Networking: Containers & Kubernetes

Aula Teórica nº12

2020/2021

Docker Networking

What does this mean for networking?

- In traditional networking services used to run on dedicated appliances – this is all now moving inside the nodes
- Running microservices means A LOT of east-west traffic – a lot more action is happening inside the nodes as well
- Linux networking stack and virtual switches are essential
- Nothing is static – containers spin up and down all the time

Docker Network Modes

- **Bridge**: default network driver. Containers are connected to a Docker0 Linux Bridge.

Used by Kubernetes

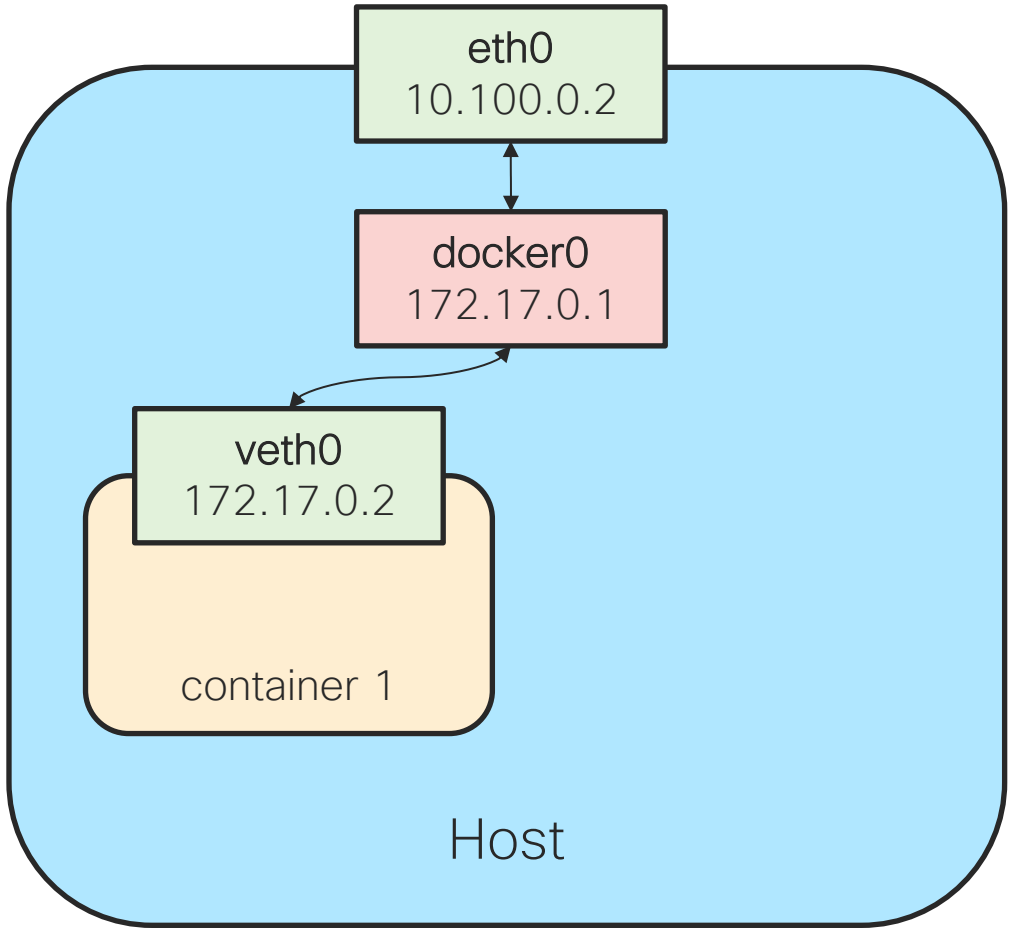
- **Host**: containers use the host networking directly (i.e. share the host network namespace)
- **None**: container not connected, or connectivity is provided by a 3rd party
- **Container**: borrow connectivity from another container

- **MacVLAN**: allows you to assign a MAC address to a container, making it appear as a physical device on your network. The Docker daemon routes traffic to containers by their MAC addresses
- **Overlay** and Third-party **Docker network plugins**.

Basic Docker Networking with Bridge Mode

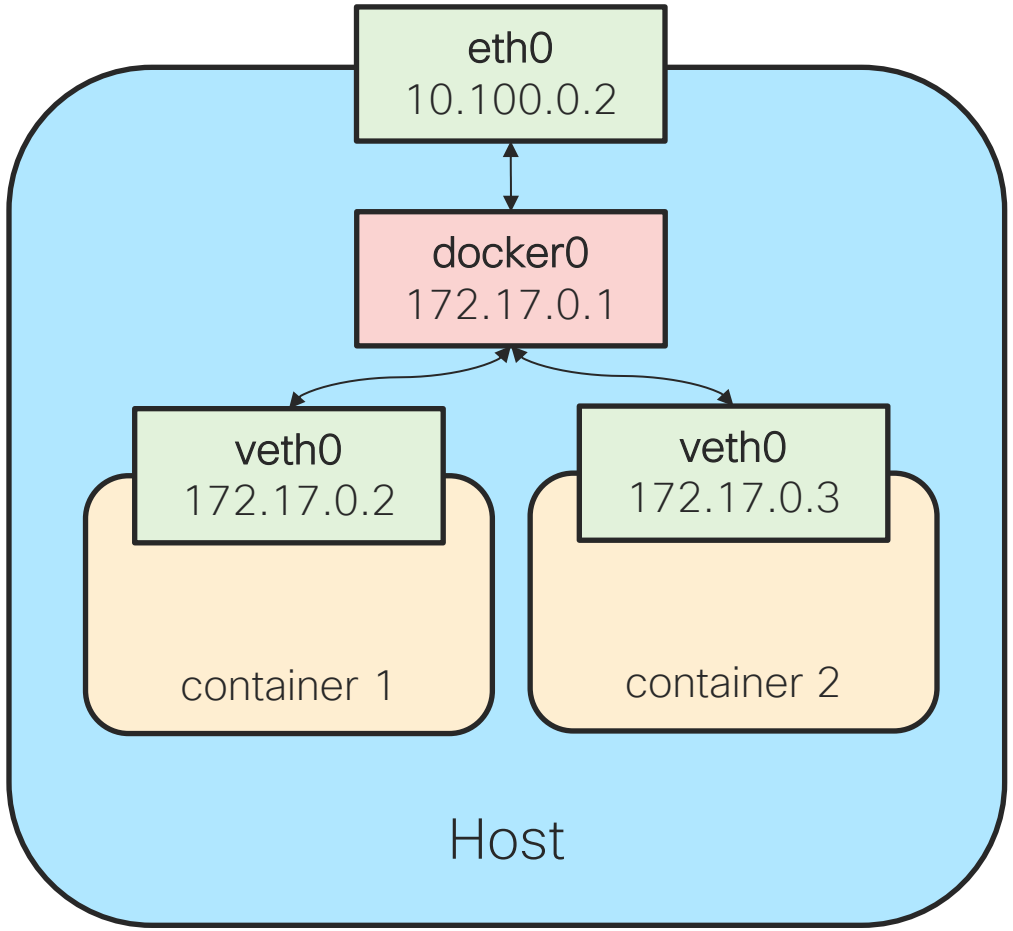
- By default Docker configures 172.17.0.1/16 on the docker0 Linux bridge.
- When a container is created, the runtime engine will connect its virtual Ethernet interface to the docker0 bridge.
- An IP Address is assigned to the container, and docker0 Bridge is the default gateway.

```
# docker inspect 925984cfda27 | grep NetworkMode
            "NetworkMode": "default",
```



