Deep Dive in Docker Overlay Networks

Laurent Bernaille

CTO, D2SI

@lbernail

Agenda

The Docker overlay

1. Getting started
2. Under the hood

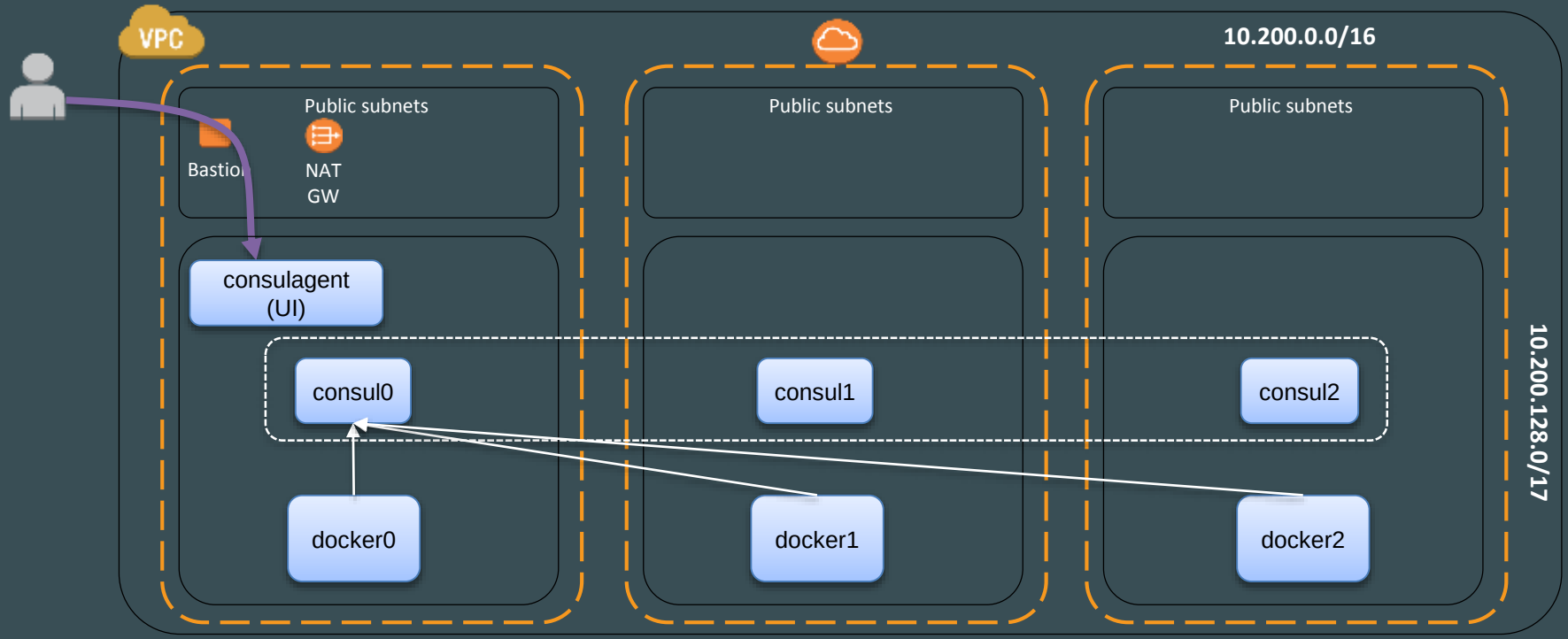
Building our overlay

1. Starting from scratch
2. Making it dynamic

Docker Overlay Networks

Getting started

Environment



dockerd -H fd:// --cluster-store=consul://consul0:8500 --cluster-advertise=eth0:2376

What is in consul? Not much for now just metadata tree

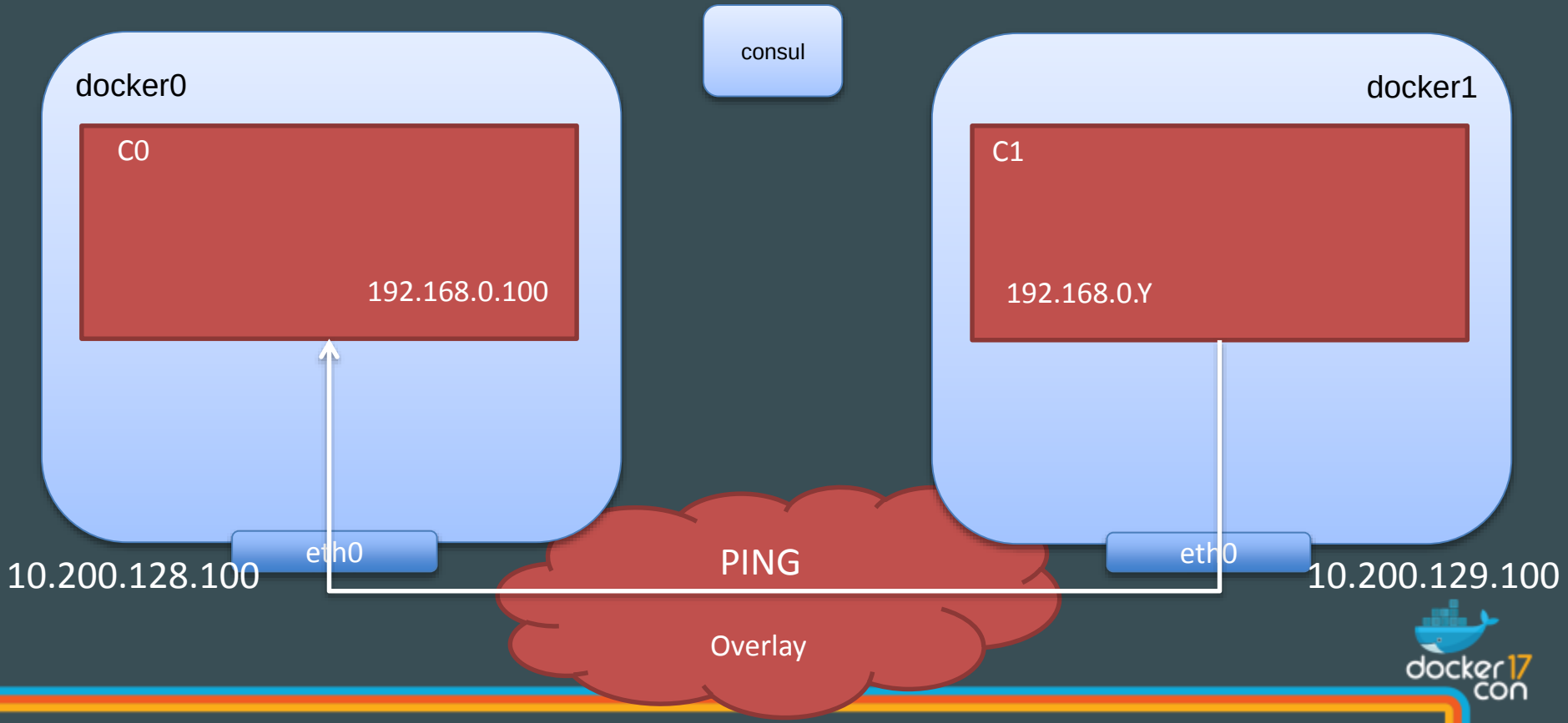
Let's create an overlay network

```
docker0:~$ docker network create --driver overlay \
                                 --internal \
                                 --subnet 192.168.0.0/24 dockercon
c4305b67cda46c2ed96ef797e37aed14501944a1fe0096dacd1ddd8e05341381

docker1:~$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
bec777b6c1f1        bridge              bridge              local
c4305b67cda4        dockercon           overlay             global
3a4e16893b16        host                host                local
c17c1808fb08        none                null                local
```

