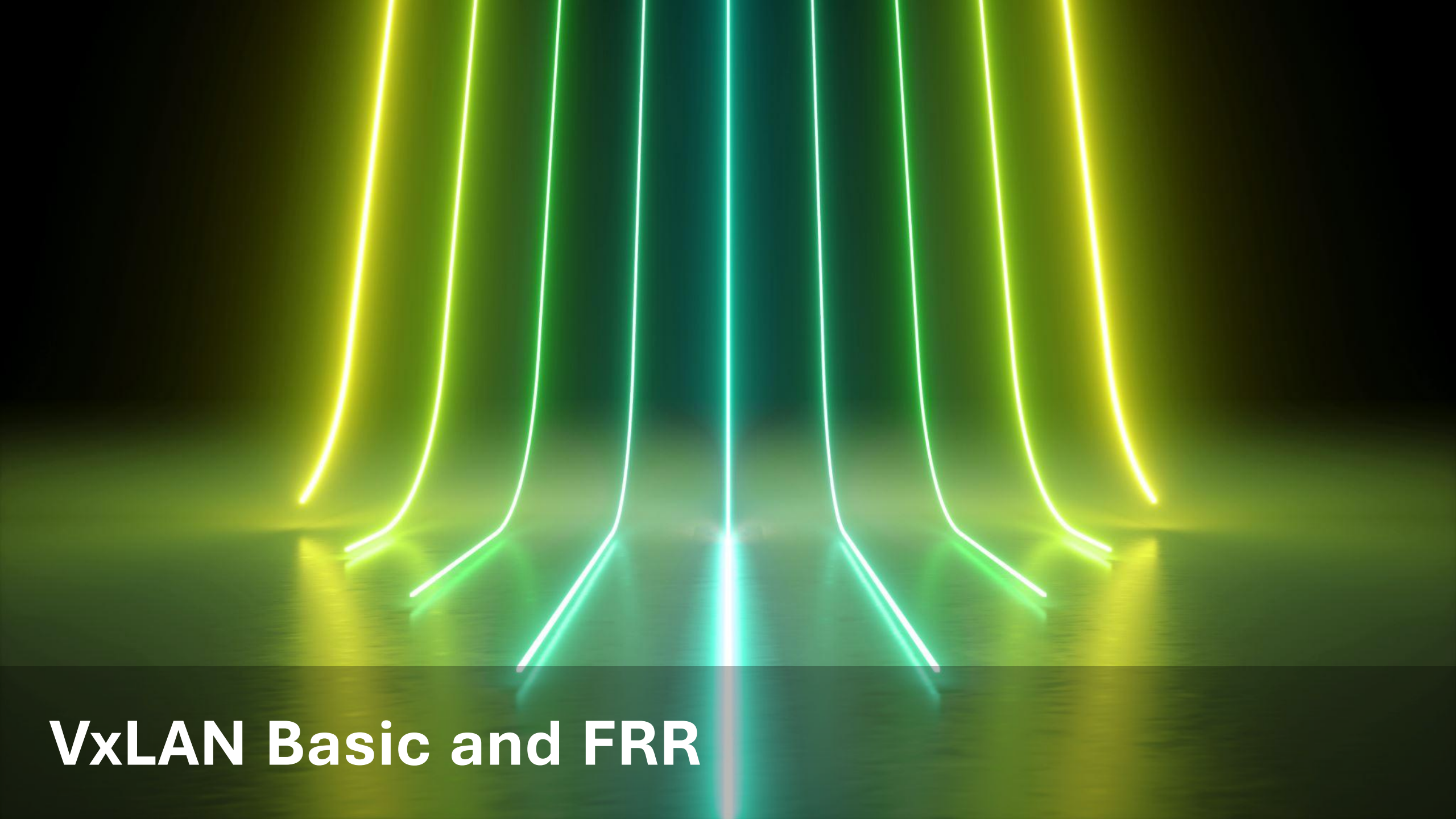
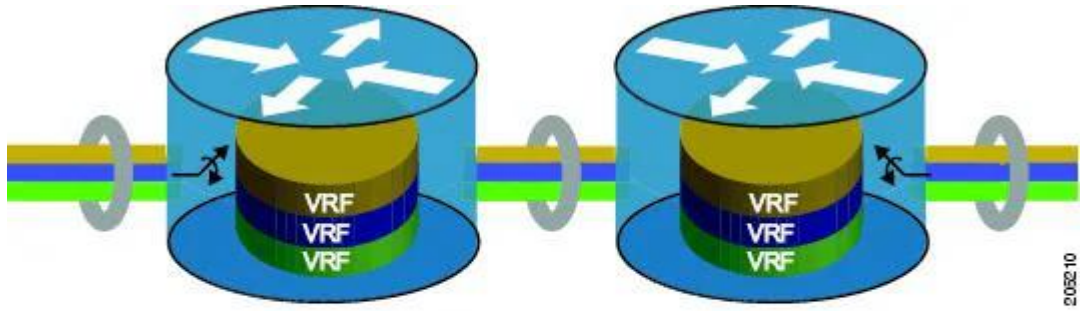
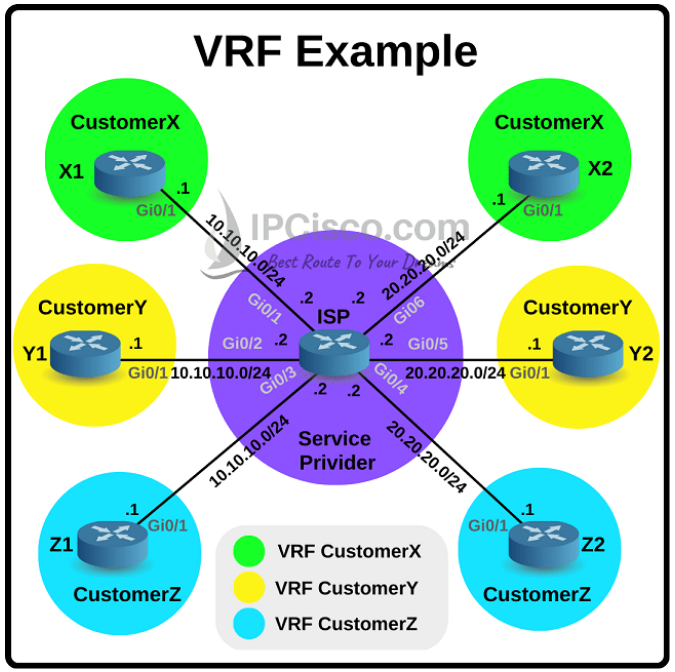


VxLAN Basic and FRR

Virtual Routing and Forwarding (VRF)

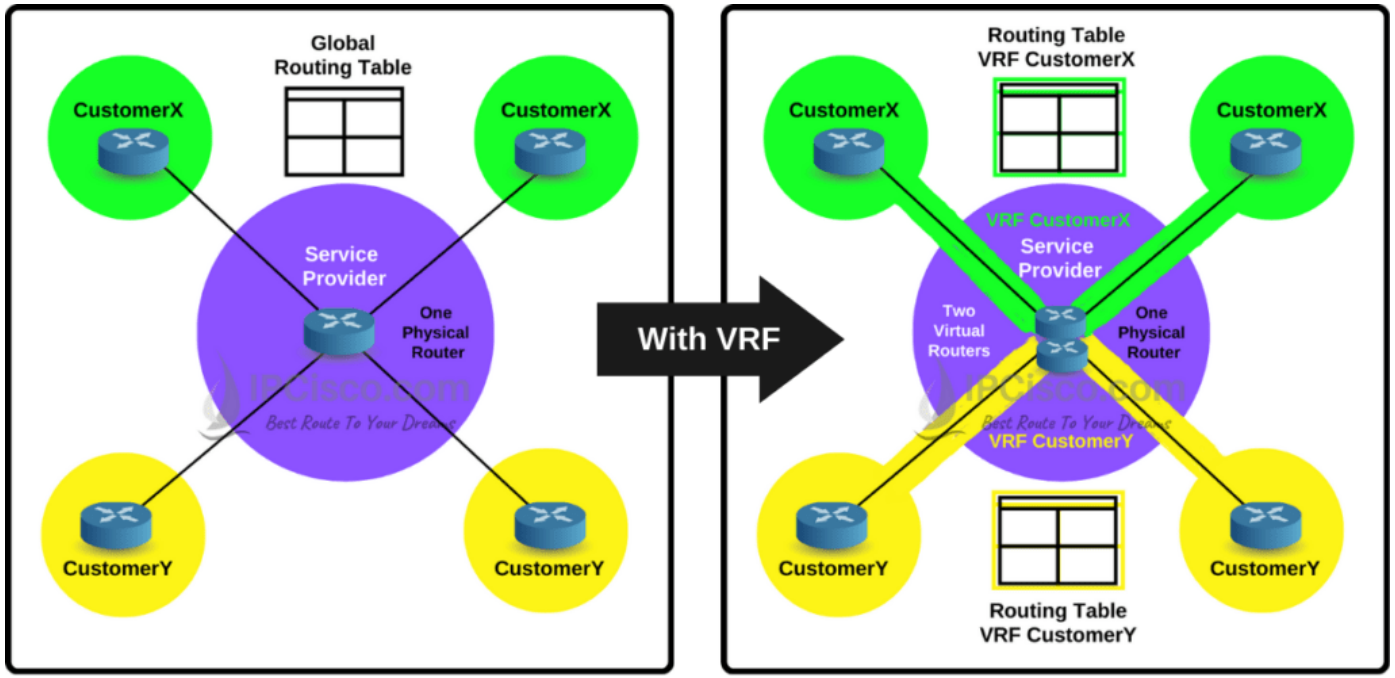
- A technique which creates multiple virtual networks within a single network entity (Figure 1). In a single network component, multiple VRF resources create the isolation between virtual networks.





VRF and VLAN

- **VRFs** are similar to VLANs used in switches at layer 2 of the OSI Reference Model. But VRF technology works at **Layer 3**. As you know, with VLANs, we can create multiple virtual switches in a physical switch. With VRFs, we can create multiple routers in a physical router.



https://ipcisco.com/lesson/cisco-virtual-routing-and-forwarding-vrf/

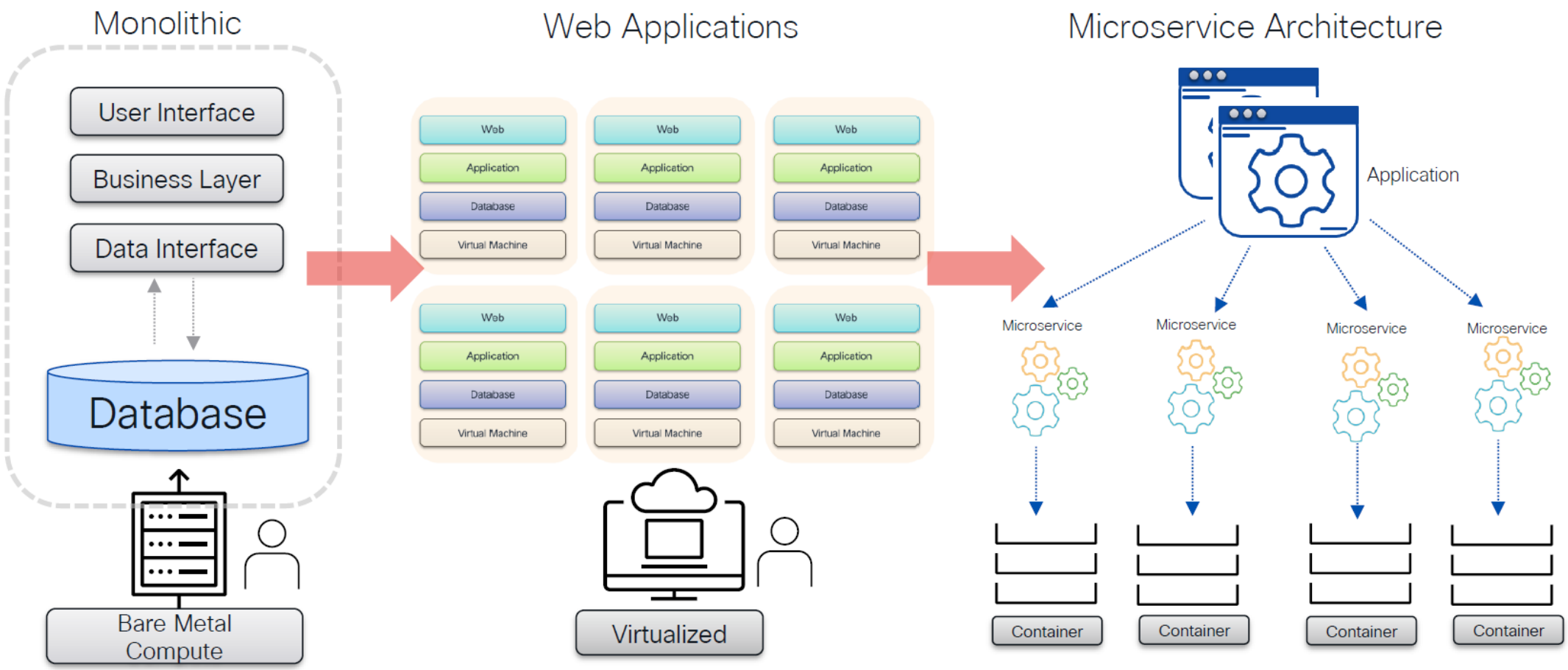
Benefits of VRFs

- With VRFs, we can create multiple virtual routers in a single physical router with isolated routing and forwarding.
- With VRFs, we can use overlapping IP addresses on created separate virtual routers.
- With separate and small routing tables, VRFs provide easy troubleshooting.
- Useful for MP-BGP and MPLS deployments.

