# Docker Networking with Linux

Guillaume Urvoy-Keller

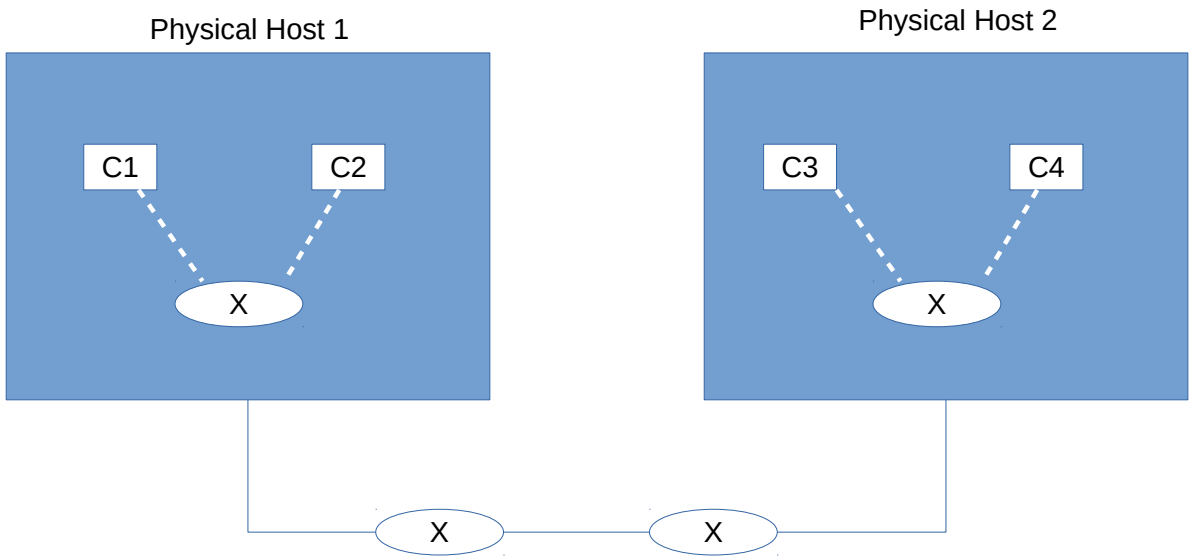
November 11, 2017

# Sources documents

- Laurent Bernaille blog: `http://techblog.d2-si.eu/2017/04/25/deep-dive-into-docker-overlay-networks-part-1.html`
- Docker Networking Cookbook, PacktPub, Jon Langemak
- Docker official documentation
- L3 + VXLAN Made Practical presentation (Openstack summit 2014) by Nolan Leake and Chet Burgess

# Outline

# Reference Scenario



# What we need

- Virtual bridges/switches virtual links inside physical hosts to interconnect:
  - Container to virtual switch
  - Physical interfaces to virtual switches
- Decoupling IP address space from tenants (containers) from the one of data center manager $\Rightarrow$ tunnelling between virtual switches
- Instantiate containers $\Rightarrow$ Docker
- As containers live in different namespaces, we need to move physical interfaces and links between containers.

Similar scenario by replacing containers with VMs

# Circuitry

Linux offers:

- native support of bridges
- native support of virtual links

# Creating a dummy interface (similar to loopback)

The "ip" command is the swiss knife of Linux for manipulating interfaces[1]

- ip link ... ⇒ manipulates interfaces / bridges
- ip add ... ⇒ assigns/removes IP addresses
- ip route ... ⇒ modifies routing tables ; e.g. *ip route show*

```
user@net2:~$ sudo apt−get install iproute2 # what you need to manipulate network
      settings
user@net2:~$ sysctl −w net.ipv4.ip_forward=1 # transforms your machine into a router
user@net2:~$ sudo ip link add dummy0 type dummy
user@net2:~$ sudo ip address add 172.16.10.129/26 dev dummy0
user@net2:~$ sudo ip link set dummy0 up
```

[1] Beware of ifconfig (for instance, it does not see all the addresses of an interface if there are multiple addresses).

# Creating a Linux Bridge

```
user@net1:~$ sudo ip link add host_bridge1 type bridge
user@net1:~$ ip link show host_bridge1
5: host_bridge1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
        DEFAULT group default
    link/ether f6:f1:57:72:28:a7 brd ff:ff:ff:ff:ff:ff
user@net1:~$ sudo ip address add 172.16.10.1/26 dev host_bridge1 # assigns an IP
        address to the interface to make it layer 3 aware (enables to use routing facility of
        kernel)
user@net1:~$ sudo ip link set dev eth1 master host_bridge1 # associate an interface to a
        bridge
user@net1:~$ sudo ip link set dev eth1 nomaster # de-associate
```

# Virtual links

- Need to connect virtual interfaces within the same host
- Linux proposes VETH: Virtual Ethernet, which are pairs of interfaces such that what is sent in one is received in the other
- They can have an IP address to be layer 3 aware.

# VETH pairs

Let us create a second bridge (the first one was host_bridge)

```
user@net1:~$ sudo ip link add edge_bridge1 type bridge
user@net1:~$ sudo ip link add host_veth1 type veth peer name edge_veth1 # create a
        VETH pair specifying the ends name
user@net1:~$ ip link show
...<Additional output removed for brevity>...
13: edge_veth1@host_veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc
        noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 0a:27:83:6e:9a:c3 brd ff:ff:ff:ff:ff:ff
14: host_veth1@edge_veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc
        noop state DOWN mode DEFAULT group default qlen 1000
            link/ether c2:35:9c:f9:49:3e brd ff:ff:ff:ff:ff:ff
```

# Side note...

Put all this up as this is not the default:

```
user@net1:~$ sudo ip link set host_bridge1 up
user@net1:~$ sudo ip link set edge_bridge1 up
user@net1:~$ sudo ip link set host_veth1 up
user@net1:~$ sudo ip link set edge_veth1 up
```

