Networking with Linux

Master Linux networking from switching to VPNs, monitoring, troubleshooting, and tuning your networks



Chandan Dutta Chowdhury

bpb

Networking with Linux

Master Linux networking from switching to VPNs, monitoring, troubleshooting, and tuning your networks



Chandan Dutta Chowdhury

bpb

Networking with Linux

Master Linux networking from switching to VPNs, monitoring, troubleshooting, and tuning your networks

Chandan Dutta Chowdhury



www.bpbonline.com

First Edition 2026

Copyright © BPB Publications, India

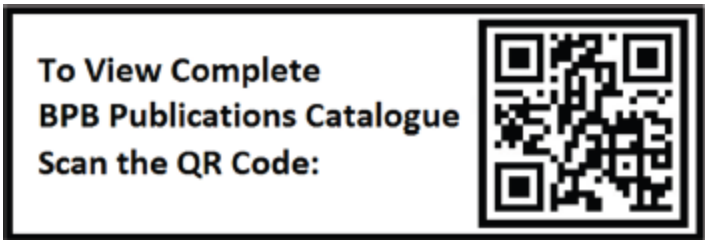
ISBN: 978-93-78546-860

All Rights Reserved. No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true and correct to the best of author's and publisher's knowledge. The author has made every effort to ensure the accuracy of these publications, but the publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.



www.bpbonline.com

Dedicated to

My parents,

Kiran Moy Dutta Chowdhury *and* ***Manju Dutta Chowdhury***

About the Author

Chandan Dutta Chowdhury is a principal engineer with over 20 years of experience in designing, securing, and troubleshooting Linux and networking. With a background in platform engineering, DevOps, and network management, he has worked with enterprises, cloud platforms, and open-source communities to build resilient and high-performance systems. Chandan has trained professionals in Linux administration and contributed to OpenStack networking and firewall implementations, and network automation through workshops and conferences. He actively contributes to open-source projects and writes technical guides to simplify complex networking concepts. Chandan has worked with multiple companies in his IT career, like HP, Juniper Networks, Cisco, Arcesium, and contributed to their product and infrastructure projects. Currently, he contributes to platform optimizations, maintaining Kubernetes clusters and Linux compute infrastructure.

About the Reviewers

- **Omkar Wagle** is an experienced distributed systems and embedded software engineer. He is currently focused on working with load balancers that manage traffic access storage on the cloud. His work primarily involves developing critical features that enhance customer experience and reduce delays due to excessive traffic ingress. He has also worked on smart home projects, developing various features for smart home solutions on ARM9 gateways, highlighting his knowledge of operating systems, threads, and low-memory SOCs.
- **Rafiuddin Syed** is a systems software engineer specializing in operating systems, real-time platforms, and high-performance embedded systems. He currently serves as director of engineering at Drako Motors (a subsidiary of IC Manage, Inc.), where he leads the architecture and implementation of next-generation compute platforms for electric vehicles. His work focuses on Quest RTOS and the Quest-V partitioning hypervisor, mixed-criticality systems, and deterministic communication frameworks for software-defined vehicles.

  Rafiuddin has extensive experience in Linux kernel development, USB subsystem engineering (xHCI/xDCI/xDBC), and real-time automotive data pipelines, including CAN-to-USB gateways and RT-DDS–based distributed systems. He actively contributes to open-source projects and collaborates with academic researchers on advanced real-time and virtualization technologies. In addition to his engineering leadership, Rafiuddin serves as a technical reviewer and mentor within the systems and embedded software community, with a strong interest in bridging industry practice and academic research.

Acknowledgement

I want to express my deepest gratitude to my family for their unwavering support and encouragement throughout the writing of this book.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey from envisioning to the completion of this book, with the valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and coworkers in the tech industry, who have taught me so much and provided valuable feedback on my work.

Preface

Linux provides a great platform for implementing network services, offering software solutions for everything from routing, load balancing, and VPNs to advanced firewall configurations. As containers and cloud computing have matured, Linux has become the platform of choice for these technologies. The foundation for these technologies has been the first-class support for virtual networks and the network stack isolation provided by Linux namespaces.

This book is designed as a ground-up exploration of networking on the Linux operating system. It builds the reader's understanding beginning with the OSI model and examining Linux's functionality at each layer. You will learn how Linux implements core concepts like switching and routing, progressing into advanced topics like link aggregation, multipath routing, policy-based routing, and BGP-based dynamic routing.

The book provides you with multiple perspectives on networking on Linux by examining both the established conventions and the emerging technology. For instance, in the case of firewalls, the book explores iptables as the dominant standard in the enterprise today, while also introducing the next generation of firewall implementation with nftables.

With this book, the readers will understand how to build load balancers, virtual networks, and container-ready infrastructures on Linux. By the end, you will possess the skills to design, configure, troubleshoot, and manage network solutions on the Linux platform.

Chapter 1: Understanding Networking on Linux — The chapter starts with the explanation of various layers of the OSI networking model and their functionality. It progresses into explaining the networking concepts of Layer 2 of the OSI model in the context of Linux and explores the physical and virtual network interfaces. The chapter further explains the concepts of bridging Layer 2 networks, VLANs, and various modes of link aggregation.

Chapter 2: Routing and Packet Forwarding — This chapter explores the Layer 3 networking concepts like IP addressing, subnetting with classful and classless addresses, and automatic network address configuration with the DHCP service. The chapter delves into routing on Linux and advanced topics like multipathing, policy-based routing, virtual routing, and forwarding. Finally, the chapter explains how to set up dynamic routing on Linux with Border Gateway Protocol.

Chapter 3: Firewalls and Packet Filtering — This chapter explains the implementation of firewalls on Linux with iptables. It explains the core concepts of the iptable, the flow of packets through the various tables that compose the firewall. The syntax of iptable rules and the actions that a rule can take are explained. The chapter also explores the concept of Network Address Translation and the use cases for source and destination address translations. IPsets and higher-layer tools like firewalld are also covered in this chapter.

Chapter 4: Network Troubleshooting and Diagnostics — This chapter explains the various tools that are used to troubleshoot and monitor networks on Linux. The chapter explores tools like ping for checking basic connectivity, traceroute to check the routing of packets, and ss and netstat for monitoring socket-level statistics. The chapter also explains tracing network connections with tcpdump, Wireshark, and tshark, and finally concludes with an explanation of NetFlow for monitoring high traffic volumes with ntopng.

Chapter 5: Network Services and Protocols — This chapter covers the configuration of the most common and essential network services like DNS, NTP, SSH, NFS, and HTTP. It explains the resolution of domain names with a recursive resolver and the configuration of the Domain Name Service. The working and configuration of the Network Time Protocol, which forms the basis of time synchronization on a network, is explained. The chapter also explains the Secure Shell service, which is the de facto standard for remote access to a Linux server. Finally, the chapter explains the Network File System for file sharing on the network and the HTTP service using NGINX configuration.

Chapter 6: Load Balancing and High Availability — This chapter explains the Load balancing and high availability services on Linux. The

concept of Layer 3 high availability and virtual IP is explained. The chapter explores the implementation of high availability with Keepalived service and BGP. The chapter then explains the implementation of Layer 4 load balancing service with iptable and IP Virtual Server. The chapter then explains load balancing at Layer 7 with HAProxy.

Chapter 7: Network Namespaces and Virtual Networking — This chapter explains network virtualization and the function of network namespaces. It explains the concept of isolated network stacks and their implementation with namespaces on Linux. The chapter demonstrates how to set up namespaces and how they function with examples. The chapter then progresses into network isolation with MACVLANS and IPVLANS and userspace networking with the TUN/TAP interfaces.

Chapter 8: Traffic Shaping and QoS — This chapter explains how quality of service can be implemented on Linux with traffic shaping. The chapter explains the concept of bandwidth as a finite resource and how to apply various traffic control mechanisms to best utilize the network bandwidth for a given scenario. The chapter delves into the various traffic queuing disciplines, applies the rate-limiting techniques, and also explains the concept of buffer bloat and mitigation.

Chapter 9: Deep Dive into TCP/IP Stack Tuning — This chapter explains the sysctl and the proc filesystem interfaces provided by Linux to tune the TCP/IP stack. It explains the scenario of tuning the system to utilize maximum bandwidth, connection handling with TCP Fast Open, and connection tear-down. Connection persistence with Keepalive and hardening against attacks like Syn-Flood are explained in the chapter. The chapter also delves into the evolution of the application-layer protocol HTTP/3 and QUIC.

Chapter 10: Virtual Private Networks on Linux — This chapter examines the virtual private network services on Linux. The chapter explains the concept of a virtual private network and its use cases for Site-to-Site and Point-to-Site configurations. The chapter explains the configuration of VPN with WireGuard and IPSEC using both certificates and pre-shared keys.

Chapter 11: Virtual Networking and Cloud Integration — The chapter explains the container and Kubernetes pod networking with various

Container Networking Interfaces plugins like Calico and the overlay network to connect the container across multiple hosts. Common overlay networks like VXLAN and GRE are explained. The concept of software defined networks with OpenVSwitch and ovsdb is covered. It also explains Virtual Private Cloud provided by popular cloud providers like AWS, Azure, and GCP, and explains how the concept of virtual networking applies to the cloud environment.

Chapter 12: Network Monitoring with eBPF — This chapter explores the modern network monitoring capability of Linux with eBPF. The chapter starts by explaining the shortcomings of traditional monitoring solutions and progresses to explain the eBPF framework to use probes and trace points to gain insight into network monitoring. The chapter provides practical examples of using XDP, the bpftrace tool, and writing a custom eBPF program for tracing and collecting statistics of network functions.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/c97796

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Networking-with-Linux**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks. You can check our social media handles below:


Instagram


Facebook


Linkedin


YouTube

Get in touch with us at: **business@bpbonline.com** for more details.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



Table of Contents

1. **Understanding Networking on Linux**
    - Introduction
    - Structure
    - Objectives
    - Introduction to OSI model of networking
        - *Implement application-specific protocols*
    - Basic network configuration on modern Linux distributions
        - *Configuring the hostname of your system*
        - *Configuring the IP address of your host*
        - *Configuring network on Rocky Linux*
        - *Configuring the network on Ubuntu*
    - Examining physical and virtual interfaces
        - *Using VETH interfaces*
        - *Bridging Layer 2 networks*
        - *VLAN networks*
    - Link aggregation and bonding
        - *Bonding with LACP*
    - Understanding Layer 2 protocols
        - *Address resolution in Layer 2 networks*
        - *Fault tolerance in Layer 2 networks*
        - *Neighbour discovery*
    - Conclusion
2. **Routing and Packet Forwarding**

Introduction
Structure
Objectives
IP addressing
Netmask and subnetting
Classful addressing
Classless addressing
DHCP service for automatic IP address configuration
Configuring DHCP service on Linux
Routing in Layer 3 networks
Equal-cost multi-path routing
Policy-based routing and VRF
Comparison with VLAN
Common VRF applications
Policy-based routing
Dynamic routing
Conclusion

3. Firewalls and Packet Filtering

Introduction
Structure
Objectives
Core concepts of iptables
Structure of iptables configuration
Firewall rules
Firewall chains
Firewall tables
Basic iptables commands and syntax
Command structure overview
Essential command categories

Example use cases for iptables
Connection tracking in iptables
Network Address Translation
Source NAT
Using ipsets with iptables
Types of ipsets
Hash-based ipsets
List-based ipsets
Special purpose ipsets
Timeout and metadata-enabled ipsets
Testing ipsets
The DShield project
Evolution from iptables to nftables
nftables families
nftables tables
nftables chains
nftables rules
Linux firewall management with firewalld
Conclusion

4. Network Troubleshooting and Diagnostics

Introduction
Structure
Objectives
Testing connectivity with ping
Useful ping command options
Using traceroute to diagnose network issues
Analyzing traceroute results
Limitations of traceroute
Using My Traceroute for network troubleshooting

Port and socket inspection
Netstat
Netcat
Checking ports open on a host
Using NC as a temporary server and client
Using Netcat for file transfer
Socket statistics
lsof
Troubleshooting using packet captures
Using tcpdump to capture packets
Basic filter structure
Host-based filtering rules
Network-based filtering rules
Network port-based filtering rules
Protocol based filtering rules
TCP flag-based filtering rules
Packet size-based rules
Combining filter rules
Limiting the size of tcpdump captures
Reading PCAP files
Using Wireshark and TShark for packet capture
Wireshark
Filtering packets
TShark
NetFlow
Enabling NetFlow on Linux
Adding hosts with nProbe NetFlow exporter
Conclusion

5. Network Services and Protocols

Introduction

Structure
Objectives
Domain Name System
DNS service on Linux using BIND
Using RNDC to administrate the BIND service
Network Time Protocol
Understanding the working of NTP
Configuring systemd-timesyncd on Linux
Configuring NTP on Linux
Configuring chrony service
Secure Shell
Configuring key-based authentication
SSH port forwarding
Network File System
Configuring NFS server
Configuring NFS client
Hypertext Transfer Protocol
Configuring NGINX HTTP szerver
Configuring SSL certificates on NGINX for HTTPS
Using NGINX as a reverse proxy
Configuring multiple backend servers with upstream
Using health checks for backend servers
Using NGINX cache to improve server performance
Conclusion

6. Load Balancing and High Availability

Introduction
Structure
Objectives
High availability at Layer 3

Installation and configuration of Keepalived
IPVS and iPTables load balancing
Installation and configuration of IPVS
Scaling a high-traffic e-commerce API
Configuration details
High-performance TLS termination cluster
Load balancing UDP-based traffic
High availability at Layer 4
Using BGP for high availability
Layer 7 load balancers
Installation and configuration of HAProxy
Global and defaults sections
Frontend section
Backend section
Stats page
Conclusion

7. Network Namespaces and Virtual Networking

Introduction
Structure
Objectives
Need for network virtualization
Use cases for the adoption of network virtualization
Network namespace
Lifecycle of a network namespace
Interaction with other namespace types
Inspection and debugging network namespaces
IPRoute2 toolkit
Core syntax and structure of ip command
Creating a network namespace and verifying isolation

Using MACVLAN for bypassing the host stack
MACVLAN modes
Configuring a MACVLAN interface in bridge mode
Scaling with shared MACs using IPVLAN
Comparing IPVLAN and MACVLAN
IPVLAN operational modes
Implementing an IPVLAN L2 network
Integrating VLANs with namespaces
Configuring a VLAN interface inside a namespace
User-space networking with TUN and TAP
Primary use cases of VPNs, hypervisors, and emulation
Creating a TUN device
Understanding Docker container networking
Docker container lifecycle
Inspecting Docker namespaces and interfaces
Docker's container network model
Kubernetes networking fundamentals
Pod's network namespace
Container network interface standard
CNI plugin workflow
Network debugging in an orchestrated world
Host-level node inspection
Security hardening and considerations
Firewall between namespaces with network policies
Performance considerations
Conclusion

8. Traffic Shaping and QoS

Introduction
Structure

Objectives
Understanding the fundamental problem
Differentiating between QoS and traffic control
Exploring the key QoS metrics
Linux traffic control system
Examining the core components of traffic control
Differentiated services code point
First-in first-out qdiscs
Rate limiting with token bucket filters
Hierarchical tree for bandwidth allocation
Stochastic fairness queuing
Prioritizing traffic flows with priority queuing
Classifying packets into streams
Active queue management
Policing traffic to enforce a strict rate limit
Applying traffic shaping to ensure predictable rates
Ingress and egress policing and shaping
Integrating traffic control with netfilter
Deploying a complete and practical traffic shaping policy
Conclusion

9. Deep Dive into TCP/IP Stack Tuning

Introduction
Structure
Objectives
The proc filesystem and sysctl
Tuning for maximum throughput
Tuning connection handling
TCP Fast Open

SYN backlog tuning
Managing connection teardown
Keepalives and connection persistence
Tuning for stability
Hardening against SYN floods
Monitoring for packet drops
TCP congestion control
CUBIC
Bottleneck bandwidth and round-trip time
Choosing an algorithm
Modern web server with QUIC and HTTP/3
Practical implementation
Conclusion

10. Virtual Private Networks on Linux
Introduction
Structure
Objectives
Understanding VPNs
Understand how VPN works
Types of VPNs
VPN protocols
Introduction to WireGuard
WireGuard core concepts
Crypto routing table
Installation of WireGuard
Generating cryptographic keys
Configuring a Point-to-Site VPN
Introduction to IPsec and strongSwan
Core IPsec concepts

IPsec tunnel and transport modes
Installing strongSwan
Authentication methods of PSK vs. certificates
Configuring a Site-to-Site VPN with PSK
Introduction to PKI and Certificate Authority
PKI hierarchy
Using strongswan-pki
Configuring remote access with certificates
WireGuard vs. strongSwan and when to use which
Security best practices
Conclusion

11. Virtual Networking and Cloud Integration

Introduction
Structure
Objectives
Virtual networking and containers
Overlay networking with VXLAN and GRE
Virtual eXtensible LAN
Generic Routing Encapsulation
Open vSwitch as an overlay engine
Creating and managing overlay tunnels
Open vSwitch and software-defined networking
Programming Open vSwitch
Docker networking models
Container network interface
Flannel CNI plugin
Flannel with Docker
Calico BGP-based CNI
Cilium eBPF-based CNI

Cloud networking
AWS VPC Cloud Networking
Microsoft Azure Virtual Network
Google Cloud VPC
Conclusion

12. Network Monitoring with eBPF

Introduction
Structure
Objectives
Limitations of traditional network monitoring
Architecture of an eBPF program
Networking and eBPF
Using eBPF tools
Types of eBPF programs
Writing an eBPF program
Program for inspecting packets with XDP
Tracing network functions
Tracepoints vs kprobes
Monitoring socket-level activity
Security considerations and best practice
Network monitoring with Tetragon
Conclusion

Index

CHAPTER 1

Understanding Networking on Linux

Introduction

Linux is a versatile, open-source operating system that forms the backbone of today's digital infrastructure. As an operating system, it serves as the crucial link between computer hardware and software, enabling devices to function and users to interact with their applications seamlessly. What sets Linux apart is its open nature; the complete source code is publicly accessible, allowing anyone to examine, customize, and share it freely.

Linux scales effortlessly across devices, from personal workstations and smartphones to massive supercomputers and global cloud platforms. Its highly adaptable architecture gives users control to tailor virtually every component to their specific requirements.

Networking capabilities are integral to Linux's functionality, facilitating device communication, internet connectivity, and resource sharing. Whether configuring a personal computer, deploying cloud services, or maintaining a home server, mastering Linux networking fundamentals unlocks the system's full potential for both personal and professional use. The combination of Linux's flexibility, reliability, and networking prowess makes it an ideal choice for diverse computing environments.

Structure

This chapter covers the following topics:

- Introduction to OSI model of networking
- Basic network configuration on modern Linux distributions
- Examining physical and virtual interfaces
- Link aggregation and bonding
- Understanding Layer 2 protocols

Objectives

By the end of this chapter, you will be able to understand the **Open Systems Interconnection** (**OSI**) model of networking, learn how to configure your Linux distribution to make it a part of the network, and configure the **Domain Name System** (**DNS**) on your host to resolve hostnames to addresses. You will also gain knowledge on how a network interface works and how the Layer 2 network functions, how to build redundancy and fault tolerance in your network. You will learn about some of the protocols that operate in the Layer 2 networks and how two or more Layer 2 networks connect.

Introduction to OSI model of networking

The OSI model serves as a conceptual framework that abstracts and standardizes network communication. It establishes a common language for diverse systems, enabling machines with different hardware and software to communicate effectively through agreed-upon networking protocols.

Network communication between devices is a complex process involving multiple interdependent systems working together. This includes:

- Establishing physical electrical connections.
- Implementing addressing schemes to identify the source and destination.

- Ensuring error-free data transmission.
- Managing data flow rates.

The OSI model organizes these networking functions into a layered architecture, creating a comprehensive networking stack. Each layer provides distinct services and capabilities, with higher layers building upon and utilizing the functionality of lower layers. This hierarchical approach enables modular design, facilitates easier troubleshooting, and promotes standardized protocol development.

The OSI model represents the network as a stack of seven different layers of protocols, as shown in the following figure:

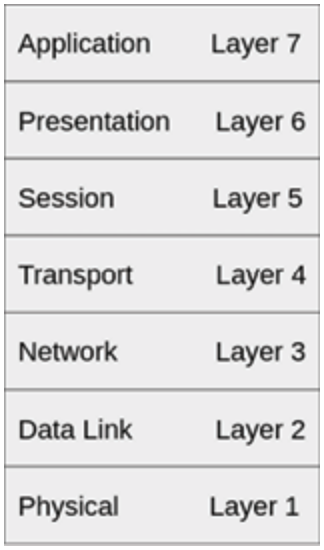


***Figure 1.1**: OSI model*

- **Physical layer**: The physical layer serves as the fundamental building block of network communication, establishing the actual physical and electrical connections between devices.

  This foundational layer:

  - Defines all hardware specifications and standards for network connections.
  - Governs physical components, including cables, connectors, and interface types.
  - Regulates electrical characteristics such as voltage levels and

signal timing.
    - Controls data transmission rates and synchronization.

  At this most basic level, all communication occurs through raw bit streams, data is transmitted and received as electrical impulses, light signals (in fiber optics), or radio waves (in wireless networks). The physical layer ensures these fundamental signals can reliably travel between networked devices.
- **Datalink layer**: The data link layer establishes and maintains reliable communication between directly connected network devices. Building upon the physical layer's raw transmission capabilities.

  This layer provides critical functions:

    - Organizes data into structured frames for transmission.
    - Detects and corrects transmission errors introduced at the physical level.
    - Regulates data transmission rates between devices with different capabilities.
    - Uses **media access control** (**MAC**) addresses to uniquely identify network interfaces.

  This layer receives raw bit streams from the physical layer and packages them into frames. It implements error correction mechanisms, adds frame headers containing source/destination MAC addresses, and controls access to shared physical media.
- **Network layer**: The network layer is responsible for end-to-end communication across interconnected networks. It provides logical addressing and routing of data. The endpoints of communication are addressed with their IP addresses. The data is represented in units of packets. This layer implements the transmission of data using route tables. This layer provides the following functions:

    - Assigning and managing IP addresses (IPv4/IPv6) to uniquely identify devices globally.
    - Structuring data into packets and determining the optimal path for transmission.

- Using routing tables and protocols (e.g., OSPF, BGP) to efficiently direct traffic between networks.

- **Transport layer**: The transport layer provides addressability to the applications running on a machine connected to the network. The data is transmitted in segments. **Transmission Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**) work in this layer. The transport layer enables process-to-process communication by providing these key functions:
  - Port-based addressing uses port numbers to identify specific applications and services on a device (e.g., HTTP=80, HTTPS=443).
  - Packages data into transmission units called segments.
  - Protocol implementation supports both TCP for reliable, connection-oriented delivery with error checking and UDP for fast, connectionless delivery for time-sensitive applications.
- **Session layer**: The session layer manages dialogue control between networked devices, establishing, maintaining, and terminating logical connections (sessions) between applications. This layer provides functions like:
  - Session establishment initiates, authenticates, and maintains connections between endpoints.
  - Session termination closes connections to free up application resources.
  - Dialogue control that manages full-duplex (two-way) or half-duplex (one-way) communication.
  - Synchronization checkpoints to recover interrupted sessions (e.g., resuming a file transfer).
- **Presentation layer**: The presentation layer acts as the network's universal translator, ensuring data is properly formatted, encrypted, and compressed for applications to understand.

  The functions provided by this layer are as follows:
  - Data translation, like handling character encoding, managing the

structure of data like JSON/XML.
    - Data compression, like zipping content.
    - Data encryption and decryption.
- **Application layer**: The application layer is the topmost layer of the OSI model, serving as the direct interface between end-users and network services. This is where human interactions with the network happen. This layer provides the following function:
    - **Network to service interface**: Enables communication between software programs and lower network layers.

Implement application-specific protocols

The OSI model serves as the fundamental framework for comprehending network protocols and their integration within the broader networking architecture. By systematically organizing communication functions into seven distinct layers, it establishes a universal reference point that clarifies how various protocols operate and interact across different levels of network communication and demonstrates the hierarchical relationship between networking components

The OSI model provides a structured approach to analyzing protocol functionality within the complete network stack. Offers a standardized vocabulary for discussing network operations and troubleshooting

In this section, we understood the OSI network model and its layers. In the rest of the chapter, we will discuss Layer 2 concepts. However, first, we will learn the basic network configuration on common Linux distributions.

Basic network configuration on modern Linux distributions

In this section, we will learn how to configure the network on Rocky Linux and Ubuntu, two of the popular Linux distributions. We will learn about setting the hostname for your Linux host, setting the IP address, the netmask, and the default gateway. We will also set the DNS resolver for the

host so that we can resolve hostnames.

Configuring the hostname of your system

To check the currently configured hostname on the machine, we can use the **hostname** command from the **hostname** package.

Setting the hostname on a modern Linux distribution can be done using systemd provided utility **hostnamectl**.

We can use the **hostnamectl** to check and set the hostname on the system as shown here:

```
[chandan@localhost ~]$ hostnamectl
   Static hostname: (unset)
Transient hostname: localhost
         Icon name: computer-vm
           Chassis: vm
        Machine ID: 040316b02b0f4b5384b32ab34b5b5faf

     Boot ID: f6a03fd1db5e459fadce2fb7f846a02e

    Virtualization: kvm Operating System: Rocky Linux 9.5 (Blue Onyx)
       CPE OS Name: cpe:/o:rocky:rocky:9::baseos
            Kernel: Linux 5.14.0-503.14.1.el9_5.x86_64
      Architecture: x86-64
   Hardware Vendor: QEMU
    Hardware Model: Standard PC Q35 + ICH9, 2009
  Firmware Version: 1.16.3-debian-1.16.3-2
```

To set the hostname of the system, we can use the following form of the command:

```
[chandan@localhost ~]$ sudo hostnamectl hostname rocky9
```

Note: Setting the hostname requires administrator privileges. The hostname configured is stored in the /etc/hostname **file.**

```
[chandan@localhost ~]$ cat /etc/hostname
rocky9
```

As this CLI is provided by systemd, it can be used on both Rocky Linux and Ubuntu systems.

Configuring the IP address of your host

The IP address on the machine can be configured using the **ip** command provided by the **iproute** package. On Ubuntu, this package is named as **iproute2**.

First, check the Ethernet interface on your machine. You can use the following **ip link show** command to do so:

```
$ ip link show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq state UP mode DEFAULT group default qlen 1000
    link/ether 52:54:00:ee:6a:84 brd ff:ff:ff:ff:ff:ff
```

The preceding output shows two interfaces on the machine: the loopback interface and **enp1s0**. The loopback network interface (usually named lo) is a virtual network interface that a computer uses to communicate with itself. It is normally configured with **ip** address of 127.0.0.1 and ::1. The physical network interface on this machine is **enp1s0**. We will discuss the network interface naming convention in Linux later in the chapter.

To check the currently configured IP address on your system, use the following command:

```
$ ip addr show
```

The output is as follows:

```
chandan@ubuntu24:~$ ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host noprefixroute
       valid_lft forever preferred_lft forever
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 52:54:00:ee:6a:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.61/24 brd 192.168.122.255 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:feee:6a84/64 scope link
       valid_lft forever preferred_lft forever
```

***Figure 1.2**: IP address configuration*

To set or add **ip** address of an interface on the system, use the following **ip** command, also illustrated in the following figure:

```
$ sudo ip addr add dev enp1s0 10.0.0.10/8
$ ip addr show dev enp1s0
```

```
chandan@ubuntu24:~$ sudo ip addr add dev enp1s0 10.0.0.10/8
chandan@ubuntu24:~$ ip addr show dev enp1s0
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 52:54:00:ee:6a:84 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.61/24 brd 192.168.122.255 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet 10.0.0.10/8 scope global enp1s0
       valid_lft forever preferred_lft forever
    inet6 fe80::5054:ff:feee:6a84/64 scope link
       valid_lft forever preferred_lft forever
chandan@ubuntu24:~$
```

***Figure 1.3**: Interface with additional IP address*

The **ip** commands we used to set the aforementioned IP address do not set the IP address configuration permanently. The changes we made previously will be lost after the system reboots. Ubuntu and Rocky Linux provide different mechanisms to make the network configuration permanent

Rocky Linux uses the NetworkManager while Ubuntu uses **netplan** with **systemd-networkd**.

Configuring network on Rocky Linux

On Rocky Linux, we can use the **nmcli** command from the NetworkManager package to set a new IP address:

```
[chandan@rocky9 ~]$ nmcli connection show
```

```
NAME    UUID                                  TYPE      DEVICE
enp1s0  2bca3654-2ba5-3988-92ba-b5c7c41cab55  ethernet  enp1s0
lo      79a796db-1ff9-4595-9ea7-496252e6f7ee  loopback  lo
[chandan@rocky9 ~]$ nmcli connection modify "enp1s0" \
   ipv4.method manual \
  ipv4.addresses 192.168.122.248/24 \
  ipv4.gateway 192.168.122.1 \
  ipv4.dns "192.168.122.1"
```

You can also use the terminal-based NetworkManager tool to set a new IP address and hostname. This wizard can be started from the **nmtui** command line, as shown here:

```
[chandan@rocky9 ~]$ sudo nmtui
```

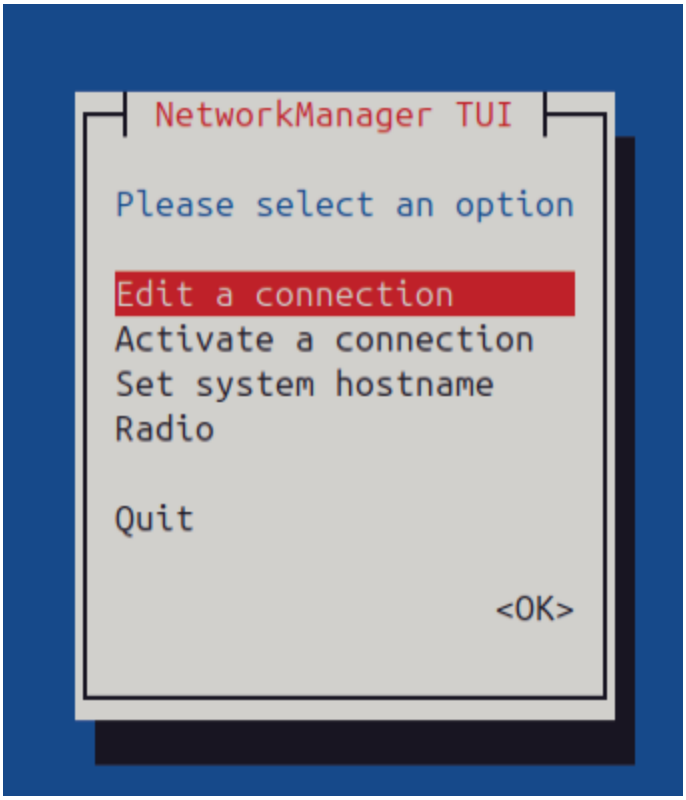


***Figure 1.4**: NetworkManager*

Proceed with **Edit** a connection and select the interface in the next screen, as shown in the following figure:

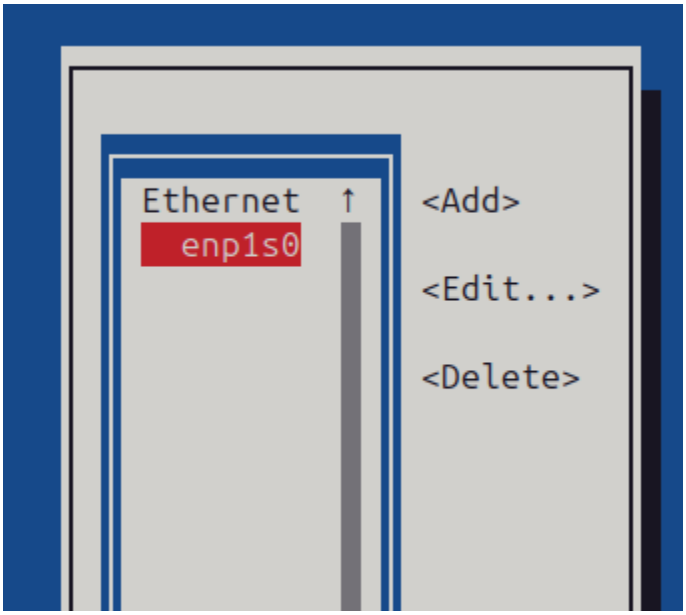


***Figure 1.5**: Select connection*

Once selected, you can enter the values, such as IP address, gateway, and DNS servers, as shown in the following figure:

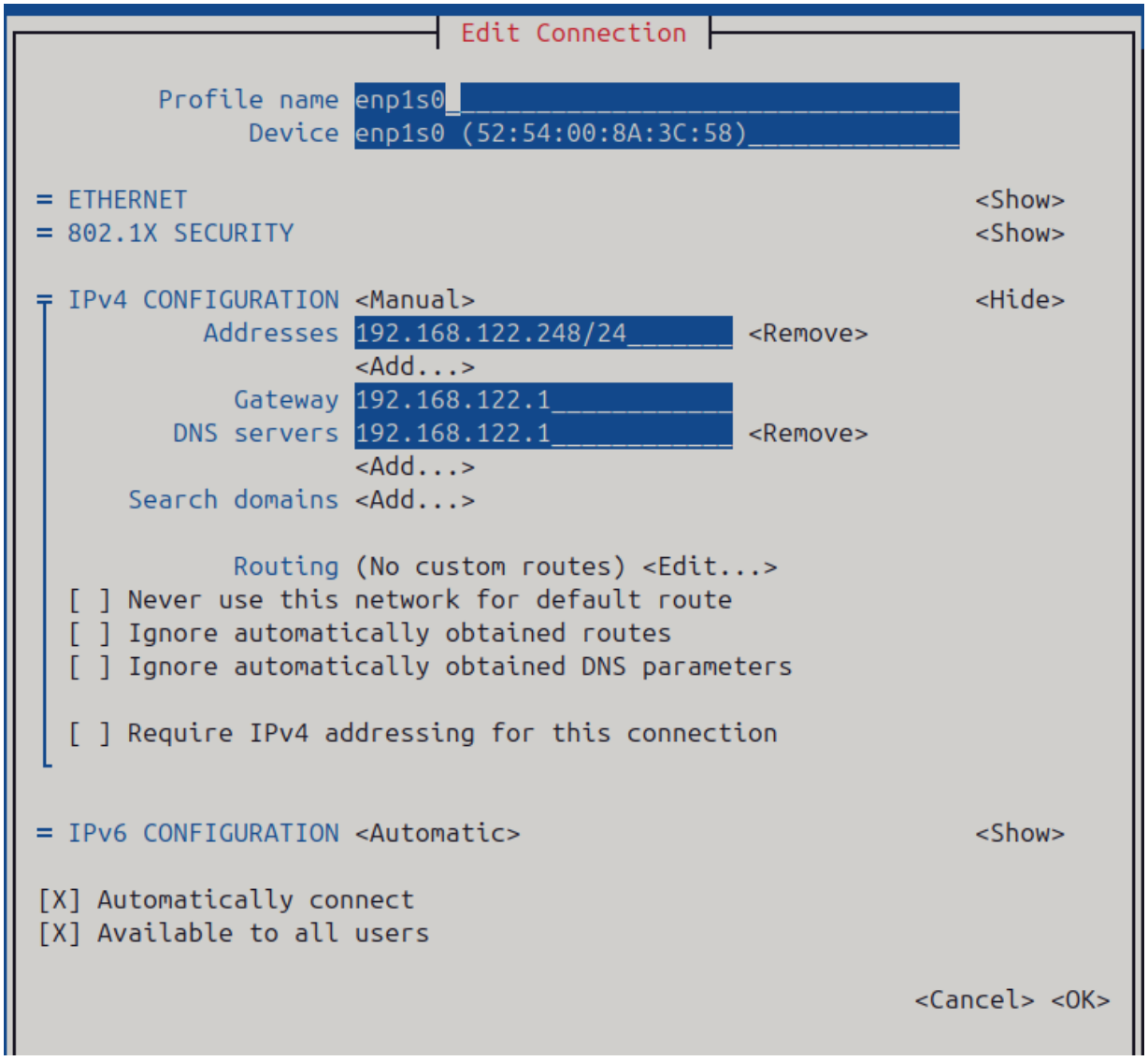


***Figure 1.6**: Edit connection*

Configuring the network on Ubuntu

On Ubuntu systems, the network configuration is done using **systemd-networkd**, but instead of directly working with the **networkd-systemd** configuration files, Ubuntu users use **netplan**, which is a higher-level tool to configure the network.

We can use the **netplan** command from the package of the same name to set the new IP Address. The configuration for the network is provided as a **.yaml** file to **netplan**, which gets converted to the **systemd-networkd** configuration. **netplan** can handle multiple YAML files, which are merged

to generate the final configuration. The configurations for **netplan** are placed in the **/etc/netplan/** directory, while the generated **systemd-networkd** configurations are placed in the **/run/systemd/network/**.

Let us configure the interface **enp1s0** on the Ubuntu host by writing the following content to a new file **/etc/netplan/10-custom.yaml**. You can find the network interfaces available on your machine using the **ip link show** command as discussed in the preceding section.

Note: netplan will complain about the permission of this file if it is readable by everyone. This can be fixed by restricting the read and write permissions of this file to the root user only.

```
$ sudo cat /etc/netplan/10-custom.yaml
network:
  version: 2
  renderer: networkd
  ethernets:
    enp1s0:
      addresses:
        - 192.168.122.61/24
      routes:
        - to: default
          via: 192.168.122.1
      nameservers:
        addresses: [192.168.122.1]
$ sudo chmod 600 /etc/netplan/10-custom.yaml
$ sudo netplan apply
```

Once the aforementioned configuration is applied using the **netplan apply** command, we can verify the content of the **systemd-networkd** configuration file:

```
$ sudo cat /run/systemd/network/10-netplan-enp1s0.network
[Match]
```

```
Name=enp1s0

[Network]
LinkLocalAddressing=ipv6
Address=192.168.122.61/24
DNS=192.168.122.1
[Route]
Destination=0.0.0.0/0
Gateway=192.168.122.1
```

We will discuss the IP address configuration and other details of Layer 3 configuration in the next chapter on routing and packet forwarding. This section is meant to get us started with a basic network setting on their hosts.

Examining physical and virtual interfaces

On Linux, a network interface can be either a physical interface or a virtual one. Although nowadays, while working with virtual machines, the distinction can be blurry, when referring to a physical interface, we expect the device to be backed by a physical device, even though those devices may be emulated hardware. Normally, the physical devices backing the interface are represented as a **peripheral component interconnect** (**PCI**) card on the Linux machine.

We can use the **lspci** command to list the PCI cards discovered by Linux:

```
$ lspci|grep Ethernet
01:00.0 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
```

This might be the right time to discuss the Linux naming convention for **network interface card** (**NIC**). Historically, the network interfaces on Linux were named as outlined in the following list:

- Wired interfaces were named as **ethN**, where N was a number to indicate the interface index, e.g., a machine with 4 interfaces had the interfaces named as **eth0**, **eth1**, **eth2**, and **eth3**.

- Wireless interfaces were named as **wlanN**, where N was a number to indicate the interface index, e.g., a machine with 2 wireless interfaces had the wireless interfaces named as **wlan0** and **wlan1**.

This approach had some limitations, like the interface names could get changed if the machine was reconfigured by adding or removing NICs. As the index of the network interface was determined by the order in which the NIC hardware was detected by Linux, e.g. if a NIC was added to a machine and Linux detected it as the first NIC the newly added card will be named eth0 and the names of the existing NICs will be changed by adding 1 to the interface index such the pre-existing **eth0** will now be called eth1. This could easily cause confusion for the user.

In modern Linux distributions, network interface names follow the persistent naming convention, which provides predictable naming (e.g., **enp3s0**, **eno0**). These names can reflect the physical location of the NIC, such as PCI bus addresses, making it easier to identify the underlying hardware. For example, the wired interface has the following naming convention:

en[prefix][location]

The term **en** indicates the interface is a wired network card. The prefix can be one of the following:

- **pXsY**, which indicates that the NIC is connected through a PCI interface on bus number X and slot Y. E.g., **enp3s0** is a wired Ethernet card on PCI bus 3 and slot 0.
- **oX**, which indicates the NIC is an on-board device on the motherboard. E.g., **eno1** is the first onboard wired NIC Ethernet card.
- **sX**, which indicates that the interface is connected using a hotplug slot. E.g., **ens1** is an Ethernet card connected through hotplug slot 1.

The wireless interfaces follow a similar naming convention as follows:

wl[prefix][location]

In this format, the term **wl** indicates that the NIC card is a wireless interface. The prefix and location follow the same convention as described previously. E.g., **wlp2s0** is a wireless NIC connected using PCI bus 2 and slot 0, whereas wlo1 is the first wireless onboard NIC.

Use the following **ethtool** command to check the properties of the interface:

```
$ sudo ethtool -i enp1s0
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:01:00.0
supports-statistics: yes
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
```

Not all properties are supported by all network interfaces. For example, on my personal laptop, the Wi-Fi interface only reports the following:

```
$ sudo ethtool wlp0s20f3
Settings for wlp0s20f3:
Link detected: yes
```

Various statistics about the traffic handled by the interface can be checked using the **-S** option to **ethtool**:

```
$ sudo ethtool -S enp1s0
NIC statistics:
rx_queue_0_packets: 165516
rx_queue_0_bytes: 230827604
rx_queue_0_drops: 0
rx_queue_0_xdp_packets: 0
rx_queue_0_xdp_tx: 0
rx_queue_0_xdp_redirects: 0
rx_queue_0_xdp_drops: 0
rx_queue_0_kicks: 4
tx_queue_0_packets: 107932
```

```
tx_queue_0_bytes: 12047124
tx_queue_0_xdp_tx: 0
tx_queue_0_xdp_tx_drops: 0
tx_queue_0_kicks: 106889
tx_queue_0_tx_timeouts: 0
```

Virtual interfaces are software-created devices that do not correspond to physical hardware but behave like network interfaces. They are normally created for special purposes. For example, the loopback interface, represented as **lo**, which is present on all Linux hosts, is used for network communication local to the host. Suppose you are developing an application that connects to a database to store data. Now, during the development, you may not have an external database host; you can set up a local database on the development machine. Now the database can be accessed over the local interface. The **lo** interface is configured with IPv4 address 127.0.0.1/8 and IPv6 address ::1/128.

