Dockercon EU 2017
Networking Workshop

Mark Church, Technical Account Manager @ Docker

Nico Kabar, Solutions Architect @ Docker

Agenda

1. Fundamentals & Network Drivers
2. Bridge Driver
3. Overlay Driver
4. Network Services (DNS, Internal and External Load Balancing, Publishing)
5. Lab I
6. BREAK (15 minutes)
7. MACVLAN Driver
8. Network Design & Best Practices
9. Network Troubleshooting
10. Lab II

Docker Networking Fundamentals

Docker Networking Design Philosophy

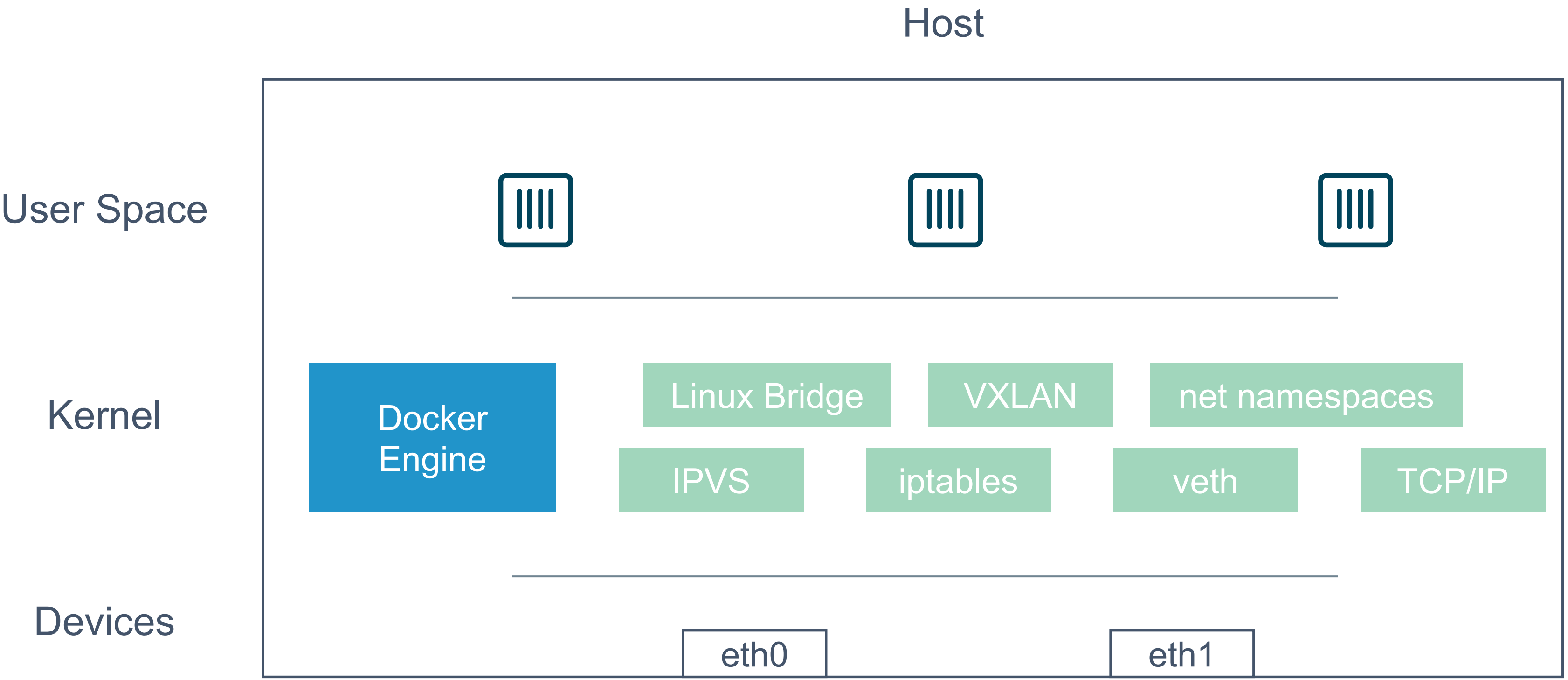
Put Users First

Developers and
Operations

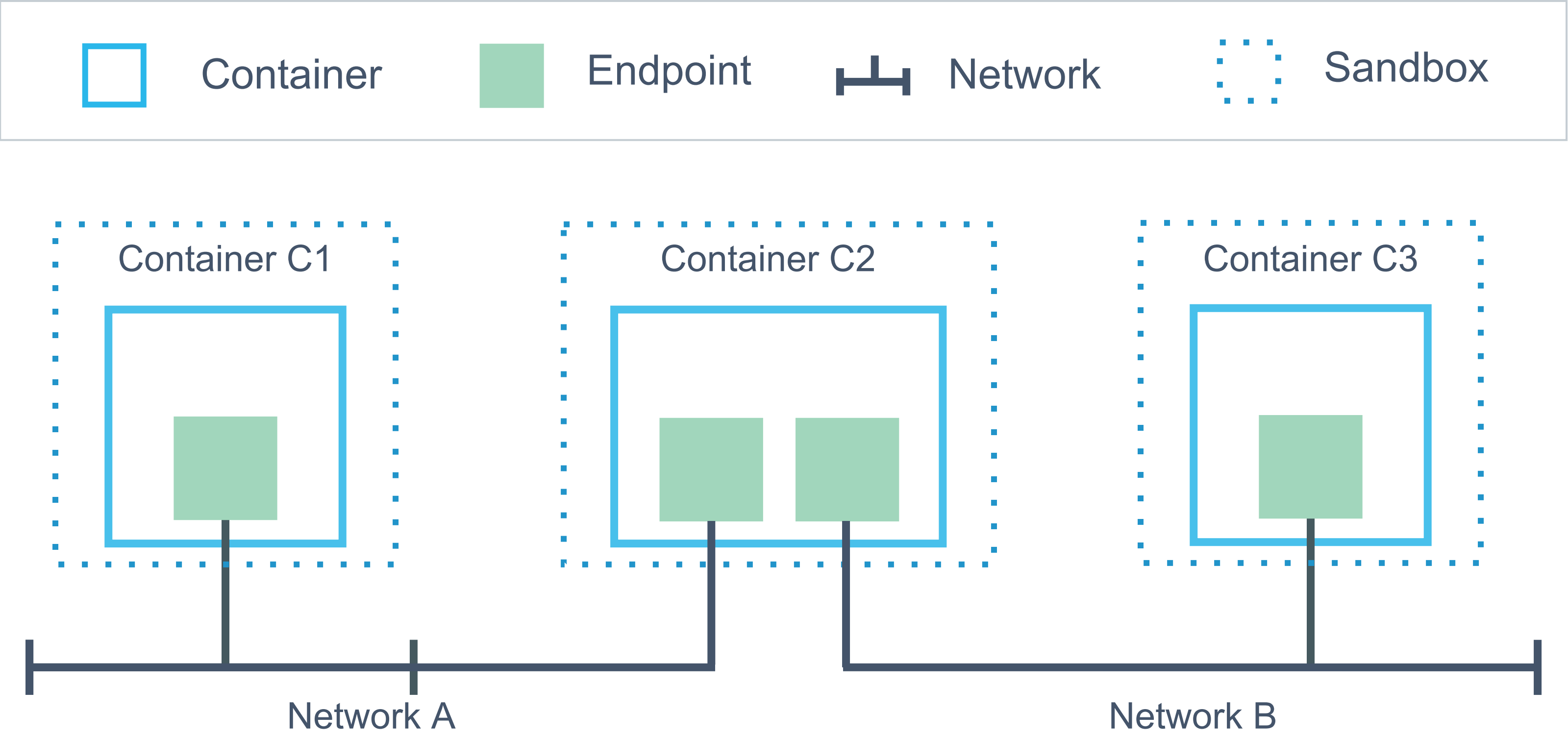
Plugin API Design

Batteries included
but removable

Docker Networking *is* Linux (and Windows) Networking



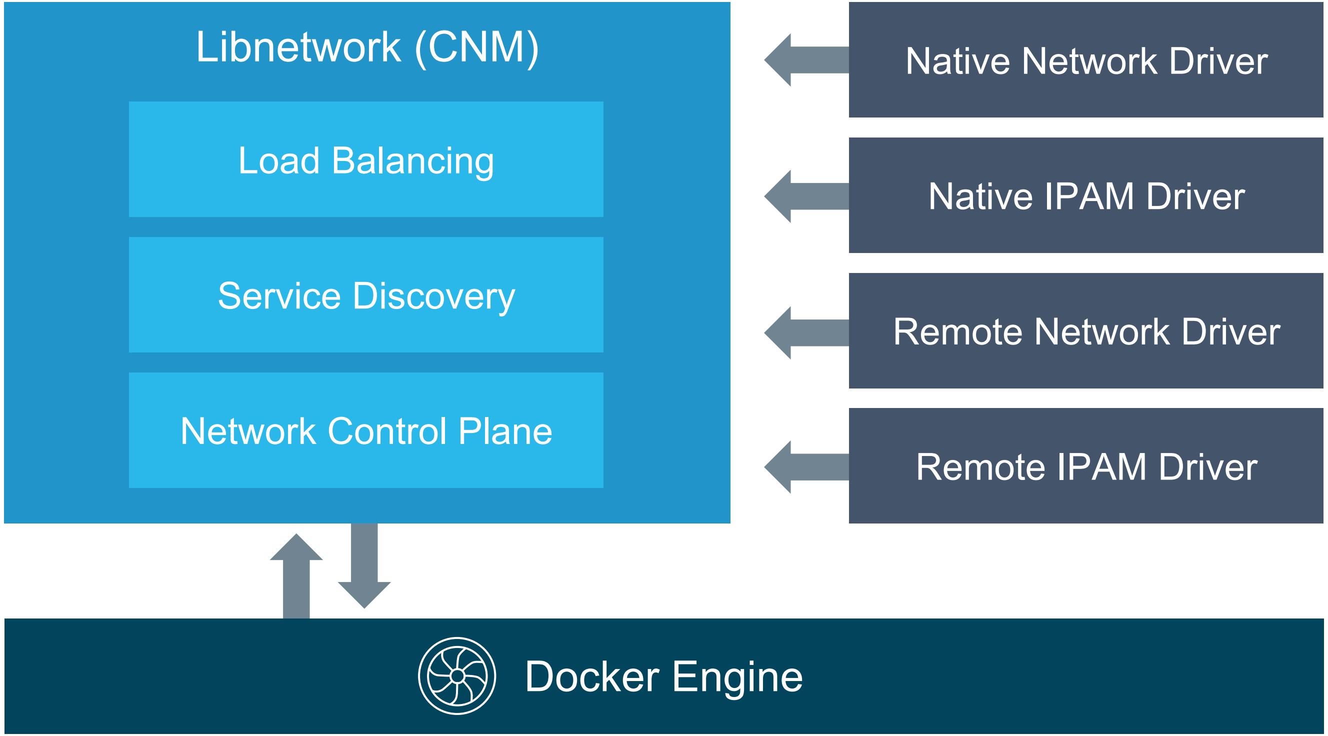
Containers and the CNM



Native Docker Networking Drivers

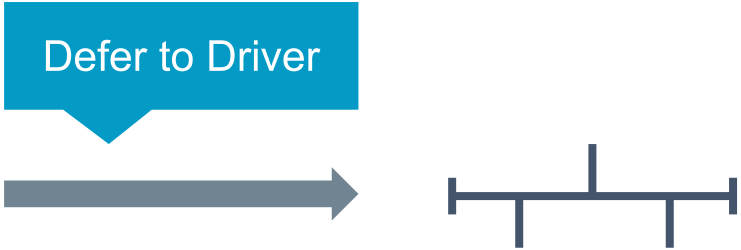
```
$ docker info

…

Plugins:
 Volume: local
 Network: bridge host ipvlan macvlan null overlay

…
```

Libnetwork Architecture



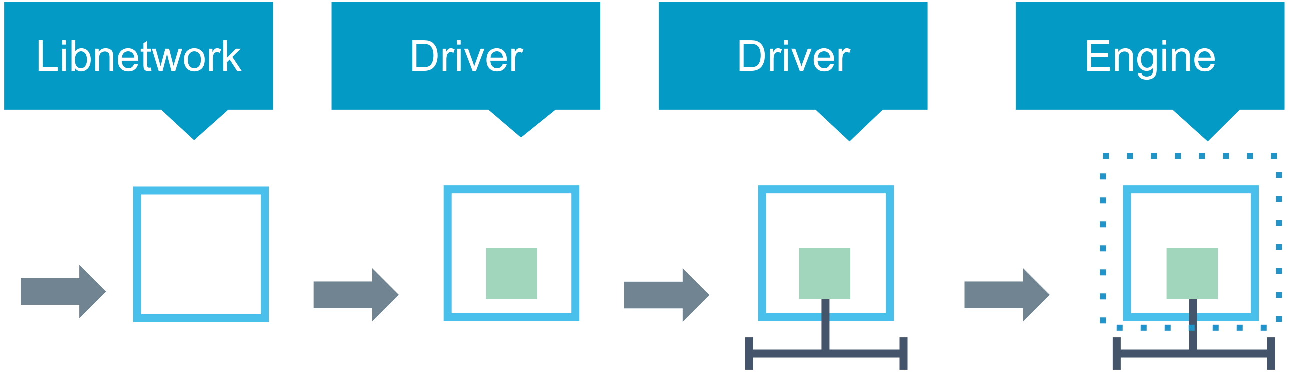
Networks and Containers

```
docker network create –d <driver> …
```



Defer to Driver

```
docker run --network
…
```



Libnetwork

Driver

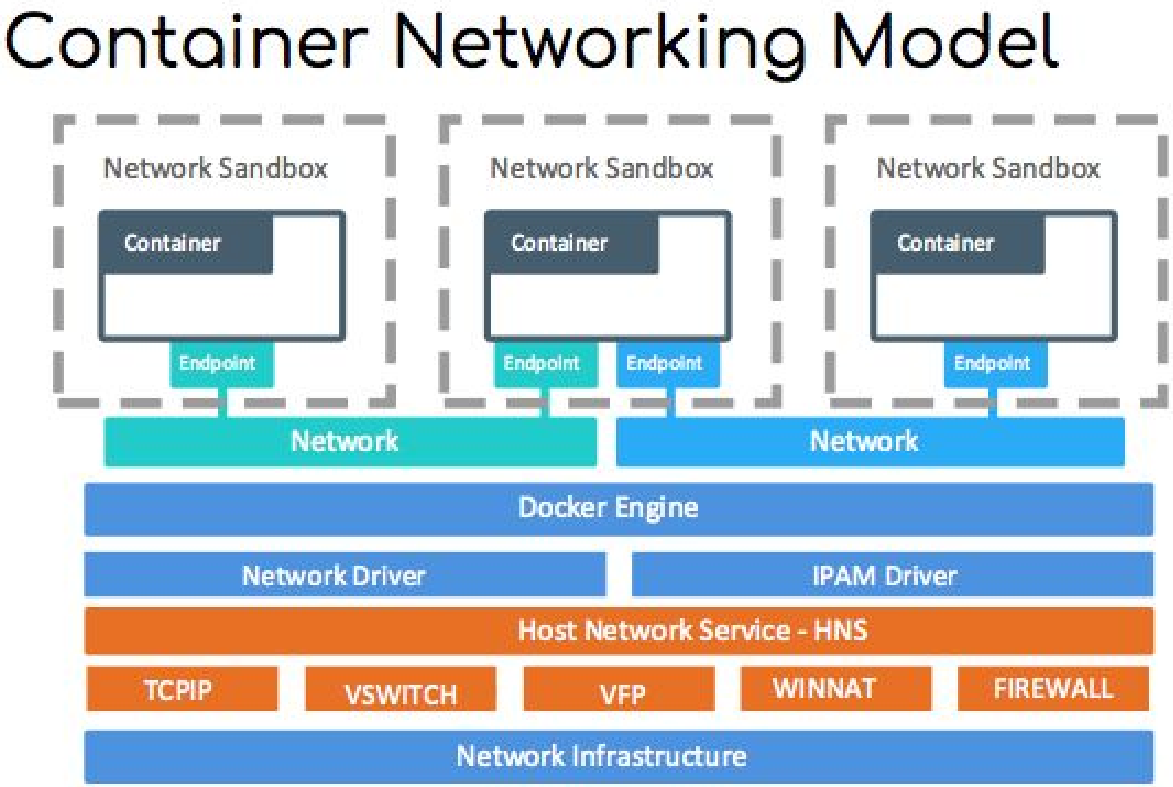
Driver

Engine

Docker Networking on Linux

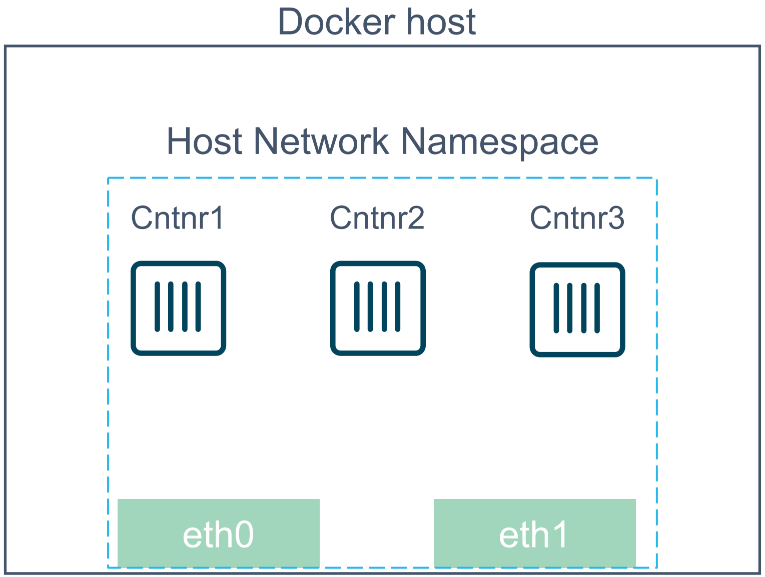
- The Linux kernel has extensive networking capabilities (TCP/IP stack, VXLAN, DNS…)
- Docker networking utilizes many Linux kernel networking features (network namespaces, bridges, iptables, veth pairs…)
- Linux bridges: L2 virtual switches implemented in the kernel
- Network namespaces: Used for isolating container network stacks
- veth pairs: Connect containers to container networks
- iptables: Used for port mapping, load balancing, network isolation…

