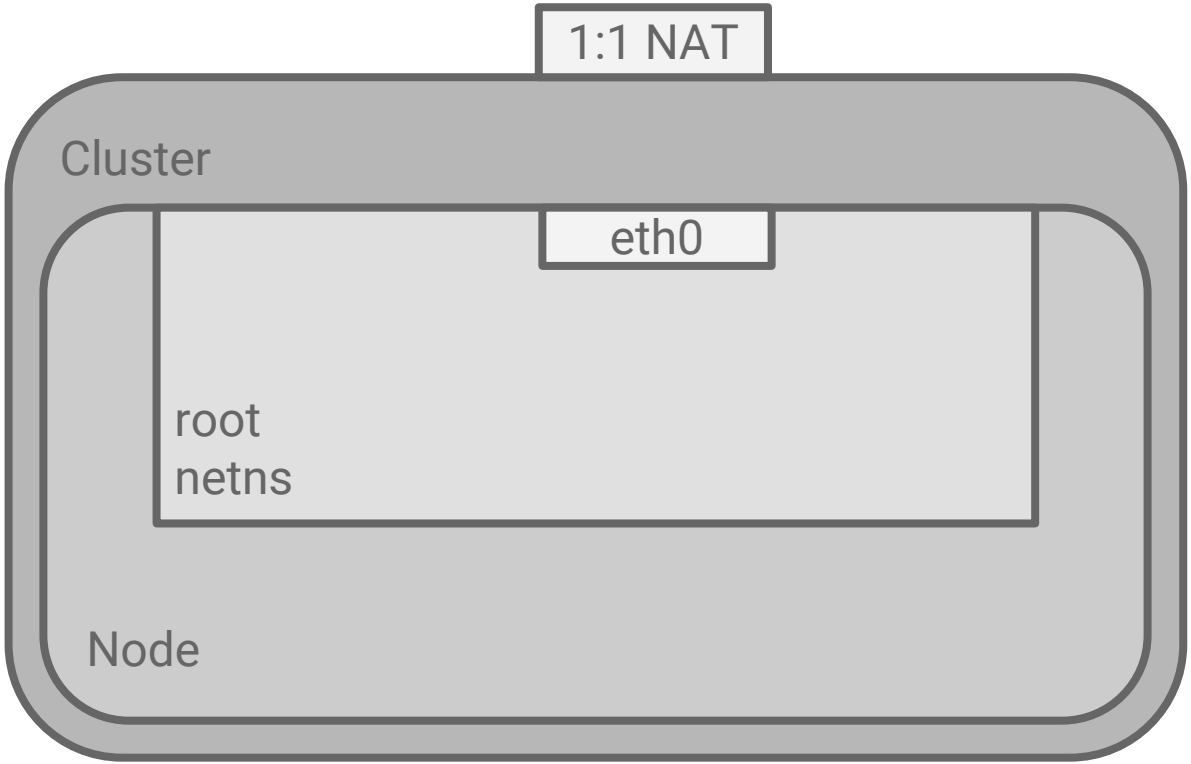
Nodes get private IPs (in 10.0.0.0/8)

Nodes can have public IPs, too

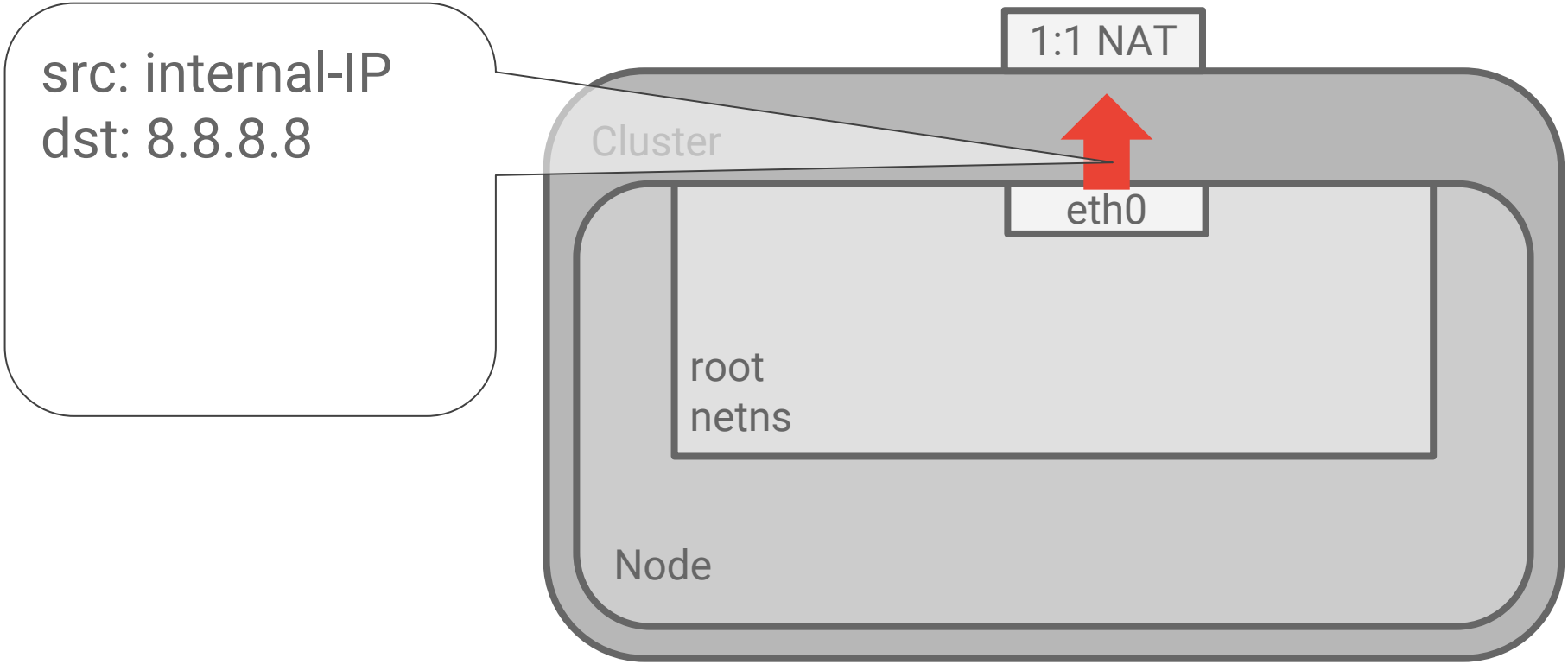
GCP: Public IPs are provided by 1-to-1 NAT



Life of a packet: Node-to-internet

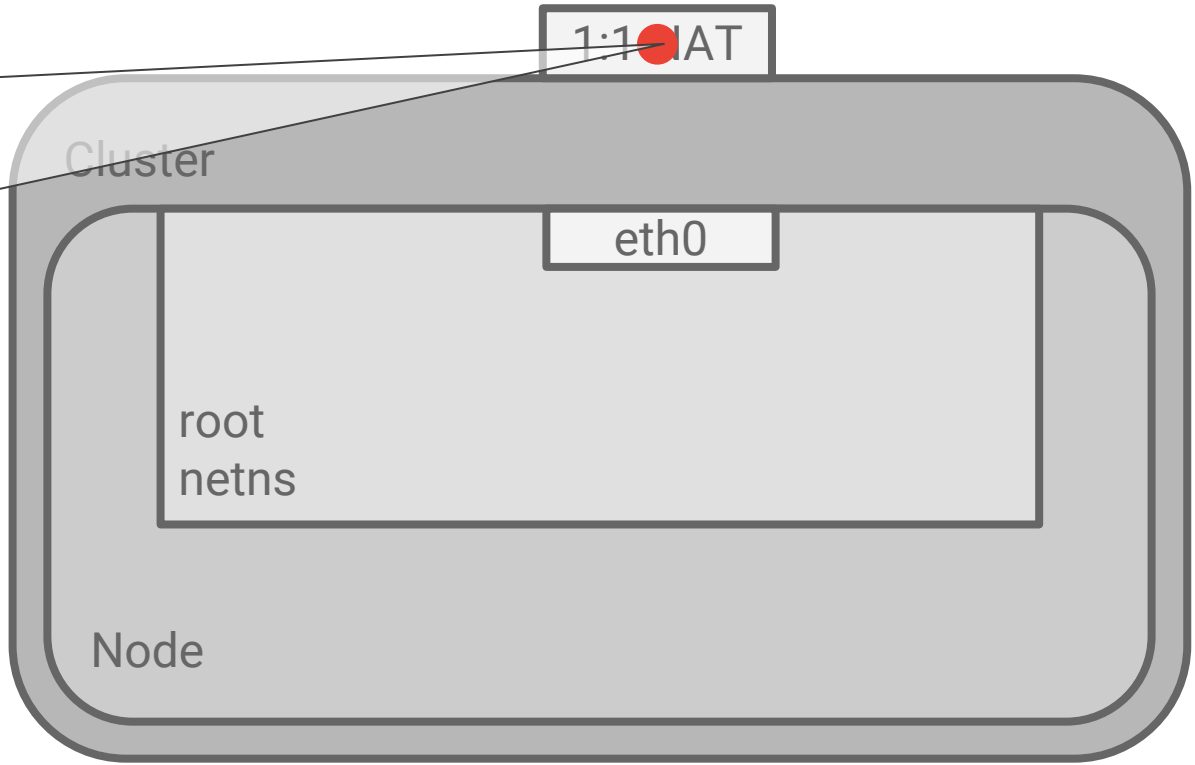


Life of a packet: Node-to-internet

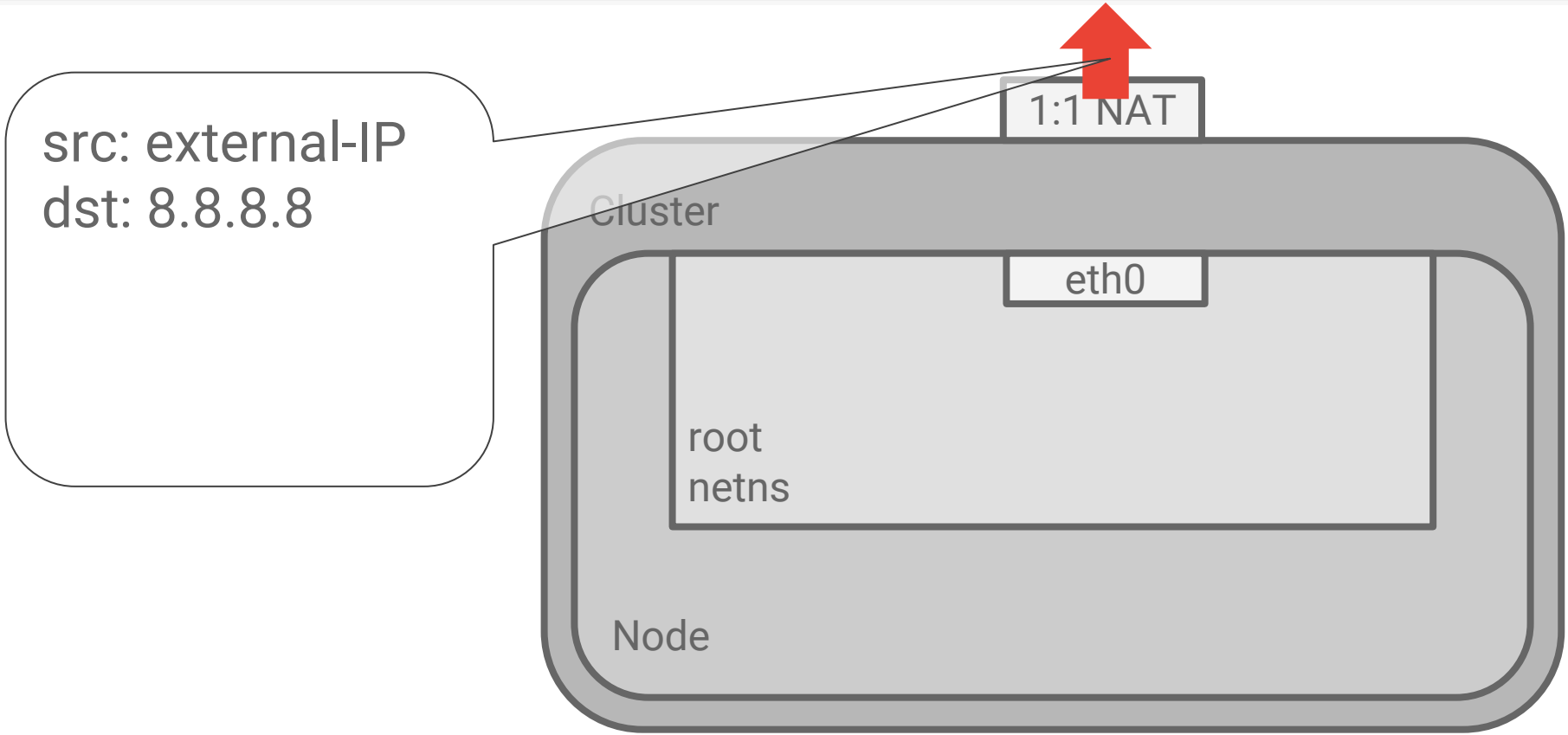


Life of a packet: Node-to-internet

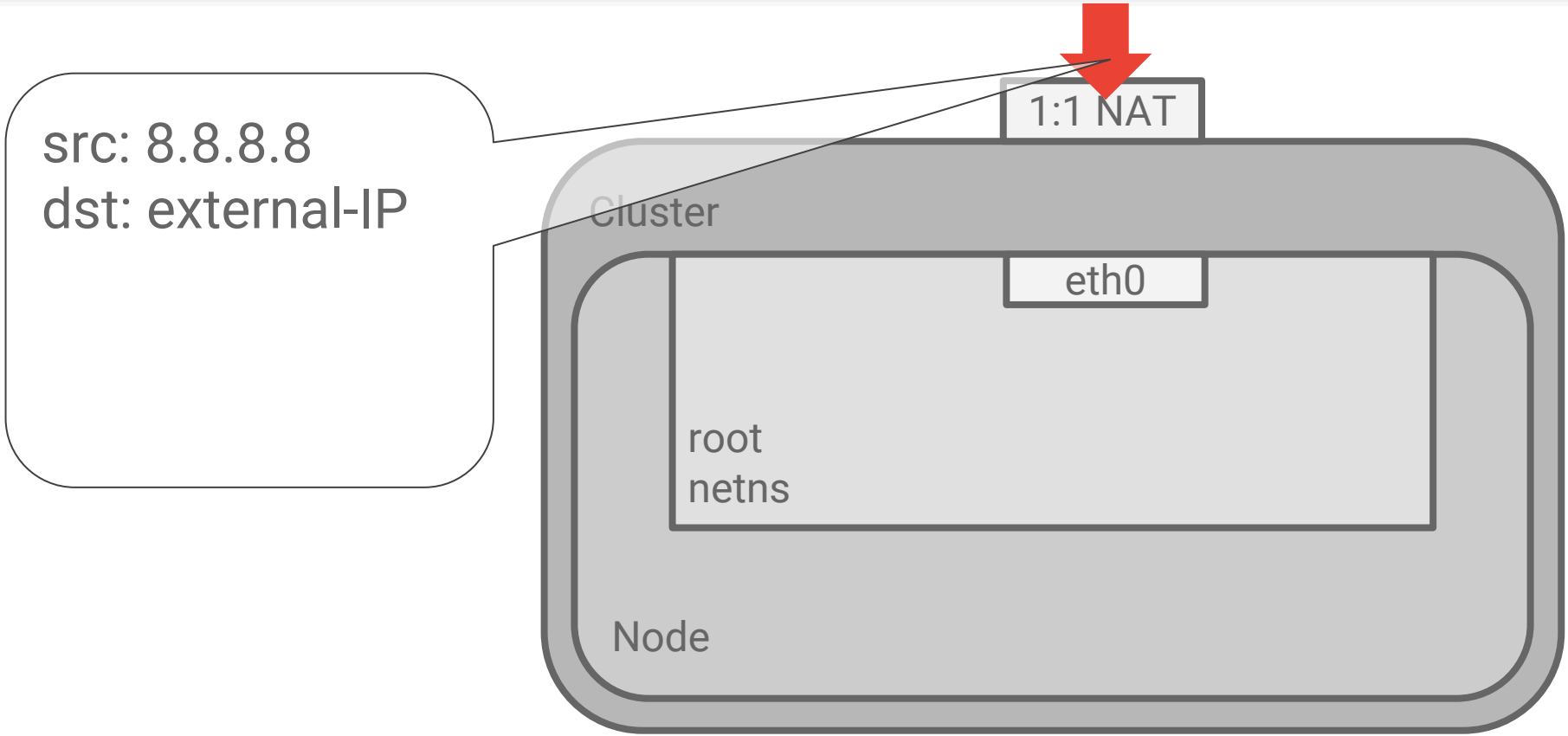
~~src: internal-IP~~
src: external-IP
dst: 8.8.8.8