VxLAN

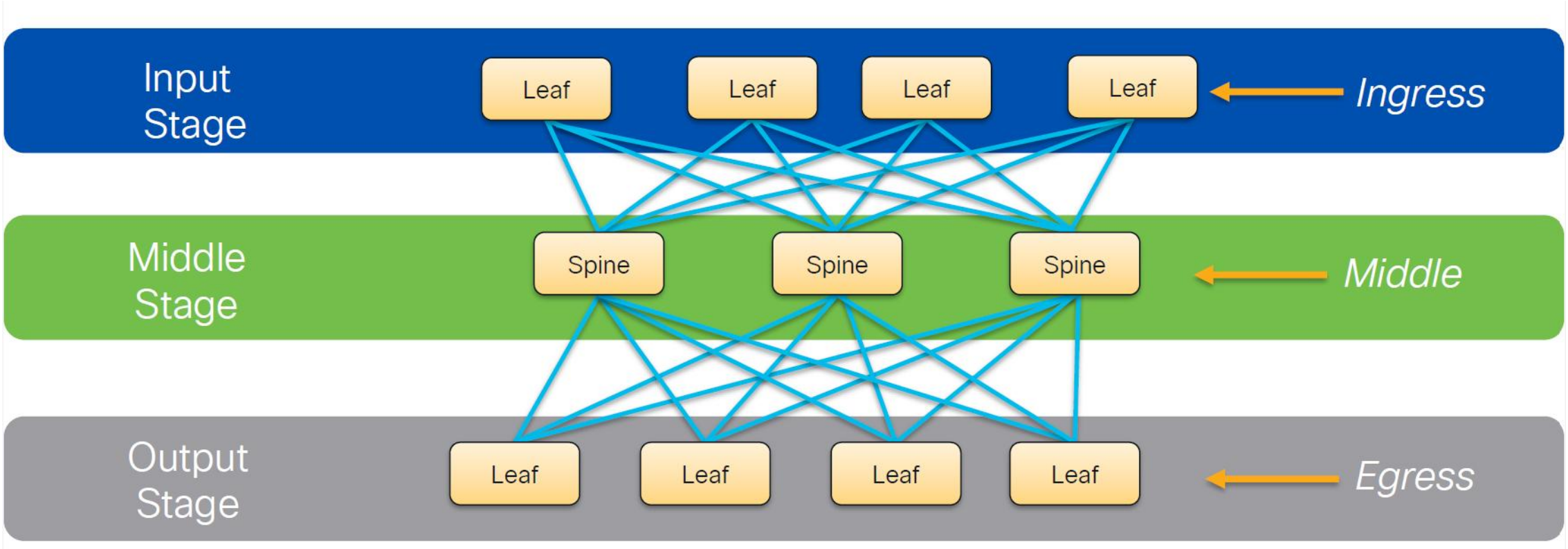
BRKDCN-1621

Evolution of the Application Architectures



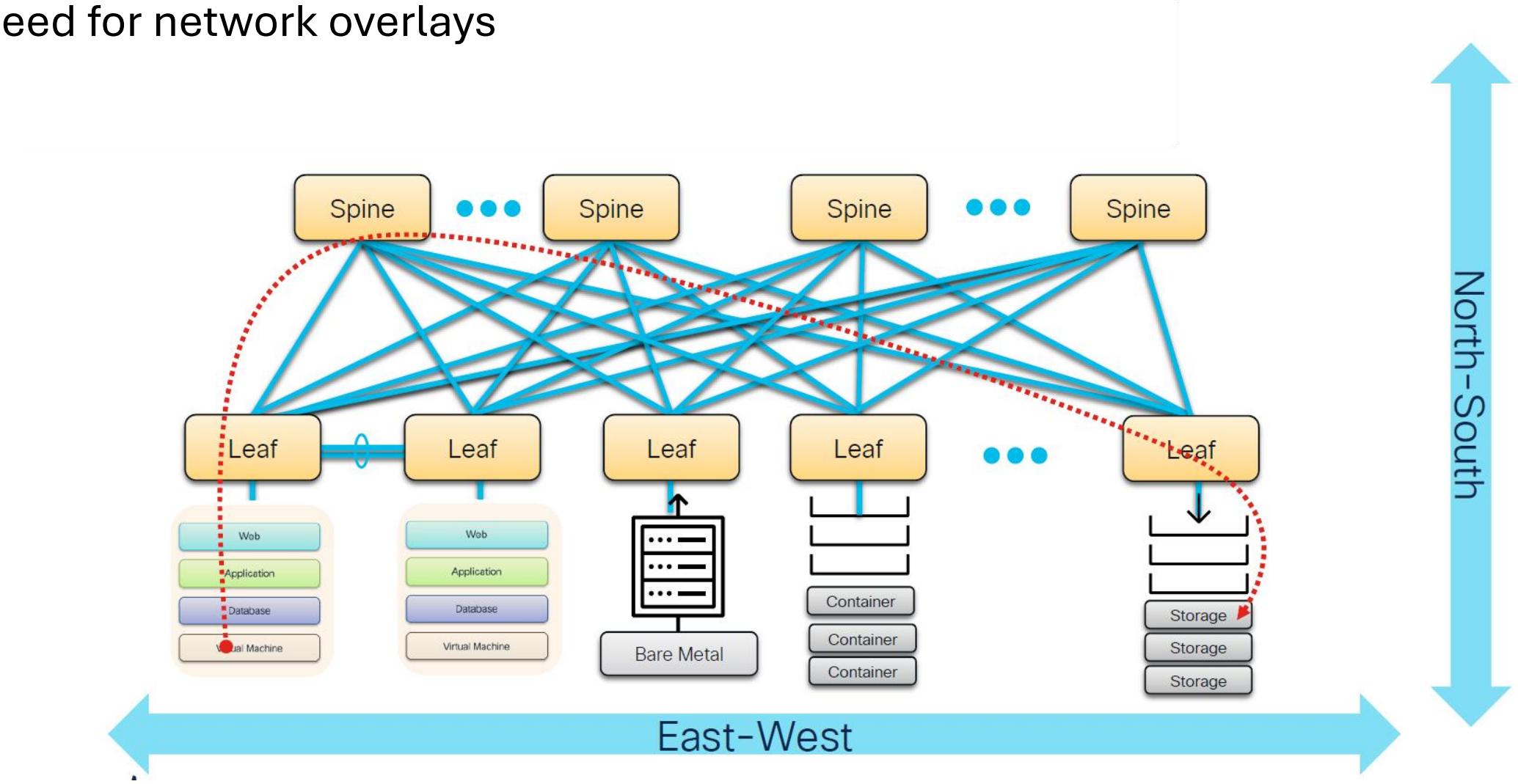
Evolution of the Data Center Network

3-Stage Clos Fabric

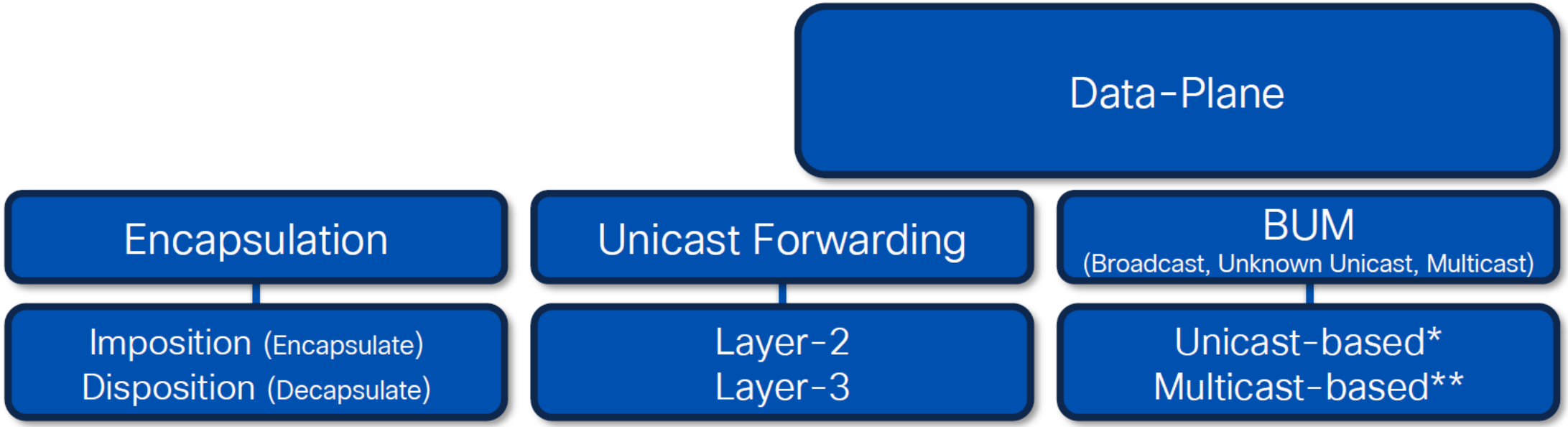


Evolution of the Data Center Network

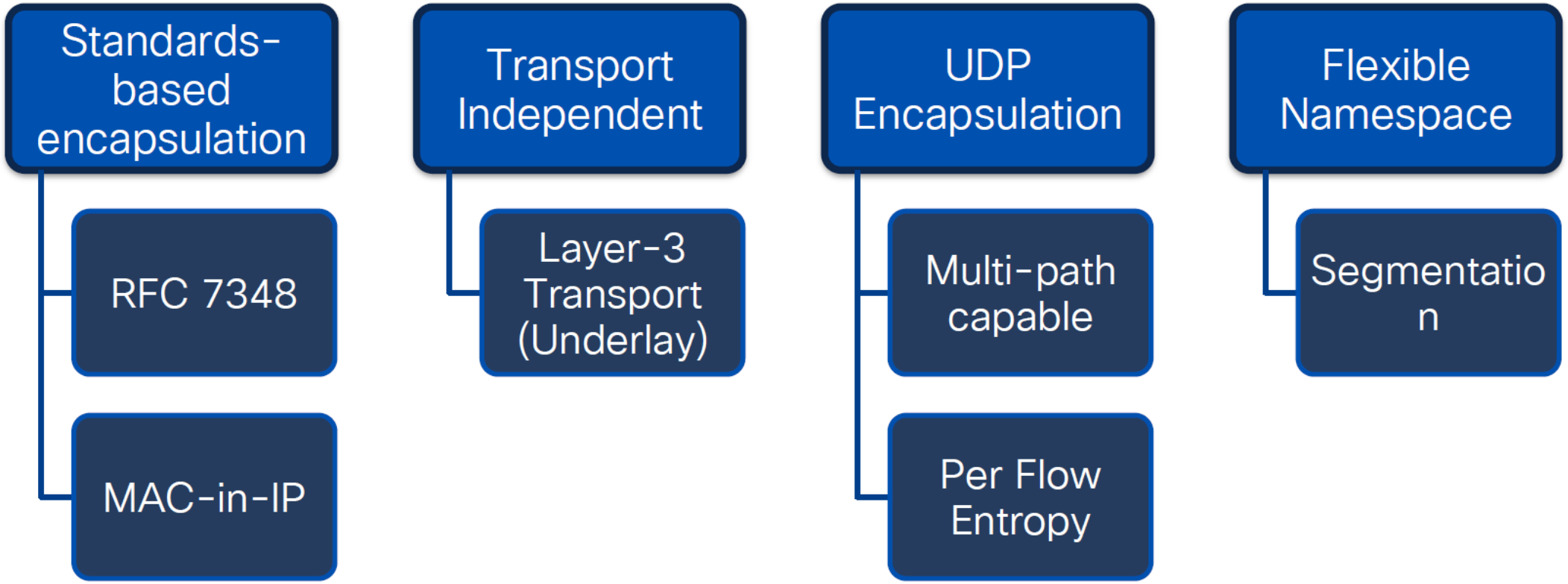
The need for network overlays



Data Plane

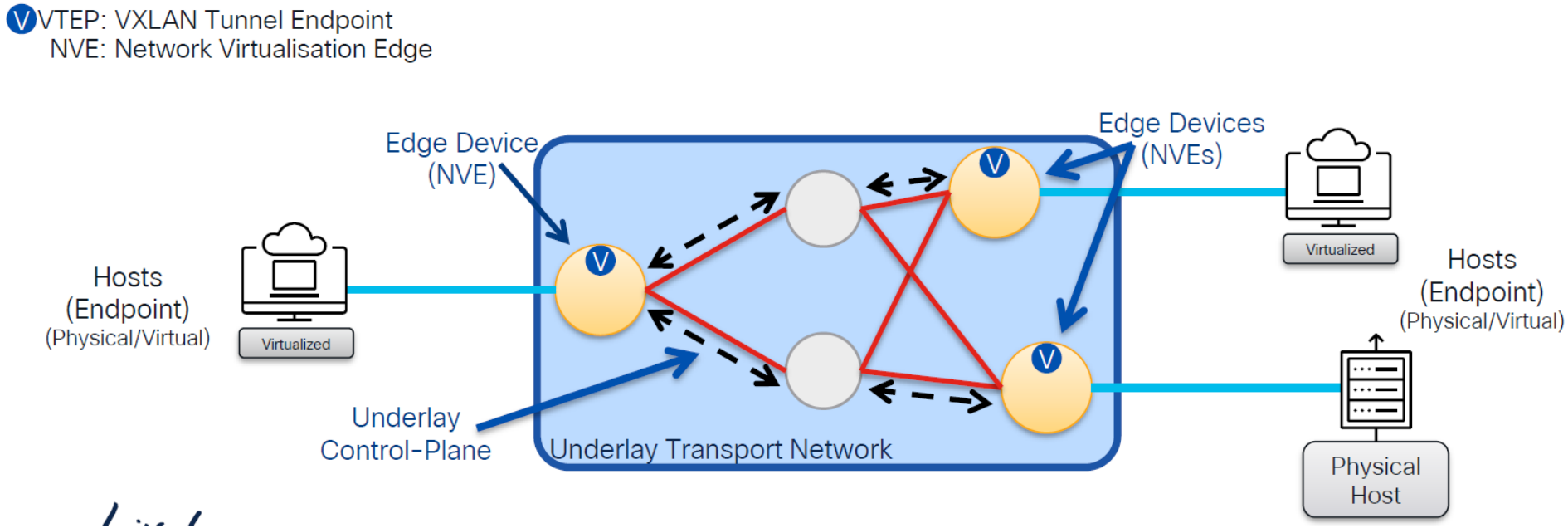

Data-Plane
Encapsulation
Unicast Forwarding
BUM
(Broadcast, Unknown Unicast, Multicast)
Imposition (Encapsulate)
Disposition (Decapsulate)
Layer-2
Layer-3
Unicast-based*
Multicast-based**

What is VXLAN?