Docker Windows Networking



Linux Networking with Containers

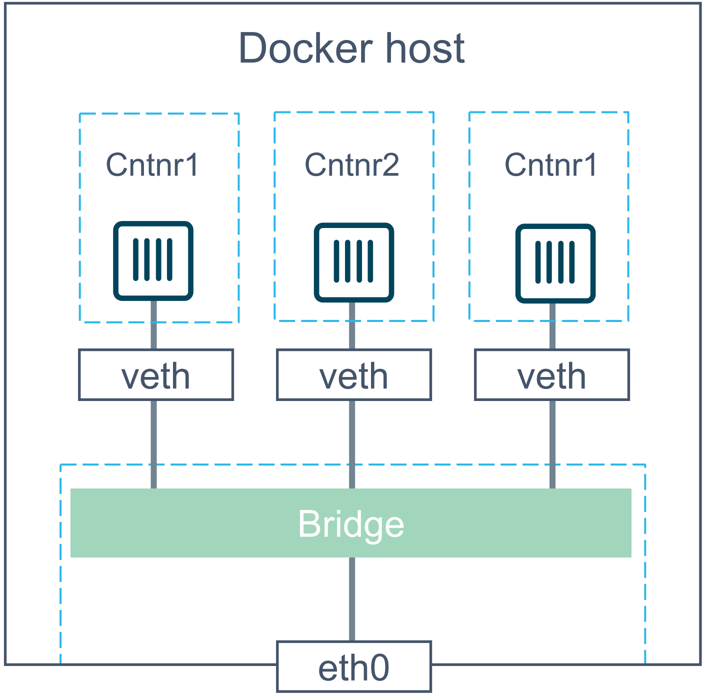
- Namespaces are used extensively for container isolation
- Host network namespace is the default namespace
- Additional network namespaces are created to isolate containers from each other



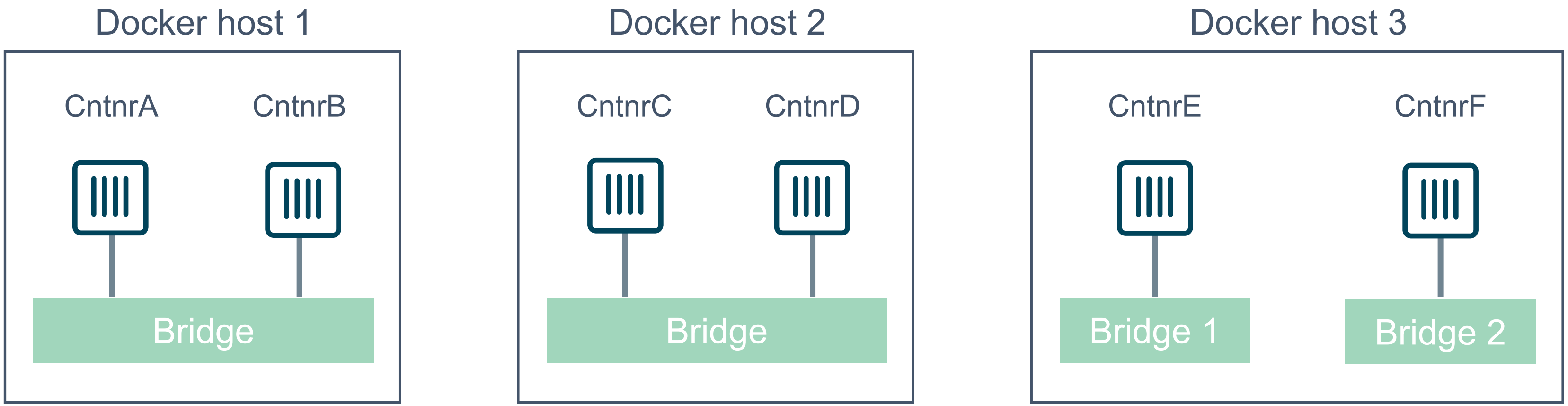
Bridge Driver

Bridge Driver in Detail

- The bridge created by the bridge driver for the pre-built bridge network is called docker0
- Each container is connected to a bridge network via a veth pair which connects between network namespaces
- Provides single-host networking
- External access requires port mapping

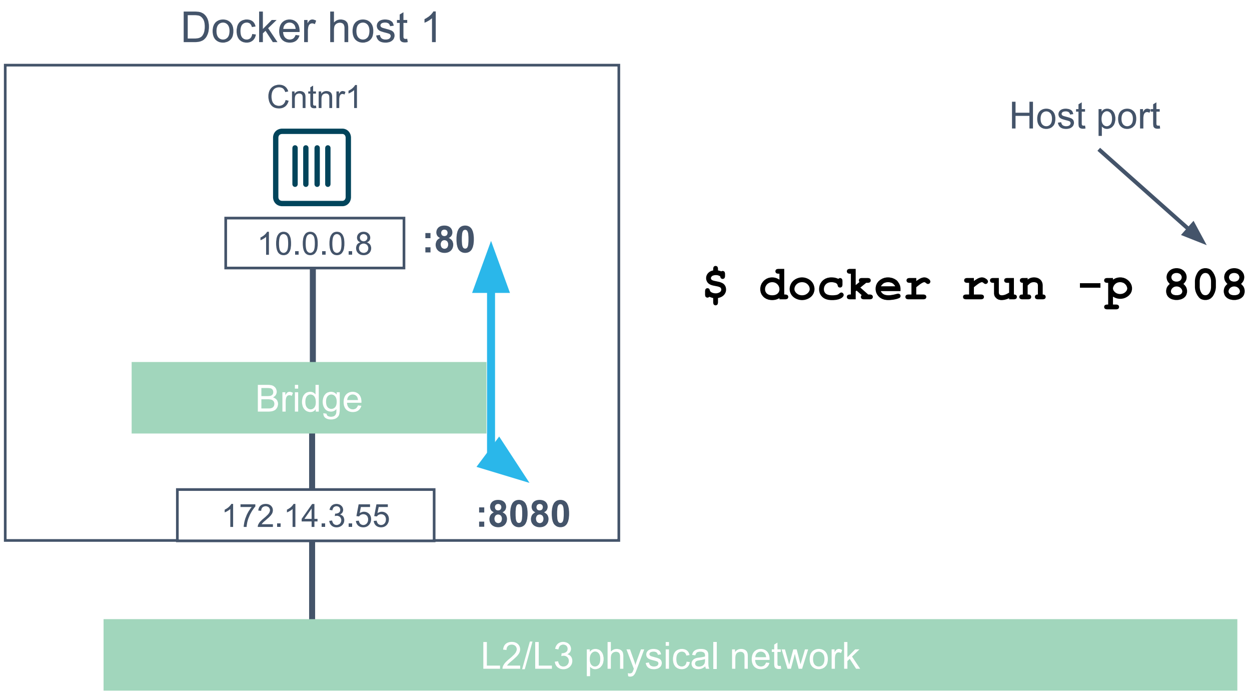


What is Docker Bridge Networking?



Containers on different **bridge** networks cannot communicate

Docker Bridge Networking and Port Mapping

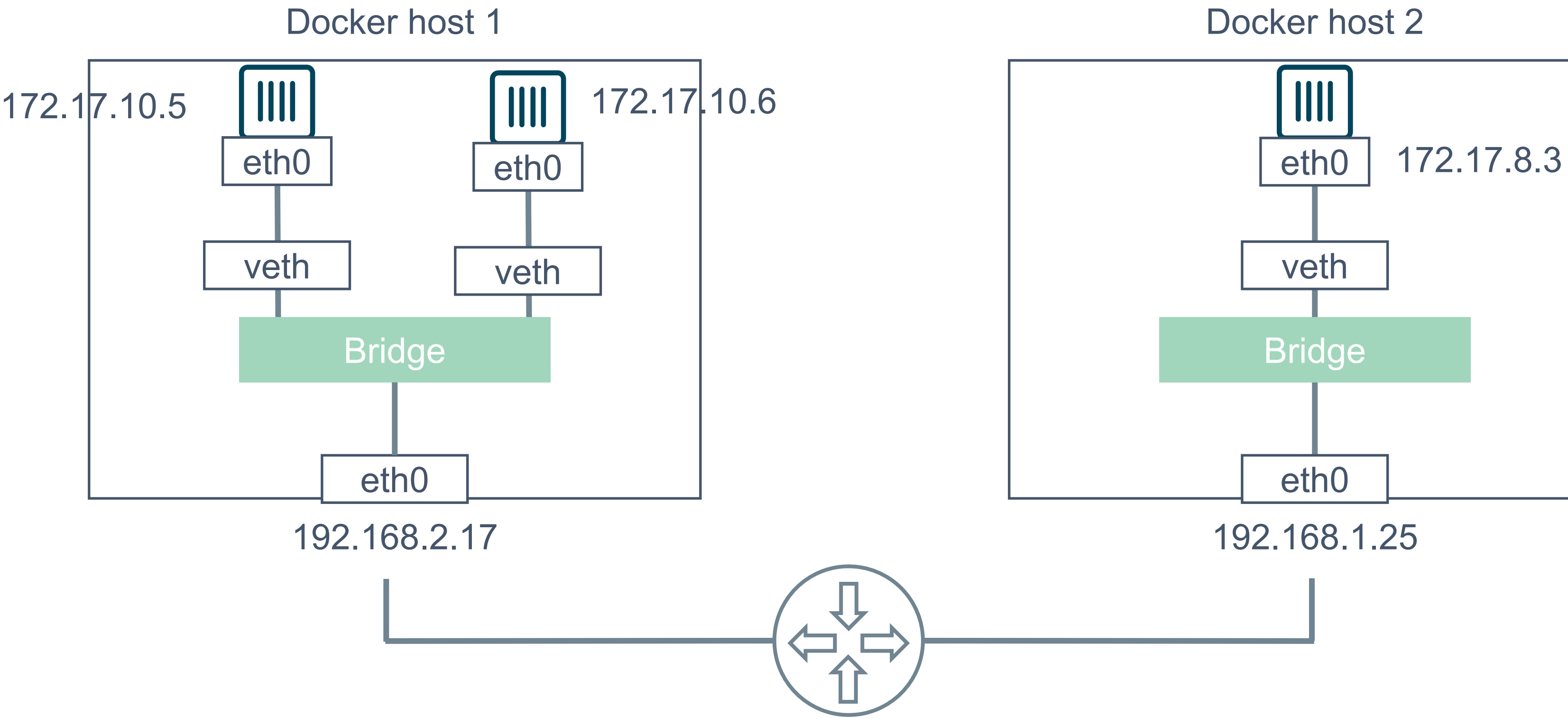


Host port

Container port

```
$ docker run -p 8080:80 ...
```

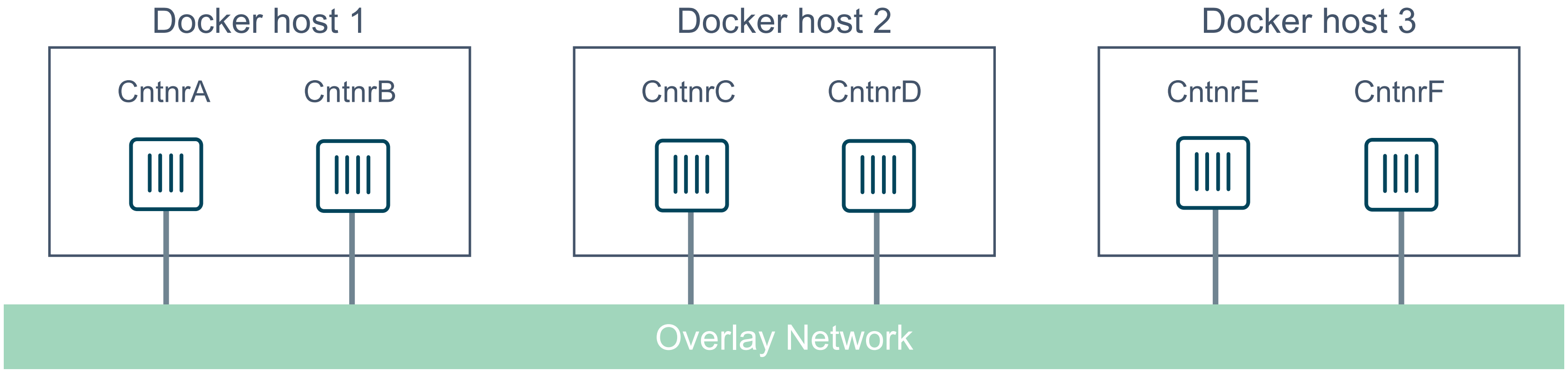
Bridge Mode Data Flow



Overlay Driver

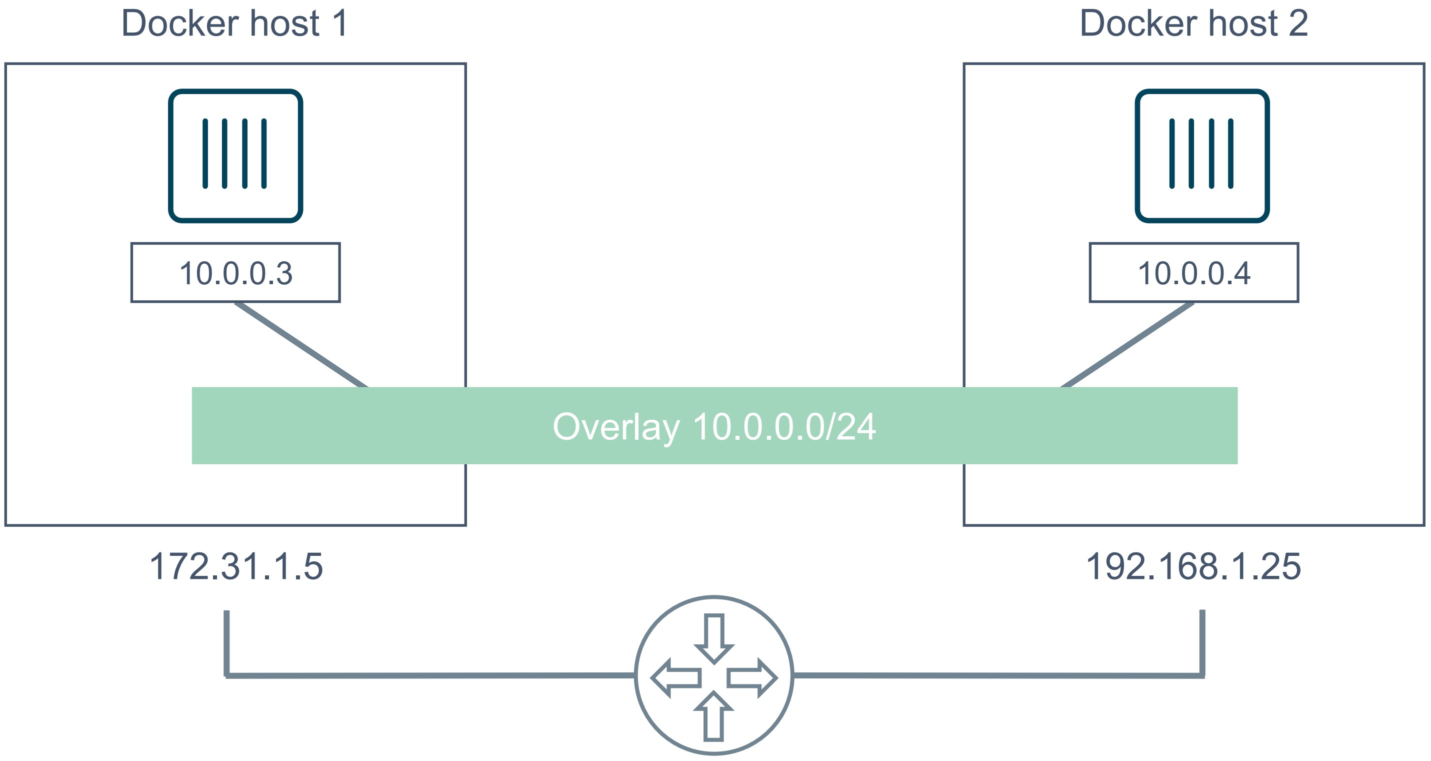
What is Docker Overlay Networking?