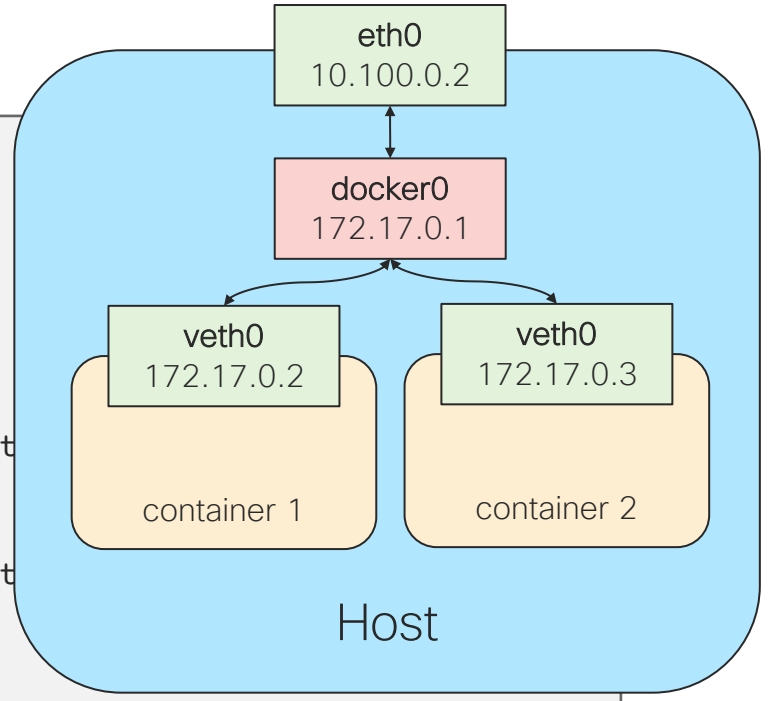
Basic Docker Networking with Bridge Mode

- Containers on the same host communicate within the same subnet via docker0 Linux Bridge.
- External communication uses the host TCP/IP stack and Docker automatically configures NAT.
- Containers can also consume Host ports to expose services from the container.



Bridge mode: what is connected to what?

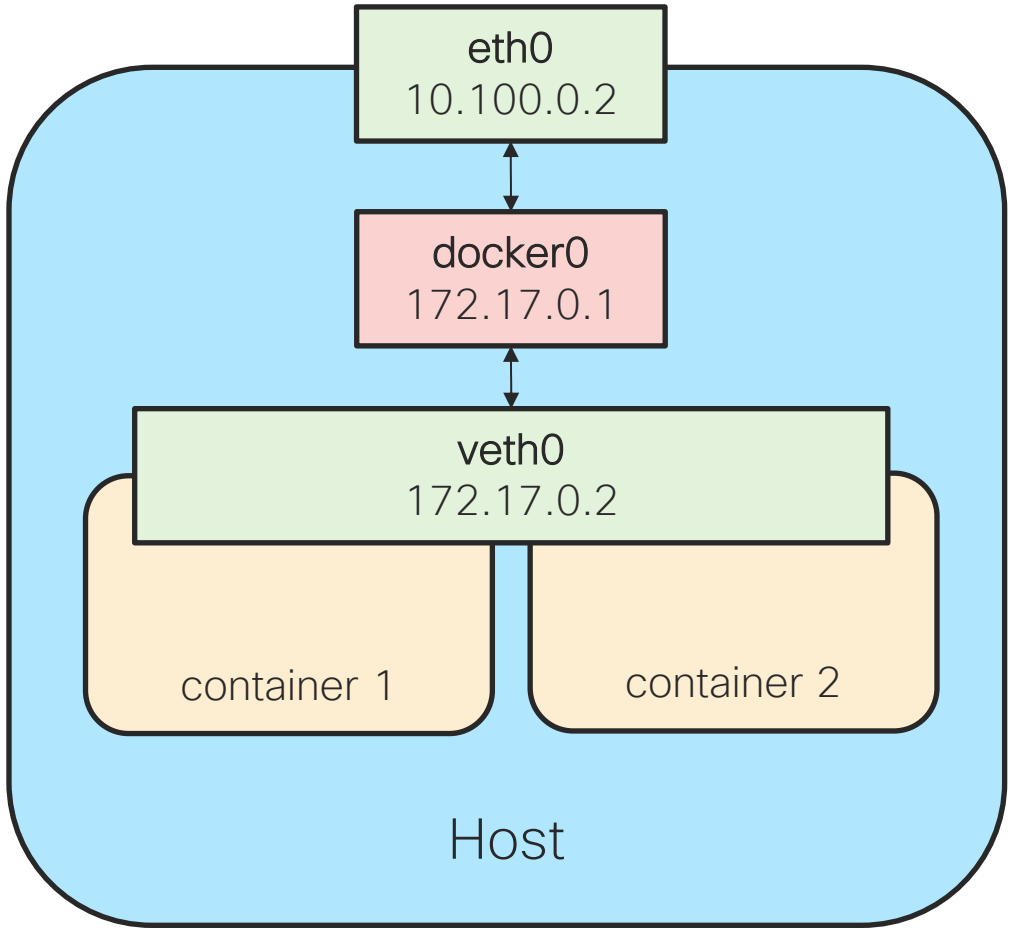
```
# docker inspect 925984cfda27 | grep Pid.:
            "Pid": 5079,

# docker inspect 5d1fe8d6c14f | grep Pid.:
            "Pid": 1617,

# nsenter -t 5079 -n ip add show eth0 | grep eth0@
59: eth0@if60: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue stat

# nsenter -t 1617 -n ip add show eth0 | grep eth0@
53: eth0@if54: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue stat


# ip add | egrep "60|54"
54: vethba1ecf2@if53: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
group default
60: veth944dfc0@if59: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
group default

# brctl show docker0
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02424bcc9785       no              veth944dfc0
                                                        vethba1ecf2
                                                        eth0
```



Containers can also share the same namespace

- It is also possible for two or more containers to share the same net namespace.
- Allows to run apps that require to talk to other services via localhost.

