Dockercon 2017 Networking Workshop

Mark Church, Technical Account Manager @ Docker

Lorenzo Fontana, Docker Captain

Nico Kabar, Solutions Architect @ Docker

Agenda

1. Container Network Model
2. Docker Networking Fundamentals
3. Bridge Driver
4. Overlay Driver
5. MACVLAN Driver
6. Network Services (Internal Load Balancing, Service Discovery)
7. Ingress Load Balancing
8. Docker Network Troubleshooting

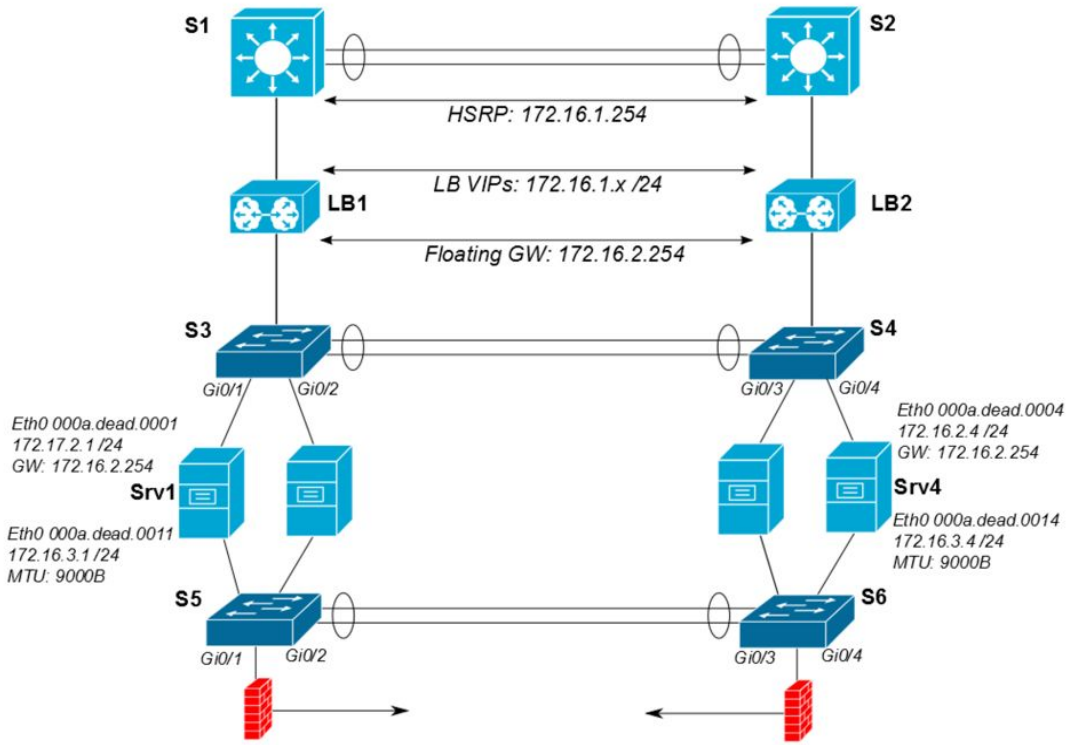
Logistics

- Logging in to lab environment
  - 2 Docker Instances: node0, node1
  - Username: ubuntu
  - Password: D0ckerCon2017
- Slack Channel:
- Survey

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

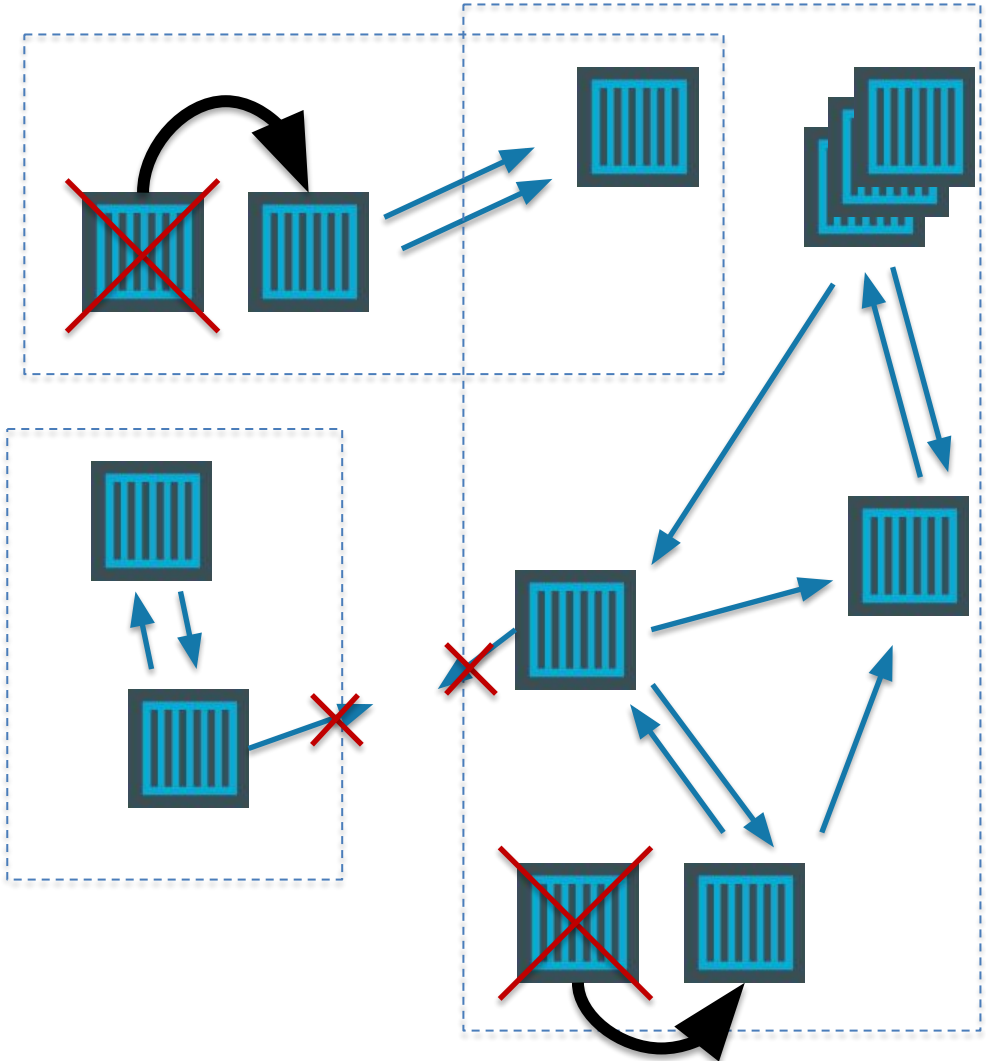


docker

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

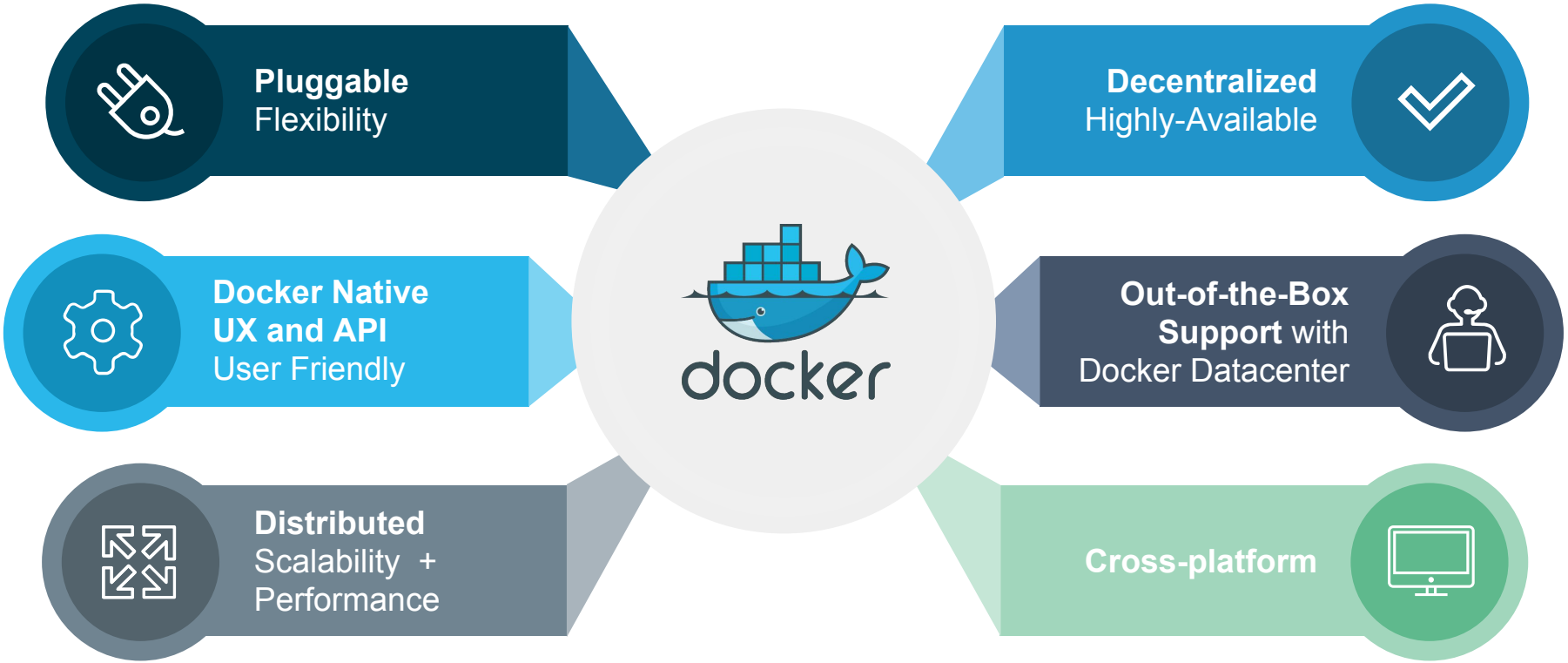
Put Users First

Developers and Operations

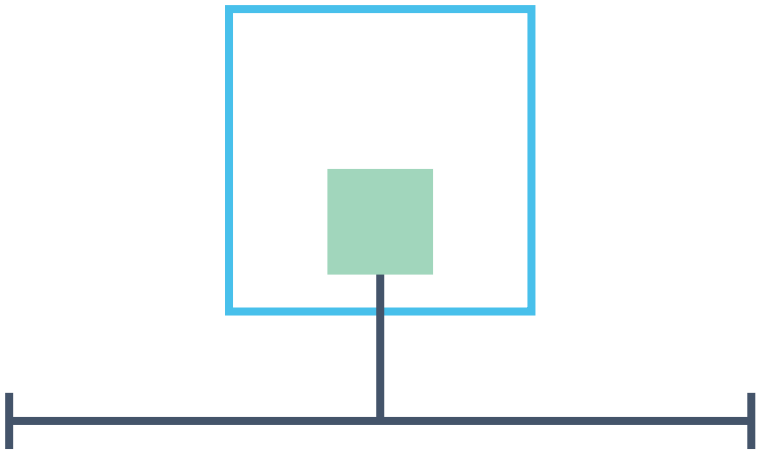
Plugin API Design

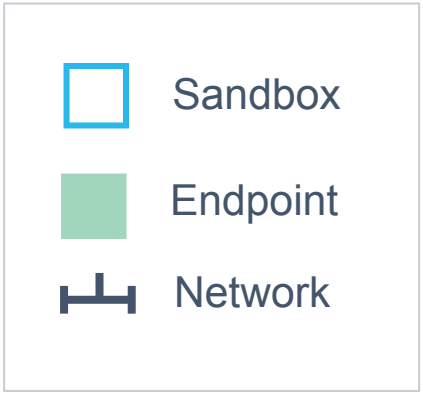
Batteries included but removable

Docker Networking Goals

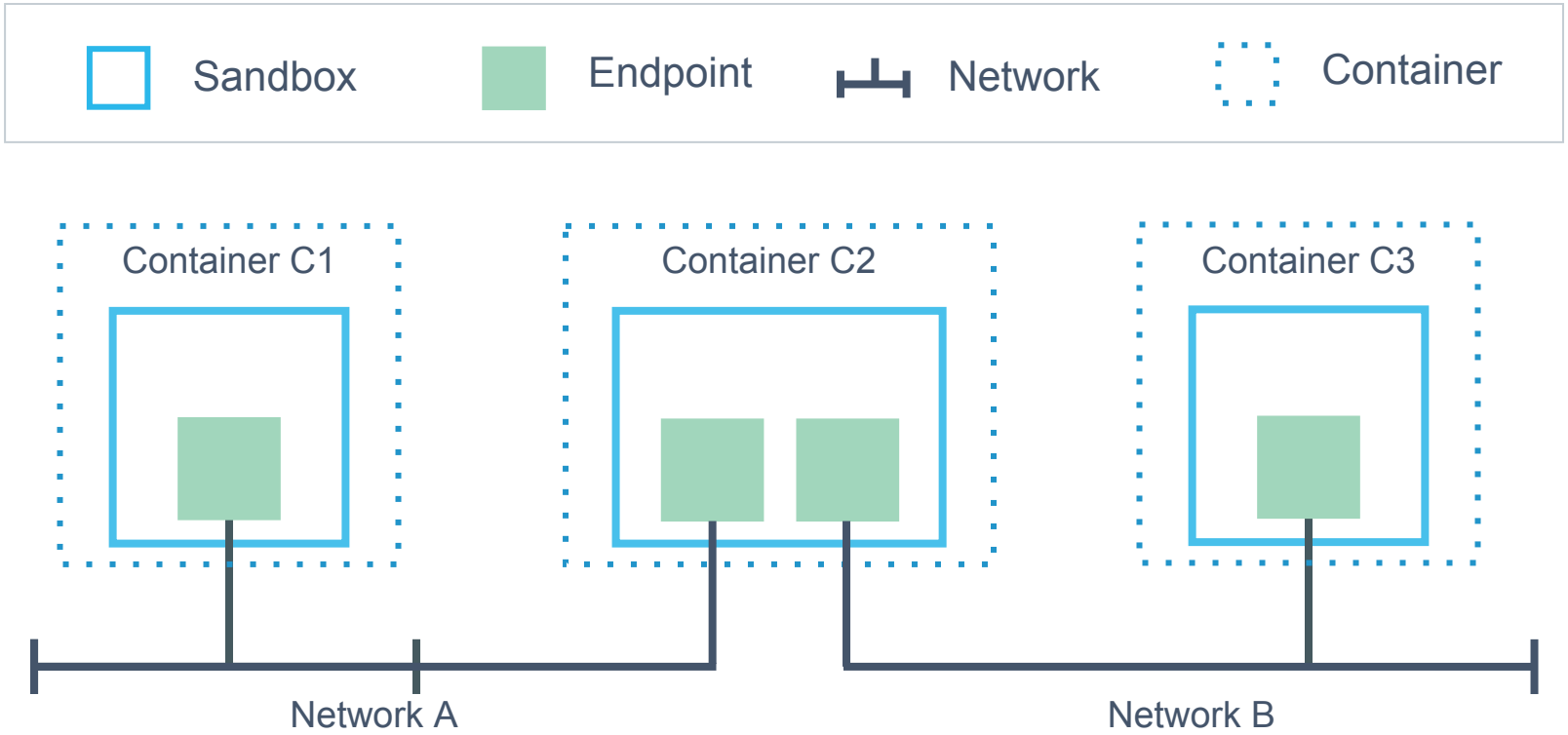


Container Network Model (CNM)



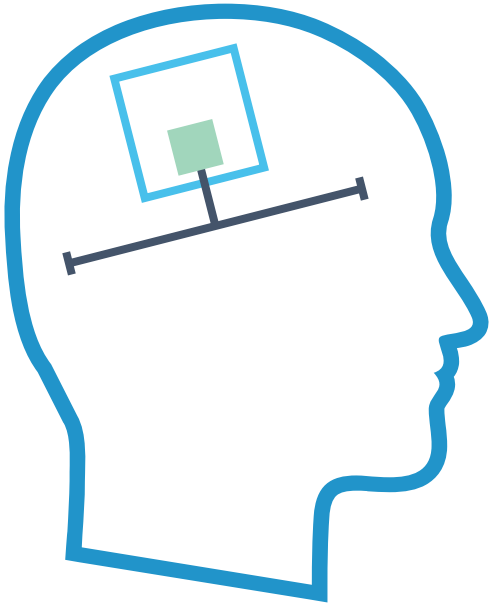


Containers and the CNM



What is Libnetwork?