The **overlay** driver enables simple and secure **multi-host** networking



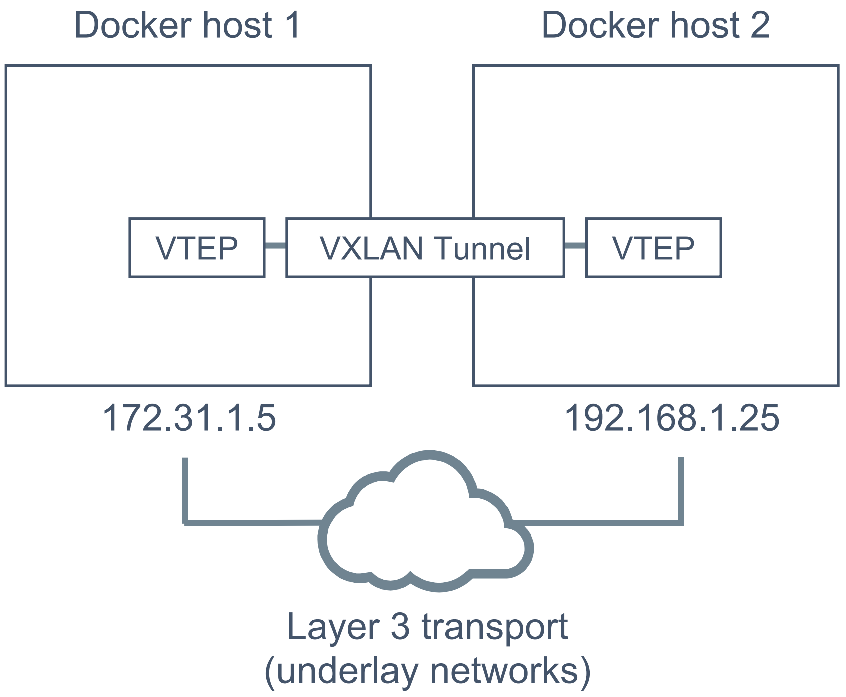
All containers on the **overlay** network can communicate!

Building an Overlay Network (High level)

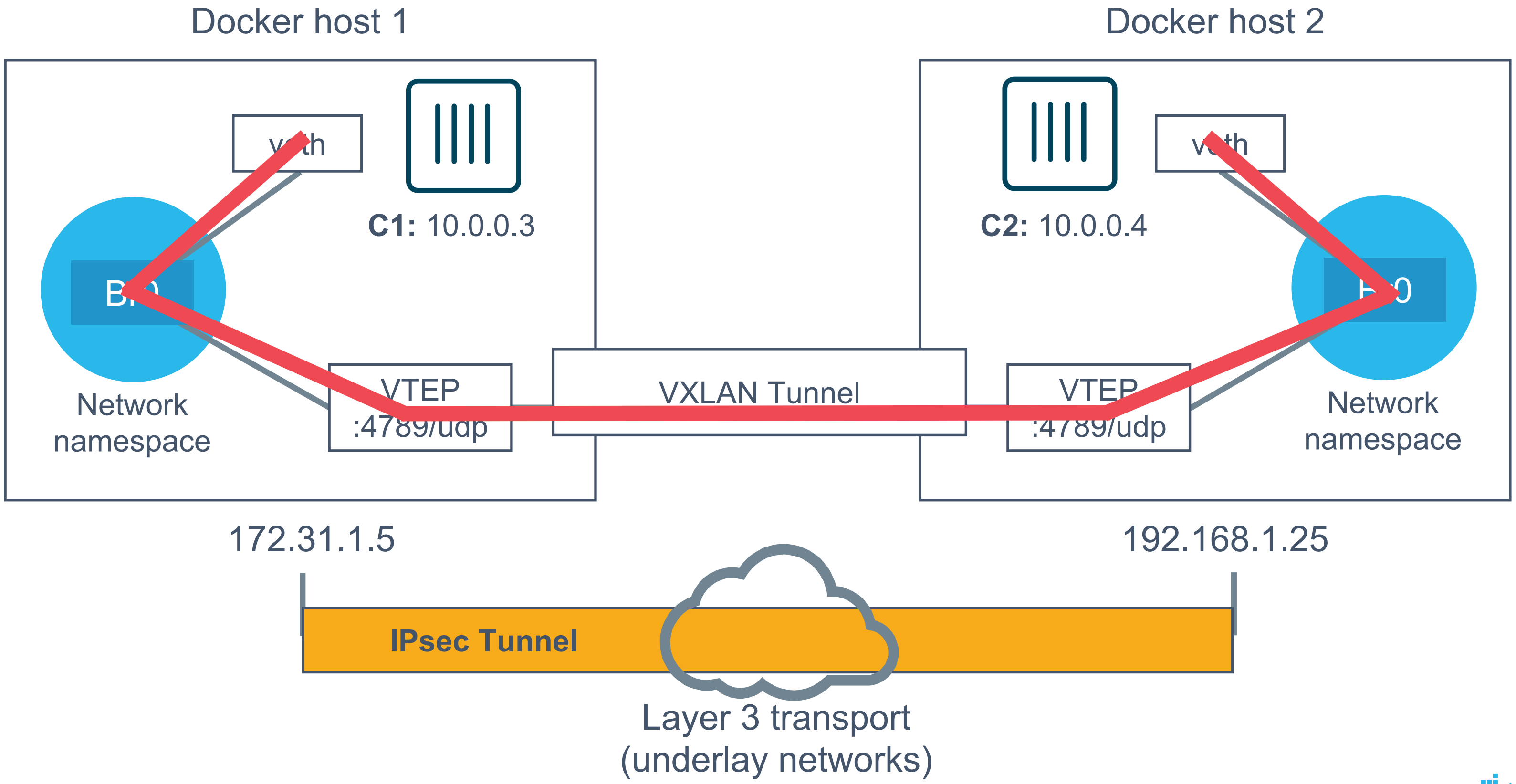


Docker Overlay Networks and VXLAN

- The **overlay** driver uses VXLAN technology to build the network
- A **VXLAN tunnel** is created through the **underlay network(s)**
- At each end of the tunnel is a VXLAN tunnel end point (**VTEP**)
- The **VTEP** performs encapsulation and de-encapsulation
- The **VTEP** exists in the Docker Host's network namespace



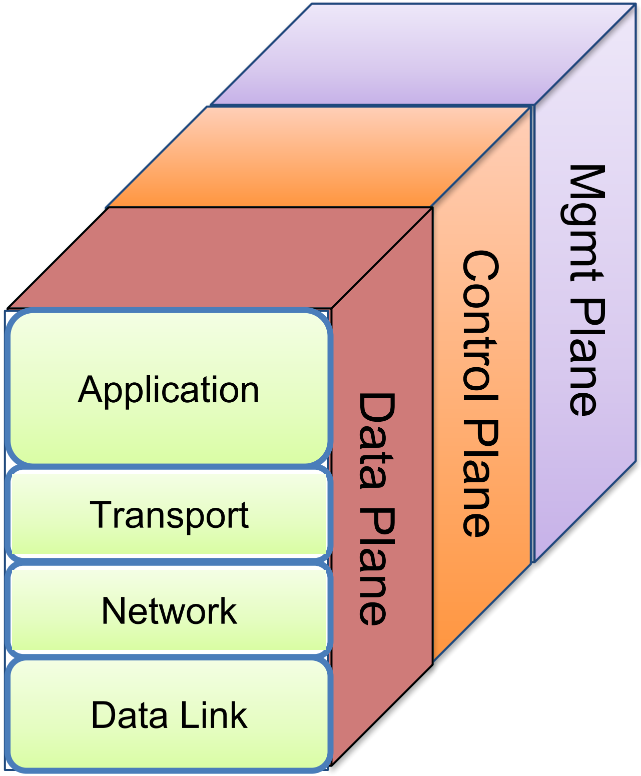
Overlay Network Encryption with IPSec



Docker Network Control Plane

Docker networking

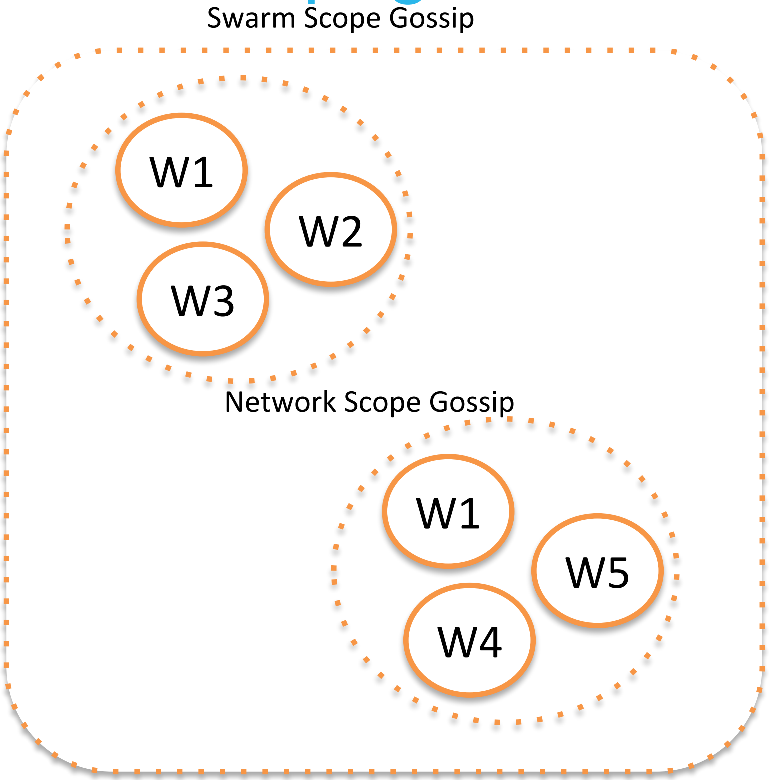
- Provides portable application services
  - Service-Discovery
  - Load-Balancing
- Built-in and pluggable network drivers
  - Overlay, macvlan, bridge
  - Remote Drivers / Plugins
- Built-in Management plane
  - API, CLI
  - Docker Stack / Compose
- Built-in distributed control plane
  - Gossip based
- Encrypted Control & Data plane



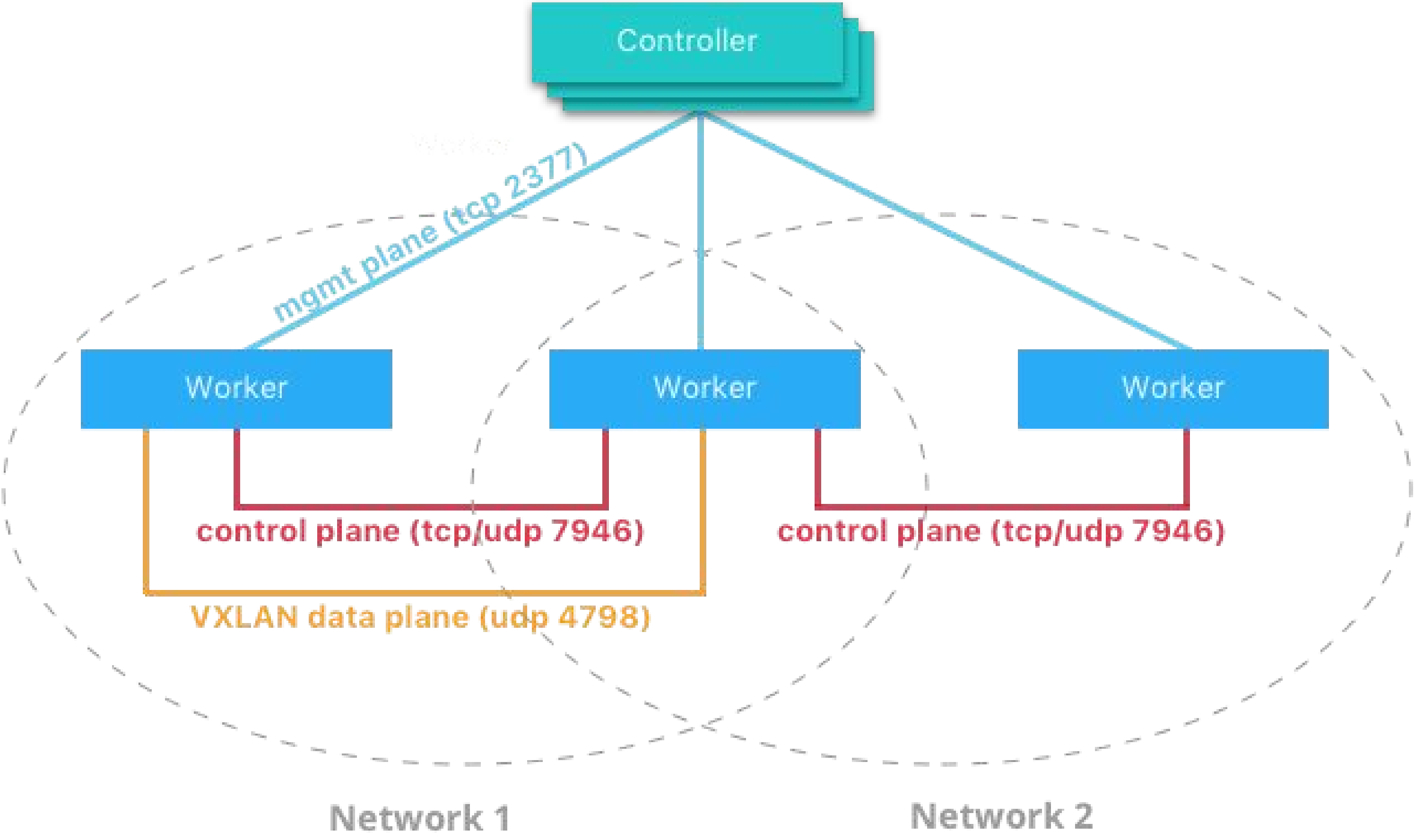
Gossip

- **Eventually consistent**
- **State dissemination through de-centralized events**
    - Service Registration
    - Load-Balancer configs
    - Routing states
- **Fast convergence**
    - ~ O(logn)
- **Highly scalable**
- **Continues to function even if all managers are Down**

