Dockercon 2017
Networking Workshop

Mark Church, Technical Account Manager @ Docker

Lorenzo Fontana, Docker Captain

Nico Kabar, Solutions Architect @ Docker

Agenda

1. Fundamentals & Network Drivers
2. Bridge Driver
3. Overlay Driver
4. MACVLAN Driver
5. Network Services: Service Discovery and Load Balancing
6. Network Design
7. Network Troubleshooting
8. Deep Dive: Network Namespaces, iptables, and VXLAN

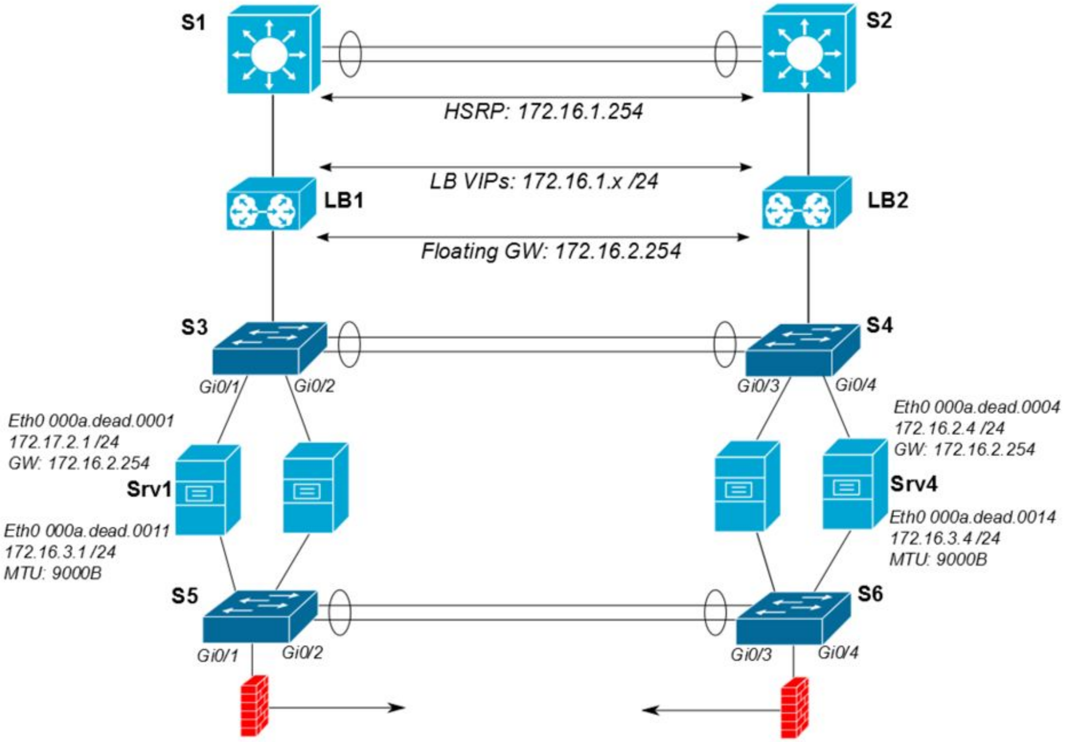
The Container Network Model (CNM)

Networking is hard!

- Distributed in nature

- Many discrete components that are managed and configured differently

- Services that need to be deployed uniformly across all of these discrete components

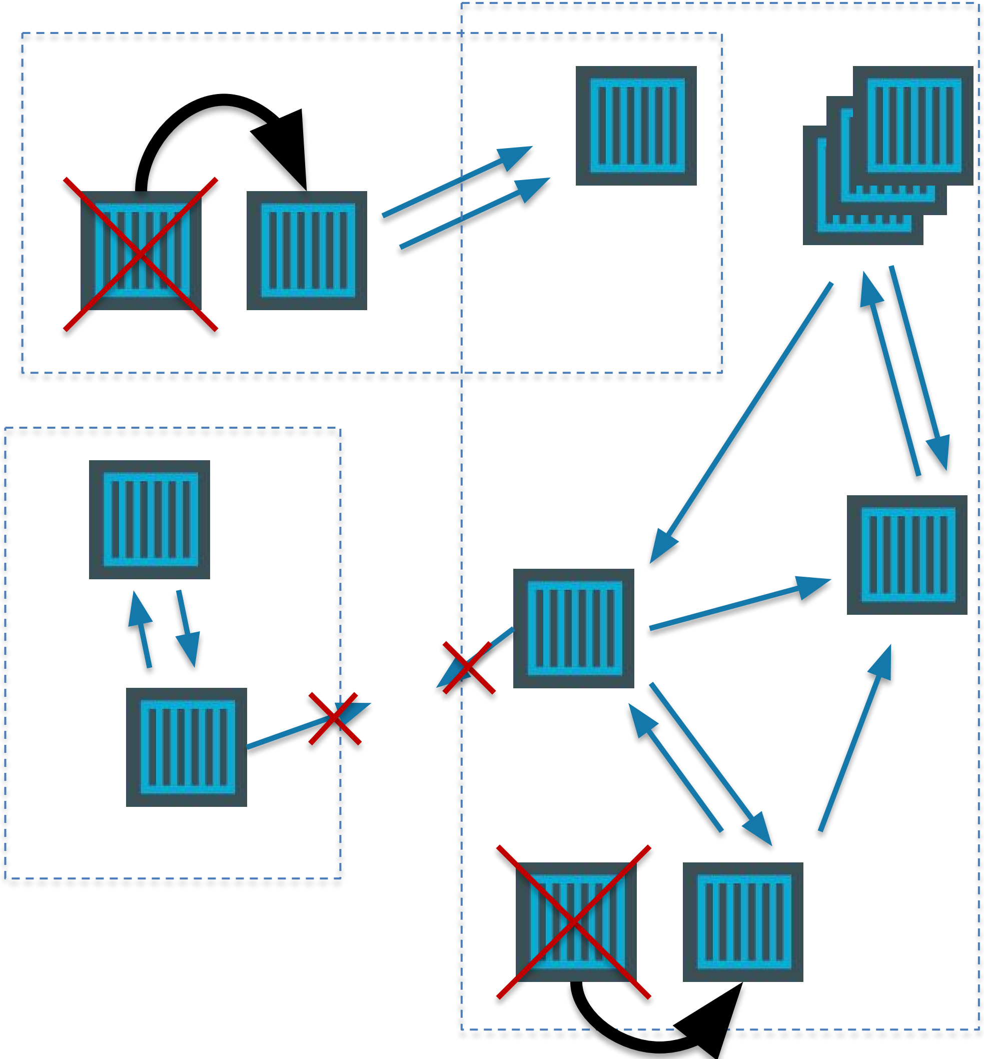


docker

Enter containers …

- 100s or 1000s of containers per host
- Containers that exist for minutes or months
- Microservices distributed across many more hosts (>>> E-W traffic)

… this is worse.



Docker Networking Design Philosophy

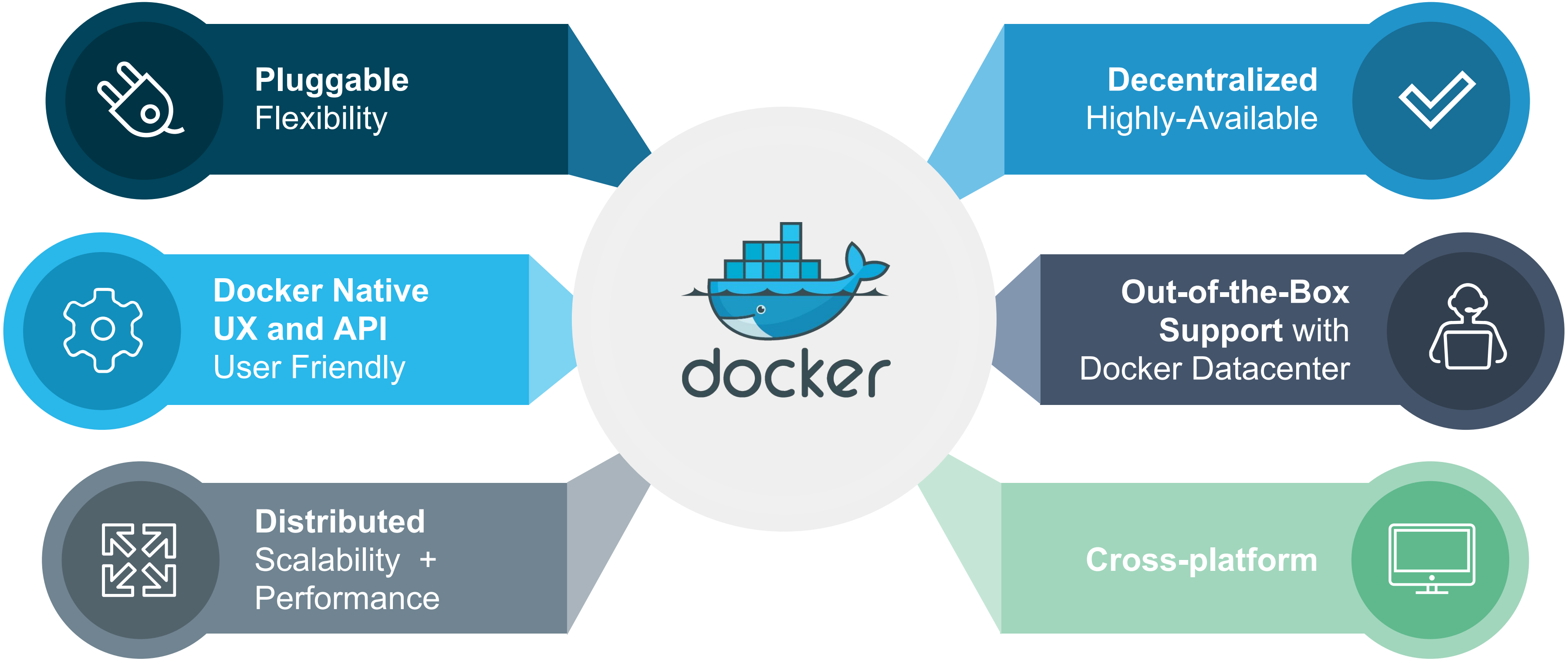
Put Users First

Developers and Operations

Plugin API Design

Batteries included but removable

Docker Networking Goals



Pluggable
Flexibility

Docker Native UX and API
User Friendly

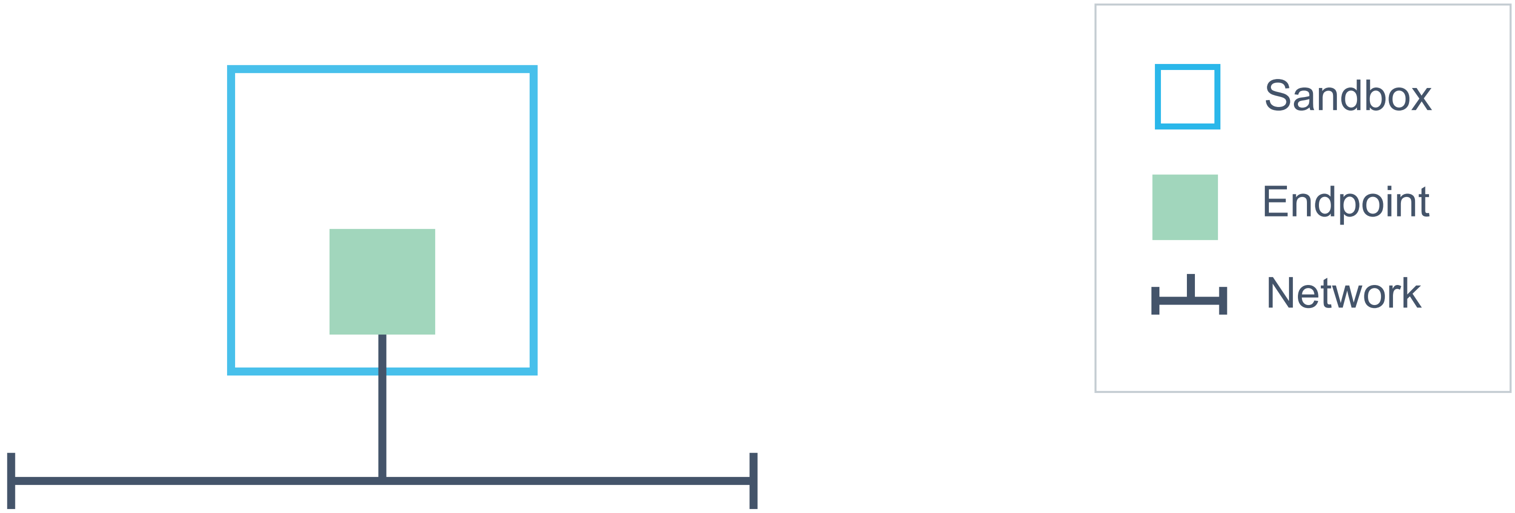
Distributed
Scalability + Performance

Decentralized
Highly-Available

Out-of-the-Box Support with Docker Datacenter

Cross-platform

Container Network Model (CNM)