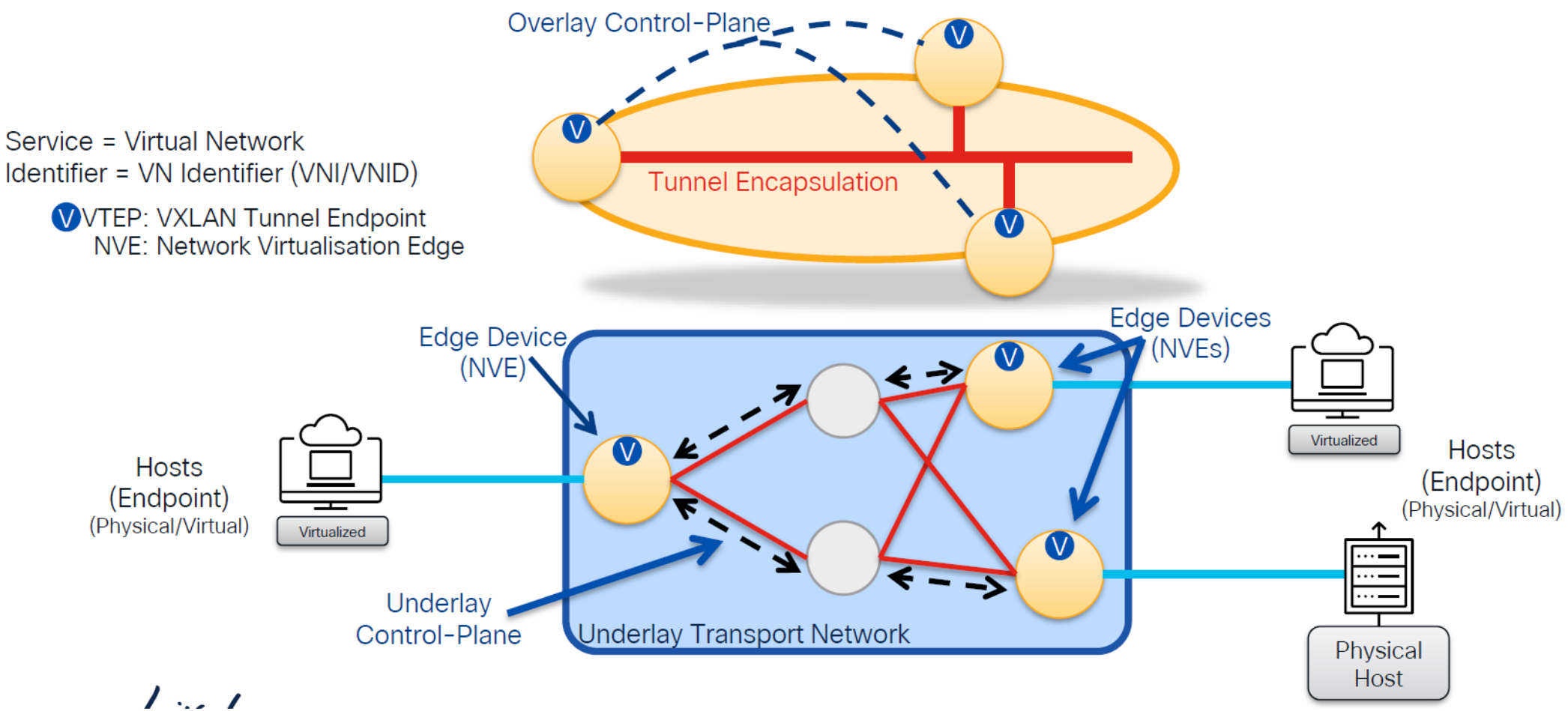


Underlay Taxonomy

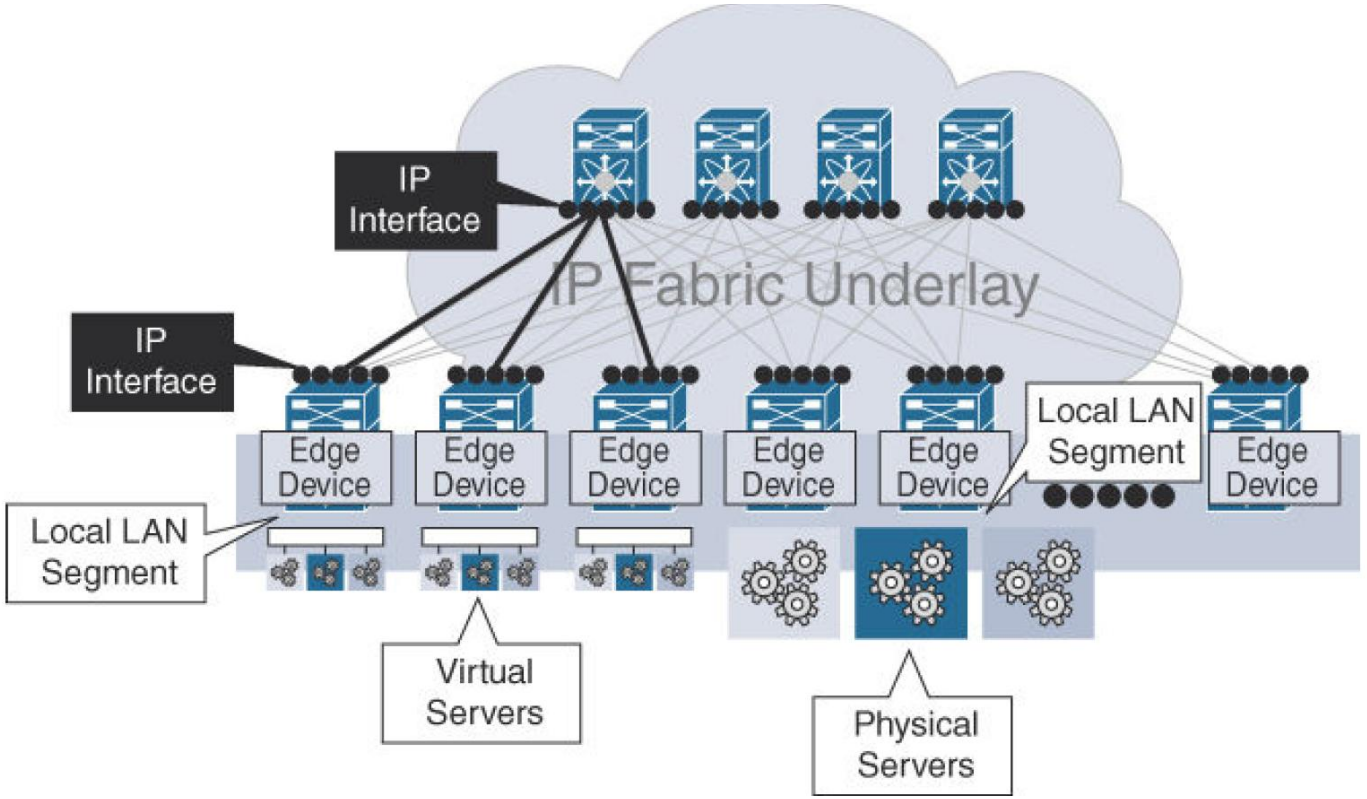
- Edge Devices host the VTEP
- Responsible for the encapsulation and decapsulation of the VXLAN Header