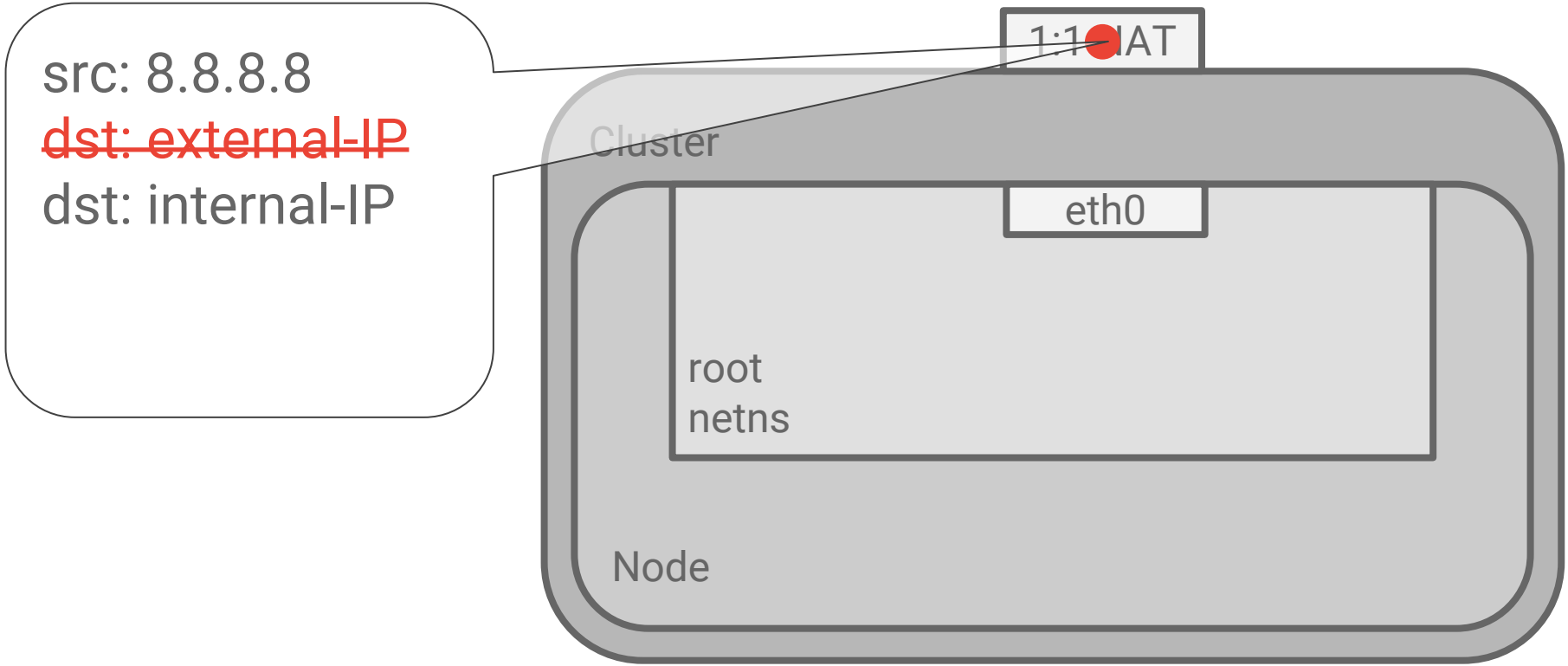
Life of a packet: Node-to-internet



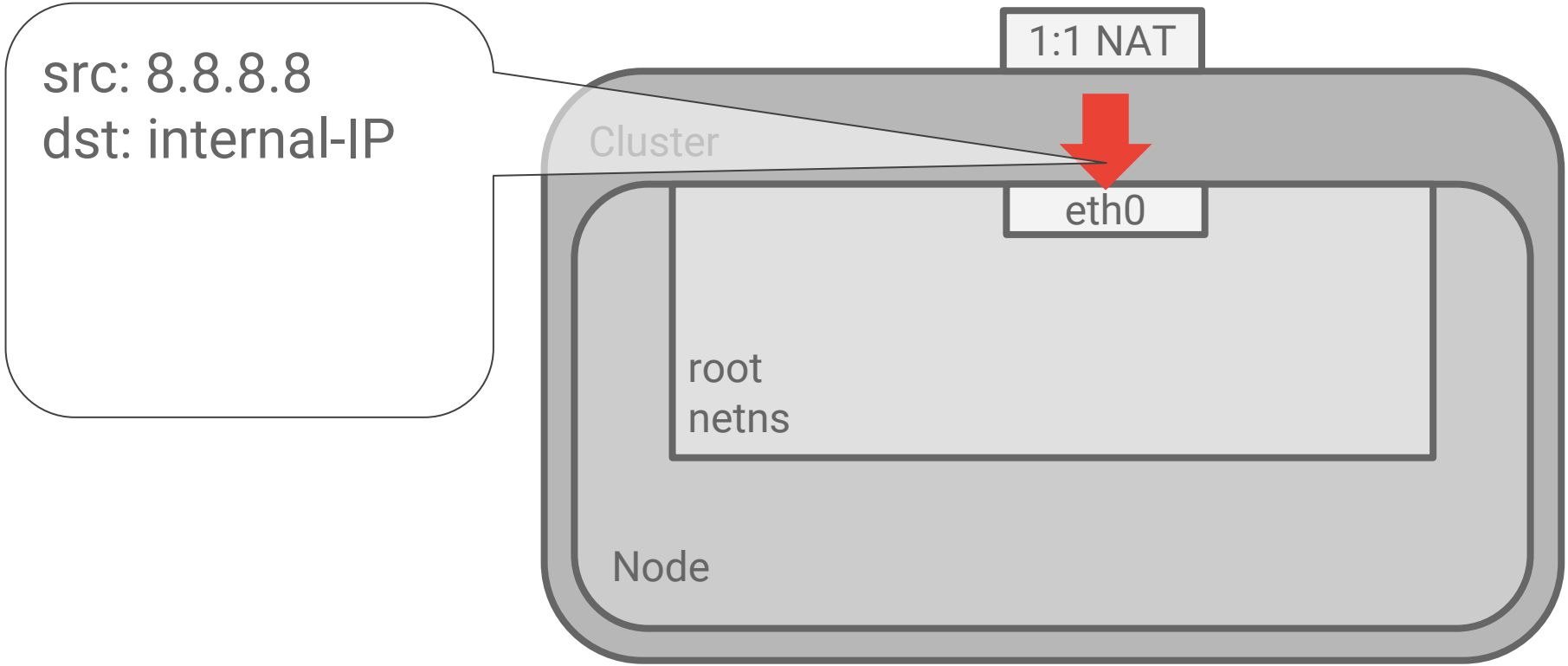
Life of a packet: Node-to-internet



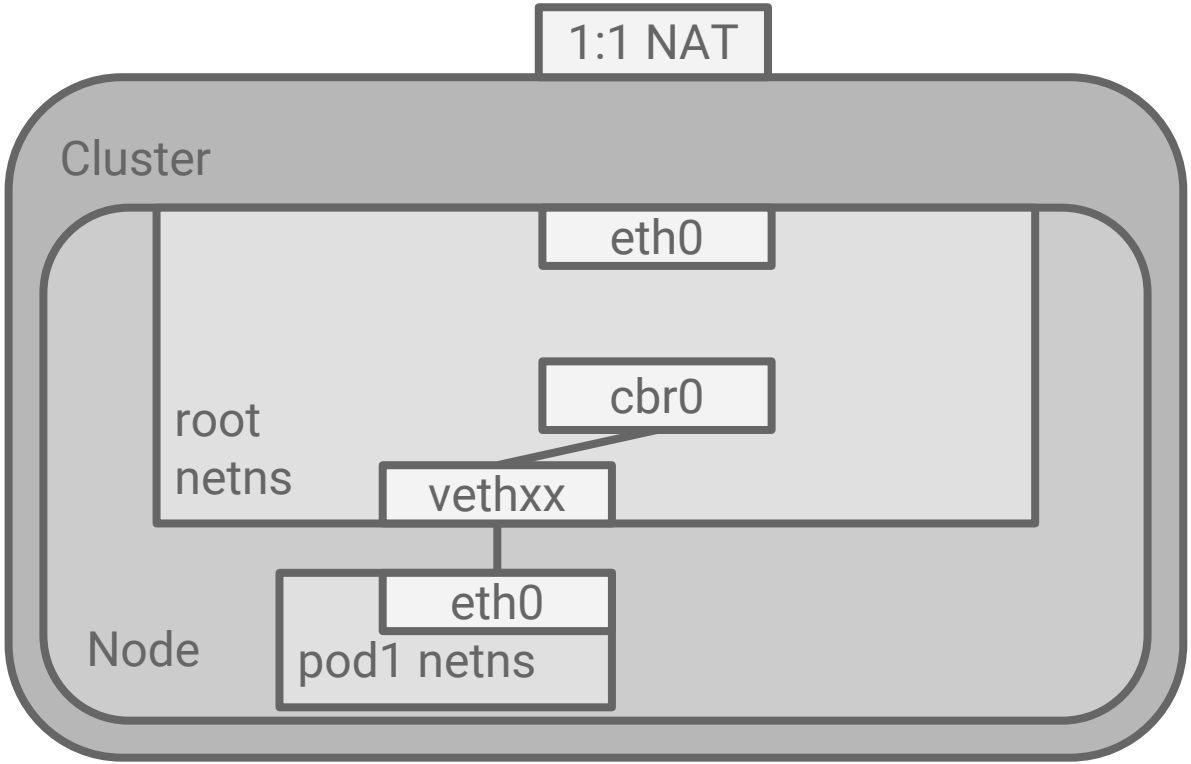
Life of a packet: Node-to-internet



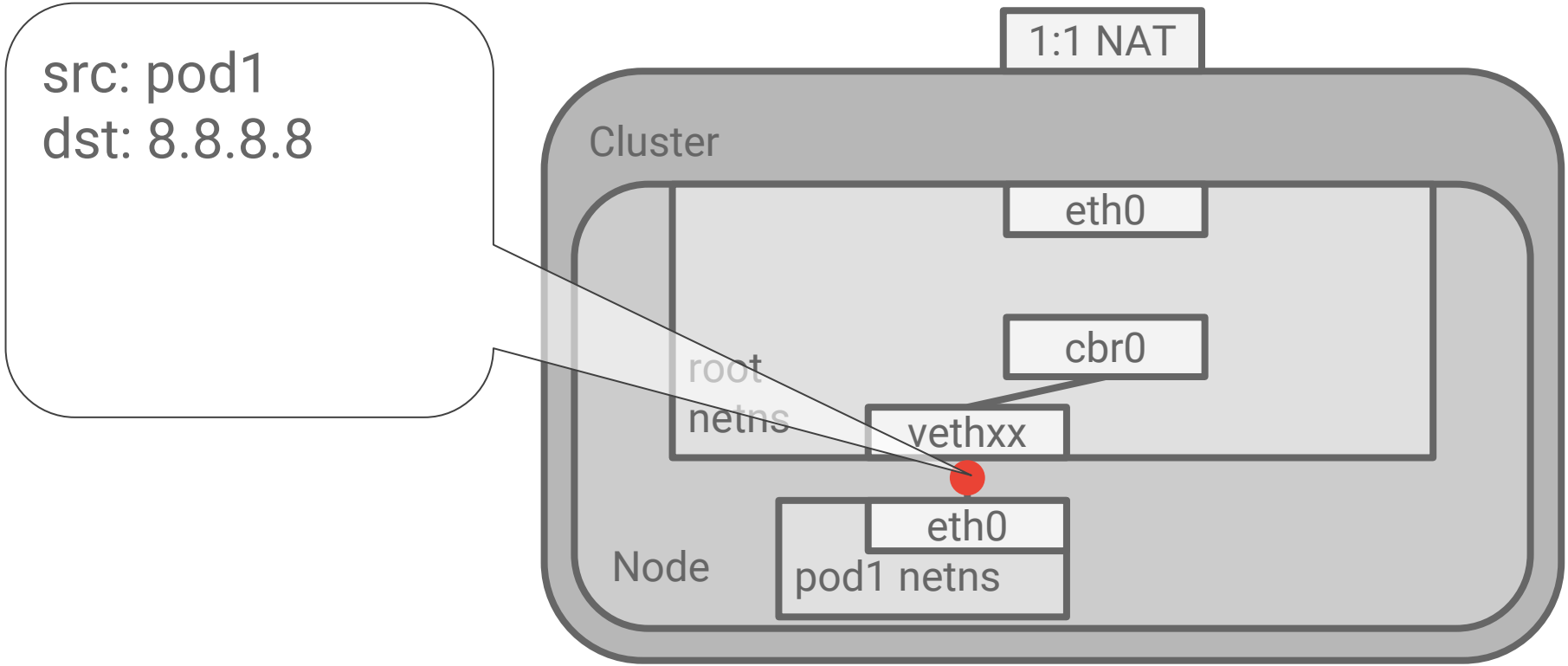
Life of a packet: Node-to-internet



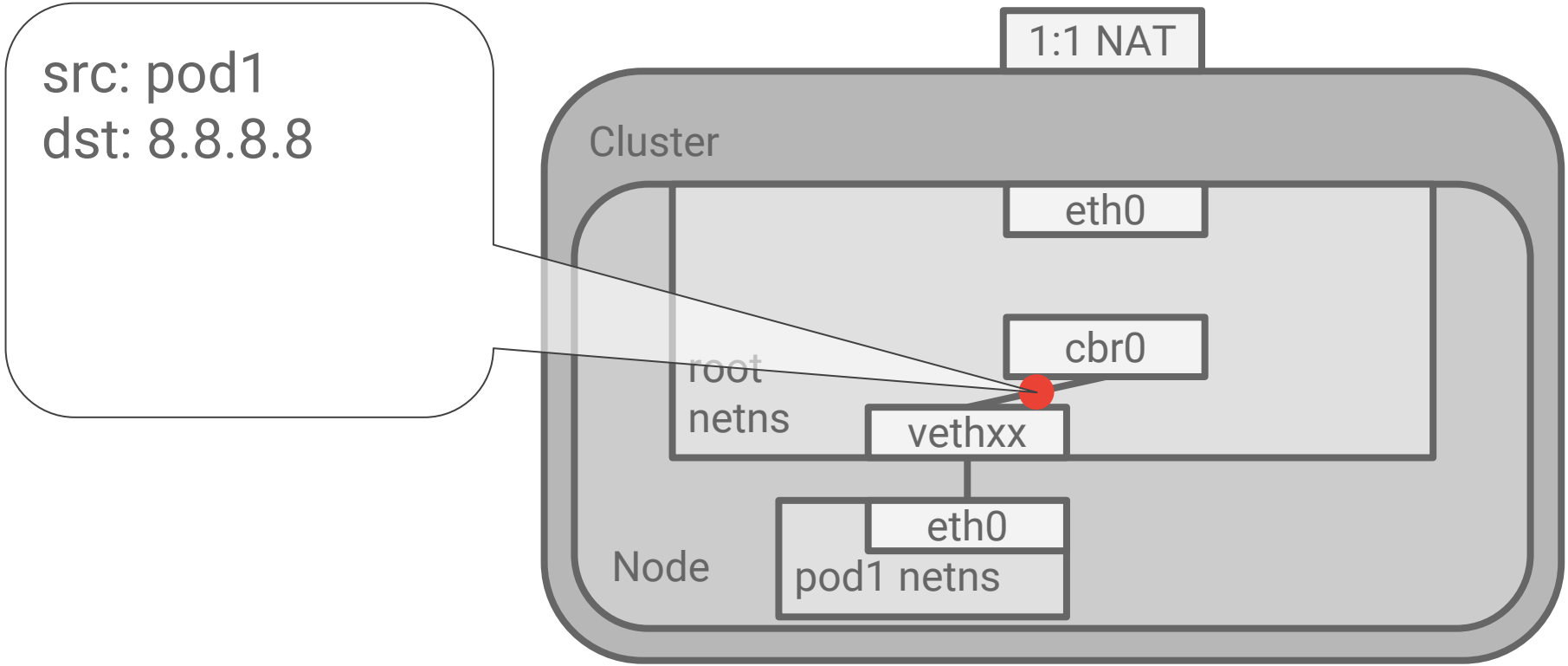
Life of a packet: pod-to-internet



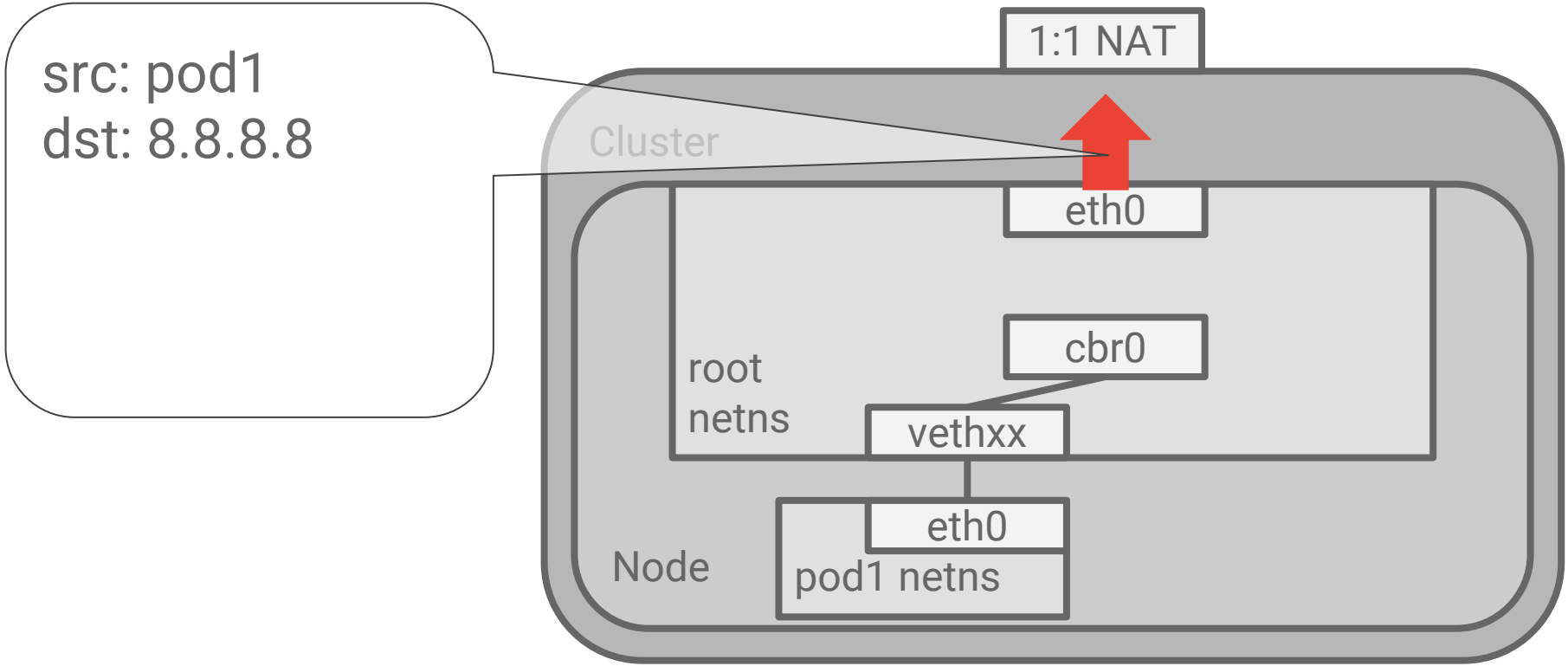
Life of a packet: pod-to-internet



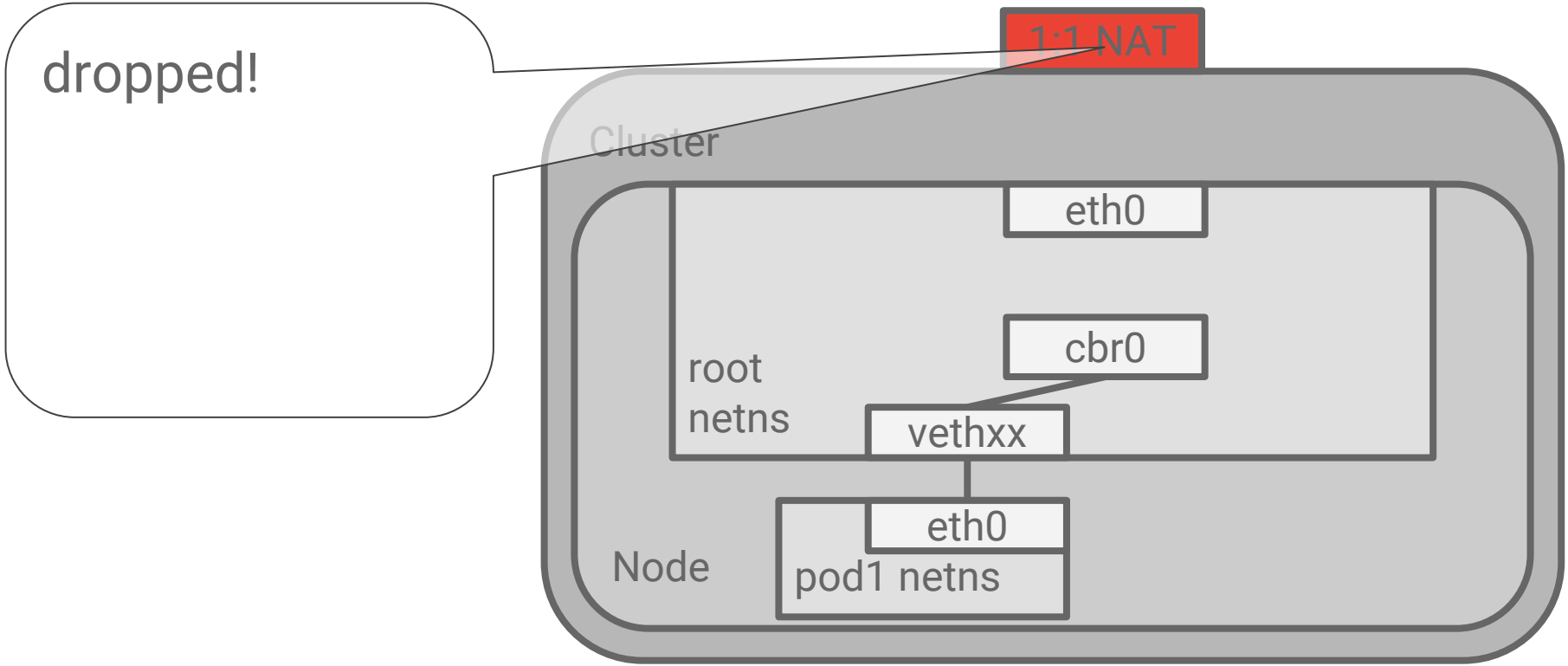
Life of a packet: pod-to-internet



Life of a packet: pod-to-internet



Life of a packet: pod-to-internet



What went wrong?

