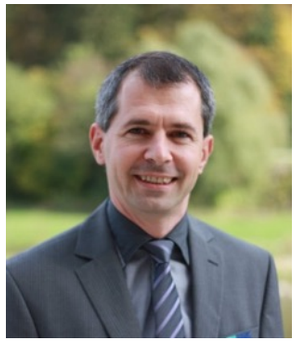
Docker Networking Deep Dive

Ivan Pepelnjak (ip@ipSpace.net)
Network Architect

ipSpace.net AG

Who is Ivan Pepelnjak (@ioshints)



Past

- Kernel programmer, network OS and web developer
- Sysadmin, database admin, network engineer, CCIE
- Trainer, course developer, curriculum architect
- Team lead, CTO, business owner

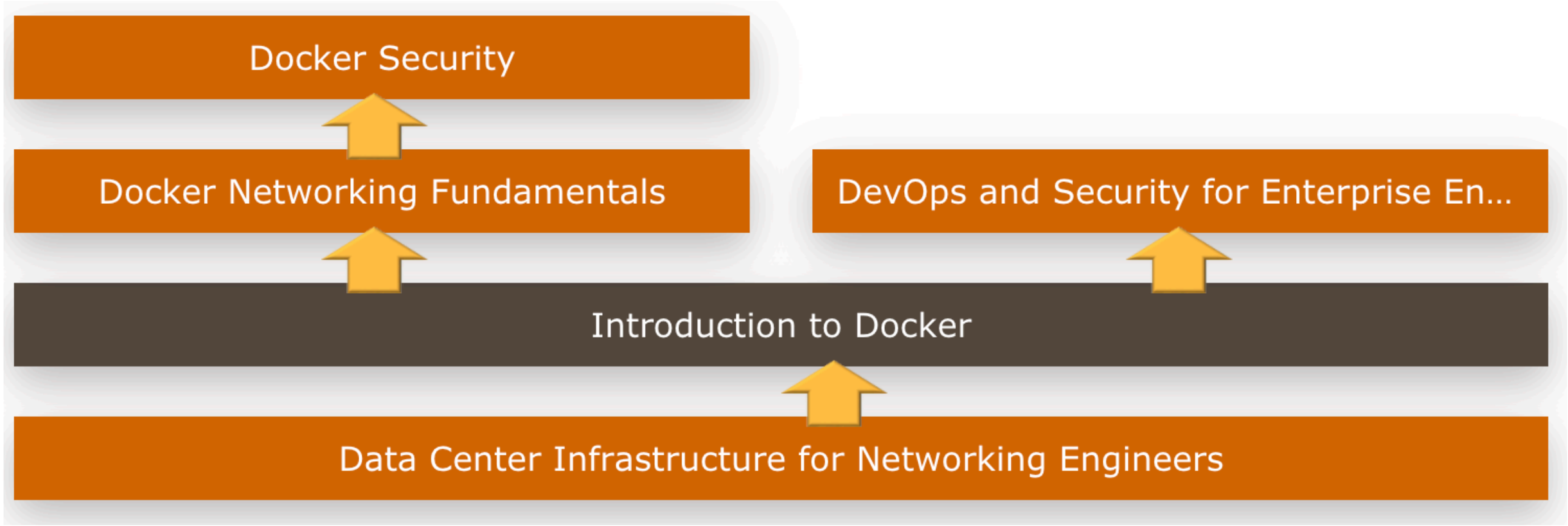
Present

- Network architect, consultant, blogger, webinar and book author

Focus

- SDN and network automation
- Large-scale data centers, clouds and network virtualization
- Scalable application design
- Core IP routing/MPLS, IPv6, VPN

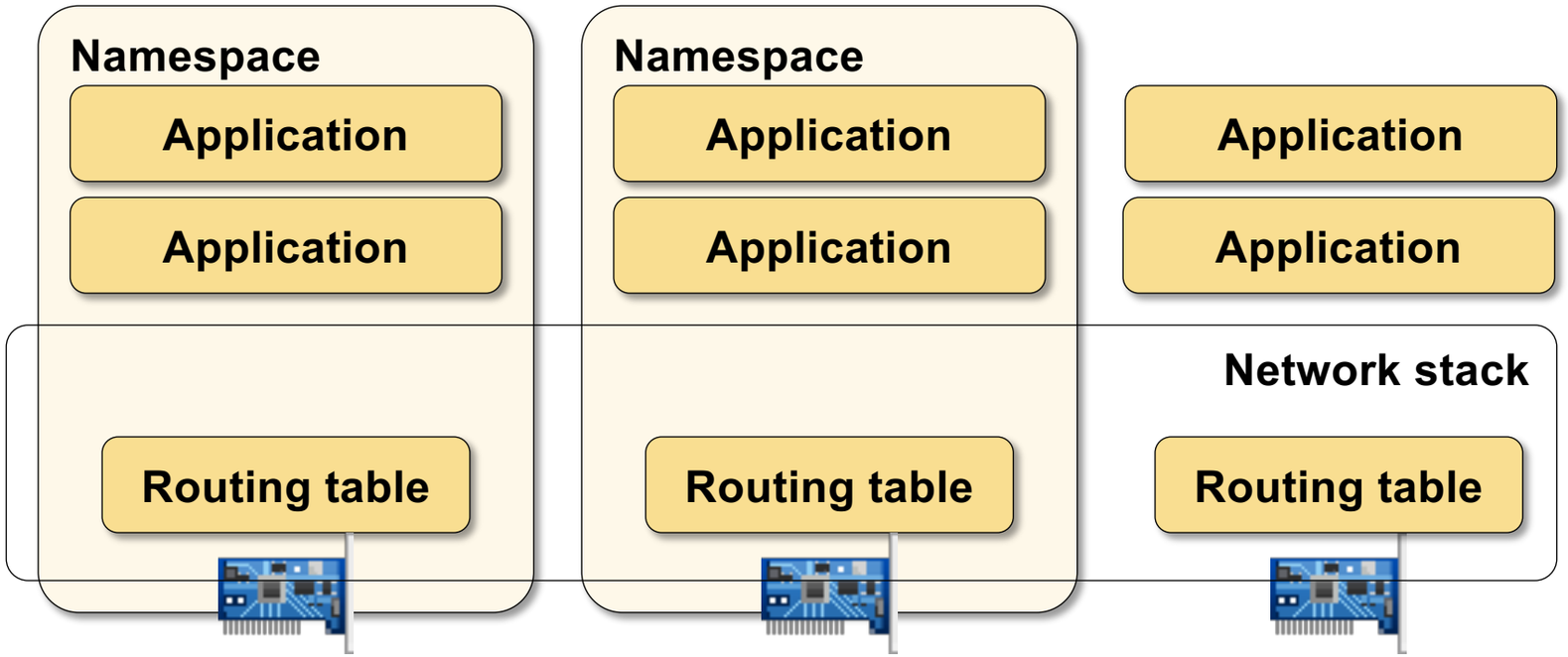
Before We Start: Assumptions



Docker Networking Overview

ipSpace

Recap: Network Namespaces



- Each Docker container is running in a separate network namespace
- Each network namespace has its own routing table and set of interfaces
- Each network namespace might need its own routing daemon (namespace != VRF)

Namespace Access to Physical Network

Each NIC (physical or virtual) is attached to a single namespace

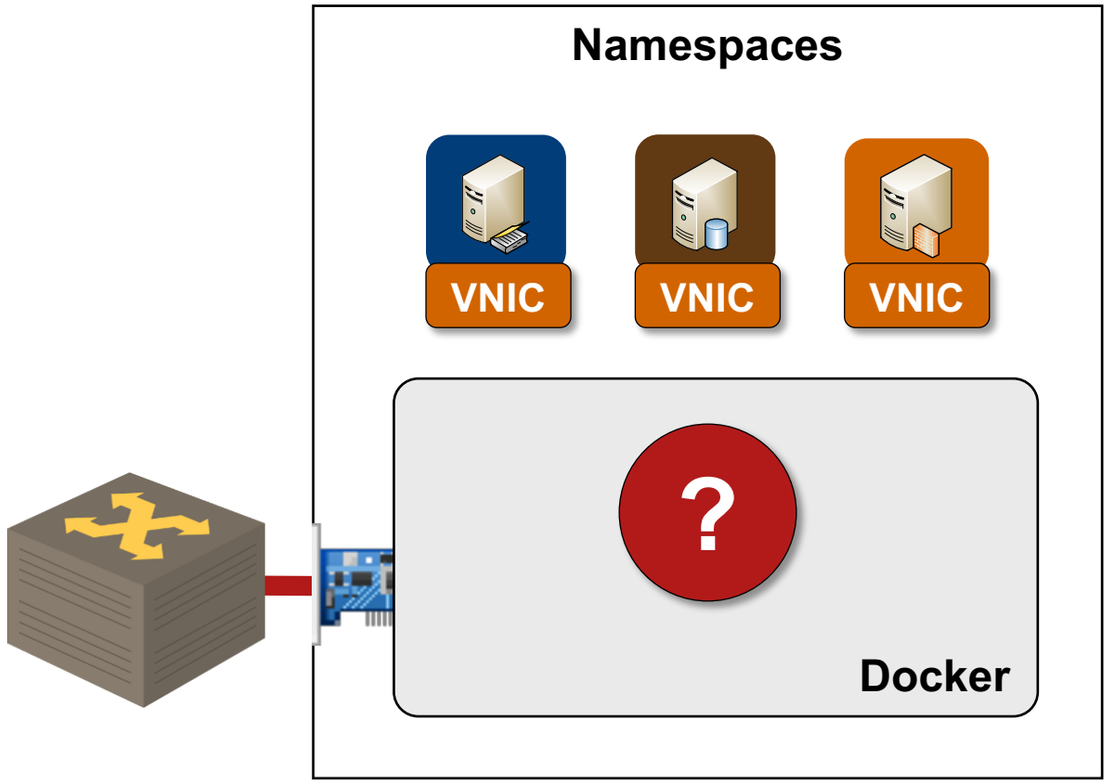
- How can containers (running in network namespaces) access external network?
- How can external clients reach container services?

High-level overview

- Create a virtual NICs
- Attach the virtual NIC to container namespace
- Make sure vNIC parameters cannot be changed within the namespace (Linux capabilities)
- Somehow connect the virtual NICs to physical network

Implementations

- Virtual NICs based on physical NICs
- vEth pairs



Virtual NICs Based on Physical NICs

Linux Virtual Interfaces (VIFs)

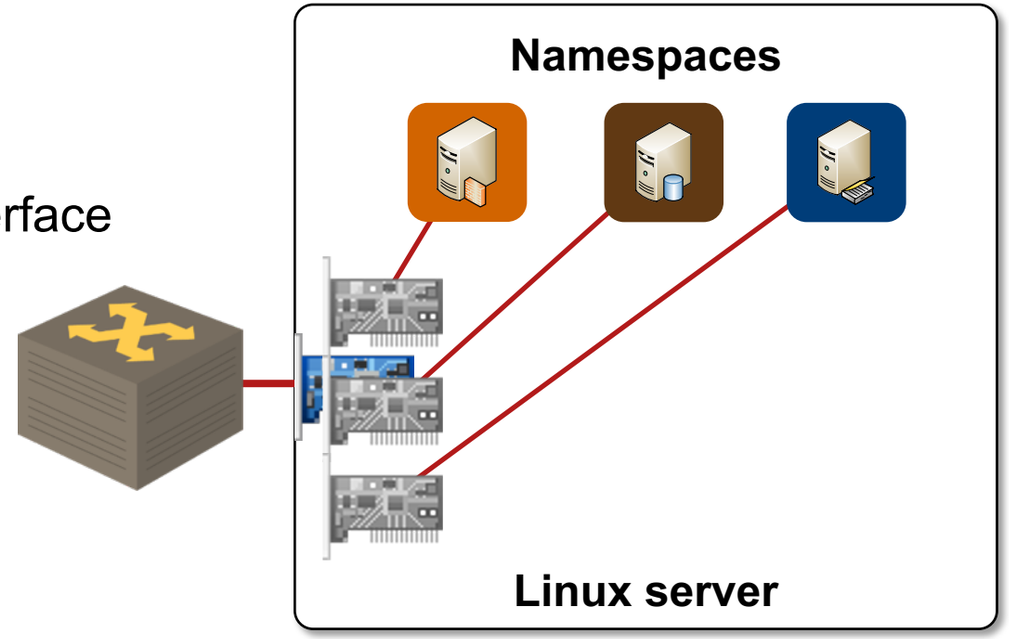
- Vlan: virtual interface for a VLAN configured on physical interface (or Linux bridge)
- Macvlan: virtual interface for a MAC address
- Ipvlan: virtual interface tied to a specific (secondary) IP addresses

Stacking virtual interfaces

- You can attach some virtual interfaces to parent interfaces
- Parent interface can be physical interface or another virtual interface (example: macvlan VIF on top of vlan VIF)

Using virtual interfaces for containers

- Parent interface resides in host namespace
- Virtual interface resides in a container

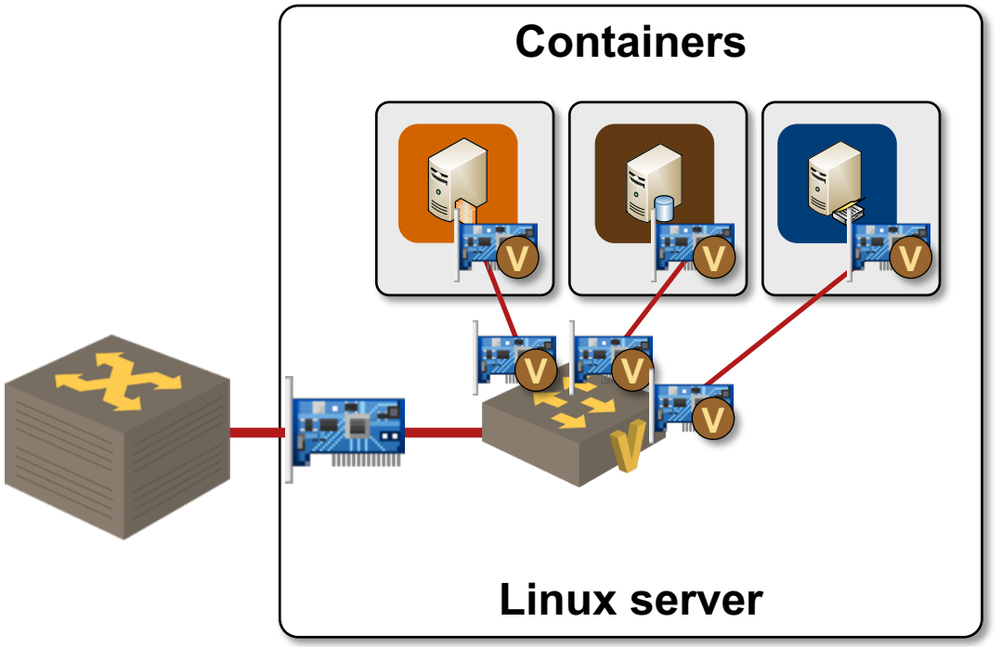


More @ https://developers.redhat.com/blog/2018/10/22/introduction-to-linux-interfaces-for-virtual-networking/

Namespace Implementations Using Virtual Switches

Typical container implementations use vEth pairs

- vEth pair = pair of virtual interfaces connected with a virtual cable
- One vEth interface is in container namespace
- Other vEth interface is in global namespace
- Interface in global namespace is attached to a port on a virtual switch
- Physical (or tunnel) interface is attached as an uplink to the virtual switch



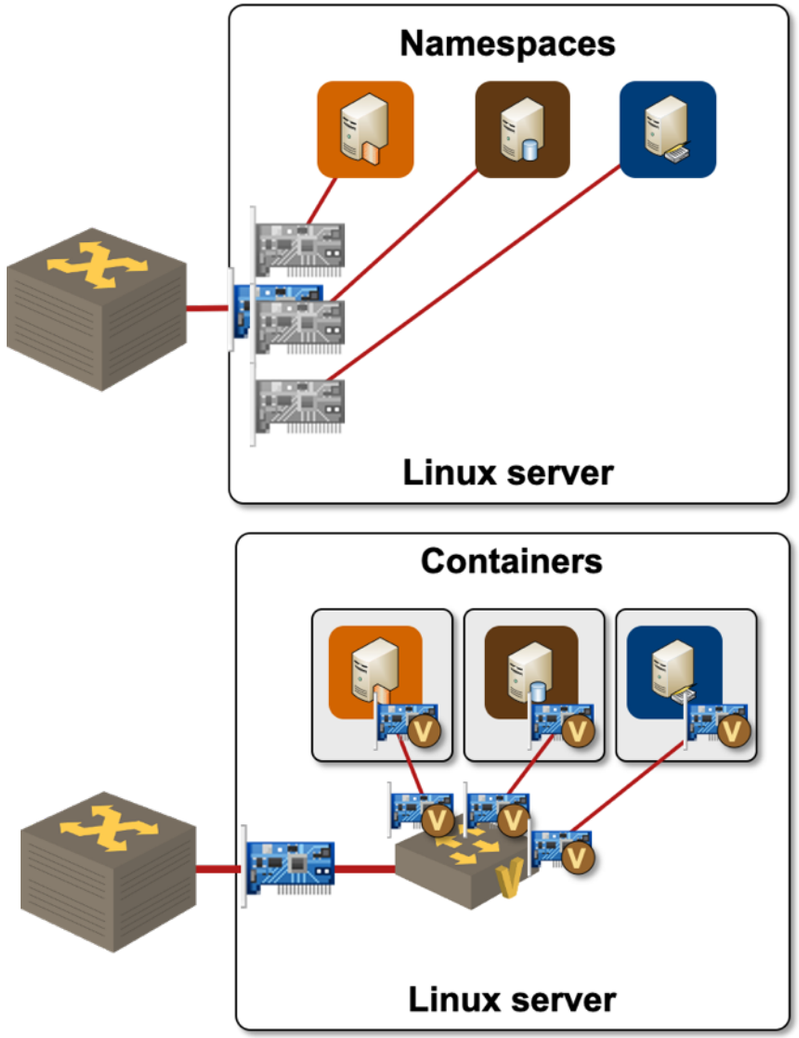
Docker Networking IPv4 Addressing

Inside IPv4 Addresses

- Default Docker bridge uses 172.17.0.0/16 (can be changed)
- You can use any IPv4 prefix you want on internal Docker networks (the recommendation to use RFC 1918 address space is obvious)

Outside IPv4 Addresses

- Macvlan and L2 ipvlan interfaces are attached to physical NIC ➔ each container has an outside MAC+IP address
- L3 ipvlan interfaces are routed ➔ each Docker host needs an IPv4 prefix
- In most other cases all containers use one of the IPv4 addresses of the Docker host (NAT)



Docker Networking IPv6 Addressing

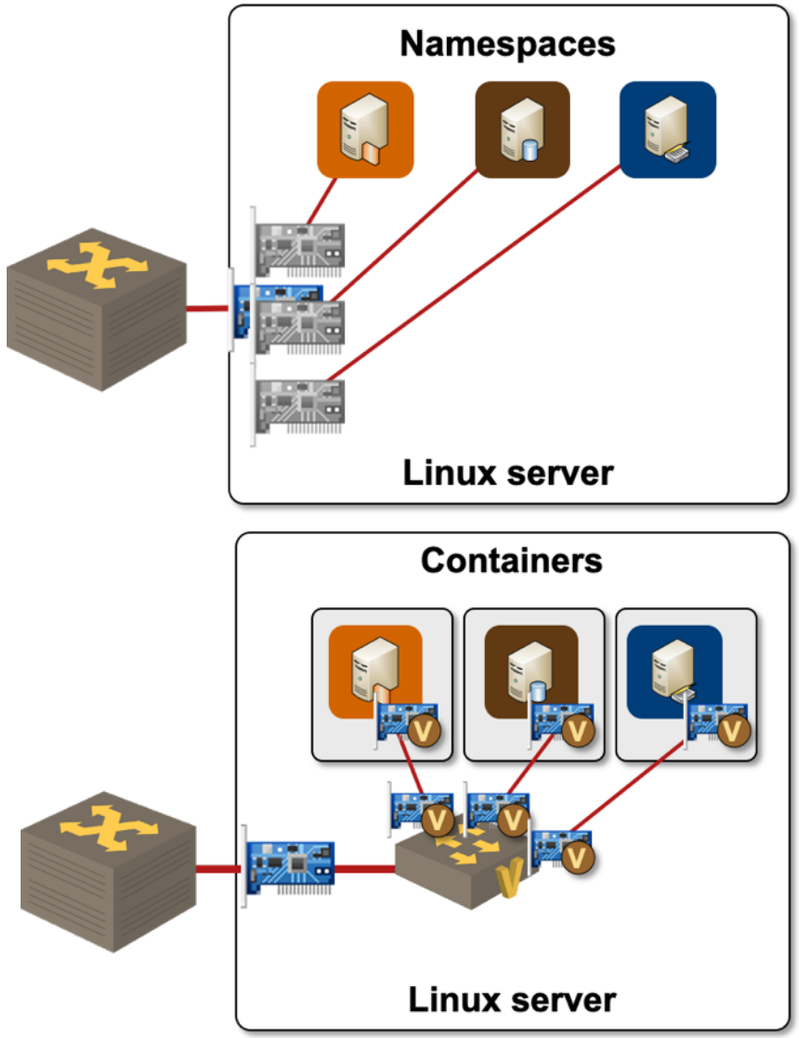
- Docker does NOT use NAT with IPv6
- Each container network needs an IPv6 prefix

Routed solution

- Assign one or more /64 prefixes to every Docker host
- Turn Docker hosts into IPv6 routers (ideally with BGP)
- Advertise IPv6 prefixes with a routing protocol

There's always a hack

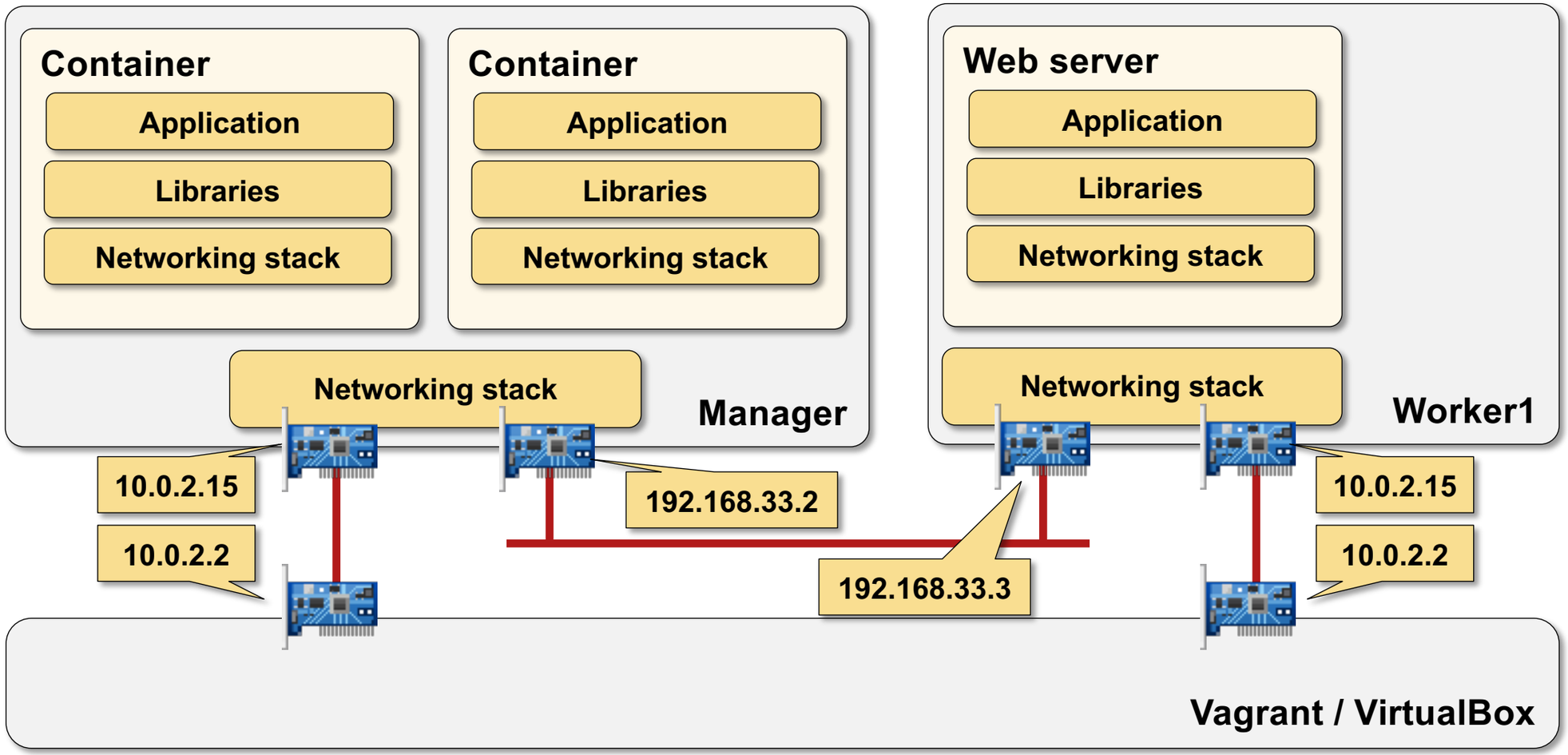
- Assign a part of outside /64 prefix to container network(s)
- Use NDP proxy to reach containers



Old docs @ https://github.com/saturnism/docker/blob/master/docs/userguide/networking/default_network/ipv6.md

Default Docker Networking

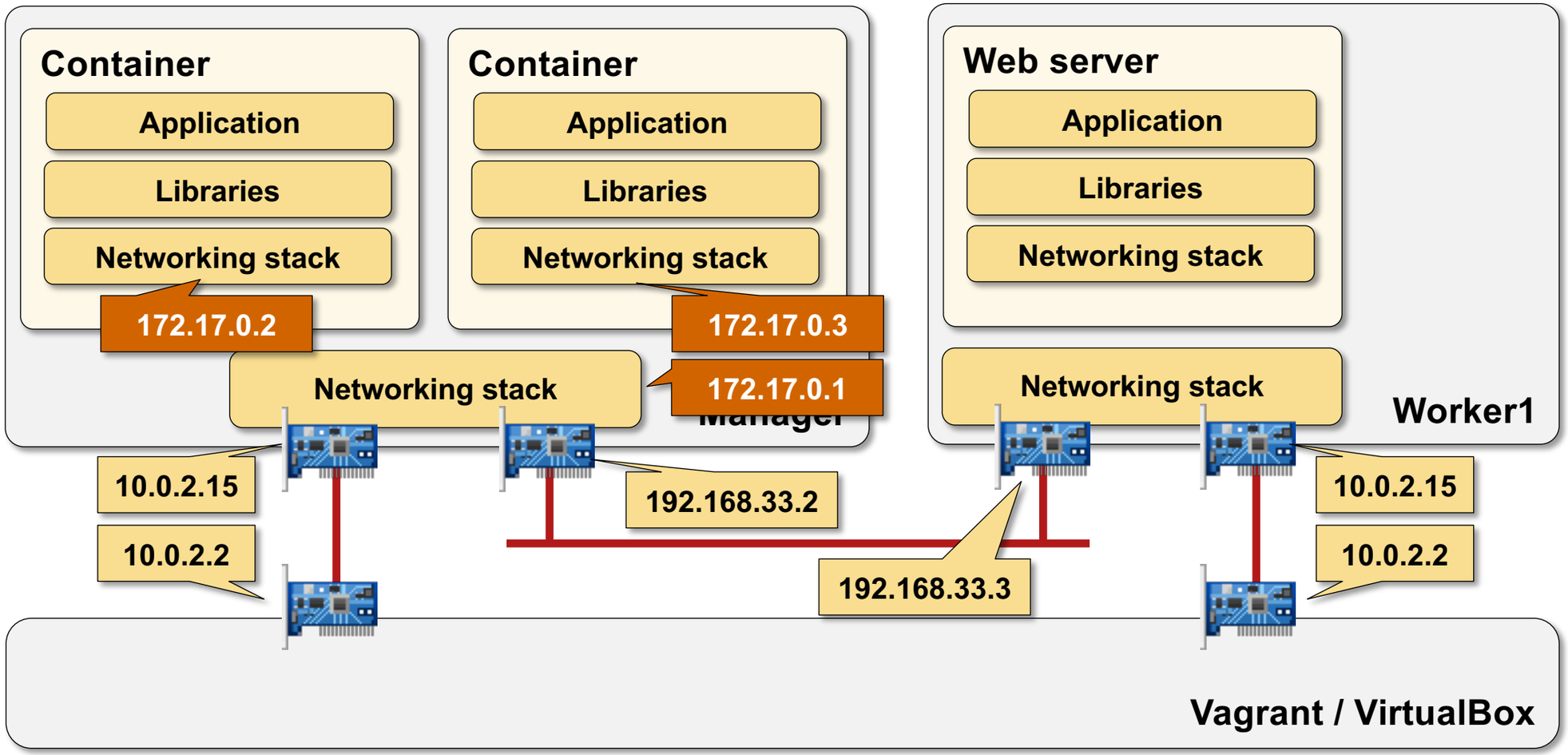
Demo Docker Setup



Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

Demo: Simple Docker Networking

Recap: IP Addressing in Our Demo



Default Docker Linux Bridge

Interfaces on a Docker Host

```
$ docker run --name C1 -itd busybox
```

```
~ $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
de2087131f95        busybox             "sh"                4 hours ago         Up 4 hours                              C1
~ $ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:8b:d5:11 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen 1000
    link/ether 08:00:27:ae:a4:7b brd ff:ff:ff:ff:ff:ff
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:51:8a:f1:38 brd ff:ff:ff:ff:ff:ff
8: veth1c9c1ad@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP mode DEFAULT group default
    link/ether 12:de:dc:93:dd:f6 brd ff:ff:ff:ff:ff:ff link-netnsid 0
~ $
```

```
~ $ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242518af138       no              veth1c9c1ad
~ $
```