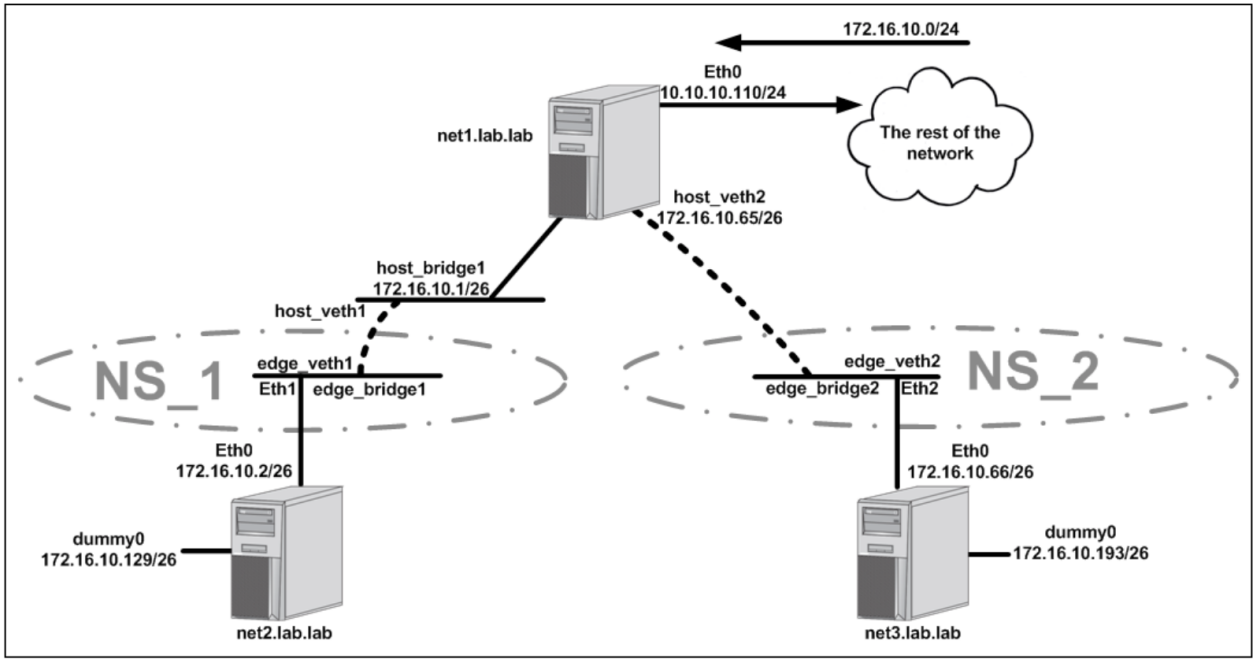
How to distinguish between a bridge or a simple interface or a veth: use **ip -d link** + name of interface:

```
root@ubuntu−xenial:/sys/class/net/enp0s3# ip −d link show dev docker0
6: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group
        default
    link/ether 02:42:86:07:6e:98 brd ff:ff:ff:ff:ff:ff promiscuity 0
    bridge forward_delay 1500 hello_time 200 max_age 2000 ageing_time 30000 stp_state 0 priority 32768 vlan_filtering
            0 vlan_protocol 802.1Q addrgenmode eui64
root@ubuntu−xenial:/sys/class/net/enp0s3# ip −d link show dev enp0s3
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DEFAULT group
        default qlen 1000
    link/ether 02:d2:3e:0e:ff:c0 brd ff:ff:ff:ff:ff:ff promiscuity 0 addrgenmode eui64
root@ubuntu−xenial:/sys/class/net/enp0s3# ip −d link show dev veth84e2b4a
17: veth84e2b4a@if16: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP
        mode DEFAULT group default
    link/ether 72:14:0f:4d:d1:28 brd ff:ff:ff:ff:ff:ff link−netnsid 0 promiscuity 1
    veth # this is a veth connected to docker0
    bridge_slave state forwarding priority 32 cost 2 hairpin off guard off root_block off fastleave off learning on flood on
            addrgenmode eui64
```

# Network Namespaces

- Network namespaces allow you to create isolated views of the network.
- Allows to mimic Virtual Routing and Forwarding (VRF) instances available in most modern networking hardware (e.g. Cisco Switches).

# Scenario to implement (Docker Networking Cookbook)



# Network Namespaces

```
user@net1:~$ sudo ip netns add ns_1
user@net1:~$ sudo ip netns add ns_2
user@net1:~$ ip netns list
ns_2
ns_1
```

## Create the bridges inside the namespaces

```
user@net1:~$ sudo ip netns exec ns_1 ip link add edge_bridge1 type bridge
user@net1:~$ sudo ip netns exec ns_2 ip link add edge_bridge2 type bridge
```

# Network Namespaces

## Do an **ip link show** inside a given ns namespace

```
user@net1:~$ sudo ip netns exec ns_1 ip link show
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: edge_bridge1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode
       DEFAULT group default
    link/ether 26:43:4e:a6:30:91 brd ff:ff:ff:ff:ff:ff
```

## We next move the interfaces eth1 and eth2 within the namespaces + one side of the VETH pairs

```
user@net1:~$ sudo ip link set dev eth1 netns ns_1
user@net1:~$ sudo ip link set dev edge_veth1 netns ns_1
user@net1:~$ sudo ip link set dev eth2 netns ns_2
user@net1:~$ sudo ip link set dev edge_veth2 netns ns_2
```

# For sake of completness

We have done the hard work. For sake of completness, we need to plug the VETH inside NS to the switchs and put everything up:

```
user@net1:~$ sudo ip netns exec ns_1 ip link set dev edge_veth1 master edge_bridge1
user@net1:~$ sudo ip netns exec ns_1 ip link set dev eth1 master edge_bridge1
user@net1:~$ sudo ip netns exec ns_2 ip link set dev edge_veth2 master edge_bridge2
user@net1:~$ sudo ip netns exec ns_2 ip link set dev eth2 master edge_bridge2
```

```
user@net1:~$ sudo ip netns exec ns_1 ip link set edge_bridge1 up
user@net1:~$ sudo ip netns exec ns_1 ip link set edge_veth1 up
user@net1:~$ sudo ip netns exec ns_1 ip link set eth1 up
user@net1:~$ sudo ip netns exec ns_2 ip link set edge_bridge2 up
user@net1:~$ sudo ip netns exec ns_2 ip link set edge_veth2 up
user@net1:~$ sudo ip netns exec ns_2 ip link set eth2 up
```

# Lab1: how a basic container is connected

Instructions:

- Start a simple debian or ubuntu container. You can deconnect (w/o killing) with ^P^Q
- Make a drawing with the interface connected to the outside of the container. Which kind of interface it is? Check also the routing table.
- From the host, find the sibling interface and where is it connected to.
- Launch a ping to the outside (the gw of your host) and check with an **watch iptables -L -v** which iptables are used for the FILTER table and a **watch iptables -L -v -t nat** for the NAT table. Explain in your report what is happening globally.

# Lab1: how a basic container is connected

- Start a container with an exposed port like 80:

```
docker run -it --name debian -p80 debian /bin/bash
```

- Check the exposed port with **docker port debian**
- Check the iptables rule
- Check what happens with a netcat on the correct port (**nc localhost exposed_port -v**). You need to be in verbose mode
- Wait a minute: there was no active web server and still, you managed to establish the TCP connection. Convince yourself with a wget or curl that it is the case.
- Do a simple **ps aux | grep docker-proxy** to understand what happens.

# Docker advanced networking functions

You have a set of predefined networks:

```
root@ubuntu−xenial: docker network ls
NETWORK ID NAME DRIVER SCOPE
bfb14981a5df bridge bridge local
b7c327787044 host host local
492f4a9fe233 none null local
```

# Docker bridge mode (this is bridge0!)

```
root@ubuntu−xenial:/sys/class/net/enp0s3# docker network inspect bridge
    {
        "Name": "bridge",
        "Id": "bfb14981a5df08fe27881557d87bc0be18185cd0ba64f7552e196fd1bbad1260",
        "Created": "2017−10−20T14:49:36.899406866Z",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
                "Config": [
                  {
                      "Subnet": "172.17.0.0/16",
                      "Gateway": "172.17.0.1"
                  }
              ]
        },
        [....]
                  },
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
```

# Docker default networking modes

- For bridge, you can adapt:
  - MTU size in case of tunnelling
  - CIDR range
  - GW address
  - ....
- Host mode is when you connect container directly to the host $\Rightarrow$ leads to port contention, e.g., you cannot run multiple replicas of a web server!
- None is.... none

# Custom networks

```
root@ubuntu-xenial:~# docker network create mynetwork
0b396f0fc9264ad7153943f293b863e028c858c4d051744b93128bc21b291336
```

