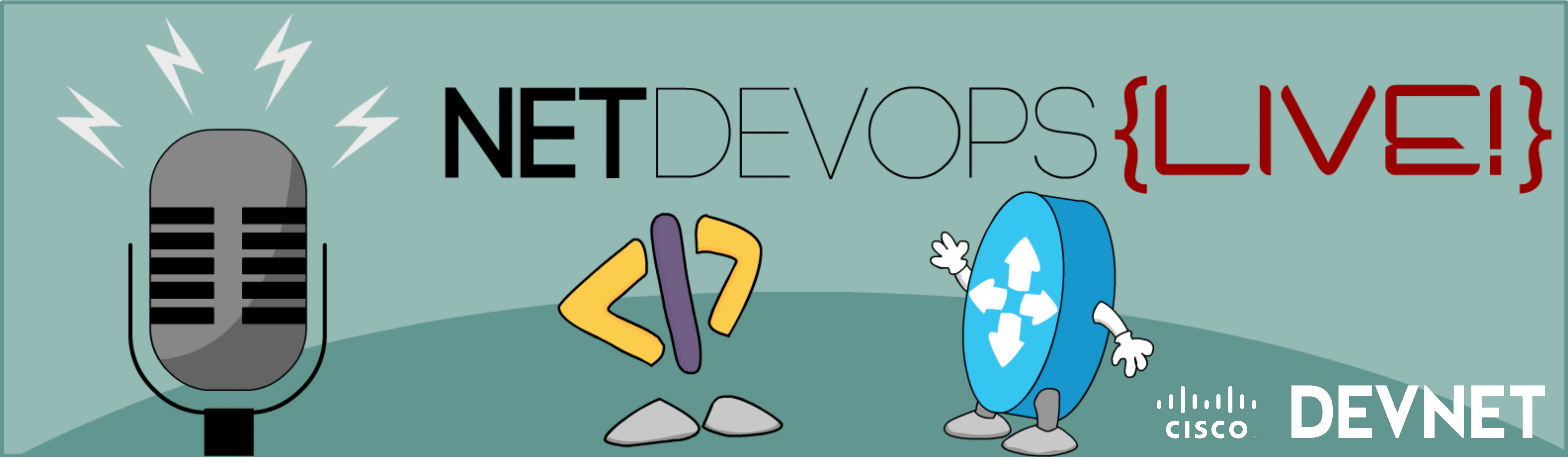


Linux Bridges, IP Tables & CNI Plug-Ins A Container Networking Deep dive

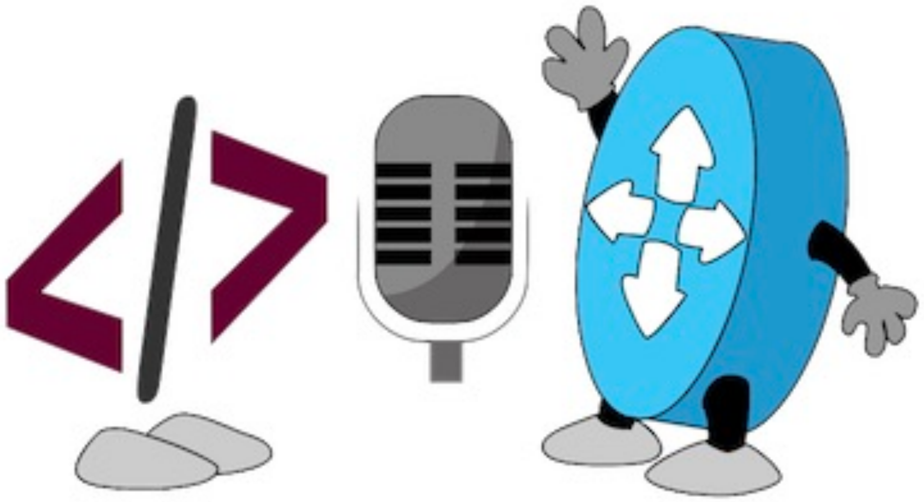
Matt Johnson
Senior Developer Advocate
Twitter: @matjohn2

Season 2, Talk 13

https://developer.cisco.com/netdevops/live

What are we going to talk about?

- Container Network Building Blocks
  - "Why container networking isn't that scary"
- Linux Network Namespaces
- Docker Networking
- Multi-Host Networking
- CNI



Linux as a software switch / router.

- Most of us likely know this is possible.
  - Interfaces
    - Physical
    - Virtual
    - Bridges
  - Routing Tables
    - Static
    - Open source routing protocol implementations (quagga/zebra etc).
  - Firewall (IPTables)
    - Filter / NAT / Mangle
  - QoS
    - tc

Linux as a software switch / router
INTERFACES

```
devbox@li642-77:~$ ip addr list
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
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether f2:3c:91:a9:46:49 brd ff:ff:ff:ff:ff:ff
    inet 212.71.255.77/24 brd 212.71.255.255 scope global dynamic eth0
       valid_lft 45405sec preferred_lft 45405sec
    inet6 2a01:7e00::f03c:91ff:fea9:4649/64 scope global dynamic mngtmpaddr noprefixroute
       valid_lft 14397sec preferred_lft 3597sec
    inet6 fe80::f03c:91ff:fea9:4649/64 scope link
       valid_lft forever preferred_lft forever
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:26:0c:b2:20 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::42:26ff:fe0c:b220/64 scope link
       valid_lft forever preferred_lft forever
5: veth53698e6@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker0 state UP group default
    link/ether ba:7d:75:56:2d:de brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::b87d:75ff:fe56:2dde/64 scope link
       valid_lft forever preferred_lft forever
```

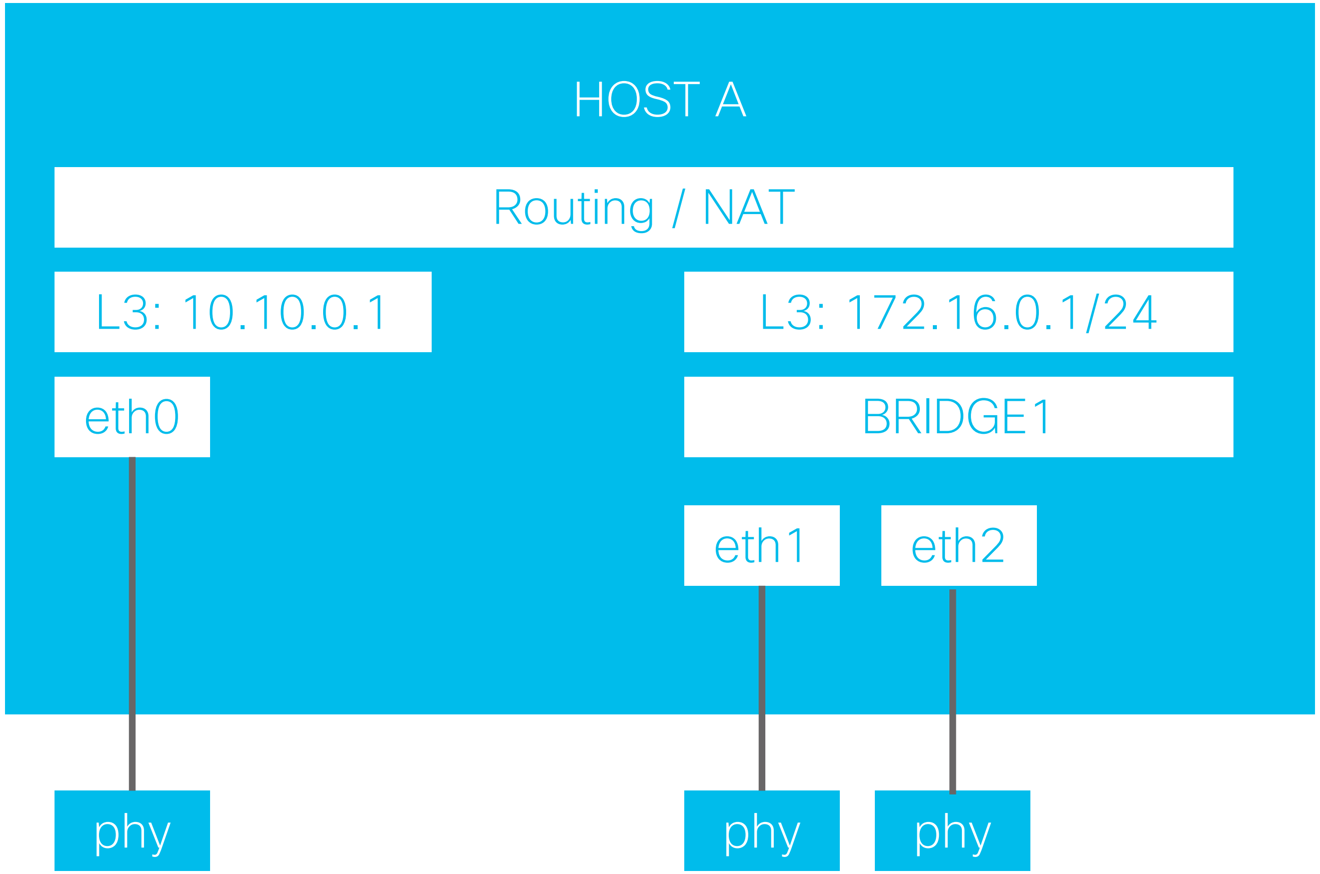
Linux as a software switch / router
BRIDGES

```
devbox@li642-77:~$ sudo ip link add  brtest1 type dummy
devbox@li642-77:~$ sudo ip link add  brtest2 type dummy
```

```
devbox@li642-77:~$ sudo brctl addbr demobr1
devbox@li642-77:~$ sudo brctl addif demobr1 brtest1 brtest2
devbox@li642-77:~$ sudo brctl show demobr1
bridge name     bridge id               STP enabled     interfaces
demobr1         8000.2a6a16177dc5       no              brtest1
                                                        brtest2
```

```
devbox@li642-77:~$ sudo brctl showmacs demobr1
port no mac addr                is local?       ageing timer
  2     2a:6a:16:17:7d:c5       yes                0.00
  2     2a:6a:16:17:7d:c5       yes                0.00
  1     92:0f:58:6a:5c:29       yes                0.00
  1     92:0f:58:6a:5c:29       yes                0.00
```