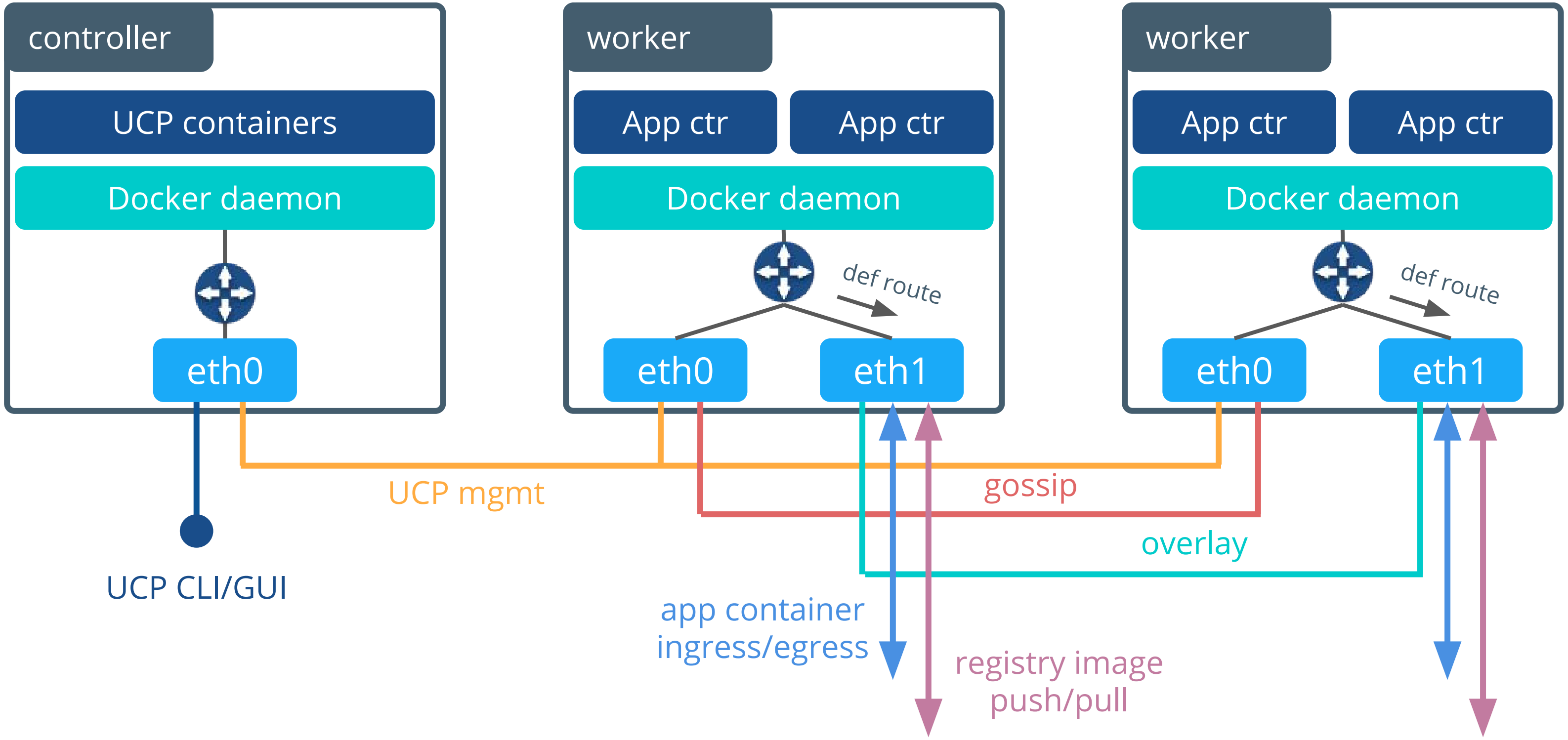
Decentralized Event Propogation

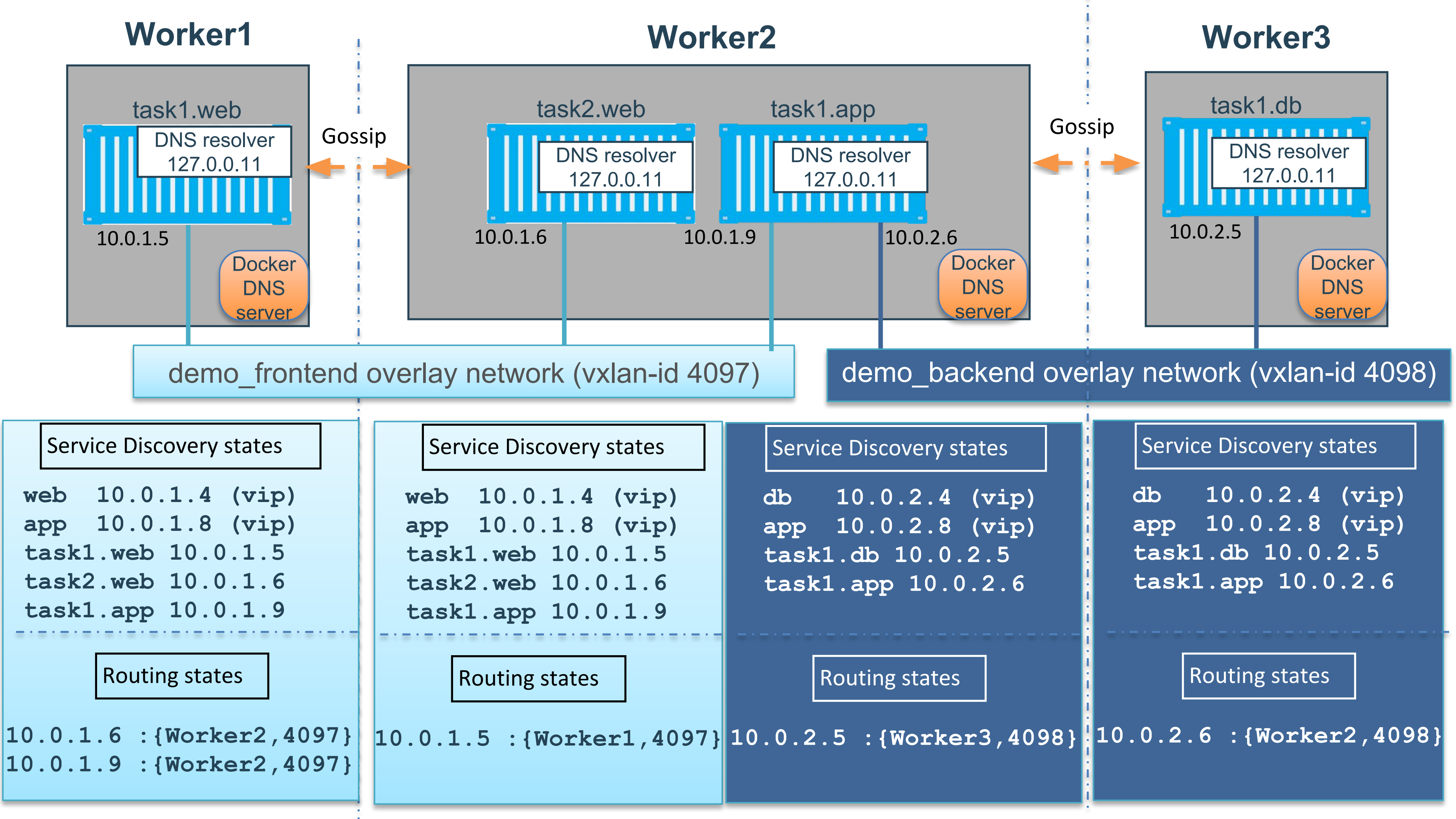


Mgmt, Control, and Data Plane in Practice


Controller
mgmt plane (tcp 2377)
Worker
Worker
Worker
control plane (tcp/udp 7946)
control plane (tcp/udp 7946)
VXLAN data plane (udp 4798)
Network 1
Network 2

Traffic Flows in Detail





Service Discovery states

```
web   10.0.1.4 (vip)
app   10.0.1.8 (vip)
task1.web 10.0.1.5
task2.web 10.0.1.6
task1.app 10.0.1.9
```

Routing states

```
10.0.1.6  :{Worker2,4097}
10.0.1.9  :{Worker2,4097}
```

Service Discovery states

```
web   10.0.1.4 (vip)
app   10.0.1.8 (vip)
task1.web 10.0.1.5
task2.web 10.0.1.6
task1.app 10.0.1.9
```

Routing states

```
10.0.1.5  :{Worker1,4097}
```

Service Discovery states

```
db    10.0.2.4 (vip)
app   10.0.2.8 (vip)
task1.db 10.0.2.5
task1.app 10.0.2.6
```

Routing states

```
10.0.2.5  :{Worker3,4098}
```

Service Discovery states

```
db    10.0.2.4 (vip)
app   10.0.2.8 (vip)
task1.db 10.0.2.5
task1.app 10.0.2.6
```

Routing states

```
10.0.2.6  :{Worker2,4098}
```

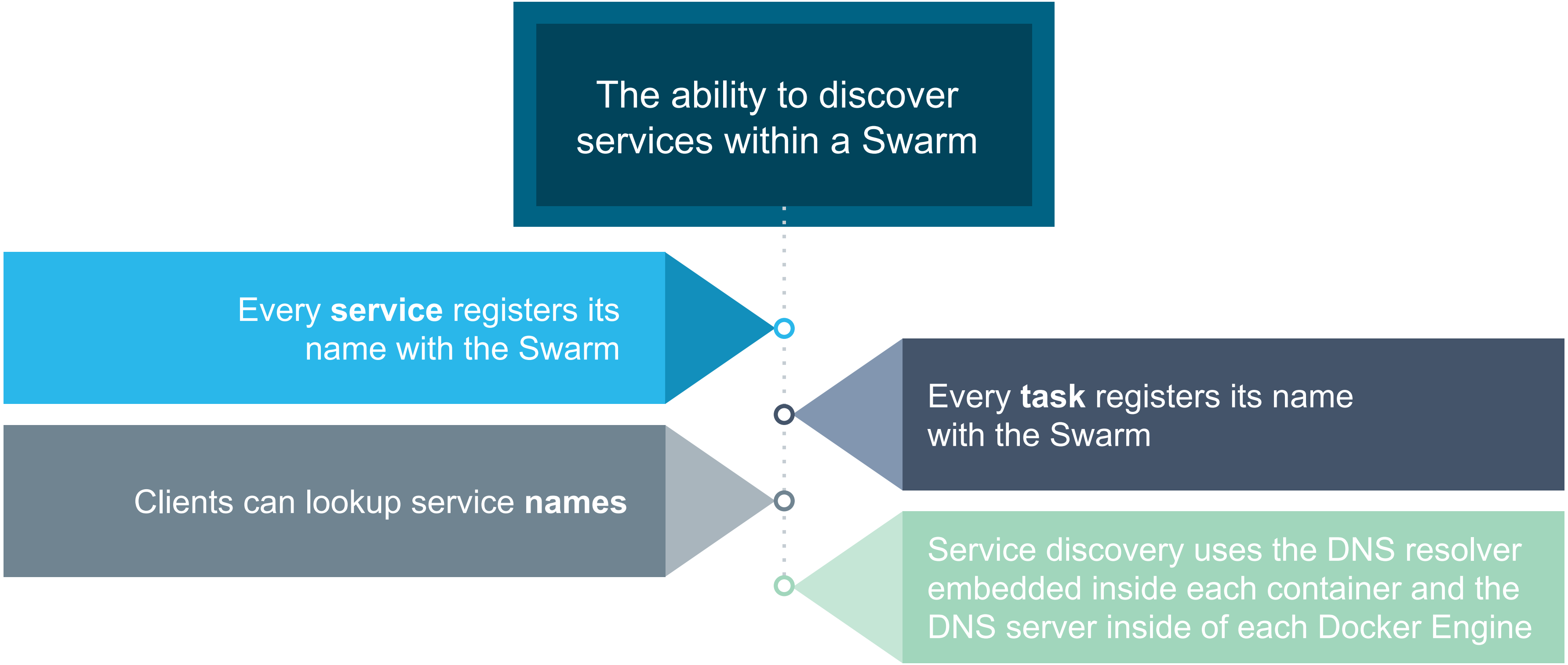
Docker Network Services

SERVICE REGISTRATION, SERVICE DISCOVERY, AND LOAD BALANCING

Docker Networking

Service Discovery

What is Service Discovery?



The ability to discover services within a Swarm

Every **service** registers its name with the Swarm

Every **task** registers its name with the Swarm

Clients can lookup service **names**

Service discovery uses the DNS resolver embedded inside each container and the DNS server inside of each Docker Engine

Service Discovery Details

Service and task registration is automatic and dynamic

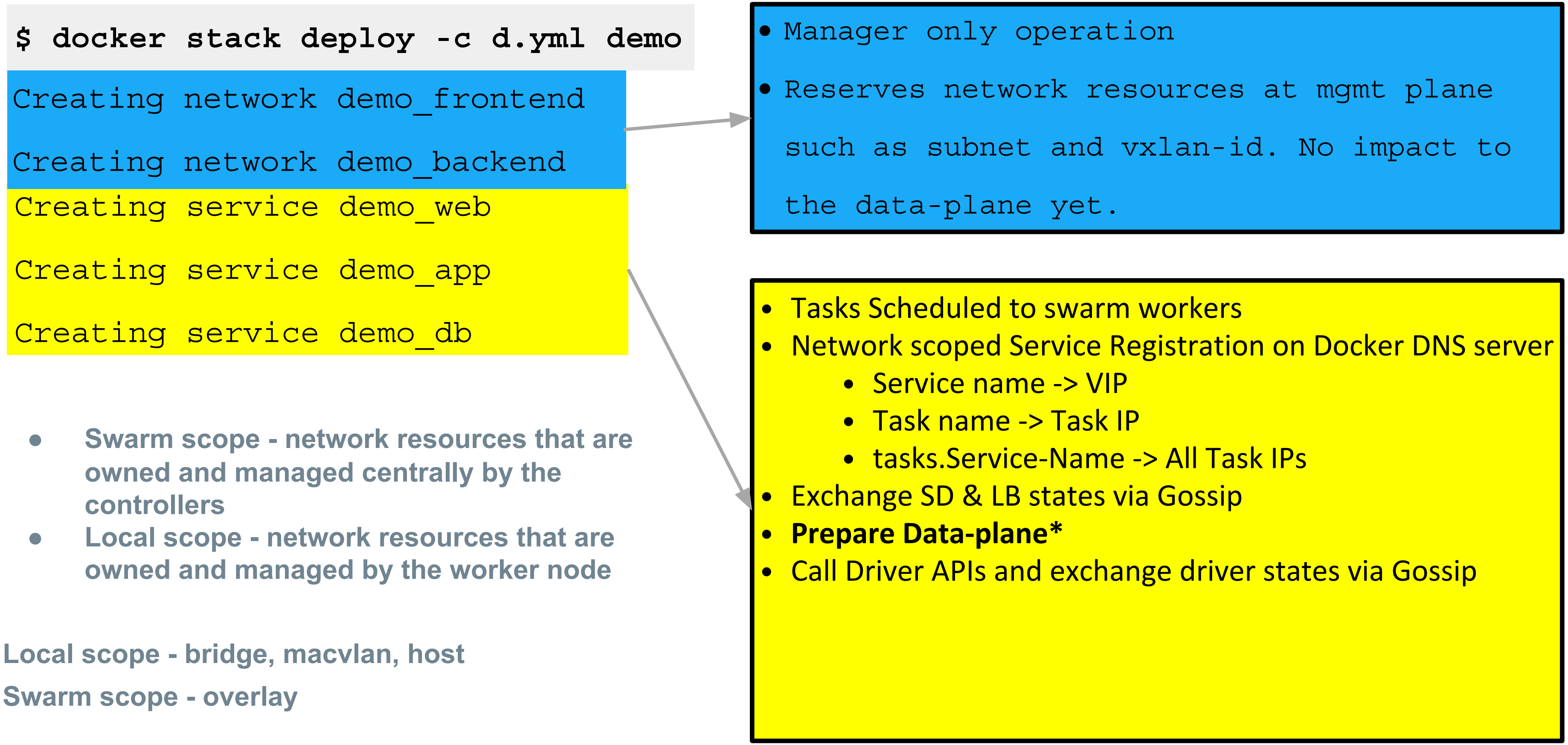
Name-IP-mappings stored in the Swarm KV store

