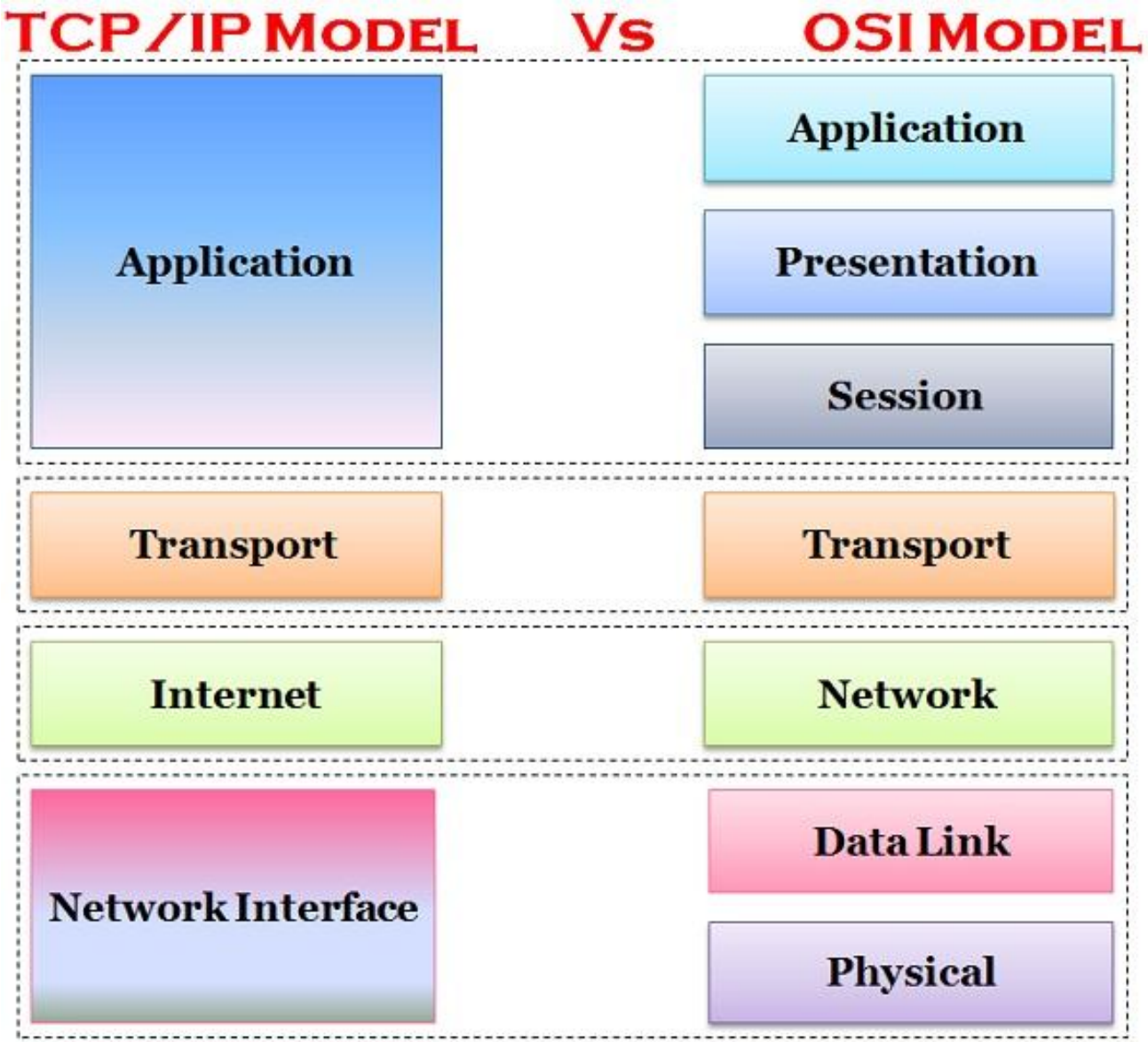
Docker

Networking

Docker Networking

Introduction to networking

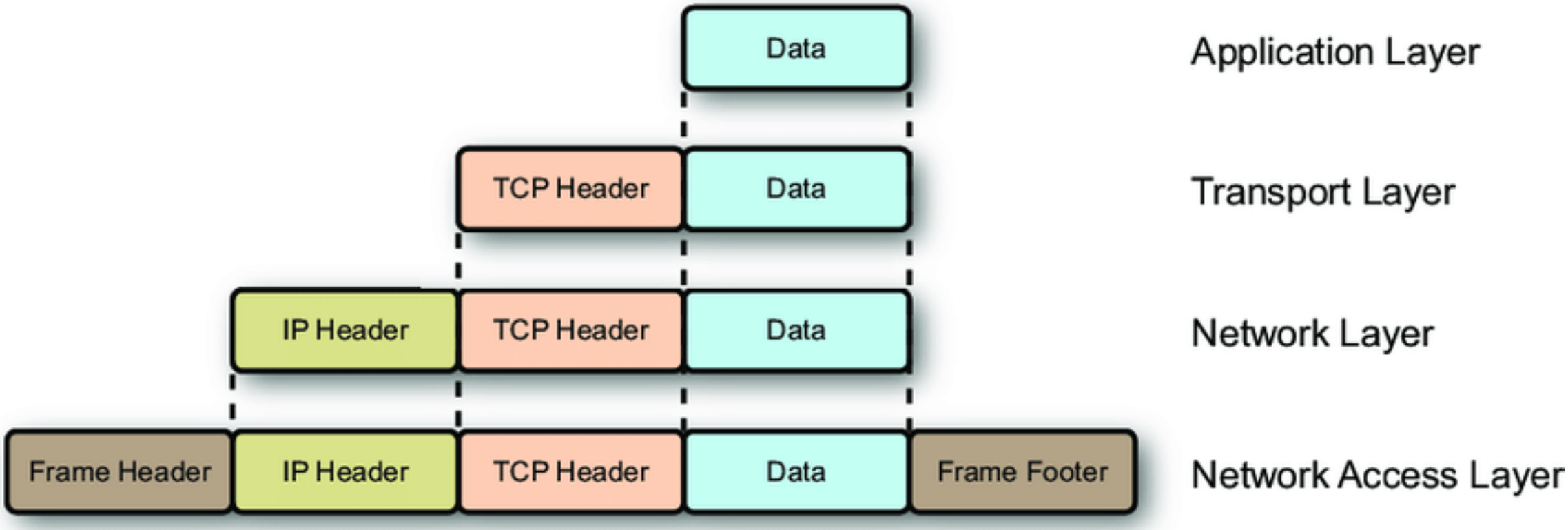
Networking layered models



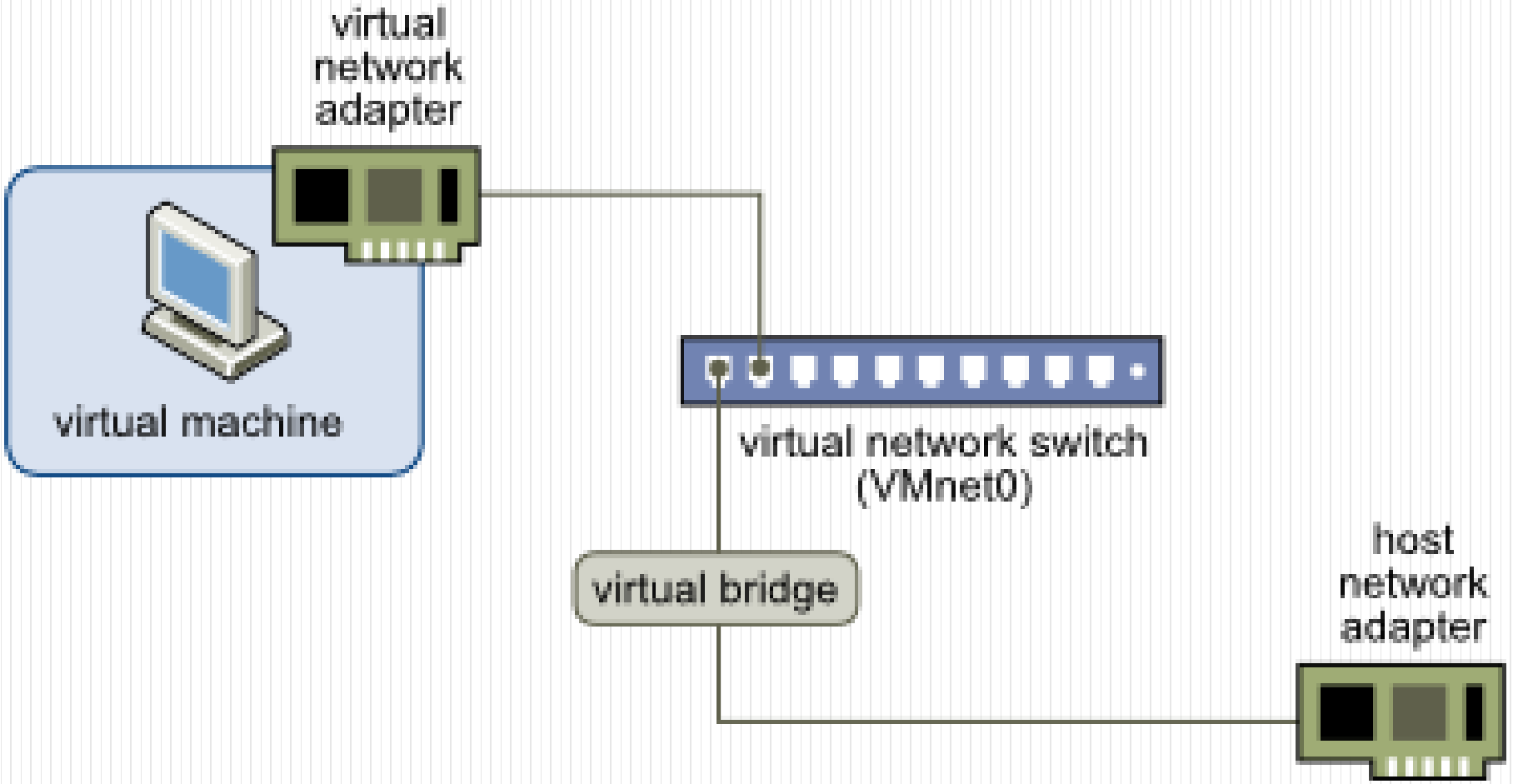
Addressing

- Protocol signature, Layer 7 – define the application protocol.
- Source and destination ports, Layer 4 – define the process of the application.
- Source and destination IPs , Layer 3 – define hosts.
- Sorce and destination MACs , Layer 2 – define network interfaces.