Linux as a software switch / router
BRIDGES

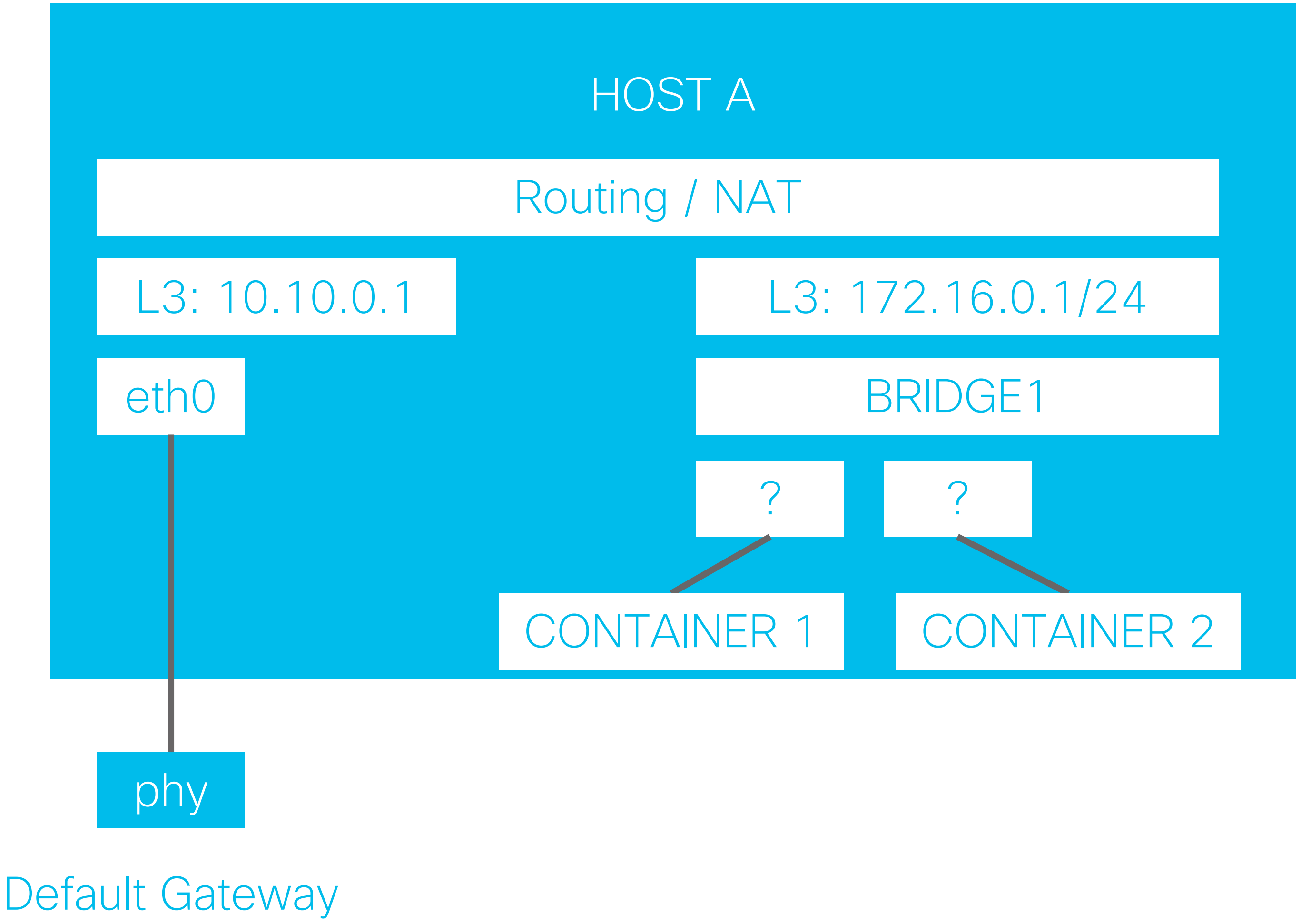


We've made a "switch" on ports eth1&2

We could give 172.16.0.1 as GW
(With some added NAT and routing etc)

Useful for **external clients** (phy interfaces)
What about **internal "clients"**?

Linux as a software switch / router
BRIDGES FOR CONTAINERS



If we could add a container to a bridge as an interface…. This could work.

We could give containers IP's in
172.16.0.0/24 range
172.16.0.1 as GW
(With some added NAT and routing etc)

Linux as a software switch / router
BRIDGES FOR CONTAINERS

```
devbox@li642-77:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.0242260cb220       no              veth53698e6
```

```
devbox@li642-77:~$ brctl showmacs docker0
port no mac addr                is local?       ageing timer
  1     ba:7d:75:56:2d:de       yes                0.00
  1     ba:7d:75:56:2d:de       yes                0.00
```

```
devbox@li642-77:~$ ip route list
default via 212.71.255.1 dev eth0 proto dhcp src 212.71.255.77 metric 1024
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
212.71.255.0/24 dev eth0 proto kernel scope link src 212.71.255.77
212.71.255.1 dev eth0 proto dhcp scope link src 212.71.255.77 metric 1024
```

Introducing vETH pairs.

The veth devices are virtual Ethernet devices.

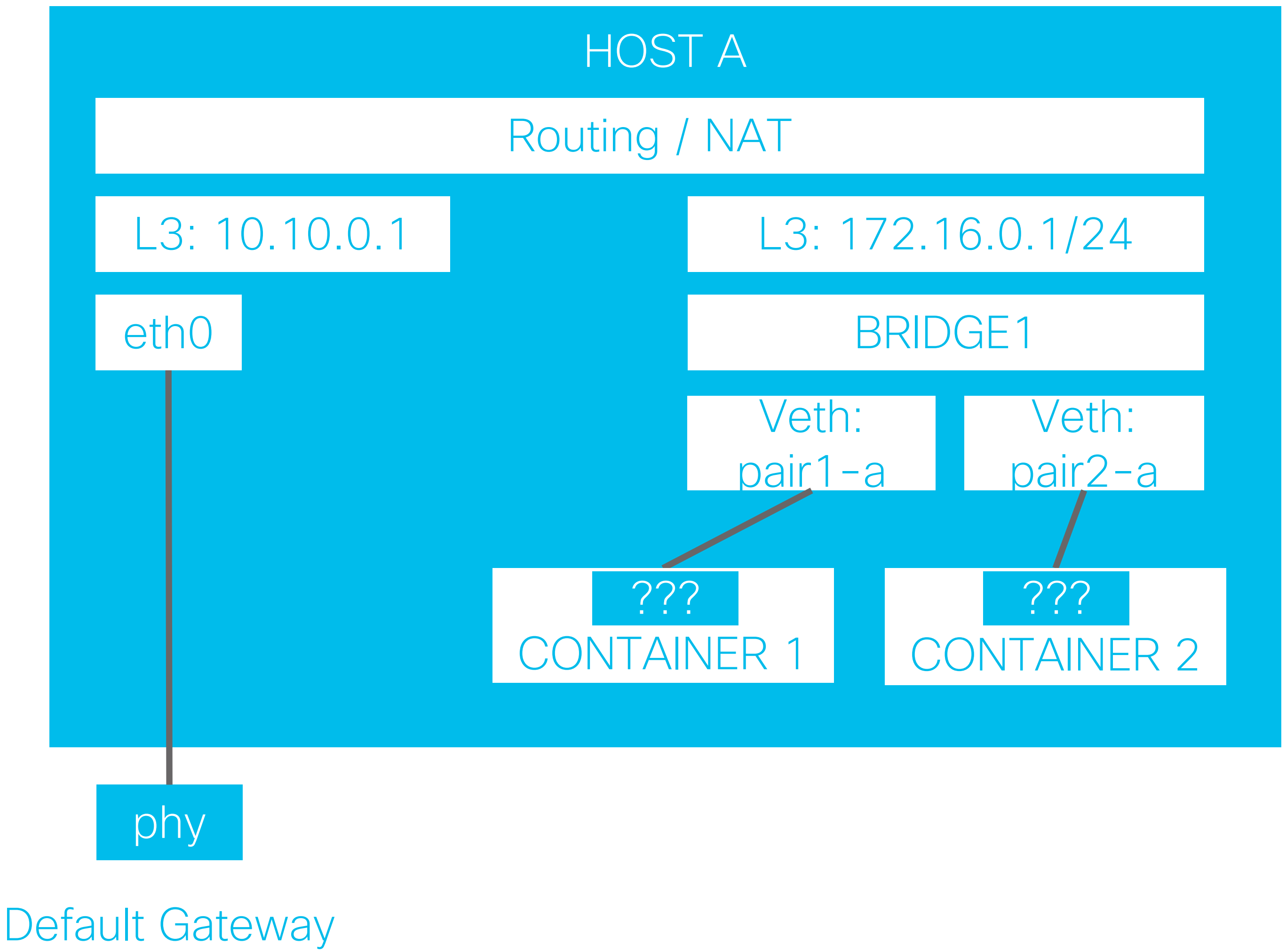
veth devices are always created in interconnected pairs.