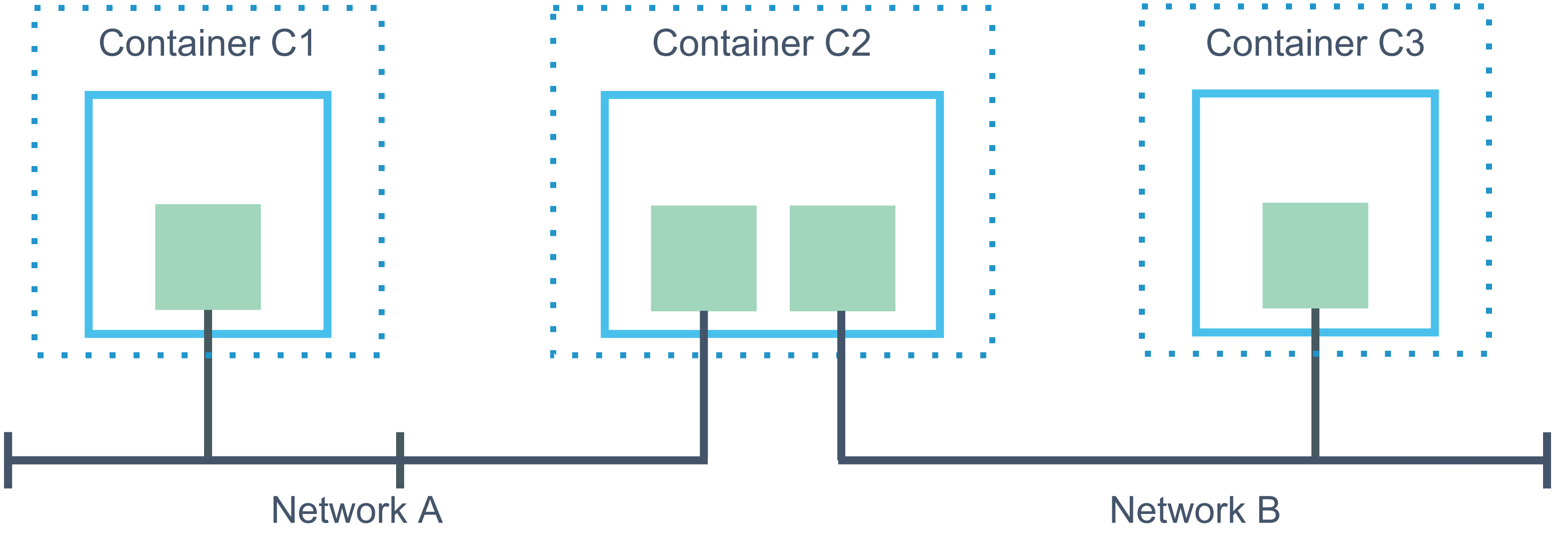
□ Sandbox

🟩 Endpoint

┴ Network

Containers and the CNM

□ Sandbox 🟩 Endpoint ┴ Network ⬚ Container



Container C1

Container C2

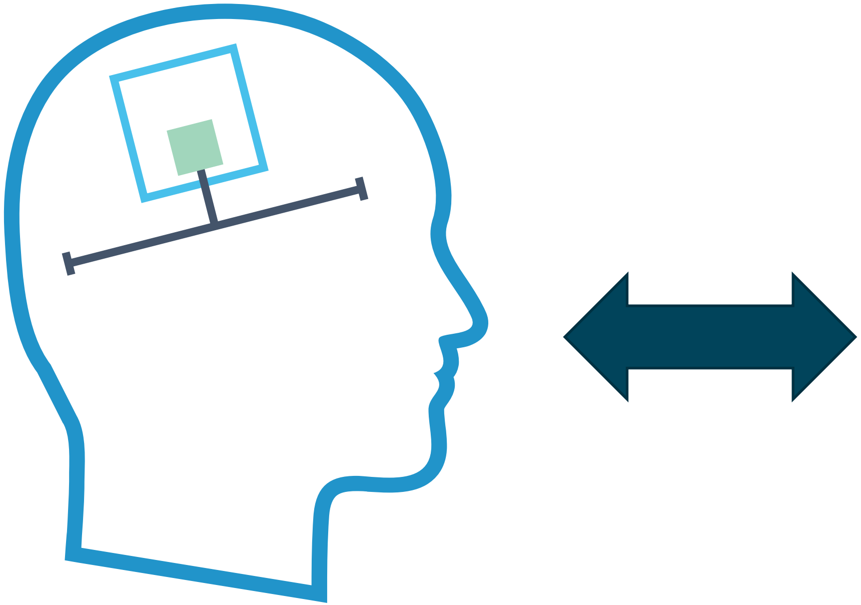
Container C3

Network A

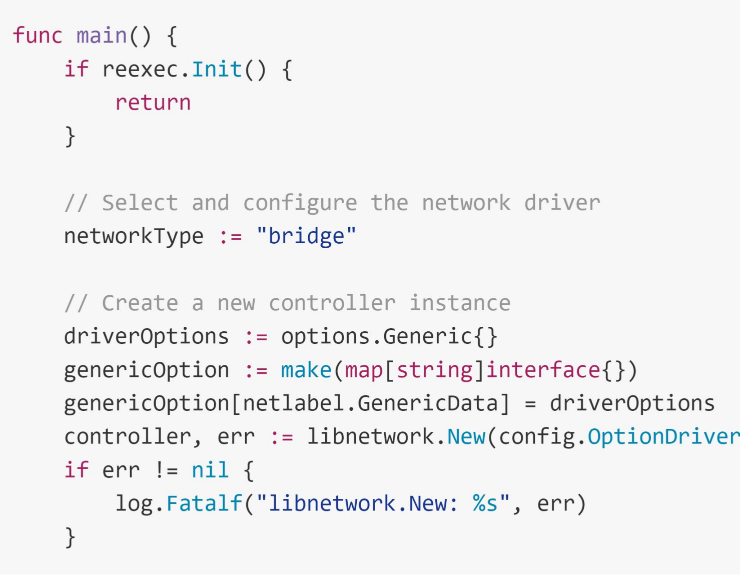
Network B

What is Libnetwork?

Libnetwork is Docker’s native implementation of the CNM



CNM



```
func main() {
    if reexec.Init() {
        return
    }

    // Select and configure the network driver
    networkType := "bridge"

    // Create a new controller instance
    driverOptions := options.Generic{}
    genericOption := make(map[string]interface{})
    genericOption[netlabel.GenericData] = driverOptions
    controller, err := libnetwork.New(config.OptionDriver
    if err != nil {
        log.Fatalf("libnetwork.New: %s", err)
    }
```

Libnetwork

What is Libnetwork?

Docker’s native implementation of the CNM

Provides built-in service discovery and load balancing

Library containing everything needed to create and manage container networks

Provides a consistent versioned API

Pluggable model (native and remote/3rd party drivers)

Multi-platform, written in Go, open source

Libnetwork and Drivers

Libnetwork has a pluggable driver interface

Drivers are used to implement different networking technologies

Built-in drivers are called local drivers, and include:
bridge, host, overlay, MACVLAN

3rd party drivers are called remote drivers, and include:
Calico, Contiv, Kuryr, Weave…

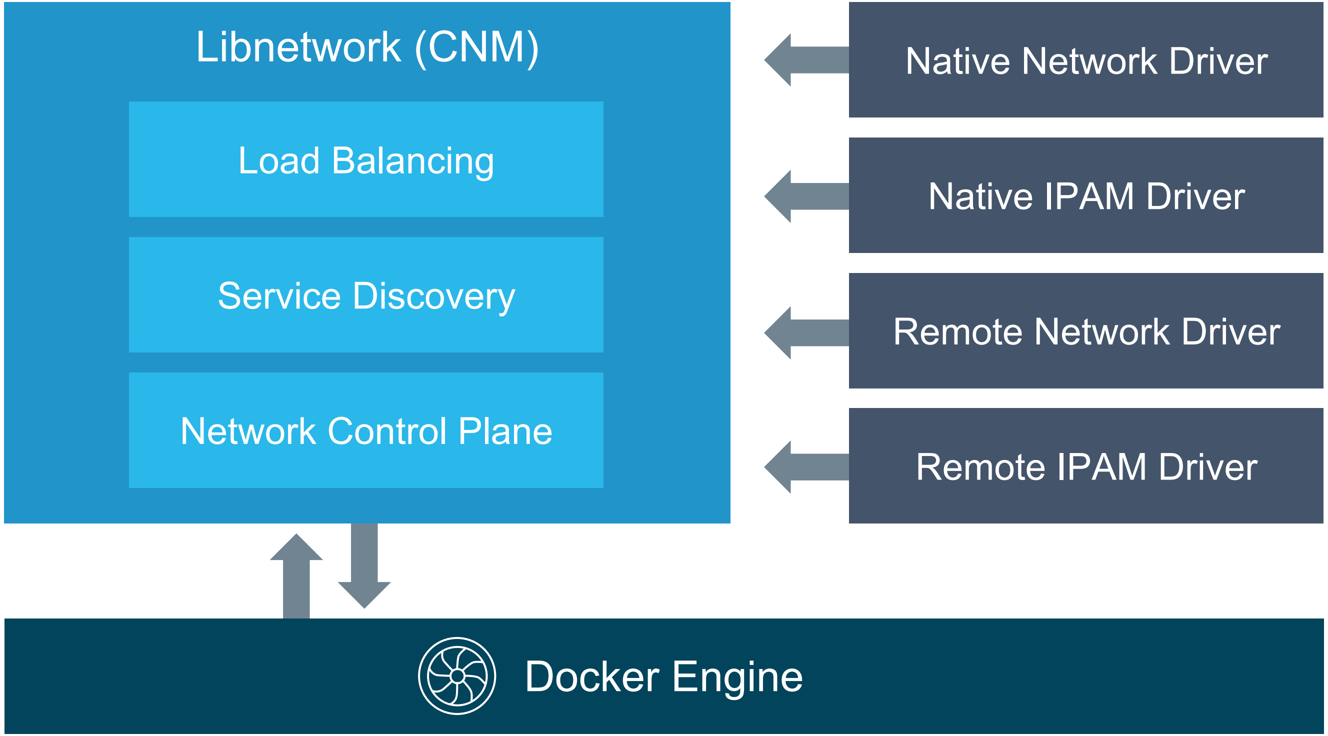
Libnetwork also supports pluggable IPAM drivers

Show Registered Drivers

```
$ docker info

Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 2
<snip>
Plugins:
 Volume: local
 Network: null bridge host overlay
...
```

Libnetwork Architecture



Libnetwork (CNM)

Load Balancing

Service Discovery

Network Control Plane

Native Network Driver

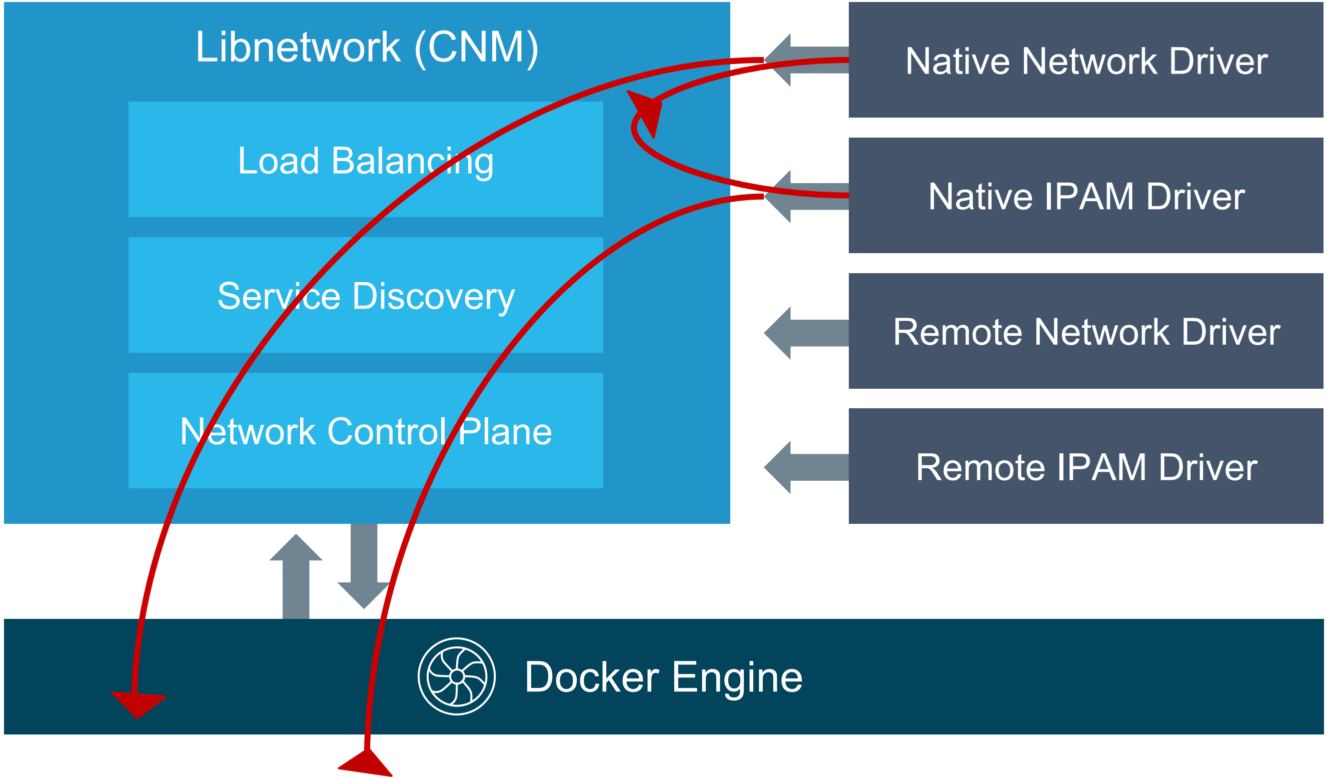
Native IPAM Driver

Remote Network Driver

Remote IPAM Driver

Docker Engine

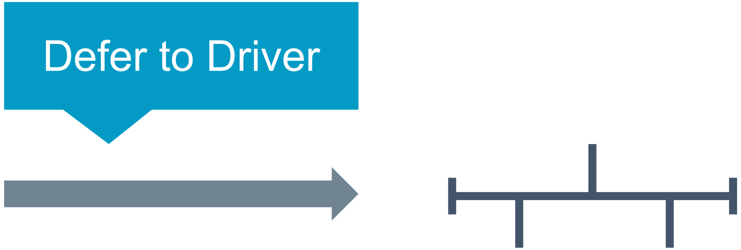
Libnetwork Communication Flow



```
docker network create -d overlay ov
```