The 1:1 NAT only understands Node IPs

- Anything else gets dropped

Pod IPs != Node IPs

When in doubt, add some more iptables

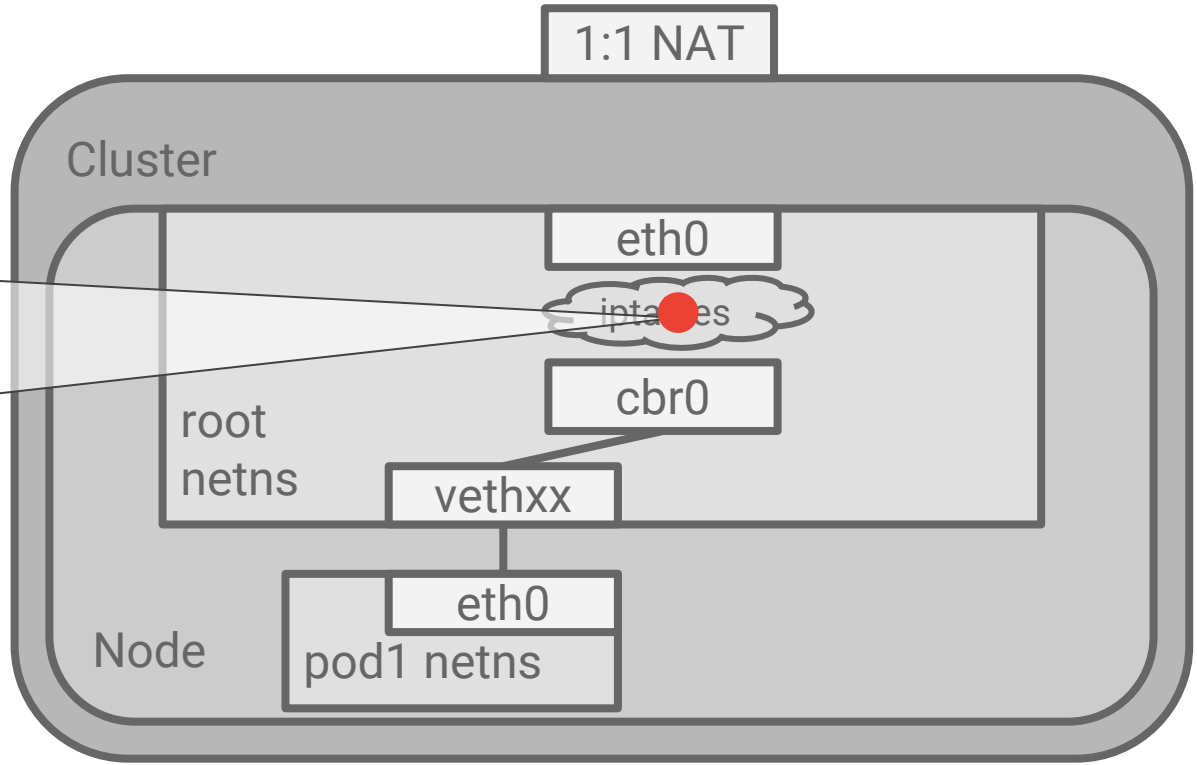
- MASQUERADE, aka SNAT

Applies to any packet with a destination *outside* of 10.0.0.0/8

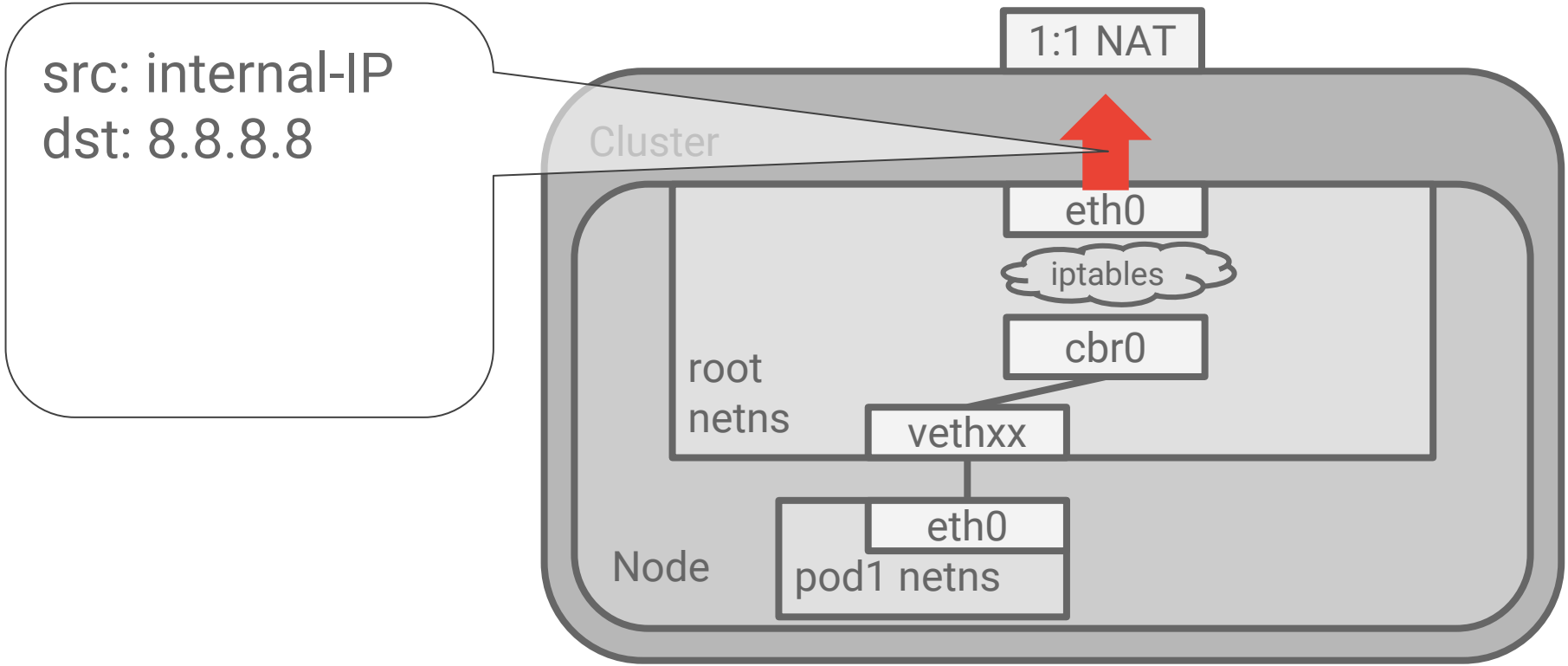
Life of a packet: pod-to-internet

~~src: pod1~~
src: internal-IP
dst: 8.8.8.8

MASQUERADE

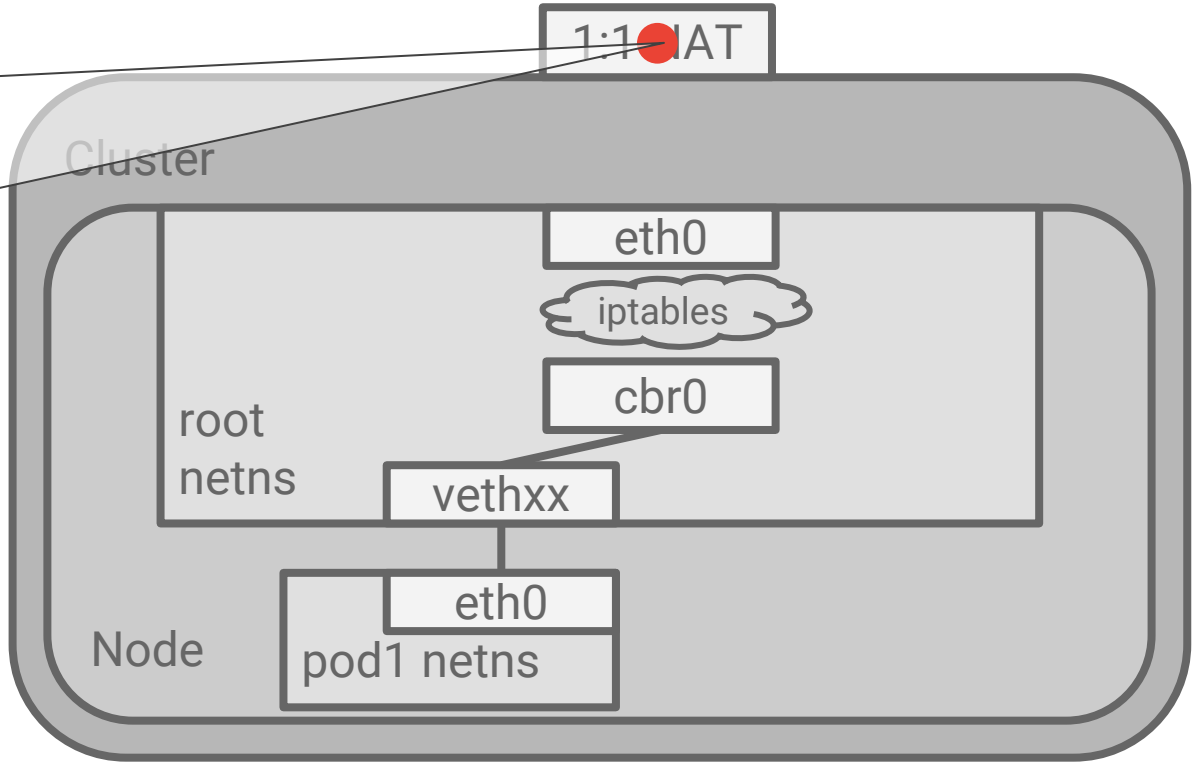


Life of a packet: pod-to-internet



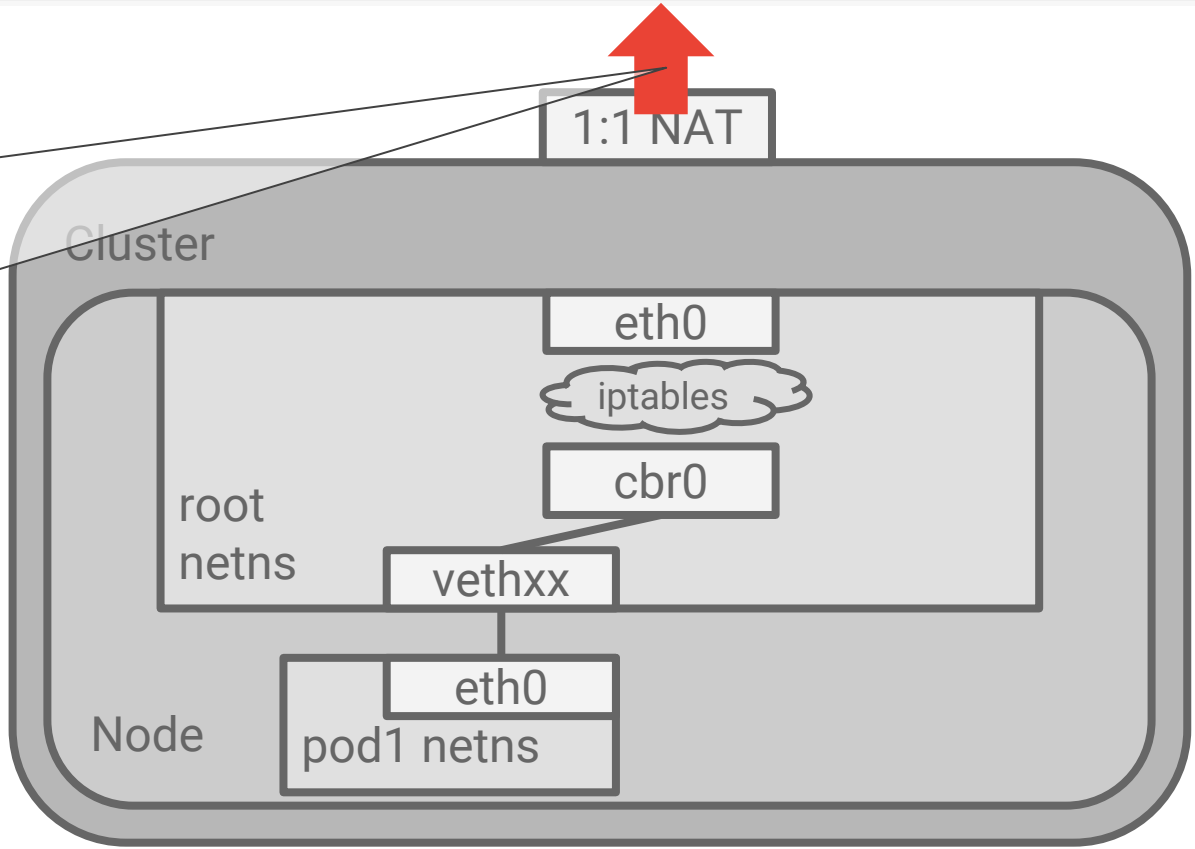
Life of a packet: pod-to-internet

~~src: internal-IP~~
src: external-IP
dst: 8.8.8.8



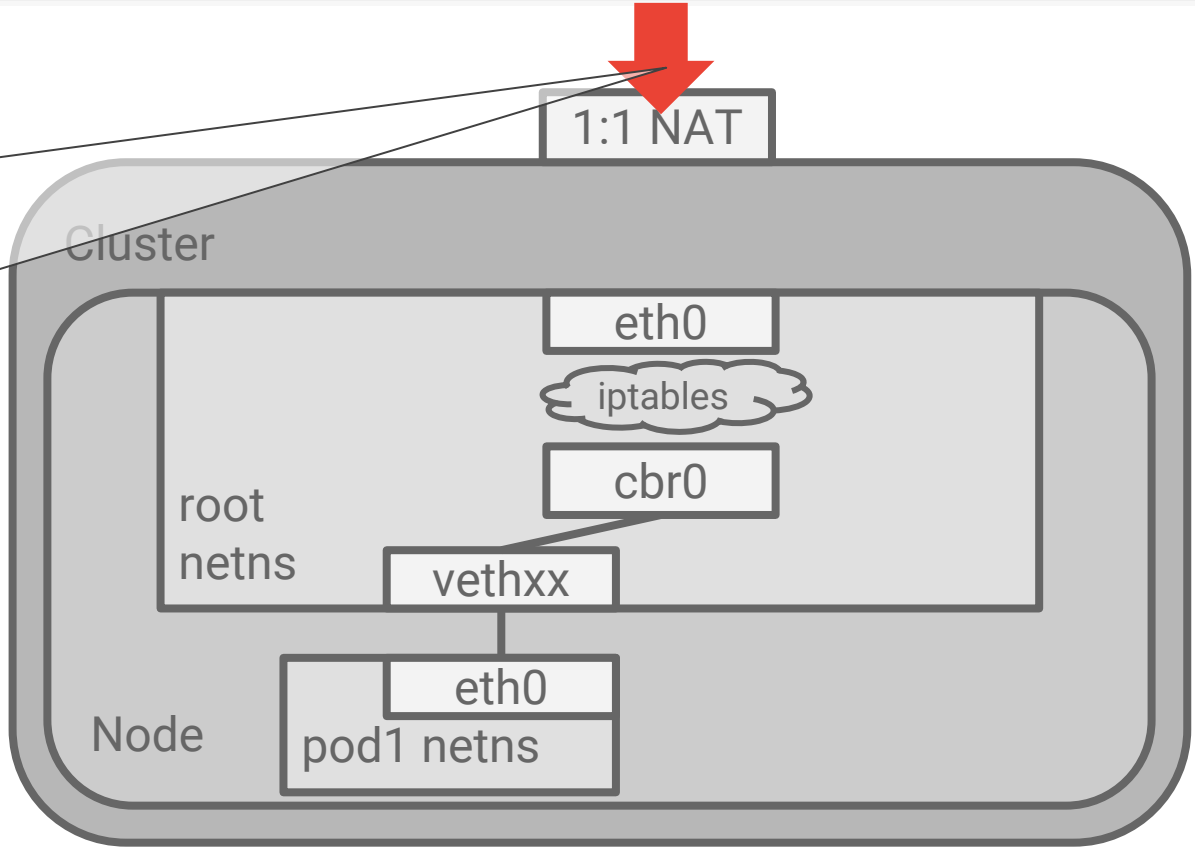
Life of a packet: pod-to-internet

src: external-IP
dst: 8.8.8.8



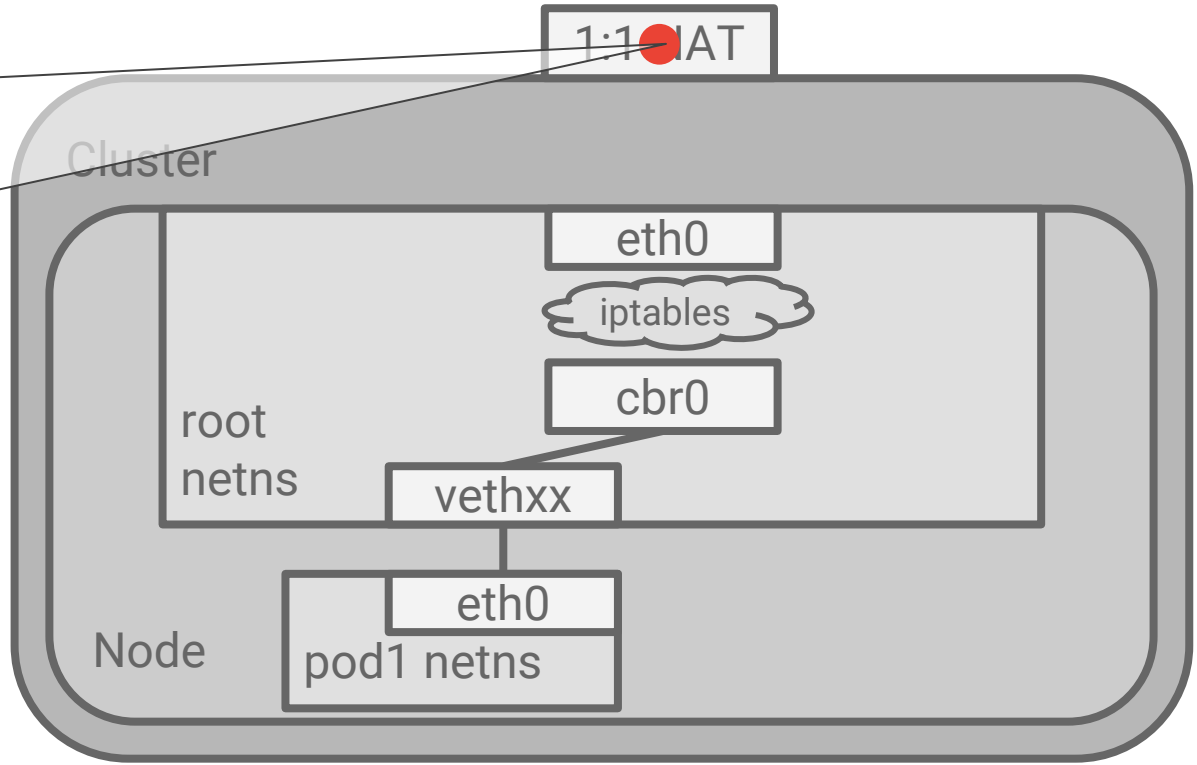
Life of a packet: pod-to-internet

src: 8.8.8.8
dst: external-IP



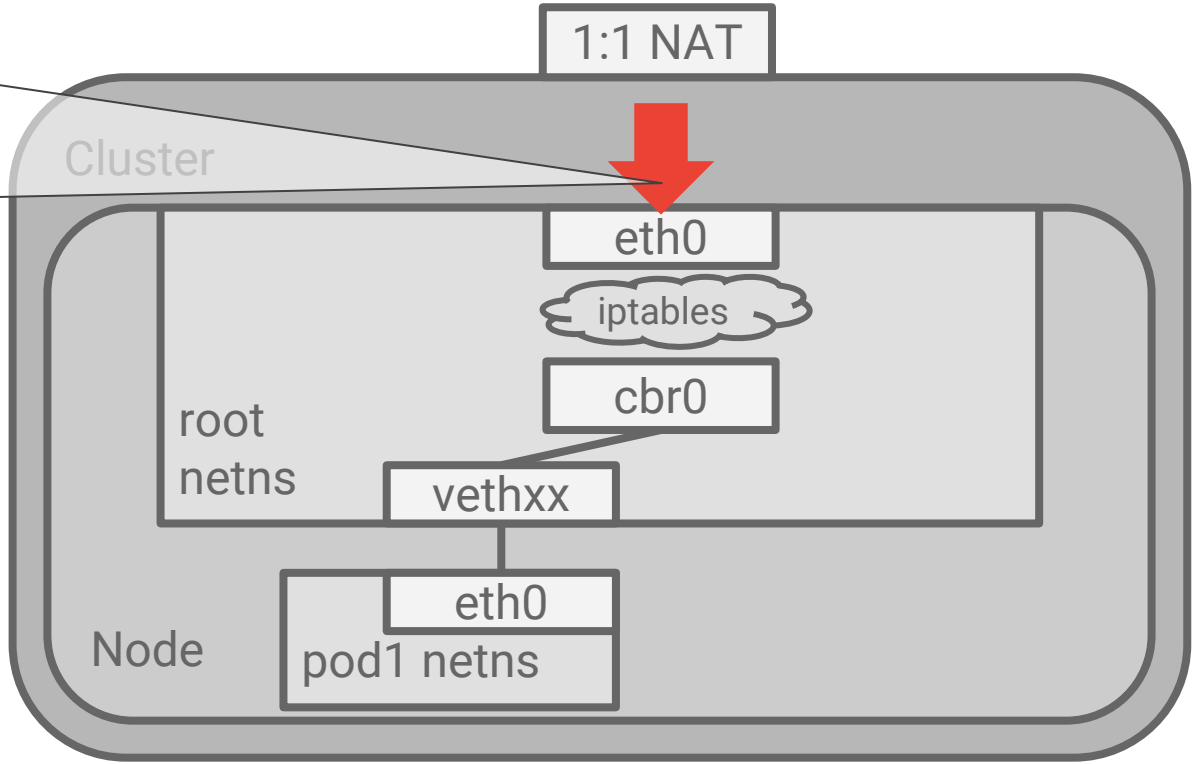
Life of a packet: pod-to-internet

src: 8.8.8.8
~~dst: external-IP~~
dst: internal-IP



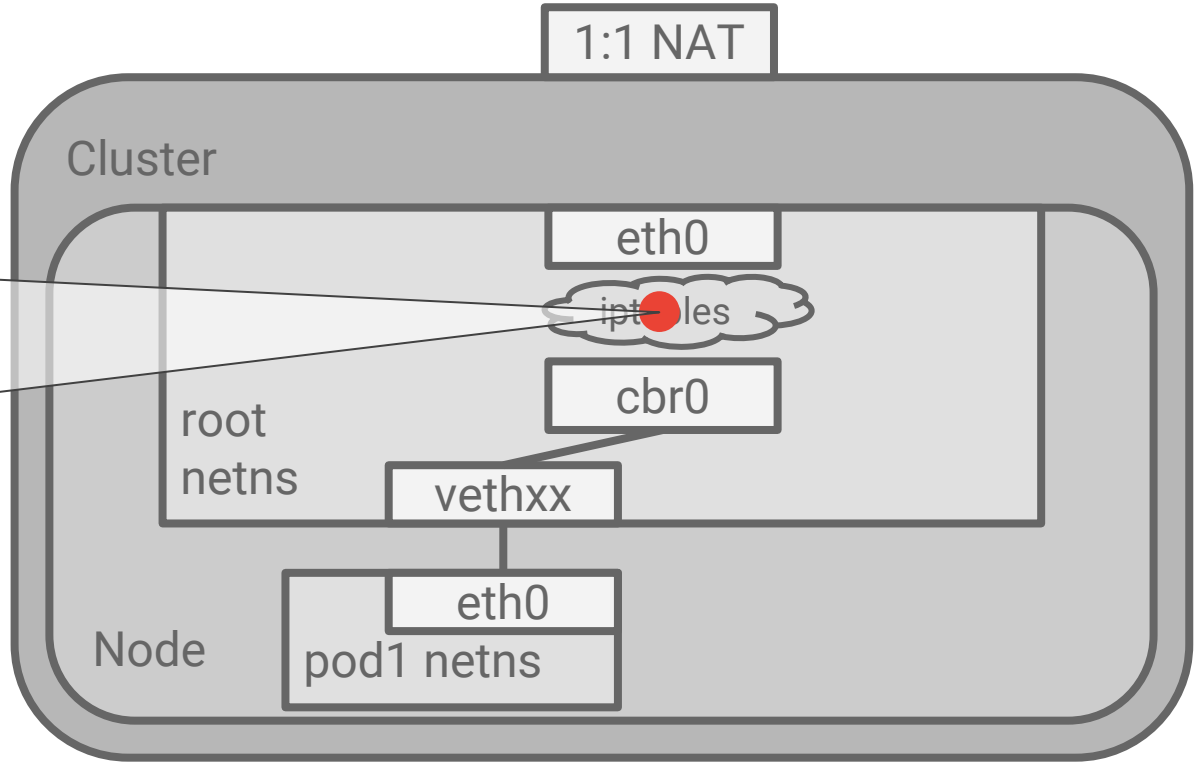
Life of a packet: pod-to-internet

src: 8.8.8.8
dst: internal-IP

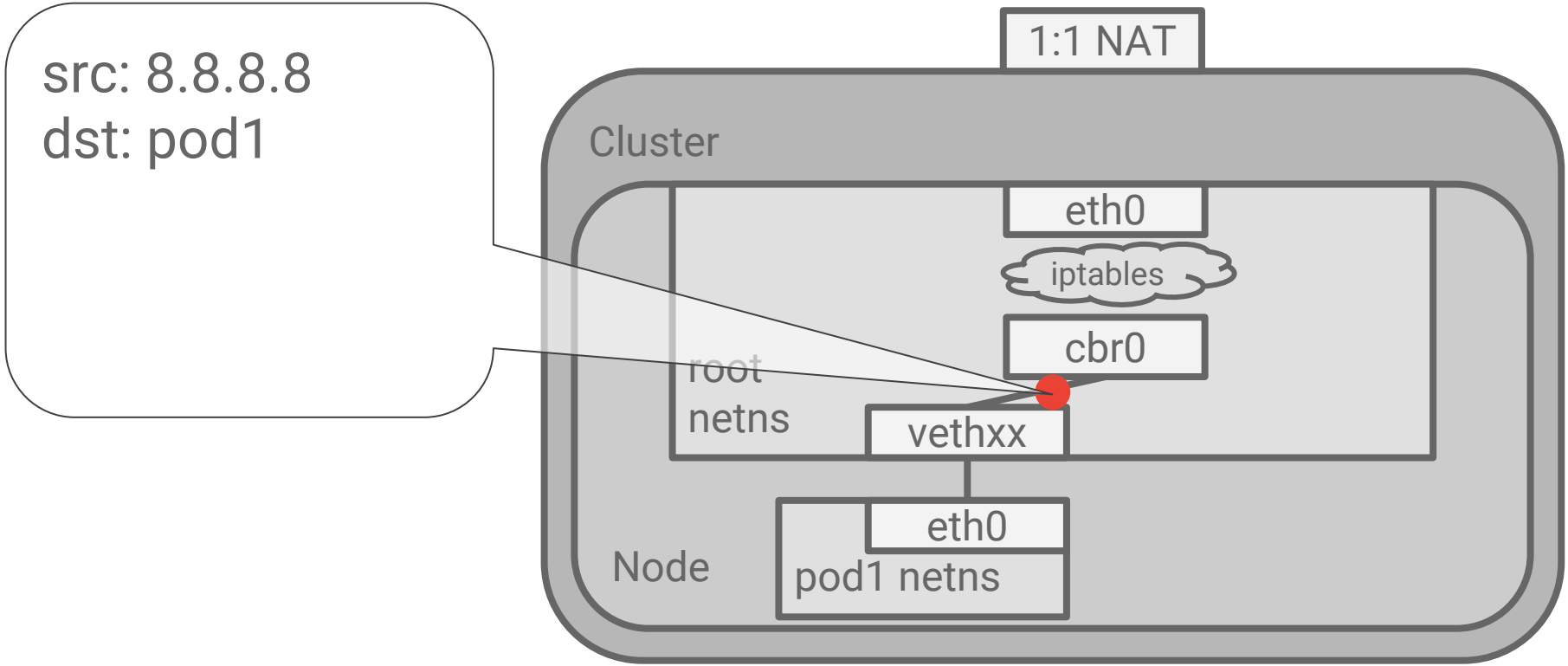


Life of a packet: pod-to-internet

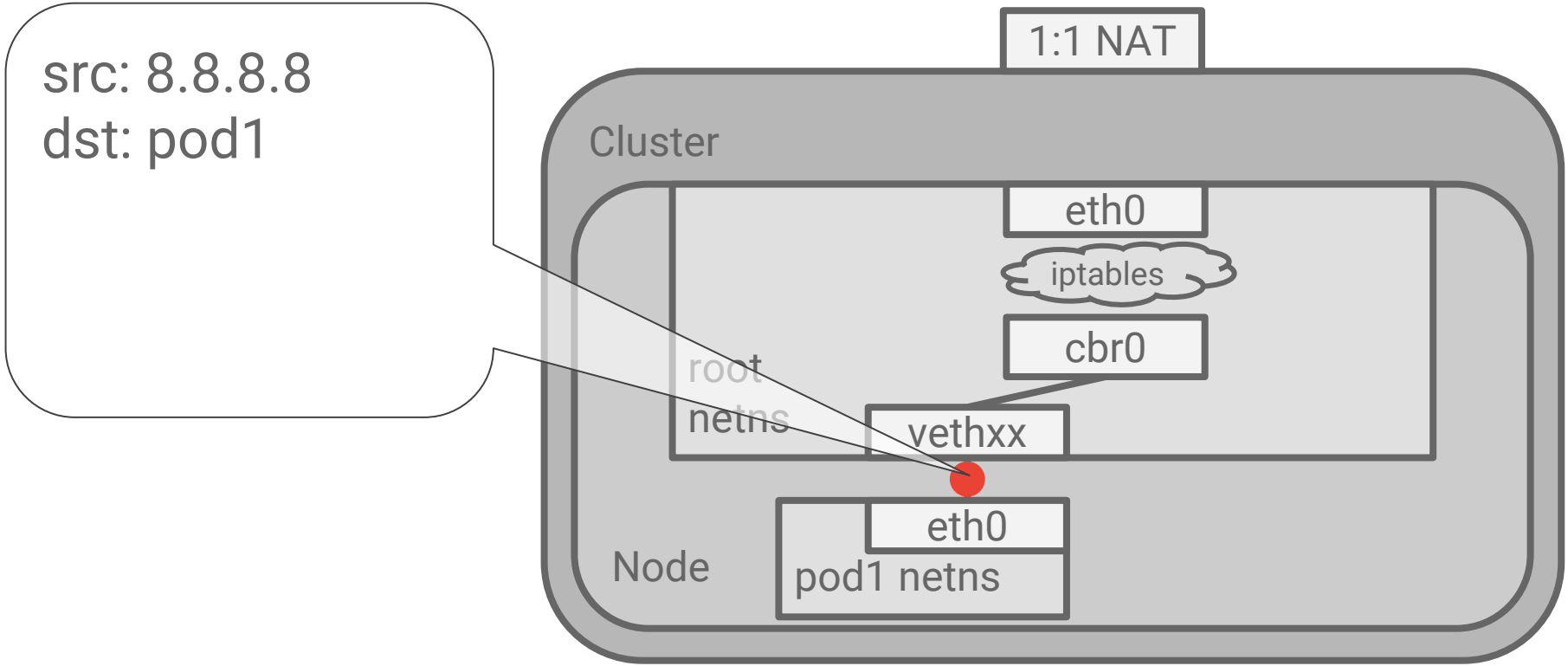
src: 8.8.8.8
~~dst: internal-IP~~
dst: pod1



Life of a packet: pod-to-internet



Life of a packet: pod-to-internet



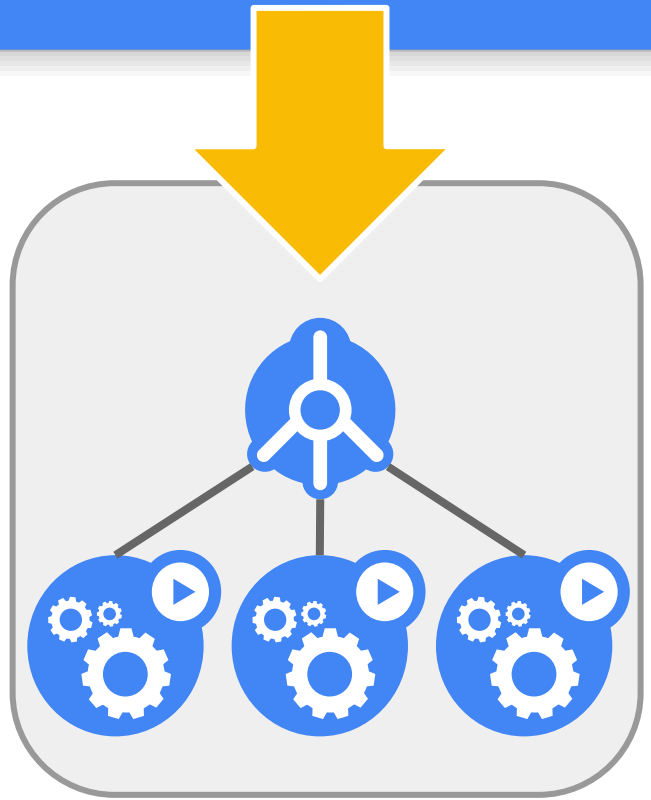
Receiving external traffic

Kubernetes builds on two:

- Network Load Balancer (L4)
- HTTP/S Load balancer (L7)

These map to Kubernetes APIs:

- `Service type=LoadBalancer`
- Ingress



L4: Service + LoadBalancer

Service

Change the type of your service

Implemented by the cloud provider controller

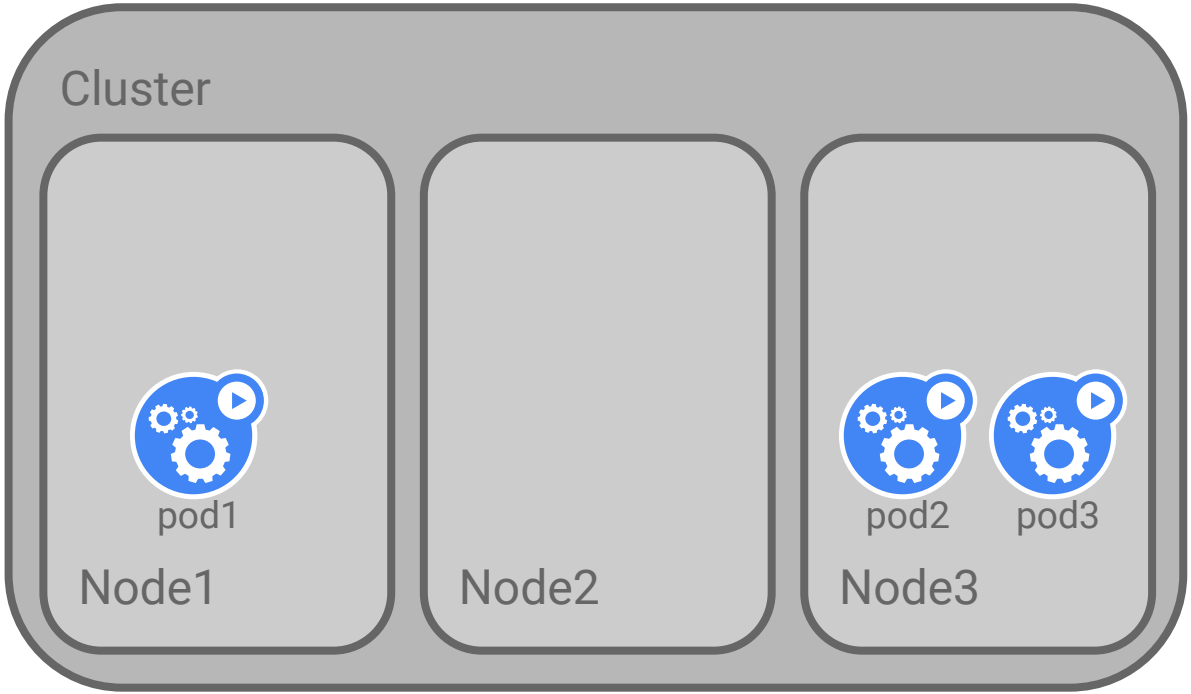
```
kind: Service
apiVersion: v1
metadata:
  name: store-be
spec:
  type: LoadBalancer
  selector:
     app: store
     role: be
  ports:
  - name: https
    port: 443
```

Service

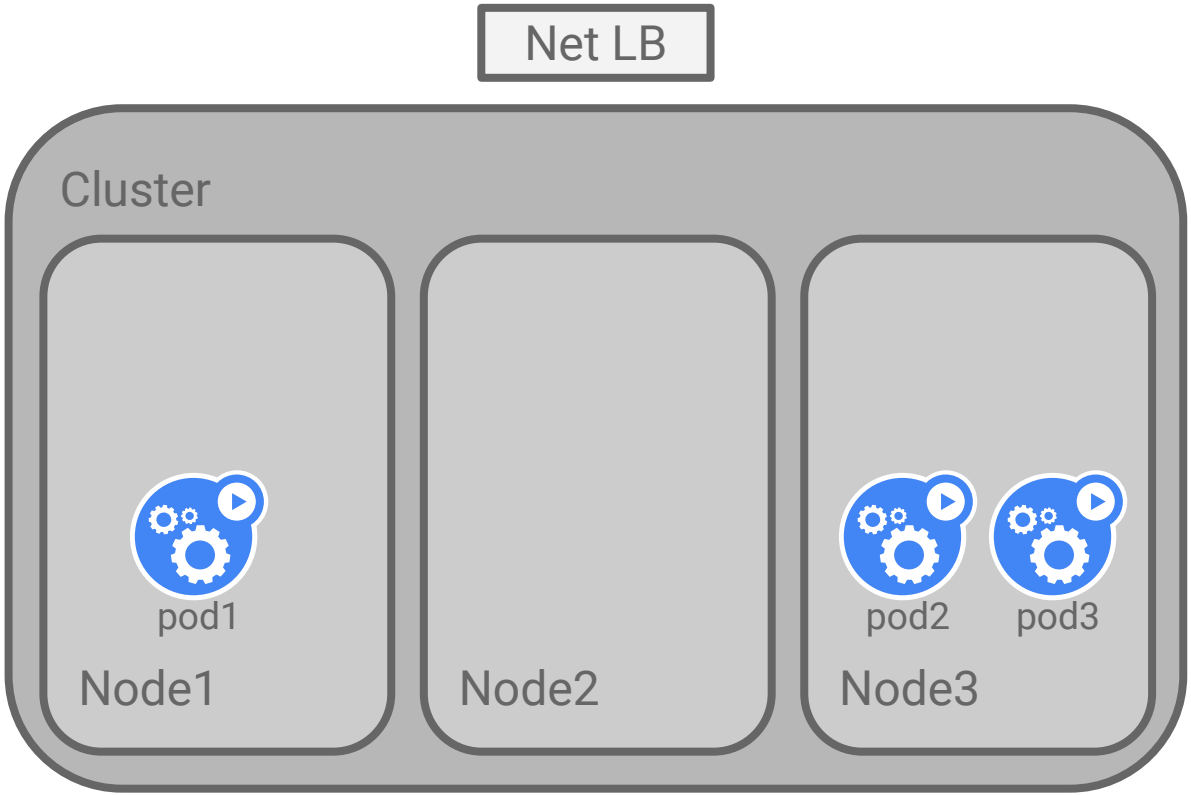
The LB info is populated when ready

```
kind: Service
apiVersion: v1
metadata:
  name: store-be
  # ...
spec:
  type: LoadBalancer
  selector:
    app: store
    role: be
  clusterIP: 10.9.3.76
  ports:
    # ...
  sessionAffinity: None
 status:
    loadBalancer:
      ingress:
      - ip: 86.75.30.9
```