Does it work?

```
docker0:~$ docker run -d --ip 192.168.0.100 --net dockercon --name C0 debian sleep 3600

docker1:~$ docker run --net dockercon debian ping 192.168.0.100
PING 192.168.0.100 (192.168.0.100): 56 data bytes
64 bytes from 192.168.0.100: seq=0 ttl=64 time=1.153 ms
64 bytes from 192.168.0.100: seq=1 ttl=64 time=0.807 ms

docker1:~$ ping 192.168.0.100
PING 192.168.0.100 (192.168.0.100) 56(84) bytes of data.
^C--- 192.168.0.100 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 3024ms
```

What did we build?



Docker Overlay Networks

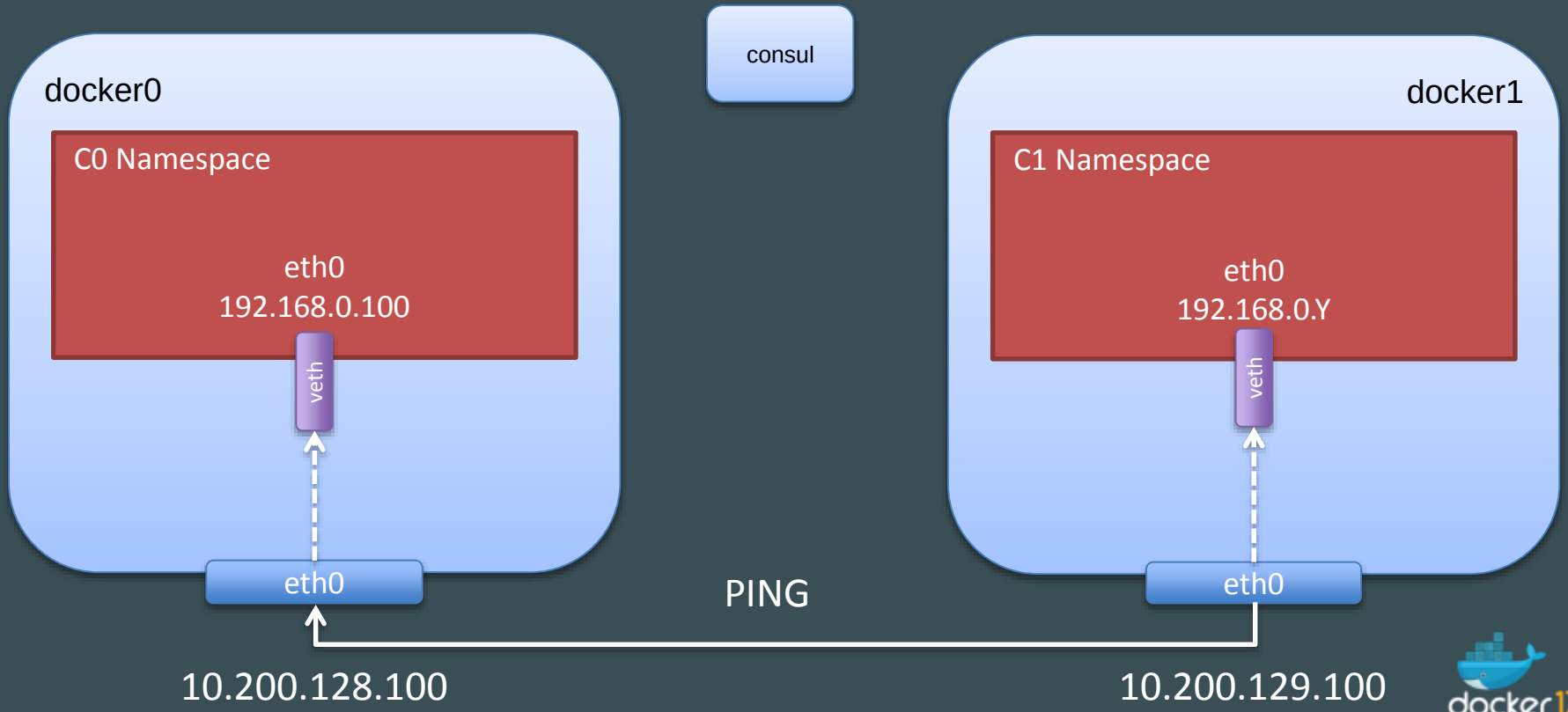
Under the hood

How does it work? Let's look inside containers

```
docker0:~$ docker exec C0 ip addr show
58: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP
    inet 192.168.0.100/24 scope global eth0
```

```
docker0:~$ docker exec C0 ip -details link show dev eth0
58: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group
default
    veth
```