This address is also mapped to the hostname localhost; this can be seen in the **/etc/hosts**:

```
$ cat /etc/hosts
127.0.0.1 localhost localhost.localdomain localhost4 localhost4.localdomain4
::1 localhost localhost.localdomain localhost6 localhost6.localdomain6
```

There are other virtual interfaces that can be commonly seen on a Linux machine. The following are the most common ones:

- **VETH**: The VETH interfaces work like pipes; they are created in pairs. The data sent on one end of the VETH interface pair shows up on the other interface of the VETH pair.
- **Bridge**: The bridge interface is used to connect two or more Layer 2 interfaces. We will be taking a more detailed look at the bridge interfaces in the subsequent section of this chapter.
- **VLAN**: Virtual local area network interfaces are used to create isolated Layer 2 networks. We will discuss the VLAN on Linux in detail.

- **Bonded**: The bonded interfaces are used to provide resilience and improved bandwidth to a network link. This works by combining two or more interfaces on the host. The bonded interface is like a logical interface where the data is sent using the individual underlying interfaces. We will explore various means in which the interfaces can be combined to add fault resilience and increase the bandwidth of a network link.
- **Dummy**: The dummy interfaces are used for testing.

Using VETH interfaces

VETH interfaces, if you recall, are created in pairs and work like a pipe. Data sent on one interface of a VETH pair will be received on the other interface of the pair.

To create a VETH pair, we can use the following command, also illustrated in the following figure:

```
$ sudo ip link add veth0 type veth peer name veth1
$ ip link show veth0
$ ip link show veth1
```

```
[chandan@rocky9 ~]$ ip link show veth0
5: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 62:c2:80:cd:c9:1c brd ff:ff:ff:ff:ff:ff
[chandan@rocky9 ~]$ ip link show veth1
4: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 7e:88:2e:e3:3d:39 brd ff:ff:ff:ff:ff:ff
[chandan@rocky9 ~]$
```

***Figure 1.7**: VETH interface status*

In the aforementioned listing, you can see that the state of these interfaces is **DOWN**. To use the VETH pair, you should set its status to **UP**. This can be done using the following command:

```
$ sudo ip link set veth0 up
$ sudo ip link set veth1 up
```

Let us set an IP address on the interfaces and check the packet flow using the following command:

```
$ sudo ip addr add dev veth0 10.0.10.1/24
$ sudo ip addr add dev veth1 10.0.10.2/24
```

We need one more setting to accept packets from local interfaces that are not a loopback interface. This can be configured using the following command:

```
$ echo 1|sudo tee /proc/sys/net/ipv4/conf/all/accept_local
```

Now we can run the **ping** command to check the connectivity:

```
$ ping -c3 -I veth0 10.0.10.2
```

The preceding **ping** command will not succeed as **veth0** and **veth1** are the two ends of the same network interface, but we can check that the packets are being received by the other end of the **veth** pair. Running **tcpdump** on the other end of the VETH pair, we can verify the packet flow using the following command:

```
$ sudo tcpdump –n –i veth1
```

The **tcpdump** command is used to check traffic on the network. We have used the **–i** parameter to select the network interface and the **–n** parameter to get numeric output. We will discuss the **tcpdump** command in detail in *Chapter 4*, *Network Troubleshooting and Diagnostics*. You can notice the **Address Resolution Protocol** (**ARP**) and **Internet Control Message Protocol** (**ICMP**) requests produced by the **ping** command received on the interface **veth1**, as shown in the following figure:

```
chandan@ubuntu24:~$ sudo tcpdump -n -i veth1
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on veth1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
17:12:35.550050 LLDP, length 219: ubuntu24
17:12:35.550064 LLDP, length 219: ubuntu24
17:12:38.105332 ARP, Request who-has 10.0.10.2 tell 10.0.10.1, length 28
17:12:38.105359 ARP, Reply 10.0.10.2 is-at fa:0d:2d:24:57:21, length 28
17:12:38.105362 IP 10.0.10.1 > 10.0.10.2: ICMP echo request, id 1973, seq 1, length 64
17:12:39.135883 IP 10.0.10.1 > 10.0.10.2: ICMP echo request, id 1973, seq 2, length 64
17:12:40.160015 IP 10.0.10.1 > 10.0.10.2: ICMP echo request, id 1973, seq 3, length 64
^C
7 packets captured
7 packets received by filter
0 packets dropped by kernel
chandan@ubuntu24:~$
```

***Figure 1.8**: tcpdump output for VETH interface*

Bridging Layer 2 networks

In the preceding section, we discussed the common virtual network interfaces in Linux, where we touched upon the bridge network interface. A bridge is a software-based Linux network device that can be used to connect Layer 2 interfaces. It provides connectivity between Layer 2 interfaces by forwarding packets between them. A Linux bridge can be viewed as a software-based switch.

To learn more about a Linux bridge, let us set up a lab to create a bridge and understand how it works.

To create a Linux bridge, we can use the **ip** command:

```
$ sudo ip link add dev br0 type bridge
$ ip link show dev br0
```

Now we can create 2 interfaces to connect to the bridge and check the connectivity. For this, we will create 2 VETH pairs representing 2 hosts connected to the bridge, as shown in the following figure and command:

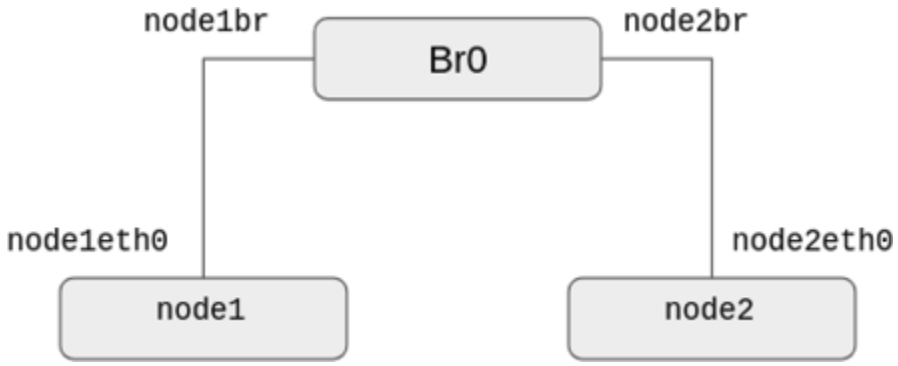


***Figure 1.9**: Setup for bridging*

```
$ sudo ip link add node1eth0 type veth peer name node1br
$ sudo ip link add node2eth0 type veth peer name node2br
```

We will set the status of the interfaces **up**:

```
$ sudo ip link set node1eth0 up
$ sudo ip link set node2eth0 up
$ sudo ip link set node1br up
$ sudo ip link set node2br up
```

```
$ sudo ip link set br0 up
```

We then add one of the interfaces in the VETH pair to the bridge:

```
$ sudo ip link set node1br master br0
$ sudo ip link set node2br master br0
```

Now we can configure the interfaces with IP addresses:

```
$ sudo ip addr add dev node1eth0 10.20.0.1/24
$ sudo ip addr add dev node2eth0 10.20.0.2/24
```

We need one more setting to make the bridge accept packets from local interfaces that are not a loopback interface. This can be configured using the following command:

```
$ echo 1|sudo tee /proc/sys/net/ipv4/conf/all/accept_local
```

Now we can check the connectivity of the interfaces:

```
$ ping -c3 -I node1eth0 10.20.0.2
```

The aforementioned setup shows how a bridge can connect multiple interfaces in a Layer 2 network and provide connectivity between them.

VLAN networks

In the preceding section, we discussed the bridge interface, which can connect multiple interfaces to a Layer 2 network. VLANs provide a way to split a single Layer 2 network into multiple isolated Layer 2 networks without needing any extra virtual or physical switch. VLANs work by adding a tag in the Ethernet frame, as shown in the following figure:

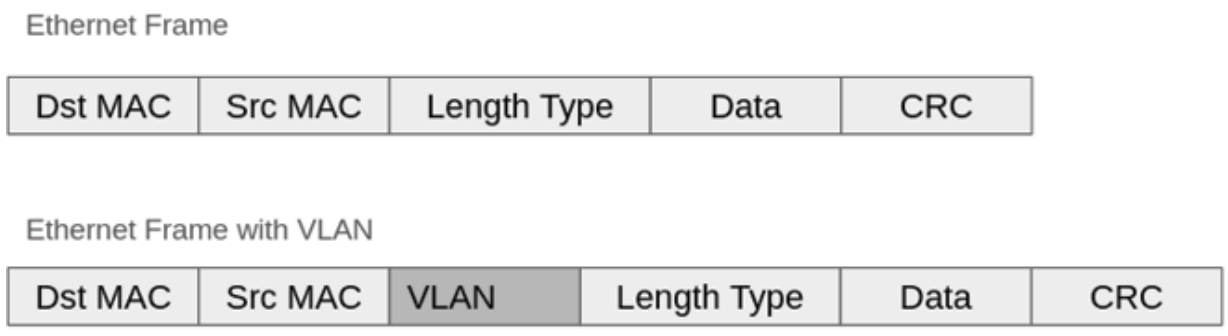


***Figure 1.10**: Ethernet frame with VLAN*

The VLAN tag contains the VLAN ID to identify the logical Layer 2 network the packet belongs to. The VLAN ID field has a length of 12 bits, so VLAN networks are restricted to a maximum count of 4096.

In a VLAN network, a connected interface can be:

- **Access interface**: An access interface will inject a VLAN tag to packets originating from it. It will reject any packets received if it does not contain a VLAN tag or the VLAN tag of the received packet does not match the VLAN ID of the access interface.
- **Trunk interface**: A trunk interface will accept packets of all VLAN IDs. These interfaces are used to carry packets from multiple VLANs and are normally used to connect switches with access ports for multiple VLANS.

In Linux, a VLAN access interface can be created at the network interface level. To do this, use the following command:

```
$ sudo ip link add link eth0 name eth0.10 type vlan id 10
```

Note: Use the ip link show command to find the network interface on your machine. Please replace eth0 with the interface to which you want to add a VLAN sub interface.

This command adds a sub-interface **eth0.10** for **eth0**, which adds a VLAN tag to network frames sent using this interface. Let us set up a lab to test the VLAN interface.

First, let us create a bridge:

```
$ sudo ip link add br0 type bridge
```

Create a Node3 interface and a VLAN 10 sub-interface for this:

```
$ sudo ip link add node3eth0 type veth peer name node3br
$ sudo ip link set node3eth0 up
$ sudo ip link set node3br up
$ sudo ip link set node3br master br0
$ sudo ip link add link node3eth0 name node3eth0.10 type vlan id 10
$ sudo ip link set node3eth0.10 up
$ sudo ip addr add dev node3eth0.10 10.10.0.3/24
```

Create a Node4 interface and a VLAN 10 sub-interface for this:

```
$ sudo ip link add node4eth0 type veth peer name node4br
$ sudo ip link set node4br up
$ sudo ip link set node4eth0 up
$ sudo ip link set node4br master br0
$ sudo ip link add link node4eth0 name node4eth0.10 type vlan id 10
$ sudo ip link set node4eth0.10 up
$ sudo ip addr add dev node4eth0.10 10.10.0.4/24
$ sudo ip link set br0 up
```

We also need to allow accepting local packets from interfaces other than the loopback interface:

```
$ echo 1|sudo tee /proc/sys/net/ipv4/conf/all/accept_local
```

Testing the VLAN interface:

```
$ ping -c3 -I node4eth0.10 10.10.0.3
```

To verify that the packets are indeed tagged with the VLAN ID, we can do a packet capture using **tcpdump**:

```
$ sudo tcpdump -n -i node4eth0 -e
```

In the following figure, we can see that the packets captured are carrying the VLAN tag 10:

```
chandan@ubuntu24:~$ sudo tcpdump -n -i node4eth0 -e
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on node4eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
04:01:00.607268 e6:d6:ea:ac:7d:32 > de:8f:9e:82:54:74, ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4
(0x0800), 10.10.0.4 > 10.10.0.3: ICMP echo request, id 1890, seq 1, length 64
04:01:01.657065 e6:d6:ea:ac:7d:32 > de:8f:9e:82:54:74, ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4
(0x0800), 10.10.0.4 > 10.10.0.3: ICMP echo request, id 1890, seq 2, length 64
04:01:02.681048 e6:d6:ea:ac:7d:32 > de:8f:9e:82:54:74, ethertype 802.1Q (0x8100), length 102: vlan 10, p 0, ethertype IPv4
(0x0800), 10.10.0.4 > 10.10.0.3: ICMP echo request, id 1890, seq 3, length 64
^C
3 packets captured
3 packets received by filter
0 packets dropped by kernel
chandan@ubuntu24:~$
```

***Figure 1.11**: tcpdump for ping with VLAN interface*

Link aggregation and bonding

Link aggregation (also called **bonding**, **teaming**, or **NIC bonding**) is a method of combining multiple network interfaces into a single logical interface to provide:

- **High availability**: Automatic failover if a network connection drops.
- **Load balancing**: Distributes traffic across multiple physical links.
- **Scalability**: Ability to add more interfaces as bandwidth needs grow.
- **Seamless operation**: Maintains a single logical interface despite using multiple physical links.

This technology is particularly valuable in enterprise environments, data centers, and any scenario where network reliability and performance are critical.

To configure bonding in Linux, we first need to load the bonding driver using the following command:

```
$ sudo modprobe bonding
```

Next, we will create multiple interfaces and combine them to create a single logical interface:

```
$ sudo ip link add dev br0 type bridge
$ sudo ip link add bond1eth0 type veth peer name bond1br
$ sudo ip link add bond2eth0 type veth peer name bond2br
$ sudo ip link set bond1br up
$ sudo ip link set bond2br up
$ sudo ip link set bond1br master br0
$ sudo ip link set bond2br master br0
$ sudo ip link add bond0 type bond
$ sudo ip link set bond1eth0 master bond0
$ sudo ip link set bond2eth0 master bond0
```

The interfaces that are part of the bond should only be brought up after making them part of the bonded interface:

```
$ sudo ip link set bond1eth0 up
$ sudo ip link set bond2eth0 up
$ sudo ip link set bond0 up
```

Check the following status of the boned interface, also illustrated in the following figure:

$ cat /proc/net/bonding/bond0

```
chandan@ubuntu24:~$ cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v6.8.0-60-generic

Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0
Peer Notification Delay (ms): 0

Slave Interface: bond1eth0
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: c6:9b:33:71:84:e9
Slave queue ID: 0

Slave Interface: bond2eth0
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 5e:3b:12:38:30:e2
Slave queue ID: 0
chandan@ubuntu24:~$
```

***Figure 1.12**: Round-robin bonding*

Let us check the interfaces connected to the bridge, also illustrated in the following figure:

$ brctl show

```
chandan@ubuntu24:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.72cd3764ab48       no              bond1br
                                                        bond2br

chandan@ubuntu24:~$
```

***Figure 1.13**: Bridge setup with peer interfaces of bond slaves*

We will set up another VETH interface to add a node to this bridge, as shown in the following figure and the command:

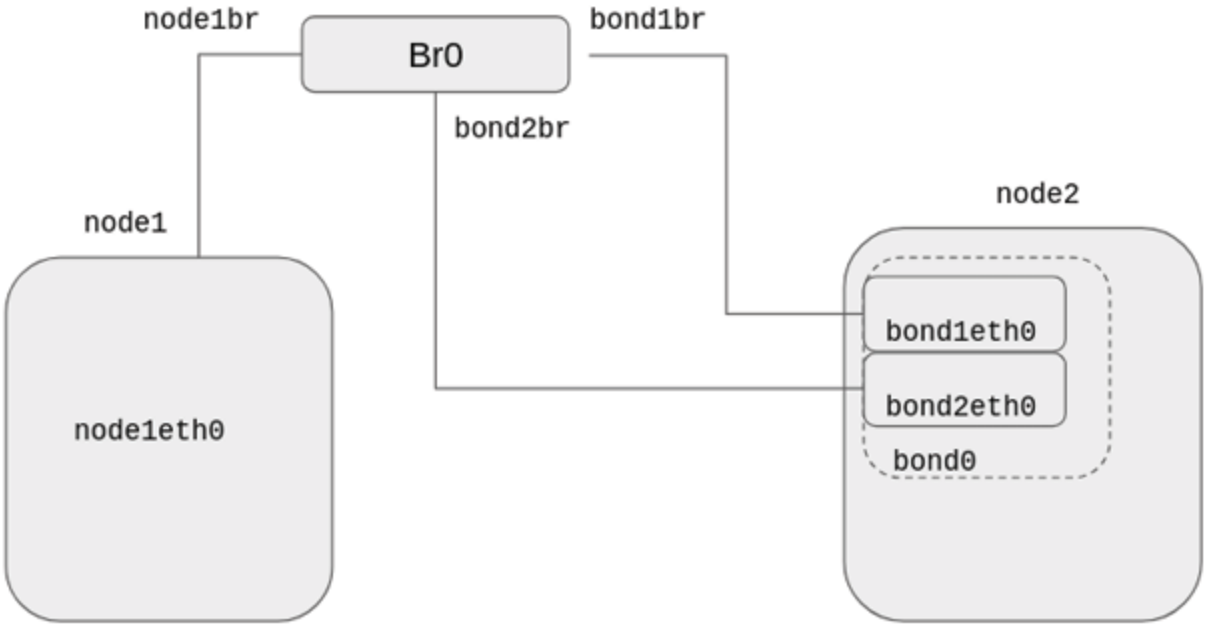


***Figure 1.14**: Setup for link aggregation*

$ sudo ip link add node1eth0 type veth peer name node1br
$ sudo ip link set node1br master br0
$ sudo ip link set node1eth0 up
$ sudo ip link set node1br up
$ sudo ip addr add dev node1eth0 10.20.0.1/24
$ sudo ip addr add dev bond0 10.20.0.2/24

The following figure illustrates a bridge setup:

```
chandan@ubuntu24:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
br0             8000.72cd3764ab48       no              bond1br
                                                        bond2br
                                                        node1br

chandan@ubuntu24:~$
```

***Figure 1.15**: Bridge setup with peer interfaces of bond slaves and node interface*

Now we can check the connectivity of the **bond0** interface, as shown here:

```
$ ping -c 3 -I bond0 10.20.0.1
```

We can verify the packets are using the current active interface in the **bond0** by running **tcpdump**, also illustrated in the following figure:

```
$ sudo tcpdump  -i bond2eth0 -i bond1eth0 -e
```

```
chandan@ubuntu24:~$ sudo tcpdump  -i bond2eth0 -i bond1eth0 -e
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on bond1eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
09:48:16.941160 6e:27:be:66:3f:bc (oui Unknown) > ae:4a:2c:aa:7f:9c (oui Unknown), ethertype IPv4 (0x0800), length 98:
 ubuntu24 > ubuntu24: ICMP echo request, id 1695, seq 1, length 64
09:48:18.988471 6e:27:be:66:3f:bc (oui Unknown) > ae:4a:2c:aa:7f:9c (oui Unknown), ethertype IPv4 (0x0800), length 98:
 ubuntu24 > ubuntu24: ICMP echo request, id 1695, seq 3, length 64
09:48:22.124610 ae:4a:2c:aa:7f:9c (oui Unknown) > 33:33:00:00:00:02 (oui Unknown), ethertype IPv6 (0x86dd), length 70:
 ubuntu24 > ip6-allrouters: ICMP6, router solicitation, length 16
09:49:11.276359 c2:01:fd:9a:82:c4 (oui Unknown) > 33:33:00:00:00:02 (oui Unknown), ethertype IPv6 (0x86dd), length 70:
 ubuntu24 > ip6-allrouters: ICMP6, router solicitation, length 16
```

***Figure 1.16**: tcpdump for ping from bonded interface*

To simulate a link failure, we can remove one of the interfaces from the **bond0** interface, also illustrated in *Figure 1.17*:

```
$ sudo ip link set bond1eth0 nomaster
$ cat /proc/net/bonding/bond0
```

```
chandan@ubuntu24:~$ cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v6.8.0-60-generic

Bonding Mode: load balancing (round-robin)
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0
Peer Notification Delay (ms): 0

Slave Interface: bond2eth0
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 5e:3b:12:38:30:e2
Slave queue ID: 0
chandan@ubuntu24:~$
```

***Figure 1.17**: Bond interface with one interface removed*

The connectivity should still be up:

$ ping -c 3 -I bond0 10.20.0.1

The bond we created in this example works in the round-robin mode. A bonded interface in Linux can be configured in different modes, as follows:

Mode	Name	Description	Switch support required
0	balance-rr	Round-robin packet distribution	Yes
1	active backup	Failover only	No
2	balance-xor	XOR of source/destination MAC	Yes
4	802.3ad	Dynamic LACP aggregation	Yes
5	balance-tlb	Adaptive transmit load balancing	No
6	balance-alb	Adaptive load balancing	No

***Table 1.1**: LACP configuration modes*

Bonding with LACP

The most common bonding mode is the one using **Link Aggregation Control Protocol** (**LACP**). Use the following command to configure the bond interface with LACP using the **802.3ad**:

```
$ sudo ip link add bond0 type bond mode 802.3ad
```

Using LACP for creating a bond requires support from the switch. The Linux bridge that we have been using for all the lab setup does not support LACP, but we can set up an LACP bond with OpenVSwitch, which is an advanced software-based switch for Linux.

To install OpenVSwitch on Linux, we can use the following command:

```
$ sudo apt install openvswitch-switch
```

To test LACP with OpenVSwitch, we will use the following setup:

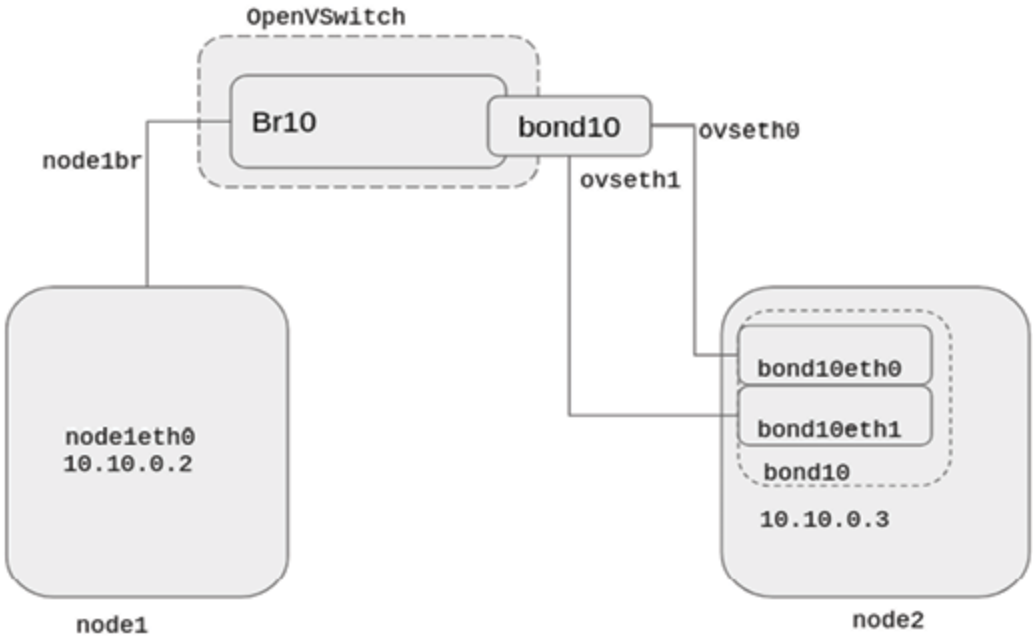


***Figure 1.18**: Setup for bonding with LACP*

Let us set up the previously mentioned topology. The following commands set up VETH pairs to form the switch and client-side bond:

```
sudo ip link add bond10eth0 type veth peer name ovseth0
sudo ip link add bond10eth1 type veth peer name ovseth1
```

The following commands set up VETH pairs for a test interface for **node1** and assign an IP address to the interface:

```
sudo ip link add node1-eth0 type veth peer name node1br
sudo ip link set node1-eth0 up
sudo ip link set node1br up
sudo ip addr add dev node1-eth0 10.10.0.2/24
```

The following commands set up the bond interface **bond10** on the client side and set an IP address on the client side:

```
sudo ip link add client-bond10 type bond mode 802.3ad
sudo ip link set bond10eth0 master client-bond10
sudo ip link set bond10eth1 master client-bond10
sudo ip link set bond10eth0 up
sudo ip link set bond10eth1 up
sudo ip link set client-bond10 up
sudo ip addr add dev client-bond10 10.10.0.3/24
```

The following commands set up the bond interface **bond10** on the switch side:

```
sudo ip link set ovseth0 up
sudo ip link set ovseth1 up
sudo ovs-vsctl add-br br10
sudo ovs-vsctl add-port br10 node1br
sudo ovs-vsctl add-bond br10 bond10 ovseth0 ovseth1 lacp=active
sudo ovs-vsctl set port bond10 lacp_system_priority=1000
sudo ovs-vsctl set port ovseth0 lacp_port_priority=10
sudo ovs-vsctl set port ovseth1 lacp_port_priority=20
sudo ovs-vsctl set port bond10 bond_mode=balance-tcp
sudo ip link set br10 up
```

The following commands can be used to inspect the status of the LACP bond on OpenVSwitch:

```
sudo ovs-vsctl show
sudo ovs-appctl lacp/show
sudo ovs-appctl bond/show
```

```
echo 1|sudo tee /proc/sys/net/ipv4/conf/all/accept_local
```

We can check the packet flow on the bonded interface with the following command:

```
$ sudo tcpdump -v -n -c3 -i client-bond10
```

While running the **ping** command, the same is illustrated in *Figure 1.19*:

```
$ ping -c3 -I node1-eth0 10.10.0.3
```

```
chandan@ubuntu24:~$ sudo tcpdump -v -n -c3 -i client-bond10
tcpdump: listening on client-bond10, link-type EN10MB (Ethernet), snapshot length 262144 bytes
10:06:13.986228 IP (tos 0x0, ttl 64, id 30591, offset 0, flags [DF], proto ICMP (1), length 84)
    10.10.0.2 > 10.10.0.3: ICMP echo request, id 1660, seq 1, length 64
10:06:14.985722 IP (tos 0x0, ttl 64, id 31009, offset 0, flags [DF], proto ICMP (1), length 84)
    10.10.0.2 > 10.10.0.3: ICMP echo request, id 1660, seq 2, length 64
10:06:16.032087 IP (tos 0x0, ttl 64, id 31624, offset 0, flags [DF], proto ICMP (1), length 84)
    10.10.0.2 > 10.10.0.3: ICMP echo request, id 1660, seq 3, length 64
3 packets captured
3 packets received by filter
0 packets dropped by kernel
chandan@ubuntu24:~$
```

***Figure 1.19**: tcpdump for ping with bonded interface using LACP*

Understanding Layer 2 protocols

In this section, we will examine some of the protocols that make the Layer 2 networks function. We will discuss the ARP protocol, which is used to resolve an IP address to a Layer 2 MAC address. We have already discussed link aggregation in the previous section; this provides a way to add fault tolerance to the host connecting to the network by utilizing multiple network interfaces. Now we will look at how to add similar capability to a network switch, but adding redundant links, and how the **Spanning Tree Protocol** (**STP**) enables us to add alternate links to the network switch without causing a loop in the network. We will also discuss the **Link Layer Discovery Protocol** (**LLDP**), which is used to discover the nodes and their capabilities connected in a Layer 2 network.

Address resolution in Layer 2 networks

In this section, we will learn about address resolution at the network Layer

2. Let us take an example of two hosts connected to a network. To communicate between the hosts, they should know each other's addresses. We normally address the node with its IP. The simplest example of this is the **ping** command, which can be used to check the connectivity between two nodes. A **ping** command works by sending an ICMP echo packet to the destination node, while the destination node responds by sending back the ICMP response. *How does the packet know which node on the network has this IP?* The ARP protocol is responsible for resolving an IP address to the corresponding MAC or hardware address of the network interface of the destination host.

Let us understand this by checking how the ARP protocol resolves the IP address of a host to the MAC address by running the **ping** command.

In my setup, we have two VMs connected to the network, and they have the following addresses:

- Ubuntu host has the IP of 192.168.122.61
- RockyLinux host has the IP of 192.168.122.248

To check what ARP does when we try to resolve the IP address to the MAC address lets run **tcpdump** and **ping**:

$ ping –c3 192.168.122.248

The following **tcpdump** was captured while running the **ping** command. As you can see, when the **ping** command is run, the source machine sends the ARP query to learn which machine on the network has the IP address of **192.168.122.248**.

The following is seen as the request:

Request who-has 192.168.122.248 tell 192.168.122.61

The following response is then received from the destination machine:

Reply 192.168.122.248 is-at 52:54:00:8a:3c:58

```
chandan@ubuntu24:~$ sudo tcpdump -n arp
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on enp1s0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
15:28:30.310542 ARP, Request who-has 192.168.122.248 tell 192.168.122.61, length 28
15:28:30.310905 ARP, Reply 192.168.122.248 is-at 52:54:00:8a:3c:58, length 28
15:28:35.668386 ARP, Request who-has 192.168.122.61 tell 192.168.122.248, length 28
15:28:35.668424 ARP, Reply 192.168.122.61 is-at 52:54:00:ee:6a:84, length 28
```

***Figure 1.20**: ARP request and response*

The ARP response contains the MAC address of the destination IP address. This response is then cached by the source node for some time so that every time a packet needs to go to this IP address, the ARP query need not be performed.

This cache can be checked with the **arp** command:

```
chandan@ubuntu24:~$ arp -n
Address                  HWtype  HWaddress           Flags Mask            Iface
192.168.122.1            ether   52:54:00:42:5c:1a   C                     enp1s0
192.168.122.248          ether   52:54:00:8a:3c:58   C                     enp1s0
```

If you run ping a second time for this IP address, you will not see the ARP query being sent on the network.

The ARP cache is cleared after a certain period. This is required to prevent the ARP table from growing out of limits. On Linux, the ARP entry caching times are controlled by the following settings:

```
/proc/sys/net/ipv4/neigh/default/gc_stale_time
/proc/sys/net/ipv4/neigh/default/base_reachable_time
```

The **gc_stale_time** is the amount of time after which Linux should check for stale entries in the ARP cache, while **base_reachable_time** determines the amount of time an ARP entry is considered valid by Linux. This time can range between *base_reachable_time/2* to *3*base_reachable_time/2*.

To check the currently configured values for the parameters, use the following commands:

```
$ cat /proc/sys/net/ipv4/neigh/default/gc_stale_time
$ cat /proc/sys/net/ipv4/neigh/default/base_reachable_time
```

These values can be set as the default or per interface using the following command:

```
$ echo 30 |sudo tee /proc/sys/net/ipv4/neigh/default/gc_stale_time
```

To set the values for an interface like **eth0**, we can use the following command:

```
$ echo 30 |sudo tee /proc/sys/net/ipv4/neigh/eth0/gc_stale_time
```

The values set with the preceding commands are temporary and will be lost after reboot. To set these values permanently, we can use the **/etc/sysctl.conf** file.

For example, adding the following entries with set the values for the given parameters:

```
net.ipv4.neigh.default.base_reachable_time=60
net.ipv4.neigh.default.gc_stale_time=120
```

Other parameters controlling the ARP cache on Linux can be found under the same path in
/proc:

```
$ find /proc/sys/net/ipv4/neigh/default/
/proc/sys/net/ipv4/neigh/default/
/proc/sys/net/ipv4/neigh/default/anycast_delay
/proc/sys/net/ipv4/neigh/default/app_solicit
/proc/sys/net/ipv4/neigh/default/base_reachable_time
/proc/sys/net/ipv4/neigh/default/base_reachable_time_ms
/proc/sys/net/ipv4/neigh/default/delay_first_probe_time /
proc/sys/net/ipv4/neigh/default/gc_interval
/proc/sys/net/ipv4/neigh/default/gc_stale_time
/proc/sys/net/ipv4/neigh/default/gc_thresh1
/proc/sys/net/ipv4/neigh/default/gc_thresh2
/proc/sys/net/ipv4/neigh/default/gc_thresh3
/proc/sys/net/ipv4/neigh/default/locktime
/proc/sys/net/ipv4/neigh/default/mcast_resolicit
/proc/sys/net/ipv4/neigh/default/mcast_solicit
```

/proc/sys/net/ipv4/neigh/default/proxy_delay
/proc/sys/net/ipv4/neigh/default/proxy_qlen
/proc/sys/net/ipv4/neigh/default/retrans_time
/proc/sys/net/ipv4/neigh/default/retrans_time_ms
/proc/sys/net/ipv4/neigh/default/ucast_solicit
/proc/sys/net/ipv4/neigh/default/unres_qlen
/proc/sys/net/ipv4/neigh/default/unres_qlen_bytes

A full description of all the entries and their functionality can be found in the man page of ARP, which can be accessed with the man **arp** command.

Fault tolerance in Layer 2 networks

STP is a network protocol designed to prevent loops in Ethernet networks. The network switches are configured with redundant links to prevent network failure when a link connecting the switch to the network fails. It is used to create a loop-free logical topology by selectively blocking redundant paths in a switched network.

STP will re-enable the blocked path if the active forwarding path fails.

In a Layer 2 network with redundant links, STP prevents broadcast storms, MAC table instability, and duplicate frames caused by network loops while still providing a fallback path in case of link failure. A broadcast storm is a network condition where an excessive number of broadcast packets flood the network, causing severe performance degradation or a complete network outage. This typically occurs due to network loops (when redundant paths are not properly managed) or misconfigured devices continuously generating broadcast traffic. A broadcast storm can cause switches to get overwhelmed, leading to high CPU usage, dropped packets, and network congestion.

So how does STP work? To understand the STP functionality, let us say we have the following setup, where three switches are connected to each other:

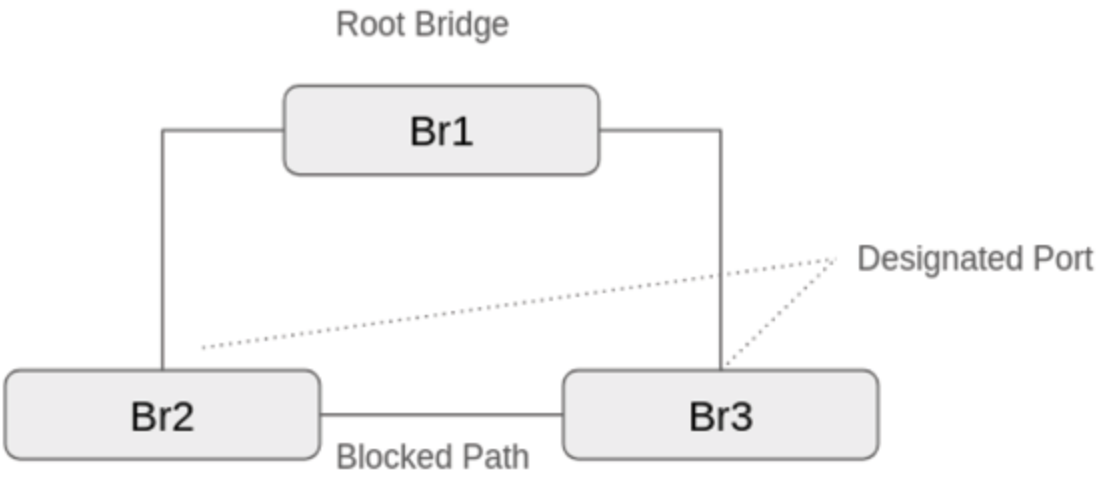


***Figure 1.21**: Setup for STP*

This topology will create a loop in the network, as a broadcast packet originating from a port on a switch can get forwarded by two different paths and can arrive back on the switch from which it originated. To prevent this, the STP protocol converts this physical topology of connected switches into a tree-like topology with one of the switches acting as the root of the tree. STP does this by electing a root bridge.

A root bridge is a switch that forms the root of this tree; it is selected based on the **Bridge ID** (**BID**) of the switch. BID is the sum of the priority of the switch, with a default value of 32768, which can be manually adjusted, and the MAC address of the switch. A switch with the lowest BID is selected as the root bridge.

The non-root switches now find the lowest cost path to the root bridge; the path that has higher bandwidth is considered lower cost. All paths to the root bridge, which are not the lowest cost paths, are blocked. The ports on the non-root bridges that connect the lowest cost path are considered designated ports or root ports.

Now that we understand the functionality of STP, let us set up a lab on a Linux host and verify our understanding.

Let us set up three bridges:

```
sudo ip link add name br1 type bridge
sudo ip link add name br2 type bridge
```

```
sudo ip link add name br3 type bridge
```

We will next enable STP on the bridges:

```
sudo ip link set br1 type bridge stp_state 1
sudo ip link set br2 type bridge stp_state 1
sudo ip link set br3 type bridge stp_state 1
```

Next, we will create links to connect the bridges with VETH pairs:

```
sudo ip link add veth1-br1 type veth peer name veth1-br2
sudo ip link add veth2-br2 type veth peer name veth2-br3
sudo ip link add veth3-br3 type veth peer name veth3-br1
```

The following command will attach the links to the bridges:

```
sudo ip link set veth1-br1 master br1
sudo ip link set veth1-br2 master br2
sudo ip link set veth2-br2 master br2
sudo ip link set veth2-br3 master br3
sudo ip link set veth3-br3 master br3
sudo ip link set veth3-br1 master br1
```

Let us bring up all the interfaces:

```
sudo ip link set br1 up
sudo ip link set br2 up
sudo ip link set br3 up
sudo ip link set veth1-br1 up
sudo ip link set veth1-br2 up
sudo ip link set veth2-br2 up
sudo ip link set veth2-br3 up
sudo ip link set veth3-br3 up
sudo ip link set veth3-br1 up
```

Now we can check the status of the bridges with the **brctl show** command, as shown in the following figure:

```
chandan@ubuntu24:~$ brctl show
bridge name     bridge id               STP enabled     interfaces
br1             8000.1e98abff5791       yes             veth1-br1
                                                        veth3-br1
br2             8000.ae79f0aee49a       yes             veth1-br2
                                                        veth2-br2
br3             8000.72e264e2f741       yes             veth2-br3
                                                        veth3-br3
chandan@ubuntu24:~$
```

***Figure 1.22**: Bridge setup with redundant path*

The individual bridge can be checked for the STP status using the **brctl showstp** command.

For example, for the **br1** command, **brctl showstp br1** shows the following output:

```
br1
 bridge id               8000.1e98abff5791
 designated root         8000.1e98abff5791
 root port                  0                    path cost                  0
 max age                   20.00                 bridge max age            20.0
 hello time                 2.00                 bridge hello time          2.0
 forward delay             15.00                 bridge forward delay      15.0
 ageing time               30.00
 hello timer                1.52                 tcn timer                  0.0
 topology change timer      5.16                 gc timer                 239.6
 flags                   TOPOLOGY_CHANGE TOPOLOGY_CHANGE_DETECTED


veth1-br1 (1)
 port id                 8001                    state                forwardin
 designated root         8000.1e98abff5791       path cost                    2
 designated bridge       8000.1e98abff5791       message age timer          0.0
 designated port         8001                    forward delay timer        0.0
 designated cost            0                    hold timer                 0.5
 flags

veth3-br1 (2)
 port id                 8002                    state                forwardin
 designated root         8000.1e98abff5791       path cost                    2
 designated bridge       8000.1e98abff5791       message age timer          0.0
 designated port         8002                    forward delay timer        0.0
 designated cost            0                    hold timer                 0.5
 flags

chandan@ubuntu24:~$
```

***Figure 1.23**: Bridges in forwarding state*

While for a non-root bridge, we can see that one of the links is blocking:

```
chandan@ubuntu24:~$ brctl showstp br2
br2
 bridge id              8000.ae79f0aee49a
 designated root        8000.1e98abff5791
 root port                 1                    path cost                  2
 max age                  20.00                 bridge max age            20.00
 hello time                2.00                 bridge hello time          2.00
 forward delay            15.00                 bridge forward delay      15.00
 ageing time             300.00
 hello timer               0.00                 tcn timer                  0.00
 topology change timer     0.00                 gc timer                 197.69
 flags


veth1-br2 (1)
 port id                8001                   state                forwarding
 designated root        8000.1e98abff5791      path cost                 2
 designated bridge      8000.1e98abff5791      message age timer        19.63
 designated port        8001                   forward delay timer       0.00
 designated cost           0                   hold timer                0.00
 flags

veth2-br2 (2)
 port id                8002                   state                  blocking
 designated root        8000.1e98abff5791      path cost                 2
 designated bridge      8000.72e264e2f741      message age timer        19.62
 designated port        8001                   forward delay timer       0.00
 designated cost           2                   hold timer                0.00
 flags

chandan@ubuntu24:~$
```

***Figure 1.24**: STP blocks one of the forwarding paths*

On checking the bridge link status, we find one of the links is blocked:

```
chandan@ubuntu24:~$ bridge link show|grep state
7: veth1-br2@veth1-br1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br2 state forwarding priority 32 cost 2
8: veth1-br1@veth1-br2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br1 state forwarding priority 32 cost 2
9: veth2-br3@veth2-br2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br3 state forwarding priority 32 cost 2
10: veth2-br2@veth2-br3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br2 state blocking priority 32 cost 2
11: veth3-br1@veth3-br3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br1 state forwarding priority 32 cost 2
12: veth3-br3@veth3-br1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master br3 state forwarding priority 32 cost 2
chandan@ubuntu24:~$
```

***Figure 1.25**: STP blocks forwarding on one of the redundant links*

Now, let us test what happens if one of the links connecting the bridge fails. This can be simulated by bringing one of the links down. For example, let us bring down the **veth1-br1**.