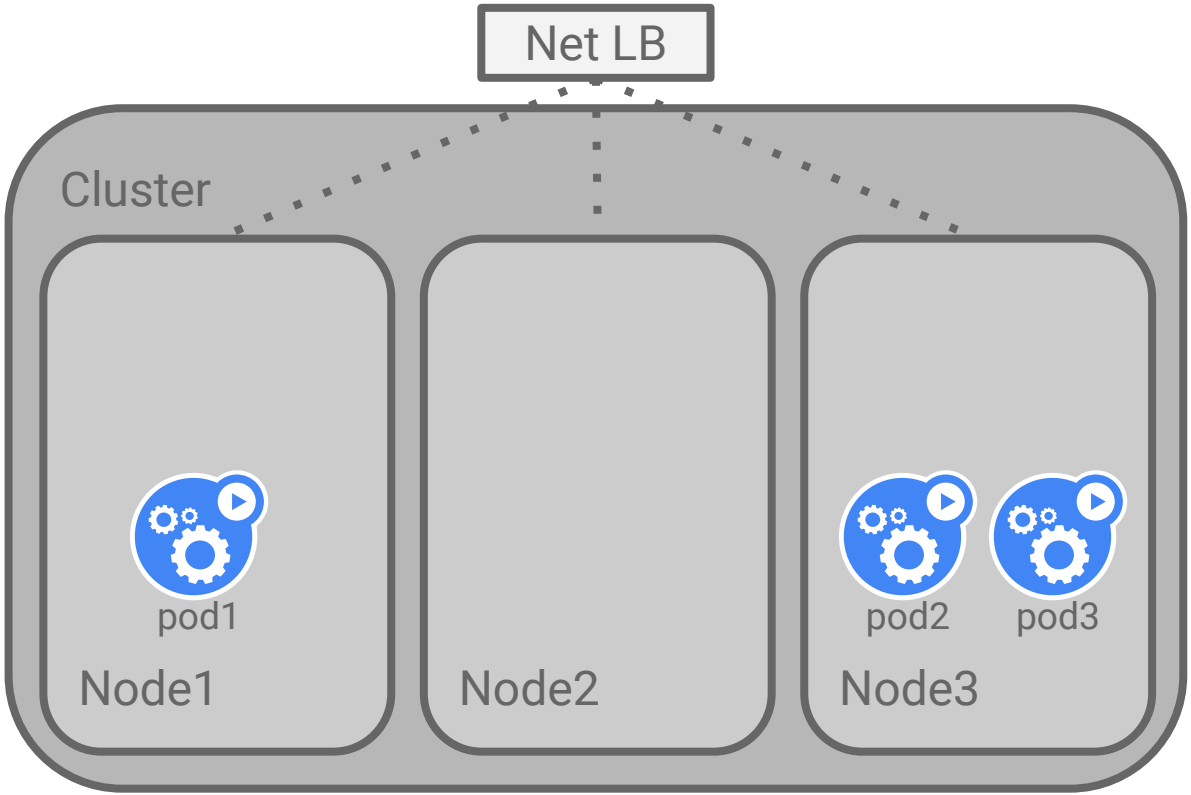
Life of a packet: external-to-service



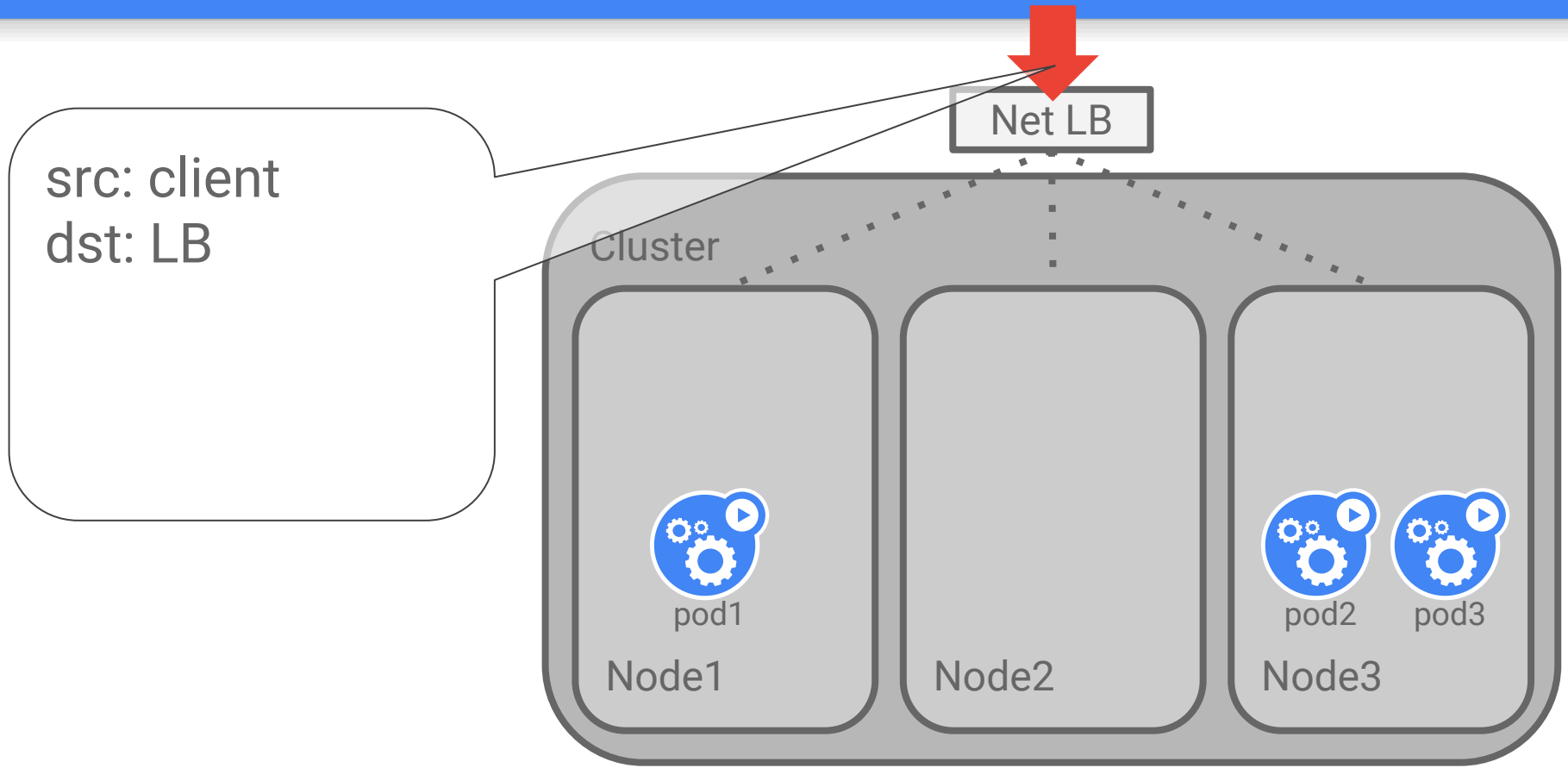
Life of a packet: external-to-service



Life of a packet: external-to-service



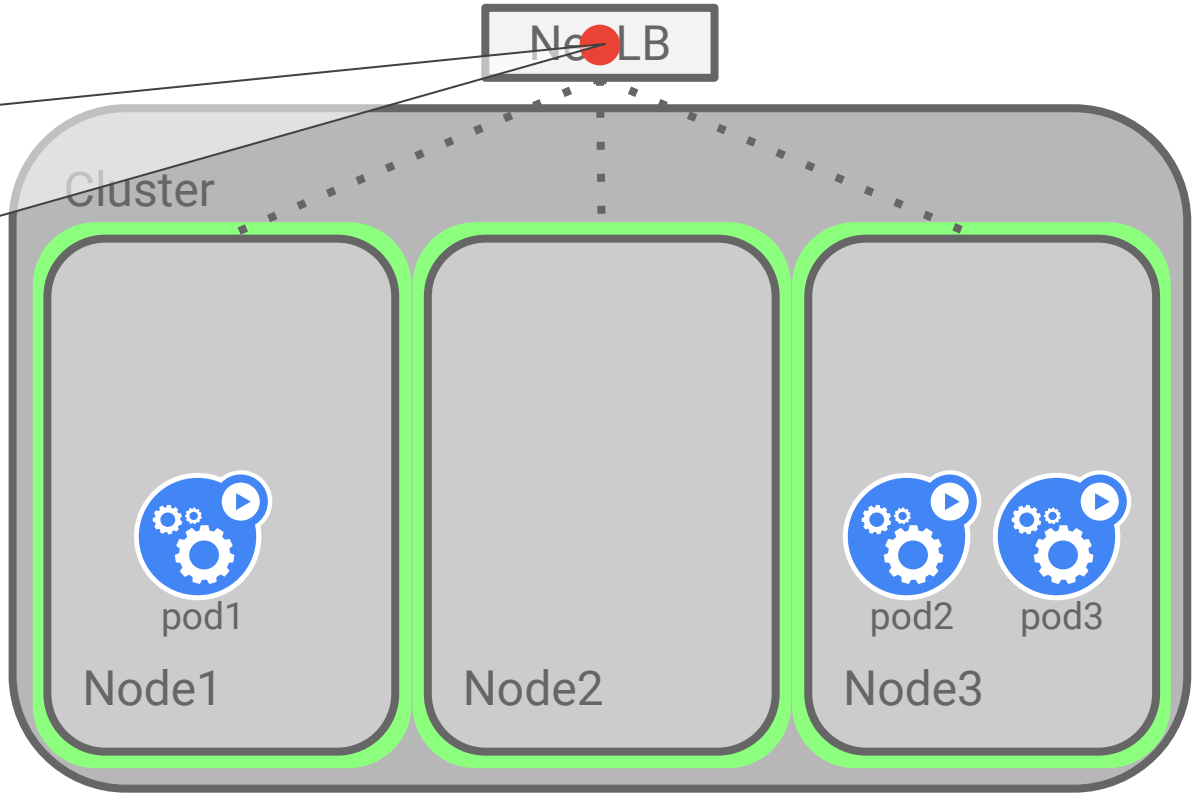
Life of a packet: external-to-service



Life of a packet: external-to-service

src: client
dst: LB

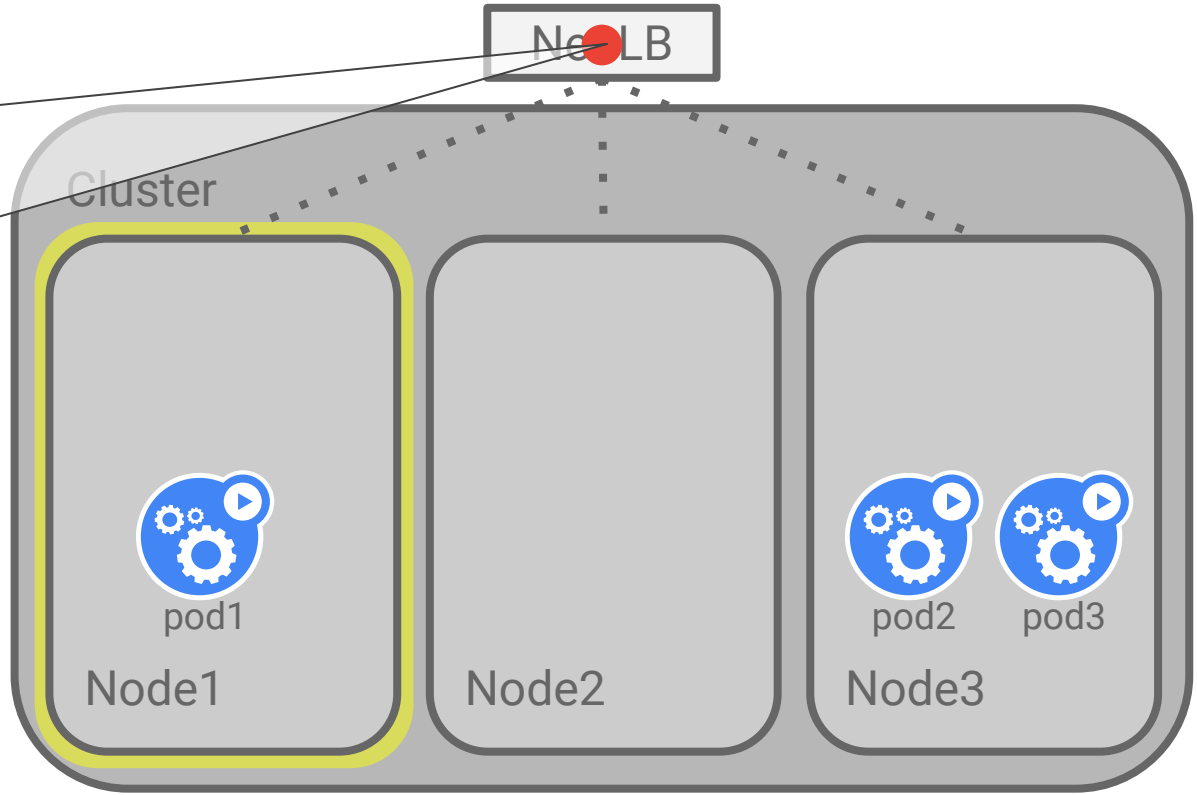
Choose a Node



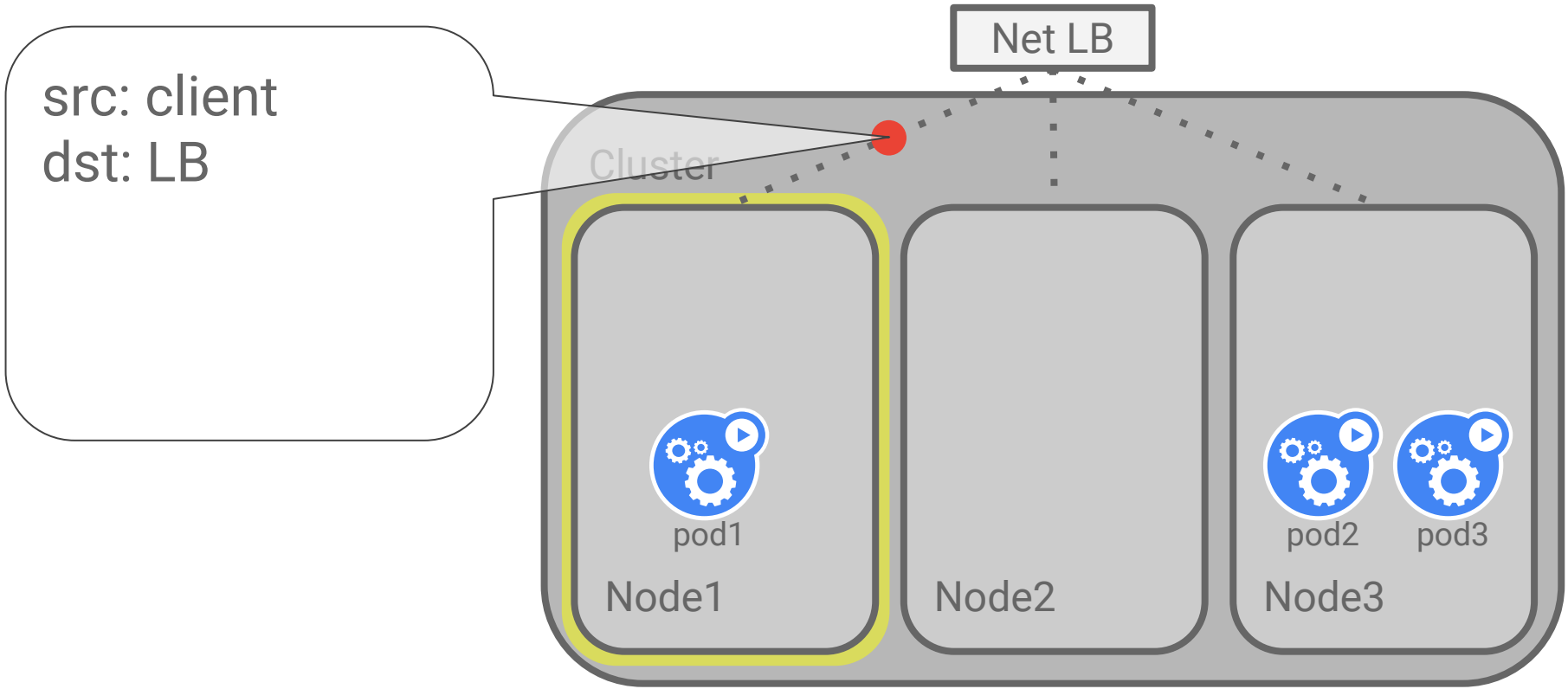
Life of a packet: external-to-service

src: client
dst: LB

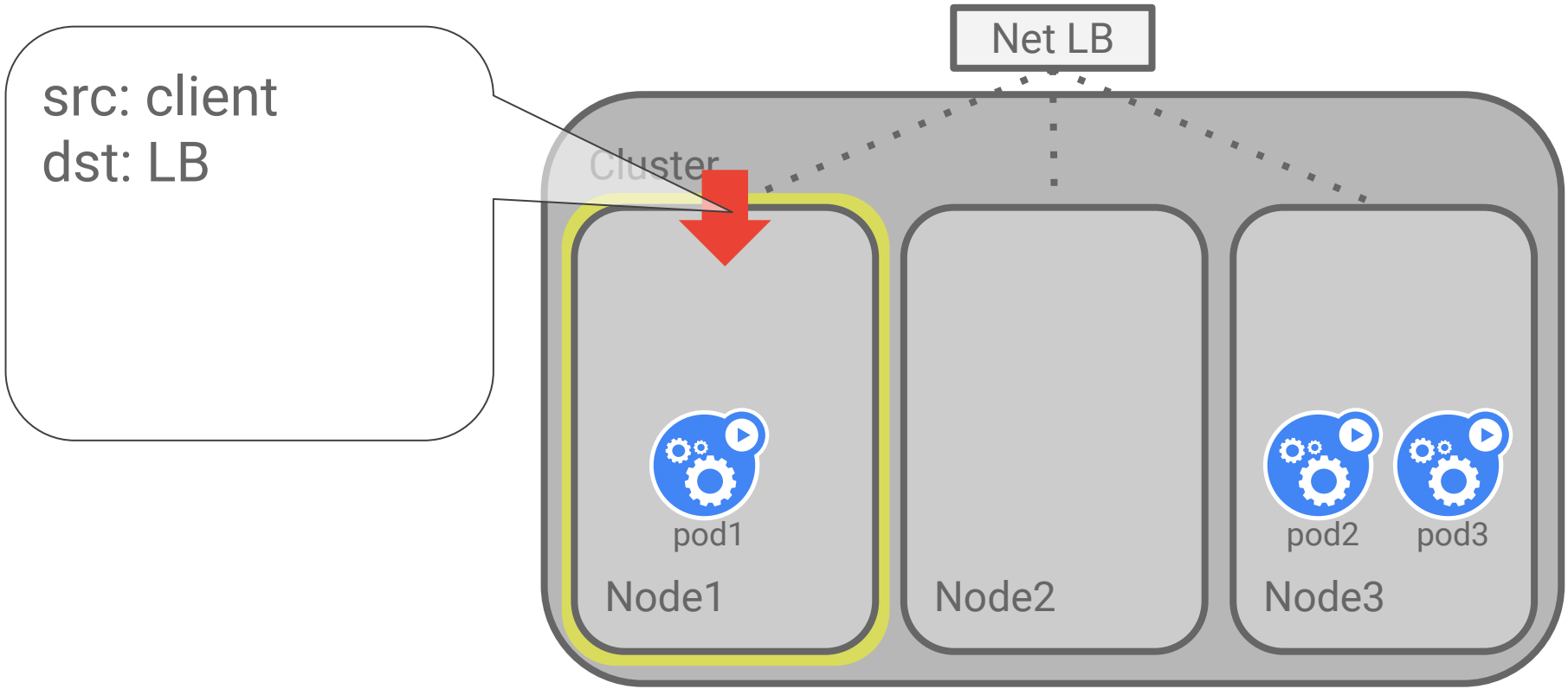
Choose a Node



Life of a packet: external-to-service



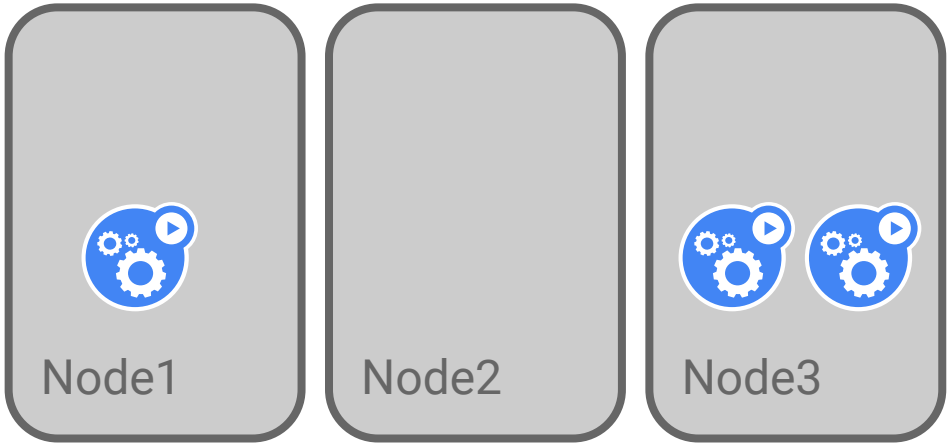
Life of a packet: external-to-service



Balancing to Nodes

Most LB only knows about Nodes

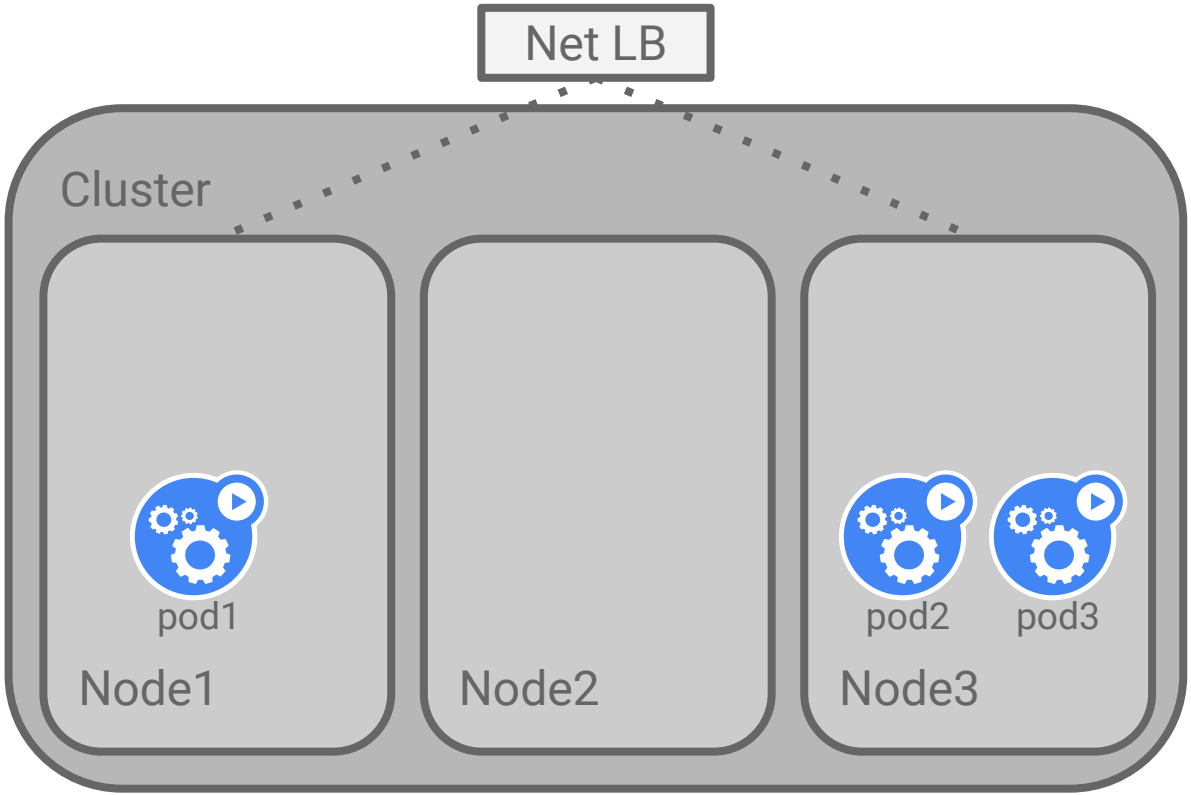
Nodes do not map 1:1 with pods



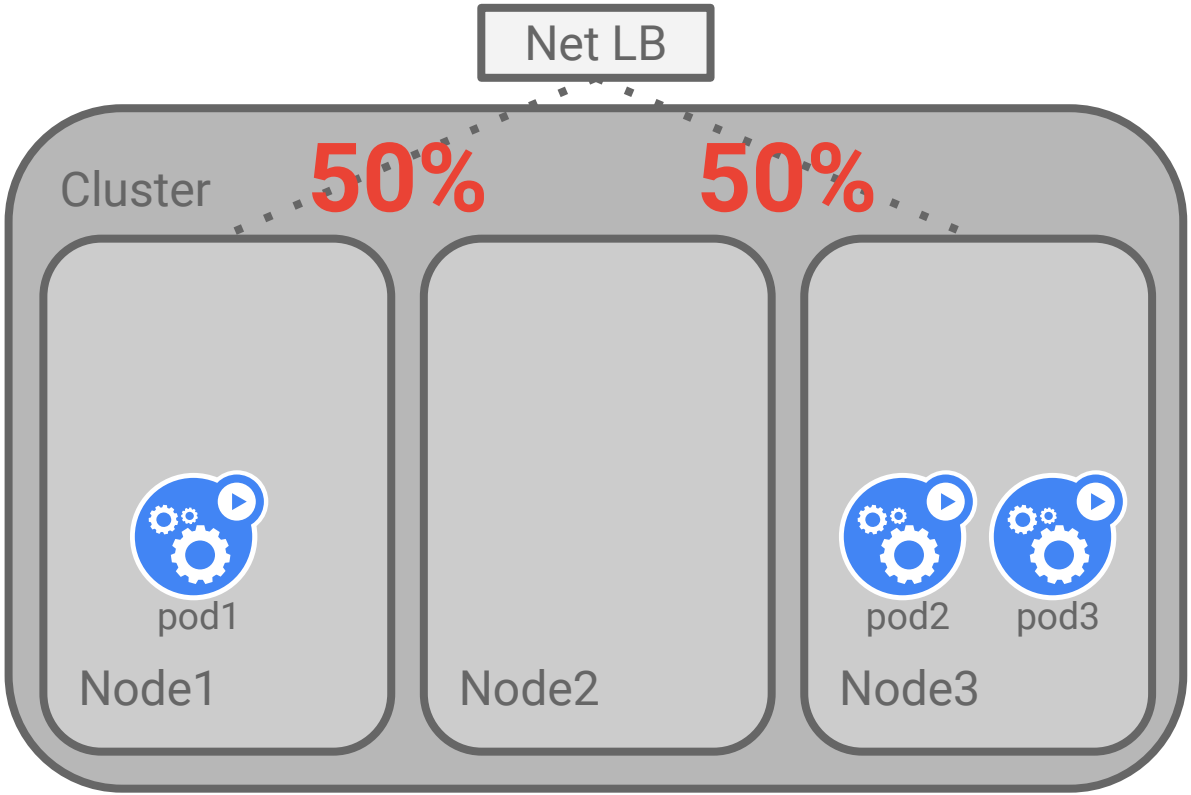
The imbalance problem

Assume the LB only hits Nodes with backend pods on them

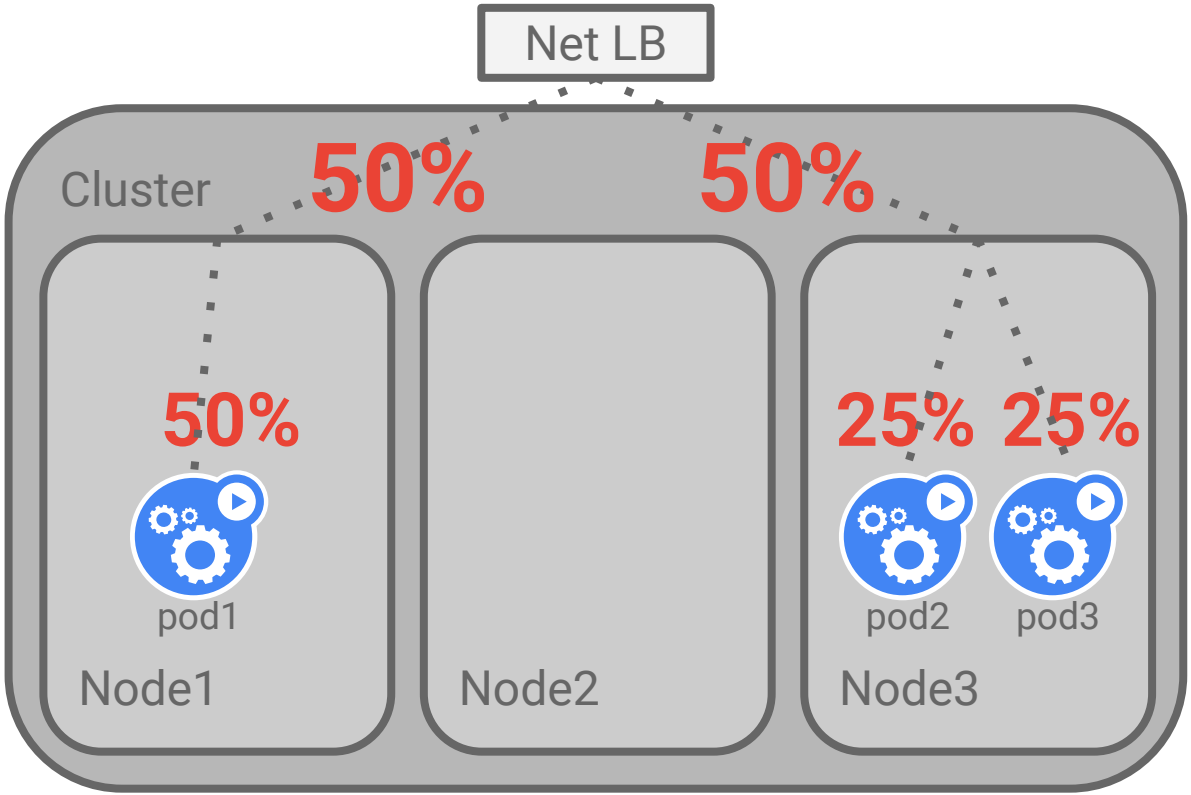
The LB only knows about Nodes



The imbalance problem



The imbalance problem



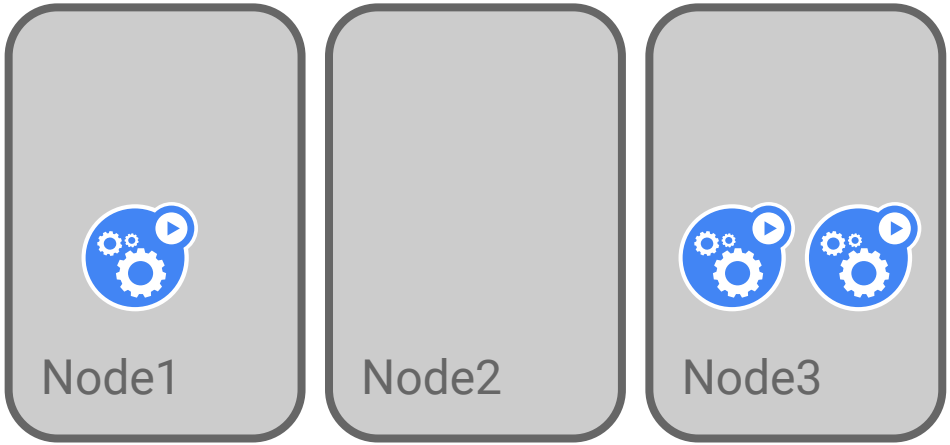
Balancing to Nodes

Most cloud LB only knows about Nodes

Nodes do not map 1:1 with pods

How do we avoid imbalance?