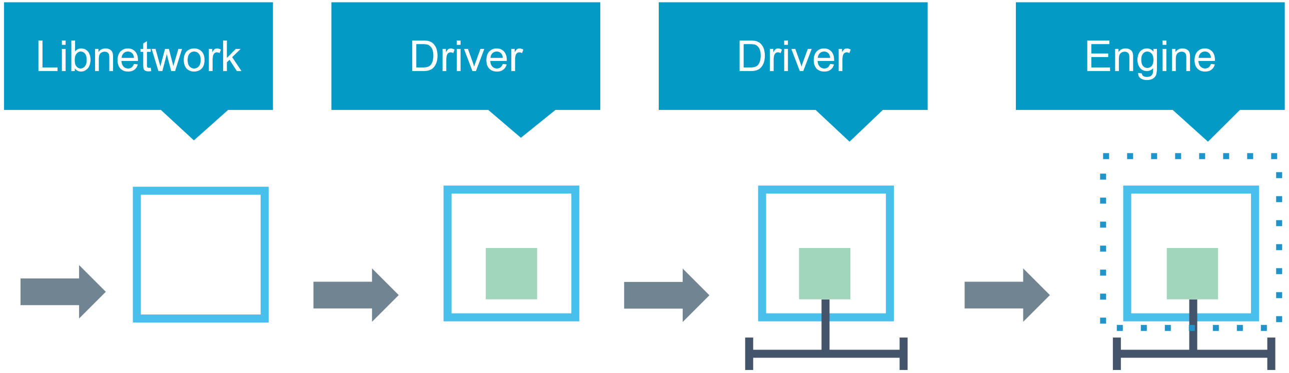
Networks and Containers

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



Defer to Driver

```
docker run --network
…
```



Libnetwork

Driver

Driver

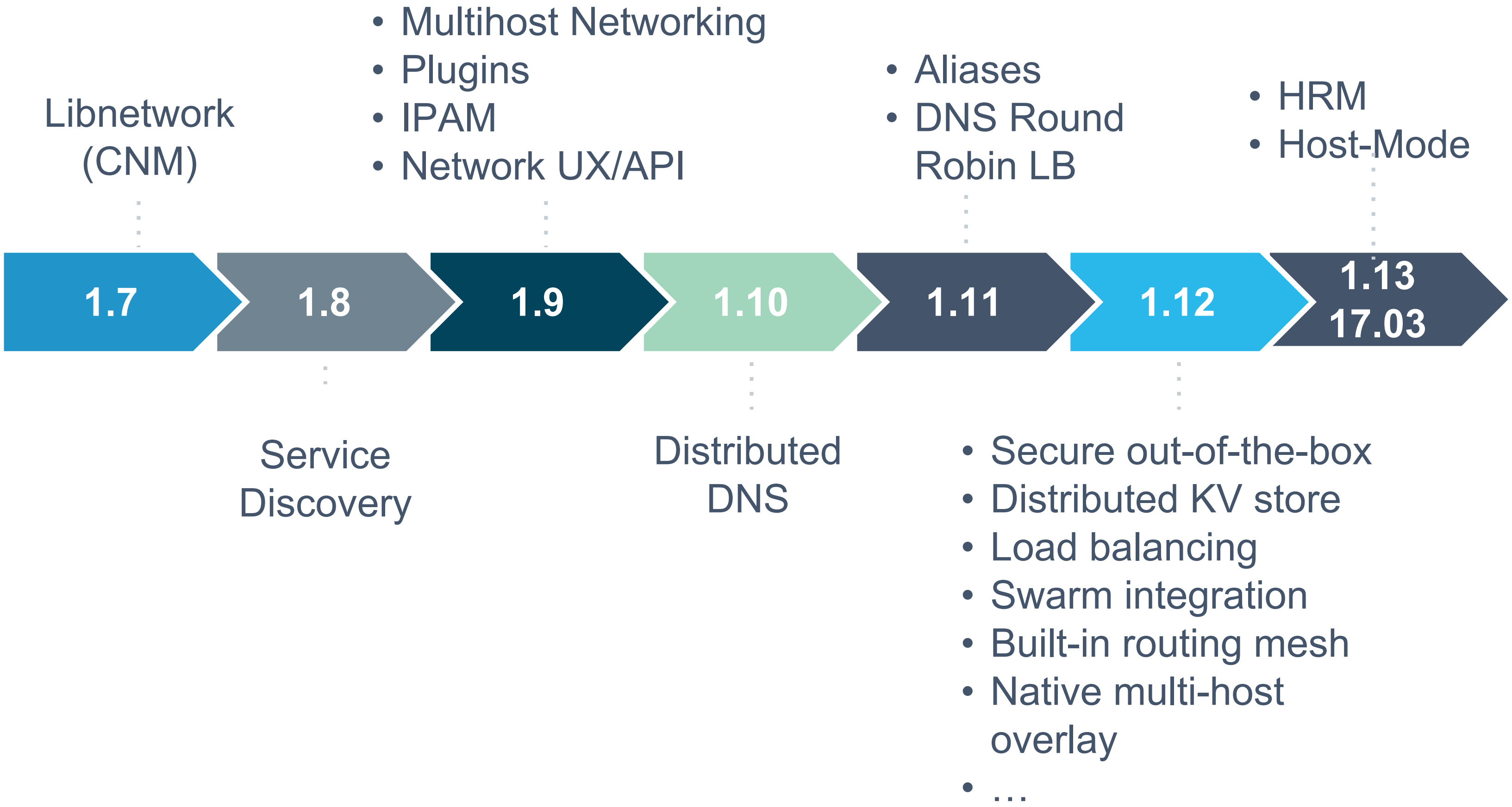
Engine

Detailed Overview: Summary

- The CNM is an open-source container networking specification contributed to the community by Docker, Inc.
- The CNM defines sandboxes, endpoints, and networks
- Libnetwork is Docker's implementation of the CNM
- Libnetwork is extensible via pluggable drivers
- Drivers allow Libnetwork to support many network technologies
- Libnetwork is cross-platform and open-source

The CNM and Libnetwork **simplify** container networking and improve **application portability**

Docker Networking Fundamentals



1.7

Libnetwork (CNM)

1.8

Service Discovery

1.9

- Multihost Networking
- Plugins
- IPAM
- Network UX/API

1.10

Distributed DNS

1.11

- Aliases
- DNS Round Robin LB

1.12

- Secure out-of-the-box
- Distributed KV store
- Load balancing
- Swarm integration
- Built-in routing mesh
- Native multi-host overlay
- …

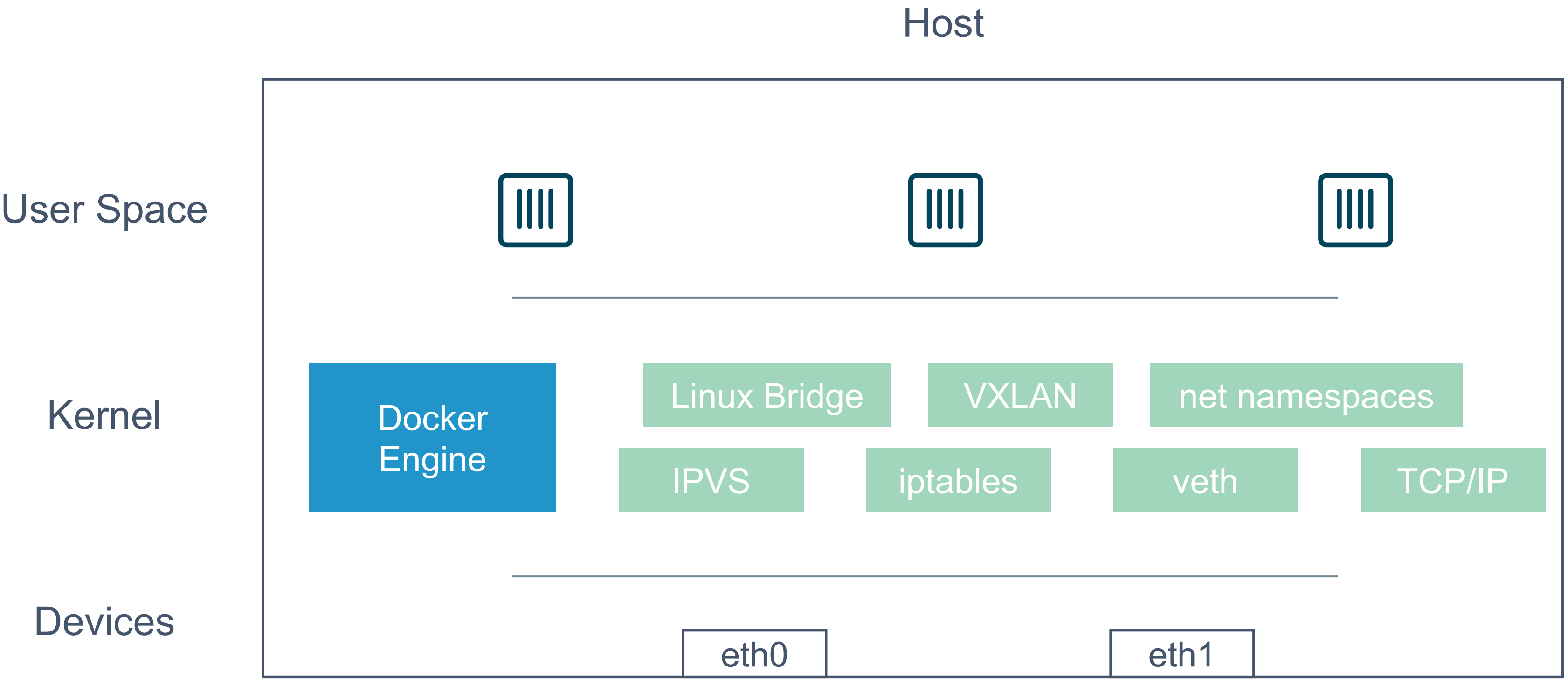
1.13
17.03

- HRM
- Host-Mode

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…

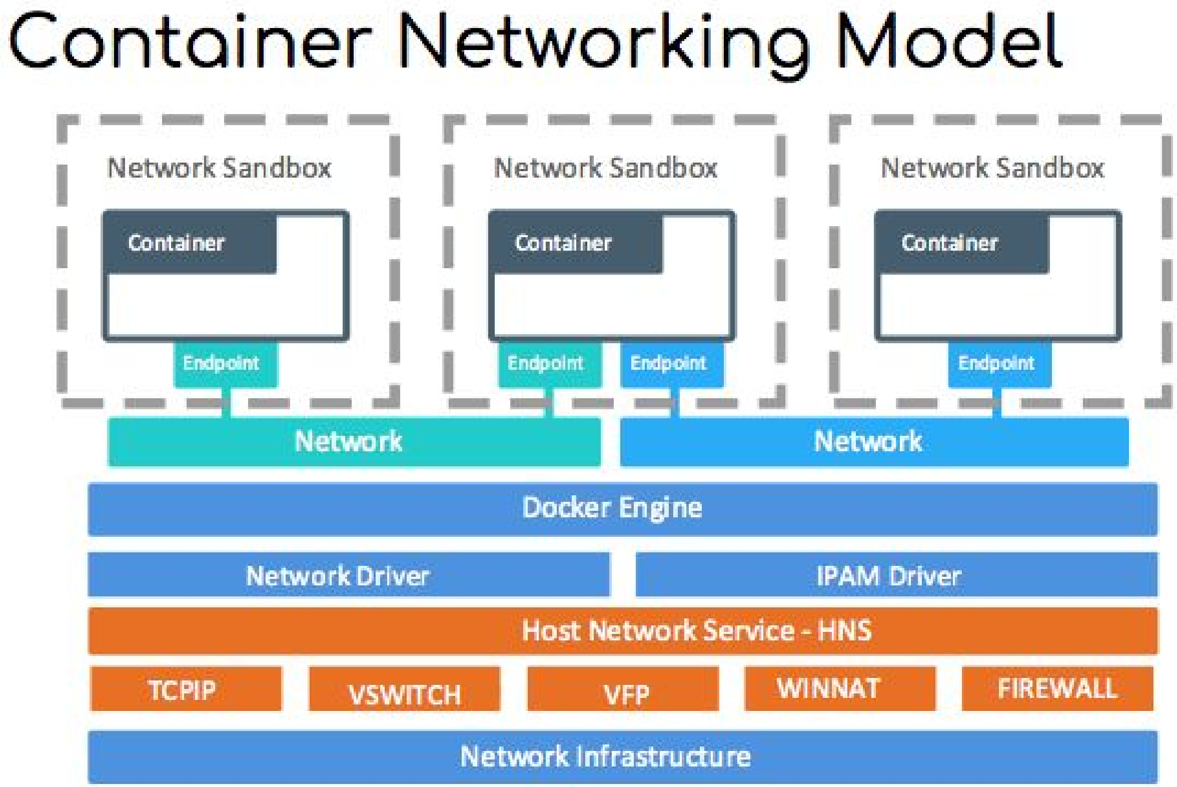
Docker Networking *is* Linux (and Windows) Networking



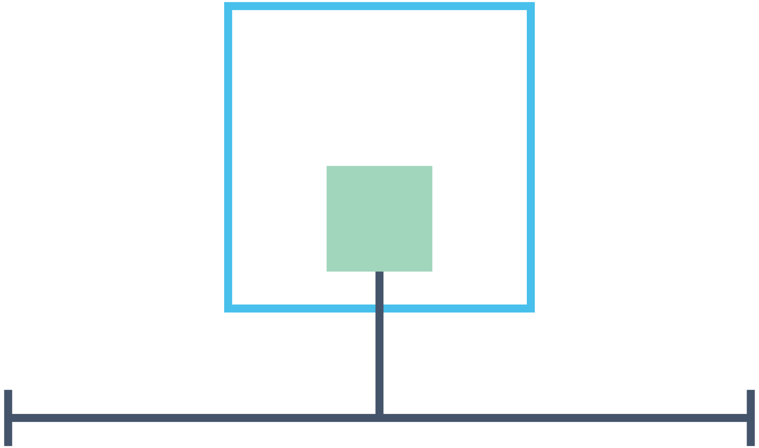
Docker Networking on Linux and Windows

Linux	Windows
Network Namespace	Network Compartments
Linux Bridge	VSwitch
Virtual Ethernet Devices	Virtual nics
IP Tables	Firewall & VFP Rules

Docker Windows Networking



Container Network Model (CNM)



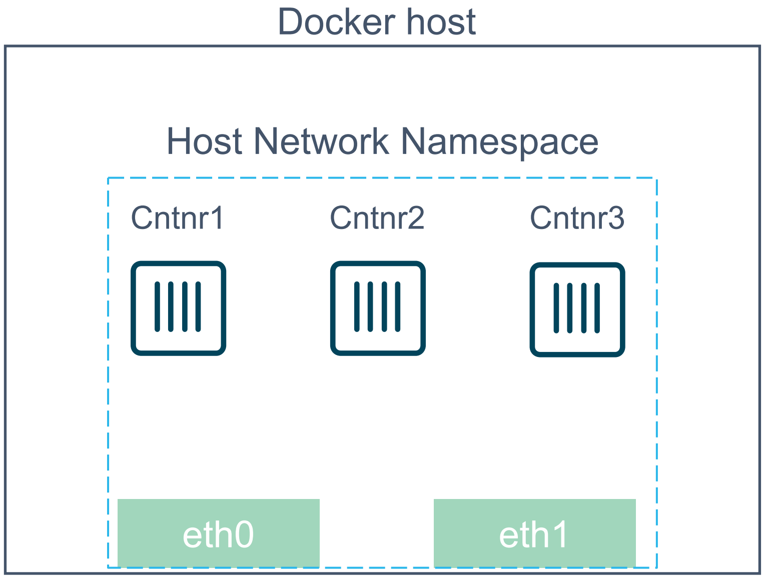
□ Sandbox

🟩 Endpoint

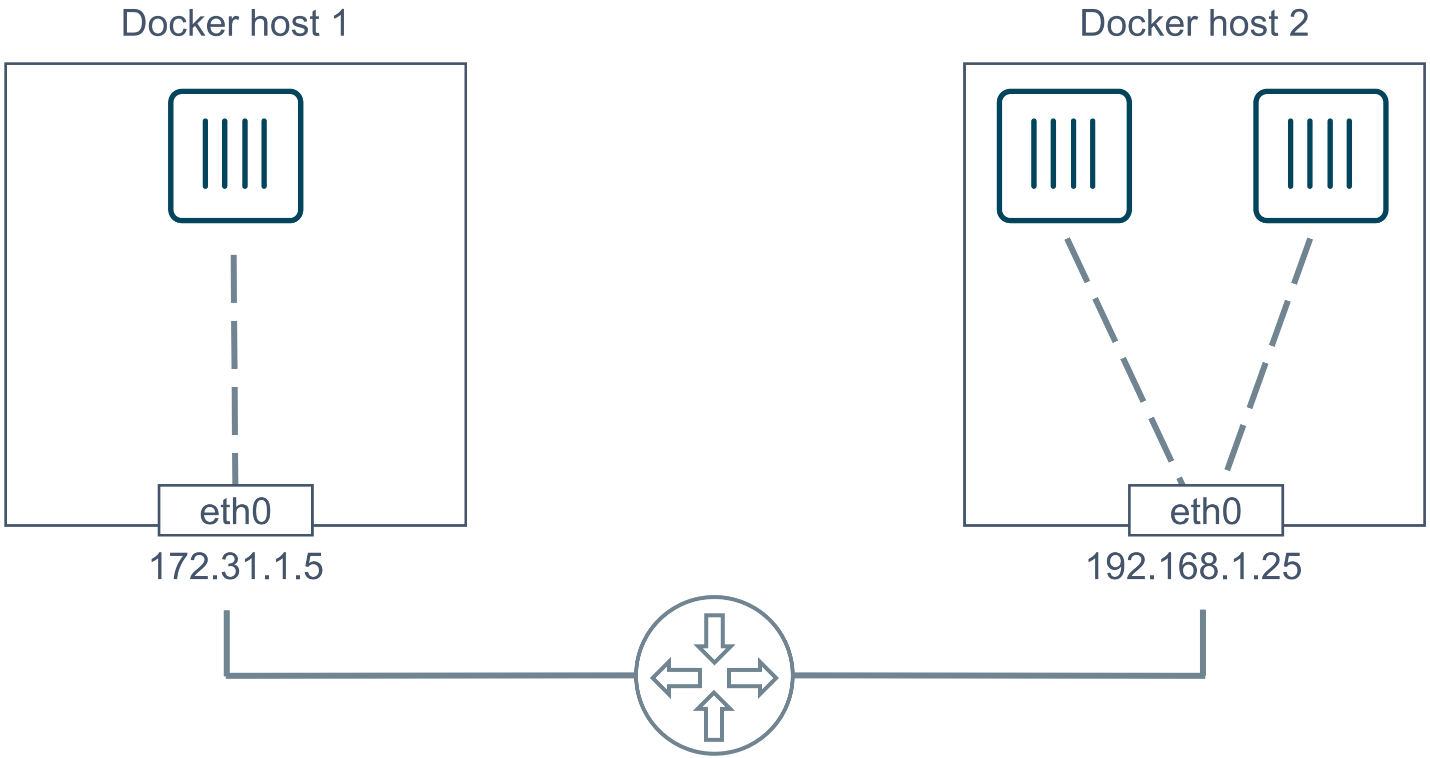
┷ Network

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



Host Mode Data Flow



Demo: Docker Networking Fundamentals

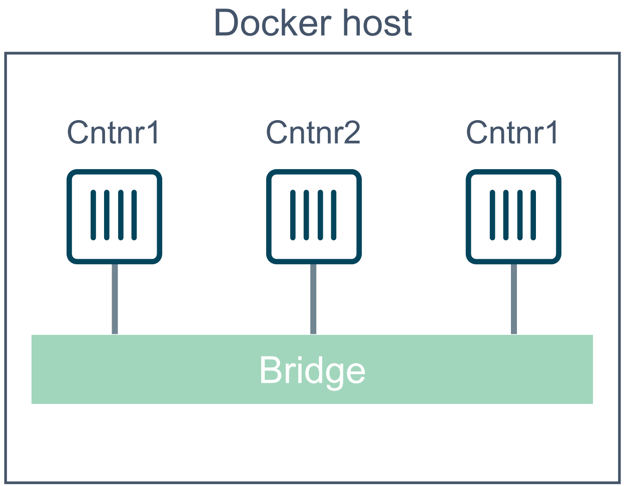
Lab Section 1

Bridge Driver

What is Docker Bridge Networking?

