CodiLime Sp. z o.o.
CodiLime, Inc.
www.codilime.com
contact@codilime.com

Into the Unknown: Deep Dive into the Linux Network Stack

Lab manual

November 2026

CodiLime Sp. z o.o.
CodiLime, Inc.
www.codilime.com
contact@codilime.com

Contents

Into the Unknown: Deep Dive into the Linux Network Stack 4
- Introduction 4
- Labs overview 4
- Requirements 4
- Running a lab 5
- Accessing the nodes: docker exec instead of SSH 5
- If something goes wrong 6
- Getting help 6

LAB 1 – Native kernel features (VRF / NETNS / VETH / MPLS) 7
- Goals 7
- Topology 8
- Deploying the lab and accessing nodes 9
- Task 0 – Check the existing control plane 9
- Task 1 – Connect PPE and LAB with a veth pair (on the host) 11
- Task 2 – Configure PPE 11
- Task 3 – Configure LAB 12
- Task 4 – Send traffic and follow it hop by hop 14
- Troubleshooting checklist 16
- Cleanup 16

LAB 2 – Linux as an EVPN/VXLAN leaf 17
- Goals 17
- Topology 18
- Deploying the lab and accessing nodes 19
- Task 0 – Check the starting point 20
- Task 1 – Create the VRF 21
- Task 2 – Create one bridge per network and put them in the VRF 21
- Task 3 – Connect the VMs to the bridges 22
- Task 4 – Configure the gateways (SVIs) 22
- Task 5 – Create the VXLAN interfaces and attach them to the bridges 22
- Task 6 – Let FRR pick up the new configuration 23
- Task 7 – Test from the VMs 25
- Task 8 – Compare with the VLAN-aware bridge on BMS2 [optional] 26
- Troubleshooting checklist 27
- Cleanup 27

LAB 3 – Alternative DP - OVS/OVN 28
- Goals 28
- Topology 29
- Deploying the lab and accessing nodes 31
- Task 0 – Check the starting point 32

Task 1 – Create the logical switches ... 33
Task 2 – Create the logical router ... 33
Task 3 – Add router ports (one per network) ... 33
Task 4 – Connect the switches to the router ... 34
Task 5 – Create the DHCP options ... 34
Task 6 – Add the VM ports ... 35
Task 7 – Check the Southbound DB and OVS ... 36
Task 8 – Test from the VMs ... 37
OVN vs. EVPN (LAB 2 vs LAB 3) ... 39
Troubleshooting checklist ... 39
Cleanup ... 39

LAB 4 – DPDK & VPP ... 40
Goals ... 40
Topology ... 40
Deploying the lab and accessing nodes ... 42
Task 0 – Check the starting point ... 42
Task 1 – R2: memif master towards R1 and vhost-user towards L2 ... 43
Task 2 – R1: memif slave towards R2 ... 44
Task 3 – R3: vhost-user towards L2 ... 46
Task 4 – L2: testpmd as a cable between R2 and R3 ... 46
Task 5 – Check end-to-end routing ... 48
Task 6 – Follow a packet through the VPP graph on R2 [optional] ... 49
Troubleshooting checklist ... 50
Cleanup ... 50

LAB 5 – VRRP + GoBGP & Bird ... 51
Goals ... 51
Topology ... 52
Deploying the lab and accessing nodes ... 53
Task 0 – Check the starting point ... 54
Task 1 – www-1: GoBGP neighbor and keepalived actions ... 55
Task 2 – www-2: GoBGP neighbor and keepalived actions ... 57
Task 3 – BIRD: route reflector with an advanced export filter ... 57
Task 4 – Verify on R2 and test the service ... 61
Task 5 – Failover: www-1 → www-2 and back ... 62
Troubleshooting checklist ... 63
Cleanup ... 63

CodiLime Sp. z o.o.
CodiLime, Inc.
www.codilime.com
contact@codilime.com

Into the Unknown: Deep Dive into the Linux Network Stack

Introduction

Most people know Linux as a server operating system. This workshop shows Linux as a **network platform**: a router, a switch, an EVPN leaf or a BGP speaker that fits into the network around it. In the labs you build the data plane by hand, look at how the kernel (and the user-space data planes that replace it) forwards packets, and connect it to a standard routing control plane.

Every lab is a small network built with **containerlab**. Each node is a container, and the links between nodes are virtual Ethernet links. You deploy the lab with one command, do the exercises, and destroy it at the end.

Labs overview

Lab	Title	Summary
1	Native kernel features (VRF / NETNS / VETH / MPLS)	You add a plain Linux host to an MPLS L3VPN network using only `iproute2`: network namespaces, veth pairs, a VRF, a TAP interface and static MPLS routes. The host then pushes its own MPLS label stack into the core.
2	Linux as an EVPN/ VXLAN leaf	You turn a Linux host into an EVPN/VXLAN leaf with a VRF, Linux bridges and VXLAN interfaces (L2 and L3 VNIs). FRR provides the BGP EVPN control plane.
3	Alternative DP – OVS/ OVN	You build the same tenant network as in LAB 2 with OVN on top of Open vSwitch, configured from one central place. OVN also provides routing and DHCP.
4	DPDK & VPP	You connect three routers that use VPP as the data plane and FRR as the control plane, with memif, vhost-user and DPDK virtio_user links. The VPP linux-cp plugin syncs addresses and routes from Linux into VPP.
5	VRRP + GoBGP & Bird	Two web servers share one service IP. keepalived (VRRP) picks the active server, GoBGP advertises the IP only from that server, and a BIRD route reflector changes route attributes based on BGP communities.

The labs are fully independent. No lab uses the results of another lab. If you skip a lab or do not finish it, the next labs still work.

Requirements

- **A dedicated VM.** Each participant gets their own virtual machine with SSH access. Everything the labs need (containerlab, Docker, container images, lab files) is already installed on it.
- **Login details.** Your login, password and the VM's domain name are given to you before the first lab.
- **An SSH client** on your laptop:
  - **Linux / macOS:** the built-in `ssh` command in a terminal.
  - **Windows:** the built-in OpenSSH client (`ssh` in PowerShell or Windows Terminal), or a client such as PuTTY.

```
ssh <login>@<vm-domain-name>
```

- **No file editing on the VM.** You type all commands yourself in the terminal. You do not need to edit any files on the VM, so you do not need a remote editor.

Running a lab

Each lab is in its own `labs1` subdirectory (`lab1` … `lab5`) in your home directory on the VM. The basic flow is the same for every lab:

```
cd ~/labs/lab1
containerlab deploy -t lab1.clab.yml     # build the lab
containerlab inspect -t lab1.clab.yml    # list nodes and management IPs
# ... exercises ...
containerlab destroy -t lab1.clab.yml    # remove the lab
```

All labs uses the same management network (`172.16.0.0/24`). **Destroy the current lab before you deploy the next one.**

Accessing the nodes: `docker exec` instead of SSH

Every node in a lab has an SSH server, and the root key is already installed, so `ssh root@<management-ip>` or `ssh root@<node-name>` works. In the labs we use `docker exec` instead, because it is faster and simpler. Example:

```
docker exec -it <node> sh        # e.g. docker exec -it ppe sh
```

What this command does:

- `docker exec` starts a new process **inside a running container** (emulating a device). Here the process is a shell (`sh` or `bash`).
- `-i` keeps the input open and `-t` allocates a terminal, so you get a normal interactive shell.
- The shell runs in the container's namespaces, so it sees the container's network interfaces, routing tables and processes, the same as a session opened with SSH.

How it differs from SSH:

	`docker exec`	`ssh root@<ip>`
Path	Local call to the Docker engine on the VM	TCP connection to the SSH server in the node, over the management network
Target	Container name	Management IP address
Authentication	None (you already have access to Docker)	Root key
Needs network and an SSH server in the node	No	Yes

For the labs there is no difference. In both cases you get a root shell in the node and see exactly the same network state.

To leave the node's shell, type `exit` or press `Ctrl+D`.

If something goes wrong

1. **Rebuild the lab.** You can always destroy the lab and deploy it again. This gives you a clean lab in its start state:

   ```
   containerlab destroy -t labN.clab.yml
   containerlab deploy -t labN.clab.yml
   ```

2. **Restore from backup.** If the step above does not help (for example, lab files were deleted or changed), run the recovery script. It restores all labs from a backup:

   ```
   /usr/local/bin/recover_labs.sh
   ```

Getting help

If you have a question or get stuck, do not hesitate to call the instructor.

There is also a Slack channel for this workshop: **#ac6-ws-a3-into-the-unknown**. If you do not have access to that Slack workspace yet, join with this invitation link:

https://join.slack.com/t/networkautomationfrm/shared_invite/zt-4c0zlio86-l~zhtYRJP8f6UTM_GLkPdA

Follow the steps of each lab precisely but most importantly - HAVE FUN!

CodiLime Sp. z o.o.
CodiLime, Inc.
www.codilime.com
contact@codilime.com

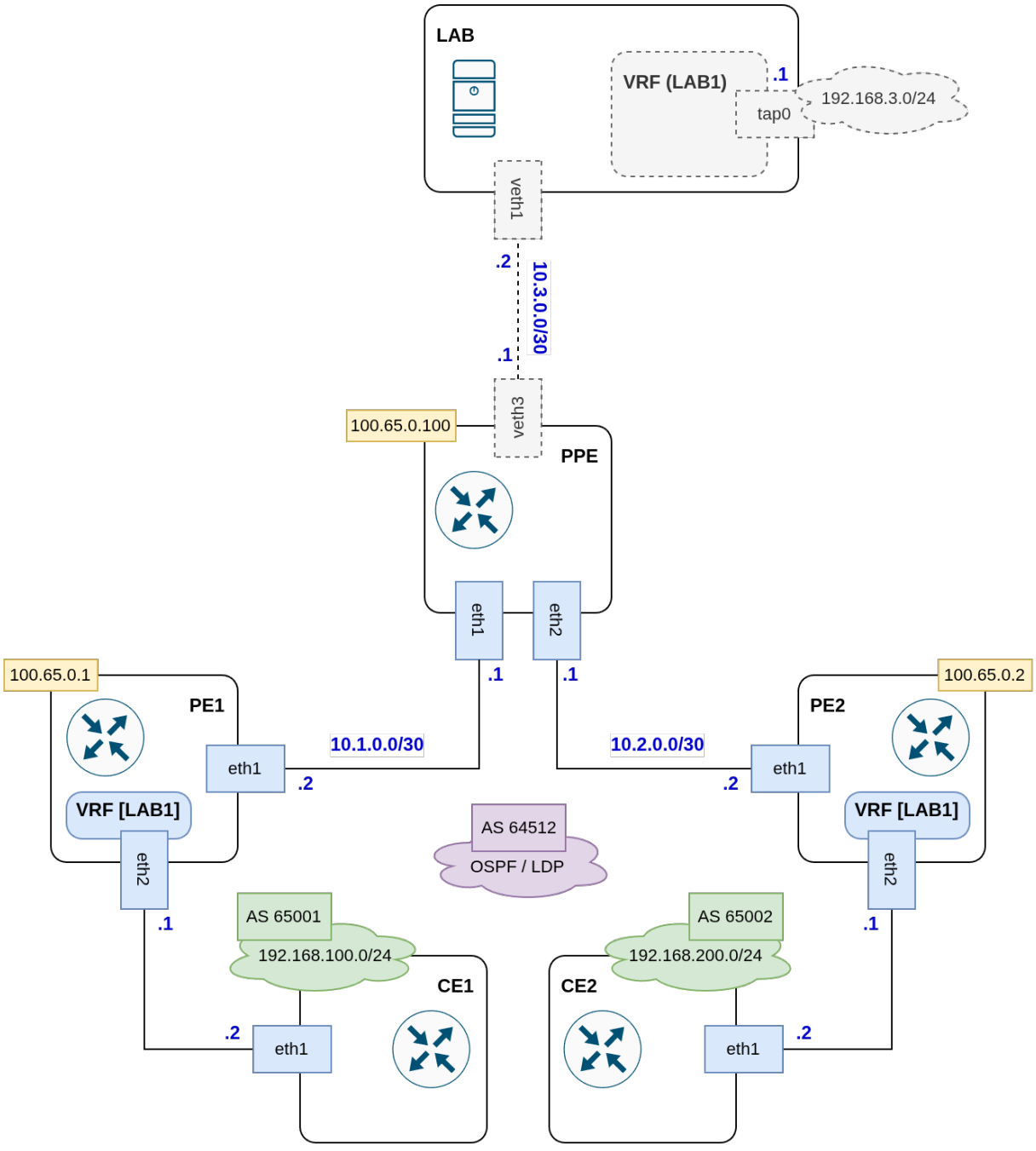
LAB 1 – Native kernel features (VRF / NETNS / VETH / MPLS)

Goals

In this lab you will build part of the data plane by hand, using only `iproute2` commands. After this lab you should be able to:

- explain how containers map to Linux **network namespaces (netns)**,
- connect two namespaces with a **veth** pair,
- create a Linux **VRF** and attach interfaces to it,
- create a **TAP** interface and inject packets into the kernel through it,
- read and modify the **IPv4** and **MPLS** routing tables (static routes, MPLS push / pop),
- check the **OSPF / LDP / BGP (L3VPN)** control plane run by FRR.

Topology



Node	Role	Interface	Address	Notes
CE1	Customer router, AS65001	eth1	192.168.1.2/30	lo: 192.168.100.1/24
PE1	PE router, AS64512	eth1	10.1.0.2/30	lo: 100.65.0.1/32
		eth2	192.168.1.1/30	in VRF `LAB1` (table 101)
PPE	P router + BGP RR	eth1	10.1.0.1/30	lo: 100.65.0.100/32
		eth2	10.2.0.1/30	
		veth3	10.3.0.1/30	created in this lab

Node	Role	Interface	Address	Notes
PE2	PE router, AS64512	eth1	10.2.0.2/30	lo: 100.65.0.2/32
		eth2	192.168.2.1/30	in VRF `LAB1` (table 101)
CE2	Customer router, AS65002	eth1	192.168.2.2/30	lo: 192.168.200.1/24
LAB	Plain Linux host	veth1	10.3.0.2/30	created in this lab
		tap0	192.168.3.1/24	in VRF `LAB1` (table 101)

Management addresses (network `172.16.0.0/24`): PPE `.10`, PE1 `.11`, PE2 `.12`, LAB `.13`, CE1 `.21`, CE2 `.22`.

What is already configured when the lab starts:

- FRR on CE1, PE1, PPE, PE2 and CE2 (OSPF + LDP in the core, iBGP VPNv4 with PPE as the route reflector, eBGP between PE and CE inside VRF `LAB1`).
- VRF `LAB1` on PE1 and PE2.
- On PE1, a static MPLS route: label `1000` → pop and look up in VRF `LAB1`.

Your job is to add the **LAB** node to this network. LAB acts like a very simple PE: it sends traffic from its own VRF, with an MPLS label stack it pushes itself, over a veth link into the MPLS core.

Deploying the lab and accessing nodes

Go to the lab directory, deploy the lab and list its nodes:

```
cd ~/labs/lab1
containerlab deploy -t lab1.clab.yml
containerlab inspect -t lab1.clab.yml
```

`containerlab inspect` lists six nodes (`ce1`, `ce2`, `lab`, `pe1`, `pe2`, `ppe`) in state `running`, with their management IP addresses.

> **Tip:** In Task 4 you will need several terminals open at the same time.

> **Note:** Do **not** run `./setup.sh` or `/tmp/setup.sh`. Those scripts are the reference solution. In this lab you type every command yourself.

Task 0 – Check the existing control plane

Before you change anything, make sure the core works.

0.1 OSPF and LDP (on PPE) [optional]

Check the OSPF neighbors, the LDP sessions and the LDP label bindings on PPE:

```
docker exec -it ppe vtysh -c "show ip ospf neighbor"
docker exec -it ppe vtysh -c "show mpls ldp neighbor"
docker exec -it ppe vtysh -c "show mpls ldp binding"
```

Expected result: PPE has two OSPF neighbors (100.65.0.1 and 100.65.0.2) in state `Full`, and two LDP sessions in state `OPERATIONAL`.