Docker Networking Uses Linux Bridges

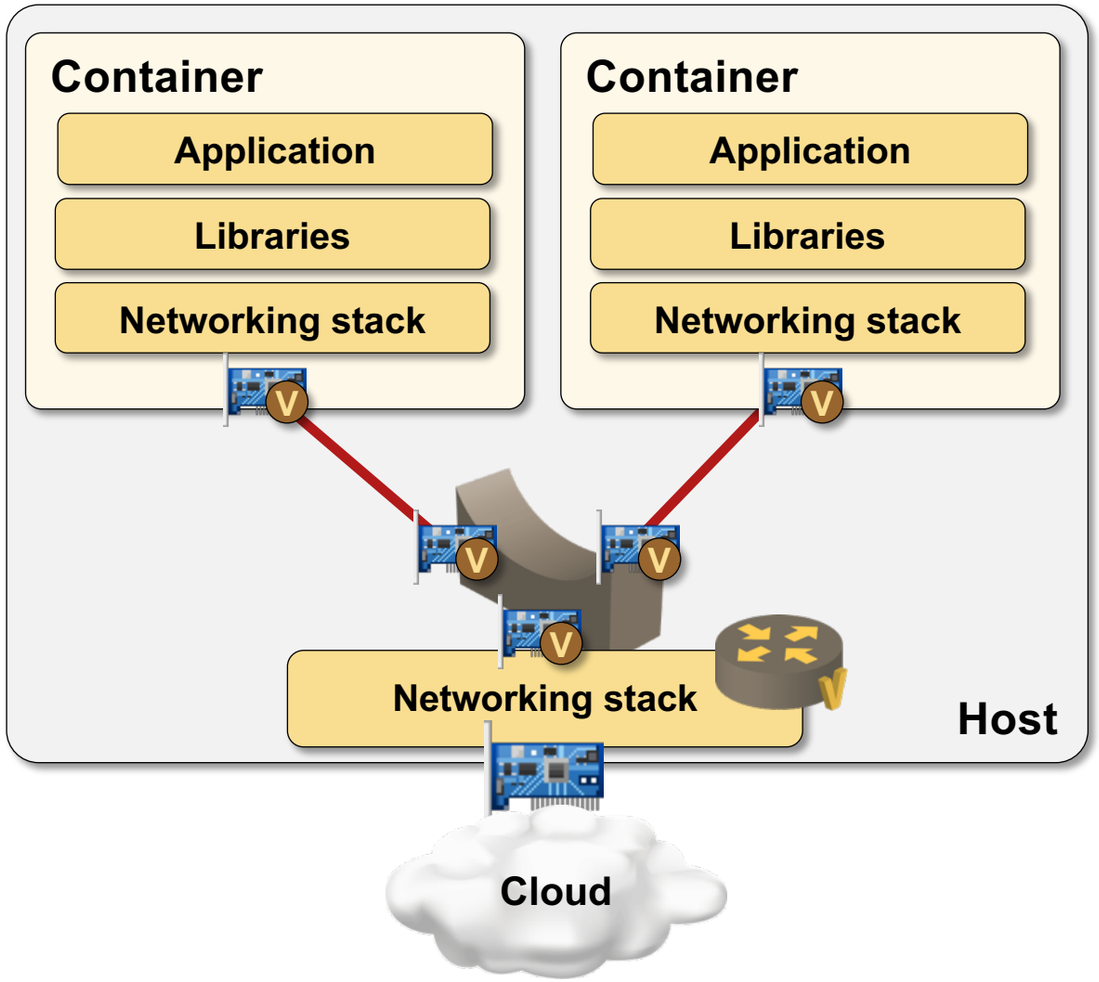
Veth pairs link namespaces

- Container side of the pair is renamed to **eth0**
- Host side of the pair is visible as veth interface

Connecting containers to outside world

- **docker0** is a bridge interface with an IP address
- veth interfaces are connected to **docker0** bridge

```
~ $ ip address show dev docker0
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:6f:f1:fa:4f brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:6fff:fef1:fa4f/64 scope link
       valid_lft forever preferred_lft forever
~ $ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02426ff1fa4f       no              veth2e50e54
                                                        veth5dd3879
```



Physical Network Access

Network access

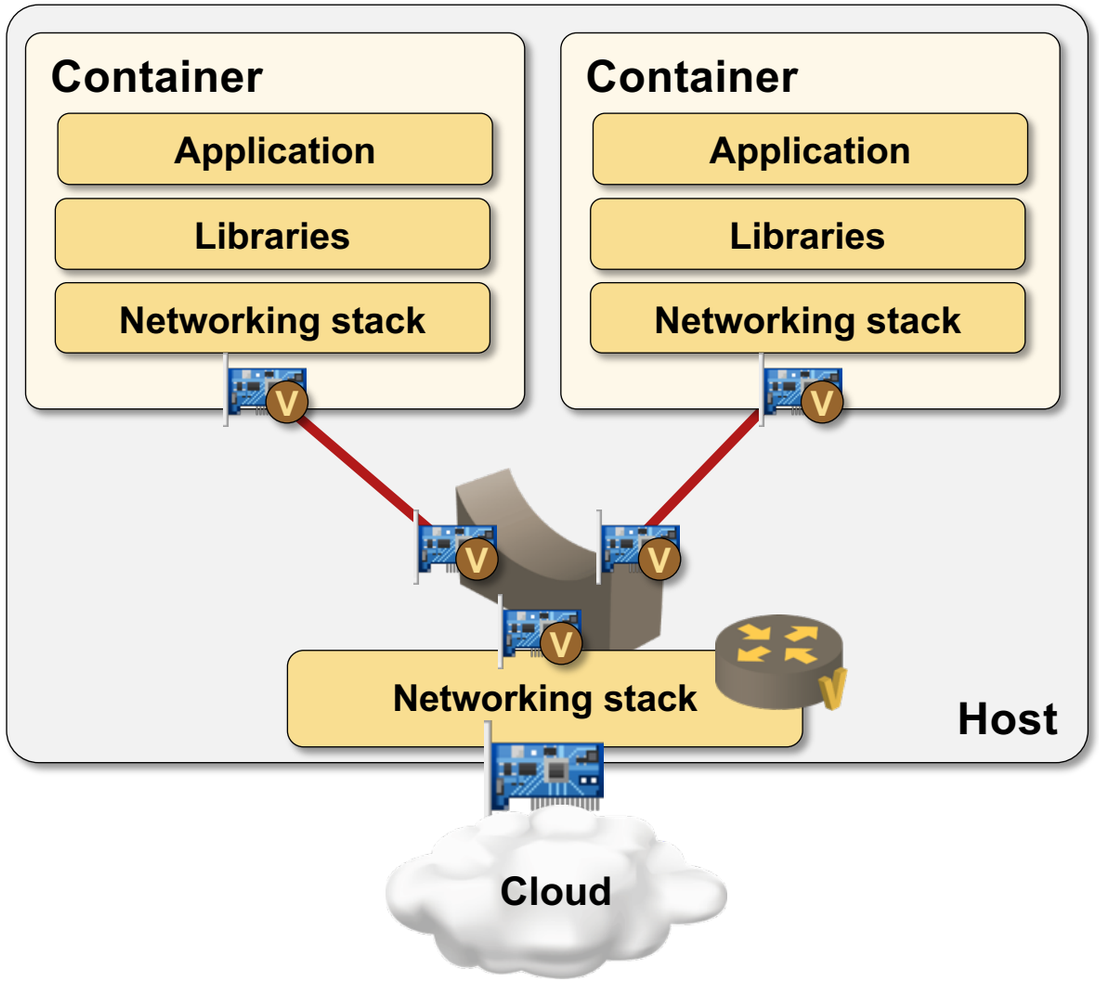
- Container can access the outside world
- NAT is used to hide containers behind host IP address

Host access

- Containers can also communicate with the host (directly with the IP address on **docker0** interface)
- Host processes can communicate with containers (IP prefix of **docker0** interface is in host routing table)

Security implications

- Containers are hosts within the host
- Traffic to containers is *forwarded*
- The usual **iptables** rules do not apply



30-Second iptables Cheat Sheet

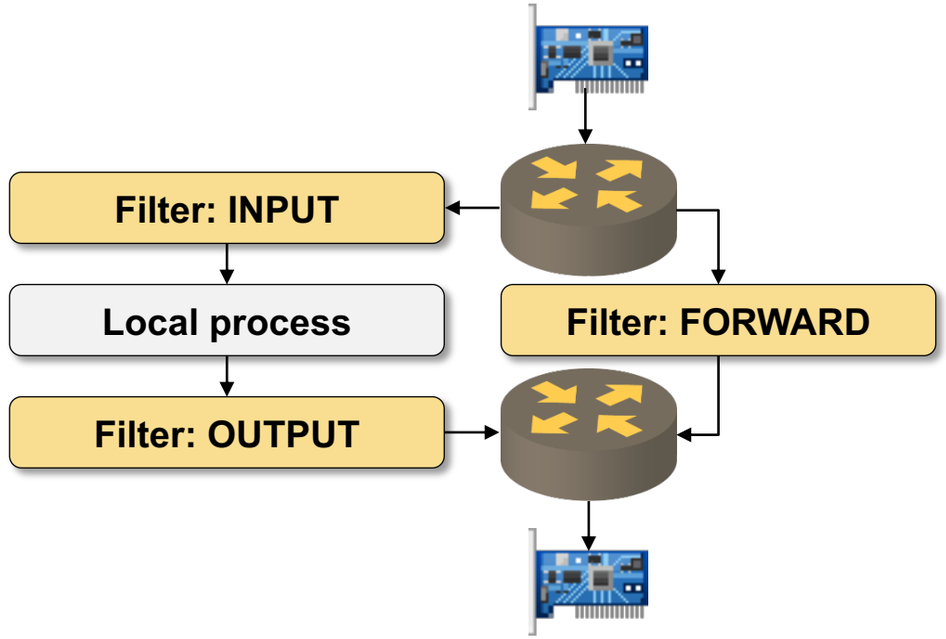
- iptables is the CLI interface to kernel netfilter hooks

Netfilter tables

- **filter** – packet filters
- **nat** – network address translation
- **mangle** – adjust IP headers (example: TTL)
- **raw** – bypass connection tracking
- **security** – set SELinux security context on packets

Netfilter hooks

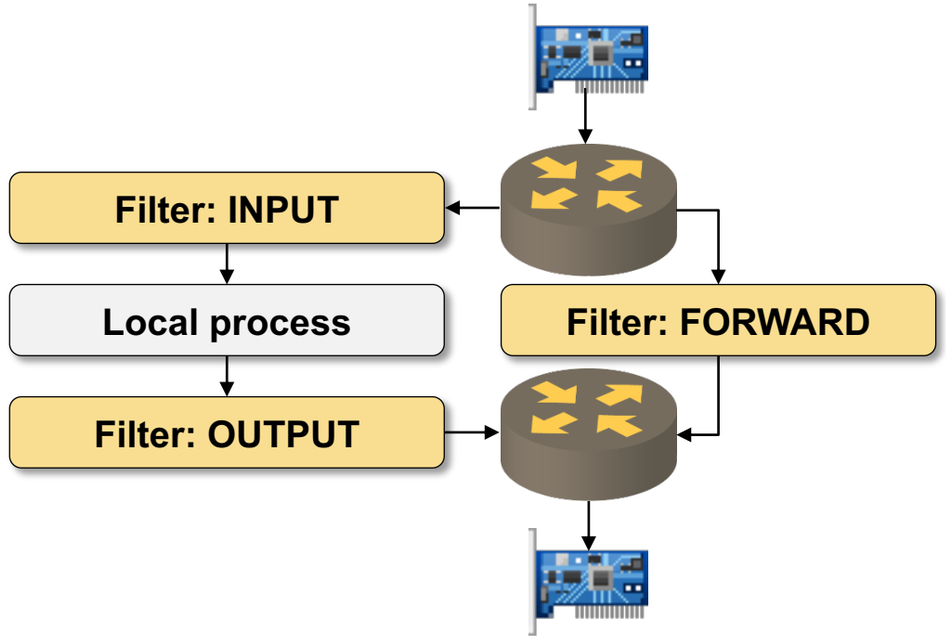
- **prerouting** – between ingress NIC and forwarding table
- **input** – between forwarding table and local process
- **forward** – within the packet forwarding process
- **output** – between local process and forwarding table
- **postrouting** – between forwarding table and egress NIC



30-Second iptables Cheat Sheet

Filter table chains

- **input** – before the packet is delivered to a local process
- **output** – when a packet is generated by a local process
- **forward** – when a packet is forwarded
- There is no **prerouting** or **postrouting** filter chain
- Every item in a chain can execute a well-known action or call another chain



Well-known filter table actions

- **DROP, ACCEPT** – obvious
- **RETURN** – return to previous chain

Important

- Packets delivered to containers are *forwarded*

More @ https://www.digitalocean.com/community/tutorials/a-deep-dive-into-iptables-and-netfilter-architecture

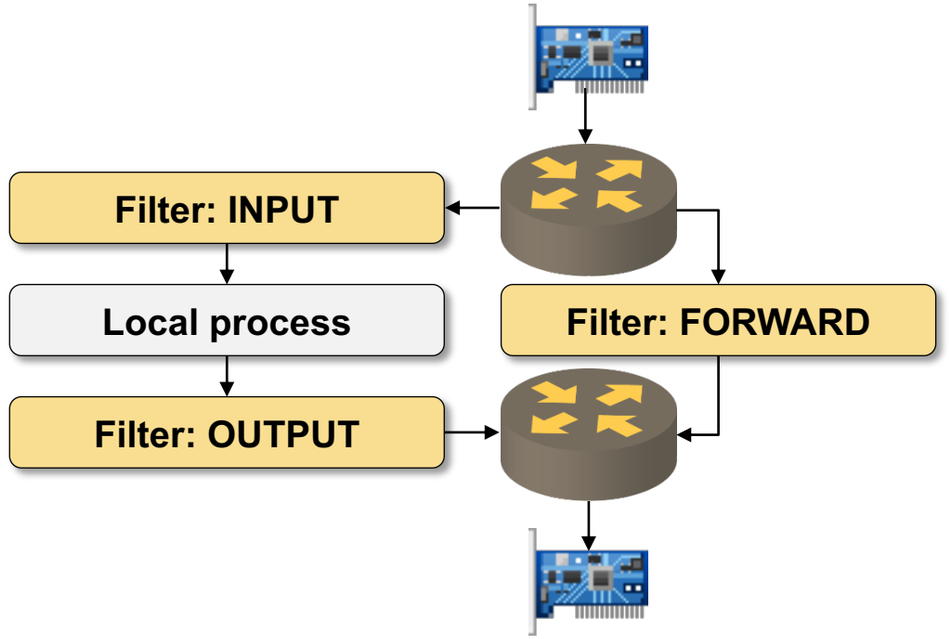
iptables Filtering Rules with Default Docker Bridge

```
~ $ sudo iptables -S
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
~ $
```

Default chain policy

Create a new chain

Add a rule to a chain



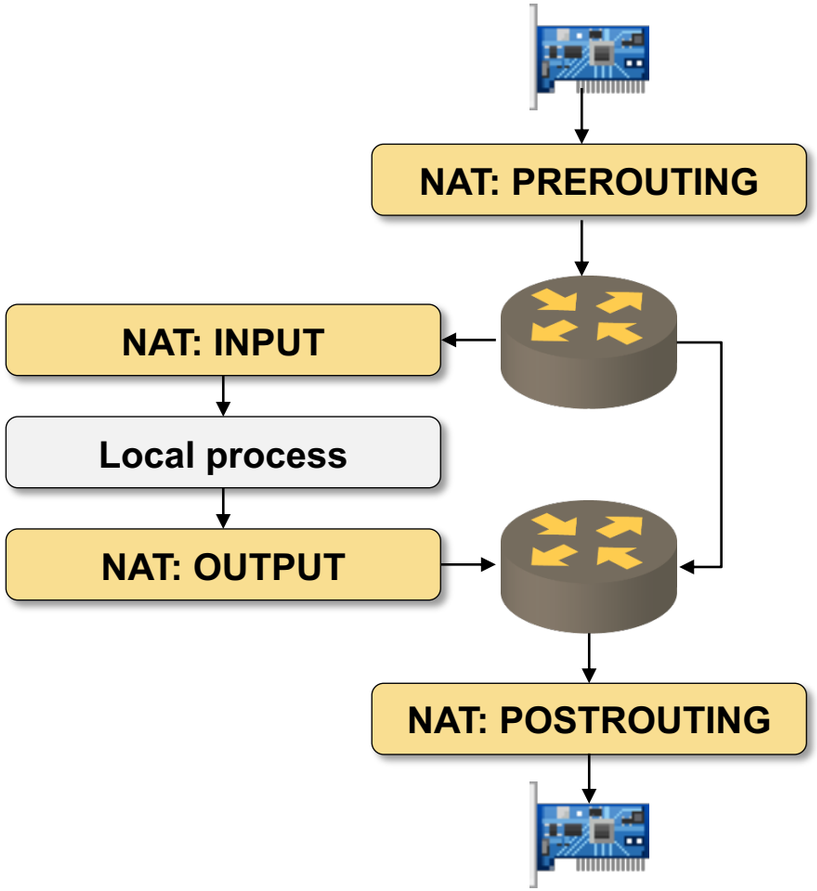
Docker iptables Filtering Rules Explained

- Docker does not use **INPUT** or **OUTPUT** filter chains
- **FORWARD** chain is heavily modified (don't touch it)
- Add your own forwarding filtering rules to **DOCKER-USER** chain
- Example: use **DOCKER-USER** chain to control access to docker daemon
- **DOCKER-ISOLATION-STAGE** chains will become meaningful with multiple networks

```
~ $ sudo iptables -S
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
~ $
```

iptables NAT Rules with Default Docker Bridge

```
~ $ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
~ $
```



Default Docker iptables NAT Rules Explained

- Docker modifies **OUTPUT**, **PREROUTING** and **POSTROUTING** NAT chains

Access to outside world: POSTROUTING

- Source IP address belongs to Docker range and output is not the Docker bridge ➔ perform masquerading (NAT)

Other uses

- **PREROUTING** and **INPUT** chains are used for exposed services ports

```
~ $ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
```

DNS Within Containers

DNS-related files are mounted as filesystems

- /etc/resolv.conf (or nsswitch.conf)
- /etc/hostname
- /etc/hosts

Changing content of DNS-related files

- Options to **docker run** command
- These files cannot be changed in a running container

```
/ # mount|grep etc
/dev/mapper/vagrant--vg-root on /etc/resolv.conf type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/mapper/vagrant--vg-root on /etc/hostname type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/dev/mapper/vagrant--vg-root on /etc/hosts type ext4 (rw,relatime,errors=remount-ro,data=ordered)
/ # cat /etc/resolv.conf
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients directly to
# all known uplink DNS servers. This file lists all configured search domains.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

nameserver 10.0.2.3
/ # cat /etc/hostname
81d898e96395
/ #
```

Default values

- Container ID is used as hostname (replace with -h option)
- Host /etc/resolv.conf file is used as a template
- Local addresses are removed from /etc/resolv.conf, Google DNS servers are added if needed

DNS With Default Docker Bridge

```
/ # cat /etc/resolv.conf
# This file is managed by man:systemd-resolved(8). Do not edit.
#
# This is a dynamic resolv.conf file for connecting local clients directly to
# all known uplink DNS servers. This file lists all configured search domains.
#
# Third party programs must not access this file directly, but only through the
# symlink at /etc/resolv.conf. To manage man:resolv.conf(5) in a different way,
# replace this symlink by a static file or a different symlink.
#
# See man:systemd-resolved.service(8) for details about the supported modes of
# operation for /etc/resolv.conf.

nameserver 10.0.2.3
/ # cat /etc/hostname
81d898e96395
/ #
```

- Containers are using the same DNS server as the Linux host
- Container **hostname** is container ID unless changed with **docker run -h** option
- Containers cannot refer to other containers by name unless you use (legacy) **--link** option

Custom Bridge Networks

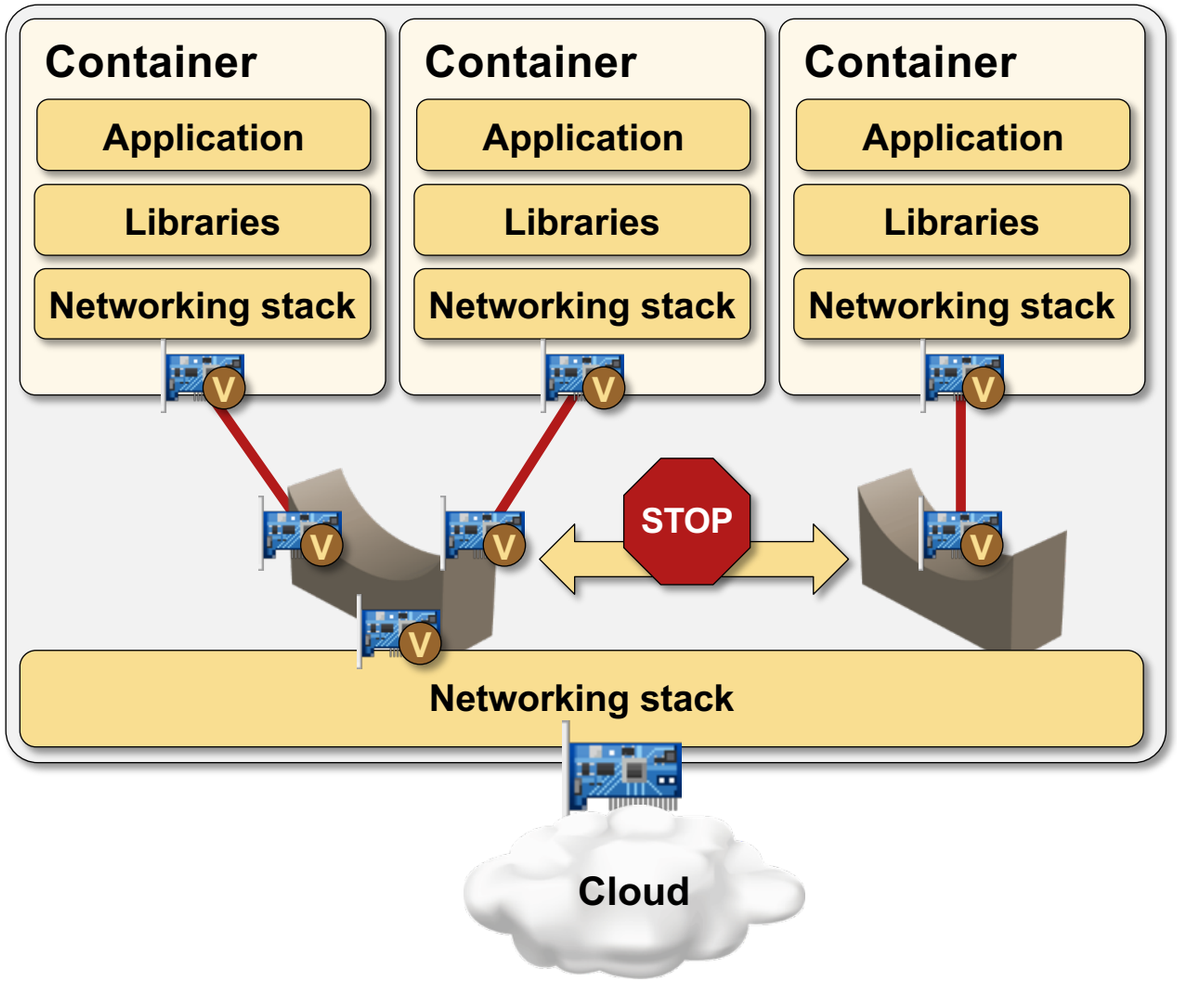
Custom Bridge Networks

Docker host can contain numerous networks

- Create with **docker network create** command
- Specify network *driver*
  (bridge, host, macvlan, ipvlan, overlay…)
- Specify additional options
  (inter-container isolation)
- Connect containers to networks with **--network** option of **docker run** command

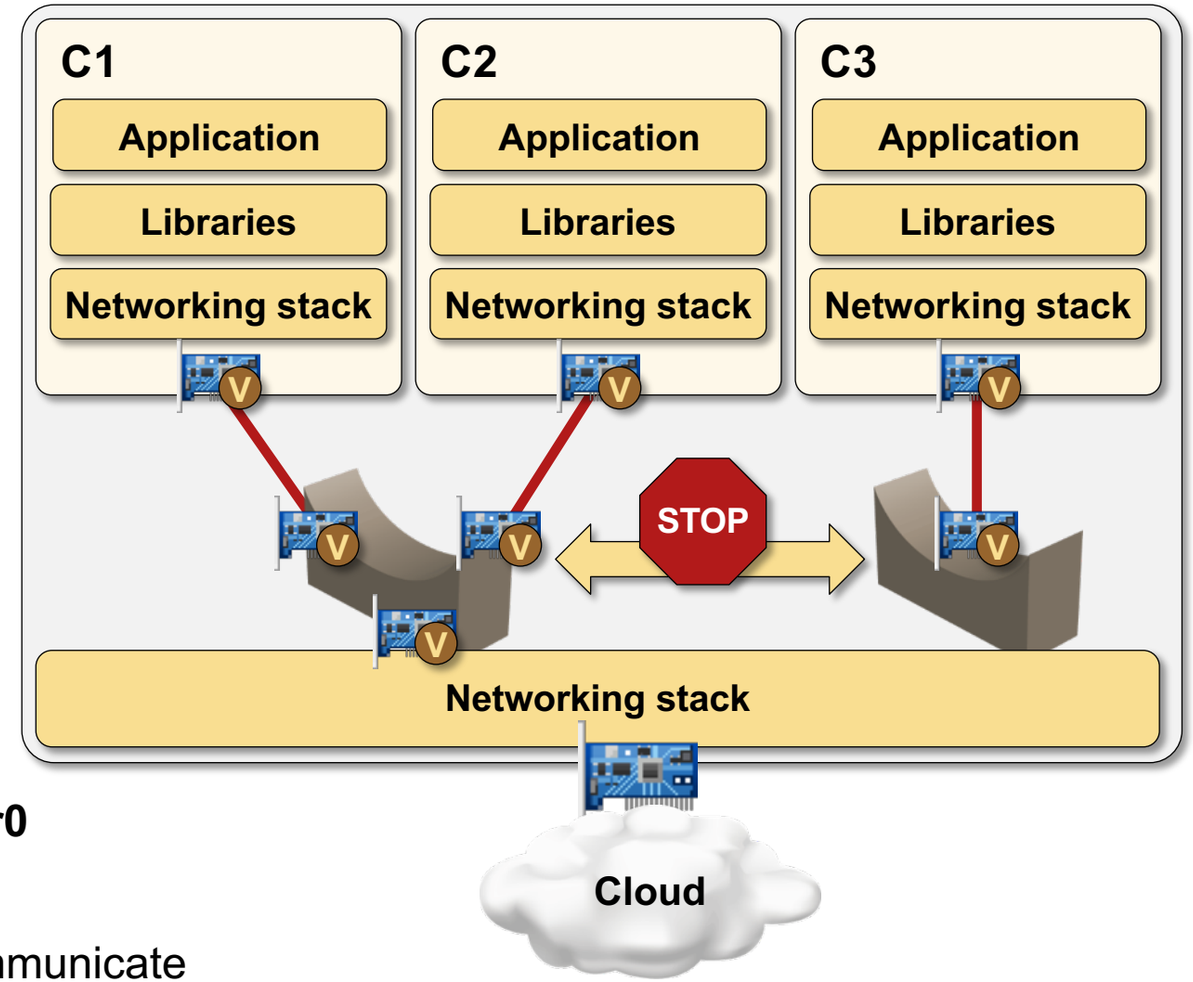
Custom bridge network implementation

- Similar to **docker0**
- iptables are used to prevent inter-network container communication



Creating a Custom Bridge Network

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -itd --name c1 \
  --network=br0 busybox
$ docker run -itd --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```



DNS and connectivity tests

- DNS works within custom bridge networks but not on **docker0**
- All containers can reach Linux host and outside world
- Containers connected to different Docker networks can't communicate

Creating a Docker Bridge Network

- Use **bridge** driver in **docker network create** command
- Specify the subnet or subnet / ip-range / gateway
- Docker bridge name does not match the Linux bridge name
  (use com.docker.network.bridge.name option to change it)
- Bridge driver options can be used to disable NAT, change MTU, or default port binding IP address

```
~ $ docker network create --driver=bridge --subnet=192.168.99.0/24 br0
2816c8c2923c7a4afe8d4650aefe6d32255568dc1240ac95fe73e77cba1bdea9
~ $ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
2816c8c2923c        br0                 bridge              local
8922d6cb90ab        bridge              bridge              local
89a2ab52049c        host                host                local
7566f4bcde97        none                null                local
~ $ brctl show
bridge name     bridge id               STP enabled     interfaces
br-2816c8c2923c         8000.0242f1d89c6d       no
docker0         8000.0242518af138       no              veth1c9c1ad
~ $
```

Advanced Docker Network Options

```
$ docker network create --driver=bridge \
    --subnet=192.168.99.0/24 --ip-range=192.168.99.16/28 \
    --gateway=192.168.99.254 \
    -o 'com.docker.network.bridge.name=br0' br0
```

```
~ $ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
e3fb7b732eb3        br0                 bridge              local
b4bb5577c0da        bridge              bridge              local
ebd51b780273        host                host                local
e25f3b41fb15        none                null                local
~ $ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.0242914287cd       no
docker0         8000.02426ff1fa4f       no
```

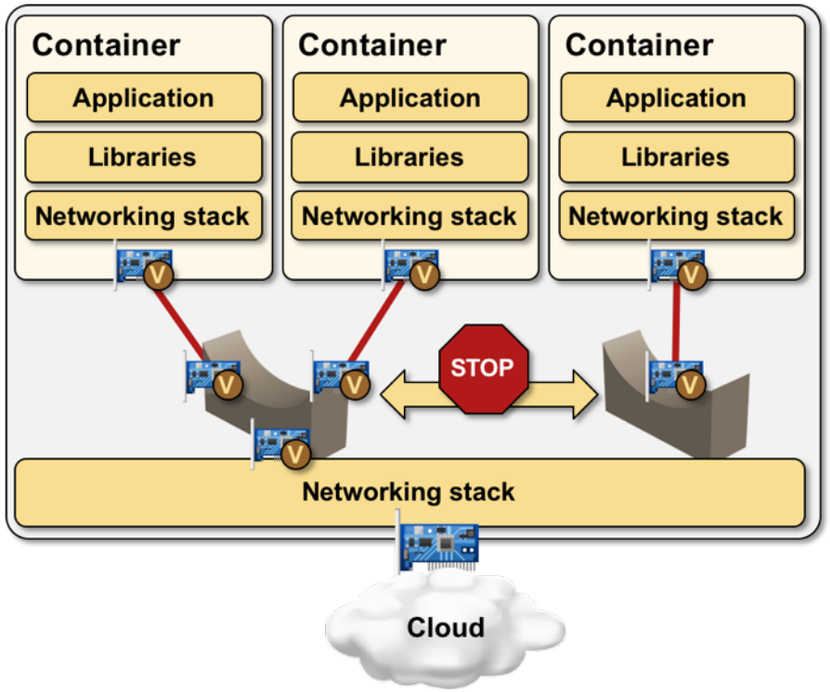
Advanced Docker Network Options

```
$ docker network create --driver=bridge \
    --subnet=192.168.99.0/24 --ip-range=192.168.99.16/28 \
    --gateway=192.168.99.254 \
    -o 'com.docker.network.bridge.name=br0' br0
$ docker run -it --network br0 --rm busybox
```

```
~ $ docker run -it --network br0 --rm busybox
/ # ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
57: eth0@if58: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:c0:a8:63:10 brd ff:ff:ff:ff:ff:ff
    inet 192.168.99.16/24 brd 192.168.99.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ip route
default via 192.168.99.254 dev eth0
192.168.99.0/24 dev eth0 scope link  src 192.168.99.16
```