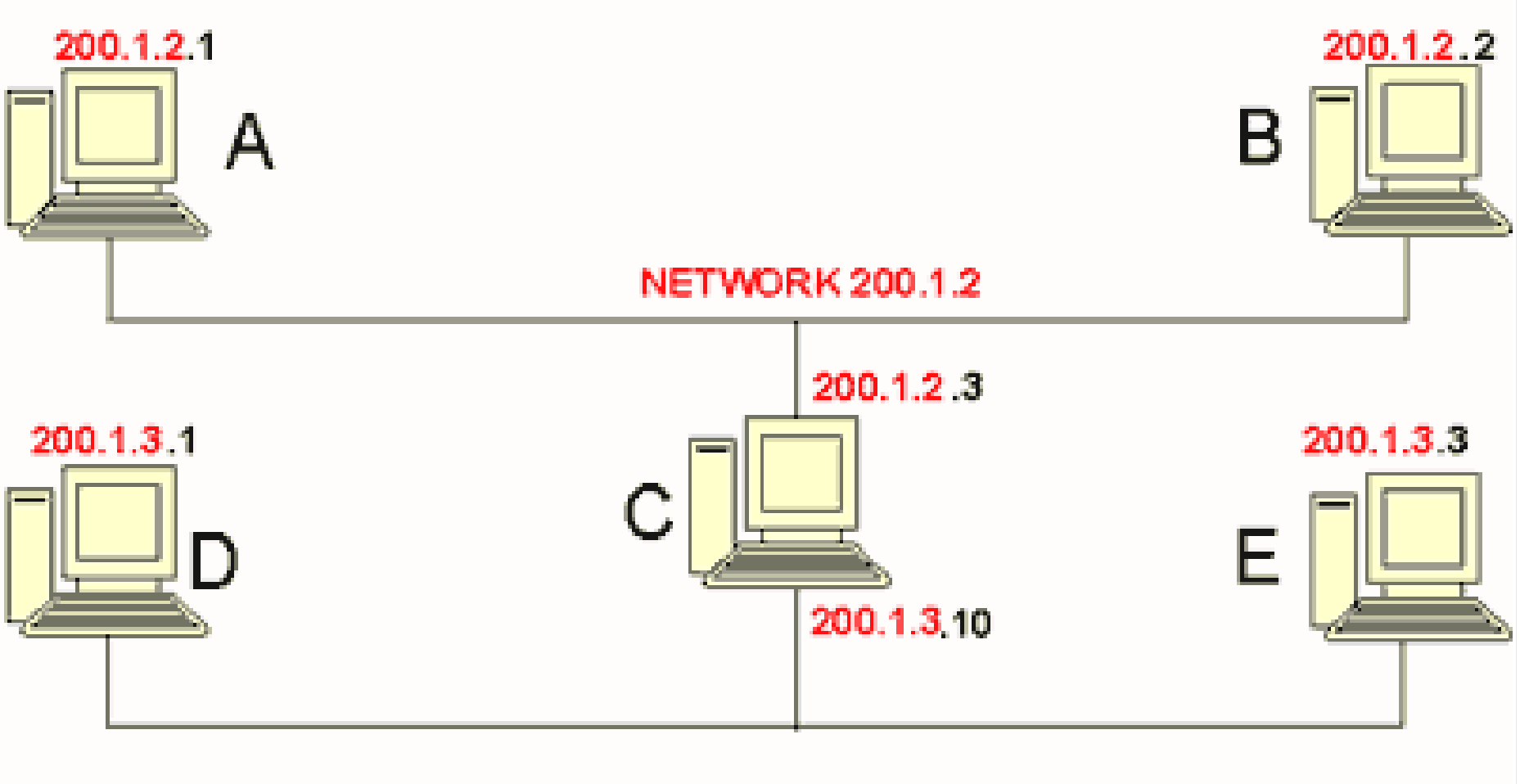
Encapsulation



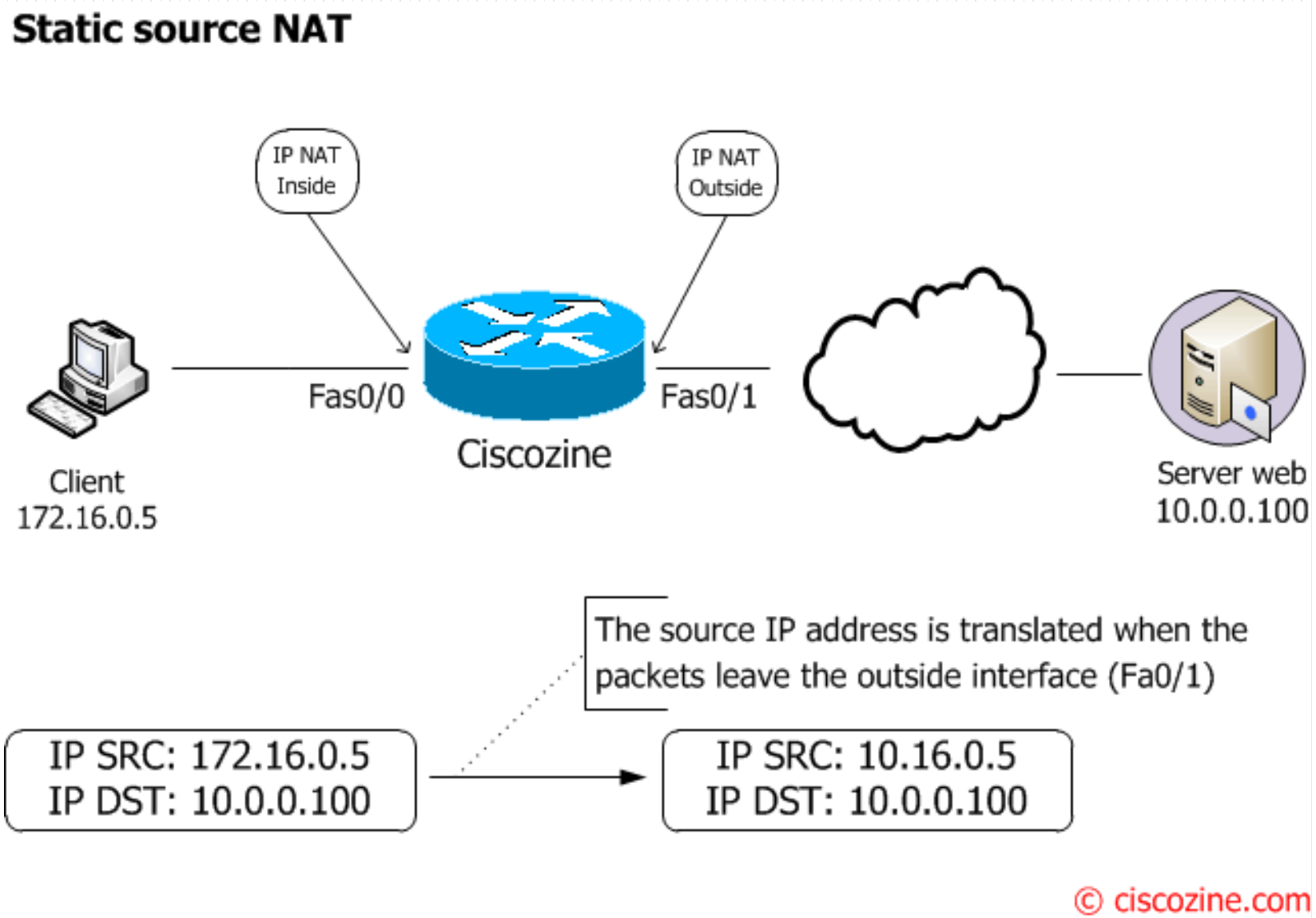
Bridging


virtual
network
adapter
virtual machine
virtual network switch
(VMnet0)
virtual bridge
host
network
adapter