0.2 MPLS forwarding table in the kernel [optional]

FRR (zebra) programs the LDP labels into the kernel. Check them with `iproute2`:

```
docker exec -it ppe ip -f mpls route show
docker exec -it pe1 ip -f mpls route show
```

On PE1 you should also see the static route that was added at deploy time:

```
1000 dev LAB1
```

0.3 VRF on PE1 [optional]

Look at VRF `LAB1` on PE1: its routing table, its interfaces and its routes:

```
docker exec -it pe1 ip vrf show
docker exec -it pe1 ip -d link show LAB1
docker exec -it pe1 ip link show master LAB1
docker exec -it pe1 ip route show vrf LAB1
```

Expected result: VRF `LAB1` uses table `101`. `eth2` is enslaved to it. The VRF table has the CE1 prefix `192.168.100.0/24` (learned by eBGP) and the CE2 prefix `192.168.200.0/24` (learned over the L3VPN, with an MPLS `encap`).

0.4 BGP / L3VPN [optional]

Check the iBGP sessions on the route reflector (PPE), the VPNv4 routes and the VRF routes on PE1, and the BGP routes that CE1 receives:

```
docker exec -it ppe vtysh -c "show bgp summary"
docker exec -it pe1 vtysh -c "show bgp ipv4 vpn"
docker exec -it pe1 vtysh -c "show bgp vrf LAB1 ipv4 unicast"
docker exec -it ce1 vtysh -c "show ip route bgp"
```

Expected result: PPE has two iBGP neighbors (`100.65.0.1` and `100.65.0.2`), each with 2 received prefixes. PE1 has the VPNv4 routes of both sites (Route Distinguishers `100.65.0.1:101` and `100.65.0.2:101`), and VRF `LAB1` on PE1 has `192.168.100.0/24` and `192.168.200.0/24`. CE1 has a BGP route to `192.168.200.0/24` via `192.168.1.1` (PE1).

0.5 End-to-end check CE1 ↔ CE2

Ping the CE2 loopback from the CE1 loopback:

```
docker exec -it ce1 ping -c 3 -I 192.168.100.1 192.168.200.1
```

The ping must work. This traffic goes CE1 → PE1 (VRF) → MPLS core → PE2 (VRF) → CE2.

Task 1 – Connect PPE and LAB with a veth pair (on the host)

Every container has its own **network namespace**. Containerlab registers these namespaces under `/var/run/netns`, using the node names. This means you can manage them from the host with `ip netns` and `ip -n <netns>`.

1.1 List the namespaces

List the network namespaces on the host:

```
sudo ip netns list
ls -l /var/run/netns/
```

You should see `ppe`, `pe1`, `pe2`, `ce1`, `ce2` and `lab`.

1.2 Create the veth pair

A veth pair is a virtual cable: whatever goes into one end comes out of the other end. Create the pair in the host namespace:

```
sudo ip link add veth3 type veth peer veth1
ip link show type veth | grep -E 'veth1|veth3'
```

You should see two interfaces, `veth1@veth3` and `veth3@veth1`, both in state `DOWN`. The name after `@` is the other end of the pair.

1.3 Move each end into its namespace

Move `veth3` into the PPE namespace and `veth1` into the LAB namespace:

```
sudo ip link set veth3 netns ppe
sudo ip link set veth1 netns lab
```

The commands print nothing when they succeed.

1.4 Verify

Check that each end is now in its namespace and not on the host:

```
ip link show veth3                          # error: the device is not in the host netns
any more
sudo ip -n ppe link show veth3              # veth3@if<N> in PPE
sudo ip -n lab link show veth1              # veth1@if<N> in LAB
```

`-n` executes command in netns.

Task 2 – Configure PPE

Open a shell on PPE:

```
docker exec -it ppe bash
```

2.1 Let FRR configure veth3

The FRR config of PPE already contains a section for `veth3` (`10.3.0.1/30`, passive OSPF, `mpls enable`). The interface did not exist when FRR started, so reload FRR to apply this config:

```
sv reload /services/frr
```

Verify (wait a few seconds):

```
ip addr show veth3                          # 10.3.0.1/30 must be present
sysctl net.mpls.conf.veth3.input            # must be 1, so the kernel accepts MPLS on
veth3
vtysh -c "show interface veth3"             # will be "protocol DOWN"
vtysh -c "show ip ospf neighbor"            # OSPF and LDP must come back in Full state
```

> `mpls enable` in FRR sets `net.mpls.conf.<if>.input=1`. Without this setting, the kernel drops every MPLS packet that arrives on the interface.

2.2 Add a static MPLS "pop" route

LAB will send packets with the label stack `1000/1000`. PPE must remove (pop) the top label and send the packet to PE1 (`10.1.0.2`) through `eth1`. The inner label `1000` stays on the packet:

```
ip -f mpls route add 1000 via inet 10.1.0.2 dev eth1
```

Verify:

```
ip -f mpls route show
ip -f mpls route get 1000
```

Expected entry:

```
1000 via inet 10.1.0.2 dev eth1
```

> There is no `as <label>` in this route. This means pop: the kernel does not push a new label. Compare it with the LDP routes in the same table, which use `as to <label>` (swap). The `as <label>` can be seen on `pe1` / `pe2` device.

Exit the remote shell using:

```
exit
```

Task 3 – Configure LAB

Open a shell on LAB:

```
docker exec -it lab bash
```

3.1 Configure the veth1 interface

Give `veth1` (the other end of `veth3`) its address and bring it up:

```
ip addr add 10.3.0.2/30 dev veth1
ip link set veth1 up
```

Verify:

```
ip addr show veth1           # state UP, 10.3.0.2/30
ping -c 3 10.3.0.1           # PPE must reply
```

3.2 Create the VRF

Create VRF `LAB1` with routing table `101` and bring it up:

```
ip link add LAB1 type vrf table 101
ip link set LAB1 up
```

Verify:

```
ip vrf show
ip -d link show LAB1
```

`ip vrf show` lists `LAB1` with table `101`, and `ip -d link show LAB1` shows `vrf table 101`.

3.3 Add a route that pushes two MPLS labels

This route goes into the VRF table (101). It pushes two labels: the outer label is for PPE, and the inner label is the "VPN label" for PE1. This way LAB works like a PE router:

```
ip route add table 101 192.168.100.0/24 encap mpls 1000/1000 via 10.3.0.1 dev veth1
```

> `ip route add vrf LAB1 ...` does the same thing as `table 101`.

Verify:

```
ip route show vrf LAB1
ip route show table 101
```

Expected entry:

```
192.168.100.0/24  encap mpls  1000/1000 via 10.3.0.1 dev veth1
```

3.4 Create the TAP interface

A TAP device is a virtual Ethernet interface. A user-space program controls the other end: every frame that the program writes to `/dev/net/tun` (tun/tap in Linux are interchangeable) shows up in the kernel as a frame **received** on `tap0`.

```
ip tuntap add dev tap0 mode tap
ip link set tap0 addr 02:00:00:00:00:02
```

The test script sends frames with destination MAC `02:00:00:00:00:02`. This is why we set the MAC of `tap0` to that value. Otherwise the kernel would treat the frames as "not for me" and drop them.

3.5 Put tap0 in the VRF and assign an IP address

Attach `tap0` to VRF `LAB1`, give it an address and bring it up:

```
ip link set tap0 master LAB1
ip addr add 192.168.3.1/24 dev tap0
ip link set tap0 up
```

Verify:

```
ip link show master LAB1         # tap0 must be listed
ip -d link show tap0             # "tun type tap", MAC 02:00:00:00:00:02
ip addr show tap0
ip route show vrf LAB1           # connected 192.168.3.0/24 + MPLS route to
192.168.100.0/24
```

> `tap0` shows `NO-CARRIER / state DOWN` until a program opens it. This is normal.

3.6 Check IP forwarding

The kernel must forward packets from `tap0` to `veth1`:

```
sysctl net.ipv4.ip_forward       # must be 1
```

If it is `0`, enable it with `sysctl -w net.ipv4.ip_forward=1`.

Exit the device/docker shell using: `exit`

Task 4 – Send traffic and follow it hop by hop

The script `/tmp/send_tap.py` on LAB opens `tap0` and writes one Ethernet frame to it:

```
Ether 02:00:00:00:00:01 -> 02:00:00:00:00:02
IP    192.168.3.99      -> 192.168.100.100
ICMP  echo request
```

4.1 Start the captures (one terminal per command)

Start `tcpdump` at four points along the path, each in its own terminal:

```
# Terminal 2 – PPE, link from LAB (expect 2 labels)
docker exec -it ppe tcpdump -nei veth3 mpls
```

```
# Terminal 3 – PPE, link to PE1 (expect 1 label)
docker exec -it ppe tcpdump -nei eth1 mpls
```

4.1.1 Start the additional captures [optional]

```
# Terminal 4 – PE1, core link (expect 1 label)
docker exec -it pe1 tcpdump -nei eth1 mpls
```

```
# Terminal 5 – PE1, link to CE1 (expect plain IP/ICMP)
docker exec -it pe1 tcpdump -nei eth2 icmp
```

4.2 Send the packet

In a first terminal, send one test frame from LAB:

```
docker exec -it lab /tmp/send_tap.py
```

Output: `Sent 42 bytes to tap0`. Run it a few times if needed.

4.3 Expected results

Capture point	What you should see
PPE `veth3`	`MPLS (label 1000 ...) (label 1000 ..., [S] ...) IP 192.168.3.99 > 192.168.100.100: ICMP echo request`
PPE `eth1`	`MPLS (label 1000, ..., [S] ...) IP 192.168.3.99 > 192.168.100.100: ICMP echo request` (outer label popped)
PE1 `eth1`	Same as PPE `eth1`
PE1 `eth2`	`IP 192.168.3.99 > 192.168.100.100: ICMP echo request` (no labels, sent to CE1)

`[S]` marks the bottom of the label stack. Also check the MPLS TTL at each hop.

> CE1 has no route back to `192.168.3.0/24`, so no echo reply is expected. The goal is to follow the request through the network.

4.4 Packet walk – what happened?

1. **LAB**: `send_tap.py` writes the frame to `tap0`. The kernel receives it on `tap0`. `tap0` is in VRF `LAB1`, so the kernel looks up the route in table 101. The route for `192.168.100.0/24` pushes the labels `1000/1000` and sends the packet out of `veth1` to `10.3.0.1`.
2. **veth pair**: the frame leaves `veth1` (netns `lab`) and arrives on `veth3` (netns `ppe`).
3. **PPE**: MPLS input is enabled on `veth3`. The kernel looks up label `1000` in the MPLS table. The rule is: pop, then send to `10.1.0.2` via `eth1`. The packet leaves with one label (`1000`).
4. **PE1**: MPLS input is enabled on `eth1` (FRR `mpls enable`). The kernel looks up label `1000`: pop and deliver to device `LAB1`. This means another IP lookup, this time in VRF `LAB1` (table 101).
5. **PE1 VRF LAB1**: `192.168.100.0/24` was learned from CE1 over eBGP. The packet leaves `eth2` as plain IP.

Troubleshooting checklist

Symptom	Check
Nothing on PPE `veth3`	LAB: `ip route show vrf LAB1`, `ip link show master LAB1`, MAC of `tap0`, `sysctl net.ipv4.ip_forward`
Packets on `veth3` but not on `eth1`	PPE: `sysctl net.mpls.conf.veth3.input`, `ip -f mpls route show`
Packets on PE1 `eth1` but not on `eth2`	PE1: `ip -f mpls route show` (route `1000 dev LAB1`), `ip route show vrf LAB1`
`encap mpls` route is rejected	Host: `lsmod | grep mpls` – modules `mpls_router` and `mpls_iptunnel` must be loaded
veth3 has no IP address	PPE: restart FRR again (`sv restart /services/frr`)

Look at kernel drop counters with `ip -s link show <if>`.

Cleanup

```
containerlab destroy -t lab1.clab.yml
```

The veth pair is deleted automatically when the `ppe` and `lab` namespaces are removed.

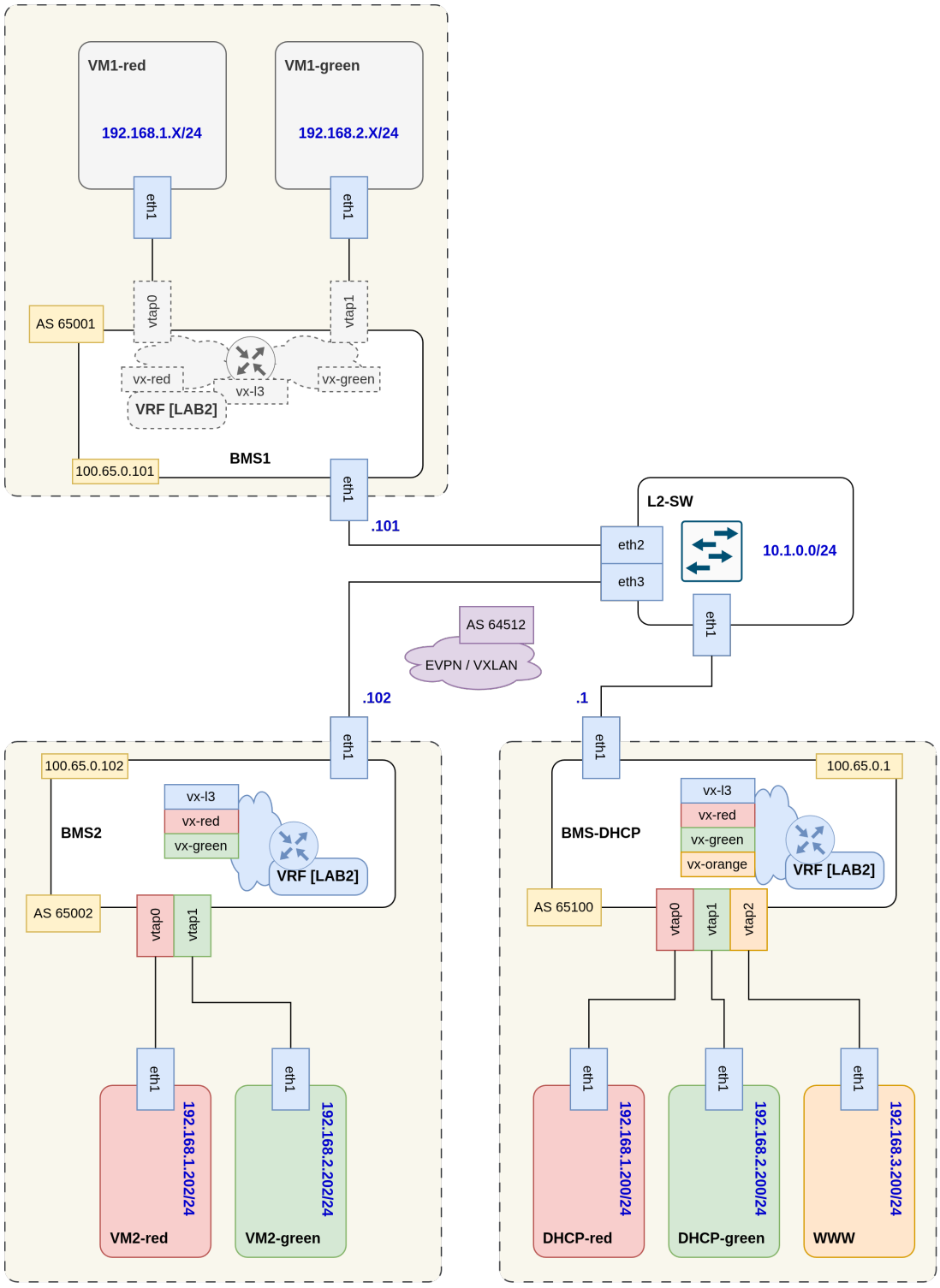
LAB 2 – Linux as an EVPN/VXLAN leaf

Goals

In this lab you turn a Linux host (**BMS1**) into an EVPN/VXLAN leaf. You build the data plane by hand with `iproute2`, and FRR provides the EVPN control plane. After this lab you should be able to:

- use a Linux **VRF** as the tenant routing table of an EVPN leaf,
- build **Linux bridges** that act as L2 segments and as gateways (SVIs),
- create **VXLAN** interfaces and attach them to bridges as **L2 VNIs** (bridging) and as an **L3 VNI** (routing between leaves),
- see how a **VLAN-aware bridge** (BMS2, BMS-DHCP) does the same job with one bridge,
- connect all of this to **FRR** (BGP EVPN), and check the result in FRR and in the kernel.