Docker Containers Connected to Multiple Networks

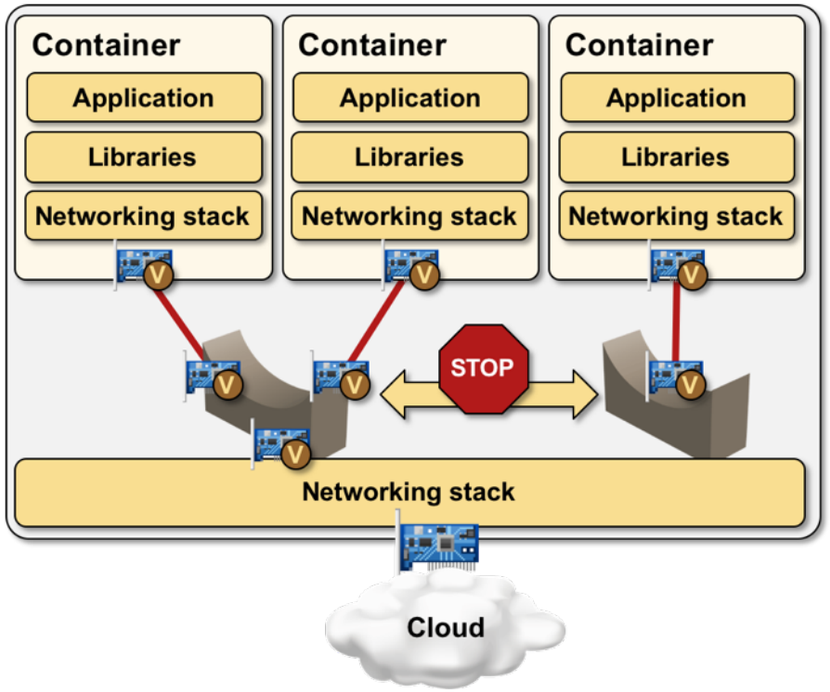
```
~ $ docker network inspect br0|jq .[0].Containers
{
  "8db02087c6c6bc159c7ff4839f889fc4df31fe0580c327c16ce8e6883d9f68d6": {
    "Name": "C2",
    "EndpointID": "af6132e8400d12360cfd682fb5adc3e48b3bd6cb5aff5617c91b007469ba746d",
    "MacAddress": "02:42:c0:a8:63:03",
    "IPv4Address": "192.168.99.3/24",
    "IPv6Address": ""
  },
  "ff5662d91a5d4e14c58026f774fec1f5f812ce193270632c4ecdf26d372db33d": {
    "Name": "C1",
    "EndpointID": "4cf42f6289011ea43a1efbf0a123bb16265ccb72c1dfed74eb0b1bc92900a6e7",
    "MacAddress": "02:42:c0:a8:63:02",
    "IPv4Address": "192.168.99.2/24",
    "IPv6Address": ""
  }
}
~ $ docker network inspect bridge|jq .[0].Containers
{
  "b5e0a9e552a8f0ba1e6ddc774e993f3daa658c85faf7d581b49c20fb7d07c960": {
    "Name": "C3",
    "EndpointID": "e113a380ba4b4b488fb601616eb3c8f4024497d094cc8c4c04bb45bb66d3b00b",
    "MacAddress": "02:42:ac:11:00:02",
    "IPv4Address": "172.17.0.2/16",
    "IPv6Address": ""
```



Docker Containers Connected to a Network with Advanced Options

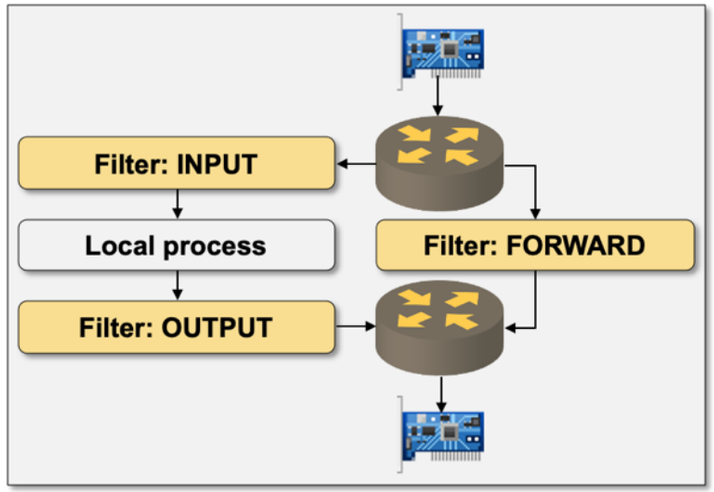
```
  .[]|{
    Name, Gateway: .IPAM.Config[]|.Gateway,
    Endpoints: [
      .Containers|to_entries[]|{
        Name: .value.Name, IPv4: .value.IPv4Address }
    ]
  }
```

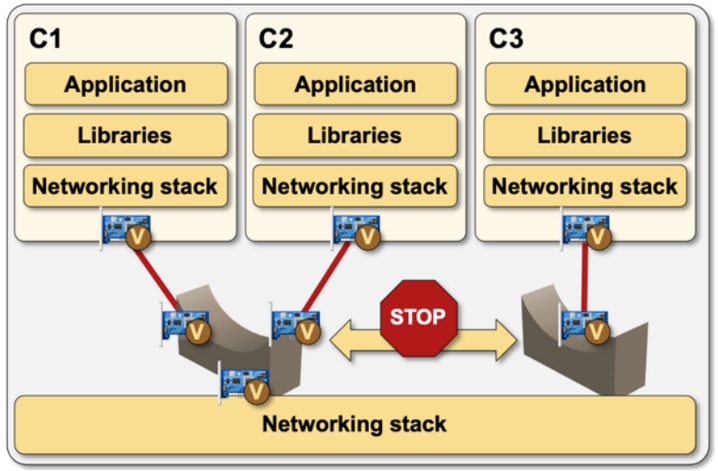
```
filter $ docker network inspect br0|jq -f /vagrant/filter/netaddr
{
  "Name": "br0",
  "Gateway": "192.168.99.254",
  "Endpoints": [
    {
      "Name": "C1",
      "IPv4": "192.168.99.16/24"
    },
    {
      "Name": "C2",
      "IPv4": "192.168.99.17/24"
    }
  ]
}
```



iptables Filtering Rules with Multiple Docker Bridges

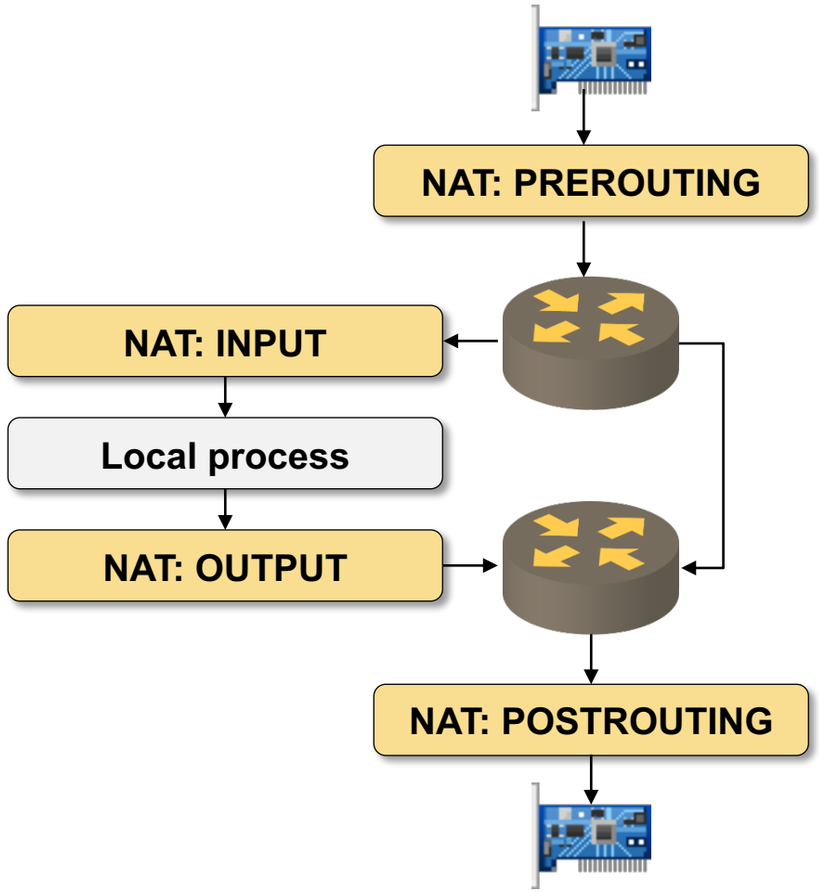
```
~ $ sudo iptables -S
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o br-2816c8c2923c -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o br-2816c8c2923c -j DOCKER
-A FORWARD -i br-2816c8c2923c ! -o br-2816c8c2923c -j ACCEPT
-A FORWARD -i br-2816c8c2923c -o br-2816c8c2923c -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i br-2816c8c2923c ! -o br-2816c8c2923c -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o br-2816c8c2923c -j DROP
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
```





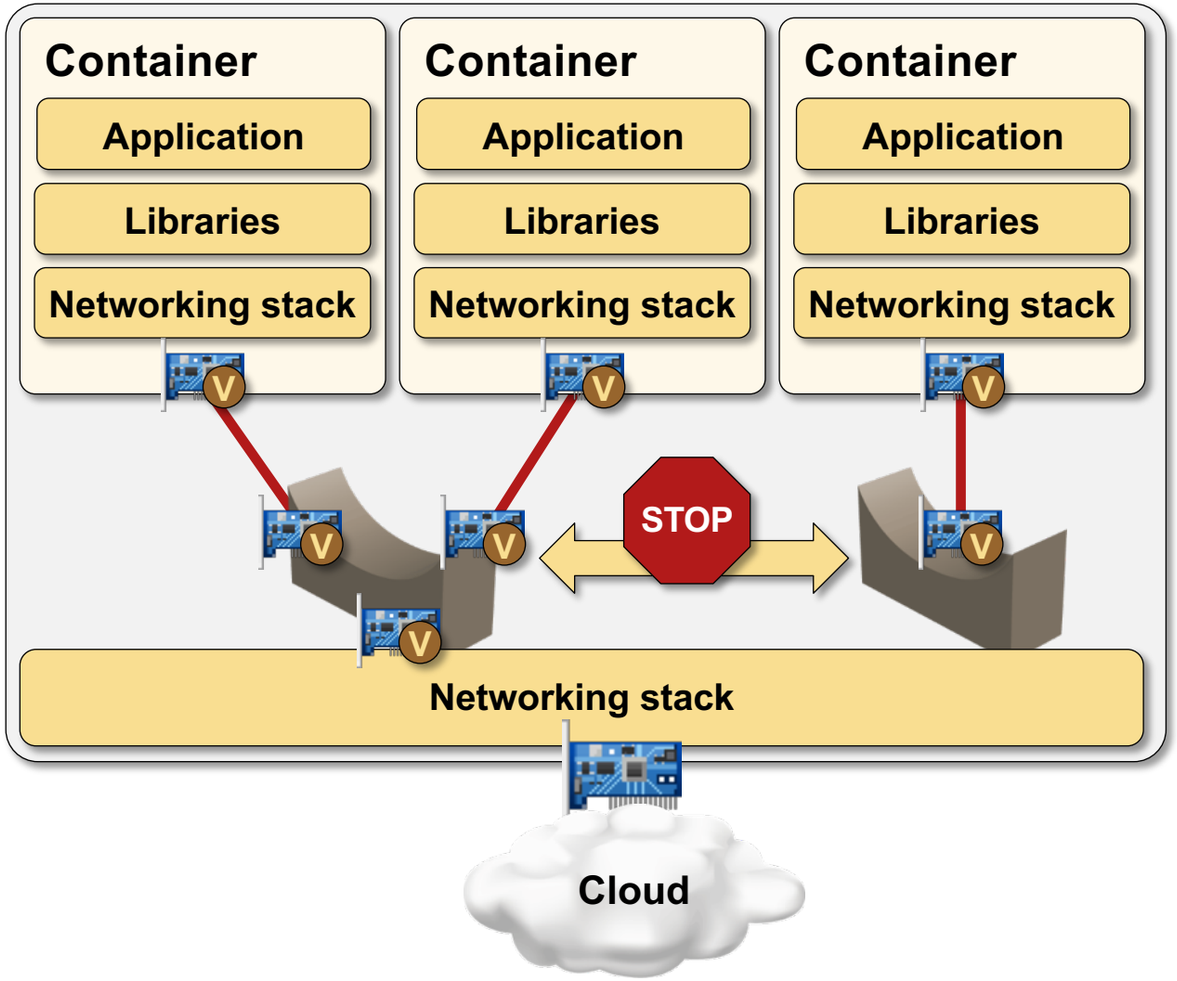
iptables NAT Rules with Multiple Docker Bridges

```
~ $ sudo iptables -S -t nat
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 192.168.99.0/24 ! -o br-2816c8c2923c -j MASQUERADE
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A DOCKER -i br-2816c8c2923c -j RETURN
-A DOCKER -i docker0 -j RETURN
~ $
```



Custom Bridge Networks – Reachability Summary

Destination	
Containers connected to the same network	
Containers connected to other networks	✗
Host processes	
External network	



DNS Server in Custom Docker Networks

```
$ docker run --network br0 -itd busybox
```

```
/ # cat /etc/resolv.conf
nameserver 127.0.0.11
options ndots:0
/ # netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         192.168.99.1    0.0.0.0         UG        0 0          0 eth0
192.168.99.0    0.0.0.0         255.255.255.0   U         0 0          0 eth0
/ # ping C1
PING C1 (192.168.99.2): 56 data bytes
64 bytes from 192.168.99.2: seq=0 ttl=64 time=0.048 ms
64 bytes from 192.168.99.2: seq=1 ttl=64 time=0.075 ms
64 bytes from 192.168.99.2: seq=2 ttl=64 time=0.074 ms
^C
--- C1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.048/0.065/0.075 ms
/ #
```

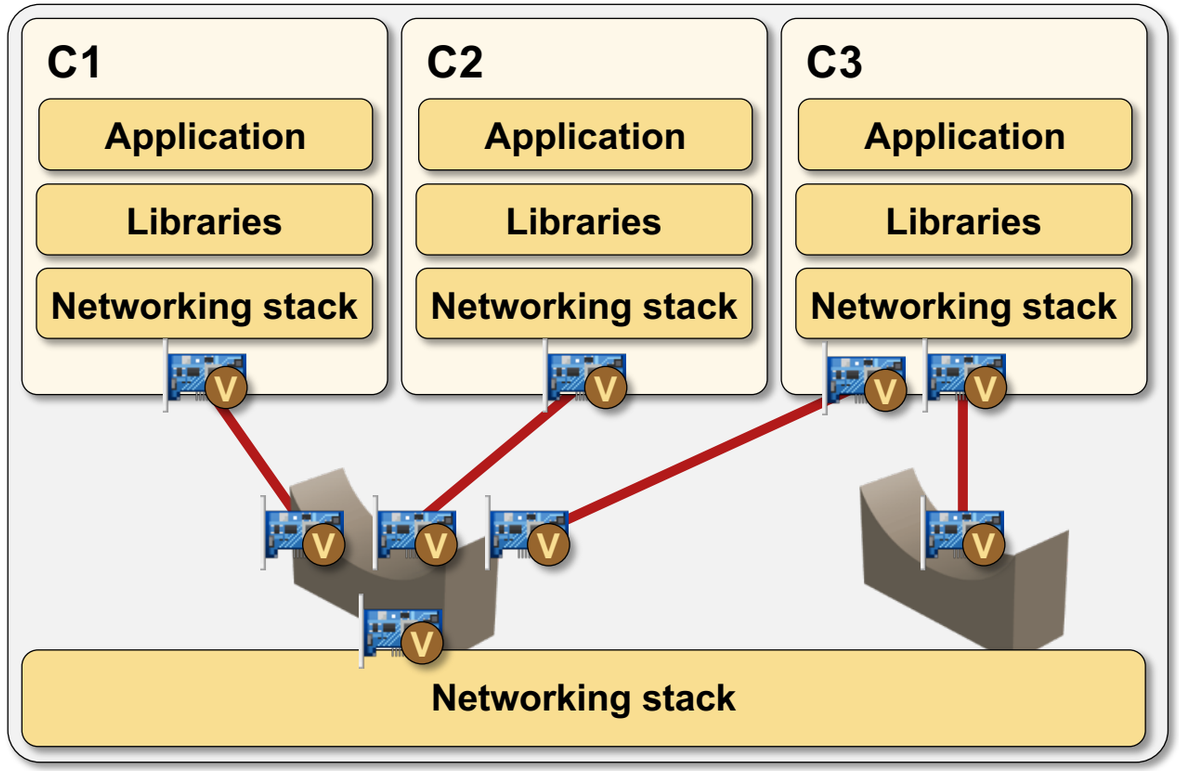
- Docker is running a DNS server for every custom network
- The Docker DNS server is always reachable on 127.0.0.11
- Docker DNS server resolves names of other containers connected to the same bridge
- DNS server provides name resolution for all networks a container is connected to

Connecting a Container to Multiple Networks

```
$ docker network connect br0 C3
```

```
/ # netstat -rn
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         192.168.99.1    0.0.0.0         UG        0 0          0 eth1
172.17.0.0      0.0.0.0         255.255.0.0     U         0 0          0 eth0
192.168.99.0    0.0.0.0         255.255.255.0   U         0 0          0 eth1
/ # cat /etc/resolv.conf
nameserver 127.0.0.11
options ndots:0
/ #
```

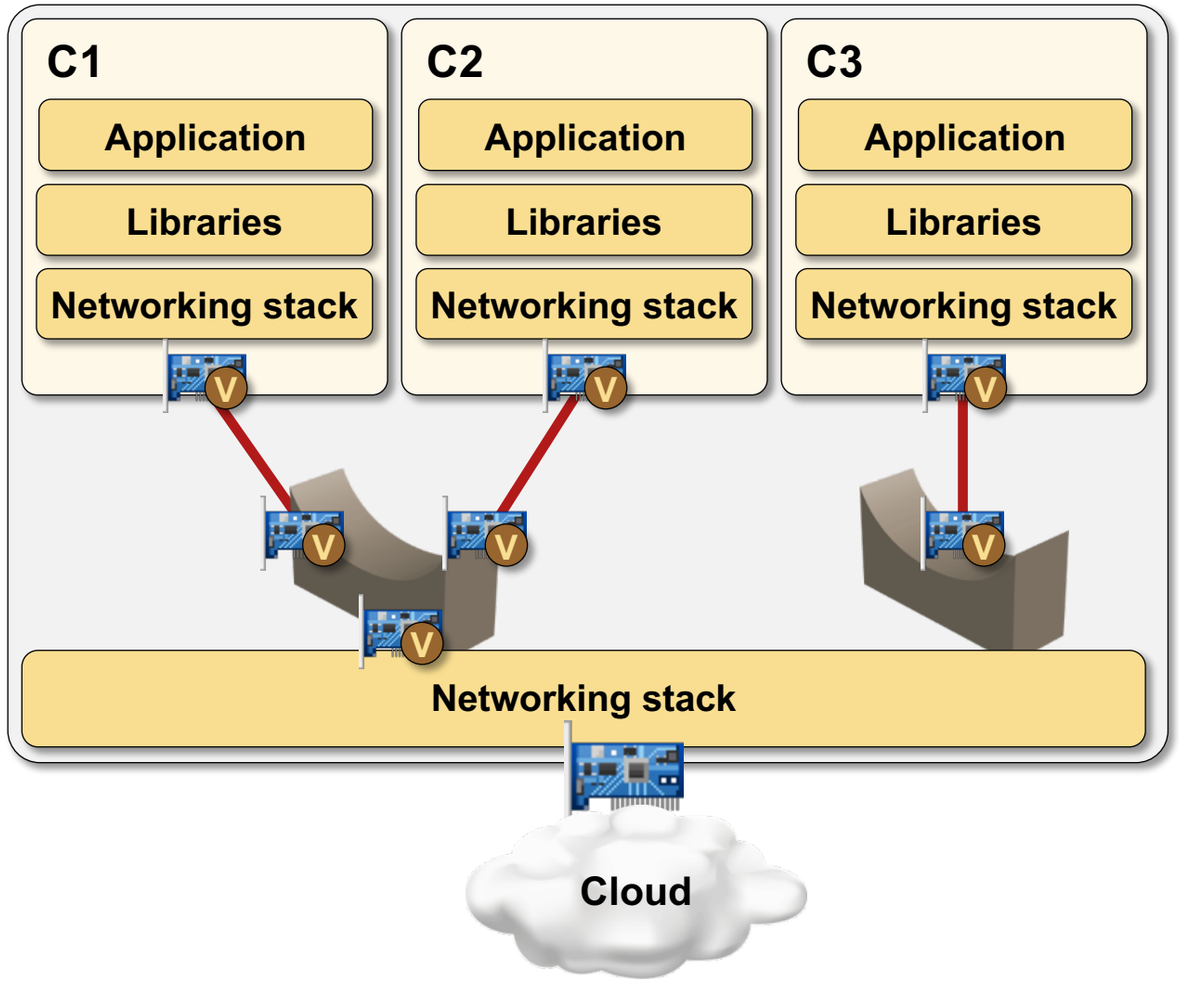
- A container can be connected to many Docker networks
- Docker automatically adds interfaces, and adjusts container routing table
- Docker DNS server is enabled as soon as the container connects to a custom network



Custom Networks with Container Isolation

Custom Bridge Network with Container Isolation

```
$ docker network create \
  --driver=bridge \
  -o "com.docker.network.bridge.
      enable_icc=false"
  --subnet=192.168.99.0/24 br0
$ docker run -itd --name c1 \
  --network=br0 busybox
$ docker run -itd --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```

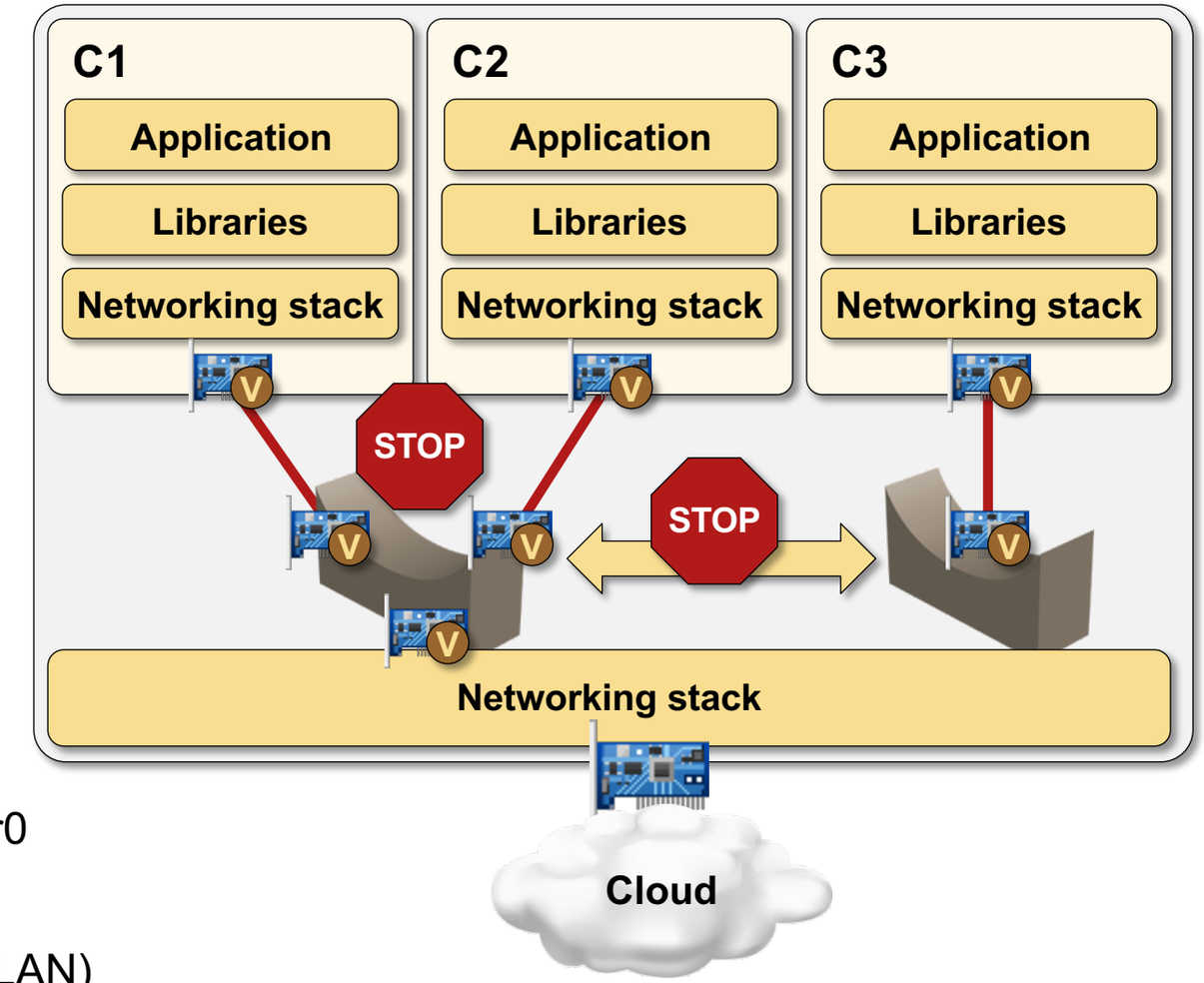


Demo: Bridge Networks without inter-container communication

Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

Custom Bridge Network with Container Isolation

```
$ docker network create \
  --driver=bridge \
  -o "com.docker.network.bridge.
      enable_icc=false"
  --subnet=192.168.99.0/24 br0
$ docker run -itd --name c1 \
  --network=br0 busybox
$ docker run -itd --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```

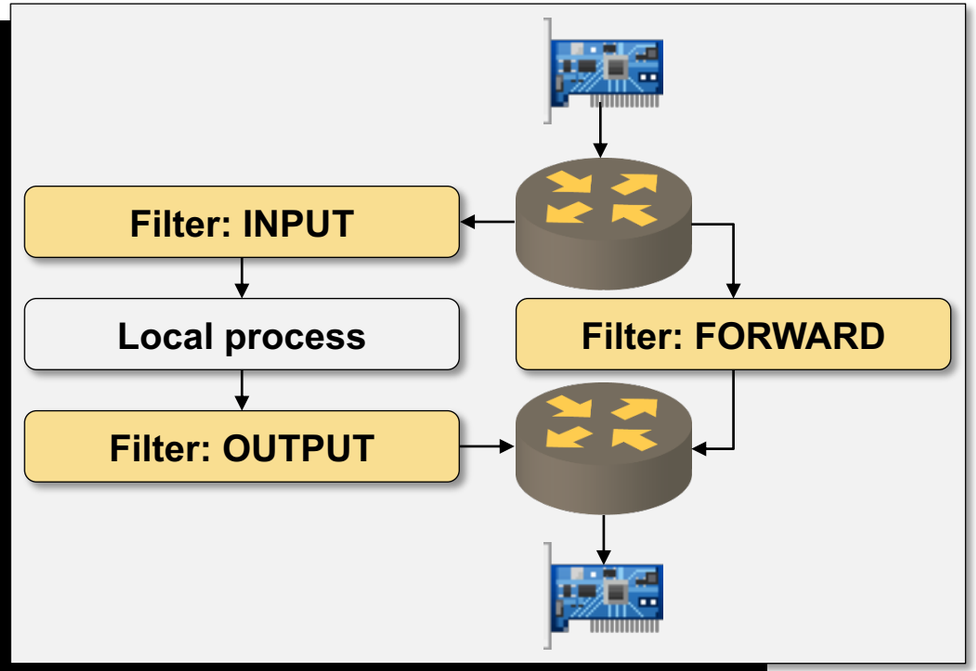


DNS and connectivity tests

- DNS works within custom bridge networks but not on docker0
- All containers can reach Linux host and outside world
- Containers in custom network cannot communicate (like PVLAN)

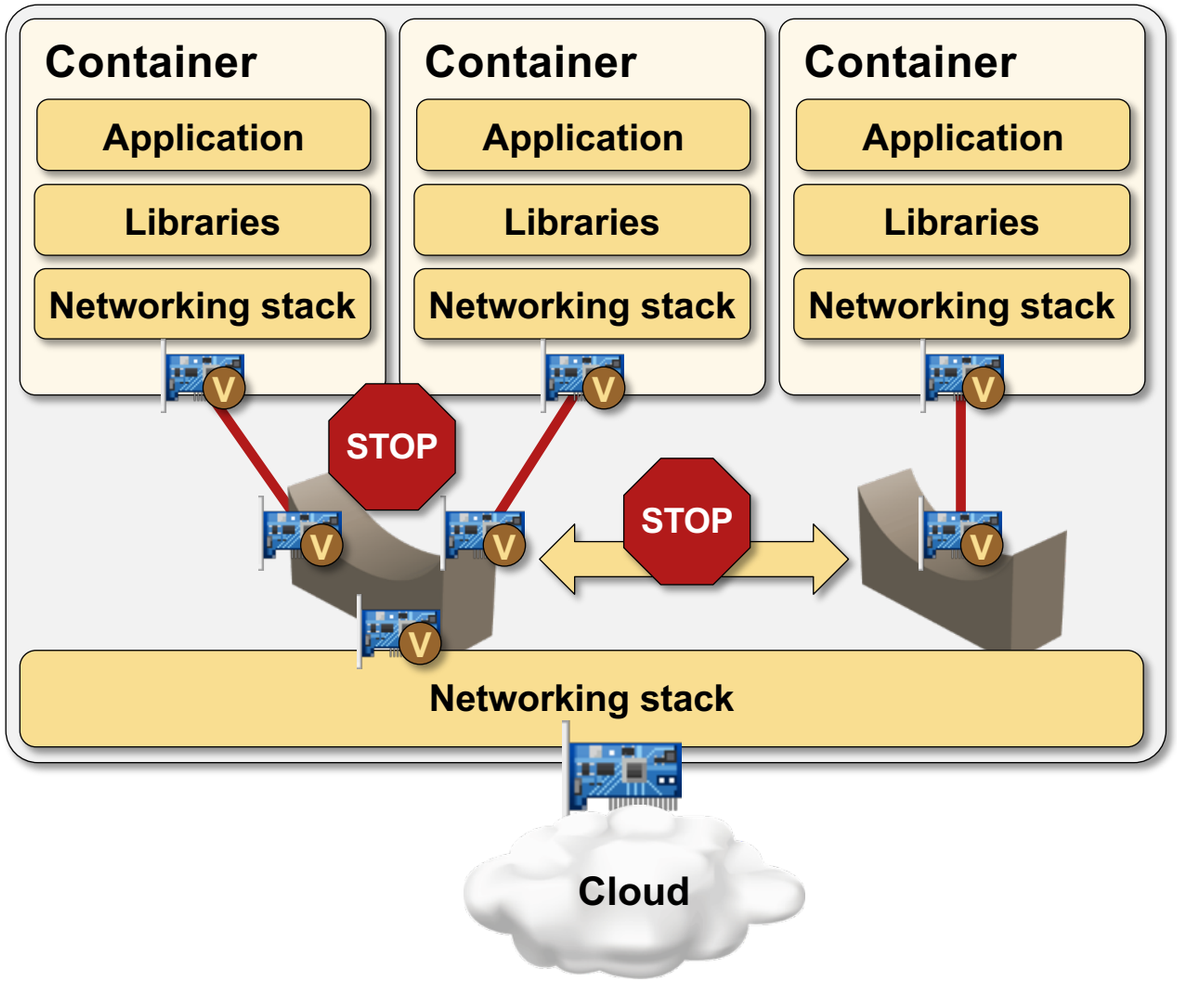
iptables Filtering Rules with Container Isolation

```
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o br-6111926ec5ec -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o br-6111926ec5ec -j DOCKER
-A FORWARD -i br-6111926ec5ec ! -o br-6111926ec5ec -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -i br-6111926ec5ec -o br-6111926ec5ec -j DROP
-A DOCKER-ISOLATION-STAGE-1 -i br-6111926ec5ec ! -o br-6111926ec5ec -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o br-6111926ec5ec -j DROP
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
~ $
```