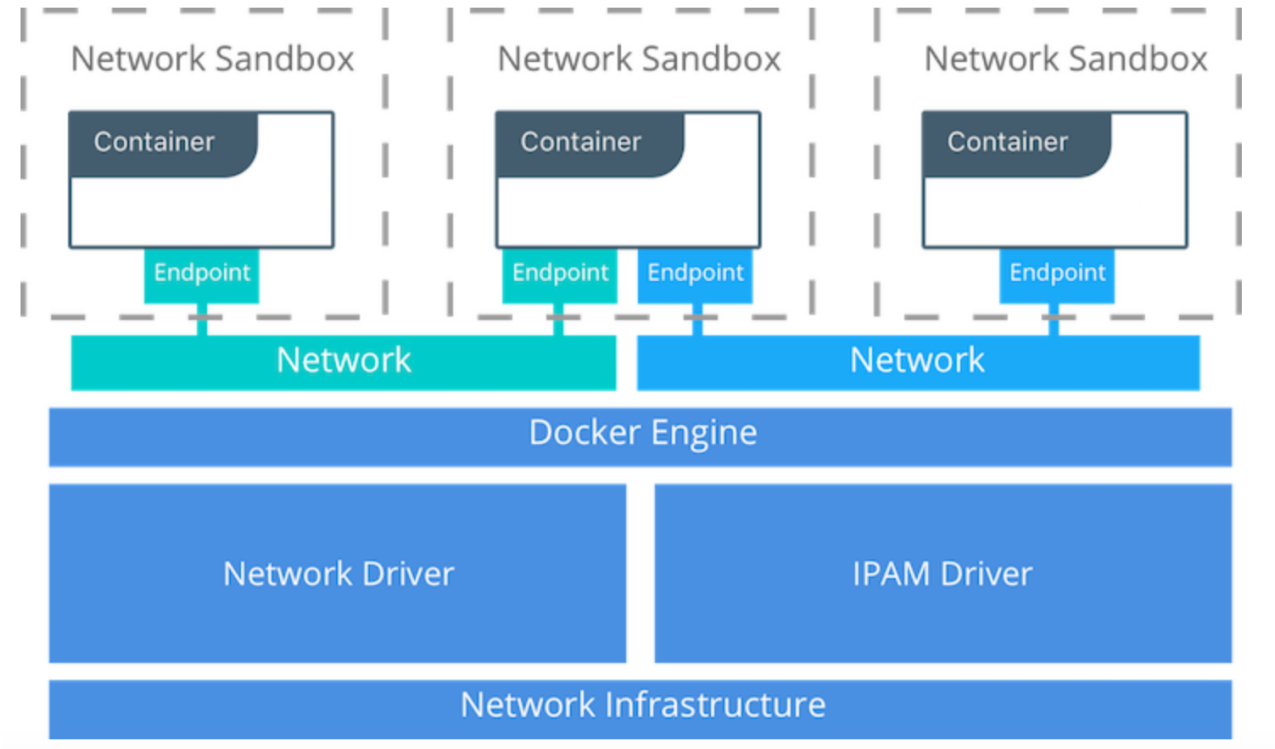
However, the scope is still local (host machine) – see last column.

The real meat will be the overlay.

```
root@ubuntu-xenial:~# docker network ls
NETWORK ID NAME DRIVER SCOPE
bfb14981a5df bridge bridge local
b7c327787044 host host local
0b396f0fc926 mynetwork bridge local
492f4a9fe233 none null local
```

# Docker Networking Model

# The Container Networking Model

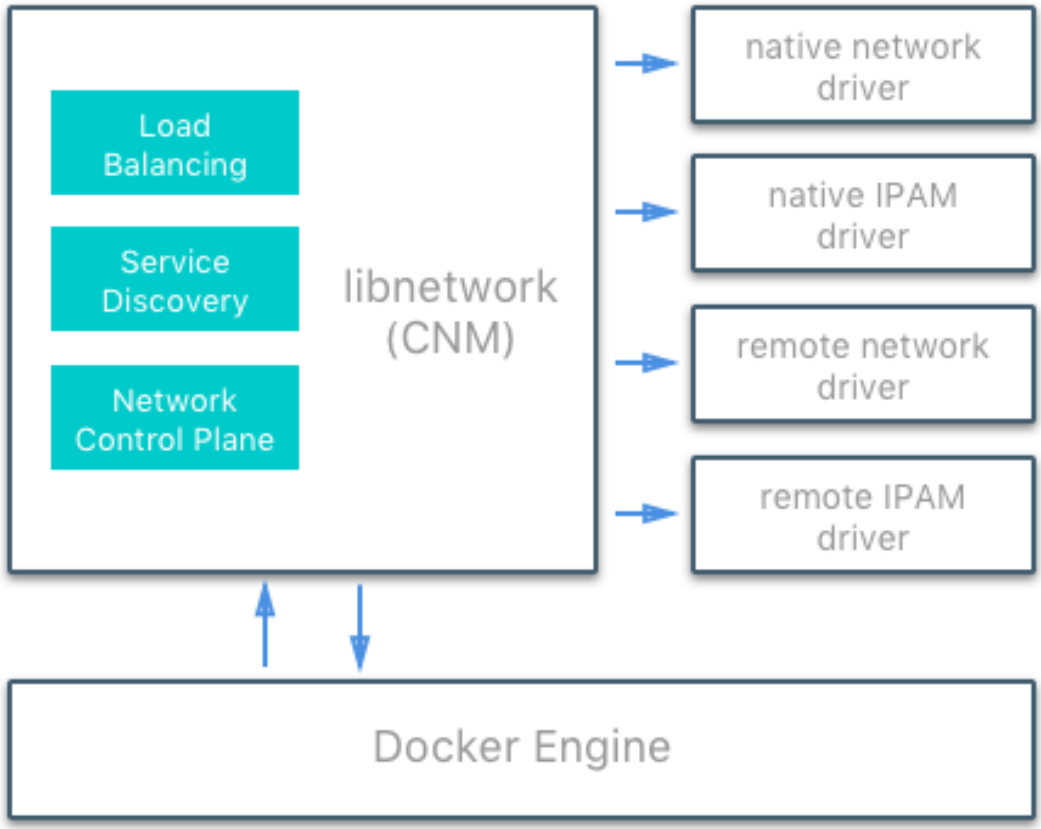



source: `https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks`

# The Container Networking Model

- "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."
- Endpoint: enable connection to the outside world, from a simple bridge to a complex overlay network
- Network driver: possibility to use Docker solution (swarm) or third party
- IPAM : IP address management - DHCP and the like

# An open Network driver Model




source: https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks

# Docker Native Network Drivers

| Driver | Description |
|---|---|
| Host | With the `host` driver, a container uses the networking stack of the host. There is no namespace separation, and all interfaces on the host can be used directly by the container |
| Bridge | The `bridge` driver creates a Linux bridge on the host that is managed by Docker. By default containers on a bridge can communicate with each other. External access to containers can also be configured through the `bridge` driver |
| Overlay | The `overlay` driver creates an overlay network that supports multi-host networks out of the box. It uses a combination of local Linux bridges and VXLAN to overlay container-to-container communications over physical network infrastructure |
| MACVLAN | The `macvlan` driver uses the MACVLAN bridge mode to establish a connection between container interfaces and a parent host interface (or sub-interfaces). It can be used to provide IP addresses to containers that are routable on the physical network. Additionally VLANs can be trunked to the `macvlan` driver to enforce Layer 2 container segmentation |
| None | The `none` driver gives a container its own networking stack and network namespace but does not configure interfaces inside the container. Without additional configuration, the container is completely isolated from the host networking stack |

source: `https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks`

# Remote Network driver

| Driver | Description |
|---|---|
| contiv | An open source network plugin led by Cisco Systems to provide infrastructure and security policies for multi-tenant microservices deployments. Contiv also provides integration for non-container workloads and with physical networks, such as ACI. Contiv implements remote network and IPAM drivers. |
| weave | A network plugin that creates a virtual network that connects Docker containers across multiple hosts or clouds. Weave provides automatic discovery of applications, can operate on partially connected networks, does not require an external cluster store, and is operations friendly. |
| calico | An open source solution for virtual networking in cloud datacenters. It targets datacenters where most of the workloads (VMs, containers, or bare metal servers) only require IP connectivity. Calico provides this connectivity using standard IP routing. Isolation between workloads — whether according to tenant ownership or any finer grained policy — is achieved via iptables programming on the servers hosting the source and destination workloads. |
| kuryr | A network plugin developed as part of the OpenStack Kuryr project. It implements the Docker networking (libnetwork) remote driver API by utilizing Neutron, the OpenStack networking service. Kuryr includes an IPAM driver as well. |

source: `https://success.docker.com/Architecture/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docker_Container_Networks`