Topology



Tenant networks (VRF `LAB2`, table 101)

Network	Subnet	Gateway (anycast)	VLAN (BMS2/BMS-DHCP)	VNI	Present on
red	192.168.1.0/24	192.168.1.1	100	100	BMS1, BMS2, BMS-DHCP

Network	Subnet	Gateway (anycast)	VLAN (BMS2/BMS-DHCP)	VNI	Present on
green	192.168.2.0/24	192.168.2.1	200	200	BMS1, BMS2, BMS-DHCP
orange	192.168.3.0/24	192.168.3.1	300	300	BMS-DHCP only
L3 VNI	–	–	1000	1000	BMS1, BMS2, BMS-DHCP

Hosts

Node	Address	Connected to	Notes
vm1-red	DHCP (red)	BMS1 `vtap0`	gets its lease from dhcp-red
vm1-green	DHCP (green)	BMS1 `vtap1`	gets its lease from dhcp-green
vm2-red	192.168.1.202/24	BMS2 `vtap0`	static
vm2-green	192.168.2.202/24	BMS2 `vtap1`	static
dhcp-red	192.168.1.200/24	BMS-DHCP `vtap0`	dnsmasq, pool 192.168.1.10–100
dhcp-green	192.168.2.200/24	BMS-DHCP `vtap1`	dnsmasq, pool 192.168.2.10–100
www	192.168.3.200/24	BMS-DHCP `vtap2`	web server

Management addresses (network `172.16.0.0/24`): BMS1 `.11`, BMS2 `.12`, BMS-DHCP `.13`, vm1-red `.21`, vm1-green `.22`, vm2-red `.23`, vm2-green `.24`, dhcp-red `.31`, dhcp-green `.32`, www `.33`, l2-switch `.40`.

Control plane (FRR)

- **Underlay:** eBGP IPv4 unicast over `10.1.0.0/24`. BMS1 (AS 65001) and BMS2 (AS 65002) peer with BMS-DHCP (AS 65100) and advertise their loopbacks. The route-map `ZEBRA-SRC` sets each loopback as the source address for these routes, so VXLAN packets leave with the loopback (VTEP) address as source.
- **Overlay:** iBGP AS 64512, address family `l2vpn evpn`, between loopbacks. BMS-DHCP is the route reflector.
- `vrf LAB2` / `vni 1000`: VRF `LAB2` uses VNI 1000 as its L3 VNI. `advertise ipv4 unicast`: the VRF's connected subnets are advertised as EVPN type-5 (prefix) routes.

What is already configured when the lab starts:

- FRR on BMS1, BMS2 and BMS-DHCP (the config above).
- The full data plane on **BMS2** and **BMS-DHCP**, using one VLAN-aware bridge `br0`.
- Static addresses on vm2-, *dhcp-* and www. The DHCP servers are running.

Your job is to build the data plane on **BMS1**. To keep it simple you do **not** use VLANs on BMS1. Instead, you create a separate bridge for each network.

Deploying the lab and accessing nodes

Go to the lab directory, deploy the lab and list its nodes:

```
cd ~/labs/lab2
containerlab deploy -t lab2.clab.yml
containerlab inspect -t lab2.clab.yml
```

`containerlab inspect` lists eleven nodes in state `running`, with their management IP addresses.

> **Note:** Do **not** run `./setup.sh` or `/tmp/setup.sh`. Those scripts are the reference solution. In this lab you type every command yourself.

Task 0 – Check the starting point

0.1 Underlay on BMS1

Check the BGP sessions and the kernel routes on BMS1, then ping the BMS2 loopback from the BMS1 loopback:

```
docker exec -it bms1 vtysh -c "show bgp ipv4 unicast summary"
docker exec -it bms1 vtysh -c "show bgp l2vpn evpn summary"
docker exec -it bms1 ip route
docker exec -it bms1 ping -c 3 -I 100.65.0.101 100.65.0.102
```

Expected result:

- The eBGP session to `10.1.0.1` is `Established`.
- The iBGP EVPN session to `100.65.0.1` is `Established` (exchanged prefixes are greater or equal 0).
- The kernel has routes to `100.65.0.1/32` and `100.65.0.102/32` (`proto bgp`, `src 100.65.0.101`).
- The loopback-to-loopback ping works.

0.2 BMS1 has no tenant config yet [optional]

List the interfaces, VRFs and EVPN VNIs on BMS1:

```
docker exec -it bms1 ip link
docker exec -it bms1 ip vrf show
docker exec -it bms1 vtysh -c "show evpn vni"
```

`vtap0` and `vtap1` (the links to vm1-red and vm1-green) exist, but they are not attached to anything. (there is no `master` field in those interfaces). There is no VRF yet. FRR shows only VNI `1000` (type `L3`) from its own config, with no VXLAN interface and tenant VRF `Unknown`.

0.3 EVPN routes from the other leaves [optional]

Show the EVPN type-3 (multicast) and type-5 (prefix) routes that BMS1 receives:

```
docker exec -it bms1 vtysh -c "show bgp l2vpn evpn route type multicast"
docker exec -it bms1 vtysh -c "show bgp l2vpn evpn route type prefix"
```

BGP already receives routes from BMS2 and BMS-DHCP: type-3 (one per VNI per VTEP) and type-5 (subnets of VRF `LAB2`). They are not used yet, because BMS1 has no matching VNI or VRF.

Task 1 – Create the VRF

Open a shell on BMS1:

```
docker exec -it bms1 bash
```

Create VRF `LAB2` with routing table `101` and bring it up:

```
ip link add LAB2 type vrf table 101
ip link set LAB2 up addrgen none
```

`addrgen none` disables automatic IPv6 link-local address generation. The VRF device, bridge ports and VXLAN devices do not need their own addresses.

Verify:

```
ip vrf show                     # LAB2 -> table 101
ip -d link show LAB2
```

`ip -d link show LAB2` shows `vrf table 101`.

Task 2 – Create one bridge per network and put them in the VRF

On BMS1 each network gets its own, non-VLAN-aware bridge:

- `br-red`: L2 segment red (VNI 100), with gateway `192.168.1.1`
- `br-green`: L2 segment green (VNI 200), with gateway `192.168.2.1`
- `br-l3`: the L3 VNI bridge (VNI 1000). It has no IP address.

```
ip link add br-red type bridge
ip link add br-green type bridge
ip link add br-l3 type bridge
```

Put the bridges in the VRF. The bridge interface itself works as the SVI, so its IP address and connected route belong to table 101:

```
ip link set br-l3 master LAB2
ip link set br-red master LAB2
ip link set br-green master LAB2
```

Verify:

```
ip -d link show br-red          # "bridge ... vlan_filtering 0"
ip link show master LAB2        # br-red, br-green, br-l3
```

Task 3 – Connect the VMs to the bridges

`vtap0` goes to vm1-red and `vtap1` goes to vm1-green:

```
ip link set vtap0 master br-red addrgen none
ip link set vtap1 master br-green addrgen none
```

Verify:

```
bridge link show                # vtap0 master br-red, vtap1 master br-green
ip link show master br-red
ip link show master br-green
```

Task 4 – Configure the gateways (SVIs)

Assign the anycast gateway addresses to the red and green bridges:

```
ip addr add 192.168.1.1/24 dev br-red
ip addr add 192.168.2.1/24 dev br-green
```

Set the MAC addresses and bring the bridges up:

```
ip link set br-red addr 00:00:5e:00:10:01 up
ip link set br-green addr 00:00:5e:00:10:01 up
ip link set br-l3 addr 00:00:5e:00:f1:01 up
```

- **br-red / br-green**: all leafs use the same gateway IP **and** the same gateway MAC (`00:00:5e:00:10:01`). This is called a *distributed anycast gateway*. A VM always sees the same default gateway, whichever leaf it is connected to.
- **br-l3**: its MAC is this leaf's **router MAC** (RMAC). BGP sends it in EVPN type-2 and type-5 routes. Remote leafs use it as the inner destination MAC when they route traffic to this leaf. The RMAC **must be unique per leaf**: BMS1 `…f1:01`, BMS2 `…f1:02`, BMS-DHCP `…f0:01`. If two leaves had the same RMAC, each would reject the other's routes.

Verify:

```
ip -br addr show vrf LAB2       # br-red and br-green has IP assigned
ip -br link show master LAB2    # MAC matches, all bridges being down is expected
ip route show vrf LAB2          # connected 192.168.1.0/24 and 192.168.2.0/24
```

Task 5 – Create the VXLAN interfaces and attach them to the bridges

Create one VXLAN device per VNI. `dstport 4789` is the standard VXLAN UDP port. `nolearning` disables data-plane MAC learning on the VXLAN device: EVPN (FRR) installs the remote MACs instead.

```
ip link add vx-l3 type vxlan dstport 4789 vni 1000 nolearning
ip link add vx-red type vxlan dstport 4789 vni 100 nolearning
ip link add vx-green type vxlan dstport 4789 vni 200 nolearning
```

Attach each one to its bridge and bring it up:

```
ip link set vx-l3 master br-l3 addrgenmode none
ip link set vx-red master br-red addrgenmode none
ip link set vx-green master br-green addrgenmode none
ip link set vx-l3 up
ip link set vx-red up
ip link set vx-green up
```

Disable MAC learning on the VXLAN bridge ports. FRR installs the remote MACs on these ports:

```
bridge link set dev vx-red learning off
bridge link set dev vx-green learning off
bridge link set dev vx-l3 learning off
```

Verify:

```
ip -d link show vx-red           # "vxlan id 100 ... dstport 4789 nolearning"
ip -d link show vx-l3            # "vxlan id 1000 ..."
bridge link show                 # vx-red -> br-red, vx-green -> br-green, vx-l3 -> br-
l3
ip link show master br-red       # vtap0 + vx-red
ip -br link show master LAB2     # br-red, br-green, br-l3
```

The data plane on BMS1 now looks like this:

```
VRF LAB2 (table 101)
 ├── br-red   192.168.1.1/24  [vtap0, vx-red   (VNI 100)]
 ├── br-green 192.168.2.1/24  [vtap1, vx-green (VNI 200)]
 └── br-l3    (no IP, RMAC)   [vx-l3 (VNI 1000)]
```

Task 6 – Let FRR pick up the new configuration

The FRR config has `vrf LAB2` / `vni 1000`. The VRF and `vx-l3` did not exist when FRR started, so reload FRR to apply the config again:

```
sv reload /services/frr
```

Wait about 15 seconds for EVPN to converge.

6.1 VNIs in FRR

Check the VNIs that FRR now knows on BMS1:

```
vtysh -c "show evpn vni"
vtysh -c "show vrf vni"
vtysh -c "show evpn vni 100"
vtysh -c "show evpn vni 1000"
```

Expected result:

- VNI `100` and `200` are type `L2` (VXLAN IF `vx-red` / `vx-green`), and VNI `1000` is type `L3` (`vx-l3`). The tenant VRF of all three is `LAB2`.
- VNI 100 and 200 each have two remote VTEPs: `100.65.0.1` and `100.65.0.102`.
- `show evpn vni 1000` shows Router MAC `00:00:5e:00:f1:01` (the MAC of `br-l3`).

6.2 EVPN routes in BGP [optional]

Check EVPN routes from FRR perspective:

```
vtysh -c "show bgp l2vpn evpn summary"                # BGP session should be
established
vtysh -c "show bgp l2vpn evpn route type multicast"   # type-3: VTEP membership per VNI
vtysh -c "show bgp l2vpn evpn route type macip"       # MAC / MAC+IP of hosts
vtysh -c "show bgp l2vpn evpn route type prefix"      # type-5: subnets in VRF LAB2
vtysh -c "show bgp vrf LAB2 ipv4 unicast"
```

You should now also see type-3 routes originated by `100.65.0.101` (BMS1).

6.3 Remote VTEPs and router MACs [optional]

Check VTEP and MACs from FRR perspective:

```
vtysh -c "show evpn rmac vni all"
vtysh -c "show evpn next-hops vni all"
```

Expected result: one RMAC, `00:00:5e:00:f0:01` → `100.65.0.1` (BMS-DHCP). BMS-DHCP is the only leaf with a subnet BMS1 doesn't have (orange, 192.168.3.0/24). BMS2 has the same subnets as BMS1, so its type-5 routes are never best on BMS1. BMS2's RMAC (`00:00:5e:00:f1:02`) shows up in Task 7, after BMS1 learns a host behind BMS2 through a type-2 MAC+IP route.

6.4 What FRR programmed into the kernel

The **flood list** (BUM traffic, head-end replication) is built from type-3 routes. It is stored as all-zero MAC entries on each L2 VXLAN device:

```
bridge fdb show dev vx-red | grep 00:00:00:00:00:00
# 00:00:00:00:00:00 dst 100.65.0.1 self permanent
# 00:00:00:00:00:00 dst 100.65.0.102 self permanent
```

Routing to other leaves goes through the L3 VNI. Remote prefixes point to the remote VTEP address `onlink` via `br-l3`. A static neighbor entry and an FDB entry map that address to the remote RMAC:

```
ip route show vrf LAB2                    # e.g. 192.168.3.0/24 via 100.65.0.1 dev br-
l3 proto bgp ... onlink
ip neigh show dev br-l3                   # 100.65.0.1 lladdr 00:00:5e:00:f0:01
extern_learn
bridge fdb show dev vx-l3                 # 00:00:5e:00:f0:01 dst 100.65.0.1 ...
extern_learn
```

`extern_learn` means that the entry was installed by the control plane (FRR/zebra), not learned from the data plane.

Exit the remote shell using:

```
exit
```

Task 7 – Test from the VMs

Keep a capture of the VXLAN traffic running on BMS1 during this task:

```
# Separate terminal
docker exec -it bms1 tcpdump -nei eth1 udp port 4789
```

7.1 DHCP over the L2 VNI

vm1-red and vm1-green have no IP address. Their DHCP servers are connected to BMS-DHCP. The DHCP broadcast reaches them only because BMS1 floods it over VNI 100 / 200 to all remote VTEPs (the flood list from Task 6.4).

[Optional] watch the DHCP server side:

```
# Separate terminal
docker exec -it dhcp-red tcpdump -nei eth1 port 67 or port 68
```

Request the leases:

```
docker exec -it vm1-red udhcpc -q -i eth1
docker exec -it vm1-green udhcpc -q -i eth1
```

Verify:

```
docker exec -it vm1-red ip addr show eth1        # 192.168.1.x/24 from the 10–100 pool
docker exec -it vm1-red ip route                 # default via 192.168.1.1
docker exec -it vm1-green ip addr show eth1      # 192.168.2.x/24
docker exec -it vm1-green ip route               # default via 192.168.2.1
```

In the BMS1 capture you should see `VXLAN ... vni 100` (and `vni 200`) frames that carry the DHCP broadcast to `100.65.0.1` and `100.65.0.102`.

7.2 Gateway and bridging (same subnet, other leaf)

Ping the local gateway, and the VM in the same subnet behind BMS2:

```
docker exec -it vm1-red ping -c 3 192.168.1.1        # local anycast gateway on BMS1
docker exec -it vm1-red ping -c 3 192.168.1.202      # vm2-red behind BMS2 – bridged
over VNI 100
docker exec -it vm1-green ping -c 3 192.168.2.202    # vm2-green behind BMS2 – bridged
over VNI 200
```

All three pings must work (`0% packet loss`). In the BMS1 capture the pings to BMS2 show up as `vni 100` and `vni 200`.

7.3 Routing between subnets

Ping from red to green, and fetch the web page from the orange network:

```
docker exec -it vm1-red ping -c 3 192.168.2.202      # red -> green
docker exec -it vm1-red curl http://192.168.3.200    # red -> orange (www)
docker exec -it vm1-green curl http://192.168.3.200  # green -> orange (www)
```

`curl` must return the page `This is a NGINX at: www-orange`.