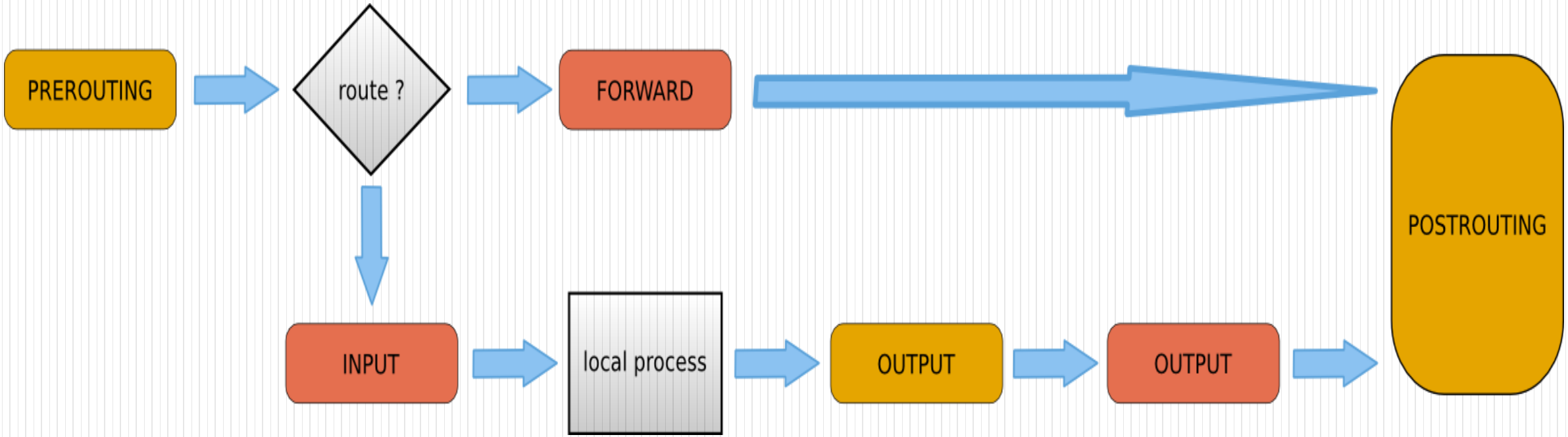
Routing


200.1.2.1
A
200.1.2.2
B
NETWORK 200.1.2
200.1.2.3
200.1.3.1
D
C
200.1.3.3
E
200.1.3.10

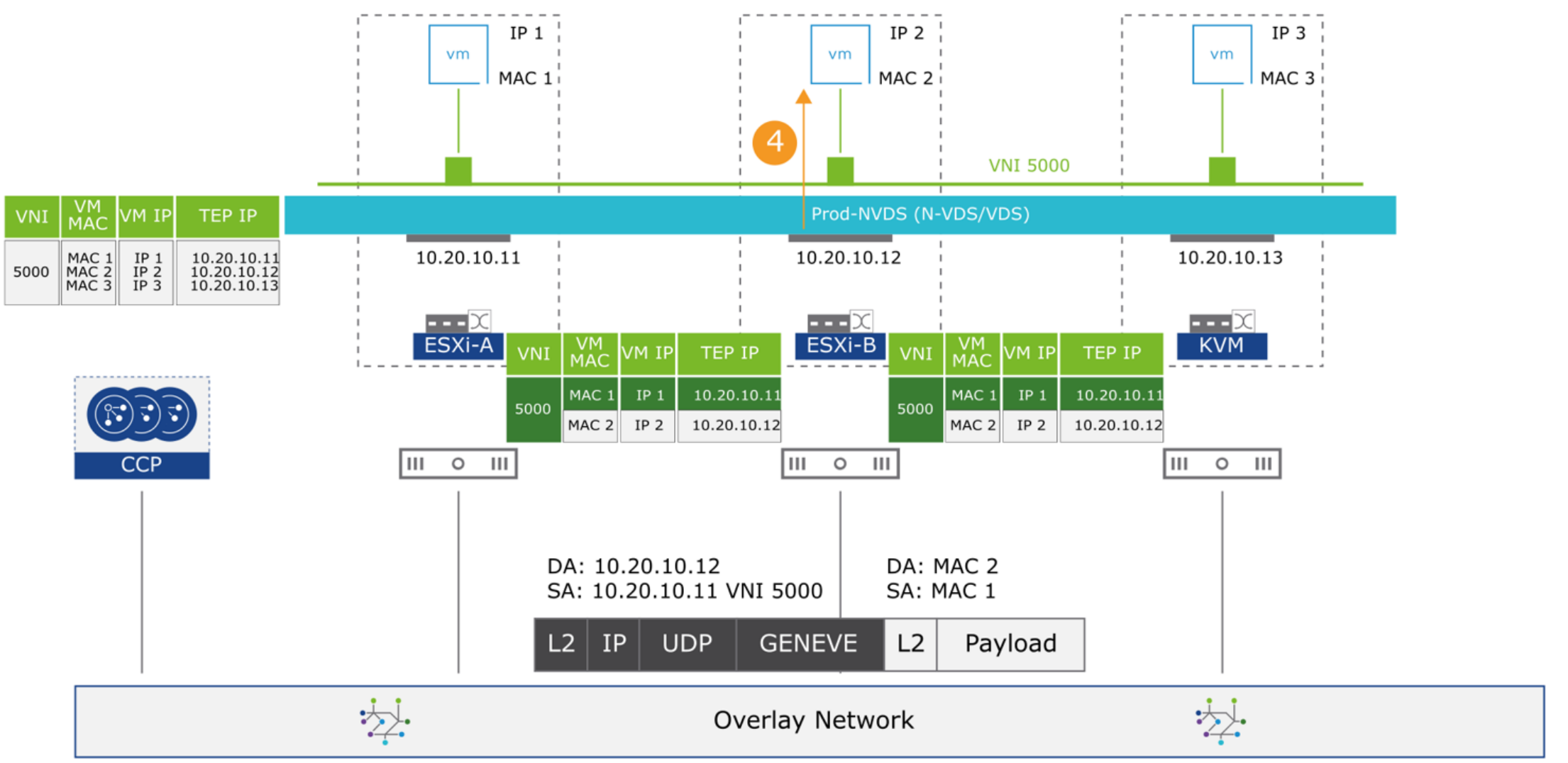
Network Address Translation



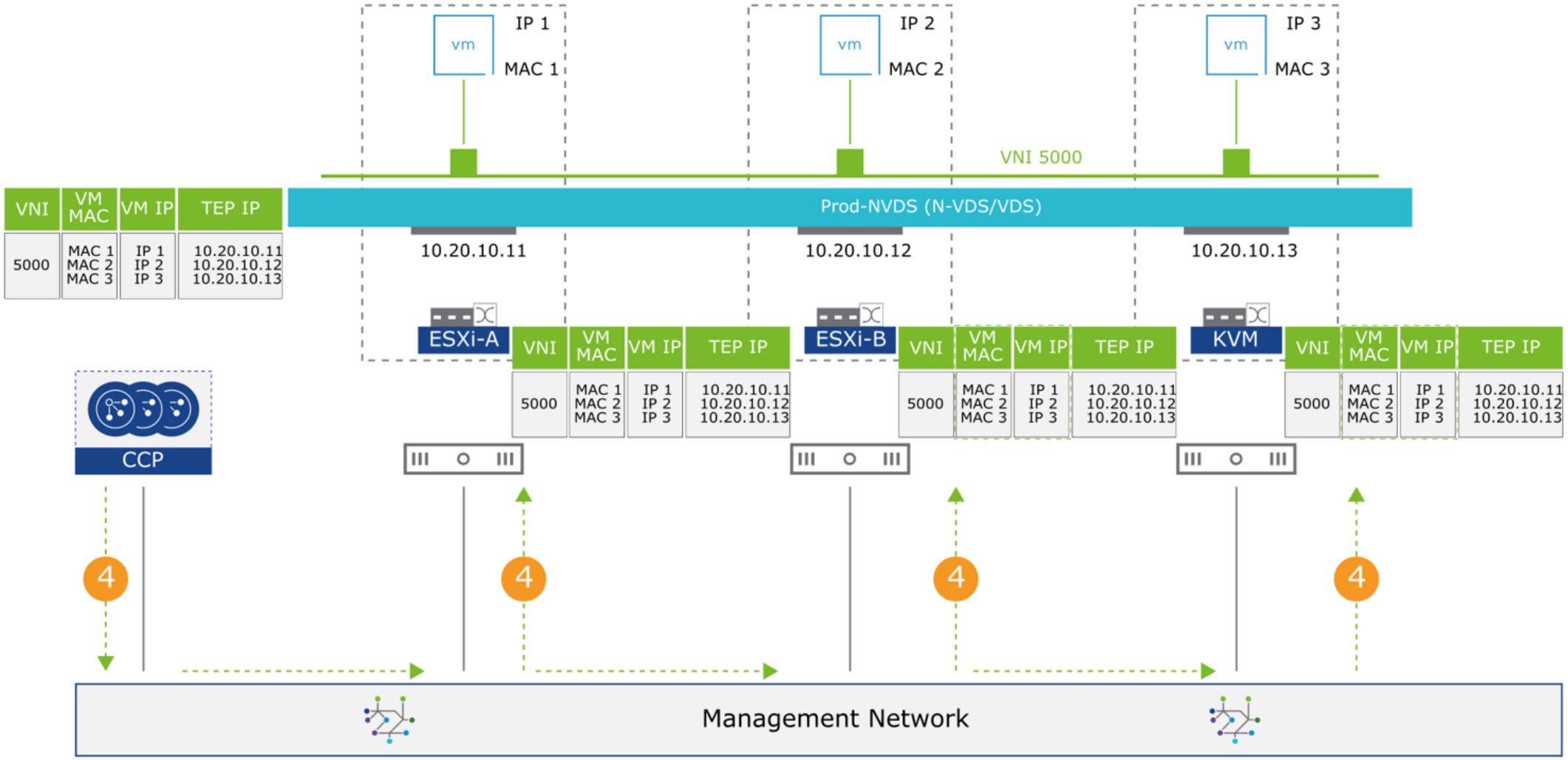
IP Filtering



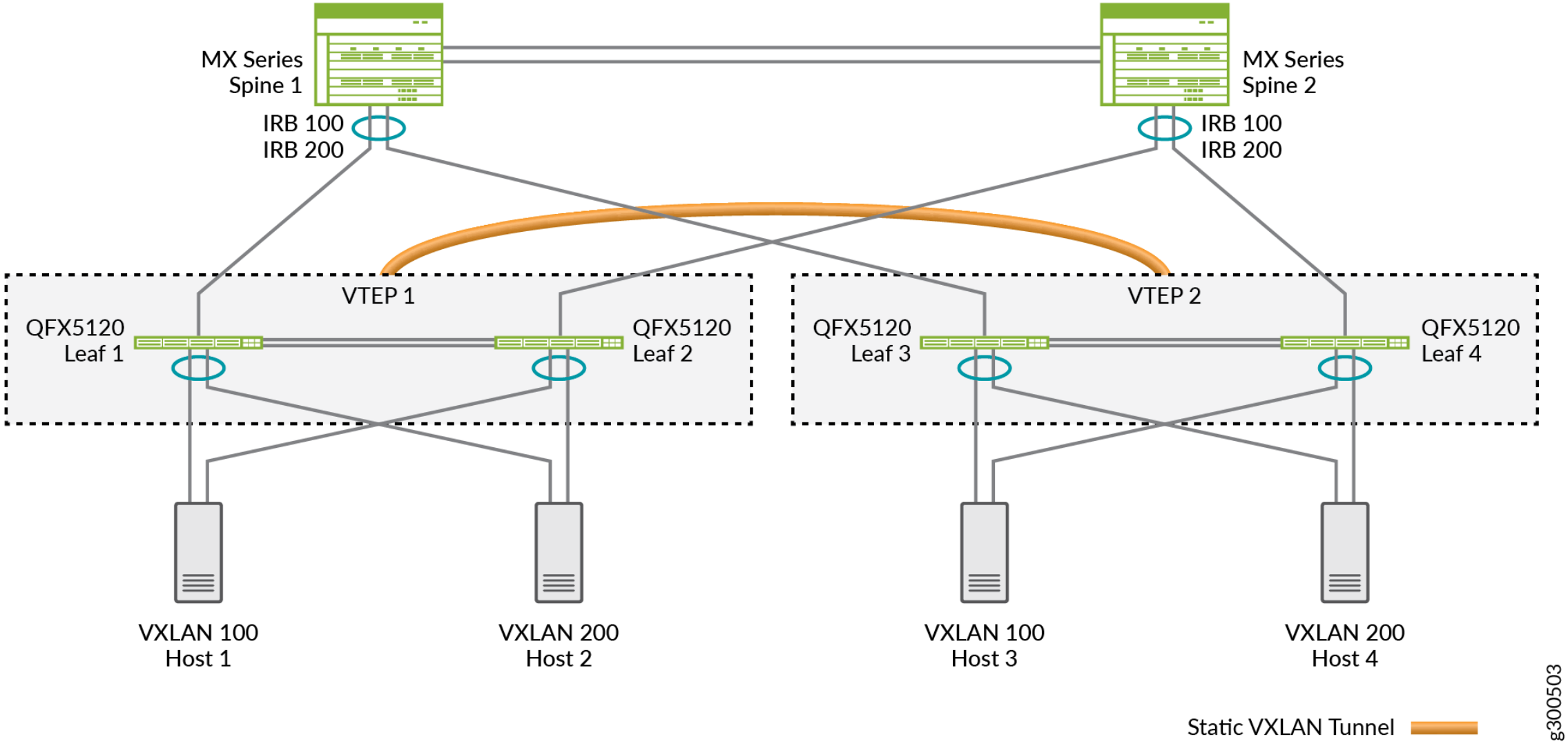
VXLan / Geneva



VXLan / Geneva



VXLan / Geneva



Docker Networking

Overview

Network driver summary

- Portability

  How do I guarantee maximum portability across diverse network environments while taking advantage of unique network characteristics?

- Service Discovery

  How do I know where services are living as they are scaled up and down?

- Load Balancing

  How do I share load across services as services themselves are brought up and scaled?

- Security

  How do I segment to prevent the wrong containers from accessing each other?

  How do I guarantee that a container with application and cluster control traffic is secure?