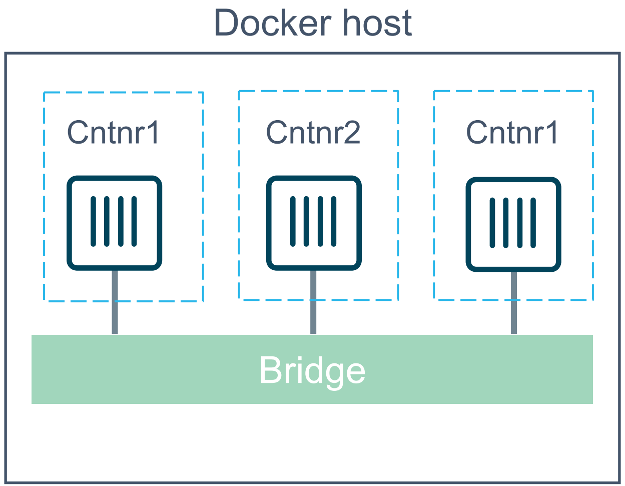
Single-host networking!

- Simple to configure and troubleshoot
- Useful for basic test and dev

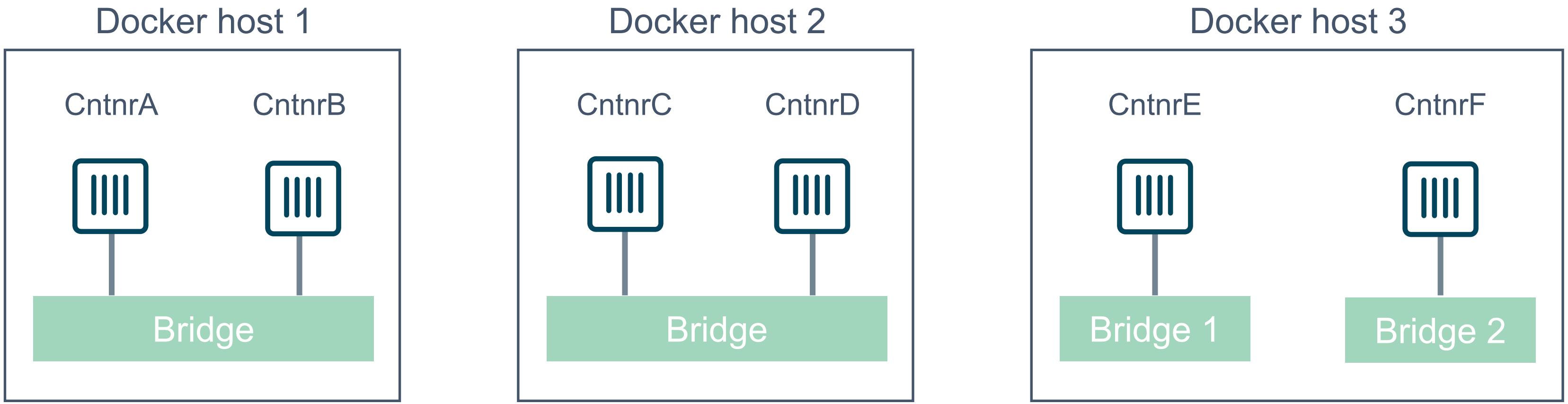


What is Docker Bridge Networking?

- Each container is placed in its own network namespace
- The bridge driver creates a bridge (virtual switch) on a single Docker host
- All containers on this bridge can communicate
- The bridge is a private network restricted to a single Docker host



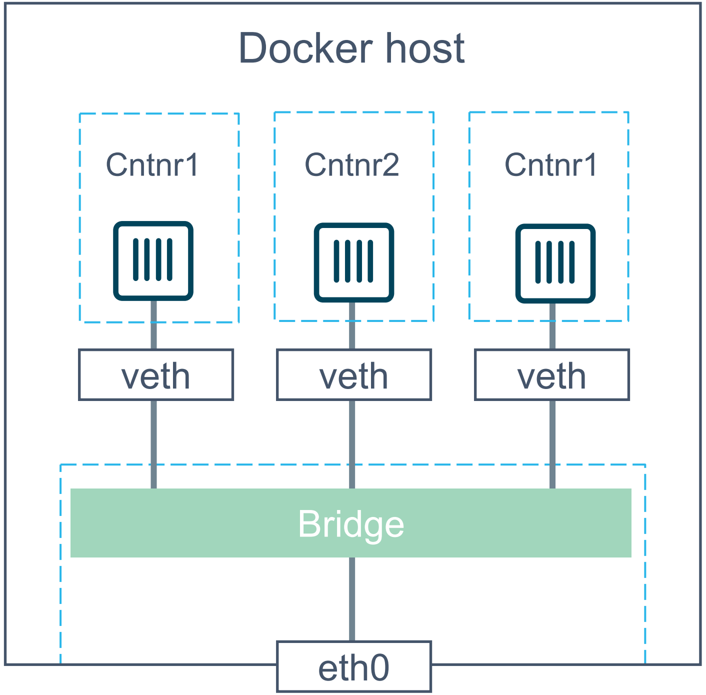
What is Docker Bridge Networking?



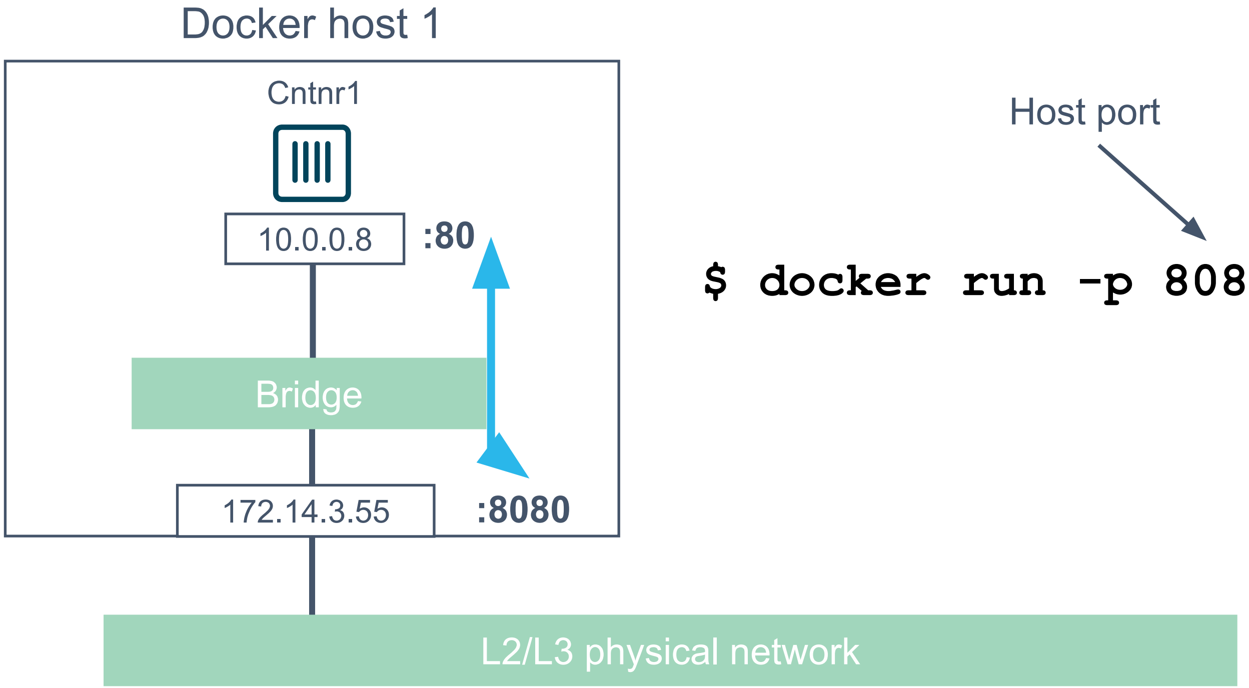
Containers on different **bridge** networks cannot communicate

Bridge Networking in a Bit More 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