Libnetwork is Docker's native implementation of the CNM

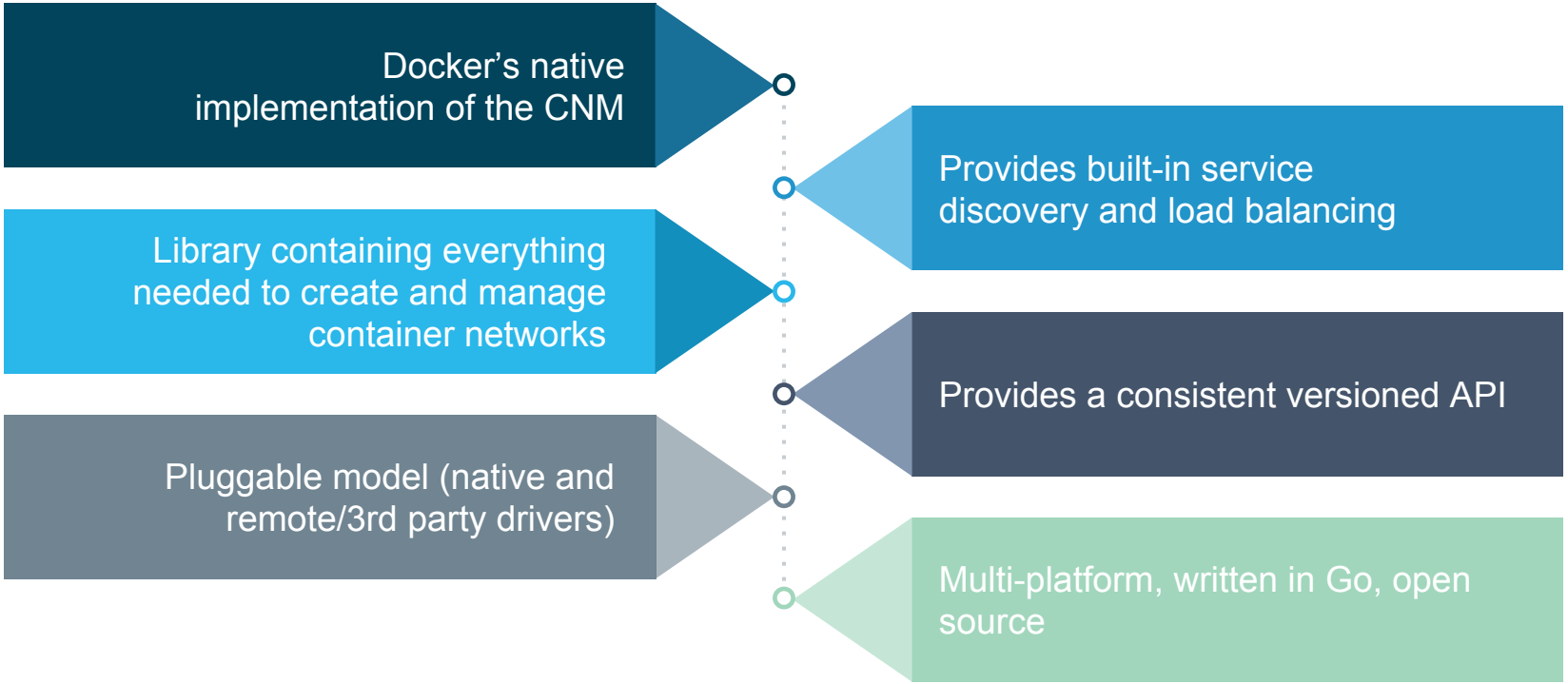


CNM

```
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?



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

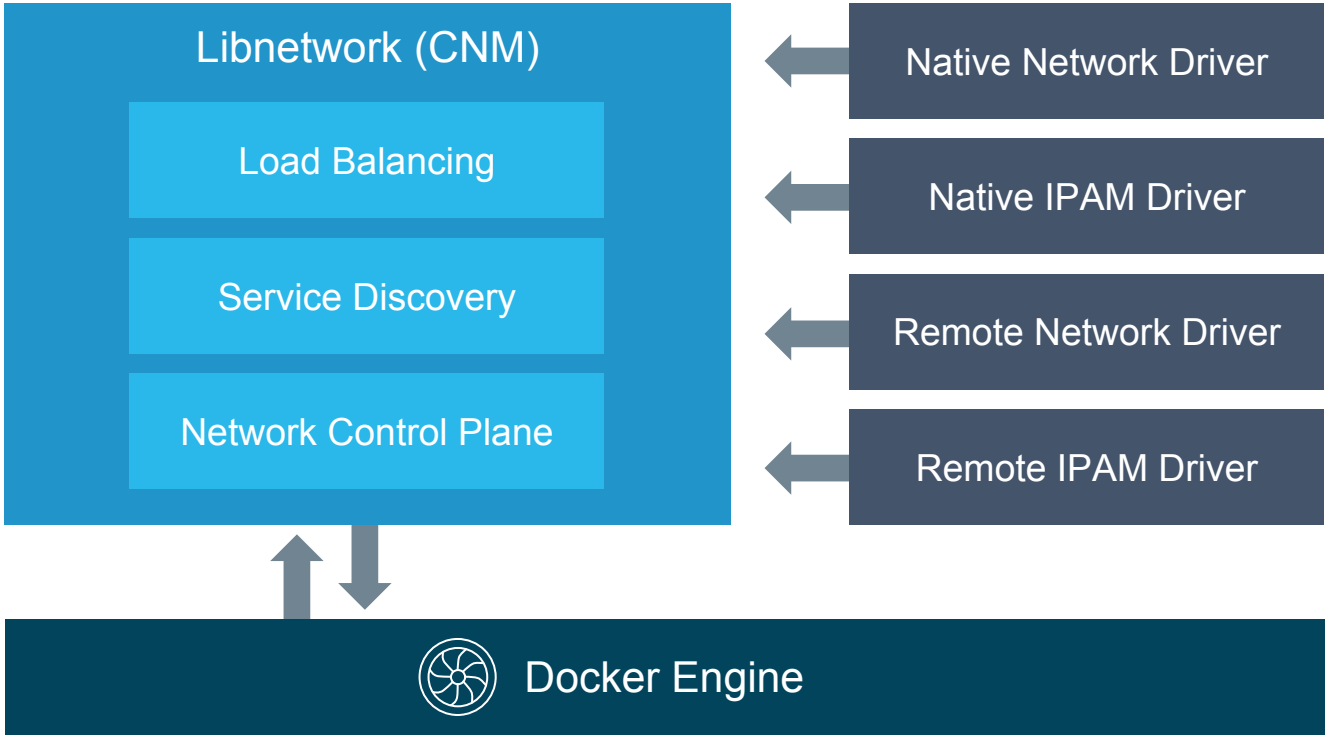
3rd party drivers are called remote drivers, and include:
Calico, Contiv, Kuryr, Weave…

Libnetwork also supports pluggable IPAM drivers

Show Registered Drivers

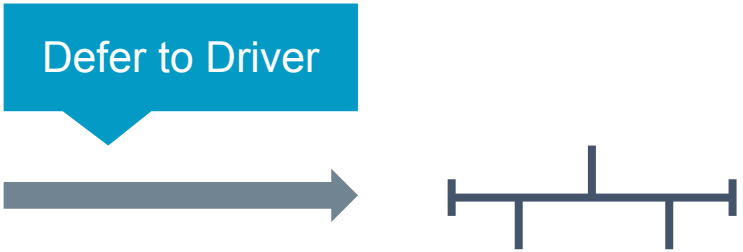
```
$ docker info

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

Libnetwork Architecture

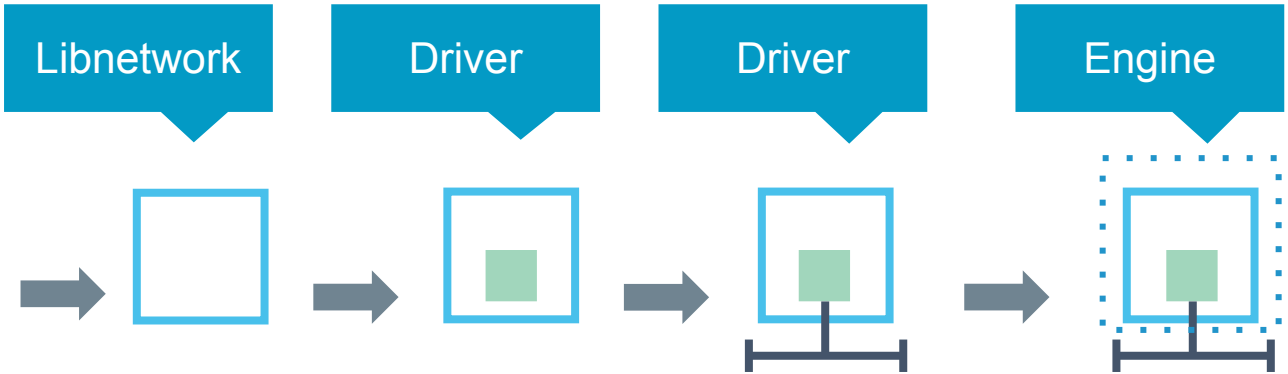


Networks and Containers

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




```
docker run --network
...
```