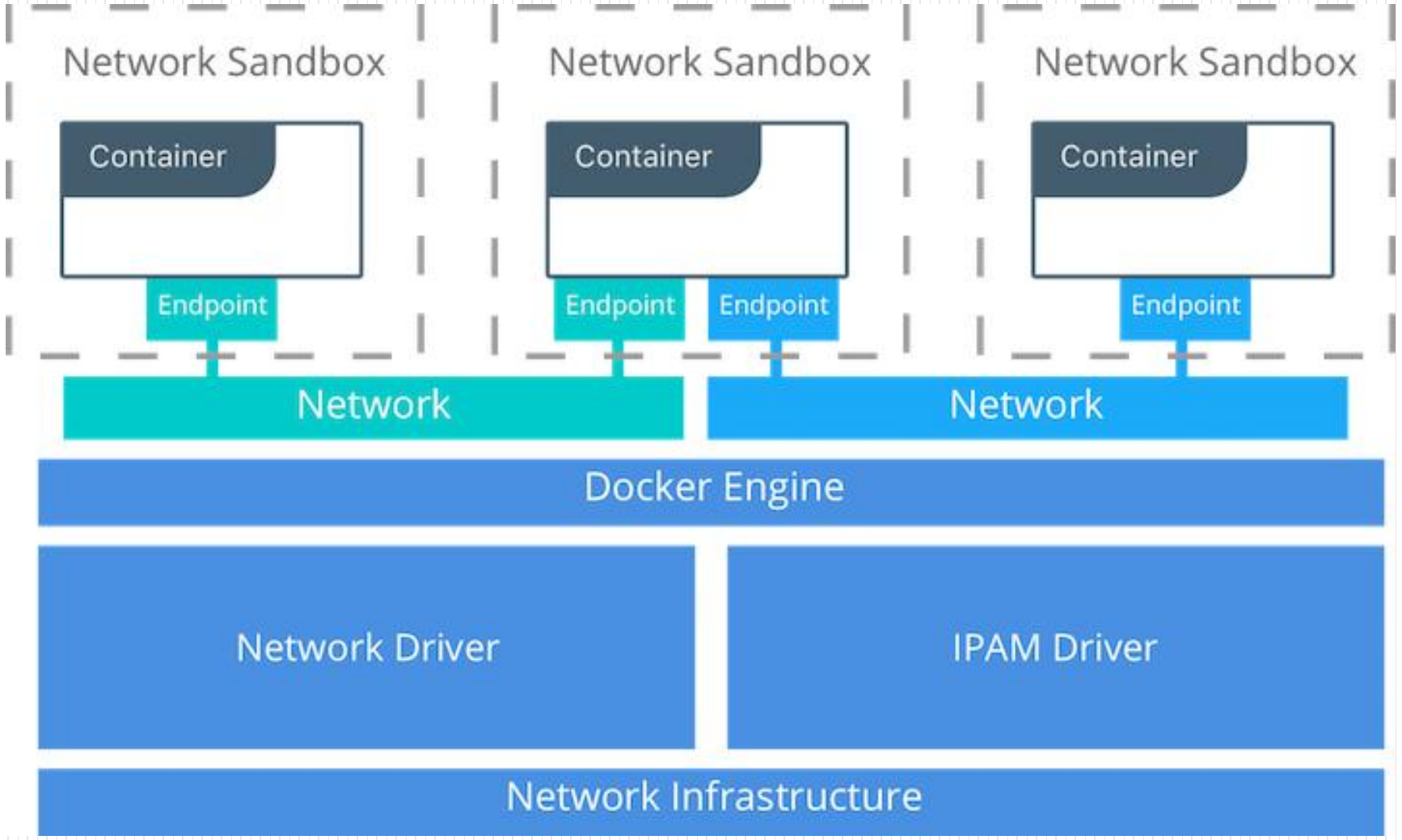
- Performance

  How do I provide advanced network services while minimizing latency and maximizing bandwidth?

- Scalability

  How do I ensure that none of these characteristics are sacrificed when scaling applications across many hosts?

The Container Networking Model

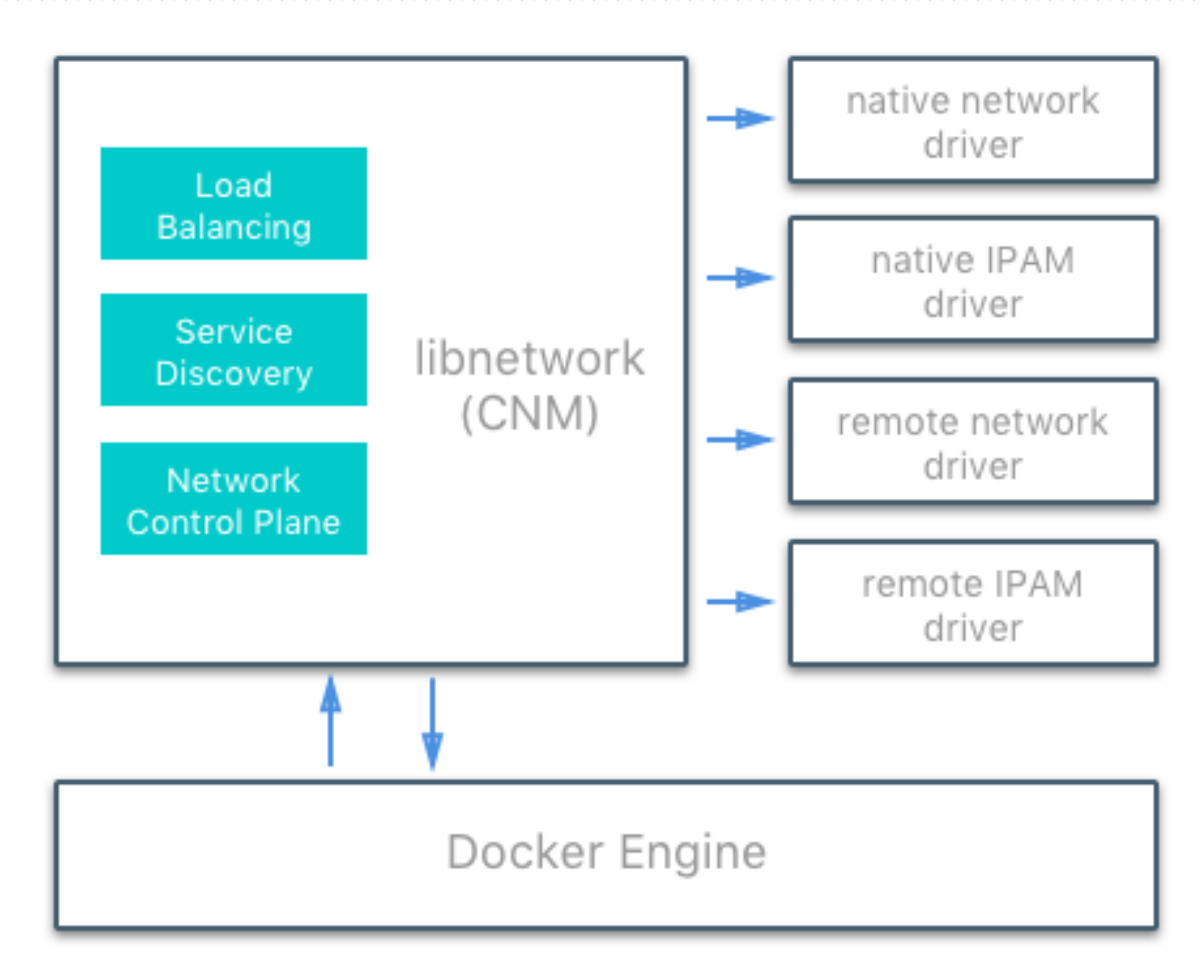


CNM Constructs

- There are several high-level constructs in the CNM. They are all OS and infrastructure agnostic so that applications can have a uniform experience no matter the infrastructure stack.
- Sandbox — A Sandbox contains the configuration of a container's network stack. This includes management of the container's interfaces, routing table, and DNS settings. An implementation of a Sandbox could be a Linux Network Namespace, a FreeBSD Jail, or other similar concept. A Sandbox may contain many endpoints from multiple networks.
- Endpoint — An Endpoint joins a Sandbox to a Network. The Endpoint construct exists so the actual connection to the network can be abstracted away from the application. This helps maintain portability so that a service can use different types of network drivers without being concerned with how it's connected to that network.
- Network — The CNM does not specify a Network in terms of the OSI model. An implementation of a Network could be a Linux bridge, a VLAN, etc. A Network is a collection of endpoints that have connectivity between them. Endpoints that are not connected to a network do not have connectivity on a network.

Network driver summary

- The following network drivers exist:
- Network Drivers
  - Native Network Drivers
  - Remote Network Drivers
- IPAM Drivers



Network driver summary

- User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.
- Host networks are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- Overlay networks are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- Macvlan networks are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- Third-party network plugins allow you to integrate Docker with specialized network stacks.

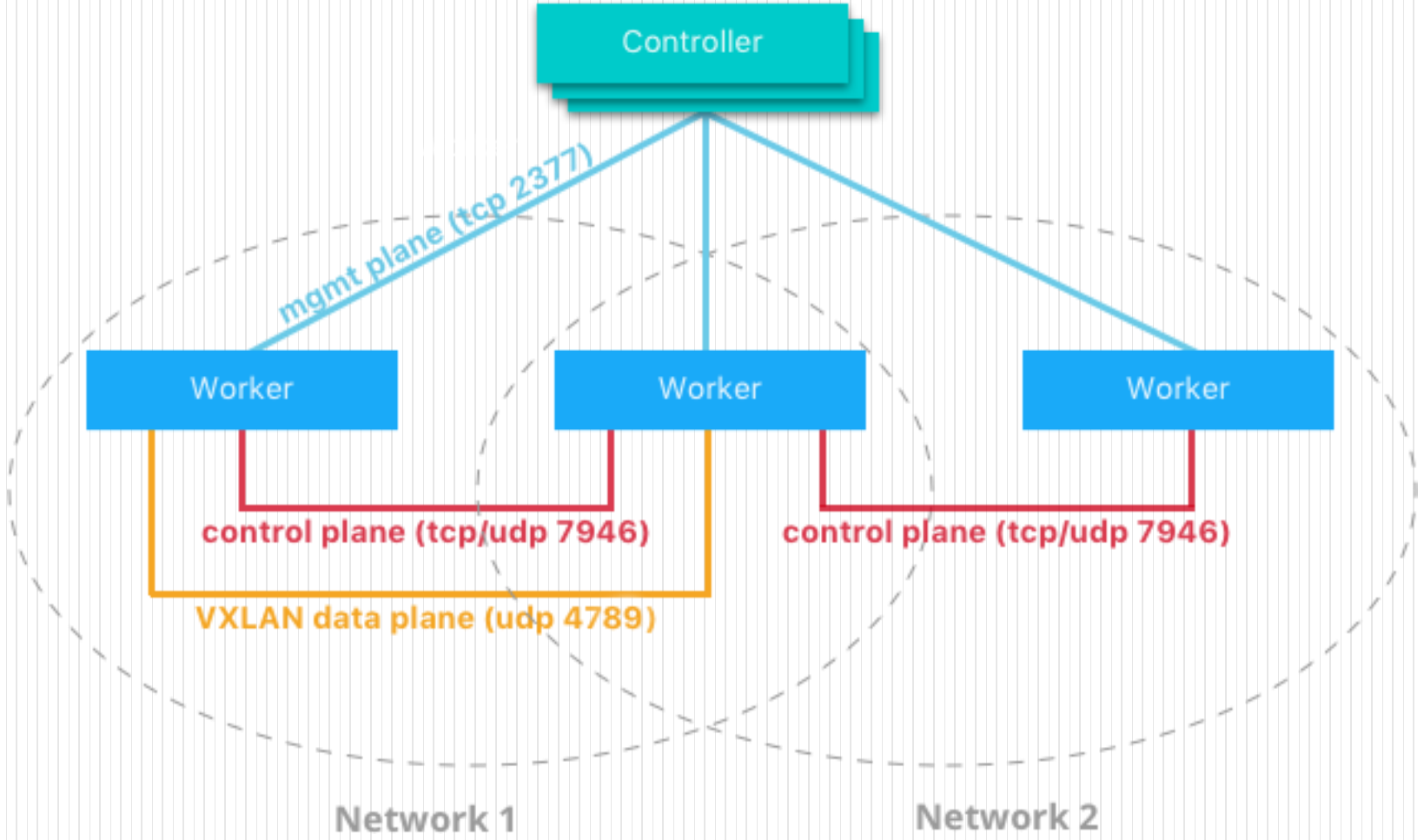
Docker EE networking features

- The following two features are only possible when using Docker EE and managing your Docker services using Universal Control Plane (UCP):

- The HTTP routing mesh allows you to share the same network IP address and port among multiple services. UCP routes the traffic to the appropriate service using the combination of hostname and port, as requested from the client.