A pair can be created using the command:
 # ip link add <p1name> type veth peer name <p2-name>
where, *p1-name* and *p2-name* are the names assigned to the two connected end points.

Packets transmitted on one device in the pair are immediately received on the other device. When either devices is down the link state of the pair is down.

http://man7.org/linux/man-pages/man4/veth.4.html

Linux as a software switch / router
BRIDGES FOR CONTAINERS



Create a vETH pair for each container.
Add one ”end” of the pair to our bridge.

Give the other end an IP address compatible with the bridge subnet.

Test traffic from pair1-b gets to the bridge BVI

Linux as a software switch / router
BRIDGES FOR CONTAINERS

```
devbox@li642-77:~$ sudo ip link add pair1-a type veth peer name pair1-b
```

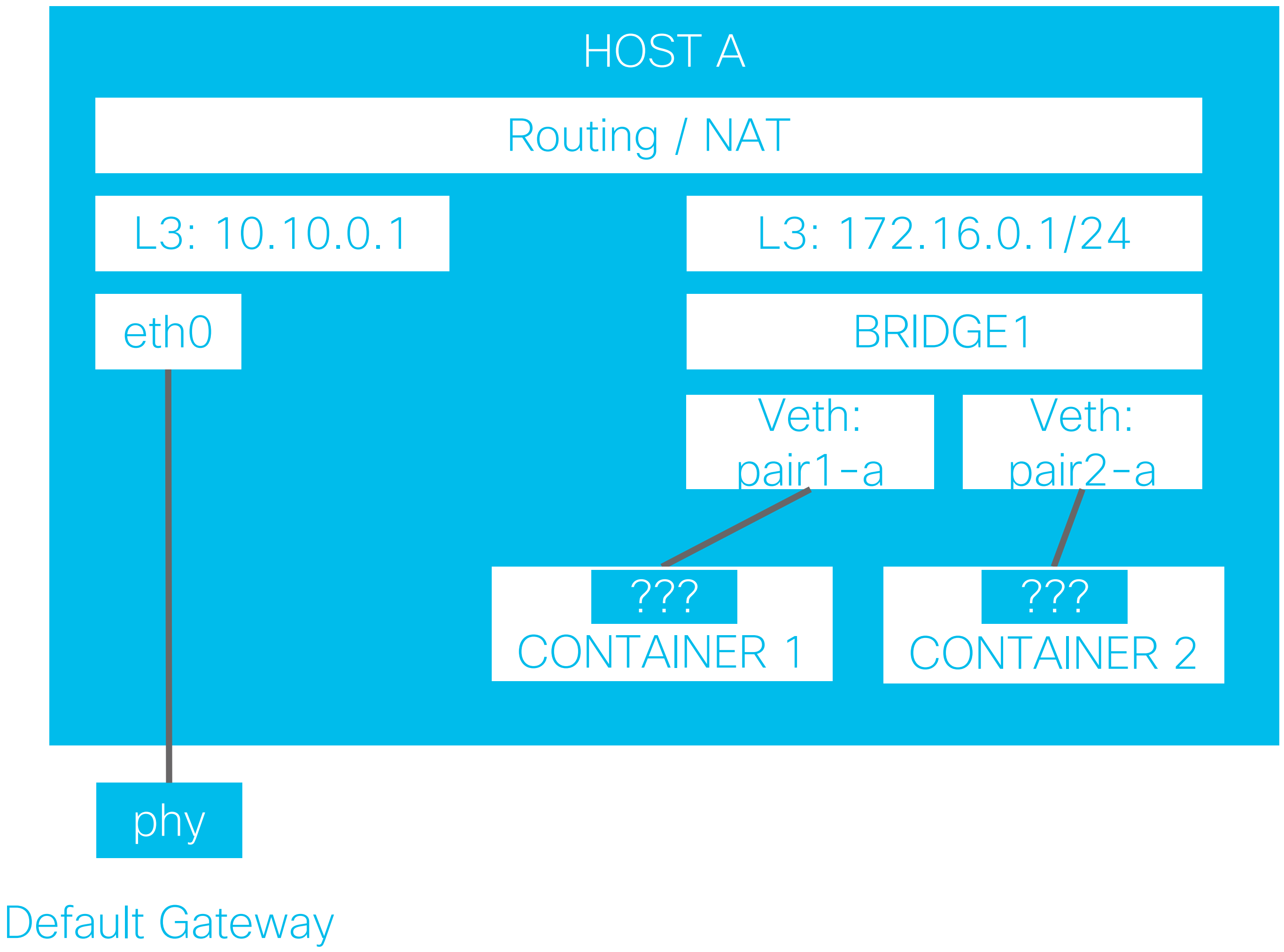
```
devbox@li642-77:~$ sudo brctl addif demobr1 pair1-a
devbox@li642-77:~$ sudo brctl show demobr1
bridge name     bridge id               STP enabled     interfaces
demobr1         8000.2a6a16177dc5       no              brtest1
                                                        brtest2
                                                        pair1-a
```

```
devbox@li642-77:~$ sudo ip link set up pair1-b
devbox@li642-77:~$ sudo ip link set up pair1-a
```

```
devbox@li642-77:~$ ping 172.16.0.1 -I pair1-b
PING 172.16.0.1 (172.16.0.1) from 172.16.0.2 pair1-b: 56(84) bytes of data.
```

```
devbox@li642-77:~$ sudo ip addr add 172.16.0.2/24 dev pair1-b
devbox@li642-77:~$ sudo tcpdump -n -i demobr1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on demobr1, link-type EN10MB (Ethernet), capture size 262144 bytes
14:23:29.041980 ARP, Request who-has 172.16.0.1 tell 172.16.0.2, length 28
14:23:30.066012 ARP, Request who-has 172.16.0.1 tell 172.16.0.2, length 28
```

Linux as a software switch / router
BRIDGES FOR CONTAINERS



Create a vETH pair for each container.
Add one ”end” of the pair to our bridge.

Give the other end an IP address compatible with the bridge subnet.

Test traffic from pair1-b gets to the bridge BVI

HOW DO WE PUT THE OTHER END INTO OUR CONTAINER?

WHAT IS A *”CONTAINER”* FROM LINUX NETWORKING VIEWPOINT?