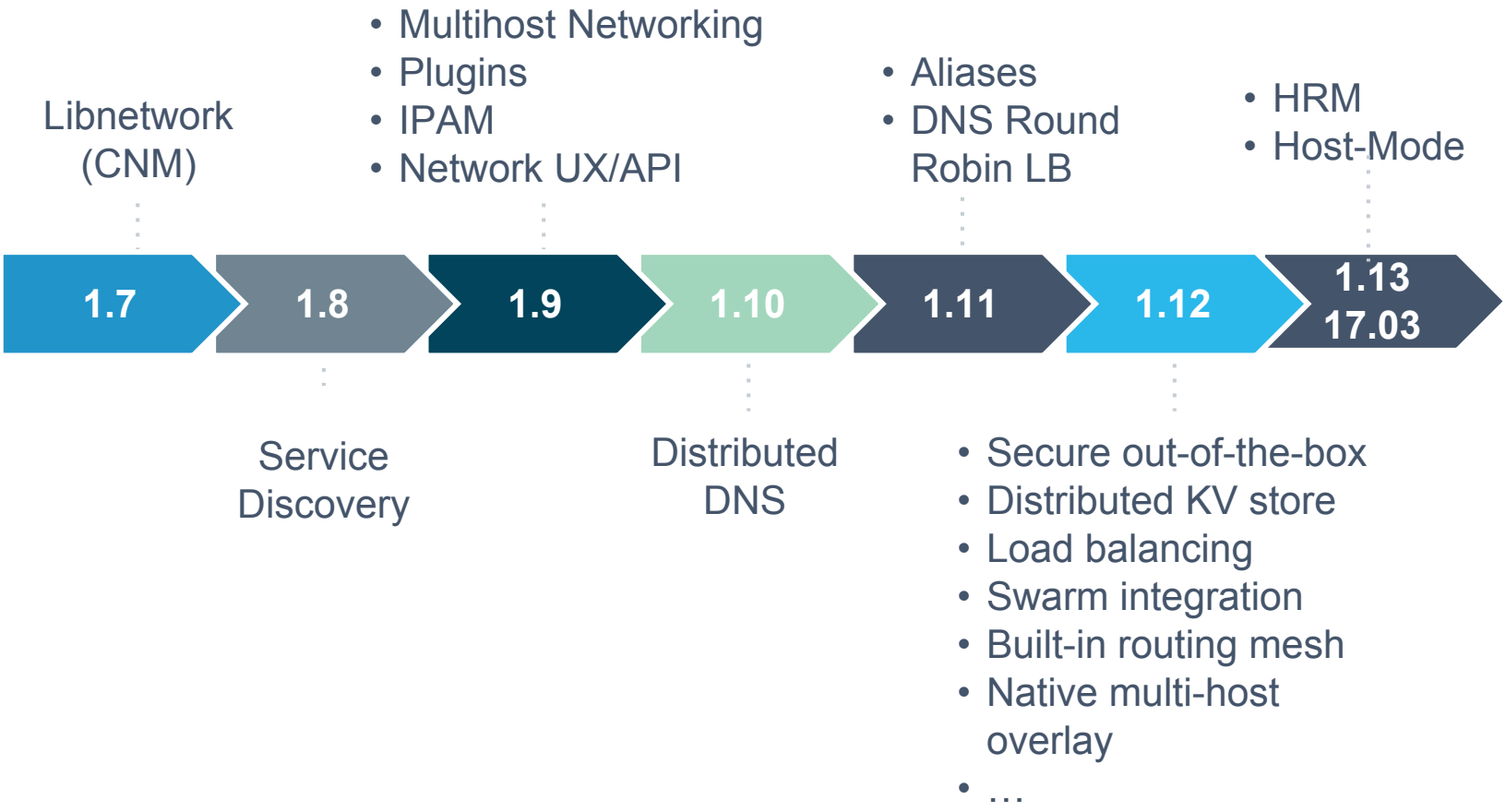


Detailed Overview: Summary

- 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**

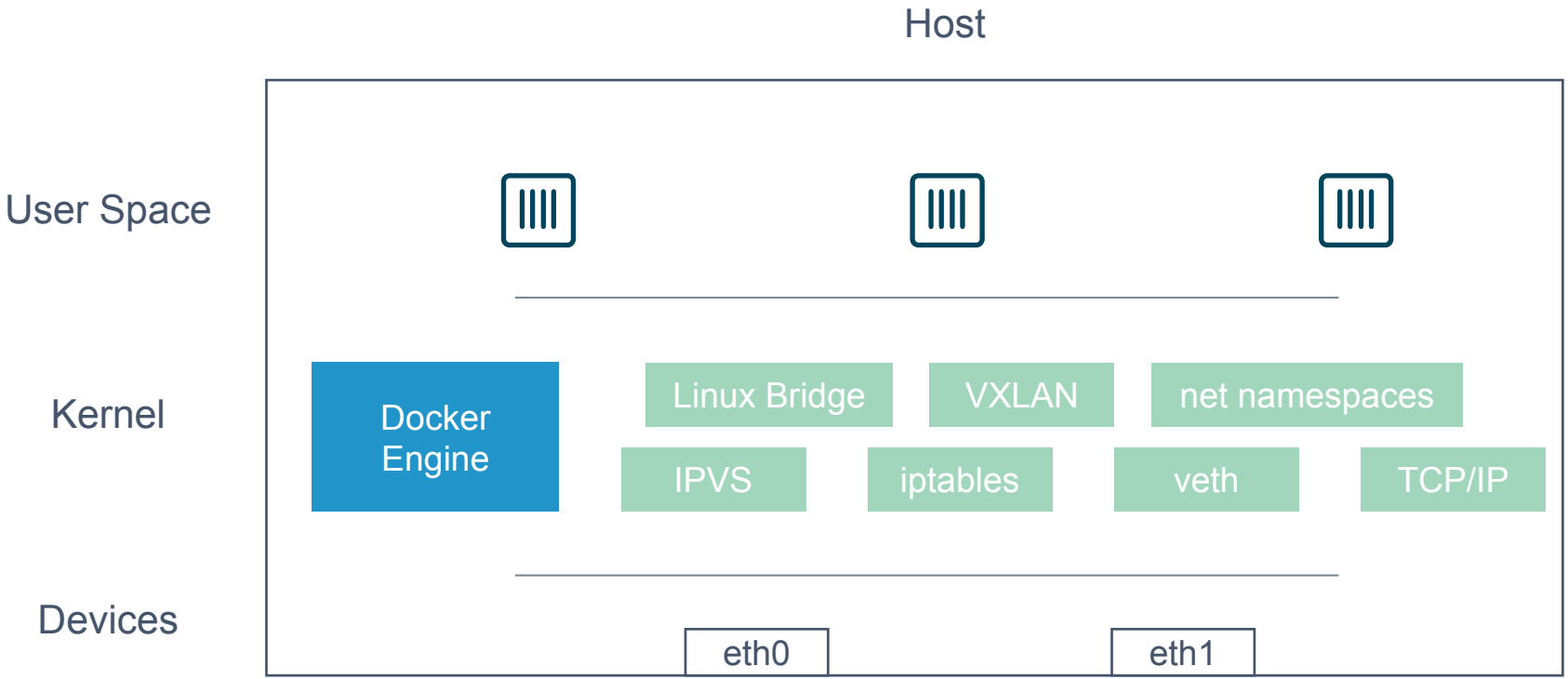
Docker Networking Fundamentals


Libnetwork (CNM)
• Multihost Networking
• Plugins
• IPAM
• Network UX/API
• Aliases
• DNS Round Robin LB
• HRM
• Host-Mode
1.7
1.8
1.9
1.10
1.11
1.12
1.13 17.03
Service Discovery
Distributed DNS
• Secure out-of-the-box
• Distributed KV store
• Load balancing
• Swarm integration
• Built-in routing mesh
• Native multi-host overlay
• …

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

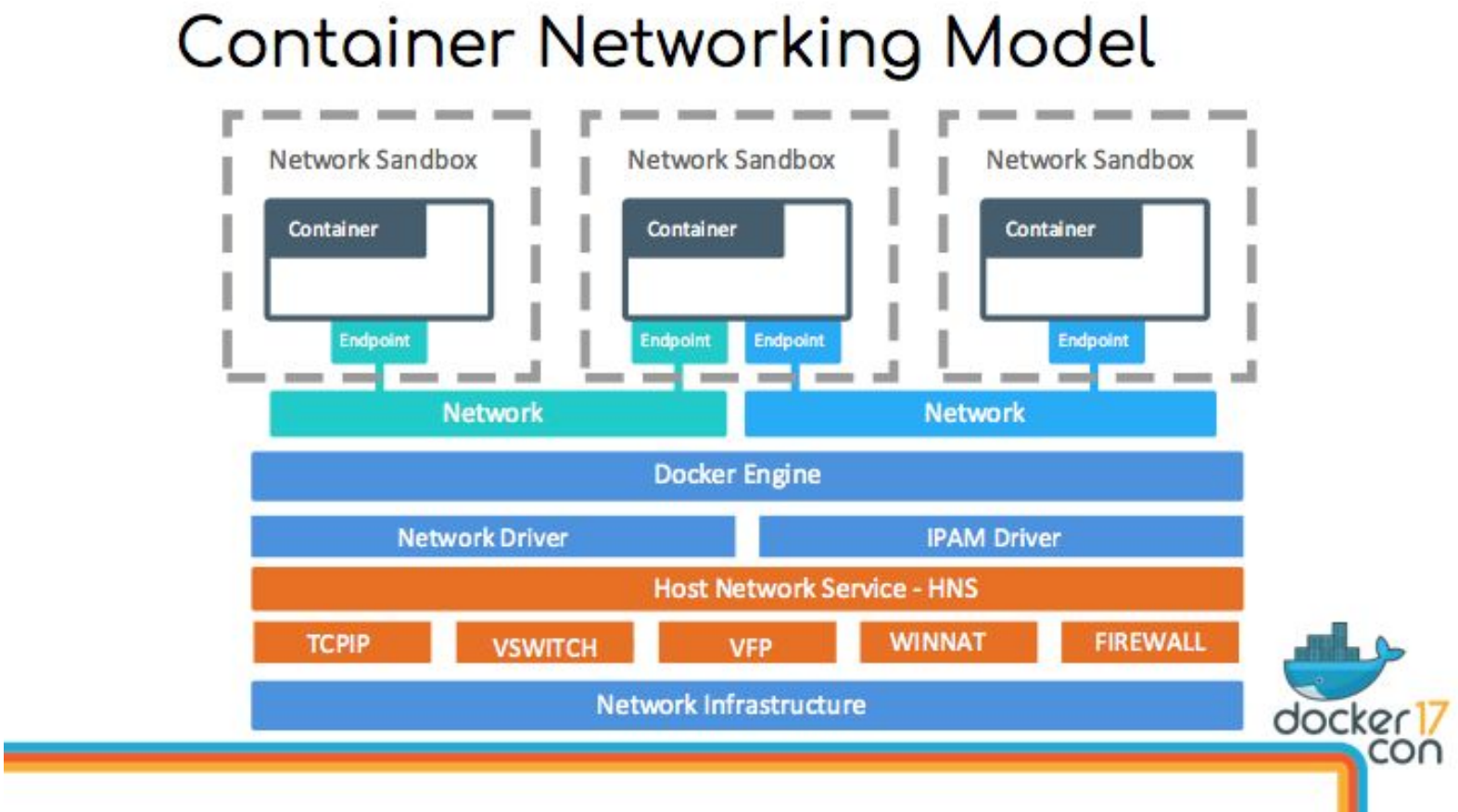
Linux

- Network Namespace
- Linux Bridge
- Virtual Ethernet Devices
- IP Tables

Windows

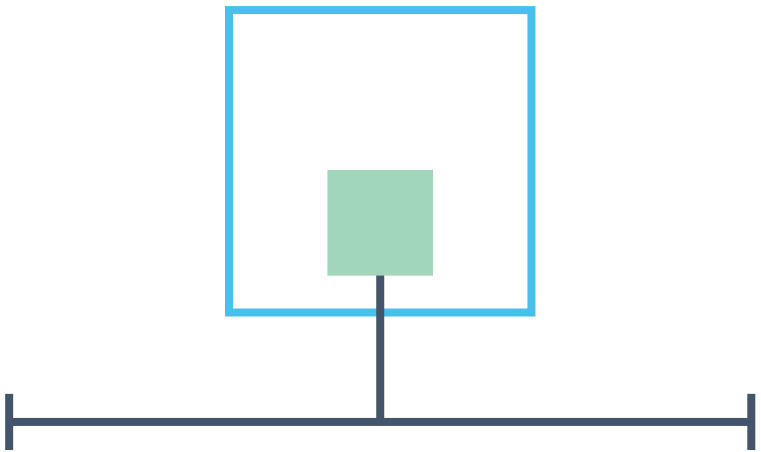
- Network Compartments
- VSwitch
- Virtual nics
- Firewall & VFP Rules

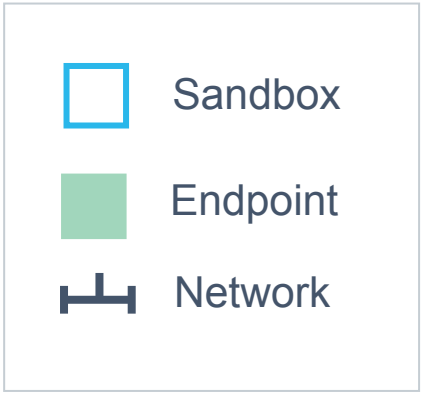
Docker Windows Networking



docker17 con
docker

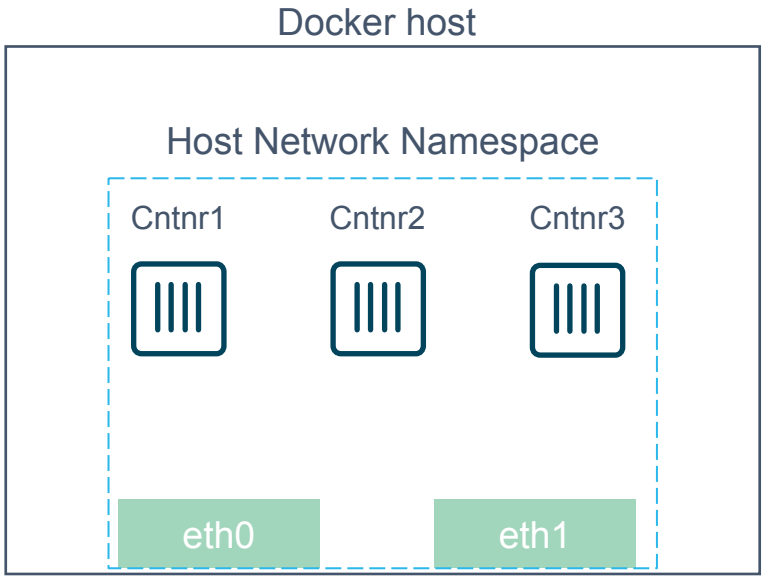
Container Network Model (CNM)



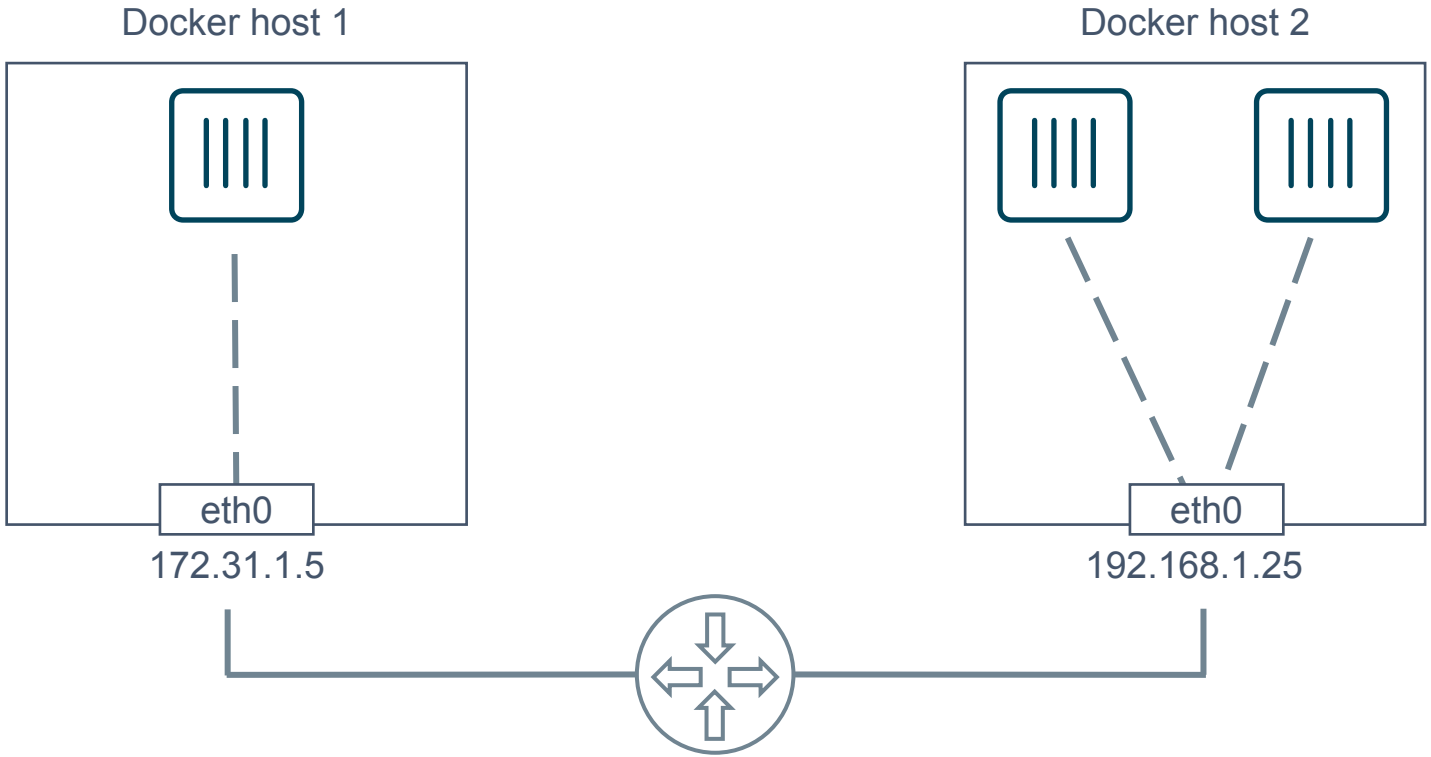


Linux Networking with Containers

- 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

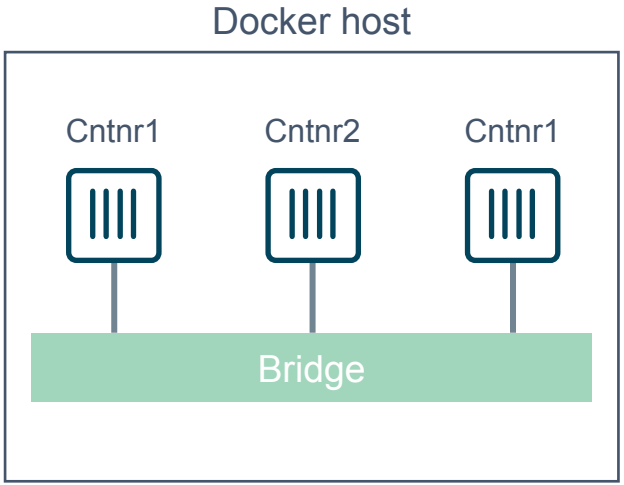
Lab Section 1

Bridge Driver

What is Docker Bridge Networking?