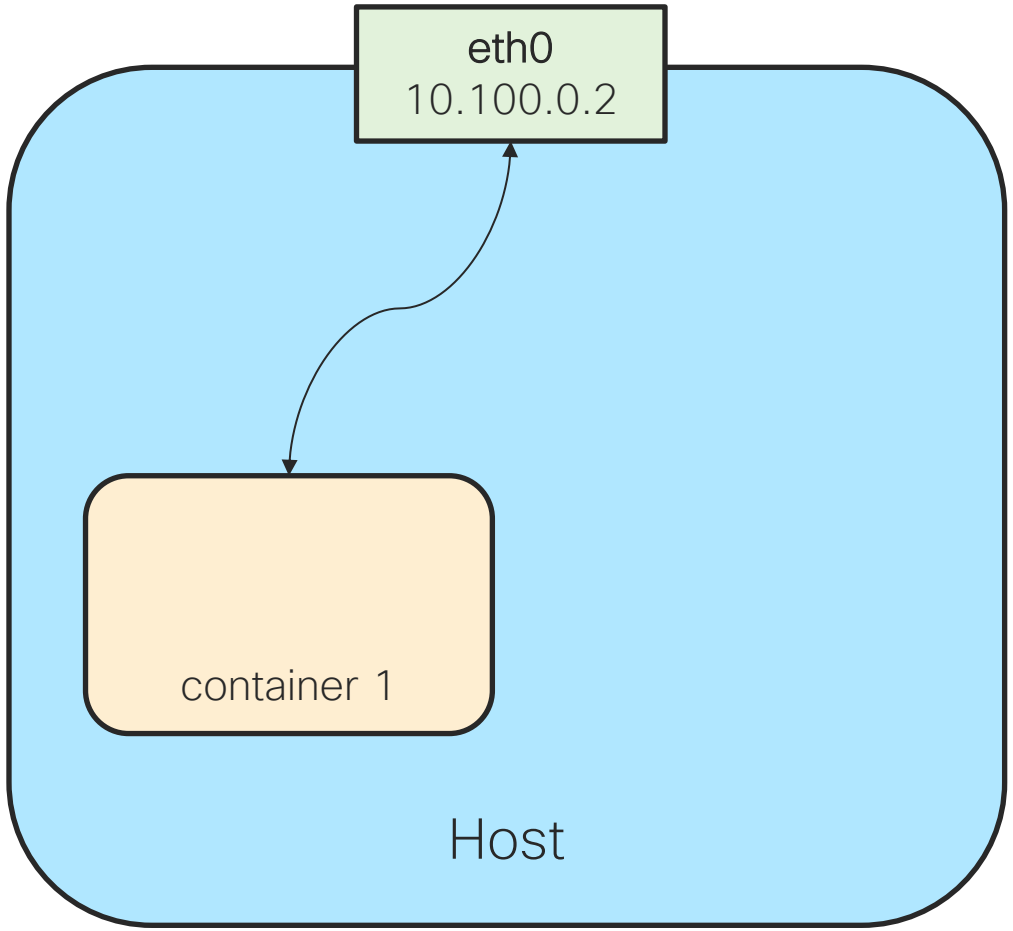
```
# docker inspect 925984cfda27 | grep NetworkMode
            "NetworkMode": "default",

# docker inspect 7357b02dcb40 | grep NetworkMode
            "NetworkMode": "container: 925984cfda27…
```



Basic Docker Networking with Host mode

- Containerized process has no network namespace
- Sees the same network interface as a non-containerized process
- Is the same as running the process on the host directly from a networking point of view.

```
# docker inspect cd786f17acab | grep NetworkMode
            "NetworkMode": "host",
```



Exposing a port

```
$ docker run -d -p 8080:8080 containers.cisco.com/ikovacev/hellocl:v1.0
3241ad886ebd6fc348bbc2f64f24281635b7a162900a42370bcf6831a19f51b2

$ curl localhost:8080
{"message":"Hello CiscoLive!","platform":"Linux","release":"4.9.125-
linuxkit","hostName":"3241ad886ebd"}
```

- Docker runtime pushes rules to local host iptables to achieve PAT
- Linux kernel netfilter then does all the work in the data-path

IP Tables

iptables

- A user-space app to configure Linux-kernel firewall
- Tables: There are four built-in tables. Each contains several chains
  - Filter table (Default): For packet filtering
  - NAT table: For network address translation
  - Mangle table: For packet manipulation
  - Raw table: For configuration exemptions
- Chains are a list of rules
  - Default: PREROUTING, INPUT, OUTPUT, FORWARD, POSTROUTING (not all tables have all)
  - Can be stitched together (i.e. daisy-chained) to enforce complex logics
- Rules: Each rule specifies the matching criteria of an IP packet and the action it should take
  - Match: source/destination IP, source/destination port, connection state, etc.
  - Actions: ACCEPT, REJECT, DROP, MARK, DNAT, SNAT, LOG, etc.

Displaying and understanding iptables

Default policy

Default chains

Packet hits

Target aka. action

Configured chains

Matching condition

```
# iptables -nvL
Chain INPUT (policy ACCEPT 1990K packets, 419M bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy ACCEPT 161M packets, 93G bytes)
 pkts bytes target     prot opt in     out     source               destination
 161M   93G DOCKER-USER  all  --  *      *       0.0.0.0/0            0.0.0.0/0
 161M   93G DOCKER-ISOLATION-STAGE-1  all  --  *      *       0.0.0.0/0            0.0.0.0/0
71896  175M ACCEPT     all  --  *      docker0  0.0.0.0/0       0.0.0.0/0      ctstate
RELATED,ESTABLISHED
    0     0 DOCKER     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
36954 1983K ACCEPT     all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
    0     0 ACCEPT     all  --  docker0 docker0  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT 717K packets, 153M bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain DOCKER (1 references)
 pkts bytes target     prot opt in     out     source               destination

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
 pkts bytes target     prot opt in     out     source               destination
36954 1983K DOCKER-ISOLATION-STAGE-2  all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
 161M   93G RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

Chain DOCKER-ISOLATION-STAGE-2 (1 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  *      docker0  0.0.0.0/0            0.0.0.0/0
36954 1983K RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0
...
```

Understanding what is iptables doing with packets (container -> external)

- Can get quite messy as chains can be referenced and nested -> iptables spaghetti
- A very rudimentary way is resetting the packet counters and see which lines get hit:

```
# iptables -Z
# nsenter -t 5079 -n ping -f -c 1234 10.48.37.151
PING 10.48.37.151 (10.48.37.151) 56(84) bytes of data.

--- 10.48.37.151 ping statistics ---
1234 packets transmitted, 1234 received, 0% packet loss, time 228ms
rtt min/avg/max/mdev = 0.099/0.166/8.104/0.231 ms, ipg/ewma 0.185/0.181 ms
# iptables -nvL | egrep "Chain|pkts|^$|1234"
Chain INPUT (policy ACCEPT 93 packets, 6308 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain FORWARD (policy ACCEPT 1717 packets, 456K bytes)
 pkts bytes target     prot opt in     out     source               destination
 1234  104K ACCEPT     all  --  *      docker0  0.0.0.0/0            0.0.0.0/0           ctstate RELATED,ESTABLISHED
 1234  104K ACCEPT     all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
 pkts bytes target     prot opt in     out     source               destination
 1234  104K DOCKER-ISOLATION-STAGE-2  all  --  docker0 !docker0  0.0.0.0/0            0.0.0.0/0
…
```

"Yes, but this is a controlled environment..."