Networks with Container Isolation – Reachability Summary

Destination	
Containers connected to the same network	✗
Containers connected to other networks	✗
Host processes	
External network	



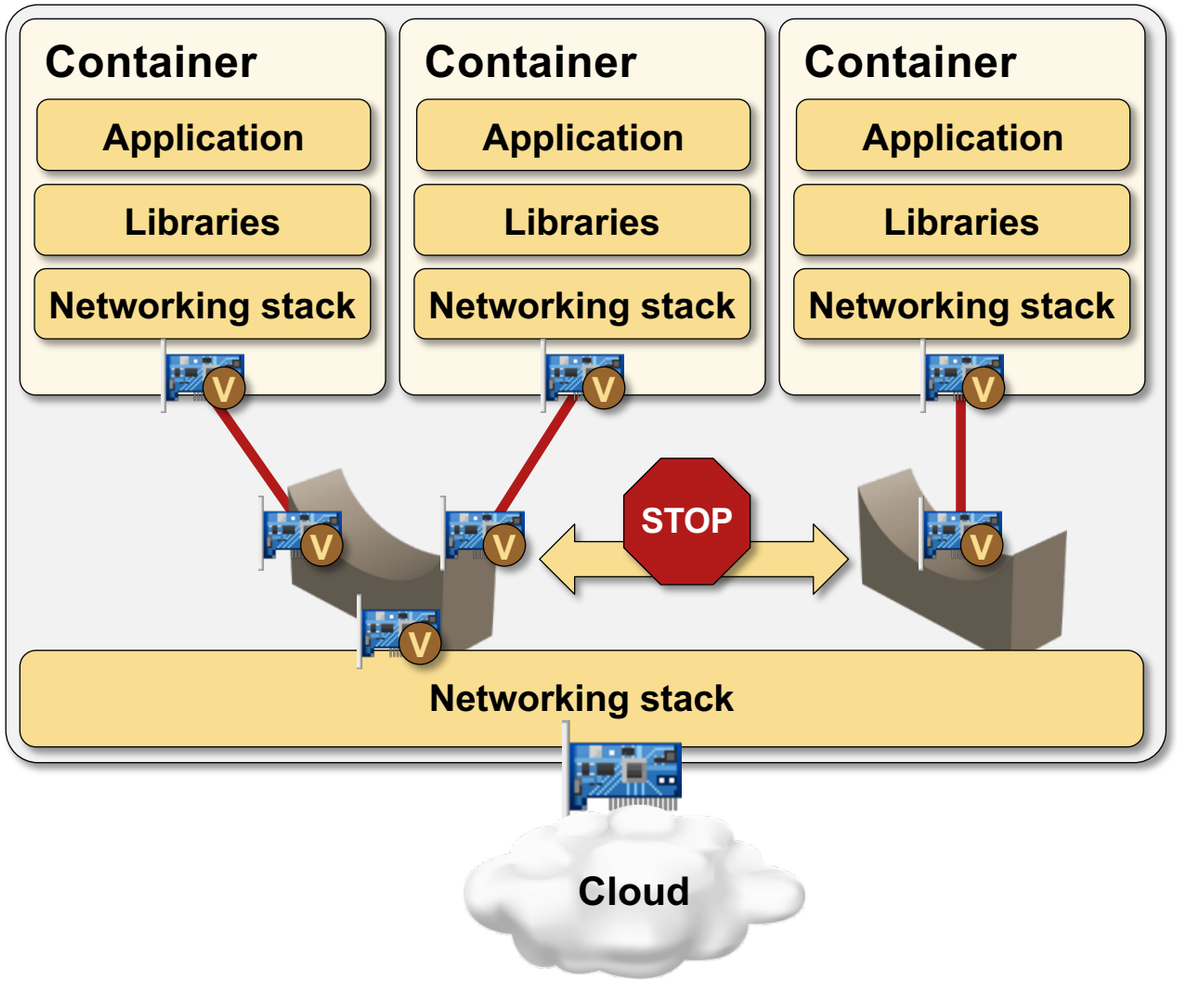
Recap: Other Docker Network Options

Bridge Networks

- Bridge (Linux interface) name
- Enable IP masquerading
- Default IP for inbound port binding
- MTU

Common to all network types

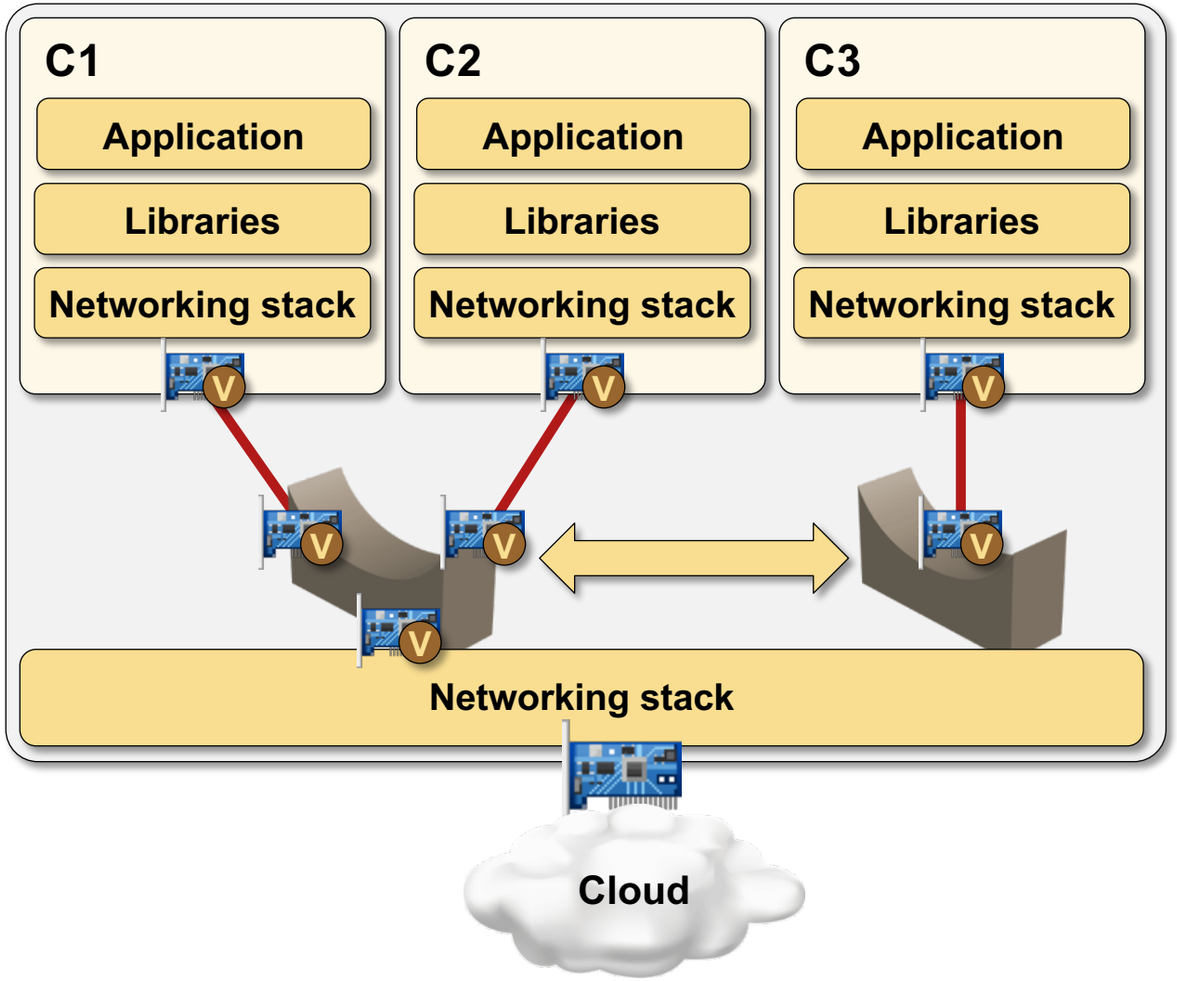
- IP range or subnet
- Gateway
- IPv6 support
- Isolated (internal) network



Isolated (Internal) Docker Networks

Isolated (Internal) Networks

```
$ docker network create \
  --driver=bridge \
  --internal \
  --subnet=192.168.99.0/24 br0
$ docker run -itd --name c1 \
  --network=br0 busybox
$ docker run -itd --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```

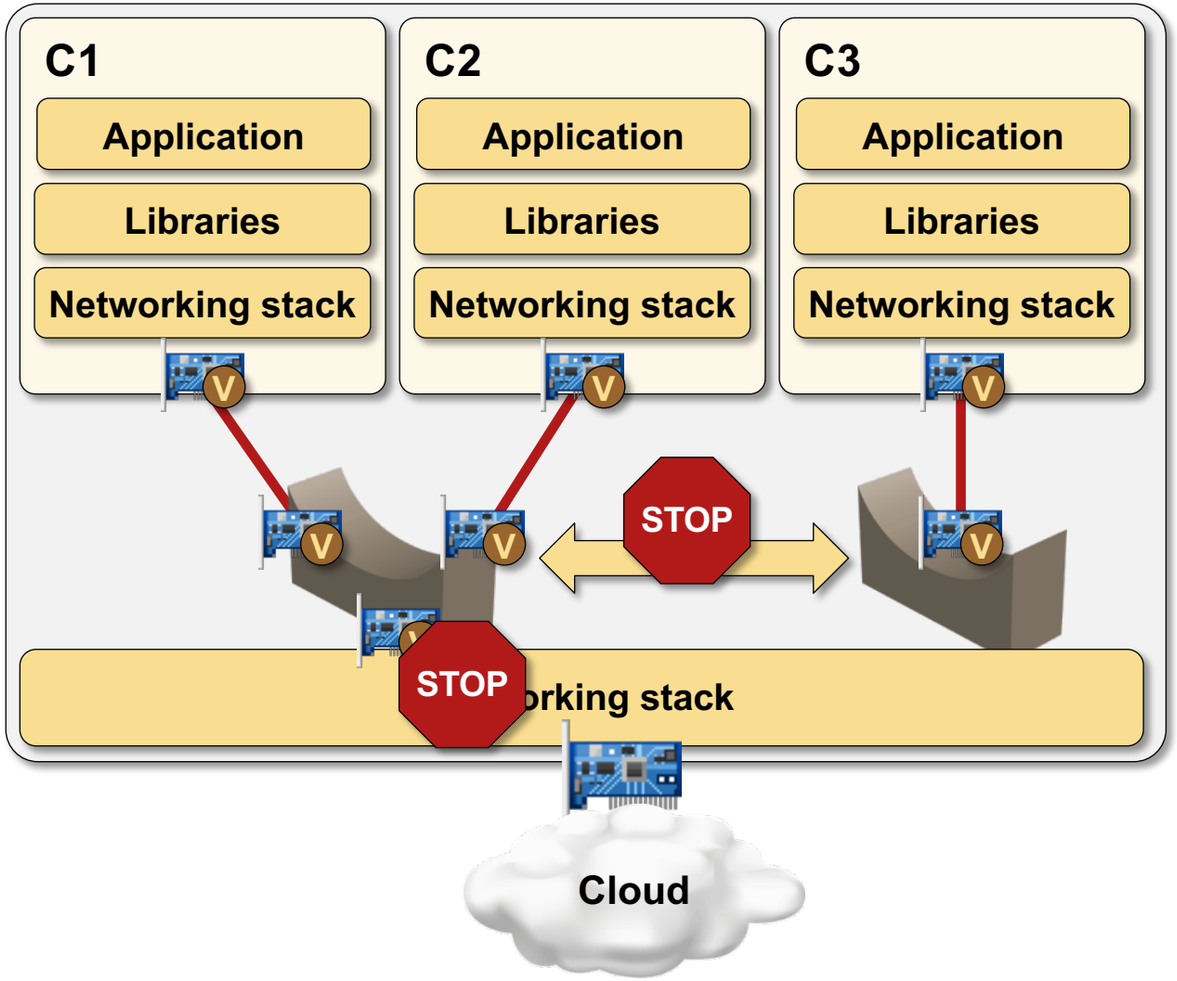


Demo: Isolated Networks

Source code @ https://github.com/ipspace/docker-examples/tree/master/labsc

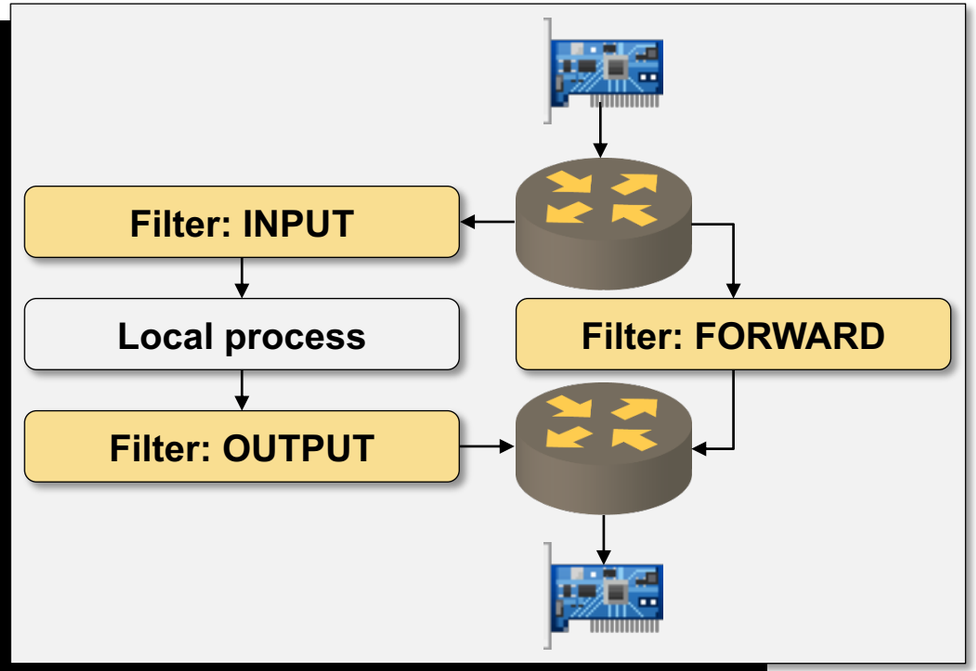
Isolated (Internal) Networks

```
$ docker network create \
  --driver=bridge \
  --internal \
  --subnet=192.168.99.0/24 br0
$ docker run -itd --name c1 \
  --network=br0 busybox
$ docker run -itd --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```



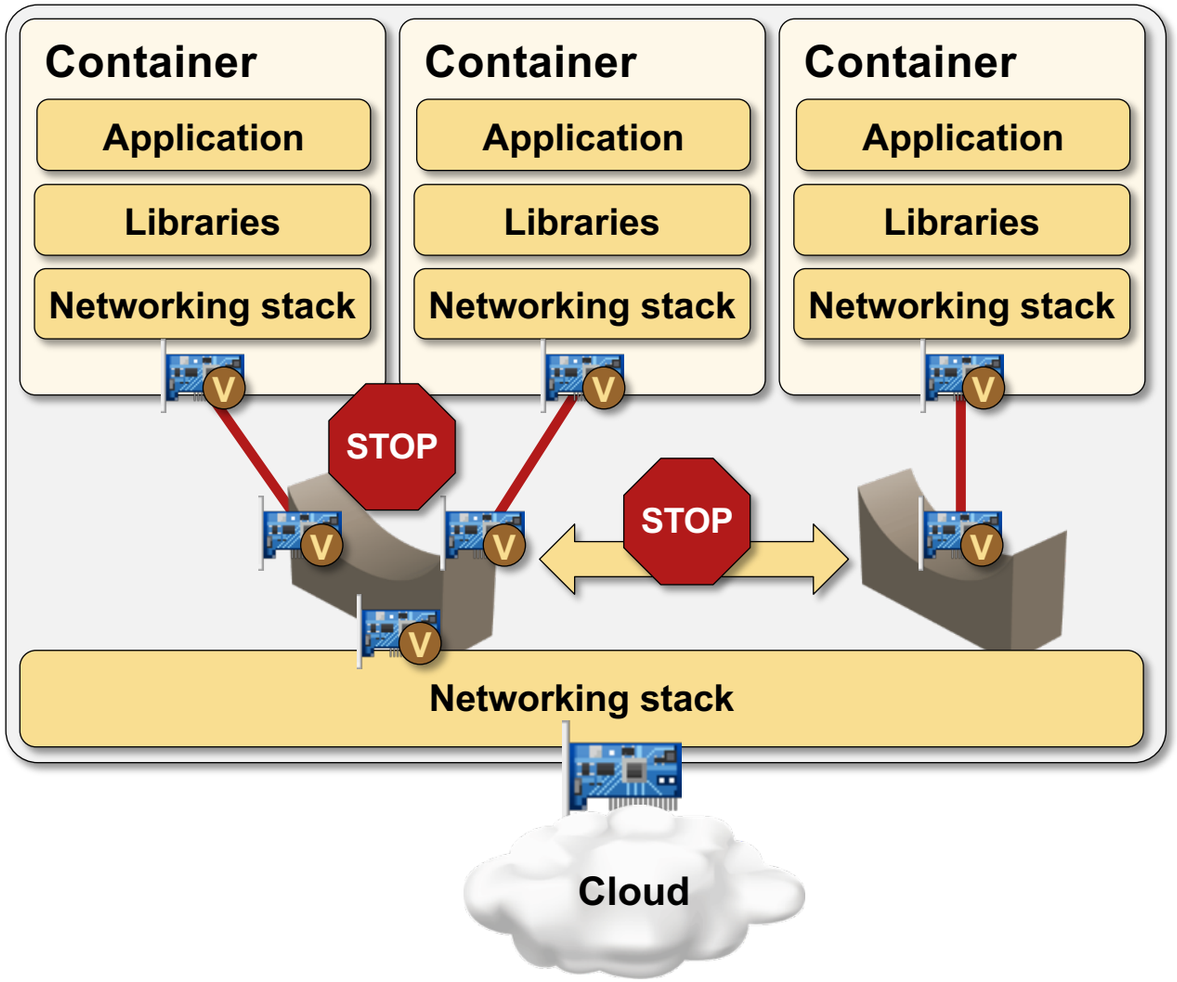
iptables Filtering Rules for Internal Networks

```
~ $ sudo iptables -S
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -i br-16b8c69a0f98 -o br-16b8c69a0f98 -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 ! -s 192.168.99.0/24 -o br-16b8c69a0f98 -j DROP
-A DOCKER-ISOLATION-STAGE-1 ! -d 192.168.99.0/24 -i br-16b8c69a0f98 -j DROP
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
~ $
```



Internal Networks – Reachability Summary

Destination	
Containers connected to the same network	
Containers connected to other networks	✗
Host processes	
External network	✗



Special Docker Network Types

Demo: Special Docker Network Types

Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

Special Docker Networking Types

Special network types

- **none** – container has no network interfaces
- **host** – container uses host TCP stack

Use cases for host networking

- Privileged networking daemons
  (example: can modify the host routing table)

```
~ $ docker run --rm -it --network none busybox
/ # ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
/ #
```

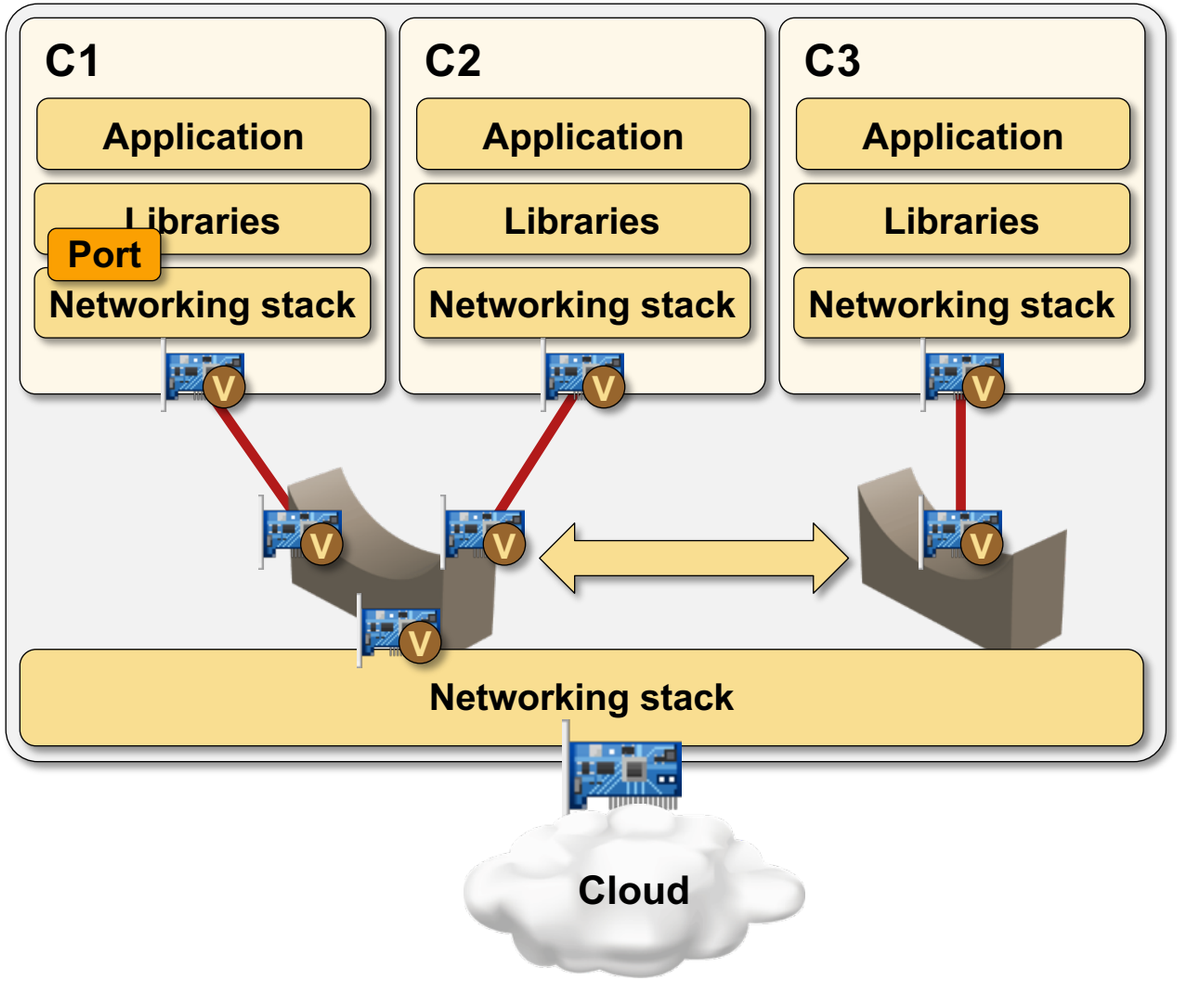
```
~ $ docker run --rm -it --network host busybox
/ # ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether 08:00:27:8b:d5:11 brd ff:ff:ff:ff:ff:ff
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel qlen 1000
    link/ether 08:00:27:ae:a4:7b brd ff:ff:ff:ff:ff:ff
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:18:74:98:72 brd ff:ff:ff:ff:ff:ff
/ #
```

Running Network Services in Docker Containers

ipSpace

Running a Container with an Exposed Port (Network Service)

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -d --name app \
  --network=br0 webapp
$ docker run -it --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```

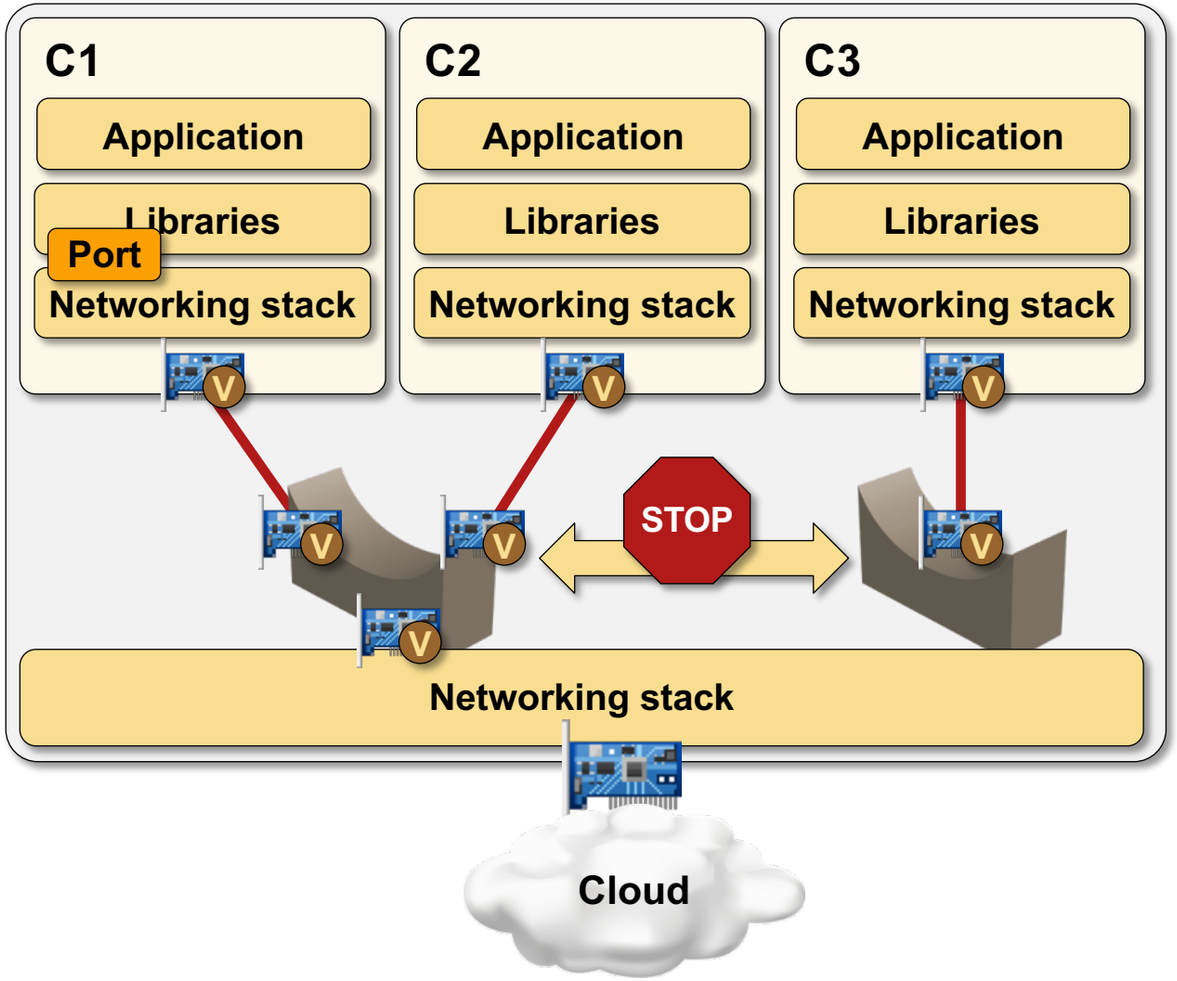


Demo: Container running a service

Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

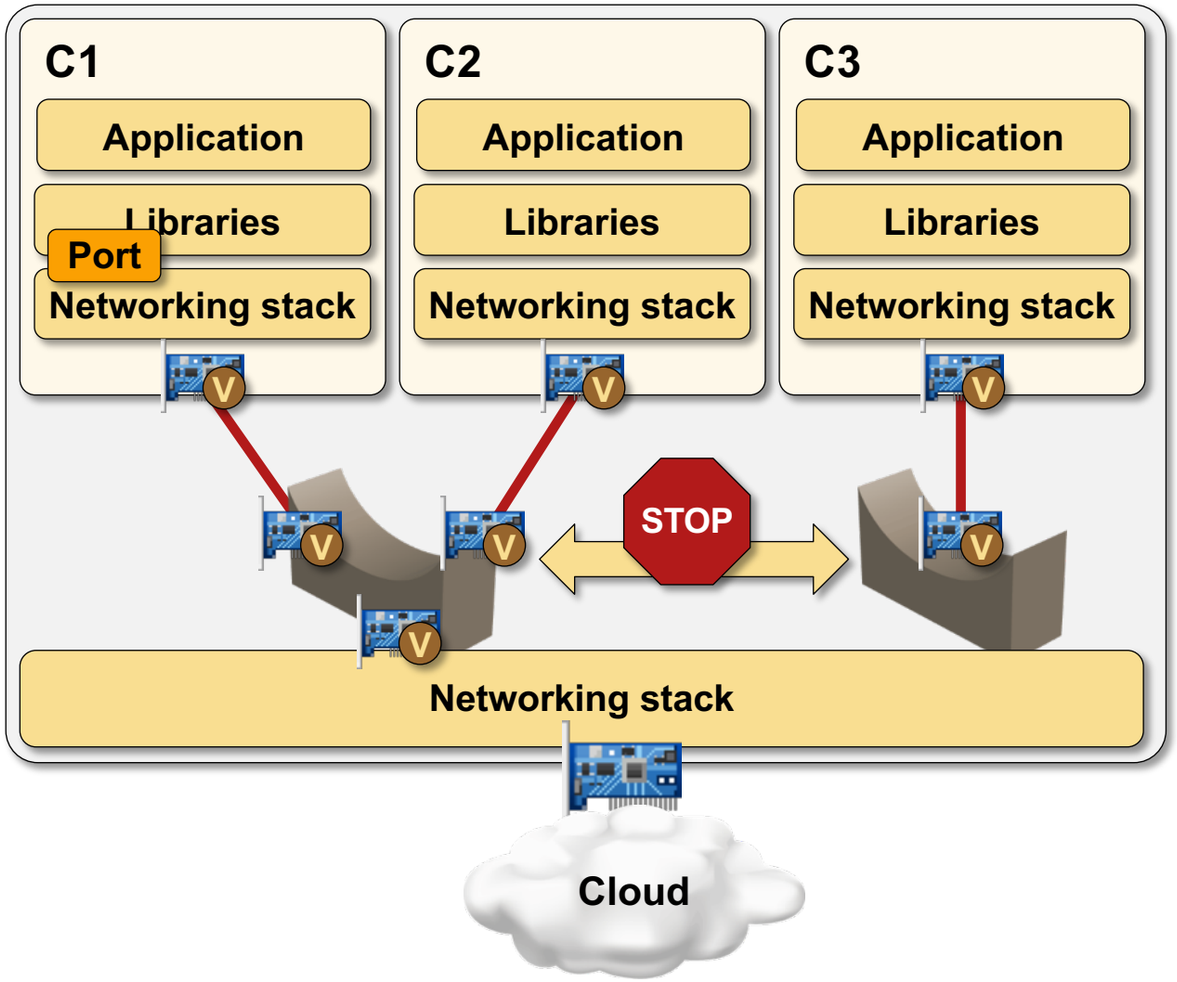
Running a Container with an Exposed Port (Network Service)

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -d --name app \
  --network=br0 webapp
$ docker run -it --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```