The orange network (VNI 300) exists only on BMS-DHCP. BMS1 has no `br-orange`, so it routes this traffic in VRF `LAB2` and sends it over the **L3 VNI**. In the capture you should see `vni 1000`, with the inner destination MAC set to the RMAC of BMS-DHCP (`00:00:5e:00:f0:01`).

7.4 MAC/IP learning after the traffic [optional]

Check the topology MAC and IP from Linux and EVPN perspective:

```
docker exec -it bms1 bridge fdb show br br-red | grep -v permanent
docker exec -it bms1 ip neigh show vrf LAB2
docker exec -it bms1 vtysh -c "show evpn mac vni 100"
docker exec -it bms1 vtysh -c "show evpn arp-cache vni 100"
docker exec -it bms1 vtysh -c "show bgp l2vpn evpn route type macip"
docker exec -it vm2-red ping -c 1 192.168.1.1         # make BMS2 learn vm2-red's IP
docker exec -it bms1 vtysh -c "show evpn rmac vni all"  # now also 00:00:5e:00:f1:02 ->
100.65.0.102
docker exec -it bms1 ip route show vrf LAB2             # 192.168.1.202 via
100.65.0.102 dev br-l3 ... onlink
```

- The MAC of vm1-red is learned **locally** on `vtap0`. FRR advertises it as a type-2 route.
- The MACs of vm2-red and dhcp-red are on `vx-red` as `extern_learn` entries with `dst 100.65.0.102` / `dst 100.65.0.1`. FRR installed them from type-2 routes.

Task 8 – Compare with the VLAN-aware bridge on BMS2 [optional]

BMS2 and BMS-DHCP build the same service with **one** VLAN-aware bridge `br0` (`vlan_filtering 1`):

- Each network is a VLAN on `br0`.
- The SVIs (`br-red`, `br-green`, `br-l3`) are VLAN sub-interfaces of `br0`.
- Each VXLAN device is a bridge port with its VLAN set as `pvid untagged`. This is how VLAN 100 maps to VNI 100, VLAN 200 to VNI 200, and VLAN 1000 to VNI 1000.

Look at the bridge, the VLANs and the FDB on BMS2:

```
docker exec -it bms2 ip -d link show br0            # "vlan_filtering 1"
docker exec -it bms2 ip -d link show br-red         # "vlan protocol 802.1Q id 100" on
top of br0
docker exec -it bms2 bridge vlan show               # VLANs per port (PVID / Egress
Untagged)
docker exec -it bms2 bridge vlan show dev vx-red    # vx-red: VLAN 100 PVID untagged
docker exec -it bms2 bridge fdb show br br0 vlan 100
docker exec -it bms2 bridge fdb show dev vx-red     # remote MACs + flood list
(00:00:00:00:00:00)
```

Item	BMS1 (this lab)	BMS2 / BMS-DHCP
Bridges	one per network (`br-red`, `br-green`, `br-l3`)	one VLAN-aware `br0`
SVI	the bridge itself	VLAN sub-interface of `br0` (`br-red` = `br0.100`)
VNI mapping	VXLAN device is a port of the bridge	VXLAN device port with `pvid untagged` VLAN
VM port mapping	port of the bridge	port with `pvid untagged` VLAN
FDB lookup key	MAC (per bridge)	MAC + VLAN

Both models work the same for FRR: zebra finds the VNI → bridge/VLAN → SVI mapping in the kernel.

Troubleshooting checklist

Symptom	Check (on BMS1)
`show evpn vni` is empty or shows no L3 VNI	`ip link show master LAB2`, `bridge link show`, then `sv reload / services/frr` again
No remote VTEPs on VNI 100/200	`vtysh -c "show bgp l2vpn evpn summary"`, `bridge fdb show dev vx-red`
DHCP does not work	`ip link show master br-red` (vtap0 + vx-red), flood list on `vx-red`
Same-subnet ping works, routing does not	`ip route show vrf LAB2`, `ip neigh show dev br-l3`, `vtysh -c "show evpn rmac vni all"`
Remote type-5 routes are not installed	MAC of `br-l3` must be unique per leaf
No VXLAN packets on `eth1`	`ip route` (loopbacks via `10.1.0.x`, `src 100.65.0.101`)

Cleanup

Destroy the lab:

```
containerlab destroy -t lab2.clab.yml
```

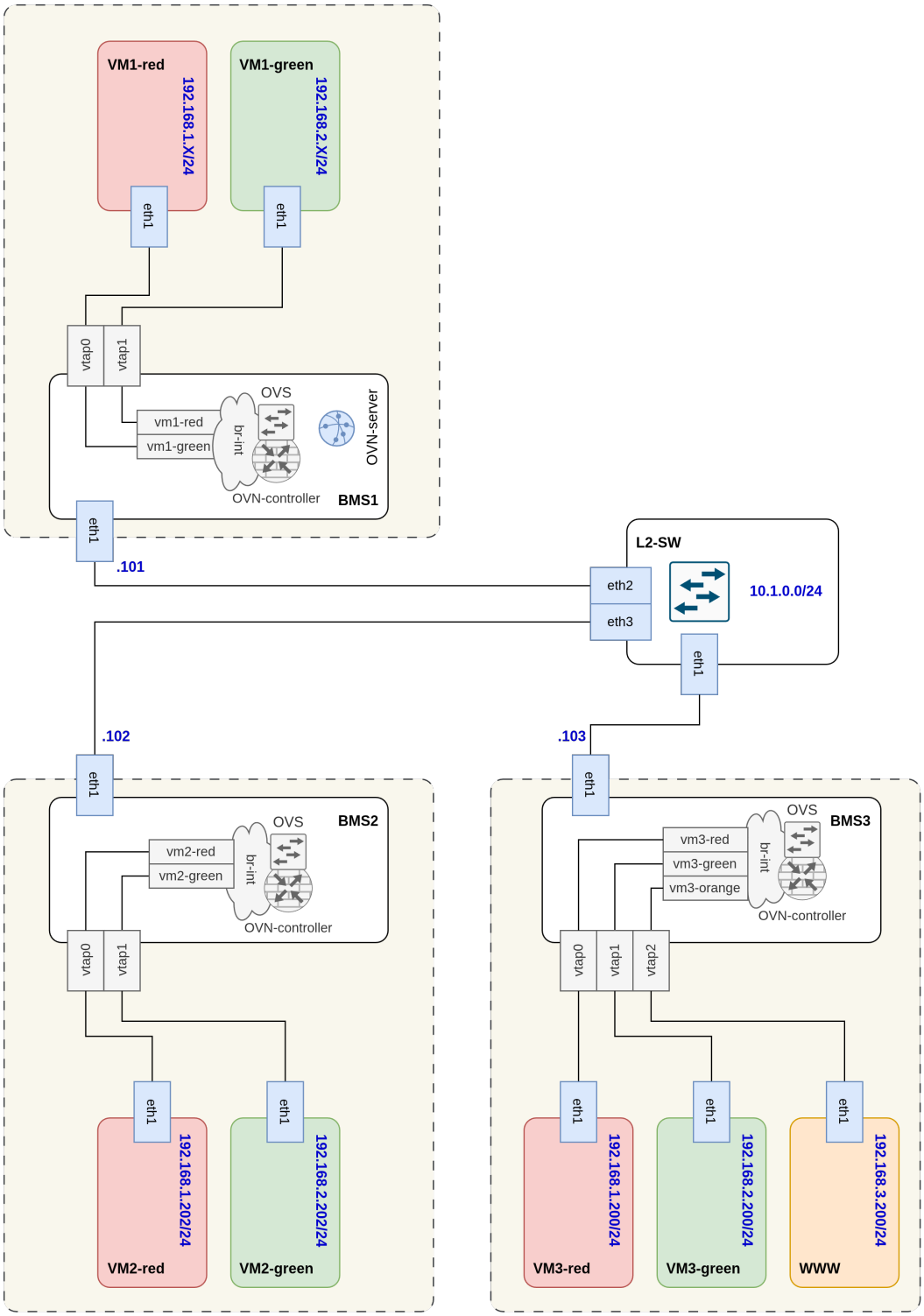
LAB 3 – Alternative DP - OVS/OVN

Goals

In this lab you build the same tenant network as in LAB 2 (red, green and orange networks connected by a router). This time you use **OVN** on top of **Open vSwitch** instead of BGP EVPN. You configure the whole network from **one** place (BMS1). After this lab you should be able to:

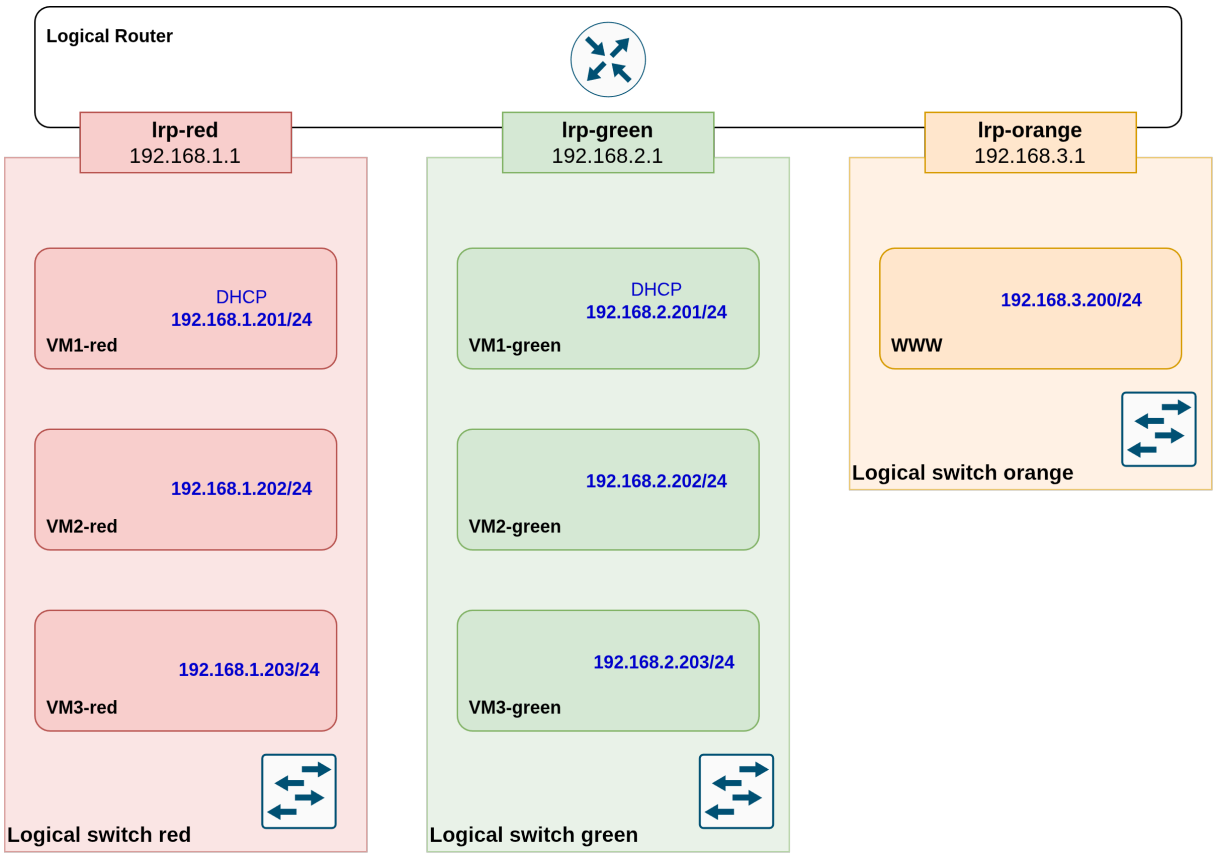
- explain the roles of **OVS** (`br-int`, OpenFlow, Geneve tunnels) and **OVN** (Northbound DB, `ovn-northd`, Southbound DB, `ovn-controller`),
- create **logical switches** (L2 networks) and connect VM ports to them,
- create a **logical router** and connect it to the switches,
- use OVN's native **DHCP**, with no separate DHCP server,
- check the result in the OVN databases, in OVS and on the wire, and trace a packet with `ovn-trace`.

Topology



Logical topology

The logical topology you build:



Tenant networks

Network	Logical switch	Subnet	Router port / gateway	DHCP
red	`red`	192.168.1.0/24	`lrp-red` 192.168.1.1, 02:ac:10:0a:00:01	OVN native, for vm1-red
green	`green`	192.168.2.0/24	`lrp-green` 192.168.2.1, 02:ac:10:0a:14:01	OVN native, for vm1-green
orange	`orange`	192.168.3.0/24	`lrp-orange` 192.168.3.1, 02:ac:10:0a:1e:01	options created, not used

Hosts

Node	Address	MAC (eth1)	Connected to	OVN logical port (`iface-id`)
vm1-red	DHCP → .1.201	02:0a:0a:00:10:11	BMS1 `vtap0`	`vm1-red`
vm1-green	DHCP → .2.201	02:0a:0a:00:20:11	BMS1 `vtap1`	`vm1-green`
vm2-red	192.168.1.202/24	random	BMS2 `vtap0`	`vm2-red`
vm2-green	192.168.2.202/24	random	BMS2 `vtap1`	`vm2-green`
vm3-red	192.168.1.203/24	random	BMS3 `vtap0`	`vm3-red`
vm3-green	192.168.2.203/24	random	BMS3 `vtap1`	`vm3-green`
www	192.168.3.200/24	random	BMS3 `vtap2`	`www`

Management addresses (network `172.16.0.0/24`): BMS1 `.11`, BMS2 `.12`, BMS3 `.13`, vm1-red `.21`, vm1-green `.22`, vm2-red `.23`, vm2-green `.24`, vm3-red `.31`, vm3-green `.32`, www `.33`, l2-switch `.40`.

OVN components

Component	Where	Role
Northbound DB	BMS1 (TCP 6641)	The desired logical network: switches, routers, ports, DHCP. You write here with `ovn-nbctl`.
`ovn-northd`	BMS1	Translates the NB DB into logical flows in the SB DB.
Southbound DB	BMS1 (TCP 6642)	Logical flows, chassis list, port bindings, MAC bindings.
`ovn-controller`	every BMS	Connects to the SB DB on BMS1. Programs OpenFlow on its local `br-int` and creates the Geneve tunnels.
OVS (`br-int`)	every BMS	The data plane. VM ports (`vtapX`) and tunnel ports are attached here.

> For simplicity all BMS nodes use the same image, so BMS2 and BMS3 also run their own NB/SB DB and `ovn-northd`. These are **not** used: every `ovn-controller` connects to `tcp:10.1.0.101:6642` (BMS1). Run all `ovn-nbctl` and `ovn-sbctl` commands on **BMS1**.

What is already configured when the lab starts:

- OVS and OVN services on BMS1, BMS2 and BMS3. Each chassis has a `system-id` (`BMS1`, `BMS2`, `BMS3`), `ovn-encap-type=geneve` and its own `ovn-encap-ip`.
- The VM ports `vtapX` are added to `br-int`. Each one has `external-ids:iface-id=<logical port name>`. When a logical port with that name appears in OVN, the chassis that has the interface **claims** it.
- Static addresses on vm2-, *vm3-* and www. vm1-red and vm1-green have fixed MAC addresses but no IP address.

Your job is to create the logical network in the OVN Northbound DB on BMS1.

Deploying the lab and accessing nodes

Go to the lab directory, deploy the lab and list its nodes:

```
cd ~/labs/lab3
containerlab deploy -t lab3.clab.yml
containerlab inspect -t lab3.clab.yml
```

`containerlab inspect` lists eleven nodes in state `running`, with their management IP addresses.

> **Note:** Do **not** run `./setup.sh` or `/tmp/setup.sh`. Those scripts are the reference solution. In this lab you type every command yourself.