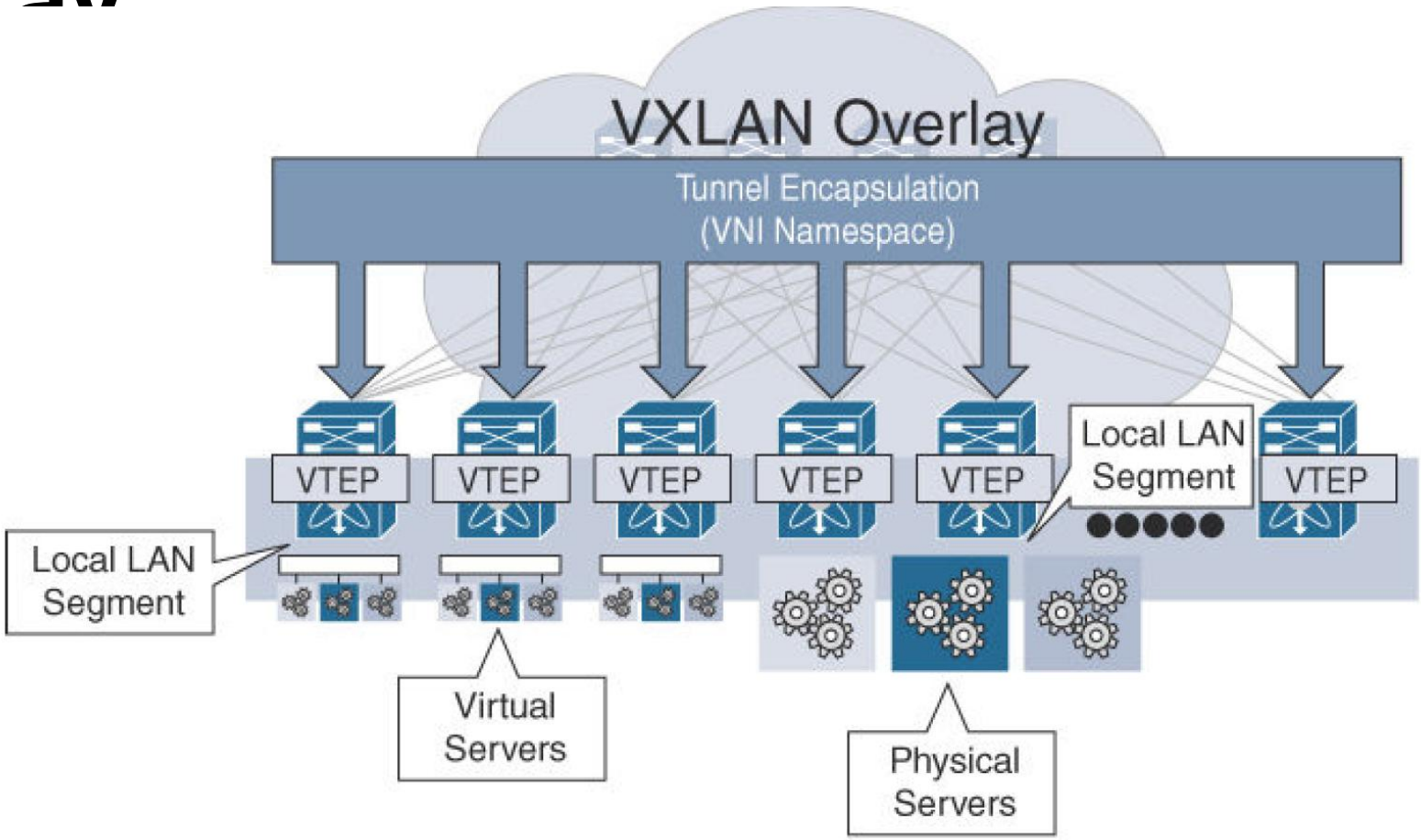
Overlay Taxonomy



Overview in Data Center: Underlay

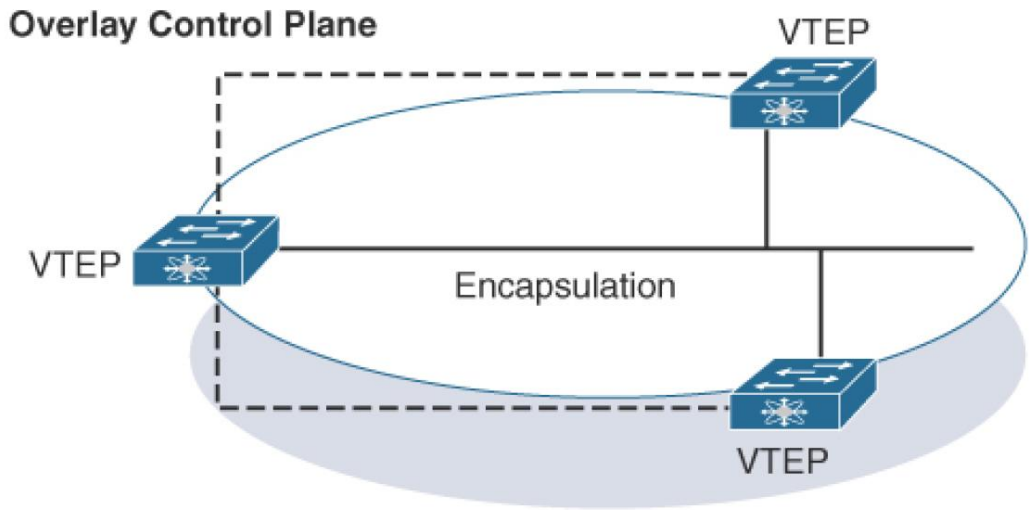


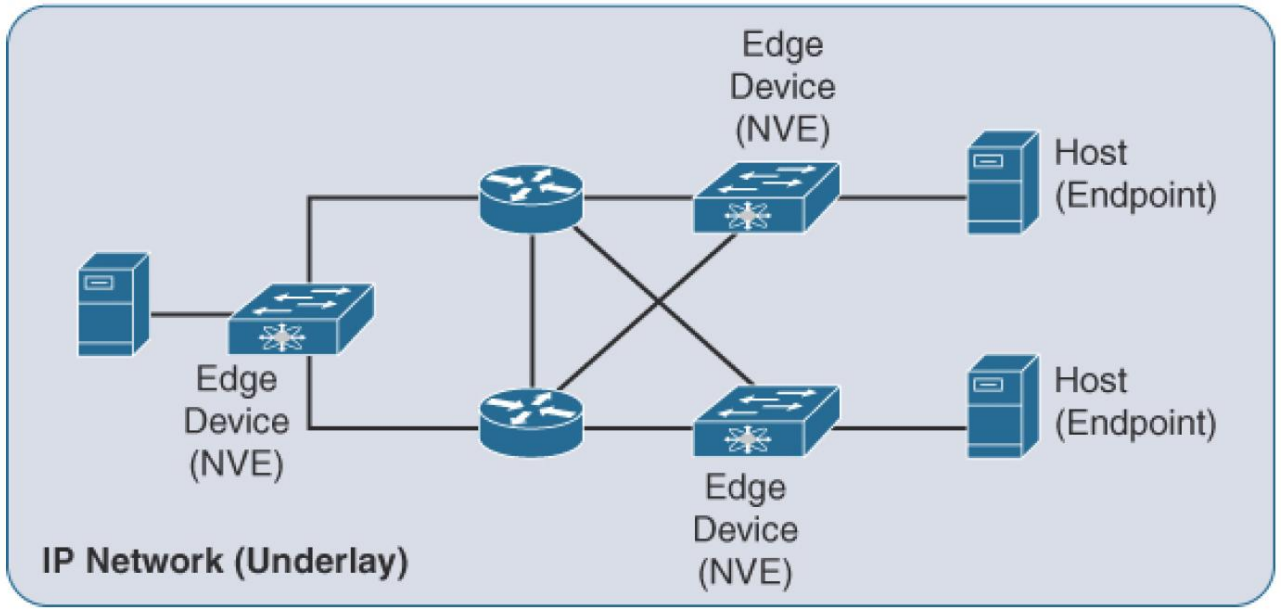
VxLAN Overlay



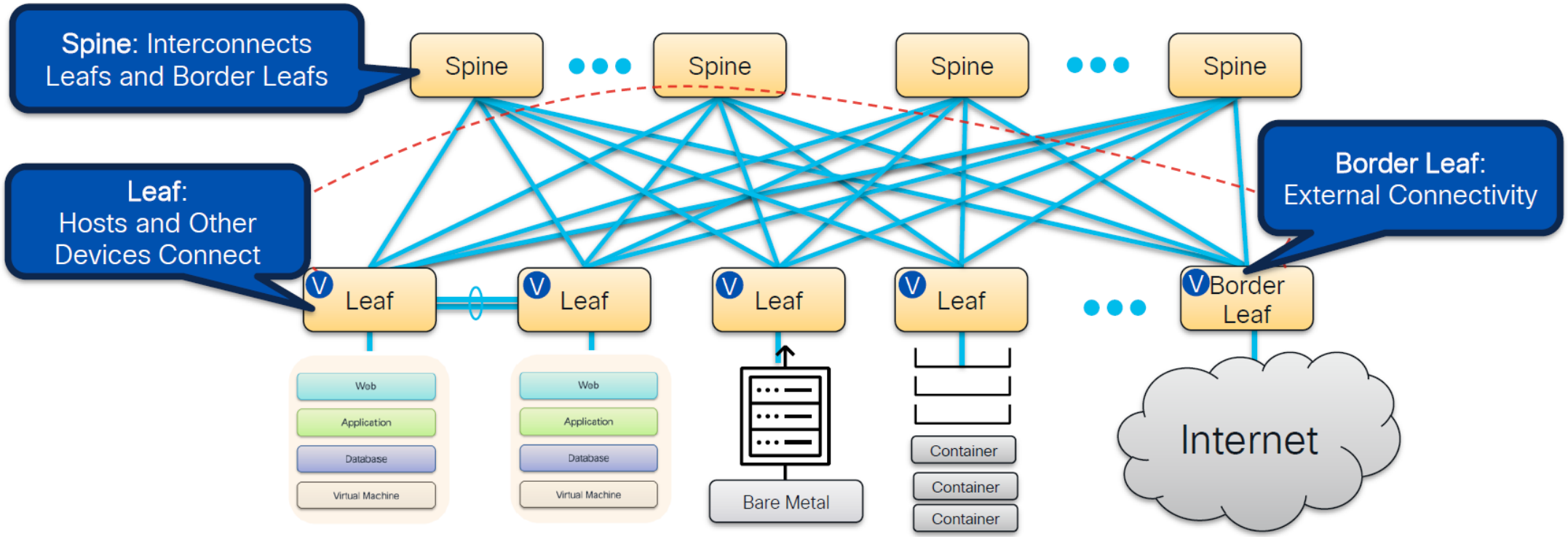
VTEP: VXLAN Tunnel Endpoint
VNI/VNID: VXLAN Network Identifier

Building data centers with VxLAN BPG EVPN

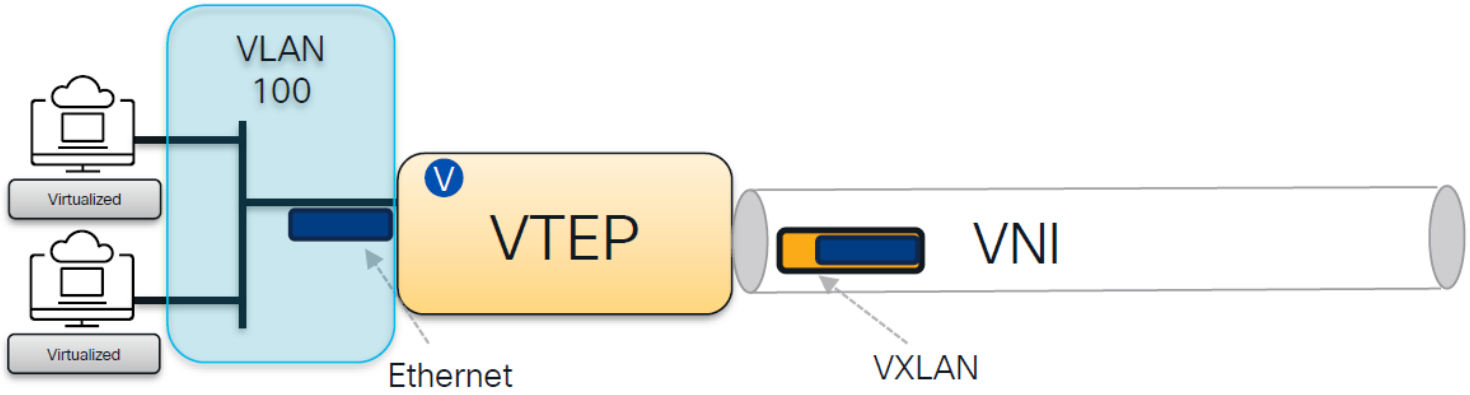

Overlay Control Plane
VTEP
VTEP
Encapsulation
VTEP


Edge
Device
(NVE)
Host
(Endpoint)
Edge
Device
(NVE)
Host
(Endpoint)
Edge
Device
(NVE)
IP Network (Underlay)

VTEP Device Roles

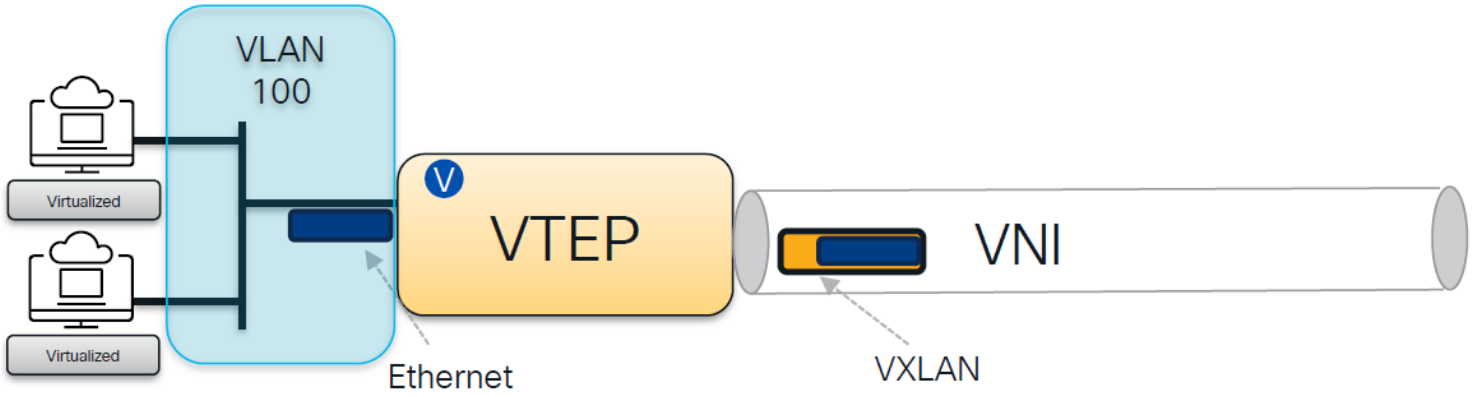


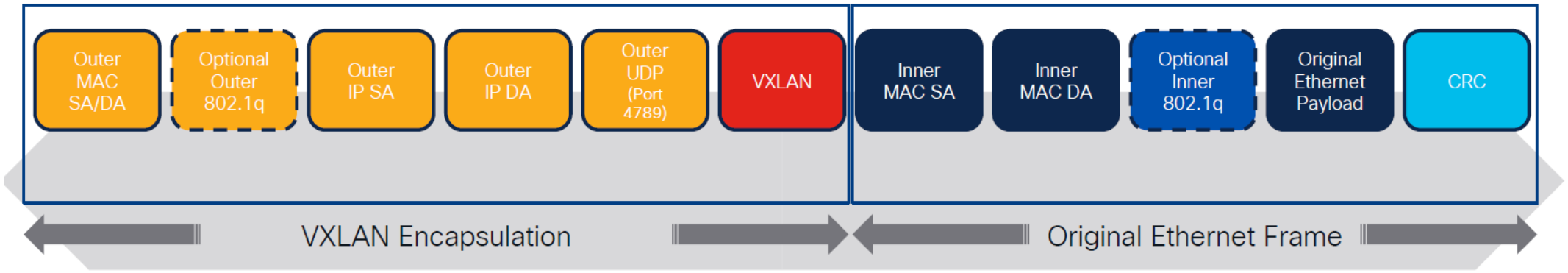
VXLAN (or Virtual) Tunnel Endpoint (VTEP)



- VXLAN Tunnel Endpoint – Network Virtualisation Edge
- Each VTEP is uniquely identified by an IP Address
- VTEP Discovers or learns remote VTEPs, and end hosts attached to them
- VTEP bridges when forwarding packets within the same VNI and Routes for Inter-VNI traffic