- Session stickiness allows you to specify information in the HTTP header which UCP uses to route subsequent requests to the same service task, for applications which require stateful sessions.

Docker Network Control Plane



Use the default bridge network

- List current networks

```
docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
17e324f45964        bridge              bridge              local
6ed54d316334        host                host                local
7092879f2cc8        none                null                local
```

- The default bridge network is listed, along with host and none.

Use the default bridge network

- List current networks

```
docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
17e324f45964        bridge              bridge              local
6ed54d316334        host                host                local
7092879f2cc8        none                null                local
```

- The default bridge network is listed, along with host and none.
- Inspect the bridge network to see what containers are connected to it.

```
docker network inspect bridge
```

Virtual Ethernet Devices

- A virtual ethernet device or veth is a Linux networking interface that acts as a connecting wire between two network namespaces.
- A veth is a full duplex link that has a single interface in each namespace. Traffic in one interface is directed out the other interface.
- Docker network drivers utilize veths to provide explicit connections between namespaces when Docker networks are created.
- When a container is attached to a Docker network, one end of the veth is placed inside the container (usually seen as the ethX interface) while the other is attached to the Docker network.

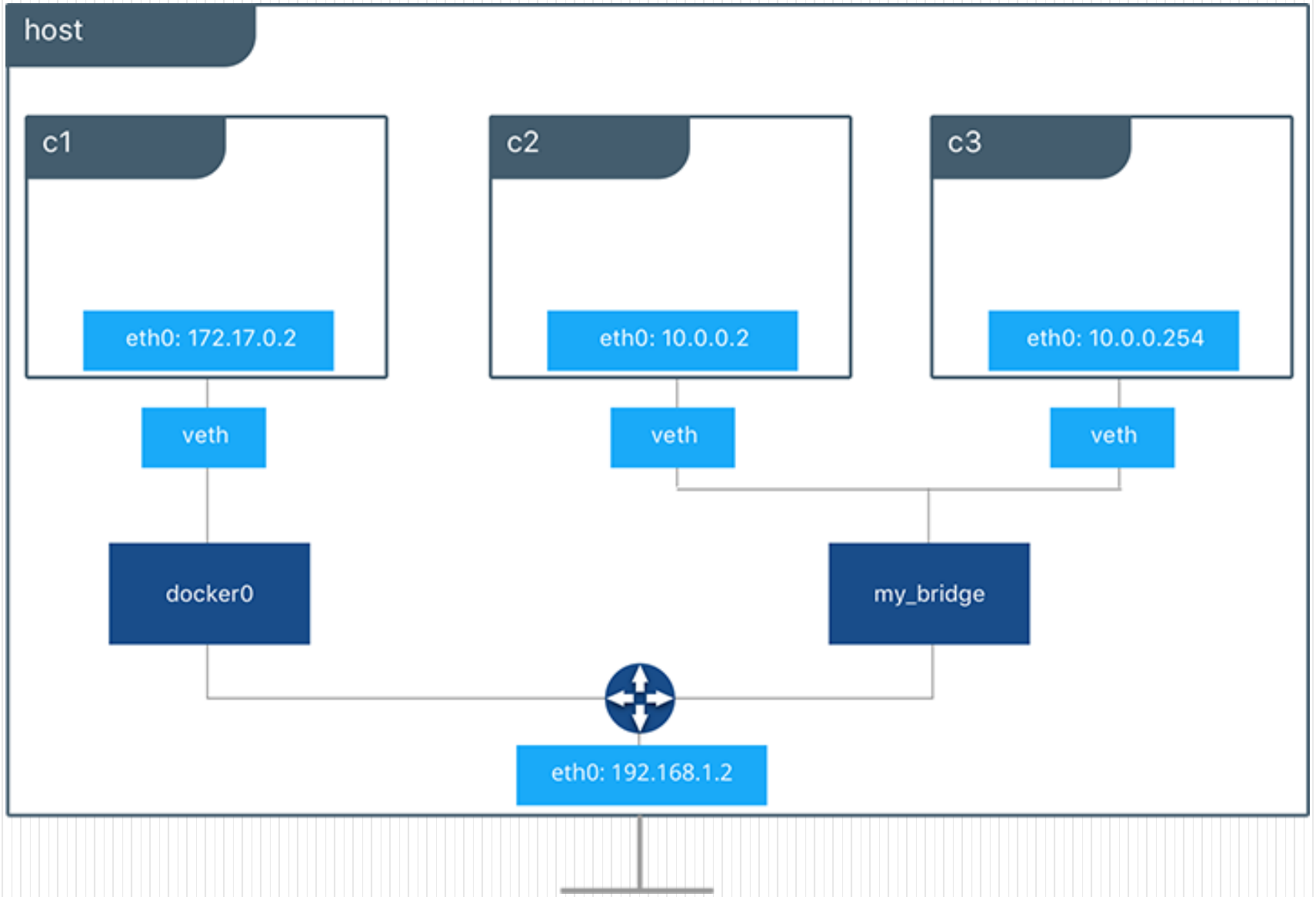
Docker Networking

Bridge network

Bridge network

- Containers connected to the same user-defined bridge network automatically expose all ports to each other, and no ports to the outside world. This allows containerized applications to communicate with each other easily, without accidentally opening access to the outside world.

Bridge network



Bridge interface

- Create interface

  docker network create **--driver bridge** my-net

- --driver bridge : default setting.

- Remove interface

  docker network rm my-net

Bridge network

- Connect to bridge interface

```
docker create --name my-nginx \
  --network my-net \
  --publish 8080:80 \
  nginx:latest
```

Bridge network

- The tool brctl on the host shows the Linux bridges that exist in the host network namespace. It shows a single bridge called docker0. docker0 has one interface, vetha3788c4, which provides connectivity from the bridge to the eth0 interface inside container c1.

```
brctl show
bridge name     bridge id           STP enabled    interfaces
docker0         8000.0242504b5200   no             vethb64e8b8
```

Bridge network

- You can only connect to one network during the docker run command, so you need to use

  docker run -dit --name alpine1 --network alpine-net alpine ash

- Connect ciontainer to the bridge network afterward

  docker network connect bridge alpine4

- Disconnect from network

  docker network disconnect bridge alpine4

Bridge network

- By default, traffic from containers connected to the default bridge network is not forwarded to the outside world. To enable forwarding, you need to change two settings.
- Configure the Linux kernel to allow IP forwarding.

```
sysctl net.ipv4.conf.all.forwarding=1
```

- Change the policy for the iptables FORWARD policy from DROP to ACCEPT.

```
iptables -P FORWARD ACCEPT
```

Configure the default bridge network

- Configure the default bridge network (file daemon.json).

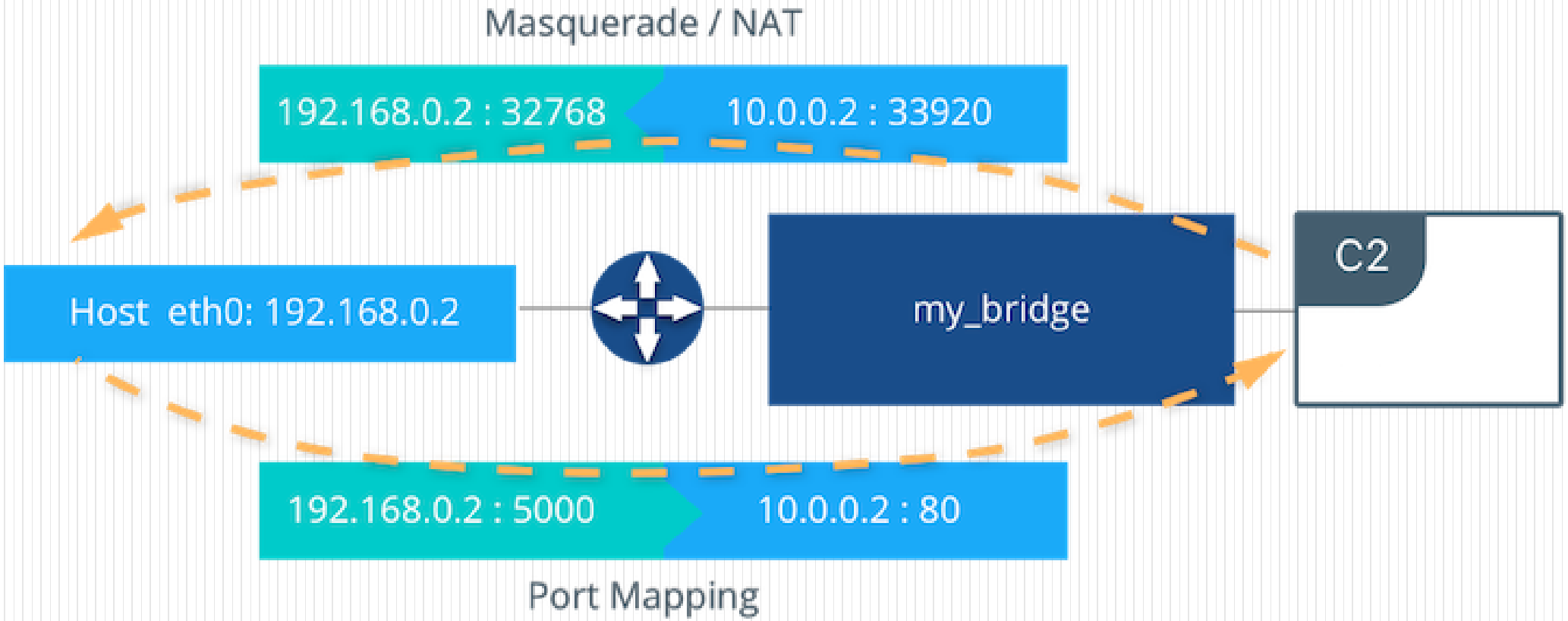
```
{
 "bip": "192.168.1.5/24",
 "fixed-cidr": "192.168.1.5/25",
 "fixed-cidr-v6": "2001:db8::/64",
 "mtu": 1500,
 "default-gateway": "10.20.1.1",
 "default-gateway-v6": "2001:db8:abcd::89",
 "dns": ["10.20.1.2","10.20.1.3"]
}
```

Configure custom bridge network

- Below a user-defined bridge network is created with two containers attached to it. A subnet is specified, and the network is named my_bridge. One container is not given IP parameters, so the IPAM driver assigns it the next available IP in the subnet. The other container has its IP specified.

```
docker network create -d bridge --subnet 10.0.0.0/24 my_bridge
docker run -itd --name c2 --net my_bridge busybox sh
docker run -itd --name c3 --net my_bridge --ip 10.0.0.254 busybox sh
```