This will remove the root path connecting the bridge **br2** to the root bridge **br1**:

```
$ sudo ip link set veth1-br1 down
```

If you check the details of bridge **br1** now, you will see that the bridge port **veth1-br2**, which was in the forwarding state, is now in the disabled state. If you continue to monitor the state of the bridge **br1**, you will see that the port **veth2-br2** now changes state from blocking to listening.

The following is the output:

```
veth1-br2 (1)
 port id               8001                    state                disabled
 designated root       8000.1e98abff5791       path cost                2
 designated bridge     8000.ae79f0aee49a       message age timer        0.00
 designated port       8001                    forward delay timer      0.00
 designated cost          2                    hold timer               0.00
 flags

veth2-br2 (2)
 port id               8002                    state                listening
 designated root       8000.1e98abff5791       path cost                2
 designated bridge     8000.72e264e2f741       message age timer       19.56
 designated port       8001                    forward delay timer      0.23
 designated cost          2                    hold timer               0.00
 flags

chandan@ubuntu24:~$
```

***Figure 1.26**: STP detects link failure and moves the redundant path to the listening state*

In some time, the state will change to learning, and finally it will be in the forward state:

```
veth1-br2 (1)
 port id               8001                    state                 disabled
 designated root       8000.1e98abff5791       path cost                2
 designated bridge     8000.ae79f0aee49a       message age timer        0.00
 designated port       8001                    forward delay timer      0.00
 designated cost          2                    hold timer               0.00
 flags

veth2-br2 (2)
 port id               8002                    state                 forwarding
 designated root       8000.1e98abff5791       path cost                2
 designated bridge     8000.72e264e2f741       message age timer       19.03
 designated port       8001                    forward delay timer      0.00
 designated cost          2                    hold timer               0.00
 flags
```

***Figure 1.27**: STP moves redundant path to forwarding state*

The STP protocol will recalculate the topology when the link fails and ensures the network is still connected by utilizing the redundant links when the link providing the root path fails.

Neighbour discovery

This section will dive into the LLDP protocol, which can be used to discover details about the directly connected nodes in a Layer 2 network. On Linux, you can find the **lldp** package for your distribution. For Ubuntu and Rocky Linux, this package can be installed with the following commands:

```
$ sudo apt install lldpd #[For Ubuntu]
$ sudo yum install lldpd #[For Rocky Linux]
```

The service can be started and enabled using the **systemctl** command:

```
chandan@ubuntu24:~$ sudo systemctl start lldpd
chandan@ubuntu24:~$ sudo systemctl enable lldpd
```

With the **lldp** package installed and service enabled on both the Ubuntu and Rocky Linux virtual machines, we can query the **lldp** daemon for the directly connected nodes. On the Hypervisor running these virtual machines, we can discover the ports the virtual machines are connected. This can be done using the following command:

$ lldpcli show neighbors

The neighbours are illustrated in the following figure:

```
Interface:    vnet3, via: LLDP, RID: 7, Time: 1 day, 21:33:26
  Chassis:
    ChassisID:    mac 52:54:00:8a:3c:58
    SysName:      rocky9
    SysDescr:     Rocky Linux 9.5 (Blue Onyx) Linux 5.14.0-503.14.1.el9_5.x86_64 #1 SMP PREEMPT_DYNAMIC Fri Nov 15 12:04:32 UTC 2024 x86_64
    MgmtIP:       192.168.122.248
    MgmtIface:    2
    MgmtIP:       fe80::5054:ff:fe8a:3c58
    MgmtIface:    2
    Capability:   Bridge, on
    Capability:   Router, on
    Capability:   Wlan, off
    Capability:   Station, off
  Port:
    PortID:       mac 52:54:00:49:93:0f
    PortDescr:    enp7s0
    TTL:          120
-------------------------------------------------------------------------------
Interface:    vnet4, via: LLDP, RID: 11, Time: 0 day, 00:10:33
  Chassis:
    ChassisID:    mac 52:54:00:ee:6a:84
    SysName:      ubuntu24
    SysDescr:     Ubuntu 24.04.2 LTS Linux 6.8.0-60-generic #63-Ubuntu SMP PREEMPT_DYNAMIC Tue Apr 15 19:04:15 UTC 2025 x86_64
    MgmtIP:       192.168.122.61
    MgmtIface:    2
    MgmtIP:       fe80::5054:ff:feee:6a84
    MgmtIface:    2
    Capability:   Bridge, off
    Capability:   Router, off
    Capability:   Wlan, off
    Capability:   Station, on
  Port:
    PortID:       mac 52:54:00:ee:6a:84
    PortDescr:    enp1s0
    TTL:          120
```

***Figure 1.28**: LLDP neighbours*

To understand how LLDP discovers the directly connected nodes and the capabilities of the directly connected nodes, we can run a **tcpdmp** and capture the LLDP messages. This can be done using the following command:

$ sudo tcpdump –vn –i vnet3

The **-v** option is needed to make **tcpdump** more verbose and capture the details of the LLDP message. As you can see in the following example, the LLDP message captured on the node contains the details of the node connected to the interface **vnet2**:

```
21:04:45.844820 LLDP, length 226
        Chassis ID TLV (1), length 7
          Subtype MAC address (4): 08:9d:f4:7f:8f:d2
        Port ID TLV (2), length 7
          Subtype MAC address (3): fe:54:00:49:93:0f
        Time to Live TLV (3), length 2: TTL 120s
        System Name TLV (5), length 15: chandan-FRI-FXX
        System Description TLV (6), length 108
          Ubuntu 24.04.2 LTS Linux 6.8.0-60-generic #63-Ubuntu SMP PREEMPT_DYNAMIC Tue Apr 15 19:04:15 UTC 2025 x86_64
        System Capabilities TLV (7), length 4
          System  Capabilities [Bridge, WLAN AP, Router, Station Only] (0x009c)
          Enabled Capabilities [Bridge, WLAN AP, Router] (0x001c)
        Management Address TLV (8), length 12
          Management Address length 5, AFI IPv4 (1): 192.168.122.1
          Interface Index Interface Numbering (2): 3
        Management Address TLV (8), length 24
          Management Address length 17, AFI IPv6 (2): fc00:f853:ccd:e793::1
          Interface Index Interface Numbering (2): 5
        Port Description TLV (4), length 5: vnet3
        Organization specific TLV (127), length 9: OUI IEEE 802.3 Private (0x00120f)
          Link aggregation Subtype (3)
            aggregation status [supported], aggregation port ID 0
        Organization specific TLV (127), length 9: OUI IEEE 802.3 Private (0x00120f)
          MAC/PHY configuration/status Subtype (1)
            autonegotiation [none] (0x00)
            PMD autoneg capability [unknown] (0x0000)
            MAU type unknown (0x0036)
        End TLV (0), length 0
```

***Figure 1.29**: LLDP packet*

The LLDP protocol can be configured by adding config files to the **/etc/lldpd.d** directory.

To view the current configuration of the **lldpd** daemon, you can run the following command:

```
$ lldpcli show configuration
```

The following is the output:

```
21:04:45.844820 LLDP, length 226
        Chassis ID TLV (1), length 7
          Subtype MAC address (4): 08:9d:f4:7f:8f:d2
        Port ID TLV (2), length 7
          Subtype MAC address (3): fe:54:00:49:93:0f
        Time to Live TLV (3), length 2: TTL 120s
        System Name TLV (5), length 15: chandan-FRI-FXX
        System Description TLV (6), length 108
          Ubuntu 24.04.2 LTS Linux 6.8.0-60-generic #63-Ubuntu SMP PREEMPT_DYNAMIC Tue Apr 15 19:04:15 UTC 2025 x86_64
        System Capabilities TLV (7), length 4
          System  Capabilities [Bridge, WLAN AP, Router, Station Only] (0x009c)
          Enabled Capabilities [Bridge, WLAN AP, Router] (0x001c)
        Management Address TLV (8), length 12
          Management Address length 5, AFI IPv4 (1): 192.168.122.1
          Interface Index Interface Numbering (2): 3
        Management Address TLV (8), length 24
          Management Address length 17, AFI IPv6 (2): fc00:f853:ccd:e793::1
          Interface Index Interface Numbering (2): 5
        Port Description TLV (4), length 5: vnet3
        Organization specific TLV (127), length 9: OUI IEEE 802.3 Private (0x00120f)
          Link aggregation Subtype (3)
            aggregation status [supported], aggregation port ID 0
        Organization specific TLV (127), length 9: OUI IEEE 802.3 Private (0x00120f)
          MAC/PHY configuration/status Subtype (1)
            autonegotiation [none] (0x00)
            PMD autoneg capability [unknown] (0x0000)
            MAU type unknown (0x0036)
        End TLV (0), length 0
```

***Figure 1.30**: LLDP configuration*

Conclusion

In this chapter, we were introduced to the basics of networking on Linux. We established the conceptual model of networking by understanding the various layers of the OSI model. We understood the functionality of the layers and how the model helps in a clear understanding of the protocols and standards. We then briefly looked at how to configure a Linux distribution with an IP address to have connectivity to the network the host is connected to.

The rest of the chapter then discussed the various virtual interfaces and Layer 2 protocols. We discussed the details of address resolution with ARP and how an IP address (Layer 3 address) gets mapped to a MAC address (Layer 2 address). We understood the concept of fault tolerance and its implementation using link aggregation and the STP protocol. We then discussed the LLDP protocol and how it can be used to discover the nodes that are directly connected through links. LLDP can also discover the capabilities of the connected nodes.

We also went through lab exercises to try the concepts we learned on Linux

using VETH interfaces, Linux Bridge, and OpenVSwitch.

In the next chapter, we will cover Layer 3 of the OSI model and understand how IP addressing and routing work together. We will learn how to configure routes manually and dynamically with routing protocols.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 2

Routing and Packet Forwarding

Introduction

In the last chapter, we discussed the **Open Systems Interconnection** (**OSI**) layer and explored the Layer 2 network protocols. In this chapter, we will look at Layer 3 in the OSI model. We will understand how the Network layer works and explore IP Addressing and routing.

Layer 3 routing on Linux involves forwarding **Internet Protocol** (**IP**) packets between networks, transforming a Linux system into a fully functional router. At the core of this functionality is IP addressing, which ensures proper network segmentation and communication. We will understand how each interface must be configured with a unique IP address, either statically or dynamically via **Dynamic Host Configuration Protocol** (**DHCP**). We will learn about how subnetting and **Classless Inter-Domain Routing** (**CIDR**) notation define the network scope, while the routing table determines how packets travel between different subnets.

Linux supports advanced routing features such as **policy-based routing** (**PBR**), dynamic routing protocols like **Open Shortest Path First** (**OSPF**) and **Border Gateway Protocol** (**BGP**), through software suites such as FRRouting or Bird. Whether used for a simple home gateway, a corporate network, or a cloud-based infrastructure, Linux provides a flexible and powerful platform for Layer 3 routing and IP traffic management.

Structure

This chapter covers the following topics:

- IP addressing
- Classless addressing
- DHCP service for automatic IP address configuration
- Routing in Layer 3 networks
- Equal-cost multi-path routing
- Policy-based routing and VRF
- Dynamic routing

Objectives

By the end of this chapter, you will have a strong understanding of IP addressing and routing. You will learn about subnetting and CIDRs, and how IP networks communicate with each other using routes. You will learn to configure DHCP to automatically configure hosts on a network with IP addresses. We will also learn about **equal-cost multi-path** (**ECMP**) and PBR, and discuss dynamic routing and how Linux can be used as a network router.

IP addressing

IP addressing is a fundamental concept in networking that enables devices to communicate over the internet or local networks. Every device connected to a network, whether a computer, smartphone, or server, requires a unique IP address to send and receive data. IP addresses ensure that data packets reach their intended destination by providing a structured way to identify and locate devices.

There are two main versions of IP addresses in use today:

- **Internet Protocol version 4 (IPv4)**: A 32-bit address format (e.g., 192.168.1.1), still the most widely used.

- **Internet Protocol version 6 (IPv6)**: A 128-bit address format (e.g., 2001:0db8:85a3::8a2e:0370:7334), designed to replace IPv4 due to its limited address space.

An IPv4 address is a 32-bit number. This 32bit number is divided into 4 groups of 8 bits called **octets**. Thus, each IPv4 address consists of four octets (8 bits each), separated by dots, sometimes referred to as the dotted representation.

The IP address can be represented in a binary or decimal format:

- **Dotted binary format**: 11000000.10101000.00000001.00000001
- **Dotted decimal format**: 192.168.1.1

In decimal representation, each octet ranges from 0 to 255, since an octet with 8 bits can have $2^8 = 256$ possible values.

For example:

- **192 in binary**: 11000000
- **168 in binary**: 10101000
- **1 in binary**: 00000001

We can also write the IP address as an integer or a hexadecimal number.

For example, the aforementioned IP address of 192.168.1.1 can be written as:

- **Integer**: 3232235777
- **Hexadecimal**: C0A80101

However, normally, the IP address is represented in dotted decimal notation for human readability. IPv6 is the successor to IPv4, designed to overcome IPv4's limitations, primarily its 32-bit address space exhaustion. IPv6 uses a 128-bit address format, providing an astronomically larger pool of addresses (approximately 3.4×10^{38}) compared to IPv4 (approximately 4.3 billion). An IPv6 address is 128 bits long, represented as 8 groups of 4 hexadecimal digits, separated by colons (:).

For example, the following is an IPv6 address:

2001:0db8:85a3:0000:0000:8a2e:0370:7334

In addition to the long representation, IPv6 also provides a shortened format of addresses. IPv6 addresses can be shortened using the following rules:

- Leading zeros in each group can be omitted:

  09C0: 9C0

  0000: 0
- Consecutive groups of all zeros can be replaced with :: (only once per address).

  2001:0db8:0000:0000:0000:0000:1428:57ab: 2001:0db8::1428:57ab

Final shortened form is as follows:

- **Original**: 2001:0db8:85a3:0000:0000:8a2e:0370:7334
- **Shortened**: 2001:db8:85a3::8a2e:370:7334

The following table compares the features of IPv4 vs. IPv6 addresses:

Features	IPv4	IPv6
Address length	32bit	128bit
Address format	Dotted decimal	Colon separated hexadecimal
Subnet mask	/24	/64 Prefix length
Broadcast	Yes	No, uses multicast instead
Auto configuration	DHCP	SLAAC and DHCP6

***Table 2.1** : Features of IPv4 vs. IPv6 addresses*

Netmask and subnetting

In the preceding section, we discussed how an IP address is a 32-bit number, and it is required by devices on the network to communicate with each other. Unlike an IP address, which identifies a specific device, the subnet mask describes the structure of the network itself. A subnet mask is a 32-bit number that defines how an IP address is divided into network and host portions. It works in conjunction with an IP address to determine which devices belong to the same local network segment and which require routing to another network.

The key distinction between an IP address and a subnet mask lies in their roles. An IP address serves as a unique identifier for a device on a network,

like a street address for a house. The subnet mask functions like a zoning map, specifying which part of the address represents the neighbourhood (network portion) and which part identifies the individual house (host portion). When combined through a bitwise AND operation, they yield the network address that routers use for making forwarding decisions.

Subnet masks use consecutive 1s followed by 0s in binary form, where the 1s mask the network portion and the 0s reveal the host portion. This differs from IP addresses, which can have any bit pattern.

A subnet mask is a 32-bit number that:

- Uses 1s to represent the network portion.
- Uses 0s to represent the host portion.

For example:

- **IP**: 192.168.1.1
- **Subnet mask**: 255.255.255.0

The subnet mask has the binary representation: 11111111.11111111.11111111. 00000000:

- The first 24 bits of the IP address represent the Network (192.168.1)
- The last 8 bits of the IP address represent the Host (.1)

Subnetting serves as a fundamental organizational tool in network architecture, addressing critical challenges that emerge as networks expand in size and complexity. By dividing a single large network into smaller, more manageable subnetworks, subnetting provides solutions to several inherent limitations of flat network designs. One primary benefit comes from controlling broadcast traffic; each subnet creates its own broadcast domain, preventing unnecessary traffic floods that would otherwise congest the entire network as more devices connect. This segmentation proves particularly valuable in large enterprise environments where thousands of devices might otherwise share the same broadcast space, leading to severe performance degradation.

For example:

- **Original network**: 192.168.0.0/16 (65,534 hosts)
- **After sub-netting**: 192.168.1.0/24 (254 hosts per subnet)

Note: The first and the last address in the host address range are reserved for special purposes and are called the network address and the broadcast address, respectively. This is one of the downsides of subnetting, as by increasing the number of networks, we also lose 2 host addresses in each host address range.

Beyond traffic management, subnetting enables logical grouping of devices based on function, location, or security requirements. Departments like finance or research can operate within isolated subnets while still maintaining controlled connectivity to shared resources. This structure supports security implementations by allowing network administrators to enforce granular access controls and firewall rules at subnet boundaries rather than attempting to manage individual devices. Security policies can be applied uniformly to entire subnets containing sensitive systems like payment processing or intellectual property storage.

The hierarchical structure introduced by subnetting simplifies troubleshooting and network management. Problems can be isolated to specific subnets, and network changes can be implemented with surgical precision rather than enterprise-wide disruption.

Classful addressing

Classful subnetting represents the original addressing scheme used in early IP networks. This subnetting approach divides IPv4 addresses into fixed categories:

- Class A
- Class B
- Class C
- Class D
- Class E

Each class had a predetermined network and host portions.

The classification system automatically assigns a default subnet mask based solely on the first few bits of the IP address, without considering the actual needs of the network. The class of the IP address was determined by the first few bits of the address.

For example, Class A addresses had:

- **Binary start**: 0 (first bit is always 0)
- **Bit pattern**: 0xxxxxxx.xxxxxxxx.xxxxxxxx.xxxxxxxx

This produces a decimal range of 1.0.0.0 to 127.255.255.255. Some portion of this range is reserved, and the only usable range is 1.0.0.1 to 126.255.255.255.

Similarly, Class B had a binary start of 10, Class C had 110, Class D had 1110, and Class E had 1111 as the binary start. The following table describes the different classful address ranges:

Class	Range	Default mask	Usage
A	1.0.0.0 - 126.255.255.255	/8 (255.0.0.0)	Large networks
B	128.0.0.0 - 191.255.255.255	/16(255.255.0.0)	Medium networks
C	192.0.0.0 - 223.266.255.255	/24(255.255.255.0)	Small networks
D	224.0.0.0 - 239.255.255.255		Multicast address
E	240.0.0.0 - 255.255.255.255		Reserved address

***Table 2.2**: Classful addressing*

The following are the problems with classful addressing:

- Inefficient use of addresses (Class A too large, Class C too small).
- No flexibility in subnet sizes.
- Led to addressing exhaustion quickly.

This limitation led to the development of CIDR.

Classless addressing

CIDR replaced classful addressing with **variable-length subnet masking** (**VLSM**), allowing more efficient allocation of IP addresses.

Key features of CIDR are:

- No fixed classes (flexible network sizes).
- Slash notation (e.g., /24) instead of default masks.
- Super-netting (combining multiple networks into one larger block).

CIDR notation is written as: IP address/Prefix length

For example, a Class C network of 192.168.1.0/255.255.255.0 can be written as 192.168.1.0/24. As CIDR supports variable-length subnets, the network can be further subnetted, with a prefix length of 26, as 192.168.1.0/26, which represents a subnet mask of 255.255.255.192, to produce 4 networks with 64 host addresses each:

- 192.168.1.0/26
- 192.168.1.64/26
- 192.168.1.128/26
- 192.168.1.192/26

Increasing the prefix length will produce more networks, but with a reduced number of host addresses in each network. E.g., a Class C network when sub-netted with:

- Prefix length of 24 produces 256 host addresses with 1 network address.
- Prefix length of 25 produces 128 host addresses with 2 network addresses.
- Prefix length of 26 produces 64 host addresses with 4 network addresses.
- Prefix length of 27 produces 32 host addresses with 8 network addresses, and so on.

CIDR also supports subnetting, which can be used to reduce the number of network addresses, e.g., in the preceding example, the network addresses 192.168.1.0/26 and 192.168.1.64/26 can be combined as 192.168.1.0/25. This is helpful in reducing the number of network routes and is called route aggregation.

The benefits of CIDR are as follows:

- Reduced IP wastage using flexible prefix length (allocates only needed addresses).

- Better route aggregation by combining subnets (reduces routing table size).
- Supports VLSM (subnets of different sizes in the same network).

DHCP service for automatic IP address configuration

DHCP is a network management protocol that automatically assigns IP addresses and other network configuration parameters to devices on a network. It eliminates the need for manual IP address assignment, making network administration automated, more efficient, and less error-prone.

One of DHCP's primary advantages is its ability to prevent IP address conflicts. By maintaining centralized control over address allocation, DHCP guarantees that no two devices on the same network receive duplicate IPs, which would otherwise disrupt communication. This is particularly important in environments with frequent device turnover, such as corporate offices or public Wi-Fi networks, where manual IP management would be impractical.

Beyond conflict prevention, DHCP optimizes IP address utilization through its lease-based system. Instead of permanently assigning addresses, DHCP temporarily allocates them, reclaiming and reassigning unused IPs as needed. This dynamic allocation is especially valuable in IPv4 networks, where address scarcity demands efficient distribution. Additionally, DHCP simplifies large-scale network administration by allowing centralized updates to critical settings such as DNS servers, gateways, and subnet masks, ensuring consistency across all connected devices.

DHCP provides the following services:

- Automatically assigns IP addresses to devices like computers, phones, etc.
- Provides subnet mask, default gateway, and **Domain Name System (DNS)** server information.
- Manages IP address leases, which are temporary IP assignments that

expire unless renewed.

- Prevents IP conflicts by centralized management so that no two devices get the same IP address simultaneously.

DHCP protocol operates in four main steps, known as the DORA process. The following table describes the process of IP address configuration using the DHCP service:

Step	DHCP message	Description
Discovery	**DHCPDISCOVER**	The device sends a broadcast message to get an IP address.
Offer	**DHCPOFFER**	The server responds with a free IP address.
Request	**DHCPREQUEST**	The device confirms and configures the IP address.
Acknowledge	**DHCPACK**	The DHCP server confirms the IP.

***Table 2.3**: DHCP messages*

Configuring DHCP service on Linux

In this section, we will discuss the installation and configuration of the DHCP service on Linux. The DHCP service on Ubuntu is provided by **isc-dhcp-server**, while on Rocky Linux, the package is called **dhcp-server**.

We can use the following commands to install the package:

- **On Ubuntu**:

```
sudo apt update
sudo apt install isc-dhcp-server –y
```

- **On Rocky Linux**:

```
sudo dnf install dhcp-server –y
```

Once installed, we can configure the service by editing the service's configuration file at:

```
/etc/dhcp/dhcpd.conf
```

This file contains fully commented examples of different configuration options.

The config file for DHCP has both global and subnet-specific options. In

the global options, we can set options like the domain name and DNS server IP address. The authoritative option tells the DHCP server that it is the only DHCP server in the network and has authority over the network. The details are as follows:

```
option domain-name "example.com";
option domain-name-servers 8.8.8.8, 8.8.4.4;
default-lease-time 600;
max-lease-time 7200;
authoritative;
```

The global options can be overridden in the subnet-specific configuration. Here is an example of a subnet configuration. In this example, we are allocating the IP address range 10.5.5.26 to 10.5.5.30 in the subnet 10.5.5.0/27. In addition, we are also setting the option for configuring the clients with DNS servers, setting the default domain name for DNS resolution. The router attribute is used to configure the gateway for the network. You can also see the default and the maximum lease time for which the IP is allocated to the device:

```
subnet 10.5.5.0 netmask 255.255.255.224 {
range 10.5.5.26 10.5.5.30;
option domain-name-servers ns1.internal.example.org;
option domain-name "internal.example.org";
option subnet-mask 255.255.255.224;
option routers 10.5.5.1;
option broadcast-address 10.5.5.31;
default-lease-time 600;
max-lease-time 7200;
}
```

We can also specify a specific IP address to be given for a certain host. The host can be identified based on its MAC address as shown here:

```
host webserver {
  hardware ethernet 0:0:c0:5d:bd:95;
  server-name "webserver.example.com";
```

```
}
```

Once you are satisfied with the configuration of the DHCP config, you can start the **dhcp** service using the following command:

- **On Ubuntu**:

```
sudo systemctl start isc-dhcp-server
```

- **On Rocky Linux**:

```
sudo systemctl start dhcpd
```

Now, let us test if the DHCP server is offering IP addresses on requests from the clients. To do this, we will set up a Linux bridge and add a VETH interface to this. We will start the DHCP server to serve on the bridge interface and request an IP address on the VETH interface.

The following commands can be used to make this setup:

```
$ sudo ip link add br1 type bridge
$ sudo ip link add veth1 type veth peer veth1br1
$ sudo ip link set veth1br1 master br1
$ sudo ip link set veth1 up
$ sudo ip link set br1 up
$ sudo ip addr add dev br1 10.5.5.1/27
```

We will now update the DHCP configuration to start the DHCP server on the bridge interface **br1**. To do this, we can edit the DHCP service configuration file **/etc/default/isc-dhcp-server** and add **br1** as the interface for the DHCP server to listen on:

```
On what interfaces should the DHCP server (dhcpd) serve DHCP requests?
Separate multiple interfaces with spaces, e.g. "eth0 eth1".
INTERFACESv4="br1"
INTERFACESv6=""
```

We will now restart **dhcp** server and make a DHCP request from the interface **veth1**. You will need the DHCP client, **dhclient**, to send a DHCP request. This command is part of the **isc-dhcp-client** package on Ubuntu and **dhcp-client** package on Rocky Linux:

```
$ sudo dhclient -4 veth1
```

The following capture of traffic with **tcpdump** shows the DHCP request and reply:

```
$ sudo tcpdump -n -i br1
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on br1, link-type EN10MB (Ethernet), snapshot length 262144 bytes
12:36:36.845102 IP 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from fa:0d:2d:24:57:21, length 300
12:36:36.854300 ARP, Request who-has 10.5.5.26 tell 10.5.5.1, length 28
12:36:37.883872 ARP, Request who-has 10.5.5.26 tell 10.5.5.1, length 28
12:36:38.110089 IP 10.5.5.1.67 > 10.5.5.26.68: BOOTP/DHCP, Reply, length 300
12:36:38.110335 IP 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from fa:0d:2d:24:57:21, length 300
12:36:38.119722 IP 10.5.5.1.67 > 10.5.5.26.68: BOOTP/DHCP, Reply, length 300
12:36:38.907617 ARP, Request who-has 10.5.5.26 tell 10.5.5.1, length 28
12:36:38.907704 ARP, Reply 10.5.5.26 is-at fa:0d:2d:24:57:21, length 28
```

After the **dhclient** command completes, the IP address on the **veth1** interface can be checked using the following command:

```
$ ip -br link show veth1
veth1@veth1br1   UP    fa:0d:2d:24:57:21 <BROADCAST,MULTICAST,UP,LOWER_UP>
$ ip -br addr show veth1
veth1@veth1br1   UP    10.5.5.26/27 fe80::f80d:2dff:fe24:5721/64
```

A DHCP server can also be used to configure the hosts for other network

services.

For example:

- **NTP servers (Option 42)**: Time servers for clock synchronization.
- **TFTP server (Option 66)**: For network booting (e.g., PXE).
- **Bootfile name (Option 67)**: File to download via TFTP (e.g., pxelinux.0).
- **SMTP/POP3 servers (Options 69/70)**: For email client configuration.
- **Static routes (Option 121)**: Non-default routes for the client.

Apart from these, it also supports providing the hosts with vendor-specific options. You can refer to the man page of the ISC DHCP server to get a complete list of options: **https://kb.isc.org/docs/isc-dhcp-44-manual-pages-dhcpdconf**

Routing in Layer 3 networks

Routing in Layer 3 networks forms the backbone of modern internet and enterprise communications by enabling intelligent path selection across interconnected networks. As discussed in the section on subnetting, it is essential to structure a flat network into smaller subnets, as this design solves key scalability and efficiency challenges. As networks grow in size and complexity, dividing them into smaller logical segments offers multiple advantages. A primary benefit is broadcast traffic containment by creating separate broadcast domains. Subnetting prevents network-wide congestion that would occur if all devices shared a single broadcast space. This proves crucial in enterprise environments where thousands of connected devices could otherwise overwhelm network performance.

Smaller networks also enable logical organization of network resources. Departments or functional units can be grouped into dedicated subnets while maintaining controlled access to shared services. Layer 3 routing provides a way to connect the smaller subnets together and allows communication between them by forwarding packets.

At its core, Layer 3 routing involves the process of forwarding IP packets between different network segments based on logical addressing, creating a

scalable framework that allows global connectivity while maintaining efficient traffic flow. Unlike Layer 2 switching, which operates within a single broadcast domain, Layer 3 routing provides the critical functionality needed to traverse multiple networks.

Layer 3 routing introduces several key architectural advantages, including logical network segmentation, improved scalability, and enhanced security controls. By operating at the network layer, routers can implement access control policies, quality of service markings, and traffic engineering features that are impossible at lower layers. The hierarchical nature of IP addressing, combined with route summarization techniques, allows for efficient management of large address spaces while minimizing routing table sizes. This becomes particularly critical in today's networks that must support increasingly complex requirements, including multi-cloud connectivity, software-defined networking, and mobile user access.

Layer 3 routing between networks is based on routing tables. A routing table is a critical data structure stored in network devices that contains information about how to forward network traffic to specific destination networks. It functions like a roadmap for data packets, determining the most efficient path they should take to reach their intended target across interconnected networks.

A routing table consists of the following components:

- **Destination network**: The target IP address or subnet (e.g., 192.168.1.0/24 or 0.0.0.0/0 for default routes).
- **Next hop**: The IP address of the next router or gateway that should receive the packet (e.g., 192.168.1.1).
- **Interface**: The physical or virtual network interface (e.g., eth0, wlan0) used to send the packet.
- **Metric**: A value indicating the *cost* of the route (lower = more preferred). Used when multiple paths exist.
- **Route type**: How the route was learned:
  - Directly connected (local network interfaces)
  - Static routes (manually configured)

- Dynamic routes (learned via protocols like OSPF, BGP)

When a router receives a packet, it compares the destination IP address against all available routes in its routing table and selects the most specific match, the route with the longest subnet mask (prefix length). This is called the **longest prefix match** (**LPM**) algorithm. It is the fundamental algorithm routers use to make forwarding decisions in IP networks.

This precise matching ensures packets take the most optimal path to their destination, even when multiple overlapping routes exist.

For example, a router receives a packet destined for 10.1.1.5 with these routes:

Prefix	Gateway	Comment
10.0.0.0/8	Gateway A	Less specific
10.1.0.0/16	Gateway B	More specific
10.1.1.0/24	Gateway C	Most specific

***Table 2.4**: Route table with overlapping networks*

In this case, all three routes match the destination. The router selects 10.1.1.0/24 (24-bit match) over the others. Traffic is forwarded through gateway C. The routes for a network can be added statically, by manually running a command, or learned dynamically using routing protocols like BGP, which we will discuss in the following sections.

On a Linux host, you can check the routing table by using the **ip route show** command:

```
$ ip route show
default via 192.168.122.1 dev enp1s0 proto static metric 100
192.168.122.0/24 dev enp1s0 proto kernel scope link src
192.168.122.248 metric 100
```

On a host with a single network interface, you will see an output as shown previously. This output says that we have a network 192.168.122.0/24, which the host is directly connected to using the enp1s0 interface. Any traffic to this interface should be forwarded directly using the interface enp1s0. Any traffic not matching this prefix of 192.168.122.0/24 should use

the default next hop address, and the packets should be forwarded to 192.168.122.1, which is called the default gateway.

On a host with multiple network interfaces, you may want to use a different interface to forward traffic for a certain network. For example, here is an example of a host with two interfaces, one of them connected to a 192.168.122.0/24 network and another connected to a 10.10.10.0/24 network.

In this case, packets destined to the directly connected network will be forwarded using the interfaces that are connected to the destination networks. Any packet that does not match the prefixes of the directly connected network will use the default route and will be forwarded to 192.168.122.1:

```
$ ip route show
default via 192.168.122.1 dev enp1s0 proto static metric 100
10.10.10.0/24 dev enp7s0 proto kernel scope link src 10.10.10.20 metric 100
192.168.122.0/24 dev enp1s0 proto kernel scope link src 192.168.122.248 metric 100
```

Let us say that we want to send some data to a destination 10.20.20.2 in this case as the prefix of the directly connected network do not match the destination address the default route is used with gateway of 192.168.122.1, but in your network design you might know that its preferable to send the packets to 10.20.20.2 using the gateway on 10.10.10.0/24, in such case you can add a route to your host for the prefix of 10.20.20.0/24.

To do this, use the following command:

```
$ sudo ip route add 10.20.20.0/24 via 10.10.10.1
$ ip route show
default via 192.168.122.1 dev enp1s0 proto static metric 100
10.10.10.0/24 dev enp7s0 proto kernel scope link src 10.10.10.20 metric 100
10.20.20.0/24 via 10.10.10.1 dev enp7s0
192.168.122.0/24 dev enp1s0 proto kernel scope link src
```

192.168.122.248 metric 100

Now the route table has a more specific route to match traffic for the network 10.20.20.0/24, and packets for this network will be forwarded using the gateway 10.10.10.1 through interface snp7s0 instead of using the default route. This example shows how the LPM algorithm selects a route based on the best match of the prefix against the destination address. In this case, we added the route manually by running a command, but the routes can be learned dynamically using routing protocols. In the aforementioned output, you will also notice a metric associated with a route; these are also referred to as the cost of the route. A route with a lower cost is preferred over one with a higher cost. In the next section, we will learn about ECMP, which uses multiple paths to forward traffic to achieve resilience and performance when there are multiple routes available to reach a destination network with equal cost.

Equal-cost multi-path routing

ECMP routing is a forwarding mechanism that enables networks to utilize multiple best paths simultaneously when routing traffic toward the same destination. Unlike traditional routing, where only a single optimal path is selected, ECMP distributes packets across multiple links of equal cost, improving bandwidth utilization, load balancing, and fault tolerance. This approach is particularly valuable in environments where redundant paths exist, such as data centers, enterprise networks, and internet backbone infrastructures.

ECMP operates by leveraging routing protocols like OSPF, IS-IS, or BGP, which identify multiple paths with the same metric to a destination. Once these paths are recognized, the router employs a hashing algorithm, typically based on packet headers such as source/destination IP, ports, or protocol, to distribute traffic across available links. This ensures that flows remain consistent (avoiding out-of-order packets) while efficiently utilizing network capacity.

A key advantage of ECMP is its ability to enhance both performance and reliability. Spreading traffic across multiple links prevents congestion on

any single path and provides automatic failover if one link becomes unavailable. This makes ECMP essential for modern high-availability architectures, including cloud networks, **software-defined wide area network** (**SD-WAN**) deployments, and spine-leaf data center designs.

Let us set up a lab to understand how we can configure ECMP on our Linux host. For this example, we will use static routing. We will discuss dynamic routing with BGP in the subsequent section.

The following figure describes the setup for this lab:

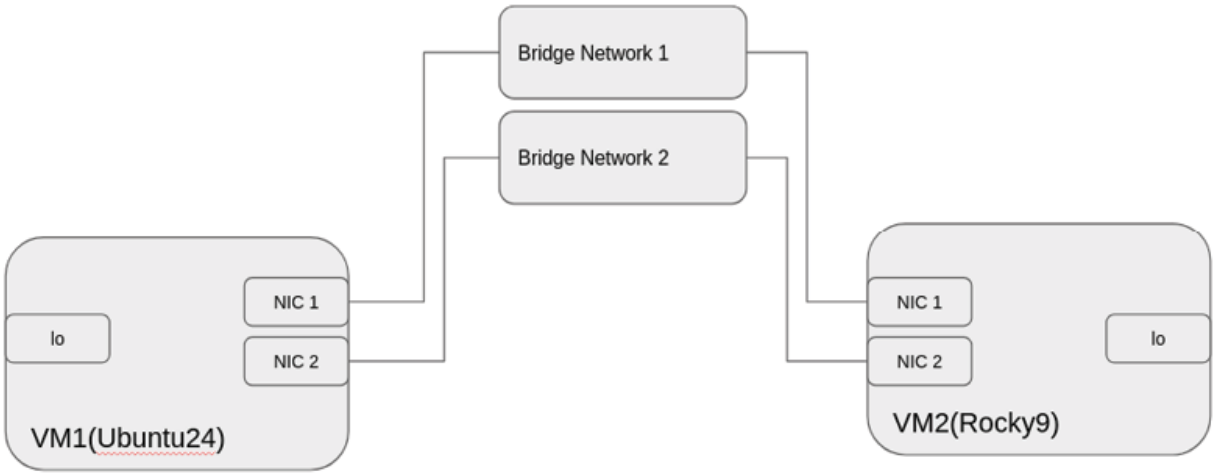


***Figure 2.1**: Setup for testing ECMP*

As we need to test multiple paths for network packets, we need to add an extra **network interface card** (**NIC**) in our virtual machines. Following is the setup with multiple NICs:

```
chandan@ubuntu24:~$ ip -br addr sh
lo               UNKNOWN        127.0.0.1/8 ::1/128
enp1s0           UP             192.168.122.61/24 fe80::5054:ff:feee:6a84/64
enp7s0           UP             10.10.10.10/24 fe80::5054:ff:fe17:9ab7/64
```

```
[chandan@rocky9 ~]$ ip -br addr sh
lo               UNKNOWN        127.0.0.1/8 ::1/128
enp1s0           UP             192.168.122.248/24 fe80::5054:ff:fe8a:3c58/64
enp7s0           UP             10.10.10.20/24 fe80::a413:85be:b97:5f5f/64
```

Now, let us set up the routes for reaching 172.16.10.1 and 172.16.20.1 from each of the virtual machines. To add the ECMP route, we can use the following command:

```
$ sudo ip route add 172.16.10.1/32 \
  nexthop via 10.10.10.20 weight 1 \
  nexthop via 192.168.122.248 weight 1
```

To test whether ECMP is using both routes, we will use a simple Python web server, use the following command:

```
$ python3 -m http.server --bind 172.16.10.1
```

To test if Linux is using both routes to send packets, use the following **curl** command to send an **Hypertext Transfer Protocol** (**HTTP**) request:

```
$ curl -s http://172.16.10.1:8000/ >/dev/null
```

The following **tcpdump** command can be used to capture the source and destination addresses:

```
$ sudo tcpdump -n port 8000|awk '/IP/{print $1" "$2" "$3" "$4" "$5}'
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on enp1s0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
13:43:18.761848 IP 192.168.122.1.40842 > 172.16.10.1.8000:
13:43:18.761940 IP 172.16.10.1.8000 > 192.168.122.1.40842:
13:43:18.762125 IP 192.168.122.1.40842 > 172.16.10.1.8000:
13:43:18.762177 IP 192.168.122.1.40842 > 172.16.10.1.8000:
13:43:18.762185 IP 172.16.10.1.8000 > 192.168.122.1.40842:
```

As we can see that requests are arriving on different IPs to reach the destination web server, we can conclude that Linux is using both routes to send and receive traffic.

How ECMP balances the traffic between multiple routes is controlled by the following **sysctl** parameters:

- **fib_multipath_hash_policy**: Controls which hash policy to use for

multipath routes. Possible values are:

- **0**: Layer 3
- **1**: Layer 4
- **2**: Layer 3 or inner Layer 3 if present
- **3**: Custom multipath hash. Fields used for multipath hash calculation are determined by the **fib_multipath_hash_fields sysctl**.

The default value for this parameter is 0.

- **fib_multipath_hash_fields:** When **fib_multipath_hash_policy** is set to 3 (custom multipath hash), the fields used for multipath hash calculation are determined by this **sysctl**. This value is a bitmask that enables various fields for multipath hash calculation. Possible fields are:
  - 0x0001 source IP address
  - 0x0002 destination IP address
  - 0x0004 IP protocol
  - 0x0008 unused (Flow label)
  - 0x0010 source port
  - 0x0020 destination port
  - 0x0040 inner source IP address
  - 0x0080 inner destination IP address
  - 0x0100 inner IP protocol
  - 0x0200 inner flow label
  - 0x0400 inner source port
  - 0x0800 inner destination port
  - The default value is 0x0007 (source IP, destination IP, and IP protocol).

The Linux kernel documentation for IP sysctl provides a full list of parameters and configuration, and can be found at the following location: **https://www.kernel.org/doc/Documentation/networking/ip-sysctl.rst**

ECMP also introduces challenges, particularly in flow distribution fairness

and handling asymmetric paths. Some implementations use advanced hashing techniques or weighted load balancing to optimize traffic distribution further. Additionally, certain applications, such as stateful firewall traversal or **Transmission Control Protocol** (**TCP**) performance tuning, may require careful consideration when ECMP is in use.

Policy-based routing and VRF

Virtual routing and forwarding (**VRF**) is a technology that creates multiple isolated routing tables on a single physical router or Linux host, allowing network paths to be logically separated at Layer 3. Each VRF instance operates like a virtual router with its own.

Using VRF, we can have:

- Independent routing table; routes are not shared between VRFs.
- Separate forwarding rules.
- Isolated address spaces.

VRF technology provides network segmentation by enabling overlapping IP address schemes, such as using 10.0.0.0/24 in both VRF-1 and VRF-2, without conflicts. Cloud providers leverage VRFs for multi-tenancy, isolating customer networks on shared physical hardware while maintaining strict separation. For security-sensitive environments, VRFs completely isolate critical traffic like management or compliant systems. Organizations also use VRFs to maintain parallel network environments, allowing test, development, and production networks to coexist on the same physical infrastructure.