Container DNS and Docker Engine DNS used to resolve names

- Every container runs a local DNS resolver (127.0.0.1:53)
- Every Docker Engine runs a DNS service

Resolution is network-scoped

Docker Stack Deploy



```
$ docker stack deploy -c d.yml demo
Creating network demo_frontend
Creating network demo_backend
Creating service demo_web
Creating service demo_app
Creating service demo_db
```

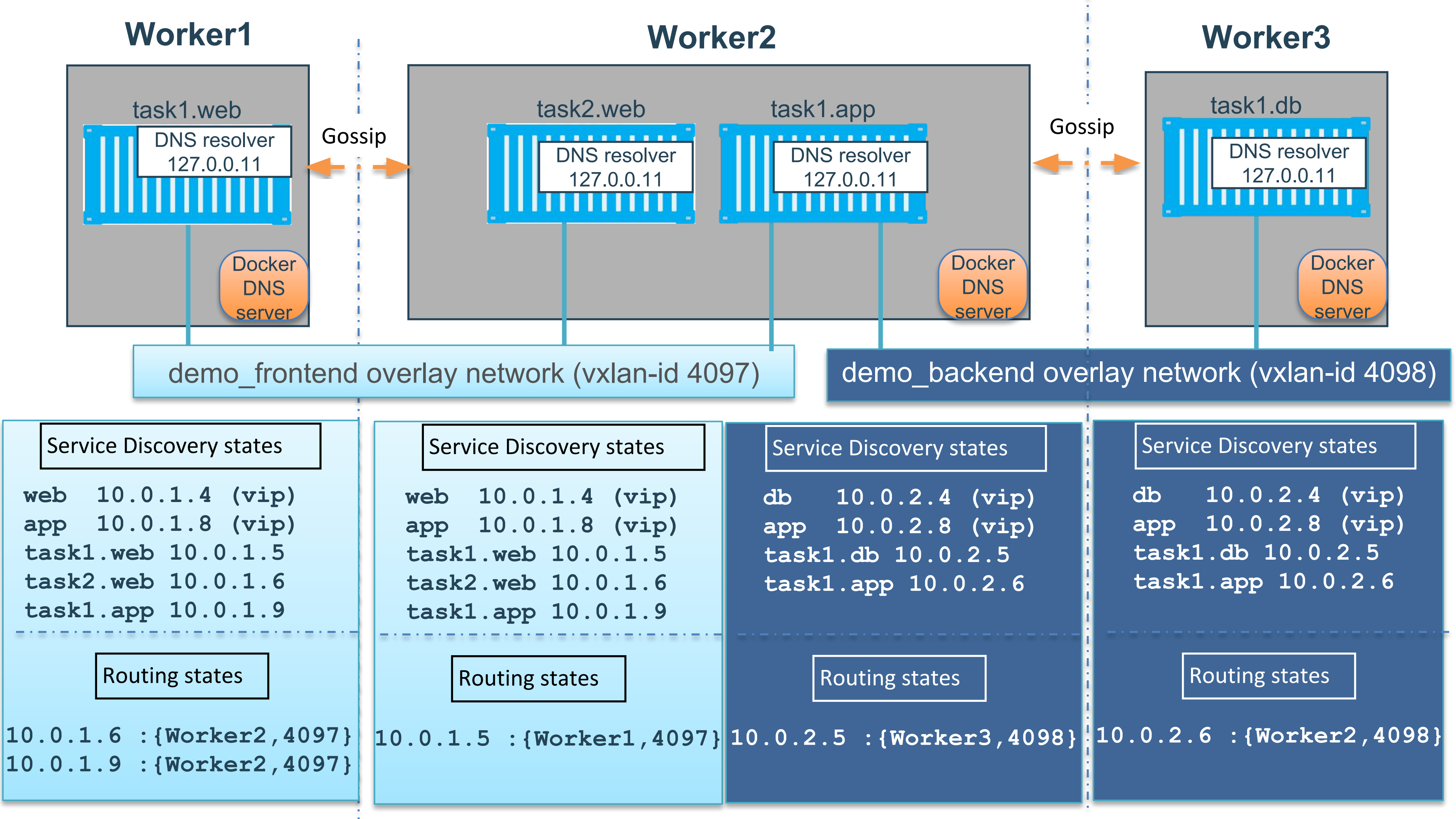
- Manager only operation
- Reserves network resources at mgmt plane such as subnet and vxlan-id. No impact to the data-plane yet.

- Tasks Scheduled to swarm workers
- Network scoped Service Registration on Docker DNS server
  - Service name -> VIP
  - Task name -> Task IP
  - tasks.Service-Name -> All Task IPs
- Exchange SD & LB states via Gossip
- **Prepare Data-plane***
- Call Driver APIs and exchange driver states via Gossip

- **Swarm scope - network resources that are owned and managed centrally by the controllers**
- **Local scope - network resources that are owned and managed by the worker node**

Local scope - bridge, macvlan, host

Swarm scope - overlay



Worker1

task1.web

DNS resolver
127.0.0.11

Docker DNS server

Gossip

Worker2

task2.web

DNS resolver
127.0.0.11

task1.app

DNS resolver
127.0.0.11

Docker DNS server

Gossip

Worker3

task1.db

DNS resolver
127.0.0.11

Docker DNS server

demo_frontend overlay network (vxlan-id 4097)

demo_backend overlay network (vxlan-id 4098)

Service Discovery states

```
web   10.0.1.4 (vip)
app   10.0.1.8 (vip)
task1.web 10.0.1.5
task2.web 10.0.1.6
task1.app 10.0.1.9
```

Routing states

```
10.0.1.6 :{Worker2,4097}
10.0.1.9 :{Worker2,4097}
```

Service Discovery states

```
web   10.0.1.4 (vip)
app   10.0.1.8 (vip)
task1.web 10.0.1.5
task2.web 10.0.1.6
task1.app 10.0.1.9
```

Routing states

```
10.0.1.5 :{Worker1,4097}
```

Service Discovery states

```
db    10.0.2.4 (vip)
app   10.0.2.8 (vip)
task1.db 10.0.2.5
task1.app 10.0.2.6
```

Routing states

```
10.0.2.5 :{Worker3,4098}
```

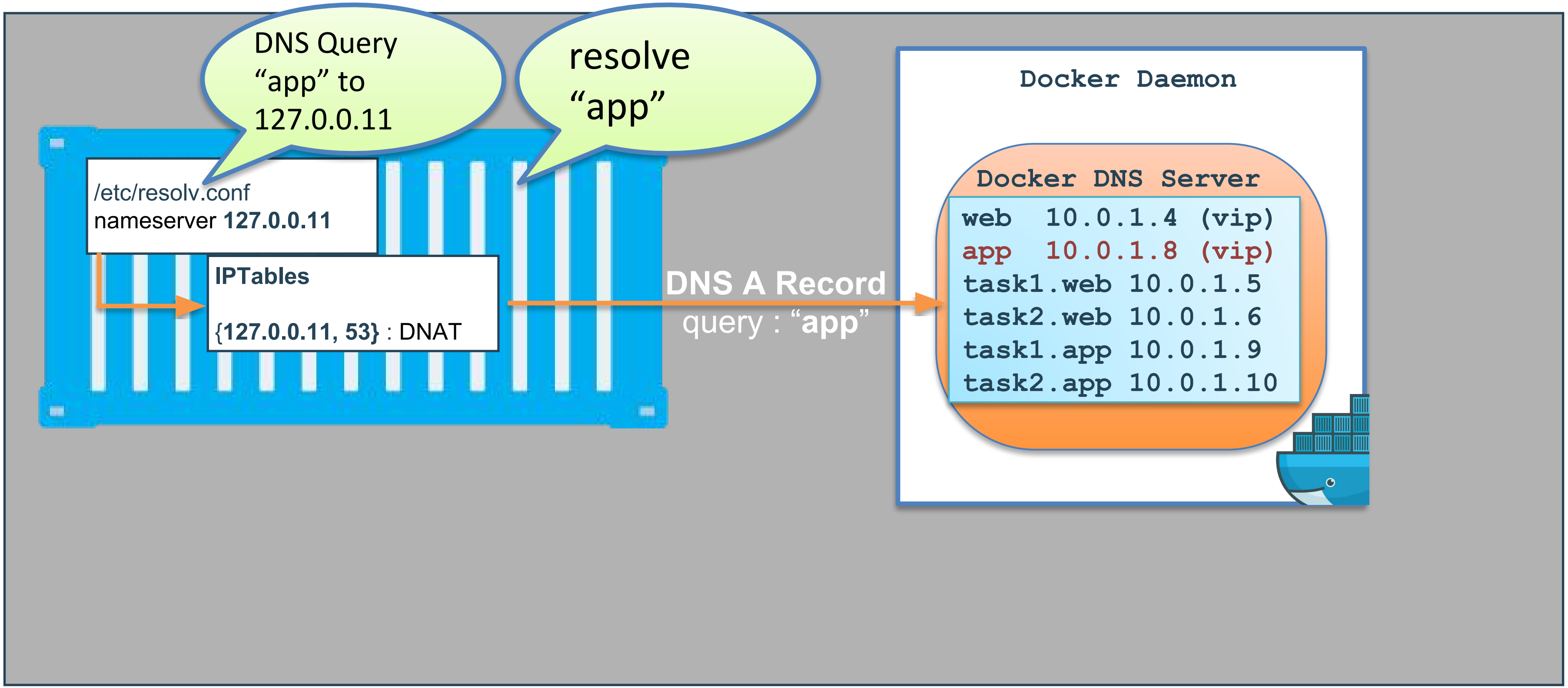
Service Discovery states

```
db    10.0.2.4 (vip)
app   10.0.2.8 (vip)
task1.db 10.0.2.5
task1.app 10.0.2.6
```

Routing states

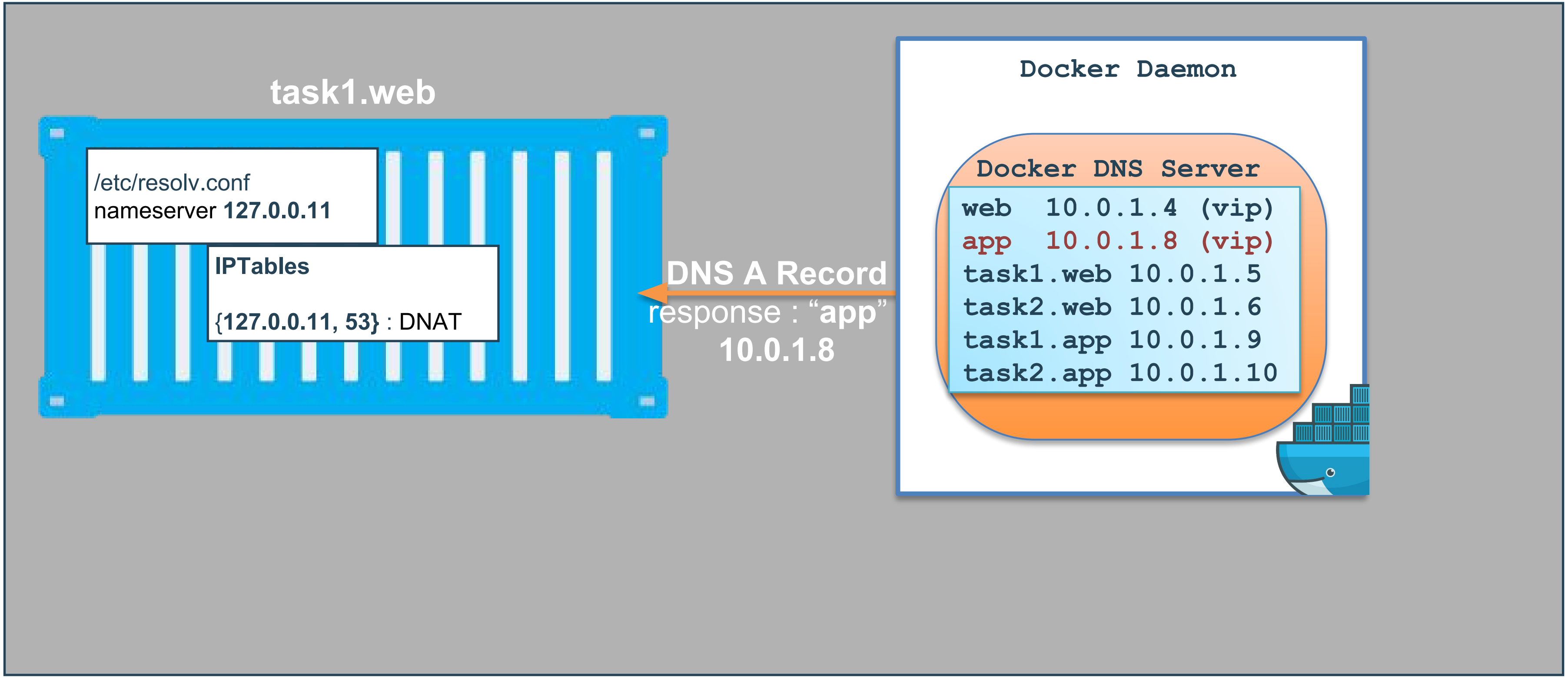
```
10.0.2.6 :{Worker2,4098}
```

Dissecting DNS Lookups



web	10.0.1.4	(vip)
app	10.0.1.8	(vip)
task1.web	10.0.1.5	
task2.web	10.0.1.6	
task1.app	10.0.1.9	
task2.app	10.0.1.10	

Dissecting DNS Lookups



task1.web

```
/etc/resolv.conf
nameserver 127.0.0.11
```

IPTables

{127.0.0.11, 53} : DNAT

DNS A Record
response : “app”
10.0.1.8

Docker Daemon

Docker DNS Server

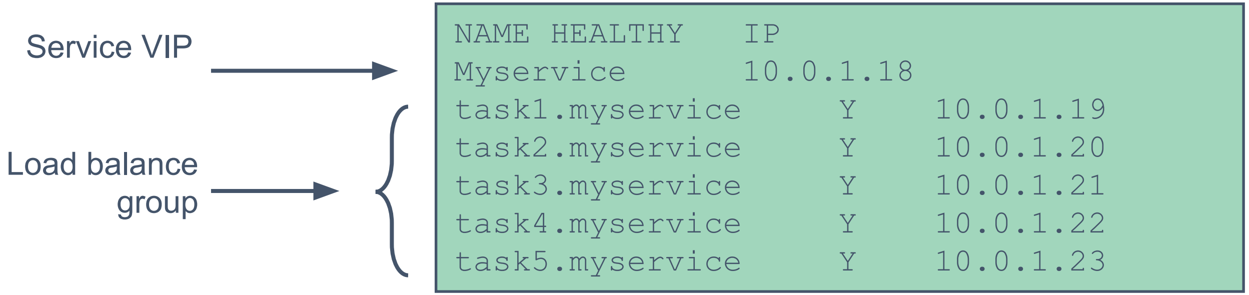
```
web       10.0.1.4 (vip)
app       10.0.1.8 (vip)
task1.web 10.0.1.5
task2.web 10.0.1.6
task1.app 10.0.1.9
task2.app 10.0.1.10
```

Docker Networking

Load Balancing

Internal LB: Service Virtual IP (VIP) Load Balancing

- Every **service** gets a **VIP** when it’s created (stays with the service for its entire life)
- Lookups against the VIP get load-balanced across all **healthy tasks** in the service
- Behind the scenes it uses Linux kernel **IPVS** to perform transport layer load balancing
- `docker inspect <service>` (shows the service VIP)