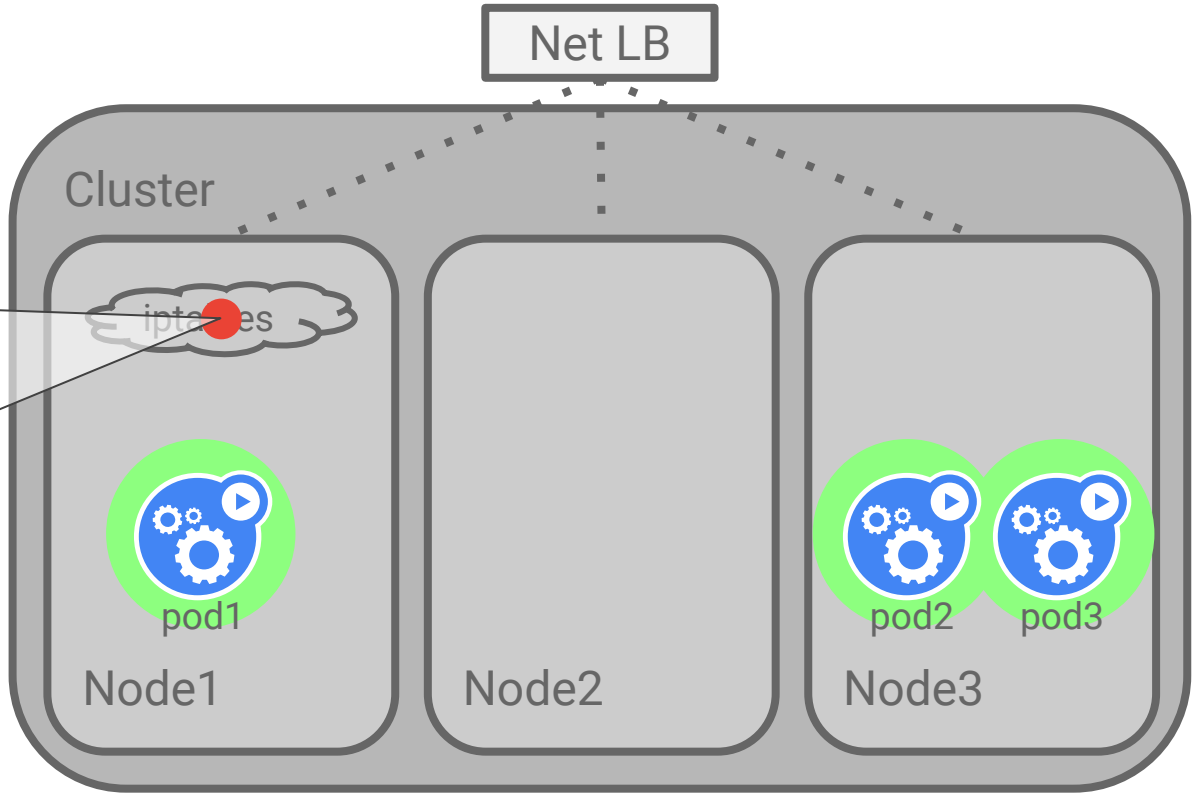
iptables, of course



Life of a packet: external-to-service

src: client
dst: LB

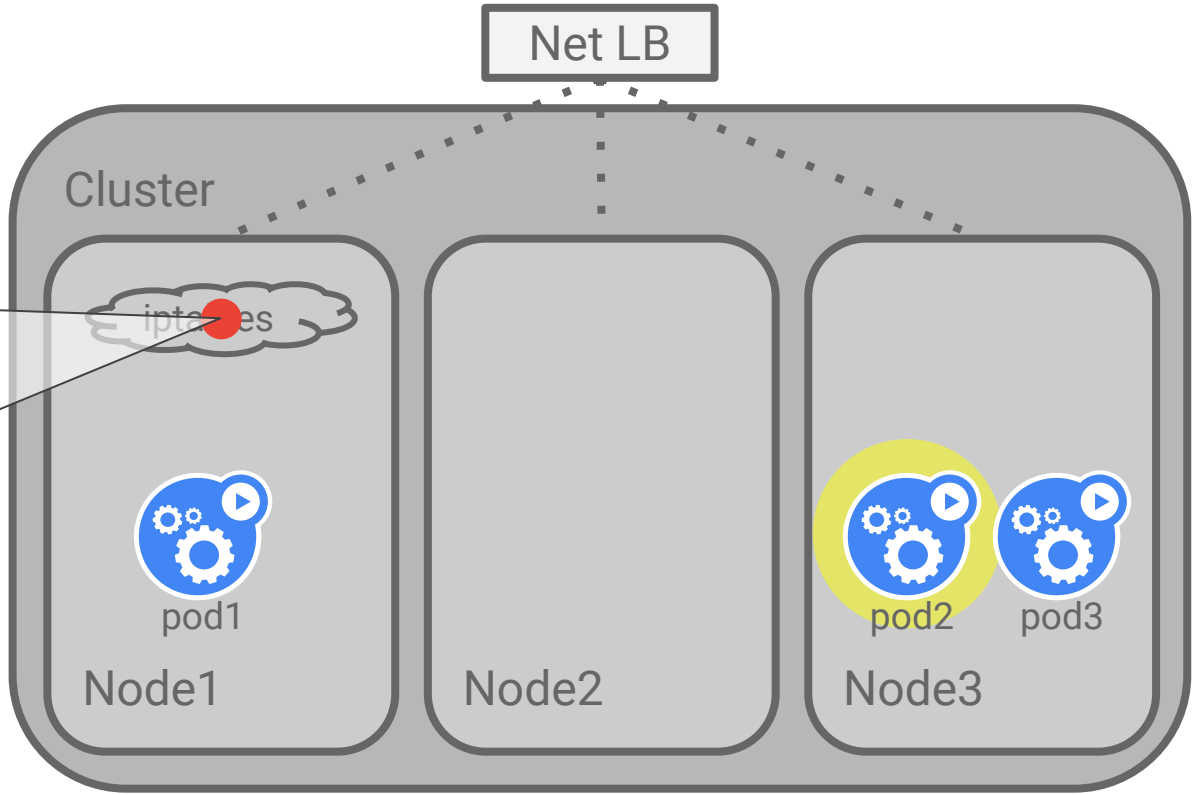
Choose a pod



Life of a packet: external-to-service

src: client
dst: LB

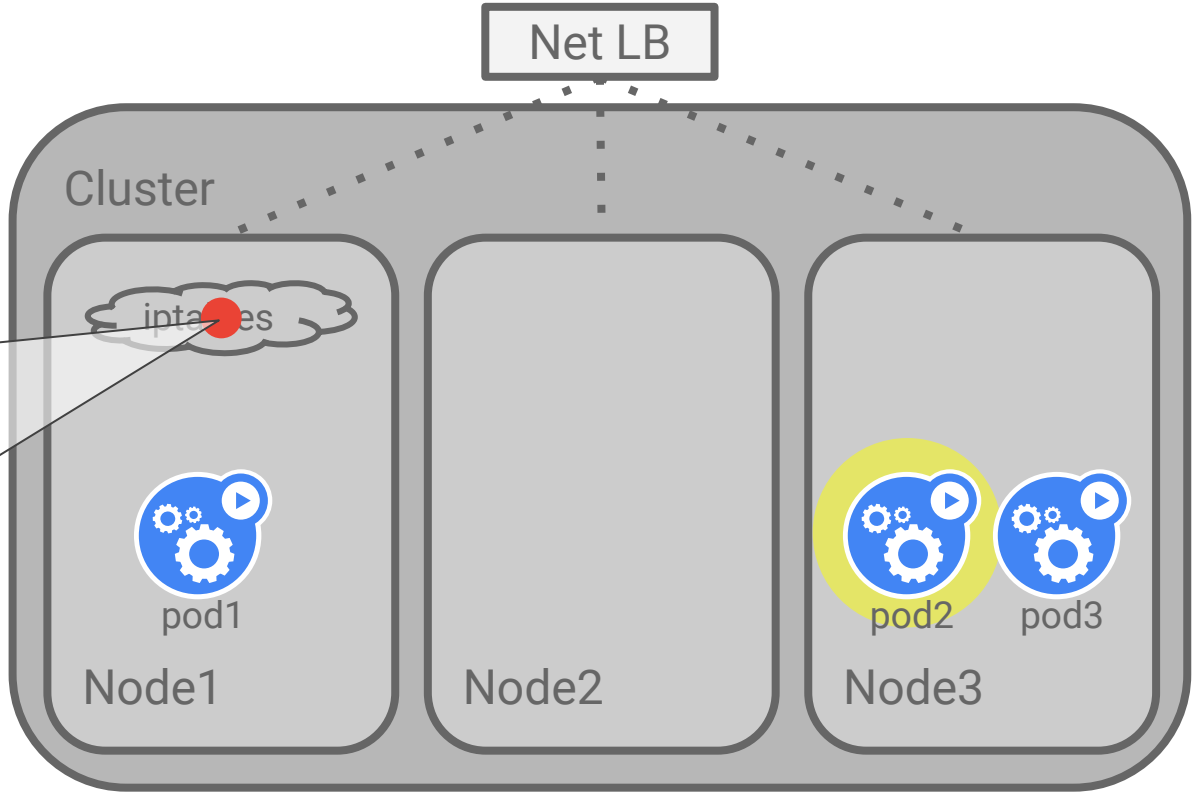
Choose a pod



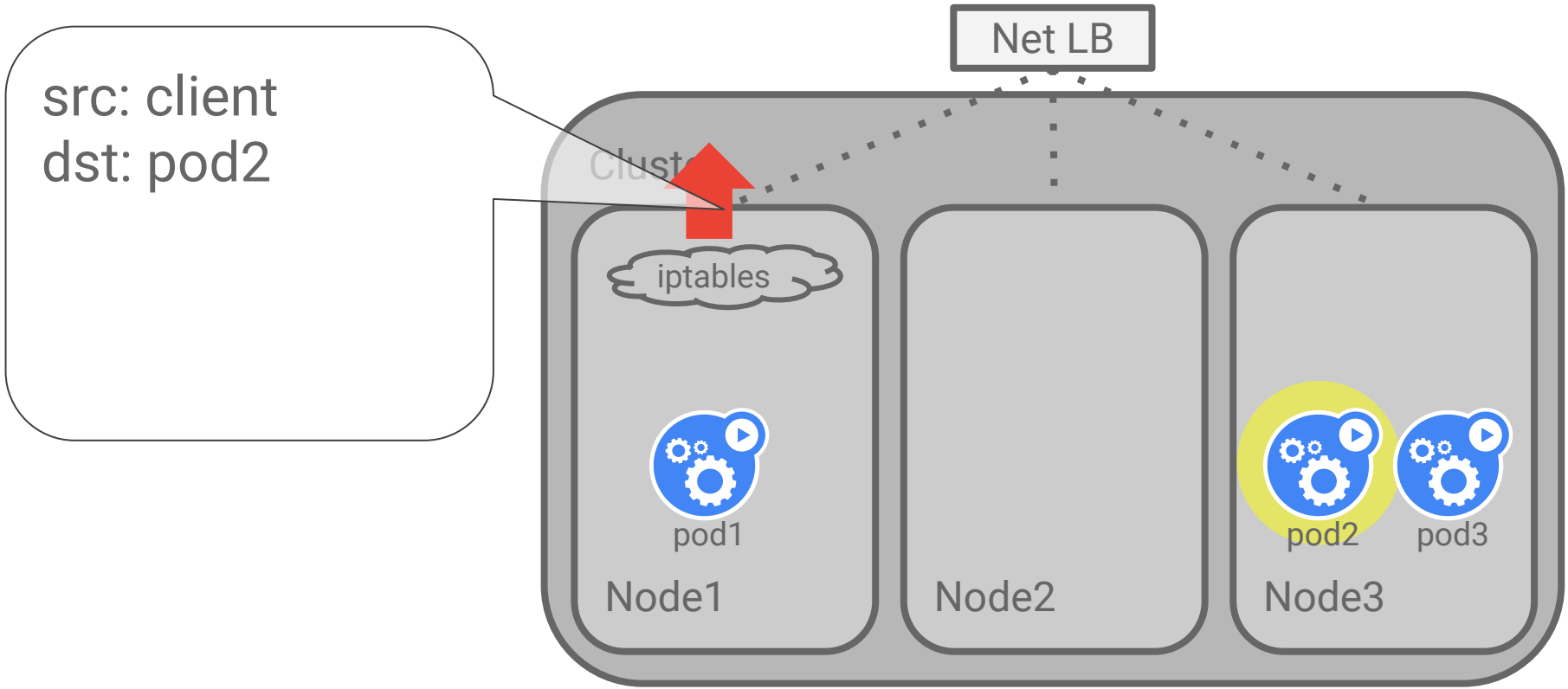
Life of a packet: external-to-service

src: client
~~dst: LB~~
dst: pod2

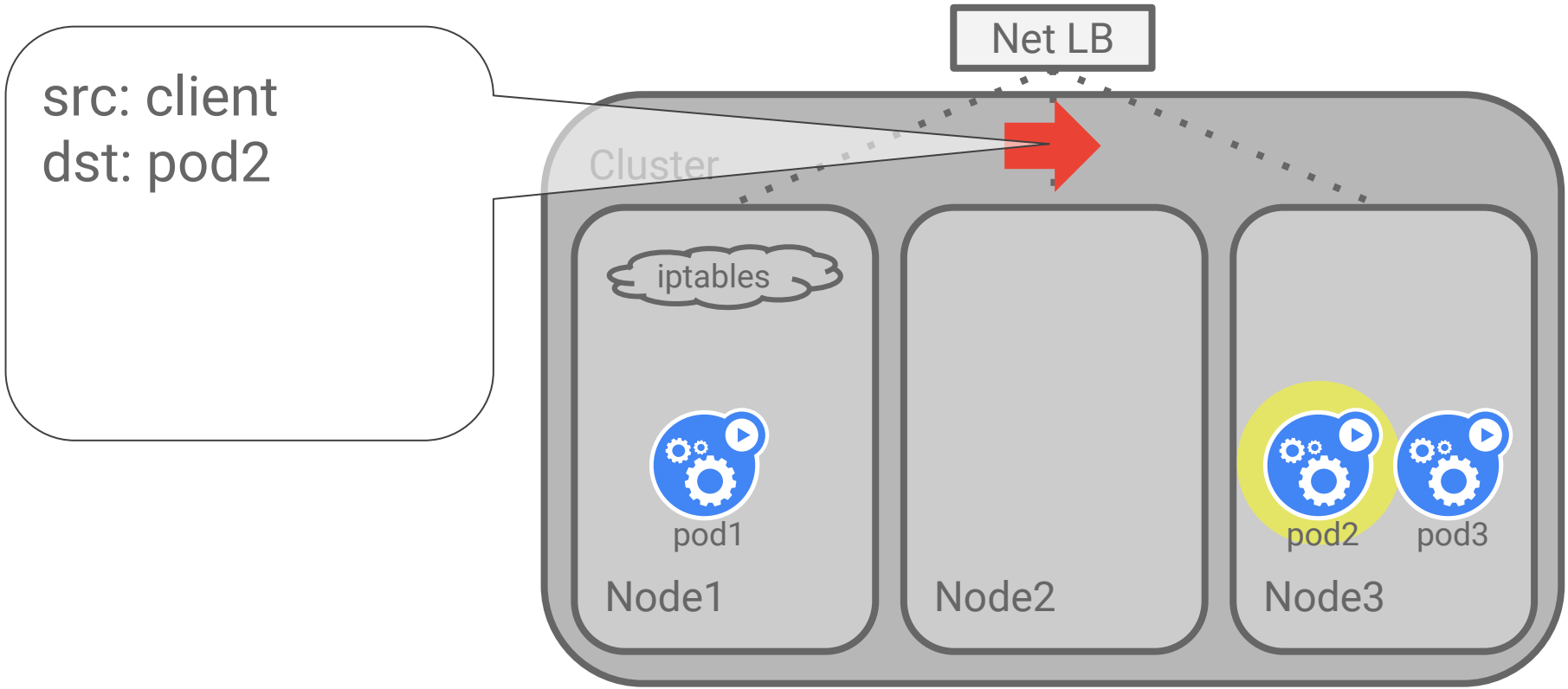
NAT



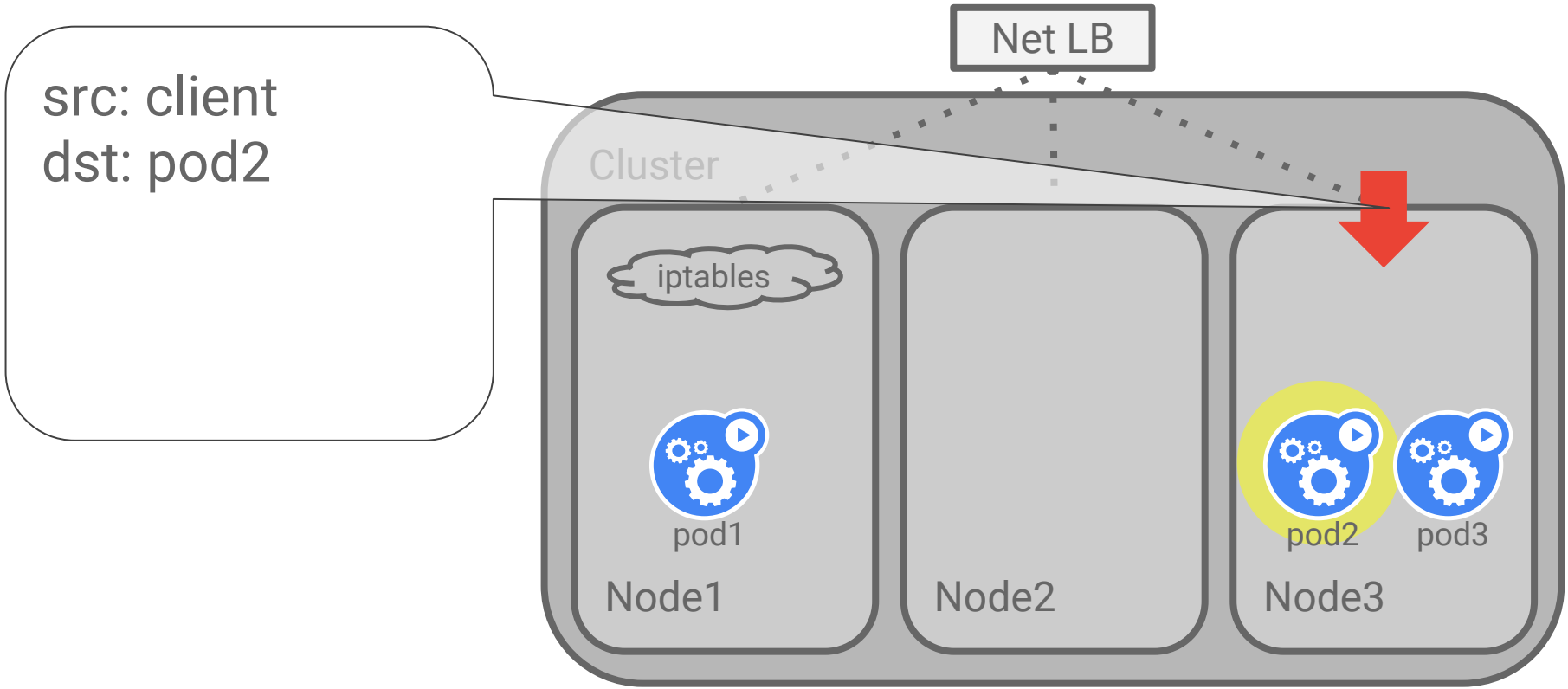
Life of a packet: external-to-service



Life of a packet: external-to-service

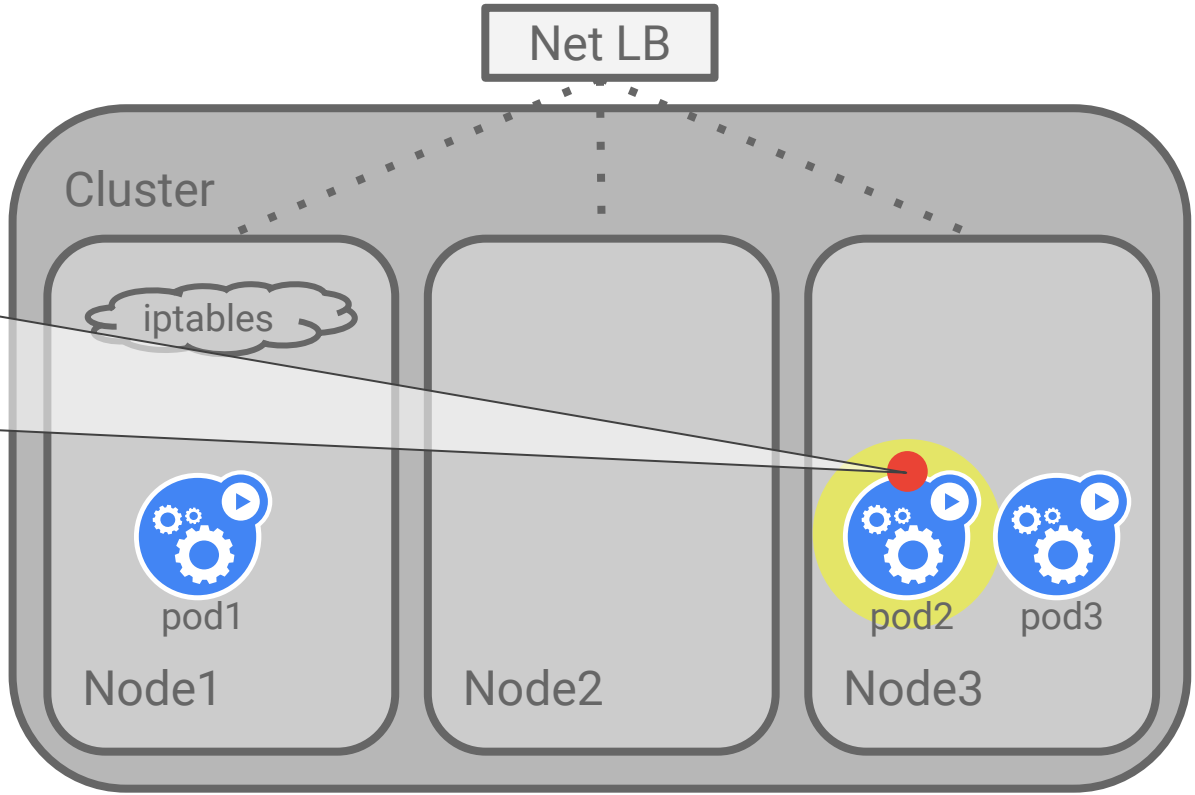


Life of a packet: external-to-service

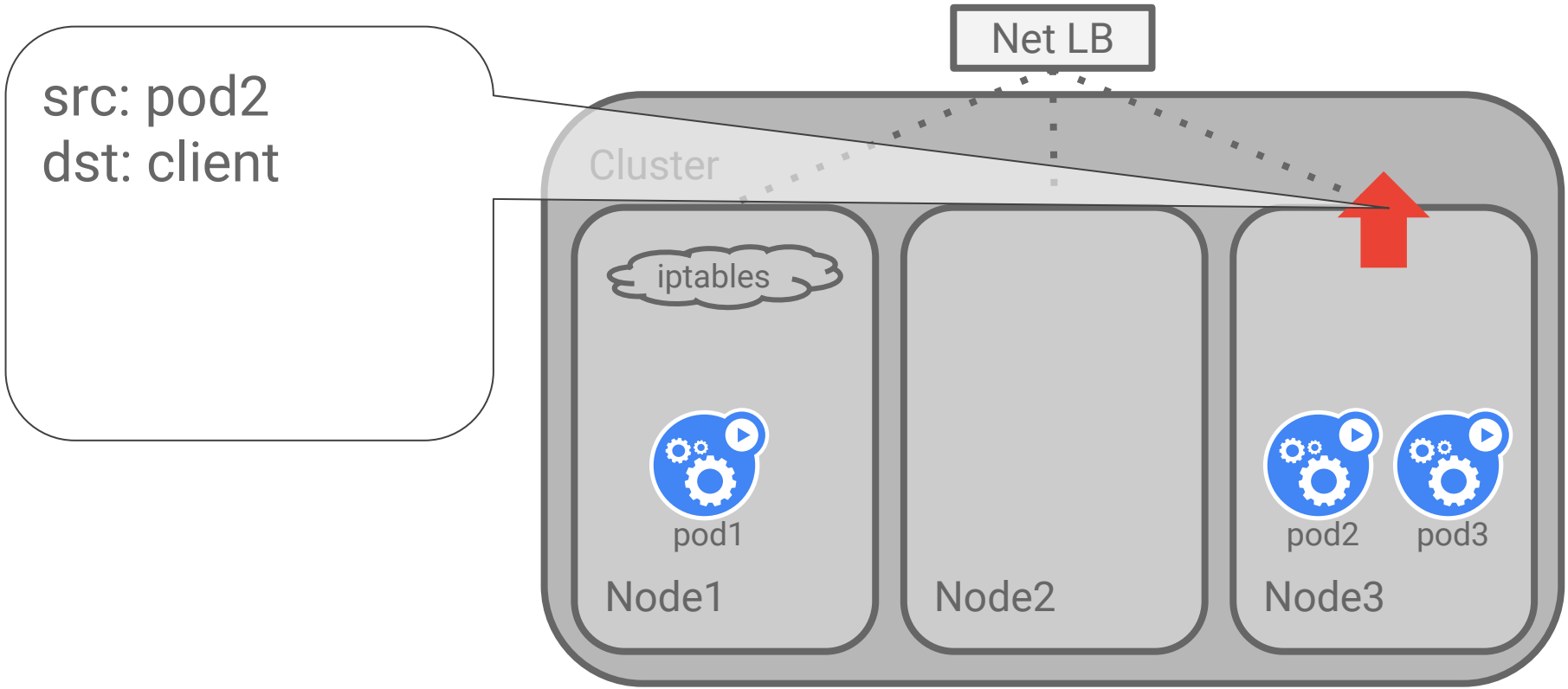


Life of a packet: external-to-service

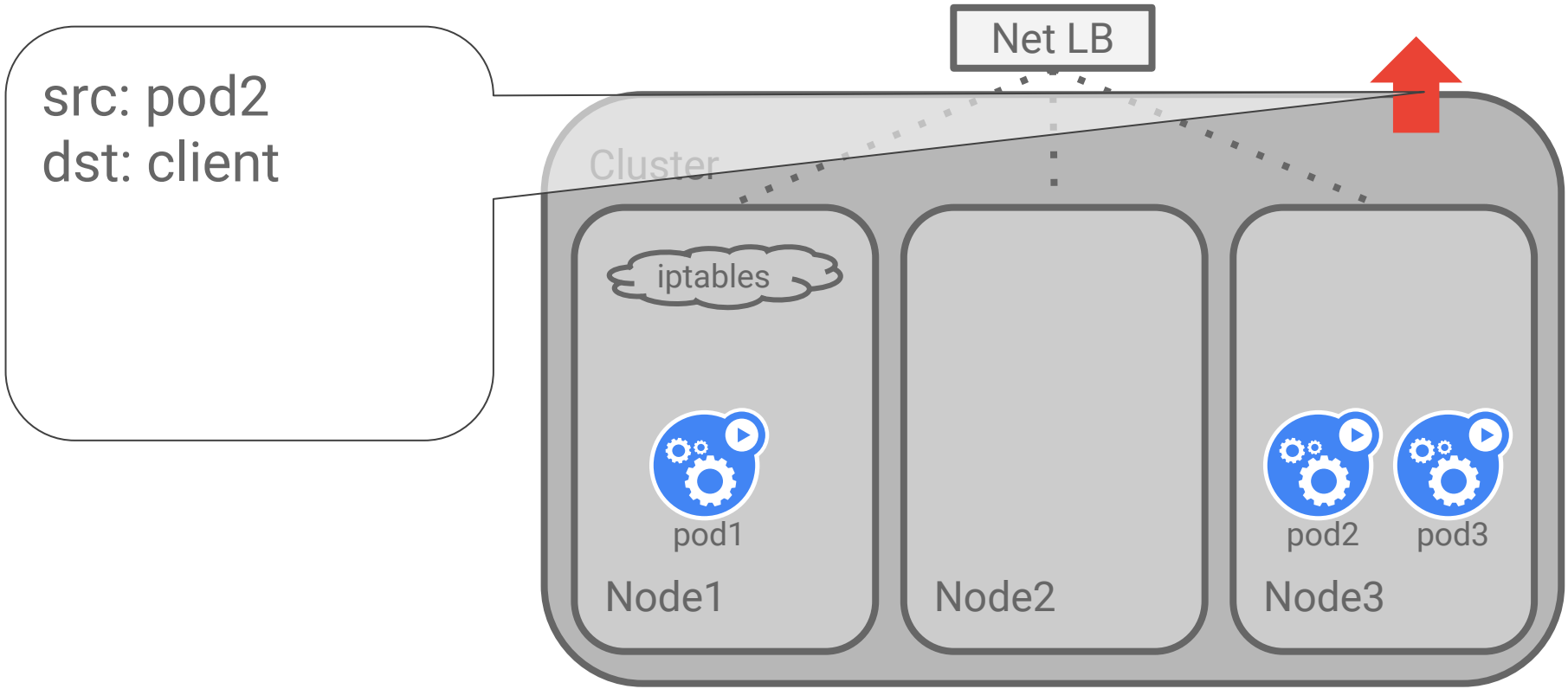
src: client
dst: pod2



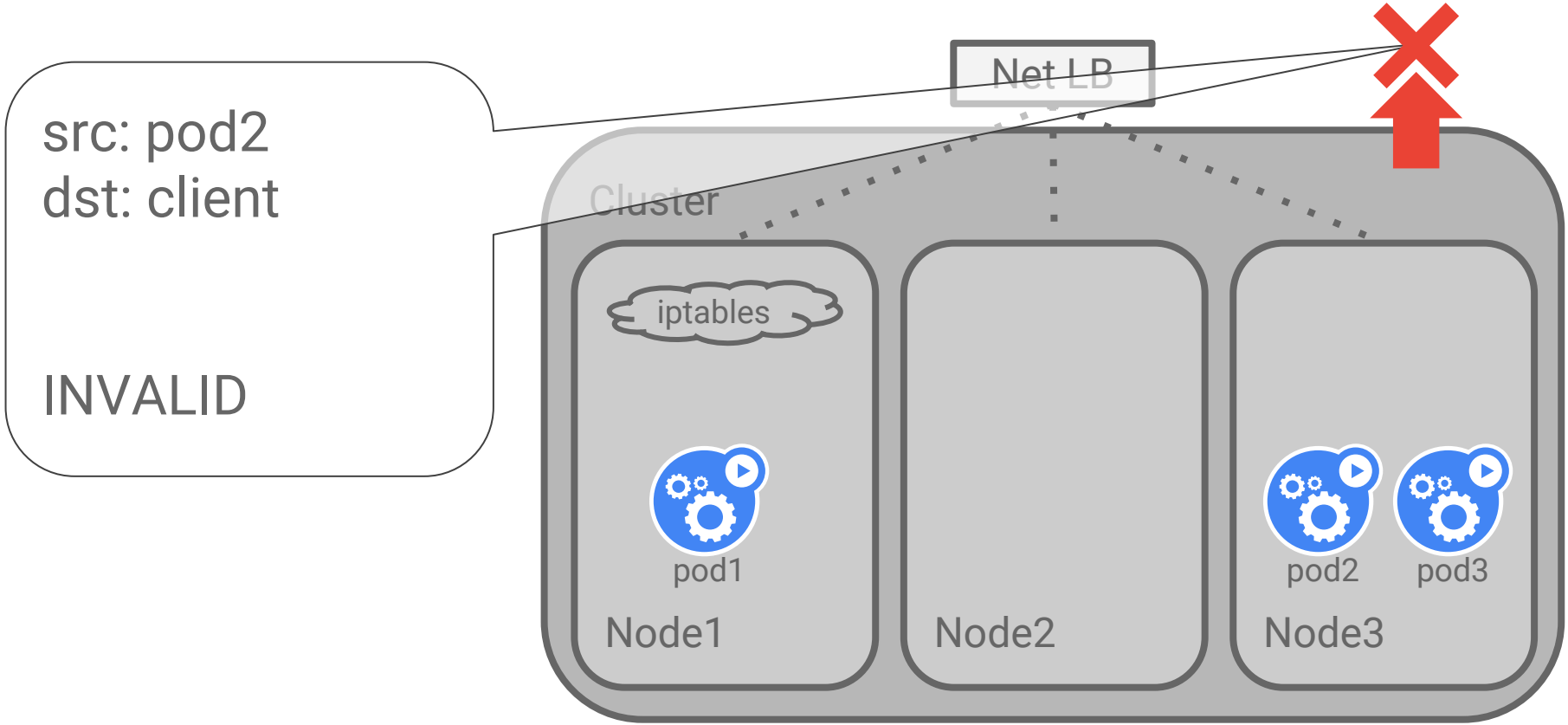
Life of a packet: external-to-service



Life of a packet: external-to-service



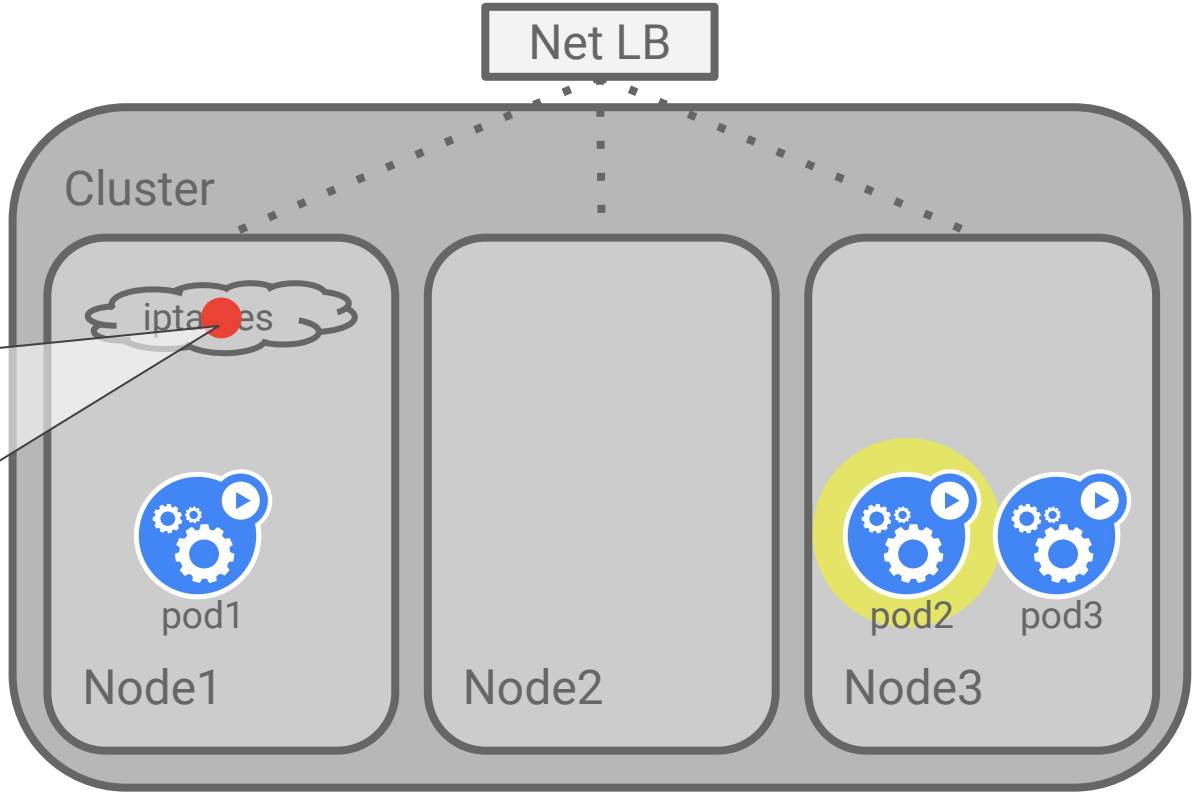
Life of a packet: external-to-service



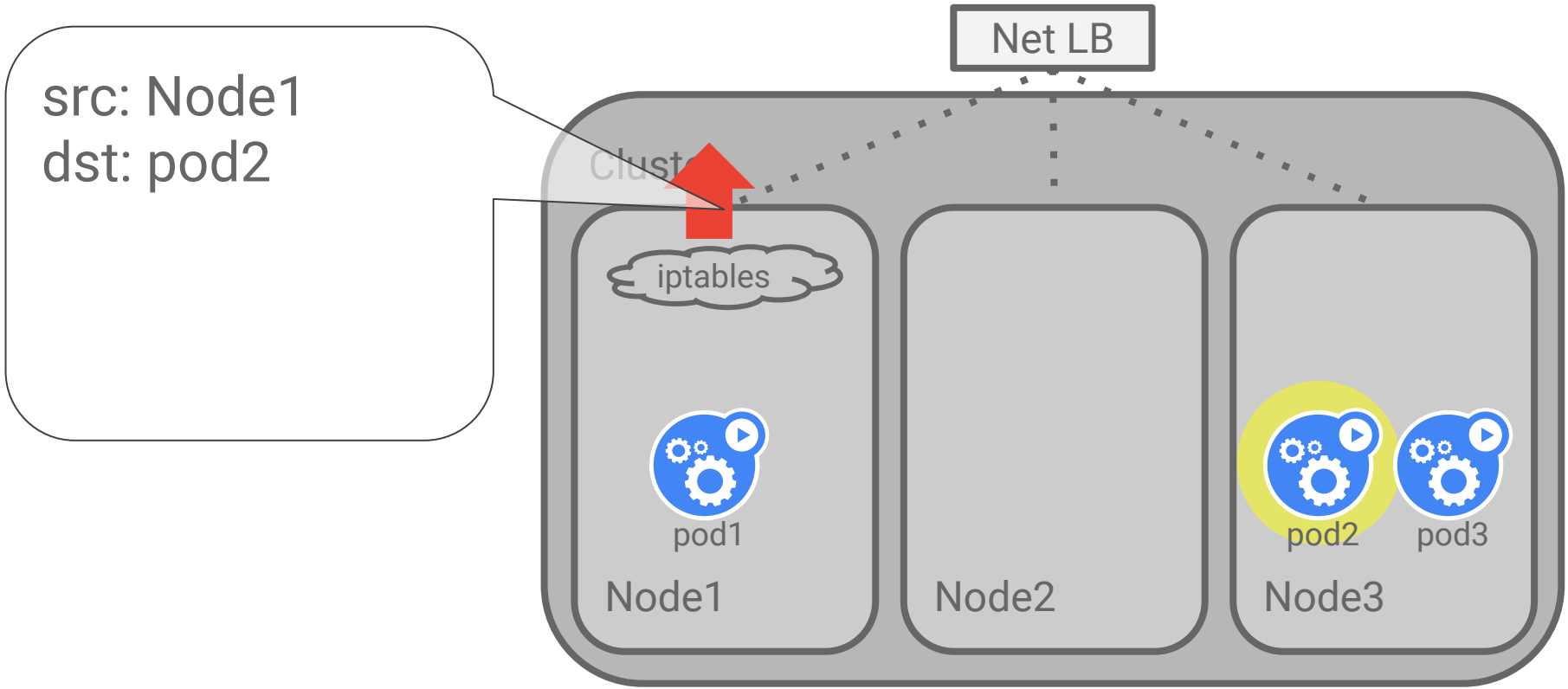
Life of a packet: external-to-service

~~src: client~~
src: Node1
~~dst: LB~~
dst: pod2

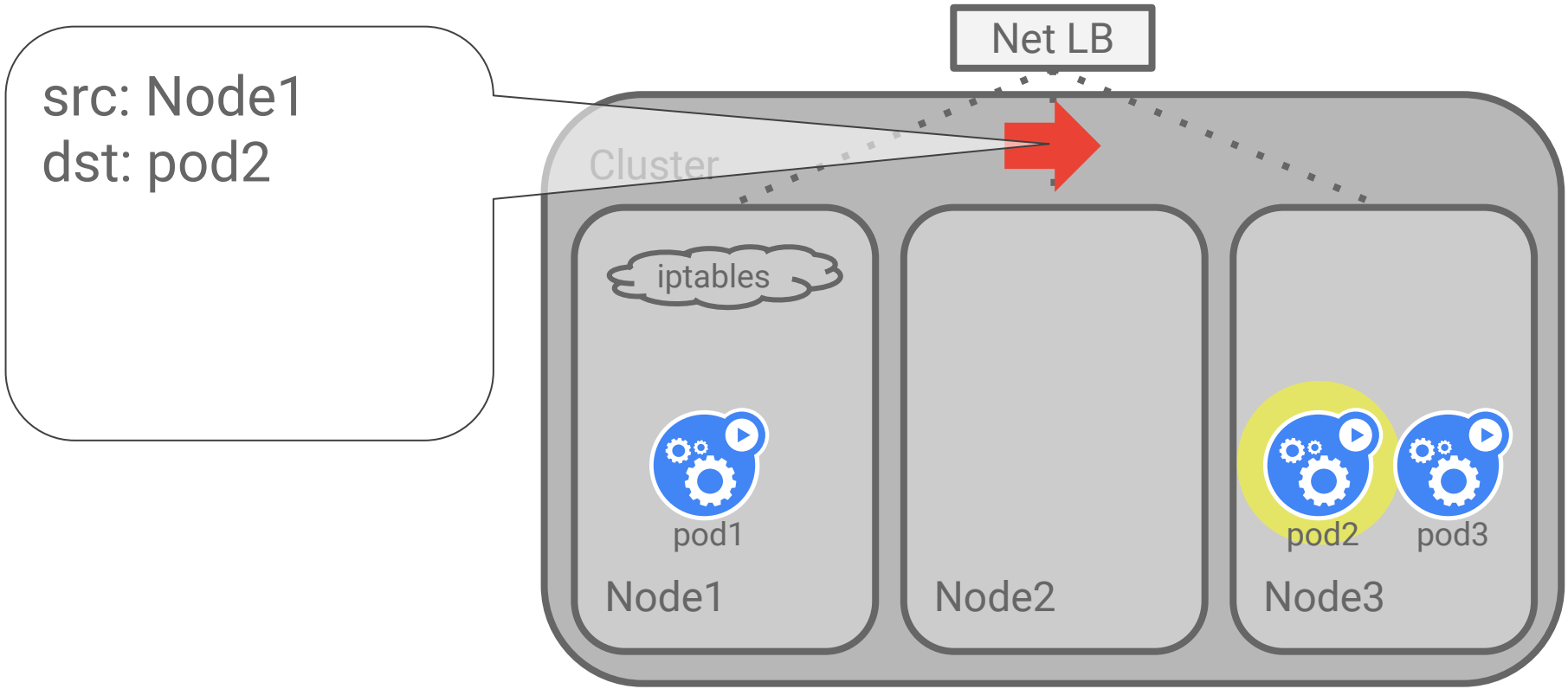
NAT



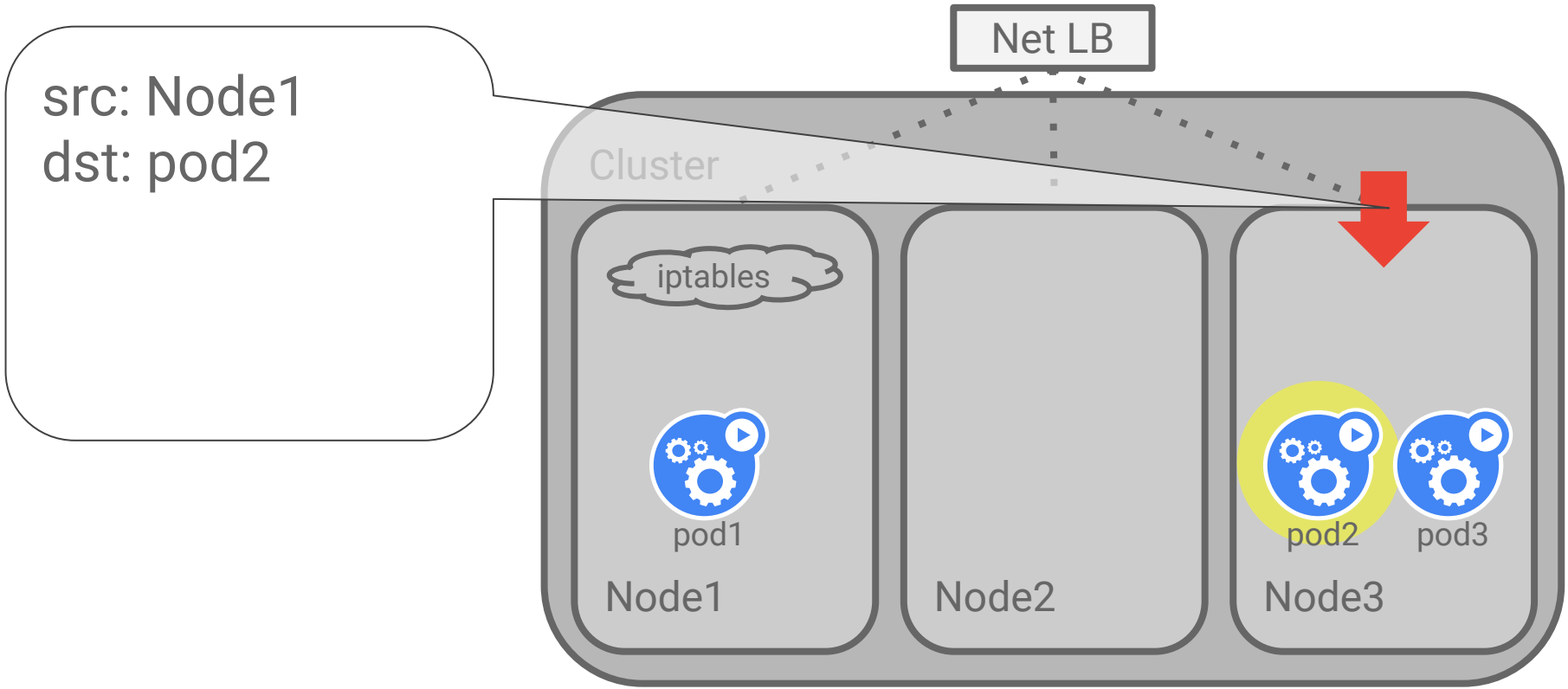
Life of a packet: external-to-service



Life of a packet: external-to-service

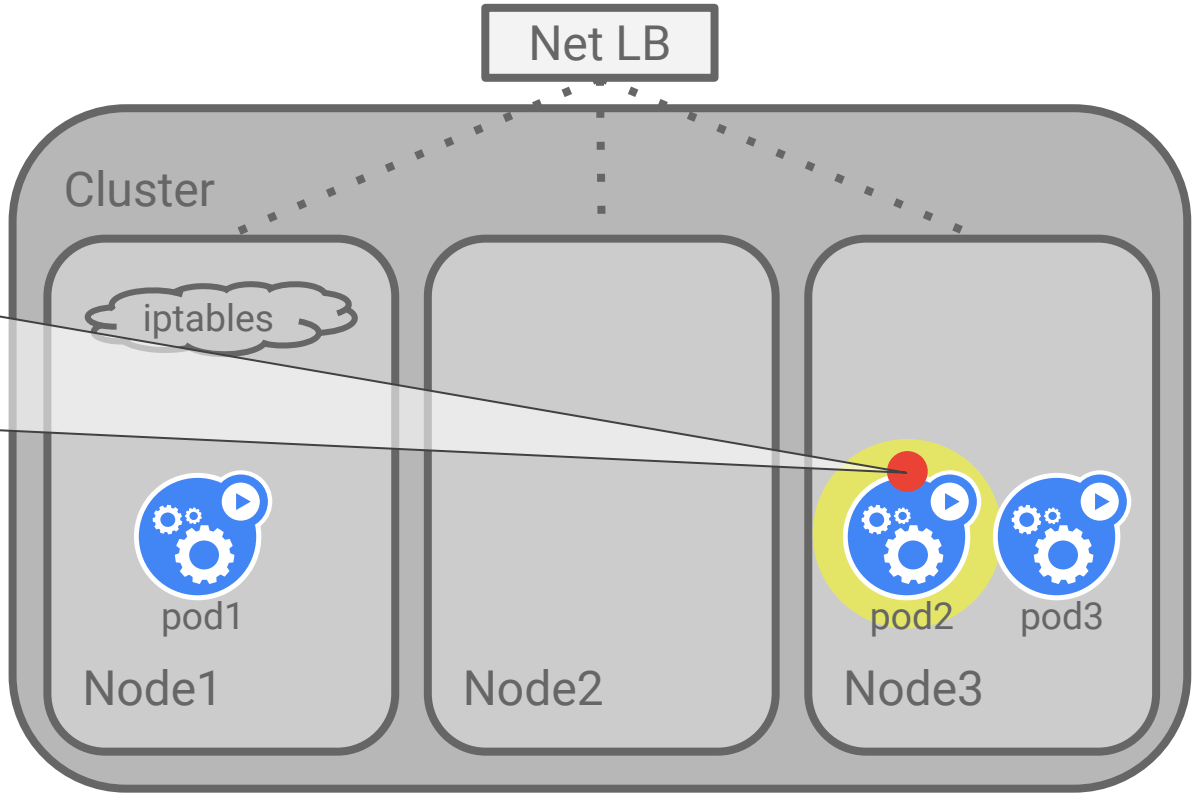