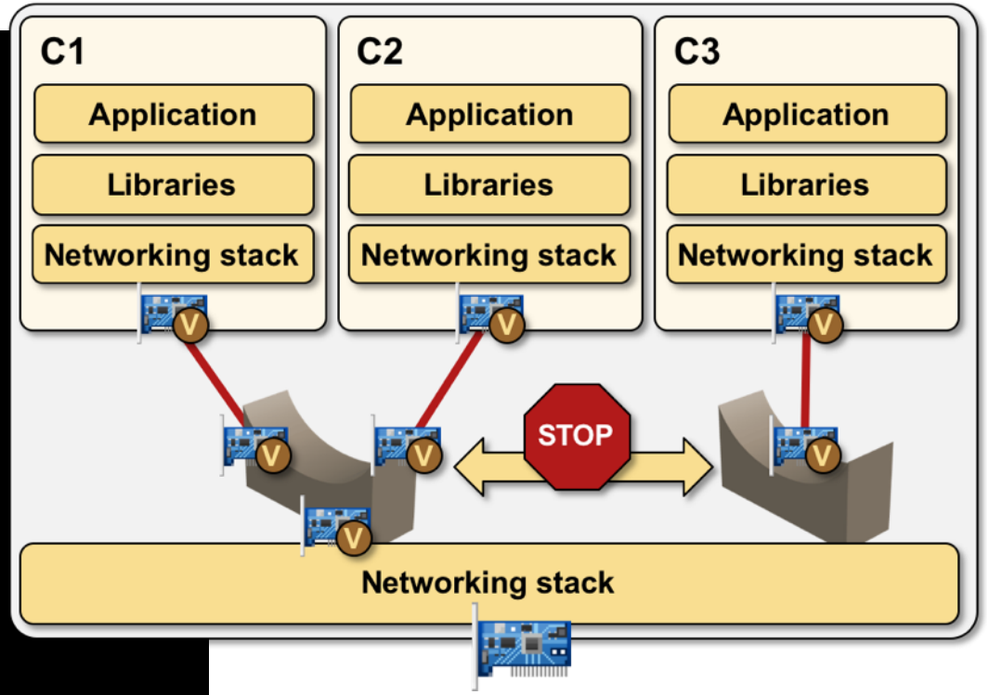
Running a Container with an Exposed Port

Reachable from	
Containers connected to the same network	
Containers connected to other networks	✗
Host processes	
External network	✗



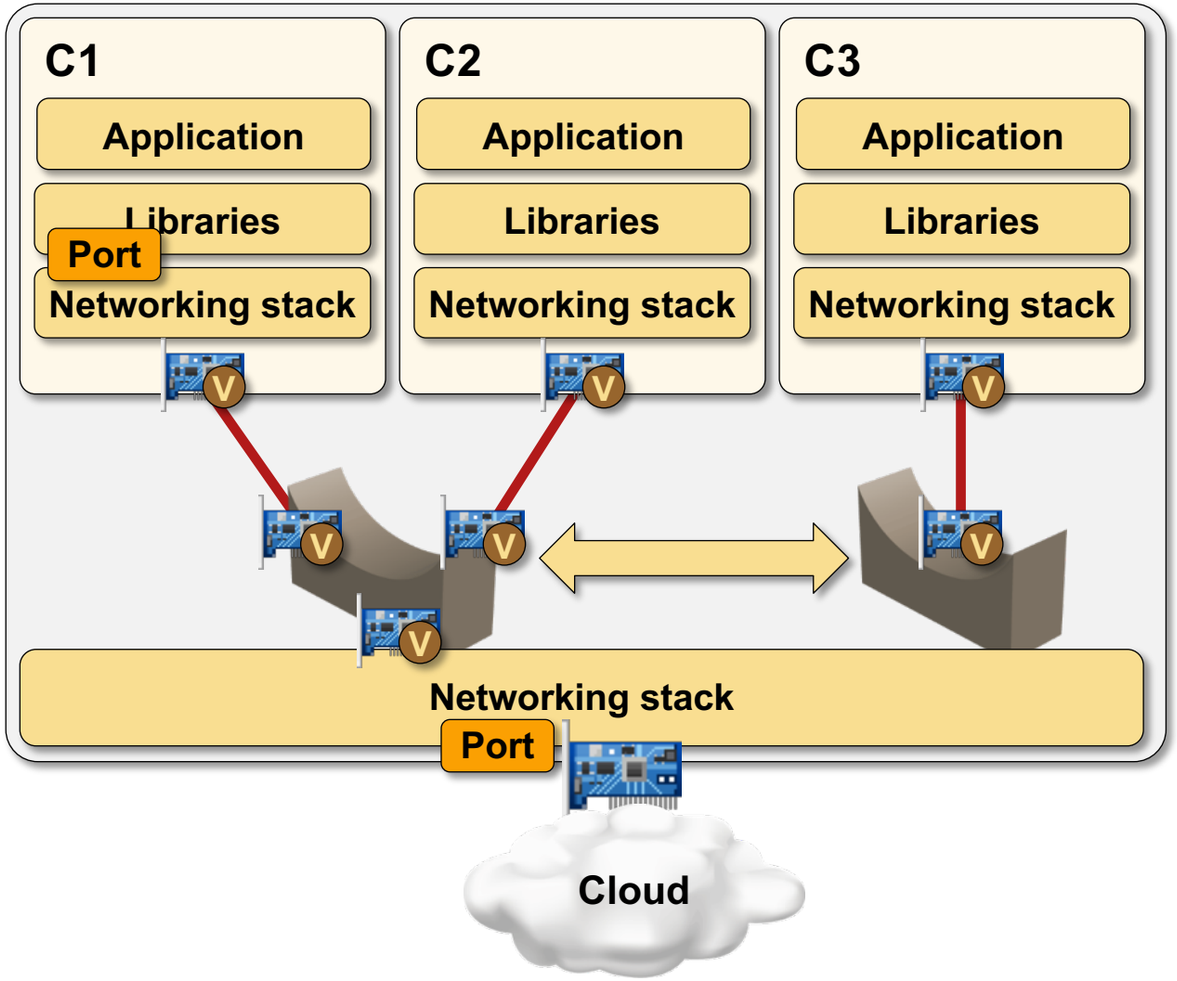
How Does It Work?

```
~ $ sudo iptables -S
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o br-2816c8c2923c -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o br-2816c8c2923c -j DOCKER
-A FORWARD -i br-2816c8c2923c ! -o br-2816c8c2923c -j ACCEPT
-A FORWARD -i br-2816c8c2923c -o br-2816c8c2923c -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A DOCKER-ISOLATION-STAGE-1 -i br-2816c8c2923c ! -o br-2816c8c2923c -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o br-2816c8c2923c -j DROP
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
```



Publishing a Container Port

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -d -p 4000:80 --name app \
  --network=br0 webapp
$ docker run -it --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```

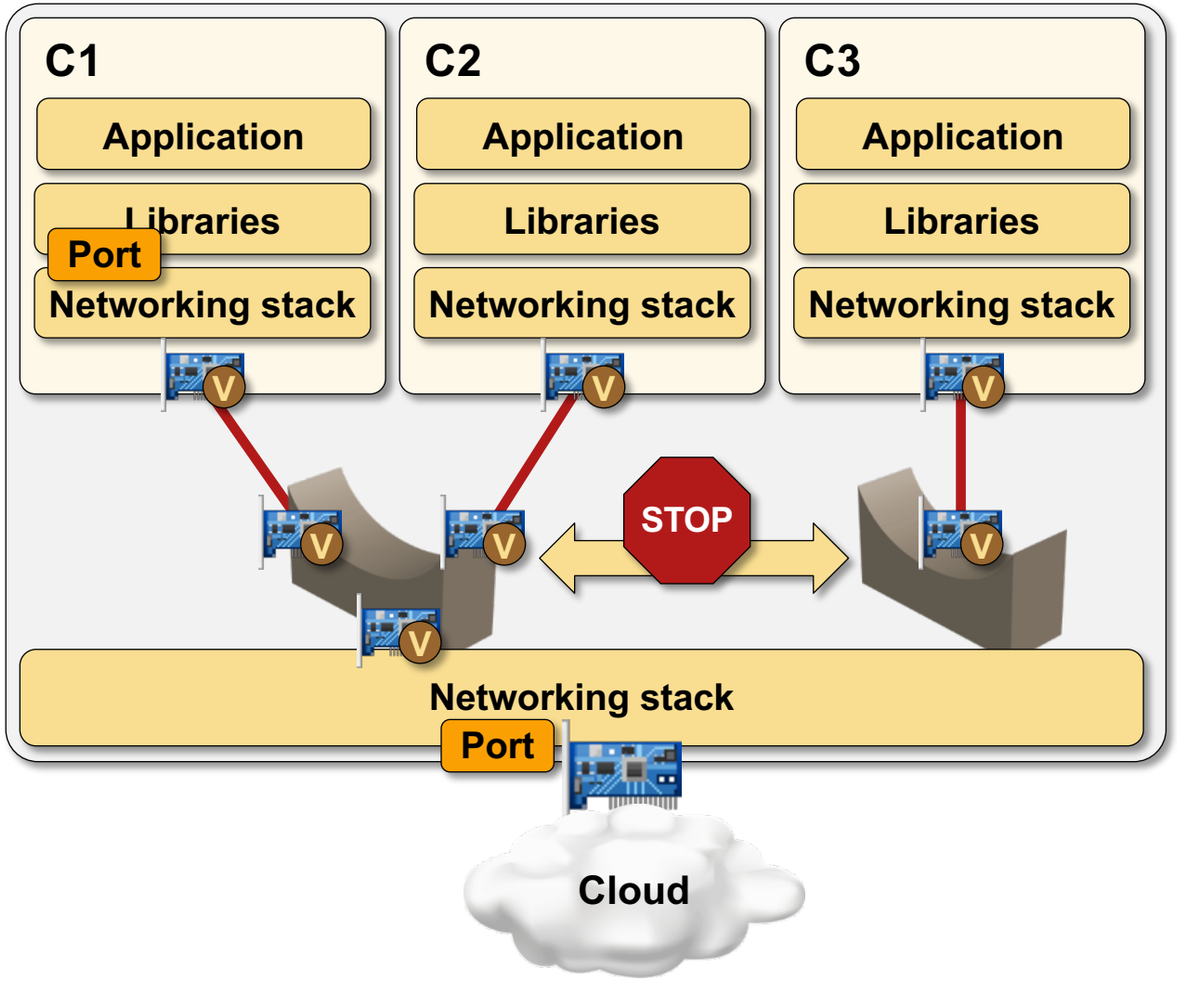


Demo: Publishing a Container Port

Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

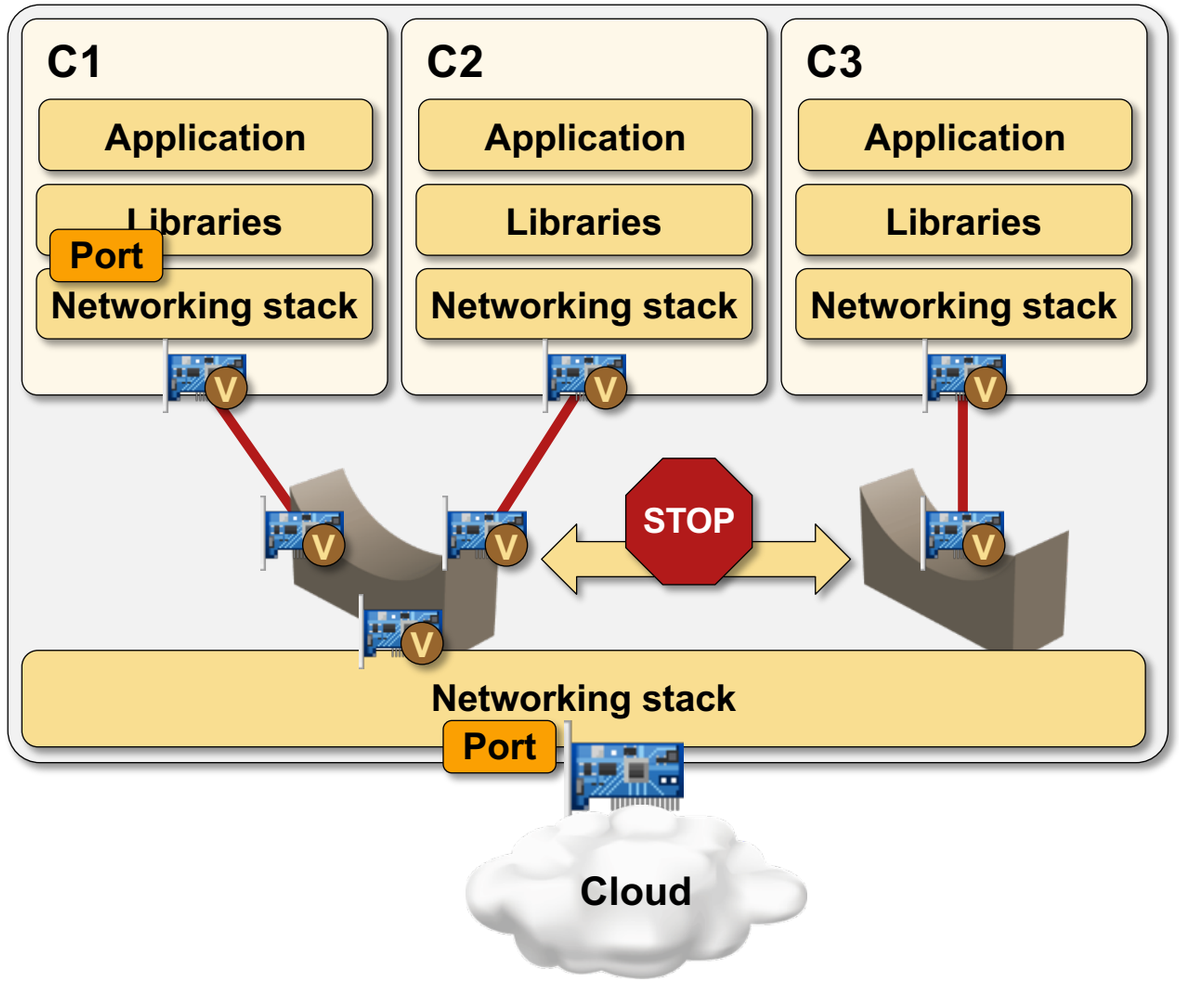
Publishing a Container Port

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -d -p 4000:80 --name app \
  --network=br0 webapp
$ docker run -it --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```



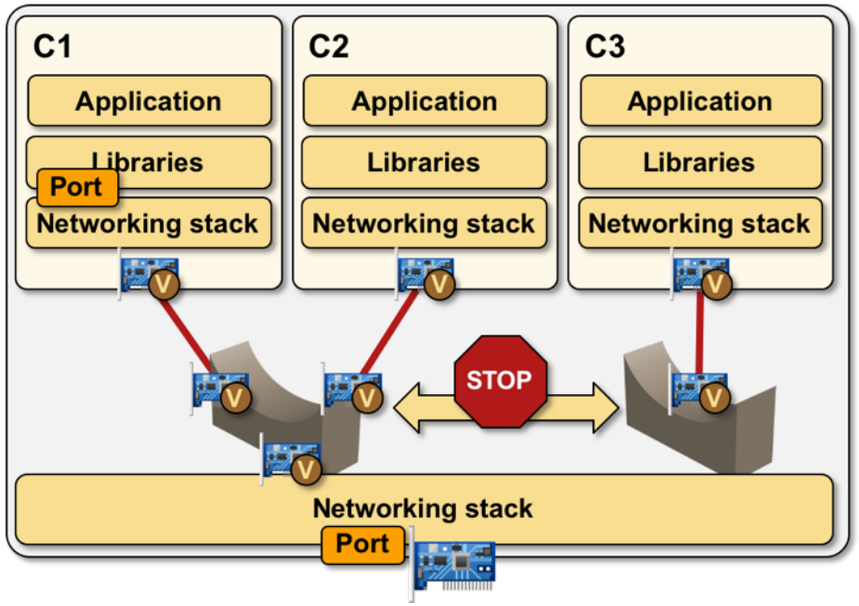
Running a Container with a Published Port

Reachable from	On container IP + port	On host IP + exposed port
Containers in the same network		
Containers connected to other networks	✗	
Host processes		
External network	✗	



Publishing a Container Port: NAT iptables Rules

```
~ $ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 192.168.0.0/24 ! -o br-87412d1d7b49 -j MASQUERADE
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 172.17.0.2/32 -d 172.17.0.2/32 -p tcp -m tcp --dport 80 -j MASQUERADE
-A DOCKER -i br-87412d1d7b49 -j RETURN
-A DOCKER -i docker0 -j RETURN
-A DOCKER ! -i docker0 -p tcp -m tcp --dport 4000 -j DNAT --to-destination 172.17.0.2:80
~ $
```

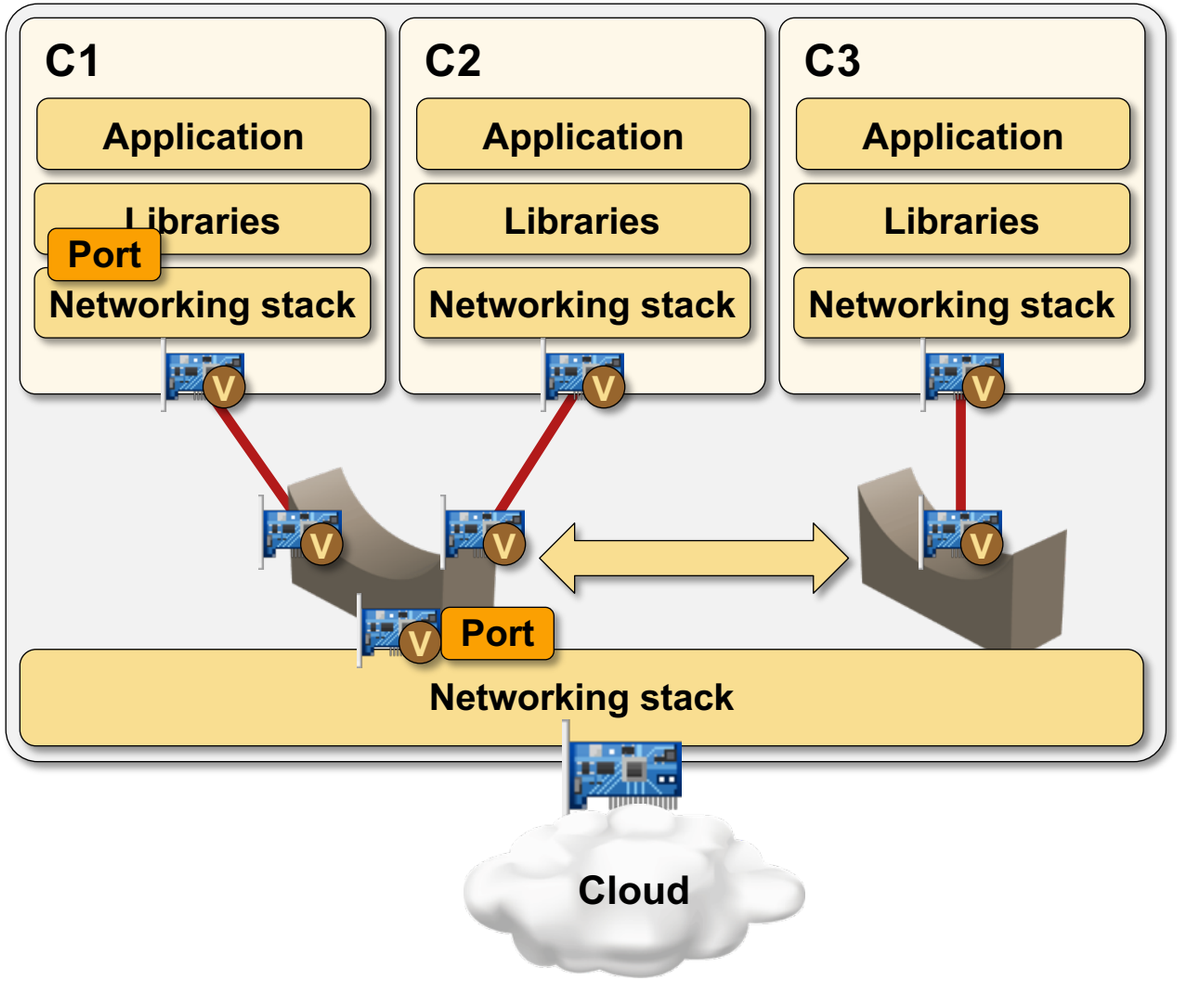


Caveats

- Published port is available on all host IP addresses
- Two containers cannot use the same port (must be solved in the orchestration system)
- Published port is not protected with **INPUT** iptables filter

Binding a Published Port to a Single IP Address

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -d \
  -p 192.168.0.1:4000:80 \
  --name app \
  --network=br0 webapp
$ docker run -it --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```

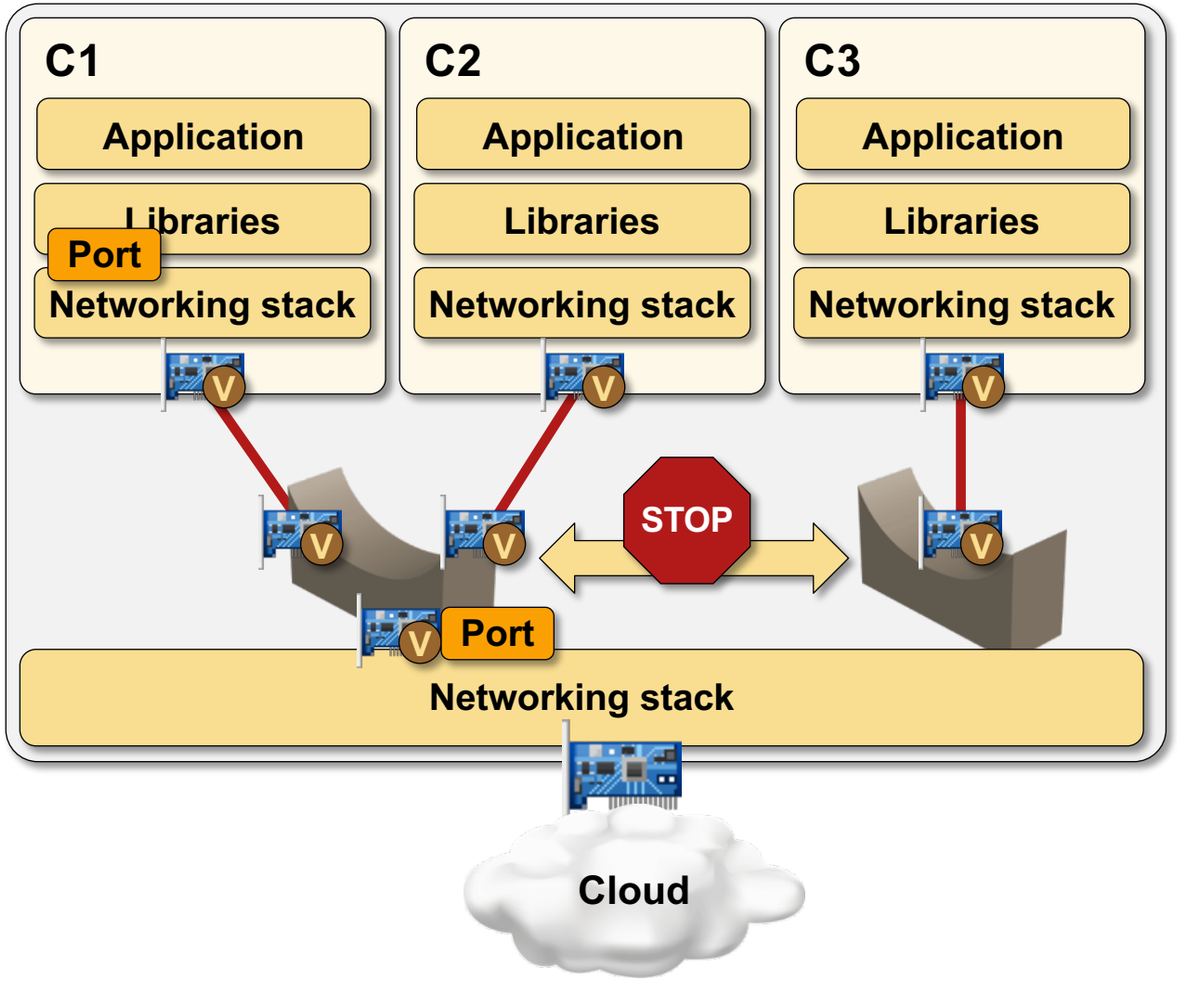


Demo: Binding a Published Port to an IP Address

Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

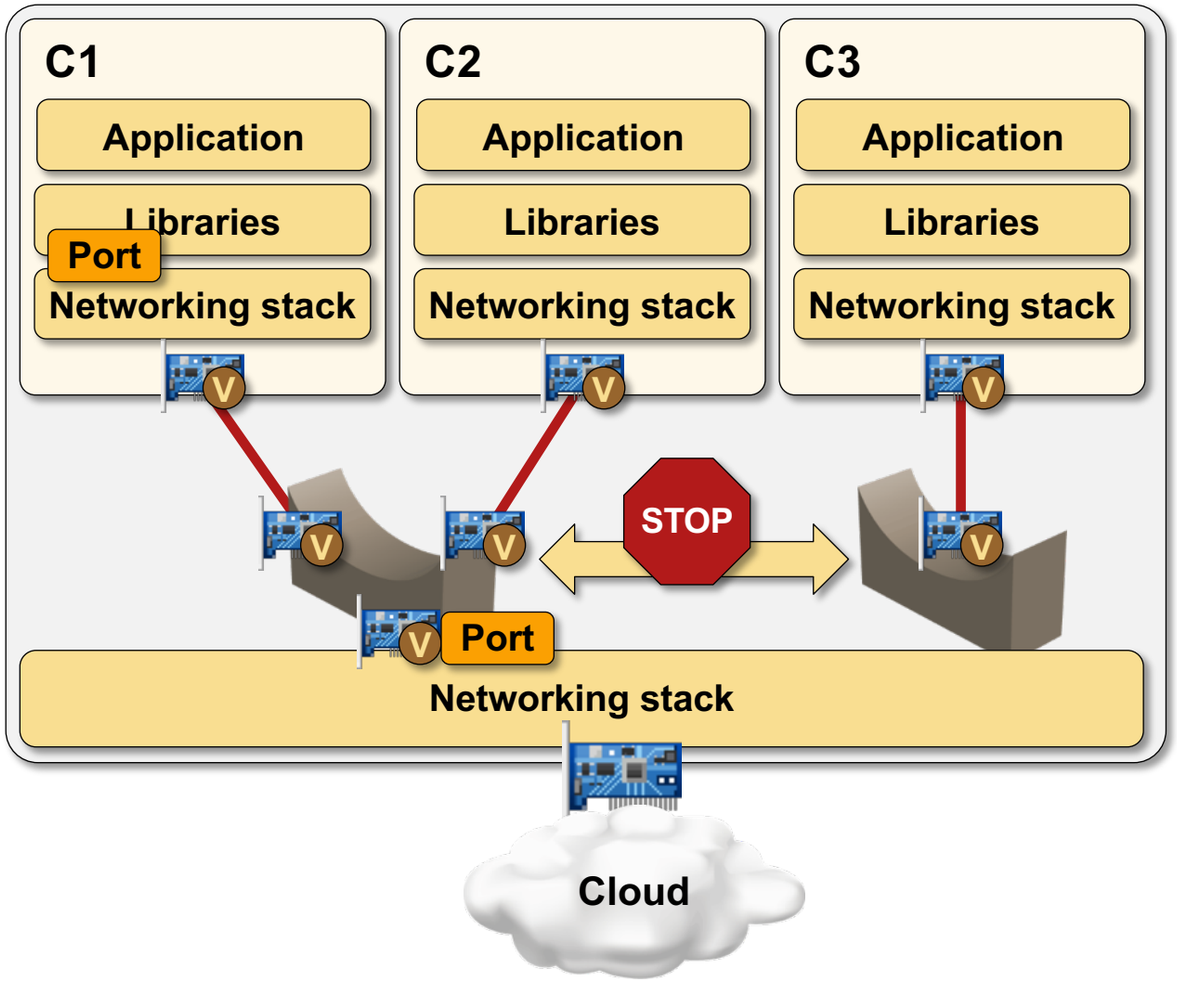
Binding a Published Port to a Single IP Address

```
$ docker network create \
  --driver=bridge \
  --subnet=192.168.99.0/24 br0
$ docker run -d \
  -p 192.168.0.1:4000:80 \
  --name app \
  --network=br0 webapp
$ docker run -it --name c3 busybox
$ docker run -it --name c2 \
  --network=br0 busybox
```



Running a Container with a Published Port Bound to IP Address

Reachable from	On container IP + port	On host IP + exposed port
Containers in the same network		
Containers connected to other networks	✗	
Host processes		
External network	✗	???