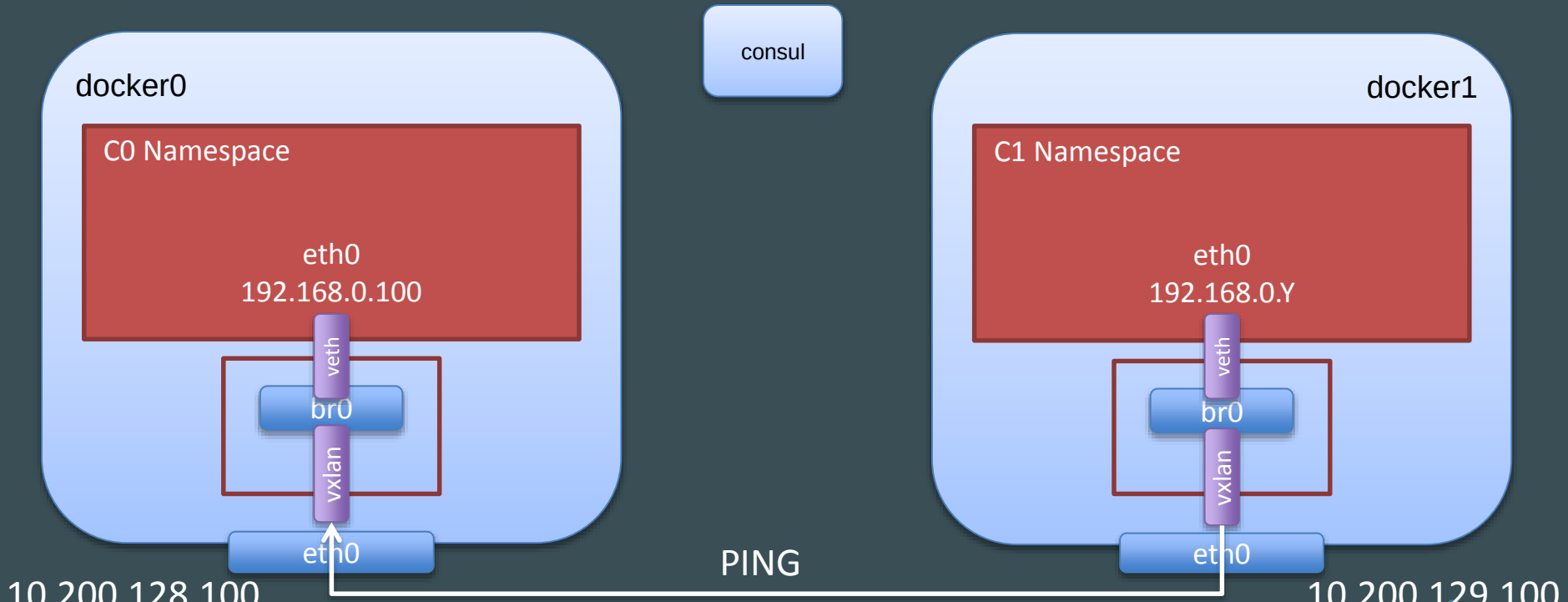
Container network configuration



Where is the other end of the veth?

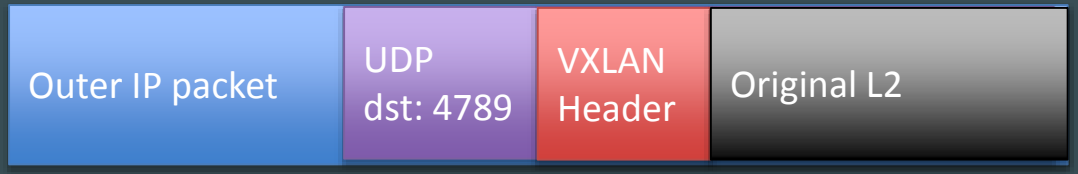
```
docker0:~$ ip link show          >> Nothing, it must be in another Namespace
docker0:~$ sudo ip netns ls
8-c4305b67cd
docker0:~$ docker network inspect dockercon -f {{.Id}}
  c4305b67cda46c2ed96ef797e37aed14501944a1fe0096dacd1ddd8e05341381

docker0:~$ overns=8-c4305b67cd
docker0:~$ sudo ip netns exec $overns ip -d link show
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default
   bridge
62: vxlan1: <..> mtu 1450 qdisc noqueue master br0 state UNKNOWN mode DEFAULT group default
   vxlan id 256 srcport 10240 65535 dstport 4789 proxy l2miss l3miss ageing 300
59: veth2: <...> mtu 1450 qdisc noqueue master br0 state UP mode DEFAULT group default
```

Update on connectivity



What is VXLAN?

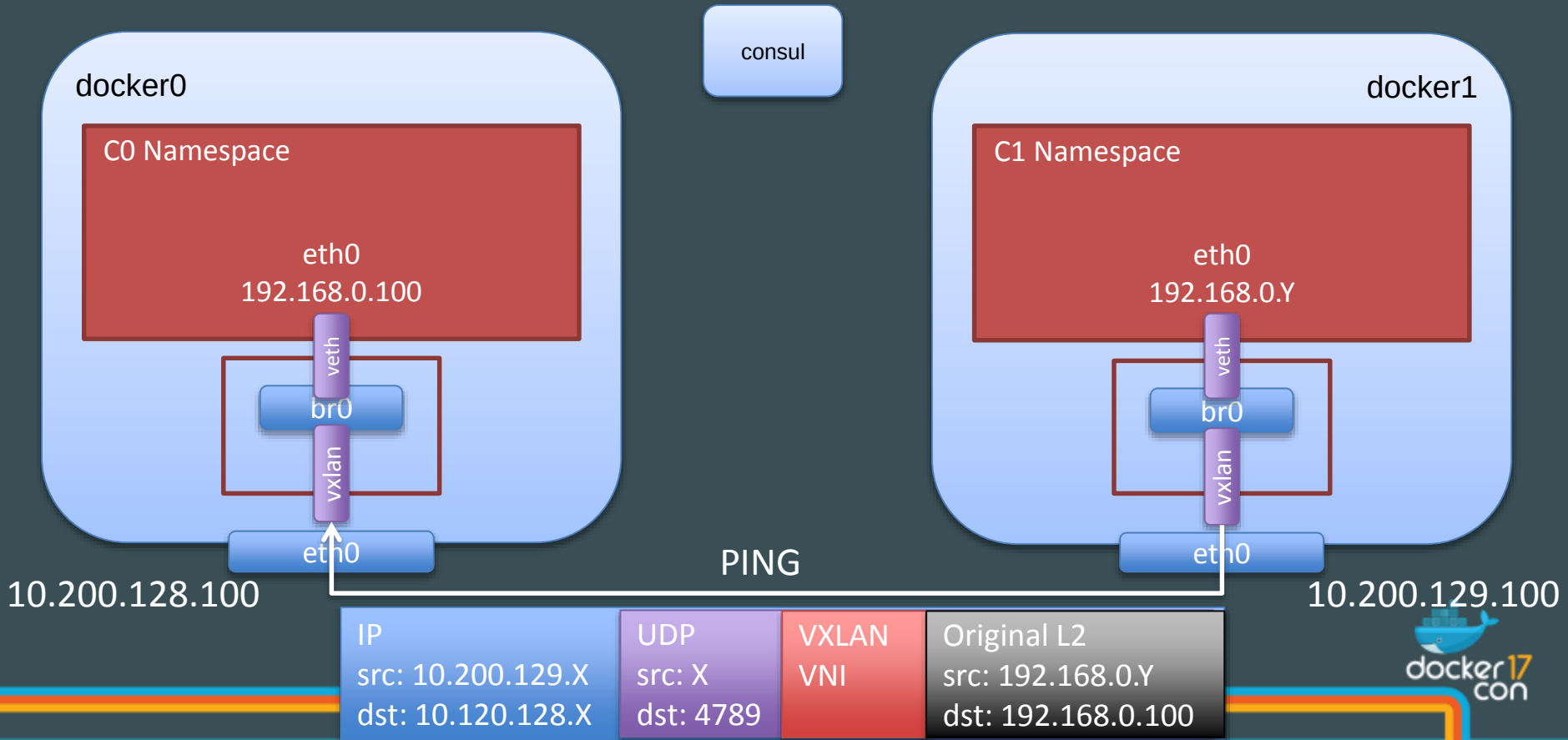
- Tunneling technology over UDP (L2 in UDP)
- Developed for cloud SDN to create multitenancy
  - Without the need for L2 connectivity
  - Without the normal VLAN limit (4096 VLAN Ids)
- In Linux
  - Started with Open vSwitch
  - Native with Kernel >= 3.7 and >=3.16 for Namespace support