Life of a packet: external-to-service

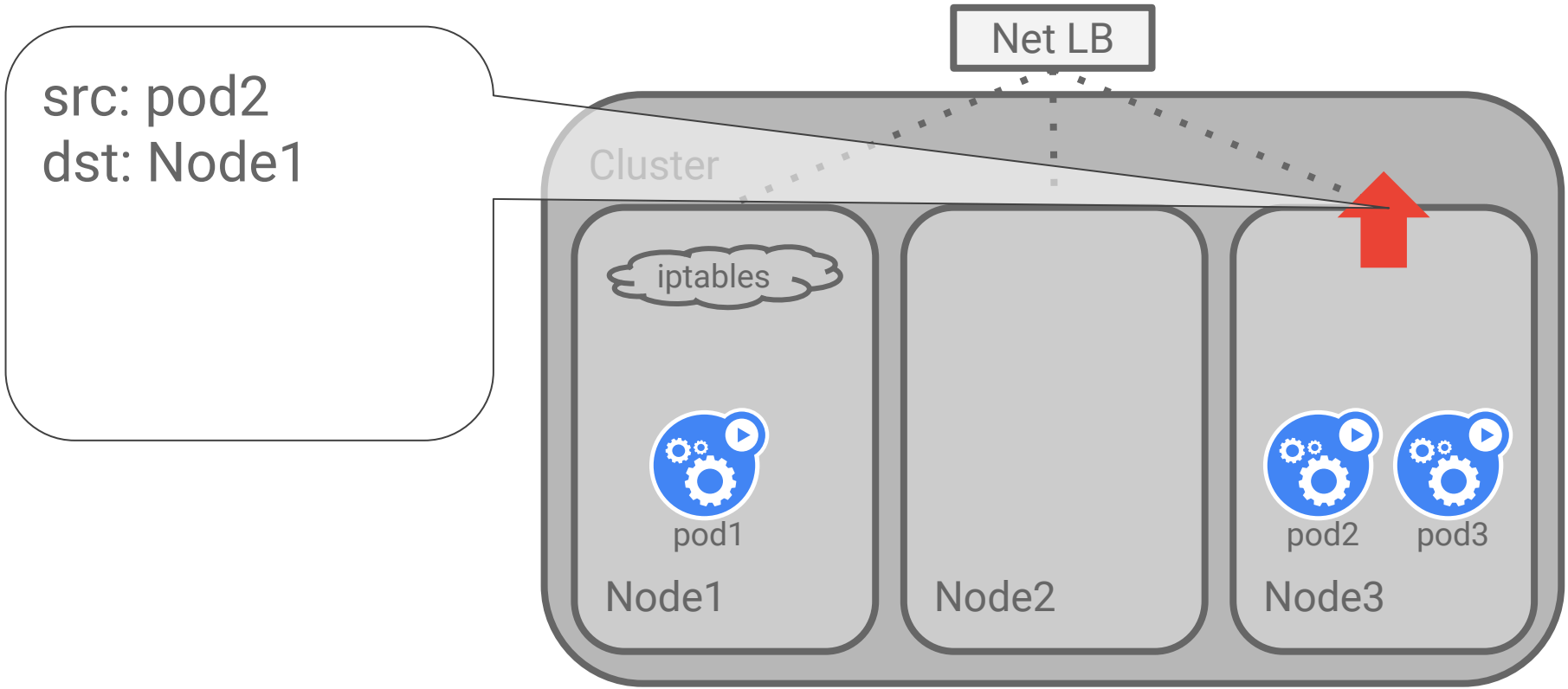


Life of a packet: external-to-service

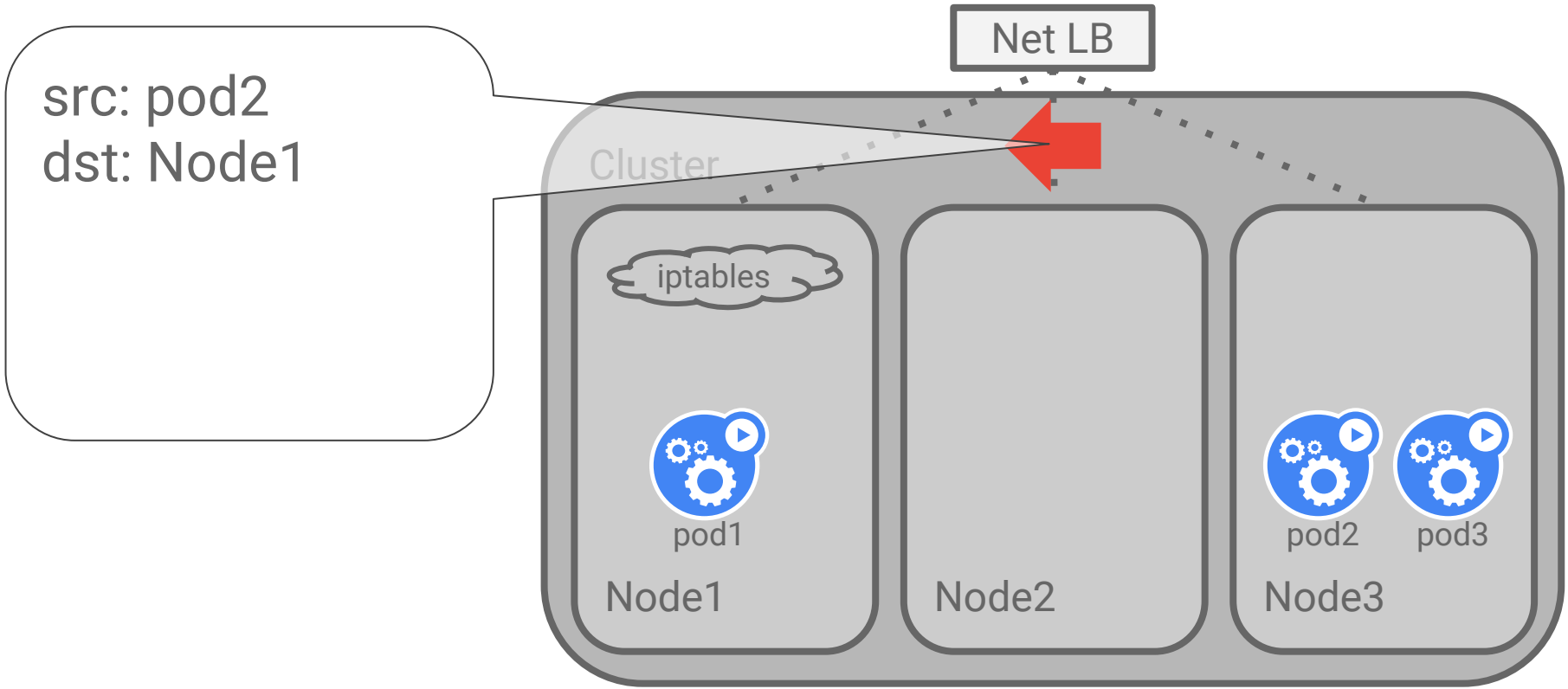
src: Node1
dst: pod2



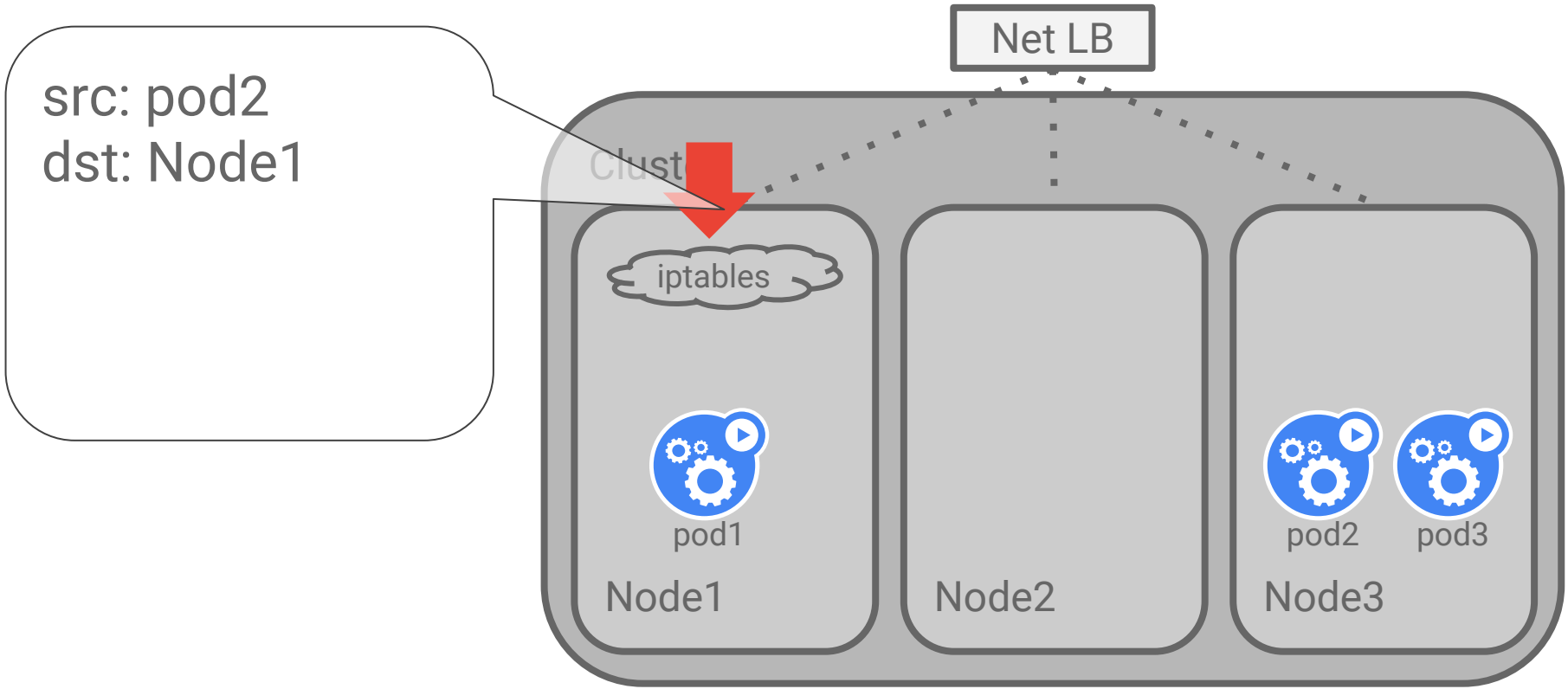
Life of a packet: external-to-service



Life of a packet: external-to-service

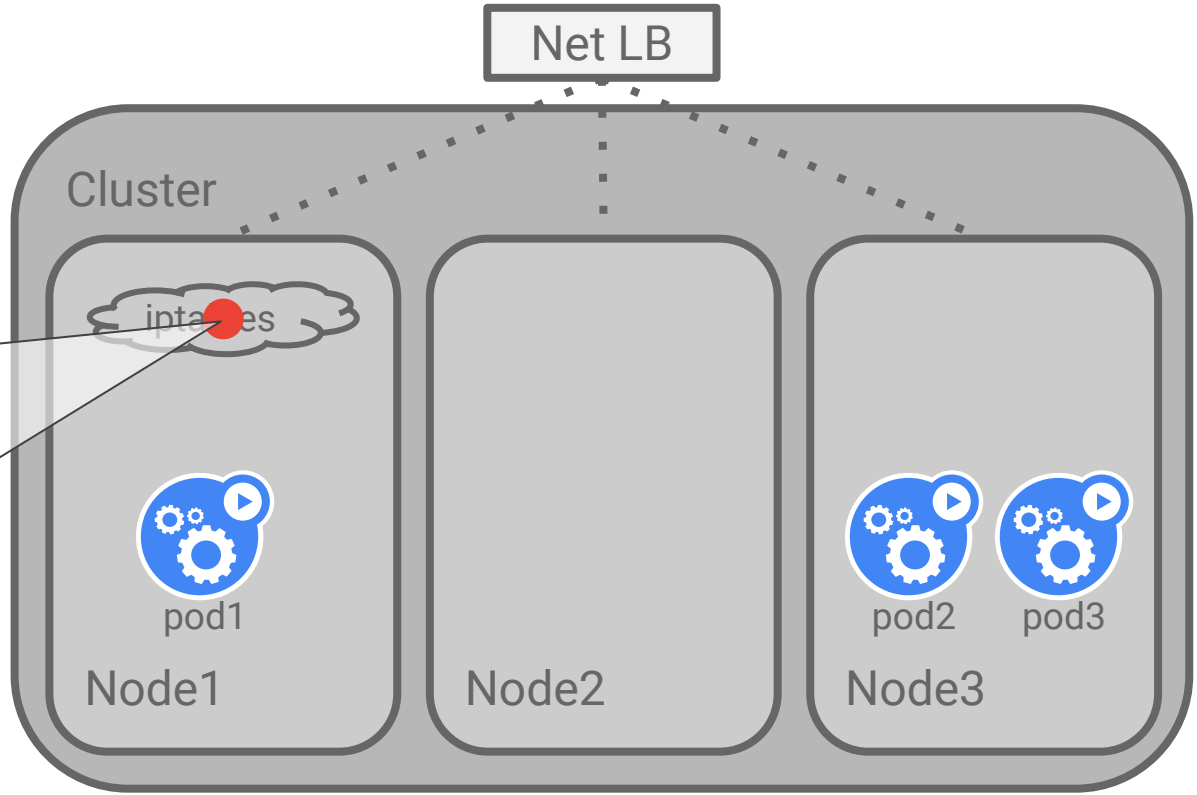


Life of a packet: external-to-service



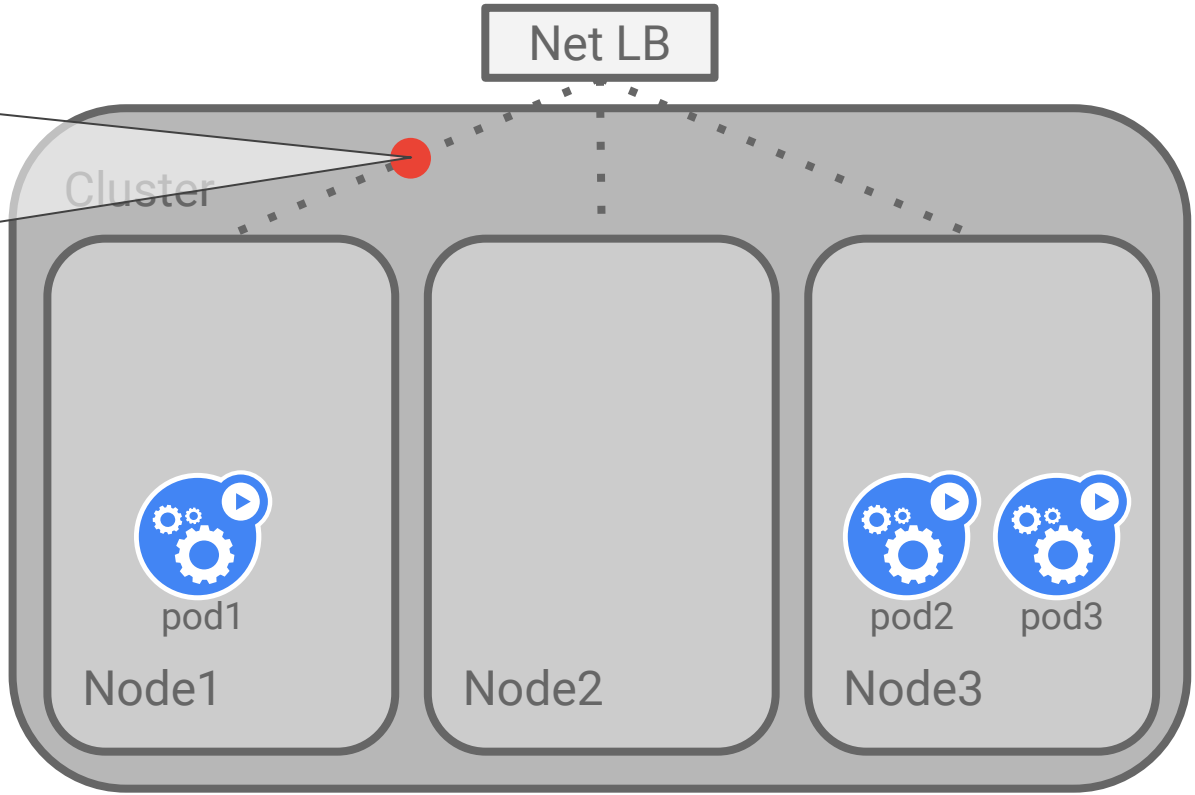
Life of a packet: external-to-service

~~src: pod2~~
src: LB
~~dst: Node1~~
dst: client

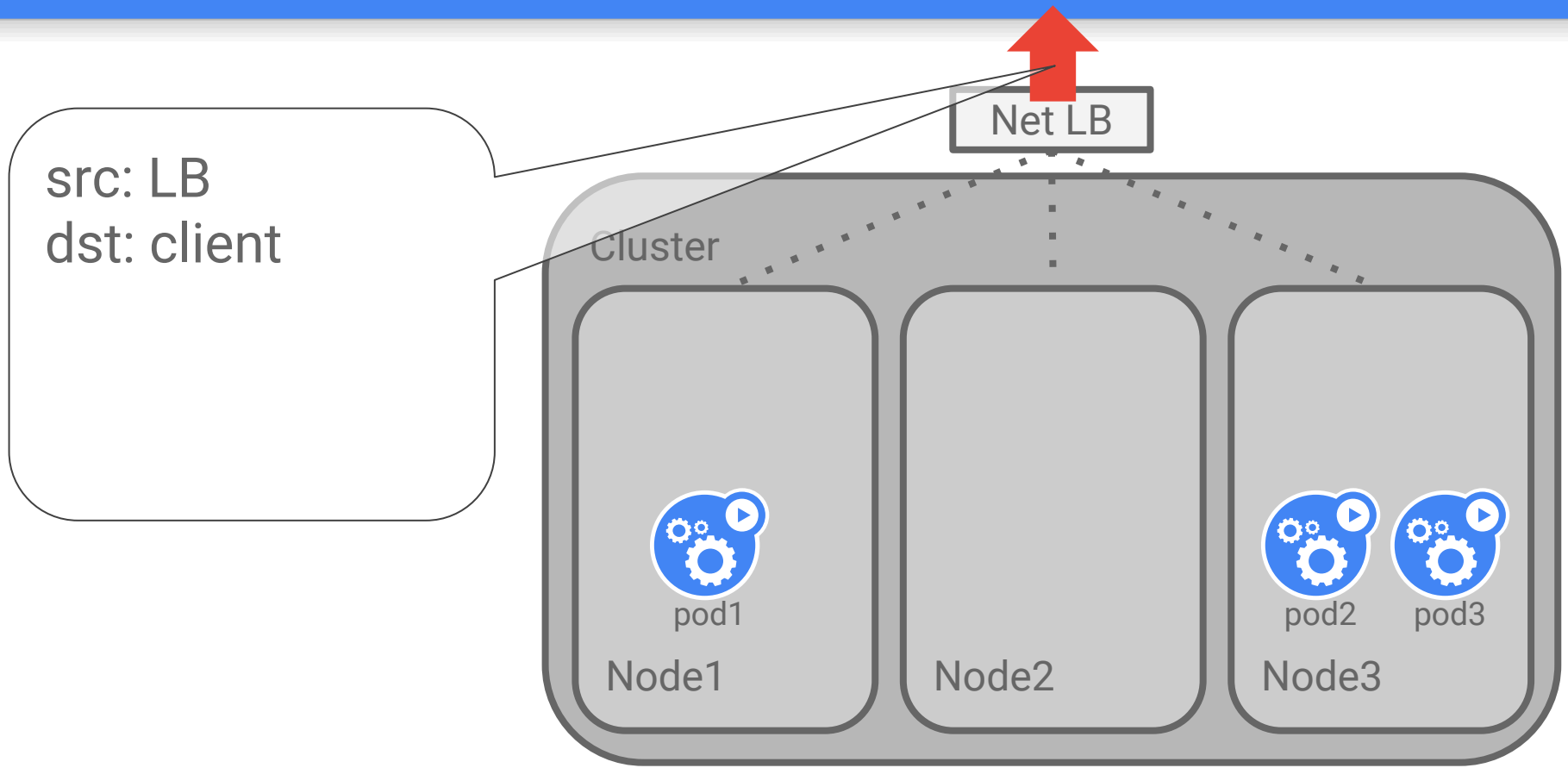


Life of a packet: external-to-service

src: LB
dst: client



Life of a packet: external-to-service



Explain the complexity

To avoid imbalance, we re-balance inside Kubernetes

A backend is chosen randomly from all pods

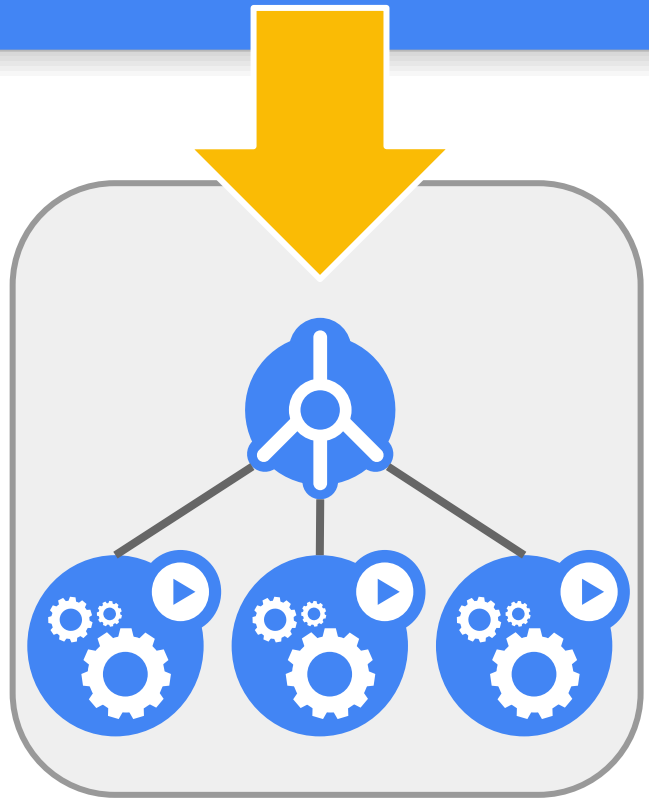
Good:

- Well balanced, in practice

Bad:

- Can cause an extra network hop
- Hides the client IP from the user's backend

Users wanted to make the trade-off themselves



OnlyLocal

Specify an external-traffic policy

iptables will always choose a pod on the same node

Preserves client IP

Risks imbalance

```
kind: Service
apiVersion: v1
metadata:
  name: store-be
  annotations:
service.beta.kubernetes.io/external-traffic:
OnlyLocal
spec:
 type: LoadBalancer
 selector:
    app: store
    role: be
  ports:
  - name: https
    port: 443
```

