Namespace and Infrastructure as Code Simulation

JOSÉ BRÁS, Instituto Superior Técnico, Portugal

This report presents a systematic methodology for emulating and validating complex network topologies entirely within a single Linux host. Beginning with isolated two-node and three-node scenarios using Linux network namespaces and virtual Ethernet (veth) pairs, it demonstrates fundamental link creation, IP addressing, and static routing. The work then scales to a full multi-subnet environment by employing Linux bridges as simulated switches and router namespaces with IPv4 forwarding and static routes, confirming end-to-end connectivity across seven hosts and two routers. Finally, the report illustrates the use of Docker and Terraform to deploy a containerized web service architecture—comprising two backend servers and an Nginx load balancer—and validates service reachability, load-balancing behavior, resilience under failure, and dynamic scaling. The experiments underscore the power of user-space network namespaces for rapid prototyping, reproducible testing, and declarative infrastructure management.

1 Introduction

Linux network **namespaces** provide isolated networking contexts within a single Linux host, allowing the simulation of multiple separate networked nodes in user space. In combination with **virtual Ethernet (veth)** pairs (which act as virtual patch cables), one can construct complex network topologies without any physical hardware. These experiments demonstrate a progressive approach to virtual networking using namespaces and veth interfaces, starting from a simple two-node link and expanding to a multi-node routed network with bridges (simulated switches). Finally, a container-based deployment is managed with **Docker** and **Terraform**, illustrating how similar networking concepts apply to container networks in a declarative infrastructure setup.

All software and configuration files used in these experiments are published in the following repository: **https://github.com/sneakyjbras/network-virtualization**.

The objectives of these experiments are to explore how isolated network stacks can communicate through virtual links, to practice setting up static routing and bridging in a fully virtual environment, and to evaluate the use of infrastructure-as-code tools for deploying networked services. Each phase builds on the previous, moving from fundamental namespace connectivity to more complex scenarios including multiple subnets, routers, and an automated containerized environment. All configurations are performed in Linux user space, making the experiments reproducible and safe to run on a single host.

2 Two-Node Network with Linux Namespaces and Veth

Simulate a minimal network with **two nodes**: the default (global) network namespace and one additional Linux network namespace, connected by a point-to-point link using a **veth pair**. This will verify that two isolated namespaces can communicate when linked and properly configured.

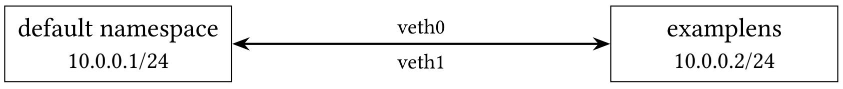

Fig. 1. Two-node network via a veth pair

- `veth0` resides in the default namespace (representing the first node). - `veth1` is moved into the `examplens` namespace (representing the second node). - Both interfaces form the two ends of a virtual Ethernet cable linking the two nodes.

Author's Contact Information: José Brás, jose.bras@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisbon, Portugal.

2.1 Steps Performed

2.1.1 1. Create Namespace. A new network namespace called `examplens` is created to represent the second node in isolation.

```
ip netns add examplens
```

2.1.2 2. Create veth Pair. A pair of virtual Ethernet interfaces is created. These two endpoints (named `veth0` and `veth1`) form a connected pair, functioning as a virtual link between the default namespace and the new `examplens` namespace.

```
ip link add veth0 type veth peer name veth1
```

2.1.3 3. Assign Interfaces to Namespace. One end of the veth pair is moved into the new namespace so that each namespace holds one end of the virtual link.

```
ip link set veth1 netns examplens
```

2.1.4 4. Assign IP Addresses. IP addresses are assigned in the same subnet to each veth endpoint, allowing IP-level connectivity.

```
ip addr add 10.0.0.1/24 dev veth0
ip netns exec examplens ip addr add 10.0.0.2/24 dev veth1
```

2.1.5 5. Bring Interfaces Up. Both veth endpoints and the loopback interface in the new namespace are activated so that the link becomes operational.

```
ip link set veth0 up
ip netns exec examplens ip link set veth1 up
ip netns exec examplens ip link set lo up
```

2.1.6 6. Test Connectivity. Bidirectional connectivity is confirmed with ICMP echo requests.

```
ping -c 4 10.0.0.2
ip netns exec examplens ping -c 4 10.0.0.1
```

2.2 Summary

- A veth pair simulated a physical link.
- One end was moved into a separate namespace.
- IPs were configured and interfaces brought up.
- Successful pings in both directions verified connectivity.