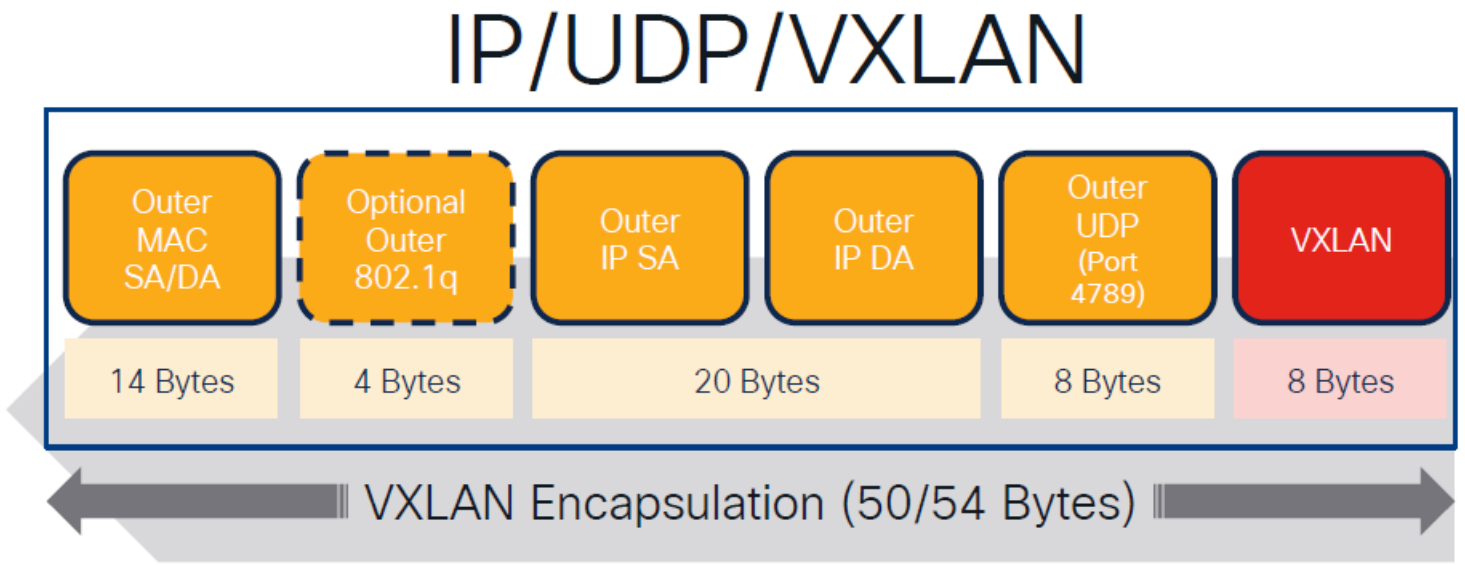
VXLAN Encapsulation / Packet Format





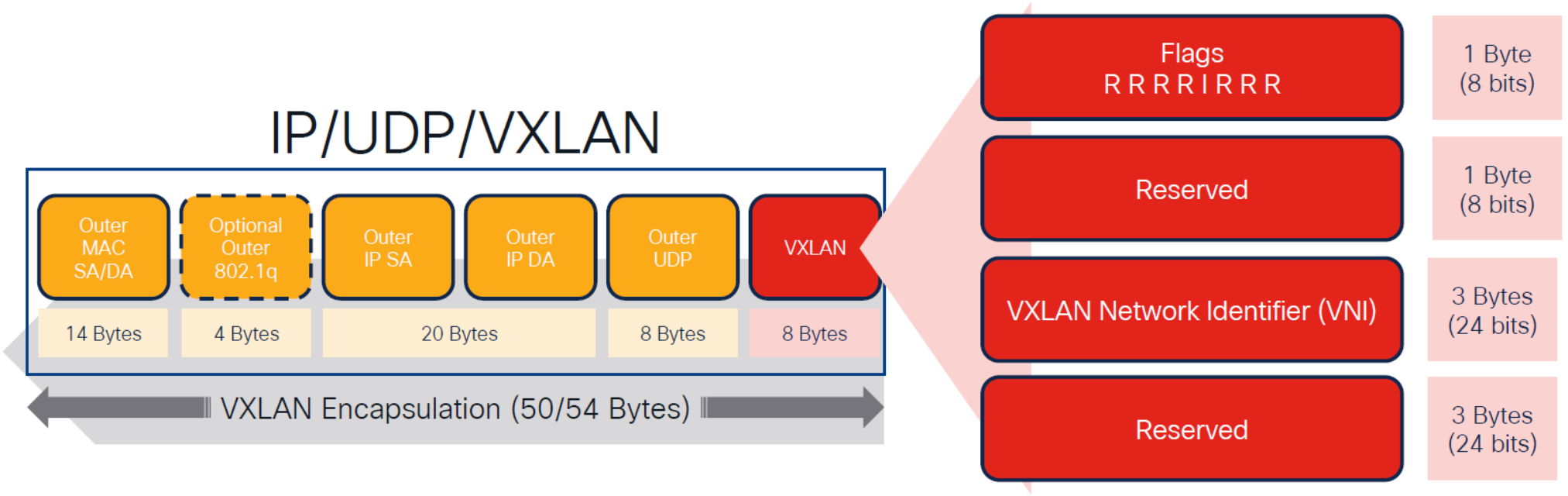
VXLAN Packet Format

- VXLAN uses MAC in UDP encapsulation



- Adds UDP and VXLAN Header before original Ethernet Frame

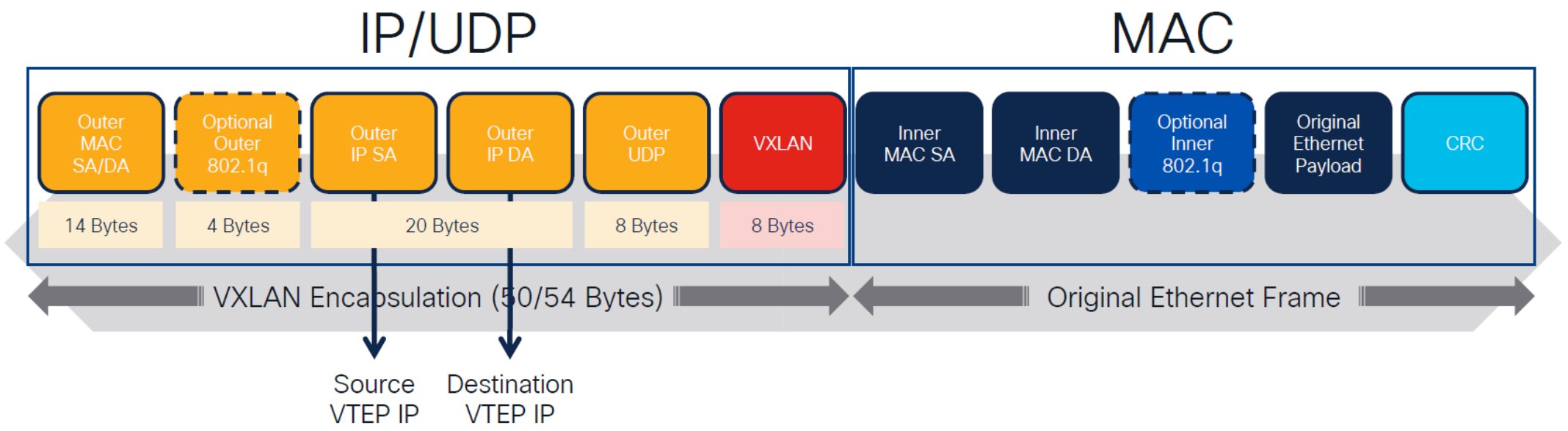
VXLAN Header Details



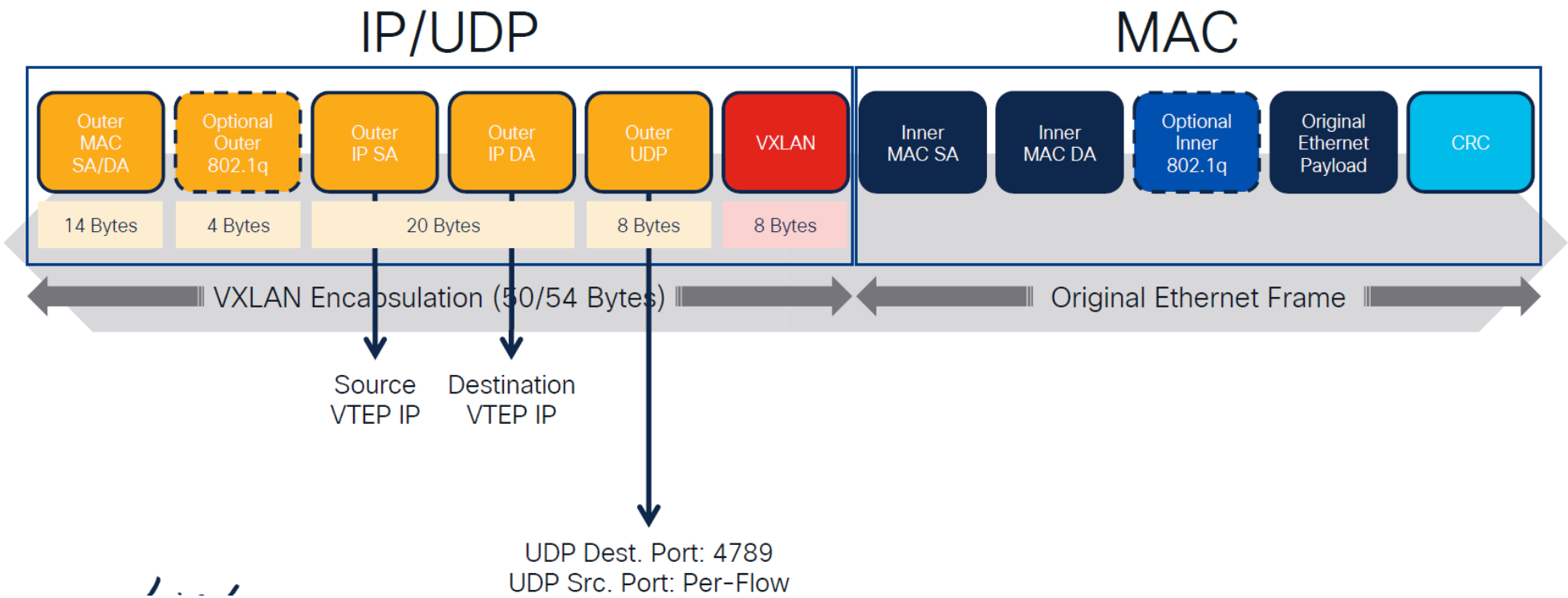
- Flags Field: I-flag (set to 1) for valid VNI. Other flags remain as R (set to 0)
- VNI Field: Allows VNI 1-16,777,215 (some implementation only 4096-16,777,215)

```
vlan 100
  vn-segment 50000
```