Docker Bridge Networking and Port Mapping

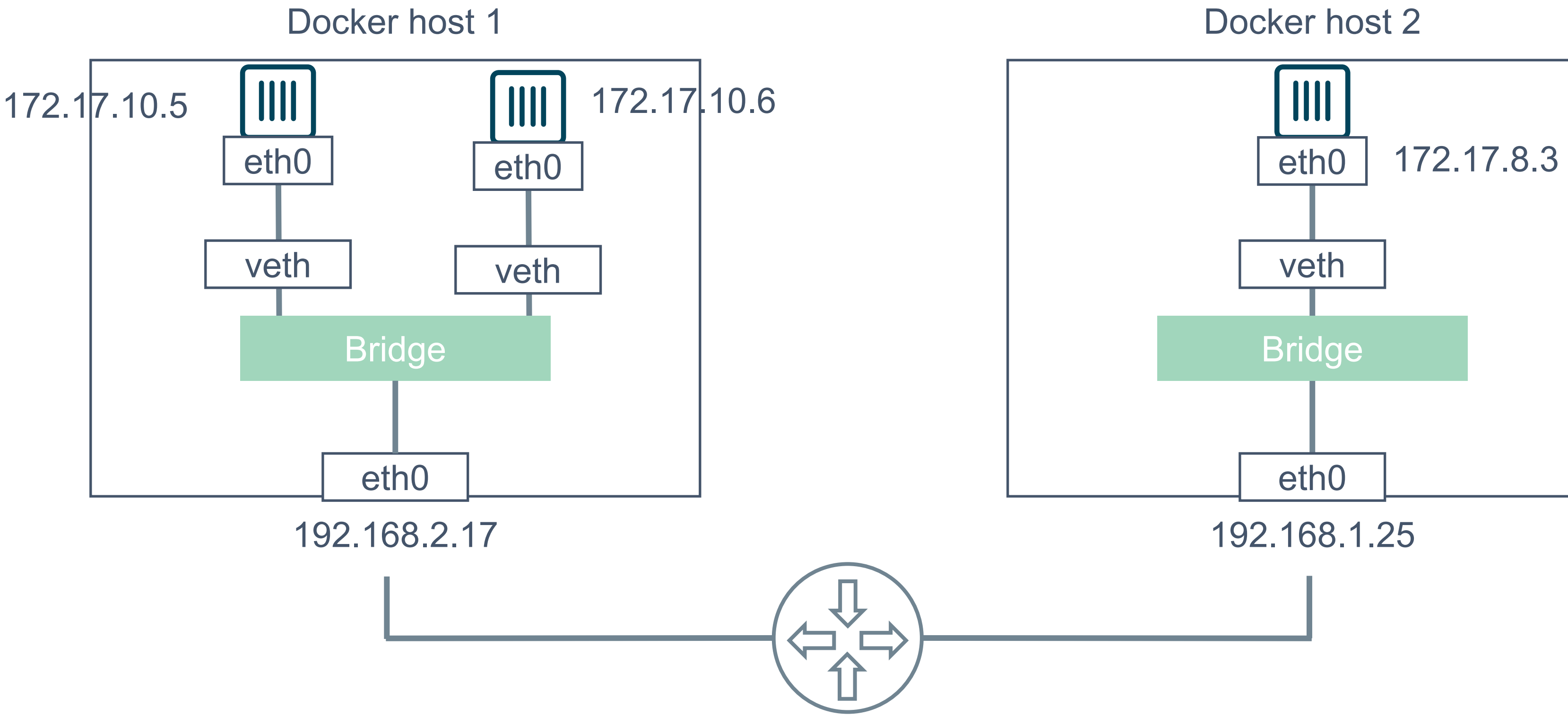


Host port

Container port

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

Bridge Mode Data Flow



Demo

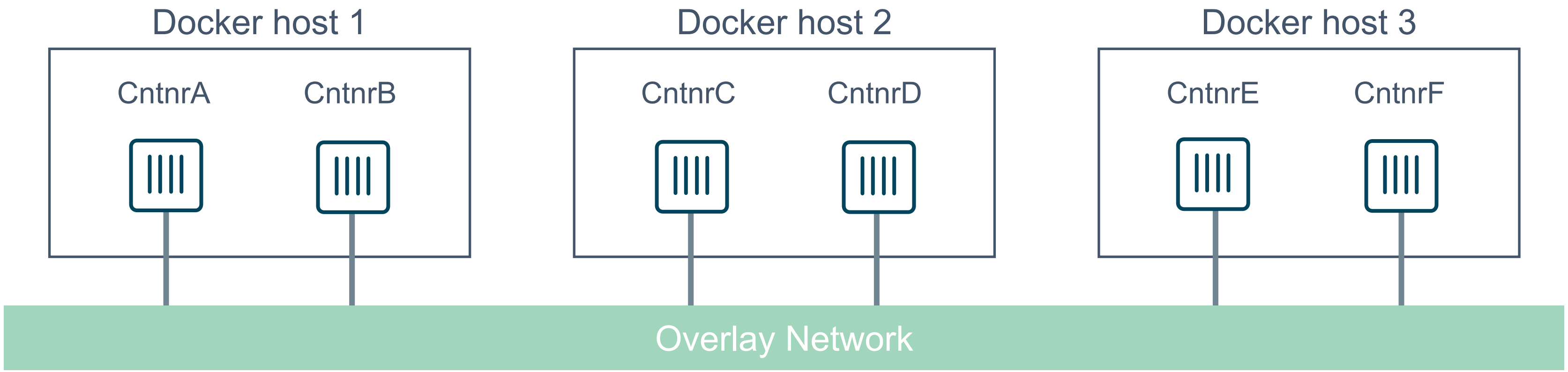
BRIDGE

Lab Section 2

Overlay Driver

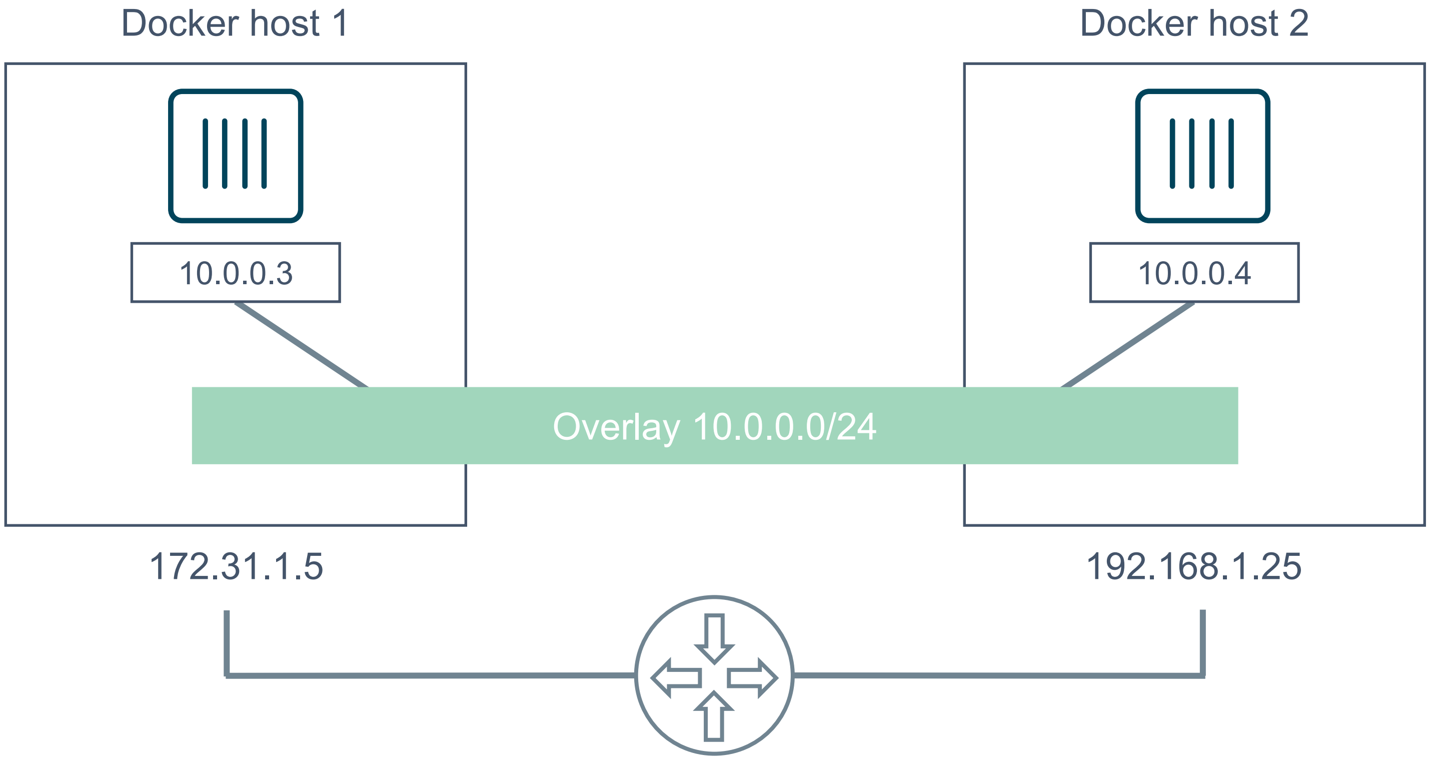
What is Docker Overlay Networking?

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



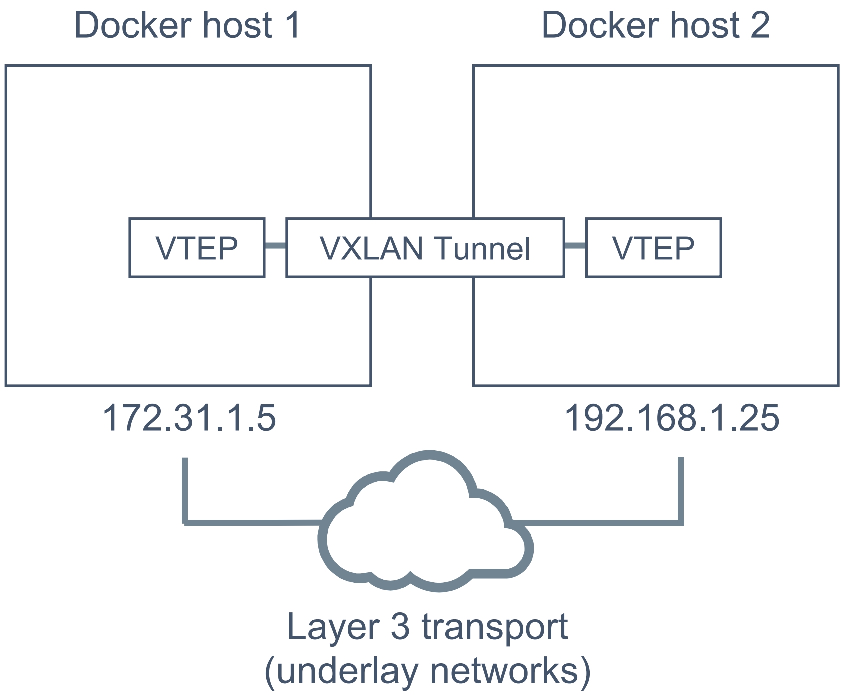
All containers on the **overlay** network can communicate!

Building an Overlay Network (High level)

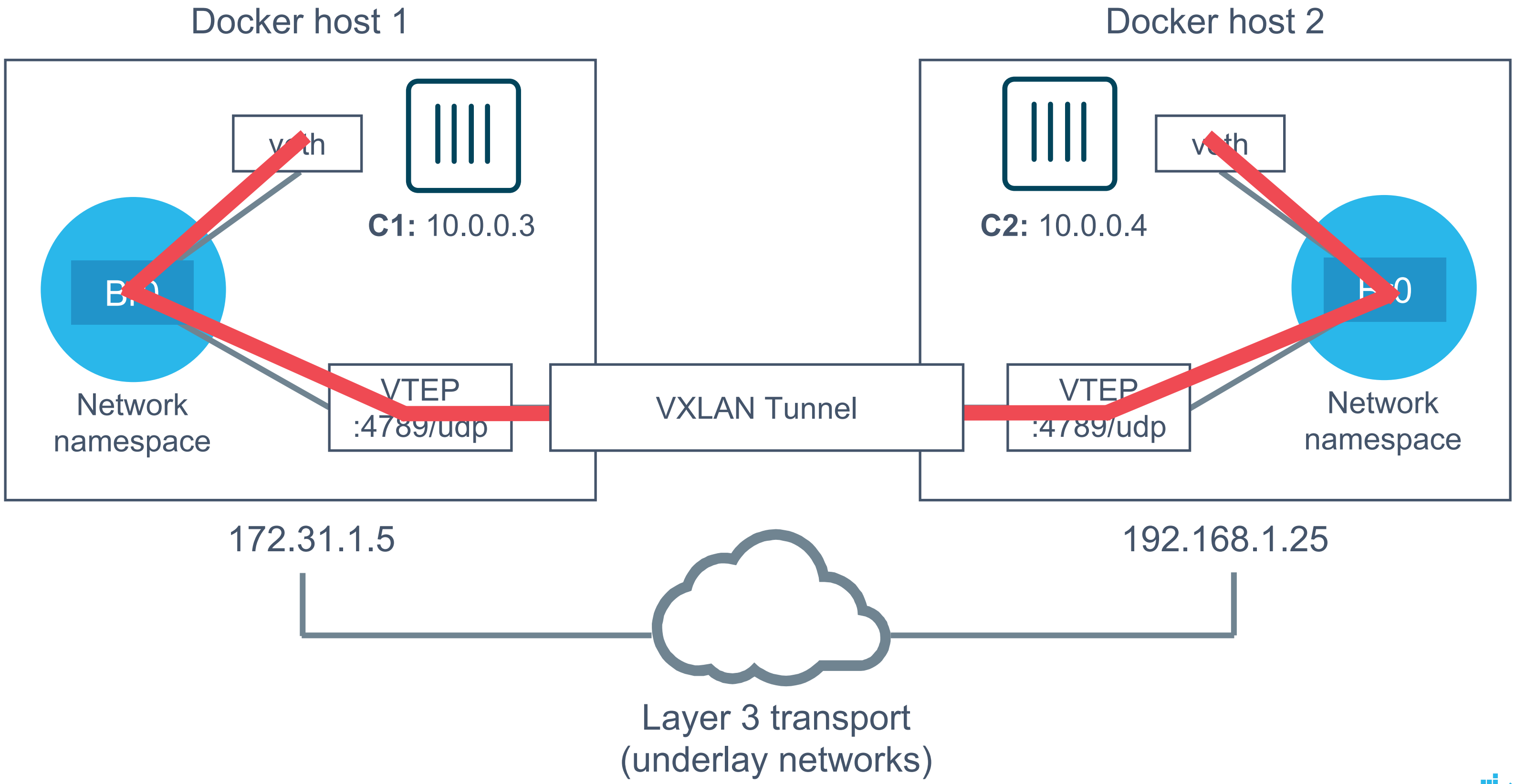


Docker Overlay Networks and VXLAN

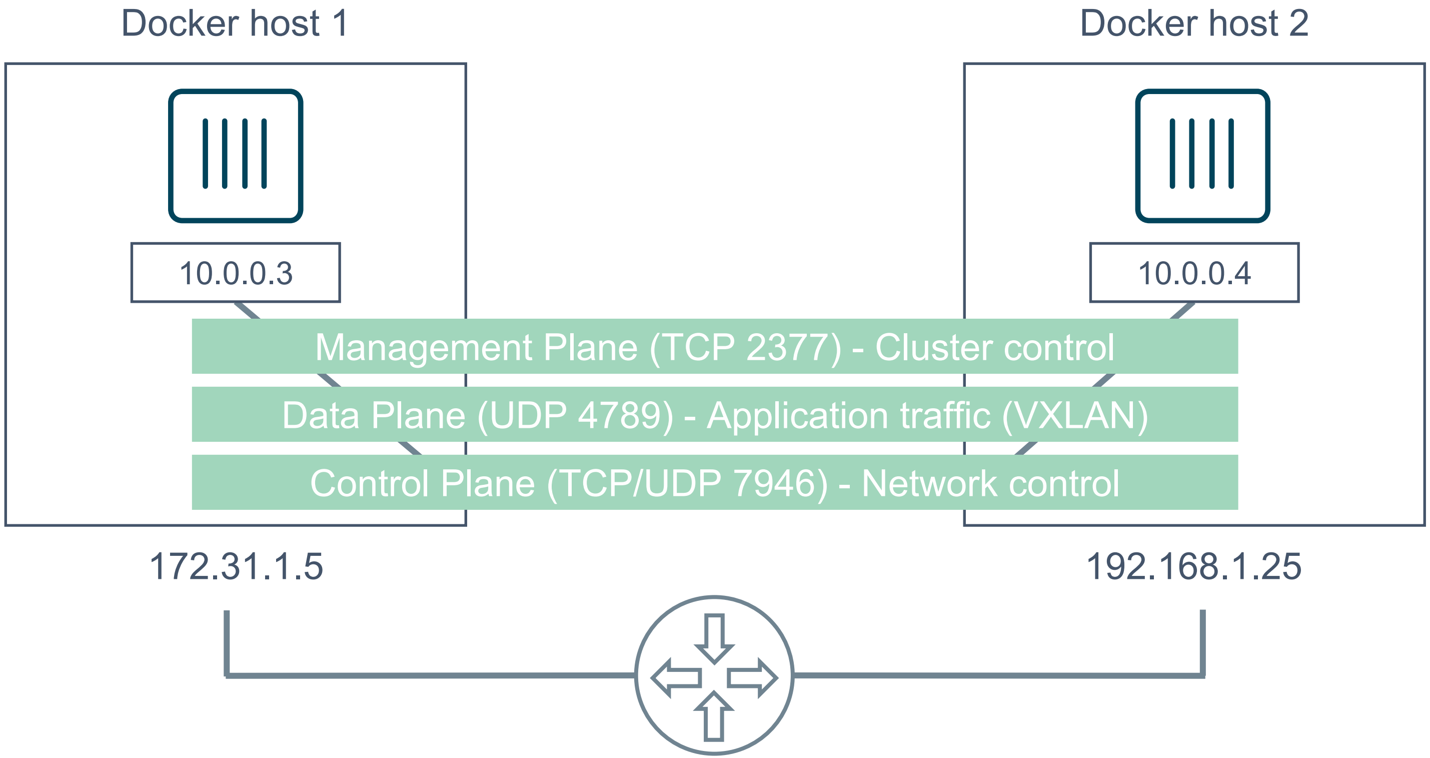
- The **overlay** driver uses VXLAN technology to build the network
- A **VXLAN tunnel** is created through the **underlay network(s)**
- At each end of the tunnel is a VXLAN tunnel end point (**VTEP**)
- The **VTEP** performs encapsulation and de-encapsulation
- The **VTEP** exists in the Docker Host's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



Building an Overlay Network (more detailed)



Overlay Networking Ports

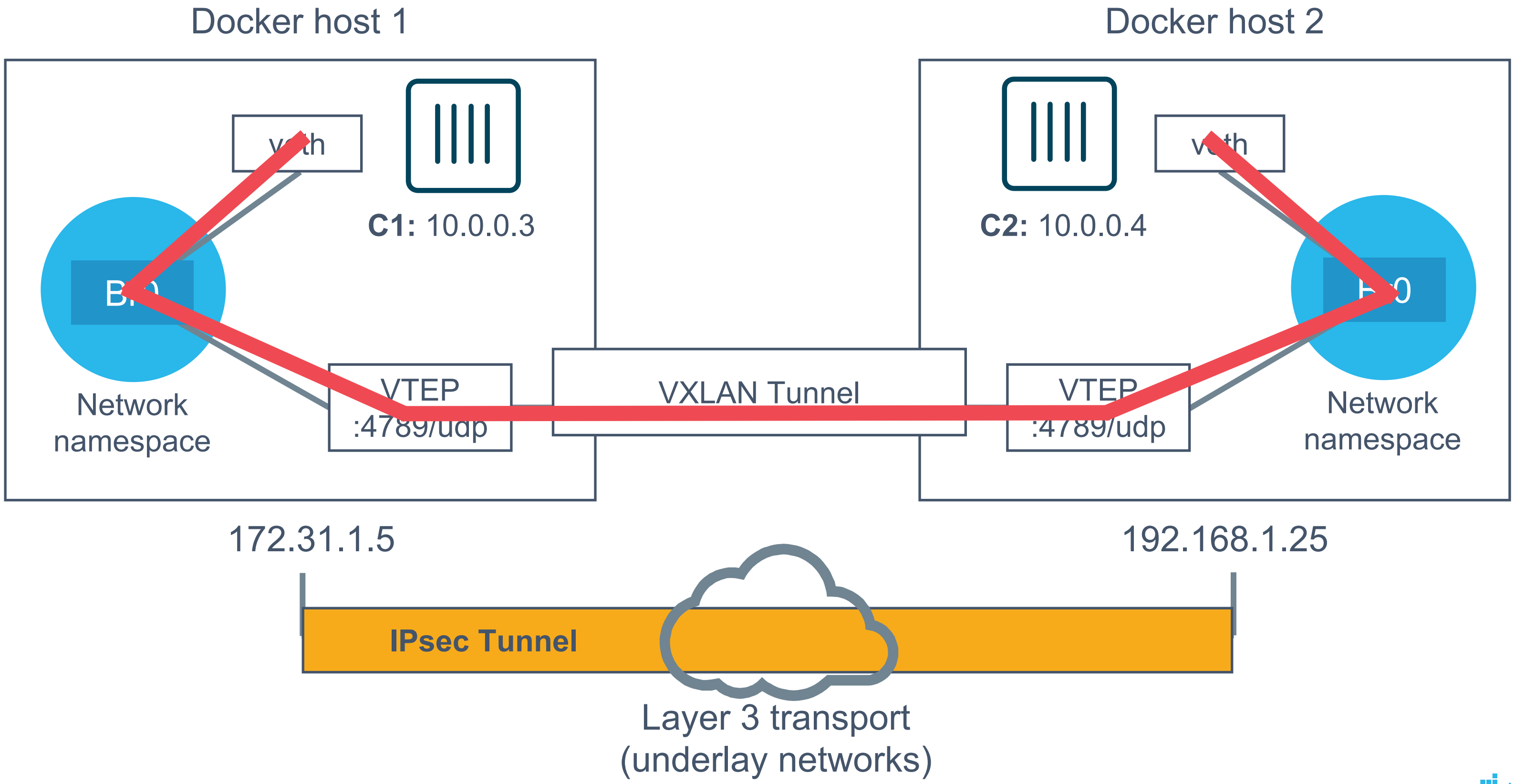


Management Plane (TCP 2377) - Cluster control

Data Plane (UDP 4789) - Application traffic (VXLAN)