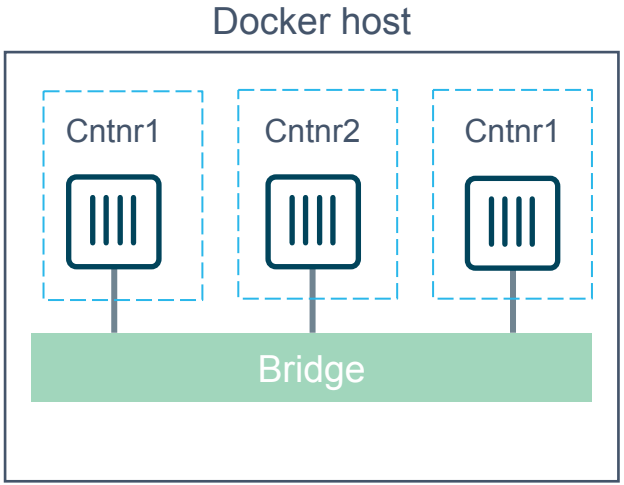
Single-host networking!

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

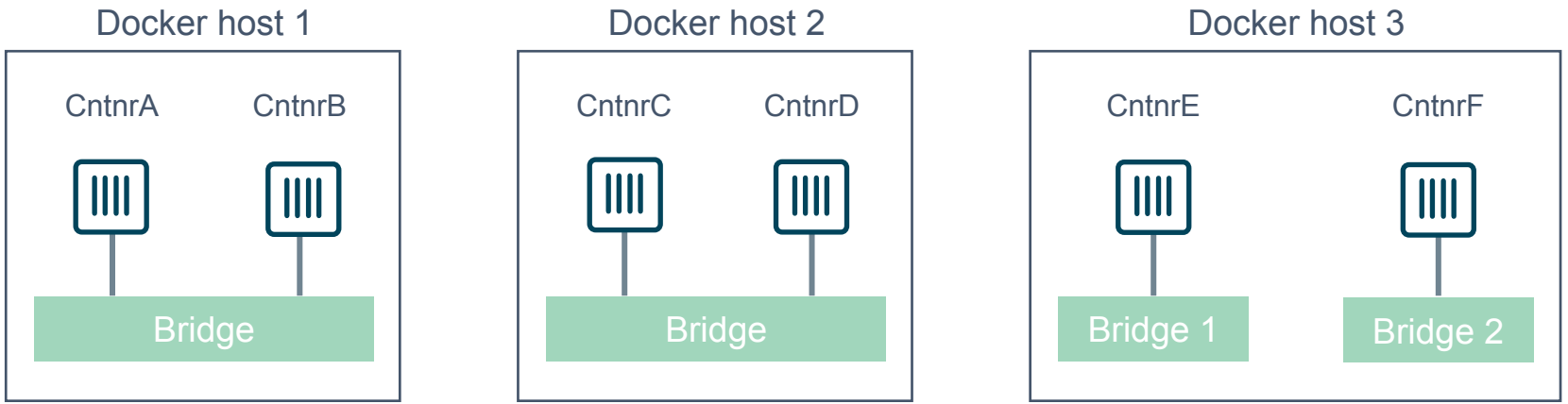


What is Docker Bridge Networking?

- 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



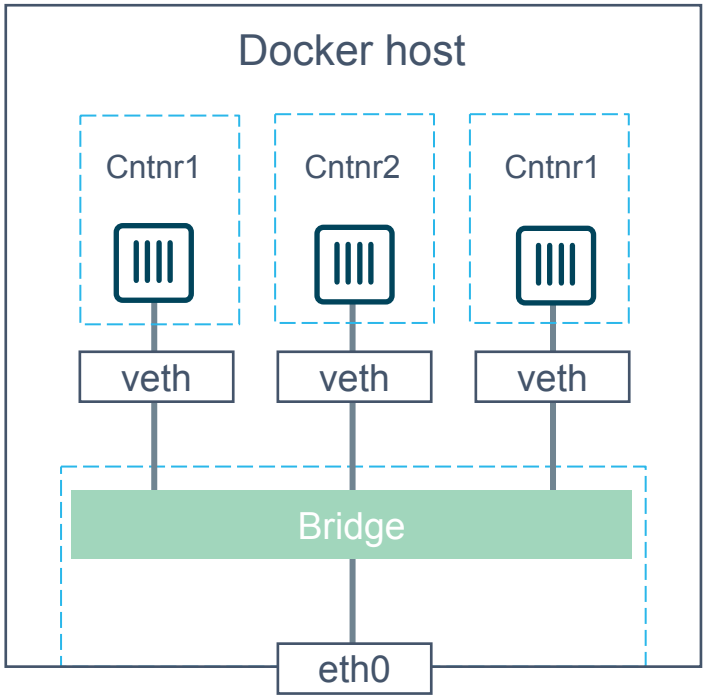
What is Docker Bridge Networking?



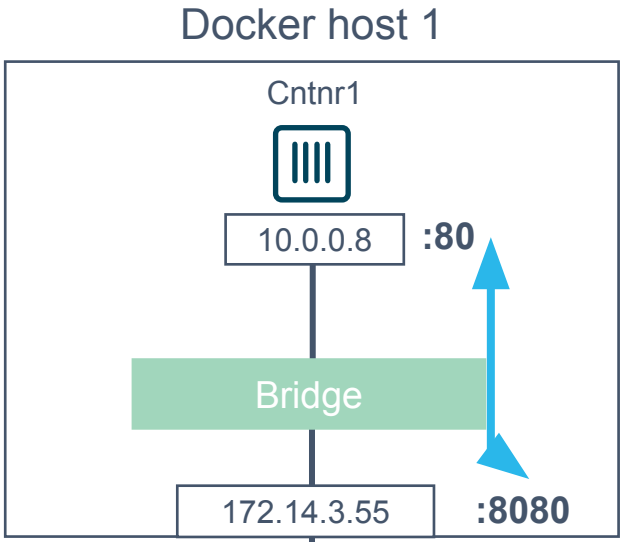
Containers on different **bridge** networks cannot communicate

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



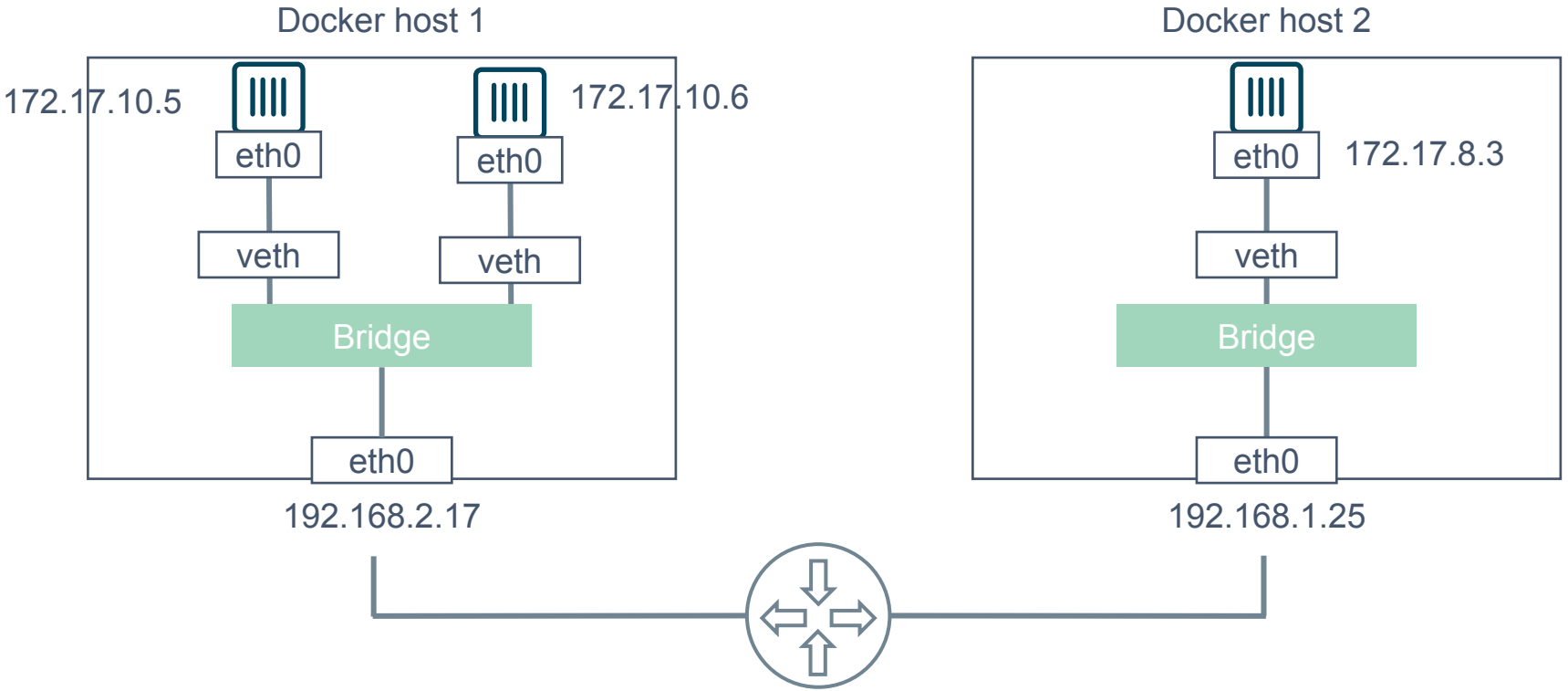
Docker Bridge Networking and Port Mapping



Host port Container port

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

Bridge Mode Data Flow



Demo

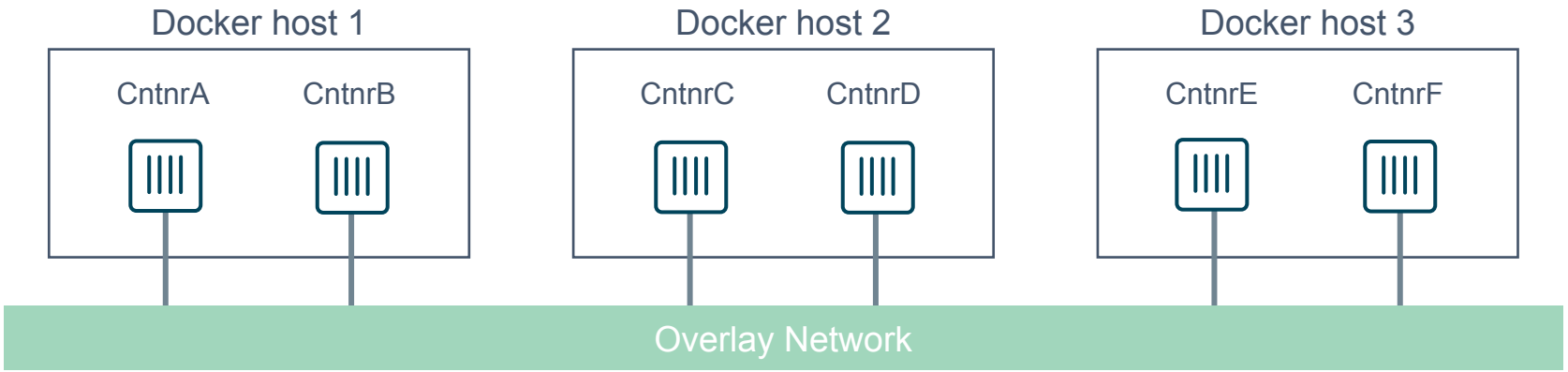
BRIDGE

Lab Section 2

Overlay Driver

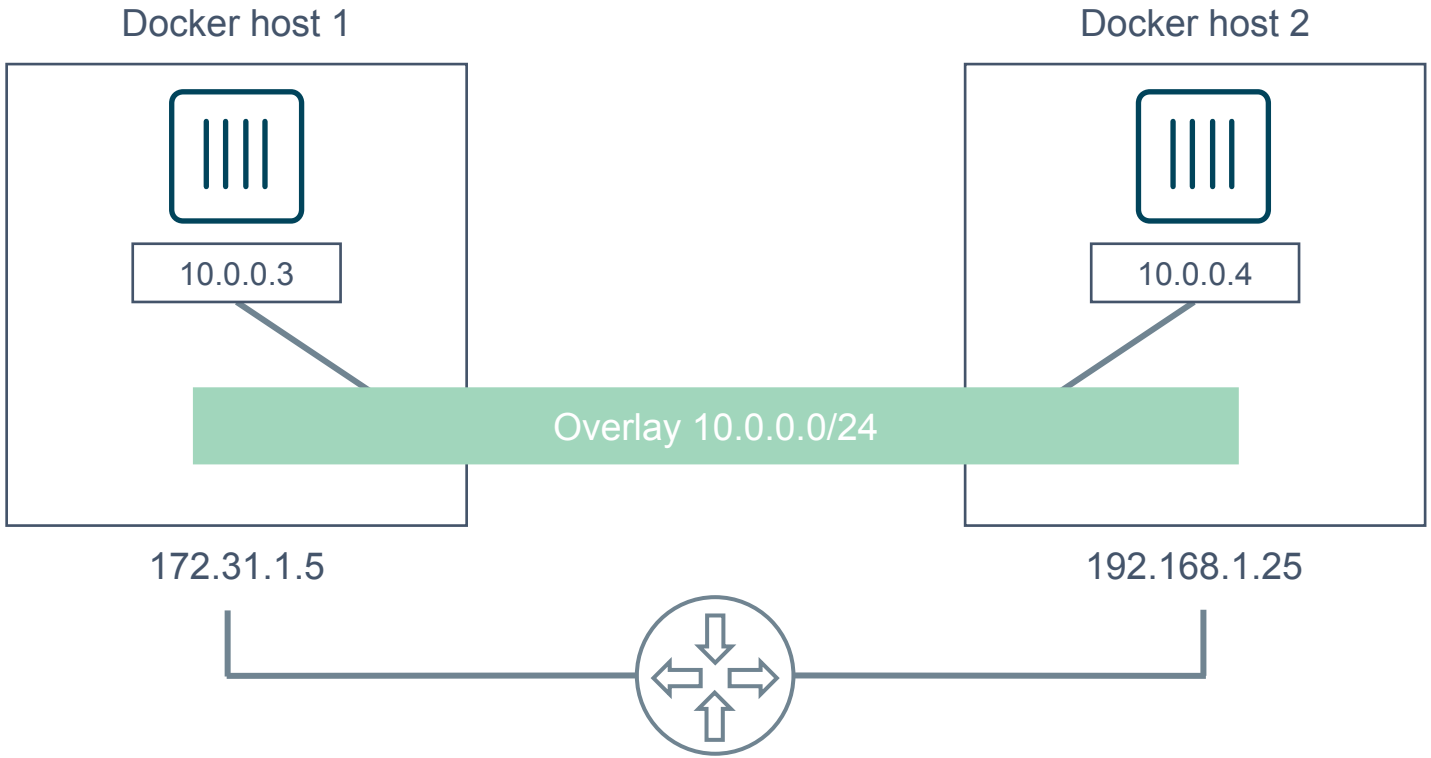
What is Docker Overlay Networking?