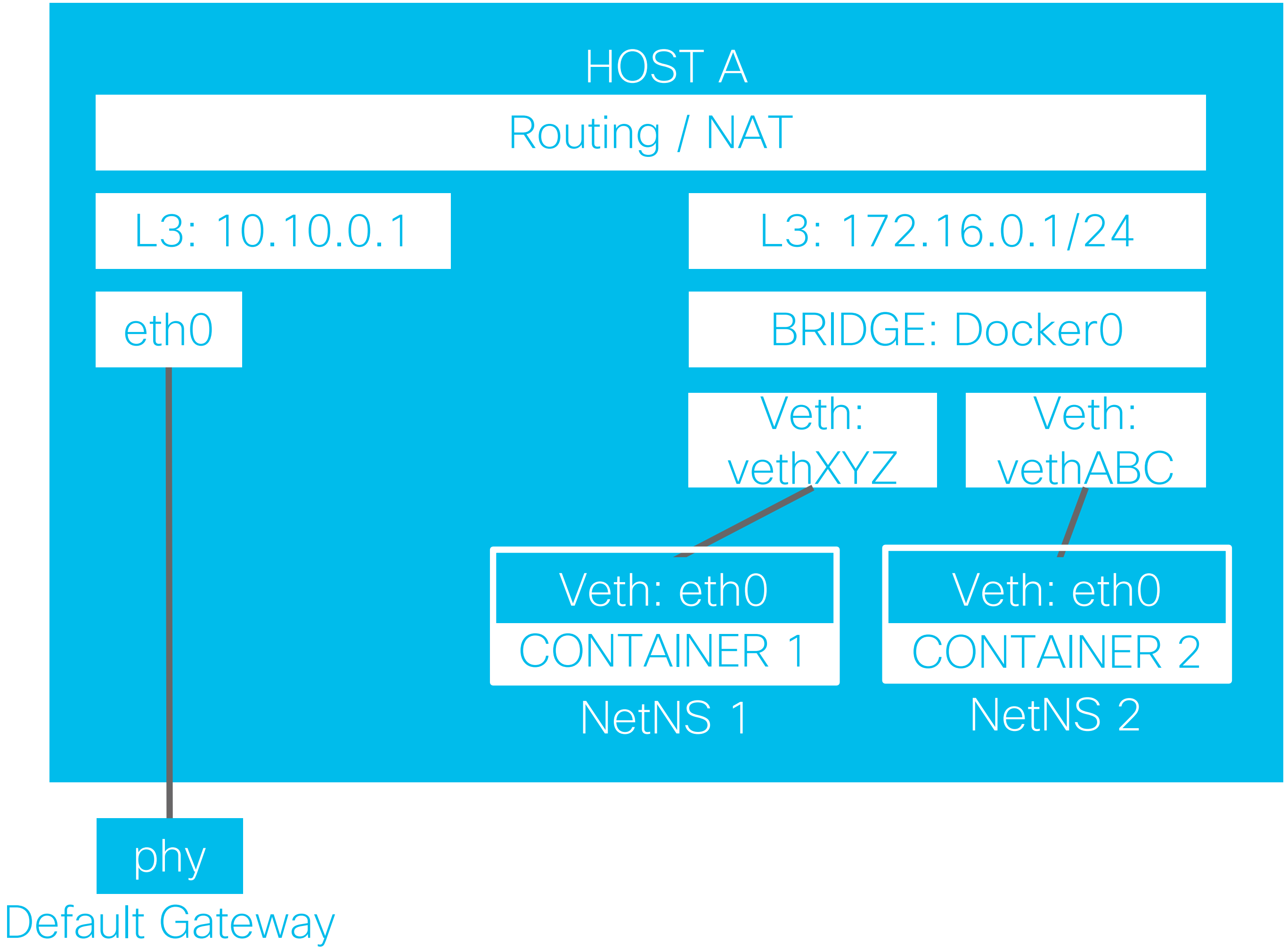
Introducing Network Namespaces … Don't they sound familiar?

- LINUX: A network namespace is logically another copy of the network stack, with its own routes, firewall rules, and network devices. By default a process inherits its network namespace from its parent. Initially all the processes share the same default network namespace from the init process.

- NETWORKS: virtual routing and forwarding (VRF) is a technology that allows multiple instances of a routing table to co-exist within the same router at the same time. One or more logical or physical interfaces may have a VRF and these VRFs do not share routes therefore the packets are only forwarded between interfaces on the same VRF.

DEMO 4: Exploring NetNS with Docker

Linux as a software switch / router
DOCKER; Bridges and NetNS



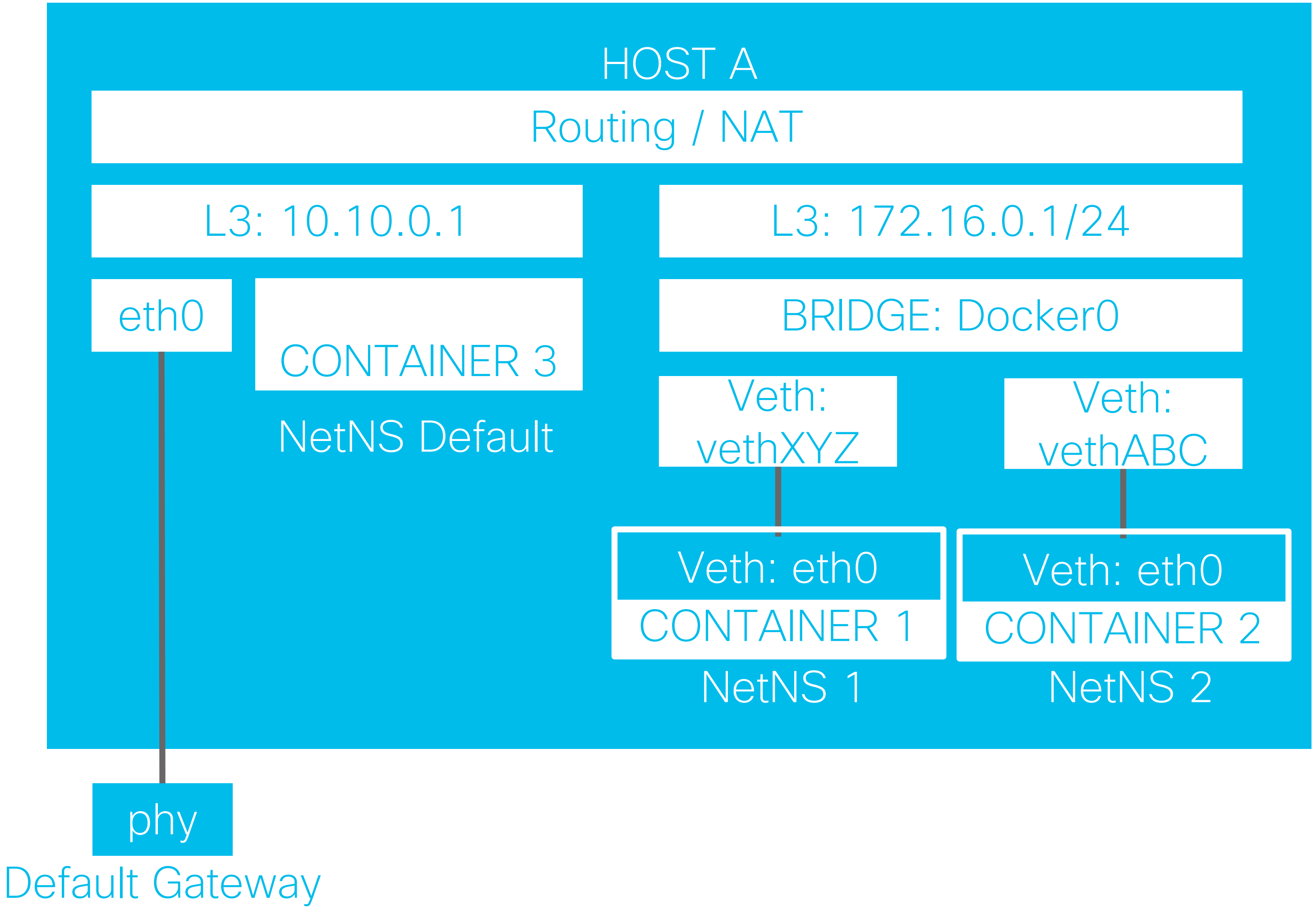
NETNS gives each container a “VRF” where the end of our Veth pair lives.

Containers see “eth0” which is the remote end of the veth pair for that container, on that host.

The remote end of the veth’s are renamed by docker to “eth0” within the container.

DEMO 5 & 6: Messing with the defaults!

Linux as a software switch / router
DOCKER; Bridges and NetNS



NETNS gives each container a “VRF” where the end of our Veth pair lives.

Containers can be placed in the host’s default netns, they will see all the “regular” hosts interface and communicate just as a regular process on the system would.

We can also show there is nothing magic by asking for no networking and doing it ourselves, manually with netns commands.

Docker Networks
"host" "none" etc..