To explore VRF on Linux, we will create a dummy interface as follows:

```
$ sudo ip link add eth2 type dummy
```

We will also add 10.0.0.1/24 address to the loopback interface to act as a gateway for our test setup, as follows:

```
$ sudo ip addr add dev lo 10.0.0.1/24
```

To create a VRF and add routes to the VRF in Linux, use the following commands:

```
$ sudo ip link add vrf-1 type vrf table 100
$ sudo ip link set vrf-1 up
$ sudo ip link set eth2 master vrf-1
$ sudo ip addr add 10.0.0.2/24 dev eth2$ sudo ip route add default via 10.0.0.1 vrf vrf-1
```

The VRF maintains an isolated routing table with id 100, which can be viewed with the following command:

```
$ ip route show vrf vrf-1
default via 10.0.0.1 dev eth2
```

In the aforementioned example, interface **eth2** is added to the **vrf-1**. The traffic on interface **eth2** now uses the route table 100 to look up routes when forwarding packets.

Comparison with VLAN

VRF operates at Layer 3 (routing) while VLAN functions at Layer 2 (switching). The isolation mechanism differs significantly between VRFs and VLANs; VRFs separate routing tables, whereas VLANs create distinct broadcast domains. A key advantage of VRF is its support for IP address reuse across instances, while VLANs require unique IP addressing within the same Layer 2 domain.

Common VRF applications

In enterprise networks, VRFs isolate departmental traffic between HR, Finance, and R&D groups. **Internet service providers** (**ISPs**) implement VRFs to separate customer networks on shared infrastructure. Modern data centers utilize VRFs extensively for multi-tenant cloud environments. **Virtual private network** (**VPN**) implementations often incorporate VRFs to maintain independent routing instances for different VPN connections and services.

PBR extends the VRF technology and applies policy rules to map traffic flows to custom routing tables.

Policy-based routing

At its core, PBR operates through the strategic application of route-maps that evaluate multiple packet attributes against predefined match conditions. When implemented on Linux systems using tools like iproute2 or network policy engines, PBR enables the creation of multiple routing tables that coexist with the main routing table. This architecture allows for exceptional flexibility, where traffic matching specific policies can bypass normal routing decisions entirely, following instead administrator-defined paths that may prioritize latency, cost, or security considerations over simple shortest-path metrics.

In Linux, PBR is implemented using multiple routing tables and writing policy rules to map packet flows to a specialized routing table.

Let us set up a lab to understand PBR on Linux for a simple example.

To do this, we will first create custom routing tables. This can be done by editing **/etc/iproute2/rt_tables** to add custom tables. The **rt_tables** entries are the id of a route table and its name:

```
$ echo "101 custom1"|sudo tee -a /etc/iproute2/rt_tables
$ echo "102 custom2" |sudo tee -a /etc/iproute2/rt_tables
```

Next, we will add rules to the custom route table. You can use the following command to add the routes:

```
$ sudo ip route add 172.16.10.1/32 via 192.168.122.248 table custom1
$ sudo ip route add 172.16.10.1/32 via 10.10.10.20 table custom2
```

The custom route table can be verified with the **ip route show** command with the table argument:

```
$ ip route show table custom1
172.16.10.1 via 192.168.122.248 dev enp1s0
$ ip route show table custom2
172.16.10.1 via 10.10.10.20 dev enp7s0
```

Now we have the custom routing tables set up with different routes for 172.16.10.1/32, with route table **custom1** using 192.168.122.248 as the gateway, while table **custom2** uses 10.10.10.20.

Now we can define the rules to map traffic flows to these custom routing tables. To test the traffic flows with custom route tables, we add two rules

that map connections based on the destination port. This can be done using the following commands:

```
$ sudo ip rule add ipproto tcp dport 5000 lookup custom1
$ sudo ip rule add ipproto tcp dport 6000 lookup custom2
```

The rules can be viewed with the **ip rule show** command as follows:

```
$ ip rule show
0:      from all lookup local
32764:  from all ipproto tcp dport 6000 lookup custom2
32765:  from all ipproto tcp dport 5000 lookup custom1
32766:  from all lookup main
32767:  from all lookup default
```

With these rules in place, if a request is made to a service running on port 5000 on IP 172.16.10.1, the route for this IP will be looked up in table **custom1**, which will use 192.168.122.248 as the gateway, while for port 6000, table **custom2** will be used with a gateway of 10.10.10.20.

To test the PBR flows, we start two web servers on 172.16.10.1 and ports 5000 and 6000:

```
$ python3 -m http.server --bind 172.16.10.1 5000 &
$ python3 -m http.server --bind 172.16.10.1 6000 &
```

Now we can send a request to the web servers using the following curl commands:

```
$ curl -s http://172.16.10.1:5000 >/dev/null
$ curl -s http://172.16.10.1:6000 >/dev/null
```

The logs for the web servers should include the following lines:

```
192.168.122.1 - - [08/Jun/2025 04:39:48] "GET / HTTP/1.1" 200 -
10.10.10.20 - - [08/Jun/2025 04:39:56] "GET / HTTP/1.1" 200 -
```

This confirms that the requests are reaching the web servers from different routes.

On Linux **ip rule** supports multiple matching criteria to determine which routing table should handle specific traffic.

Here are some of the available match options:

- Source address matching:

  ip rule add from <IP/CIDR> lookup <tables name>

- Destination address matching:

  ip rule add to <IP/CIDR> lookup <tables name>

- Firewall mark (fwmark) matching, this requires prior packet marking:

  iptables -t mangle -A PREROUTING -p tcp --dport 80 -j MARK --set-mark 1
  ip rule add fwmark lookup <tables name>

- Protocol/Port matching:

  ip rule add ipproto tcp dport <destination port> lookup <tables name>
  ip rule add ipproto udp sport <source port> lookup <tables name>

- Interface-based matching:

  ip rule add <incoming interface> lookup <tables name>
  ip rule add <outgoing interface> lookup <tables name>

- **Type of service** (**TOS**) matching:

  ip rule add tos <TOS_VALUE> lookup <TABLE_ID>

- UID/GID matching (User/Session based):

  ip rule add uidrange <start uid>-<end uid> - lookup <table name>
  ip rule add supplgroup <gid> lookup <tables name>

- Packet length matching:

  ip rule add length <min>-<max> lookup <tables name>

In summary, PBR enables granular traffic control beyond standard destination-based routing, making it invaluable for complex network scenarios. It allows routing decisions based on source IP, application type, **quality of service** (**QoS**) markings, or interface, enabling:

- **Multi-wide area network (WAN) load balancing**: Distribute traffic

across multiple ISPs based on the application or source subnet.

- **Service chaining**: Force specific flows through firewalls, analysers, or VPN tunnels.
- **QoS implementation**: Route latency-sensitive traffic (VoIP, video) over premium paths.
- **Traffic segregation**: Isolate guest, host, or management traffic.
- **Cloud/Multi-tenant routing**: Maintain separate paths for different customers or applications.

PBR provides network architects control over traffic flows, improving performance, security, and resource utilization while supporting business and technical requirements that standard routing cannot address. Its flexibility makes it essential for modern multi-cloud and enterprise networks.

Dynamic routing

Dynamic routing is a critical component of modern networking, allowing routers to automatically exchange routing information and adapt to changes in network topology. Unlike static routing, where paths are manually configured, dynamic routing protocols enable routers to communicate with each other, sharing updates about available routes, link failures, and optimal paths. This capability is essential for scalable, fault-tolerant networks, particularly in environments with frequent changes or complex architectures.

On Linux, FRR is one of the leading open-source solutions for implementing dynamic routing. FRR is a feature-rich routing suite that supports industry-standard protocols such as OSPF, BGP, RIP, IS-IS, and EIGRP, making it suitable for enterprise networks, data centres, and internet service providers. Originally derived from Quagga, FRR has evolved into a high-performance, modular routing platform that integrates seamlessly with Linux-based networking.

One of FRR’s key strengths is its ability to turn a Linux system into a full-fledged router capable of participating in dynamic routing. Whether

deploying OSPF for internal enterprise routing or BGP for internet peering, FRR provides the necessary tools to build resilient and efficient networks. Its modular architecture allows administrators to enable only the required protocols, optimizing resource usage while maintaining flexibility.

FRR also supports advanced features such as route redistribution, PBR, and VRF, making it a versatile choice for complex networking scenarios. With its CLI modelled after Cisco’s IOS, network engineers familiar with traditional routers will find FRR intuitive to configure and manage.

By leveraging FRR on Linux, organizations can achieve scalability, automation, and high availability without relying on proprietary hardware. FRR empowers administrators to build adaptive networks that respond dynamically to real-time conditions.

Let us set up a lab to explore and understand the FRR service on Linux. We will examine dynamic routing with FRR using BGP to advertise a route from one Linux host to another. BGP serves as the fundamental routing protocol that glues together the global Internet, enabling data exchange between autonomous systems across organizational and national boundaries.

An **Autonomous System** (**AS**) in the context of BGP represents a fundamental organizational unit of the Internet's routing architecture. It refers to a collection of IP networks and routers under the control of a single administrative entity that presents a common, clearly defined routing policy to the Internet. Each AS is assigned a globally unique **Autonomous System Number** (**ASN**) by regional Internet registries, which serves as its identifier in BGP routing exchanges.

BGP specializes in scalable path vector routing between independently managed networks. Each BGP speaker maintains its own routing policies while exchanging reachability information with neighbouring AS through TCP connections on port 179. This design provides exceptional stability and control. BGP updates propagate incrementally, and the protocol can handle the massive routing tables of the global Internet while allowing each participant to enforce local preferences and business relationships.

The following figure describes the setup to test the FRR service:

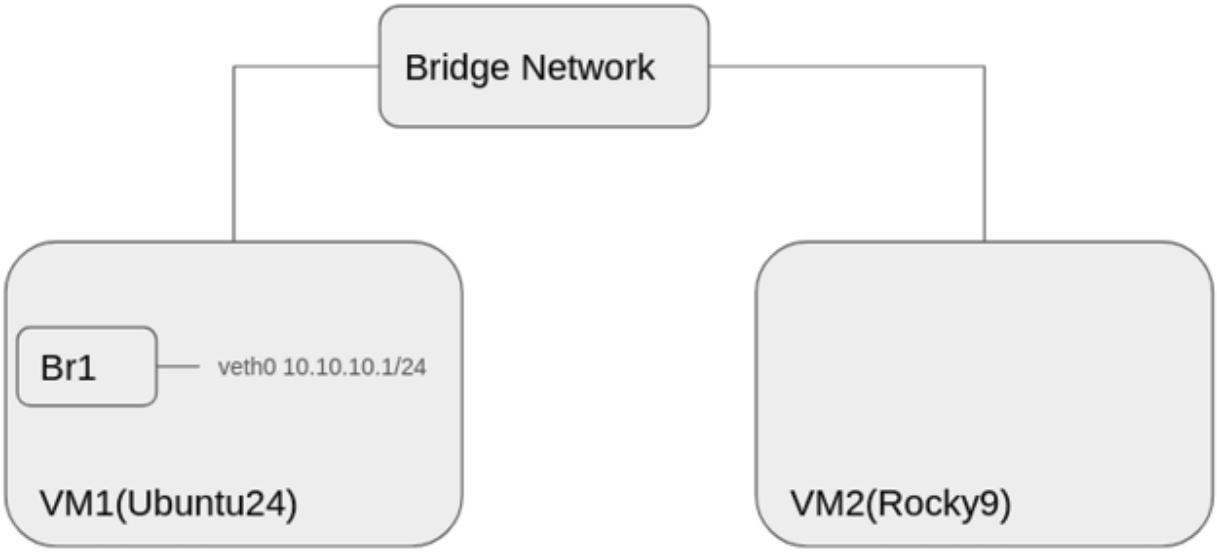


***Figure 2.3**: Setup for testing BGP based dynamic routes*

To install **frr** package, use the following command:

On the Ubuntu host, use **apt** to install the package:

```
$ sudo apt install -y frr
```

On Rocky Linux, use **yum** to install the package:

```
$ sudo yum install -y frr
```

Now we should check if **frr** service is running with **systemctl** command:

```
$ systemctl status frr
```

If the service is not running, we can start the service with the following command:

```
$ sudo systemctl start frr
```

Now we will configure the FRR service to enable and start the BGP protocol.

To do this, edit the **frr** config file **/etc/frr/daemons** and enable BGP. Following is the configuration we have set for **frr**, the **bgpd** daemon is set to yes:

```
bgpd=yes
ospfd=no
ospf6d=no
```

```
ripd=no
ripngd=no ...
```

Now we will restart the **frr** service to activate the configuration change. This can be done using the **systemctl** command:

```
$ sudo systemctl restart frr
```

We can use the **vtysh** command to configure the BGP daemon. Once the command is executed, you will see the following banner:

```
chandan@ubuntu24:~$ sudo vtysh
Hello, this is FRRouting (version 8.4.4).
Copyright 1996-2005 Kunihiro Ishiguro, et al.
ubuntu24#
```

We will configure the two Linux hosts as BGP peers

On the Ubuntu24 host, we enter the following configuration in the **vtysh**:

```
ubuntu24#
ubuntu24# configure terminal
ubuntu24(config)# router bgp 65001
ubuntu24(config-router)# neighbor 192.168.122.248 remote-as 65002
ubuntu24(config-router)# exit
ubuntu24(config)# exit
```

To check the configuration of **frr** daemon, we can run the following command:

```
ubuntu24# show running-config
Building configuration...
```

Current configuration:

```
!
frr version 8.4.4
frr defaults traditional
hostname ubuntu24
log syslog informational
no ipv6 forwarding
```

```
service integrated-vtysh-config
!
router bgp 65001
neighbor 192.168.122.248 remote-as 65002
exit
!
end
ubuntu24#
```

Now we can configure the Rocky Linux host:

```
rocky9# configure terminal
rocky9(config)# router bgp 65002
rocky9(config-router)# neighbor 192.168.122.61 remote-as 65001
rocky9(config-router)# exit
rocky9(config)# exit
```

To check the status of BGP:

```
rocky9# show bgp summary
IPv4 Unicast Summary (VRF default):
BGP router identifier 192.168.122.248, local AS number 65002 vrf-id 0
BGP table version 0
RIB entries 0, using 0 bytes of memory
Peers 1, using 725 KiB of memory
Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
192.168.122.61  4      65001         4         4        0    0    0 00:01:16 (Policy) (Policy) N/A

Total number of neighbors 1
rocky9#
```

Next, we will add a local network on the Ubuntu24 host and advertise this network to the Rocky9 host using BGP. To do this, we will add a bridge

interface on the Ubuntu24 host.

First, we will check the routing table of the Ubuntu24 host:

```
chandan@ubuntu24:~$ ip route show
default via 192.168.122.1 dev enp1s0 proto static
192.168.122.0/24 dev enp1s0 proto kernel scope link src
192.168.122.61
```

Now we will add the bridge interface and configure a local network attached to this bridge:

```
chandan@ubuntu24:~$ sudo ip link add br1 type bridge
chandan@ubuntu24:~$ sudo ip link add veth0 type veth peer vethbr
chandan@ubuntu24:~$ sudo ip link set vethbr master br1
chandan@ubuntu24:~$ sudo ip link set vethbr up
chandan@ubuntu24:~$ sudo ip link set veth0 up
chandan@ubuntu24:~$ sudo ip link set br1 up
chandan@ubuntu24:~$ ip link show br1
5: br1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 1e:98:ab:ff:57:91 brd ff:ff:ff:ff:ff:ff
```

Now we will add the local network of 10.10.10.0/24 for this interface:

```
chandan@ubuntu24:~$ sudo ip addr add dev br1 10.10.10.1/24
```

The route table after the bridge is configured as follows:

```
chandan@ubuntu24:~$ ip route show
default via 192.168.122.1 dev enp1s0 proto static
10.10.10.0/24 dev br1 proto kernel scope link src 10.10.10.1 linkdown
192.168.122.0/24 dev enp1s0 proto kernel scope link src 192.168.122.61
```

You can see that the route table now shows the newly added local network on Ubuntu24. Now we will configure the BGP daemon to advertise the local network on host Ubuntu24:

```
chandan@ubuntu24:~$ sudo vtysh
Hello, this is FRRouting (version 8.4.4).
Copyright 1996-2005 Kunihiro Ishiguro, et al.
```

```
ubuntu24# configure terminal
ubuntu24(config)# router bgp 65001
ubuntu24(config-router)# network 10.10.10.0/24
ubuntu24(config-router)# exit
ubuntu24(config)# ip forwarding
ubuntu24(config)# exit
ubuntu24# show bgp summary
IPv4 Unicast Summary (VRF default):
BGP router identifier 192.168.122.61, local AS number 65001 vrf-id 0
BGP table version 0
RIB entries 1, using 192 bytes of memory
Peers 1, using 724 KiB of memory
Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ
Up/Down State/PfxRcd   PfxSnt Desc
192.168.122.248 4      65002        26        26        0    0    0 00:23:50
(Policy) (Policy) N/A
Total number of neighbors 1
ubuntu24#
```

We need to allow the propagation of routes by updating the router policy. First, we can check the route table on the **Rocky9** host, as shown here:

```
[chandan@rocky9 ~]$ ip route show
default via 192.168.122.1 dev enp1s0 proto static metric 100
192.168.122.0/24 dev enp1s0 proto kernel scope link src
192.168.122.248 metric 100
```

We can update the router policy, using the following command:

```
[chandan@rocky9 ~]$ sudo vtysh
Hello, this is FRRouting (version 8.5.3).
Copyright 1996-2005 Kunihiro Ishiguro, et al.
rocky9# configure terminal
rocky9(config)# route-map PERMIT-ALL permit 10
rocky9(config-route-map)# exit
```

```
rocky9(config)# router bgp 65002
rocky9(config-router)# neighbor 192.168.122.61 route-map PERMIT-ALL in
rocky9(config-router)# neighbor 192.168.122.61 route-map PERMIT-ALL out
rocky9(config-router)# exit
rocky9(config)# exit
rocky9# show bgp summary
IPv4 Unicast Summary (VRF default):
BGP router identifier 192.168.122.248, local AS number 65002 vrf-id 0
BGP table version 5
RIB entries 1, using 192 bytes of memory
Peers 1, using 725 KiB of memory
Neighbor        V         AS   MsgRcvd   MsgSent   TblVer  InQ OutQ  Up/Down State/PfxRcd   PfxSnt Desc
192.168.122.61  4      65001        75        75        0    0    0 01:04:49            1        1 N/A
Total number of neighbors 1
rocky9#
```

We can check the route table from **vtysh** to see which routes are added by BGP:

```
rocky9# show ip route
Codes: K - kernel route, C - connected, S - static, R - RIP,
       O - OSPF, I - IS-IS, B - BGP, E - EIGRP, N - NHRP,
       T - Table, v - VNC, V - VNC-Direct, F - PBR,
       f - OpenFabric,
       > - selected route, * - FIB route, q - queued, r - rejected, b - backup
       t - trapped, o - offload failure
K>* 0.0.0.0/0 [0/100] via 192.168.122.1, enp1s0, 06:02:31
```

```
B>* 10.10.10.0/24 [20/0] via 192.168.122.61, enp1s0, weight 1, 00:04:55
C>* 192.168.122.0/24 is directly connected, enp1s0, 06:02:31
rocky9#
```

Routes can also be checked from Linux with **ip** command:

```
[chandan@rocky9 ~]$ ip route show
default via 192.168.122.1 dev enp1s0 proto static metric 100
10.10.10.0/24 nhid 8 via 192.168.122.61 dev enp1s0 proto bgp metric 20
192.168.122.0/24 dev enp1s0 proto kernel scope link src 192.168.122.248 metric 100
```

Now we can reach the **ip** on the host Ubuntu24 from Rocky9:

```
[chandan@rocky9 ~]$ ping -c3 10.10.10.1
PING 10.10.10.1 (10.10.10.1) 56(84) bytes of data.
64 bytes from 10.10.10.1: icmp_seq=1 ttl=64 time=0.709 ms
64 bytes from 10.10.10.1: icmp_seq=2 ttl=64 time=0.809 ms
64 bytes from 10.10.10.1: icmp_seq=3 ttl=64 time=1.31 ms
--- 10.10.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2026ms
rtt min/avg/max/mdev = 0.709/0.943/1.311/0.263 ms
```

This lab shows how routes can be exchanged between Linux hosts acting as BGP speakers. FRR also supports OSPF, RIP, EIGRP, etc.

Conclusion

Layer 3 routing in Linux transforms the operating system into a fully functional router capable of sophisticated traffic management across complex networks. By leveraging the kernel's native IP stack and tools like iproute2, Linux provides a robust, flexible platform for implementing both basic and advanced routing features, from static route configuration to dynamic protocols via FRR. The system's support for PBR, VRF isolation,

and ECMP load balancing enables enterprise-grade network architectures without specialized hardware.

The Linux routing subsystem excels in scenarios requiring customization, such as multi-homed networks, cloud deployments, and SDN environments, where traditional routers might lack the necessary flexibility. Its ability to integrate with modern technologies, containers, virtual networks, and IPv6 makes it future proof for evolving infrastructure demands.

While Linux routing may not match dedicated hardware in raw throughput for some high-performance use cases, its cost efficiency, programmability, and deep integration with the broader Linux ecosystem make it an indispensable tool for network engineers. Whether used for simple gateway functions or as part of a distributed routing fabric, Linux proves that advanced networking no longer requires proprietary solutions.

In the next chapter, we will continue our journey into firewalls and explore packet filtering and Network Address Translation.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 3

Firewalls and Packet Filtering

Introduction

In the previous chapters, we discussed various networking concepts and understood how hosts on a network communicate with each other. We looked at the networking protocols and discussed how packets are forwarded at Layer 2, how routing works at Layer 3, and how to configure those services on Linux. In this chapter, we will explore the realm of network security and understand what network security features are provided by Linux.

Networking security is a fundamental aspect of system administration. One of the most powerful tools available for managing network traffic in Linux is iptables. As a user-space utility program, iptables allows administrators to configure the rulesets that govern packet filtering, **Network Address Translation** (**NAT**), and other packet manipulations in the Linux kernel. iptables operates by interacting with the netfilter framework, which is the subsystem in the Linux kernel responsible for controlling network packets as they traverse various stages of the network stack. Understanding iptables is essential for anyone looking to secure a Linux system.

One of the key strengths of iptables is its flexibility. It can be used for simple tasks, like blocking an **Internet Protocol** (**IP**) address, or for complex configurations involving port forwarding, NAT, and stateful packet

inspection.

In recent years, newer alternatives like nftables have emerged as successors to iptables, offering a more streamlined syntax and improved performance. However, iptables remains widely used, with many existing systems relying on its mature and battle-tested functionality.

Structure

This chapter will cover the following topics:

- Core concepts of iptables
- Structure of iptables configuration
- Basic iptables commands and syntax
- Connection tracking in iptables
- Network Address Translation
- Using ipsets with iptables
- The DShield project
- Evolution from iptables to nftables
- Linux firewall management with firewalld

Objectives

By the end of this chapter, you will develop a comprehensive understanding of how to configure, manage, and troubleshoot Linux firewall rules to control network traffic effectively. By mastering iptables, you would be able to define and manipulate rules within various tables and chains, ensuring proper packet filtering, NAT, and traffic logging. Additionally, you will learn to implement security policies that protect systems from unauthorized access while allowing legitimate traffic, as well as gain proficiency in advanced features such as stateful firewalling, connection tracking, and rule optimization. You will gain practical skills, such as writing and testing iptables rules in real-world scenarios, which are essential to avoid common pitfalls and ensure reliable network security.

Ultimately, the goal is to equip the reader with the expertise needed to design and maintain robust firewall configurations in Linux environments.

Core concepts of iptables

In the world of Linux networking and security, iptables stands as one of the most powerful and widely used firewall management tools. It serves as the front-end interface for configuring the netfilter framework, a Linux kernel subsystem that controls how network packets are processed, filtered, and modified. For system administrators, security professionals, and network engineers, mastering iptables is essential for enforcing security policies, preventing unauthorized access, and managing network traffic efficiently.

Iptables is a command-line utility that allows users to define rules for packet filtering, NAT, and packet mangling. These rules determine how incoming, outgoing, and forwarded traffic should be handled by the Linux kernel. Unlike application-level firewalls, iptables operates at the network layer, giving it granular control over IP packets before they reach applications or services.

Netfilter provides hooks at different stages of packet processing (e.g., when a packet arrives, before it is routed, or before it leaves the system), allowing iptables rules to intercept and manipulate traffic at these critical points.

Despite the emergence of newer alternatives like nftables (its successor) and firewalld (a higher-level firewall manager), iptables remains relevant for several reasons:

- **Widespread adoption**: Many production servers, cloud instances, and embedded systems still rely on iptables.
- **Granular control**: It offers fine-tuned rule definitions, making it ideal for complex security requirements.
- **Scriptability**: Rules can be automated via shell scripts, making iptables useful in DevOps and infrastructure as code environments.
- **Fundamental knowledge**: Understanding iptables provides a strong foundation for learning modern firewall technologies.

Structure of iptables configuration

The iptables configuration is structured into individual rules, which are composed of a match condition and an action to be performed when a packet matches the condition. The individual rules are grouped into chains. A network packet is matched against rules in a chain in a sequential order. Finally, the chains are associated with tables.

To effectively work with iptables, it is crucial to understand its three fundamental building blocks: rules, chains, and tables. These components work together to control network traffic, but it's often easiest to start with the smallest unit (rules) and expand to the broader structures (chains and tables) that organize them.

The following figure shows the organization of firewall rules into chains and how multiple chains compose a firewall table:

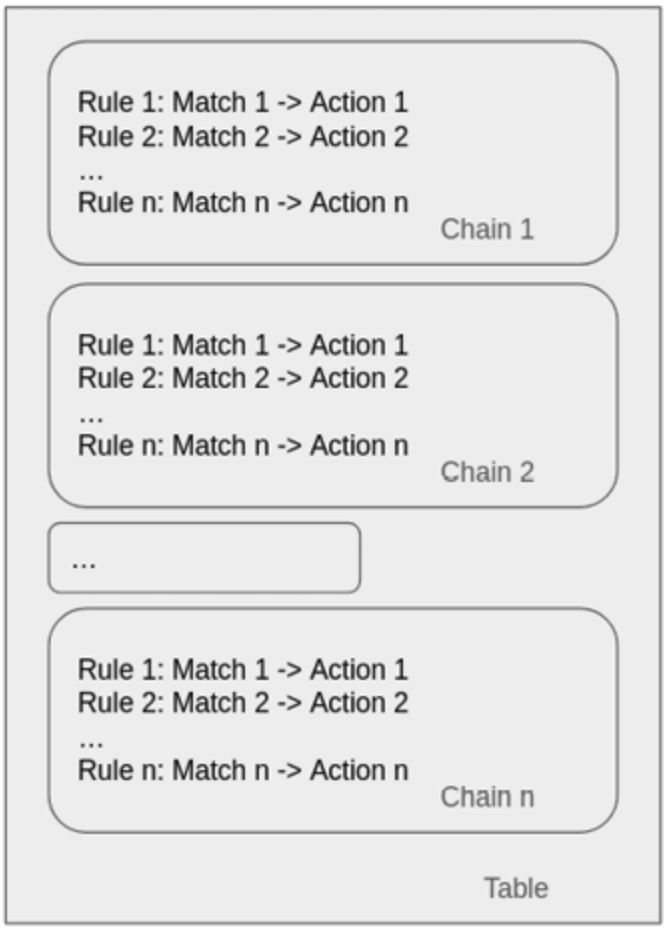


***Figure 3.1**: iptables rule organization*

Firewall rules

At the most basic level, iptables operates through rules, which define what should happen to network packets. Each rule consists of two main parts—a matching condition (such as a source IP, destination port, or protocol) and a target action (what to do if the packet matches). Common targets include the following:

- **ACCEPT**: Allow the packet.
- **DROP**: Silently discard the packet.
- **REJECT**: Block the packet and send an error response.
- **LOG**: Record the packet for debugging.

Rules are evaluated in order, so their sequence is critical. If the packet does not match the rule, the next rule is tried in the sequence. Once a packet

matches a rule, the specified action is taken, and no further rules in that chain are checked. An exception to this flow is the Jump action, which specifies another chain of rules that the packet should be matched against.

Firewall chains

Rules do not exist in isolation; they are grouped into chains, which determine when and how packets are inspected. Iptables includes several built-in chains. The details are as follows:

- **INPUT**: Processes incoming packets destined for the local system.
- **OUTPUT**: Handles packets generated by the local system.
- **FORWARD**: Manages packets being routed through the system, used in gateways.
- **PREROUTING**: Alters packets before routing decisions, used for NAT or marking.
- **POSTROUTING**: Modifies packets after routing, used for source NAT.

Chains process packets in a linear fashion, checking each rule in sequence. Administrators can also create custom chains to organize complex rule sets more efficiently.

Firewall tables

While chains group rules, tables group chains based on their purpose. Each table contains a specific set of chains and is used for different types of packet manipulation. The details are as follows:

- **Filter**: The default table for basic packet filtering, like accepting or blocking traffic.
- **Nat**: Handles NAT, e.g., port forwarding, IP masquerading.
- **Mangle**: Used for specialized packet alterations, e.g., modifying headers or QoS marking.
- **Raw**: Bypasses connection tracking for performance-sensitive traffic.

When a packet is processed, it moves through chains in different tables depending on its path. For example, an incoming packet first hits the mangle tables PREROUTING chains before reaching the filter table's

INPUT chain. The following figure shows the traversal of a packet and the netfilter tables it passes through at various stages of forwarding. The netfilter tables and their chains vary depending on whether the packet is destined for a local process or is to be forwarded.

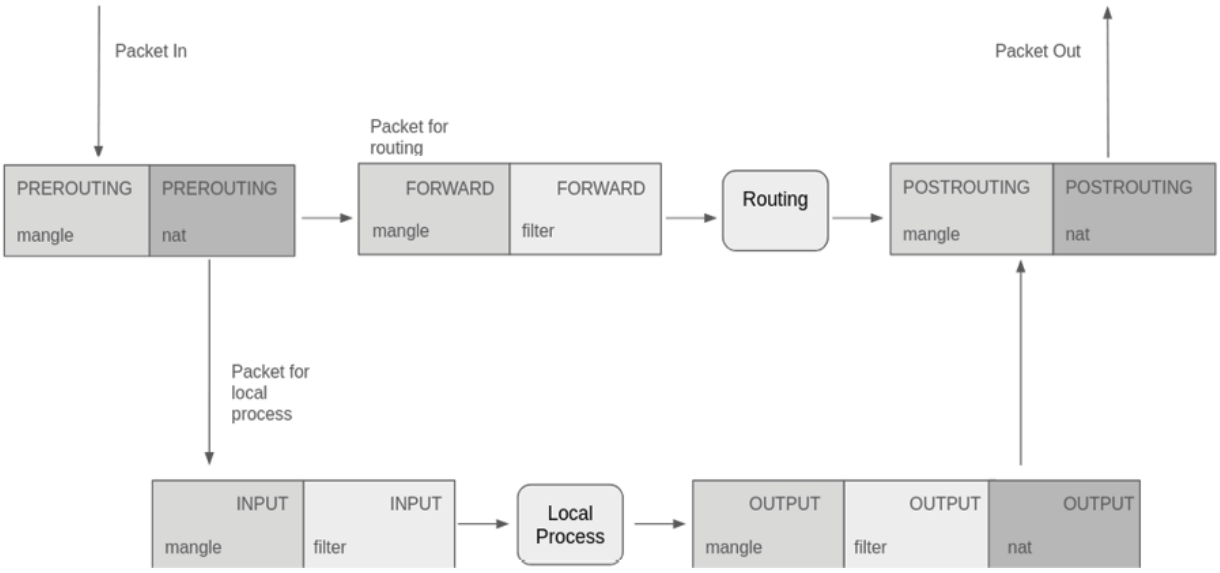


***Figure 3.2**: iptables packet flow for local and forwarding*

Understanding the hierarchy, rules within chains, and chains within tables helps build a clear foundation for learning iptables. The following sections will explore how to define and manage these components in practice.

Basic iptables commands and syntax

Mastering iptables begins with understanding its fundamental commands and syntax structure. This section provides a comprehensive guide to the essential commands you will use daily when working with Linux firewalls. Unlike graphical firewall tools, iptables operates entirely through the command line, requiring correct syntax to implement effective network security policies.

Command structure overview

Every iptables command follows a consistent pattern with three main

components:

- Table specification (optional, defaults to filter table)
- Chain selection (INPUT, OUTPUT, FORWARD, etc.)
- Rule parameters and targets (matching criteria and actions)

Essential command categories

The following are the categories of **iptables** commands for viewing and managing firewall rules:

- **Viewing current rules**: Before making changes, let us inspect existing rules. The most common viewing commands include:

```
iptables -L          # Lists all rules in the filter table
iptables -L -v       # Shows verbose output with packet counters
iptables -L -n       # Displays numeric output (no DNS resolution)
iptables -t nat -L   # Views rules in the NAT table
iptables -S          # Outputs rules in iptables-save format
```

  Here is an example of the output:

```
chandan@ubuntu24:~$ sudo iptables -L -t filter -n -v
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source               destination

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

Chain OUTPUT (policy ACCEPT 0 packets, 0 bytes)
pkts bytes target     prot opt in     out     source               destination
chandan@ubuntu24:~$
```

- **Rule management commands**: These are the core commands for manipulating firewall rules.

  The basic command format is:

```
iptables [-t table] <command> <chain> <rule parameters> -j
```

<target>

The table parameter is optional; if not specified, the filter table is assumed by the command.

Let us discuss some common rule parameters for matching a packet. When creating rules, these parameters help define matching criteria:

```
-p tcp/udp/icmp          # Protocol specification
--dport 80               # Destination port
--sport 1024:65535       # Source port range
-s 192.168.1.0/24        # Source network
-d 10.0.0.5              # Destination IP
-i eth0 -o eth1          # Input and output interfaces
```

This is not an exhaustive list of match parameters, and the full list of parameters can be found in the main page of iptables.

Let us now configure the rules in the **INPUT** chain of the filter table by executing the following commands:

```
$ sudo iptables -A INPUT -p tcp --dport 22 -j ACCEPT
$ sudo iptables -I INPUT 1 -s 192.168.1.100 -j DROP
```

The first example adds a rule with the **-A** command. As the table is not specified, the rule is added to the **INPUT** table of the filter table. The **-A** command adds a rule to the end of the chain. It matches the TCP protocol and destination port 22, which is the port used by the **Secure Shell** (**SSH**) service. The parameter **-j** specifies the action, which in this case is to **ACCEPT** the packet.

The next example uses the **-I** command with the **INPUT** 1 parameter. This command inserts a rule at position 1 of the **INPUT** chain instead of appending to the end of the chain.

After executing the preceding command, we can check the state of the **INPUT** chain for the filter table:

```
$ sudo iptables -L -t filter -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
```

```
DROP       0    --  192.168.1.100       0.0.0.0/0
ACCEPT     6    --  0.0.0.0/0           0.0.0.0/0           tcp dpt:22

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

You can notice that even though the second example was executed after the first, the rule is inserted at position **1** as **-I INPUT** 1 parameter was used. The **-v** parameter with the list command **-L** will show the packet count, which is the number of packets that have matched the rule.

The following shows an example output:

```
chandan@ubuntu24:~$ sudo iptables -L -t filter -n -v
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       0    --  *      *       192.168.1.100        0.0.0.0/0
  428 37516 ACCEPT     6    --  *      *       0.0.0.0/0            0.0.0.0/0
tcp dpt:22
```

The **-Z** command can be used to reset this counter, which can be very helpful in debugging rules. For example, the following command will reset the counter for the rule at position 2 of the **INPUT** chain:

```
$ sudo iptables -Z INPUT 2
```

You can see the packet counter has now restarted the count in the following output:

```
chandan@ubuntu24:~$ sudo iptables -L -t filter -n -v
Chain INPUT (policy ACCEPT 9 packets, 588 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       0    --  *      *       192.168.1.100        0.0.0.0/0
  145 12740 ACCEPT     6    --  *      *       0.0.0.0/0            0.0.0.0/0
tcp dpt:22
```

The **-D** command can be used to delete a rule in a chain for a table. For example, the following deletes the rule at position 1 in the **INPUT** chain of the filter table:

```
iptables -D INPUT 1
```

The complete rule specification can also be used to match the rule while deleting the rule. E.g., the following will remove the matching rule:

```
iptables -D INPUT -p tcp --dport 22 -j ACCEPT
```

The **-R** command can be used to replace the rule. This example replaces the rule at position 2 of the **OUTPUT** chain with a rule to **DROP icmp** packets:

```
iptables -R OUTPUT 2 -p icmp -j DROP
```

The following are the **iptables** commands for managing firewall chains and persisting them across reboots:

- **Chain management**: These commands control the chains themselves. A new chain can be created with the **-N** command, and the **-X** command can be used to delete the chain, as shown in the following code:

  ```
  iptables -N CUSTOM_CHAIN
  iptables -X CUSTOM_CHAIN
  ```

  A custom chain can be used to hold rules for a specific use case. For example, if you have multiple interfaces in a machine and you want to handle the rules for different interfaces in their own chains, in this case, a custom chain can be very handy. This also makes updating the rules for a specific interface.

  A chain can be used as an action in a rule with the **–j** parameter. When a packet matches a rule that specifies a chain as its action. The packet is then matched against rules in the specified chain.

  The **-F** command can be used to remove all rules from a chain.

  ```
  iptables -F INPUT
  ```

- **Persistence commands**: **iptables** rules do not survive reboots by default. The following command can be used to maintain persistence by saving the state of the iptable into a file; the same file can be used to

restore the state of the **iptables** configuration after a reboot.

```
iptables-save > /etc/iptables.rules     # Saves current rules
iptables-restore < /etc/iptables.rules  # Restores saved rules
```

Example use cases for iptables

Let us look at some examples where iptables can be used to specifically open ports for a given service.

In the following example, we open ports for a basic Web Server Protection. Once applied, these rules will only allow packets that are connecting to ports 80 **Hypertext Transfer Protocol** (**HTTP**) and 443 **Hypertext Transfer Protocol Secure** (**HTTPS**). It is a good practice to add a default deny as the last rule of a chain. This ensures that no port other than the ones specifically opened is blocked.

Note: Be careful with this example, as the last command will block access to other ports, including SSH, if you are connected remotely over SSH, you will be locked out of the system.

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
iptables -A INPUT -p tcp --dport 443 -j ACCEPT
iptables -A INPUT -j DROP  # Default deny policy
```

Here is another example of protecting SSH access to your machine; in this case, the SSH access is restricted to only hosts connecting from the **Classless Inter-Domain Routing** (**CIDR**) 192.168.1.0/24.

```
iptables -A INPUT -p tcp --dport 22 -s 192.168.1.0/24 -j ACCEPT
iptables -A INPUT -p tcp --dport 22 -j DROP
```

The following list highlights the things to keep in mind while working with iptable rules:

- **Rule order matters**: **iptables** processes rules sequentially until it finds a match.
- **Persistence**: Forgetting to save rules means they will disappear after reboot.
- **Locking out access**: Adding restrictive rules before permitting management access.

- **Lack of specificity**: Not specifying details of a rule, as interfaces can lead to unexpected behavior.

These basic commands provide the foundation for more advanced iptables configurations.

Connection tracking in iptables

Connection tracking, also known as **connection tracking** (**conntrack**), is a fundamental netfilter feature that enables iptables to function as a stateful firewall. Unlike simple packet filtering that examines each packet in isolation, conntrack maintains awareness of network connections, allowing for more intelligent and secure traffic management.

Refer to the following list to understand how the conntrack system works. When a packet arrives, the conntrack system:

- Classifies the packet as part of a new or existing connection.
- Tracks the connection state (NEW, ESTABLISHED, RELATED, INVALID).
- Updates connection timers and status information.
- Provides this state information to iptables rules.

Connection states define the state of a connection. A connection can start as NEW when the connection is initiated, and on completing the TCP handshake, the connection is classified as established. The following states show the classification of connections by the conntrack system:

- **NEW**: First packet of a connection, e.g., **Synchronize** (**SYN**) packet in TCP.
- **ESTABLISHED**: Recognized as part of an existing connection.
- **RELATED**: Associated with an established connection [like **File Transfer Protocol** (**FTP**) data channels or **Internet Control Message Protocol** (**ICMP**) errors].
- **INVALID**: Packets that do not make sense in the current context (malformed, suspicious).

The conntrack state can be used to match packets in iptables with the --

cstate parameter. For example, the following commands show basic stateful firewall rules:

```
iptables -A INPUT -m conntrack --ctstate
ESTABLISHED,RELATED -j ACCEPT
iptables -A INPUT -m conntrack --ctstate INVALID -j DROP
```

These rules **ACCEPT** any packet from connections that are already in **ESTABLISHED** state and **DROP** packets classified as **INVALID** state.