Exposing a port in Docker

```
# docker run -d -p 8080:8080 containers.cisco.com/ikovacev/hellocl:v1.0
e8e6ad133e79bf90f75e541719abd93ceac409228ce47ce1733068a74311c925
```

Docker pushes NAT rules to iptables

```
# iptables -nvL -t nat --line-numbers
Chain PREROUTING (policy ACCEPT 411 packets, 32926 bytes)
num   pkts bytes target     prot opt in     out     source               destination
1     9422  579K DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0        tcp dpt:3389
to:192.168.1.2:3389
2     352K   83M DOCKER     all  --  *      *       0.0.0.0/0            0.0.0.0/0        ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 107 packets, 6432 bytes)
num   pkts bytes target     prot opt in     out     source               destination
1       71  4504 MASQUERADE  all  --  *      !docker0  172.17.0.0/16        0.0.0.0/0
2    2063K  126M MASQUERADE  all  --  *      ens160  0.0.0.0/0            0.0.0.0/0
3        0     0 MASQUERADE  tcp  --  *      *       172.17.0.4           172.17.0.4       tcp dpt:8080

Chain DOCKER (1 references)
num   pkts bytes target     prot opt in     out     source               destination
1        0     0 RETURN     all  --  docker0 *       0.0.0.0/0            0.0.0.0/0
2        3   192 DNAT       tcp  --  !docker0 *      0.0.0.0/0            0.0.0.0/0        tcp dpt:8080 to:172.17.0.4:8080
```

TIP: grep for the port# to quickly verify iptables config

Kubernetes Networking

Key components

Kube-proxy

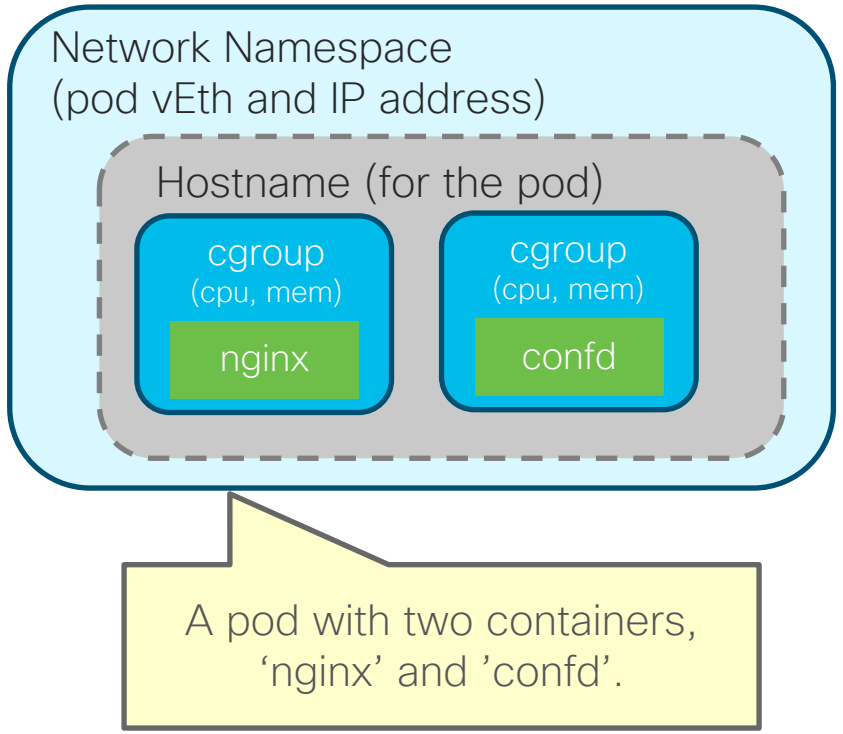
- K8s native iptables/ipvs proxy
- Implements K8s services by programming rules on the local cluster node
- Runs as a pod on each node (DaemonSet)

CNI

- The Kubernetes networking plugin
- Provides connectivity for pods and more
- Can take over K8s services implementation from kube-proxy
- Runs as a binary on each node called by kubelet

Kubernetes Pods = unit of deployment

- A *Pod* is a group of one or more containers with **shared storage and network**
- A pod models an application-specific "logical host" – an analogue to physical or **virtual machine**.
- Containers within a pod share an IP address and port space, and can find each other via ***localhost***



Kubernetes Service types

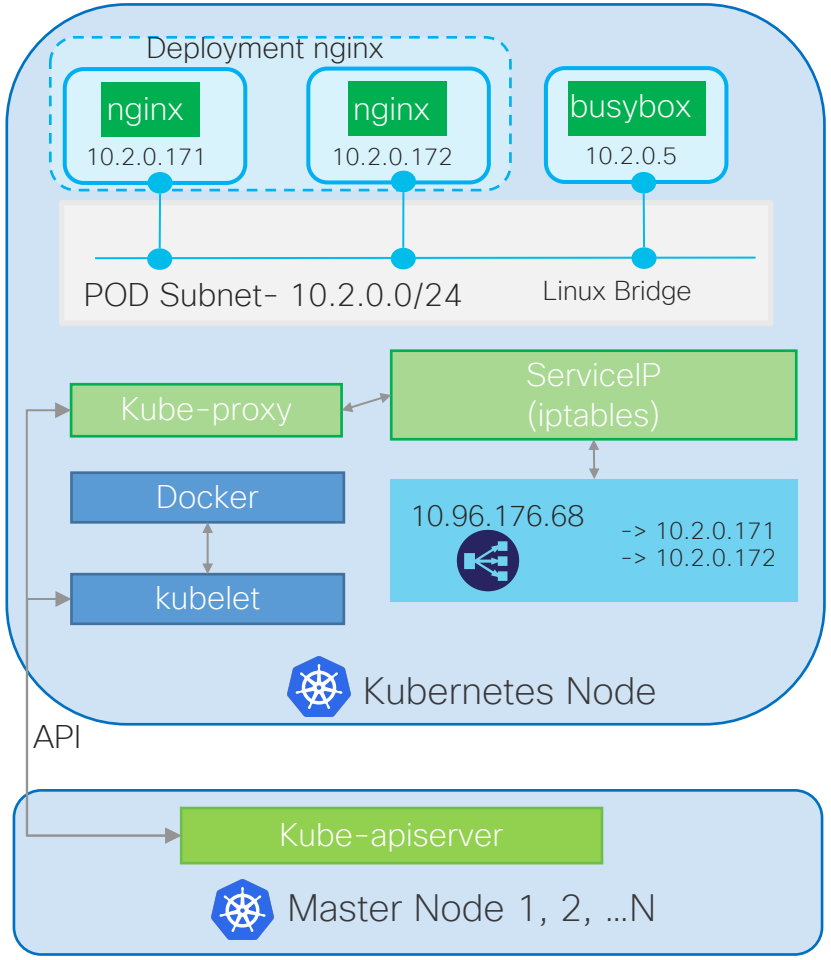
- There are currently four types of Kubernetes Services:
  - ClusterIP – exposes a service **internally**, using a virtual IP address that is permanent but only visible in the cluster (for POD to POD conversations)
  - NodePort – exposes a service **externally** by mapping the service port to a port on the Node IP.
  - LoadBalancer – uses a cloud provider to expose a service **externally** via a dedicated external IP address (statically or dynamically assigned)
  - ExternalName – used to reference endpoints that are external to the cluster.