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



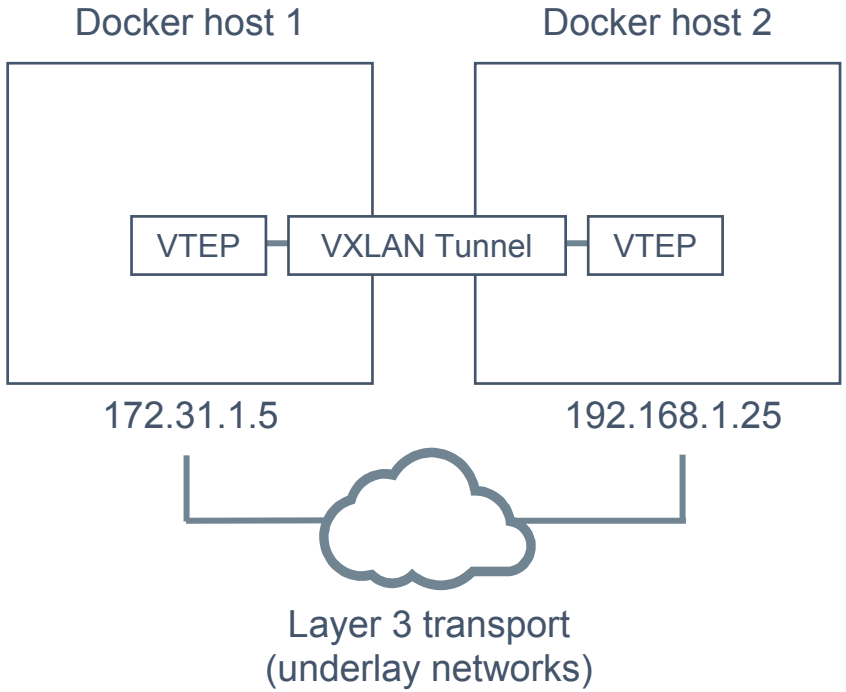
All containers on the **overlay** network can communicate!

Building an Overlay Network (High level)

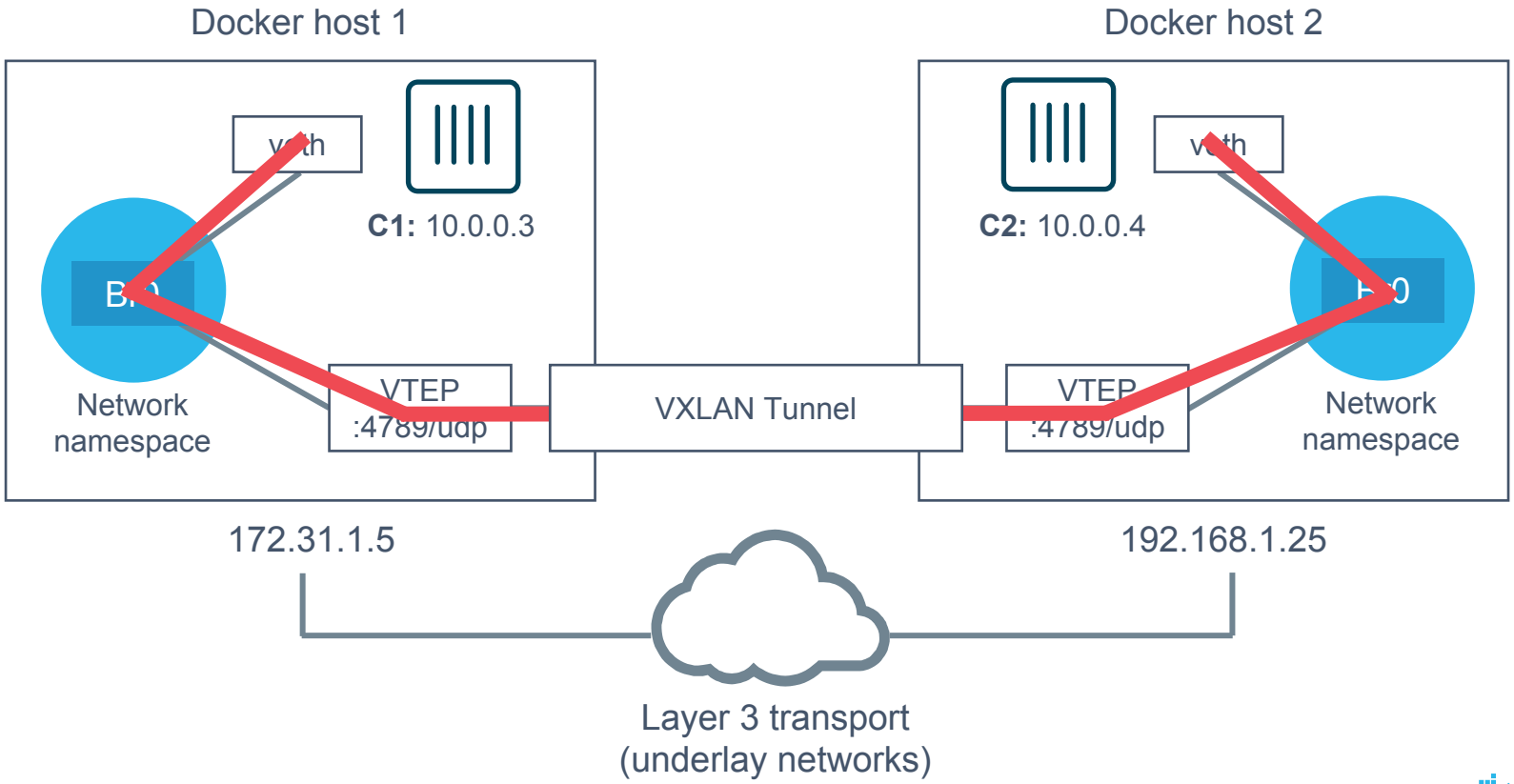


Docker Overlay Networks and VXLAN

- 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

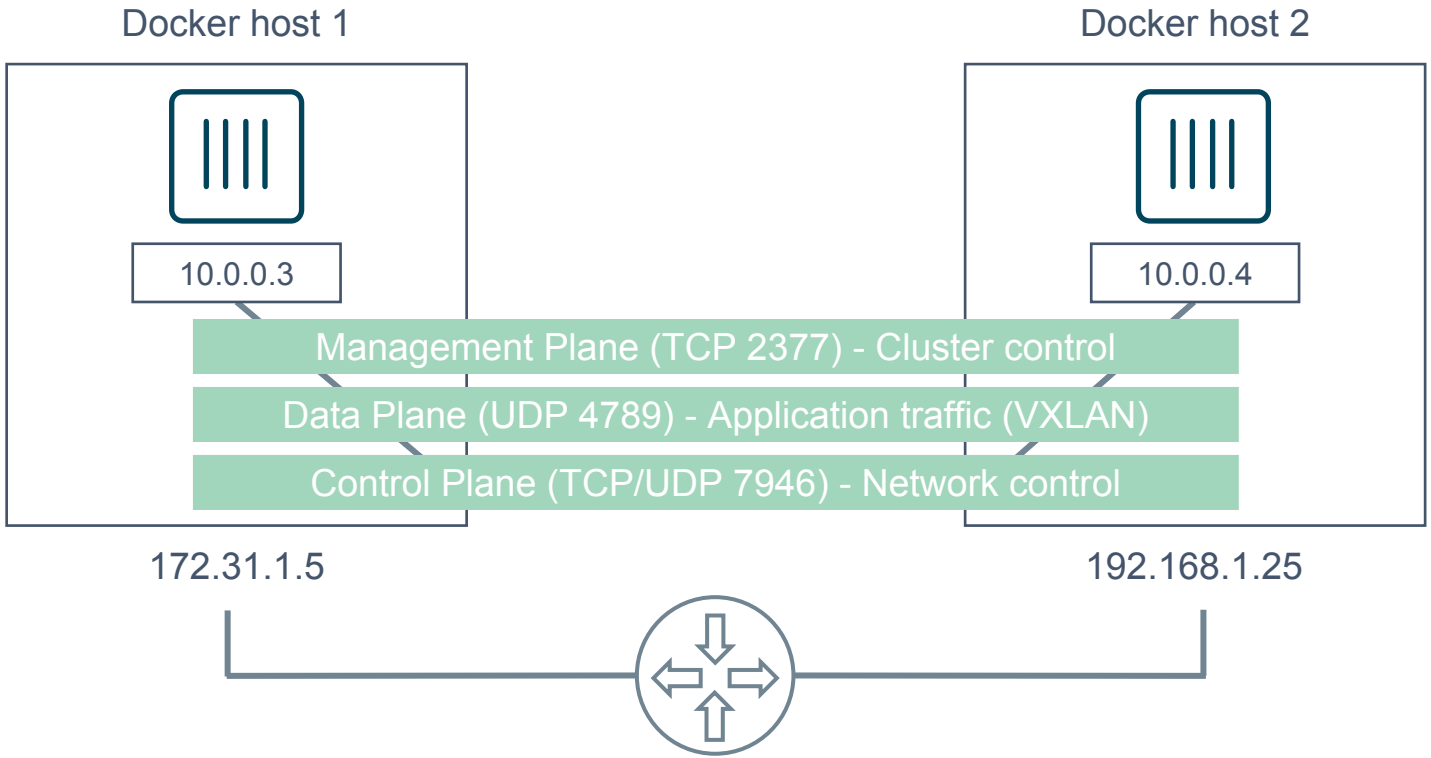


Building an Overlay Network (more detailed)

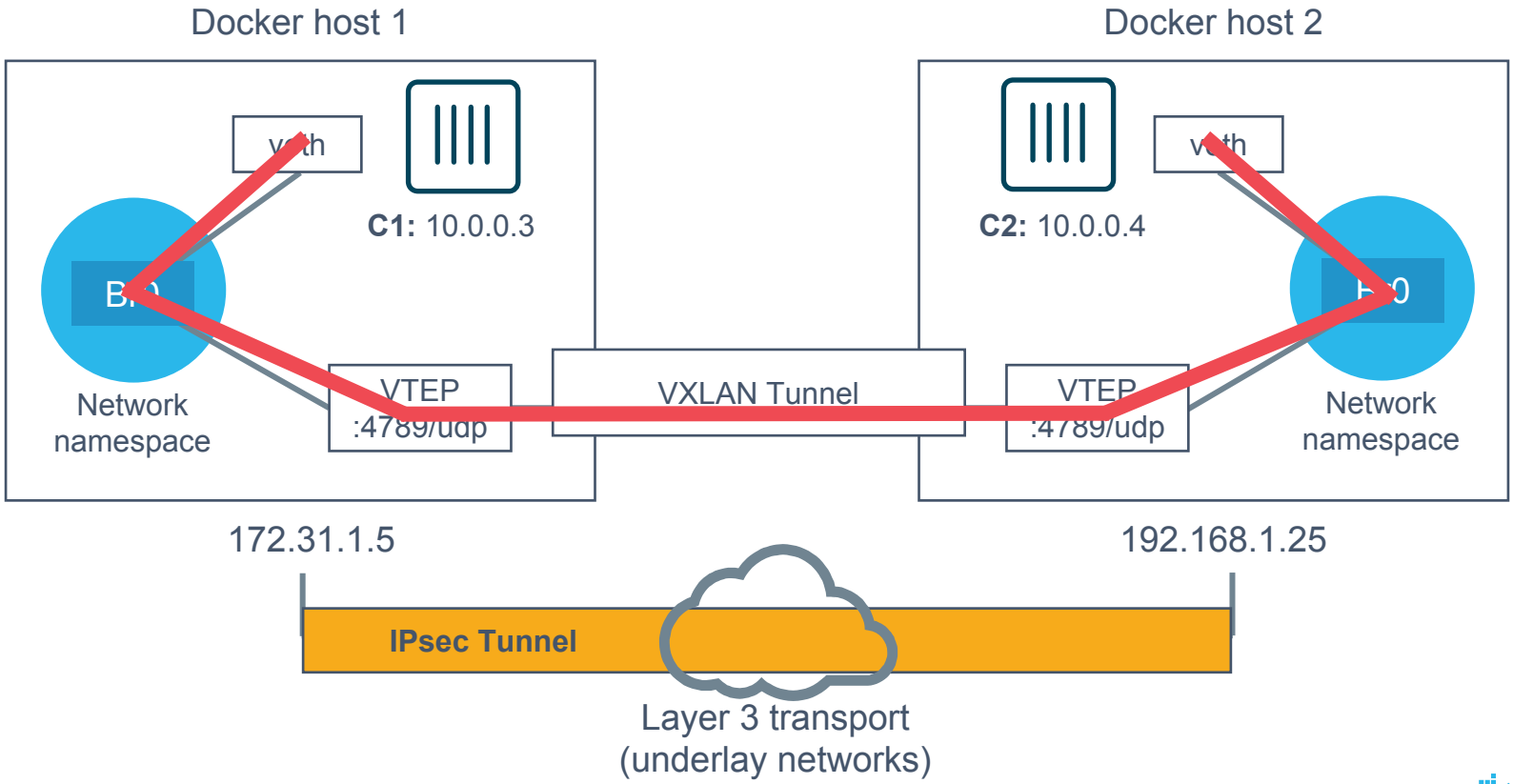


docker

Overlay Networking Ports



Overlay Network Encryption with IPSec



docker

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

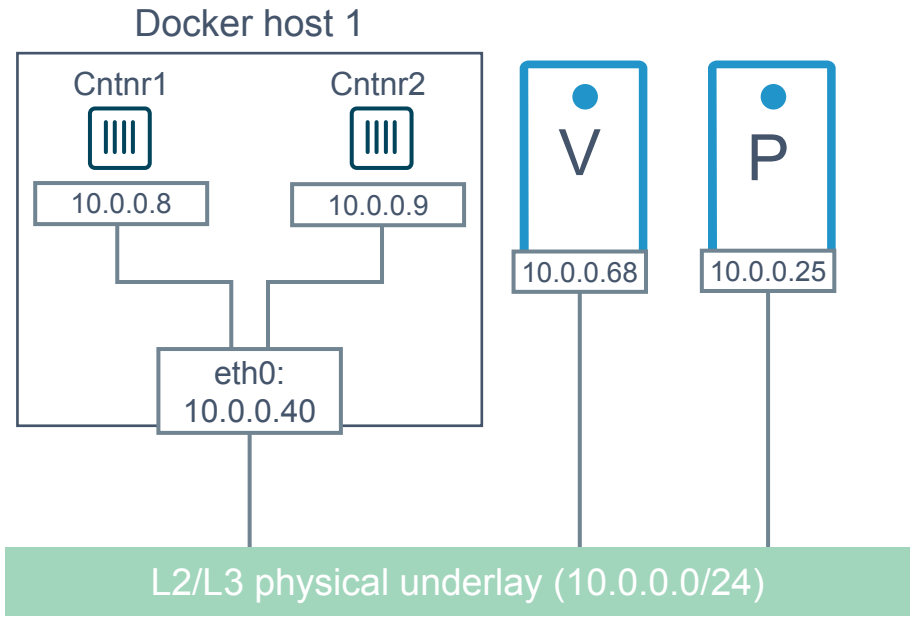
Demo

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?