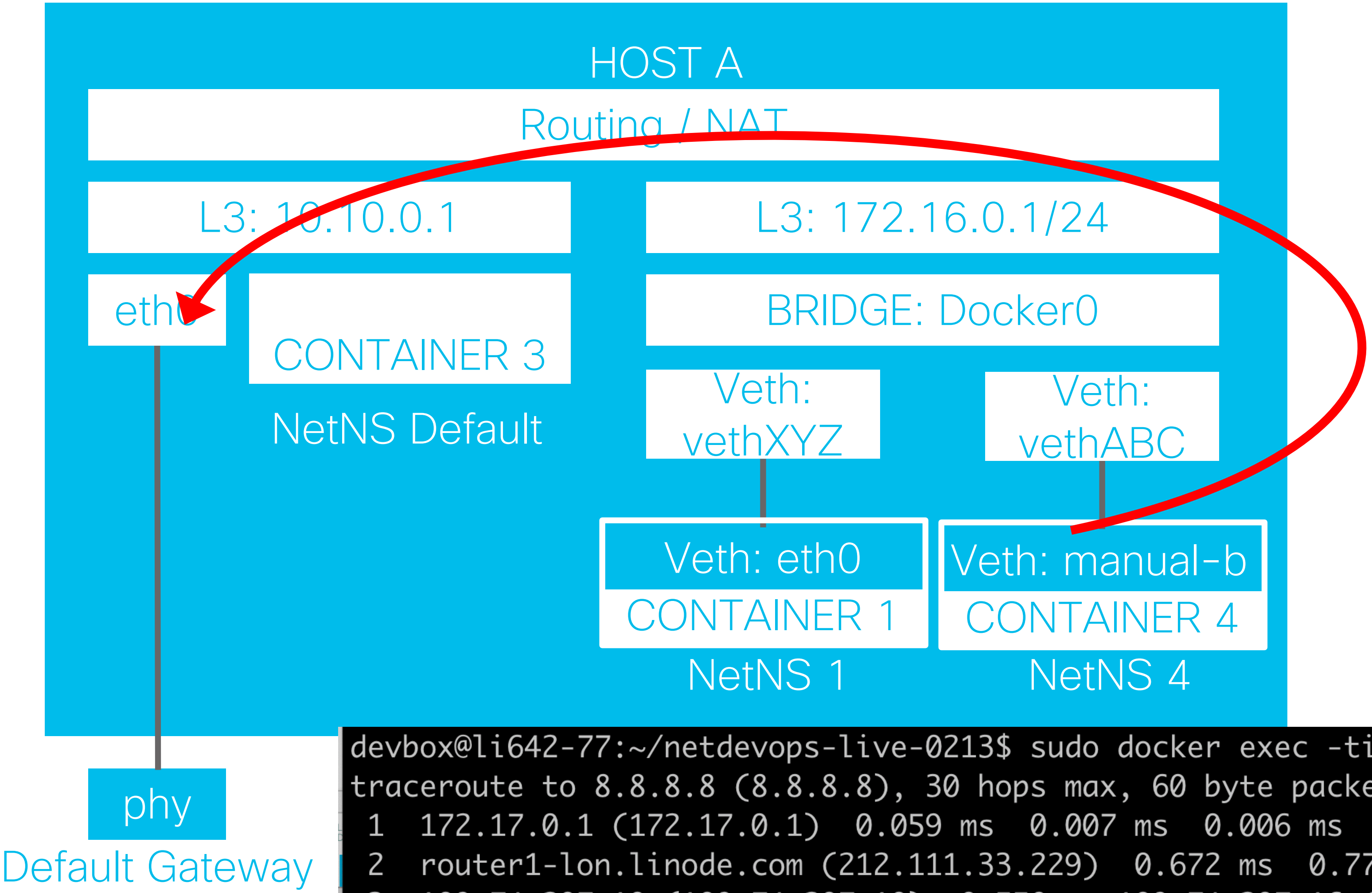
```
devbox@li642-77:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
936f5c268e8f        bridge              bridge              local
9f041bd11a1e        host                host                local
b9533c3efd7a        none                null                local
```

```
devbox@li642-77:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
936f5c268e8f        bridge              bridge              local
9f041bd11a1e        host                host                local
b9533c3efd7a        none                null                local
d8fbbc99ee9c        test                bridge              local
devbox@li642-77:~$
devbox@li642-77:~$
devbox@li642-77:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
br-d8fbbc99ee9c         8000.02423fdb4749       no
docker0         8000.0242260cb220       no
```

Docker Networks
“host” ”none” etc..

```
devbox@li642-77:~$ sudo docker network inspect 936f5c268e8f
[
    {
        "Name": "bridge",
        "Id": "936f5c268e8f48eca920818e5a5335eca9aa48aa3190721c5e545f79a6b40d77",
        "Created": "2019-06-24T01:19:05.228876918Z",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": null,
            "Config": [
                {
                    "Subnet": "172.17.0.0/16",
                    "Gateway": "172.17.0.1"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
```

A question remains. Internet access?



Any IP on our Docker0 bridge seems to have internet access via eth0…

Guess whats' happening?

```
devbox@li642-77:~/netdevops-live-0213$ sudo docker exec -ti container4 traceroute 8.8.8.8
traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
 1  172.17.0.1 (172.17.0.1)  0.059 ms  0.007 ms  0.006 ms
 2  router1-lon.linode.com (212.111.33.229)  0.672 ms  0.772 ms  0.863 ms
 3  109.74.207.10 (109.74.207.10)  0.558 ms 109.74.207.16 (109.74.207.16)  0.796 ms 109.74.207.26 (109.74.20
 4  google1.lonap.net (5.57.80.136)  0.902 ms  0.889 ms  0.886 ms
 5  74.125.242.97 (74.125.242.97)  2.671 ms 74.125.242.65 (74.125.242.65)  1.650 ms 108.170.246.129 (108.170
 6  172.253.68.213 (172.253.68.213)  1.392 ms 172.253.66.99 (172.253.66.99)  2.205 ms 108.170.238.117 (108.1
 7  dns.google (8.8.8.8)  1.264 ms  0.973 ms  1.912 ms
```

Docker Default IPTables NAT

- IPTables Source PAT rule for Docker0 interface.
- You can also see another for the new “docker network” we created.

```
devbox@li642-77:~/netdevops-live-0213$ sudo iptables -t nat -L -v
Chain PREROUTING (policy ACCEPT 705 packets, 40299 bytes)
 pkts bytes target     prot opt in     out     source               destination
 8073  437K DOCKER     all  --  any    any     anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 338 packets, 20284 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 257 packets, 18587 bytes)
 pkts bytes target     prot opt in     out     source               destination
   10   840 DOCKER     all  --  any    any     anywhere            !localhost/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 257 packets, 18587 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  any    !br-d8fbbc99ee9c  172.18.0.0/16        anywhere
  116  7353 MASQUERADE  all  --  any    !docker0  172.17.0.0/16        anywhere

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  br-d8fbbc99ee9c any     anywhere             anywhere
    0     0 RETURN     all  --  docker0 any     anywhere             anywhere
```

Docker Default IPTables NAT

IPTables is pretty common everywhere you will see containers.
If traffic is getting into, or out of an IP address, or your service is being exposed on a port you didn't expect,
Chances are there will be a some DNAT, SNAT rules being generated for you.

Example, exposing a docker container to the outside world…

```
^C^C^C^Cdevbox@li642-77:~/netdevops-live-0213$ sudo docker run --name container5 -p8000:8000 ubuntu:latest sleep 1000000 &
[1] 27273
devbox@li642-77:~/netdevops-live-0213$ sudo iptables -t nat -L -v
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
 8089  437K DOCKER     all  --  any    any     anywhere             anywhere             ADDRTYPE match dst-type LOCAL

Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination

Chain OUTPUT (policy ACCEPT 1 packets, 73 bytes)
 pkts bytes target     prot opt in     out     source               destination
   10   840 DOCKER     all  --  any    any     anywhere            !localhost/8          ADDRTYPE match dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 1 packets, 73 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MASQUERADE  all  --  any    !br-d8fbbc99ee9c  172.18.0.0/16        anywhere
  116  7353 MASQUERADE  all  --  any    !docker0  172.17.0.0/16        anywhere
    0     0 MASQUERADE  tcp  --  any    any     172.17.0.2           172.17.0.2           tcp dpt:8000

Chain DOCKER (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 RETURN     all  --  br-d8fbbc99ee9c any     anywhere             anywhere
    0     0 RETURN     all  --  docker0 any     anywhere             anywhere
    0     0 DNAT       tcp  --  !docker0 any     anywhere             anywhere             tcp dpt:8000 to:172.17.0.2:8000
devbox@li642-77:~/netdevops-live-0213$
```

IPTables for security

IPTables is also commonly used to enforce security policy especially in multi-host clustered container environments. This is usually a central control plane (container orchestrator) deciding which IPTables rules to apply on which hosts to protect the containers running there.