Control Plane (TCP/UDP 7946) - Network control

Overlay Network Encryption with IPSec



Overlay Networking Under the Hood

- Virtual eXtensible LAN **(VXLAN)** is the **data transport** (RFC7348)
- Creates a new L2 network over an L3 transport network
- Point-to-Multi-Point tunnels
- VXLAN Network ID (**VNID**) is used to map frames to VLANs
- Uses Proxy ARP
- Invisible to the container
- The **docker_gwbridge** virtual switch per host for default route
- Leverages the distributed KV store created by Swarm
- Control plane is encrypted by default
- Data plane can be encrypted if desired

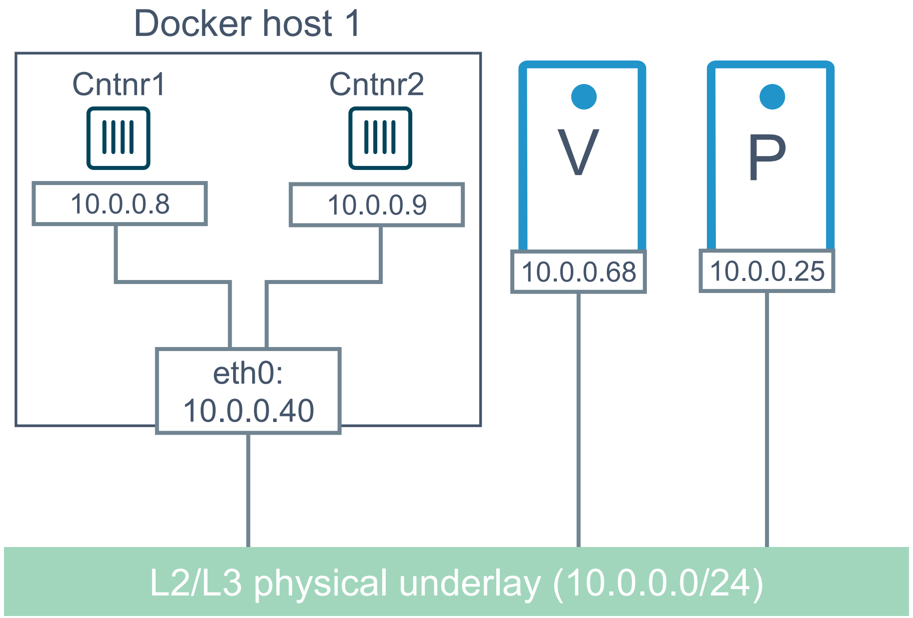
Demo

OVERLAY

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

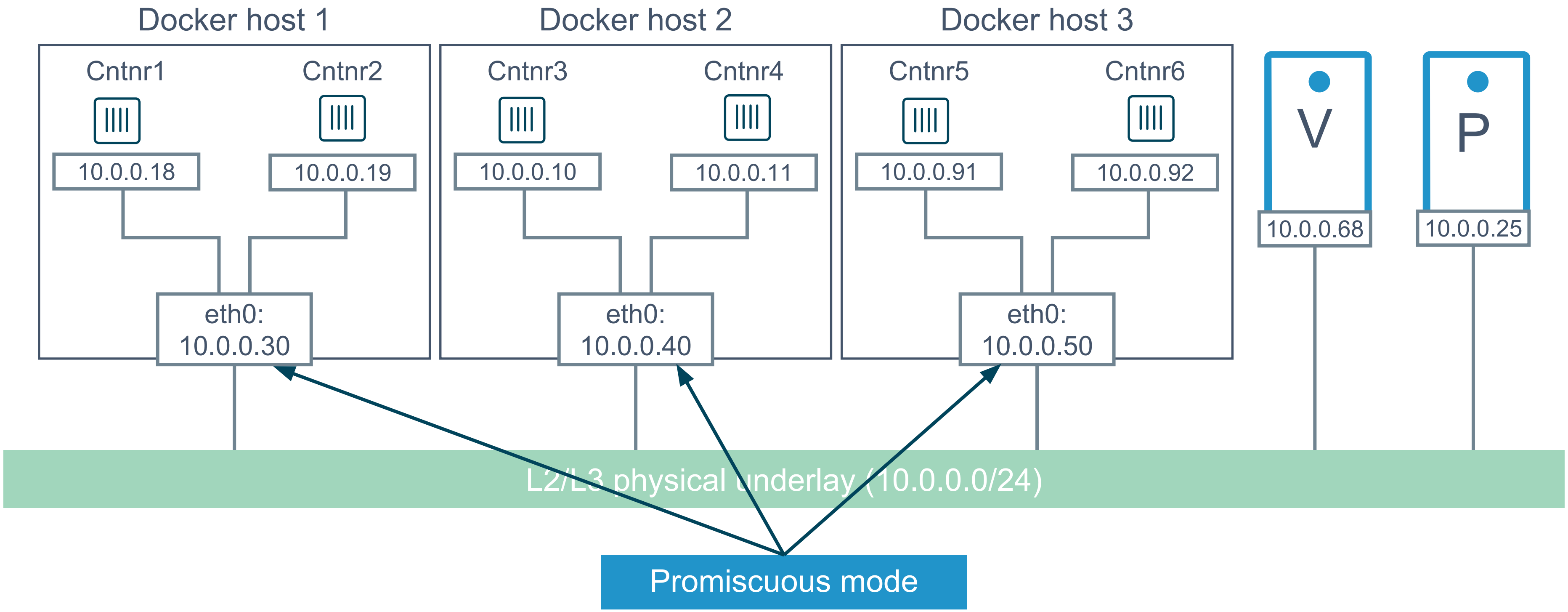
Parent interface has to be connected to physical underlay

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?



Docker host 1

Cntnr1

10.0.0.18

Cntnr2

10.0.0.19

eth0:
10.0.0.30

Docker host 2

Cntnr3

10.0.0.10

Cntnr4

10.0.0.11

eth0:
10.0.0.40

Docker host 3

Cntnr5

10.0.0.91

Cntnr6

10.0.0.92

eth0:
10.0.0.50

V

10.0.0.68

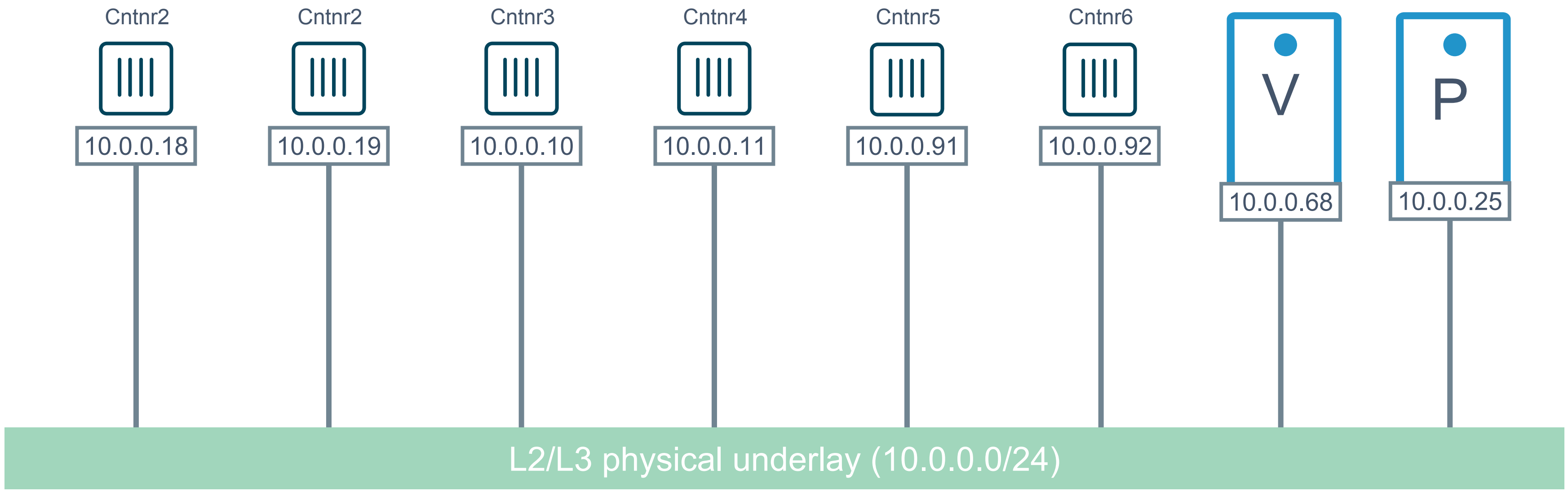
P

10.0.0.25

L2/L3 physical underlay (10.0.0.0/24)

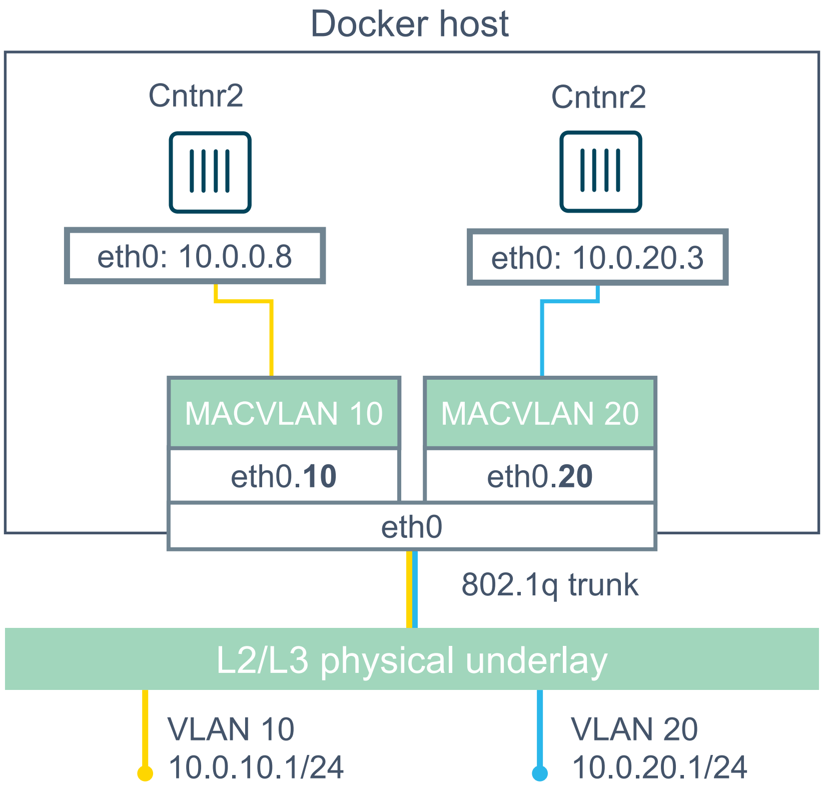
Promiscuous mode

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!

Demo

MACVLAN

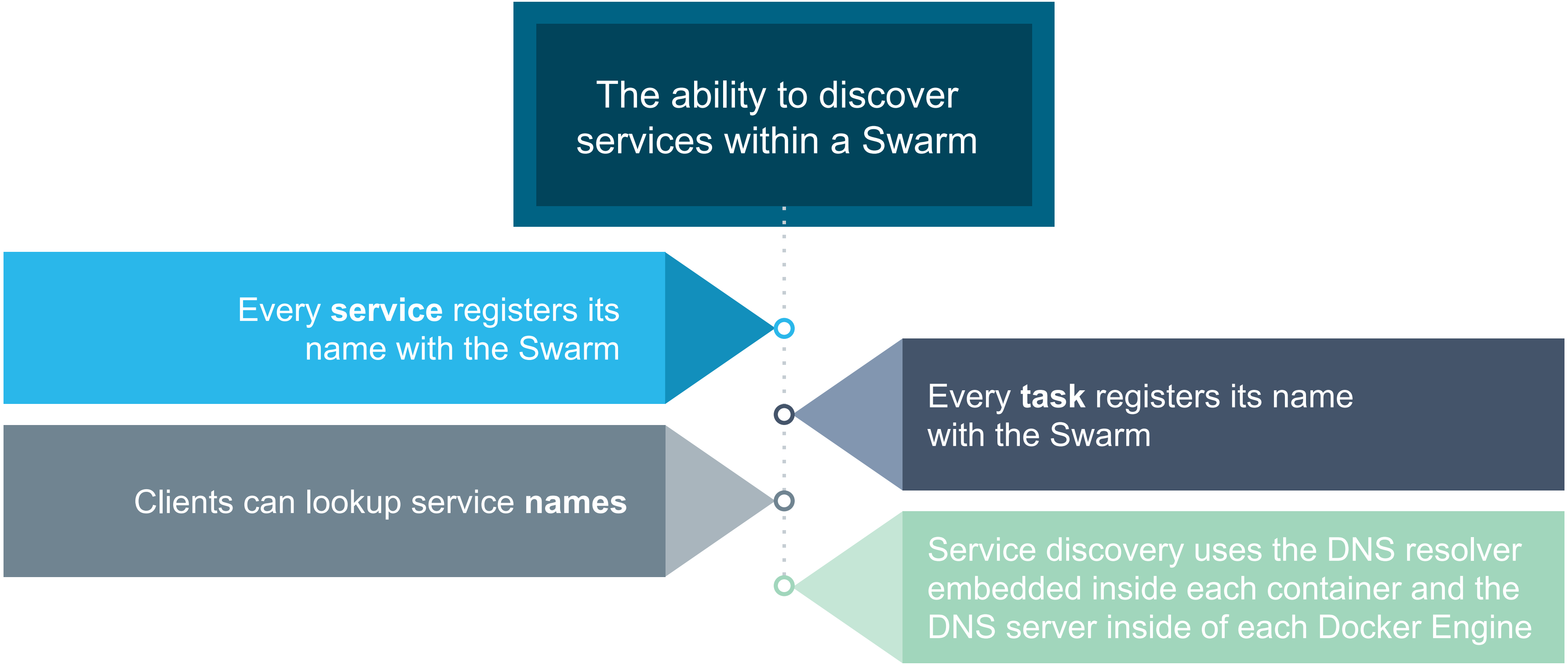
Use Cases Summary

- The bridge driver provides simple single-host networking
  - Recommended to use another more specific driver such as overlay, **MACVLAN** etc…
- The overlay driver provides native out-of-the-box multi-host networking
- The MACVLAN driver allows containers to participate directly in existing networks and VLANs
  - Requires promiscuous mode
- Docker networking will continue to evolve and add more drivers and networking use-cases

Docker Network Services

SERVICE REGISTRATION, SERVICE DISCOVERY, AND LOAD BALANCING

What is Service Discovery?