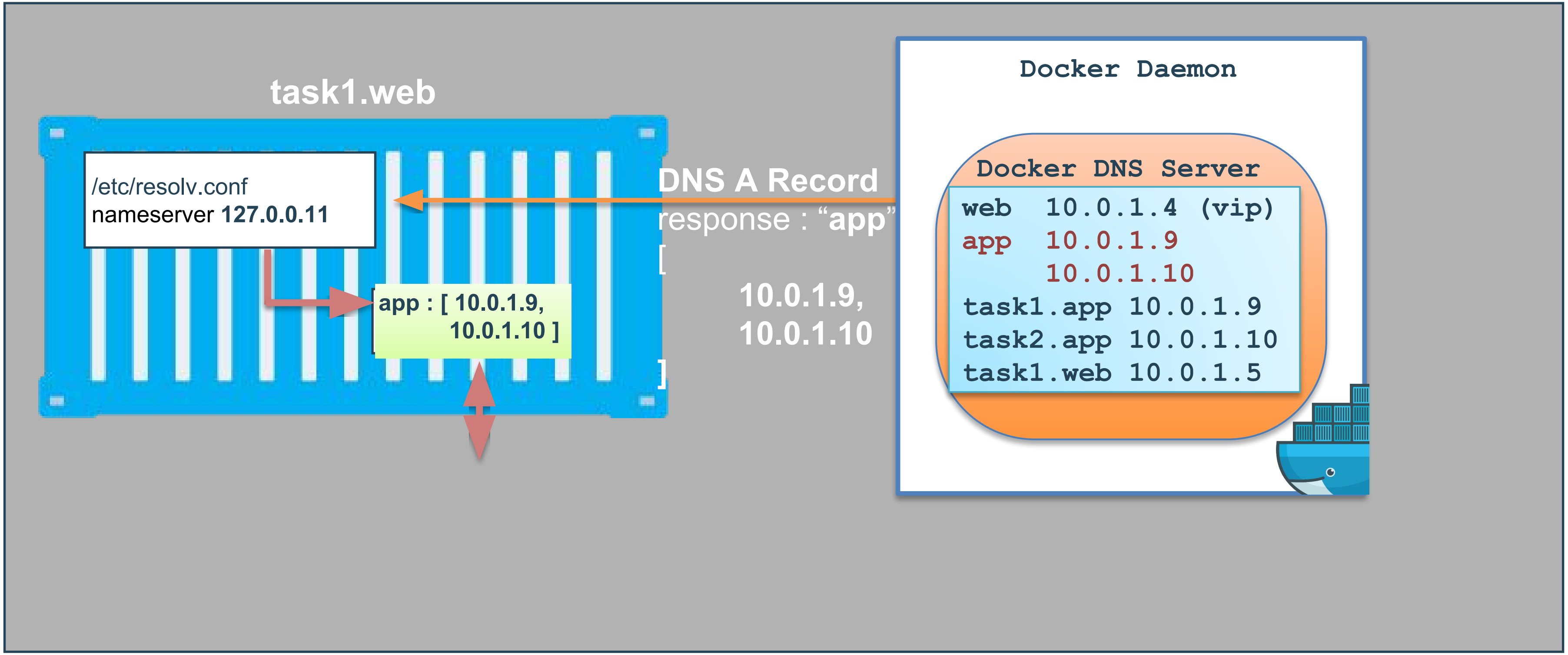
Service VIP

Load balance group

```
NAME HEALTHY   IP
Myservice      10.0.1.18
task1.myservice     Y   10.0.1.19
task2.myservice     Y   10.0.1.20
task3.myservice     Y   10.0.1.21
task4.myservice     Y   10.0.1.22
task5.myservice     Y   10.0.1.23
```

Internal LB: DNS RR Load Balancing

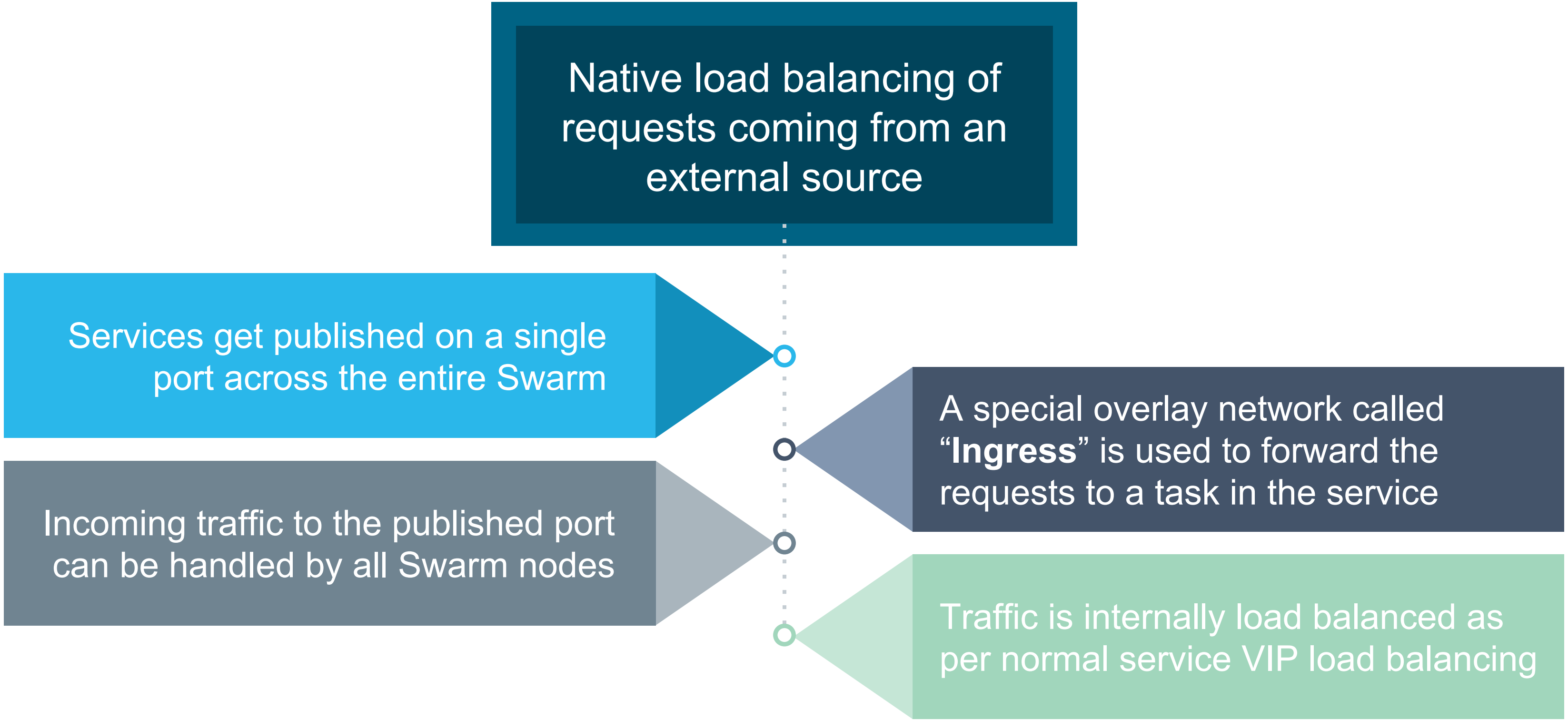
docker service create —name=app **—endpoint-mode=dns-rr** demo/my-app



```
web   10.0.1.4 (vip)
app   10.0.1.9
      10.0.1.10
task1.app 10.0.1.9
task2.app 10.0.1.10
task1.web 10.0.1.5
```

Publishing Services

What is the Routing Mesh?



Native load balancing of requests coming from an external source

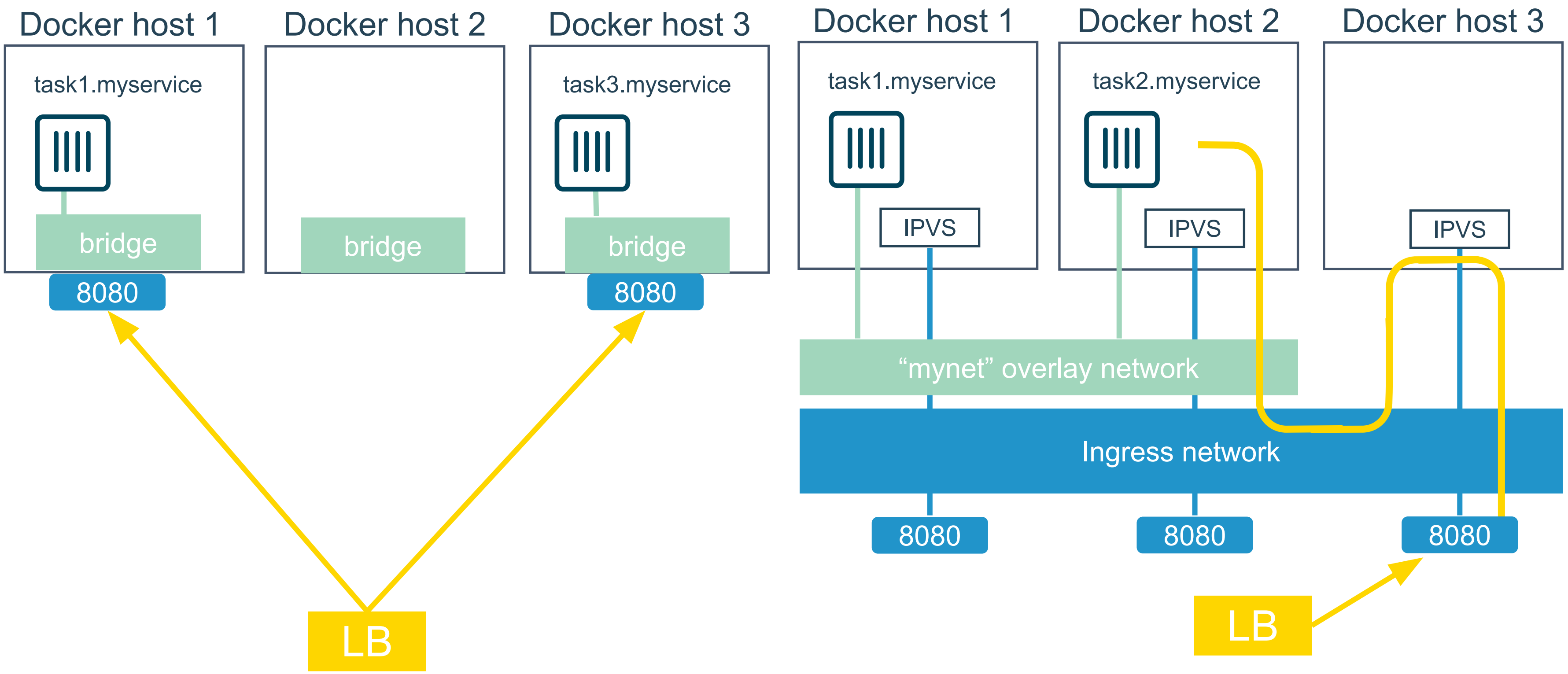
Services get published on a single port across the entire Swarm

A special overlay network called “**Ingress**” is used to forward the requests to a task in the service

Incoming traffic to the published port can be handled by all Swarm nodes

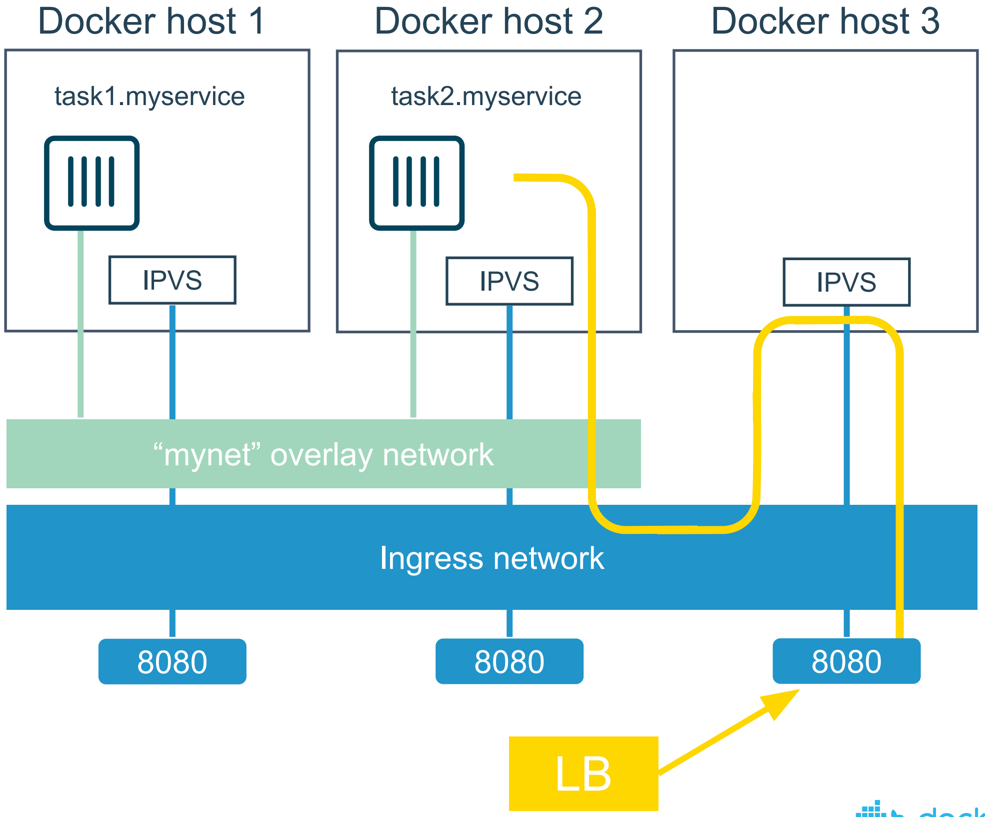
Traffic is internally load balanced as per normal service VIP load balancing

Publish Mode: Host vs Ingress

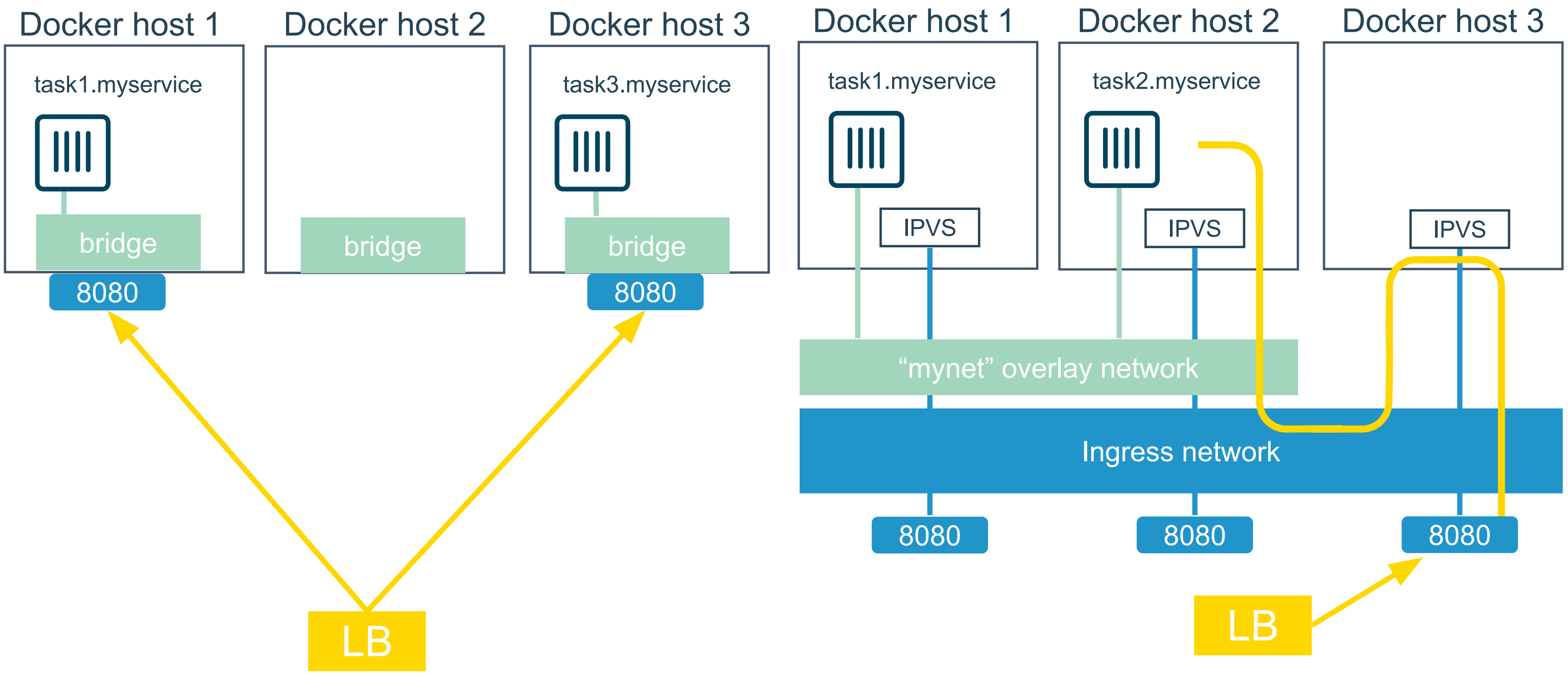


External LB: Ingress Routing Mesh

1. Three Docker hosts
2. New service with 2 tasks
3. Connected to the **mynet** overlay network
4. Service published on port 8080 swarm-wide
5. External LB sends request to Docker host 3 on port 8080
6. Routing mesh forwards the request to a healthy task using the ingress network