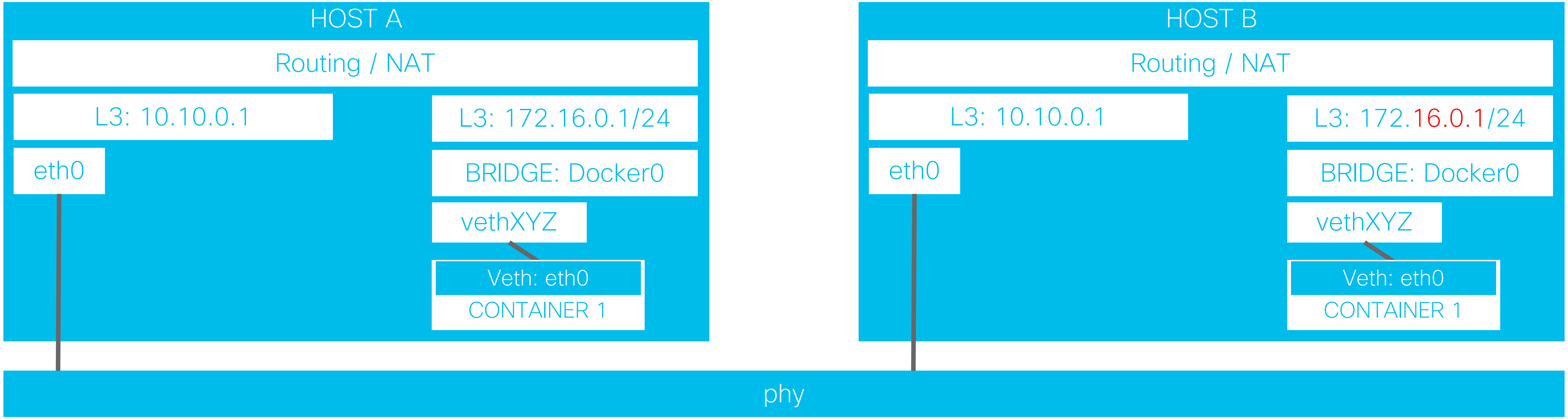
```
Chain DOCKER (2 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.17.0.2           tcp dpt:8000

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
target     prot opt source               destination
DOCKER-ISOLATION-STAGE-2  all  --  anywhere             anywhere
DOCKER-ISOLATION-STAGE-2  all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere

Chain DOCKER-ISOLATION-STAGE-2 (2 references)
target     prot opt source               destination
DROP       all  --  anywhere             anywhere
DROP       all  --  anywhere             anywhere
RETURN     all  --  anywhere             anywhere

Chain DOCKER-USER (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             anywhere
```

Multi-Host.

Multi-Host: How

Your container is a VRF, with a connection out to a bridge, with a L3 BVI, how would you make it multi-host?

Now we know what the foundations of “container networking” are, Implementations for moving beyond single host docker should be apparent.



Multi-Host: How

L2 VLAN to span Docker0 Bridge

- Hairpin L3 Routing Concerns
- Broadcast Domain
- IP addressing (would need central state)

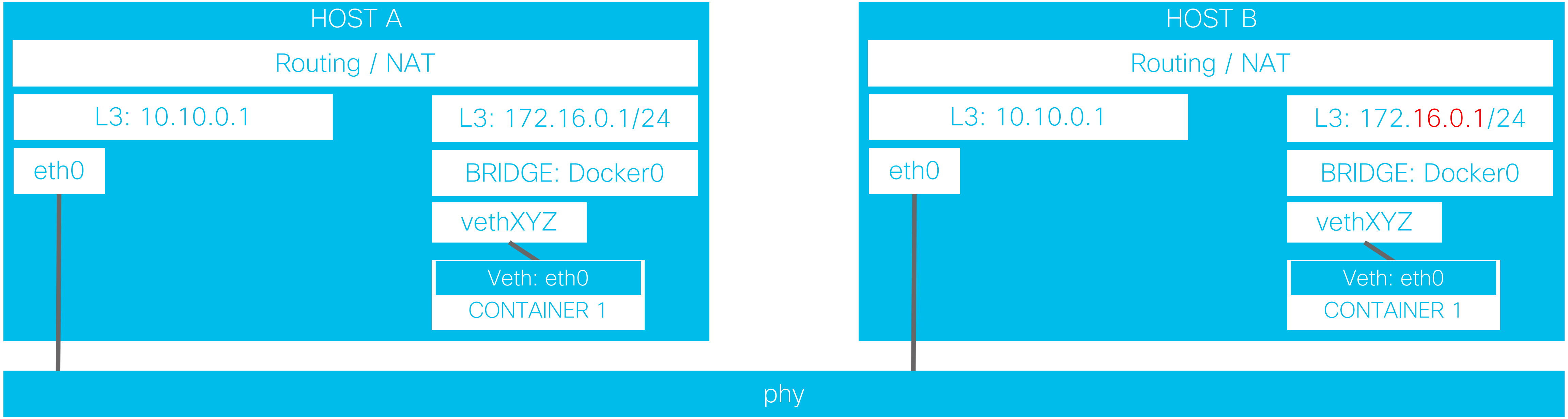
Existing Network Environment (VLAN-in-VLAN?)

Multi-Tenancy?

GRE Tunnels

L3 Routes to each host, with a separate docker0 subnet on each

- Managing routing tables
- Static vs Routing Protocol
- Managing hosts to be networked (state)



Multi-Host: Solutions

One size does not fit all.

Pluggable solutions

Flannel
(https://coreos.com/flannel/docs/latest/running.html)

Calico
(https://docs.projectcalico.org)

Weave
(https://www.weave.works/docs/net/latest/overview/)

Contiv (ACI)
(https://contiv.io/documents/networking/aci_ug.html)

Recommended backends

VXLAN

Use in-kernel VXLAN to encapsulate the packets.

Type and options:

- `Type` (string): `vxlan`
- `VNI` (number): VXLAN Identifier (VNI) to be used. Defaults to 1.
- `Port` (number): UDP port to use for sending encapsulated packets. Defa
currently 8472.
- `GBP` (Boolean): Enable VXLAN Group Based Policy. Defaults to `false`.
- `DirectRouting` (Boolean): Enable direct routes (like `host-gw`) when the
subnet. VXLAN will only be used to encapsulate packets to hosts on diffe
to `false`.

host-gw

Use host-gw to create IP routes to subnets via remote machine IPs. Requires
connectivity between hosts running flannel.

host-gw provides good performance, with few dependencies, and easy set u

Type:

- `Type` (string): `host-gw`

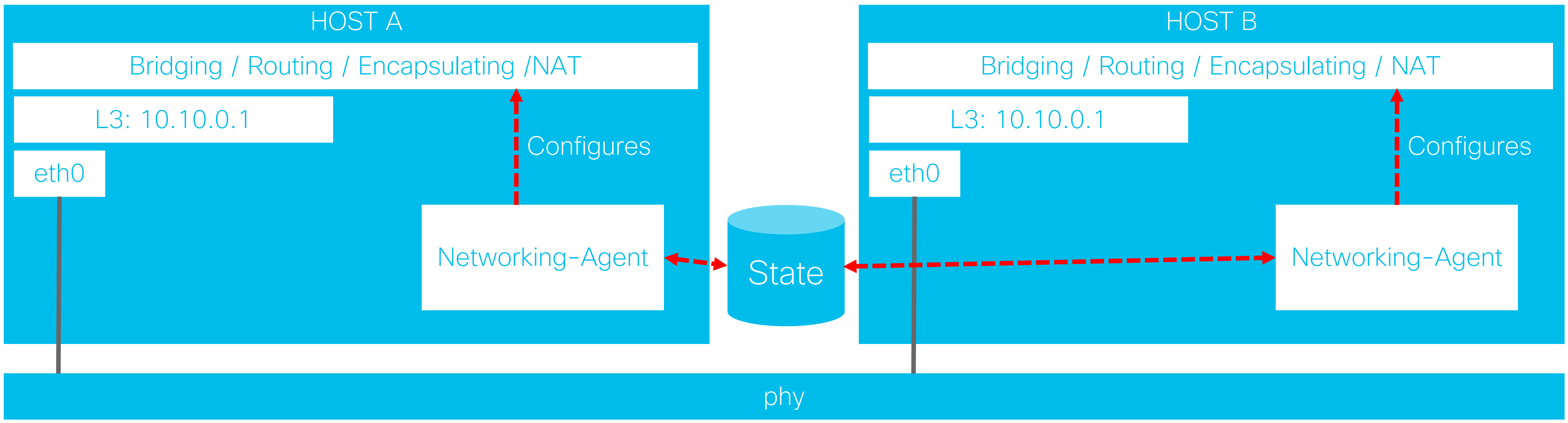
https://coreos.com/flannel/docs/latest/backends.htr

Multi-Host: Solutions

Most solutions run some form of agent on each host, talking to a central data store.

The agent inserts/configures connectivity to other hosts and maintains IP addressing.

Weave-Mesh does not need a state store, but does require direct L2 connectivity between all nodes.