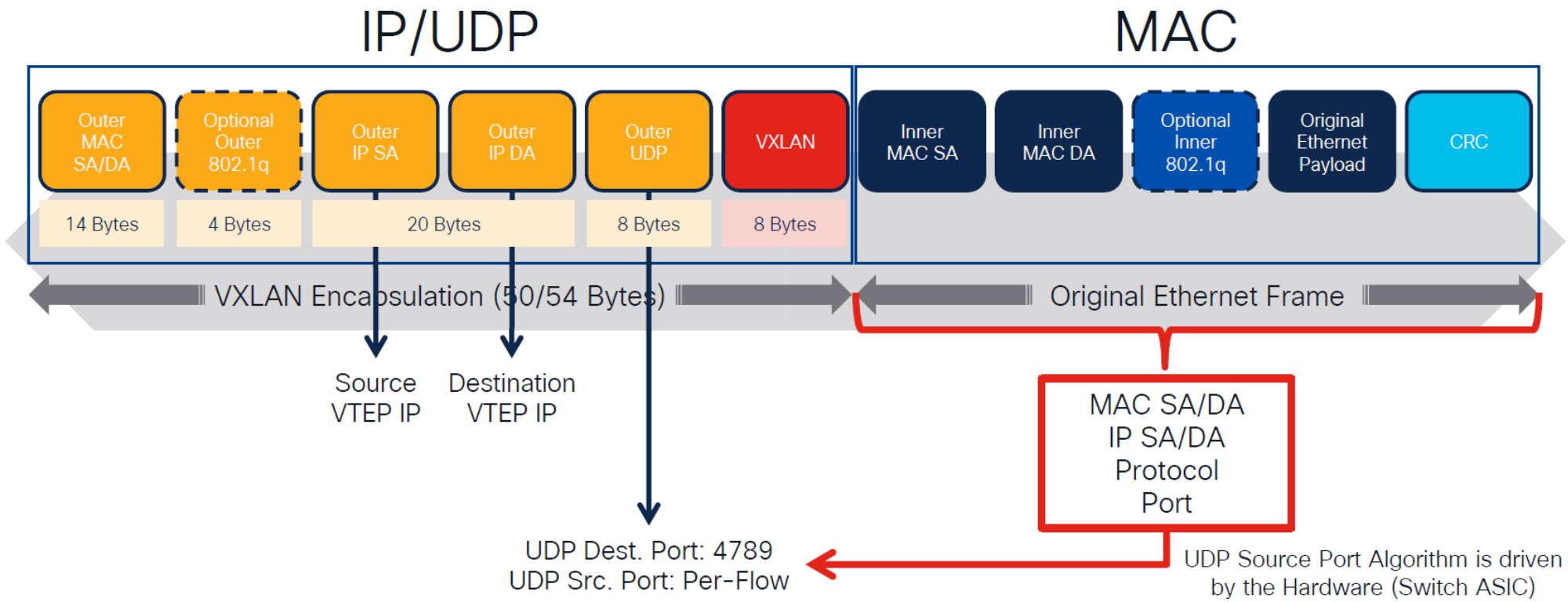
Transport Independence



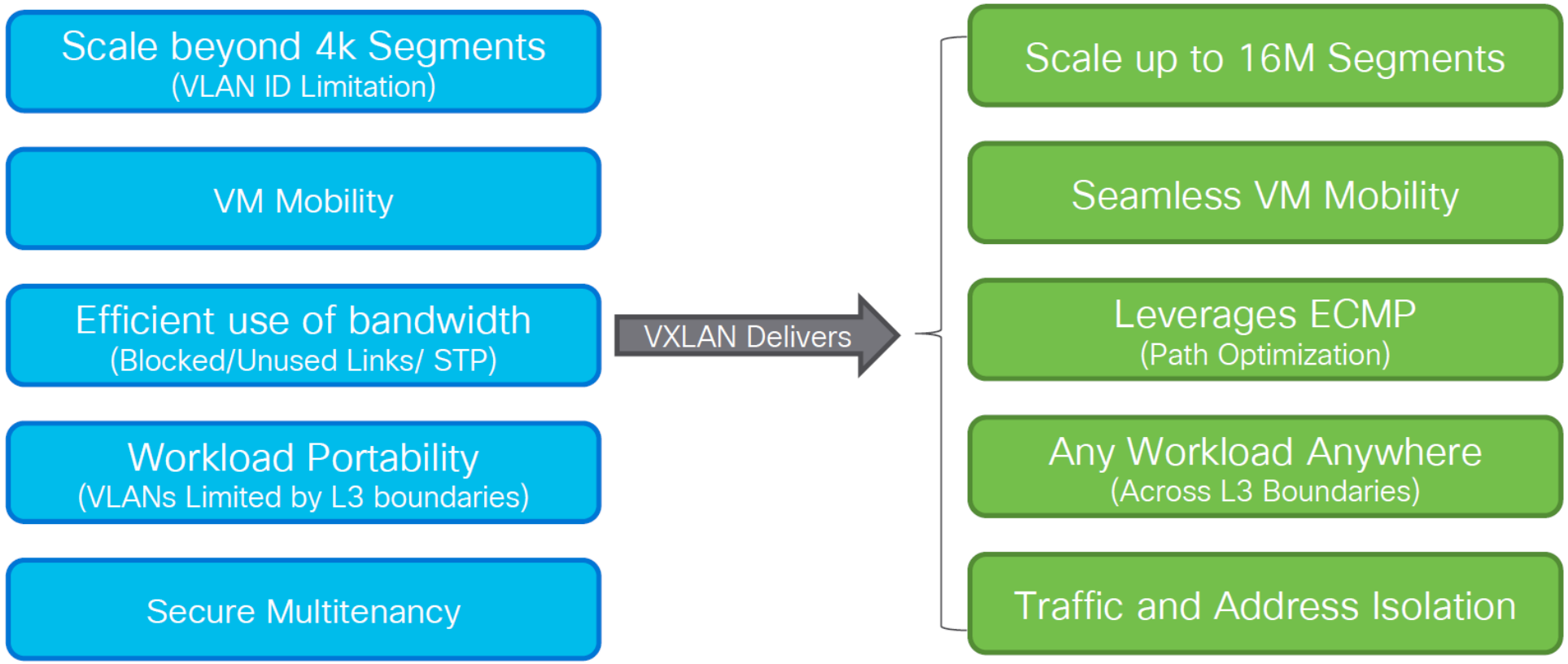
Multipath Capable



Multipath Capable

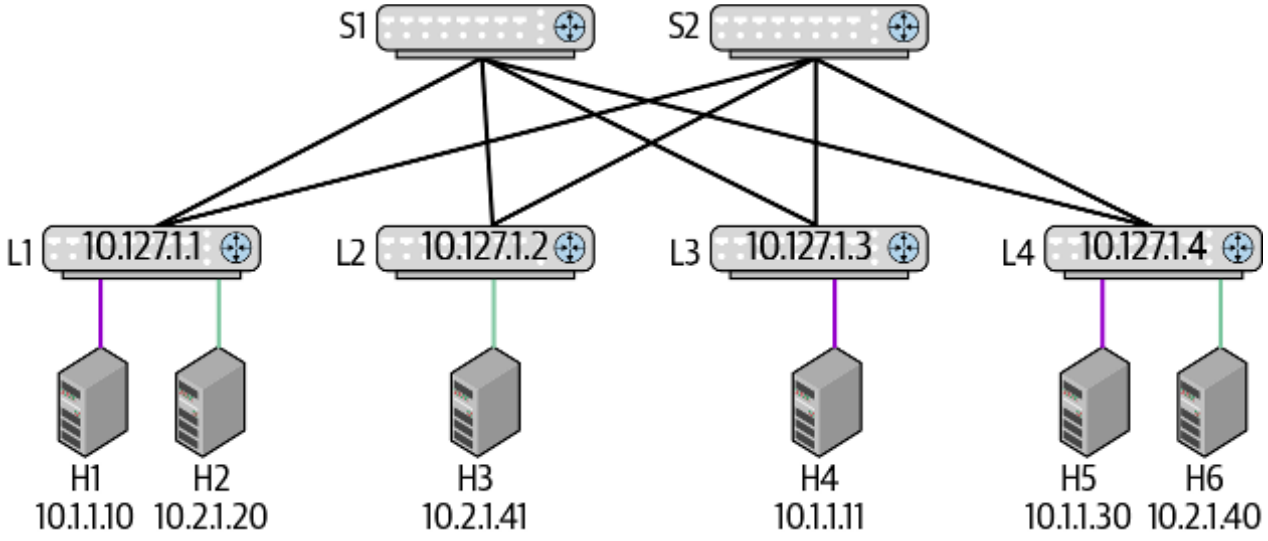


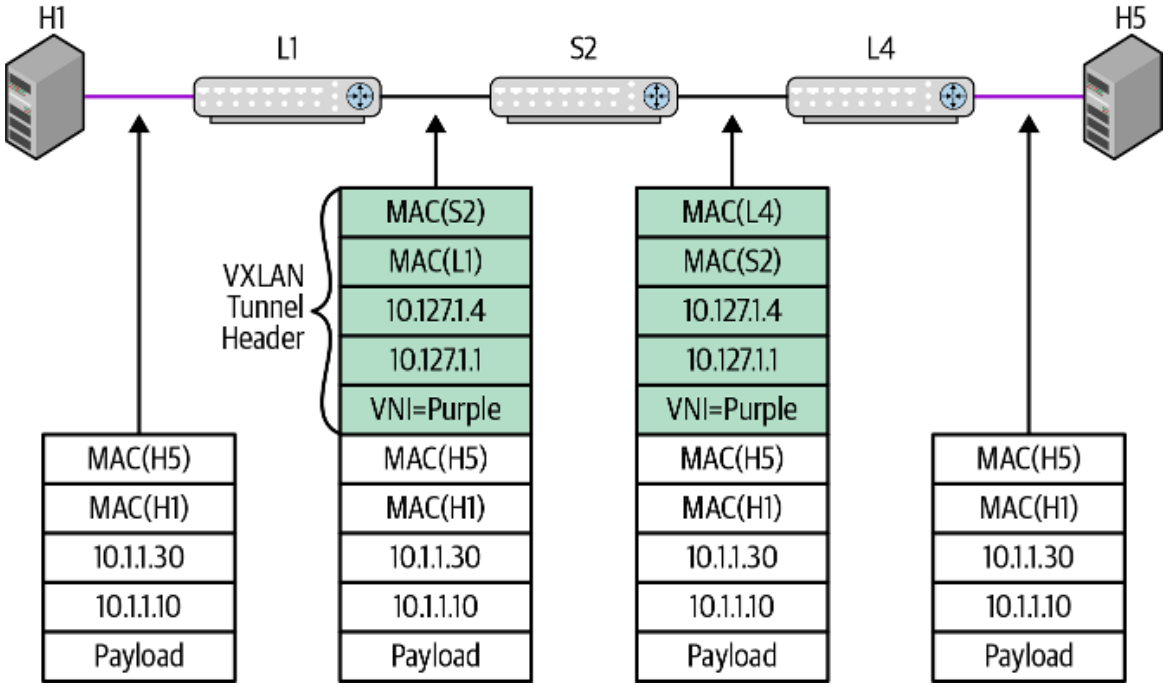
Why VXLAN? - How did we get here?



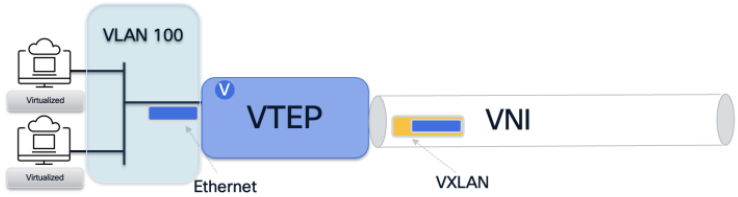
VxLAN Bridging H1 to H5

- Same subnet forward





Packet Capture - Host to VTEP




```
> Frame 2: 102 bytes on wire (816 bits), 102 bytes captured (816 bits)
v Ethernet II, Src: 52:0d:cf:cf:1b:08 (52:0d:cf:cf:1b:08), Dst: 20:20:00:00:00:aa (20:20:00:00:00:aa)
  > Destination: 20:20:00:00:00:aa (20:20:00:00:00:aa)
  > Source: 52:0d:cf:cf:1b:08 (52:0d:cf:cf:1b:08)
    Type: 802.1Q Virtual LAN (0x8100)
    [Stream index: 1]
v 802.1Q Virtual LAN, PRI: 0, DEI: 0, ID: 100
    000. .... .... .... = Priority: Best Effort (default) (0)
    ...0 .... .... .... = DEI: Ineligible
    .... 0000 0110 0100 = ID: 100
    Type: IPv4 (0x0800)
v Internet Protocol Version 4, Src: 100.100.100.2, Dst: 200.200.200.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0xbd56 (48470)
  > 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 255
    Protocol: ICMP (1)
    Header Checksum: 0xa520 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 100.100.100.2
    Destination Address: 200.200.200.2
    [Stream index: 0]
v Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x15f4 [correct]
    [Checksum Status: Good]
    Identifier (BE): 13605 (0x3525)
    Identifier (LE): 9525 (0x2535)
    Sequence Number (BE): 52225 (0xcc01)
    Sequence Number (LE): 460 (0x01cc)
    [Response frame: 3]
    Timestamp from icmp data: Nov  7, 2024 08:14:13.468214000 IST
    [Timestamp from icmp data (relative): 58158.750333000 seconds]
  > Data (40 bytes)
```

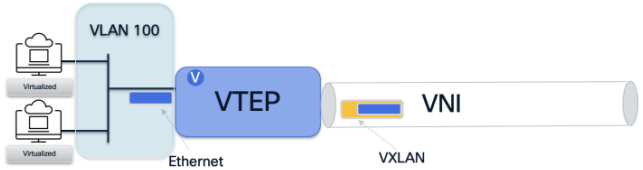
VLAN 100

Ethernet Header

IP Header

Payload

Packet Capture - VTEP to Spine




```
Frame 2: 148 bytes on wire (1184 bits), 148 bytes captured (1184 bits)
Ethernet II, Src: 52:0d:18:e0:1b:08 (52:0d:18:e0:1b:08), Dst: 52:03:7e:03:1b:08 (52:03:7e:03:1b:08)
Internet Protocol Version 4, Src: 10.3.0.1, Dst: 10.3.0.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 134
    Identification: 0x0000 (0)
  > 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 255
    Protocol: UDP (17)
    Header Checksum: 0xa75e [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 10.3.0.1
    Destination Address: 10.3.0.2
    [Stream index: 0]
> User Datagram Protocol, Src Port: 53936, Dst Port: 4789
v Virtual eXtensible Local Area Network
  > Flags: 0x0800, VXLAN Network ID (VNI)
    Group Policy ID: 0
    VXLAN Network Identifier (VNI): 50000
    Reserved: 0
Ethernet II, Src: 52:0d:18:e0:1b:08 (52:0d:18:e0:1b:08), Dst: 52:11:5f:78:1b:08 (52:11:5f:78:1b:08)
  > Destination: 52:11:5f:78:1b:08 (52:11:5f:78:1b:08)
  > Source: 52:0d:18:e0:1b:08 (52:0d:18:e0:1b:08)
    Type: IPv4 (0x0800)
    [Stream index: 1]
Internet Protocol Version 4, Src: 100.100.100.2, Dst: 200.200.200.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0xc086 (49286)
  > 000. .... = Flags: 0x0
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 254
    Protocol: ICMP (1)
    Header Checksum: 0xa2f0 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 100.100.100.2
    Destination Address: 200.200.200.2
    [Stream index: 1]
> Internet Control Message Protocol
```

IP Header

VTEP IP Addresses

UDP Header

VXLAN Encapsulation

VNI 50000

Payload

VLAN vs VxLAN Setup

- **VLAN setup:**
- Create VLAN (e.g., VLAN 10)
- Assign switch ports to VLAN
- Configure trunk ports between switches
- Devices communicate within layer-2 domain

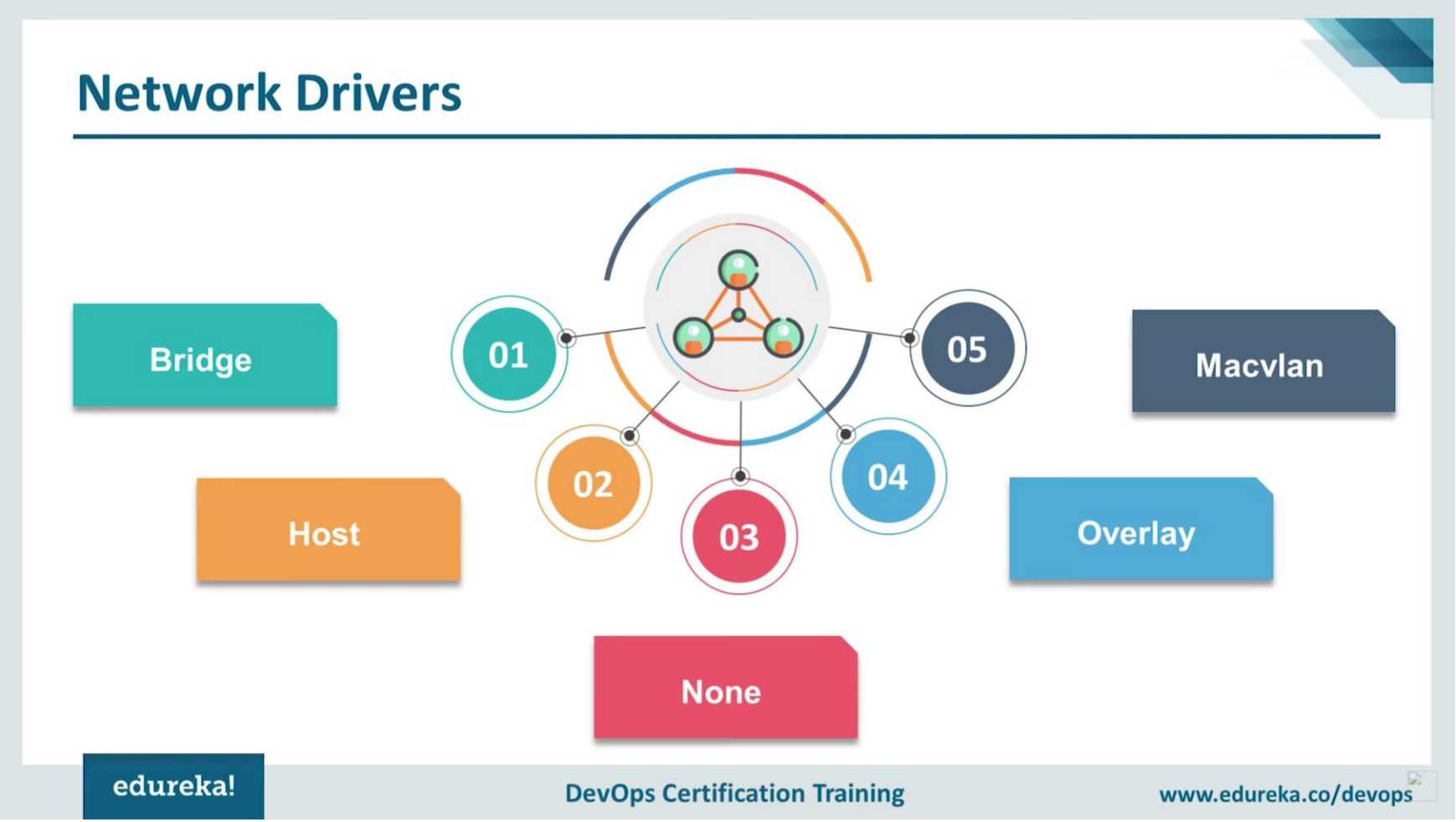
- **VXLAN setup:**
- Configure VTEP on each device/switch (assign IP address)
- Create VNI (e.g., VNI 5000)
- Map VNI to local VLAN
- Configure peer discovery (multicast group, static peers, or BGP EVPN)
- VXLAN tunnels form automatically between VTEPs

Key differences:
- VLAN: Layer-2 only, limited to 4096 IDs, direct switch connectivity
- VXLAN: Layer-3 transport, 16M IDs, works across routed networks
- Creation order for VXLAN: VTEP → VNI → VNI-to-VLAN mapping → Peer discovery
- The VTEP is the foundational element - without it, VXLAN cannot function.

Testing...