Kubernetes Service implementation

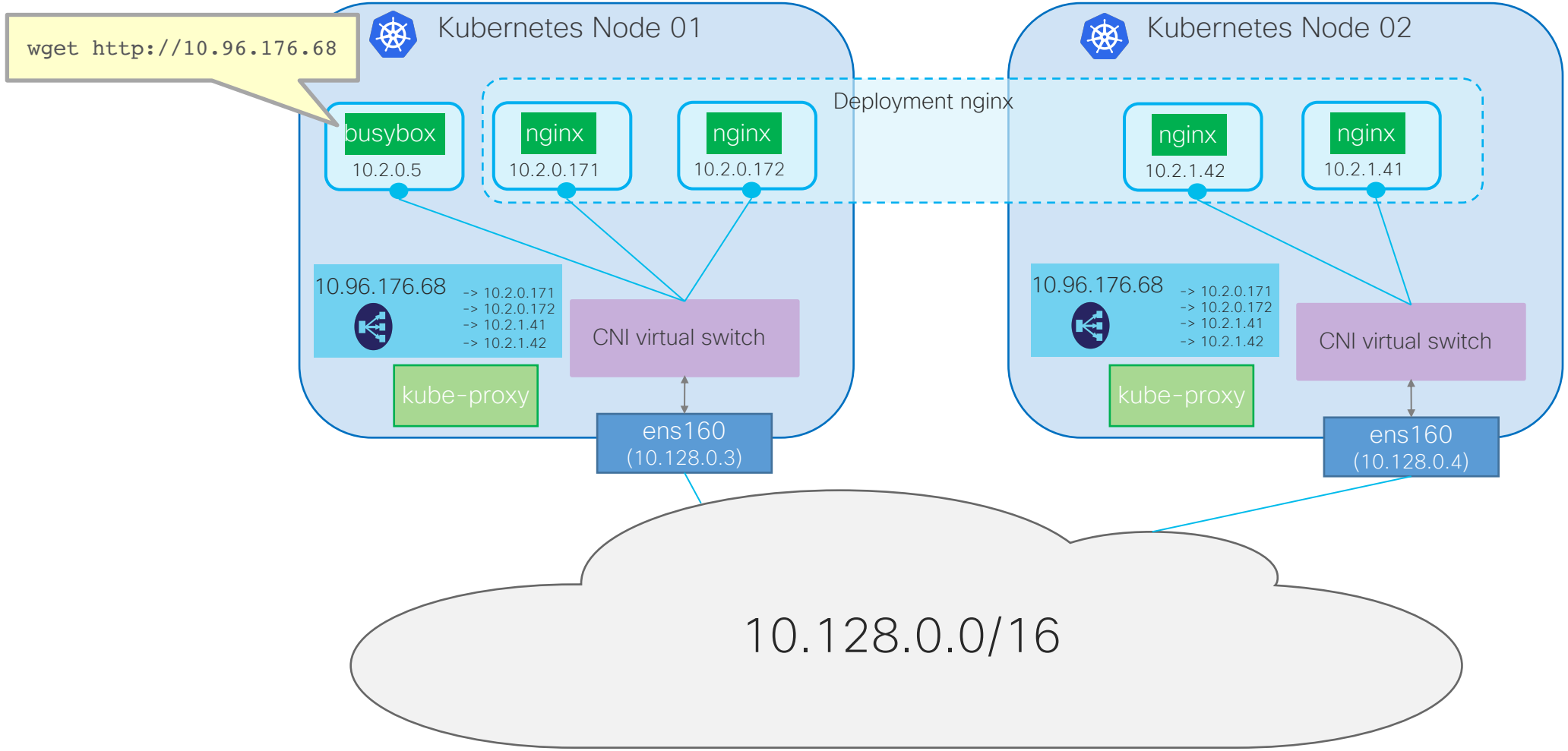
- ClusterIP and NodePort:
  - By default implemented by **kube-proxy**
  - However, some CNIs will take over ClusterIP
  - NodePort typically stays in iptables since traffic lands on the node IP
- LoadBalancer:
  - By default implemented outside of Kubernetes and provisioned by kube-cloud-controller
  - Deviations: In-cluster (quazi-)LoadBalancer (MetalLB) or bundled with CNI

ClusterIP – EastWest Load Balancing

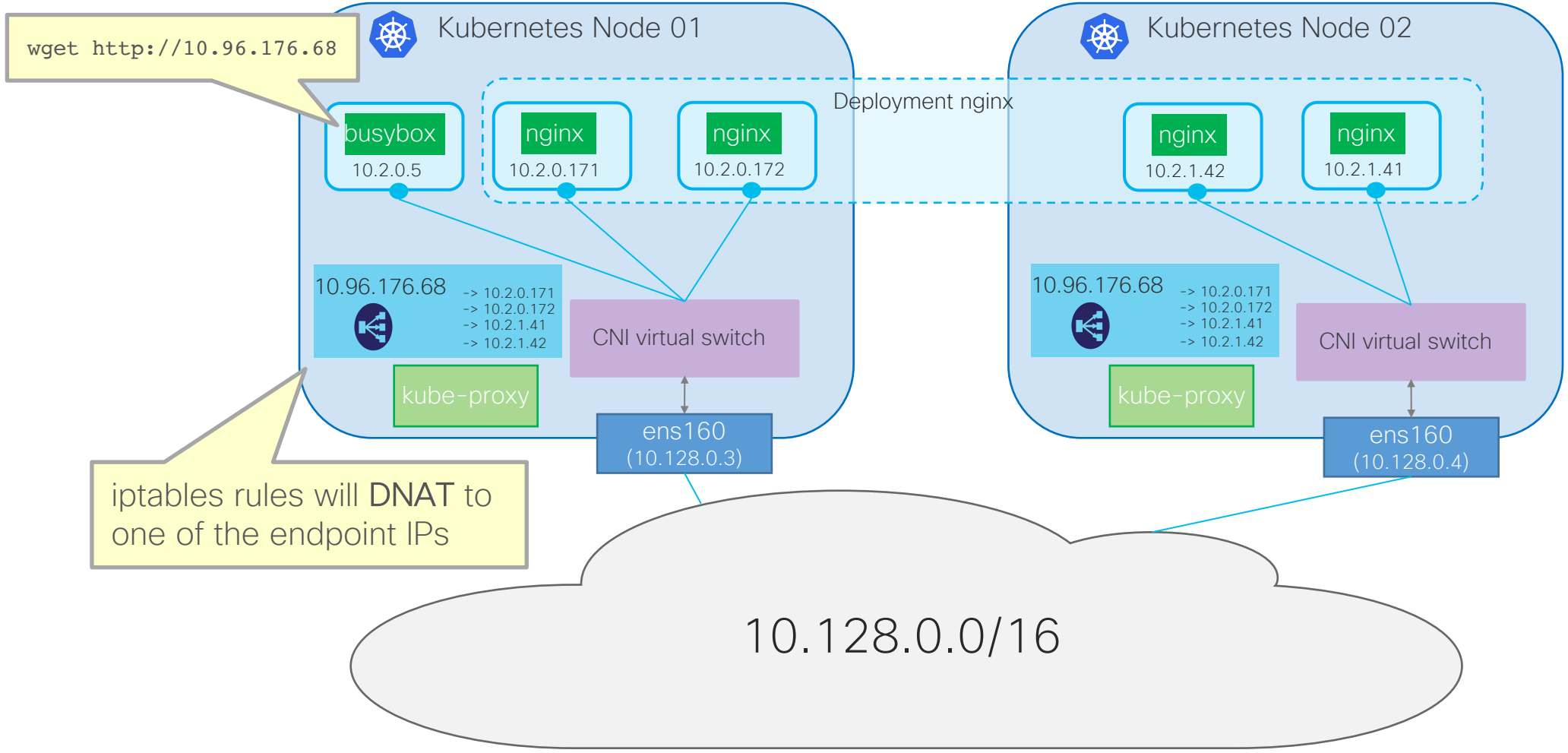
- Kubernetes assigns a persistent **virtual IP** to the service (can be dynamically or statically assigned)
- PODs can communicate to the service ClusterIP which in turn load balances and **NATs** to the POD IP addresses.
- By default this is implemented by kube-proxy using **iptables** or lately **IPVS**