External Access for Standalone Containers



External Access for Standalone Containers

- For most types of Docker networks (bridge and overlay included) external ingress access for applications must be explicitly granted.
- This is done through internal port mapping. Docker publishes ports exposed on host interfaces to internal container interfaces.
- The port can be set to listen on a specific (or all) host interfaces, and all traffic is mapped from this port to a port and interface inside the container.

```
docker run -d --name C2 --net my_bridge -p 5000:80 nginx
```

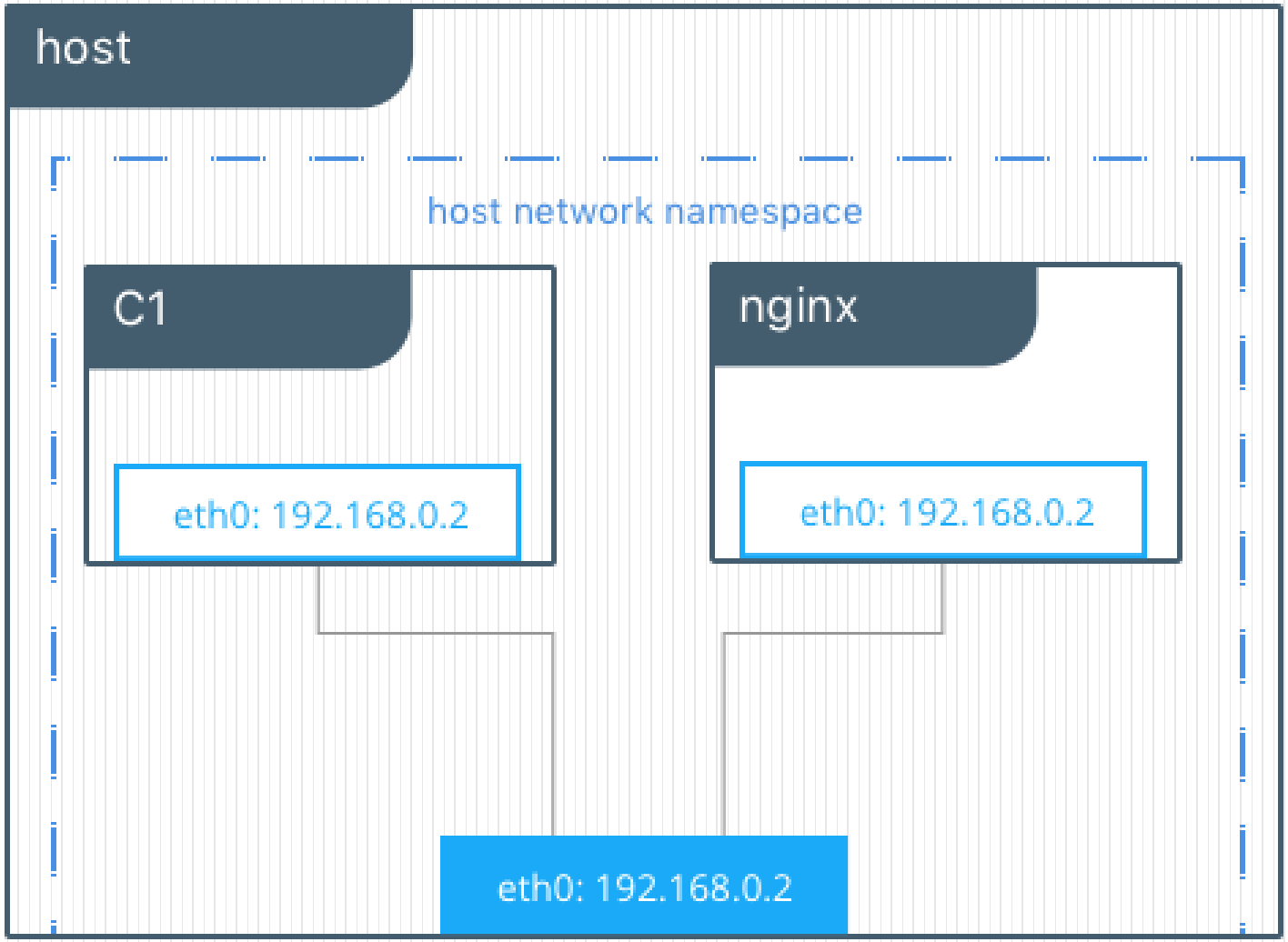
Docker Networking

Host network

Docker Host Network Driver

- The host network driver is most familiar to those new to Docker because it's the same networking configuration that Linux uses without Docker.
- --net=host effectively turns Docker networking off and containers use the host (or default) networking stack of the host operating system.
- Typically with other networking drivers, each container is placed in its own network namespace (or sandbox) to provide complete network isolation from each other. With the host driver containers are all in the same host network namespace and use the network interfaces and IP stack of the host. All containers in the host network are able to communicate with each other on the host interfaces. From a networking standpoint this is equivalent to multiple processes running on a host without containers. Because they are using the same host interfaces, no two containers are able to bind to the same TCP port. This may cause port contention if multiple containers are being scheduled on the same host.

Host network



Docker Host Network Driver

- #Create containers on the host network

```
docker run -itd --net host --name C1 alpine sh
docker run -itd --net host --name nginx
```

- #Show host eth0

```
ip add | grep eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group default qlen 1000
    inet 172.31.21.213/20 brd 172.31.31.255 scope global eth0
```

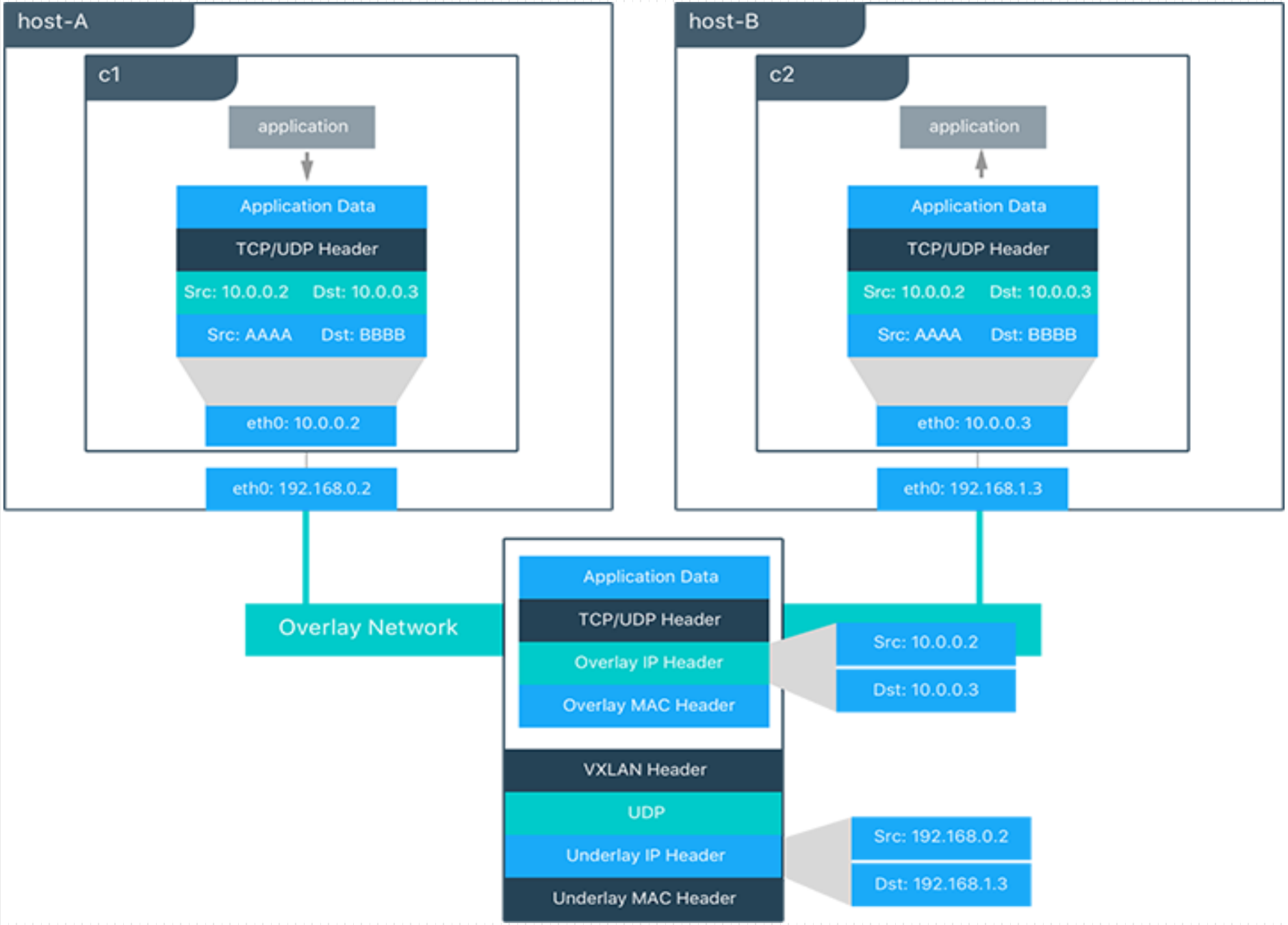
- #Show eth0 from C1

```
docker run -it --net host --name C1 alpine ip add | grep eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP qlen 1000
    inet 172.31.21.213/20 brd 172.31.31.255 scope global eth0
```