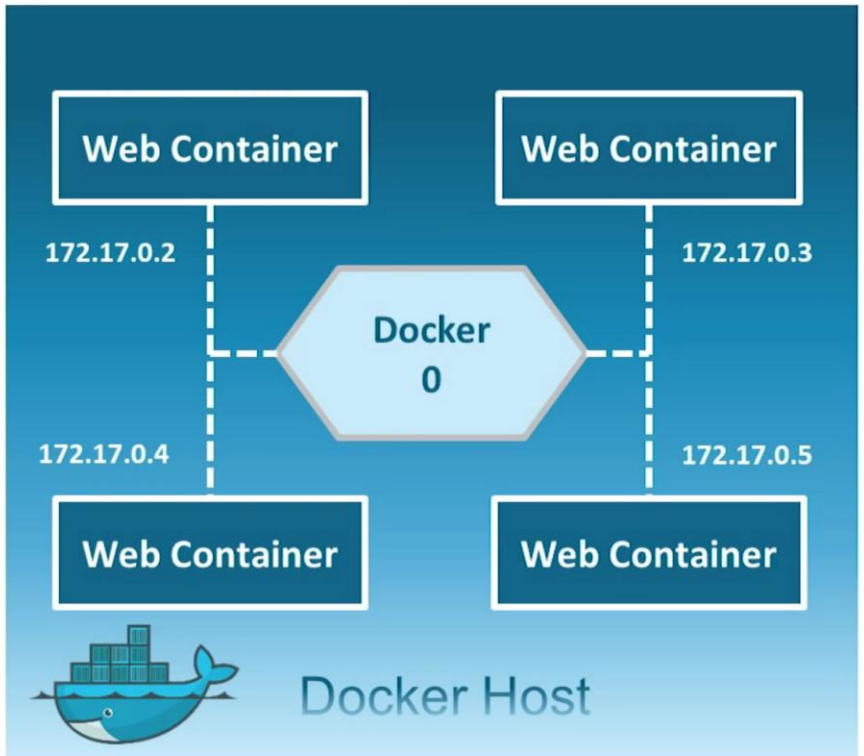
- https://github.com/faysalmehedi/vxlan-ovs-docker-lab?fbclid=IwAR0OW7MTpkOc0nfHGey5otzmpo_ng5YRQm0ITF6JsjbAbFAVAjf0Svz7sd4

Docker Networks

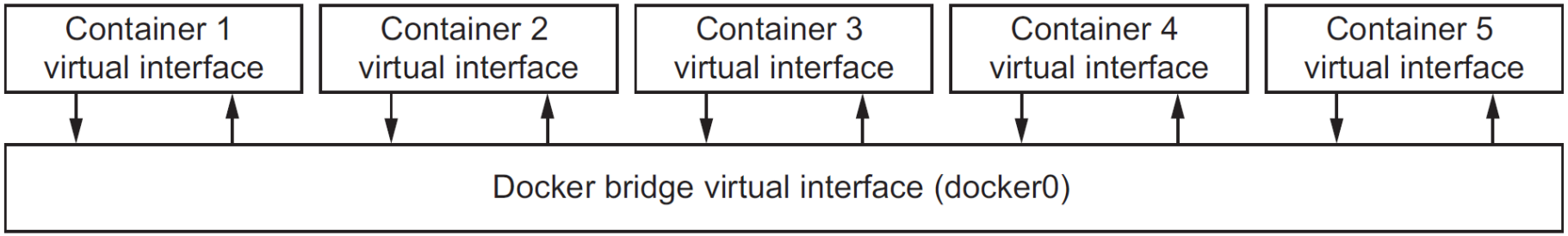


Network Drivers: Bridge

The default network driver. If you don't specify a driver, this is the type of network you are creating. Bridge networks are usually used when your applications run in standalone containers that need to communicate.



Bridged Container: Inter-container communication



User can

- Define the address and subnet of the bridge
- Define the range of IP addresses that can be assigned to containers
- Define the maximum transmission unit (MTU)

Docker network

```
modern@ubuntu:~$ docker network

Usage:  docker network COMMAND

Manage networks

Commands:
  connect     Connect a container to a network
  create      Create a network
  disconnect  Disconnect a container from a network
  inspect     Display detailed information on one or more networks
  ls          List networks
  prune       Remove all unused networks
  rm          Remove one or more networks

Run 'docker network COMMAND --help' for more information on a command.
```

Check for Network installed (from install docker)

- Name
  - Just Name ..
- Scope
  - Local = single host
  - Swarm = multi-host

'bridge' is the default one where all new containers will connect to unless specified otherwise, and it maps to docker0 virtual bridge (the one you see when running ifconfig in the host)
'host' maps the container directly to the host, so it is not recommended due to security concerns
'none' provides no connectivity at all

```
root@ubuntu:/home/modern# docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
03e9693f71df        bridge              bridge              local
51c819f4deb6        host                host                local
acf2bf319f06        none                null                local
root@ubuntu:/home/modern#
```

Inspect

```
root@ubuntu:/home/modern# docker network inspect bridge
[
    {
        "Name": "bridge",
        "Id": "03e9693f71df12e37b7ca9c8606f8f3c22d44e9943ac95432751ffa94475e4c4"
,
        "Created": "2019-12-24T08:16:41.135287876-08:00",
        "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": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {
            "com.docker.network.bridge.default_bridge": "true",
            "com.docker.network.bridge.enable_icc": "true",
            "com.docker.network.bridge.enable_ip_masquerade": "true",
            "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",
            "com.docker.network.bridge.name": "docker0",
            "com.docker.network.driver.mtu": "1500"
        },
        "Labels": {}
    }
```

No Container Connected Yet

- No IPv6
- IPAM = IP Address Management
- Default subnet = 172.17.0.0 and gateway = 172.17.0.1

Create a bridge

- Bridge name ps-bridge

```
modern@ubuntu:~$ sudo docker network create -d bridge --subnet 10.0.0.1/24 ps-bridge
[sudo] password for modern:
913ba9ee873a476c82b39814dce1c8fa080ea8a1c73131c8c7bf72529a6d27fe
```

- Check it

```
modern@ubuntu:~$ sudo docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
03e9693f71df        bridge              bridge              local
51c819f4deb6        host                host                local
acf2bf319f06        none                null                local
913ba9ee873a        ps-bridge           bridge              local
modern@ubuntu:~$
```

ตรวจสอบที่สร้างขึ้น

```
root@ubuntu:/home/modern# docker network inspect ps-bridge
[
    {
        "Name": "ps-bridge",
        "Id": "913ba9ee873a476c82b39814dce1c8fa080ea8a1c73131c8c7bf72529a6d27fe",
        "Created": "2019-12-24T08:51:58.481215154-08:00",
        "Scope": "local",
        "Driver": "bridge",
        "EnableIPv6": false,
        "IPAM": {
            "Driver": "default",
            "Options": {},
            "Config": [
                {
                    "Subnet": "10.0.0.1/24"
                }
            ]
        },
        "Internal": false,
        "Attachable": false,
        "Ingress": false,
        "ConfigFrom": {
            "Network": ""
        },
        "ConfigOnly": false,
        "Containers": {},
        "Options": {},
        "Labels": {}
    }
]
```

Install bridge function module

- Install

```
root@ubuntu:/home/modern# apt-get install bridge-utils
Reading package lists... Done
Building dependency tree
Reading state information... Done
bridge-utils is already the newest version (1.5-15ubuntu1).
bridge-utils set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 128 not upgraded.
```

- Show

```
root@ubuntu:/home/modern# brctl show
bridge name     bridge id               STP enabled     interfaces
br-913ba9ee873a         8000.02420439398f       no
docker0         8000.02428309553e       no
root@ubuntu:/home/modern#
```

  - The first on -> one we just create
  - The second one -> relate to default bridge

Check

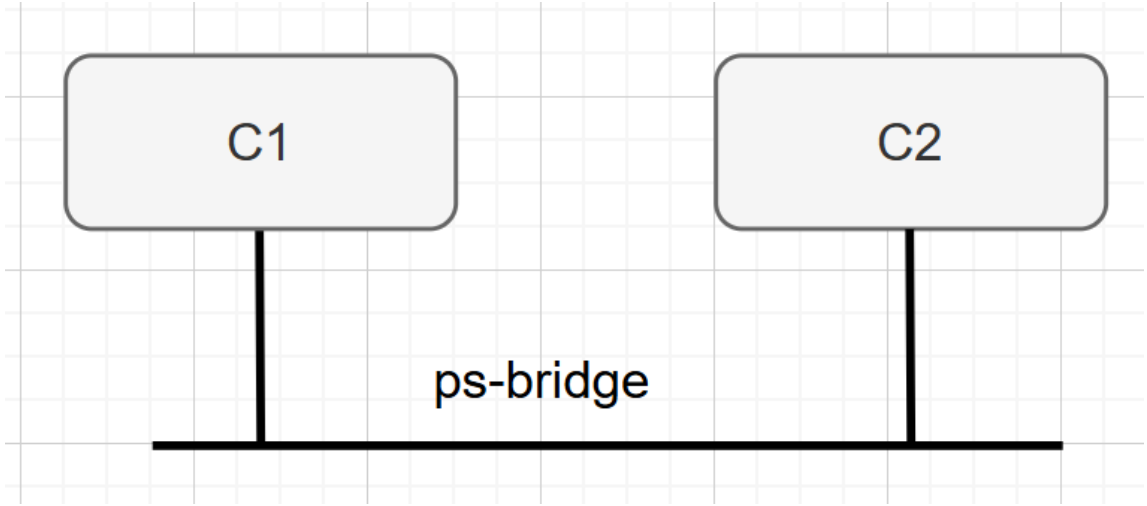
```
root@ubuntu:/home/modern# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT grou
p default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEF
AULT group default qlen 1000
    link/ether 00:0c:29:84:89:f3 brd ff:ff:ff:ff:ff:ff
3: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mod
e DEFAULT group default
    link/ether 02:42:83:09:55:3e brd ff:ff:ff:ff:ff:ff
4: br-913ba9ee873a: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state
DOWN mode DEFAULT group default
    link/ether 02:42:04:39:39:8f brd ff:ff:ff:ff:ff:ff
```

Create container to connect

- C1, C2 container

```
root@ubuntu:/home/modern# docker run -dt --name c1 --network ps-bridge alpine sleep 1d
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
e6b0cf9c0882: Pull complete
Digest: sha256:21716586201556792 40babee0a7714f6509fae66898db422ad803b951257db78
Status: Downloaded newer image for alpine:latest
859dab0048a0c8a4bae6c08d9b65d3868686849775477dc18f6d3575e85165ba
root@ubuntu:/home/modern#
```

```
root@ubuntu:/home/modern# docker run -dt --name c2 --network ps-bridge alpine sleep 1d
c21ecdce9ad926eb6eda4c84cb7f66eb8ad2f0d72188a293008a7deab2e1ef38
```



Inspect

- docker network inspect ps-bridge


```
"Containers": {
    "859dab0048a0c8a4bae6c08d9b65d3868686849775477dc18f6d3575e85165ba": {
        "Name": "c1",
        "EndpointID": "ba183b618d72d649f0f4a22380a7ab8b5280cea8c2d67b64e74f874998a47e0a",
        "MacAddress": "02:42:0a:00:00:02",
        "IPv4Address": "10.0.0.2/24",
        "IPv6Address": ""
    },
    "c21ecdce9ad926eb6eda4c84cb7f66eb8ad2f0d72188a293008a7deab2e1ef38": {
        "Name": "c2",
        "EndpointID": "d448349e9dfe0ea5279a3ac14e94565b8cace218df2412789aa96eb62ba36434",
        "MacAddress": "02:42:0a:00:00:03",
        "IPv4Address": "10.0.0.3/24",
        "IPv6Address": ""
    }
},
```

Bridge show

- Brctl show
  - Two virtual Ethernet (vethxxxx)

```
root@ubuntu:/home/modern# brctl show
bridge name     bridge id               STP enabled     interfaces
br-913ba9ee873a         8000.02420439398f       no              veth250bf5d
                                                                vethe5aade0
docker0         8000.02428309553e       no
```

Testing Connectivity

- Go inside container
- Check ip address
- Ping ->ip (can we ping by name … testing ping c2)

```
root@ubuntu:/home/modern# docker exec -it c1 sh
/ # ip a
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
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
5: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:0a:00:00:02 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.2/24 brd 10.0.0.255 scope global eth0
       valid_lft forever preferred_lft forever
/ # ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3): 56 data bytes
64 bytes from 10.0.0.3: seq=0 ttl=64 time=3.816 ms
64 bytes from 10.0.0.3: seq=1 ttl=64 time=0.085 ms
64 bytes from 10.0.0.3: seq=2 ttl=64 time=0.084 ms
^C
--- 10.0.0.3 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.084/1.328/3.816 ms
```

FRRouting

- FRR is free software that implements and manages various IPv4 and IPv6 routing protocols. It runs on nearly all distributions of Linux and BSD and supports all modern CPU architectures.
- FRR currently supports the following protocols: BGP, OSPF, RIP, IS-IS, PIM, LDP, BFD, Babel, PBR, OpenFabric and VRRP, with alpha support for EIGRP and NHRP.

History of FRR

~1996 - Zebra development begins

↓

~2002 - Quagga forked from Zebra

↓

2016 - FRR forked from Quagga

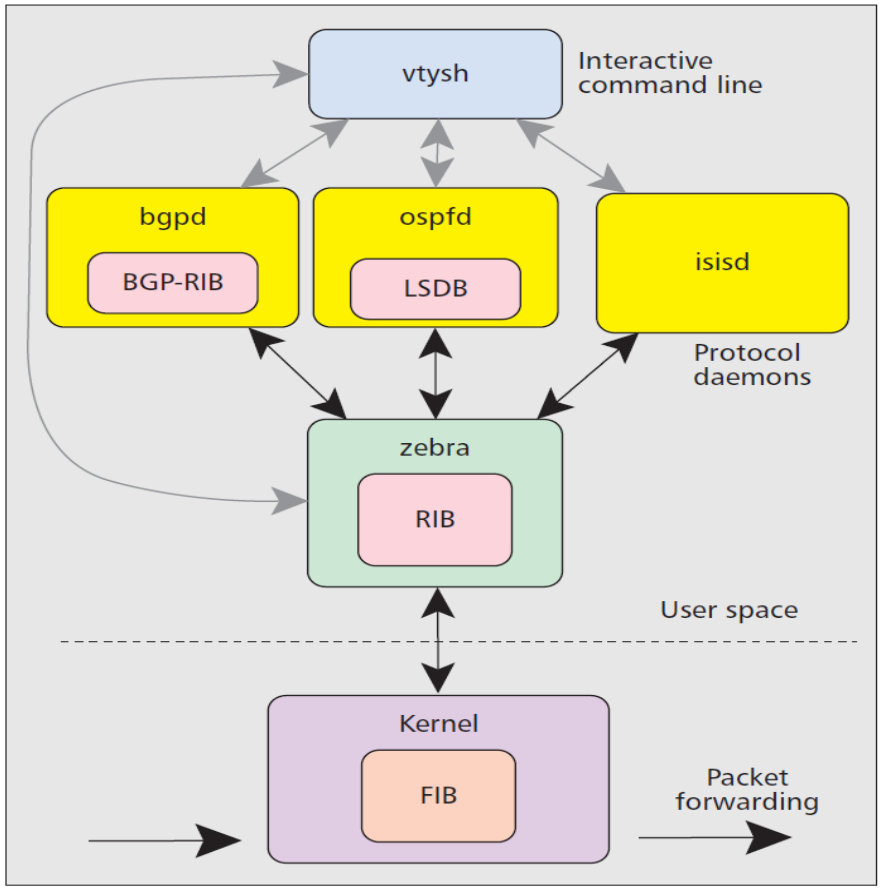
↓

🎉 Jan. 2017 - First Release - FRR 2.0 🎉

↓

June 11, 2018 - Latest Stable - FRR 5.0

- FRR has its roots in the Quagga project.