The ability to discover services within a Swarm

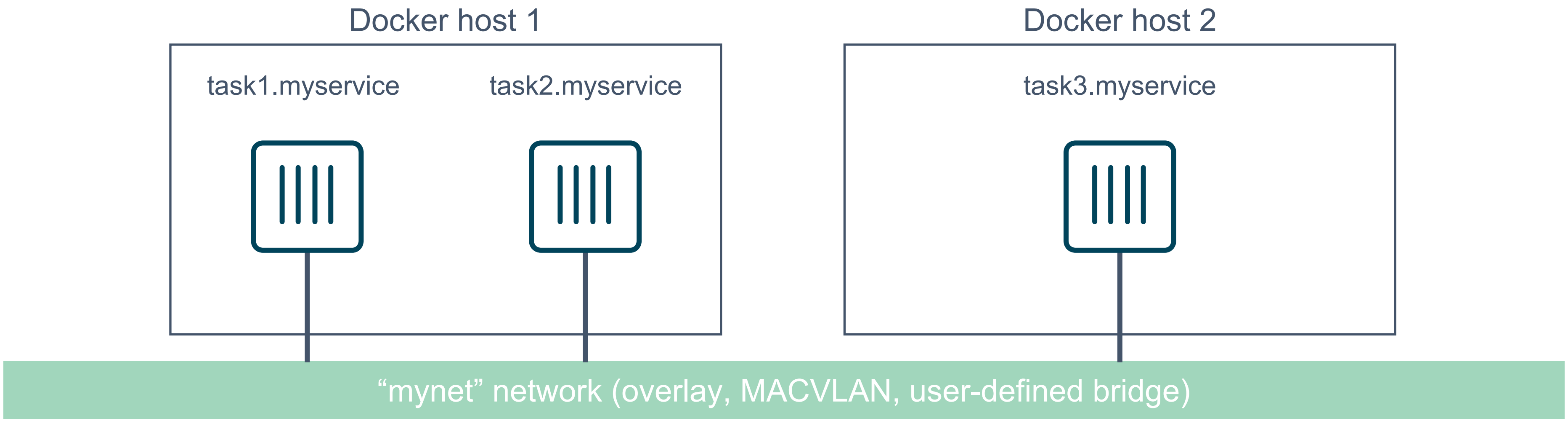
Every **service** registers its name with the Swarm

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 in a Bit More Detail



Docker host 1

task1.myservice

task2.myservice

Docker host 2

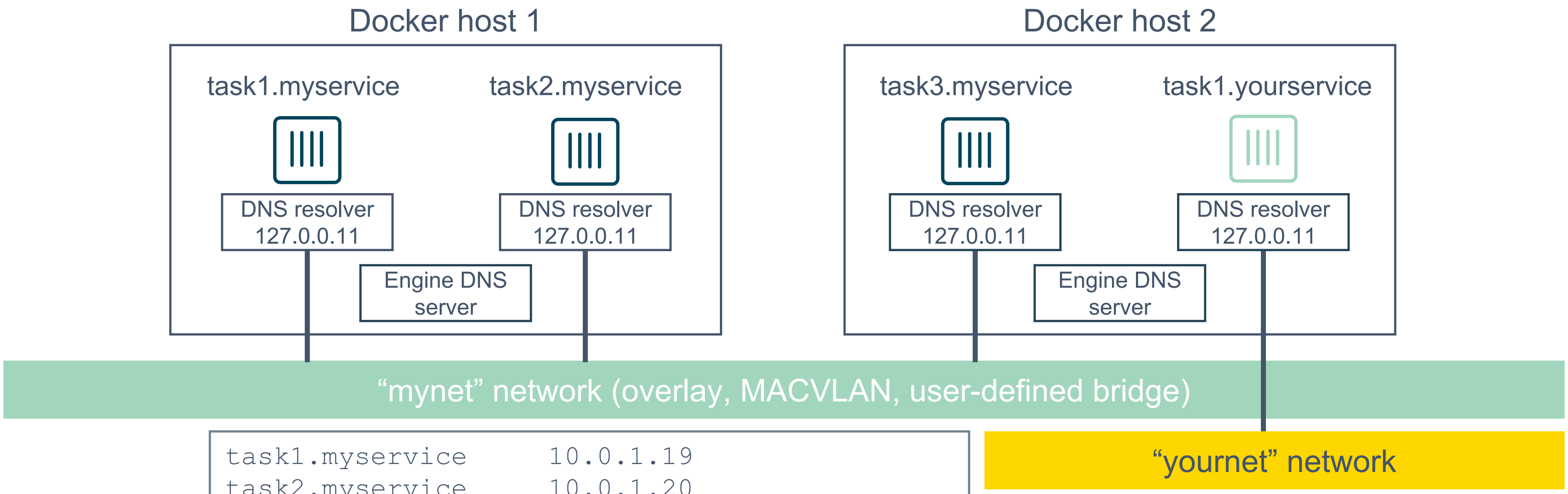
task3.myservice

“mynet” network (overlay, MACVLAN, user-defined bridge)

task1.myservice	10.0.1.19
task2.myservice	10.0.1.20
task3.myservice	10.0.1.21
myservice	10.0.1.18

Swarm DNS (service discovery)

Service Discovery in a Bit More Detail



Docker host 1

task1.myservice

task2.myservice

DNS resolver 127.0.0.11

DNS resolver 127.0.0.11

Engine DNS server

Docker host 2

task3.myservice

task1.yourservice

DNS resolver 127.0.0.11

DNS resolver 127.0.0.11

Engine DNS server

“mynet” network (overlay, MACVLAN, user-defined bridge)

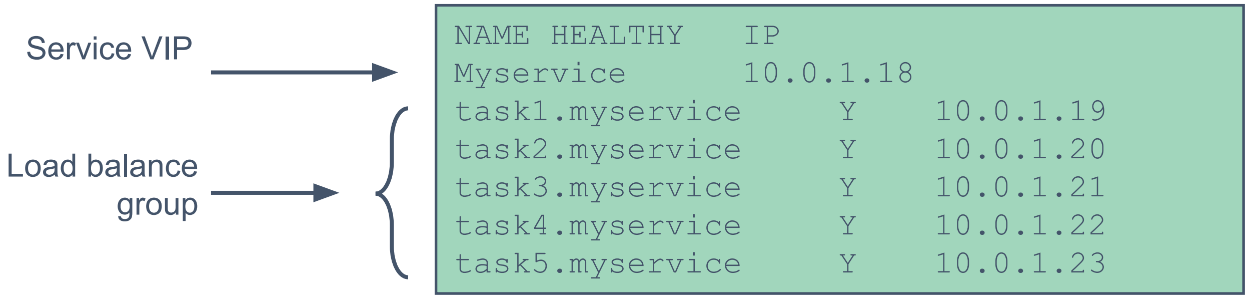
“yournet” network

task1.myservice	10.0.1.19
task2.myservice	10.0.1.20
task3.myservice	10.0.1.21
myservice	10.0.1.18
task1.yourservice	192.168.56.51
yourservice	192.168.56.50

Swarm DNS (service discovery)

Service Virtual IP (VIP) Load Balancing

- Every **service** gets a **VIP** when it's created
  - This 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
```

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

Q & A

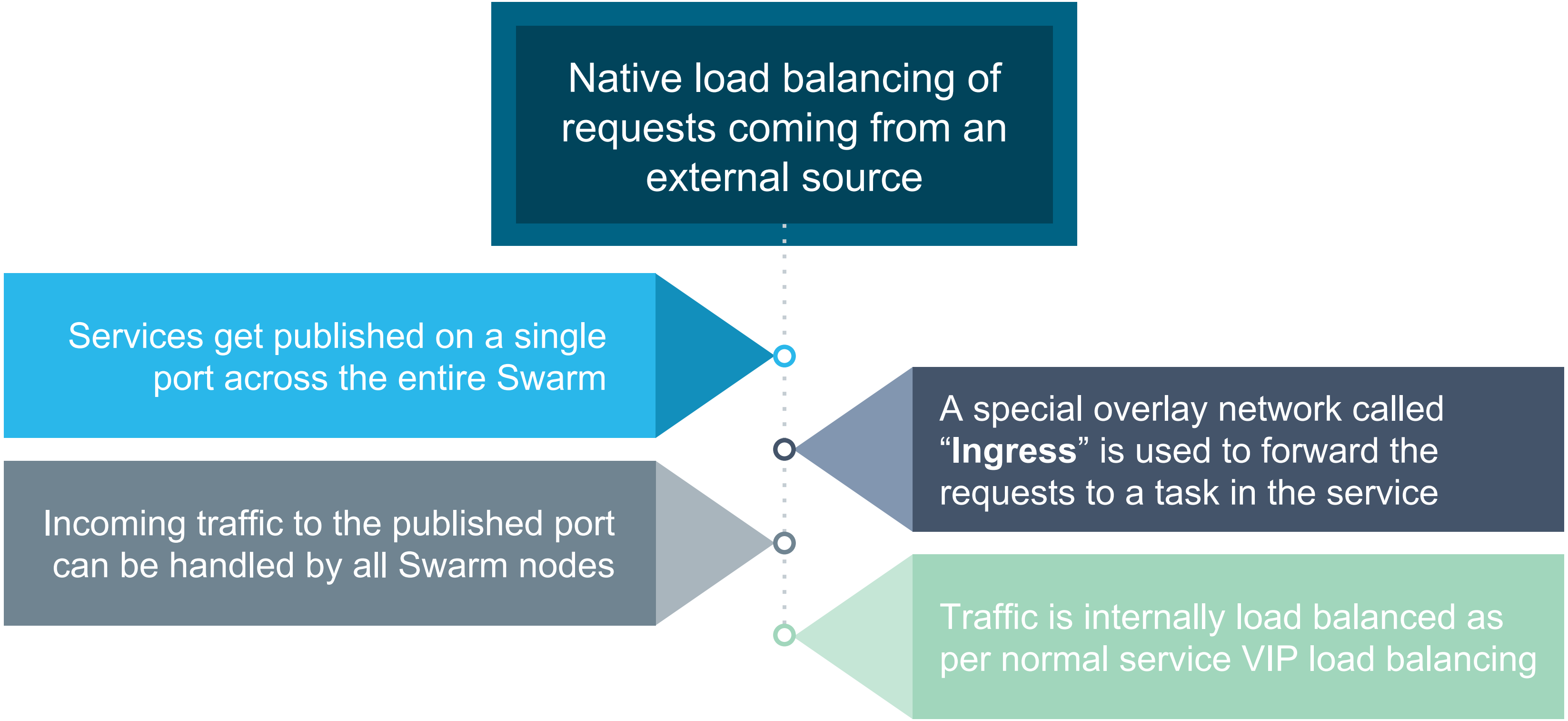
Demo

SERVICE DISCOVERY

Load Balancing External Requests

ROUTING MESH

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

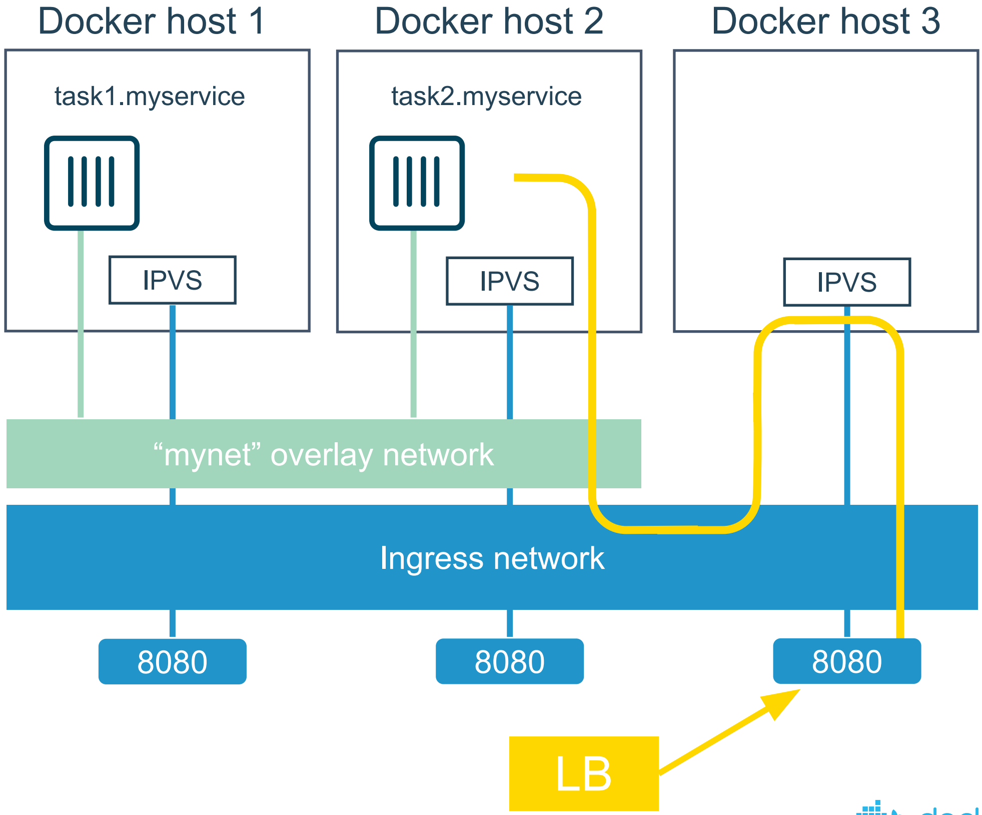
A special overlay network called "**Ingress**" is used to forward the requests to a task in the service

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

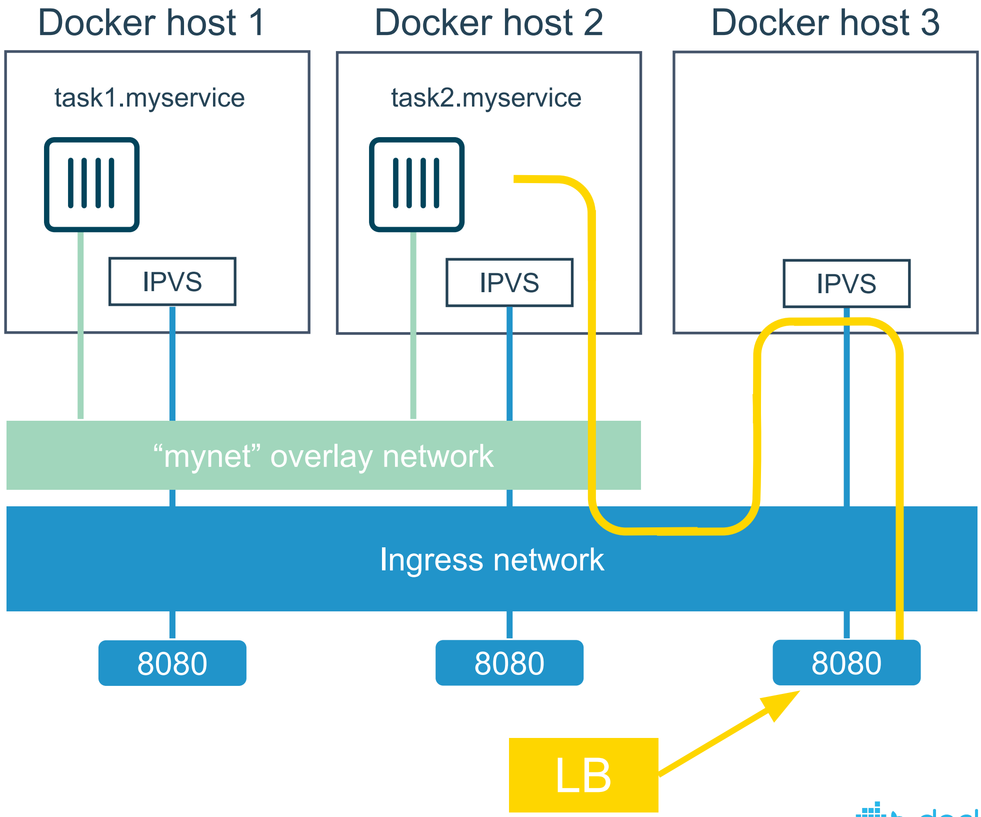
Routing Mesh Example

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

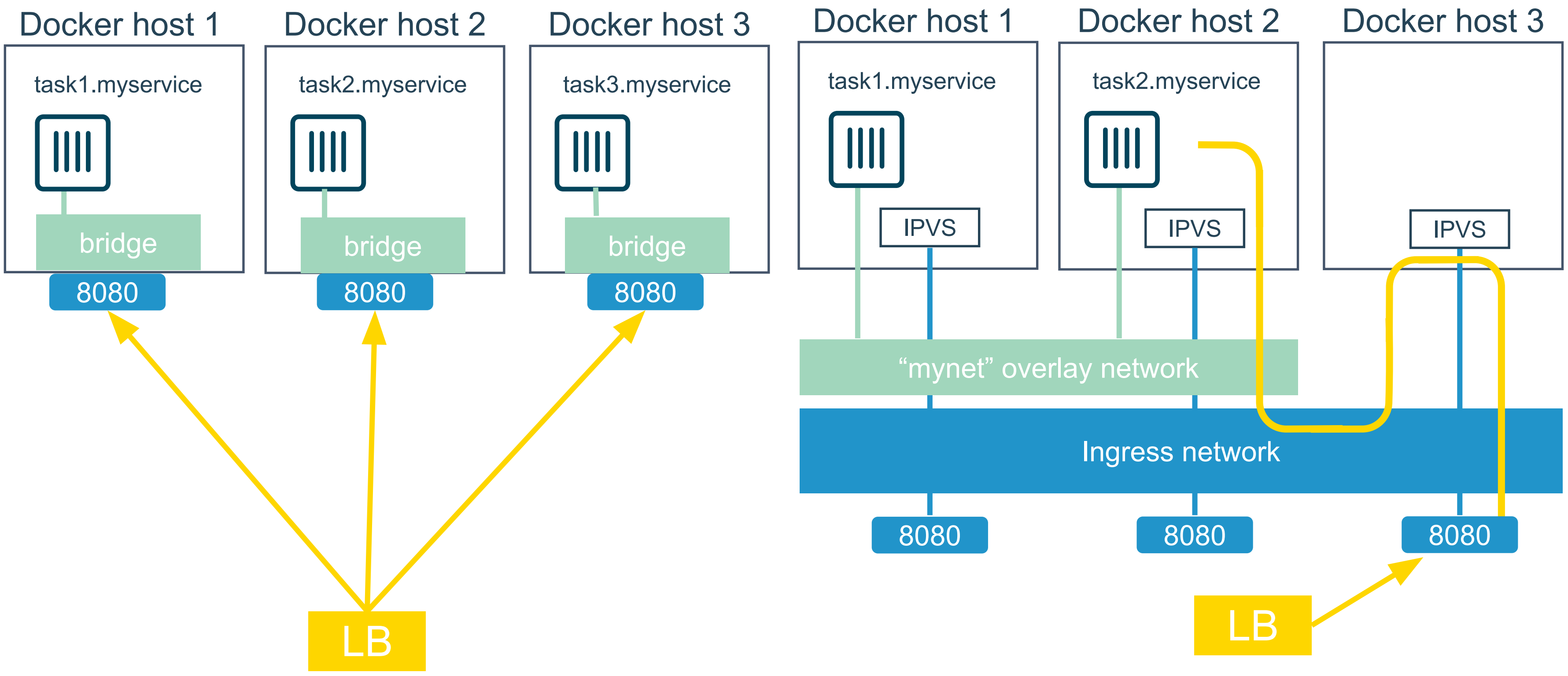


Routing Mesh Example

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



Demo

ROUTING MESH

HTTP Routing Mesh (HRM) with Docker Datacenter

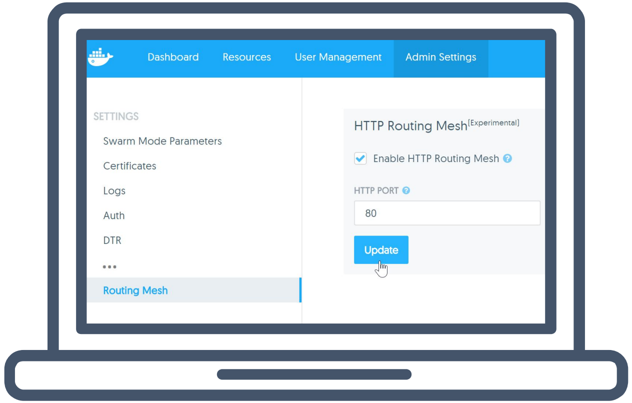
APPLICATION LAYER LOAD BALANCING (L7)

What is the HTTP Routing Mesh (HRM)?

Native **application layer (L7)** load balancing of requests coming from an external source

- Load balances traffic based on hostnames from HTTP headers
- Allows multiple services to be accessed via the same published port
- Requires Docker Enterprise Edition
- Builds on top of transport layer routing mesh

Enabling and Using the HTTP Routing Mesh



HTTP Routing Mesh[Experimental]

☑ Enable HTTP Routing Mesh

HTTP PORT: 80