Let's verify this

```
docker0:~$ sudo tcpdump -nn -i eth0 "port 4789"

docker1:~$ docker run --net dockercon debian ping 192.168.0.100
PING 192.168.0.100 (192.168.0.100): 56 data bytes
64 bytes from 192.168.0.100: seq=0 ttl=64 time=1.153 ms
64 bytes from 192.168.0.100: seq=1 ttl=64 time=0.807 ms

docker0:~$
listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
14:05:04.041366 IP 10.200.129.98.34922 > 10.200.128.130.4789: VXLAN, flags [I] (0x08), vni 256
   IP 192.168.0.2 > 192.168.0.100: ICMP echo request, id 256, seq 62903, length 64
14:05:04.041429 IP 10.200.128.130.59164 > 10.200.129.98.4789: VXLAN, flags [I] (0x08), vni 256
   IP 192.168.0.100 > 192.168.0.2: ICMP echo reply, id 256, seq 62903, length 64
```

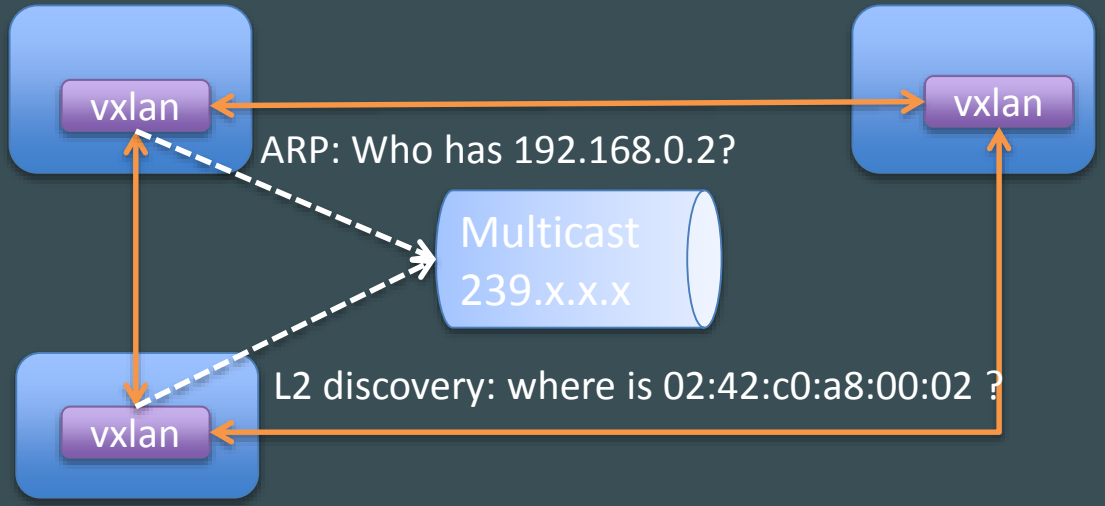
Full connectivity with VXLAN



How does docker0 know about containers on C1

```
docker0:~$ sudo ip netns exec $overns ip neighbor show
  192.168.0.2 dev vxlan1 lladdr 02:42:c0:a8:00:02 PERMANENT
docker0:~$ sudo ip netns exec $overns bridge fdb show br br0
  02:42:c0:a8:00:02 dev vxlan1 dst 10.200.129.98 self permanent

docker1:~$ docker run -d --ip 192.168.0.200 --net dockercon --name C1 debian sleep 3600
docker0:~$ sudo ip netns exec $overns ip neighbor show
  192.168.0.2 dev vxlan1 lladdr 02:42:c0:a8:00:02 PERMANENT
  192.168.0.20 dev vxlan1 lladdr 02:42:c0:a8:00:14 PERMANENT
docker0:~$ sudo ip netns exec $overns ip neighbor show
  02:42:c0:a8:00:02 dev vxlan1 dst 10.200.129.98 self permanent
  02:42:c0:a8:00:14 dev vxlan1 dst 10.200.129.98 self permanent
```

VXLAN L2/L3 resolution - Option 1: Multicast



Use a multicast group to send traffic for unknown L3/L2 addresses

- PROS: simple and efficient
- CONS: Multicast connectivity not always available (on public clouds for instance)

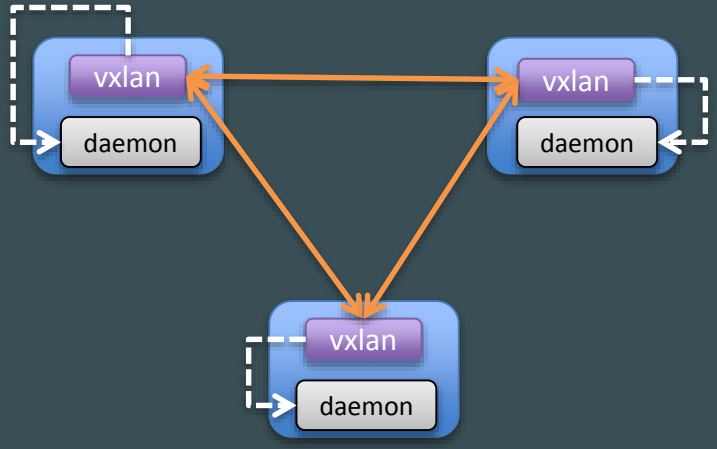
VXLAN L2/L3 resolution - Option 2: Point-to-point

Configure a remote IP address where to send traffic for unknown addresses

- PROS: simple, not need for multicast, very good for two hosts
- CONS: difficult to manage with more than 2 hosts

VXLAN L2/L3 resolution - Option 3: Manual

Manual (with a daemon modifying ARP/FDB)
ARP: Do you know 192.168.0.2?
L2: where is 02:42:c0:a8:00:02 ?