Publish Mode: Host vs Ingress



--publish-mode=host

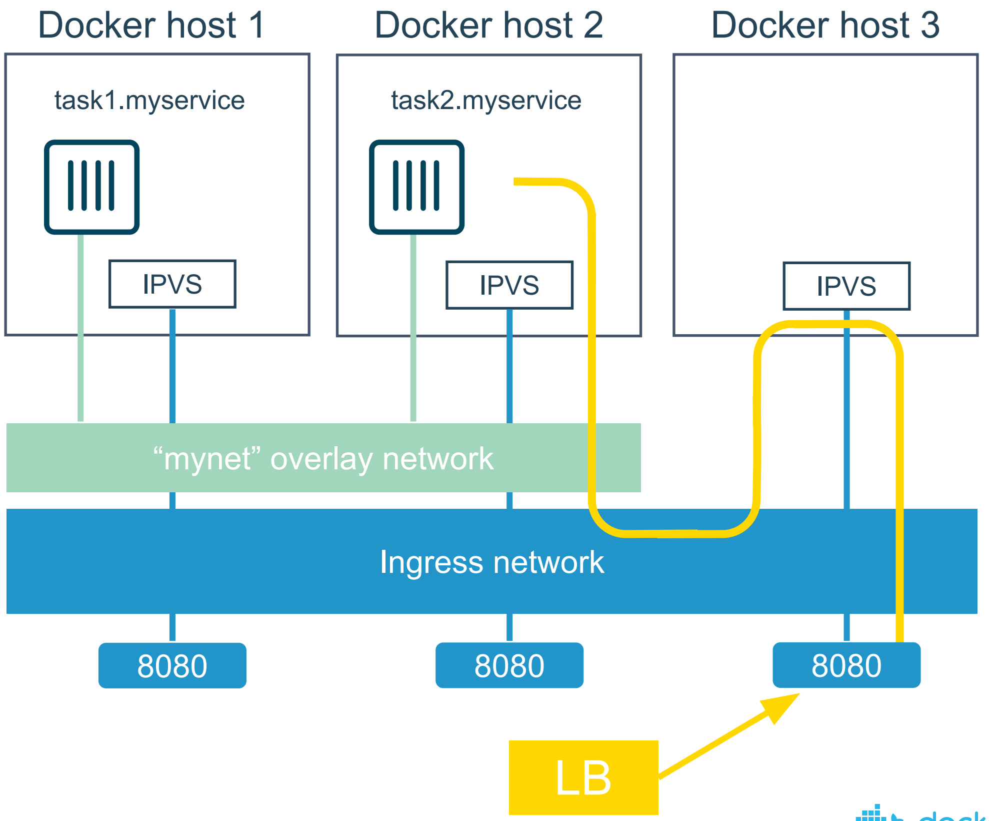
--publish-mode=ingress

Load Balancing External Requests

ROUTING MESH

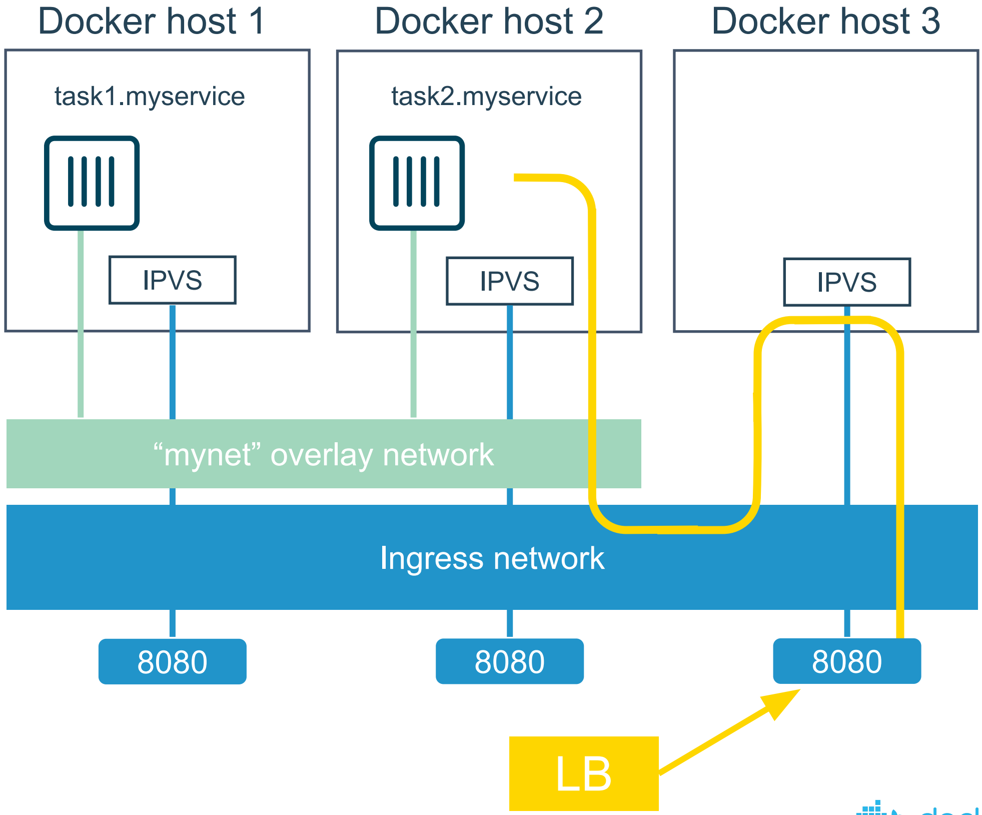
Routing Mesh Example

1. Three Docker hosts
2. New service with 2 tasks
3. Connected to the **mynet** overlay network
4. Service published on port 8080 swarm-wide
5. External LB sends request to Docker host 3 on port 8080
6. Routing mesh forwards the request to a healthy task using the ingress network

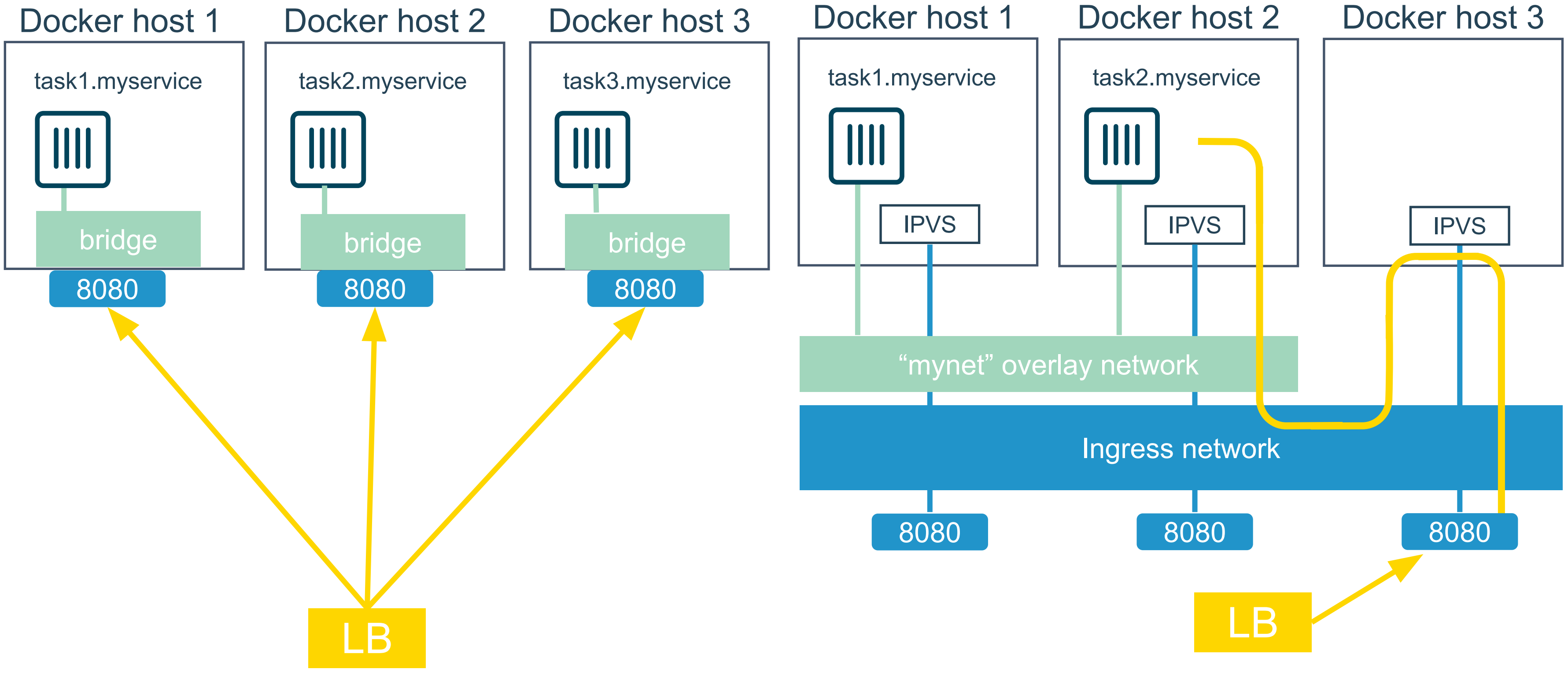


Routing Mesh Example

1. Three Docker hosts
2. New service with 2 tasks
3. Connected to the **mynet** overlay network
4. Service published on port 8080 swarm-wide
5. External LB sends request to Docker host 3 on port 8080
6. Routing mesh forwards the request to a healthy task using the ingress network



Host Mode vs Routing Mesh



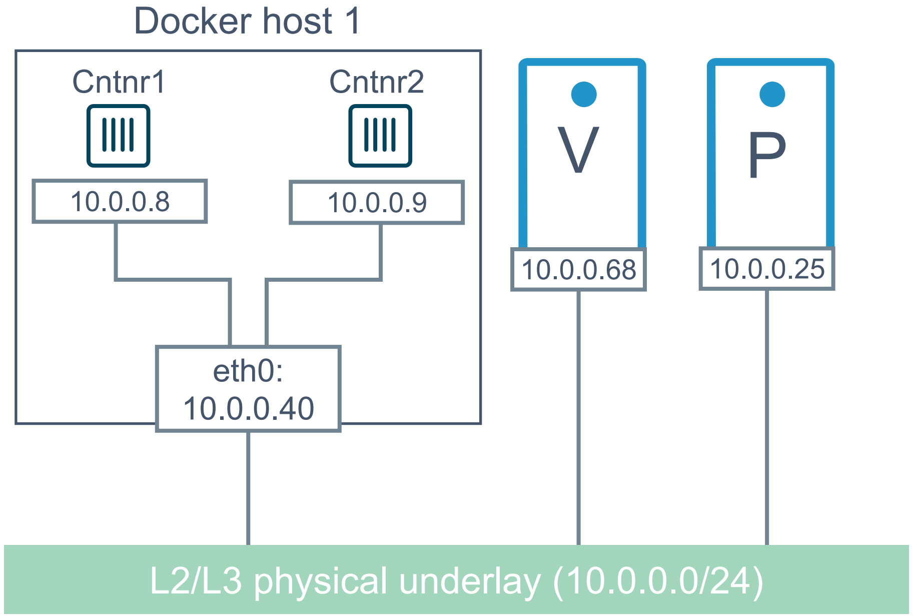
Lab I

BREAK

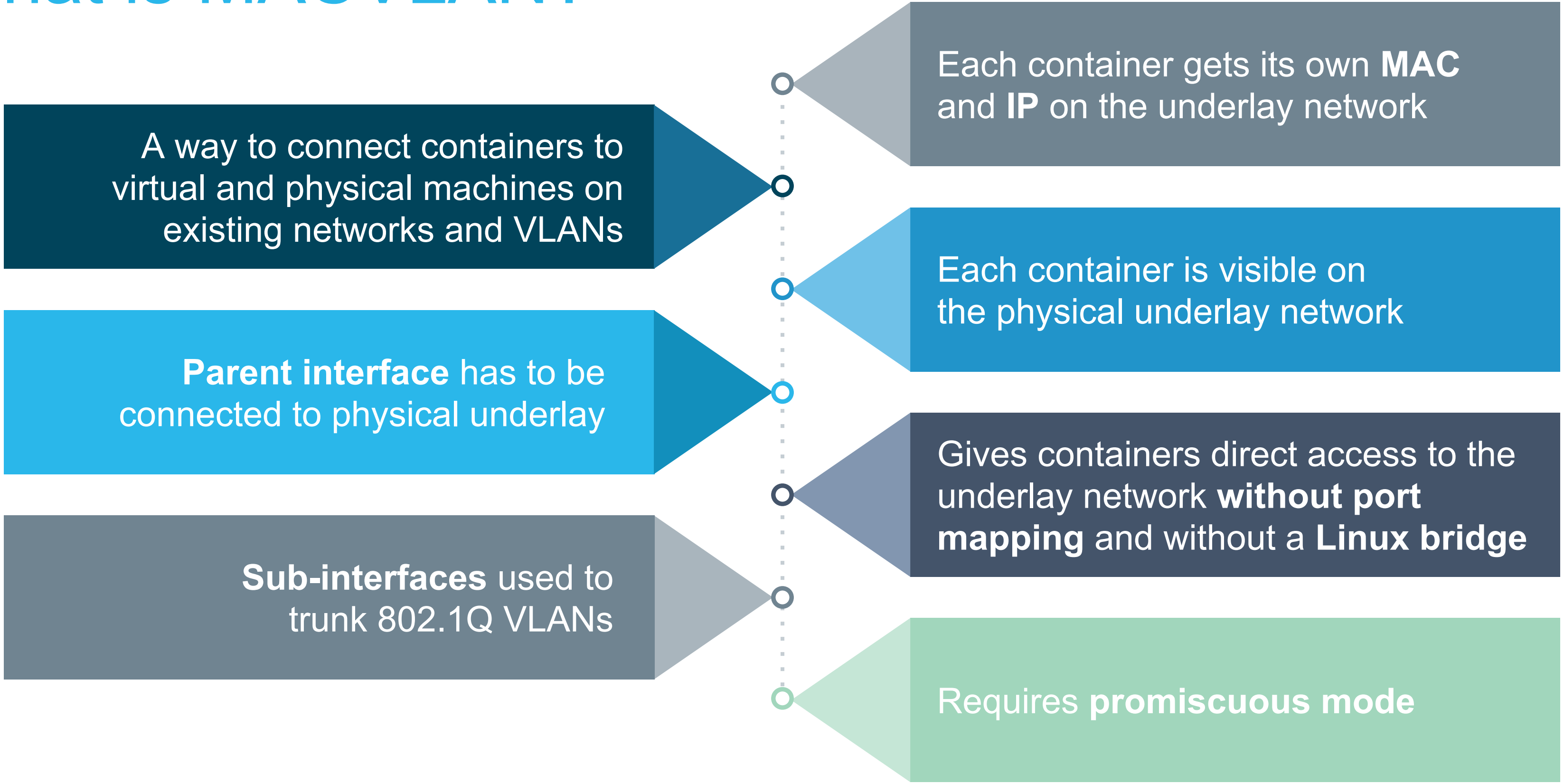
MACVLAN Driver

What is MACVLAN?