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

Parent interface has to be connected to physical underlay

Sub-interfaces used to trunk 802.1Q VLANs

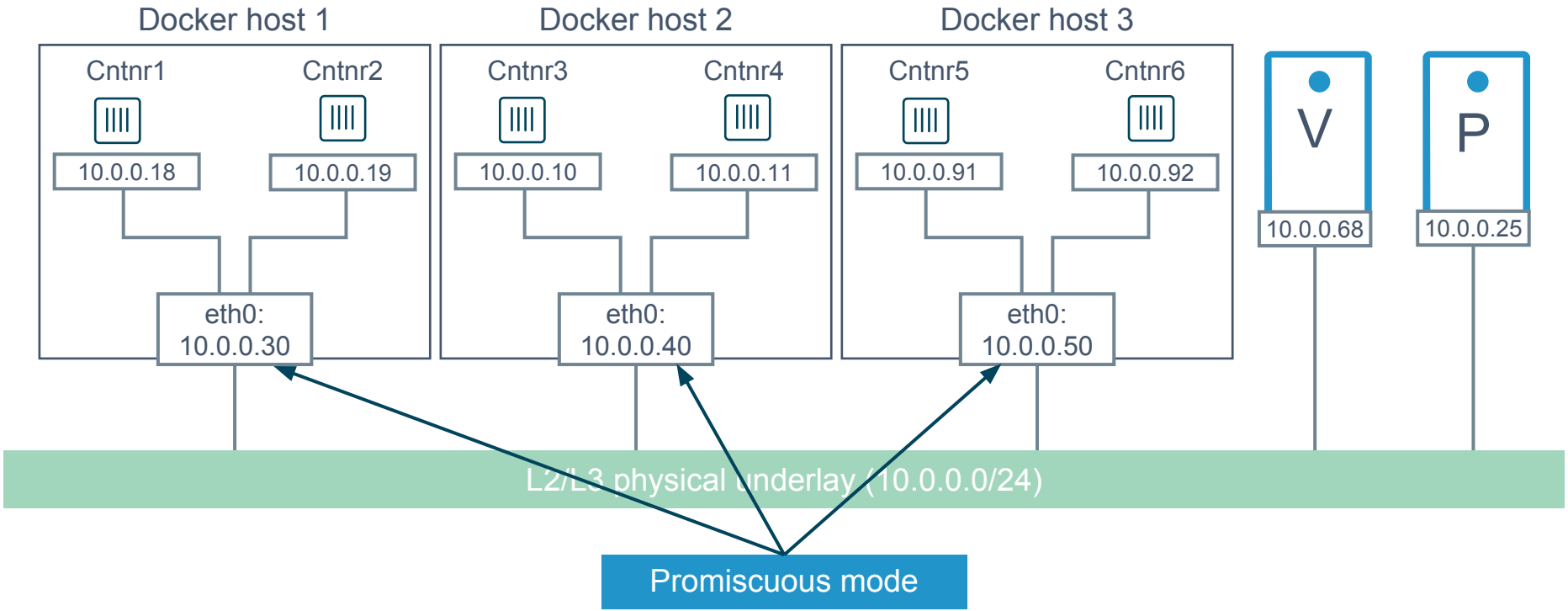
Each container gets its own **MAC** and **IP** on the underlay network

Each container is visible on the physical underlay network

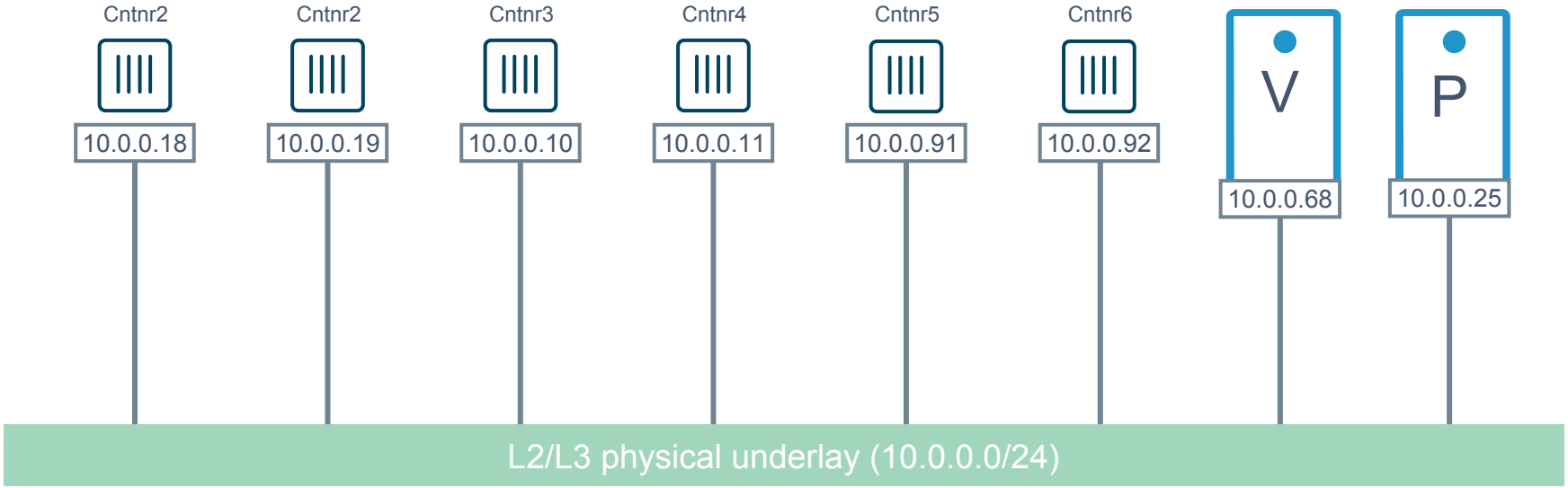
Gives containers direct access to the underlay network **without port mapping** and without a **Linux bridge**

Requires **promiscuous mode**

What is MACVLAN?

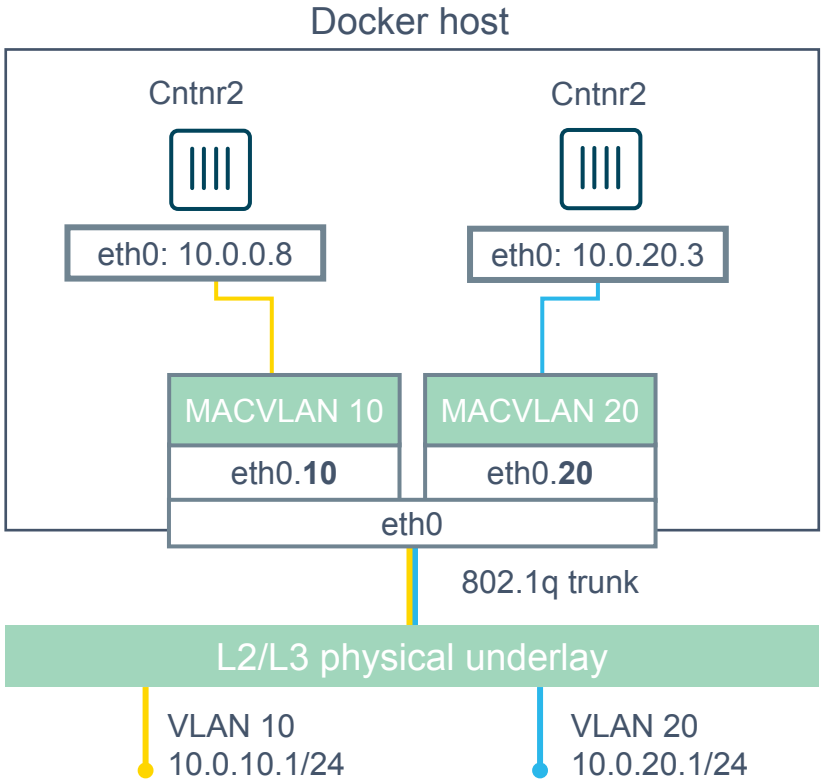


What is MACVLAN?



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



docker

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

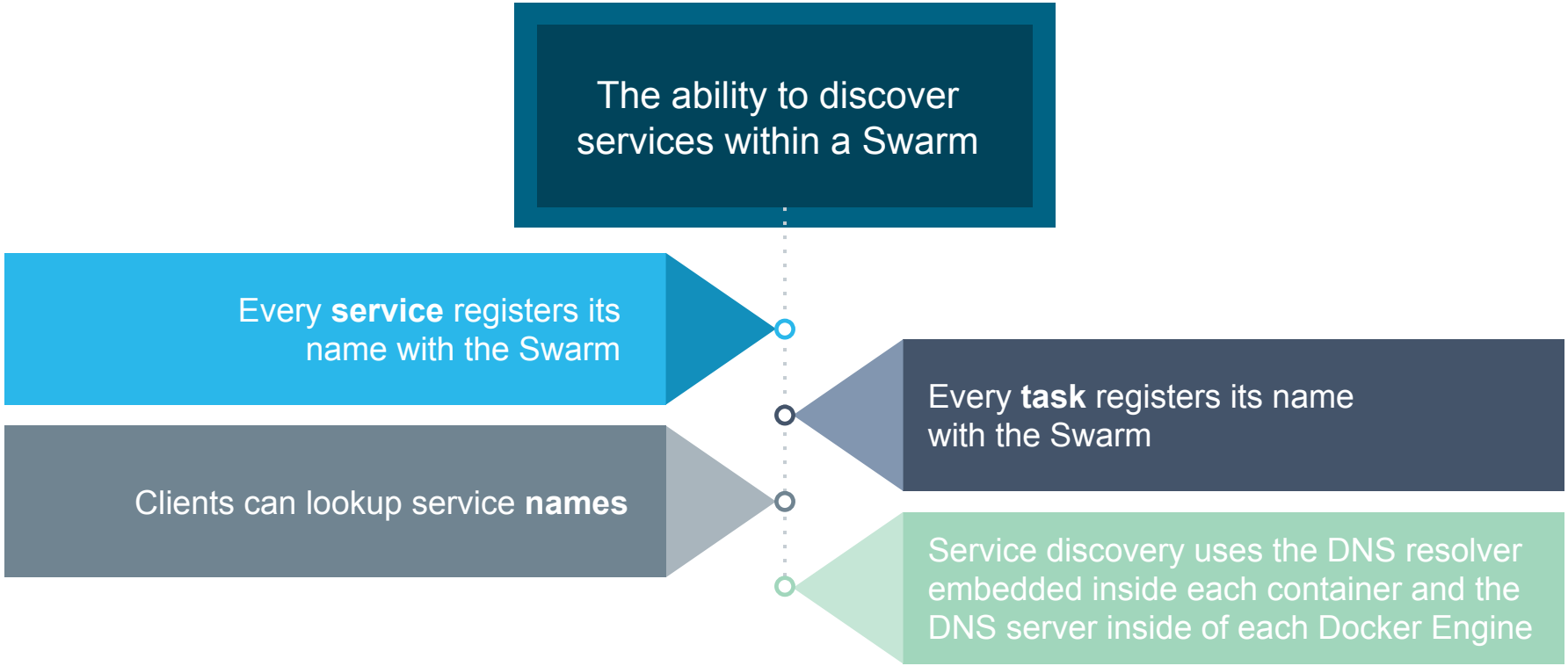
Use Cases Summary

- 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

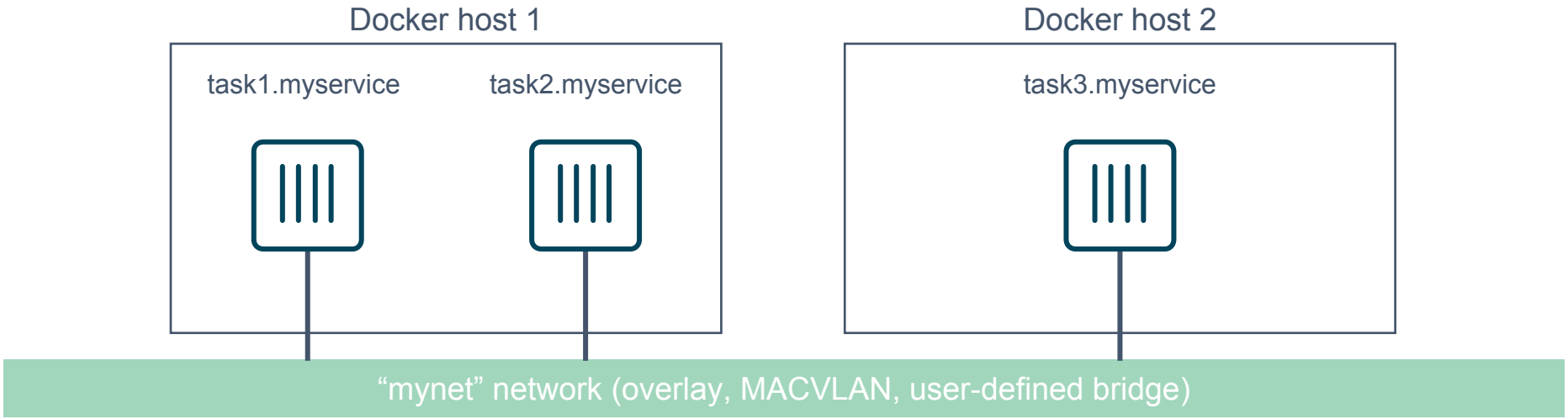
Docker Network Services

SERVICE REGISTRATION, SERVICE DISCOVERY, AND LOAD BALANCING

What is Service Discovery?