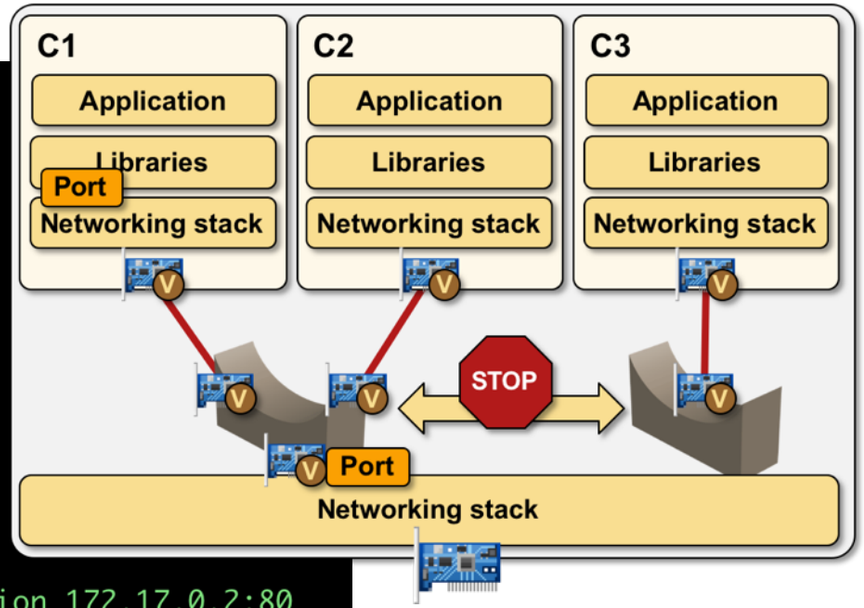


Notes

- External hosts could be able to access the published port (depends on networking setup)
- Published port could be bound to 127.0.0.1 ➔ only host processes would be able to access it

Binding a Container Port to Host IP Address: NAT iptables Rules

```
~ $ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 192.168.0.0/24 ! -o br-87412d1d7b49 -j MASQUERADE
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 172.17.0.2/32 -d 172.17.0.2/32 -p tcp -m tcp --dport 80 -j MASQUERADE
-A DOCKER -i br-87412d1d7b49 -j RETURN
-A DOCKER -i docker0 -j RETURN
-A DOCKER -d 192.168.0.1/32 ! -i docker0 -p tcp -m tcp --dport 4000 -j DNAT --to-destination 172.17.0.2:80
~ $
```



Notes

- Multiple containers could use the same port on different IP addresses
- Published port is not protected with **INPUT** iptables filter

Summary: Docker Networking Using Linux Bridges

Takeaways

- Single-host Docker networks use Linux bridge
- Containers are connected to the default network unless specified otherwise
- Default network allows external access and inter-container communication

You can also

- Create additional networks
- Create completely isolated networks
- Disable inter-container communication and IP masquerading (outgoing NAT)
- Select a host IP address to use for port binding
- Select a different outgoing IP address for IP masquerading (advanced, requires **iptables** manipulation)

Takeaways

- Docker containers can expose service ports
- Other containers attached to the same network can access those services directly
- Docker DNS server performs name resolution on non-default bridged networks

You can also

- Map an exposed container port to a published port on Docker host
- Bind a published port to a specific IP address (or all host IP addresses)
- Docker host performs DNAT from host-IP:published-port to container-ip:exposed-port

Network Security in Docker

Default

- All containers within a network can communicate with one another
- Containers can access host services
- Containers can access outside services using the IP address of the Docker host (NAT masquerading)
- Container services are **not** protected with INPUT iptables rules

Docker security options

- Isolated networks
- Private VLANs
- Binding exposed ports to specified IP address(es)

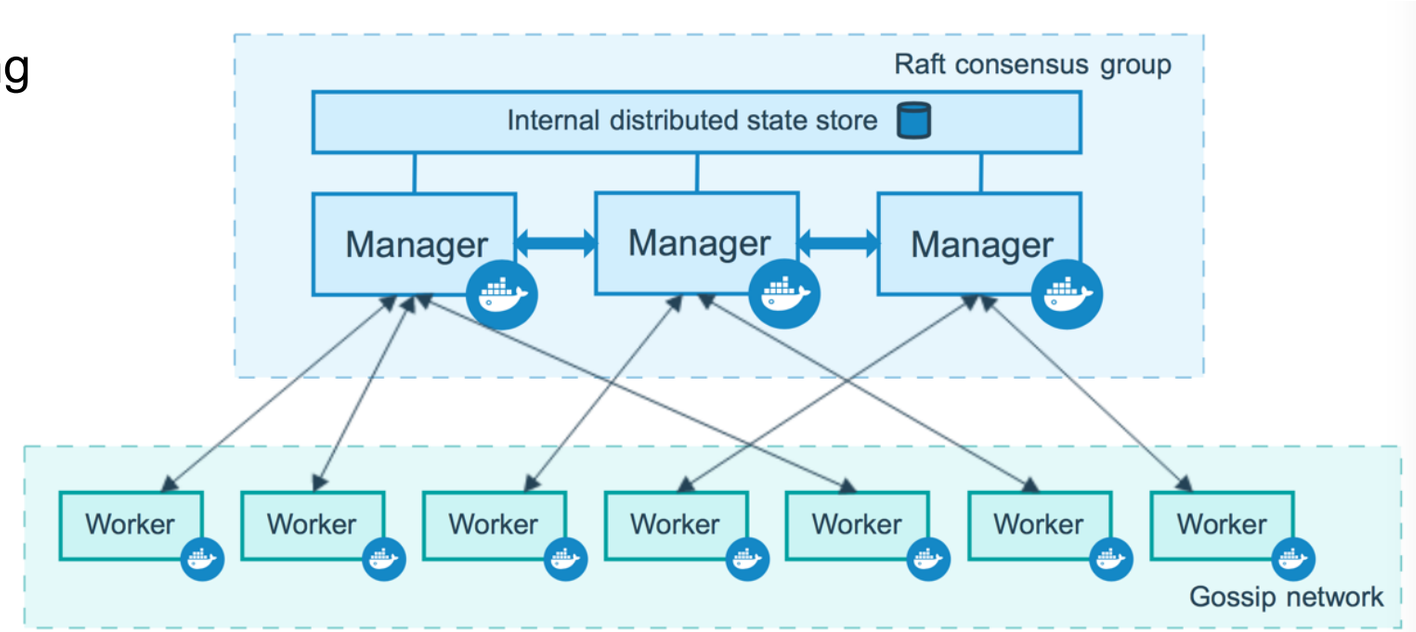
Third-party solutions: Cilium, Trireme, Calico, Contiv…

Docker Swarm Networking

Refresh: Docker Swarm Fundamentals

Docker Swarm = high-availability cluster of Docker hosts

- Cluster-wide virtual networks
- Cluster-wide service or application stack deployment
- Automatic restart of failed containers
- Horizontal service scaling and load balancing
- Service discovery using DNS
  (alternatives: consul, etcd…)
- Service deployment using *compose*-like
  YAML files



More in *Introduction to Docker* webinar

Docker Swarm Control- and Data-Plane Protocols

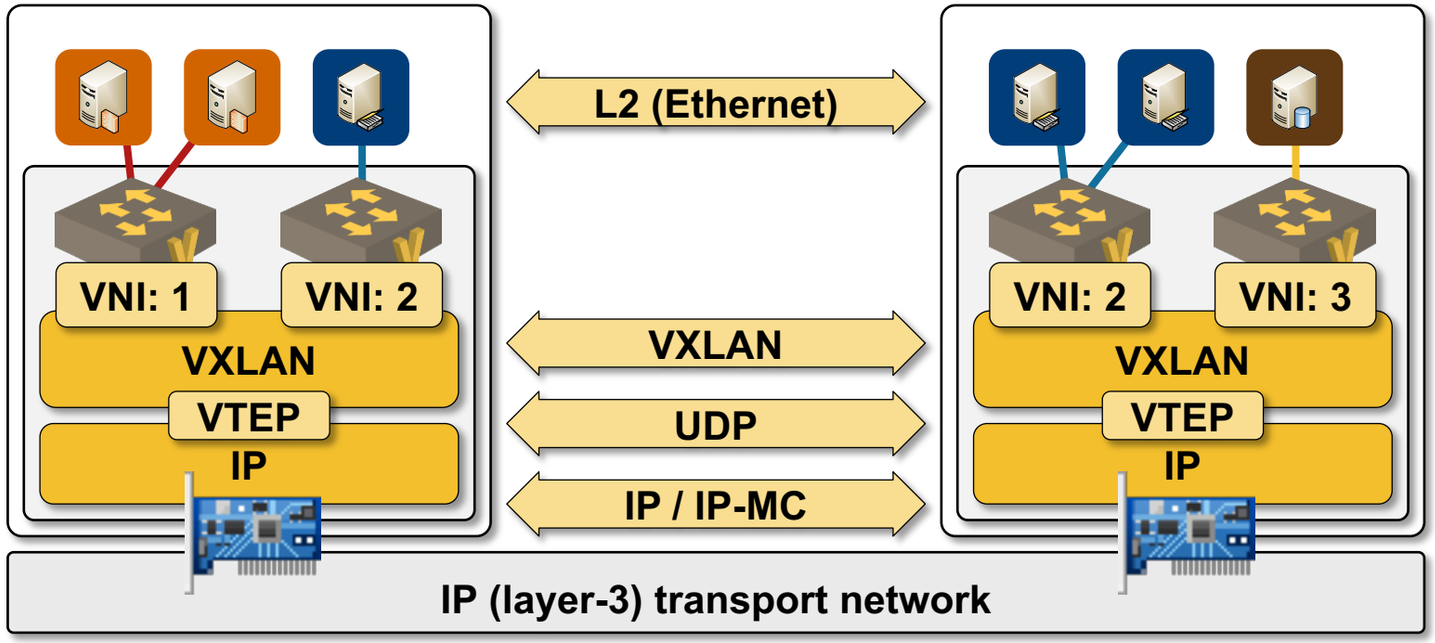
```
~ $ sudo netstat -utlp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:sunrpc          0.0.0.0:*               LISTEN      464/rpcbind
tcp        0      0 localhost:domain        0.0.0.0:*               LISTEN      515/systemd-resolve
tcp        0      0 0.0.0.0:ssh             0.0.0.0:*               LISTEN      857/sshd
tcp6       0      0 [::]:sunrpc             [::]:*                  LISTEN      464/rpcbind
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN      857/sshd
tcp6       0      0 [::]:2377               [::]:*                  LISTEN      838/dockerd
tcp6       0      0 [::]:7946               [::]:*                  LISTEN      838/dockerd
udp        0      0 0.0.0.0:sunrpc          0.0.0.0:*                           464/rpcbind
udp        0      0 0.0.0.0:640             0.0.0.0:*                           464/rpcbind
udp        0      0 0.0.0.0:4789            0.0.0.0:*                           -
udp        0      0 localhost:domain        0.0.0.0:*                           515/systemd-resolve
udp        0      0 manager:bootpc          0.0.0.0:*                           1659/systemd-networ
udp6       0      0 [::]:sunrpc             [::]:*                              464/rpcbind
udp6       0      0 [::]:640                [::]:*                              464/rpcbind
udp6       0      0 [::]:7946               [::]:*                              838/dockerd
```

TCP 2377 – RPC interface for Docker Swarm (IANA registration)

TCP/UDP 7946 – Gossip protocol (Docker-specific port)

UDP 4789 – VXLAN

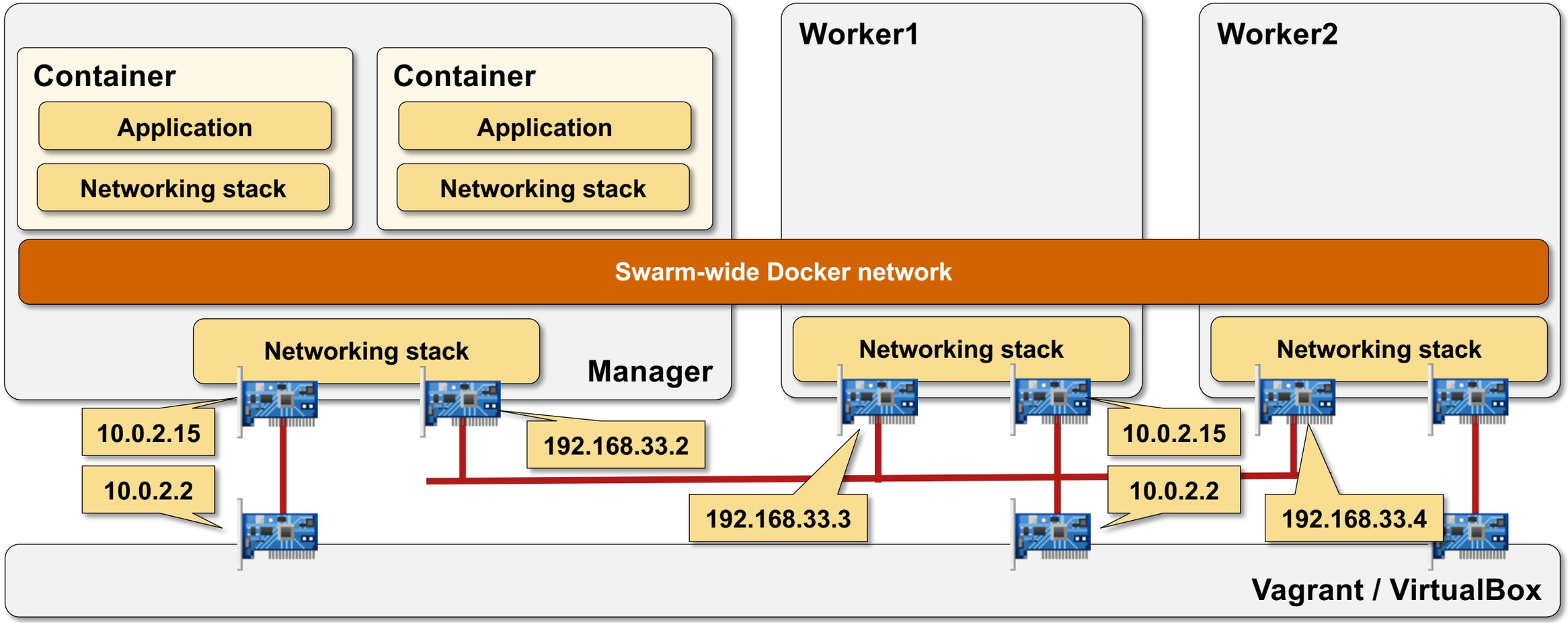
Refresh: Swarm-Wide Networks



Docker Swarm uses overlay driver to build virtual cluster-wide L2/L3 networks

- VXLAN encapsulation
- Container-MAC to VTEP mapping with Gossip protocol
- Optional IPsec/AES encryption (keys managed by Swarm managers)

Demo Swarm Setup



Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

Default Docker Swarm Networking

Docker Networks Created on Swarm Initialization

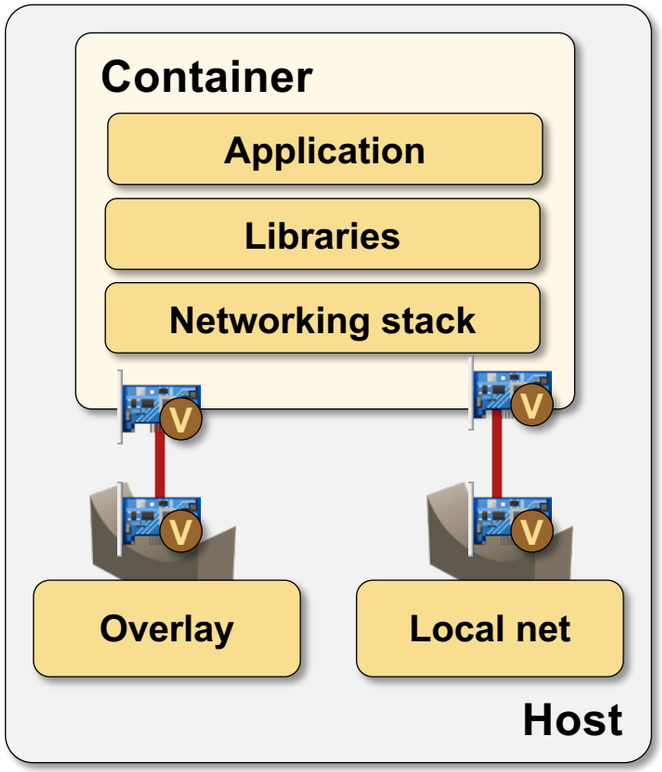
```
~ $ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
222343ddec1d        bridge              bridge              local
ebd51b780273        host                host                local
e25f3b41fb15        none                null                local
~ $ docker swarm init --advertise-addr 192.168.33.2
Swarm initialized: current node (n7kxqx6c1t1phll8qv0ue29by) is now a manager.
```

```
~ $ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
222343ddec1d        bridge              bridge              local
ebc800a6a81f        docker_gwbridge     bridge              local
ebd51b780273        host                host                local
kyr5v5lidnwr        ingress             overlay             swarm
e25f3b41fb15        none                null                local
~ $
```

- **ingress** – network connecting Swarm services (including ingress load balancer) across Swarm members
- **docker_gwbridge** – connects containers using overlay networks to the local host

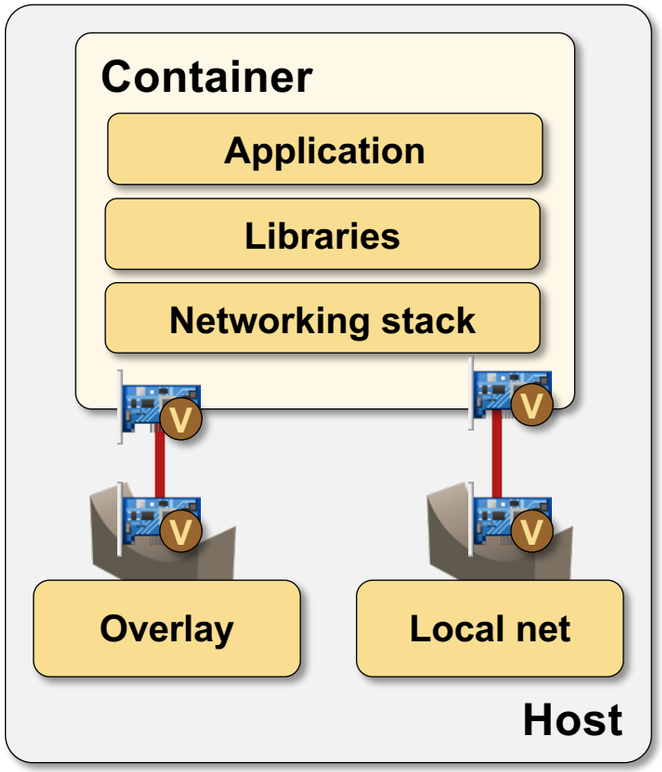
Containers Connected to Swarm Networks Use docker_gwbridge

```
~ $ docker network create --driver=overlay --subnet=192.168.1.0/24 --attachable ov0
1agoctdzy7yzucmq7s2tdeu5h
~ $ docker run --rm --network ov0 busybox ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:C0:A8:01:02
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:15:00:03
          inet addr:172.21.0.3  Bcast:172.21.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:90 (90.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
```



Using docker_gwbridge

- Every container connected to a Swarm network is also connected to **docker_gwbridge** – a Docker network using Linux bridge with disabled inter-container communication
- The connection to **docker_gwbridge** is not included in **docker inspect** data
- The in-container default route points to **docker_gwbridge**, enabling communication with the outside world
- You can disconnect a container from **docker_gwbridge** with **docker network disconnect** command (expect weird results)

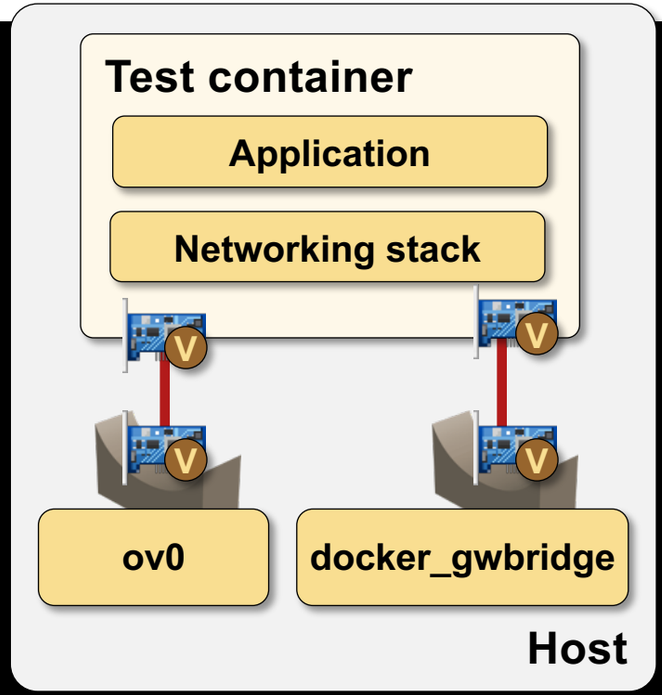


Modifying docker_gwbridge parameters

- Docker Swarm needs a **bridge** network named **docker_gwbridge**
- You can create that network *before starting/joining Swarm* and change any parameters you wish (subnet, **bridge** driver settings…)

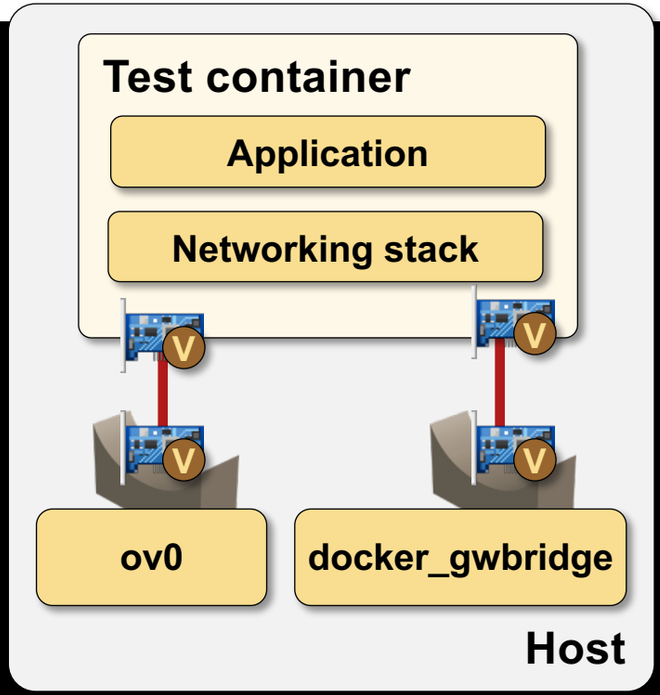
Container Networks Displayed By Docker Inspect Command

```
$ docker run -itd --rm --network ov0 --name test alpine
9d1edc03b558b7e2bffa08ab4881051370bc21d03e513772c4cdbca5d7e2af6a
$ docker inspect test|jq '.[]|.NetworkSettings.Networks'
{
  "ov0": {
    "IPAMConfig": {
      "IPv4Address": "192.168.1.14"
    },
    "Links": null,
    "Aliases": [
      "9d1edc03b558"
    ],
    "NetworkID": "3e0ykh53lm910bvkd3l0phiku",
    "EndpointID": "516b809444af4bce9111ad7060c8da2c5620b111ad70d076b386411ce2a221a9",
    "Gateway": "",
    "IPAddress": "192.168.1.14",
    "IPPrefixLen": 24,
    "IPv6Gateway": "",
    "GlobalIPv6Address": "",
    "GlobalIPv6PrefixLen": 0,
    "MacAddress": "02:42:c0:a8:01:0e",
    "DriverOpts": null
  }
}
$
```



Connections to docker_gwbridge Network

```
$ docker network inspect docker_gwbridge|jq ".[].Containers"
{
  "9d1edc03b558b7e2bffa08ab4881051370bc21d03e513772c4cdbca5d7e2af6a": {
    "Name": "gateway_28ad45dd1bef",
    "EndpointID": "944603db6251a39f79629e26186ecce61b58bf728fa5ba6ac2941dea33536d86",
    "MacAddress": "02:42:ac:12:00:05",
    "IPv4Address": "172.18.0.5/16",
    "IPv6Address": ""
  },
  "ep-137028609f9ed06e14abb7aed4cada02217197d4012ff1fadcfaee27e99e2d04": {
    "Name": "gateway_253e745a506b",
    "EndpointID": "137028609f9ed06e14abb7aed4cada02217197d4012ff1fadcfaee27e99e2d04",
    "MacAddress": "02:42:ac:12:00:04",
    "IPv4Address": "172.18.0.4/16",
    "IPv6Address": ""
  },
  "ep-36447bd0a13456b2f724a98ac56e96a7b44b6e542c83a47cc5a86df0e9651759": {
    "Name": "gateway_9d2f2fa911a6",
    "EndpointID": "36447bd0a13456b2f724a98ac56e96a7b44b6e542c83a47cc5a86df0e9651759",
    "MacAddress": "02:42:ac:12:00:03",
    "IPv4Address": "172.18.0.3/16",
    "IPv6Address": ""
  },
  "ingress-sbox": {
    "Name": "gateway_ingress-sbox",
```



Disconnecting a Container from docker_gwbridge

```
$ docker run -itd --rm --network ov0 --name test alpine
fe49936ac596cfef3806ca99d2d8c494c5037c8576a84fe60b9134ab84909e33
$ docker exec test ip route
default via 172.18.0.1 dev eth1
172.18.0.0/16 dev eth1 scope link  src 172.18.0.4
192.168.1.0/24 dev eth0 scope link  src 192.168.1.8
$ docker network disconnect -f docker_gwbridge test
Error response from daemon: container fe49936ac596cfef3806ca99d2d8c494c5037c8576a84fe60b9134ab84909e33 failed to leave network docker_g
wbridge: container fe49936ac596cfef3806ca99d2d8c494c5037c8576a84fe60b9134ab84909e33: endpoint create on GW Network failed: endpoint wit
h name gateway_253e745a506b already exists in network docker_gwbridge
$ docker exec test ip route
192.168.1.0/24 dev eth0 scope link  src 192.168.1.8
$ docker exec test ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
41: eth0@if42: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue state UP
    link/ether 02:42:c0:a8:01:08 brd ff:ff:ff:ff:ff:ff
```

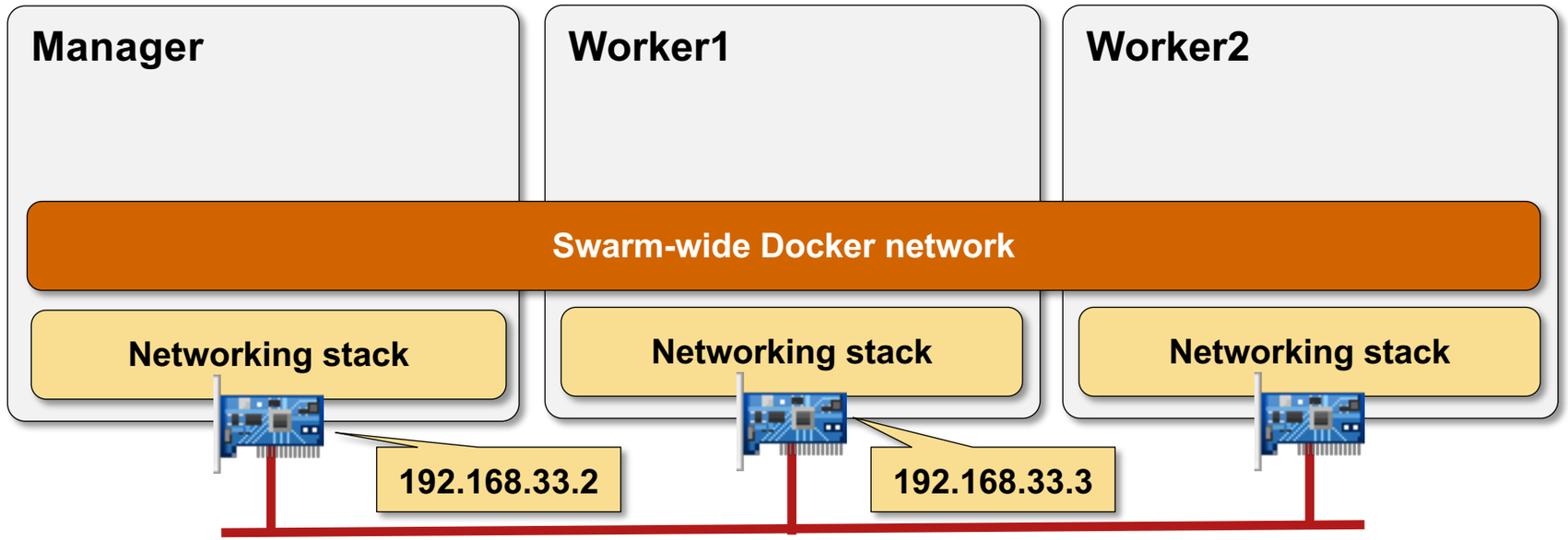
- You can request to disconnect a container from **docker_gwbridge** network
- The API call fails after disconnecting the container
- In a word: Don’t

Swarm Overlay Networks

Creating a Swarm-wide Overlay Network

```
$ docker network create --attachable --driver overlay --subnet 192.168.1.0/24 ov0
```

- Use **attachable** option if you want to connect individual containers to overlay networks
- Default overlay network subnets: a /24 prefix from 10.0.0.0/8 (configurable in /etc/docker/daemon.json)



See https://capstonec.com/2019/10/18/configure-custom-cidr-ranges-in-docker-ee/ for details on changing Docker default prefixes

Overlay Networks Don’t Appear in iptables

```
~ $ sudo iptables -S
-P INPUT ACCEPT
-P FORWARD DROP
-P OUTPUT ACCEPT
-N DOCKER
-N DOCKER-ISOLATION-STAGE-1
-N DOCKER-ISOLATION-STAGE-2
-N DOCKER-USER
-A FORWARD -j DOCKER-USER
-A FORWARD -j DOCKER-ISOLATION-STAGE-1
-A FORWARD -o docker_gwbridge -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker_gwbridge -j DOCKER
-A FORWARD -i docker_gwbridge ! -o docker_gwbridge -j ACCEPT
-A FORWARD -o docker0 -m conntrack --ctstate RELATED,ESTABLISHED -j ACCEPT
-A FORWARD -o docker0 -j DOCKER
-A FORWARD -i docker0 ! -o docker0 -j ACCEPT
-A FORWARD -i docker0 -o docker0 -j ACCEPT
-A FORWARD -i docker_gwbridge -o docker_gwbridge -j DROP
-A DOCKER-ISOLATION-STAGE-1 -i docker_gwbridge ! -o docker_gwbridge -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -i docker0 ! -o docker0 -j DOCKER-ISOLATION-STAGE-2
-A DOCKER-ISOLATION-STAGE-1 -j RETURN
-A DOCKER-ISOLATION-STAGE-2 -o docker_gwbridge -j DROP
-A DOCKER-ISOLATION-STAGE-2 -o docker0 -j DROP
-A DOCKER-ISOLATION-STAGE-2 -j RETURN
-A DOCKER-USER -j RETURN
```