- A way to attach containers to existing networks and VLANs
- Ideal for apps that are not ready to be fully containerized
- Uses the well known MACVLAN Linux network type



What is MACVLAN?



Each container gets its own **MAC** and **IP** on the underlay network

A way to connect containers to virtual and physical machines on existing networks and VLANs

Each container is visible on the physical underlay network

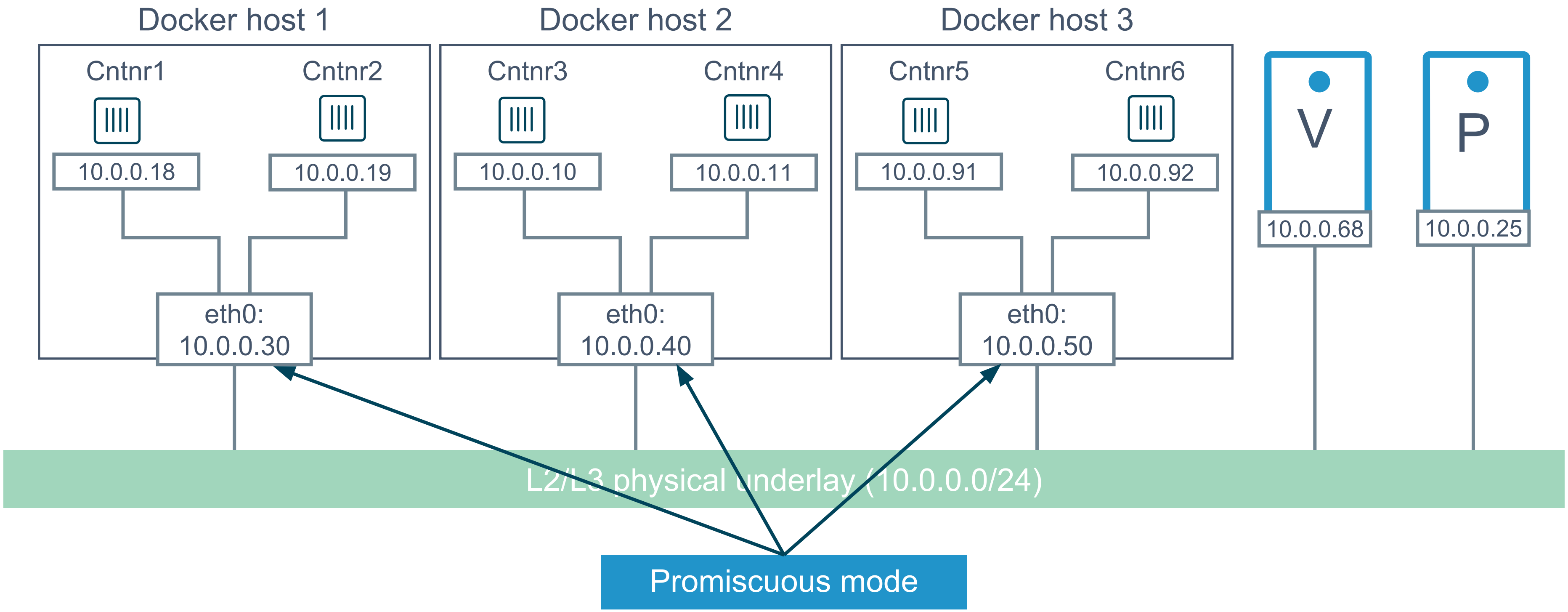
Parent interface has to be connected to physical underlay

Gives containers direct access to the underlay network **without port mapping** and without a **Linux bridge**

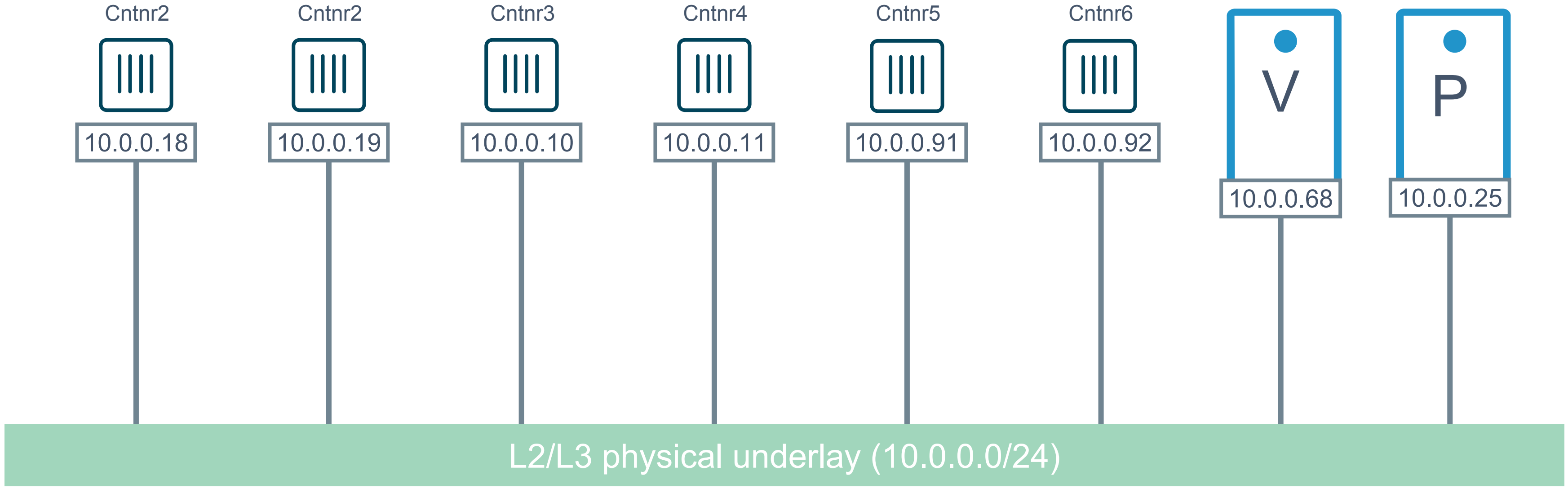
Sub-interfaces used to trunk 802.1Q VLANs

Requires **promiscuous mode**

What is MACVLAN?

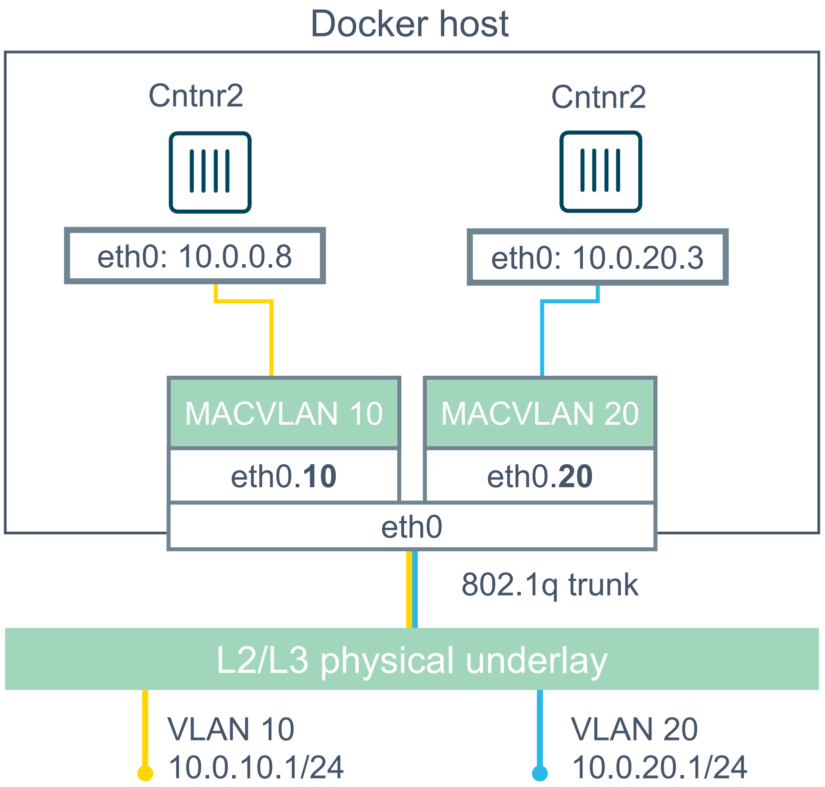


What is MACVLAN?



MACVLAN and Sub-interfaces

- MACVLAN uses **sub-interfaces** to process 802.1Q VLAN tags.
- In this example, two sub-interfaces are used to enable two separate VLANs
- Yellow lines represent VLAN 10
- Blue lines represent VLAN 20



MACVLAN Summary

- Allow containers to be plumbed into existing VLANs
- Ideal for integrating containers with existing networks and apps
- High performance (no NAT or Linux bridge…)
- Every container gets its own **MAC** and **routable IP** on the physical underlay
- Uses **sub-interfaces** for 802.1q VLAN tagging
- Requires **promiscuous** mode!

MACVLAN Summary

- Allow containers to be plumbed into existing VLANs
- Ideal for integrating containers with existing networks and apps
- High performance (no NAT or Linux bridge…)
- Every container gets its own **MAC** and **routable IP** on the physical underlay
- Uses **sub-interfaces** for 802.1q VLAN tagging
- Requires **promiscuous** mode!

Docker Network Design Best Practices

General Networking Design Guidelines

- Make sure to have the required Docker EE TCP/UDP ports open
- Place managers close to each other. Latency can impact raft traffic.
- Pick the right subnet based on the application requirement.
- Dedicate subnets from the underly to be used as overlay subnets
- Monitor your network (rtt, packets drops, tcp retransmits)
- Use Labels!

Networking Reference Architecture

- https://success.docker.com/article/Docker_Reference_Architecture-_Designing_Scalable,_Portable_Docker_Container_Networks

Docker Network Troubleshooting

Common Network Issues

Blocked ports, ports required to be open for network mgmt, control, and data plane

Iptables issues

Used extensively by Docker Networking, must not be turned off

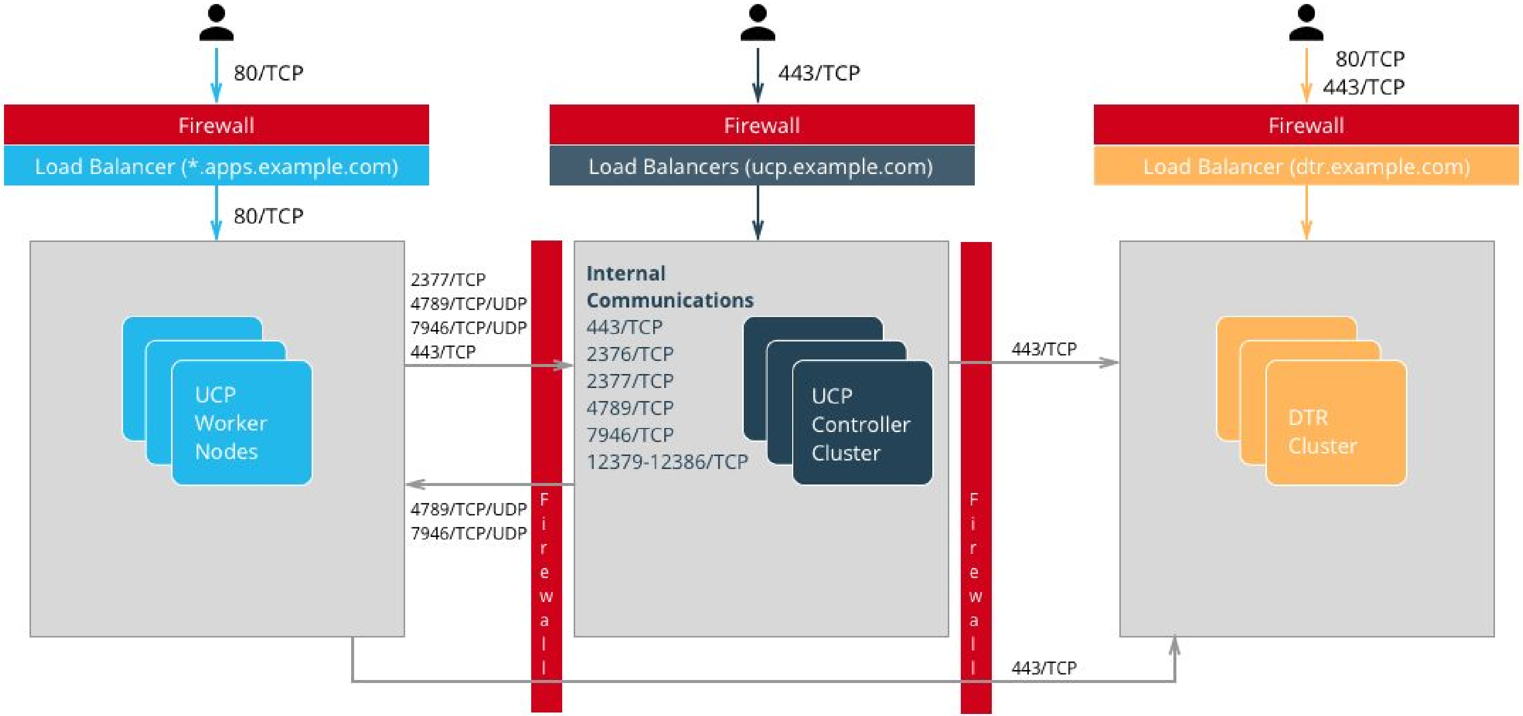
List rules with $ iptables -S, $ iptables -S -t nat

Network state information stale or not being propagated

Destroy and create networks again with same name

General connectivity problems

Required Ports



General Connectivity Issues

Network always gets blamed first :(

Eliminate or prove connectivity first, connectivity can be broken at service discovery or network level

Service Discovery

Test service name resolution or container name resolution

```
drill <service name> (returns the service VIP DNS record)
```

```
drill tasks.<service name> (returns all task DNS records)
```

Network Layer

Test reachability using VIP or container IP

```
task1$ nc -l 5000, task2$ nc <service ip> 5000
```

```
ping <container ip>
```

Netshoot Tool

Has most of the tools you need **<u>in a container</u>** to troubleshoot common networking problems

```
iperf, tcpdump, netstat, iftop, drill, netcat-openbsd, iproute2,
util-linux(nsenter), bridge-utils, iputils, curl, ipvsadmin, ethtool…
```

Two Uses

Connect it to a specific **network namespace** (such as a container's) to view the network from that container's perspective

Connect it to a **docker network** to test connectivity on that network

Netshoot Tool

Connect to a container namespace

```
docker run -it --net container:<container_name> nicolaka/netshoot
```

Connect to a network

```
docker run -it --net host nicolaka/netshoot
```

Once inside the **netshoot** container, you can use any of the network troubleshooting tools that come with it

Network Troubleshooting Tools

Capture all traffic to/from port 999 on eth0 on a myservice container

```
docker run -it --net
container:myservice.1.0qlf1kaka0cq38gojf7wcatoa  nicolaka/netshoot
tcpdump -i eth0 port 9999 -c 1 -Xvv
```

See all network connections to a specific task in myservice

```
docker run -it --net
container:myservice.1.0qlf1kaka0cq38gojf7wcatoa  nicolaka/netshoot
netstat -taupn
```

Network Troubleshooting Tools

Test DNS service discovery from one service to another

```
docker run -it --net
container:myservice.1.bil2mo8inj3r9nyrss1g15qav  nicolaka/netshoot drill
yourservice
```

Show host routing table from inside the netshoot container

```
docker run -it --net host nicolaka/netshoot ip route show
```

Lab 2

THANK YOU