3 Three-Node Network with Namespaces and Static Routing

Extend the virtual network to **three nodes** in a linear topology. One node (`ns2`) will function as a router interconnecting two subnets via static routing and IP forwarding.

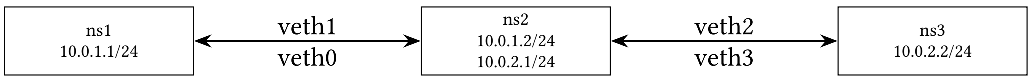

Fig. 2. Compressed three-node routed network with namespaces

- `ns1` and `ns3` are endpoints on separate subnets. - `ns2` routes between subnets 10.0.1.0/24 and 10.0.2.0/24.

3.1 Steps Performed

3.1.1 1. Cleanup. Any existing namespaces or veth interfaces are removed to ensure a pristine environment.

```
ip netns add ns1
ip netns add ns2
ip netns add ns3


ip link add veth0 type veth peer name veth1
ip link add veth2 type veth peer name veth3
```

Interfaces are assigned:

```
ip link set veth1 netns ns1
ip link set veth0 netns ns2
ip link set veth2 netns ns2
ip link set veth3 netns ns3


ip netns exec ns1 ip addr add 10.0.1.1/24 dev veth1
ip netns exec ns2 ip addr add 10.0.1.2/24 dev veth0
ip netns exec ns2 ip addr add 10.0.2.1/24 dev veth2
ip netns exec ns3 ip addr add 10.0.2.2/24 dev veth3


ip netns exec ns1 ip link set veth1 up
ip netns exec ns2 ip link set veth0 up
ip netns exec ns2 ip link set veth2 up
ip netns exec ns3 ip link set veth3 up
ip netns exec ns1 ip link set lo up
ip netns exec ns2 ip link set lo up
ip netns exec ns3 ip link set lo up


ip netns exec ns2 sysctl -w net.ipv4.ip_forward=1


ip netns exec ns1 ip route add 10.0.2.0/24 via 10.0.1.2
ip netns exec ns3 ip route add 10.0.1.0/24 via 10.0.2.1


ip netns exec ns1 ping -c 4 10.0.2.2
ip netns exec ns3 ping -c 4 10.0.1.1
```

3.2 Summary

- Three namespaces connected by veth.
- IP forwarding enabled on ns2.
- Static routes configured.
- End-to-end pings confirmed routing functionality.

4 Full Multi-Subnet Network Emulation with Bridges and Routers

Simulate a multi-router environment with seven host namespaces (H1–H7), two routers (R1, R2), and three LAN segments bridged by Linux bridges.

Table 1. Topology & IP Plan

Segment	Bridge	Subnet	Devices
A	brA	10.0.1.0/24	H1 (10.0.1.11), H2 (10.0.1.12)
B	brB	10.0.2.0/24	H5 (10.0.2.51), H6 (10.0.2.52), H7 (10.0.2.53)
C	brC	10.0.3.0/24	H3 (10.0.3.31), H4 (10.0.3.32)
D (core)	none	10.0.4.0/30	R1↔R2 core link

4.1 Implementation Steps

```
# Full setup and connectivity test outline:

echo "=== Cleanup ==="
# Remove existing namespaces, veths, and bridges.

echo "=== Create namespaces ==="
# Create H1 toH7, R1, R2.

echo "=== Create bridges A, B, C ==="
# Create brA, brB, brC.

echo "=== Attach hosts to bridges ==="
# For each host, create and attach veth pair to appropriate bridge.

echo "=== Routers & core link ==="
# Attach R1 to brA and brB; R2 to brC.
# Create veth pair for core link between R1 and R2.

echo "=== IP addressing & bring-up ==="
# Assign IPs as per plan; bring interfaces and bridges up.

echo "=== Enable routing ==="
```

```
# Enable IPv4 forwarding on R1 and R2.

echo "=== Static routes ==="
# Hosts: default route via respective router.
# Routers: routes to other LANs via core link.

echo "=== Connectivity Tests ==="
# Ping between hosts on different segments to verify routing.
```

4.2 Summary

- Emulated LAN segments with Linux bridges.
- Connected hosts and routers via veth.
- Enabled routing and static routes.
- Verified inter-LAN connectivity across R1 and R2.