To view active connections tracked by the conntrack system, use the following command:

```
$ conntrack –L
```

Here is an example output of the command:

```
chandan@ubuntu24:~$ sudo conntrack -L
tcp      6 26 TIME_WAIT src=192.168.122.1 dst=192.168.122.61
sport=51778 dport=22 src=192.168.122.61 dst=192.168.122.1
sport=22 dport=51778 [ASSURED] mark=0 use=1
tcp      6 299 ESTABLISHED src=192.168.122.1 dst=192.168.122.61
sport=49376 dport=22 src=192.168.122.61 dst=192.168.122.1
sport=22 dport=49376 [ASSURED] mark=0 use=1
conntrack v1.4.8 (conntrack-tools): 2 flow entries have been shown.
```

To monitor connection events in real-time, use the **–E** parameter:

```
conntrack –E
chandan@ubuntu24:~$ sudo conntrack -E
    [NEW] tcp      6 120 SYN_SENT src=192.168.122.1
dst=192.168.122.61 sport=51778 dport=22 [UNREPLIED]
src=192.168.122.61 dst=192.168.122.1 sport=22 dport=51778
[UPDATE] tcp      6 60 SYN_RECV src=192.168.122.1
dst=192.168.122.61 sport=51778 dport=22 src=192.168.122.61
dst=192.168.122.1 sport=22 dport=51778
[UPDATE] tcp      6 432000 ESTABLISHED src=192.168.122.1
dst=192.168.122.61 sport=51778 dport=22 src=192.168.122.61
dst=192.168.122.1 sport=22 dport=51778 [ASSURED]
[UPDATE] tcp      6 120 FIN_WAIT src=192.168.122.1
```

```
dst=192.168.122.61 sport=51778 dport=22 src=192.168.122.61
dst=192.168.122.1 sport=22 dport=51778 [ASSURED]
[UPDATE] tcp      6 30 LAST_ACK src=192.168.122.1
dst=192.168.122.61 sport=51778 dport=22 src=192.168.122.61
dst=192.168.122.1 sport=22 dport=51778 [ASSURED]
[UPDATE] tcp      6 120 TIME_WAIT src=192.168.122.1
dst=192.168.122.61 sport=51778 dport=22 src=192.168.122.61
dst=192.168.122.1 sport=22 dport=51778 [ASSURED]
```

Network Address Translation

NAT is a networking technique that enables the modification of IP address and port information in packet headers as they traverse a routing device. In the context of iptables, NAT functionality is implemented through the NAT table, which contains specialized chains for altering packet addressing at different stages of processing. This capability serves as the backbone for several critical network operations that have become essential in modern networking environments.

The primary purpose of NAT is to solve the problem of IP address scarcity by allowing multiple devices on a private network to share a single public IP address when communicating with external networks. When an internal host initiates a connection to an external server, the NAT gateway modifies the source address in the outgoing packets, replacing the private IP address with its own public address. As response packets return, the NAT device reverses this translation, ensuring packets reach the correct internal host. This process, known as **Source NAT** (**SNAT**) or masquerading, occurs in the POSTROUTING chain, where packets are processed after routing decisions are made.

Source NAT

Source NAT is used when the source IP of the packet is in a non-routable network range and should not be exposed to the external world. Source NAT has the following properties:

- Changes the source address of outgoing packets.
- Typically used for internal clients accessing the internet.

The following figure describes a typical source NAT setup with the address translation applied at the gateway:

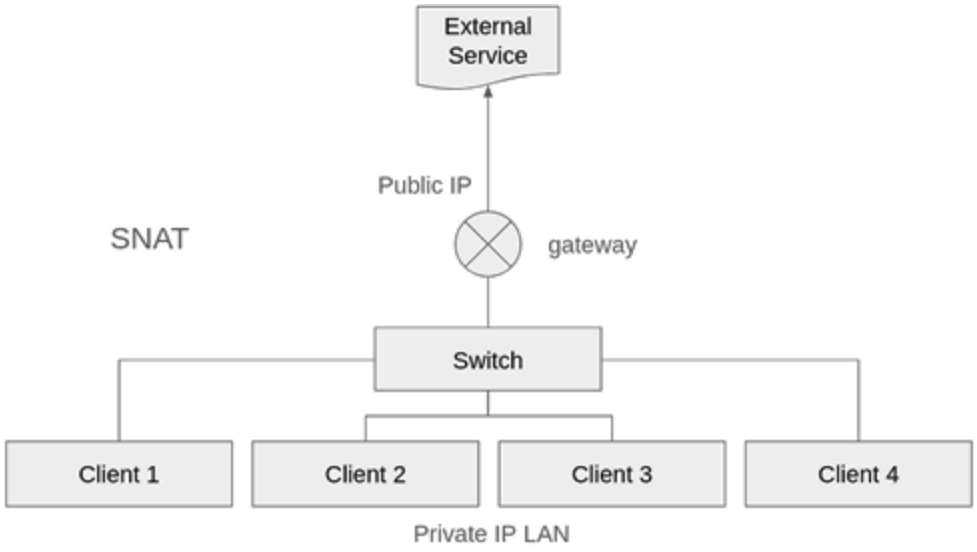


***Figure 3.3**: Source NAT*

Here is an example of SNAT, which sets the source IP of the packet going out of the router to 49.207.58.13:

```
iptables -t nat -A POSTROUTING -o eth0 -j SNAT --to-source 49.207.58.13
```

The **MASQUERADE** target offers a specialized form of SNAT for dynamic environments where the external IP address may change, such as **Dynamic Host Configuration Protocol** (**DHCP**) assigned interfaces or dial-up connections. Unlike static SNAT, which requires specifying a fixed IP address, **MASQUERADE** automatically uses whatever address is assigned to the output interface, making it ideal for residential gateways or mobile deployments.

MASQUERADE (Dynamic SNAT) has the following properties:

- Automatically uses the interface's IP address.
- Ideal for dynamic IP connections (like DHCP).

To use Dynamic SNAT, we can use the following command with the

MASQUERADE action:

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

In this case, the source address of the packet going out will be modified to the IP address configured on eth0.

Destination NAT (**DNAT**) performs the inverse operation, typically used to expose internal services to external networks. When configured in the PREROUTING chain, DNAT alters the destination address of incoming packets before routing occurs, enabling features like port forwarding. For example, an external request to a router's public IP on port 80 can be transparently redirected to an internal web server's private IP address. This capability makes services hosted on private networks accessible from the internet while maintaining network segregation.

Destination NAT has the following properties:

- Changes the destination address of incoming packets.
- Used for port forwarding and load balancing.

The following figure illustrates destination NAT:

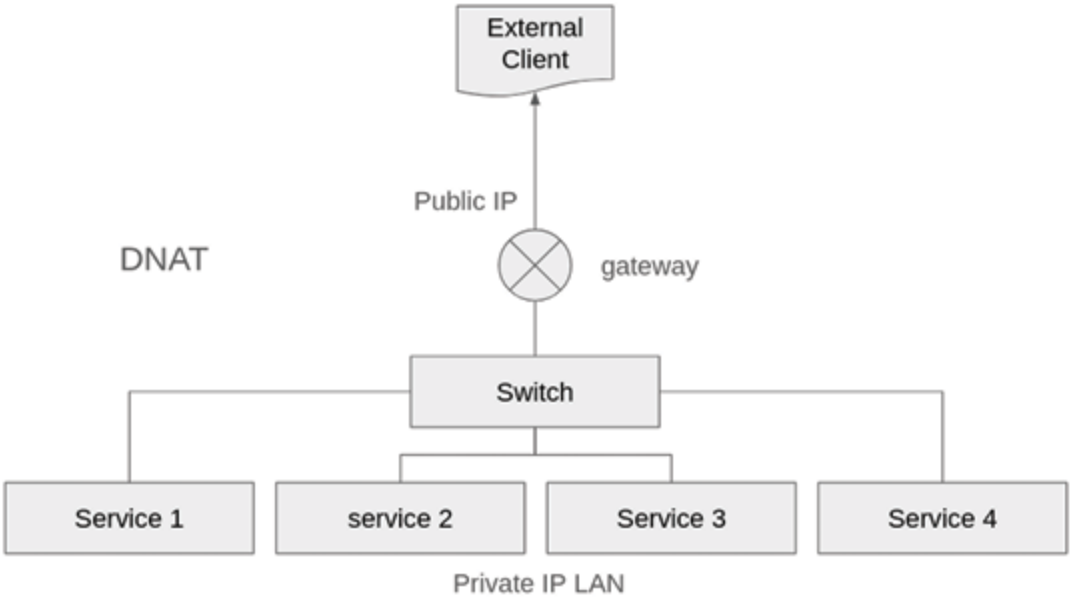


***Figure 3.4**: Destination NAT*

To configure DNAT for exposing a service on an internal host, we can use the following command:

```
iptables -t nat -A PREROUTING -p tcp --dport 80 -j DNAT --to-destination 192.168.1.10
```

In this case, a service running on port 80 on an internal host with an IP address of 92.168.1.10 can be accessed through the public IP of the gateway.

conntrack works in tandem with NAT to maintain the state of translated connections. The system records each NAT mapping and uses this information to properly reverse translations for reply packets. This stateful approach ensures bidirectional communication works seamlessly despite address modifications. The connection tracking system also handles protocol-specific complexities, particularly for applications like FTP or VoIP that embed addressing information within packet payloads.

Beyond basic address translation, iptables' NAT implementation supports more advanced scenarios. Load balancing can be achieved by distributing incoming connections across multiple servers through DNAT rules. Network redundancy becomes possible by maintaining multiple NAT paths with different upstream providers. The NAT system also interacts with QoS mechanisms, allowing prioritized handling of specific translated flows.

Let us explore an example of a load balancer setup with DNAT:

1. First, we create a custom chain **WEIGHTED_LB**:

   ```
   iptables -t nat -N WEIGHTED_LB
   ```

2. Then we add the traffic distribution for load balancing with 50% distribution between the two servers:

   ```
   iptables -t nat -A WEIGHTED_LB -m statistic --mode random --probability 0.5 \
   -j DNAT --to-destination 192.168.1.101
   iptables -t nat -A WEIGHTED_LB -m statistic --mode random --probability 0.5 \
   -j DNAT --to-destination 192.168.1.102
   ```

3. Route incoming HTTP/HTTPS traffic:

   ```
   iptables -t nat -A PREROUTING -p tcp --dport 80 -j WEIGHTED_LB
   ```

```
iptables -t nat -A PREROUTING -p tcp --dport 443 -j WEIGHTED_LB
```

4. Enable masquerading:

```
iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

NAT configuration requires careful consideration of packet flow through the various chains. **PREROUTING** changes affect both forwarded and locally destined packets, while **POSTROUTING** modifications apply to both routed and locally generated outgoing traffic. Understanding these subtleties is crucial when designing complex network architectures involving multiple NAT operations.

While NAT provides important benefits, it also introduces certain limitations and considerations. Some protocols require special helpers to function correctly through NAT. Performance overhead increases with the number of concurrent translations being maintained. Debugging NAT-related issues often requires examining both the NAT rules and the connection tracking table to fully understand translation behaviour.

Using ipsets with iptables

Ipsets represent a powerful extension to iptables that solves a critical performance challenge in large-scale firewall implementations. Traditional iptables rules require linear processing of individual rules for IP address matching, which becomes increasingly inefficient when dealing with hundreds or thousands of IP addresses. Ipsets introduce a specialized framework for storing and matching large collections of network attributes in optimized data structures, dramatically improving firewall performance.

An ipset is essentially a structured database of network elements. IP addresses, networks, ports, or combinations thereof that can be referenced as a single entity within iptables rules. The system implements various set types optimized for different matching scenarios, including hash-based storage for IP addresses, networks, or port numbers. When iptables encounters a rule referencing an IPset, it performs a single efficient lookup against the entire collection rather than processing numerous individual

rules.

The Linux kernel's netfilter framework includes special hooks for IPset integration, allowing matches to occur at the packet processing layer with minimal overhead. Sets maintain their contents in kernel memory, with userspace tools providing management interfaces. This architecture enables matching operations to complete in constant time (O(1)) regardless of set size, a critical advantage over linear rule processing.

Ipsets support multiple data types and matching modes that extend beyond simple IP address lists. The system can store combinations of IP addresses with port numbers and protocols, enabling complex matching criteria in a single rule. Dynamic sets allow runtime modifications without requiring firewall rule changes. This allows administrators to add or remove entries while maintaining active traffic filtering.

A typical ipset deployment involves three phases:

- **Set creation**: Define the set's parameters, including type and storage method:

```
ipset create malicious_hosts hash:ip timeout 86400
```

- **Population**: Add network elements to the set:

```
ipset add malicious_hosts 192.0.2.45
ipset add malicious_hosts 203.0.113.0/24
```

- **iptables integration**: Reference the set in firewall rules:

```
iptables -A INPUT -m set --match-set malicious_hosts src -j DROP
```

The efficiency gains of Ipsets become particularly evident in scenarios requiring:

- Large blocklists (thousands of malicious IPs)
- Frequent updates (dynamic threat intelligence feeds)
- Complex matching (IP+port combinations)
- High traffic volumes (servers under DDoS attack)

Types of ipsets

Ipsets provide specialized data structures for efficiently storing and

matching groups of network-related elements in iptables/nftables rules. The different ipset types optimize for various firewall use cases by using distinct storage and lookup methods.

Hash-based ipsets

Hash ipsets use hash tables for O(1) lookup times, making them ideal for large collections of individual elements. These come in several variants, some of which are as follows:

- **hash:ip**: Stores individual IP addresses (IPv4 or IPv6) with no additional data. Useful for maintaining blocklists or allowlists of specific hosts. This can be used for blocking known malicious IPs.
- **hash:net**: Stores network ranges in CIDR notation (e.g., 192.168.1.0/24). More memory-efficient than individual IPs when dealing with entire networks. Commonly used for geo-blocking or organizational access control.
- **hash**:**ip,port**: Stores combinations of IP addresses and port numbers. Each entry pairs an IP with a specific port (e.g., 192.168.1.5,80). Useful for service-specific filtering.
- **hash**:**net,port**: Extends **hash:net** by including port numbers with network ranges. Enables rules like *block all SSH access from this entire network range*.
- **hash**:**ip,port,ip**: Stores tuples of (source IP, port, destination IP). Used for complex scenarios involving specific communication paths.
- **hash**:**ip,port,net**: Similar but with destination networks instead of IPs (source IP, port, destination network).
- **hash**:**mac**: Stores MAC addresses instead of IPs. Useful for layer-2 filtering on local networks.

List-based ipsets

list:set is a list of ipsets. Stores other ipsets as elements, creating nested set structures. Allows building hierarchical policies (e.g., combining country blocks with service-specific rules).

Special purpose ipsets

The following special purpose ipsets can be used to check **ip** and **port** in a firewall rule:

- **bitmap**:**ip**: Uses a memory-efficient bitmap to represent IP ranges. Ideal for contiguous IP ranges (e.g., 192.168.1.0-192.168.1.255). This scheme provides a fixed size with lower memory overhead than **hash:net**.
- **bitmap**:**port**: Bitmap implementation for port numbers. Efficient for well-defined port ranges like ephemeral ports (32768-60999).

Timeout and metadata-enabled ipsets

Most hash-based types support optional timeout values per entry. This ipset will retain the individual entries for 300 seconds:

```
ipset create temp_block hash:ip timeout 300
```

You can add a counter to an ipset to get the number of packets that match the ipset:

```
ipset create tracked_hosts hash:ip counters
```

Similarly, a comment can be added to an **ipset:**

```
ipset create noted_ips hash:ip comment.
```

Testing ipsets

To ensure your ipsets are working correctly, we can run tests to validate the proper function of ipsets. To check if an IP is part of the **ipset**, use the following test command:

```
root@ubuntu24:~# ipset create test_block hash:ip
root@ubuntu24:~# ipset test test_block 8.8.8.8
8.8.8.8 is NOT in set test_block.
root@ubuntu24:~# ipset add test_block 8.8.8.8
root@ubuntu24:~# ipset test test_block 8.8.8.8
Warning: 8.8.8.8 is in set test_block.
```

To test ipsets with iptables, we can create an iptables rule to reference the

ipset. For example, in the following test, we create an **ipset** to block an IP and reference it in the iptables rule. The **ipset** is created with a timeout of 300 seconds, which means an entry added to this set will be part of the **ipset** only for 300 seconds. We can also add a counter to the **ipset**, which maintains a count of the number of lookups that matched a member. This can be done using the following commands:

```
root@ubuntu24:~# ipset create test_block hash:ip timeout 300 counters
root@ubuntu24:~# ipset add test_block 8.8.8.8
root@ubuntu24:~# iptables -A INPUT -m set --match-set test_block src -j DROP
```

To view the state of the **ipset**, use the list command as outlined here:

```
root@ubuntu24:~# ipset list test_block
Name: test_block
Type: hash:ip
Revision: 6
Header: family inet hashsize 1024 maxelem 65536 timeout 300 counters bucketsize 12 initval 0xf149a303
Size in memory: 312
References: 1
Number of entries: 1
Members:
8.8.8.8 timeout 208 packets 0 bytes 0
```

By running the list command again, you can check the change in state of the **ipset**. In this example, we note that the timeout for the member 8.8.8.8 is reducing, and the counter value is changed to reflect the number of matched lookups:

```
root@ubuntu24:~# ipset list test_block
Name: test_block
Type: hash:ip
Revision: 6
```

```
Header: family inet hashsize 1024 maxelem 65536 timeout 300 counters bucketsize 12 initval 0xf149a303
Size in memory: 312
References: 1
Number of entries: 1
Members:
8.8.8.8 timeout 105 packets 10 bytes 840
```

Benchmarks demonstrate orders-of-magnitude performance improvements compared to equivalent iptables-only implementations, especially for rulesets matching against extensive IP collections. The memory footprint also remains predictable regardless of set size due to the hash-based storage.

To run a test against a large set of IPs in a set and test input, the following example can be used:

```
root@ubuntu24:~# ipset create large_set hash:ip
root@ubuntu24:~# for i in {1..40}; do ipset add large_set 10.10.0.$i; done
root@ubuntu24:~# for i in {1..40}; do ipset add large_set 10.20.0.$i; done
root@ubuntu24:~# /usr/bin/time bash -c "for i in {1..1000}; do ipset test large_set 10.10.0.$((RANDOM%256)); done "
```

The time command will provide details of the **central processing unit** (**CPU**) and memory utilization of the test run.

Ipsets include robust management features, which are as follows:

- Automatic expiration of temporary entries.
- Counters for tracking matches per entry.
- Backup/restore functionality.
- Integrity checking for stored data.
- Max element limits to prevent memory exhaustion.

Administrators should monitor set memory usage and consider appropriate hash size parameters during creation to optimize performance. The system also supports binding sets to specific network interfaces when appropriate.

Even though they are powerful, ipsets require careful implementation:

- Kernel memory usage for large sets.

- Atomicity requirements during bulk updates.
- Synchronization needs in clustered environments.
- Validation requirements for dynamic updates.

Ipsets represent one of the most significant optimizations available for iptables deployments, solving fundamental scaling challenges while enabling new functionality. Their integration maintains compatibility with existing iptables rules while providing a path to handle network security requirements efficiently.

The DShield project

DShield represents one of the internet's collaborative cybersecurity initiatives, operating as the foundational data collection mechanism for the **Internet Storm Center** (**ISC**). Established during the early days of widespread internet attacks, this volunteer-powered platform has evolved into a critical early warning system that aggregates and analyzes security data from thousands of contributors across the globe. The project's fundamental premise revolves around the concept that shared security intelligence creates stronger collective defence. As participants contribute firewall logs, intrusion attempts, and malicious activity reports, the system processes this information to identify emerging threats and provide actionable insights back to the community.

At its technical core, DShield functions as a sophisticated data correlation engine that processes millions of security events daily from diverse sources, including enterprise firewalls, home network devices, and honeypots. The platform employs advanced normalization techniques to anonymize and standardize incoming data while preserving the tactical value of attack patterns. This processed information fuels multiple security products, including real-time attack maps that visualize global threat activity, dynamically updated blocklists of malicious IP addresses, and detailed analytical reports on emerging attack vectors. The system particularly excels at detecting distributed scanning activities, brute force campaigns, and vulnerability exploitation attempts by correlating seemingly isolated

events across its sensor network.

The practical outputs of DShield take multiple valuable forms. Security teams leverage its continuously updated blocklists to harden perimeter defenses, while researchers analyze its historical datasets to identify long-term attack trends. The platform's daily diary entries provide concise analysis of significant developments, often spotting new attack methods before they achieve widespread attention.

In the last section, we discussed ipsets. Let us set up a dynamic **ipset** to block IPs published by DShield blocklist. The list of blocked IPs is available at **https://www.dshield.org/block.txt**.

The header of the list defines the format of the data:

```
#    Columns (tab delimited):
#
#    (1) start of netblock
#    (2) end of netblock
#    (3) subnet (/24 for class C)
#    (4) number of targets scanned
#    (5) name of Network
#    (6) Country
#    (7) contact email address
#
```

The following steps describe how to set up a dynamic **ipset** to block IPs published by the DShield blocklist:

1. We first create an **ipset dshield_block** as follows:

   ```
   $ sudo ipset create dshield_block hash:ip hashsize 4096 maxelem 100000
   ```

2. Next, we will set up an iptables rule to block access from the IPs in the preceding **ipset**, using the following command:

   ```
   $ sudo iptables -I INPUT -m set --match-set dshield_block src -j DROP
   ```

3. Next, we will write a script to get the list of block IPs from DShield:

   ```
   #!/bin/bash
   ```

```
# DShield blocklist URL
BLOCKLIST_URL="https://www.dshield.org/block.txt"
TMP_FILE="/tmp/dshield_block.txt"

# Download latest list
wget "$BLOCKLIST_URL" -O "$TMP_FILE" || exit 1
# Create the main ipset if it does not exist.
sudo ipset list dshield_block >/dev/null 2>&1|| sudo ipset create dshield_block hash:ip hashsize 4096 maxelem 100000

# Create a temporary ipset (avoids locking the main set during updates)
sudo ipset create dshield_block_temp hash:ip hashsize 4096 maxelem 100000

# Add IPs to the temp set
grep -Eo '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' "$TMP_FILE" | while read -r ip; do
    sudo ipset add dshield_block_temp "$ip"
done

# Swap the temp set into production (minimizes downtime)
sudo ipset swap dshield_block_temp dshield_block

# Cleanup
sudo ipset destroy dshield_block_temp
rm "$TMP_FILE"

echo "DShield blocklist updated: $(date)" |sudo tee -a /var/log/dshield_ipset.log
```

4. We will save this script at **/usr/local/bin/update_dshield_blocklist.sh** and set up a cron job to schedule a run of this script to keep the **ipset** up-to-date. This can be done with:

```
$ sudo crontab -e
And adding the following entry to run the script every 8 hours
0 */8 * * * /usr/local/bin/update_dshield_blocklist.sh
```

In the script, we create a temporary **ipset** which we update with the list of IPs from the DShield block list and then swap the temporary ipset with the one referred by the iptables rule to reduce disruptions in traffic.

A typical output of the script **update_dshield_blocklist.sh** is as follows:

```
chandan@ubuntu24:~$ /usr/local/bin/update_dshield_blocklist.sh
--2026-02-07 06:51:04--  https://www.dshield.org/block.txt
Resolving www.dshield.org (www.dshield.org)... 172.67.70.195,
104.26.2.15, 104.26.3.15, ...
Connecting to www.dshield.org
(www.dshield.org)|172.67.70.195|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2109 (2.1K) [text/plain]
Saving to: '/tmp/dshield_block.txt'

/tmp/dshield_block.txt                100%
[=============================================
==================================>]   2.06K  --.-KB/s
in 0s

2026-02-07 06:51:04 (13.9 MB/s) - '/tmp/dshield_block.txt' saved
[2109/2109]

DShield blocklist updated: Sat Feb  7 06:51:04 AM UTC 2026
```

The cron job logs are captured by syslog, the default log file location is **/var/log/syslog** in Ubuntu and **/var/log/messages** in Rocky Linux. You should see logs like the following in your syslog file for every invocation of the script by cron.

CRON[4205]: (chandan) CMD (/usr/local/bin/update_dshield_blocklist.sh)

Evolution from iptables to nftables

The next-generation packet filtering subsystem nftables, has gradually replaced iptables as the default firewall solution in modern Linux distributions. The new nftables frontend to netfilter was designed to overcome fundamental architectural limitations in iptables while providing:

- A more streamlined syntax
- Better performance characteristics
- Enhanced scalability
- Unified management of IPv4/IPv6 rulesets
- Native support for sophisticated packet processing

At its core, nftables provides a more efficient, flexible, and unified approach to network traffic management while maintaining the familiar netfilter hooks in the Linux kernel.

The architecture of nftables differs significantly from its predecessor through its use of a virtual machine-based approach to packet processing. Rather than relying on linear rule evaluation like iptables, nftables compiles rules into bytecode that executes on a compact virtual machine within the kernel. This design enables more sophisticated packet processing without the performance penalties associated with complex iptables rulesets. The bytecode approach also allows for better optimization of rule evaluation and reduces context switching between kernel and userspace operations.

nftables families

One of the most notable improvements in nftables is its unified handling of multiple network protocols. Where iptables required separate utilities for IPv4 (iptables), IPv6 (ip6tables), nftables uses families to group protocols. The following are the currently supported families in nftables:

- **ip**: Family represents IPv4 packets.
- **ip6**: Family represents IPv6 packets.

- **inet**: Represents both IPv4 and IPv6.
- **arp**: Represents the layer2 packet flows.
- **bridge**: Represents packets flowing through a bridge device.
- **netdev**: Represents packets flowing through a single interface.

The main command for using nftables is **nft**. Unlike iptables, nftables do not have any predefined tables or chains.

nftables tables

Tables are containers for organizing rules. Here is an example table creation with nftables. To create the table, use the **nft** add table command as follows:

```
$ sudo nft add table inet filter
```

This command creates a table filter with family **inet** for handling IPv4/IPv6. You can list the tables with the **nft** list tables command:

```
$ sudo nft list tables
table ip filter
table inet filter.0
```

The table can be deleted with the **nft delete** command. The generic format of the table command is as follows:

```
nft (add | delete | flush) table [<family>] <name>
```

nftables chains

Chains are packet processing points with configurable types that determine the kind of processing done by the chain. The following types are supported:

- filter (standard processing)
- route (for packet rerouting)
- nat

A chain also refers to a hook, which represents the point in the processing of a packet when the chain is called. The following table outlines the hooks that are supported for the given family:

Family	Hooks

inet, ip, ip6	prerouting input forward output postrouting
arp	input output
netdev	ingress egress

***Table 3.1** : nftable hooks*

For example, here we are adding a chain called input to the filter table of the **inet** family. You can note that a chain also specifies the type of filter and priority 0, along with the hook:

```
$ sudo nft add chain inet filter input { type filter hook input priority 0 \; }
```

To list the chain, we can use the following **nft** list command:

```
$ sudo nft list chain inet filter input
table inet filter {
        chain input {
                type filter hook input priority filter; policy accept;
        }
}
```

nftables rules

Rules are the actual filtering instructions with enhanced matching capabilities:

```
$ sudo nft add rule inet filter input tcp dport {22, 80, 443} accept
$ sudo nft list chain inet filter input
table inet filter {
        chain input {
                type filter hook input priority filter; policy accept;
                tcp dport { 22, 80, 443 } accept
```

```
        }
}
```

To delete a rule, first find the handle name for the rule. This can be done using the **-a** option to the **nft** list command:

```
$ sudo nft -a list chain inet filter input
table inet filter {
        chain input { # handle 1
                type filter hook input priority filter; policy accept;
                tcp dport { 22, 80, 443 } accept # handle 3
        }
}
```

Now we can delete the rule using the following command:

```
$ sudo nft delete rule inet filter input handle 3
$ sudo nft -a list chain inet filter input
table inet filter {
        chain input { # handle 1
                type filter hook input priority filter; policy accept;
        }
}
```

The nftables syntax represents a clean break from **iptables** command-like syntax. Rules follow a more logical, expressive format that reduces the need for complex command-line options. A complete nftables configuration can be written in a declarative style like modern configuration languages, with support for variables, includes, and other features that improve readability and maintainability. The framework introduces native support for sophisticated data structures like sets, maps, and dictionaries, eliminating the need for external utilities like **ipset** in most scenarios. For example, here are the commands to set up an IP allow and block list with nftables with native support for sets. The following is an example of using native sets in nftables to get the same result as ipsets:

1. Create a filter table and add a chain input:

```
nft add table inet filter
nft add chain inet filter input { type filter hook input priority 0 \; }
```

2. Create sets for allowed and blocked IPs:

```
nft add set inet filter allowed_ips { type ipv4_addr \; flags interval \; }
nft add set inet filter blocked_ips { type ipv4_addr \; }
```

3. Populate the sets with IPs:

```
nft add element inet filter allowed_ips { 192.168.1.0/24, 10.0.0.0/8 }
nft add element inet filter blocked_ips { 45.227.253.109, 185.142.236.34 }
```

4. Add rules to accept allowed IPs and drop blocked ones:

```
nft add rule inet filter input ip saddr @allowed_ips accept
nft add rule inet filter input ip saddr @blocked_ips drop
```

5. View the chain configuration:

```
nft -a list chain inet filter input
table inet filter {
        chain input { # handle 1
                type filter hook input priority filter; policy accept;
                ip saddr @allowed_ips accept # handle 10
                ip saddr @blocked_ips drop # handle 11
        }
}
```

You can list the sets defined with the **list sets** command as follows:

```
nft list sets
table ip filter {
}
table inet filter {
        set allowed_ips {
                type ipv4_addr
                flags interval
```

```
        elements = { 10.0.0.0/8, 192.168.1.0/24 }
    }
    set blocked_ips {
        type ipv4_addr
        elements = { 45.227.253.109, 185.142.236.34 }
    }
}
```

To save the **nftable** configuration, you can use the following command:

```
nft list ruleset
```

The list of rules can be saved to the **/etc/nftables.conf** file, which is used by **nftables** service to restore the rules on reboot.

Performance characteristics of **nftables** demonstrate measurable improvements over traditional iptables. Independent benchmarks consistently show 2-3x faster rule evaluation, particularly for complex rule sets. Memory usage is significantly reduced, especially in deployments requiring large numbers of rules or extensive blocklists. The architecture also scales better with modern multi-core processors, distributing firewall processing more efficiently across available CPU resources.

Practical advantages of **nftables** become particularly apparent in complex deployment scenarios. The framework natively supports features like interval matching for IP ranges, time-based rules, and sophisticated connection tracking without requiring numerous kernel modules. Configuration management is simplified through atomic ruleset updates, where entire configurations can be replaced without temporary rule inconsistencies. The system also provides better integration with other Linux networking features like traffic control and network namespaces.

For administrators transitioning from iptables, **nftables** provides compatibility tools, including translation utilities that can convert existing iptables rules to **nftables** syntax. Major Linux distributions have embraced **nftables** as their default firewall solution, with RHEL 8+, Debian 10+, and Ubuntu 18.04+ all featuring native support.

The **nftables** framework continues to evolve, with recent developments

adding support for hardware offloading of certain operations through flow tables, enhanced NAT capabilities, and improved integration with Linux security modules. Its design positions it as the foundation for Linux packet filtering well into the future.

Linux firewall management with firewalld

Firewalld represents a significant evolution in Linux firewall management, designed to simplify network security configuration while maintaining robust protection. As the default firewall interface in Red Hat Enterprise Linux and its derivatives, firewalld operates as a dynamic firewall daemon that builds upon either **iptables** or **nftables** as its backend, abstracting away much of the complexity associated with direct rule management.

Unlike traditional static firewall configurations, firewalld introduces the concept of runtime and permanent configurations, allowing immediate application of temporary rules while maintaining a separate set of persistent rules that survive reboots. This dual-configuration approach enables network administrators to test rules before making them permanent, reducing the risk of accidental lockouts.

The system organizes firewall rules through zones, predefined security contexts that can be assigned to network interfaces or connections. Each zone contains a set of rules governing what services and ports are available, how masquerading and port forwarding are handled, and which sources are trusted. Common zones include:

- Public (default zone for untrusted networks)
- Trusted (allow all traffic)
- Home (slightly restricted for home networks)
- Internal (more permissive for internal networks)
- **Demilitarized zone systems** (**dmz**)

Traffic between different zones is blocked, while the traffic within a single zone is allowed by default. The following commands can be used to check the firewalld zones:

- To check the status of the firewalld service, use the following

command:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --state
running
```

- To get all zones, run the following command:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --get-zones
block dmz drop external home internal nm-shared public trusted work
```

- The active zones can be seen with the following command:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --get-active-zones
public
  interfaces: enp1s0 enp7s0
```

At its core, firewalld operates through several abstraction layers that simplify common firewall tasks:

- **Services**: Predefined configurations for common applications (HTTP, SSH, etc.).
- **Ports**: Direct port-based rules when services are not predefined.
- **Rich rules**: Advanced syntax for complex scenarios.
- **Direct** Interface: Escape hatch for custom iptables/nftables rules.

The following are some of the common examples of using the **firewall-cmd** to manage the firewall abstractions like services and ports described in the preceding section:

- To list the services that are allowed, use the command:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --list-service
cockpit dhcpv6-client ssh
```

- To allow connections to a service, use the following command:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --add-service=https
Success
```

- Now you can view the services that are allowed in the zone with the following command:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --list-service
```

```
cockpit dhcpv6-client https ssh
```

- You can also use the port parameter to add and check enabled ports:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --add-port=3020/tcp
success
[chandan@rocky9 ~]$ sudo firewall-cmd --list-port
3020/tcp
```

- You can use the **–list-all** command to list all services, zones, and policies:

```
[chandan@rocky9 ~]$ sudo firewall-cmd --list-services
public (active)
  target: default
  icmp-block-inversion: no
  interfaces: enp1s0 enp7s0
  sources:
  services: cockpit dhcpv6-client ssh
  ports:
  protocols:
  forward: yes
  masquerade: no
  forward-ports:
  source-ports:
  icmp-blocks:
  rich rules:
```

The **firewall-cmd** utility serves as the primary command-line interface, while graphical tools like **firewall-config** provide a user-friendly alternative. Under the hood, firewalld translates these high-level configurations into the appropriate iptables or nftables rules, automatically handling the backend implementation details.

Firewalld's most significant innovation is its support for runtime modifications without service interruptions. Traditional firewall solutions require flushing and reloading entire rule sets for changes to take effect,

potentially dropping existing connections. Firewalld maintains stateful connections during updates, applying changes incrementally while preserving established sessions.

The system also supports automatic zone assignment based on network interface characteristics, particularly useful for mobile devices moving between trusted and untrusted networks. When combined with NetworkManager integration, this allows dynamic firewall policy adjustments as network conditions change.

Conclusion

This chapter has explored the critical tools and frameworks that define network security in Linux, from the classic iptables to the modern nftables, along with complementary systems like ipsets and DShield. Each of these technologies represents a step forward in managing and securing network traffic, reflecting the evolving challenges of cybersecurity.

iptables laid the foundation, offering powerful packet filtering, NAT, and stateful inspection. However, its syntax and performance limitations led to the development of ipsets, which introduced efficient set-based matching for large blocklists and complex rules. Meanwhile, DShield demonstrated the power of collaborative security, aggregating global threat data to enhance protection beyond individual network boundaries.

The transition to nftables marks a significant leap forward, unifying firewall management under a more flexible, performant framework. With streamlined syntax, native support for sets, and better scalability, nftables addresses many of iptables' shortcomings while maintaining backward compatibility.

In the next chapter, we will continue our journey to explore tools and techniques of troubleshooting the network. We will understand how to test reachability, latency, and routing paths. Effective diagnostics will help us maintain reliability, optimize performance, and troubleshoot failures.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 4

Network Troubleshooting and Diagnostics

Introduction

Troubleshooting network issues on Linux can be a challenging yet essential skill for system administrators and users alike. Given the complexity of modern networks, problems can arise from various sources, including misconfigured settings, hardware failures, or connectivity disruptions. Linux provides a robust set of tools and commands to diagnose and resolve network-related problems efficiently. This chapter explores fundamental techniques for diagnosing network problems on Linux, covering common utilities such as ip, ping, traceroute, netstat, and many others. By following a structured approach, checking physical connections, verifying configurations, testing connectivity, and analyzing logs, administrators can pinpoint and resolve network issues effectively, ensuring reliable communication and service availability.

Structure

This chapter covers the following topics:

- Testing connectivity with ping
- Using traceroute to diagnose network issues
- Using My Traceroute for network troubleshooting
- Port and socket inspection
- Netstat
- Netcat
- lsof
- Troubleshooting using packet captures
- Tshark
- NetFlow

Objectives

This chapter aims to provide you with the knowledge and skills to diagnose and resolve common network issues on Linux systems. By the end of this chapter, you will be able to systematically troubleshoot connectivity problems, analyze network configurations, and apply appropriate fixes using essential Linux networking tools.

Testing connectivity with ping

Network connectivity is a fundamental aspect of system administration, and one of the most basic yet powerful tools for testing it is the **ping** command. Short for **Packet Internet Groper**, (**ping**) is used to verify whether a remote host is reachable and measure the round-trip time for messages sent between the source and destination. It operates by sending **Internet Control Message Protocol** (**ICMP**) Echo Request packets and waiting for Echo Reply responses. While ping is simple to use, it offers various options that enhance its functionality for troubleshooting network issues.

The simplest form of the **ping** command sends continuous ICMP packets to a specified host until manually stopped:

```
ping example.com.
```

Useful ping command options

The following list outlines some of the common and useful options used with ping:

- **Limiting the number of packets (-c)**: By default, **ping** runs indefinitely, but the **-c** (count) option allows specifying how many packets to send before stopping:

  ```
  ping -c 4 example.com
  ```

  This sends exactly 4 packets and then exits, providing a concise summary.

- **Setting timeout interval (-W)**: If a network is slow or unresponsive, ping waits for a reply before timing out. The **-W** (wait) option adjusts this timeout period (in seconds):

  ```
  ping -W 2 example.com
  ```

  Here, **ping** waits 2 seconds for each reply before considering the packet lost.

- **Adjusting packet size (-s)**: The default ICMP packet size is 56 bytes (64 bytes including headers). The **-s** option allows sending larger or smaller packets, useful for testing network performance or **maximum transmission unit** (**MTU**) issues:

  ```
  ping -s 1000 example.com
  ```

  This sends 1000-byte packets, helping identify fragmentation or latency problems.

  MTU is the largest size of a single data packet that can be transmitted over a network without fragmentation. It is measured in bytes and includes both the payload (data) and headers (e.g., IP, TCP, ICMP). The following are some scenarios of different MTU sizes:

  - Standard Ethernet MTU is 1500 bytes, which is most common for home and business networks.
  - Some high-performance networks support larger MTUs, e.g., 9000 bytes called **Jumbo Frames**.
  - VPNs, PPPoE, and some ISPs may reduce MTU; lower MTUs

accommodate extra headers used by VPNs and other services.

If a packet exceeds the MTU of a network link, it gets fragmented (split into smaller pieces) or dropped, leading to connectivity issues. Since **ping** sends ICMP packets, it can help detect MTU-related issues by testing different packet sizes.

Check the default MTU on your system using the following command:

```
$ ip link show
```

To test MTU with **ping**, use the -s (size) flag to set the payload size excluding headers. The total overhead due to headers is calculated as follows:

ICMP Header (8 bytes) + IP Header (20 bytes) = Total overhead (28 bytes)

To test if a 1500-byte MTU works, we need to use a payload of 1472 (MTU size excluding the headers):

```
$ ping -s 1472 -M do example.com
```

The **-M** flag with the do option prevents fragmentation of the packet by setting the Do not fragment flag.

If ping is successful, it means the network supports the tested MTU size. The test can be performed with various packet sizes to find the correct MTU supported by the network. E.g., if **ping -s 1452** works but **-s 1453** fails, actual *MTU = 1452 + 28 = 1480 bytes*.

Once found, the MTU can be set for the interface using the following command:

```
$sudo ip link set eth0 mtu 1480
```

- **Flood ping (-f)**: For stress testing, the **-f** (flood) option sends packets as fast as possible, which can help identify network bottlenecks. Flood ping testing technique sends ICMP packets as fast as possible to identify:
  - Bandwidth limitations
  - Router or switch performance issues
  - Packet loss under heavy load

- Network congestion points

sudo ping -f -c 1000 destination_ip_or_hostname

However, this requires root privileges and should be used cautiously, as this can cause disruption in the network. Also to be noted that this technique should only be used in networks administered by you; testing external networks with flood pings can be seen as a violation of the fair usage policy. Flood ping is particularly effective for:

- Identifying weak **Wireless Fidelity** (**Wi-Fi**) connections.
- Finding overloaded routers
- Testing QoS configurations.
- Benchmarking network upgrades.

Let us analyze the results of a typical **ping** command. The following results for a flood **ping** can provide some insights into the network:

1000 packets transmitted, 1000 received, 0% packet loss, time 2518ms

rtt min/avg/max/mdev = 1.289/4.143/140.627/13.111 ms, pipe 13, ipg/ewma 2.520/1.849 ms

The following information can be deciphered from the preceding result:

- **Packet transmission**: 1000 packets transmitted, 1000 received, 0% packet loss. No packets were lost during the test. Indicates stable physical connectivity and no major congestion.
- **Round-trip time (RTT)**: *min/avg/max = 1.289/4.143/140.627 ms*
- **Minimum latency**: 1.289ms shows ideal best-case.
- **Average latency**: 4.143ms, average metrics, which should be good for most applications.
- **Maximum latency**: 140.627ms denotes the spike value.
- **Jitter (mdev)**: mdev measures latency variability of the network; a lower value is better. 13.111 ms is moderate and suggests some network inconsistency.
- **Pipe:** It is the maximum number of outstanding packets.

- **Ipg/ewma:** Inter-packet gap and exponential weighted moving average. The inter-packet gap is the average time between sending consecutive ping packets. The value of 2.520ms means a packet was sent every IPG interval. In this case, in 1 second or 1000 milliseconds, 1000/2.520 (~ 396.82) packets were sent.

- **Setting time interval between packets (-i)**: The **-i** (interval) option controls the delay between sending each packet. The default is 1 second. For faster checks, a shorter interval can be set:

```
$ ping -i 0.5 example.com
```

Conversely, a longer interval (e.g., **-i 5**) reduces network load during prolonged tests.