Task 0 – Check the starting point

0.1 OVS on each BMS [optional]

Show the OVS bridge, the VM port settings and the OVN settings of BMS1:

```
docker exec -it bms1 ovs-vsctl show
docker exec -it bms1 ovs-vsctl --columns=name,external_ids list Interface vtap0 vtap1
docker exec -it bms1 ovs-vsctl get Open_vSwitch . external_ids
```

Expected result:

- `br-int` has the ports `vtap0` and `vtap1`.
- There are tunnel ports of type `geneve` to the other chassis (`remote_ip="10.1.0.102"`, `remote_ip="10.1.0.103"`).
- `vtap0` has `iface-id=vm1-red`, and `vtap1` has `iface-id=vm1-green`.
- `external_ids` has `system-id=BMS1`, `ovn-remote="tcp:10.1.0.101:6642"`, `ovn-encap-type=geneve` and `ovn-encap-ip="10.1.0.101"`.

You might want run the same commands on `bms2` and `bms3`. BMS3 also has `vtap2` (`iface-id=www`).

0.2 Chassis registered in the Southbound DB

Show the chassis that are registered in the Southbound DB:

```
docker exec -it bms1 ovn-sbctl show
```

Expected result: three chassis, `BMS1`, `BMS2` and `BMS3`, each with a `geneve` encap on its `10.1.0.10x` address. No port is bound yet.

0.3 The Northbound DB is empty [optional]

Show the logical network in the Northbound DB:

```
docker exec -it bms1 ovn-nbctl show
```

The output is empty. No logical switches or routers exist yet.

0.4 Underlay [optional]

Ping the tunnel (encap) addresses of BMS2 and BMS3 from BMS1:

```
docker exec -it bms1 ping -c 3 10.1.0.102
docker exec -it bms1 ping -c 3 10.1.0.103
```

Both pings must work (`0% packet loss`). The Geneve tunnels use these addresses.

Task 1 – Create the logical switches

Open a shell on BMS1 for Tasks 1–6:

```
docker exec -it bms1 bash
```

Each logical switch is one L2 network, the same as a bridge/VLAN plus L2 VNI in LAB 2. It spans all chassis automatically.

```
ovn-nbctl ls-add red
ovn-nbctl ls-add green
ovn-nbctl ls-add orange
```

Verify [optional]:

```
ovn-nbctl ls-list
ovn-nbctl show
```

Both commands list the three switches `green`, `orange` and `red`. They have no ports yet.

Task 2 – Create the logical router

Create the logical router `lr-bms`. It routes between the three networks:

```
ovn-nbctl lr-add lr-bms
```

Verify [optional]:

```
ovn-nbctl lr-list
```

The output lists one router, `lr-bms`.

Task 3 – Add router ports (one per network)

Each logical router port has a MAC address and the gateway IP of its network:

```
ovn-nbctl lrp-add lr-bms lrp-red    02:ac:10:0a:00:01 192.168.1.1/24
ovn-nbctl lrp-add lr-bms lrp-green  02:ac:10:0a:14:01 192.168.2.1/24
ovn-nbctl lrp-add lr-bms lrp-orange 02:ac:10:0a:1e:01 192.168.3.1/24
```

Verify [optional]:

```
ovn-nbctl show lr-bms
ovn-nbctl lrp-list lr-bms
ovn-nbctl lr-route-list lr-bms
```

`lr-route-list` is empty. The three subnets are directly connected to the router, so they need no static routes.

Task 4 – Connect the switches to the router

Each switch needs a port of type `router` that is the peer of a router port:

- `lsp-set-type ... router`: this switch port connects to a logical router.
- `lsp-set-options ... router-port=lrp-xxx`: the router port on the other side.
- `lsp-set-addresses ... router`: take the MAC/IP from the peer router port. With this, the switch can answer ARP for the gateway and send gateway-bound frames to the router.

```
ovn-nbctl lsp-add red lsp-rt-red
ovn-nbctl lsp-set-type lsp-rt-red router
ovn-nbctl lsp-set-options lsp-rt-red router-port=lrp-red
ovn-nbctl lsp-set-addresses lsp-rt-red router

ovn-nbctl lsp-add green lsp-rt-green
ovn-nbctl lsp-set-type lsp-rt-green router
ovn-nbctl lsp-set-options lsp-rt-green router-port=lrp-green
ovn-nbctl lsp-set-addresses lsp-rt-green router

ovn-nbctl lsp-add orange lsp-rt-orange
ovn-nbctl lsp-set-type lsp-rt-orange router
ovn-nbctl lsp-set-options lsp-rt-orange router-port=lrp-orange
ovn-nbctl lsp-set-addresses lsp-rt-orange router
```

Verify [optional]:

```
ovn-nbctl show
ovn-nbctl lsp-list red
ovn-nbctl lsp-get-type lsp-rt-red          # router
ovn-nbctl lsp-get-options lsp-rt-red       # router-port=lrp-red
```

In `ovn-nbctl show`, each switch now has a port `lsp-rt-xxx` with `type: router` and `router-port: lrp-xxx`.

Task 5 – Create the DHCP options

OVN has a built-in DHCP server. `ovn-controller` answers DHCP requests locally, on the chassis where the VM is connected, so no DHCP server container is needed. A `DHCP_Options` row describes one subnet. The options `server_id`, `server_mac` and `lease_time` are mandatory.

`ovn-nbctl create` prints the UUID of the new row. Save it in a shell variable, because you need it here and in Task 6:

```
DHCP_RED=$(ovn-nbctl create DHCP_Options cidr=192.168.1.0/24)
DHCP_GREEN=$(ovn-nbctl create DHCP_Options cidr=192.168.2.0/24)
DHCP_ORANGE=$(ovn-nbctl create DHCP_Options cidr=192.168.3.0/24)
```

Configure all required options:

```
ovn-nbctl dhcp-options-set-options $DHCP_RED  lease_time=3600 router=192.168.1.1
server_id=192.168.1.1 server_mac=c0:ff:ee:00:00:01
ovn-nbctl dhcp-options-set-options $DHCP_GREEN  lease_time=3600 router=192.168.2.1
server_id=192.168.2.1 server_mac=c0:ff:ee:00:00:01
```

Verify:

```
echo $DHCP_RED $DHCP_GREEN
ovn-nbctl dhcp-options-list
ovn-nbctl dhcp-options-get-options $DHCP_RED
```

`dhcp-options-list` shows three UUIDs, the same as in the `echo` output. `dhcp-options-get-options` shows `lease_time=3600`, `router=192.168.1.1`, `server_id=192.168.1.1` and `server_mac=c0:ff:ee:00:00:01`.

> If you lose the shell variables (for example after opening a new shell), find the UUID again: `DHCP_RED=$(ovn-nbctl --bare --columns=_uuid find DHCP_Options cidr="192.168.1.0/24")`

Task 6 – Add the VM ports

The name of each logical switch port must match the `iface-id` that is set on the `vtapX` interface of the chassis. This is how OVN knows where each port is.

The VMs use two kinds of addresses:

- **vm1-red / vm1-green**: the MAC and IP are known (`"<mac> <ip>"`). OVN uses them for the DHCP reply. `ovn-controller` answers ARP for them and delivers frames to them by MAC.
- **vm2-*, vm3-*, www**: `unknown`. OVN does not know their addresses. Frames to unknown MAC addresses are delivered to these ports. The router learns their MAC from ARP and stores it in the `MAC_Binding` table.

6.1 red

Add the three VM ports of the red network. Only vm1-red gets DHCP:

```
ovn-nbctl lsp-add red vm1-red
ovn-nbctl lsp-set-addresses vm1-red "02:0a:0a:00:10:11 192.168.1.201"
ovn-nbctl lsp-set-dhcpv4-options vm1-red "$DHCP_RED"

ovn-nbctl lsp-add red vm2-red
ovn-nbctl lsp-set-addresses vm2-red unknown

ovn-nbctl lsp-add red vm3-red
ovn-nbctl lsp-set-addresses vm3-red unknown
```

6.2 green

Add the three VM ports of the green network. Only vm1-green gets DHCP:

```
ovn-nbctl lsp-add green vm1-green
ovn-nbctl lsp-set-addresses vm1-green "02:0a:0a:00:20:11 192.168.2.201"
ovn-nbctl lsp-set-dhcpv4-options vm1-green "$DHCP_GREEN"

ovn-nbctl lsp-add green vm2-green
ovn-nbctl lsp-set-addresses vm2-green unknown

ovn-nbctl lsp-add green vm3-green
ovn-nbctl lsp-set-addresses vm3-green unknown
```

6.3 orange

Add the port of the web server:

```
ovn-nbctl lsp-add orange www
ovn-nbctl lsp-set-addresses www unknown
```

6.4 Wait until all chassis have applied the config

```
ovn-nbctl --wait=hv sync
```

`--wait=hv` returns when every `ovn-controller` has installed the new configuration into its OVS.

Verify the Northbound DB:

```
ovn-nbctl show
ovn-nbctl lsp-get-addresses vm1-red
ovn-nbctl lsp-get-dhcpv4-options vm1-red
```

`lsp-get-addresses` shows `02:0a:0a:00:10:11 192.168.1.201`, and `lsp-get-dhcpv4-options` shows the UUID of `DHCP_RED` with `(192.168.1.0/24)`.

Task 7 – Check the Southbound DB and OVS

7.1 Port bindings

Show the chassis and the ports bound to them:

```
ovn-sbctl show
```

Expected result: each chassis has claimed the ports of its VMs:

Chassis	Ports
BMS1	`vm1-red`, `vm1-green`
BMS2	`vm2-red`, `vm2-green`
BMS3	`vm3-red`, `vm3-green`, `www`

7.2 Datapaths and logical flows [optional]

`ovn-northd` creates one datapath per logical switch and router. Each one has a tunnel key, which is also used as the Geneve VNI:

```
ovn-sbctl --columns=external_ids,tunnel_key list Datapath_Binding
ovn-sbctl lflow-list red | head -40
ovn-sbctl lflow-list lr-bms | grep -c table
```

There are four datapaths: `red`, `green`, `orange` and `lr-bms`, each with its own `tunnel_key`. The router alone has more than a hundred logical flows.

7.3 OpenFlow programmed by ovn-controller [optional]

`ovn-controller` turns the logical flows into OpenFlow rules on `br-int`:

```
ovs-ofctl dump-flows br-int | wc -l
ovs-vsctl show                                  # br-int, vtap0, vtap1 and the geneve
ports
ovs-vsctl --columns=name,ofport list Interface   # OpenFlow port numbers
```

`br-int` now has several hundred OpenFlow rules, even for this small network.

7.4 ovn-trace

`ovn-trace` simulates a packet through the logical pipeline and shows each logical table it passes. Trace a packet from vm1-red to www:

```
ovn-trace --summary red 'inport == "vm1-red" && eth.src == 02:0a:0a:00:10:11 && eth.dst
== 02:ac:10:0a:00:01 && ip4.src == 192.168.1.201 && ip4.dst == 192.168.3.200 && ip.ttl
== 64'
```

Expected result: the packet enters switch `red`, goes to `lr-bms` through `lsp-rt-red` → `lrp-red`, is routed (TTL decremented) out of `lrp-orange` into switch `orange`, and is output to port `www`. If OVN does not know the MAC of `192.168.3.200` yet, the trace ends with an ARP request from the router.

Exit the remote shell using:

```
exit
```

Task 8 – Test from the VMs

In separate terminal keep a capture of the Geneve traffic running on BMS1 during this task:

```
# Separate terminal
docker exec -it bms1 tcpdump -nei eth1 udp port 6081
```

8.1 OVN native DHCP

Request a DHCP lease on vm1-red and vm1-green:

```
docker exec -it vm1-red udhcpc -q -i eth1
docker exec -it vm1-green udhcpc -q -i eth1
```

Verify:

```
docker exec -it vm1-red ip addr show eth1        # 192.168.1.201/24
docker exec -it vm1-red ip route                 # default via 192.168.1.1
docker exec -it vm1-green ip addr show eth1      # 192.168.2.201/24
docker exec -it vm1-green ip route               # default via 192.168.2.1
```

The DHCP reply comes from `server_mac` `c0:ff:ee:00:00:01` and is generated by `ovn-controller` on BMS1. The Geneve capture shows **no** DHCP packets, because nothing leaves BMS1.

8.2 Gateway and L2 (same subnet, other chassis)

Ping the router port, and the VMs in the same subnet on BMS2 and BMS3. All pings must work (`0% packet loss`):

```
docker exec -it vm1-red ping -c 3 192.168.1.1        # logical router port
docker exec -it vm1-red ping -c 3 192.168.1.202      # vm2-red on BMS2
docker exec -it vm1-red ping -c 3 192.168.1.203      # vm3-red on BMS3
docker exec -it vm1-green ping -c 3 192.168.2.202    # vm2-green on BMS2
docker exec -it vm1-green ping -c 3 192.168.2.203    # vm3-green on BMS3
```

In the capture, the traffic to vm2-red / vm3-red goes in Geneve packets to `10.1.0.102` / `10.1.0.103`.

8.3 Routing between subnets

Ping from red to green, and fetch the web page from the orange network:

```
docker exec -it vm1-red ping -c 3 192.168.2.202      # red -> green
docker exec -it vm1-red curl http://192.168.3.200    # red -> orange (www)
docker exec -it vm1-green curl http://192.168.3.200  # green -> orange (www)
```

`curl` must return the page `This is a NGINX at: www-orange`.

The logical router is **distributed**. The packet is routed on BMS1, the chassis where it enters, and goes in a Geneve packet directly to BMS3. In the capture, the inner frame already has the router MAC `02:ac:10:0a:1e:01` as source and a decremented TTL.

8.4 What OVN learned [optional]

Check MACs that OVN learned due to previous steps:

```
docker exec -it bms1 ovn-sbctl --columns=logical_port,ip,mac list MAC_Binding
```

`MAC_Binding` holds the MACs that the router learned through ARP when it routed traffic: `192.168.3.200` (www) on `lrp-orange`, `192.168.2.202` (vm2-green) on `lrp-green`, and the return path to vm1-red and vm1-green. Same-subnet traffic (Task 8.2) does not go through the router, so vm2-red and vm3-* are not listed.

OVN vs. EVPN (LAB 2 vs LAB 3)

This table compares two approches:

Item	LAB 2 – EVPN/VXLAN (FRR + Linux)	LAB 3 – OVN/OVS
Control plane	Distributed: BGP EVPN between leaves, with an RR	Central: NB/SB OVSDB on BMS1, `ovn-northd`
Where you configure	Every leaf (VRF, bridges, VXLAN, FRR)	Only BMS1 (`ovn-nbctl`)
Encapsulation	VXLAN (UDP 4789), VNI per network	Geneve (UDP 6081), tunnel key per datapath
L2 network	Linux bridge (or VLAN) + L2 VNI	Logical switch
Gateway / routing	Anycast SVI on every leaf + L3 VNI (symmetric IRB)	Distributed logical router (routed on the ingress chassis)
MAC/IP distribution	Learned locally, advertised as type-2/ type-5 routes	Set in NB (`lsp-set-addresses`), or learned by ARP (`MAC_Binding`)
BUM traffic	Head-end replication, flood list from type-3 routes	Replication to the chassis that have ports on the switch
DHCP	Separate DHCP server containers, reached over L2 VNI	Native, answered locally by `ovn-controller`
Data plane	Linux kernel (bridge, vxlan, VRF)	OVS kernel datapath, programmed with OpenFlow

Troubleshooting checklist

Symptom	Check (on BMS1)
A chassis is missing in `ovn-sbctl show`	On that BMS: `ovs-vsctl get Open_vSwitch . external_ids`, `sv status /services/ovn-controller`
A port is not bound (`chassis` empty)	The LSP name must match `iface-id` on the BMS: `ovs-vsctl --columns=name,external_ids list Interface`
No DHCP lease	`ovn-nbctl lsp-get-dhcpv4-options vm1-red`, `ovn-nbctl lsp-get-addresses vm1-red` (the MAC must match the VM's `eth1`)
Same-subnet ping to BMS2/BMS3 fails	`ovs-vsctl show` (geneve ports), `ping 10.1.0.102`
Routing fails	`ovn-nbctl show lr-bms`, the `lsp-rt-*` options, then `ovn-trace`
`ovn-nbctl` hangs	`sv status /services/ovsdb-server-ovn /services/ovn-northd`

Logs: `/var/log/ovn/ovn-controller.log`, `/var/log/ovn/ovn-northd.log`, `/var/log/openvswitch/ovs-vswitchd.log`.