Overlay Networks Are Not Displayed as Linux Bridges

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cbb38666f839        bridge              bridge              local
ebc800a6a81f        docker_gwbridge     bridge              local
ebd51b780273        host                host                local
kyr5v5lidnwr        ingress             overlay             swarm
e25f3b41fb15        none                null                local
3e0ykh53lm91        ov0                 overlay             swarm
$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.02427872ce0f       no
docker_gwbridge         8000.0242019e71e2       no              vethe1686ee
$
```

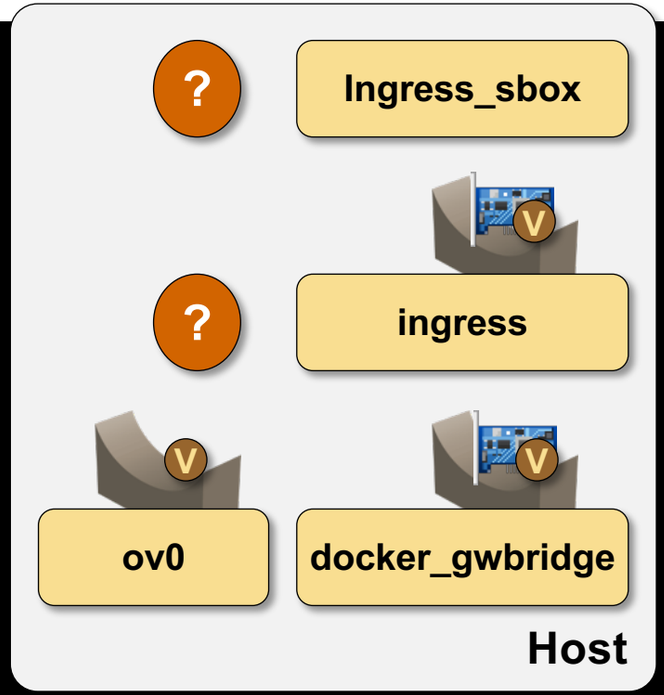
- Overlay Docker Swarm networks don’t appear as Linux bridges
- They behave in exactly the same way as isolated bridge-based Docker networks

Mystery solved:

- Overlay networks are created in dedicated namespaces
- Those namespaces are created in **/var/run/docker/netns** directory (and thus somewhat hard to see)

Each Overlay Network Is in a Separate Namespace

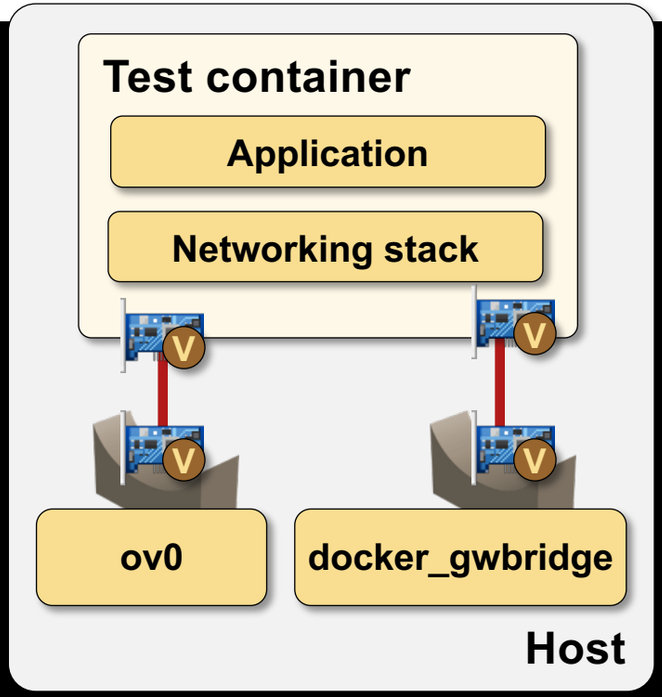
```
$ sudo ln -s /var/run/docker/netns /var/run
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cbb38666f839        bridge              bridge              local
ebc800a6a81f        docker_gwbridge     bridge              local
ebd51b780273        host                host                local
kyr5v5lidnwr        ingress             overlay             swarm
e25f3b41fb15        none                null                local
3e0ykh53lm91        ov0                 overlay             swarm
$ sudo ip netns
1-kyr5v5lidn (id: 0)
ingress_sbox (id: 1)
$ sudo ip netns exec ingress_sbox brctl show
bridge name     bridge id               STP enabled     interfaces
$ sudo ip netns exec 1-kyr5v5lidn brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.2aa76be99dac       no              veth0
                                                        vxlan0
```



- Link **/var/run/docker/netns** to **/var/run/netns** to enable **ip netns** to display Docker namespaces
- **ingress_sbox** namespace belongs to ingress load balancer container
- Namespace id 0 belongs to **ingress** overlay network

Overlay Networks (and Namespaces) Are Instantiated When Needed

```
$ docker run -itd --rm --network ov0 --name test alpine
b8ad4bd40cae5b573ce11685153764a461ab370c2264b7e35a1f56441baf5aee
$ sudo ip netns
21381c04f838 (id: 4)
1-3e0ykh53lm (id: 2)
lb_3e0ykh53l (id: 3)
1-kyr5v5lidn (id: 0)
ingress_sbox (id: 1)
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cbb38666f839        bridge              bridge              local
ebc800a6a81f        docker_gwbridge     bridge              local
ebd51b780273        host                host                local
kyr5v5lidnwr        ingress             overlay             swarm
e25f3b41fb15        none                null                local
3e0ykh53lm91        ov0                 overlay             swarm
$ sudo ip netns exec 1-3e0ykh53lm brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.2ed36baab436       no              veth0
                                                        veth1
                                                        vxlan0
```



- Namespace, Linux bridge and VXLAN interface are created when an overlay network is first used
- Docker networks on worker nodes are also instantiated on as-needed basis

Docker Swarm Network – Local Components

```
$ sudo ip netns exec 1-3e0ykh53lm ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 2e:d3:6b:aa:b4:36 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.1/24 brd 192.168.1.255 scope global br0
       valid_lft forever preferred_lft forever
12: vxlan0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UNKNOWN group default
    link/ether 3e:41:64:d0:34:80 brd ff:ff:ff:ff:ff:ff link-netnsid 0
14: veth0@if13: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP group default
    link/ether 3e:e1:92:f5:80:74 brd ff:ff:ff:ff:ff:ff link-netnsid 1
16: veth1@if15: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP group default
    link/ether 2e:d3:6b:aa:b4:36 brd ff:ff:ff:ff:ff:ff link-netnsid 2
$
```

- **br0** - Linux bridge connecting containers with VXLAN transport interface (the same IP address is configured on all Swarm nodes)
- Multi-point VXLAN transport interface
- Virtual Ethernet interfaces connecting containers to overlay network

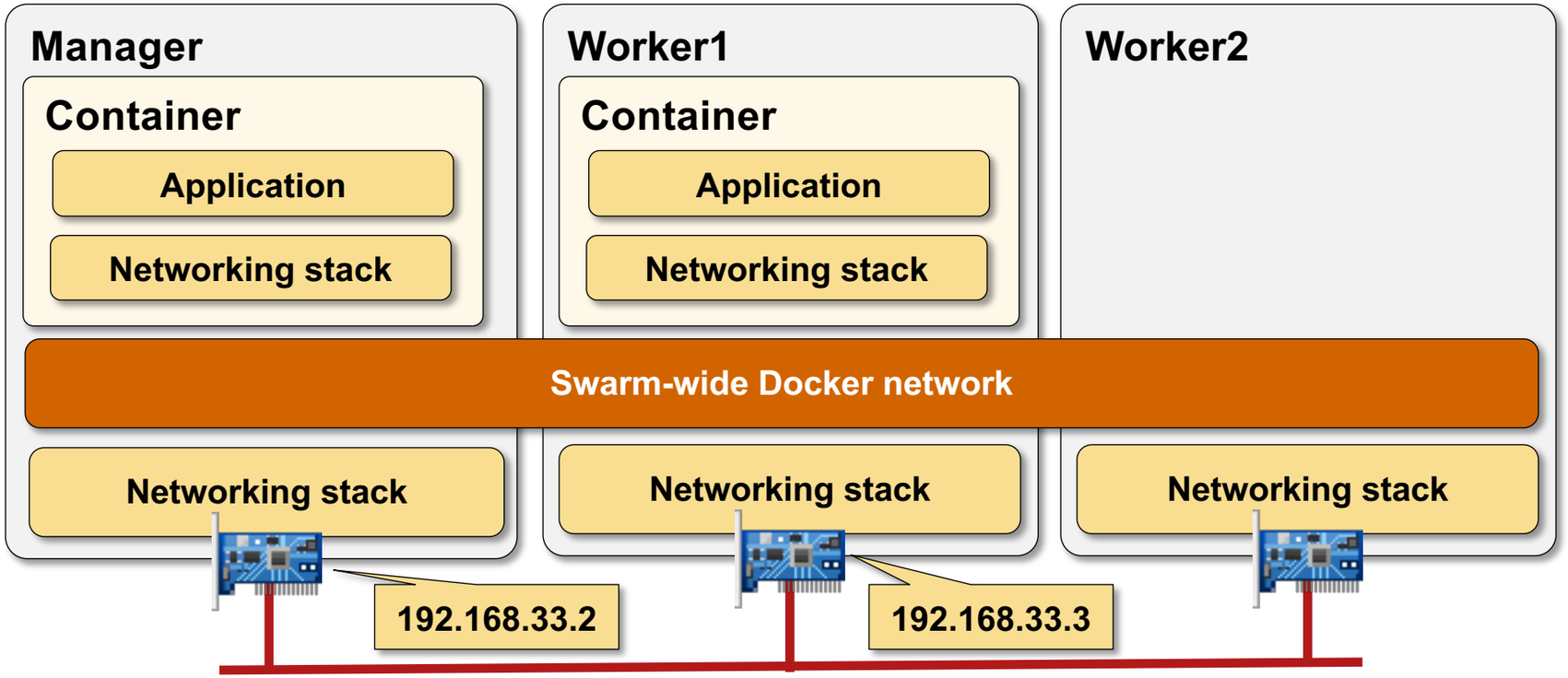
Deploying Containers on Multiple Swarm Hosts

Next steps

- Deploy containers on multiple hosts

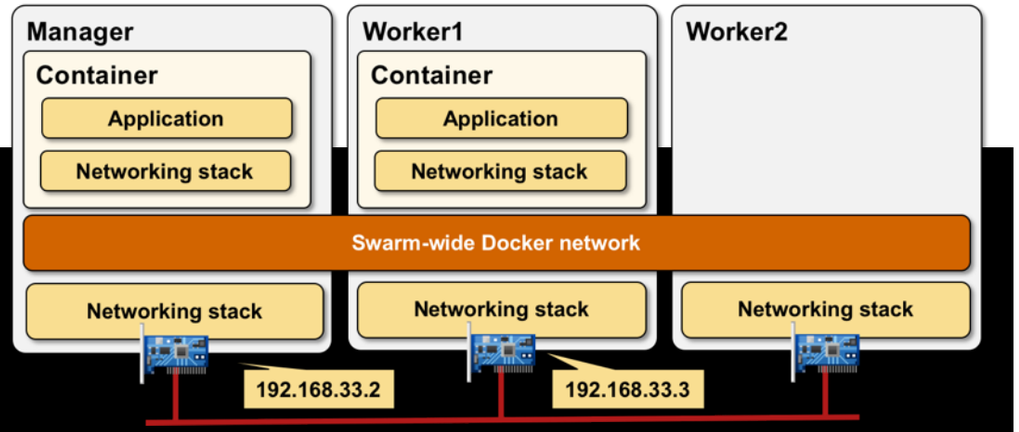
Inspect

- VXLAN interface parameters
- VXLAN (MAC-to-VTEP) mappings
- ARP cache



Source code @ https://github.com/ipspace/docker-examples/tree/master/labs

Docker VXLAN Transport




```
$ sudo ip netns exec 1-3e0ykh53lm bridge fdb show dev vxlan0
3e:41:64:d0:34:80 master br0 permanent
02:42:c0:a8:01:04 dst 192.168.33.3 link-netnsid 0 self permanent
02:42:c0:a8:01:05 dst 192.168.33.3 link-netnsid 0 self permanent
$ sudo ip netns exec 1-3e0ykh53lm ip neighbor
192.168.1.5 dev vxlan0 lladdr 02:42:c0:a8:01:05 PERMANENT
192.168.1.4 dev vxlan0 lladdr 02:42:c0:a8:01:04 PERMANENT
$ sudo ip netns exec 1-3e0ykh53lm ip -d link show vxlan0
12: vxlan0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UNKNOWN mode DEFAULT group default
    link/ether 3e:41:64:d0:34:80 brd ff:ff:ff:ff:ff:ff link-netnsid 0 promiscuity 1
    vxlan id 4097 srcport 0 0 dstport 4789 proxy l2miss l3miss ttl inherit ageing 300 udpcsum noudp6zerocsumtx noudp6zerocsumrx
    bridge_slave state forwarding priority 32 cost 100 hairpin off guard off root_block off fastleave off learning on flood on port_id
0x8001 port_no 0x1 designated_port 32769 designated_cost 0 designated_bridge 8000.2e:d3:6b:aa:b4:36 designated_root 8000.2e:d3:6b:aa:b4
:36 hold_timer    0.00 message_age_timer    0.00 forward_delay_timer    0.00 topology_change_ack 0 config_pending 0 proxy_arp off proxy
_arp_wifi off mcast_router 1 mcast_fast_leave off mcast_flood on neigh_suppress off group_fwd_mask 0x0 group_fwd_mask_str 0x0 vlan_tunn
el off addrgenmode eui64 numtxqueues 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
$
```

- Docker is using unicast VXLAN with statically-configured MAC-to-VTEP mappings and ARP entries
- Proxy ARP is enabled on VXLAN interface – all ARP requests are answered locally
- The kernel asks userland program to populate L2 (FDB) or L3 (ARP) entries when needed

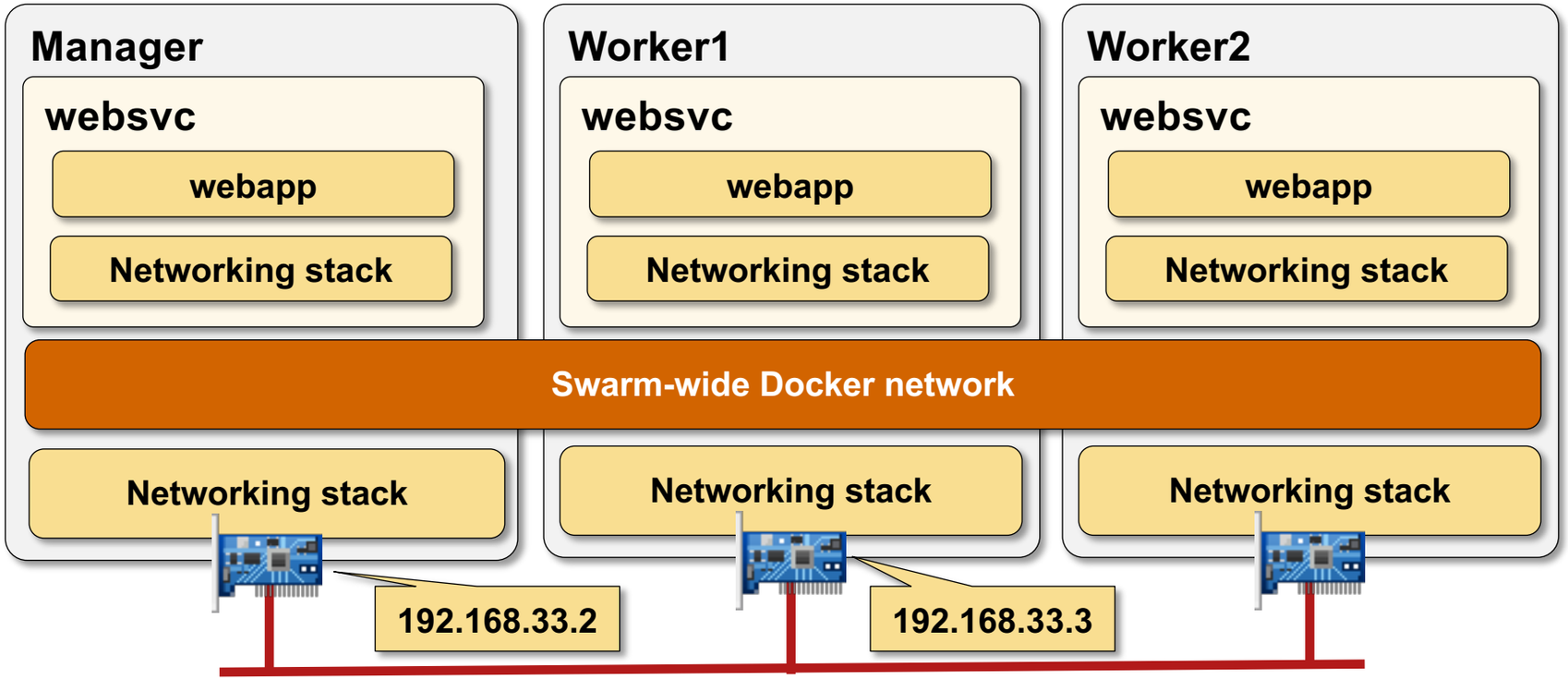
Port Publishing for Standalone Containers

- Standalone containers are connected to overlay and **docker_gwbridge** networks
- Port publishing creates a NAT rule mapping host port to a container port on **docker_gwbridge**

```
$ docker run -itd --rm --network ov0 -p 8080:80 --name test alpine
575c9c78bfcab1160d2988add8874ec2c2d844844ec17752ed363ac9d37e30cb
$ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 172.18.0.0/16 ! -o docker_gwbridge -j MASQUERADE
-A POSTROUTING -s 172.18.0.3/32 -d 172.18.0.3/32 -p tcp -m tcp --dport 80 -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
-A DOCKER -i docker_gwbridge -j RETURN
-A DOCKER ! -i docker_gwbridge -p tcp -m tcp --dport 8080 -j DNAT --to-destination 172.18.0.3:80
$ docker exec test ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 02:42:AC:12:00:03
          inet addr:172.18.0.3  Bcast:172.18.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:796 (796.0 B)  TX bytes:0 (0.0 B)
```

Load Balancing in Docker Swarm Node

Deploying Swarm-Wide Service



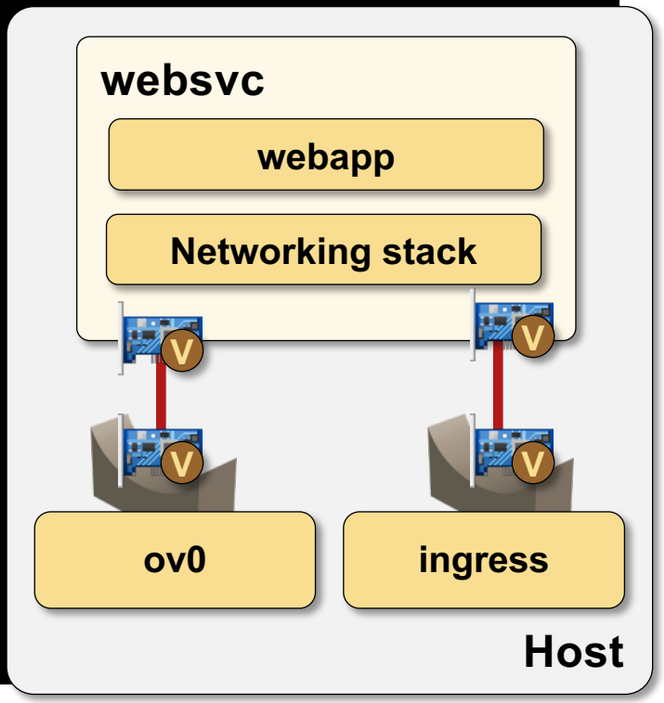
- Deploy Swarm-wide application web server (see *Introduction to Docker* for details) publishing port 80
- Explore port publishing and load balancing in Docker Swarm

Create a Multi-Instance Service

```
$ docker service create --name websvc --network ov0 --publish 8080:80 --replicas 3 webapp
image webapp:latest could not be accessed on a registry to record
its digest. Each node will access webapp:latest independently,
possibly leading to different nodes running different
versions of the image.

gq0dac81d1xzxoxx792n83sne
overall progress: 3 out of 3 tasks
1/3: running   [==================================================>]
2/3: running   [==================================================>]
3/3: running   [==================================================>]
verify: Service converged
$ docker service ls
ID                  NAME                MODE                REPLICAS            IMAGE               PORTS
gq0dac81d1xz        websvc              replicated          3/3                 webapp:latest       *:8080->80/tcp
$ docker service ps websvc
ID                  NAME                IMAGE               NODE                DESIRED STATE       CURRENT STATE                ERROR
              PORTS
lpsn4y3g1ct9        websvc.1            webapp:latest       worker2             Running             Running about a minute ago

le6d644qp2zf        websvc.2            webapp:latest       manager             Running             Running about a minute ago

u2k3cd58sl9g        websvc.3            webapp:latest       worker1             Running             Running about a minute ago

$
```

Service Containers Are Attached to Ingress Docker Network

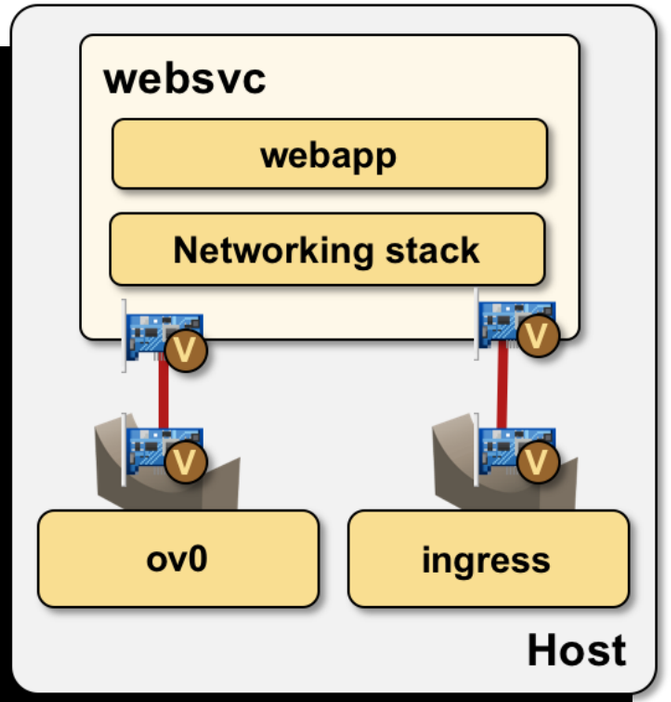
```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
3f24944d4757        webapp:latest       "python app.py"     7 minutes ago       Up 7 minutes        80/tcp              websvc.2.le6d64
4qp2zfd7nu7d0ybgtc5
$ docker inspect $(docker ps -q)|jq -f /vagrant/filter/ipaddr
{
  "Name": "/websvc.2.le6d644qp2zfd7nu7d0ybgtc5",
  "Network": {
    "net": "ingress",
    "ip": "10.0.0.8",
    "gw": ""
  }
}
{
  "Name": "/websvc.2.le6d644qp2zfd7nu7d0ybgtc5",
  "Network": {
    "net": "ov0",
    "ip": "192.168.1.12",
    "gw": ""
  }
}
```



- Containers running as part of a service are connected to **ingress** network (and an overlay network)
- **Ingress** network handles incoming connections to published service ports

Service Containers Are Also Attached to docker_gwbridge

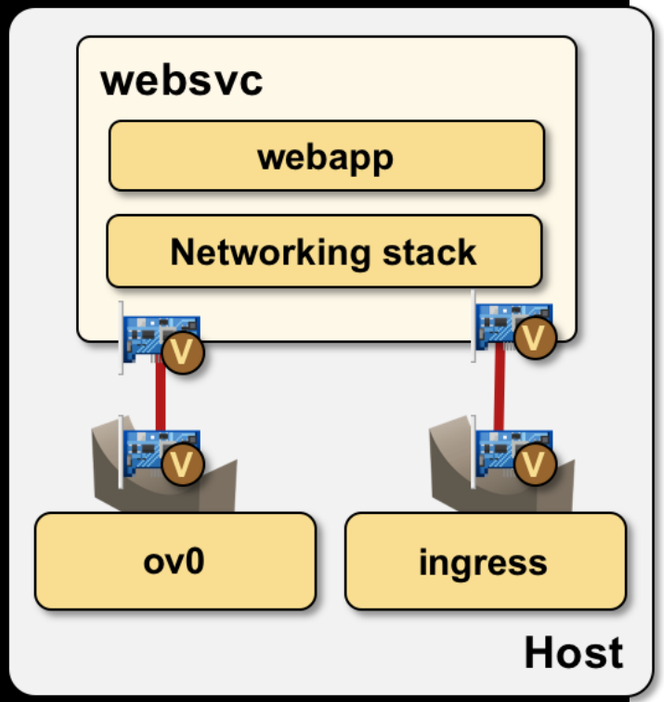
```
$ sudo ip netns exec fc8de92089a5 ip address show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
37: eth1@if38: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:c0:a8:01:0c brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet 192.168.1.12/24 brd 192.168.1.255 scope global eth1
       valid_lft forever preferred_lft forever
39: eth2@if40: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff link-netnsid 2
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth2
       valid_lft forever preferred_lft forever
41: eth0@if42: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:08 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.8/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
$
```



- Similar to standalone containers, service containers connect to **docker_gwbridge** network
- The **docker_gwbridge** network is used for outbound connections, and does not appear in **docker inspect**

Service Containers Are Also Attached to docker_gwbridge (2)

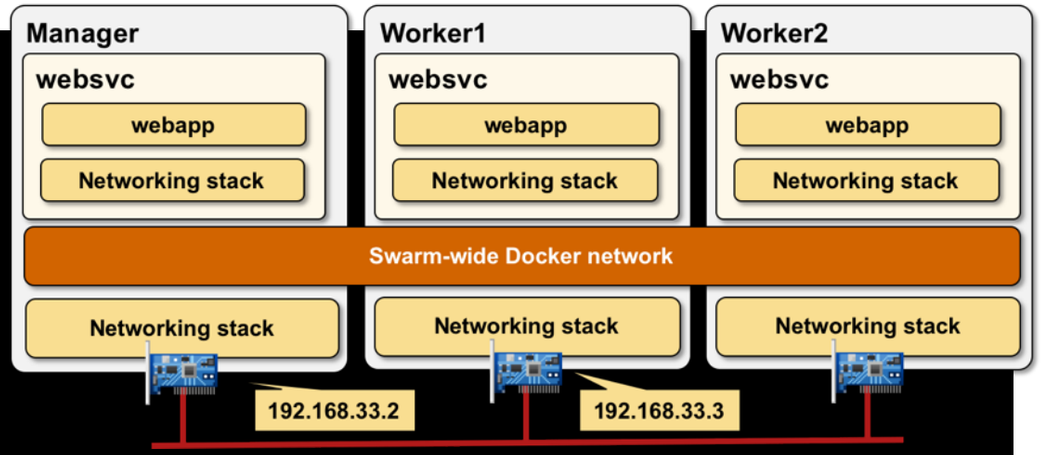
```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
3f24944d4757        webapp:latest       "python app.py"     2 hours ago         Up 2 hours          80/tcp              websvc.2.le6d64
4qp2zfd7nu7d0ybgtc5
$ docker network inspect docker_gwbridge|jq '.[]|.Containers'
{
  "3f24944d475741d5455554dd5284ab018cdb84267bbe9b5b46ed0f87616e350b": {
    "Name": "gateway_fc8de92089a5",
    "EndpointID": "a4b82d62006eb35f0a8d05dc3f8c684d736136e89c56dc645ed476fddd018bf6",
    "MacAddress": "02:42:ac:12:00:03",
    "IPv4Address": "172.18.0.3/16",
    "IPv6Address": ""
  },
  "ingress-sbox": {
    "Name": "gateway_ingress-sbox",
    "EndpointID": "1acfc9348b147c07228483d120f7530878a63b78fad514a573a559432e0a305b",
    "MacAddress": "02:42:ac:12:00:02",
    "IPv4Address": "172.18.0.2/16",
    "IPv6Address": ""
  }
}
$
```



You can see the hidden container attachment with docker network inspect command

NAT Table Translates Published Port to a Port on docker_gwbridge

```
$ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER
-N DOCKER-INGRESS
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER-INGRESS
-A PREROUTING -m addrtype --dst-type LOCAL -j DOCKER
-A OUTPUT -m addrtype --dst-type LOCAL -j DOCKER-INGRESS
-A OUTPUT ! -d 127.0.0.0/8 -m addrtype --dst-type LOCAL -j DOCKER
-A POSTROUTING -o docker_gwbridge -m addrtype --src-type LOCAL -j MASQUERADE
-A POSTROUTING -s 172.17.0.0/16 ! -o docker0 -j MASQUERADE
-A POSTROUTING -s 172.18.0.0/16 ! -o docker_gwbridge -j MASQUERADE
-A DOCKER -i docker0 -j RETURN
-A DOCKER -i docker_gwbridge -j RETURN
-A DOCKER-INGRESS -p tcp -m tcp --dport 8080 -j DNAT --to-destination 172.18.0.2:8080
-A DOCKER-INGRESS -j RETURN
```



- NAT rules are mapping published service port to an IP address on **docker_gwbridge** network
- The destination IP address is not a container IP address

NAT Table Maps Service Port to ingress_sbox

```
$ sudo iptables -t nat -S
-P PREROUTING ACCEPT
-A DOCKER -i docker_gwbridge -j RETURN
-A DOCKER-INGRESS -p tcp -m tcp --dport 8080 -j DNAT --to-destination 172.18.0.2:8080
-A DOCKER-INGRESS -j RETURN
```

```
$ docker network inspect docker_gwbridge|jq -f /vagrant/filter/netaddr
{
  "Name": "docker_gwbridge",
  "Gateway": "172.18.0.1",
  "Endpoints": [
    {
      "Name": "gateway_fc8de92089a5",
      "IPv4": "172.18.0.3/16"
    },
    {
      "Name": "gateway_ingress-sbox",
      "IPv4": "172.18.0.2/16"
    }
  ]
}
```

What exactly is ingress-sbox?

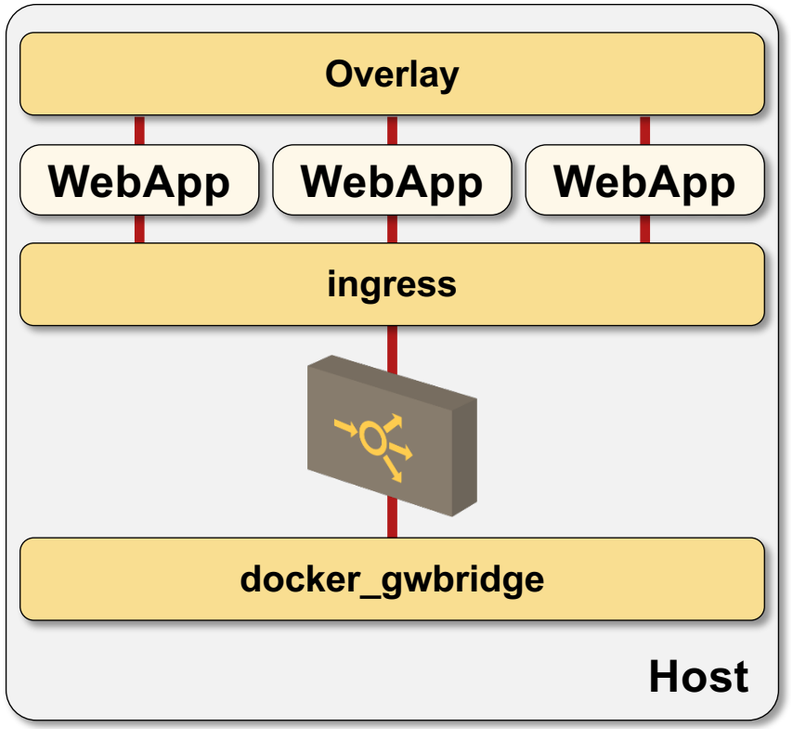
Load Balancing in Docker Swarm Node

Containers started as part of a service connect to:

- Overlay virtual network (if specified)
- **ingress** overlay virtual network
- **docker_gwbridge** network (for outbound connections)

Inbound connections

- Host TCP stack maps published port to port on **ingress_sbox** container
- **ingress_sbox** container is a simple load balancer that maps inbound requests to exposed TCP port

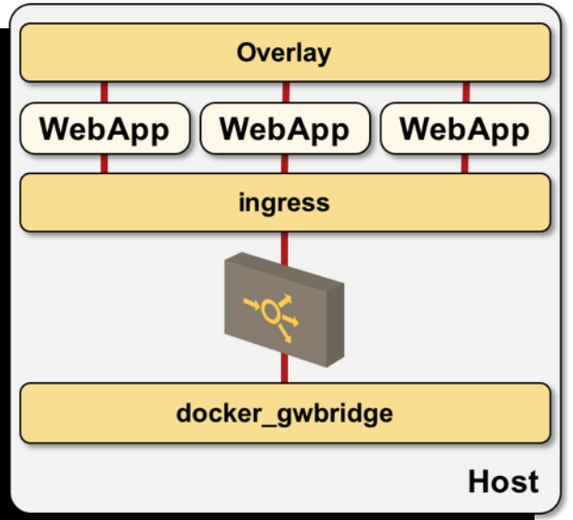


Where Is the ingress_sbox Process?