# Docker Swarm 101

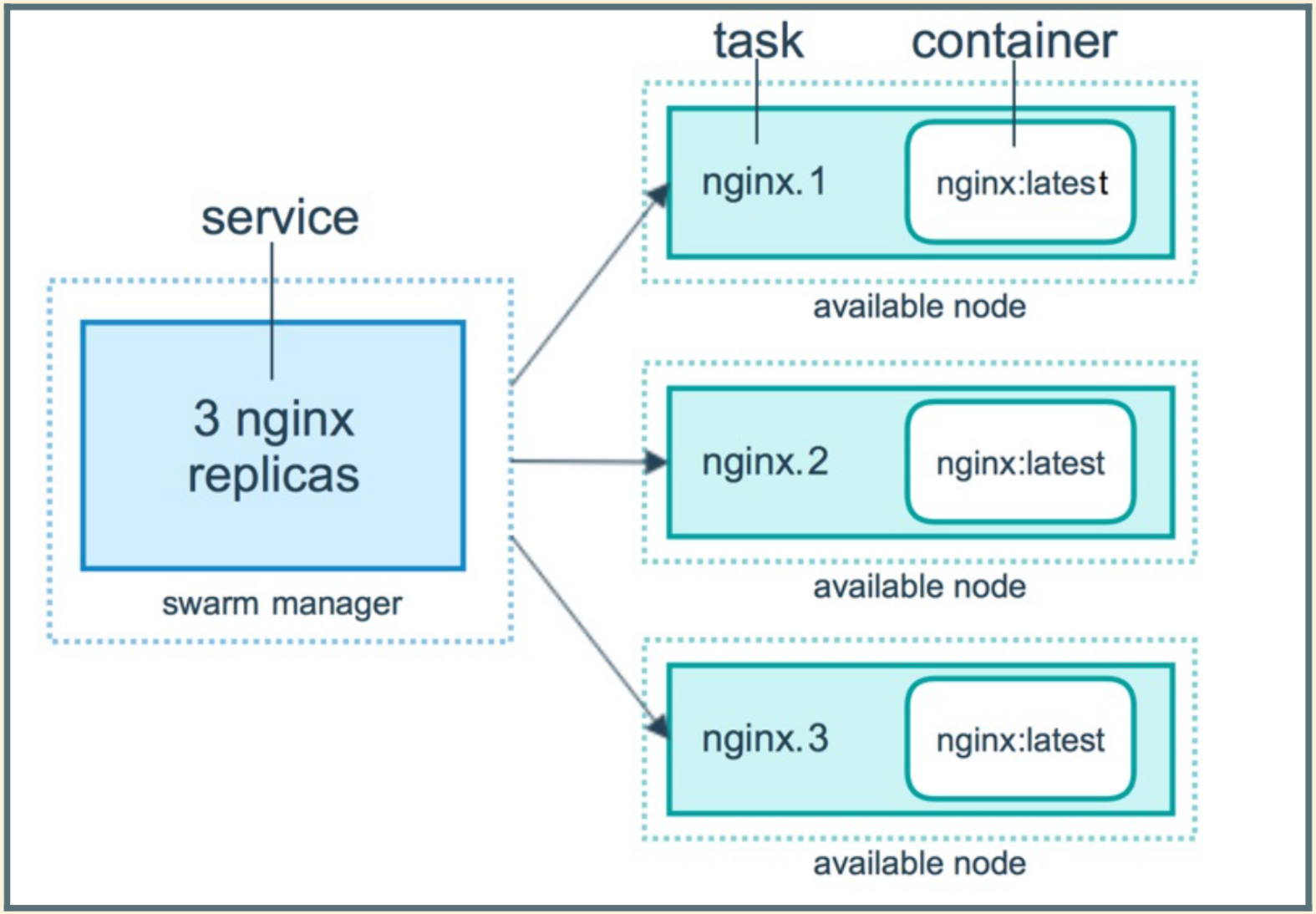
# Docker swarm

Several Docker Hosts $\Rightarrow$ Use them in Cluster

Docker Engine 1.12:

- natively supports swarm
- Clusters organized into workers, managers and leaders
- Dispatching of services : tasks to be executed by servers

# Swarm tasks dispatching



# Swarm operations

```
$ docker swarm init --advertise-addr <MANAGER-IP>

# Swarm initialized: current node (8jud...) is now a manager. To add a worker to this swarm
      , run the following command:

docker swarm join --token SWMTKN-1-59fl4ak4nqjmao1ofttrc4eprhrola2l87... \
      172.31.4.182:2377
```

## Check state:

```
$ docker info Swarm: active
NodeID: 8jud7o8dax3zxbags3f8yox4b
Is Manager: true
ClusterID: 2vcw2oa9rjps3a24m91xhvv0c
```

## You have created a first node in the swarm (your host)

```
$ docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS
8jud...ox4b * ip-172-31-4-182 Ready Active Leader
```

# Docker swarm

Docker has generated tokens to join the swarm:

```
$ docker swarm join−token worker
$ docker swarm join−token manager
```

You can then join by issuing on the second host:

```
$ docker swarm join −−token TOKEN−WORKER... 172.31.4.182:2377
```

If this works, you should have

```
$ docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS
8jud...ox4b ∗ ip−172−31−4−182 Ready Active Leader
ehb0...4fvx ip−172−31−4−180 Ready Active
```

# Docker swarm

You can now execute a service :

```
root@ubuntu-xenial: docker service create --replicas 1 --name helloworld alpine ping
    docker.com
```

and observe the services in general or a specific service

```
root@ubuntu-xenial: docker service create --replicas 1 --name helloworld alpine ping docker.com
2klpz2bef3ez7w498hw17bwbw

root@ubuntu-xenial: docker service ls
ID NAME MODE REPLICAS IMAGE PORTS
2klpz2bef3ez helloworld replicated 1/1 alpine:latest
root@ubuntu-xenial: docker service ps helloworld
ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR PORTS
5uwod1wobk0m helloworld.1 alpine:latest ubuntu-xenial Running Running 35 seconds ago
```

# Docker Network Overlay

# Docker Overlay

- Enables multi-host networking
  - A host here is a physical or virtual machine that features the docker daemon
  - Docker hosts be created independently or from a central place using docker-machine
- Docker overlay driver enables to create a VLAN for groups of distributed (over the Docker hosts) containers

# Docker Machine


- Create a VM with Docker engine that can be remotely controlled. This VM can be local (Virtualbox or Hyper-V) or distant in the cloud (Amazon Web Service, Digital Ocean).
- For cloud deployment, docker-machine superseded by docker Cloud

# Docker Machine with local provisioning using Virtualbox

## Creating VM

```
$ docker−machine create −−driver virtualbox default
  Creating machine...
  (staging) Copying /Users/ripley/.docker/machine/cache/boot2docker.iso to /Users/ripley/.docker/machine/machines/
            default/boot2docker.iso...
  (staging) Creating VirtualBox VM...
  (staging) Creating SSH key...
  (staging) Starting the VM...
  Provisioning with boot2docker...
  Copying certs to the remote machine...
  Setting Docker configuration on the remote daemon...
  Checking connection to Docker...
  Docker is up and running!
```

# Docker Machine with local provisioning using Virtualbox

## Listing current docker machines

```
$ docker-machine ls
 NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERRORS
 default * virtualbox Running tcp://192.168.99.187:2376 v1.9.1
```

## Listing and Changing env variables to control a given docker machine:

```
$ docker-machine env default
 export DOCKER_TLS_VERIFY="1"
 export DOCKER_HOST="tcp://172.16.62.130:2376"
 export DOCKER_CERT_PATH="/Users/<yourusername>/.docker/machine/machines/default"
 export DOCKER_MACHINE_NAME="default"
 # Run this command to configure your shell:
 # eval "$(docker-machine env default)"
```

```
$ eval "$(docker-machine env default)"
```