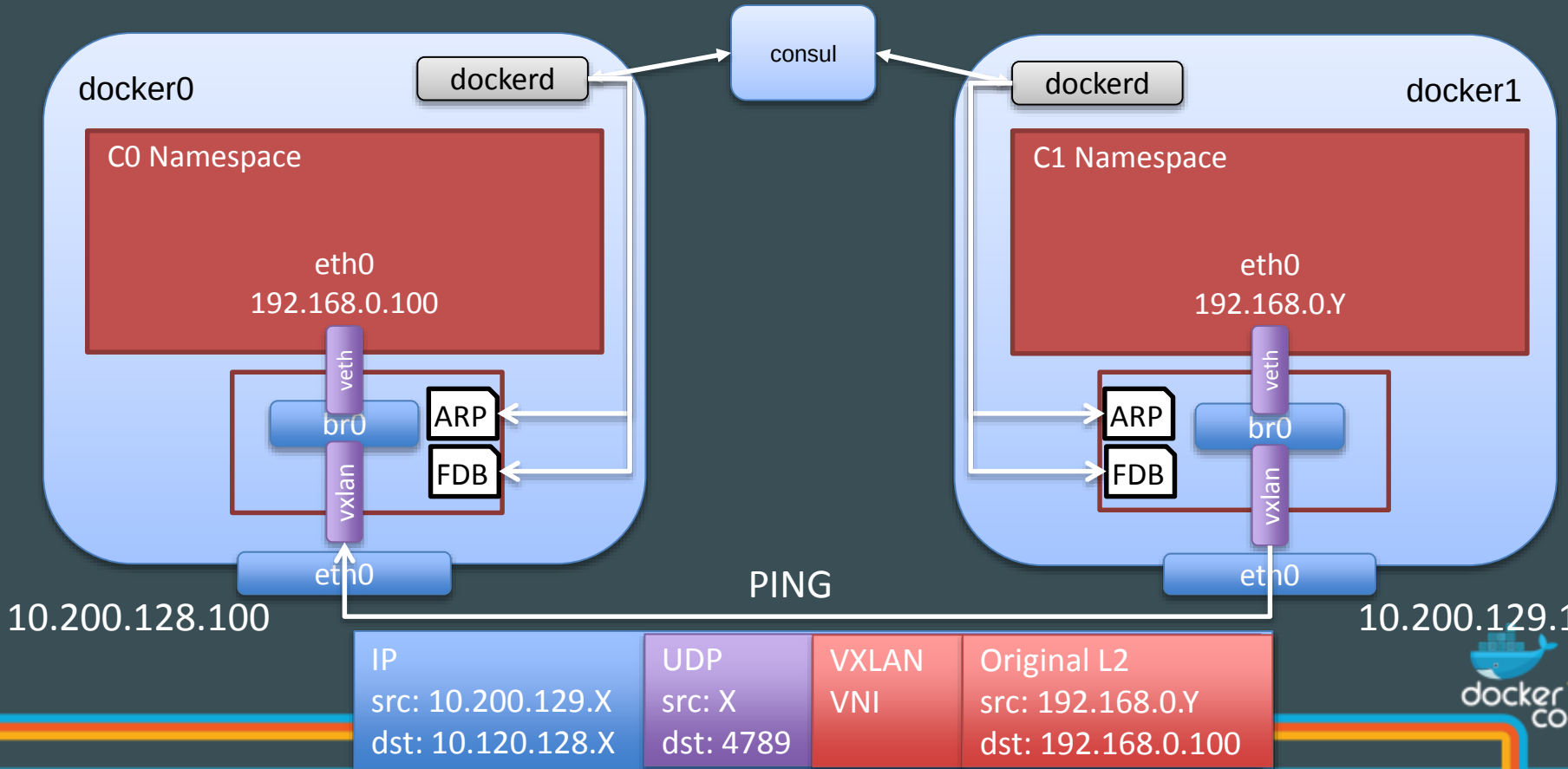
Do nothing, provide ARP / FDB information from outside

- PROS: very flexible
- CONS: requires a daemon and a centralized database of addresses

What's inside consul?

```
docker0:~$ docker network inspect dockercon -f {{.Id}}
docker0:~$ docker inspect C0 -f {{.NetworkSettings.Networks.dockercon.EndpointID}}
docker0:~$ net=
docker0:~$ curl -s http://consul1:8500/v1/kv/docker/network/v1.0/network/${net}/
docker0:~$ python/dump_endpoints.py
```

Overview


consul
docker0
dockerd
C0 Namespace
eth0
192.168.0.100
veth
br0
vxlan
ARP
FDB
eth0
10.200.128.100
docker1
dockerd
C1 Namespace
eth0
192.168.0.Y
veth
br0
vxlan
ARP
FDB
eth0
10.200.129.100
PING
IP
src: 10.200.129.X
dst: 10.120.128.X
UDP
src: X
dst: 4789
VXLAN
VNI
Original L2
src: 192.168.0.Y
dst: 192.168.0.100

Building our overlay

Starting from scratch

Clean up

```
docker0:~$ docker rm -f $(docker ps -aq)
docker0:~$ docker network rm dockercon
docker1:~$ docker rm -f $(docker ps -aq)
```