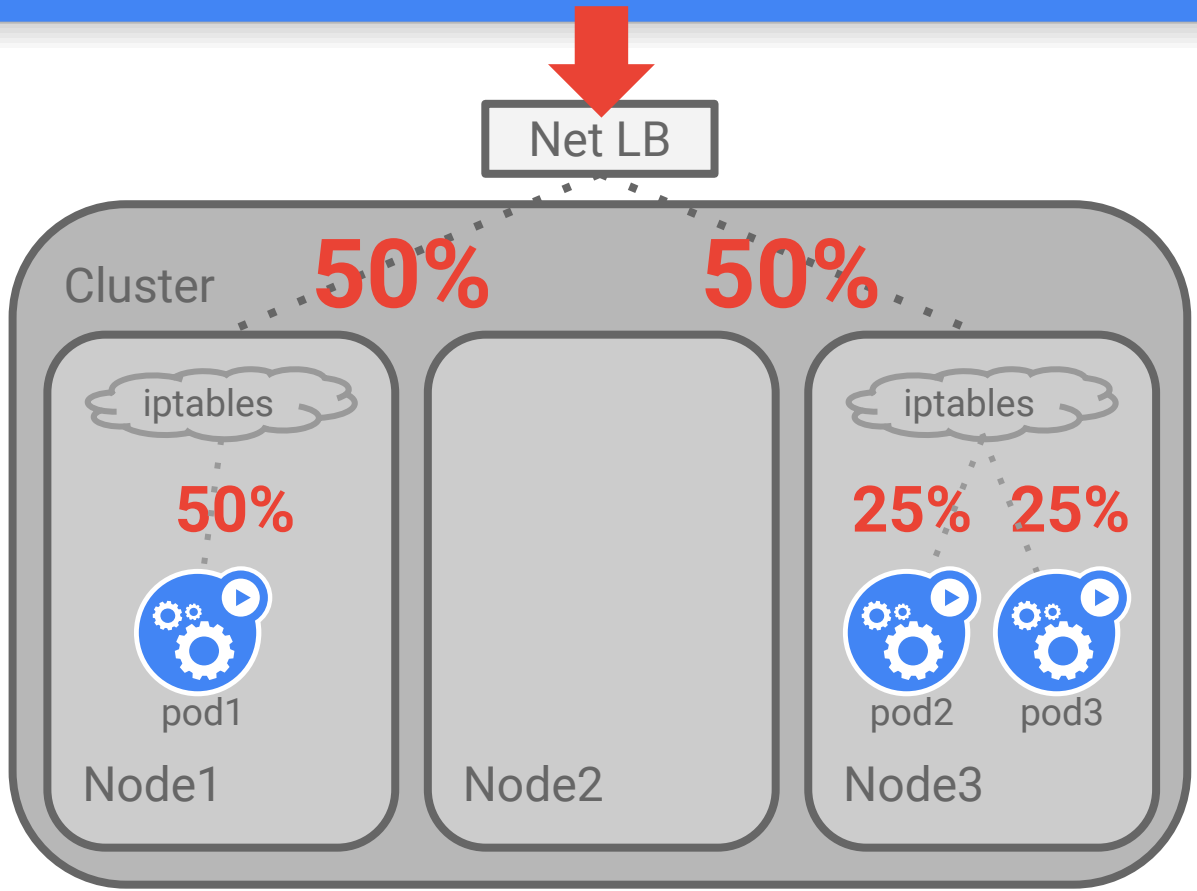
Opt-in to the imbalance problem

In practice Kubernetes spreads pods across nodes

If pods >> nodes: OK

If nodes >> pods: OK

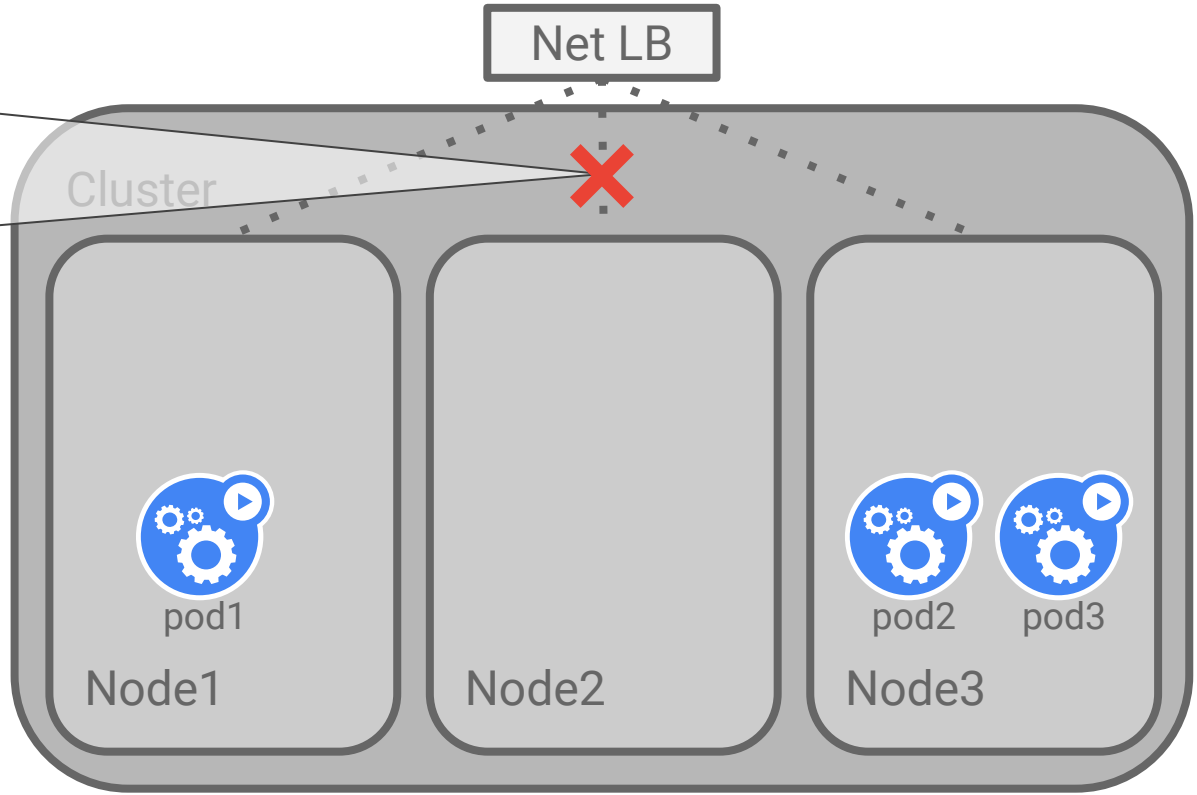
If pods ~= nodes: risk



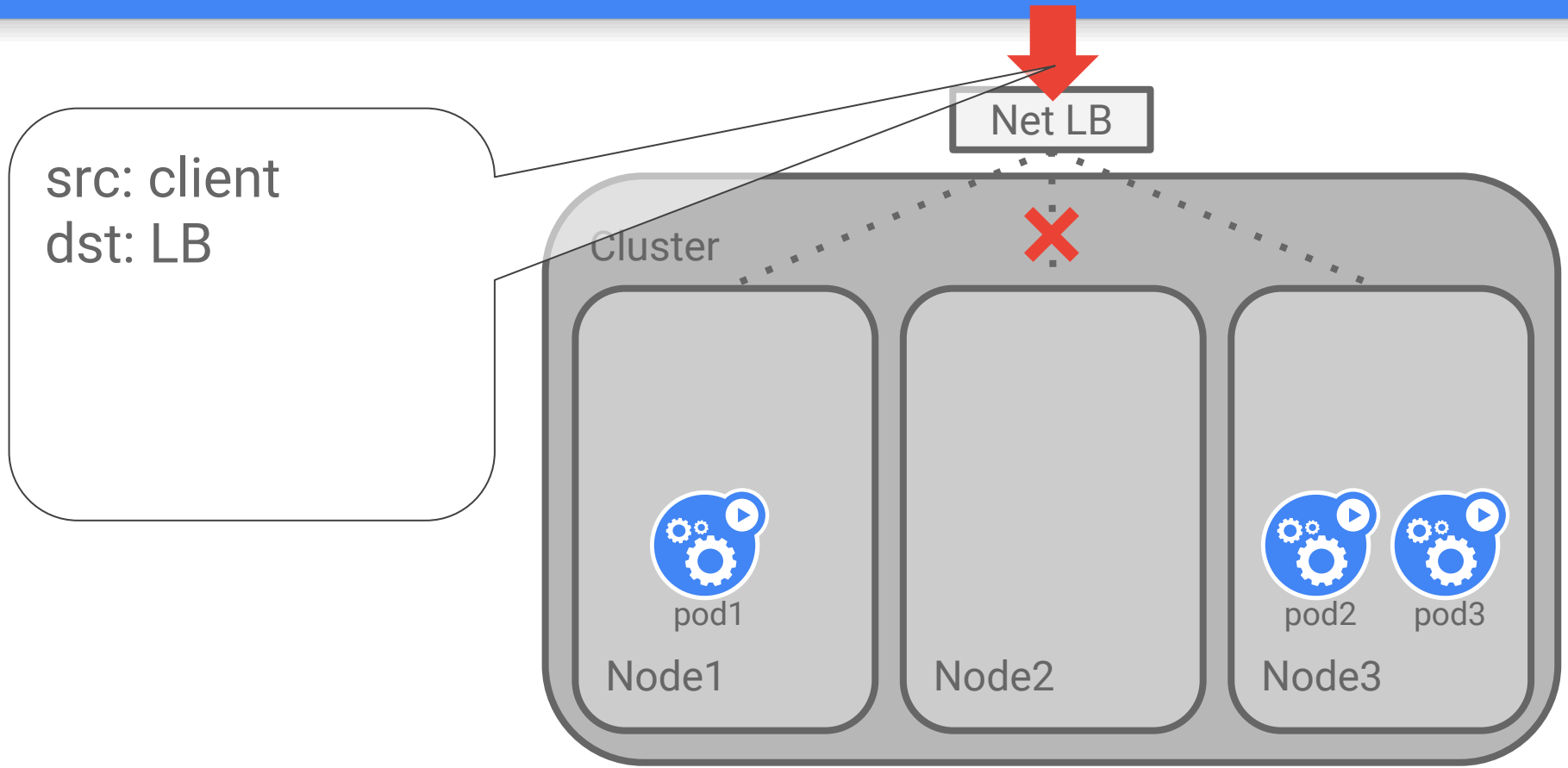
Life of a packet: external-to-service

Not considered

Health-check fails if no backends



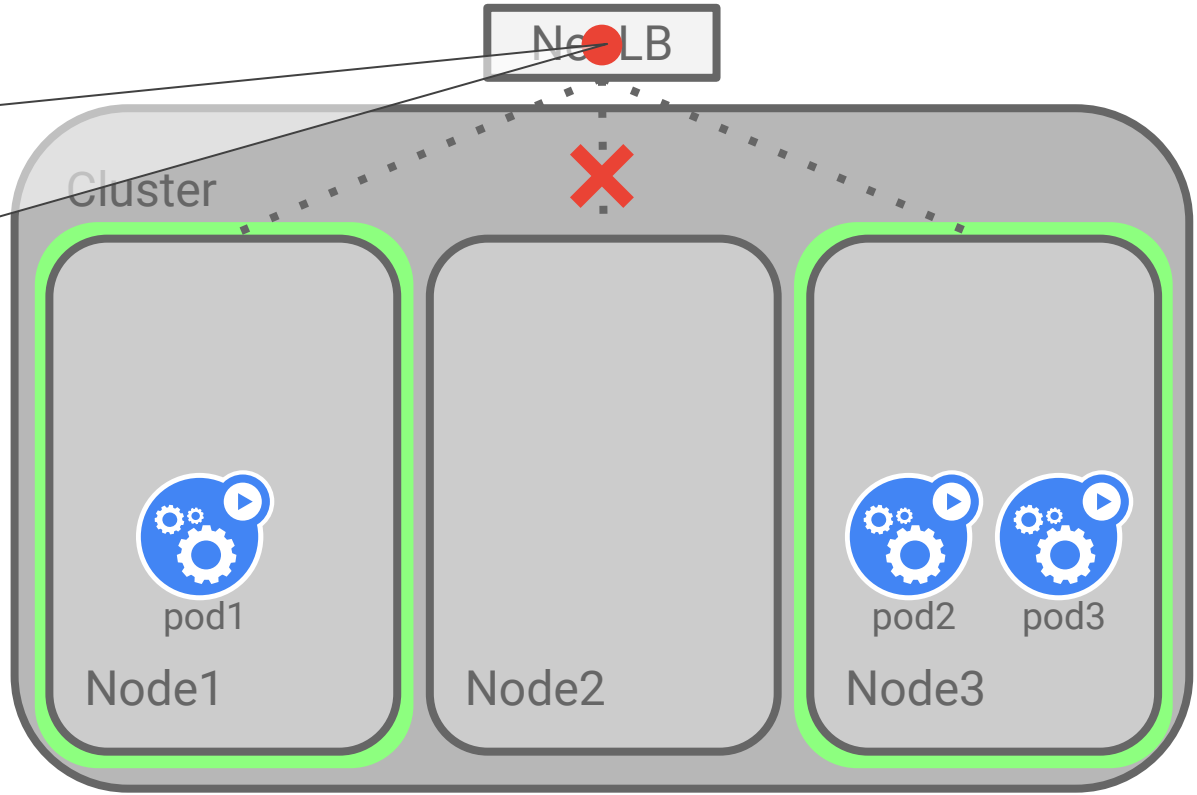
Life of a packet: external-to-service



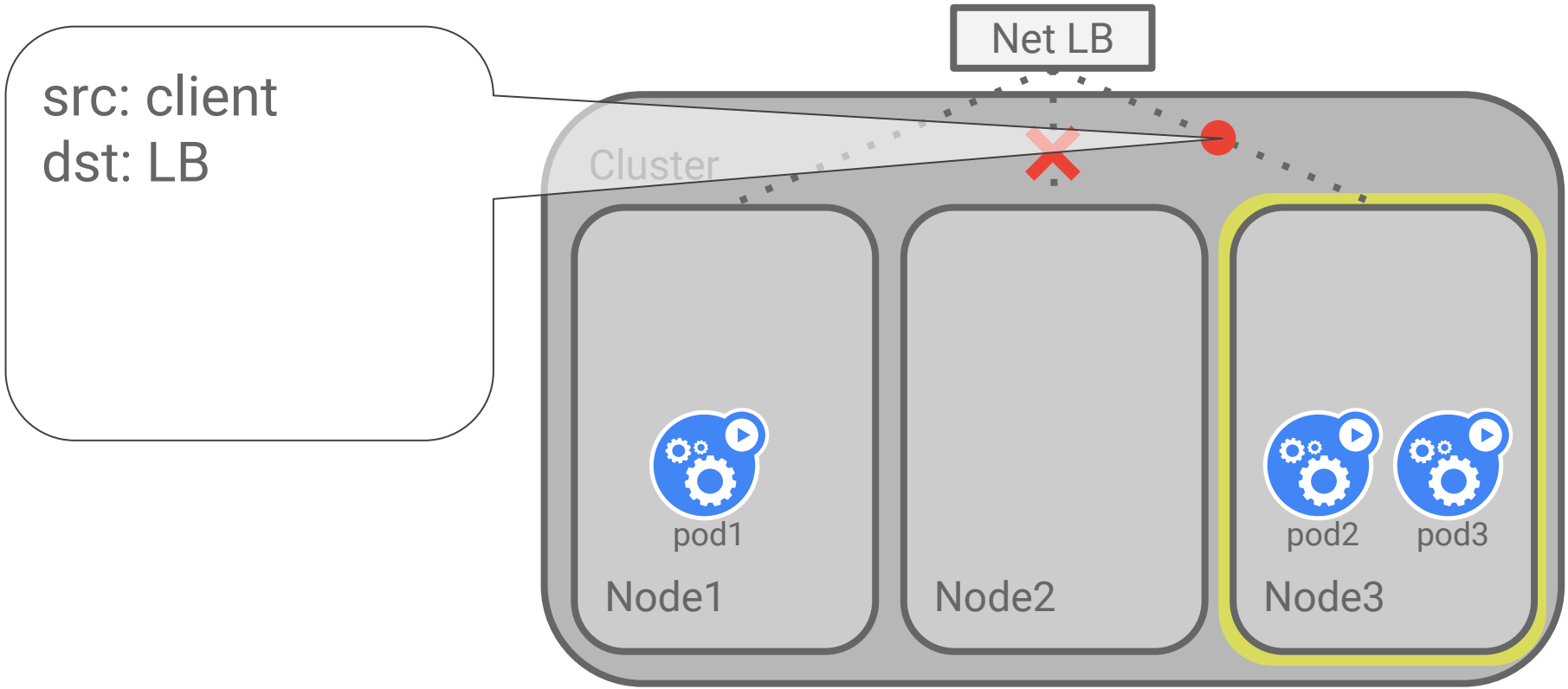
Life of a packet: external-to-service

src: client
dst: LB

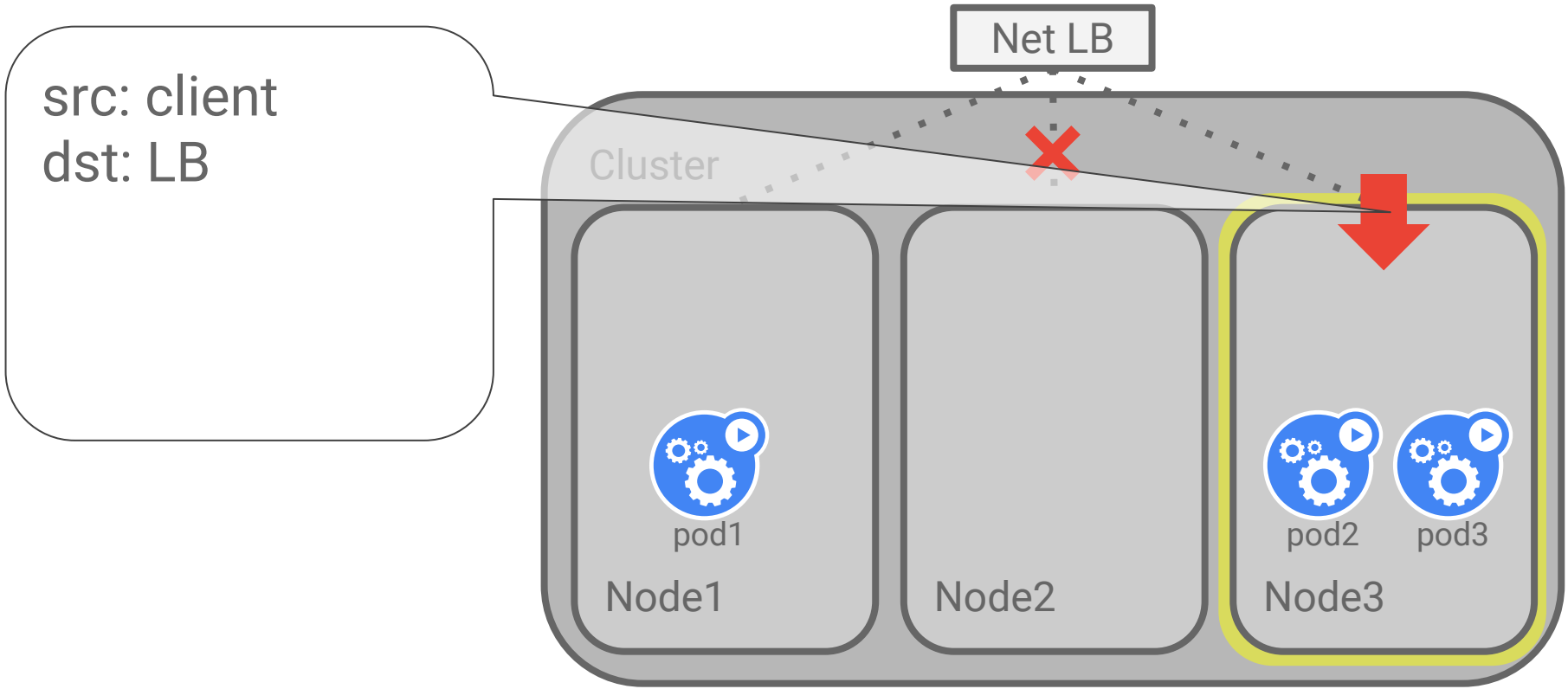
Choose a Node



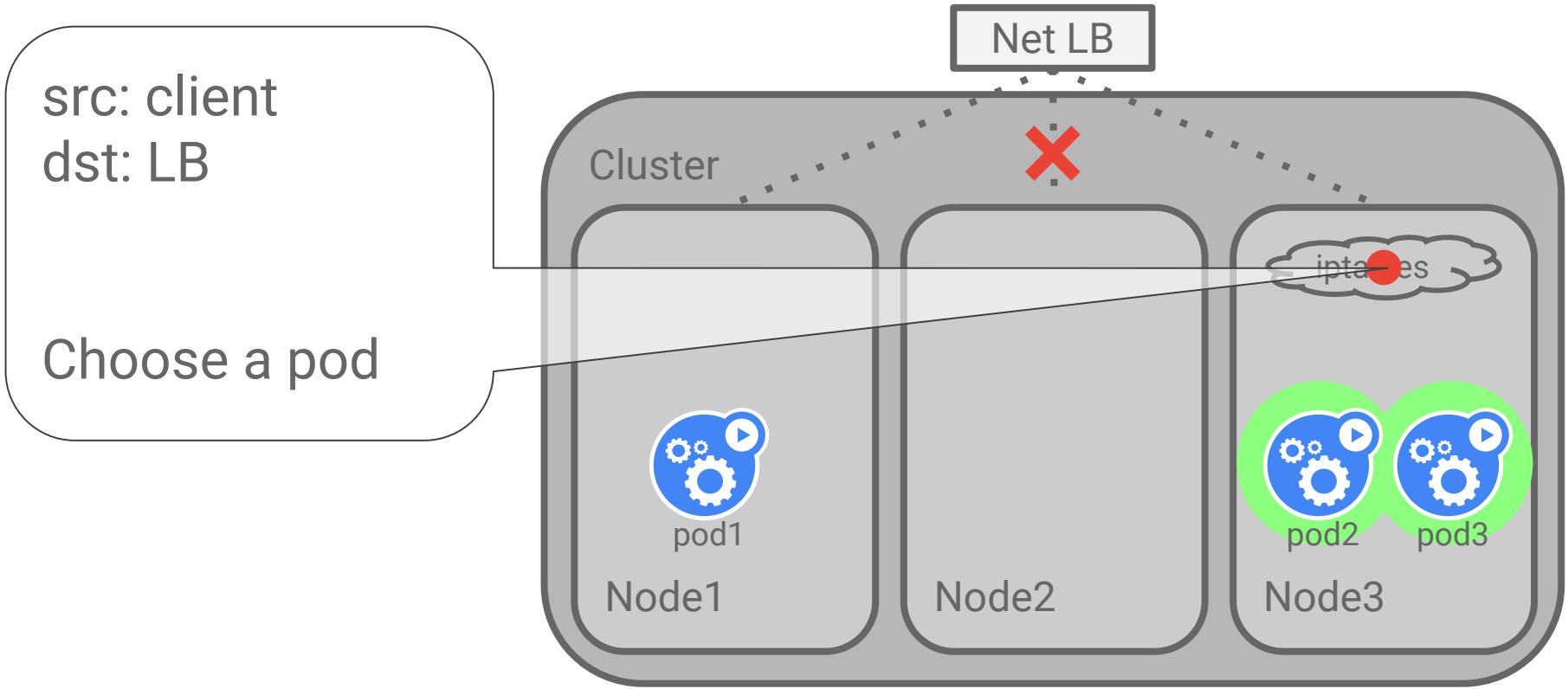
Life of a packet: external-to-service



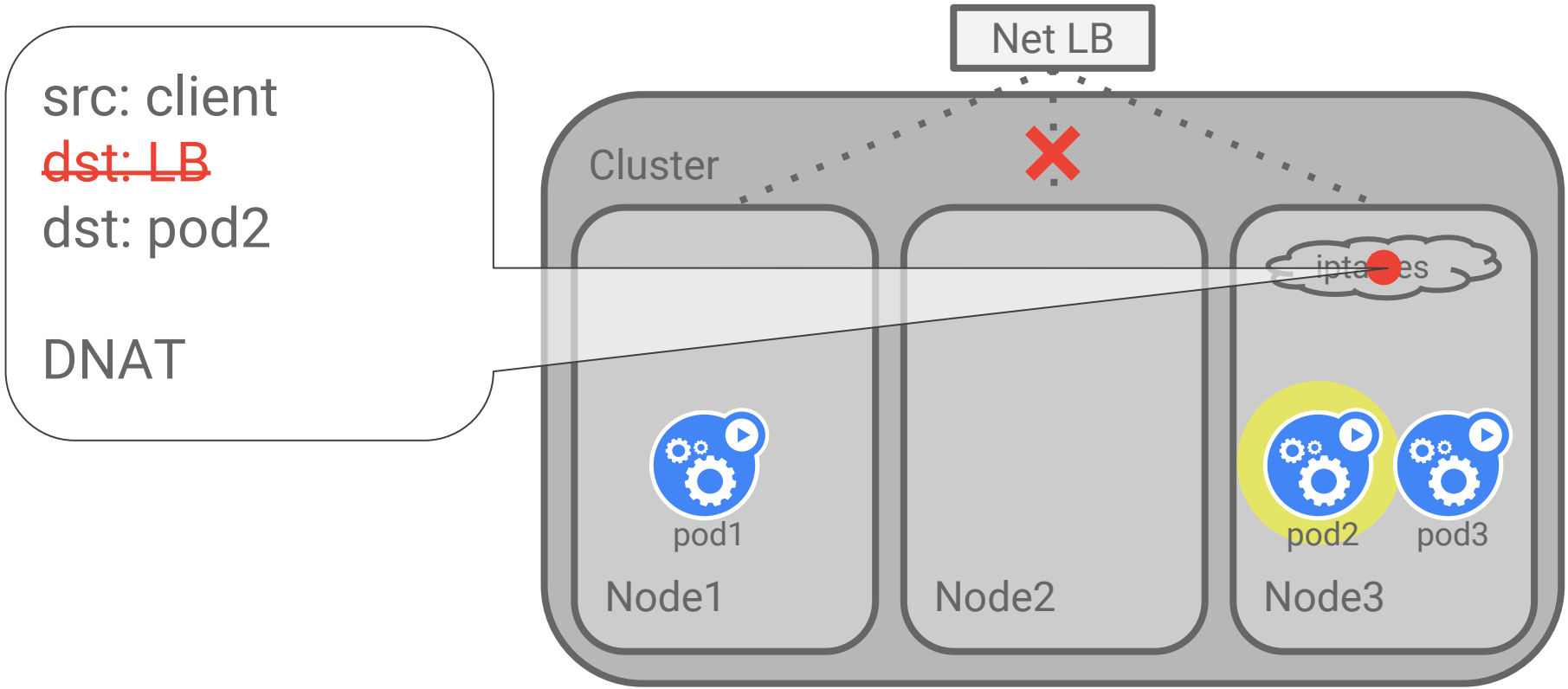
Life of a packet: external-to-service



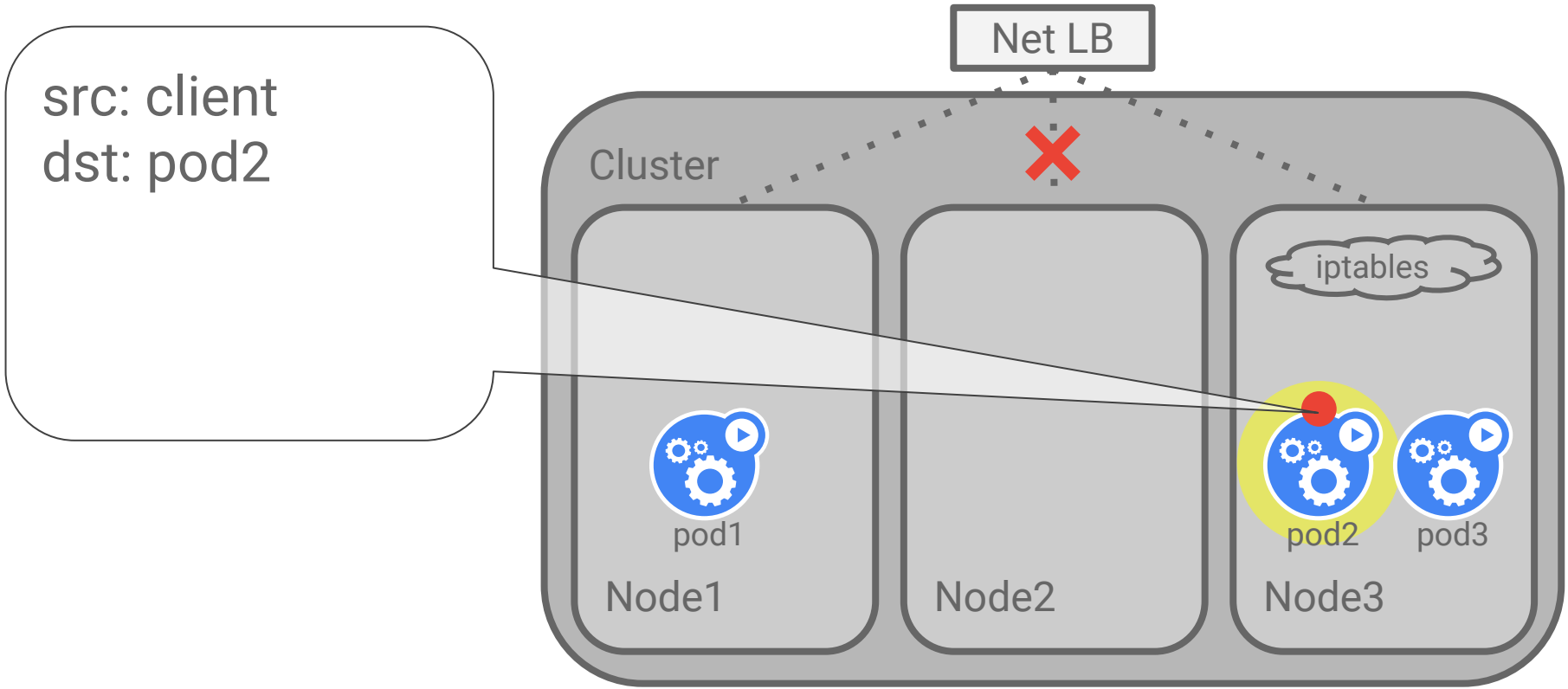
Life of a packet: external-to-service



Life of a packet: external-to-service

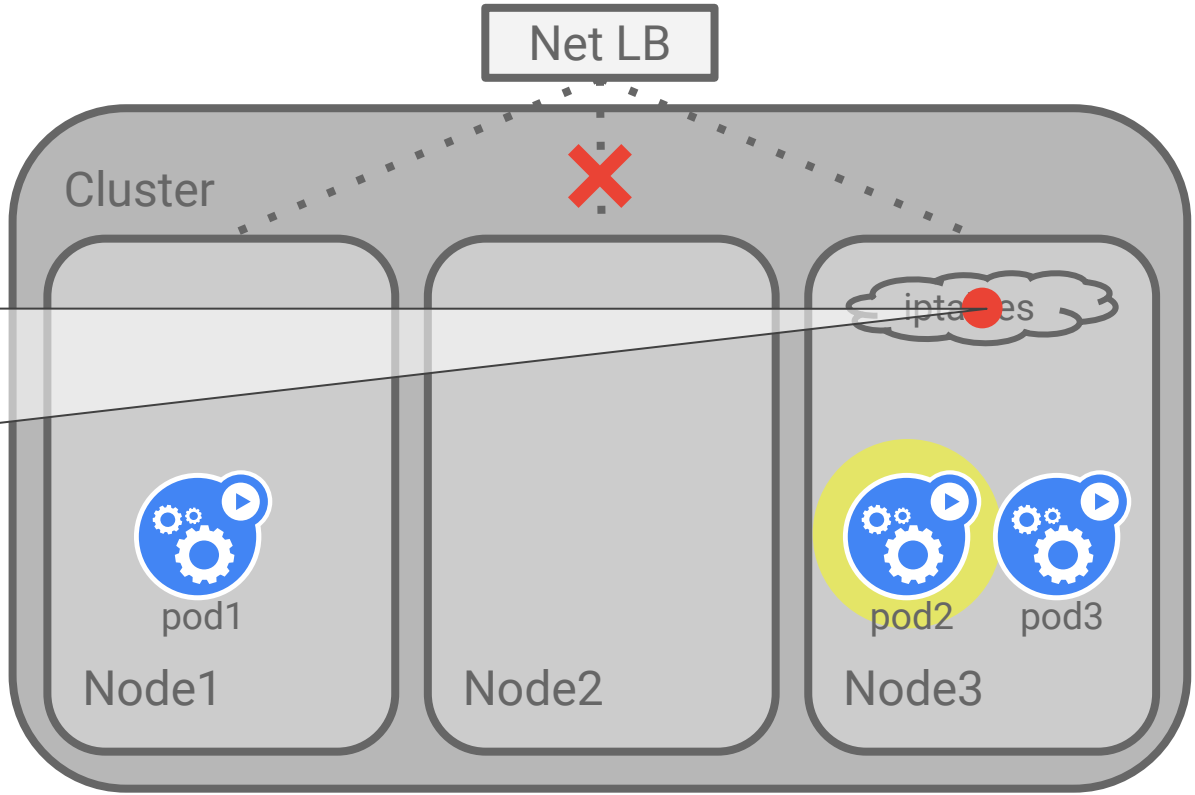


Life of a packet: external-to-service

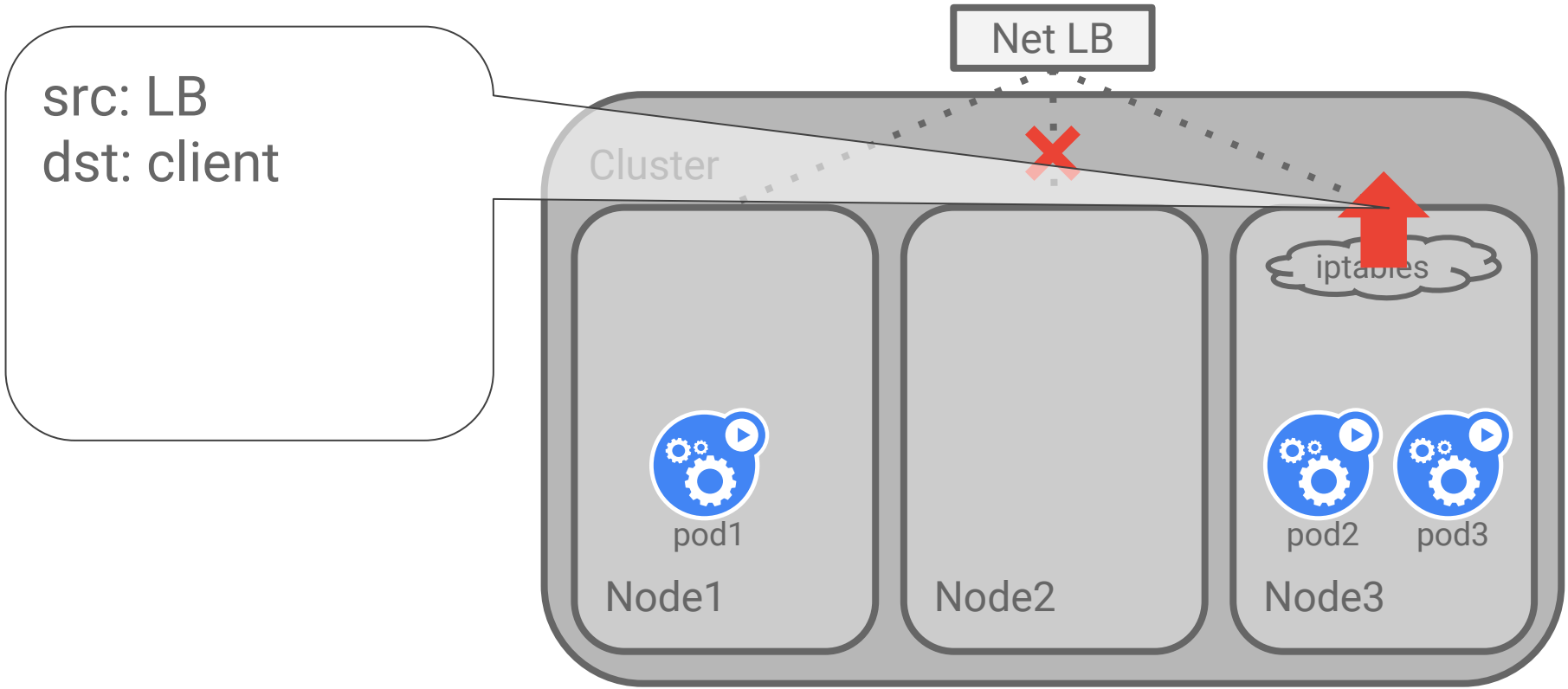


Life of a packet: external-to-service

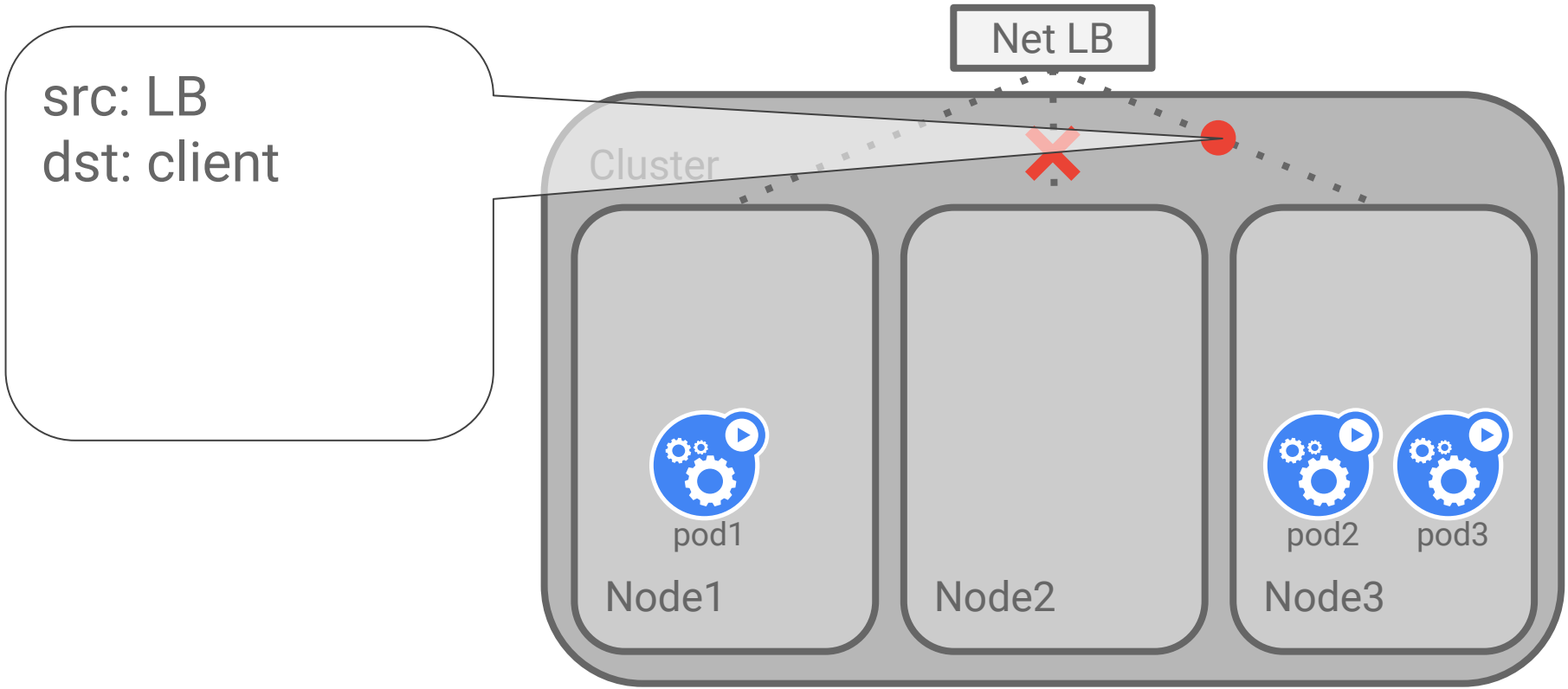
~~src: pod2~~
src: LB
dst: client



Life of a packet: external-to-service

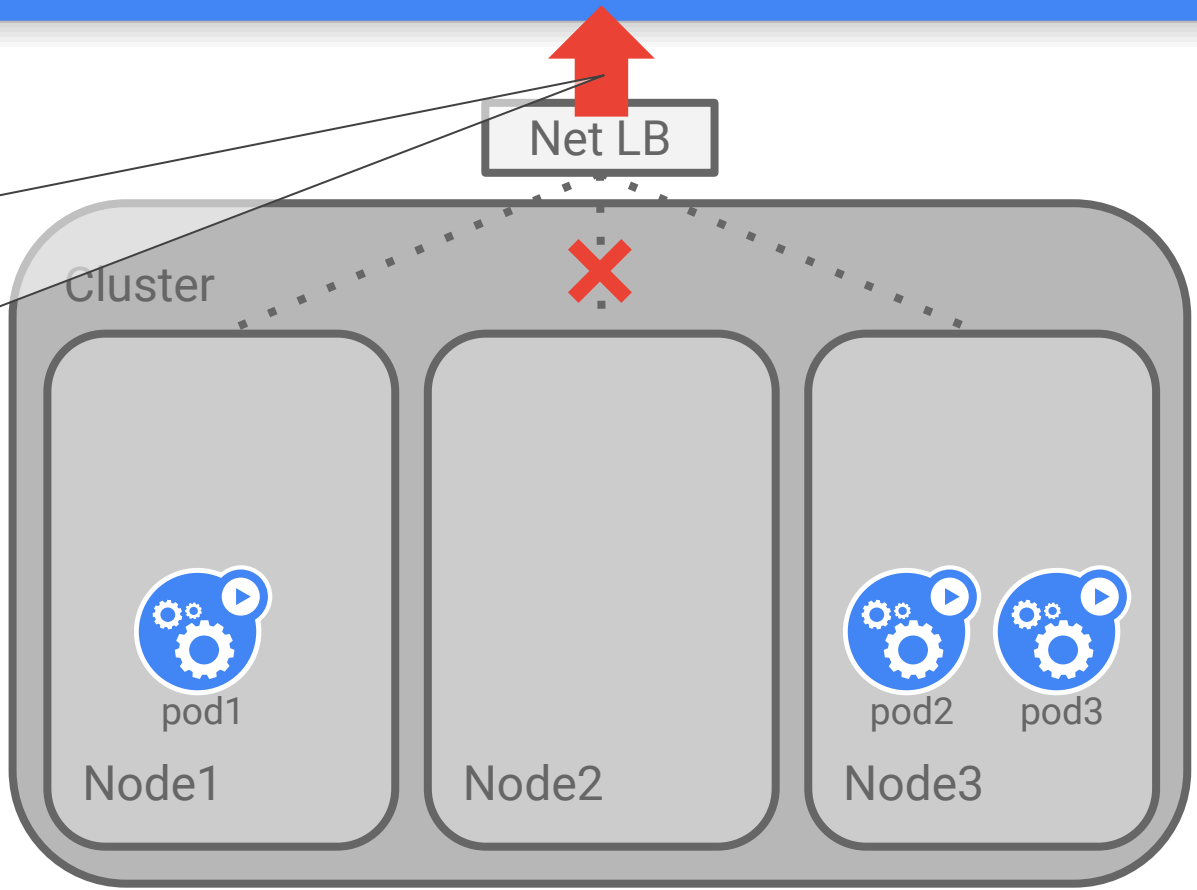


Life of a packet: external-to-service



Life of a packet: external-to-service

src: LB
dst: client



Network Policy

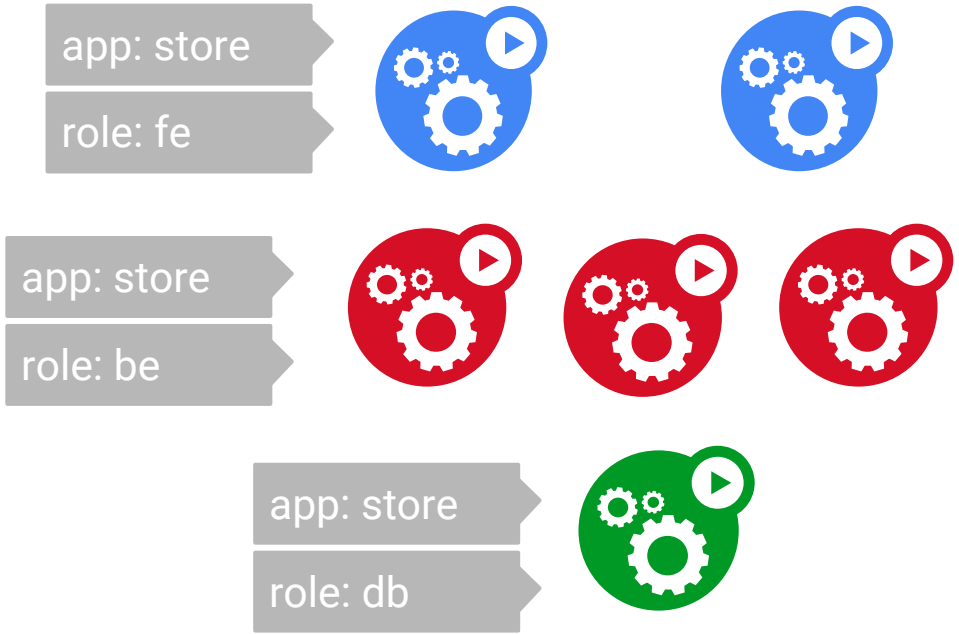
Network Policy

A common pattern for applications is to organize into micro-services or tiers

Example: The classic three-tier app

Users want to “lock down” the network.

Allow some tiers to communicate with others, but not a free-for-all.

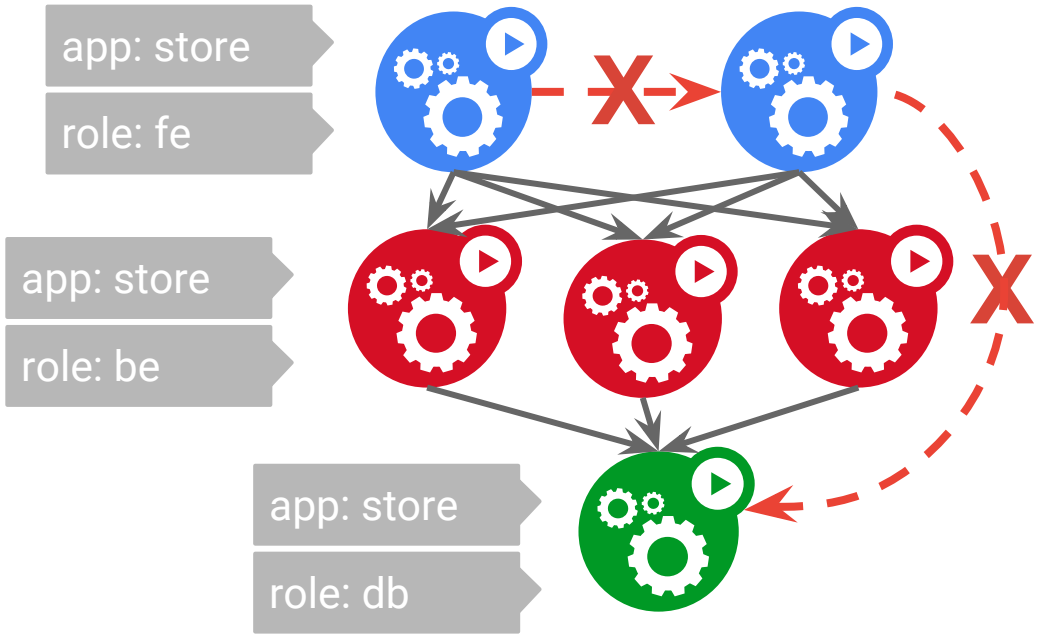


Network Policy

Networks can disallow communication between tiers that are not allowed

- An FE should never reach around to the DB
- An FE should never talk to another FE

Labels are dynamic so these rules must be so also



Namespaces

“Private” scope for creating and managing objects (Pods, Services, NetworkPolicies...)

Namespaced objects are always namespaced

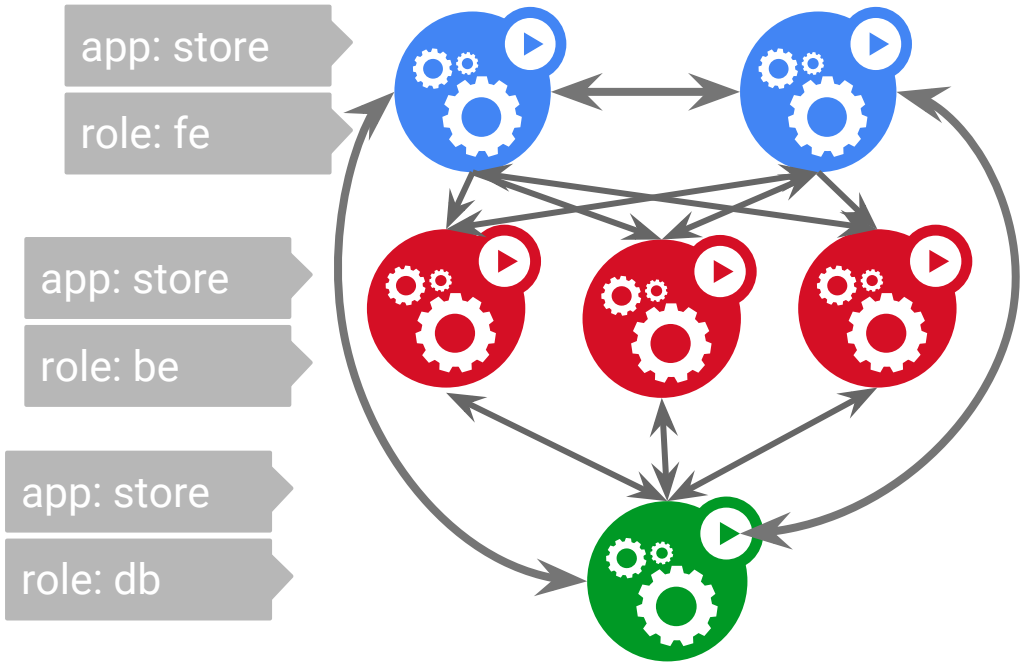
- If you don’t specify a namespace in YAML, use kubectl command-line or the current “context”

```
kubectl -n my-namespace create -f file.yaml