## New docker host ready to be integrated in swarm!

# Docker Network Overlay

## Create an overlay

```
$ docker network create --driver overlay my-network
```

## Inspect network

```
$ docker network inspect my-network
[
    {
        "Name": "my-network",
        "Id": "fsf1dmx3i9q75an49z36jycxd",
        "Created": "0001-01-01T00:00:00Z",
        "Scope": "swarm",
        "Driver": "overlay",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": []
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "Containers": null,
        "Options": {
            "com.docker.network.driver.overlay.vxlanid_list": "4097"
        },
        "Labels": null
    }
]
```

# Docker Network Overlay

What is important in previous listing:

- The driver : overlay!
- The scope : swarm $\Rightarrow$ network extends to a swarm, not local to host
- Attached containers are listed in the docker inspect

You can now attach a service (set of containers) to the overlay

```
$ docker service create replicas 3 −−name my−web −−network my−network nginx
```

# Docker Ovelay Network: What is behind the hood?

## IETF RFC 7348

Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks

- Tunnelling of Ethernet frames within UDP packets

# VXLAN

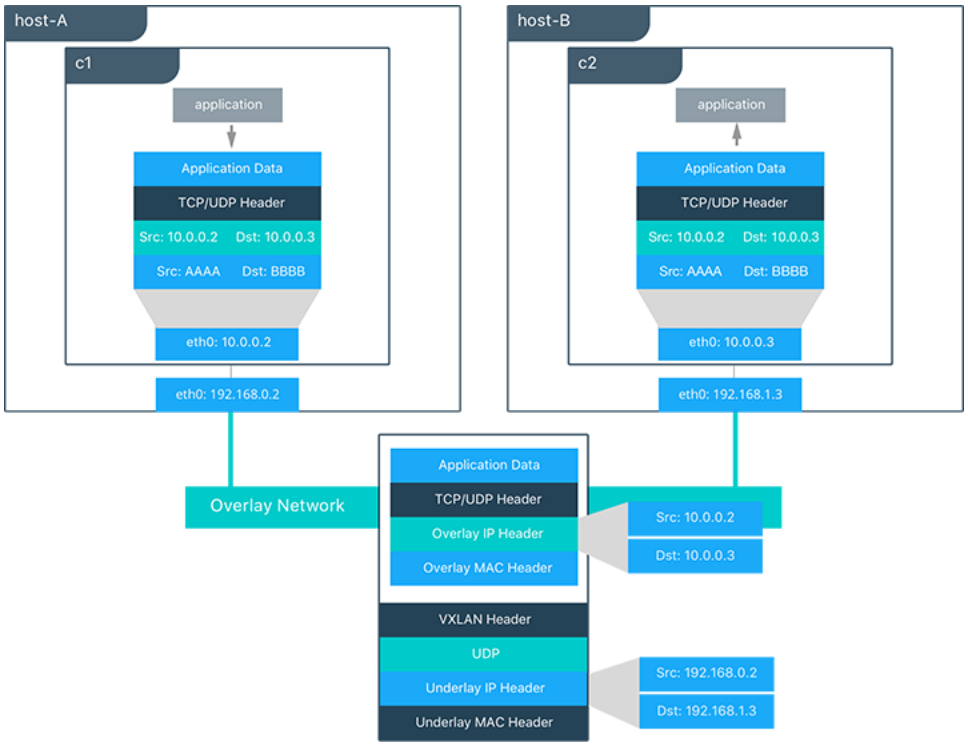



Figure: source: docker web site

"c1 does a DNS lookup for c2. Since both containers are on the same overlay network the Docker Engine local DNS server resolves c2 to its overlay IP address 10.0.0.3." $\Rightarrow$ need to discuss additional services like DNS, load balancing

# Docker overlay - what is added in a given host (and in each container)

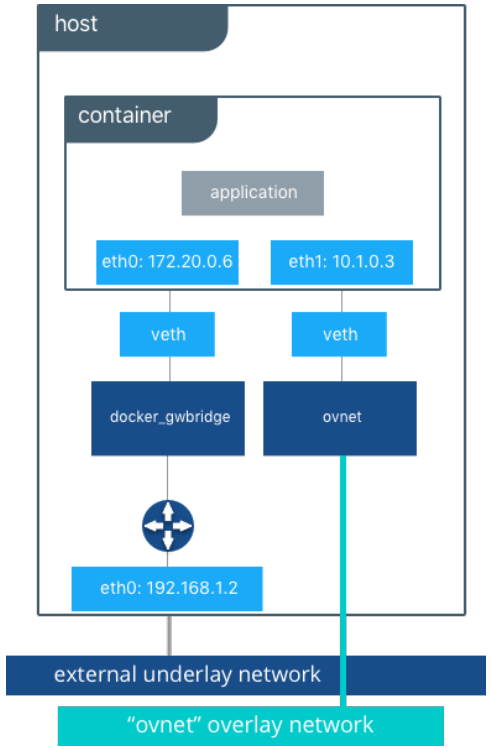



Figure: source: docker web site

# Docker overlay - what is added in a given host (and in each container)

- Docker_gwbridge: the egress bridge for traffic that goes outside (underlay means 'not any VXLAN interface')
- ovnet: one bridge per overlay.
  - Created on each docker host with containers in this overlay, ie. no a priori creation on all swarm nodes (just where needed)
    - Called the egress bridge
    - One per host
    - Constitutes the so-called VXLAN Tunnel End Point (VETP).
    - VETPs communicate with each other to maintain the overlay
  - Uses a Linux VXLAN port to attach to the outside.

# VXLAN

- VNI–VXLAN Network Identifier
  - 24 bit number (16M+ unique identifiers)
  - Part of the VXLAN Header
  - Similar to VLAN ID
  - Limits broadcast domain
  -
- VTEP–VXLAN Tunnel End Point
  - Originator and/or terminator of VXLAN tunnel for a specific VNI
  - Outer DIP/Outer SIP

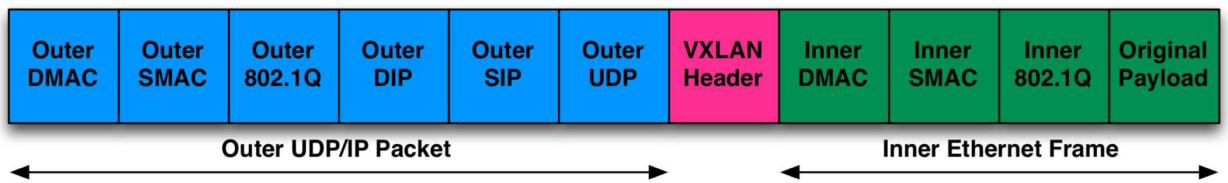



Figure: source: Nolan Leake (cumulus) and Chet Burgess (Metacloud)

# VXLAN

## Sending a packet

- ARP table is checked for IP/MAC/Interface mapping
- L2 FDB is checked to determine IP of destination on VTEP for destination on MAC on source VTEP
- Source VETP then encapsulates frame with correct destination VETP and destination decapsulates....

```
root@mcpl.dev4.mc:~$ ip addr show dev vxlan12345
140: vxlan12345: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
 group default
    link/ether 4e:3c:82:2e:09:0e brd ff:ff:ff:ff:ff:ff
    inet 10.9.8.1/24 scope global vxlan12345
       valid_lft forever preferred_lft forever
root@mcpl.dev4.mc:~$ arp -an | grep 10.9.8.2
? (10.9.8.2) at 7a:79:96:de:2c:a2 [ether] on vxlan12345
root@mcpl.dev4.mc:~$ bridge fdb show dev vxlan12345
7a:79:96:de:2c:a2 dst 172.16.3.102 self
4e:3c:82:2e:09:0e dst 127.0.0.1 self
```

Figure: source: Nolan Leake (cumulus) and Chet Burgess (Metacloud)

# VXLAN

- Need a mechanism to lookup behind which VETP is a container MAC.
- Option in VXLAN
  - IP Multicast group per overlay (per VXLAN)
  - Unicast also possible