Start from scratch

Create overlay components

```
ip netns add overns
ip netns exec overns ip link add dev br0 type bridge
ip netns exec overns ip addr add dev br0 192.168.0.1/24
```

create overlay NS
create bridge in NS

```
ip link add dev vxlan1 type vxlan id 42 proxy learning l2miss l3miss dstport 4789
ip link set vxlan1 netns overns
ip netns exec overns ip link set vxlan1 master br0
```

create VXLAN interface
move it to NS
add it to bridge

```
ip netns exec overns ip link set vxlan1 up
ip netns exec overns ip link set br0 up
```

bring all interfaces up

Step 1: overlay Namespace

Create containers and connect them

```
docker0
docker run -d --net=none --name=demo debian sleep 3600

ctn_ns_path=$(docker inspect --format="{{ .NetworkSettings.SandboxKey}}" demo)
ctn_ns=${ctn_ns_path##*/}

ip link add dev veth1 mtu 1450 type veth peer name veth2 mtu 1450
ip link set dev veth1 netns overns
ip netns exec overns ip link set veth1 master br0
ip netns exec overns ip link set veth1 up

ip link set dev veth2 netns $ctn_ns
ip netns exec $ctn_ns ip link set dev veth2 name eth0 address 02:42:c0:a8:00:02
ip netns exec $ctn_ns ip addr add dev eth0 192.168.0.2
ip netns exec $ctn_ns ip link set dev eth0 up


docker1
Same with 192.168.0.2 / 02:42:c0:a8:00:02
```