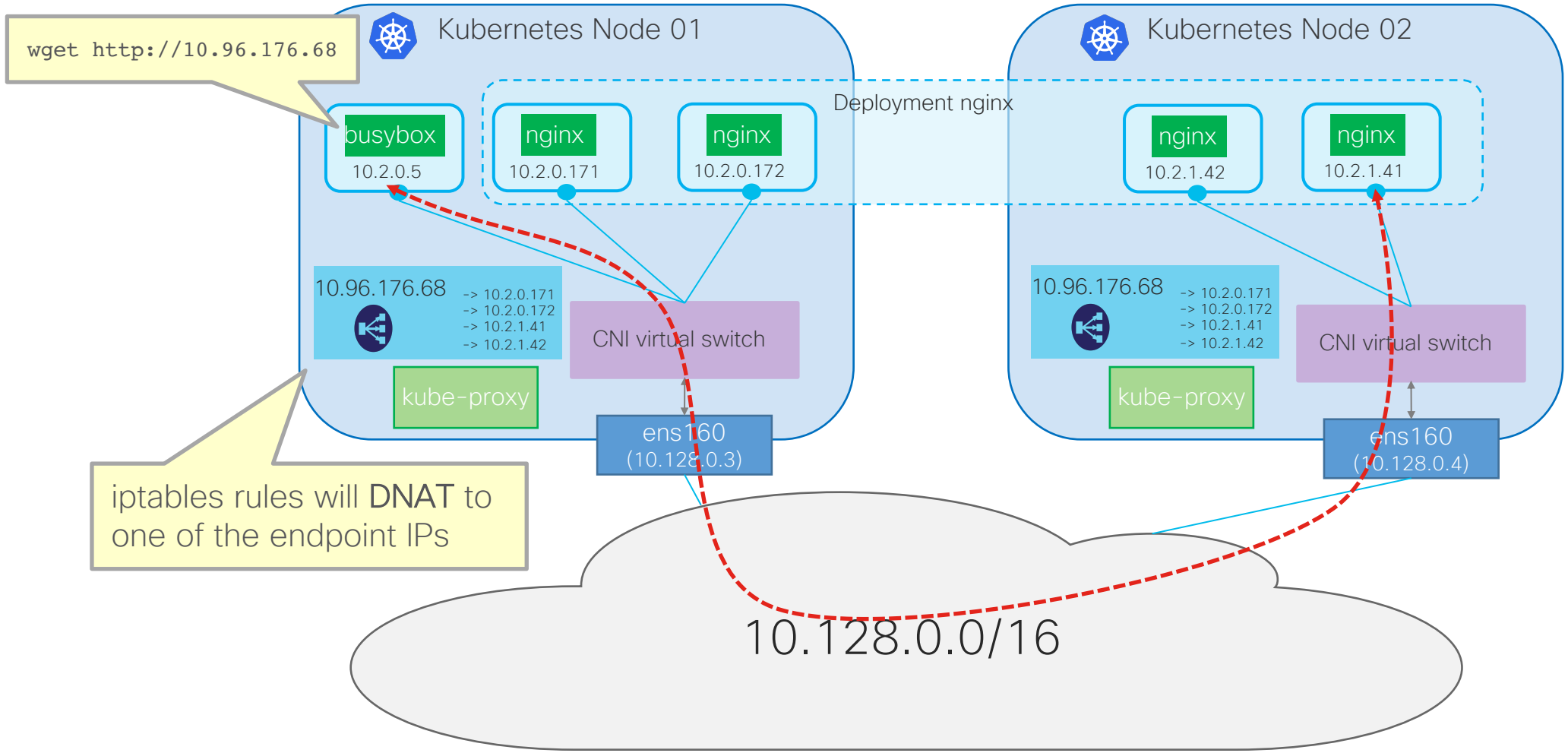
Service ClusterIP becomes a distributed L4 load balancer



Service ClusterIP becomes a distributed L4 load balancer

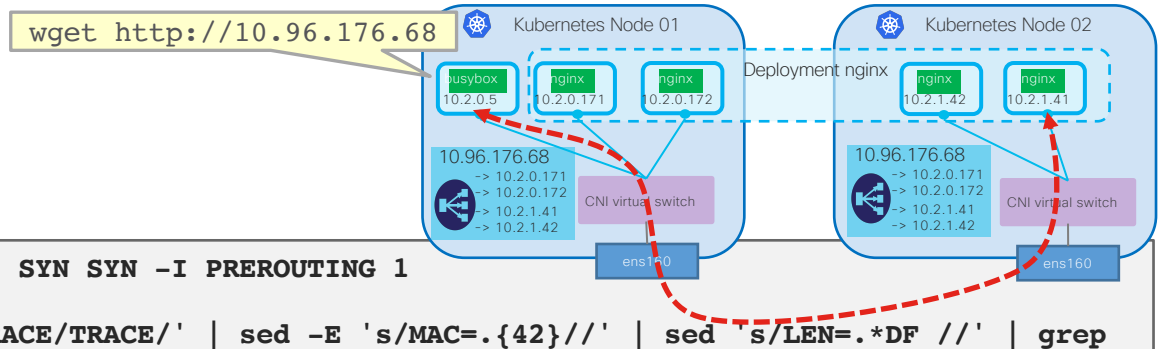


Service ClusterIP becomes a distributed L4 load balancer



Verifying ClusterIP packet processing

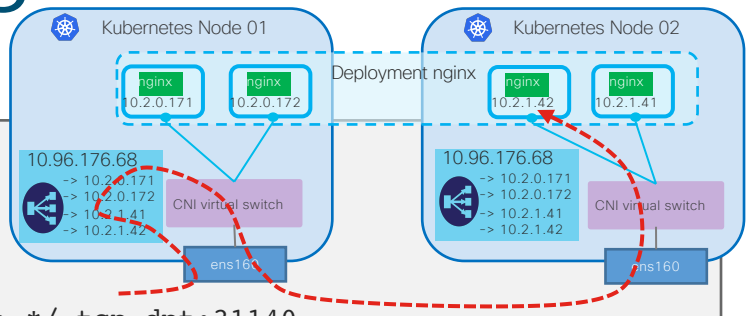
- The node that runs the pod accessing the ClusterIP is doing all the heavy lifting
- Two distinct cases:
  - Source and Destination pods are on the same node
  - Source and Destination pods are on different nodes:




```
# iptables -t raw -j TRACE -d 10.96.176.68 -p tcp --dport 8080 --tcp-flags SYN SYN -I PREROUTING 1
# journalctl -f > trace &
# grep TRACE trace | sed 's/IN=/\; IN=/' | column -t -s $';' | sed 's/.*TRACE/TRACE/' | sed -E 's/MAC=.{42}//' | sed 's/LEN=.*DF //' | grep
-v cali- | cut -c -140
TRACE: raw:PREROUTING:rule:2                    IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: raw:PREROUTING:policy:3                  IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: mangle:PREROUTING:rule:1                 IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: mangle:PREROUTING:policy:2               IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: nat:PREROUTING:rule:1                    IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: nat:PREROUTING:rule:2                    IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: nat:KUBE-SERVICES:rule:14                IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: nat:KUBE-SVC-TWVLBX4WCEZSIVWL:rule:1     IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: nat:KUBE-SEP-K2D2A2RTG5KDZHX4:rule:2     IN=cali6867d3c3750 OUT= SRC=10.2.0.5 DST=10.96.176.68 PROTO=TCP SPT=40185 DPT=8080 SEQ=
TRACE: mangle:FORWARD:policy:1                  IN=cali6867d3c3750 OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=
TRACE: filter:FORWARD:rule:1                    IN=cali6867d3c3750 OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=
TRACE: mangle:POSTROUTING:policy:1              IN= OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=1078874980 ACK=
TRACE: nat:POSTROUTING:rule:1                   IN= OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=1078874980 ACK=
TRACE: nat:POSTROUTING:rule:2                   IN= OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=1078874980 ACK=
TRACE: nat:KUBE-POSTROUTING:return:2            IN= OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=1078874980 ACK=
TRACE: nat:POSTROUTING:policy:4                 IN= OUT=tunl0 SRC=10.2.0.5 DST=10.2.1.41 PROTO=TCP SPT=40185 DPT=80 SEQ=1078874980 ACK=

# iptables -t nat -nvL POSTROUTING | grep Chain
Chain POSTROUTING (policy ACCEPT 9 packets, 540 bytes)
```