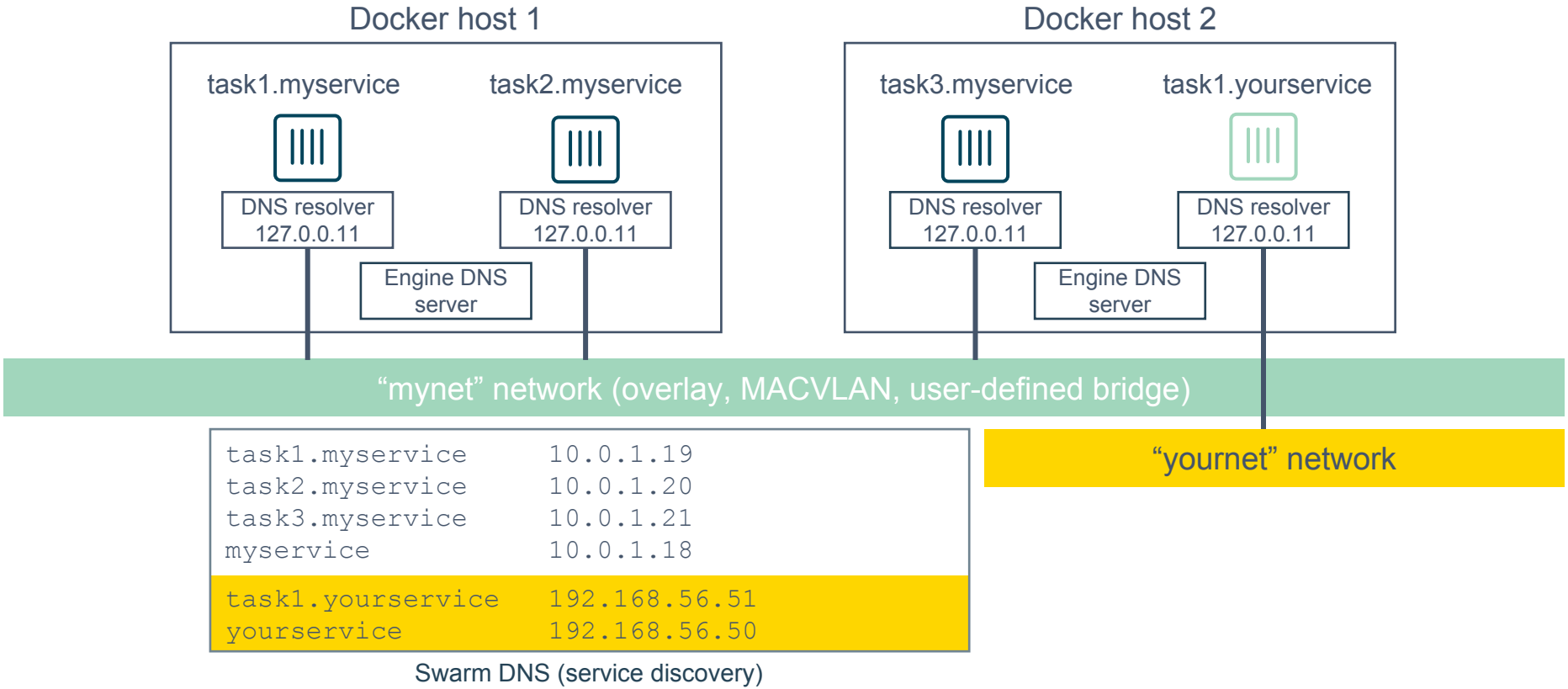
Service Discovery in a Bit More Detail




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

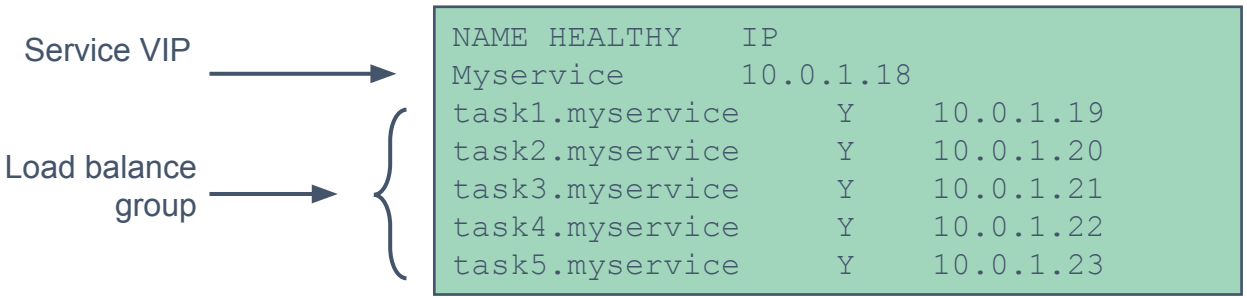
Swarm DNS (service discovery)

Service Discovery in a Bit More Detail



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 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

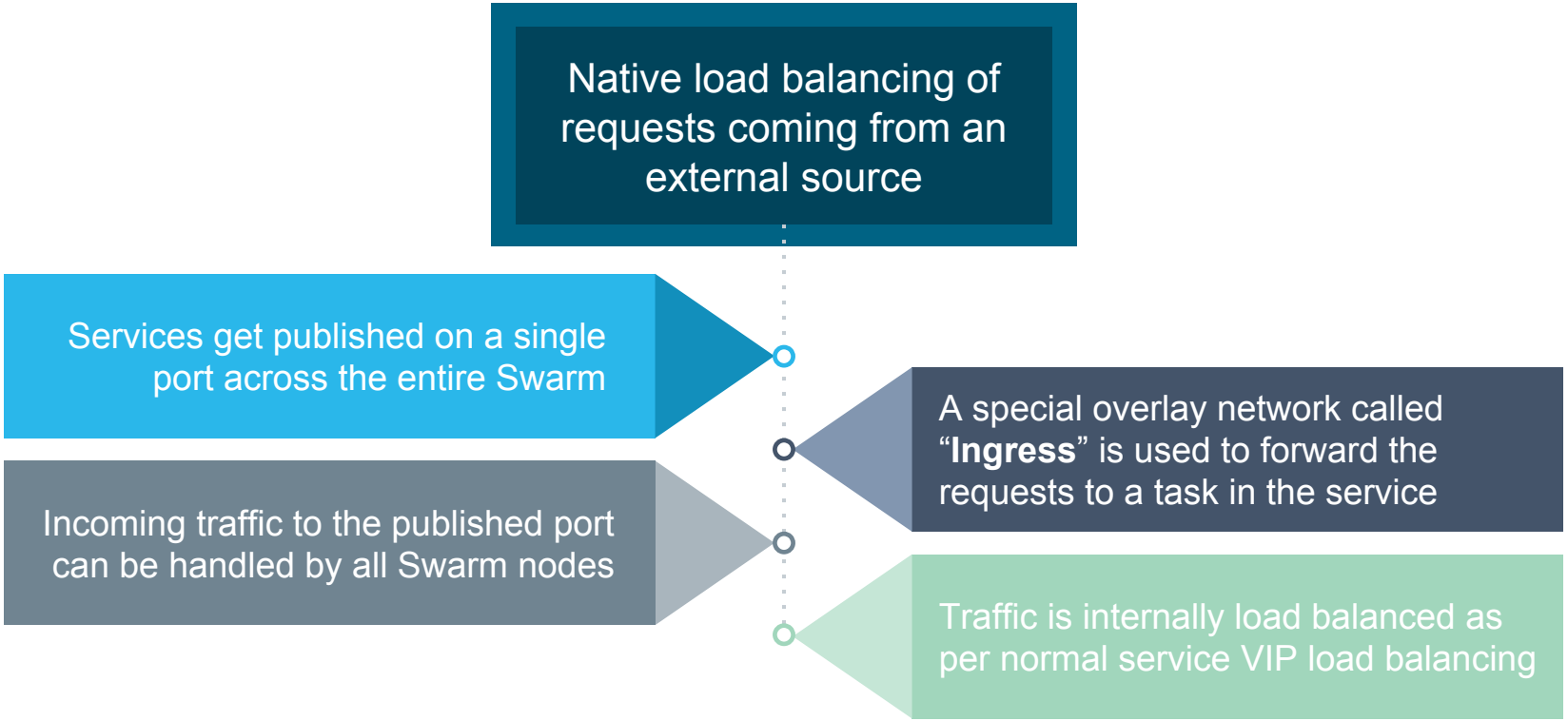
Demo

SERVICE DISCOVERY

Load Balancing External Requests

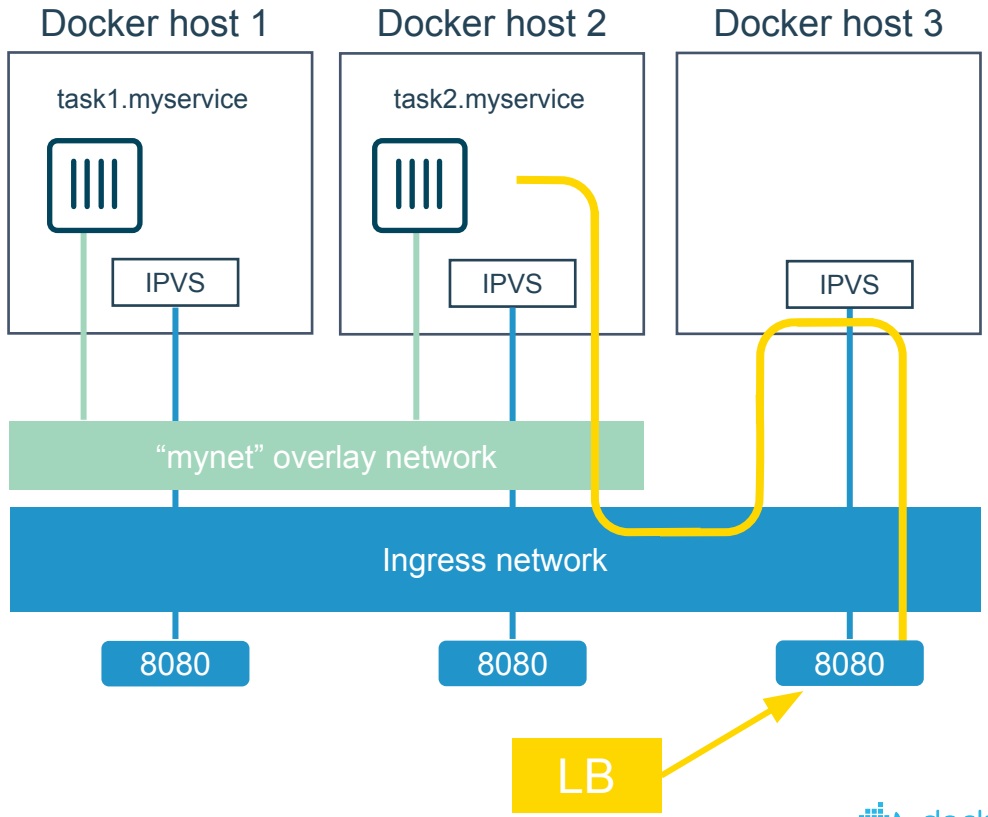
ROUTING MESH

What is the Routing Mesh?



Routing Mesh Example

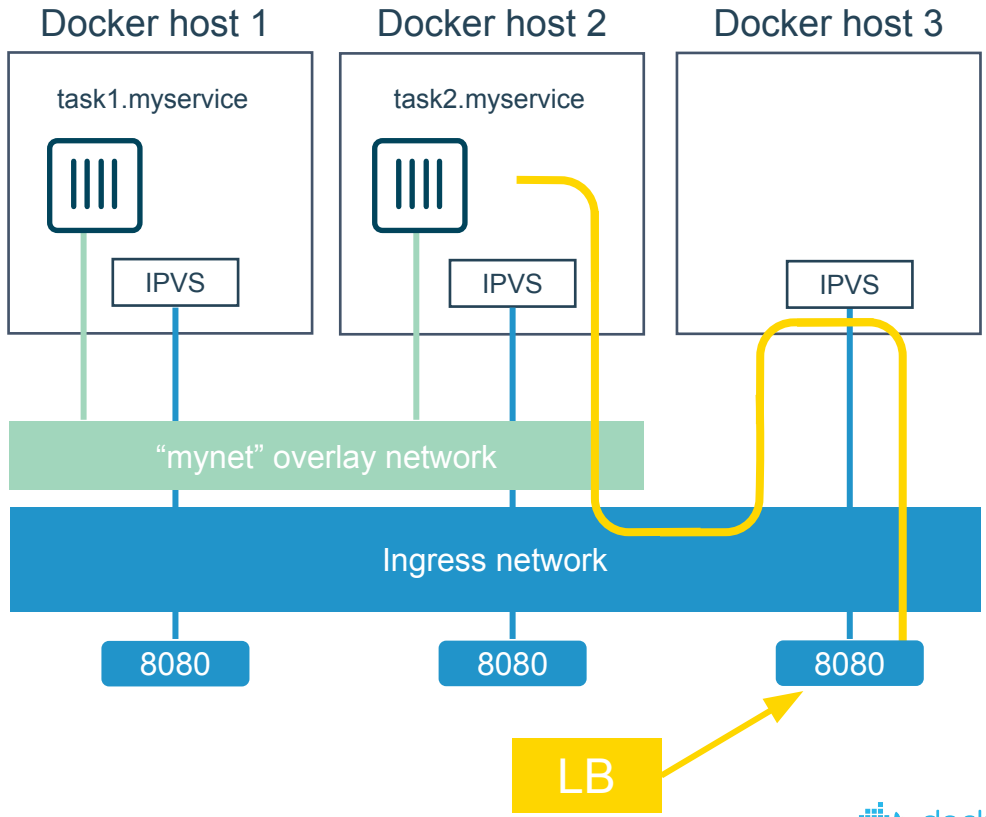
1. Three Docker hosts
2. New service with 2 tasks
3. Connected to the **mynet** overlay network
4. Service published on port 8080 swarm-wide
5. External LB sends request to Docker host 3 on port 8080
6. Routing mesh forwards the request to a healthy task using the ingress network