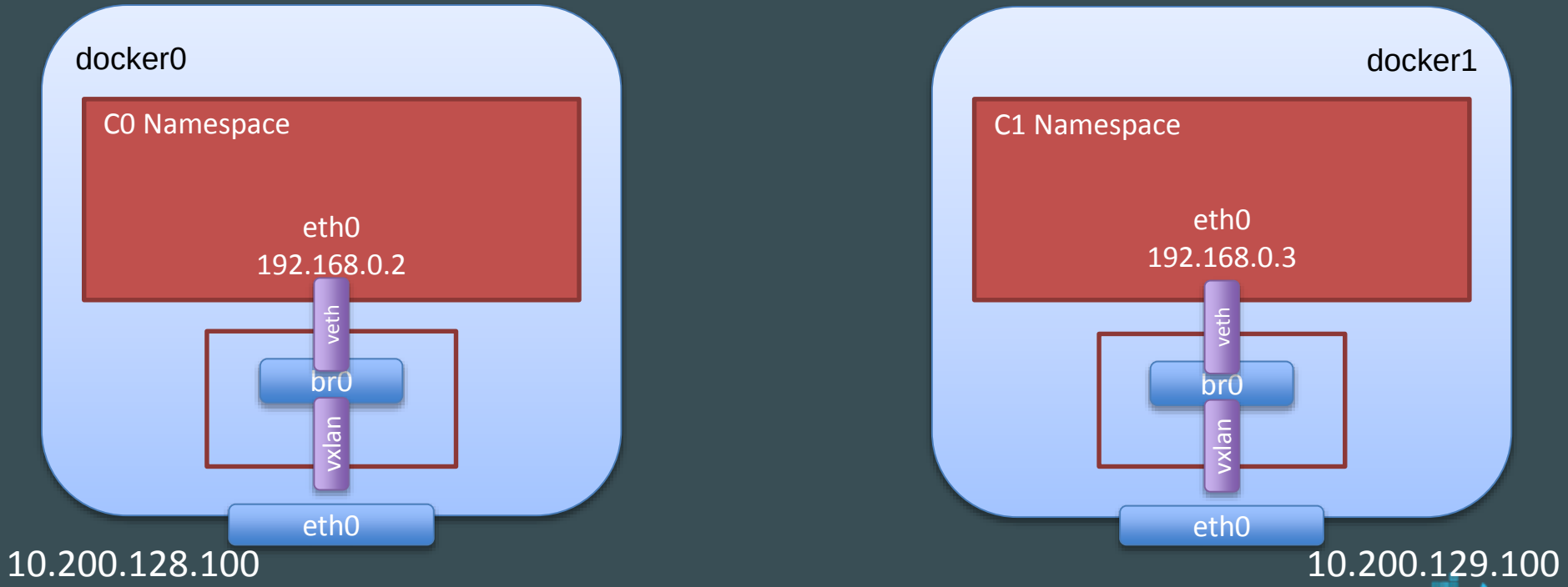
Create container without net

Get NS for container

Create veth
Send veth1 to overlay NS
Attach it to overlay bridge

Send veth2 to container
Rename & Configure

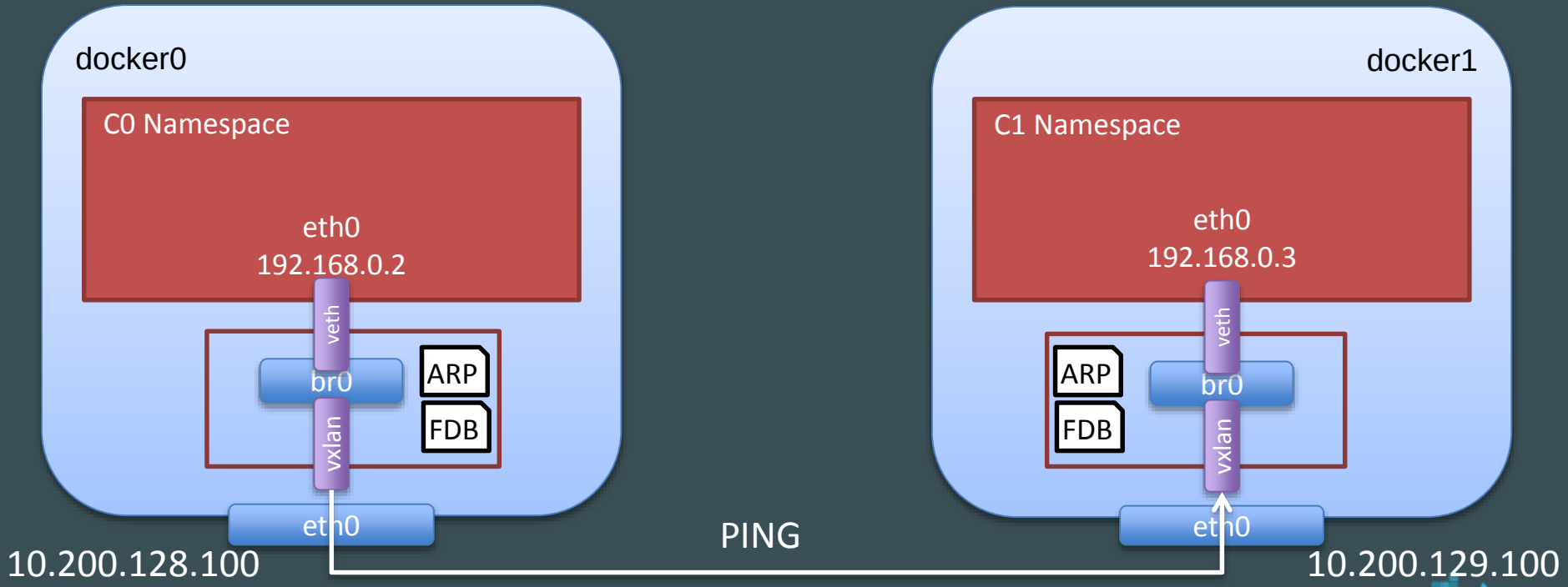
With container interfaces



Does it ping?

```
docker0:~$ docker exec -it demo ping 192.168.0.3
PING 192.168.0.3 (192.168.0.3): 56 data bytes
92 bytes from 192.168.0.2: Destination Host Unreachable

docker0:~$ sudo ip netns exec overns ip neighbor show

docker0:~$ sudo ip netns exec overns ip neighbor add 192.168.0.3 lladdr 02:42:c0:a8:00:03 dev vxlan1

docker0:~$ sudo ip netns exec overns bridge fdb add 02:42:c0:a8:00:03 dev vxlan1 self dst 10.200.129.98

\           vni 42 port 4789
docker0:~$ docker exec -it demo ping 192.168.0.3

docker1: Look at ARP / FDB tables

=> FDB has "learned" about 02:42:c0:a8:00:02 but ARP does not know about 192.168.0.2

docker1:~$ sudo ip netns exec overns ip neighbor add 192.168.0.2 lladdr 02:42:c0:a8:00:02 dev vxlan1
```

Final result



Docker Overlay Network

Making it dynamic

Recreate overlay without static entries

```
docker0:~$ sudo ip netns delete overns


Recreate overlay Namespace, bridge and VXLAN

Connect demo container to overlay NS bridge


docker0:~$ sudo ip netns exec overns ip link show

docker0:~$ sudo ip netns exec overns ip neighbor show

docker0:~$ sudo ip netns exec overns bridge fdb show
```

Catching network events: NETLINK

- Kernel interface for communication between Kernel and userspace
- Designed to transfer networking info (used by iproute2)
- Several protocols
  - **NETLINK_ROUTE**
  - NETLINK_FIREWALL
- Several notification types, for *NETLINK_ROUTE* for instance:
  - LINK
  - **NEIGHBOR**
- Many events
  - LINK: NEWLINK, GETLINK
  - **NEIGHBOR: GETNEIGH <= information on ARP, L2 discovery queries**

Catching L2/L3 misses

```
#!/usr/bin/env python

# Create the netlink socket and bind to NEIGHBOR NOTIFICATION
s = socket.socket(socket.AF_NETLINK, socket.SOCK_RAW, socket.NETLINK_ROUTE)
s.bind((os.getpid(), RTMGRP_NEIGH))

while True:
    data = s.recv(65535)
    msg_len, msg_type, flags, seq, pid = struct.unpack("=LHHLL", data[:16])

    # We fundamentally only care about GETNEIGH messages
    if msg_type != RTM_GETNEIGH:
        continue
```

```
data=data[16:]
ndm_family, _, _, ndm_ifindex, ndm_state, ndm_flags, ndm_type = struct.unpack("=BBHiHBB", data[:12])
logging.debug("Received a Neighbor miss")
logging.debug("Family: {}".format(if_family.get(ndm_family,ndm_family)))
logging.debug("Interface index: {}".format(ndm_ifindex))
logging.debug("State: {}".format(nud_state.get(ndm_state,ndm_state)))
logging.debug("Flags: {}".format(ndm_flags))
logging.debug("Type: {}".format(type.get(ndm_type,ndm_type)))
```

```
data=data[12:]
rta_len, rta_type = struct.unpack("=HH", data[:4])
logging.debug("RT Attributes: Len: {}, Type: {}".format(rta_len,nda_type.get(rta_type,rta_type)))
```

```
data=data[4:]
if nda_type.get(rta_type,rta_type) == "NDA_DST":
  dst=socket.inet_ntoa(data[:4])
  logging.info("L3Miss: Who has IP: {}?".format(dst))

if nda_type.get(rta_type,rta_type) == "NDA_LLADDR":
  mac="%02x:%02x:%02x:%02x:%02x:%02x" % struct.unpack("BBBBBB",data[:6])
  logging.info("L2Miss: Who has MAC: {}?".format(mac))
```

Total Length		
Msg Type (**GETNEIGH**)	Msg Flags	
Sequence Number		
PID		
Family	padding	
Interface Index		
State	Flags	Type
Attribute Length	Attribute Type (**NDA_DST**)	
IP Address		

nlmsghdr (Netlink msg hdr)

ndmsg (network discovery)

rtattr header (route attribute)

rtattr

>> use pyroute2

Catching L2/L3 misses

```
docker0-1:~$ sudo ip netns exec overns python/l2l3miss.py
docker0-2:~$ docker exec -it demo ping 192.168.0.3
docker0-1:~$
INFO:root:L3Miss: Who has IP: 192.168.0.3?
```