NodePort? Practically the same thing!

- All the action is on the node where traffic lands first




```
$ kubectl get svc nginx
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)         AGE
nginx        NodePort    10.96.176.220    <none>        80:31140/TCP    4m56s

# iptables -t nat -nvL | grep 31140
    0     0 KUBE-MARK-MASQ  tcp  --  *      *       0.0.0.0/0       0.0.0.0/0        /* default/nginx: */ tcp dpt:31140
    0     0 KUBE-SVC-TFDSBUMN6I7R7DUY  tcp  --  *      *       0.0.0.0/0       0.0.0.0/0        /* default/nginx: */ tcp dpt:31140

# iptables -t raw -j TRACE -p tcp --dport 31140 --tcp-flags SYN SYN -I PREROUTING 1

<snip>
TRACE: nat:KUBE-SERVICES:rule:26                IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: nat:KUBE-NODEPORTS:rule:3                IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: nat:KUBE-MARK-MASQ:rule:1                IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: nat:KUBE-MARK-MASQ:return:2              IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: nat:KUBE-NODEPORTS:rule:4                IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: nat:KUBE-SVC-TFDSBUMN6I7R7DUY:rule:4     IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: nat:KUBE-SEP-SQNRY57BXQNVGBT4:rule:2     IN=ens192 OUT= SRC=192.168.1.1 DST=10.128.0.3 PROTO=TCP SPT=41618 DPT=31140 SEQ=24907083
TRACE: mangle:FORWARD:policy:1                  IN=ens192 OUT=tunl0 SRC=192.168.1.1 DST=10.2.1.41 PROTO=TCP SPT=41618 DPT=80 SEQ=2490708
...
TRACE: nat:POSTROUTING:rule:1                   IN= OUT=tunl0 SRC=192.168.1.1 DST=10.2.1.41 PROTO=TCP SPT=41618 DPT=80 SEQ=2490708399 AC
TRACE: nat:POSTROUTING:rule:2                   IN= OUT=tunl0 SRC=192.168.1.1 DST=10.2.1.41 PROTO=TCP SPT=41618 DPT=80 SEQ=2490708399 AC
TRACE: nat:KUBE-POSTROUTING:rule:1              IN= OUT=tunl0 SRC=192.168.1.1 DST=10.2.1.41 PROTO=TCP SPT=41618 DPT=80 SEQ=2490708399 AC
</snip>


# iptables -t nat -nvL KUBE-POSTROUTING
Chain KUBE-POSTROUTING (1 references)
 pkts bytes target     prot opt in     out     source       destination
    0     0 MASQUERADE  all  --  *      *       0.0.0.0/0    0.0.0.0/0    /* kubernetes service traffic requiring SNAT */ mark match 0x4000/0x4000
```

```
# ip add show tunl0 | grep inet
    inet 10.2.26.128/32 brd 10.2.26.128 scope global tunl0
# tcpdump -ennali tunl0
15:17:07.758001 ip: 10.2.26.128.41618 > 10.2.2.41.80: Flags [S],
```

LoadBalancer

- The action is mostly outside of the Kubernetes cluster
- Main purpose is to evenly distribute traffic between cluster nodes
- Once the traffic lands, it is the same story as with ClusterIP but this time with externally routable IPs
- There are many different solutions and implementations, each has it own inner workings and specifics