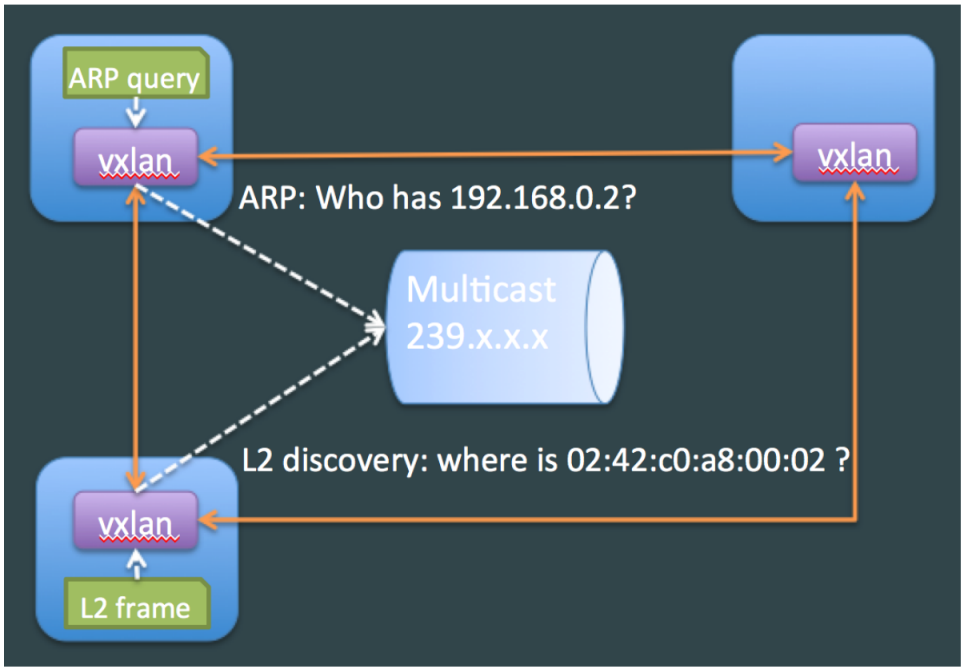



Figure: source: Nolan Leake (cumulus) and Chet Burgess (Metacloud)

# VXLAN support in Linux

- Well supported in most modern Linux Distros – Linux Kernel 3.10+
  - Linux uses UDP port 8472 instead of IANA issued 4789 – iproute2 3.7+
  - Configured using ip link command

```
root@mcpl.dev4.mc:~$ ip link add vxlanl2345 type vxlan id 12345 remote 172.16.3.100
root@mcpl.dev4.mc:~$ ip link set dev vxlanl2345 up
root@mcpl.dev4.mc:~$ ip -d link show dev vxlanl2345
140: vxlanl2345: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN
 mode DEFAULT group default
    link/ether 4e:3c:82:2e:09:0e brd ff:ff:ff:ff:ff:ff promiscuity 0
    vxlan id 12345 remote 172.16.3.100 port 32768 61000 ageing 300
```

Figure: source: Nolan Leake (cumulus) and Chet Burgess (Metacloud)

# VLXAN - The Docker way

- Key advantage: Docker knows where MAC addresses appear in the overlay when it creates the containers
- Propagate to each VETP the MAC/VETP mapping

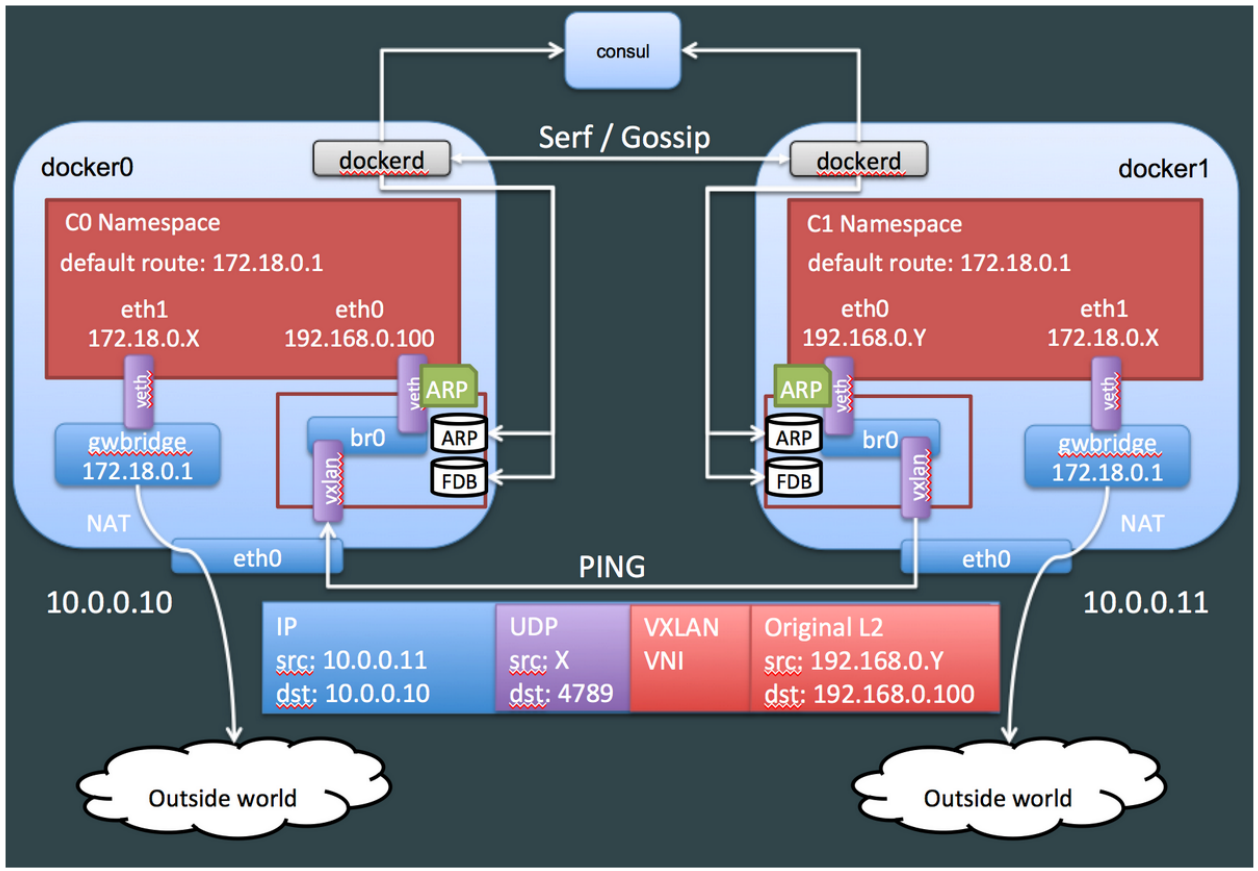



Figure: source: Laurent Bernaille

# Docker Network Control Plane Model

Relies on a gossip protocol (SWIM) to propagate network state information and topology across Docker container clusters. A complex task at large scale when one must reach a consensus. Part of Swam.