Add MAC entry for other container

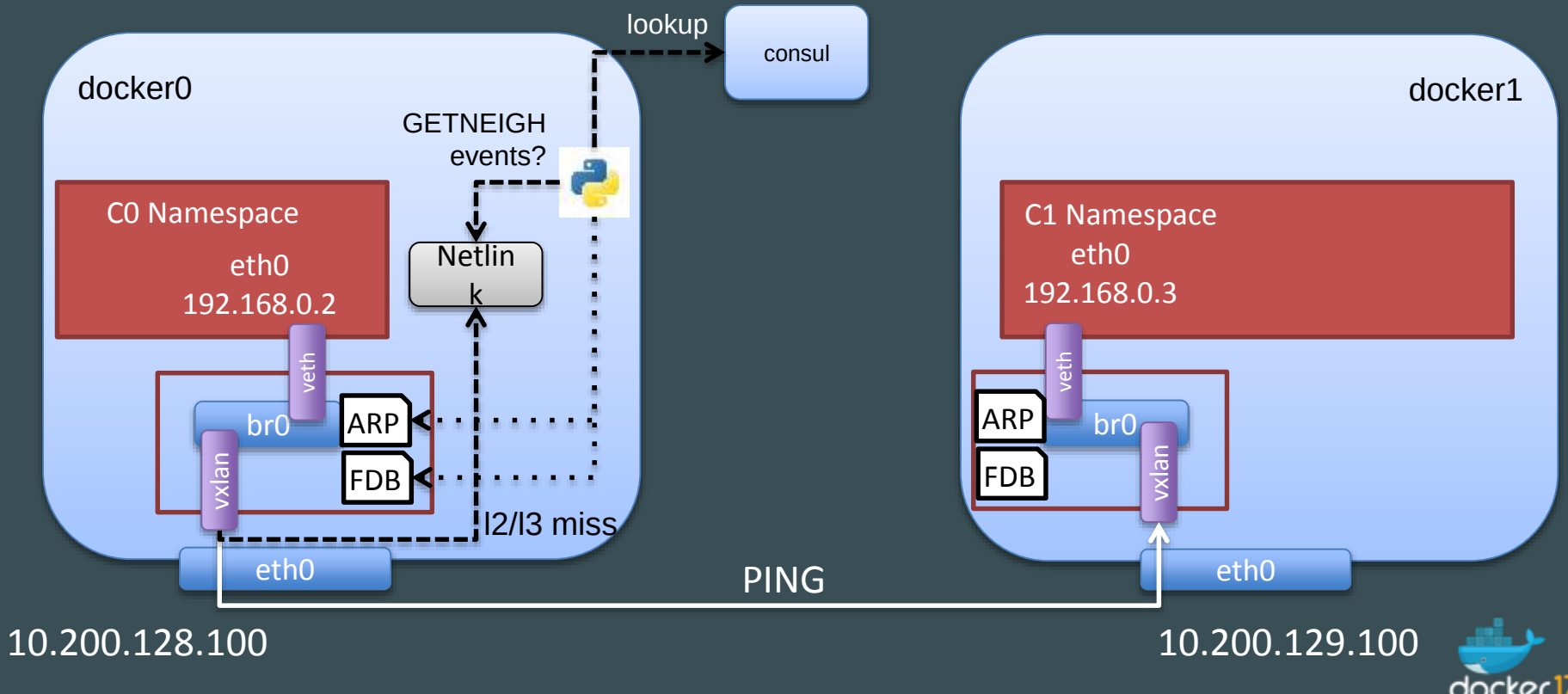
```
docker0-1:~$
INFO:root:L2Miss: Who has MAC: 02:42:c0:a8:00:03?
```

Add FDB entry >> It pings

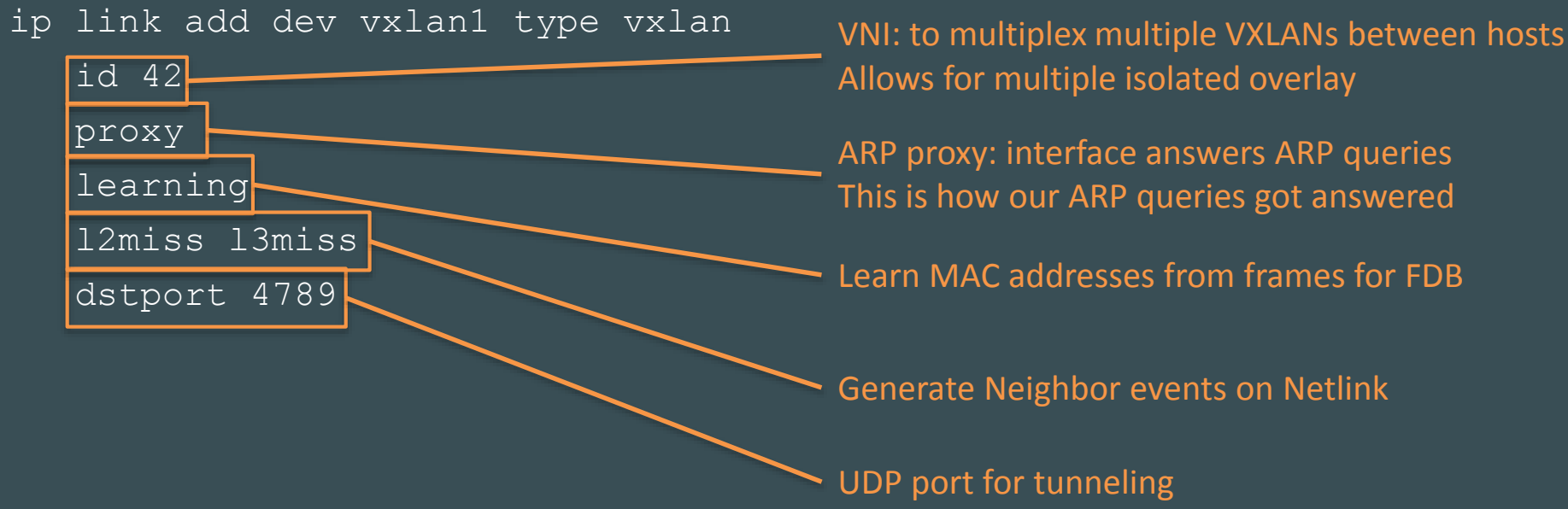
Storing MAC, FDB info in Consul

```
Recreate overlay, attach container


docker0:~$ sudo python/arpd-consul.py
docker0:~$ docker exec -it demo ping 192.168.0.3
INFO Starting new HTTP connection (1): consul1
INFO L3Miss on vxlan1: Who has IP: 192.168.0.3?
INFO Populating ARP table from Consul: IP 192.168.0.3 is 02:42:c0:a8:00:03
INFO L2Miss on vxlan1: Who has Mac Address: 02:42:c0:a8:00:03?
INFO Populating FIB table from Consul: MAC 02:42:c0:a8:00:03 is on host 10.200.129.98
```

Final result



Quick summary on VLXAN options



A few tricky implementation details

- ip netns commands do not work by default with docker net NS
  - Workaround: nsenter or symlink /var/run/docker/netns to /var/run/netns
- vxlan must be created in host Network NS and moved in the overlay NS
  - Keeps a link with the host eth0 interface
  - Otherwise vxlan will not be able to go outside the host
- l2miss / l3miss
  - By default GETNEIGH events are not sent on Netlink
  - Alternative: use /proc/sys/net/ipv4/neigh/eth0/app_solicit
- Consul python script runs in host network namespace
  - If it runs in the overlay namespace it can not access consul
  - The script binds the Netlink socket inside the overlay Namespace

Thank You!

https://github.com/lbernail/dockercon2017

@lbernail

#dockercon