Cleanup

Destroy the lab:

```
containerlab destroy -t lab3.clab.yml
```

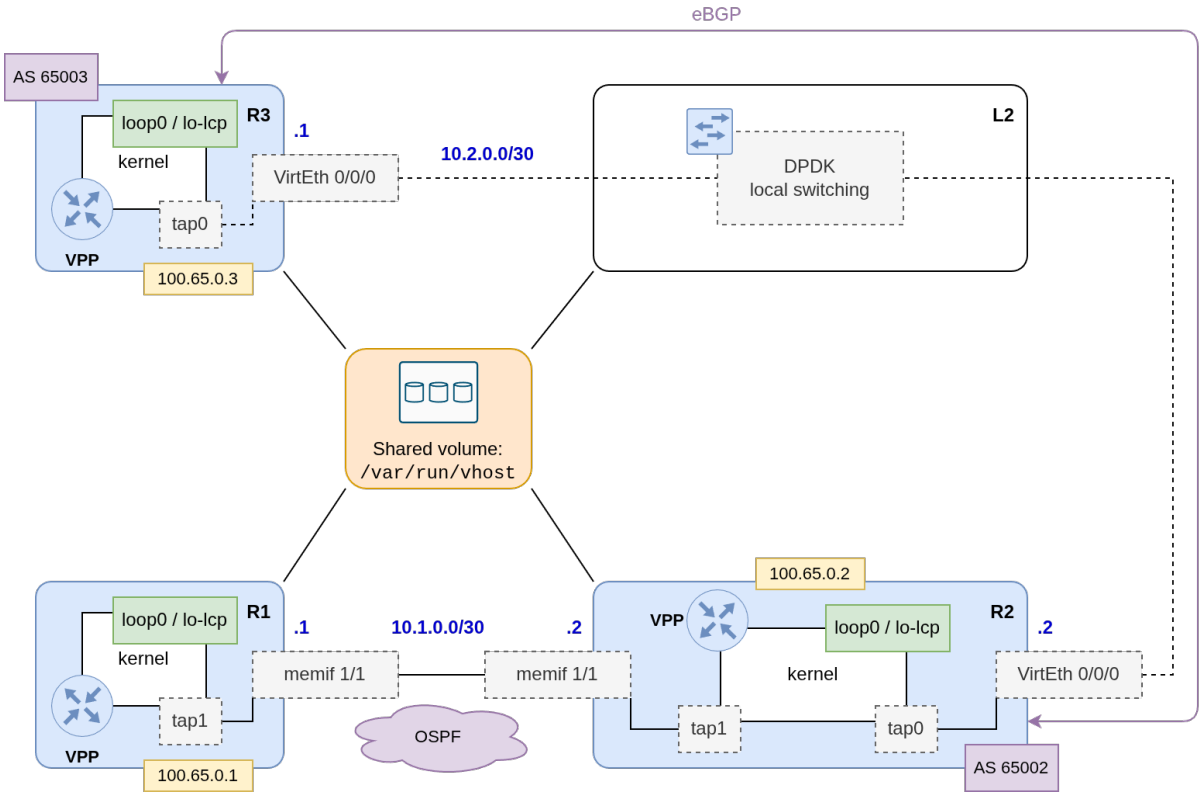
LAB 4 – DPDK & VPP

Goals

In this lab you connect three routers. Each router has **VPP** as the data plane and **FRR** in Linux as the control plane. You build the links between them by hand. After this lab you should be able to:

- create a **memif** link between two VPP instances (shared memory, master / slave),
- create a **vhost-user** interface in VPP (backend) and connect a DPDK **virtio_user** device (frontend) to it,
- run **DPDK testpmd** as a transparent "cable" between two virtio_user ports,
- use the VPP **linux-cp** plugin to mirror VPP interfaces into Linux as TAP interfaces, and see how IP addresses, routes and ARP entries are **synced** from the kernel (FRR) into the VPP FIB,
- check the result in VPP (`vppctl`), in Linux (`ip`) and in FRR (`vtysh`).

Topology



Addressing

Node	VPP interface	Linux TAP (linux-cp)	Address	Peer
R1	`loop0`	`lo-lcp`	100.65.0.1/32	–
	`memif1/1` (slave)	`tap1`	10.1.0.1/30	R2 `memif1/1`
R2	`loop0`	`lo-lcp`	100.65.0.2/32	–

Node	VPP interface	Linux TAP (linux-cp)	Address	Peer
	`memif1/1` (master)	`tap1`	10.1.0.2/30	R1 `memif1/1`
	`VirtualEthernet0/0/0`	`tap0`	10.2.0.2/30	L2 `net_virtio_user0`
R3	`loop0`	`lo-lcp`	100.65.0.3/32	–
	`VirtualEthernet0/0/0`	`tap0`	10.2.0.1/30	L2 `net_virtio_user1`

Management addresses (network `172.16.0.0/24`): R1 `.11`, R2 `.12`, R3 `.13`, L2 `.40`.

Control plane (FRR)

- **R1 ↔ R2**: OSPF area 0 on `tap1` (point-to-point). The loopbacks (`lo-lcp`) are passive.
- **R2 ↔ R3**: eBGP, AS 65002 ↔ AS 65003, over `tap0`. Both routers advertise their loopback.
- **R2** redistributes BGP into OSPF and OSPF into BGP. R1 learns `100.65.0.3/32`, and R3 learns `100.65.0.1/32`.

How the pieces fit together

- **VPP** owns the data-plane interfaces (`memif1/1`, `VirtualEthernet0/0/0`, `loop0`) and forwards packets in user space.
- **linux-cp** creates a TAP pair for each VPP interface:
  - one end is in VPP (`tap4096`, `tap4097`, …),
  - the other end is in Linux (`lo-lcp`, `tap1`, `tap0`).

  Control-plane packets (OSPF, BGP, ARP, ping to the router itself) are punted from VPP to Linux through this pair. Packets that Linux sends go back out through VPP.
- **lcp-sync** (in `startup.conf`): IP addresses, link state, routes and ARP neighbors that FRR/Linux set on the TAP interfaces are copied into VPP via netlink. FRR never talks to VPP directly; it only programs the Linux kernel.
- **memif**: a shared-memory packet interface between two processes. The **master** creates the socket and the shared memory. The **slave** connects to it.
- **vhost-user / virtio_user**: VPP is the vhost-user **backend** in `server` mode, so it creates the socket. testpmd uses DPDK **virtio_user** devices as the **frontend** (client). They connect to the sockets and share their memory with VPP.
- **testpmd** in `io` forwarding mode sends everything from port 0 to port 1 and back, without changes. It works like a cable between R2 and R3.

What is already configured when the lab starts:

- VPP on R1, R2 and R3, with the `linux_cp`, `linux_nl` and `vhost` plugins enabled and DPDK disabled.
- `loop0` in VPP, mirrored to Linux as `lo-lcp` (from `/etc/vpp/vpp-exec.conf`).
- FRR. It starts only after VPP has finished its startup config. Its config already has sections for `tap1` / `tap0`, but these interfaces do not exist yet.
- The shared volume `/var/run/vhost` in all four containers, used for the memif and vhost-user sockets.

Your job is to create the memif and vhost-user interfaces, mirror them into Linux with linux-cp, and start testpmd.

Deploying the lab and accessing nodes

Go to the lab directory, deploy the lab and list its nodes:

```
cd ~/labs/lab4
containerlab deploy -t lab4.clab.yml
containerlab inspect -t lab4.clab.yml
```

`containerlab inspect` lists four nodes (`l2`, `r1`, `r2`, `r3`) in state `running`, with their management IP addresses.

> **Tip:** testpmd (Task 4) runs in the foreground and keeps its own prompt. Use a separate terminal for it and leave it running.

> **Note:** Do **not** run `./setup.sh` or `/tmp/setup.sh`. Those scripts are the reference solution. In this lab you type every command yourself.

Task 0 – Check the starting point

Run on each router `r1` [optional: `r2`, `r3`]:

```
docker exec -it r1 vppctl show version
docker exec -it r1 vppctl show interface
docker exec -it r1 vppctl show interface addr
docker exec -it r1 vppctl show lcp
docker exec -it r1 ip -br addr
```

Expected result (R1):

```
loop0 (up):
  L3 100.65.0.1/32
tap4096 (up):

lcp lcp-sync on
...
itf-pair: [0] loop0 tap4096 lo-lcp 3 type tap

lo-lcp           UP             100.65.0.1/32 ...
```

- VPP has only `local0`, `loop0` and `tap4096`, the VPP end of the linux-cp pair for `loop0`.
- In Linux, `lo-lcp` has the loopback address, set by FRR.
- `100.65.0.1/32` shows up on `loop0` in VPP because of lcp-sync.

Check the shared socket directory:

```
docker exec -it l2 ls -la /var/run/vhost/ # Should not return any error
```

> Socket files from an earlier run can stay in `/var/run/vhost` (it is a docker volume). This is fine: the master/server side creates them again.

Task 1 – R2: memif master towards R1 and vhost-user towards L2

Open a shell on R2:

```
docker exec -it r2 bash
```

1.1 Create the memif socket and the memif interface (master)

Create the memif socket file, a memif interface in master mode on it, and bring the interface up:

```
vppctl create memif socket id 1 filename /var/run/vhost/r1-r2-memif.sock
vppctl create interface memif id 1 socket-id 1 master
vppctl set int state memif1/1 up
```

The interface name `memif1/1` is `memif<socket-id>/<id>`.

Verify:

```
vppctl show memif
```

Expected result: socket `1` is a `listener` (R2 is master). The interface is `admin-up` but not `connected` yet, with `num-s2m-rings 0`, because R1 does not exist on the other side yet.

```
sockets
  id  listener    filename
  1   yes (1)     /var/run/vhost/r1-r2-memif.sock

interface memif1/1
  socket-id 1 id 1 mode ethernet
  flags admin-up
```

1.2 Mirror memif1/1 into Linux as tap1

Create the linux-cp pair for `memif1/1`. The Linux end is called `tap1`:

```
vppctl lcp create memif1/1 host-if tap1
```

Verify:

```
vppctl show lcp | grep itf-pair
ip -br addr show tap1
vppctl show int addr
```

Expected result:

```
itf-pair: [1] memif1/1 tap4097 tap1 4 type tap

tap1              DOWN           10.1.0.2/30 ...  <NO-CARRIER,...,UP>

memif1/1 (up):
  L3 10.1.0.2/30
```

- FRR sees the new `tap1` and sets `10.1.0.2/30` on it right away.
- lcp-sync copies the address to `memif1/1` in VPP.
- `tap1` has `NO-CARRIER`, because linux-cp mirrors the link state of `memif1/1`, which is not connected yet.

1.3 Create the vhost-user interface (server) and mirror it as tap0

Create the vhost-user interface in server mode, bring it up and mirror it into Linux as `tap0`:

```
vppctl create vhost-user socket /var/run/vhost/r2-l2.sock server
vppctl set int state VirtualEthernet0/0/0 up
vppctl lcp create VirtualEthernet0/0/0 host-if tap0
```

`create vhost-user` prints the name of the new interface, `VirtualEthernet0/0/0`.

Verify:

```
vppctl show vhost-user
vppctl show int addr
ip -br addr show tap0               # Down status is expected
```

Expected result: `socket filename /var/run/vhost/r2-l2.sock type server`. `Memory regions (total 0)` and `features (0x0)`, because no frontend is connected yet. `VirtualEthernet0/0/0` has `10.2.0.2/30`.

1.4 Reload FRR

Reload FRR on R2:

```
sv reload /services/frr
```

This makes FRR apply the whole config again (OSPF on `tap1`, BGP over `tap0`) now that the interfaces exist.

Exit the remote shell using:

```
exit
```

Task 2 – R1: memif slave towards R2

Open a shell on R1:

```
docker exec -it r1 bash
```

Create memif interface and add it to `linux-cp` and reload FRR:

```
vppctl create memif socket id 1 filename /var/run/vhost/r1-r2-memif.sock
vppctl create interface memif id 1 socket-id 1 slave
vppctl set int state memif1/1 up
vppctl lcp create memif1/1 host-if tap1
sv reload /services/frr
```

2.1 Verify the memif connection

Show the memif sockets and interfaces on R1:

```
vppctl show memif
```

Expected result: socket `1` is **not** a listener (R1 is slave). The interface is `connected`, and the rings and shared-memory regions are set up:

```
interface memif1/1
  remote-name "VPP 26.06-release"
  remote-interface "memif1/1"
  socket-id 1 id 1 mode ethernet
  flags admin-up slave connected zero-copy
  num-s2m-rings 1 num-m2s-rings 1 buffer-size 2048 num-regions 2
```

Exit the remote shell using:

```
exit
```

On R2, `tap1` now has carrier:

```
docker exec -it r2 ip -br link show tap1      # UP ...
<BROADCAST,MULTICAST,UP,LOWER_UP>
docker exec -it r2 vppctl show memif memif1/1 # connected
```

2.2 Verify the link and OSPF

Open again a shell on R1:

```
docker exec -it r1 bash
```

Check the address in VPP, ping R2 over the memif link and check the OSPF neighbor:

```
vppctl show int addr                # memif1/1: L3 10.1.0.1/30
ping -c 2 10.1.0.2
vtysh -c "show ip ospf neighbor"    # 100.65.0.2  Full/-  tap1
```

The ping must work (`0% packet loss`). OSPF needs up to about 40 seconds to reach `Full`.

2.3 Verify the kernel → VPP sync (linux-cp)

Compare the Linux tables with the VPP tables:

```
ip route                              # 100.65.0.2 via 10.1.0.2 dev tap1 proto ospf
vppctl show ip fib 100.65.0.2/32
ip neigh show dev tap1                # 10.1.0.2 lladdr 02:fe:... REACHABLE/STALE/DELAY
vppctl show ip neighbors              # 10.1.0.2  D  02:fe:...  memif1/1
```

Expected result in the VPP FIB: the route has source `lcp-rt-dynamic`, and its path points to `memif1/1`, the VPP interface, not `tap1`:

```
100.65.0.2/32 fib:0 index:19 locks:2
  lcp-rt-dynamic refs:1 src-flags:added,contributing,active,
      ...
        10.1.0.2 memif1/1
      [@0]: ipv4 via 10.1.0.2 memif1/1: mtu:9000 ...
```

Exit the remote shell using:

```
exit
```

Task 3 – R3: vhost-user towards L2

Open a shell on R3:

```
docker exec -it r3 bash
```

Create vhost-user interface (server) and mirror it as tap0 interface and reload FRR at the end:

```
vppctl create vhost-user socket /var/run/vhost/r3-l2.sock server
vppctl set int state VirtualEthernet0/0/0 up
vppctl lcp create VirtualEthernet0/0/0 host-if tap0
sv reload /services/frr
```

Verify:

```
vppctl show int addr                  # VirtualEthernet0/0/0: L3 10.2.0.1/30
vppctl show vhost-user                # type server, Memory regions (total 0)
ip -br addr show tap0                 # DOWN, 10.2.0.1/30
```

`tap0` stays `DOWN` and the BGP session to R2 cannot come up yet: nothing connects the two vhost-user sockets.

Task 4 – L2: testpmd as a cable between R2 and R3

Open a **new terminal** and a shell on L2:

```
docker exec -it l2 bash
```

Start testpmd:

```
dpdk-testpmd --lcores='(0-1)@0' \
        --no-pci \
        --in-memory \
        --single-file-segments \
        --vdev 'net_virtio_user0,path=/var/run/vhost/r2-l2.sock,queues=1' \
        --vdev 'net_virtio_user1,path=/var/run/vhost/r3-l2.sock,queues=1' \
        -- -i --forward-mode=io --auto-start --total-num-mbufs=8192 --nb-cores=1
```

Option	Meaning
`--lcores='(0-1)@0'`	EAL: lcores 0 and 1 (main + 1 forwarding) both run on CPU 0
`--no-pci`	EAL: do not scan or probe PCI devices
`--in-memory`	EAL: no hugepage files or runtime files on disk
`--single-file-segments`	EAL: fewer, larger memory files (needed to share memory with vhost-user)
`--vdev 'net_virtio_user0,path=...'`	virtual device: virtio_user frontend connected to R2's vhost-user socket → port 0
`--vdev 'net_virtio_user1,path=...'`	the same, connected to R3's socket → port 1
`--`	end of EAL options, start of testpmd options
`-i`	interactive mode (`testpmd>` prompt)
`--forward-mode=io`	forward port 0 ↔ port 1 without touching the packets
`--auto-start`	start forwarding right away
`--total-num-mbufs=8192`	size of the packet buffer pool
`--nb-cores=1`	one forwarding core

At the `testpmd>` prompt, check:

```
show port info all
show config fwd
show port stats all
```

Expected result: two ports, `net_virtio_user0` (port 0) and `net_virtio_user1` (port 1), both with link up. Forwarding mode is `io`, with the stream `port 0 ↔ port 1`. `show port stats all` shows RX/TX counters going up once the routers exchange traffic. Run it again later to see the counters grow.