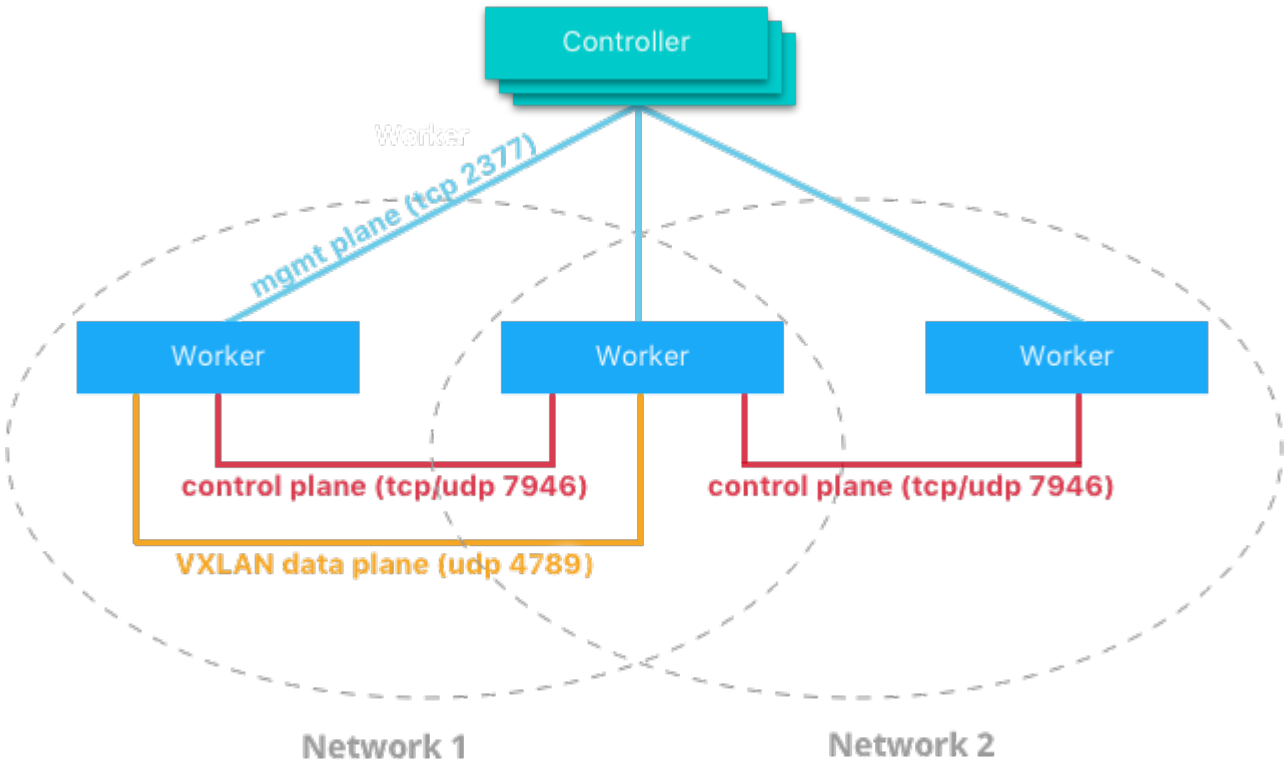



Figure: source: docker web site

# Docker networking: What is still missing?

- Port publishing at swarm level and load balancing
- DNS support

# Exposing ports

- Ports are exposed on
- Two modes:
  - Host mode: only exposes the IP of hosts with a replica
  - Ingress mode (default mode) : any host even without replica will answer.

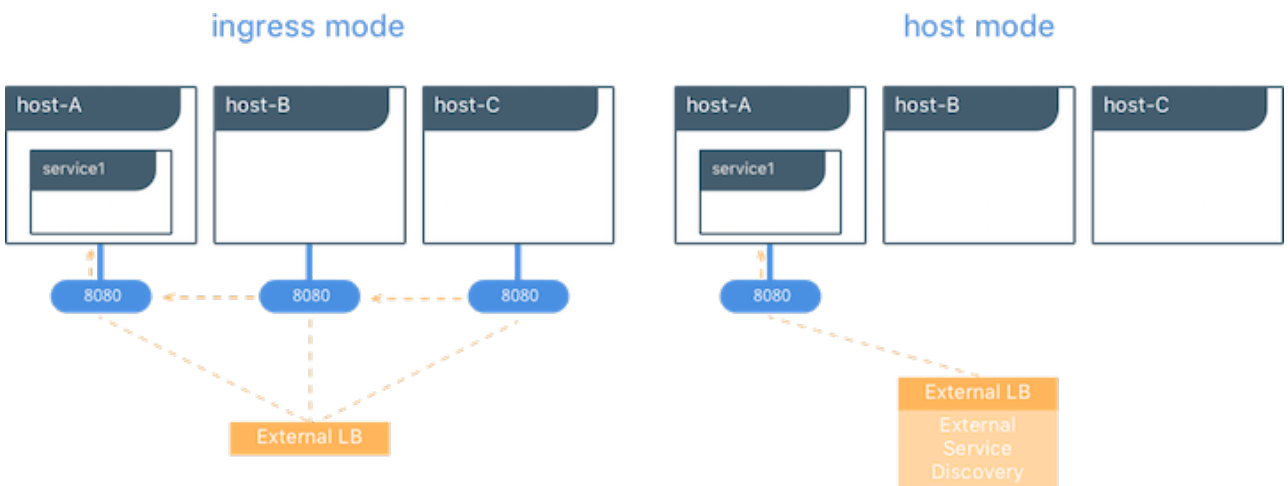



Figure: source: docker web site

# Exposing port: Ingress mode

- Ingress mode relies on Swarm Routing mesh protocol (control plane TCP port on docker network control plane slide)
- In general, usage of an external (to docker) load balancer like HAproxy

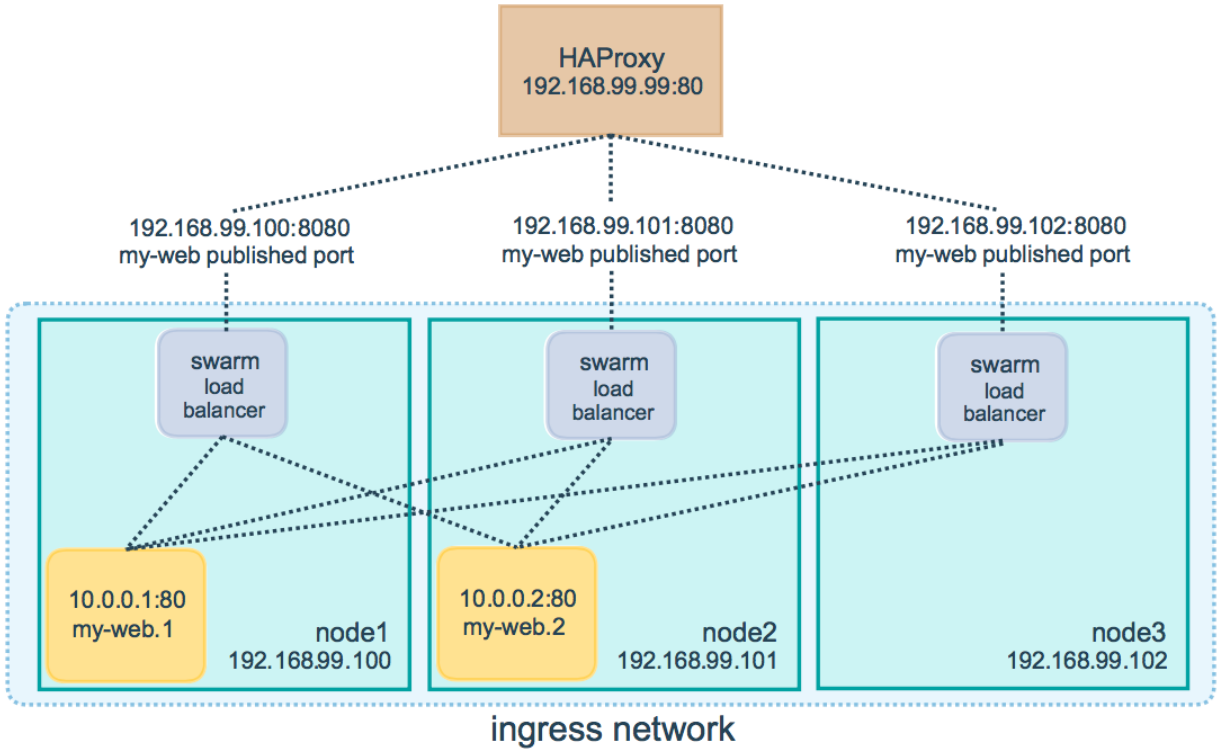



Figure: source: docker web site

# Exposing port: Ingress mode

Relies on ipvs (load balacing at layer 4) and iptables (firewall/NAT) of Linux kernel