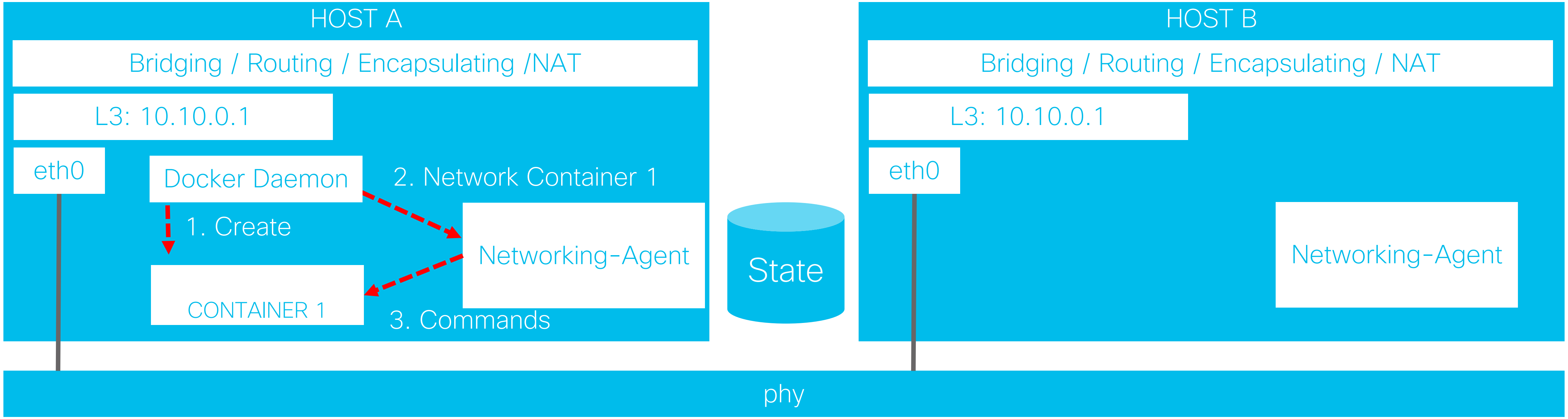


Multi-Host: Solutions

Plugin packaging / format defined by Container solution, in this case, Docker.

”What should I do to network this container”
“What should I do to include this host”

*Solution has the flexibility to *not* be software.*
EG. ACI plugin mapping VXLAN tag to an EPG



Calico – BGP vs Layers of Tunnels.

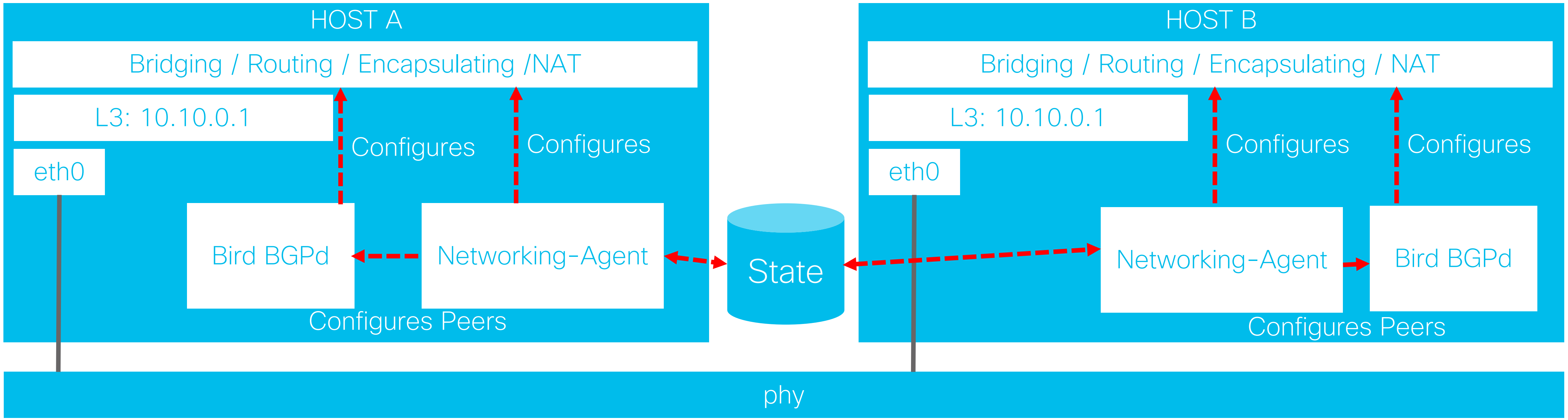
No overlay required

Why add another layer of overhead when you don't need it?

Sometimes, an overlay network (encapsulating packets inside an extra IP header) is necessary. Often, though, it just adds unnecessary overhead, resulting in multiple layers of nested packets, impacting performance and complicating trouble-shooting. Wouldn't it be nice if your virtual networking solution adapted to the underlying infrastructure, using an overlay only when required? That's what Calico does. In most environments, Calico simply routes packets from the workload onto the underlying IP network without any extra headers. Where an overlay is needed – for example when crossing availability zone boundaries in public cloud – it can use lightweight encapsulation including IP-in-IP and VxLAN. Project Calico even supports both IPv4 and IPv6 networks!

https://docs.projectcalico.org/v2.6/usage/configuration/bgp

Calico BGP



Configuring BGP Peers

This document describes the commands available in `calicoctl` for managing BGP. It is intended primarily for users who are running on private cloud and would like to peer Calico with their underlying infrastructure.

This document covers configuration of:

- Global default node AS Number
- The full node-to-node mesh
- Global BGP Peers

https://docs.projectcalico.org/v2.6/usage/configuration/bgp

Modularity: CNI

Usually, you'll be running a container orchestrator.

CNI: Container Network Interface

Plugin standard for networking plugins

Compatible with Kubernetes

All solutions discussed provide CNI plugins

Docker itself uses a different plugin mechanism

https://github.com/containernetworking/cni

Who is using CNI?

Container runtimes

- rkt - container engine
- Kubernetes - a system to simplify container operations
- OpenShift - Kubernetes with additional enterprise featur
- Cloud Foundry - a platform for cloud applications
- Apache Mesos - a distributed systems kernel
- Amazon ECS - a highly scalable, high performance conta
- Singularity - container platform optimized for HPC, EPC,
- OpenSVC - orchestrator for legacy and containerized ap

Modularity: Kubernetes NetworkPlugin

Network Policies

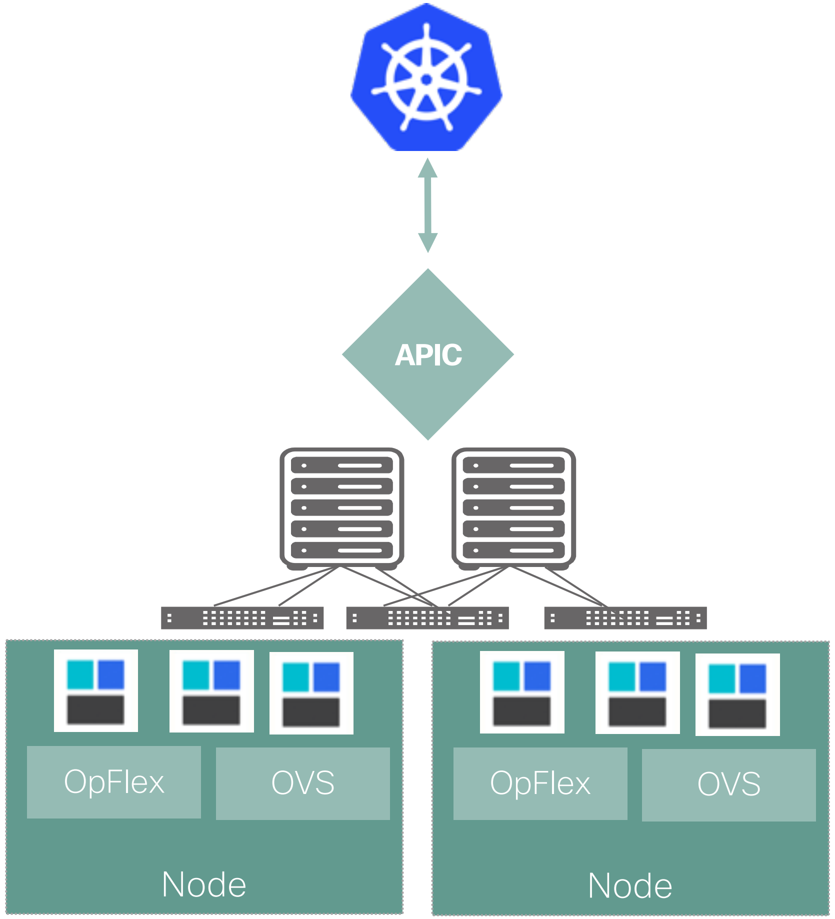
A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

`NetworkPolicy` resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods.

Prerequisites

Network policies are implemented by the network plugin, so you must be using a networking solution which supports `NetworkPolicy` - simply creating the resource without a controller to implement it will have no effect.

Cisco ACI CNI for Container Integration



ACI and Containers

- Unified networking: Containers, VMs, and bare-metal
- Micro-services load balancing integrated in fabric for HA / performance
- Secure multi-tenancy and seamless integration of Kubernetes network policies and ACI policies
- Visibility: Live statistics in APIC per container and health metrics

See Season 1, Episode 7 for more details! https://developer.cisco.com/netdevops/live/#s01t07

Industry Developments

Future Developments & State of the art

kubernetes

Documentation Blog Partners Community Case Studies English v1.15

Using eBPF in Kubernetes

Thursday, December 07, 2017

Using eBPF in Kubernetes

Introduction

Kubernetes provides a high-level API and a set of components that hides almost all of the intricate and—to some of us—interesting details of what happens at the systems level. Application developers are not required to have knowledge of the machines' IP tables, cgroups, namespaces, seccomp, or, nowadays, even the container runtime that their application runs on top of. But underneath, Kubernetes and the technologies upon which it relies (for example, the container runtime) heavily leverage core Linux functionalities.