```

Network Policy

When first created, the namespace allows all pods to reach each other.

Remember we said all pods can reach each other.

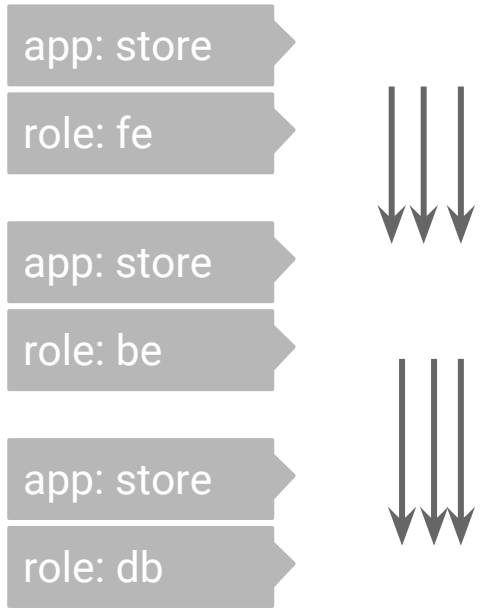


Network Policy

Describe the allowed links

Example:

- Pods labelled “role: be” can receive traffic from “role: fe”
- Pods labelled “role: db” can receive traffic from “role: be”



Network Policy

Install policies

Per-namespace API

Specify:

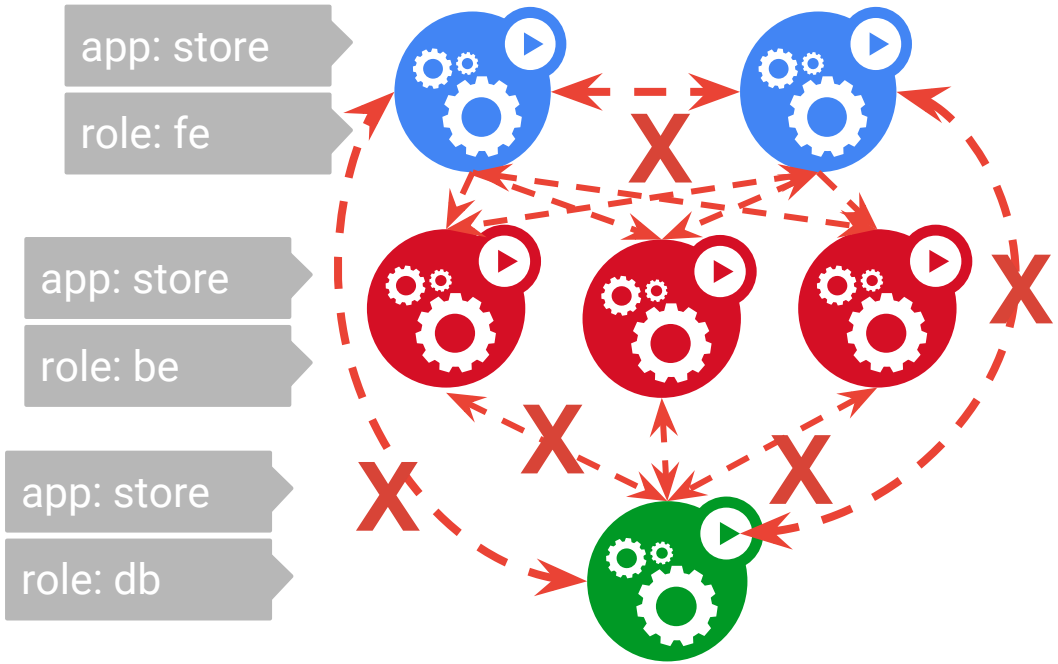
- Pods subject to this policy
- Pods allowed to connect to the subjects
- Port(s) allowed

```
apiVersion: extensions/v1beta1
kind: NetworkPolicy
metadata:
  name: store-net-policy
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
      matchLabels:
        role: be
   ports:
   - protocol: tcp
     port: 6379
```

Network Policy

Switch the network isolation mode to “DefaultDeny”

Absent policies, no network traffic can flow



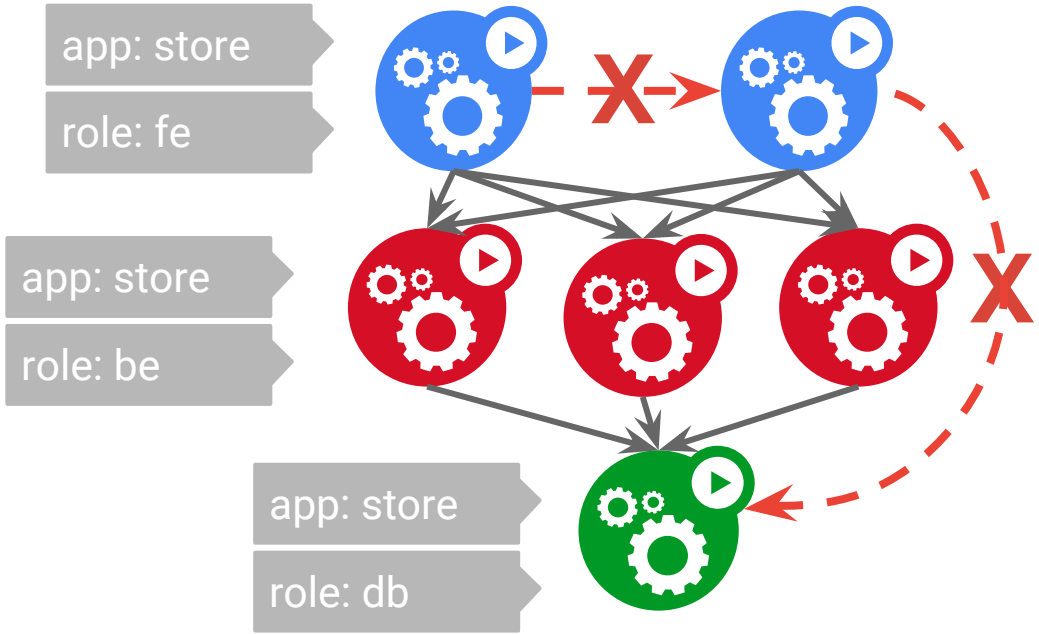
Network Policy

```
kind: Namespace
apiVersion: v1
metadata:
  name: store-namespace
  annotations:
    net.beta.kubernetes.io/network-policy: |
      {
        "ingress": {
          "isolation": "DefaultDeny"
        }
      }
```

Network Policy

Connections allowed by NetworkPolicies are OK

Ordering is very important for these steps



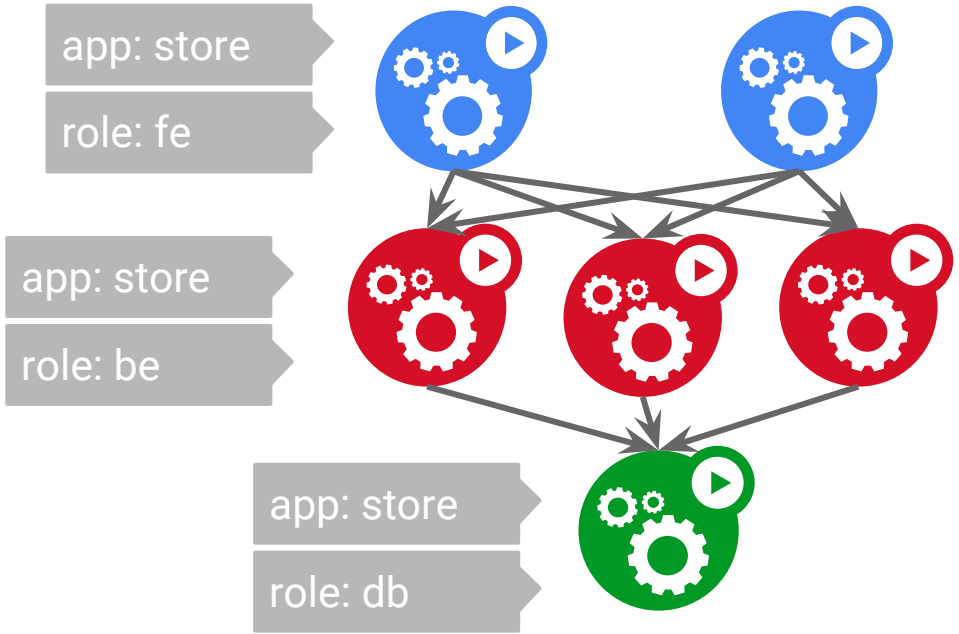
Network Policy

There may be multiple NetworkPolicies in a Namespace

- Purely additive

There is no way to specify “deny” in NetworkPolicy

Implemented at L3/L4, no L7 support



Open implementation

Beta in v1.6
Expected GA in v1.7

Kubernetes allows the user to select the best method to implement Network policy.

Today there is a wide range of choices:

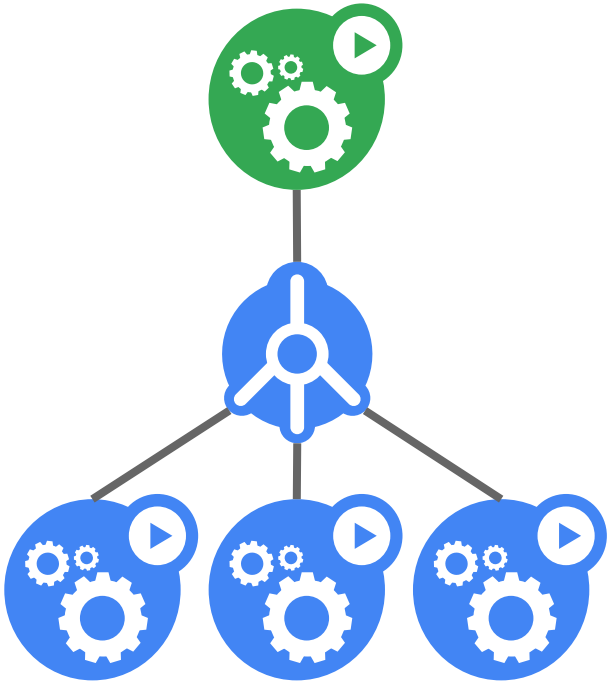
- Calico
- Contiv
- Openshift
- Romana
- Trireme
- WeaveNet
- And more...

Watch this space

Kubernetes Networking is a moving target

The efforts of Open Source developers continue to improve and simplify the system

Hopefully the next KubeCon we will have the opportunity to present more.



Kubernetes is Open

 open community

 open source

 open design

 open to ideas

https://kubernetes.io

Code: github.com/kubernetes/kubernetes
Chat: slack.k8s.io
Twitter: @kubernetesio