5 Docker Container Deployment with Terraform (Load Balancer & Web Servers)

Terraform was used to deploy two web server containers (`web1`, `web2`) and one Nginx load balancer container (`load_balancer`) on Docker. A test script (`run.sh`) validated connectivity, load balancing behavior, resilience, and scaling.

5.1 Terraform Workflow and Infrastructure Setup

```
terraform init
terraform plan
terraform apply -auto-approve
terraform destroy -auto-approve
```

- **init:** downloads provider and prepares state.
- **plan:** previews infrastructure changes.
- **apply:** provisions containers and network.
- **destroy:** cleans up resources.

5.2 IP Address Allocation

Container	Internal IP	Host Port Mapping
web1	172.18.0.2	—
web2	172.18.0.3	—
load_balancer	172.18.0.4	8080 $\longrightarrow$ 80

5.3 Web Server Accessibility Testing

```
curl http://172.18.0.2
curl http://172.18.0.3
```

Both servers responded correctly, confirming reachability.

5.4 Load Balancer Request Distribution

```
for i in {1..4}; do curl -s http://172.18.0.4; echo; done
```

Responses alternated between Web Server 1 and 2, demonstrating round-robin distribution.

5.5 Port Mapping and Browser Access

- Host port 8080 is mapped to LB container port 80.
- Access via `http://localhost:8080/` shows alternating content in a browser.

5.6 Resilience to a Backend Server Failure

```
docker stop web1
for i in {1..3}; do curl -s http://localhost:8080; echo; done
```

All responses came from Web Server 2 only, showing LB resilience.

5.7 Scaling Out to Three Web Servers

(1) Update Terraform config:

```
-count = 2
+count = 3
 message = "Web Server ${count.index + 1}"
```

(2) Apply changes:

```
terraform plan
terraform apply
```

(3) Verify distribution:

```
for i in {1..6}; do curl -s http://localhost:8080; echo; done
```

Responses cycled through Web Server 1, 2, 3.

5.8 Evaluation of Design Choices

The following evaluation outlines the principal architectural and tool decisions made during the Docker/Terraform deployment. Each choice is weighed in terms of its benefits for reproducibility, maintainability, and operational simplicity, as well as potential drawbacks or alternative approaches.

These summarized trade-offs provide a clear basis for refining the deployment architecture in future iterations, guiding choices around orchestration platforms, load-balancing strategies, and testing frameworks.

Table 3. Evaluation of Design Choices

Decision	Pros	Cons / Alternatives
Terraform + Docker provider	Declarative, reproducible, stateful; easy teardown	Tightly coupled to Docker; consider Kubernetes or cloud LB for production scalability.
`count` for scaling	Simple numeric scaling	Lacks descriptive resource naming; modules or dynamic blocks offer more maintainability.
Nginx load balancer in container	Lightweight and familiar; quick deployment	Single point of failure; HAProxy or multiple LB instances provide better high availability.
Bash test script (`run.sh`)	Quick sanity checks; integrates with apply	Not part of CI/CD; Terratest or similar frameworks enable automated integration testing.

6 Conclusion

This report has demonstrated a progressive methodology for emulating complex network topologies entirely within a single Linux host. Through sequential experiments, the following achievements were realized:

- **Namespace Isolation & Veth Linking:** Two-node and three-node scenarios validated that Linux network namespaces, paired with veth interfaces, can simulate point-to-point links and multi-subnet routing with explicit IP addressing and static routes.
- **Layer-2 Bridging & Multi-Router Emulation:** The full multi-subnet lab employed Linux bridges to emulate LAN segments, and two router namespaces with IP forwarding and static routes to interconnect them, proving end-to-end connectivity across seven hosts and two routers.
- **Infrastructure-as-Code Deployment:** The Docker/Terraform phase illustrated how containerized web services and an Nginx load balancer can be provisioned declaratively, tested for connectivity, and shown to support resilience (backend failure) and dynamic scaling (adding a third web server).

Collectively, these experiments confirm that sophisticated network behaviors—spanning from fundamental Layer-2 links to full Layer-3 routed and load-balanced service deployments—can be modeled, tested, and iterated entirely in user space. This approach enables rapid prototyping, reproducible testing, and seamless integration with automation tools such as Terraform, making it a valuable methodology for both educational purposes and pre-production network design validation.