Docker Networking

Overlay network

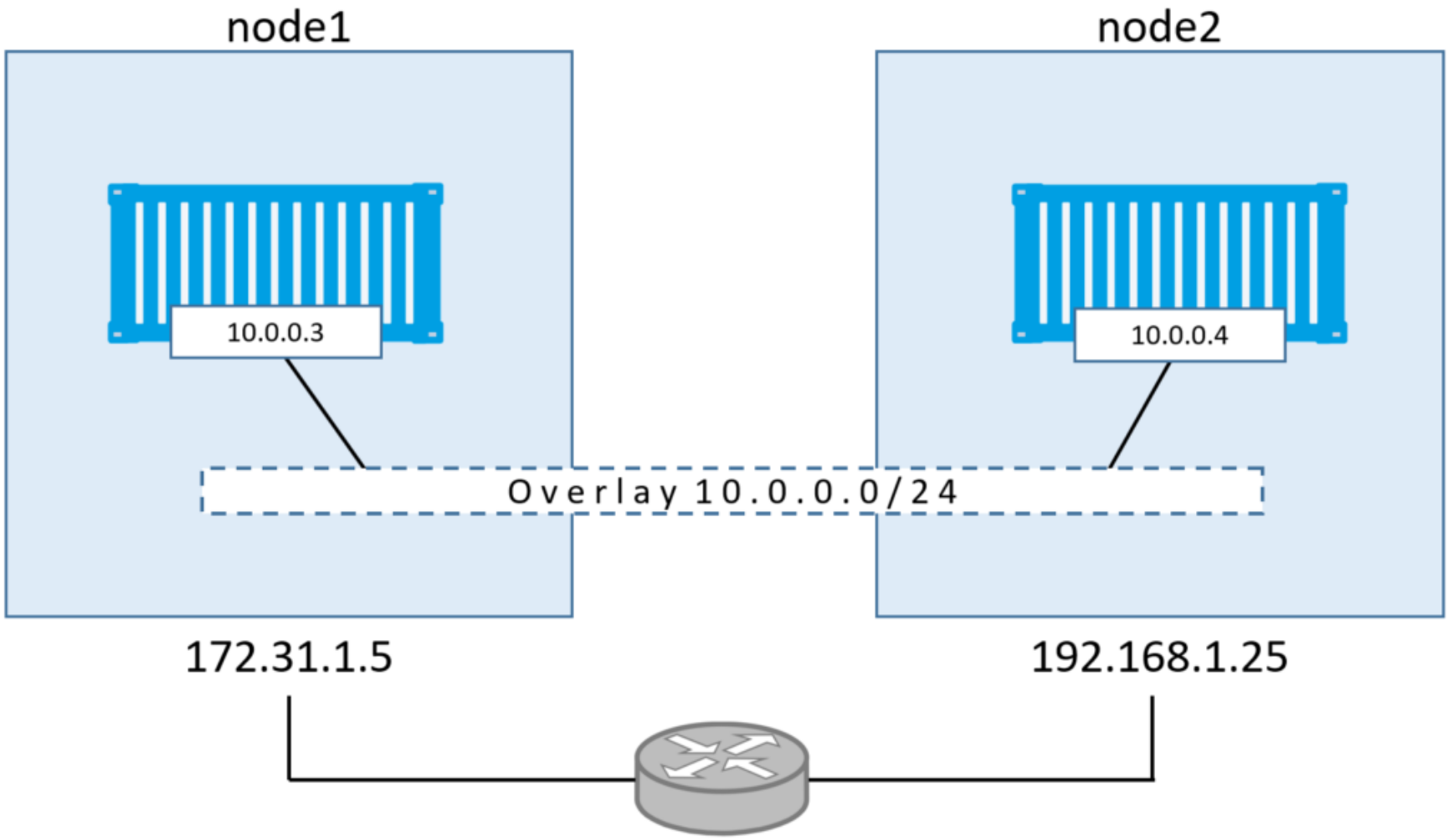
Overlay network



Overlay network

- The overlay network driver creates a distributed network among multiple Docker daemon hosts. This network sits on top of (overlays) the host-specific networks, allowing containers connected to it (including swarm service containers) to communicate securely. Docker transparently handles routing of each packet to and from the correct Docker daemon host and the correct destination container.

Overlay network



Overlay network

- To create an overlay network for use with swarm services, use a command like the following:

```
docker network create -d overlay my-overlay
```

- To create an overlay network which can be used by swarm services or standalone containers to communicate with other standalone containers running on other Docker daemons, add the --attachable flag:

```
docker network create -d overlay --attachable my-attachable-overlay
```

Overlay network

- You can use the overlay network feature with both --opt encrypted –attachable, define custom ip range and gateway and attach unmanaged containers to that network:

```
docker network create --opt encrypted --driver overlay \
        --attachable  --subnet=10.11.0.0/16 \
        --gateway=10.11.0.2  \
        --opt com.docker.network.driver.mtu=1200 \
        my-attachable-multi-host-network
```

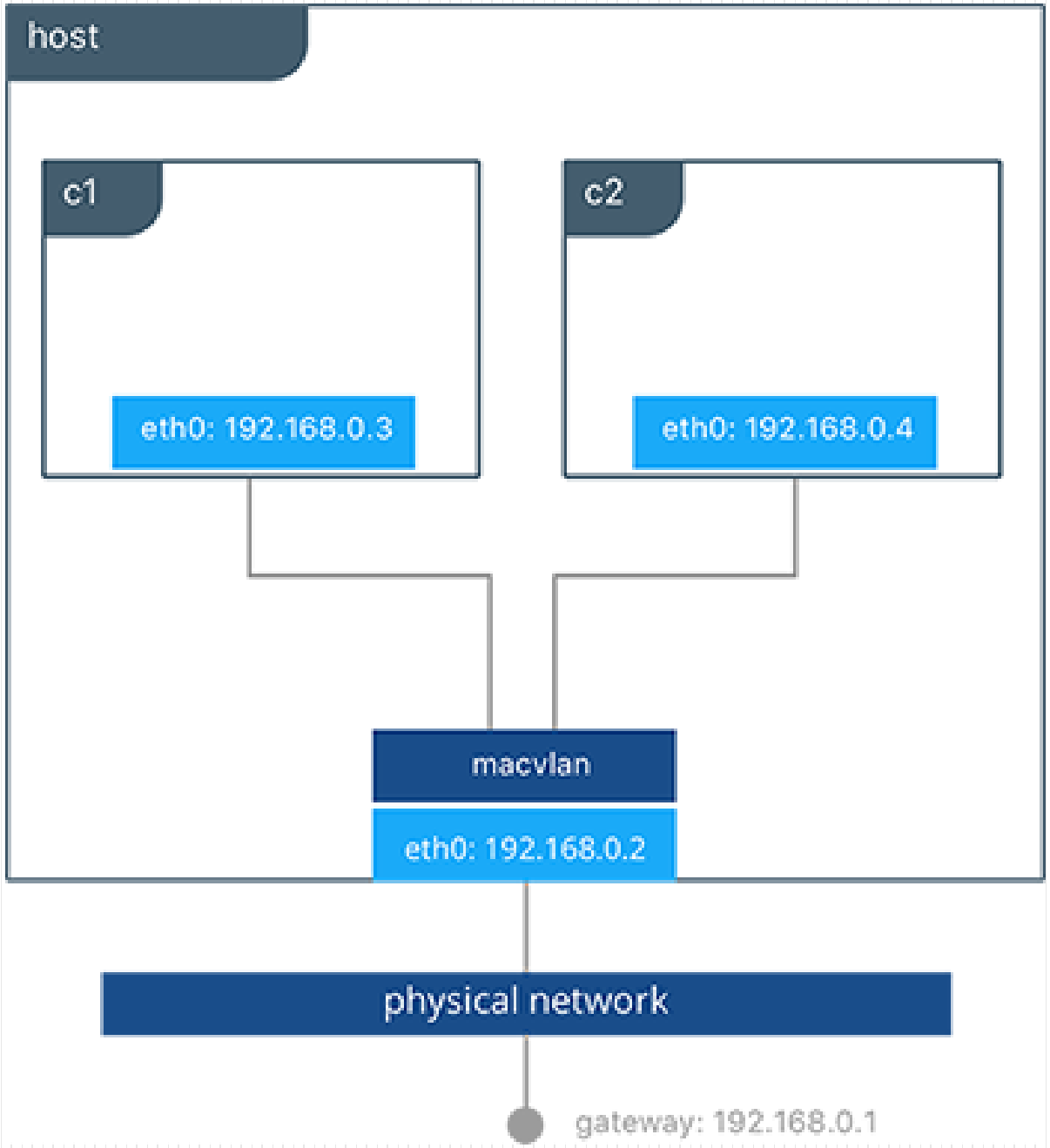
Docker Networking

Macvlan

Macvlan network

- Some applications, especially legacy applications or applications which monitor network traffic, expect to be directly connected to the physical network. In this type of situation, you can use the macvlan network driver to assign a MAC address to each container’s virtual network interface, making it appear to be a physical network interface directly connected to the physical network.

Macvlan network



Macvlan network

- The macvlan driver is a new implementation of the tried and true network virtualization technique.
- The Linux implementations are extremely lightweight because rather than using a Linux bridge for isolation, they are simply associated with a Linux Ethernet interface or sub-interface to enforce separation between networks and connectivity to the physical network.
- MACVLAN use-cases may include:
  - Very low-latency applications
  - Network design that requires containers be on the same subnet as and using IPs as the external host network

Macvlan network

- To create a Macvlan network which bridges with a given physical network interface

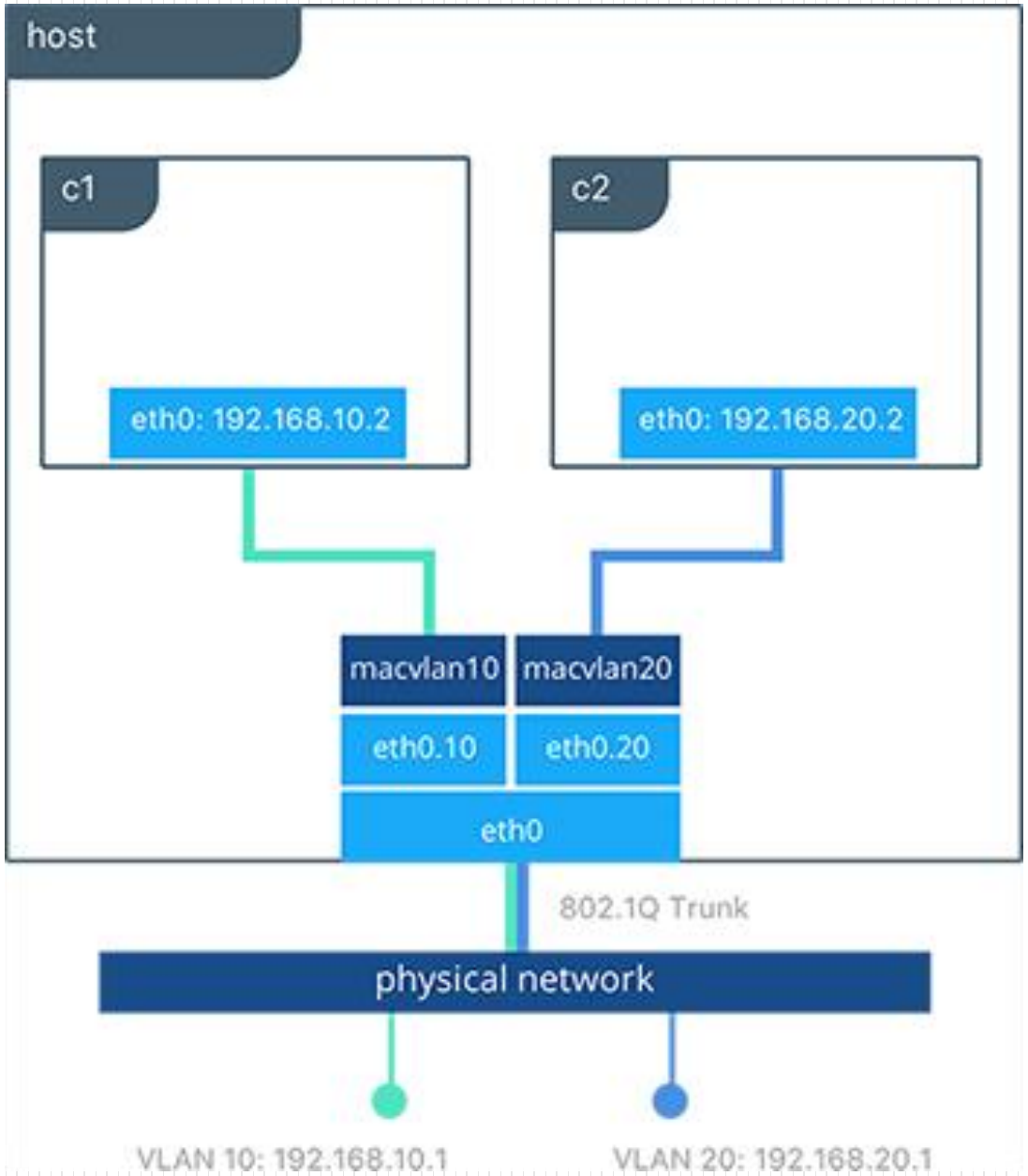
```
docker network create -d macvlan \
  --subnet=172.16.86.0/24  --gateway=172.16.86.1  \
  -o parent=eth0 pub_net
```