docker

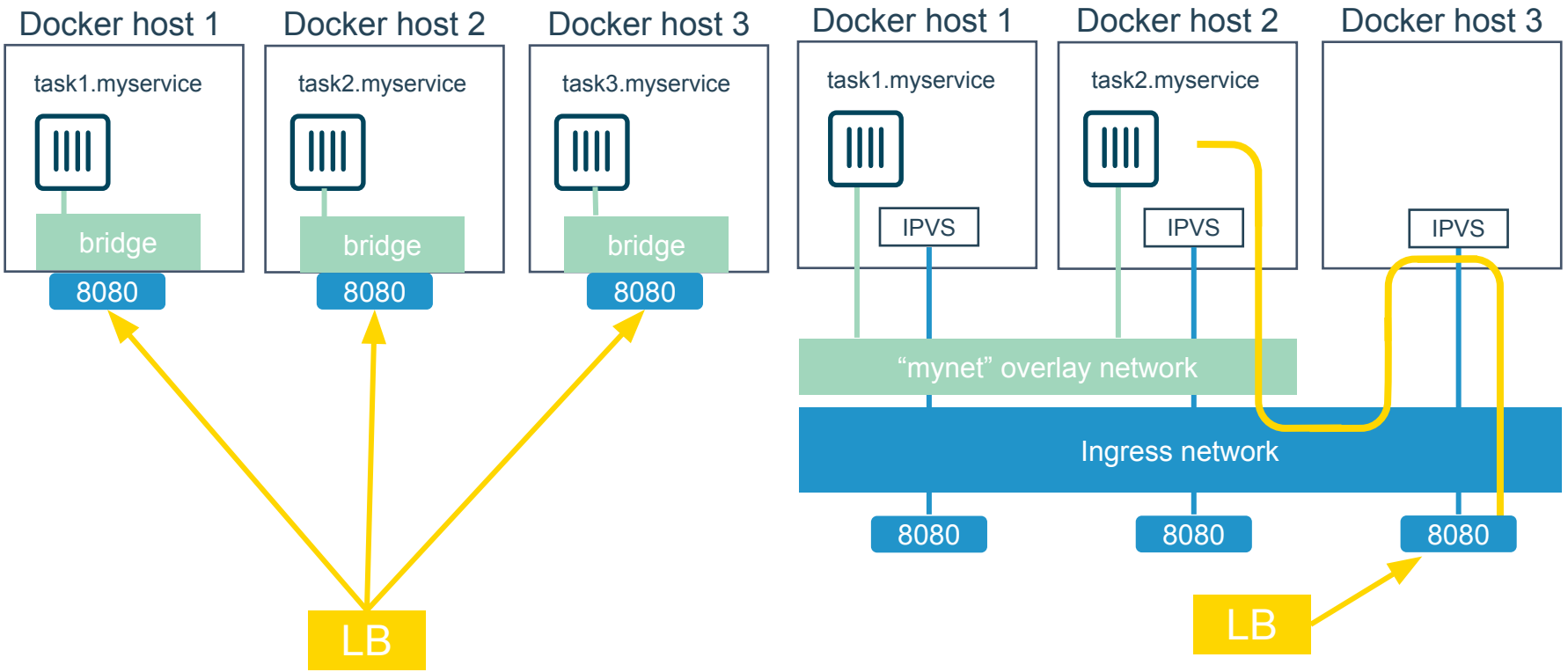
Routing Mesh Example

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



docker

Host Mode vs Routing Mesh



docker

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

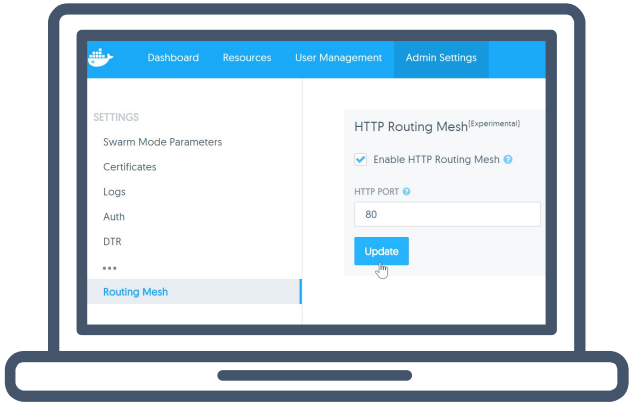
Load balances traffic based on hostnames from HTTP headers

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




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

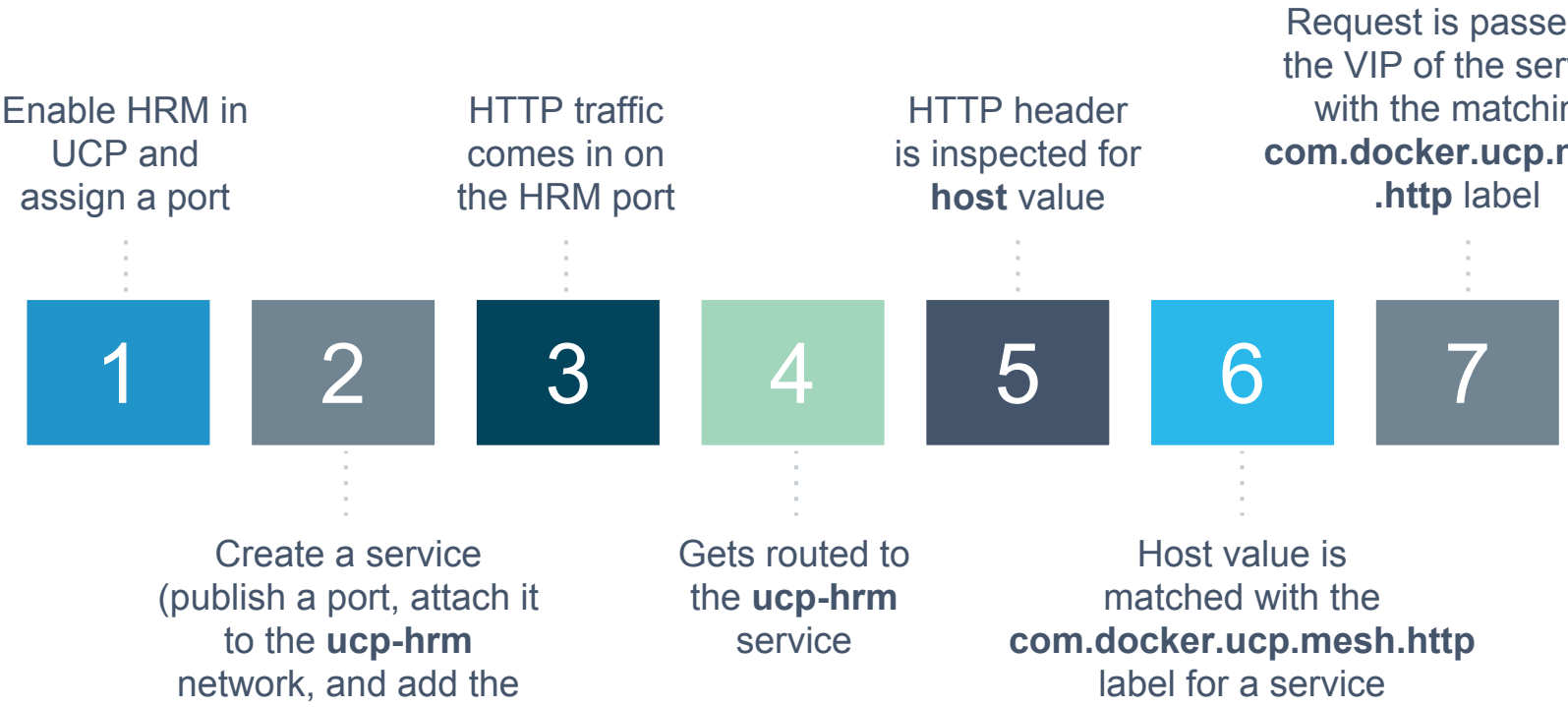
Enable HTTP routing mesh in UCP

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

Create new service

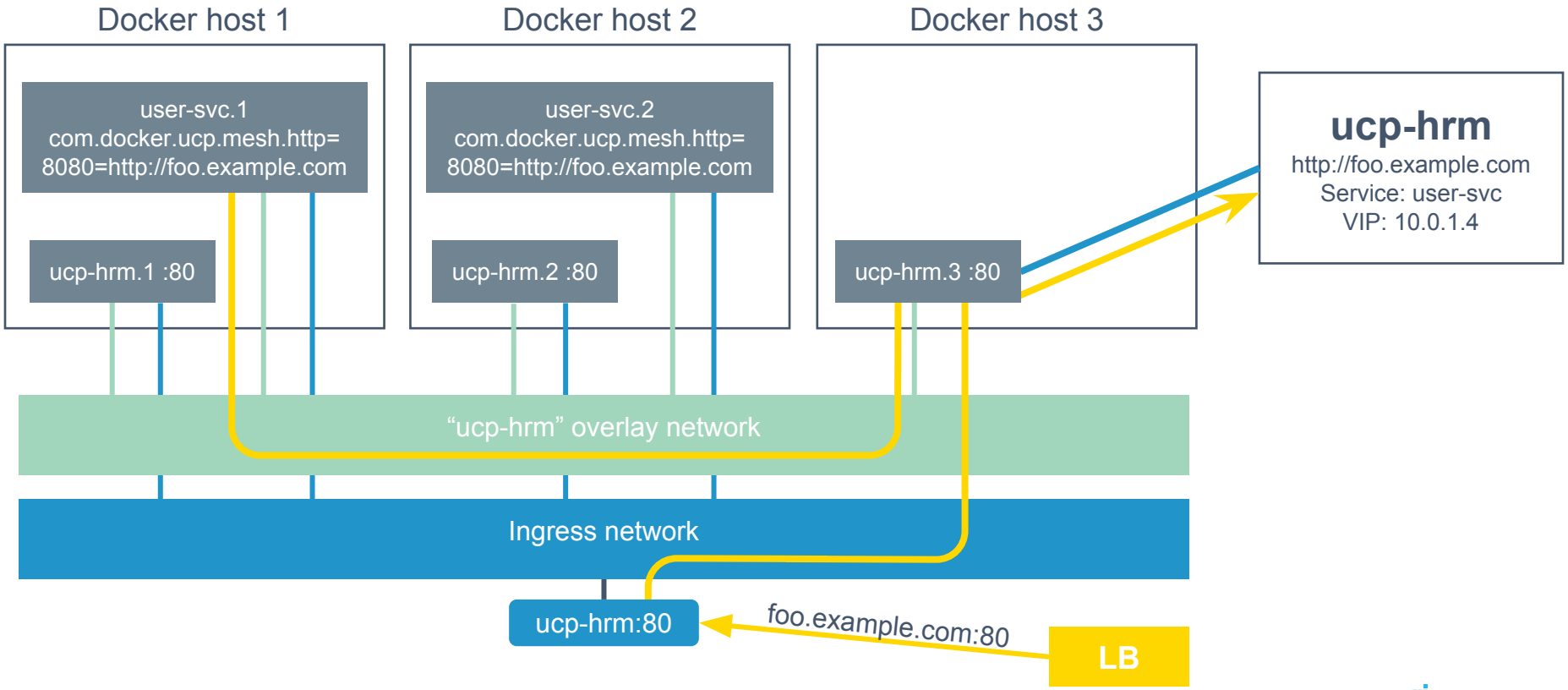
2. a) Add to **ucp-hrm** network
   b) Assign **label** specifying hostname
   (links service to http://foo.example.com)

HTTP Routing Mesh (HRM) Flow



docker

HTTP Routing Mesh Example



docker

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

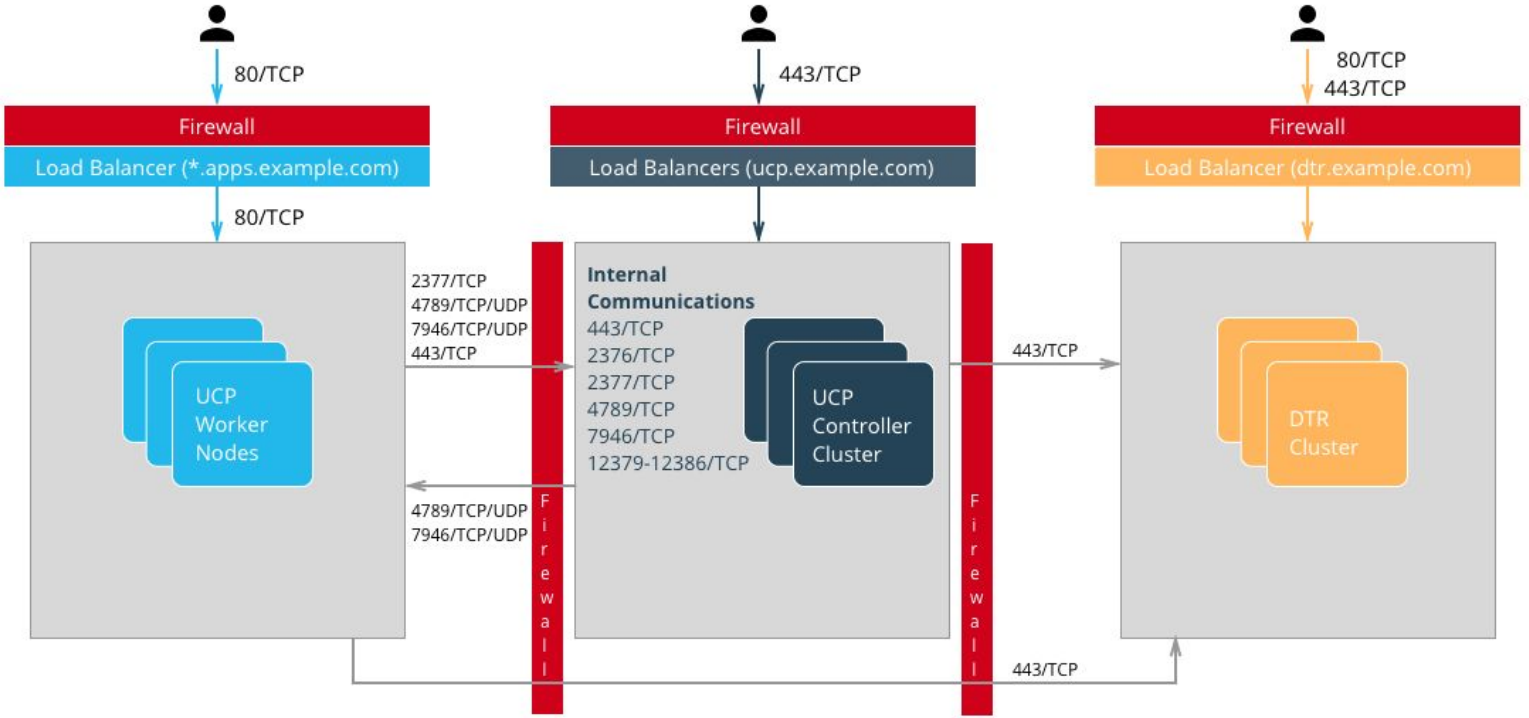
List rules with $ iptables -S, $ iptables -S -t nat

Network state information stale or not being propagated

Destroy and create networks again with same name

General connectivity problems

Required Ports



docker

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

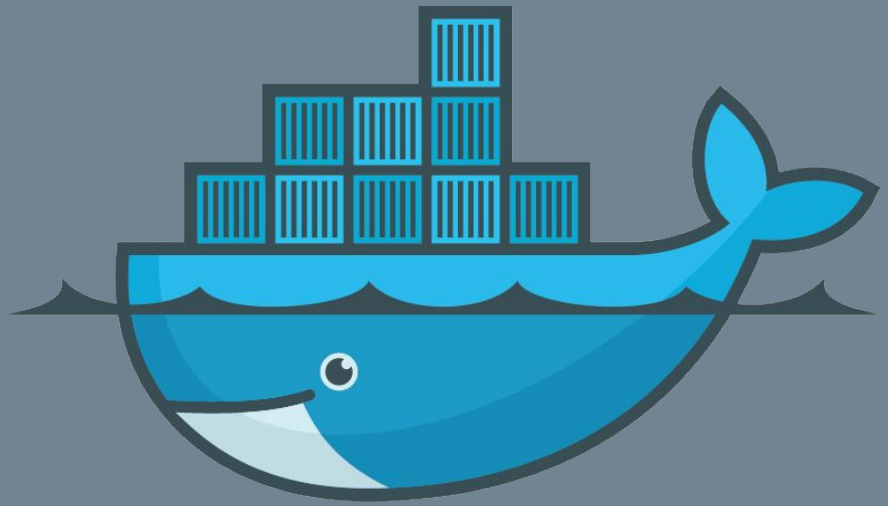
Test DNS service discovery from one service to another

```
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