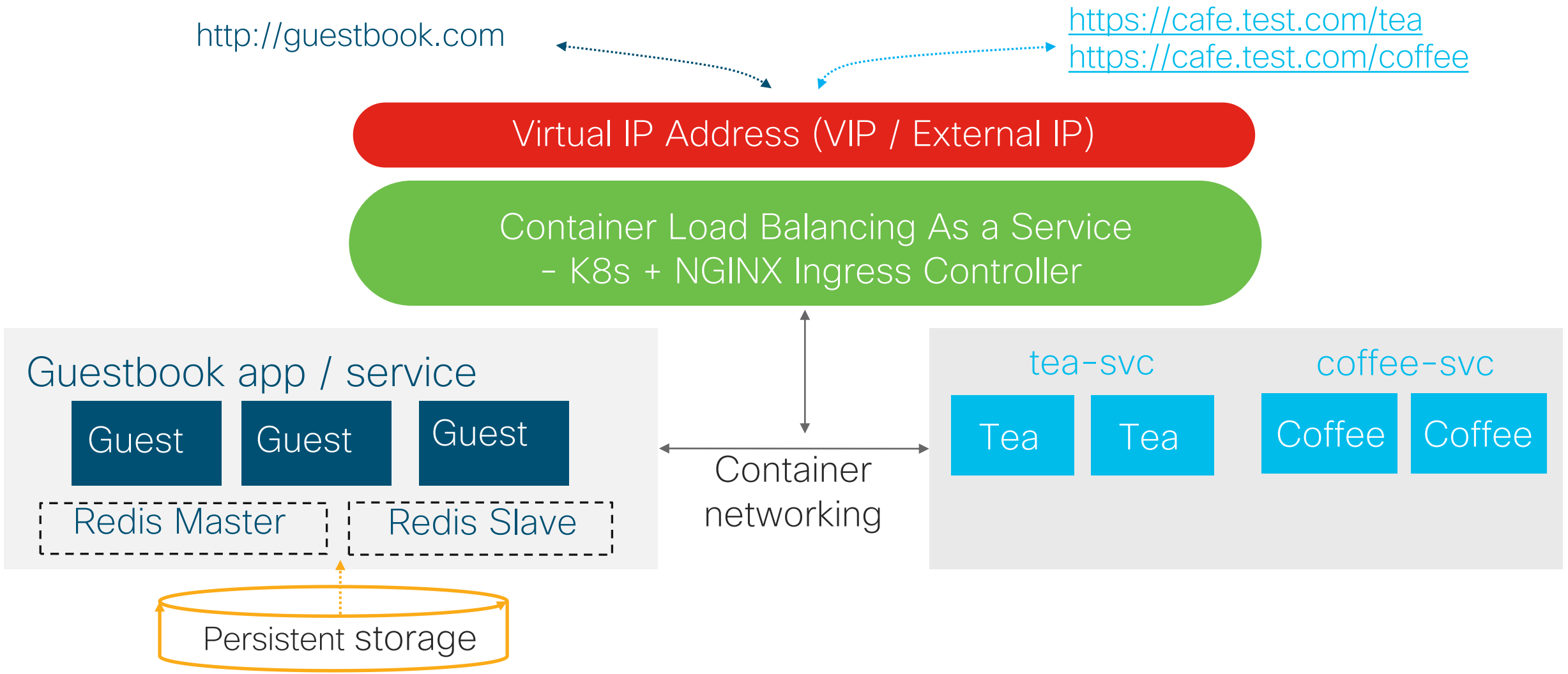
Ingress Controller

- LoadBalancers burn a lot of IPs (one per exposed object)
- An ingress controller is another method of exposing a cluster service to the outside world that will not require a Load Balancer for each service.
- These are L7 HTTP(S) load balancers that can offer also SSL termination, name-based virtual hosting, etc.
- The Ingress Controller is deployed as a POD that can run on one or more nodes on the cluster
- Orchestrated and configured via Kubernetes standard API object "ingress"
- There are multiple options: NGINX, HAProxy, Envoy, Traefik, etc.

NGINX Ingress Controller

- Based on NGINX webserver/proxy/load-balancer
- Two parts: IC + NGINX (same pod/container)
- Ingress Controller (control-plane):
  - Monitors for additions and changes of ingress objects from the kube-apiserver
  - Generates a config file for the NGINX based on the ingress object config and state
  - Manages the NGINX process
- NGINX (data-plane):
  - Does all the work by rewriting HTTP headers, etc.
  - Can also terminate SSL
- Needs to be exposed via LoadBalancer or NodePort

NGINX Ingress topology



Container Network Interfaces (CNIs)

CNI: solving the Kubernetes Network puzzle

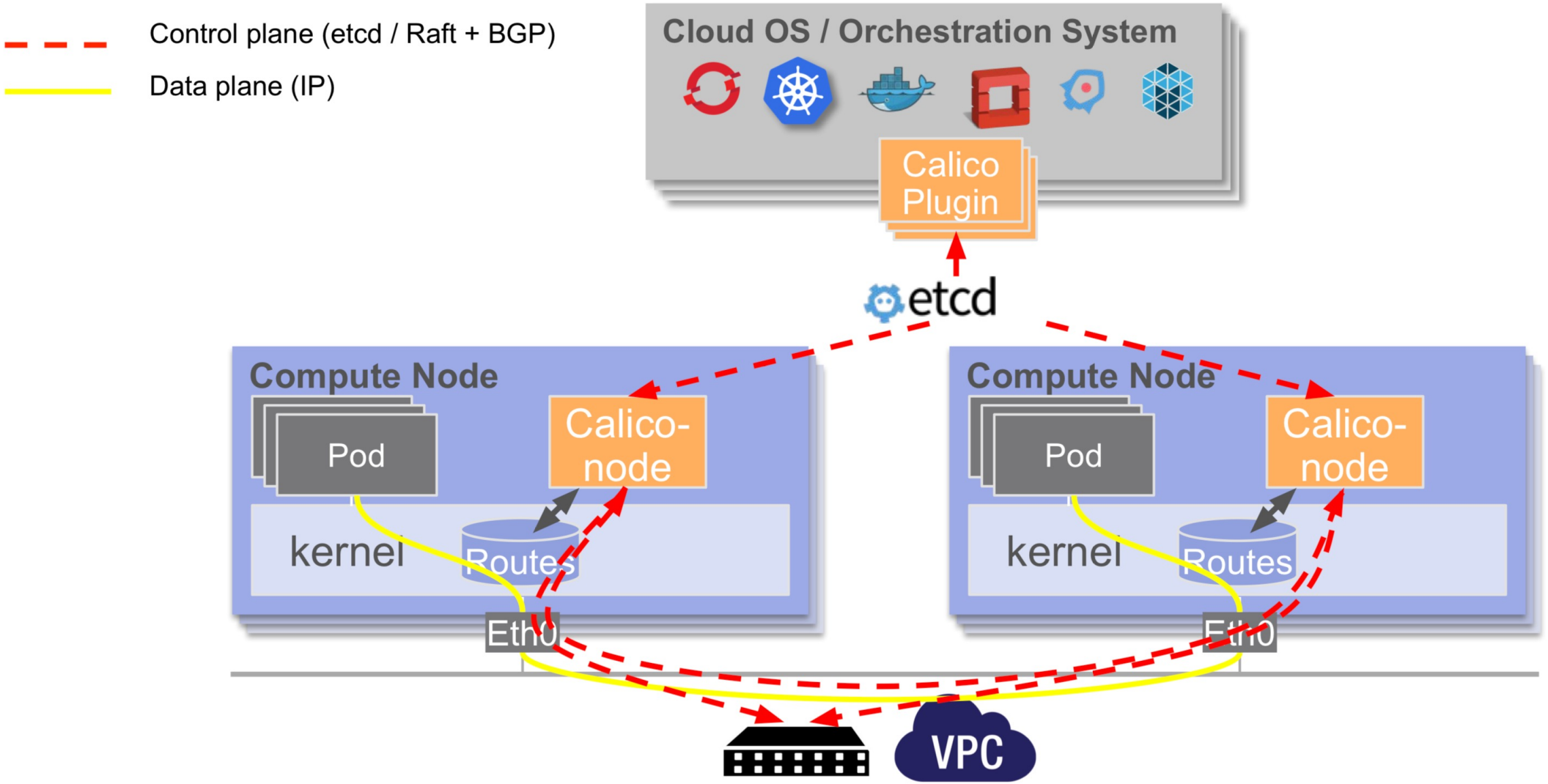
- Kubernetes makes individual Docker nodes come together
- CNI makes them play together nicely
- The main issue that CNI solves is how to bridge the networking gap between Docker nodes
- All CNIs do it differently, but all of them play with the following functionality:
  - Transporting the packets between the node, usually via some sort of overlay
  - Employ a data-path
  - Knowing which pod is running where
  - Handling IPAM
  - Implementing a layer of security within the cluster (optional)

Calico CNI properties

https://docs.projectcalico.org/v3.10/getting-started/kubernetes/

- One of the most popular CNIs
- Leverages Linux's built-in efficient network stack for data-path
  - Linux IP routing
  - iptables (for security policy)
- Implements lightweight IP-in-IP tunneling between the nodes
- Uses BGP daemons in the control-plane for distributing pod reachability info

Calico Architecture



Calico-node pod

3 components:

- **Felix**
  - Programs routes and iptables on the host to provide desired connectivity
  - Programs interface information to the kernel for outgoing endpoint traffic
  - Instructs the host to respond to ARPs for workloads with the MAC address of the host.
- **BIRD**
  - BGP client that is used to exchange routing information between hosts
- **confd**
  - for managing and applying BIRD config

Calico data plane plumbing