> Leave testpmd running. If you `quit`, the R2 ↔ R3 link goes down.

4.1 Verify the vhost-user side in VPP

In your first terminal exit docker shell if any:

```
exit
```

Then execute:

```
docker exec -it r2 vppctl show vhost-user
docker exec -it r2 ip -br link show tap0
```

Then execute [optional]:

```
docker exec -it r2 vppctl show hardware-interfaces VirtualEthernet0/0/0
docker exec -it r3 vppctl show vhost-user
docker exec -it r3 ip -br link show tap0
```

Expected result: the frontend has connected and negotiated virtio features (`features (0x150008000)`, `protocol features (0x1)`). There is `Memory regions (total 1)`, which is testpmd's memory mapped into VPP. The link is `up`, and `tap0` is now `UP ... LOWER_UP`.

Task 5 – Check end-to-end routing

5.1 BGP R2 ↔ R3 [optional]

Check BGP status:

```
docker exec -it r3 vtysh -c "show bgp summary"
docker exec -it r3 vtysh -c "show ip route bgp"
```

Expected result:

```
Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/
PfxRcd   PfxSnt
10.2.0.2        4      65002         6         6        3    0    0 00:00:19
2        3

B>* 100.65.0.1/32 [20/20] via 10.2.0.2, tap0, weight 1, 00:00:18
B>* 100.65.0.2/32 [20/0] via 10.2.0.2, tap0, weight 1, 00:00:18
```

5.2 OSPF on R1 (BGP routes redistributed by R2) [optional]

Check OSPF status:

```
docker exec -it r1 vtysh -c "show ip route ospf"
```

Expected result:

```
O>* 10.2.0.0/30 [110/20] via 10.1.0.2, tap1, weight 1
O>* 100.65.0.2/32 [110/20] via 10.1.0.2, tap1, weight 1
O>* 100.65.0.3/32 [110/20] via 10.1.0.2, tap1, weight 1
```

5.3 The same routes in VPP

Check that the routes learned by FRR are also in the VPP FIB:

```
docker exec -it r1 vppctl show ip fib 100.65.0.3/32     # lcp-rt-dynamic, via 10.1.0.2
memif1/1
```

[optional]:

```
docker exec -it r3 vppctl show ip fib 100.65.0.1/32     # lcp-rt-dynamic, via 10.2.0.2
VirtualEthernet0/0/0
docker exec -it r2 vppctl show ip fib                   # the full FIB of the transit
router
```

5.4 Ping loopback to loopback

Ping between the loopbacks of R1 and R3, in both directions:

```
docker exec -it r1 ping -c 3 -I 100.65.0.1 100.65.0.3
docker exec -it r3 ping -c 3 -I 100.65.0.3 100.65.0.1
```

Expected result: replies with `ttl=63`, so one hop (R2) routed the packet.

Task 6 – Follow a packet through the VPP graph on R2 [optional]

R2 forwards R1 ↔ R3 traffic entirely in VPP. The Linux kernel on R2 never sees these packets. Check this with a VPP packet trace:

```
docker exec -it r2 vppctl clear trace
docker exec -it r2 vppctl trace add memif-input 5
docker exec -it r1 ping -c 1 -I 100.65.0.1 100.65.0.3
docker exec -it r2 vppctl show trace
```

Expected path of the echo request through the VPP graph nodes:

```
memif-input
ethernet-input
ip4-input
ip4-lookup
ip4-rewrite
VirtualEthernet0/0/0-output
VirtualEthernet0/0/0-tx
```

The packet comes in on memif, is routed with the FIB entry that linux-cp synced from FRR, and leaves on the vhost-user interface. It does not pass through `tap*` or `linux-cp-*` nodes.

To trace the opposite direction (from R3), use the vhost-user input node:

```
docker exec -it r2 vppctl clear trace
docker exec -it r2 vppctl trace add vhost-user-input 5
docker exec -it r3 ping -c 1 -I 100.65.0.3 100.65.0.1
docker exec -it r2 vppctl show trace
```

Other useful VPP commands:

```
docker exec -it r2 vppctl show runtime           # per-node counters (memif-input,
vhost-user-input, *-tx)
docker exec -it r2 vppctl show errors            # drop / error counters
docker exec -it r2 vppctl show interface         # rx/tx counters per interface
```

In the testpmd terminal, `show port stats all` now shows the forwarded packets on both ports.

Troubleshooting checklist

Symptom	Check
`memif1/1` not `connected`	Both sides use the same socket file. R2 is `master` and R1 is `slave`. `vppctl show memif` on both
`tap1` / `tap0` has no IP address	`vtysh -c "show running-config"`, then `sv reload /services/frr`
IP on the TAP, but not in VPP	`vppctl show lcp` (`lcp-sync on`, the `itf-pair` exists)
testpmd fails to start	R2 and R3 must create the vhost-user sockets (server) **before** testpmd starts. Check the hugepages (`grep -i huge /proc/meminfo`)
`Memory regions (total 0)` on VPP	testpmd is not running or not connected: check the `testpmd>` terminal
BGP R2 ↔ R3 stays `Active` / `Connect`	testpmd: `show port info all` (link up), `show port stats all`; VPP: `show vhost-user`
Route in FRR, but not in VPP	`vppctl show ip fib <prefix>` (source `lcp-rt-dynamic`), `ip route`

Cleanup

Stop testpmd (`quit` at the `testpmd>` prompt), then:

```
containerlab destroy -t lab4.clab.yml
```

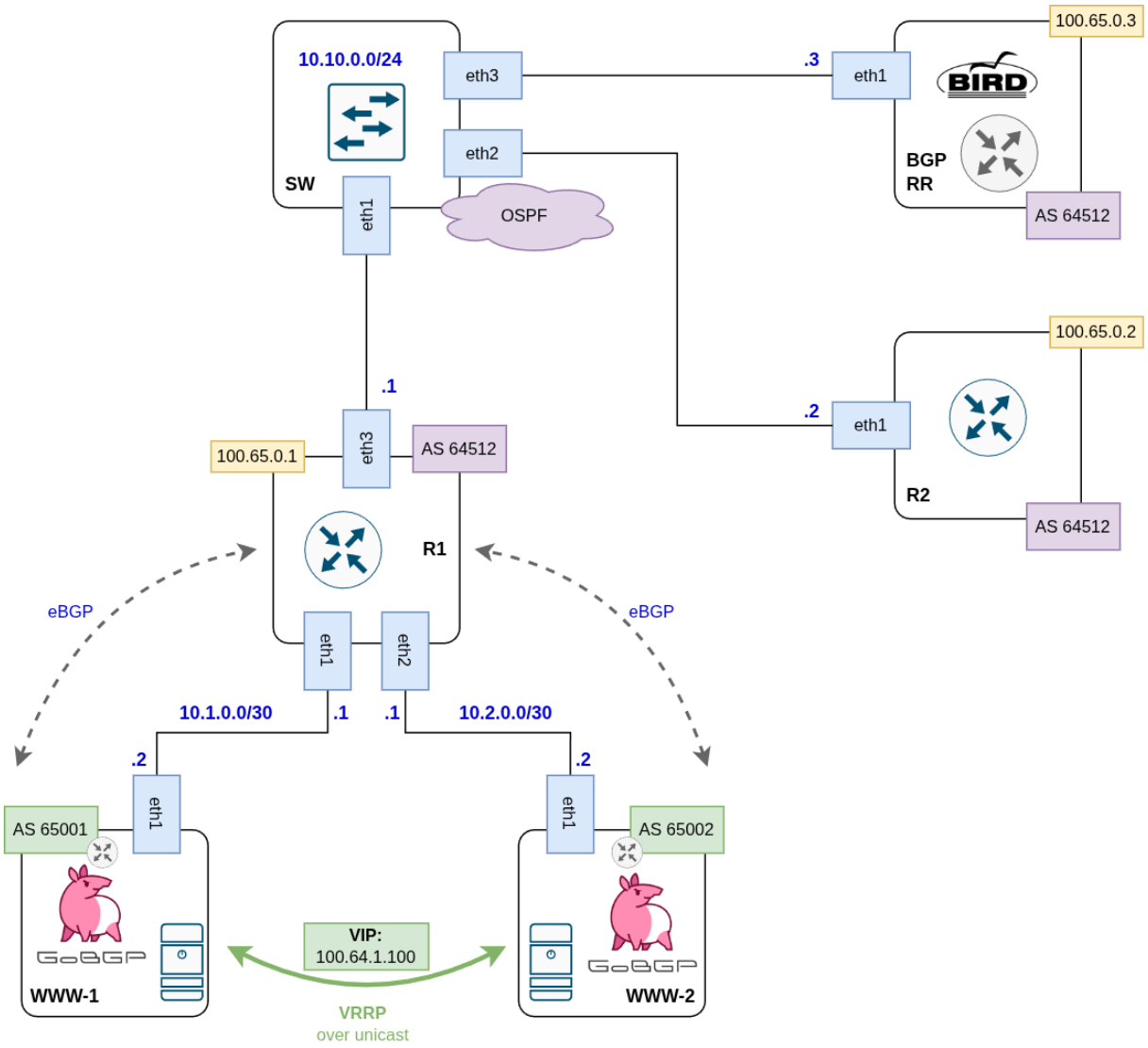
LAB 5 – VRRP + GoBGP & Bird

Goals

This lab is not about the Linux data plane. It is about tools that connect a Linux server to the network around it. Two web servers share one service IP (VIP). keepalived decides which server is active, and GoBGP advertises the VIP only from the active server. A BIRD route reflector rewrites the route attributes based on BGP communities. After this lab you should be able to:

- use **GoBGP** as a BGP speaker on a server and advertise a prefix with **large communities** from the CLI,
- use **keepalived** (VRRP + health check) to start actions (advertise / withdraw) when a server becomes master or backup,
- configure **BIRD** as an iBGP **route reflector**, and read an advanced BIRD export filter that uses communities as "action" signals,
- check the result in GoBGP, BIRD (`birdc`) and FRR (`vtysh`), and test a failover.

Topology



Addressing

Node	Interface	Address	Software
www-1	eth1	10.1.0.2/30 (+ VIP when master)	nginx, keepalived, GoBGP (AS 65001)
www-2	eth1	10.2.0.2/30 (+ VIP when master)	nginx, keepalived, GoBGP (AS 65002)
R1	eth1	10.1.0.1/30	FRR (AS 64512)
	eth2	10.2.0.1/30	
	eth3	10.10.0.1/24	
	lo	100.65.0.1/32	
R2	eth1	10.10.0.2/24	FRR (AS 64512)
	lo	100.65.0.2/32	
bgp-rr	eth1	10.10.0.3/24	BIRD 3 (AS 64512, route reflector)

Node	Interface	Address	Software
	lo	100.65.0.3/32	

Service VIP: **100.64.1.100/32**.

Management addresses (network `172.16.0.0/24`): R1 `.11`, R2 `.12`, bgp-rr `.13`, www-1 `.21`, www-2 `.22`, l2-switch `.40`.

Control plane

- **OSPF area 0** between R1, R2 and bgp-rr (BIRD) on `10.10.0.0/24`. It carries the loopbacks and the server subnets. R1's `eth1` / `eth2` are passive in OSPF.
- **eBGP** R1 (AS 64512) ↔ www-1 (AS 65001) and R1 ↔ www-2 (AS 65002). On R1 this is the peer group `gobgp` with `remote-as external` and `passive`: R1 never opens the BGP connection itself, and only accepts connections on TCP 179. GoBGP on the server opens the session after `gobgp neighbor add`.
- **iBGP AS 64512**: R1 and R2 are route-reflector clients of BIRD (`100.65.0.3`). R1 sends all paths to the RR (`addpath-tx-all-paths`) and all communities (`send-community all`).
- **VRRP (keepalived)** between www-1 and www-2, in **unicast** mode (`unicast_peer`). The servers are on different subnets, so multicast VRRP would not reach the other server.

What is already configured when the lab starts:

- FRR on R1 and R2 (OSPF + BGP). The BGP sessions to the servers and to the RR are not up yet.
- GoBGP on www-1 and www-2: only the global part (`AS` and `router-id` in `/etc/gobgp/gobgp.toml`), with no neighbors.
- keepalived on www-1 and www-2, with the VRRP instance, the health check and the VIP. It has no notify actions yet. The config includes `/etc/keepalived.d/*.conf` inside the `vrrp_instance`, and this directory is empty.
- BIRD on bgp-rr with OSPF, the kernel protocol and the export filter. It has no BGP sessions yet. The config includes `/etc/bird.d/*.conf`, and this directory is empty.

Your job is to:

1. add the BGP neighbor to GoBGP on both servers,
2. make keepalived advertise / withdraw the VIP through GoBGP,
3. add the BGP route-reflector config to BIRD.

Deploying the lab and accessing nodes

Go to the lab directory, deploy the lab and list its nodes:

```
cd ~/labs/lab5
containerlab deploy -t lab5.clab.yml
containerlab inspect -t lab5.clab.yml
```

`containerlab inspect` lists six nodes in state `running`, with their management IP addresses.

> **Note:** Do **not** run `./setup.sh` or `/tmp/setup.sh`. Those scripts are the reference solution. In this lab you type every command yourself.

Task 0 – Check the starting point

0.1 OSPF [optional]

Check OSPF on R1 and on the route reflector (BIRD), and the routes that BIRD installed in the kernel:

```
docker exec -it r1 vtysh -c "show ip ospf neighbor"
docker exec -it bgp-rr birdc show protocols
docker exec -it bgp-rr birdc show ospf neighbors
docker exec -it bgp-rr ip route
```

Expected result: R1 has two OSPF neighbors on `eth3`, `100.65.0.2` and `100.65.0.3`, in state `Full`. BIRD shows `ospf_main ... Running`. The kernel on bgp-rr has `proto bird` routes to `10.1.0.0/30`, `10.2.0.0/30` and the loopbacks.

0.2 VRRP – the VIP is on www-1

Check which server has the VIP on `eth1`:

```
docker exec -it www-1 ip -br addr show eth1      # 10.1.0.2/30 100.64.1.100/32
docker exec -it www-2 ip -br addr show eth1      # 10.2.0.2/30 only
```

keepalived writes its log to the container output. Read it from the host with `docker logs`:

```
docker logs www-1 2>&1 | grep Keepalived_vrrp
docker logs www-2 2>&1 | grep Keepalived_vrrp
```

Expected result on www-1:

```
Keepalived_vrrp[66]: (lab5) Entering BACKUP STATE (init)
Keepalived_vrrp[66]: VRRP_Script(check_http) succeeded
Keepalived_vrrp[66]: (lab5) Changing effective priority from 110 to 130
Keepalived_vrrp[66]: (lab5) Entering MASTER STATE
```

> SECURITY VIOLATIONS warning are expected. The `keepalived` configuration was kept to bare minimum (for lab simplicity)

On www-2 the effective priority changes from `100` to `120`, and the instance stays in `BACKUP STATE`.

> Use `docker logs -f www-1` to follow the log live. This is useful during the failover test in Task 5.

How the priorities work:

- Base priority: www-1 `110`, www-2 `100`.

- The health check `check_http` runs `curl http://127.0.0.1/` every 2 s, with `weight 20`. While the check succeeds, keepalived adds 20 to the priority: www-1 `130`, www-2 `120`.
- If nginx on www-1 fails (`fall 2` = two failed checks), its priority drops to `110`. That is lower than `120`, so www-2 becomes master.
- When the check passes again (`rise 2`), www-1 is back at `130` and takes the VIP back (preemption).