```
docker service create -p 8080
\
--network ucp-hrm \
--label
com.docker.ucp.mesh.http=8080=
http://foo.exsample.org
...
```

1

Enable HTTP routing mesh in UCP

a) Creates **ucp-hrm** *network*

b) Creates **ucp-hrm** *service* and exposes it on a port (80 by default)

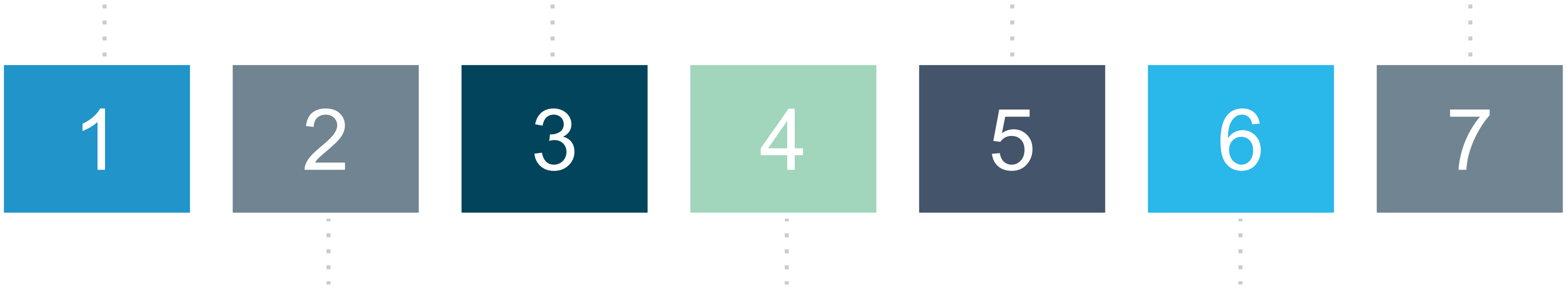
2

Create new service

a) Add to **ucp-hrm** network

b) Assign **label** specifying hostname
(links service to http://foo.example.com)

HTTP Routing Mesh (HRM) Flow



Enable HRM in UCP and assign a port

Create a service (publish a port, attach it to the **ucp-hrm** network, and add the **com.docker.ucp.mesh .http** label)

HTTP traffic comes in on the HRM port

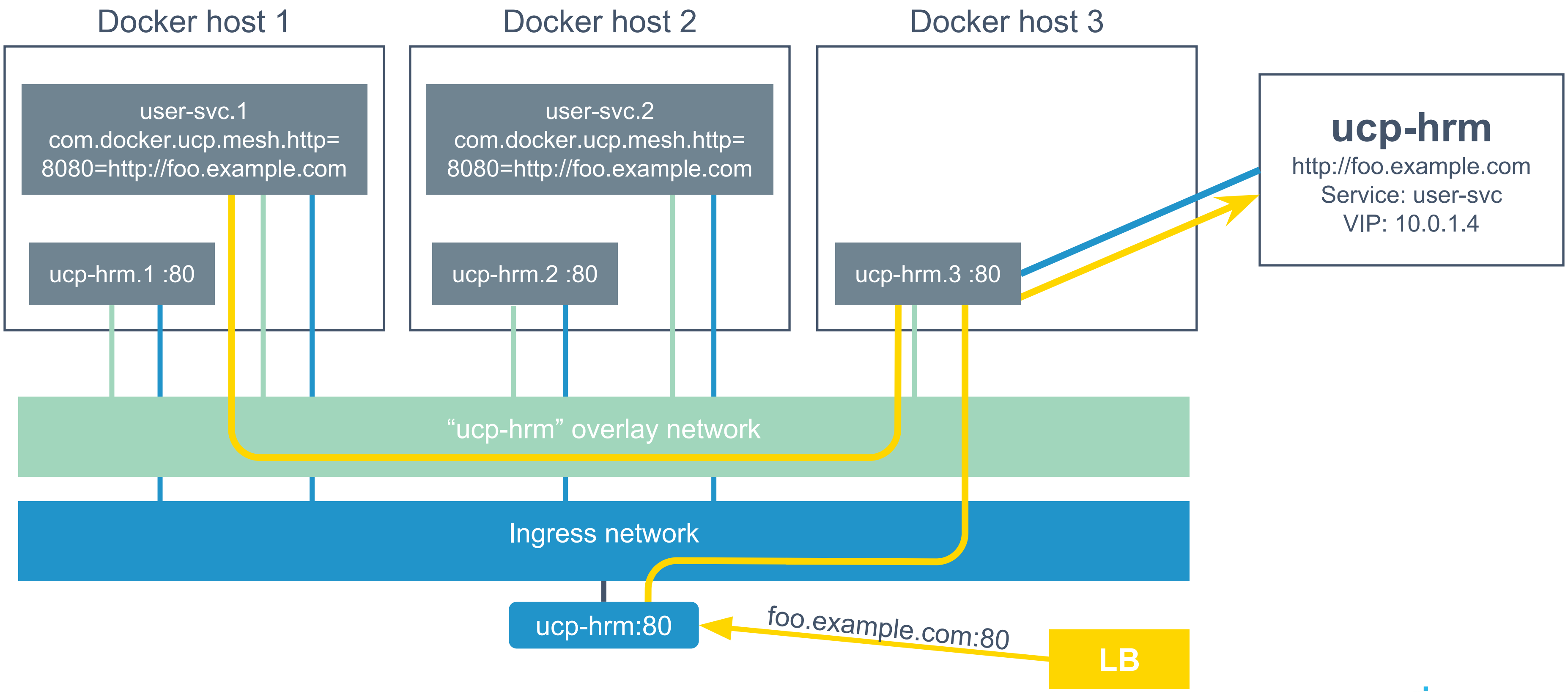
Gets routed to the **ucp-hrm** service

HTTP header is inspected for **host** value

Host value is matched with the **com.docker.ucp.mesh.http** label for a service

Request is passed to the VIP of the service with the matching **com.docker.ucp.mesh .http** label

HTTP Routing Mesh Example



Demo

HRM

Q & A

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

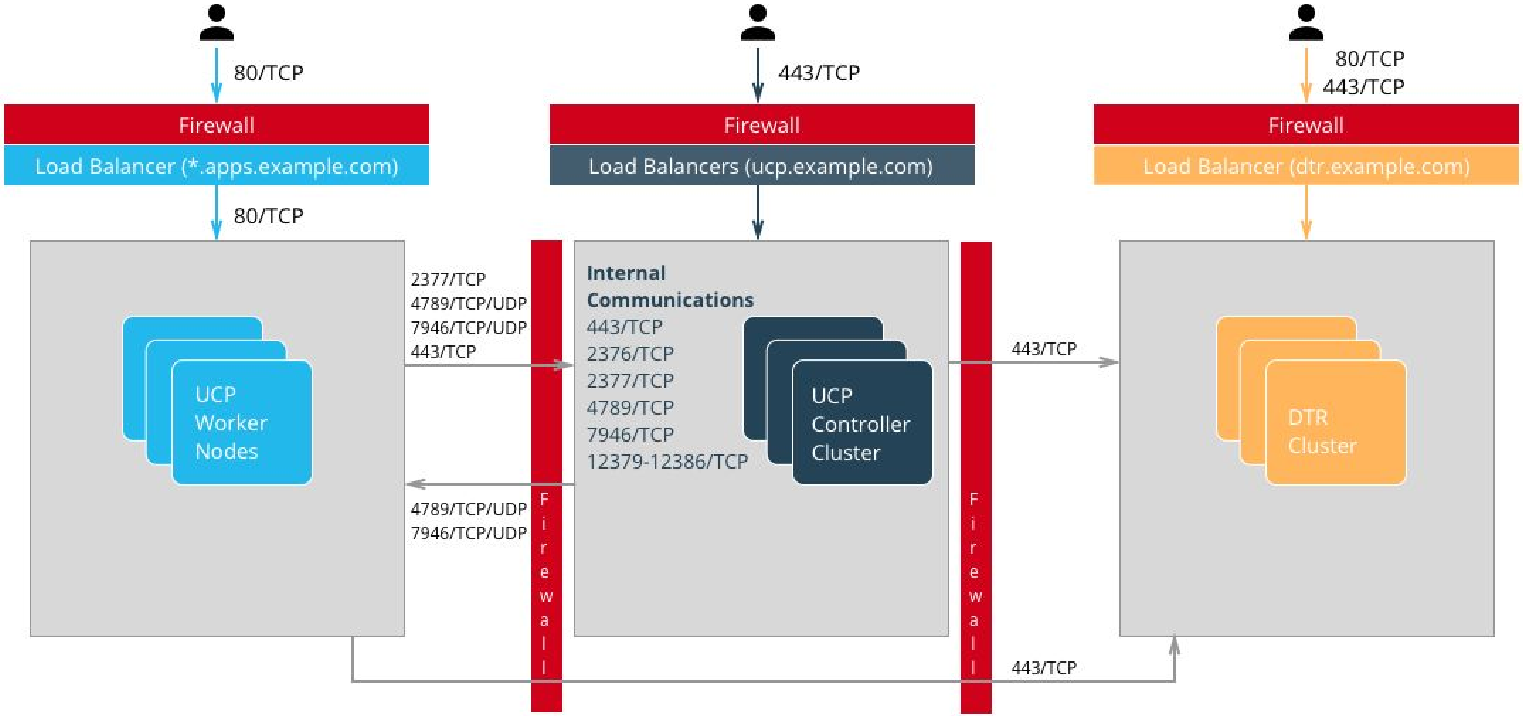
List rules with $ iptables -S, $ iptables -S -t nat

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 **in a container** 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 Section 3

THANK YOU