```
root         696     1  0 15:24 ?        00:00:00 /usr/sbin/cron -f
root         698     1  0 15:24 ?        00:00:00 /usr/bin/containerd
root         773     1  0 15:24 ?        00:00:00 /usr/lib/policykit-1/polkitd --no-debug
root         956     1  0 15:24 ?        00:00:00 /usr/sbin/VBoxService --pidfile /var/run/vboxadd-service.sh
root         970     1  0 15:24 ?        00:00:23 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
root         989     1  0 15:24 tty1     00:00:00 /sbin/agetty -o -p -- \u --noclear tty1 linux
root         990     1  0 15:24 ?        00:00:00 /usr/sbin/sshd -D
systemd+    1777     1  0 15:24 ?        00:00:00 /lib/systemd/systemd-networkd
root        4465   990  0 16:27 ?        00:00:00 sshd: vagrant [priv]
vagrant     4467     1  0 16:27 ?        00:00:00 /lib/systemd/systemd --user
vagrant     4468  4467  0 16:27 ?        00:00:00 (sd-pam)
vagrant     4580  4465  0 16:27 ?        00:00:00 sshd: vagrant@pts/0
vagrant     4581  4580  0 16:27 pts/0    00:00:00 -bash
root        4656     2  0 16:27 ?        00:00:00 [kworker/0:1]
root        5658   698  0 16:40 ?        00:00:00 containerd-shim -namespace moby -workdir /var/lib/containerd/io.containerd.runtime.v1.l
root        5684  5658  0 16:40 ?        00:00:00 python app.py
root        5882   990  0 16:41 ?        00:00:00 sshd: vagrant [priv]
vagrant     5964  5882  0 16:41 ?        00:00:00 sshd: vagrant@pts/1
vagrant     5965  5964  0 16:41 pts/1    00:00:00 -bash
root        6409     2  0 16:55 ?        00:00:00 [kworker/u2:0]
root        6539     2  0 17:00 ?        00:00:00 [kworker/0:0]
root        6851     2  0 17:09 ?        00:00:00 [kworker/u2:1]
root        7009     2  0 17:15 ?        00:00:00 [kworker/u2:2]
vagrant     7025  4581  0 17:15 pts/0    00:00:00 ps -ef
$
```

Exploring iptables in ingress_sbox Network Namespace

```
$ sudo ip netns exec ingress_sbox iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER_OUTPUT
-N DOCKER_POSTROUTING
-A OUTPUT -d 127.0.0.11/32 -j DOCKER_OUTPUT
-A POSTROUTING -d 127.0.0.11/32 -j DOCKER_POSTROUTING
-A POSTROUTING -d 10.0.0.0/24 -m ipvs --ipvs -j SNAT --to-source 10.0.0.2
-A DOCKER_OUTPUT -d 127.0.0.11/32 -p tcp -m tcp --dport 53 -j DNAT --to-destination 127.0.0.11:39605
-A DOCKER_OUTPUT -d 127.0.0.11/32 -p udp -m udp --dport 53 -j DNAT --to-destination 127.0.0.11:58377
-A DOCKER_POSTROUTING -s 127.0.0.11/32 -p tcp -m tcp --sport 39605 -j SNAT --to-source :53
-A DOCKER_POSTROUTING -s 127.0.0.11/32 -p udp -m udp --sport 58377 -j SNAT --to-source :53
$
```



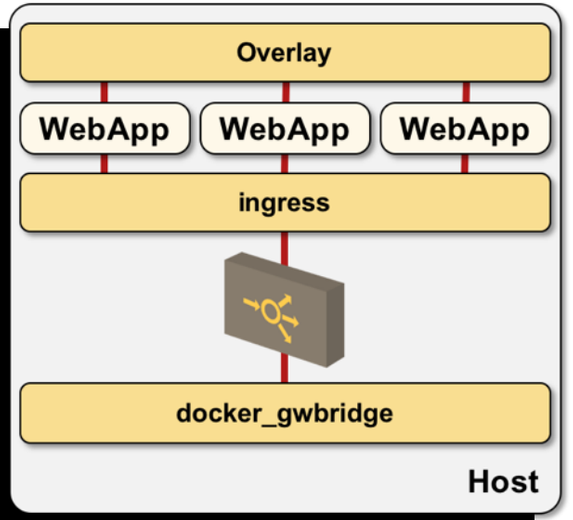
- DNS mapping rules (similar to most containers)
- IPVS masquerading rule ➔ Docker might use IPVS load balancer

IPVS basics

- Simple load balancer implemented in Linux kernel
- Uses direct server return or source NAT (the option Docker selected)

IPVS Setup in ingress_sbox Namespace

```
$ sudo ip netns exec ingress_sbox ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
FWM  256 rr
  -> 10.0.0.7:0                   Masq    1      0          0
  -> 10.0.0.8:0                   Masq    1      0          0
  -> 10.0.0.9:0                   Masq    1      0          0
$ sudo ip netns exec ingress_sbox iptables -t mangle -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P FORWARD ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-A PREROUTING -p tcp -m tcp --dport 8080 -j MARK --set-xmark 0x100/0xffffffff
-A INPUT -d 10.0.0.6/32 -j MARK --set-xmark 0x100/0xffffffff
```

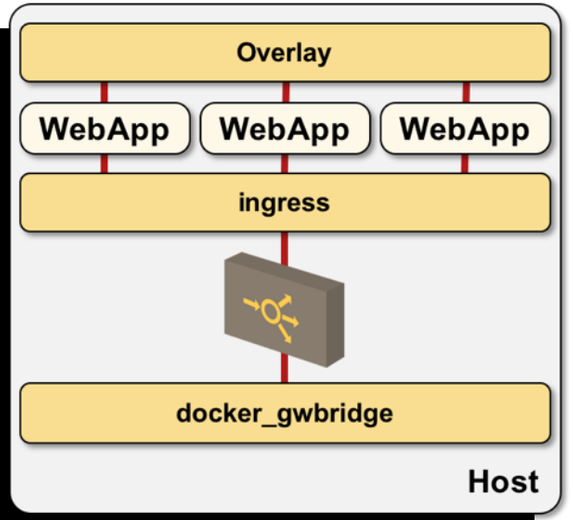


- Firewall Mark (internal packet marking) is used to select IPVS service
- Firewall Mark is set by **mangle** iptables rules
- Mangle rules set a different Firewall Mark for every Docker Swarm service based on destination TCP port

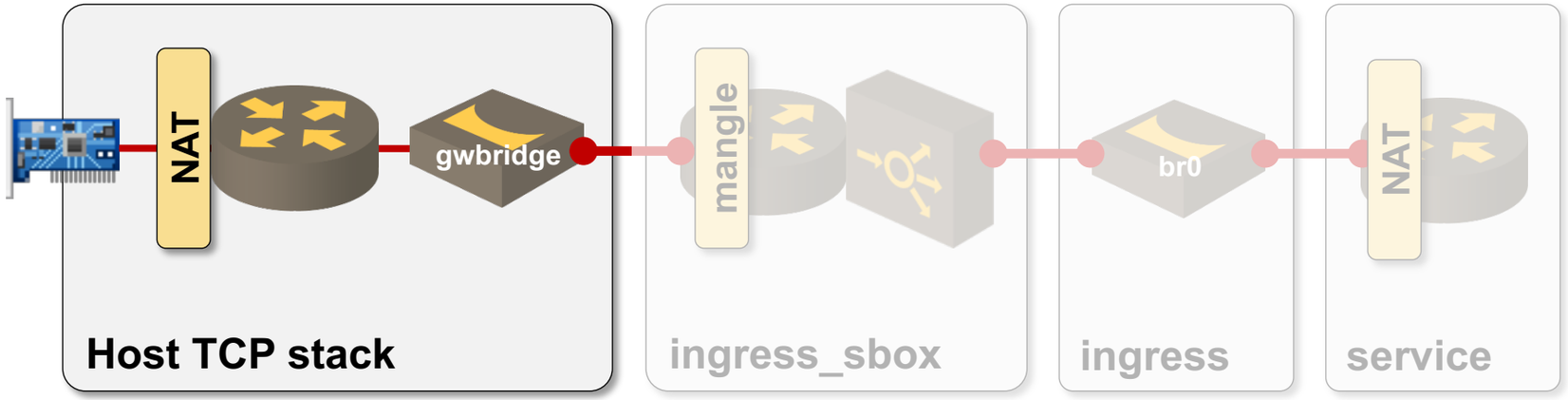
Destination TCP port after a packet exits IPVS is still 8080. What next?

Final Port Remapping Happens in Service Container Namespace

```
$ sudo ip netns
fc8de92089a5 (id: 4)
1-3e0ykh53lm (id: 2)
lb_3e0ykh53l (id: 3)
1-kyr5v5lidn (id: 0)
ingress_sbox (id: 1)
$ sudo ip netns exec fc8de92089a5 ip address show dev eth0
41: eth0@if42: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:00:00:08 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.0.0.8/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
$ sudo ip netns exec fc8de92089a5 iptables -t nat -S
-P PREROUTING ACCEPT
-P INPUT ACCEPT
-P OUTPUT ACCEPT
-P POSTROUTING ACCEPT
-N DOCKER_OUTPUT
-N DOCKER_POSTROUTING
-A PREROUTING -d 10.0.0.8/32 -p tcp -m tcp --dport 8080 -j REDIRECT --to-ports 80
-A OUTPUT -d 127.0.0.11/32 -j DOCKER_OUTPUT
-A POSTROUTING -d 127.0.0.11/32 -j DOCKER_POSTROUTING
-A DOCKER_OUTPUT -d 127.0.0.11/32 -p tcp -m tcp --dport 53 -j DNAT --to-destination 127.0.0.11:32845
-A DOCKER_OUTPUT -d 127.0.0.11/32 -p udp -m udp --dport 53 -j DNAT --to-destination 127.0.0.11:41827
-A DOCKER_POSTROUTING -s 127.0.0.11/32 -p tcp -m tcp --sport 32845 -j SNAT --to-source :53
-A DOCKER_POSTROUTING -s 127.0.0.11/32 -p udp -m udp --sport 41827 -j SNAT --to-source :53
```

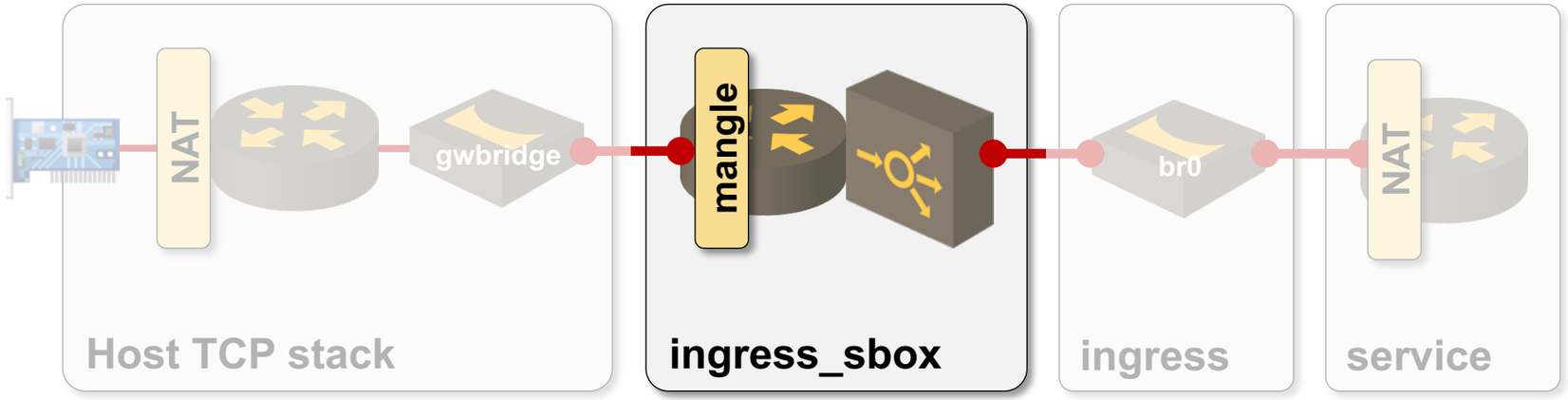


Swarm Service Incoming Connections: End-to-End Picture (Part 1)



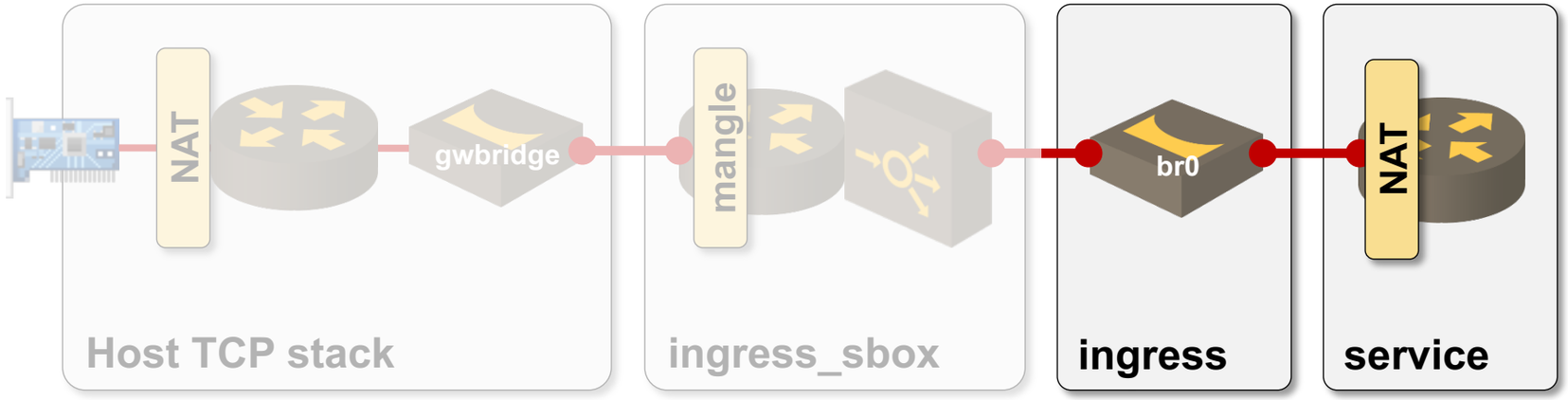
- Ethernet NIC receives a packet addressed to a published port on host IP address
- NAT rules match on TCP port and map destination IP address to **ingress_sbox** IP address
- Modified packet is routed to **docker_gwbridge** subnet and bridged to **veth** link to **ingress_sbox** namespace

Swarm Service Incoming Connections: End-to-End Picture (Part 2)



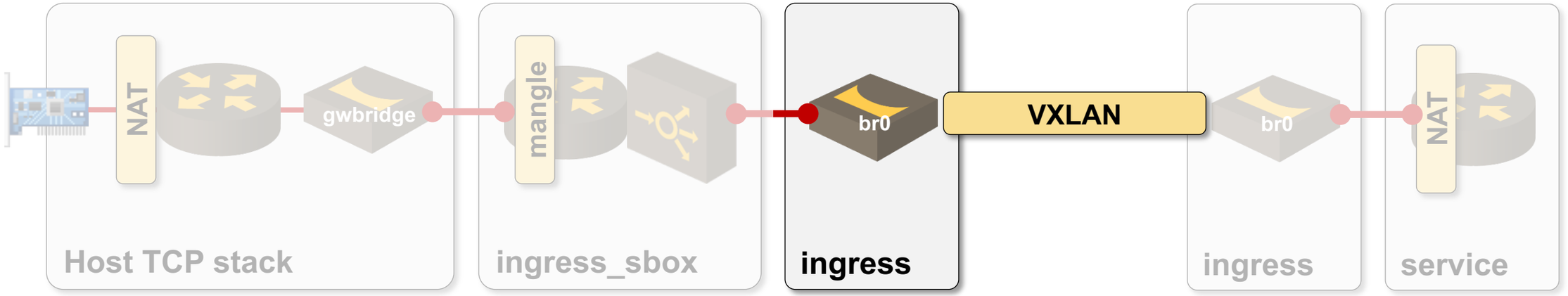
- Packet is received on **veth** interface in **ingress_sbox** namespace
- Mangle rules set firewall mark
- IPVS intercepts the packet, selects a load balancing service based on the firewall mark, and selects an IP address of one of the available service containers
- Destination IP address is rewritten
- Packet is routed within the **ingress_sbox** namespace and sent to **veth** interface toward **ingress** namespace

Swarm Service Incoming Connections: End-to-End Picture (Part 3)



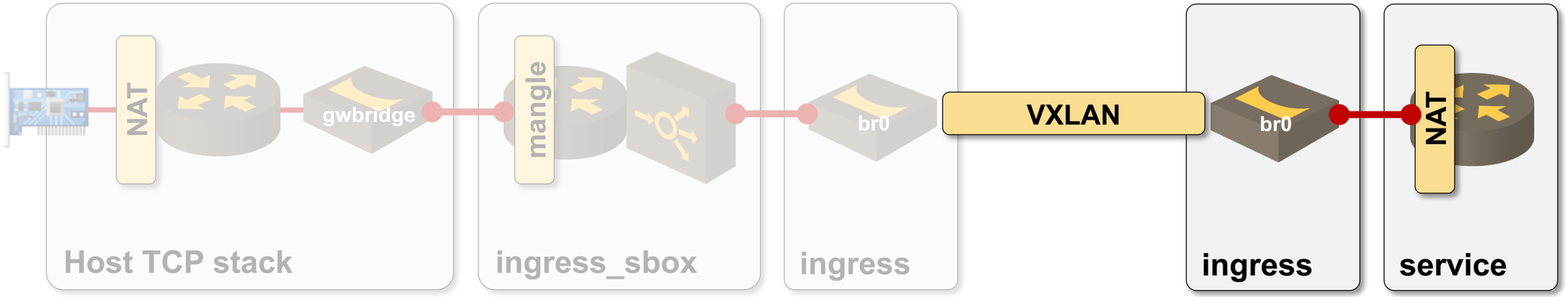
- Packet is received on **veth** interface in **ingress** namespace
- Destination MAC address points to outgoing **veth** interface
  ➔ packet is bridged toward a service container
- Packet is received on **veth** interface in a service container
- NAT rules rewrite destination port number to actual service port number
- Packet is routed to local process

Swarm Service Incoming Connections: Remote Containers (Part 1)



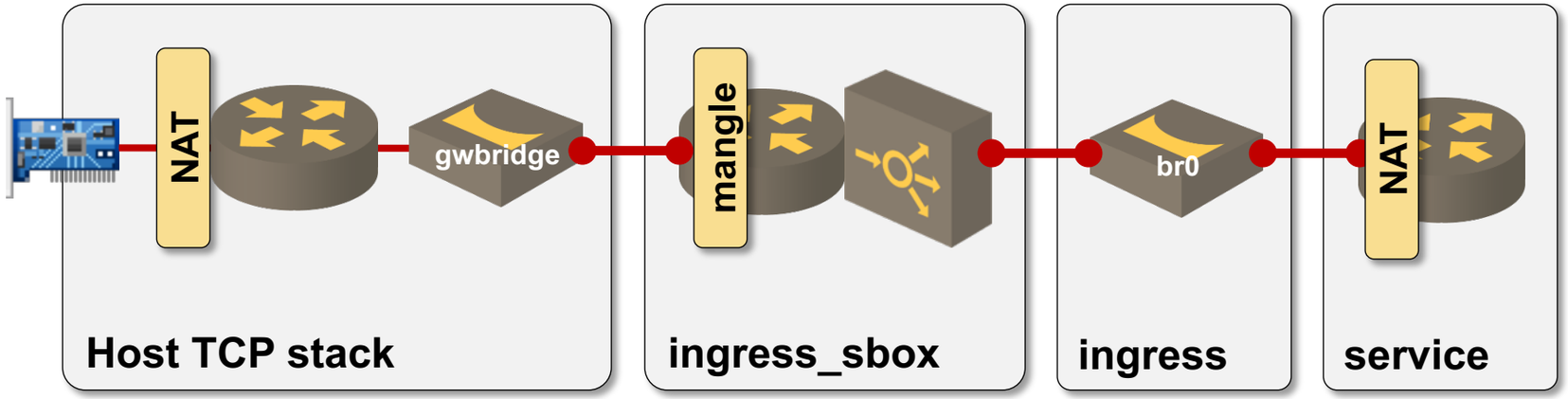
- Packet is received on **veth** interface in **ingress** namespace
- Destination MAC address points to outgoing VXLAN interface
  ➔ packet is bridged to **vxlan0** interface
- Static MAC-to-VTEP mapping is used to find destination IP address
- VXLAN packet is generated and routed through host TCP/IP stack to Ethernet NIC

Swarm Service Incoming Connections: Remote Containers (Part 2)



- VXLAN packet is received by destination Swarm node
- VXLAN interface is selected based on VNI ➔ packet is passed to **vxlan0** interface in **ingress** namespace
- Destination MAC address points to outgoing **veth** interface
  ➔ packet is bridged toward a service container

Swarm Service Incoming Connections: Return Traffic



- Connection tracking entries are created by every NAT rule and IPVS
- Service container sends return traffic to **ingress** interface of **ingress_sbox** load balancer
- NAT connection entry in service container remaps port number
- IPVS connection entry changes IP addresses: source to **ingress_sbox** VIP address, destination to original client's IP address
- NAT connection entry in host TCP/IP stack changes source IP address to host IP address

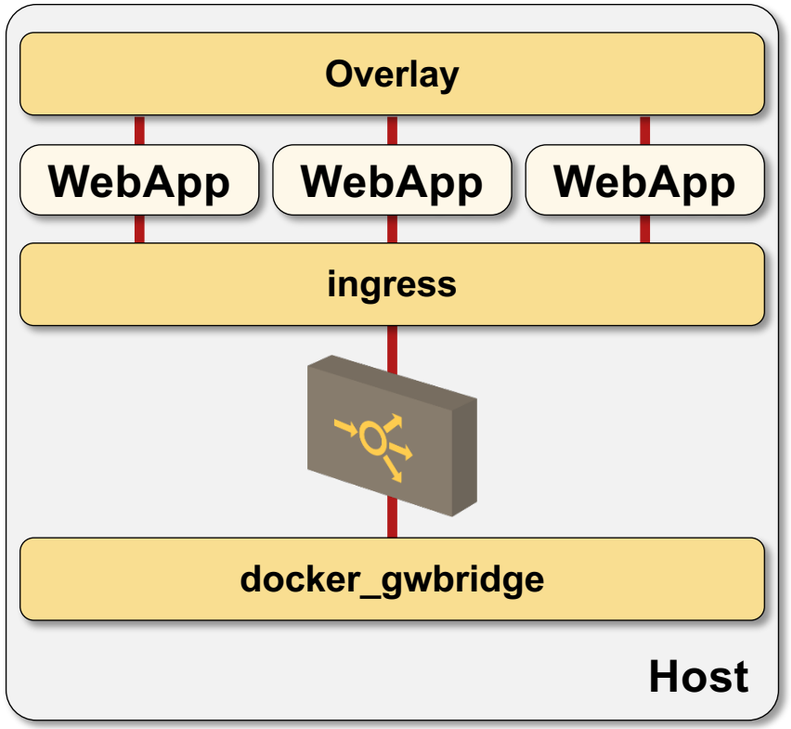
Docker Swarm Networking Summary

From a Single Node to a Swarm

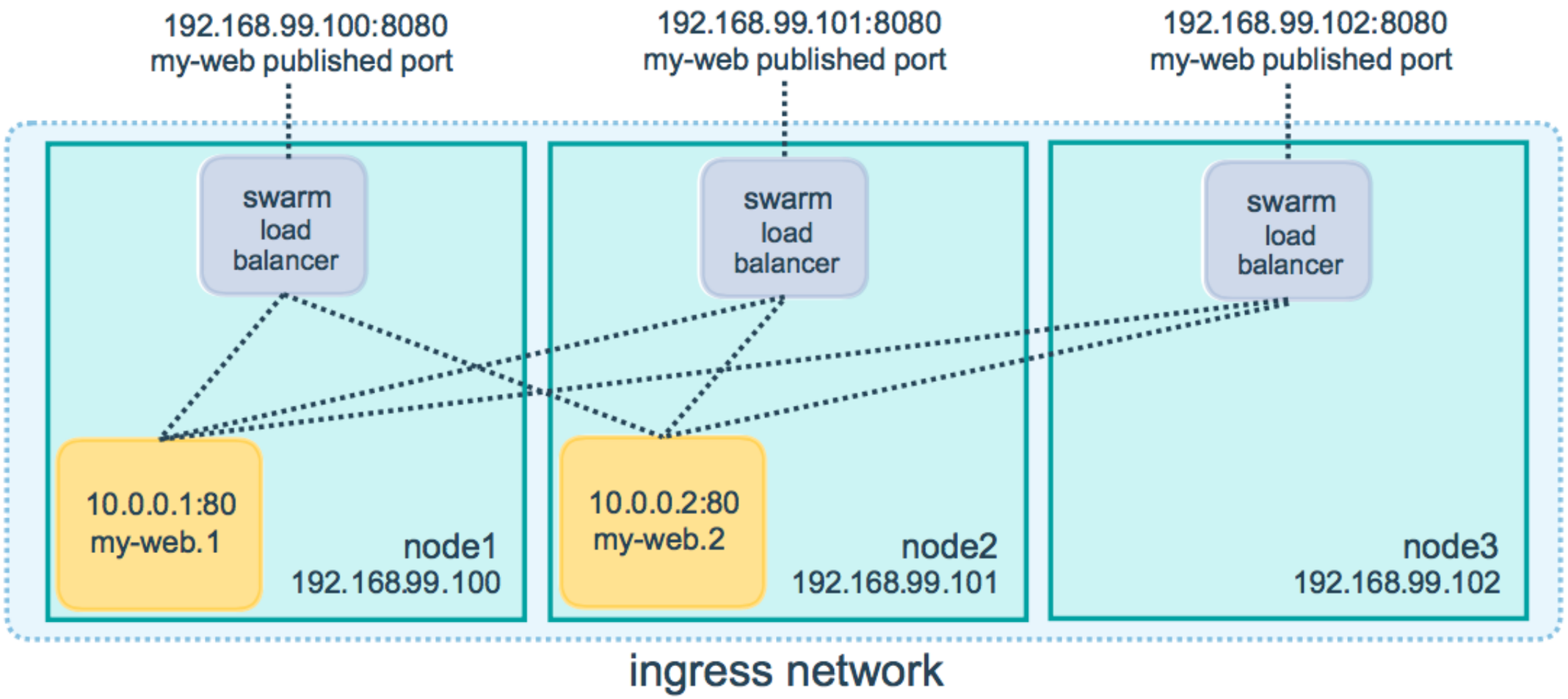
- All overlay networks are stretched across all swarm members
- Docker networks are instantiated on as-needed basis
- VXLAN interfaces and namespaces are created when a container is attached to a Docker network
- Published ports work exactly like on standalone Docker nodes

Swarm-wide services

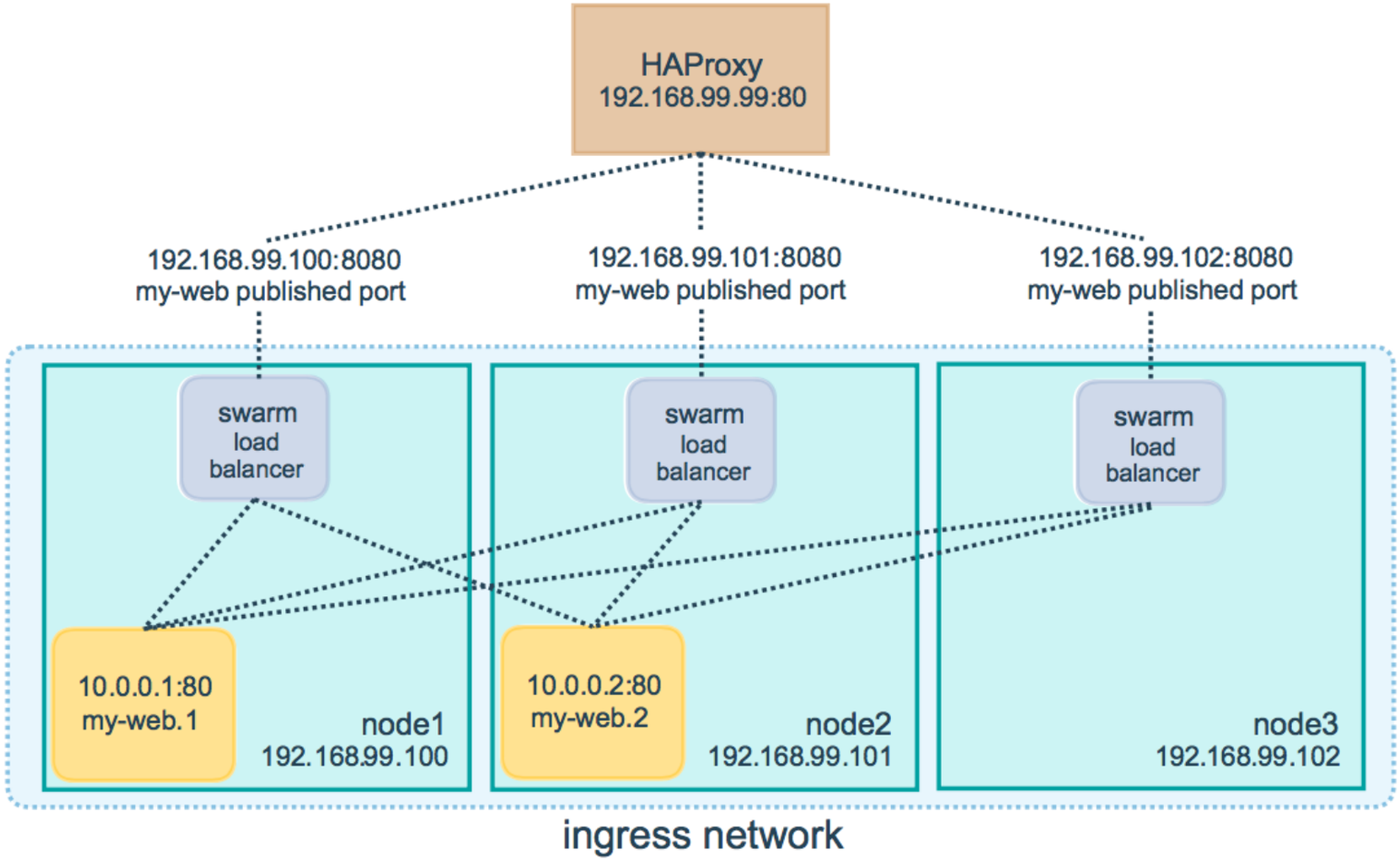
- **docker service** deploys containers across swarm members
- **ingress_sbox** container is deployed on every swarm member node
- **ingress** overlay virtual network connects load balancers on all swarm member nodes to all containers with published ports
- Incoming TCP session can arrive to any swarm member node
- **ingress_sbox** load balancer can map inbound requests to any container (including containers on other swarm nodes)



Load Balancing in Docker Swarm



Adding External Load Balancer



Questions?

Web:	ipSpace.net
Blog:	blog.ipSpace.net
Email:	ip@ipSpace.net
Twitter:	@ioshints
Data center:	ipSpace.net/NextGenDC
Automation:	ipSpace.net/NetAutSol
Public cloud:	ipSpace.net/PubCloud
Webinars:	ipSpace.net/Webinars
Consulting:	ipSpace.net/Consulting