0.3 BGP is not up yet [optional]

Check GoBGP on www-1, the BGP sessions on R1 and the BIRD include directory:

```
docker exec -it www-1 gobgp global
docker exec -it www-1 gobgp neighbor              # empty
docker exec -it r1 vtysh -c "show bgp summary"    # 10.1.0.2, 10.2.0.2, 100.65.0.3:
Active or Connect
docker exec -it bgp-rr ls /etc/bird.d/            # empty
```

`gobgp global` shows `AS: 65001` and `Router-ID: 10.1.0.2`. On R1 no session is established yet (`Up/Down` is `never`).

Task 1 – www-1: GoBGP neighbor and keepalived actions

Open a shell on www-1:

```
docker exec -it www-1 bash
```

1.1 Add the BGP neighbor (R1) to GoBGP

Add R1 as a BGP neighbor of GoBGP:

```
gobgp neighbor add 10.1.0.1 as 64512
```

Verify:

```
gobgp neighbor
gobgp neighbor 10.1.0.1
```

Expected result:

```
Peer        AS  Up/Down State       |#Received  Accepted
10.1.0.1 64512 00:00:04 Establ      |        0         0

  BGP state = SESSION_STATE_ESTABLISHED, up for 00:00:04
  Local address is 10.1.0.2, local ASN: 65001
```

> This neighbor is added at run time through the GoBGP API. It is not saved to `gobgp.toml`, so it is lost if GoBGP restarts.

1.2 Add the notify actions to keepalived

keepalived runs a `notify_master` command when the instance becomes master, and a `notify_backup` command when it becomes backup. Use them to advertise or withdraw the VIP in GoBGP. The route gets two **large communities**. BIRD reads them as instructions (see Task 3.1):

- `64512:1:777`: set MED to 777
- `64512:2:3`: prepend the AS path 3 times

```
cat  > /etc/keepalived.d/notify.conf <<EOF

notify_master "/usr/local/bin/gobgp global rib add -a ipv4 100.64.1.100/32 large-
community 64512:1:777,64512:2:3"
notify_backup "/usr/local/bin/gobgp global rib del 100.64.1.100/32 -a ipv4"
EOF
```

Restart keepalived so it reads the new file, and runs `notify_master` when it becomes master again:

```
sv restart /services/keepalived
```

Verify (wait a few seconds):

```
ip -br addr show eth1           # the VIP appeared on eth1
gobgp global rib
```

Expected result:

```
   Network              Next Hop             AS_PATH              Age        Attrs
*> 100.64.1.100/32      0.0.0.0                                   00:00:02
[{Origin: ?} {LargeCommunity: [ 64512:1:777, 64512:2:3]}]
```

Exit the remote shell using:

```
exit
```

1.3 Verify on R1

Check the BGP session to www-1 and the VIP route on R1:

```
docker exec -it r1 vtysh -c "show bgp summary"
docker exec -it r1 vtysh -c "show bgp ipv4 unicast 100.64.1.100/32"
```

Expected result: the session to `10.1.0.2` is up, with `PfxRcd 1`. For prefix:

```
  65001
    10.1.0.2 from 10.1.0.2 (10.1.0.2)
      Origin incomplete, valid, external, best (First path received)
      Large Community: 64512:1:777 64512:2:3
```

Task 2 – www-2: GoBGP neighbor and keepalived actions

Open a shell on www-2:

```
docker exec -it www-2 bash
```

www-2 uses the same commands, but its own neighbor address and different community values:

- `64512:1:888`: MED 888
- `64512:2:1`: prepend once

Add neighbor, keepalived scripts and restart keepalived:

```
gobgp neighbor add 10.2.0.1 as 64512

cat  > /etc/keepalived.d/notify.conf <<EOF

notify_master "/usr/local/bin/gobgp global rib add -a ipv4 100.64.1.100/32 large-
community 64512:1:888,64512:2:1"
notify_backup "/usr/local/bin/gobgp global rib del 100.64.1.100/32 -a ipv4"
EOF

sv restart /services/keepalived
```

Verify:

```
gobgp neighbor                  # 10.2.0.1 64512 ... Establ
ip -br addr show eth1           # no VIP: www-2 is BACKUP
gobgp global rib
```

Expected result: www-2 is backup, so it does **not** advertise the VIP. Its RIB has only the route that R1 sends to it, which is www-1's route:

```
   Network              Next Hop             AS_PATH              Age        Attrs
*> 100.64.1.100/32      10.2.0.1             64512 65001          00:00:05
[{Origin: ?} {LargeCommunity: [ 64512:1:777, 64512:2:3]}]
```

Exit the remote shell using:

```
exit
```

On R1, both sessions to the servers are now up:

```
docker exec -it r1 vtysh -c "show bgp summary"
```

Task 3 – BIRD: route reflector with an advanced export filter

3.1 The existing BIRD config

First read the config that is already there:

```
docker exec -it bgp-rr cat /etc/bird/bird.conf
```

It has these parts:

`router id 100.65.0.3;`: the BIRD router ID.

`protocol device`: reads the interface list from the kernel every 5 s. Other protocols need it.

`protocol ospf v2 ospf_main`: OSPF area 0 on `eth1` (broadcast) and `lo` (stub). `import all` takes all OSPF routes into the BIRD table `master4`. The export filter rejects BGP routes, so BGP is never redistributed into OSPF.

`protocol kernel`: writes routes from `master4` into the Linux routing table. Only OSPF routes are exported (`RTS_OSPF`, `RTS_OSPF_IA`, `RTS_OSPF_EXT1/2`). BGP routes stay in BIRD only. The RR is not in the traffic path, so it does not need them in its kernel.

`function prepend_times(int n)`: a reusable function with a parameter. It prepends the AS path up to 3 times. Each prepend uses a different (private) ASN, so you can see in the path how many prepends were done:

`n`	Prepended (leftmost first)
1	`64521`
2	`64522 64521`
3	`64523 64522 64521`

`filter export_to_clients`: applied to every route the RR sends to its clients. `lc c;` declares a local variable of type *large community*. The filter runs these steps in order:

Step	Code	What it does
–	`if source != RTS_BGP then reject;`	Export only BGP routes. OSPF/device routes are never sent over BGP.
(1)	`if bgp_large_community ~ [(64512, 1, *)] then { c = filter(...).min; bgp_med = c.data2; }`	**Action community "set MED"**: if the route has a community `64512:1:X`, set MED = X. `filter(...)` keeps only the matching communities, and `.min` picks the lowest if there are several. `data2` is the third field (X).
(2)	`if bgp_large_community ~ [(64512, 2, *)] then { ...; prepend_times(c.data2); }`	**Action community "prepend N times"**: the originator chooses how much to make its route less preferred.
(3)	`bgp_local_pref = 200 - bgp_path.len * 10;`	**Local-pref from AS path length**: a shorter path gives a higher local-pref. The length is measured **after** the prepend in step (2), so prepending lowers local-pref.
(4)	`bgp_path.delete([65000..65100]);`	**Strip an ASN range**: removes all ASNs from 65000 to 65100 from the path. This removes the servers' private ASNs (65001, 65002).
(5)	`bgp_large_community.add((64512, 9, net.len));`	**Metadata community**: adds `64512:9:<prefix length>`, for example `64512:9:32` for a /32.
–	`bgp_large_community.delete([(64512, 1, *), (64512, 2, *)]);`	**Cleanup**: removes the action communities, so they are not sent further.

Step	Code	What it does
–	`accept;`	Send the changed route.

At the end, `include "/etc/bird.d/*.conf";` loads the extra config files. You add the BGP part in that directory.

3.2 Add the BGP configuration

Open a shell on bgp-rr:

```
docker exec -it bgp-rr bash
```

The config uses a **template**. All settings shared by the RR clients are defined once, and each `protocol bgp` inherits them with `from`:

- `local 100.65.0.3 as 64512`: source address (loopback) and local AS.
- `rr client`: this neighbor is a route-reflector client.
- `rr cluster id 100.65.0.3`: the cluster ID added to `CLUSTER_LIST` on reflected routes.
- `add paths rx`: accept multiple paths per prefix from the client (R1 sends them with `addpath-tx-all-paths`).
- `import all`: accept every route from the clients.
- `export filter export_to_clients`: apply the filter from Task 3.1 to every route sent to the clients.

Add configuration:

```
cat  > /etc/bird.d/bgp.conf <<EOF

template bgp ibgp_rr_client {
    local 100.65.0.3 as 64512;
    rr client;
    rr cluster id 100.65.0.3;

    ipv4 {
        add paths rx;
        import all;
        export filter export_to_clients;
    };
}

protocol bgp ibgp_r1 from ibgp_rr_client {
    description "RR client R1";
    neighbor 100.65.0.1 as 64512;
}

protocol bgp ibgp_r2 from ibgp_rr_client {
    description "RR client R2";
    neighbor 100.65.0.2 as 64512;
}

EOF
```

3.3 Check the config and apply it

First check the syntax only (`-p` = parse and exit). Nothing is changed:

```
bird -p -c /etc/bird/bird.conf
echo $?                          # 0 = config OK
```

Then tell the running BIRD to load the new config. This does not restart it:

```
birdc configure
```

Expected result:

```
Reading configuration from /etc/bird/bird.conf
Reconfigured
```

3.4 Verify the BGP sessions

Wait 10 seconds for BGP to establish:

```
birdc show protocols
birdc show protocols all ibgp_r1
```

Expected result:

```
Name       Proto      Table      State  Since         Info
ibgp_r1    BGP        ---        up     14:24:44.172  Established
ibgp_r2    BGP        ---        up     14:24:44.447  Established
ospf_main  OSPF       master4    up     14:18:03.086  Running
```

3.5 What the RR received (before the filter)

Show the routes that BIRD received from R1, with all attributes:

```
birdc show route protocol ibgp_r1 all
```

Expected result: the route is exactly as www-1 sent it. BIRD resolves the next hop `10.1.0.2` recursively through OSPF (`hostentry`, `igp_metric`).

```
100.64.1.100/32      unicast [ibgp_r1 ... from 100.65.0.1] * (100/20) [AS65001?]
        via 10.10.0.1 on eth1
        hostentry: via 10.1.0.2 table master4
        bgp_path: 65001
        bgp_next_hop: 10.1.0.2
        bgp_local_pref: 100
        bgp_large_community: (64512, 1, 777) (64512, 2, 3)
```

3.6 What the RR sends to R2 (after the filter)

Show the routes that BIRD sends to R2, after the export filter:

```
birdc show route export ibgp_r2 all
```

Expected result:

```
100.64.1.100/32      unicast [ibgp_r1 ... from 100.65.0.1] * (100/20) [AS64521?]
        bgp_path: 64523 64522 64521
        bgp_next_hop: 10.1.0.2
        bgp_med: 777
        bgp_local_pref: 160
        bgp_large_community: (64512, 9, 32)
```

Step by step for www-1's route:

Step	Result
in	path `65001`, LC `(64512,1,777) (64512,2,3)`
(1)	MED = `777`
(2)	prepend 3× → path `64523 64522 64521 65001`
(3)	path length 4 → local-pref `200 - 40 = 160`
(4)	`65001` deleted → path `64523 64522 64521`
(5)	LC `(64512,9,32)` added
out	the action communities are removed, only `(64512,9,32)` is left

```
birdc show route export ibgp_r1 all
```

This is empty. BIRD does not send a route back to the client it came from.

Exit the remote shell using:

```
exit
```

Task 4 – Verify on R2 and test the service

Check the iBGP session to the RR and the VIP route on R2:

```
docker exec -it r2 vtysh -c "show bgp summary"
docker exec -it r2 vtysh -c "show bgp ipv4 unicast 100.64.1.100/32"
```

Expected result:

```
  64523 64522 64521
    10.1.0.2 (metric 20) from 100.65.0.3 (100.65.0.1)
      Origin incomplete, metric 777, localpref 160, valid, internal, best (First path
received)
      Large Community: 64512:9:32
      Originator: 100.65.0.1, Cluster list: 100.65.0.3
```

- The route was reflected: `Originator 100.65.0.1` (R1) and `Cluster list 100.65.0.3`.
- MED, local-pref, AS path and communities are the values set by the BIRD filter.
- The next hop `10.1.0.2` (www-1) is reachable through OSPF via R1.

Test the service from R2:

```
docker exec -it r2 ip route get 100.64.1.100     # via 10.10.0.1 (R1)
docker exec -it r2 curl -s http://100.64.1.100/
```

The page must contain `www-1`.

Task 5 – Failover: www-1 → www-2 and back

Keep one terminal open to watch R2:

```
docker exec -it r2 sh -c 'while true; do curl -s --max-time 1 http://100.64.1.100/ |
grep "<p>"; sleep 1; done'
```

5.1 Stop nginx on www-1

Stop the web server on www-1, so that its health check fails:

```
docker exec -it www-1 sv stop /services/nginx
```

Within a few seconds, the following happens:

1. `check_http` on www-1 fails twice (`fall 2`). The priority drops from 130 to 110.
2. www-2 (120) becomes **MASTER**. It takes the VIP and runs its `notify_master`, so GoBGP advertises the VIP with `64512:1:888,64512:2:1`.
3. www-1 becomes **BACKUP**. It removes the VIP and runs its `notify_backup`, so GoBGP withdraws the VIP.
4. R1 → BIRD → R2: R2 now gets the route from www-2.

Verify:

```
docker exec -it www-1 ip -br addr show eth1      # no VIP
docker exec -it www-2 ip -br addr show eth1      # 100.64.1.100/32
docker exec -it bgp-rr birdc show route export ibgp_r2 all
docker exec -it r2 vtysh -c "show bgp ipv4 unicast 100.64.1.100/32"
```

Expected result on R2:

```
  64521
    10.2.0.2 (metric 20) from 100.65.0.3 (100.65.0.1)
      Origin incomplete, metric 888, localpref 180, valid, internal, best
      Large Community: 64512:9:32
```

Step by step for www-2's route: MED `888`, 1 prepend → path `64521 65002` (length 2) → local-pref `200 - 20 = 180`, `65002` deleted → path `64521`.

The loop on R2 now prints `www-2`.

5.2 Start nginx on www-1 again [optional]

Start the web server on www-1 again:

```
docker exec -it www-1 sv start /services/nginx
```

After `rise 2`, www-1 is back at priority 130 and takes over again (preemption). It advertises the VIP with its own communities, and www-2 withdraws.

Verify:

```
docker exec -it www-1 ip -br addr show eth1      # 100.64.1.100/32
docker exec -it www-2 ip -br addr show eth1      # no VIP
docker exec -it r2 vtysh -c "show bgp ipv4 unicast 100.64.1.100/32"   # 64523 64522
64521, metric 777, localpref 160
```

The loop on R2 prints `www-1` again. Stop it with `Ctrl+C`.

Troubleshooting checklist

Symptom	Check
GoBGP neighbor not `Establ`	`gobgp neighbor 10.1.0.1`, R1: `vtysh -c "show bgp neighbors 10.1.0.2"`, `ping 10.1.0.1`
VIP on the server, but not in `gobgp global rib`	`cat /etc/keepalived.d/notify.conf`, then `sv restart /services/keepalived` (notify runs only on a state change)
Both servers are MASTER	VRRP unicast between `10.1.0.2` ↔ `10.2.0.2`: `ping` between the servers, `auth_pass` and `virtual_router_id` must match
`bird -p` reports an error	The error shows the file and line, usually in `/etc/bird.d/bgp.conf`
BIRD sessions not `Established`	`birdc show protocols all ibgp_r1`, OSPF route to `100.65.0.1` (`birdc show route 100.65.0.1/32`)
R2 does not have the route	`birdc show route protocol ibgp_r1 all`, `birdc show route export ibgp_r2 all`
Route attributes on R2 are not as expected	`birdc show route export ibgp_r2 all`, then go through the filter steps in Task 3.1

keepalived log: `docker logs www-1 2>&1 | grep Keepalived_vrrp`. Use `docker logs -f www-1` to follow it live.

Cleanup

Destroy the lab:

```
containerlab destroy -t lab5.clab.yml
```