This article focuses on a core Linux functionality increasingly used in networking, security and auditing, and tracing and monitoring tools. This functionality is called extended Berkeley Packet Filter (eBPF)

Note: *In this article we use both acronyms: eBPF and BPF. The former is used for the extended BPF functionality, and the latter for "classic" BPF functionality.*

What is BPF?

BPF is a mini-VM residing in the Linux kernel that runs BPF programs. Before running, BPF programs are loaded with the bpf() syscall and are validated for safety: checking for loops, code size, etc. BPF programs are attached to kernel objects and executed when events happen on those objects—for example, when a network interface emits a packet.

RSS Feed

Submit a Post

@Kubernetesio

View on GitHub

#kubernetes-users

Stack Overflow

Forum

Download Kubernetes

Future Developments & State of the art

Kubefed (V2) – Federated Kubernetes

Edge / Fog / Remote location workloads

- AutoVPN
- SDWAN
- Potential for CNI or workload integration

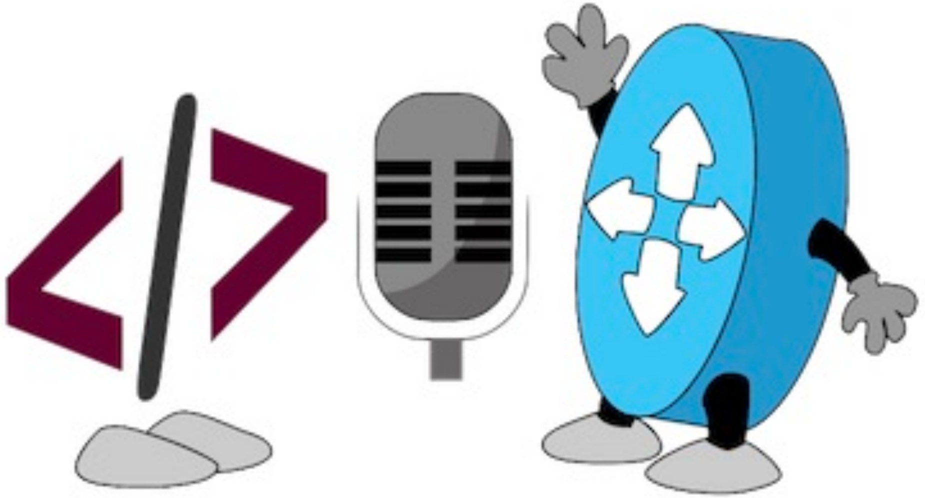
Service Mesh

- Still relies on a underlying IP fabric

Summing up

What did we talk about?

- Linux Networking
- How Linux Networking became container Networking
  - Namespaces
  - vETH
- Scaling to multiple hosts
- Pluggability and CNI



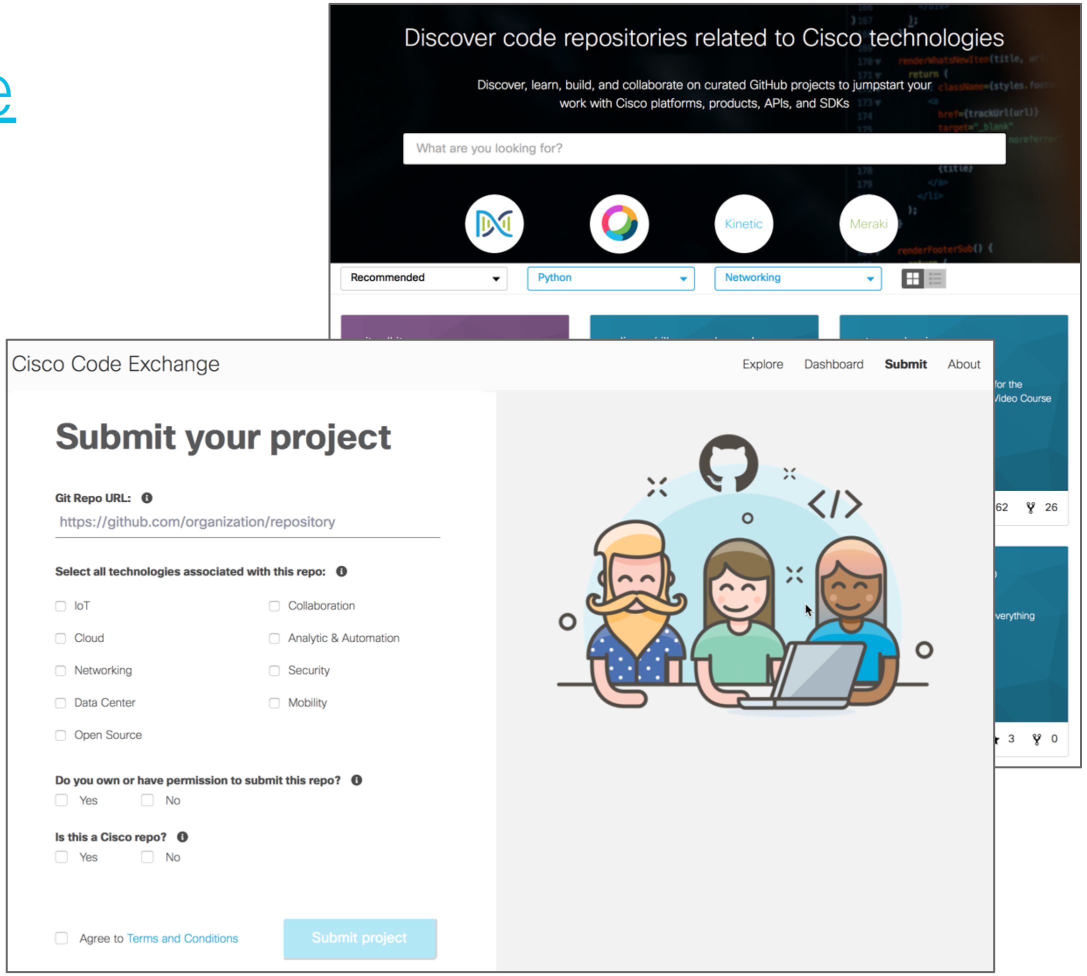
Webinar Resource List

- Learning Labs
  - Microservices and Containers Intro DEVNET Module https://developer.cisco.com/learning/modules/cloud-native-development
- DevNet Sandboxes
  - Kubernetes CNI/ACI Sandbox http://cs.co/sbx-acik8s
- Code Samples and CLI Snippets
  - https://github.com/ciscodevnet/netdevops-live-0213/

NetDevOps Live! Code Exchange Challenge

developer.cisco.com/codeexchange

"Containerize" your favorite network automation script for easier portability!



Example: Include a Dockerfile in the repo that describes and installs all necessary Python dependencies.

Looking for more about NetDevOps?

- NetDevOps on DevNet
  developer.cisco.com/netdevops
- NetDevOps Live!
  developer.cisco.com/netdevops/live
- NetDevOps Blogs
  blogs.cisco.com/tag/netdevops
- Network Programmability Basics Video Course
  developer.cisco.com/video/net-prog-basics/

Got more questions? Stay in touch!

DEVNET

LEARN CODE INSPIRE CONNECT

developer.cisco.com

@mattdashj

https://github.com/matjohn2

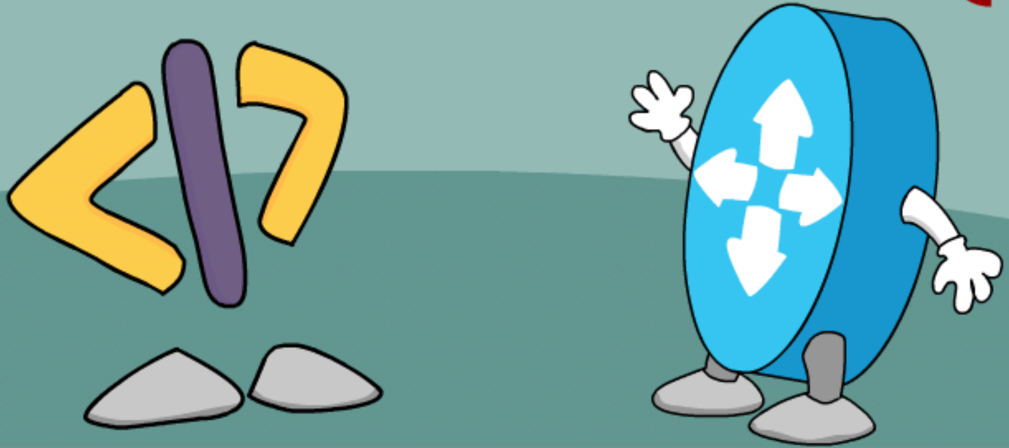
@CiscoDevNet

facebook.com/ciscodevnet/

http://github.com/CiscoDevNet

NETDEVOPS {LIVE!}

CISCO DEVNET



https://developer.cisco.com/netdevops/live

@netdevopslive