Quagga architecture

Introduction to the Quagga Routing Suite, IEEE Network 2014

Releases

- https://frrouting.org/release/
  - Latest Nov 9, 2025
- https://docs.frrouting.org/en/latest/

What kernel do you need?

- VRF - Depends on l3mdev
  - 4.4 for Limited Functionality, 4.8 for full functionality
- BGP EVPN - Depends on NTF_EXT_LEARNED and ARP Suppression
  - 4.17 and 4.14
- PIM-SM - Depends on IGMPMSG_WRVIFWHOLE and RTN_MULTICAST netlink messages
  - 4.18
- These features are not currently available for non-Linux distributions

Starting/Stopping FRR

- systemctl <start|restart|reload|stop> frr
- Config file: /etc/frr/frr.conf
- The interactive shell
  - vtysh

```
root@r1:/home/cumulus# systemctl start frr
root@r1:/home/cumulus# vtysh

Hello, this is FRRouting (version 4.0+cl3u2).
Copyright 1996-2005 Kunihiro Ishiguro, et al.

r1.rdu.cumulusnetworks.com# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, D -
SHARP,
       F - PBR,
       > - selected route, * - FIB route

K>* 0.0.0.0/0 [0/0] via 10.50.11.1, eth0, 00:00:06
C>* 10.50.11.0/24 is directly connected, eth0, 00:00:06
C>* 192.168.210.0/24 is directly connected, swp1, 00:00:06
C>* 192.168.213.0/24 is directly connected, swp2, 00:00:06
C>* 192.168.214.0/24 is directly connected, swp3, 00:00:06
C>* 192.168.240.1/32 is directly connected, lo, 00:00:06
```

Logging

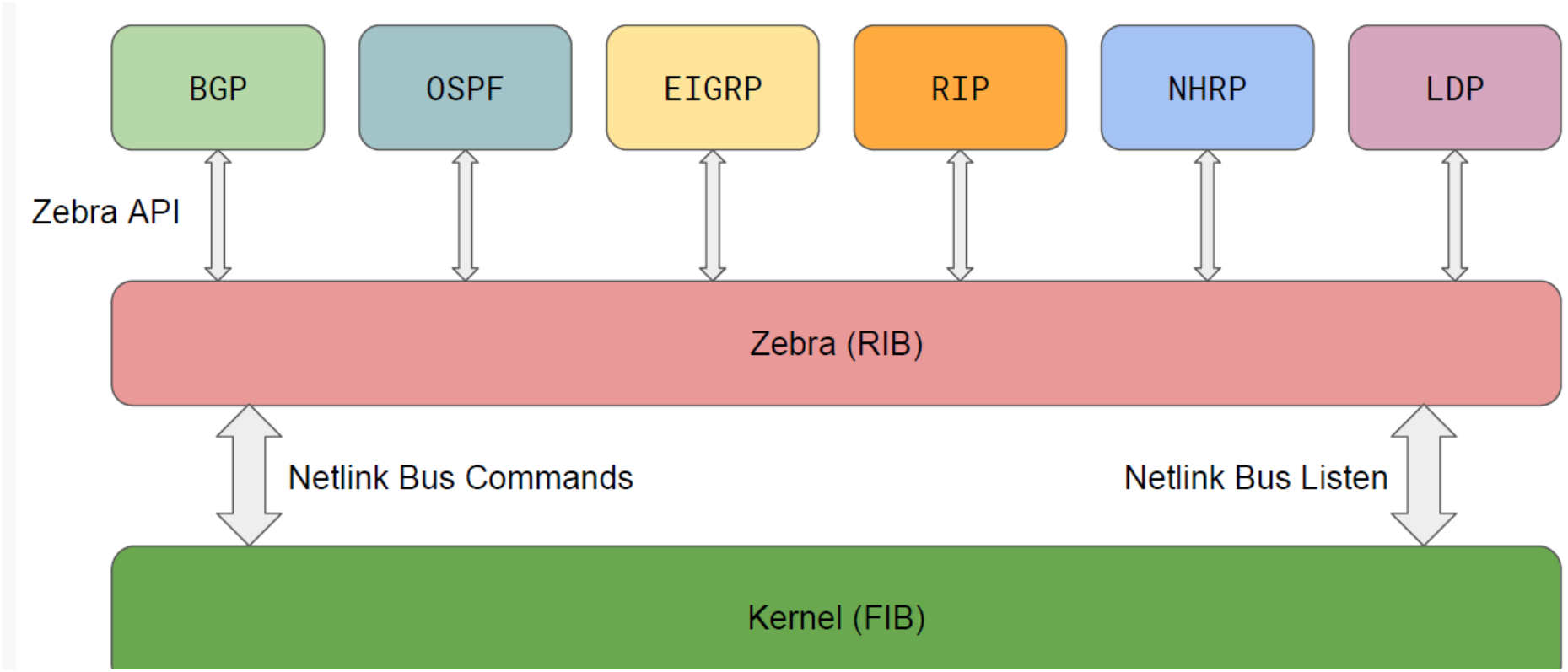
- Where do my logs go?
  - **show logging** in vtysh
  - 3 supported logging destinations:
    1. Syslog - **log syslog [level]**
    2. File - **log file [name]**
    3. Standard output - **log stdout**
- The log file has no output
- Designed to not be chatty unless debugs are turned on
- If you are not sure what is going on use debug commands liberally
- **debug [protocol] ?** if you are not sure
- what to turn on.

```
frr(config)# log file /tmp/frr.log
frr(config)# log syslog
frr(config)# log stdout
frr(config)#
frr# show logging

Logging configuration for zebra:
Syslog logging: level debugging, facility
daemon, ident zebra
Stdout logging: level debugging
Monitor logging: level debugging
File logging: disabled
Protocol name: ZEBRA
Record priority: disabled
Timestamp precision: 6

...
```

FRR Architecture



RIB -vs- FIB

- Routing Information Base
  - A.k.a ‘Control Plane’
  - This is in FRRouting
- Forwarding Information Base
  - A.k.a ‘Data plane’
  - This is in the Kernel

FRR

```
robot# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, A - Babel, D - SHARP,
       F - PBR,
       > - selected route, * - FIB route

K * 0.0.0.0/0 [0/100] via 10.0.2.2, enp0s3 inactive, 01:24:15
D   4.3.2.1/32 [150/0] via 192.168.209.44, enp0s9, 00:00:02
K>* 4.3.2.1/32 [0/14000] is directly connected, enp0s8, 00:01:05
S   4.3.2.1/32 [1/0] is directly connected, enp0s10, 16:30:20
K * 4.3.2.1/32 [255/8192] is directly connected, enp0s9, 16:33:31
```

Kernel

```
sharpd@robot ~> ip route show
default via 10.0.2.2 dev enp0s3 proto static metric 100
4.3.2.1 dev enp0s8 scope link metric 14000
4.3.2.1 dev enp0s9 scope link metric 4278198272
```

OSPF Overview

- Considered an IGP (Interior Gateway Protocol)
  - Uses Link State Routing
  - Every router floods information, so that every router has all the information
  - Dijkstra's Algorithm
- Uses IP (Layer 3) to advertise routing information
- OSPFv2 -vs- OSPFv3
  - IPv4 -vs- IPv6
- Supports multiple areas (and types of areas)
  - If you do more than one area you must have area 0

OSPF Basic Setup

```
# Put this in all FRR.conf
!
router ospf
network 0.0.0.0/0 area 0.0.0.0
!
```

OSPF Debugging

- show ip ospf interface
  - What Interfaces is OSPF using?
- show ip ospf neighbor
  - What neighbors have I formed?
- show ip ospf database
  - Dump of the OSPF database
- show ip ospf database router
  - Dump of what each router sent you
- debug ospf packet
  - Look at data sent/received from peers, stored in log file
- debug ospf nsm
  - Look at neighbor state change events, stored in log file

Create Linux Router

1. ดาวน์โหลดอิมเมจ Alpine

```
docker pull alpine:latest
```

2. Create 2 router containers: สร้างเร้าเตอร์จากนั้นกำหนดค่าต่าง ๆ ด้วยการสร้างเน็ตเวิร์กที่ชื่อว่า routernet และสร้าง router1 และ router2

```
docker network create --subnet=172.20.0.0/16 routernet

docker run -dit --name router1 --hostname router1 \
  --network routernet --ip 172.20.0.2 \
  --cap-add=NET_ADMIN --cap-add=NET_RAW \
  alpine:latest

docker run -dit --name router2 --hostname router2 \
  --network routernet --ip 172.20.0.3 \
  --cap-add=NET_ADMIN --cap-add=NET_RAW \
  alpine:latest
```

3. ติดตั้ง FRR ให้กับ router1 และ router2 ด้วยการเข้าไปใน shell ของเร้าเตอร์ทั้ง 2 ตัว

```
docker exec -it router1 sh

apk update
apk add frr frr-openrc ip6tables iptables
```

4. ติดตั้ง ospf ตั้งค่า /etc/frr/daemons เป็นรองรับการทำงานของ OSPF และ zebra (สามารถทำผ่าน nano) เพื่อกำหนดให้ค่า ospfd=yes และ zebra=yes

```
# Edit /etc/frr/daemons
sed -i s/ospfd=no/ospfd=yes/ /etc/frr/daemons
sed -i s/zebra=no/zebra=yes/ /etc/frr/daemons

# Enable IP forwarding
echo "net.ipv4.ip_forward=1" >> /etc/sysctl.conf
sysctl -p
```

5. ติดตั้ง OSPF:

```
# Start FRR
/usr/lib/frr/frrinit.sh start

# Configure via vtysh
vtysh

configure terminal
router ospf
 network 172.20.0.0/16 area 0
exit
exit
write memory
```

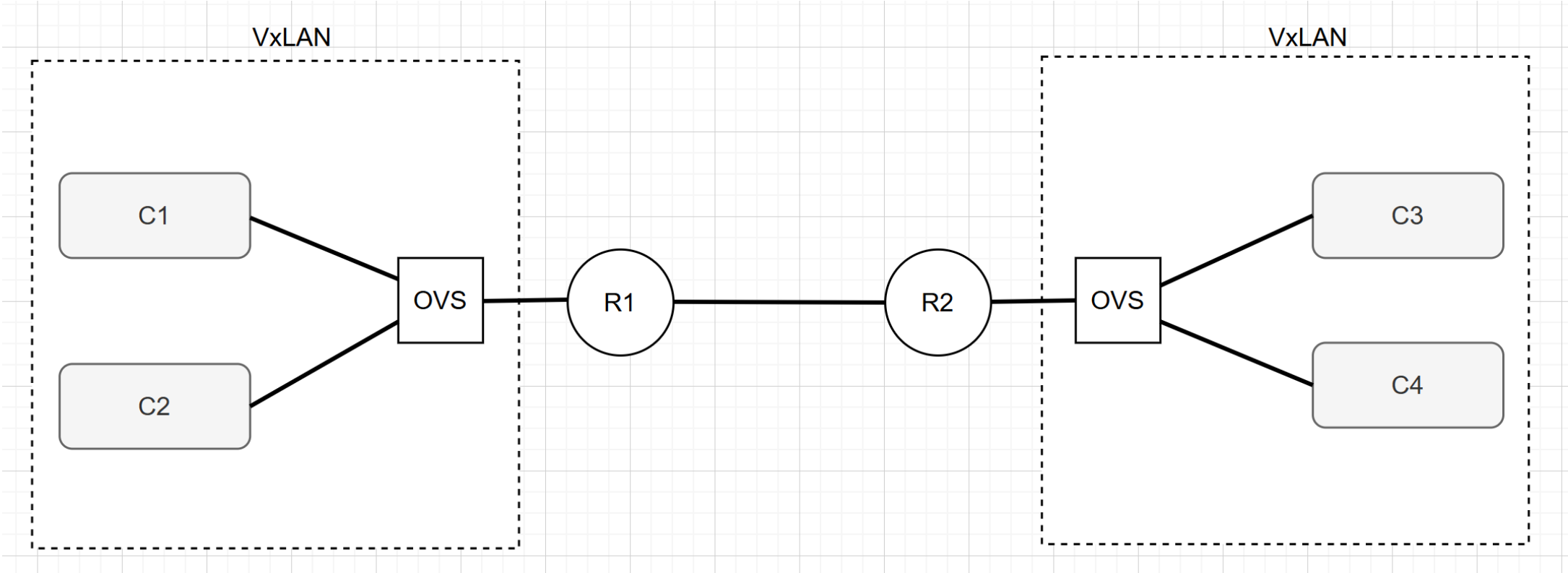
6. Verify:

```
# Check OSPF neighbors
vtysh -c "show ip ospf neighbor"

# Check routes
vtysh -c "show ip route"
```

กลับไปทำขั้นตอนที่ 3 - 5 ใหม่ที่ router 2 เพื่อให้ router ทั้งสองสามารถสื่อสารกันได้

Create this network



สร้าง Tolopogy hosts -- OVS -- routers -- OVS -- hosts.

```
# Install OVS on host
apk add openvswitch
rc-service ovs-modules start
rc-service ovsdb-server start
rc-service ovs-vswitchd start

# Create 2 OVS bridges
ovs-vsctl add-br ovs1
ovs-vsctl add-br ovs2

# Create router containers with multiple networks
docker run -dit --name router1 --hostname router1 \
  --cap-add=NET_ADMIN --cap-add=NET_RAW --network none alpine:latest

docker run -dit --name router2 --hostname router2 \
  --cap-add=NET_ADMIN --cap-add=NET_RAW --network none alpine:latest

# Create host containers
docker run -dit --name host1 --network none alpine:latest
docker run -dit --name host2 --network none alpine:latest

# Connect using ovs-docker (utility script)
ovs-docker add-port ovs1 eth0 host1 --ipaddress=10.1.1.10/24
ovs-docker add-port ovs1 eth0 router1 --ipaddress=10.1.1.1/24
ovs-docker add-port ovs2 eth0 host2 --ipaddress=10.2.2.10/24
ovs-docker add-port ovs2 eth0 router2 --ipaddress=10.2.2.1/24

# Create router interconnect network
docker network create --subnet=192.168.0.0/30 router-link
docker network connect router-link router1 --ip=192.168.0.1
docker network connect router-link router2 --ip=192.168.0.2
```

จาก **Topology** เดิม

- ทดสอบการสื่อสาร และสร้าง VxLAN ต่อไป