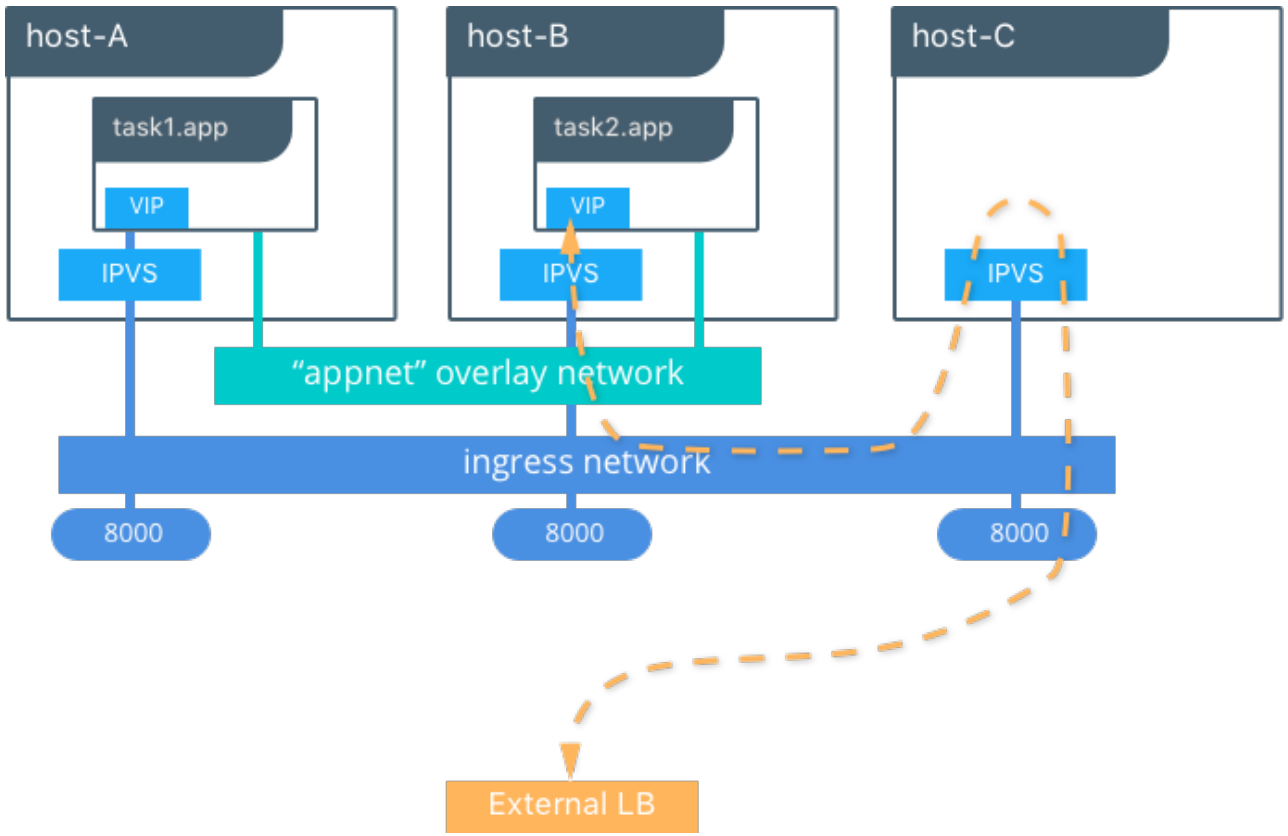



Figure: source: docker web site

# Service discovery

Step 1: DNS (from docker web site)

- Each Docker container ( or task in Swarm mode) has a DNS resolver that forwards DNS queries to Docker Engine, which acts as a DNS server.
- Service discovery is network-scoped (per overlay)
- If the destination container or service does not belong on the same network(s) as the source container, then Docker Engine forwards the DNS query to the configured default DNS server.

# Service discovery

- Docker assigns a virtual IP for each service at creation

```
# Create myservice with 2 replicas as part of that network
$ docker service create --network mynet --name myservice --replicas 2 busybox
        ping localhost
8t5r8cr0f0h6k2c3k7ih4l6f5

# See the VIP that was created for that service
$ docker service inspect myservice
...

"VirtualIPs": [
                    {
                        "NetworkID": "a59umzkdj2r0ua7x8jxd84dhr",
                        "Addr": "10.0.0.3/24"
                    },
]
```

# Service discovery

- Load balancing on VIP on the different replicas addresses.

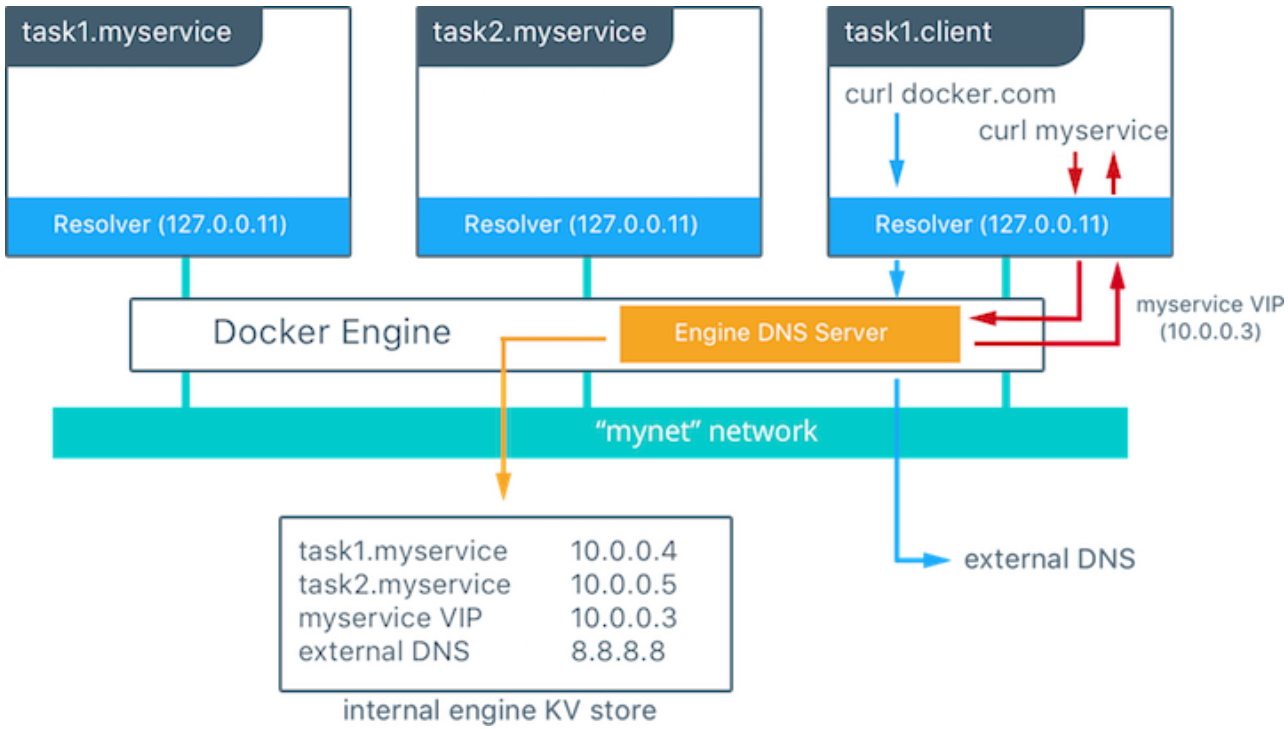



Figure: source: docker web site