Macvlan network

- To connect containers to the Macvlan network

```
docker run -itd --name c1 --net mvnet --ip 192.168.0.3 busybox sh
docker run -it --name c2 --net mvnet --ip 192.168.0.4 busybox sh
```

Macvlan network with vlans



Macvlan network with vlans

- #Creation of macvlan10 network in VLAN 10

```
docker network create -d macvlan --subnet 192.168.10.0/24 \
    --gateway 192.168.10.1 -o parent=eth0.10 macvlan10
```

- #Creation of macvlan20 network in VLAN 20

```
docker network create -d macvlan --subnet 192.168.20.0/24 \
    --gateway 192.168.20.1 -o parent=eth0.20 macvlan20
```

- #Creation of containers on separate MACVLAN networks

```
docker run -itd --name c1--net macvlan10 --ip 192.168.10.2 busybox sh
docker run -it --name c2--net macvlan20 --ip 192.168.20.2 busybox sh
```

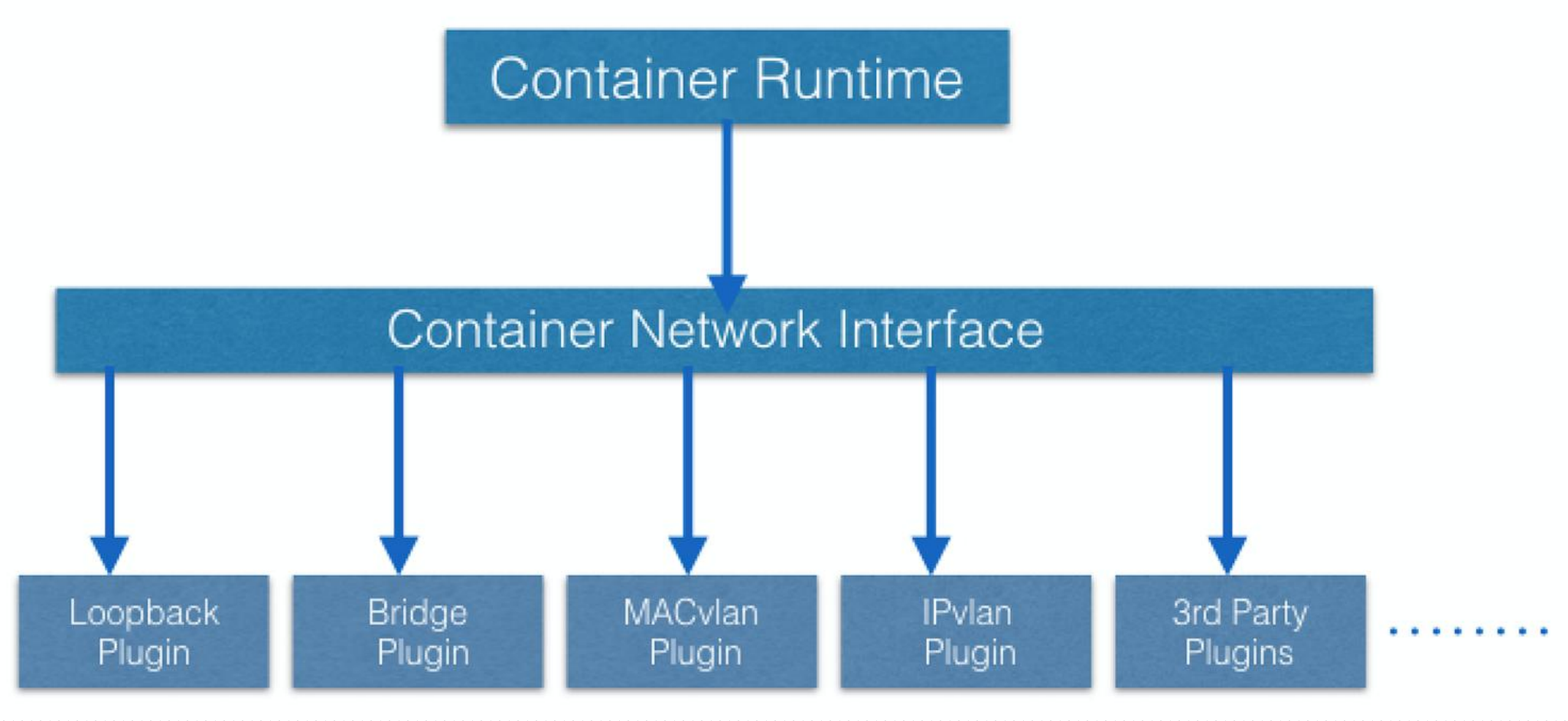
Kubernetes Networking

Basic concepts

Networking problems

- Highly-coupled container-to-container communications: this is solved by pods and localhost communications.
- Pod-to-Pod communications.
- Pod-to-Service communications: this is covered by services.
- External-to-Service communications: this is covered by services.

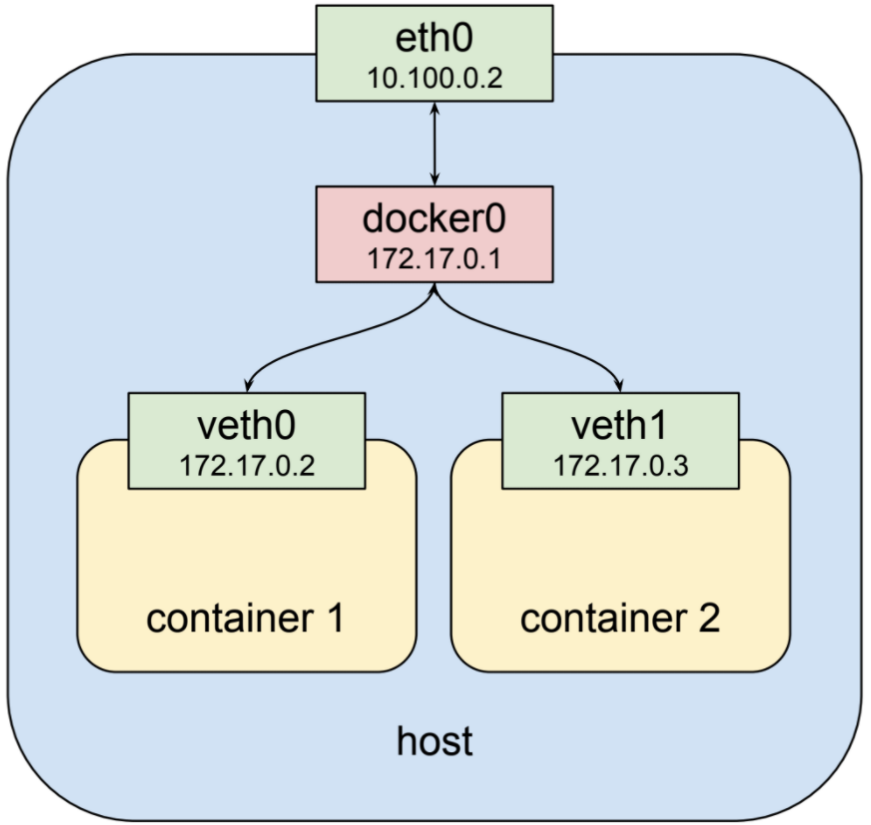
Kubernetes network core plugins



Kubernetes network model

- Every Pod gets its own IP address.
- This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports.
- This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes pod: networking



Pod networking

- Every Pod gets its own IP address.
- This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports.
- This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration..

Pod networking

- Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):
    - pods on a node can communicate with all pods on all nodes without NAT
    - agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Network Model Implememtations

- There are a number of ways that this network model can be implemented.
- Azure CNI for Kubernetes
- Google Compute Engine (GCE)
- OpenVSwitch, k-vswitch (simple Kubernetes networking plugin based on Open vSwitch).
- Project Calico.
- Weave Net from Weaveworks.

Kubernetes Networking

Ingress network

Ingress

- Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. Traffic routing is controlled by rules defined on the Ingress resource.
- An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name based virtual hosting.
- An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

Ingress Controller

- You must have an ingress controller to satisfy an Ingress. Only creating an Ingress resource has no effect.
- You may need to deploy an Ingress controller such as ingress-nginx. You can choose from a number of Ingress controllers.
- Kubernetes as a project currently supports and maintains GCE and nginx controllers.

Ingress Resource

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:

nginx.ingress.kubernetes.io/rewrite-target: /
spec:
rules:
- http:
    paths:
    - path: /testpath
      pathType: Prefix
      backend:
        serviceName: test
        servicePort: 80
```

Ingress Class

```
apiVersion: networking.k8s.io/v1beta1
kind: IngressClass
metadata:
  name: external-lb
spec:
  controller: example.com/ingress-controller
  parameters:
    apiGroup: k8s.example.com/v1alpha
    kind: IngressParameters
    name: external-lb
```

Ingress Example

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-
target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

Ingress Example

```
Name:             simple-fanout-example
Namespace:        default
Address:          178.91.123.132
Default backend:  default-http-backend:80 (10.8.2.3:8080)
Rules:
  Host        Path  Backends
  ----        ----  --------
  foo.bar.com
              /foo   service1:4200 (10.8.0.90:4200)
              /bar   service2:8080 (10.8.0.91:8080)
Annotations:
  nginx.ingress.kubernetes.io/rewrite-target:  /
Events:
  Type     Reason  Age              From                  Message
  ----     ------  ----             ----                  -------
  Normal   ADD     22s              loadbalancer-controller  default/test
```

Ingress TLS (1 od 2)

```
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
```

Ingress TLS (2 od 2)

```
apiVersion:
networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts:
      - sslexample.foo.com
    secretName: testsecret-tls
rules:
- host: sslexample.foo.com
  http:
    paths:
    - path: /
      backend:
        serviceName: service1
        servicePort: 80
```

Kubernetes Networking

Network Policy

Network Policy

- A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.
- NetworkPolicy resources use labels to select pods and define rules which specify
- By default, pods are non-isolated; they accept traffic from any source.what traffic is allowed to the selected pods.

Network Policy Terms

- podSelector: Each NetworkPolicy includes a podSelector which selects the grouping of pods to which the policy applies. The example policy selects pods with the label “role=db”. An empty podSelector selects all pods in the namespace

Network Policy Terms

- policyTypes: Each NetworkPolicy includes a policyTypes list which may include either Ingress, Egress, or both. The policyTypes field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no policyTypes are specified on a NetworkPolicy then by default Ingress will always be set and Egress will be set if the NetworkPolicy has any egress rules.

Network Policy Terms

- ingress: Each NetworkPolicy may include a list of whitelist ingress rules. Each rule allows traffic which matches both the from and ports sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an ipBlock, the second via a namespaceSelector and the third via a podSelector.

- egress: Each NetworkPolicy may include a list of whitelist egress rules. Each rule allows traffic which matches both the to and ports sections. The example policy contains a single rule, which matches traffic on a single port to any destination in 10.0.0.0/24.

Network Policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Questions?

andry.cheredarchuk@gmail.com