- **Bypassing Domain Name System (DNS) (-n)**: The flag **-n** can be used to skip DNS resolution and display **Internet Protocol** (**IP**) addresses directly.
- **Time to live (TTL) settings (-t)**: The **-t** option sets the TTL value for outgoing packets, which determines how many hops (routers) the packet can traverse before being discarded:

```
$ ping -t 30 example.com
```

This can help diagnose routing loops or overly restrictive firewall rules. In the following example, we see how the **–t** option can be used to find the number of hops needed to reach the destination:

```
$ ping -c3 -t14 google.com
PING google.com (2404:6800:4002:824::200e) 56 data bytes
From 2404:6800:8112::1 icmp_seq=1 Time exceeded: Hop limit
From 2404:6800:8112::1 icmp_seq=2 Time exceeded: Hop limit
From 2404:6800:8112::1 icmp_seq=3 Time exceeded: Hop limit
```

While with 20 hops, the **ping** succeeds:

```
$ ping -c3 -t20 google.com
PING google.com (2404:6800:4002:824::200e) 56 data bytes
64 bytes from del12s07-in-x0e.1e100.net
(2404:6800:4002:824::200e): icmp_seq=1 ttl=118 time=62.3 ms
```

64 bytes from del12s07-in-x0e.1e100.net (2404:6800:4002:824::200e): icmp_seq=2 ttl=118 time=63.1 ms
64 bytes from del12s07-in-x0e.1e100.net (2404:6800:4002:824::200e): icmp_seq=3 ttl=118 time=64.3 ms

To summarize, the **ping** command is a versatile and essential tool for network diagnostics. While its basic usage is straightforward, its various options allow for deeper troubleshooting, from measuring latency to stress-testing connections.

Using traceroute to diagnose network issues

Network connectivity problems can be complex, involving multiple potential points of failure between a source and destination. The traceroute command is a fundamental tool for identifying where these failures occur by mapping the path packets take across a network. Unlike simpler tools like ping, which only confirm end-to-end connectivity, traceroute reveals the journey packets undergo, making it invaluable for diagnosing latency, packet loss, and routing failures.

Traceroute operates by exploiting the TTL field in IP packet headers. Each router that forwards a packet decrements the TTL by one. When the TTL reaches zero, the router discards the packet and sends back an ICMP **Time Exceeded** message. Traceroute begins by sending packets with a TTL of 1, forcing the first hop router to respond. It then increments the TTL with each subsequent packet, allowing it to discover each hop along the path until the destination is reached.

By default, traceroute sends **User Datagram Protocol** (**UDP**) packets, though modern implementations often support **Transmission Control Protocol** (**TCP**) and other protocols for bypassing firewalls that block traditional traceroute methods. Each hop is probed multiple times, providing latency measurements that help identify performance bottlenecks.

A typical traceroute output lists each hop along the route, showing:

- The hop number.
- The router’s hostname (if resolvable) and IP address.

- **Round-trip times** (**RTT**) for each probe packet.

Here is an example of running traceroute:

```
$ traceroute -n google.com
traceroute to google.com (142.250.183.238), 30 hops max, 60 byte packets
 1  192.168.122.1  0.107 ms  0.089 ms  0.083 ms
 2  192.168.29.1  1.741 ms  2.126 ms  2.361 ms
 3  10.17.40.1  7.458 ms  7.453 ms  7.448 ms
 4  172.16.80.129  12.448 ms 172.16.80.133  16.951 ms  15.945 ms
 5  192.168.157.34  16.096 ms  15.300 ms 192.168.157.30  14.633 ms
 6  192.168.232.229  14.267 ms  17.526 ms  17.519 ms
 7  192.168.232.194  18.692 ms 192.168.232.195  17.725 ms  16.758 ms
 8  192.168.148.144  15.787 ms  11.548 ms 192.168.148.148  13.117 ms
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  72.14.195.22  49.567 ms  47.210 ms 72.14.195.34  44.885 ms
15  209.85.252.44  45.733 ms * *
16  142.250.183.238  48.281 ms 142.251.54.92  47.223 ms
142.251.52.220  47.891 ms
```

Analyzing traceroute results

Analyzing the result for traceroute provides the following insights:

- **Detecting network congestion**: If a particular hop shows significantly higher latency than others, it may indicate congestion, an overloaded router, or a suboptimal routing path. Consistently high latency at multiple hops suggests general network congestion, while a single problematic hop points to a localized issue.
- **Detecting faulty devices or saturated link**: If some probes to a hop

time out (marked with *), it suggests packet loss at that router or the preceding link. Occasional timeouts may be normal due to rate-limiting, but consistent loss indicates a faulty device or saturated network link. This can also happen when Firewalls/**Access Control List** (**ACLs**) block ICMP/UDP traceroute probes, or if the routers are configured not to send *Time Exceeded* replies

- **Detecting network loop**: If the same IP address appears multiple times in succession, a routing loop may exist, causing packets to cycle endlessly between routers. This often appears as a repeating sequence of hops without ever reaching the destination.
- **Detecting asymmetric paths**: Networks do not always use the same path in both directions. If a traceroute to a destination succeeds but return traffic fails, an asymmetric routing issue may be present, often due to misconfigured routing policies. In the traceroute output, you can observe multiple IP addresses listed for some hops (e.g., hop 4: 172.16.80.129 and 172.16.80.133). This occurs due to load balancing and path redundancy in modern networks.

Traceroute is an indispensable tool for network diagnostics, offering visibility into the path and performance of traffic across complex networks. Here are some flags that can be used with traceroute, which might help diagnose network issues in certain situations, which are as follows:

- Many firewalls block ICMP/UDP traceroutes but permit TCP traffic. Tools like **tcptraceroute** or **traceroute -T** can use **Transmission Control Protocol Synchronize** (**TCP SYN**) packets (e.g., mimicking **Hypertext Transfer Protocol Secure** (**HTTPS**) traffic on port 443) to bypass these restrictions:

```
$ sudo traceroute -T -p 443 example.com
```

Here is an example of using TCP traffic for traceroute:

```
$ sudo traceroute -n -T -p 443 google.com
traceroute to google.com (142.250.183.238), 30 hops max, 60 byte packets
 1  192.168.122.1  0.209 ms  0.166 ms  0.135 ms
 2  192.168.29.1  26.450 ms  26.828 ms  26.812 ms
```

```
 3  10.17.40.1  27.395 ms  27.362 ms  27.346 ms
 4  172.16.80.133  27.343 ms  27.307 ms 172.16.80.129  27.437 ms
 5  192.168.157.34  27.420 ms 192.168.157.30  27.215 ms
192.168.157.28  27.201 ms
 6  192.168.232.229  27.165 ms  27.105 ms  27.075 ms
 7  192.168.232.195  27.225 ms 192.168.232.194  11.206 ms
192.168.232.195  11.640 ms
 8  192.168.148.144  12.241 ms 192.168.148.150  10.805 ms  13.202
ms
 9  * * *
10  * * *
11  * * *
12  * * *
13  * * *
14  209.85.148.118  45.448 ms 72.14.195.34  46.958 ms
209.85.148.118  44.928 ms
15  142.251.249.3  46.008 ms 172.253.69.191  46.917 ms  43.591 ms
16  142.251.76.173  51.663 ms 142.251.76.175  46.226 ms
142.251.76.173  43.922ms
17  142.250.183.238  46.121 ms  47.362 ms  43.486 ms
```

- Reverse DNS lookups slow down traceroute or clutter output. Disabling DNS resolution (**-n**) provides cleaner results:

```
$ sudo traceroute -n example.com
```

- Increasing the number of probes per hop (**-q**) helps identify intermittent issues, while adjusting packet size (**-s**) can test MTU-related problems:

```
$ traceroute -q 5 -s 1500 example.com
```

Limitations of traceroute

While traceroute is a powerful tool to diagnose network failures, it has limitations, which are as follows:

- **Firewalls and rate-limiting**: Many networks block ICMP or rate-limit

responses, causing false timeouts.

- **Load balancing**: Paths may change between probes, creating confusing output.
- **Incomplete paths:** Some routers never send *Time Exceeded* messages, creating *blind spots*.

By systematically analyzing each hop, administrators can pinpoint failures, optimize routing, and resolve connectivity issues that simpler tools like ping cannot detect. While it requires careful interpretation, especially in modern networks with load balancing and firewalls, its insights remain crucial for effective network troubleshooting.

Using My Traceroute for network troubleshooting

Network diagnostics often require more than just a one-time path analysis, especially when dealing with intermittent connectivity issues or fluctuating latency. **My Traceroute** (**MTR**) is a dynamic tool that continuously monitors network performance, making it superior for real-time troubleshooting. Unlike traditional traceroute, which provides a static snapshot of the network path, MTR updates statistics in real time, offering deeper insights into packet loss, latency variations, and routing instability.

MTR operates by sending a sequence of packets with incrementing TTL values, like traceroute, but instead of stopping after one pass, it repeatedly probes each hop along the path. This continuous sampling generates live statistics, including:

- RTT averages, minimums, and maximums.
- Jitter or latency variation across multiple probes.
- Packet loss percentage at each hop.

By default, MTR refreshes its output every second, allowing administrators to observe how network conditions evolve. This makes it particularly effective for detecting transient issues that might disappear in a single traceroute run.

To start MTR, simply run **mtr** hostname. E.g., the following listing runs **mtr** in report mode and numeric output:

```
$ mtr --report -n google.com
Start: 2025-07-12T05:25:17+0000
HOST: ubuntu24                    Loss%   Snt   Last   Avg  Best  Wrst StDev
  1.|-- 192.168.122.1             0.0%    10    0.3   0.3   0.2   0.4   0.1
  2.|-- 192.168.29.1              0.0%    10    2.2   6.0   1.6  41.6  12.5
  3.|-- 10.17.40.1                0.0%    10   27.6   7.3   3.0  27.6   7.2
  4.|-- 172.16.80.133              0.0%    10   10.3  11.1  10.1  12.4   0.8
  5.|-- 192.168.157.32             0.0%    10   10.4  10.4   9.6  11.5   0.6
      192.168.157.34
  6.|-- 192.168.232.229            0.0%    10   11.2  10.6   9.7  11.2   0.5
  7.|-- 192.168.232.194            0.0%    10    9.7  16.0   9.7  58.5  15.0
  8.|-- 192.168.148.144            0.0%    10   10.5  11.0  10.3  13.2   0.9
      192.168.148.146
  9.|-- ???                       100.0    10    0.0   0.0   0.0   0.0   0.0
 10.|-- ???                       100.0    10    0.0   0.0   0.0   0.0   0.0
 11.|-- ???                       100.0    10    0.0   0.0   0.0   0.0   0.0
 12.|-- ???                       100.0    10    0.0   0.0   0.0   0.0   0.0
 13.|-- ???                       100.0    10    0.0   0.0   0.0   0.0   0.0
 14.|-- 72.14.195.22              0.0%    10   48.5  46.6  44.8  48.5   1.1
 15.|-- 142.251.249.5             0.0%    10   47.6  46.4  45.2  47.6   0.8
 16.|-- 142.251.76.175            0.0%    10   44.8  46.5  44.8  47.6   0.9
 17.|-- 142.250.183.238           0.0%    10   44.9  45.6  44.4  47.2   1.0
```

For more detailed analysis, additional flags can be used:

- **-r**: Generates a static summary after a set number of cycles. This is useful for logging.
- **-c (count)**: Specifies the number of packets sent per hop before exiting.
- **-n**: Disables reverse DNS lookups for faster output.
- **-P (port)**: Uses TCP or UDP probes (helpful for bypassing ICMP blocks).

When diagnosing a network issue, observe the following in MTR’s output:

- **Identifying packet loss**: Unlike traceroute, which may show sporadic timeouts, MTR calculates packet loss percentages for each hop. If loss occurs at a specific router but not beyond it, that device may be dropping packets due to congestion or misconfiguration. Consistent loss across multiple hops suggests a broader network issue, such as a faulty backbone link.
- **Analysing latency spikes:** MTR's continuous probing reveals whether high latency is persistent or intermittent. A sudden spike in RTT at a particular hop indicates a potential bottleneck, such as an overloaded router or a suboptimal peering connection. If latency increases gradually across multiple hops, the issue may be general congestion rather than a single faulty node.
- **Detecting routing fluctuations**: Since MTR updates in real time, it can expose route flapping, where paths change frequently due to dynamic routing protocols like **Border Gateway Protocol** (**BGP**). If the same destination shows different hops in successive refreshes, it may explain inconsistent connectivity.
- **Comparing forward and reverse paths**: Networks often route traffic asymmetrically, meaning the path to a destination may differ from the return path. While traceroute only shows the forward route, MTR's persistent probing can sometimes hint at return-path issues if packets disappear after a certain hop.

MTR is a powerful evolution of traceroute, offering real-time, statistical insights into network performance. Its ability to continuously monitor latency, packet loss, and routing changes makes it indispensable for diagnosing intermittent or complex connectivity issues. While traditional traceroute remains useful for quick checks, MTR's dynamic approach provides a deeper, more actionable view of network behaviour.

While both traceroute and MTR map network paths, MTR offers several advantages over traditional traceroute:

- **Continuous monitoring**: Traceroute provides a single snapshot of the network path, which may miss intermittent problems. MTR, by contrast, keeps probing indefinitely, making it ideal for catching

transient issues like periodic packet loss or temporary latency spikes.

- **Statistical insights**: Traceroute shows per-probe latency, but MTR aggregates data into averages, minimums, maximums, and loss percentages. This makes it easier to spot trends, such as a specific router consistently adding delay.
- **User-friendly interface**: MTR's real-time display is more intuitive for ongoing diagnostics, whereas interpreting multiple traceroute runs requires manual comparison. The color-coded output in some MTR implementations (like mtr-curses) further highlights problematic hops.
- **Flexible probe options**: While traceroute primarily uses UDP or ICMP, MTR supports TCP probes (**-P** flag), which can bypass firewalls that block traditional traceroute methods. This is particularly useful for diagnosing issues in restrictive networks.

However, traceroute still has niche uses like quick one-off path checks where continuous monitoring is not needed. It provides compatibility with older systems where MTR is not installed. For scripting and automation, the output from traceroute is easier to parse in some cases.

Port and socket inspection

Port and socket inspection provides critical insights into connectivity issues, service availability, and security threats. By examining open ports, active connections, and socket states, administrators can identify service availability, diagnose connectivity failures, and analyze network performance.

Netstat

Network statistics (**Netstat**) is a command-line tool available on Linux that provides critical network connection information. It helps diagnose connectivity problems, detect unauthorized connections, and analyze network performance.

To view all active connections on the host, you can use the netstat **-a**

command. The following listing shows a typical output:

```
$ netstat -a|head -30
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 _localdnsproxy:domain   0.0.0.0:*               LISTEN
tcp        0      0 _localdnsstub:domain    0.0.0.0:*               LISTEN
tcp        0      0 localhost:2616          0.0.0.0:*               LISTEN
tcp        0      0 localhost:zebra         0.0.0.0:*               LISTEN
tcp        0      0 localhost:bgpd          0.0.0.0:*               LISTEN
tcp        0      0 localhost:41947         0.0.0.0:*               LISTEN
tcp6       0      0 [::]:ssh                [::]:*                  LISTEN
tcp6       0     52 ubuntu24:ssh            chandan-FRI-FXX:43252   ESTABLISHED
udp        0      0 _localdnsproxy:domain   0.0.0.0:*
udp        0      0 _localdnsstub:domain    0.0.0.0:*
raw6       0      0 [::]:ipv6-icmp          [::]:*                  7
raw6       0      0 [::]:ipv6-icmp          [::]:*                  7
raw6       0      0 [::]:ipv6-icmp          [::]:*                  7
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   Path
unix  3      [ ]         STREAM     CONNECTED     10783
unix  2      [ ACC ]     STREAM     LISTENING     3772     /run/systemd/journal/io.systemd.journal
unix  2      [ ]         DGRAM                    10833    /run/user/1000/systemd/notify
unix  3      [ ]         STREAM     CONNECTED     9403
unix  2      [ ACC ]     STREAM     LISTENING     10836    /run/user/1000/systemd/private
unix  2      [ ACC ]     STREAM     LISTENING     10845    /run/user/1000/bus
```

The output from the command displays the first 30 lines of active network connections and listening sockets on a Linux system, broken into two main sections, which consist of Active Internet connections and Active UNIX domain sockets. The Internet connections section lists TCP, UDP, and RAW socket details, showing columns such as protocol (Proto), receive and send queue sizes (Recv-Q, Send-Q), local and foreign addresses, and the state of the connection (e.g., LISTEN, ESTABLISHED). For example, it reveals services like **Secure Shell** (**SSH**) listening on all interfaces ([::]:ssh) and an established SSH connection from a remote host, along with DNS-related services listening locally. The UNIX domain sockets section shows **inter-process communication** (**IPC**) sockets used by systemd and user sessions, with details like reference counts, socket types, and states (e.g., LISTENING, CONNECTED). LISTEN states indicate services waiting for incoming connections (e.g., SSH on port 22), helping identify open ports and potential entry points. ESTABLISHED states show live connections, useful for spotting active communication, such as the SSH session in the example.

This example includes TCP, UDP, and Unix socket connections on the host. To list only TCP or UDP connections, you can use the **-t** or **-u** option with netstat, respectively. The following listing shows the result of running netstat to get all TCP-based connections that are in listening mode and displays the results with IP addresses instead of hostnames. The listening connection is selected using **-l**, and the **-n** option displays numeric addresses instead of hostnames.

```
$ sudo netstat -ltn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 127.0.0.54:53           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:2616          0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:2601          0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:2605          0.0.0.0:*               LISTEN
```

```
tcp        0      0 127.0.0.1:41947         0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
```

The output of the command reveals a focused view of the system's listening TCP sockets. This command filters the results to show only sockets actively waiting for incoming connections, with all addresses and ports displayed in numeric form. The columns indicate the protocol, queue sizes, local listening address, allowed remote address range, and the socket state. Most services in the above output are bound strictly to loopback addresses such as 127.0.0.1, 127.0.0.53, and 127.0.0.54. This means services like the local DNS resolvers on port 53 and the various applications on ports 2601, 2605, 2616, and 41947 are only accessible from the machine itself, which is a strong security practice that minimizes external attack vectors.

Adding the **-p** option will also list the processes that created the listening connection.

The following listing shows a typical output of this command:

```
$ sudo netstat –ltnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address
State       PID/Program name
tcp        0      0 127.0.0.54:53           0.0.0.0:*               LISTEN
511/systemd-resolve
tcp        0      0 127.0.0.53:53           0.0.0.0:*               LISTEN
511/systemd-resolve
```

Note: Using the -p option to list the process with netstat requires elevated privilege and hence must be run with sudo.

This is helpful to identify issues when you receive an address already in use error, which is a common error when multiple processes try to open the same port on the host. Here is an example to reproduce this issue. The following command starts a web server on port 8000:

```
chandan@ubuntu24:~$ python3 -m http.server &
[1] 17617
```

```
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Now, if we try the same command again, we will hit the address already in use error:

```
chandan@ubuntu24:~$ python3 -m http.server & [2] 17618
Traceback (most recent call last):
File "", line 198, in _run_module_as_main
File "", line 88, in _run_code
...
File "/usr/lib/python3.12/http/server.py", line 136, in server_bind socketserver.TCPServer.server_bind(self)
File "/usr/lib/python3.12/socketserver.py", line 473, in server_bind self.socket.bind(self.server_address)
OSError: [Errno 98] Address already in use
```

In this case, we can use the **netstat** command to find out the process that has opened port 8000 as follows:

```
chandan@ubuntu24:~$ sudo netstat -ltnp |grep 8000
tcp        0      0 0.0.0.0:8000            0.0.0.0:*               LISTEN      17617/python3
```

Now you can decide if you need to release the port used by the old process by killing it or starting your second server on a different port.

You can get the network statistics of your host using the netstat command with the **-s** option. The netstat **-s** command displays comprehensive network statistics for all supported protocols, including IP, ICMP, TCP, and UDP. It provides detailed metrics such as packets transmitted and received, errors, retransmissions, and connection failures. For IP, it shows fragmentation and reassembly data. ICMP statistics include message counts and error rates. TCP metrics cover segments sent/received, retransmissions, and connection timeouts. UDP data shows datagrams and errors. These statistics help diagnose network issues like packet loss, congestion, or protocol-specific failures. The following is a partial listing of the output:

```
$ sudo netstat -s
Ip:
```

Forwarding: 1
73243117 total packets received
145 with invalid headers
737712 forwarded
0 incoming packets discarded
72504154 incoming packets delivered
37863951 requests sent out
338 outgoing packets dropped
14489 dropped because of missing route
OutTransmits: 38601663

Icmp:
10306 ICMP messages received
32 input ICMP message failed
ICMP input histogram:
destination unreachable: 9002
timeout in transit: 23
echo requests: 1281
59405 ICMP messages sent
0 ICMP messages failed
OutRateLimitGlobal: 22
OutRateLimitHost: 1676
ICMP output histogram:
destination unreachable: 58123
echo requests: 1
echo replies: 1281

Tcp:
190928 active connection openings
9368 passive connection openings
359 failed connection attempts
11237 connection resets received

```
25 connections established
35441925 segments received
```

This hows the overall summary of the network state of the host.

Netcat

Netcat or **nc** command, which is often called the **Swiss Army knife of networking**, is a versatile command-line tool for reading and writing data across network connections. It is invaluable for testing connectivity, port scanning, transferring files, and debugging network services.

Checking ports open on a host

To check if a port is open on the host, you can use the **nc** command as follows:

```
$ nc -zv example.com 80
```

This will scan port 80 on the server **example.com** and return success if the port is found to be open. Following is a typical output of this command:

```
$ nc -zv localhost 8000
Connection to localhost (127.0.0.1) 8000 port [tcp/*] succeeded!
```

To scan for a UDP port the **-u** option can be used as follows:

```
$ nc -zvu 192.168.122.1 53
Connection to 192.168.122.1 53 port [udp/domain] succeeded!
```

Multiple ports can be scanned by providing a range of ports, as shown here:

```
$ nc -zv localhost 2600-2607
nc: connect to localhost (127.0.0.1) port 2600 (tcp) failed: Connection refused
Connection to localhost (127.0.0.1) 2601 port [tcp/zebra] succeeded!
nc: connect to localhost (127.0.0.1) port 2602 (tcp) failed: Connection refused
nc: connect to localhost (127.0.0.1) port 2603 (tcp) failed: Connection refused
```

```
nc: connect to localhost (127.0.0.1) port 2604 (tcp) failed: Connection refused
Connection to localhost (127.0.0.1) 2605 port [tcp/bgpd] succeeded!
nc: connect to localhost (127.0.0.1) port 2606 (tcp) failed: Connection refused
```

Using NC as a temporary server and client

Netcat serves as a powerful temporary server or client for quick network testing and debugging. As a server, it can listen on any port with the **-l** option to receive connections:

```
$ nc -l -p 8080
```

Incoming data is printed on the screen; this is ideal for verifying firewall rules or testing remote access. As a client, it can connect to services. Here is an example of using **nc** as a client to a web server:

```
$ nc example.com 80
```

This can be used as a **s** quick check by manually sending raw protocols like **Hyper Text Transfer Protocol** (**HTTP**), **Simple Mail Transfer Protocol** (**SMTP**), or checking port availability. E.g., in the following listing, we can see **nc** used as an HTTP client:

```
chandan@ubuntu24:~$ mkdir nctest
chandan@ubuntu24:~$ cd nctest
chandan@ubuntu24:~/nctest$ echo "This is a test for using Netcat as a server" > index.html
chandan@ubuntu24:~/nctest$ python3 -m http.server 4000 &
[4] 68804
chandan@ubuntu24:~/nctest$ Serving HTTP on 0.0.0.0 port 4000 (http://0.0.0.0:4000/) ...
chandan@ubuntu24:~/nctest$ echo -e "GET /\n\n” |nc localhost 4000
127.0.0.1 - - [12/Jul/2025 06:31:47] "GET /" 200 -
This is a test for using Netcat as a server
```

In this example, we started an HTTP server and used **nc** to act as a client to

access the content on the server. We can also use **nc** as a server, as shown in the following example:

```
chandan@ubuntu24:~/nctest$ cat index.html |nc -l 4001 &
[5] 68841
chandan@ubuntu24:~/nctest$ nc localhost 4001
This is a test for using Netcat as a server
```

Using Netcat for file transfer

Netcat enables instant file transfers between hosts. On the receiving side, start an **nc** processing in listening mode and redirect the output to a file:

```
$ nc -l -p 1234 > file
```

Next, on the sender's side, use the **nc** command as a client to read the data from a file to be copied to the remote host:

```
$ nc host 1234 < file
```

This is perfect for ad-hoc diagnostics, though unencrypted traffic should be avoided in production. This example assumes that port 1234 is not blocked by a firewall.

Socket statistics

The **socket statistics** (**ss**) command is a Linux tool for network troubleshooting, replacing the older netstat. It provides detailed socket information and is faster for analyzing connections.

To list all connections with **ss** command, use the following options:

```
$ ss -tulnp
```

The options used in the following command are like the ones used with netstat, **-t** to specify selection of TCP connections, **-u** for UDP connections, **-n** to specify numeric output instead of DNS names, and **-p** for output process names that created the connections.

```
$ sudo ss -tulnp
```

The **ss** command also takes an optional **FILTER** parameter to select connections in a certain state. E.g., the following command will only

display TCP connections in the connected state:

```
$sudo ss -t state connected
```

The **ss** command recognizes connected, synchronized, bucket, and big as some of the connection states:

- The connected state includes all the states except for listening and closed.
- Synchronized state includes all the connected states except for syn-sent.

The man page of **ss** command provides details of all the states that can be used with the state filter.

You can also use a filter based on an expression. We have already discussed the filter parameter; the expression can select a connection based on the source and destination IP, source and destination port, etc.

Here is an example of a filter expression running on an SSH server that selects connections based on destination IP and source port:

```
$ sudo ss -p  '( sport = :ssh and dst = 192.168.122.1 )'
Netid  State  Recv-Q  Send-Q  Local Address:Port  Peer Address:Port  Process
tcp  ESTAB  0  52  [::ffff:192.168.122.61]:ssh  [::ffff:192.168.122.1]:43252  users:(("sshd",pid=1449,fd=4),("sshd",pid=1332,fd=4))
```

The **ss** command provides real-time network diagnostics, offering speed, precision, and deep TCP insights that netstat cannot match.

lsof

List open files (**lsof**) command is a powerful Unix/Linux utility that reveals which processes have opened files, sockets, or network connections. Since Linux treats network sockets as files, **lsof** is invaluable for diagnosing network issues. To list all open network connections, use the **lsof** with **-i** option. E.g., the following listing shows a typical output of this command:

```
$ sudo lsof -i
```

```
COMMAND    PID           USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
systemd      1           root  101u  IPv6   6110      0t0  TCP *:ssh (LISTEN)
systemd-r  511 systemd-resolve   14u  IPv4   5750      0t0  UDP _localdnsstub:domain
systemd-r  511 systemd-resolve   15u  IPv4   5751      0t0  TCP _localdnsstub:domain (LISTEN)
systemd-r  511 systemd-resolve   16u  IPv4   5752      0t0  UDP _localdnsproxy:domain
systemd-r  511 systemd-resolve   17u  IPv4   5753      0t0  TCP _localdnsproxy:domain (LISTEN)
container 1033           root   10u  IPv4   9420      0t0  TCP localhost:41947 (LISTEN)
sshd      1330           root    3u  IPv6   6110      0t0  TCP *:ssh (LISTEN)
zebra    15097            frr   23u  IPv4 208588      0t0  TCP localhost:zebra (LISTEN)
bgpd     15102            frr   18u  IPv4 208596      0t0  TCP localhost:bgpd (LISTEN)
staticd  15109            frr   12u  IPv4 208604      0t0  TCP localhost:2616 (LISTEN)
python3  17617        chandan    3u  IPv4 414041      0t0  TCP *:8000 (LISTEN)
```

To display connections for TCP or UDP protocols, use the **lsof -i <protocol>** command. The **-n** option can be used to get numeric output instead of using DNS names. For example, the following output shows only the UDP connections:

```
$ sudo lsof -i UDP -n
COMMAND    PID           USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
systemd-r  511 systemd-resolve   14u  IPv4   5750      0t0  UDP 127.0.0.53:domain
```

```
systemd-r 511 systemd-resolve   16u  IPv4   5752      0t0  UDP
127.0.0.54:domain
```

To get a list of listening ports you can use **lsof -i -s <PROTOCOL>:LISTEN** command, as shown in the following listing:

```
$ sudo lsof -i -s TCP:LISTEN -n
COMMAND     PID            USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
systemd       1            root  101u  IPv6   6110      0t0  TCP *:ssh (LISTEN)
systemd-r   511 systemd-resolve   15u  IPv4   5751      0t0  TCP 127.0.0.53:domain (LISTEN)
systemd-r   511 systemd-resolve   17u  IPv4   5753      0t0  TCP 127.0.0.54:domain (LISTEN)
container  1033            root   10u  IPv4   9420      0t0  TCP 127.0.0.1:41947 (LISTEN)
sshd       1330            root    3u  IPv6   6110      0t0  TCP *:ssh (LISTEN)
zebra     15097             frr   23u  IPv4 208588      0t0  TCP 127.0.0.1:zebra (LISTEN)
bgpd      15102             frr   18u  IPv4 208596      0t0  TCP 127.0.0.1:bgpd (LISTEN)
staticd   15109             frr   12u  IPv4 208604      0t0  TCP 127.0.0.1:2616 (LISTEN)
python3   17617         chandan    3u  IPv4 414041      0t0  TCP *:8000 (LISTEN)
```

To get network usage for a given remote address, the following command can be used:

```
$ sudo lsof -i @192.168.122.1 -n
COMMAND  PID  USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
sshd    1332  root    4u  IPv6   9821      0t0  TCP 192.168.122.61:ssh->192.168.122.1:43252 (ESTABLISHED)
```

```
sshd    1449 chandan    4u  IPv6   9821      0t0  TCP 192.168.122.61:ssh->192.168.122.1:43252 (ESTABLISHED)
```

To get a list of connections sorted by process name, the following command can be used. It uses awk to select only certain fields of **lsof** output and gets a count of connections by process ID using **sort** and **uniq**. This can help in finding out the number of connections made by a process.

```
$ lsof -i -nP| awk '!/COMMAND/{print $1,$2,$5,$8}' | sort | uniq -c | sort -rn|head -10
     11 firefox 2878106 IPv4 TCP
     10 firefox 2878106 IPv6 TCP
      2 systemd-r 316012 IPv4 UDP
      2 systemd-r 316012 IPv4 TCP
      2 dnsmasq 1911 IPv4 UDP
      2 avahi-dae 1344 IPv6 UDP
      2 avahi-dae 1344 IPv4 UDP
      1 systemd 1 IPv6 TCP
      1 sshd 2753609 IPv6 TCP
      1 ssh 2799852 IPv4 TCP
```

lsof can be used for identifying port conflicts, debugging service binding issues, and monitoring process-level network activity. Combining it with commands like **grep**, **awk**, and **kill** provides an efficient network troubleshooting and diagnostic tool.

Troubleshooting using packet captures

Packet capturing is a fundamental technique for network troubleshooting because it provides raw, unfiltered visibility into network traffic. By recording actual packets traversing a network, administrators can:

- **Verify connectivity issues**: Captures can confirm whether packets reach their destination or get dropped due to firewalls, routing errors, or faulty hardware.
- **Identify latency problems**: Analyzing TCP retransmissions, duplicate

acknowledgements (**ACKs**), and round-trip times can pinpoint bottlenecks causing slow performance.

- **Diagnose protocol errors**: Inspecting packet headers and payloads reveals malformed packets, misconfigured services, or protocol violations (e.g., **Secure Sockets Layer** (**SSL**)/**Transport Layer Security** (**TLS**) handshake failures).
- **Detect security threats**: Captures expose malicious traffic like port scans, **distributed-denial-of-service** (**DDoS**) attacks, or data exfiltration attempts.
- **Troubleshoot application issues**: By examining HTTP, DNS, or database queries, admins can determine if problems stem from the network or the application layer.

Tools like **tcpdump** and Wireshark decode packets, showing:

- Source/destination IPs/ports
- TCP flags
- Payload data

Unlike synthetic tests (e.g., **ping**), packet captures provide ground truth, revealing exactly what occurs on the wire. Whether diagnosing intermittent failures, optimizing performance, or investigating breaches, packet analysis is indispensable for root-cause analysis in networks.

Using tcpdump to capture packets

Network administrators and engineers frequently encounter connectivity issues that require analysis beyond simple diagnosis with tools like **ping** or **traceroute**. In these situations, **tcpdump** emerges as an indispensable tool for deep packet inspection and network traffic analysis. As a command-line packet analyser, **tcpdump** provides raw visibility into network communications, allowing for granular troubleshooting of connectivity problems, performance issues, and security incidents. Unlike higher-level tools that provide abstracted summaries of network behaviour, **tcpdump** exposes the actual packets traversing a network interface, revealing the truth about what is occurring at the protocol level.

The power of **tcpdump** lies in its ability to capture and display network

traffic in real time or save it to a file for later analysis. When a user reports that an application cannot connect to a server, for instance, **tcpdump** can verify whether connection attempts are leaving the host, whether responses are being received, and if so, what those responses contain. This level of detail proves invaluable when diagnosing complex issues involving firewalls, routing problems, or application-layer protocols.

One of the most common use cases for **tcpdump** involves verifying basic network connectivity. When a server appears unreachable, running **tcpdump** on both the client and server can determine where packets are being dropped. On the client side, the absence of outgoing packets suggests an application issue, while outgoing packets with no replies indicate a network or firewall problem. The server-side capture might reveal whether packets are arriving at all, and if they are, whether the server is responding. This bidirectional analysis often uncovers misconfigured firewalls, routing issues, or host-based filtering rules that prevent communication:

```
$ sudo tcpdump -i eth0 port 80 –nn
```

-i eth0 tells **tcpdump** to listen on interface eth0. The port 80 options filter for HTTP traffic. **-nn** disables DNS and port resolution for faster output.

If no packets appear, the server may not be receiving traffic. If SYN packets appear without SYN-ACK replies, the service may not be running

The tool supports filtering expressions that allow administrators to focus on specific types of traffic, reducing noise and helping isolate the relevant packets from what could otherwise be an overwhelming stream of data. The effectiveness of **tcpdump** depends heavily on proper filter construction. The tool's filtering language allows examination of traffic based on numerous criteria, including source and destination addresses, port numbers, protocol types, and even specific bits within packet headers. Mastering this syntax enables targeted captures that exclude irrelevant traffic while highlighting packets of interest. For example, a filter can isolate DNS queries to troubleshoot name resolution problems or focus on HTTP traffic when debugging web applications. This precision becomes critical in busy network environments where capturing all traffic would generate unmanageable amounts of data.

Basic filter structure

Basic tcpdump rule follows this pattern:

```
tcpdump [options] [filter expression]
```

Where the filter expression consists of:

- Primitives (host, port, net)
- Qualifiers (src, dst, proto)
- Operators (and, or, not)
- Protocol-specific filters

Host-based filtering rules

The host-based filtering rule selects the packets to dump by matching the primitive host as the source or destination of the traffic. Here are some examples of host-based filtering:

$ sudo tcpdump host 192.168.1.100

This filter will select traffic to or from the IP address 192.168.1.100. To select a traffic source or destination IP address, use the **src** and **dst** qualifiers, respectively. For example, the following command shows filtering traffic from the source IP address 192.168.1.100:

```
$ sudo tcpdump src host 192.168.1.100
```

While the following filter will select traffic to the destination IP address 192.168.1.100:

```
$ sudo tcpdump dst host 192.168.1.100
```

Network-based filtering rules

Similarly, for using network-based filtering following rules can be used:

```
$ sudo tcpdump net 192.168.1.0/24
$ sudo tcpdump src net 10.0.0.0/8
$ sudo tcpdump dst net 172.16.0.0/16
```

In the preceding example, traffic from or to a network is selected with the net primitive, while **src** and **dst** qualifiers can be used to select the source

and destination network, respectively.

Network port-based filtering rules

Network port-based filtering also follows the same pattern. The following example shows the port, source, and destination port-based filtering rules, respectively:

```
$ sudo tcpdump port 80
$ sudo tcpdump src port 22
$ sudo tcpdump dst port 53
```

In addition to the preceding code, which is based on a single network port, the **portrange**-based filtering can be used to filter packets matching a range of ports as follows:

```
$ sudo tcpdump portrange 1000-2000
```

Protocol based filtering rules

A protocol-based filtering rule can be used to select packets based on the protocol. **tcpdump**'s ability to decode various protocol layers helps identify whether problems occur at the network (IP), transport (TCP/UDP), or application layer. In the following example, **icmp** protocol is used:

```
$ sudo tcpdump icmp
```

Other protocols like TCP, UDP, and ARP can also be used to filter packets.

TCP flag-based filtering rules

TCP flags can be used to match the packets for a certain connection state. The following rules match connection state SYN, ACK, and FIN, respectively, by comparing the values of these flags in the packets:

```
$ sudo tcpdump 'tcp[tcpflags] & tcp-syn != 0'
$ sudo tcpdump 'tcp[tcpflags] & tcp-ack != 0'
$ sudo tcpdump 'tcp[tcpflags] & tcp-fin != 0'
```

Packet size-based rules

The packets can also be matched against their size with comparison terms like less than or greater than, as shown in the following code:

```
$ sudo tcpdump less 64
$ sudo tcpdump greater 1500
```

Exact size match is also possible with the = operator:

```
$ sudo tcpdump length = 128
```

Combining filter rules

In all the previous examples we discussed, the filtering rule is applied based on a single parameter. What makes tcpdump's rules so powerful is the ability to combine the filters to create complex rules. This ability is very helpful while debugging a networking issue on a busy server. For example, if you want to trace the packets for a request to a web server for a specific host with IP address 192.168.1.100, you can combine the **src** host and **dst** port rule as shown here:

```
$ sudo tcpdump 'src host 192.168.1.100 and dst port 80'
```

Another example is to use the not option to negate a match. In the following example, we are interested in all traffic from host 192.168.1.100, but not to port 22 (SSH traffic):

```
$ sudo tcpdump 'host 192.168.1.100 and not port 22'
```

The following shows a more complex example depicting how multiple rules can be combined with a logical or operator:

```
$ sudo tcpdump '(src net 10.0.0.0/8 and dst port 443) or (src net 172.16.0.0/16 and dst port 80)'
```

Similarly, the following rule uses the logical and operator to monitor DNS traffic for a specific domain:

```
$ sudo tcpdump 'port 53 and udp and host example.com'
```

Limiting the size of tcpdump captures

On a busy network connection, it is often not possible to analyze the information dumped by **tcpdump** as it appears on the screen. In such

situations, it is helpful to write the captured packets to a file. This can be done using the **-w** option to **tcpdump**:

```
$ sudo tcpdump -w /tmp/dump.pcap
```

The packet dump can be later analyzed by running **tcpdump** against the dump file:

```
$ sudo tcpdump -r /tmp/dump.pcap
```

While debugging a transient or difficult-to-reproduce network issue, you might need to run **tcpdump** for a long time, which often leads to a very large dump file size. There are a few options that are useful to learn, which help to deal with this situation by rotating the dump file based on its size or at a certain time limit. The following command will create a new dump file every 60 seconds the **tcpdump** runs:

```
$ sudo tcpdump -G 60 -w /tmp/dump.pcap
```

To rotate the dump file after a certain file size is reached, you can use the **-C** option. In the following example, **tcpdump** will create a new file after the current dump file reaches the size of 100MB:

```
$ sudo tcpdump -C 100 -w /tmp/dump.pcap
```

Despite its power, **tcpdump** does present some challenges. The volume of output can overwhelm beginners, and interpreting packet details requires a solid understanding of network protocols. The tool also consumes significant system resources when capturing high-speed traffic, potentially affecting performance on production systems. For these reasons, many administrators use **tcpdump** for targeted, short-term captures rather than continuous monitoring. When long-term analysis is needed, the output is typically redirected to a file for later processing with **tcpdump** itself or more specialized analysis tools like Wireshark.

In modern network environments, **tcpdump** remains relevant even as more user-friendly packet analyzers emerge. Its lightweight nature makes it ideal for use on servers where installing graphical tools is not practical, and its availability on nearly all Unix-like systems means it is always at hand when needed. The skills learned from **tcpdump** also transfer well to other packet analysis tools, forming a foundation for more advanced network

diagnostics.

Reading PCAP files

The format of the dump file generated by **tcpdump** with **-w** option is called **Packet Capture** (**PCAP**). It is the standard binary format for storing network traffic captures. PCAP format is used by tools like **tcpdump**, Wireshark, and TShark. A PCAP files contain raw packet data along with metadata about when each packet was captured.

A PCAP file consists of:

- **Global header (24 bytes)**: Contains metadata about the entire capture.
- **Packet headers and packet data (Repeated for each packet)**: Timestamp and packet length info, followed by the actual packet bytes:

  [Global Header]
  [Packet Header][Packet Data]
  [Packet Header][Packet Data]
  ...

The global header has a size of 24 bytes and appears once at the start of the file and defines the capture parameters. The following table describes the fields in the global header:

Field	Size (bytes)	Description
Magic number	4	Identifies byte order (0xA1B2C3D4 for big-endian, 0xD4C3B2A1 for little-endian).
Major version	2	PCAP format major version (usually 0x02).
Minor version	2	PCAP format minor version (usually 0x04).
Timezone offset	4	Timezone correction (usually 0).
Timestamp accuracy	4	Accuracy of timestamps (usually 0).
Snapshot length	4	Max packet length captured (65535 for full packets).
Link-layer type	4	Data link type (e.g., 1 for Ethernet, 101 for raw IP).

***Table 4.1** : Fields in PCAP global header*

Each packet in the PCAP file is preceded by a header of size 16 bytes, containing metadata about the packet as described in the following table:

Field	Size (bytes)	Description
Timestamp (Seconds)	4	Unix epoch seconds when captured
Timestamp (Microseconds)	4	Microsecond fraction of timestamp
Captured length	4	Bytes actually captured
Original length	4	Original packet length (before truncation)

***Table 4.2**: Fields in PCAP packet header*

The raw packet bytes immediately follow each packet header. The format depends on the link-layer type specified in the global header. Common link layer types are as follows:

- Ethernet
- Raw IP (no link-layer header)
- Linux *cooked* capture address **Socket Link Layer** (**SLL**)

There are multiple libraries that can be used to read PCAP files. These can be handy if you want to generate any custom report on packets captured by tcpdump. Following is an example Python script that reads a PCAP file and prints the MAC and IP addresses for the first 3 captured packets:

```
#================================================================#
# read_pcap.py
#================================================================#
import dpkt
import socket
def read_pcap(file_path):
    """
    Reads a PCAP file and prints Ethernet (MAC) and IP addresses for each packet.
    """
```

```
with open(file_path, 'rb') as f:
    pcap = dpkt.pcap.Reader(f)

    count = 0
    for timestamp, buf in pcap:
        count +=1
        if count > 3:
            break
        try:
            # Parse Ethernet frame
            eth = dpkt.ethernet.Ethernet(buf)

            # Print MAC addresses
            print(f"\nTimestamp: {timestamp}")
            print(f"Source MAC: {':'.join(f'{x:02x}' for x in eth.src)}")
            print(f"Destination MAC: {':'.join(f'{x:02x}' for x in eth.dst)}")

            # Check if packet contains IP
            if isinstance(eth.data, dpkt.ip.IP):
                ip = eth.data

                # Print IP addresses
                src_ip = socket.inet_ntoa(ip.src)
                dst_ip = socket.inet_ntoa(ip.dst)
                print(f"Source IP: {src_ip}")
                print(f"Destination IP: {dst_ip}")
            else:
                print("Non-IP Packet")

        except Exception as e:
```

```
            print(f"Error processing packet: {e}")
if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <pcap_file>")
        sys.exit(1)

    read_pcap(sys.argv[1])
#=============================================================#
```

Running the script against a **pcap** file will generate output as shown here, which might be useful to automate report generation:

```
$ python read_pcap.py dump.pcap
Timestamp: 1752327458.165801
Source MAC: 52:54:00:ee:6a:84
Destination MAC: 52:54:00:42:5c:1a
Source IP: 192.168.122.61
Destination IP: 192.168.122.1
...
```

Using Wireshark and TShark for packet capture

Wireshark, a graphical network protocol analyzer, and TShark, its command-line counterpart, are among the most powerful tools for network packet capture and analysis. Together, they enable network administrators, security analysts, and engineers to capture, dissect, and interpret network traffic in real time or from saved capture files.

Wireshark and TShark are part of the same suite of tools developed by the Wireshark Foundation. Both support hundreds of protocols and provide deep packet inspection capabilities, but they cater to different use cases:

- Wireshark offers a user-friendly **graphical user interface** (**GUI**), ideal for interactive analysis.
- TShark is a command-line tool, better suited for automation, scripting,

and remote captures.

Wireshark

To start **wireshark** you can execute the **wireshark** command from the shell as follows:

```
$ sudo wireshark
```

This will open the **Welcome to Wireshark** screen. Here you can select the interface to start the packet capture:

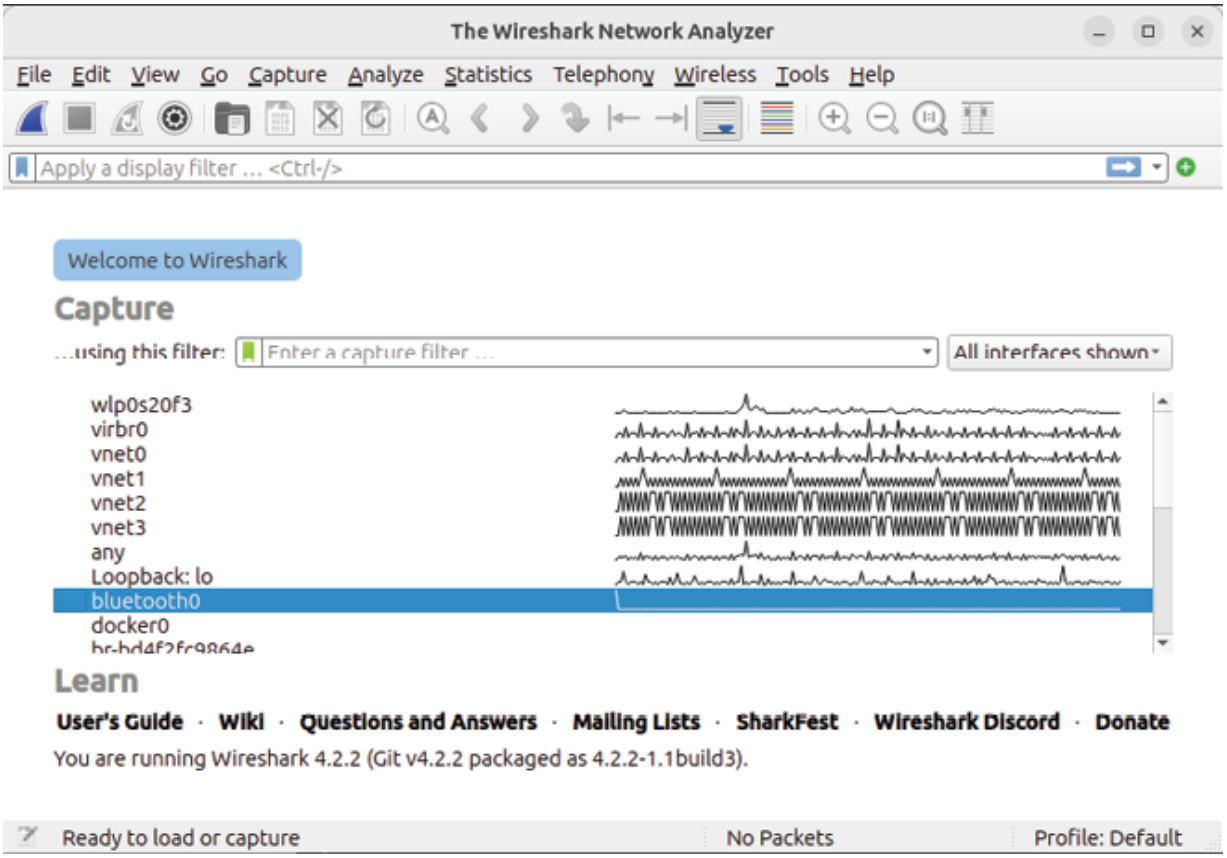


***Figure 4.1**: Wireshark welcome screen*

Once the interface is selected, you should see the following packet capture screen. Here you can see three panes; the top pane shows the packets captured, which provides similar details as the **tcpdump** output. With **wireshark**, you can view all the details of the packet, including the **hexdump**.

The lower two panes show the details of the packet, like the Frame, Ethernet, and IP headers on the left pane, while the right pane shows the **hexdump** of the packet:

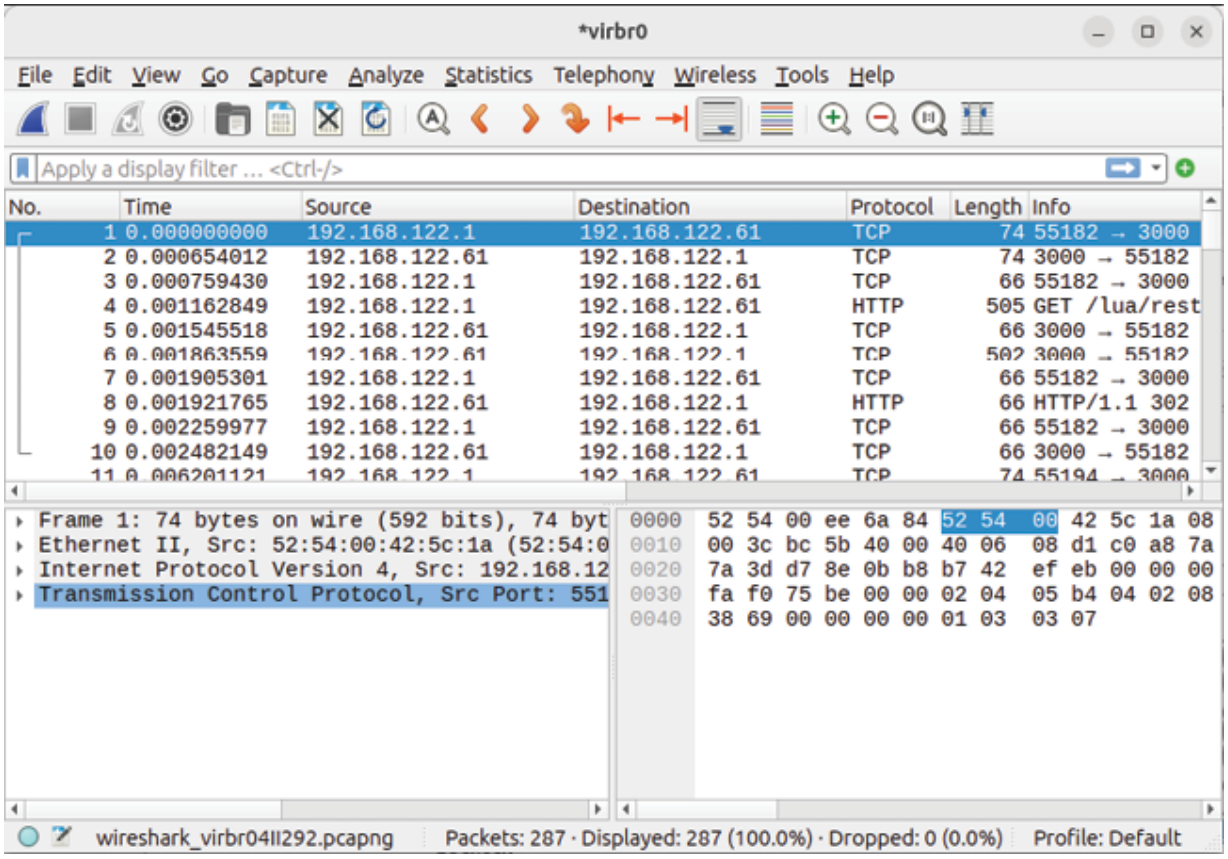


***Figure 4.2**: Wireshark packet capture*

The headers in the details pane can be expanded to view the fields and the values in each header. E.g., the following figures show the Ethernet header expanded.

Filtering packets

Wireshark provides two types of filters:

- The capture filter limits the packets that are captured by Wireshark. The capture filter follows the same filtering rule format as **tcpdump**. At its core, it uses **libpcap** filters, which are also used by **tcpdump**. In the following figure, we have set the capture filter to capture only SSH

traffic from host 192.168.122.1:

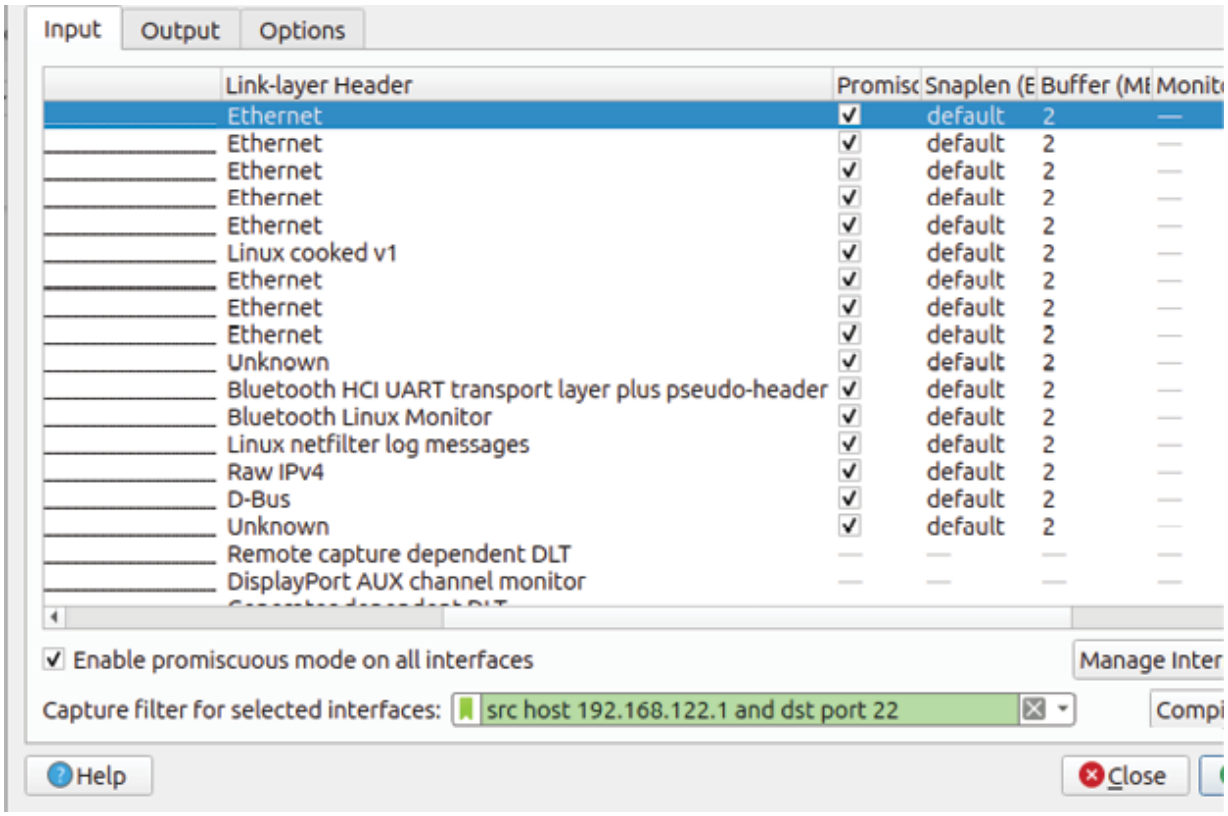


***Figure 4.3**: Wireshark capture filter*

- The display filters, on the other hand, limit the number of packets displayed by Wireshark. Attributes of the packet can be applied as a display filter. In the following figure, we configured the status code of the HTTP packet as a filter:

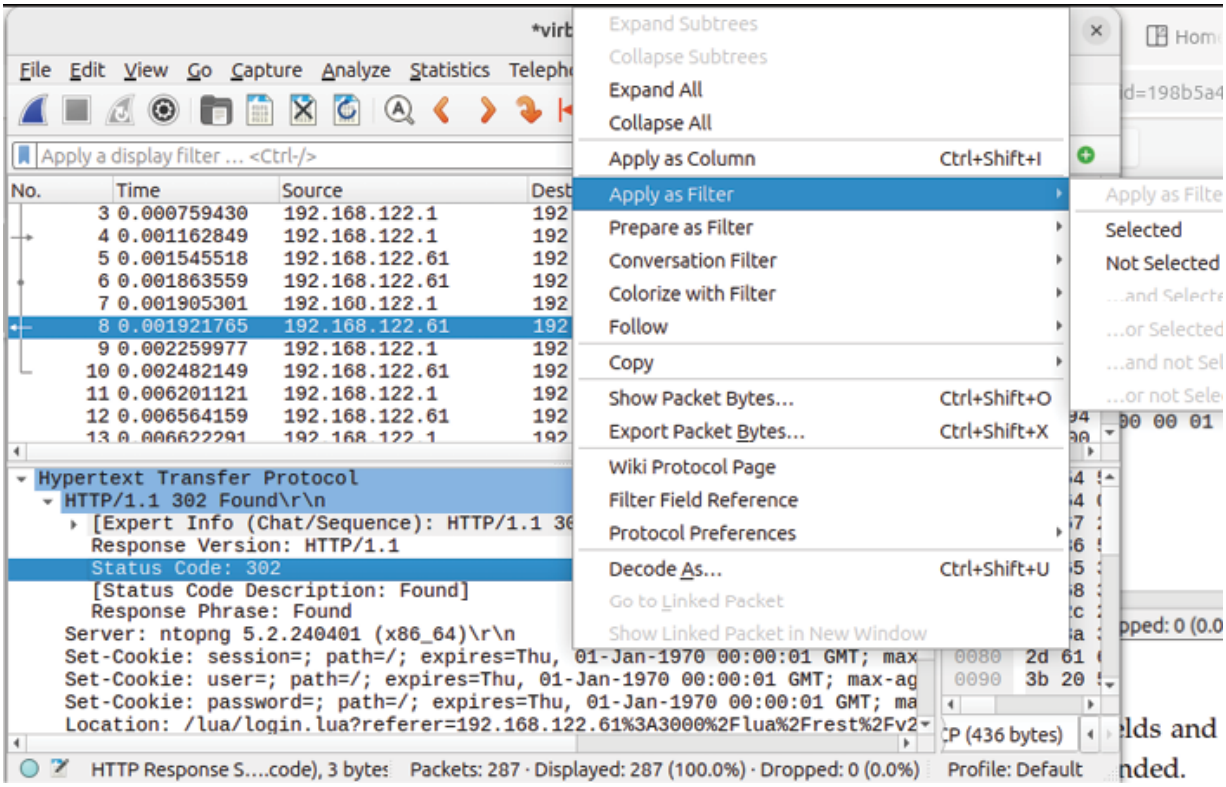


***Figure 4.4**: Wireshark display filter*

This filter will only display the packets where the status code 302 was seen.

TShark

TShark is the command-line version of Wireshark. Its functionality is like **tcpdump**, although **tcpdump** is a lightweight, command-line packet capture tool focused on raw packet collection with basic filtering. **tcpdump** prints packet summaries or writes them to a file (**pcap**) for later analysis.

TShark, on the other hand, is designed for packet analysis. It decodes protocols, applies advanced filters, and generates detailed statistics beyond just capturing traffic. It supports the display filter of Wireshark in addition to the capture filter, which is also supported by **tcpdump**. In the following example, you can see the filters running with **tcpdump** and TShark:

```
$ sudo tcpdump -n -c 3 src host 192.168.122.1 and dst port 22
tcpdump: verbose output suppressed, use -v[v]... for full protocol
```

```
decode
listening on enp1s0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
13:08:49.274871 IP 192.168.122.1.36558 > 192.168.122.61.22: Flags [.], ack 3597838395, win 1208, options [nop,nop,TS val 1421983801 ecr 4220438989], length 0
13:08:49.279060 IP 192.168.122.1.36558 > 192.168.122.61.22: Flags [P.], seq 0:36, ack 117, win 1208, options [nop,nop,TS val 1421983805 ecr 4220439004], length 36
13:08:49.299106 IP 192.168.122.1.36558 > 192.168.122.61.22: Flags [P.], seq 36:72, ack 153, win 1208, options [nop,nop,TS val 1421983825 ecr 4220439008], length 36
3 packets captured
7 packets received by filter
0 packets dropped by kernel

$ sudo tshark -n -c 3 src host 192.168.122.1 and dst port 22
Running as user "root" and group "root". This could be dangerous.
Capturing on 'enp1s0'
    1 0.000000000 192.168.122.1 → 192.168.122.61 SSH 102 Client: Encrypted packet (len=36)
    2 0.018490895 192.168.122.1 → 192.168.122.61 SSH 102 Client: Encrypted packet (len=36)
    3 0.037907350 192.168.122.1 → 192.168.122.61 SSH 102 Client: Encrypted packet (len=36)
3 packets captured
```

It should be noted that TShark can decode the protocol to SSH.

In summary, **tcpdump** is a lightweight tool that can be used to collect packet dumps and get high level analysis of traffic. TShark is more resource-intensive but is able to provide better analysis of the packets. It is possible to combine these tools using **tcpdump** to collect data from a production system into a **pcap** file and use **tshark** or **wireshark** to analyse the data offline.

NetFlow

NetFlow is a network protocol that collects and analyzes IP traffic data, providing visibility into bandwidth usage, traffic patterns, and potential network issues. It works by capturing metadata about network flows sharing common attributes like source/destination IP, port numbers, and protocol. When enabled on routers or switches, NetFlow exports flow records to a collector for analysis. This data helps identify bandwidth hogs, detect anomalies like DDoS attacks, and optimize traffic routing. Unlike packet captures, NetFlow is lightweight and scalable, making it ideal for monitoring large networks without excessive storage or performance overhead.

NetFlow is particularly useful for troubleshooting because it offers aggregated traffic insights without inspecting individual packets. For example, it can reveal sudden spikes in traffic to a specific port, indicating a potential attack or misconfigured service. It also helps track application performance by showing which protocols consume the most bandwidth. Security teams use NetFlow to detect suspicious flows, such as data exfiltration attempts or communication with known malicious IPs. Capacity planning benefits from long-term flow data, highlighting growth trends and congestion points.

Compared to tools like **tcpdump** or Wireshark, NetFlow provides a broader, more manageable view of network behavior, making it essential for enterprise troubleshooting, security monitoring, and performance optimization. Its flow-based approach complements deep packet inspection, offering both high-level trends and granular traffic analysis when needed.

Enabling NetFlow on Linux

To understand the NetFlow protocol and get a hands-on experience of NetFlow, we will deploy **ntopng** on Linux and use it to collect NetFlow data and analyze the same. **ntopng** is a high-performance, open-source network traffic monitoring and analysis tool. It provides real-time visibility into network usage, helping administrators analyze bandwidth consumption, detect anomalies, and troubleshoot performance issues. It supports various

traffic monitoring methods, including NetFlow.

To install **ntopng** we can follow the instructions in the installation guide of **ntopng**. For this test, we will run the following commands on our Ubuntu host to configure the apt repository for **ntopng** and install **ntopng** using the **apt** command:

```
$ wget https://packages.ntop.org/apt/ntop.key | sudo apt-key add -
$ sudo add-apt-repository "deb
https://packages.ntop.org/apt/$(lsb_release -cs) ./"
$ sudo apt update
$ sudo apt install ntopng
```

Once the installation is completed, we can check if the **ntopng** service is running using the following command:

```
$ systemctl status ntopng
● ntopng.service - ntopng - High-Speed Web-based Traffic Analysis
and Flow Collection Tool
    Loaded: loaded (/usr/lib/systemd/system/ntopng.service; enabled;
preset: enabled)
    Active: active (running) since Thu 2025-07-10 12:21:20 UTC;
16min ago
      Docs: man:ntopng(8)
            file:/usr/share/doc/ntopng/README.Debian
            file:/usr/share/doc/ntopng/UserGuide.pdf.gz
   Process: 1003 ExecStart=/usr/sbin/ntopng /etc/ntopng.conf
(code=exited, status=0/SUCCESS)
```

Once we have verified the service is running, we can access the **ntopng webui** by navigating to **http://hostip:3000**.

On opening the above **Uniform Resource Locator** (**URL**) in your browser, you will be asked to enter the username and password to access the content. Once you enter the default username and password, you will be asked to reset the password.

The **ntopng** web UI provides various dashboards that provide a summary of the hosts doing most network communications, as shown in the following

figure:

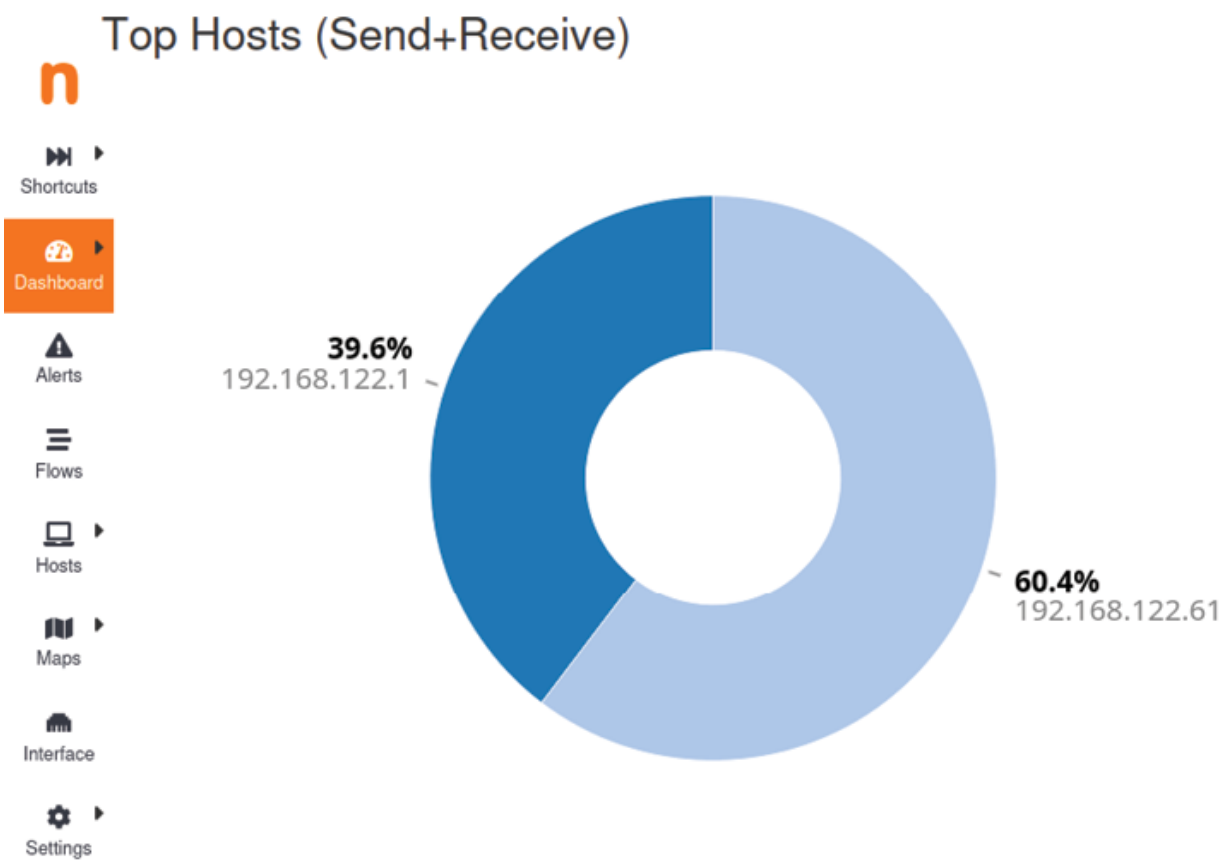


***Figure 4.5**: ntopng top hosts*

You can also check the top application ports and how they contribute to the network usage. In the following figure, you can notice the top services are SSH, HTTP, and **ntop** itself:

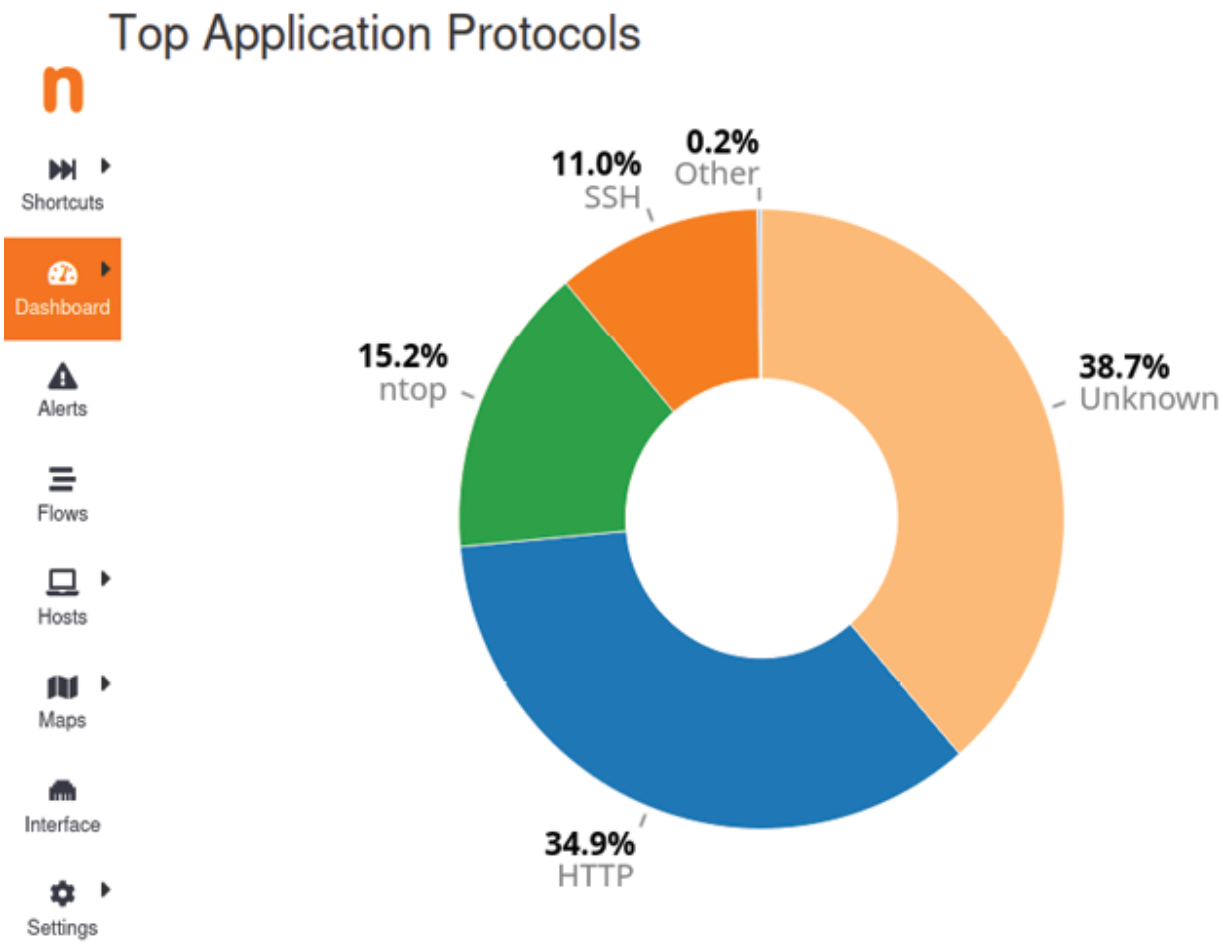


***Figure 4.6**: ntopng top applications*

In the active flow dashboard, you can check the currently active flows or connections on the host:

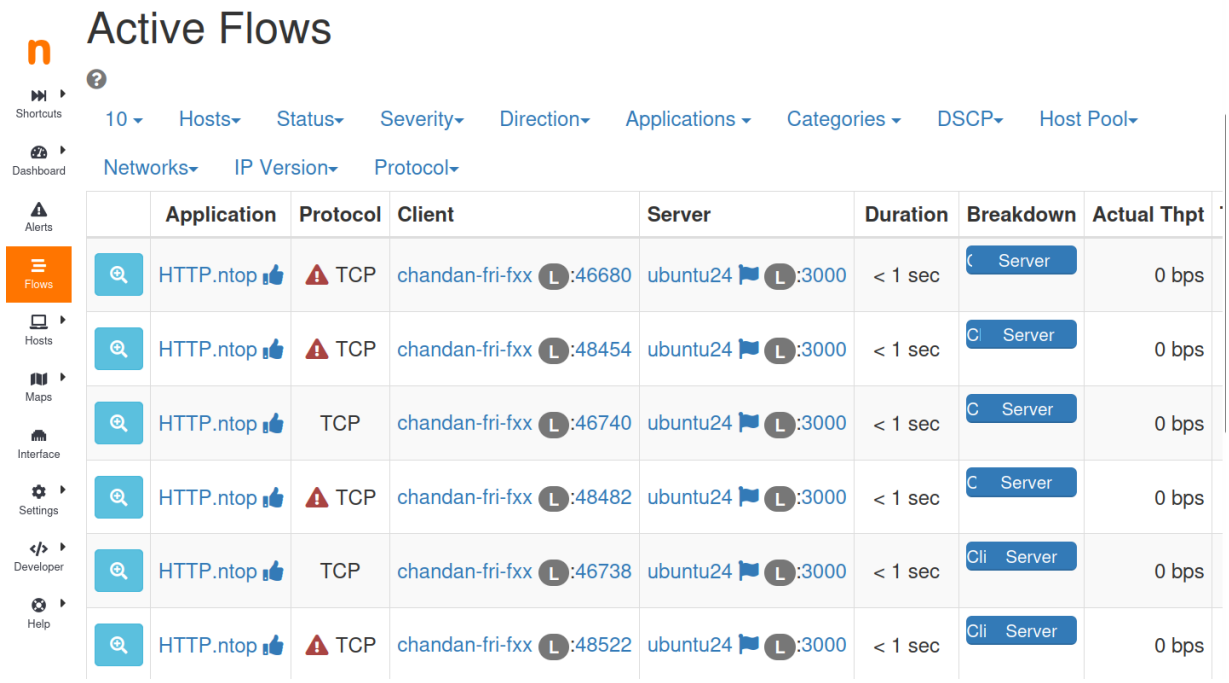


***Figure 4.7**: ntopng active flows*

Adding hosts with nProbe NetFlow exporter

The dashboards that we discussed show the NetFlow data on a single host. It is possible to export NetFlow data from other hosts into **ntopng** NetFlow collector. This can be done using the nProbe program. nProbe is a high-performance NetFlow/sFlow/IPFIX probe developed by **ntop** (creators of **ntopng**). It collects network traffic data, converts it into flow records, and exports them to collectors like **ntopng** for analysis. nProbe can be installed from the **ntop** package repository. The following command will enable the **ntop yum** repository:

```
$ curl https://packages.ntop.org/centos-stable/ntop.repo |sudo tee /etc/yum.repos.d/ntop.repo
$ sudo dnf config-manager --set-enabled crb
$ sudo dnf install epel-release
$ sudo dnf install nprobe
```

Next update the **ntopng** config file to include the following options to enable NetFlow collector in **ntopng**:

```
--interface=enp1s0
```

```
--netflow-collector=2055
```

This will require a restart of the **ntopng** service. You can use the following command to restart the service:

```
$ sudo systemctl restart ntopng
```

Finally, start the **nprobe** daemon on the remote host to start the NetFlow export as follows:

```
$sudo nprobe -i enp1s0 -n 192.168.122.61:2055
```

In this example, the **ntopng** NetFlow collector is running at 192.168.122.61. You should now be able to view the NetFlow data from a remote host running **nprobe** on the **ntopng** web UI in the following figure:

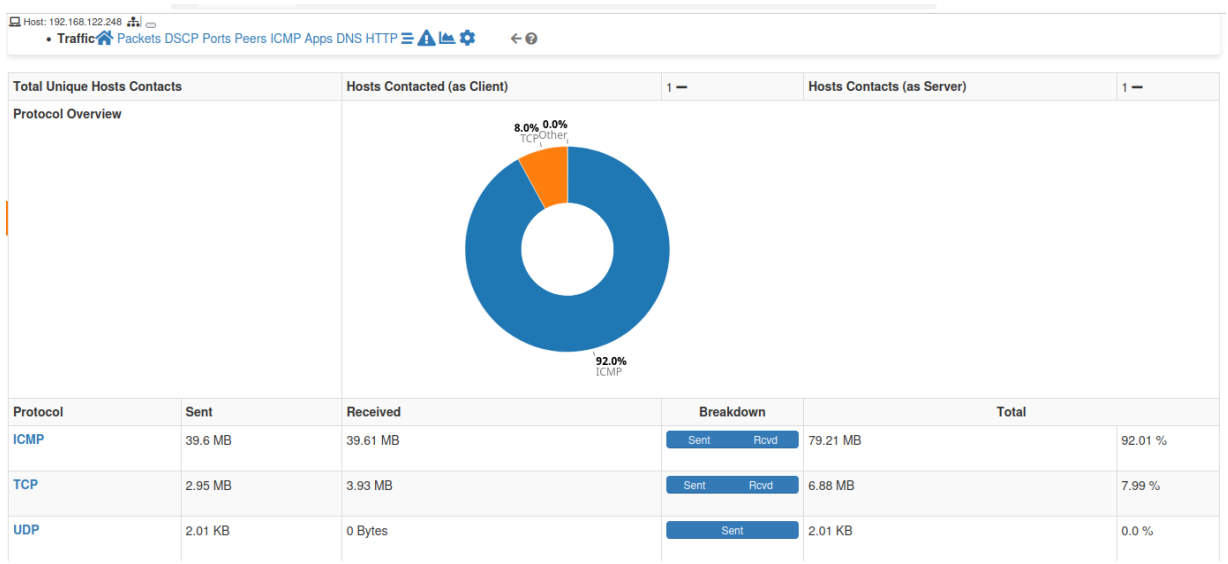


***Figure 4.8**: ntopng remote host with nprobe*

Conclusion

Network troubleshooting requires a systematic approach and the right tools to diagnose and resolve issues effectively. Commands like ping and traceroute provide quick connectivity checks, while netstat and ss reveal active connections and listening ports, helping identify service availability and unexpected traffic. Tools like lsof take this further by linking network activity to specific processes, making it easier to pinpoint misbehaving

applications. For deeper analysis, tcpdump and Wireshark offer packet-level inspection, exposing protocol errors, latency spikes, and security threats. Meanwhile, NetFlow provides a broader view of traffic patterns, aiding in bandwidth monitoring and anomaly detection.

Mastering these tools allows network administrators to move from basic connectivity tests to advanced traffic analysis, ensuring optimal performance and security. While each tool serves a distinct purpose, combining them creates a powerful troubleshooting workflow.

In the next chapter, we will learn about some of the common application services and understand the Layer 4 networking needs of applications. Our understanding of debugging techniques learned in this chapter will be helpful for overcoming issues in configuring these services.

CHAPTER 5

Network Services and Protocols

Introduction

In the last few chapters, we discussed and understood the protocols and services at Layer 2 and Layer 3. This chapter explores the critical network applications and protocols at Layer 4 and the succeeding layers. A Linux administrator and user must understand how to configure, manage, and troubleshoot these network applications effectively, as they provide core networking infrastructure used by other applications. We will examine key protocols such as **Domain Name System** (**DNS**), **Network Time Protocol** (**NTP**), **Secure Shell** (**SSH**), **Network File System** (**NFS**), etc. By the end of this chapter, you will have a thorough understanding of how these services and protocols function in a Linux environment, how to configure them securely, and how they interact to form the foundation of modern networked systems.

Structure

This chapter covers the following topics:

- Domain Name System
- Network Time Protocol

- Configuring systemd-timesyncd on Linux
- Configuring NTP on Linux
- Configuring chrony service
- Secure Shell
- Network File System
- Hypertext Transfer Protocol

Objectives

This chapter aims to provide a comprehensive understanding of essential network services and protocols in Linux, enabling you to configure, manage, and secure them effectively. You will learn how DNS resolves domain names to **Internet Protocol** (**IP**) addresses and how to set up a basic DNS resolver. The chapter covers SSH for secure remote access, including key-based authentication, and explores **Hypertext Transfer Protocol** (**HTTP**) for web communication along with basic web server setup. You will gain practical skills in system clock synchronization via NTP for accurate timekeeping. Additionally, the chapter explains how to implement network file sharing using NFS while maintaining proper permissions. Throughout, emphasis will be placed on security best practices to safeguard these critical services against common vulnerabilities. By the end of this chapter, you will have both the theoretical knowledge and hands-on skills needed to work confidently with these fundamental Linux network services.

Domain Name System

DNS acts as the internet's address book, converting easy-to-remember domain names like google.com into numerical IP addresses that computers use to locate each other. DNS resolution employs a recursive query mechanism that systematically traverses a hierarchical server structure to translate domain names into IP addresses. When a client device initiates a DNS lookup, the process begins with the local resolver, typically operated

by the **internet service provider** (**ISP**) or configured as a public DNS service. This resolver assumes full responsibility for completing the entire query chain on behalf of the client, demonstrating the recursive nature of the system.

The recursion unfolds through multiple tiers of DNS infrastructure. The resolver first examines its cache for previously resolved queries to optimize performance. For un-cached requests, it systematically contacts the root servers, which form the apex of the DNS hierarchy. These root servers do not possess the final answer but provide referrals to the appropriate **top-level domain** (**TLD**) servers corresponding to the domain's extension (**.com**, .**org**, etc.).

The resolver then queries these TLD servers, which in turn direct it to the authoritative name servers for the specific domain. These authoritative servers, managed by the domain's owner, finally provide the definitive IP address mapping. Throughout this process, the resolver recursively follows each referral, handling all intermediate queries autonomously until it obtains the final answer to return to the original client. The following figure describes the recursive call flow for DNS name resolution:

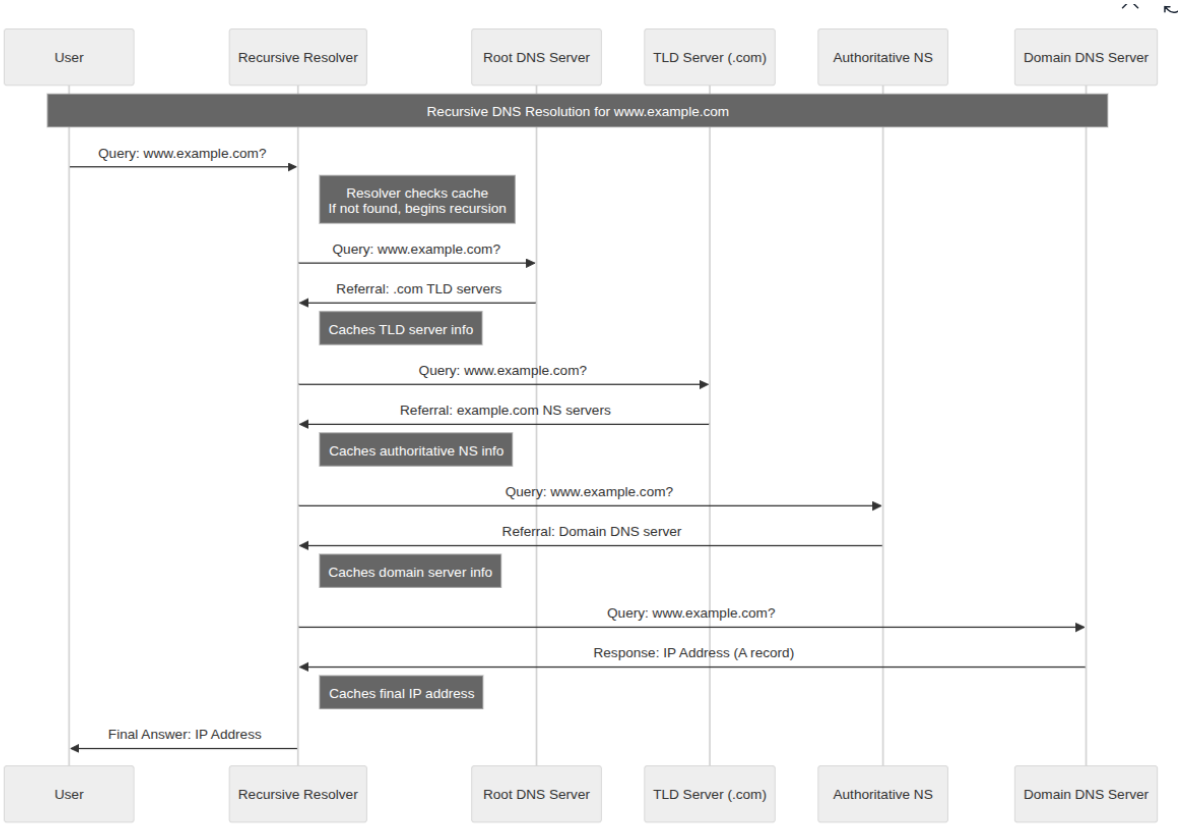


***Figure 5.1**: Recursive resolution of DNS names*

The following steps are performed by the recursive DNS resolver:

1. **User query**: The User (e.g., a web browser) asks the recursive resolver (like Google DNS, Cloudflare, or an ISP's DNS) for the IP address of **www.example.com**.
2. **Resolver cache check**: The resolver first checks its local cache to see if it already has the IP for **www.example.com**. If not cached (or the cache has expired), it starts the recursive resolution process.
3. **Query root DNS server**: The resolver queries a root DNS server. The root server does not know the IP for **www.example.com**, but it knows who manages the **.com** domain. It refers the resolver to the **top-level domain** (**TLD**) servers for **.com**.
4. **Cache TLD info**: The resolver caches the TLD server information for future **.com** queries. Caching the TLD server information improves performance for subsequent queries for the same domain.

5. **Query TLD server**: The resolver queries a **.com** TLD server for the authoritative name servers for **example.com**.
6. **Cache authoritative name server (NS) info**: The resolver caches the authoritative name server records for example.com.
7. **Query authoritative NS**: The resolver queries one of the authoritative NS for example.com. The NS may refer to the actual domain DNS server that hosts the A record for **www.example.com**, or it might delegate to a DNS server for the subdomain.
8. **Cache domain server info**: In case the authoritative name servers respond with the DNS server for a subdomain, the resolver caches the domain DNS server reference.
9. **Query domain DNS server**: The resolver queries the final domain DNS server. The domain DNS server returns the IP address (A record) for **www.example.com**.
10. **Cache final IP**: The resolver caches the IP address with a TTL so future requests can be answered faster.
11. **Return answer to user**: The resolver returns the IP address to the User.

This recursive architecture provides several key advantages. It offloads complexity from end-user devices, centralizes caching for improved efficiency, and enables a distributed, fault-tolerant system where no single entity needs to know all possible domain mappings. The resolver's ability to persistently chase down referrals through multiple hierarchical levels while maintaining the original query context exemplifies the truly recursive nature of DNS resolution.

During hostname resolution on a Linux system, the **/etc/nsswitch.conf** file plays a pivotal role in determining the sequence and methods used to translate hostnames into IP addresses. When an application attempts to resolve a hostname, like ping trying to reach another machine, the system consults the hosts line in **nsswitch.conf** to decide the resolution strategy. The following is an example of typical **nsswitch.conf** found on Linux systems:

```
$ cat /etc/nsswitch.conf
```

```
# /etc/nsswitch.conf
#
# Example configuration of GNU Name Service Switch functionality.
# If you have the `glibc-doc-reference' and `info' packages installed, try:
# `info libc "Name Service Switch"' for information about this file.

passwd:         files systemd sss
group:          files systemd sss
shadow:         files systemd sss
gshadow:        files systemd

hosts:          files mdns4_minimal [NOTFOUND=return] dns mymachines
networks:       files

protocols:      db files
services:       db files sss
ethers:         db files
rpc:            db files

netgroup:       nis sss
automount:  sss
```

The line **hosts**: **files mdns4_minimal [NOTFOUND=return] dns mymachines** in the **/etc/nsswitch.conf** file configures the order and behaviour of the system's hostname resolution sources. Each entry is a database or service that will be queried, in sequence, to translate hostnames into IP addresses.

Here is an explanation of each component in the order they are used:

- **files**: This instructs the system to first consult the local **/etc/hosts** file. This file contains static mappings of hostnames to IP addresses,

manually maintained by the system administrator or local applications. It is checked first because it is fast and authoritative for the local machine.

- **mdns4_minimal [NOTFOUND=return]**: This is a specialized source for multicast DNS (mDNS). It is primarily used to resolve hostnames ending in the **.local** top-level domain on a local area network without needing a central DNS server. If the hostname is a **.local** name and is not found, the search stops and fails.
- **dns**: This is the traditional DNS, which queries the DNS servers listed in **/etc/resolv.conf**. It is used to resolve standard domain names on the internet (e.g., example.com). This source is only reached for queries that are not resolved by files and are not **.local** queries (which would have been stopped by the previous rule).
- **mymachines**: This is a service provided by systemd that can resolve the hostnames of local containers and virtual machines managed by systemd-machined or systemd-nspawn. It provides dynamic resolution for hostnames of other local system instances without needing to edit **/etc/hosts**.

The resolution process follows the exact order specified in **nsswitch.conf**, typically starting with local files (checking **/etc/hosts**) before potentially proceeding to DNS lookups or other network-based name services:

```
$ cat /etc/hosts
127.0.0.1 localhost
127.0.1.1 ubuntu24

# The following lines are desirable for IPv6 capable hosts
::1     ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

This ordered approach allows for efficient resolution, where frequently

accessed local hostnames can be resolved immediately without network queries, while still falling back to DNS for unknown hosts. The configuration also enables complex enterprise environments to incorporate additional name services like **Lightweight Directory Access Protocol** (**LDAP**) into the resolution chain when needed.

The file's settings directly impact both the speed and reliability of hostname resolution, as well as the system's ability to function during network outages (when local host files may provide critical resolution) or in disconnected environments. Misconfiguration can lead to resolution failures, slow network operations, or unexpected behaviour when applications cannot properly resolve hostnames. System administrators often tune these settings to optimize performance for their specific environment, such as prioritizing DNS for cloud systems or emphasizing local files for isolated networks.

DNS service on Linux using BIND

Berkeley Internet Name Domain (**BIND**) is the most widely used DNS server software on Linux, providing robust domain name resolution capabilities. In this section, we will discuss how to set up a BIND server to provide DNS service.

On most Linux distributions (Debian/Ubuntu, Rocky Linux/CentOS), install BIND using the package manager:

```
$ sudo apt install bind9 bind9-utils
$ sudo yum install bind bind-utils
```

Let us check the configuration file for the BIND server at **/etc/bind/named.conf**. This is the main configuration file for the bind server, and it includes the config files for different zones:

```
chandan@ubuntu24:~$ tail -5 /etc/bind/named.conf
// If you are just adding zones, please do that in
/etc/bind/named.conf.local
include "/etc/bind/named.conf.options";
include "/etc/bind/named.conf.local";
include "/etc/bind/named.conf.default-zones";
```

To add the DNS zone configuration for example.com, we will edit the **/etc/bind/named.conf.local**:

1. Add the forward and reverse zones as follows:

```
chandan@ubuntu24:~$ tail /etc/bind/named.conf.local

zone "example.com" {
    type master;
    file "/etc/bind/db.example.com";  # Path to zone file
};

zone "1.168.192.in-addr.arpa" {
    type master;
    file "/etc/bind/db.192.168.1";
};
```

   The zone definition configures the type and the path of the zone database file.

2. The following listing describes the forward and reverse mapping database format:

```
chandan@ubuntu24:~$ cat /etc/bind/db.example.com
$TTL 86400
@   IN  SOA  ns1.example.com. admin.example.com. (
              2025072501 ; Serial
              3600       ; Refresh
              1800       ; Retry
              604800     ; Expire
              86400      ; Min TTL
)
@       IN  NS    ns1.example.com.
ns1     IN  A     192.168.1.10
www     IN  A     192.168.1.20
```

```
mail    IN  A     192.168.1.30
chandan@ubuntu24:~$ cat /etc/bind/db.192.168.1
$TTL 86400
@   IN  SOA  ns1.example.com. admin.example.com. (
             2023072401 ; Serial
             3600       ; Refresh
             1800       ; Retry
             604800     ; Expire
             86400      ; Min TTL
)
@       IN  NS    ns1.example.com.
10      IN  PTR   ns1.example.com.
20      IN  PTR   www.example.com.
30      IN  PTR   mail.example.com.
```

3. Once the zone files are created, we will verify the configuration if valid with the **named-checkconf** and **named-checkzone** commands and follow:

```
chandan@ubuntu24:~$ sudo named-checkconf
chandan@ubuntu24:~$ echo $?
0
chandan@ubuntu24:~$ sudo named-checkzone example.com
/etc/bind/db.example.com
zone example.com/IN: loaded serial 2025072501
OK
```

4. Now we can validate if the DNS is able to resolve the address resolution queries for the **example.com** domain. This can be done using the **dig** command, which is as follows:

```
chandan@ubuntu24:~$ dig @localhost mail.example.com

; <<>> DiG 9.18.30-0ubuntu0.24.04.2-Ubuntu <<>> @localhost
mail.example.com
```

```
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36547
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; COOKIE: c36bf59ca1b0d22a01000000688312d97e36e00ad5fb0fd5 (good)
;; QUESTION SECTION:
;mail.example.com.              IN      A

;; ANSWER SECTION:
mail.example.com.       86400   IN      A       192.168.1.30

;; Query time: 0 msec
;; SERVER: 127.0.0.1#53(localhost) (UDP)
;; WHEN: Fri Jul 25 05:15:05 UTC 2025
;; MSG SIZE  rcvd: 89
```

5. You can get a short answer to the query by using the **+short** argument as shown here:

```
chandan@ubuntu24:~$ dig @localhost +short mail.example.com
192.168.1.30
```

6. Use the **dig -x** command to get the reverse mapping from IP address to name:

```
chandan@ubuntu24:~$ dig -x  192.168.1.10 +short
ns1.example.com.
chandan@ubuntu24:~$ dig -x  192.168.1.20 +short
```

```
www.example.com.
chandan@ubuntu24:~$ dig -x  192.168.1.30 +short
mail.example.com.
```

7. Now, let us try adding TXT records to bind to provide information or metadata about a host or service. To do this, we will edit the zone database and add the following configuration:

```
build1  IN  A    192.168.1.41
build2  IN  A    192.168.1.42
build3  IN  A    192.168.1.43
build4  IN  A    192.168.1.44


; TXT Records
@       IN      TXT     "This is a general TXT record for example.com"
secret  IN      TXT     "This is hidden data"
build   IN      TXT     "Build servers are [build1, build2, build3, build4]."
```

   Remember to update the serial of the database file after you change the database. Once the file is updated, validate the zone file and reload the bind service as follows:

```
$ sudo named-checkzone example.com /etc/bind/db.example.com
$ sudo systemctl restart bind9
```

8. Now we can query the DNS server for the TXT records:

```
$ dig +short TXT @localhost build.example.com
"Build servers are [build1, build2, build3]."
```

Editing zone files to add records can be cumbersome and error prone. In the following section, we will discuss the **Remote Name Daemon Control** (**RNDC**) tool to update records in a DNS zone.

Using RNDC to administrate the BIND service

RNDC is a command-line utility designed to securely manage and control a

running BIND DNS server (named). It enables administrators to perform critical operations such as reloading DNS zones, flushing caches, toggling query logging, checking server status, and shutting down BIND gracefully without requiring a full-service restart.

RNDC communicates with the BIND server over **Transmission Control Protocol** (**TCP**) port 953 by default, using a shared secret key, typically **Hash-based Message Authentication Code using the Secure Hash Algorithm** (**HMAC-SHA256**) for authentication, ensuring secure remote administration. It allows administrators to:

- Reload zones (without restarting BIND)
- Flush DNS caches
- Enable/disable query logging
- Check server statusShut down BIND gracefully.

To configure **rndc**, refer to the following steps:

1. First, create a **rndc** key using the following code:

```
$ sudo rndc-confgen -a -k rndc-key
wrote key file "/etc/bind/rndc.key"
```

   The **rndc** key file is created with restricted permission and ownership, with read-write access to the bind user and read access to the bind group:

```
$ ls -l /etc/bind/rndc.key
-rw-r----- 1 bind bind 100 Aug  4 10:45 /etc/bind/rndc.key
```

   The RNDC key file contains the shared secret used to authenticate the **rndc command line interface** (**CLI**) with the BIND server.

2. Now we will configure the bind server to reference the RNDC key file. To do this, edit the **/etc/bind/named.conf.options** file and add the following:

```
include "/etc/bind/rndc.key";

controls {
   inet 127.0.0.1 port 953
```

```
    allow { 127.0.0.1; } keys { "rndc-key"; };
};
```

3. To control the **example.com** domain with **rndc**, update the zone in the **/etc/bind/named.conf.local** file to allow update with **rndc** as follows:

```
zone "example.com" {
    type master;
    file "/etc/bind/db.example.com";  # Path to zone file
    allow-update { key "rndc-key"; };
};
```

4. Once done, restart the named service:

```
$ sudo systemctl restart named
```

5. Now you can check the status of **rndc**:

```
$ sudo rndc status
```

6. The **rndc** CLI can be used now to add records to the zone database. To do this, we will use the following command:

```
$ sudo nsupdate -k /etc/bind/rndc.key
> server 127.0.0.1
> update add newtxt.example.com 86400 TXT "Dynamic TXT record"
> send
> quit
```

7. To verify that the record has been added, use the following command:

```
$ dig +short TXT @localhost newtxt.example.com
"Dynamic TXT record"
```

RNDC utility offers significant advantages for managing BIND DNS servers efficiently and securely. Unlike manual restarts, RNDC allows dynamic reloading of zones and configurations without service interruptions, ensuring continuous DNS availability. It provides granular control, enabling administrators to flush caches, toggle query logging, and check server status in real-time. In this example, RNDC uses HMAC/ **Transaction Signature** (**TSIG**) shared-secret authentication (e.g., HMAC-

SHA256), enhancing security over plaintext methods. Its ability to manage BIND remotely simplifies administration in distributed environments.

Network Time Protocol

Time synchronization is a critical yet often overlooked aspect of system administration, ensuring that all machines in a network operate with consistent and accurate timestamps. NTP serves as the backbone of this synchronization, allowing Linux systems to maintain precise timekeeping by aligning their clocks with trusted time sources.

In a Linux environment, NTP operates as a background service that continuously adjusts the system clock by communicating with reference time servers, typically provided by global NTP pools or internal infrastructure. Proper time synchronization is essential for a wide range of operations, including log file consistency, security certificate validation, database transactions, and distributed computing tasks. Even minor time discrepancies can lead to authentication failures, data corruption, or debugging challenges in networked systems.

Linux systems commonly implement NTP through software packages such as the traditional **NTP daemon** (**ntpd**), systemd-timesyncd, or chrony (a modern alternative optimized for variable network conditions). These services work by periodically querying upstream time servers, calculating clock drift, and making gradual adjustments to keep the system time accurate. Additionally, NTP can be configured in a hierarchical manner, where some machines act as time sources for others, ensuring reliable synchronization across large-scale deployments without creating a high load on top-level servers.

By maintaining synchronized time, Linux systems improve security, operational reliability, and compliance with protocols that depend on accurate timestamps. Whether in a data center, cloud environment, or embedded system, NTP ensures that all components work in harmony, preventing subtle but disruptive timing-related issues.

Understanding the working of NTP

The NTP operates through a hierarchical, decentralized system of time synchronization that ensures clocks across networked computers maintain consistent and accurate time. At its core, NTP functions by establishing a reference timekeeping structure where reliable time sources, known as stratum servers, provide authoritative time data to client systems.

The synchronization process begins when an NTP client initiates communication with one or more configured time servers. These servers typically exist in a tiered hierarchy, with Stratum 0 devices being the most accurate [atomic clocks or **Global Positioning System** (**GPS**) time sources], followed by Stratum 1 servers directly connected to these reference time servers, and subsequent strata that distribute time further down the network. The client exchanges timestamped packets with its chosen time servers, carefully measuring network latency to account for transmission delays.

Through sophisticated algorithms, NTP analyses multiple time samples, discarding outliers and statistically filtering the most reliable time data. It continuously adjusts the system clock in small increments to avoid sudden jumps, a process known as **clock disciplining**. This gradual correction maintains stability while compensating for both time drift and network variability. The protocol maintains persistent associations with its time sources, periodically rechecking and readjusting to maintain microsecond-level precision.

NTP's true robustness lies in its fault-tolerant design. Clients automatically evaluate the reliability of multiple time sources, preferring more accurate stratum servers while seamlessly failing over to backups if primary sources become unavailable. The protocol also supports peer-to-peer synchronization among client systems when external references are inaccessible, creating a self-correcting network of timekeepers. This elegant combination of hierarchical time distribution, statistical filtering, and adaptive correction enables NTP to maintain consistent time across global networks, even under challenging network conditions.

NTP measures time offset using four critical timestamps in each exchange:

- $\mathbf{T_1}$: Client’s send time (when the request leaves the client).
- $\mathbf{T_2}$: Server’s receive time (when the request arrives at the server).
- $\mathbf{T_3}$: Server’s send time (when the response leaves the server).
- $\mathbf{T_4}$: Client’s receive time (when the response arrives back at the client).

These timestamps are embedded in NTP packets during synchronization.

NTP first computes the total round-trip delay (δ) between the client and server:

$$\delta=(T4-T1)-(T3-T2)$$

This represents the total time the packet spent in transit (minus the server’s processing time). Assuming symmetric network latency (i.e., the request and response take equal time), the time offset (θ) between the client and server is:

$$\theta=((T2-T1)+(T3-T4))/2$$

If $\theta > 0$, the client’s clock is ahead of the server.

If $\theta < 0$, the client’s clock is behind the server.

This formula averages the forward and backward path delays to cancel out network jitter. The following figure describes the process of calculating the round-trip delay and the time offset of the NTP client:

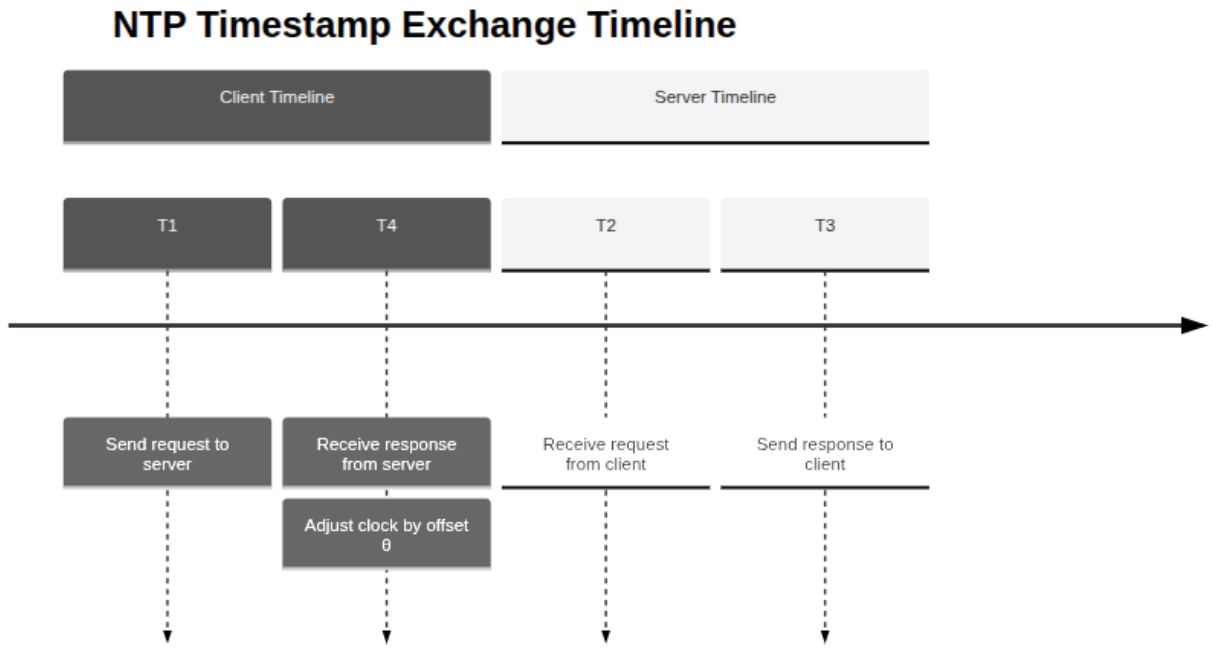


***Figure 5.2**: NTP offset calculation*

Since network conditions are not perfectly symmetric, NTP uses:

- **Multiple samples**: Collects several offset measurements over time.
- **Outlier rejection**: Discards inconsistent data (e.g., due to packet loss or spikes in latency).
- **Weighted averaging**: Prioritizes servers with lower latency and higher stratum (trust level).

Tools like clock discipline algorithms then apply small, gradual adjustments to avoid sudden time jumps.

Modern Linux systems provide multiple options to manage time synchronization:

- **systemd-timesyncd** is a service provided by systemd.
- **ntpd** is the classic time synchronization service.
- **chrony** is the modern alternative to NTP.

In the following section, we discuss the configuration of these services in brief.

Configuring systemd-timesyncd on Linux

To configure **systemd-timesyncd**, we will first install the package. This can be done using the following command:

```
$ sudo apt install systemd-timesyncd
```

The main configuration file for this service is **/etc/systemd/timesyncd.conf**.

Here,

- To configure the service, add the following to the config file:

```
[Time]
NTP=time.cloudflare.com
FallbackNTP=0.pool.ntp.org 1.pool.ntp.org 2.pool.ntp.org
```

- You can use the following **timedatectl** command to check the status of the service:

```
$ timedatectl
               Local time: Fri 2025-07-25 07:10:14 UTC
           Universal time: Fri 2025-07-25 07:10:14 UTC
                 RTC time: Fri 2025-07-25 07:10:14
                Time zone: Etc/UTC (UTC, +0000)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

Configuring NTP on Linux

To configure NTP on Linux, we will first install the **ntp** package using the following code:

```
$ sudo apt install ntp
```

The main configuration file for **ntp** is /etc/ntpsec/ntp.conf**.**

1. Use the following command to configure **ntp** service, we add the **ntp** servers to the file:

```
pool 0.ubuntu.pool.ntp.org iburst
pool 1.ubuntu.pool.ntp.org iburst
pool 2.ubuntu.pool.ntp.org iburst
pool 3.ubuntu.pool.ntp.org iburst


# Use Ubuntu's ntp server as a fallback.
server ntp.ubuntu.com
restrict default kod nomodify nopeer noquery limited
restrict 127.0.0.1
restrict ::1
```

2. Use the following **ntpq** command to check the status of **ntp** service:

```
$ ntpq -np
     remote           refid      st t when poll reach   delay   offset  jitter
```

```
==========================================================================
====================================
0.ubuntu.pool.ntp.org            .POOL.          16 p    -  256    0   0.0000   0.0000   0.0001
1.ubuntu.pool.ntp.org            .POOL.          16 p    -  256    0   0.0000   0.0000   0.0001
2.ubuntu.pool.ntp.org            .POOL.          16 p    -  256    0   0.0000   0.0000   0.0001
#185.125.190.57                 194.121.207.249  2 u   63  128  377 +162.159.200.1
10.217.8.105     3 u    9  128  377  25.6044 -34.9136   2.8125
-192.46.211.253                 129.69.253.1     2 u    -  128  377  65.0817 -38.6399  18.8290
*14.139.60.103                  .PPS.            1 u  201  128  376  42.5902 -38.0099   4.7777
+139.84.142.141                 35.73.197.144    2 u    6  128  377  59.7312 -40.4711
10.9830
+192.46.215.60                  193.190.230.65   2 u   66  128  375  61.2995 -38.1417
8.1948
+86.107.100.245                 15.207.248.194   5 u  133  128  377  60.6173 -37.4156
5.7222
```

3. The **ntpdig** command provides a more detailed state. Use the following command to check the various time offsets:

```
$ ntpdig -d
ntpdig: querying 127.0.0.1 (localhost)
org t1: ec2da4f9.f08b3000 rec t2: ec2da4f9.f0901d8c
xmt t3: ec2da4f9.f092848d dst t4: ec2da4f9.f094c000
org t1: 1753425529.939624 rec t2: 1753425529.939699
xmt t3: 1753425529.939736 dst t4: 1753425529.939770
rec-org t21: 0.000075  xmt-dst t34: -0.000034
2025-07-25 06:38:49.939736 (+0000) +0.000021 +/- 0.000055 localhost 127.0.0.1 s2 no-
leap
```

Configuring chrony service

Chrony is lightweight, works well on unstable networks, and is the default on many modern Linux distributions (Rocky Linux 8+, Ubuntu 18.04+).

You can use the following command to install the **chrony** package:

```
$ sudo apt install chrony
```

To check the status of the **chrony** service, use the **chronyc** tracking:

```
$ chronyc tracking
Reference ID    : 67BDBF0F (103.189.191.15)
Stratum         : 3
Ref time (UTC)  : Fri Jul 25 06:55:17 2025
System time     : 0.003661128 seconds fast of NTP time
Last offset     : +0.002993165 seconds
RMS offset      : 0.003702105 seconds
Frequency       : 4.657 ppm slow
Residual freq   : +0.000 ppm
Skew            : 30.799 ppm
Root delay      : 0.051735412 seconds
Root dispersion : 0.002603510 seconds
Update interval : 64.8 seconds
Leap status     : Normal
chandan@ubuntu24:~$ chronyc sources -v
  .-- Source mode  '^' = server, '=' = peer, '#' = local clock.
/ .- Source state '*' = current best, '+' = combined, '-' = not combined,
| /             'x' = may be in error, '~' = too variable, '?' = unusable.
||                                                 .- xxxx [ yyyy ] +/- zzzz
||     Reachability register (octal) -.           |  xxxx = adjusted offset,
||     Log2(Polling interval) --.      |          |  yyyy = measured offset,
||                                \    |          |  zzzz = estimated error.
||                                 |   |          \
MS Name/IP address         Stratum Poll Reach LastRx Last sample
===============================================================================
^- 185.125.190.58                2   6   377    53  +8450us[+8697us] +/-
```

```
93ms
^- prod-ntp-4.ntp1.ps5.cano>    2   6   377    53    +59ms[  +59ms] +/-
150ms
^- prod-ntp-3.ntp1.ps5.cano>    2   6   377   120  +1219us[+2047us] +/-
95ms
^- alphyn.canonical.com         2   6   377   119    +12ms[  +13ms] +/-
171ms
^* 103.189.191.15               2   6   377    30  -2236us[-1963us] +/-
27ms
^- 17.253.18.99                 2   6   377    56    +11ms[  +11ms] +/-   86ms
^- 139.84.142.141               2   6   377    57  +1099us[+1347us] +/-
99ms
^+ 160.250.111.68               5   6   377    60  +3174us[+3420us] +/-
35ms
```

It should be noted that only one of the preceding options should be used to synchronize the time of the system with the upstream time servers.

Secure Shell

SSH is an essential network protocol that provides a secure, encrypted channel for remote administration of Linux systems. Unlike older protocols like Telnet or **File Transfer Protocol** (**FTP**) that transmit data in plaintext, SSH encrypts all communication, ensuring confidentiality and integrity even over untrusted networks. This makes SSH the gold standard for securely managing servers, transferring files, and executing commands remotely.

At its core, SSH operates on a client-server model. The **SSH daemon** (**sshd**) runs on the remote machine, listening for incoming connections on port 22 by default. When a user connects using an SSH client, the two systems authenticate each other through cryptographic key pairs or passwords, then establish an encrypted tunnel for all subsequent communication. This tunnel protects against eavesdropping, tampering, and unauthorized access.

Beyond basic remote shell access, SSH supports advanced features like port forwarding (for securely tunnelling other protocols), **secure file transfers** (**SFTP**), and SSH keys (for password-less authentication). Its versatility makes it indispensable for system administrators, developers, and cloud engineers who need to manage distributed systems securely.

Proper SSH configuration is crucial for both functionality and security. Best practices include disabling root login, enforcing key-based authentication, and using firewall rules to restrict access. When properly implemented, SSH provides a robust and efficient way to maintain control over Linux systems, whether they are in a local data center or a cloud environment halfway around the world.

SSH is a cryptographic network protocol that enables secure communication between two systems over an unsecured network, such as the internet. It provides encrypted data exchange, strong authentication, and integrity protection, making it the standard method for remote server administration, secure file transfers, and tunneling other protocols.

To initiate a SSH connection, the client first connects to the SSH port, which is 22 by default. The client and server then negotiate the encryption algorithm for setting up a session. The client and server must also set up a shared key, which is used to encrypt the connection. One of the most common mechanisms to set up the shared key is the **Diffie-Hellman** (**DH**) key exchange protocol.

Note: The modern implementation of SSH uses Elliptic-curve Diffie–Hellman (ECDH) key exchange protocol, which is an evolution of the original Diffie-Hellman protocol. DH and ECDH are mathematically equivalent protocols that solve the same problem of secure key exchange using different mathematical foundations, but ECDH is computationally more efficient and produces smaller exchange messages owing to its use of elliptic curves for key exchange message generation. In this section, we will discuss the original DH key exchange protocol.

The DH key exchange is a cryptographic protocol that allows two parties to securely establish a shared secret over an insecure communication channel (like the internet). It enables encrypted communication without pre-shared secrets, forming the basis for protocols like **Transport Layer Security** (**TLS**), SSH, and **virtual private networks** (**VPNs**). DH relies on the mathematical properties of modular exponentiation and the discrete

logarithm problem. To understand how the shared secret is established between the two parties, refer to the following steps:

1. We start with public parameters, i.e., known to all:
   a. A large prime number p is also known as a modulus.
   b. A base number g, also known as a generator, typically a small integer like 2 or 5.
2. Next, we have private keys, which are secret:
   a. Party Alice picks a private key a (random number).
   b. Party Bob picks a private key b (random number).
3. Public keys are computed and exchanged over the network:
   a. Alice computes her public key:
   $$A=g^a \bmod p$$
   b. Bob computes his public key:
   $$B=g^b \bmod p$$
4. Shared secret calculated as follows:
   a. Alice computes the shared secret:
   $$S=B^a \bmod p$$
   b. Bob computes the shared secret:
   $$S=A^b \bmod p$$
5. Both arrive at the same value S due to modular arithmetic properties:
   $$g^{ab} \bmod p = g^{ba} \bmod p$$
   Eavesdroppers see only p, g, A, and B. Calculating a or b from A or B requires solving the discrete logarithm problem, which is computationally infeasible for large primes (e.g., 2048-bit or 4096-bit p). This ensures that even if an attacker intercepts traffic, they cannot decrypt it without the session keys.

To understand SSH authentication, refer to the following steps:

1. **Server authentication with host key verification**: To prevent any host from impersonating as the SSH server, the client verifies the SSH server with its host key as follows:
   a. The server presents its public host key (stored in

~/.ssh/known_hosts on the client).

b. The client checks if this key is trusted (to prevent man-in-the-middle attacks).

c. If the key is unknown, the user is prompted to verify it before proceeding.

2. **Client authentication with password or key-based**: Once the server is verified, the client must authenticate itself using one of these methods:

   a. **Password authentication**: The user enters a password (encrypted over SSH).

   b. **Public key authentication (More secure)**: The client proves ownership of a private key matching a public key stored in **~/.ssh/authorized_keys** on the server.

3. **Encrypted session (Advanced Encryption Standard (AES), ChaCha20, etc.)**: After the client authentication step completes, an encrypted session is set up between the client and the server.

   a. Once authenticated, all communication (commands, file transfers, etc.) is encrypted using symmetric encryption (e.g., AES-256).

   b. The session remains secure until the user disconnects or the session times out.

Configuring key-based authentication

SSH key-based authentication provides a more secure alternative to password logins by using cryptographic key pairs instead of traditional passwords. This method enhances security by eliminating the risk of brute-force attacks while simplifying access through passwordless logins. The following is a step-by-step explanation of how to set it up:

1. On your local machine, create a new SSH key pair using the **ssh-keygen** command. This generates a public key (shared with servers) and a private key (kept secure on your local machine):

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
```

2. The **-t ed25519** parameter specifies the key type, and the **-C** parameter

adds a comment:

```
$ ssh-keygen -t ed25519 -C
"chandan@ubuntu24.mydomain.com"
Generating public/private ed25519 key pair.
Enter file in which to save the key
(/home/chandan/.ssh/id_ed25519):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in
/home/chandan/.ssh/id_ed25519
Your public key has been saved in
/home/chandan/.ssh/id_ed25519.pub
The key fingerprint is:
SHA256:Q8vFWu8eZGoMLcmSi0eoLFGfLQLLejyD+aBGYwyd
V+M chandan@ubuntu24.mydomain.com
The key's randomart image is:
+--[ED25519 256]--+
|                 |
|      o  .       |
|.... o .. +      |
|o+o..+E= B .     |
|=...= = S . +    |
|.@ o + o = =     |
|O O . o   + o    |
|.* o .   . . .   |
|o .         .    |
+----[SHA256]-----+
```

3. The public key is saved at **/home/chandan/.ssh/id_ed25519.pub**:

```
$ cat /home/chandan/.ssh/id_ed25519.pub
ssh-ed25519                    AAAAC3NzaC1l...zxahDcO
chandan@ubuntu24.mydomain.com
```

4. Use **ssh-copy-id** to securely transfer your public key to the remote server:

```
$ ssh-copy-id -i <path to public key> <ssh server>
```

In the following example, the **ssh** public key is copied to the server 192.168.122.248:

```
$ ssh-copy-id -i /home/chandan/.ssh/id_ed25519.pub 192.168.122.248
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/home/chandan/.ssh/id_ed25519.pub"
The authenticity of host '192.168.122.248 (192.168.122.248)' can't be established.
ED25519 key fingerprint is SHA256:PHR4j2h+8zI84jjnnjqaKZuU9EbL0zMYU3W3+OiSqA4.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
chandan@192.168.122.248's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh '192.168.122.248'"
and check to make sure that only the key(s) you wanted were added.
```

Now you can confirm that we can log in to the host without a password:

```
chandan@ubuntu24:~$ ssh 192.168.122.248
Enter passphrase for key '/home/chandan/.ssh/id_ed25519':
```

```
Last login: Sat Aug  2 17:55:53 2025 from 192.168.122.61
[chandan@rocky9 ~]$
```

In the preceding example, you will be prompted to provide the password for loading the **ssh** public key. This can be avoided by using the **ssh-agent** and adding the public key to it:

```
chandan@ubuntu24:~$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-CvUnm3O0NDe5/agent.1947; export SSH_AUTH_SOCK;
SSH_AGENT_PID=1948; export SSH_AGENT_PID;
echo Agent pid 1948;
```

Now you can export the environment variables generated by the **ssh-agent** command and add your **ssh** public key to the agent:

```
chandan@ubuntu24:~$ ssh-add
Enter passphrase for /home/chandan/.ssh/id_ed25519:
Identity added: /home/chandan/.ssh/id_ed25519 (chandan@ubuntu24.mydomain.com)
```

Now you can continue to log in to the server multiple times without being prompted to load the **ssh** public key every time:

```
chandan@ubuntu24:~$ ssh 192.168.122.248
Last login: Sat Aug  2 18:04:12 2025 from 192.168.122.61
[chandan@rocky9 ~]$
logout
Connection to 192.168.122.248 closed.
chandan@ubuntu24:~$ ssh 192.168.122.248
Last login: Sat Aug  2 18:25:10 2025 from 192.168.122.61
[chandan@rocky9 ~]$
```

5. For improved security, disable password authentication on the remote server:

   The **sshd** configuration file **/etc/ssh/sshd_config** should be edited to set the following options:

```
PasswordAuthentication no
```

```
PubkeyAuthentication yes
```

Restart the SSH service to load the updated configuration for the SSH daemon.

```
$ sudo systemctl restart sshd
```

SSH port forwarding

SSH port forwarding, also known as **SSH tunnelling**, enables the redirection of network traffic from one port to another, either locally or across remote systems. This technique is particularly useful for bypassing firewalls, accessing restricted services, or encrypting traffic that would otherwise be transmitted in plaintext. There are three primary types of SSH port forwarding:

- **Local port forwarding**: Redirects traffic from a local port to a remote server, allowing secure access to services running on a private network.

  For example, to access a private database (MySQL on port 3306) hosted on a remote server (**remote-server.com**) from your local machine, use the following command:

  ```
  $ ssh -L 3306:localhost:3306 user@remote-server.com
  ```

  This will connect the port 3306 on your local machine to the port 3306 of remote-server.com

- **Remote port forwarding**: Makes a local service available on a remote machine, often used to expose internal applications to the internet securely.

  For example, to expose a local service like a web app on your host, you can connect the local port of the application to a port on a public server using the following command:

  ```
  $ ssh -R 8080:localhost:8000 user@public-server.com
  ```

  This command will connect the port 8000 on your local host to the port 8080 on the public-server.com.

- **Dynamic port forwarding**: Creates a flexible proxy, routing all traffic through the SSH server for full encrypted browsing and application access.

For example, to route all browser traffic through a remote SSH server (proxy-server.com) for secure, encrypted browsing, dynamic port forwarding can be used as follows:

```
$ ssh -D 1080 user@proxy-server.com
```

This will create a dynamic proxy that is agnostic to the application protocol and can handle both TCP and **User Datagram Protocol** (**UDP**) traffic. This is called a **SOCKS proxy**. To use this, you can configure your web browser to use a SOCKS proxy configuration of localhost and port 1080.

By leveraging SSH port forwarding, users can enhance security, circumvent network restrictions, and securely manage services without exposing them directly to untrusted networks.

Network File System

The NFS is a distributed file-sharing protocol that allows Linux and Unix systems to access files and directories over a network as if they were stored locally. Originally developed by *Sun Microsystems* in the 1980s, NFS remains a fundamental technology for sharing storage resources across servers, workstations, and data centers.

NFS operates on a client-server model, where an NFS server exports specific directories, making them available to authorized NFS clients. These clients can then mount the remote directories into their local filesystem hierarchy, enabling seamless file access just like local storage. This transparency makes NFS particularly useful for centralized storage management, allowing multiple systems to work with the same files simultaneously.

One of NFS's key strengths is its stateless design, meaning the server does not track client activity between requests. This improves reliability, as client disconnections do not disrupt operations when a client reconnects, it simply resumes file access. Modern NFS implementations (NFSv4 and later) also include enhanced security features such as Kerberos authentication and stronger file-locking mechanisms.

NFS is widely used in enterprise environments for scenarios like shared home directories, application data storage, and virtualization host storage. When appropriately configured with appropriate permissions and network security, it provides an efficient way to centralize data while maintaining performance and accessibility across networked systems.

Configuring NFS server

NFS allows sharing directories between Linux/Unix systems over a network, also called **NFS exports**. The following is a step-by-step guide to setting up an NFS server and client:

1. To install the NFS server packages, run the following command on Ubuntu:

```
$ sudo apt install nfs-kernel-server
```

2. On Rocky Linux, use the following command to install the packages:

```
$ sudo yum install nfs-utils
```

3. To export a directory from the NFS server, we will first prepare the path that we want to share. This can be done using the following commands:

```
chandan@ubuntu24:~$ sudo mkdir -p /srv/nfs_share
chandan@ubuntu24:~$ touch /srv/nfs_share/file-on-nfs
chandan@ubuntu24:~$ sudo chown nobody:nogroup
/srv/nfs_share
chandan@ubuntu24:~$ sudo chmod 777 /srv/nfs_share
```

4. Next, we will edit the NFS configuration to share the path as an NFS export:

```
chandan@ubuntu24:~$ sudo vi /etc/exports
/srv/nfs_share 192.168.122.0/24(rw,sync,no_subtree_check)
```

5. Finally, run the following commands to load the updated NFS configuration:

```
chandan@ubuntu24:~$ sudo exportfs -a
```

Configuring NFS client

The NFS share can be accessed by the clients by mounting it on the local filesystem. The files on the NFS share can then be accessed as if the file were locally available on your host.

The following steps describe the configuration of the NFS client to access an NFS share or export:

1. The NFS clients that will access the NFS exports from the server should install the following packages:

```
[chandan@rocky9 ~]$ sudo apt install nfs-common
```

2. On Rocky Linux, the NFS client is part of the **nfs-utils** package.
3. Next, prepare the directory where the NFS shares will be mounted using the following code:

```
[chandan@rocky9 ~]$ sudo mkdir -p /mnt/nfs_client
```

4. You can check the shares available on the NFS server with the following command:

```
[chandan@rocky9 ~]$ showmount -e 192.168.122.61
Export list for 192.168.122.61:
/srv/nfs_share 192.168.122.0/24
```

5. Finally, you can mount the NFS export from the server:

```
[chandan@rocky9 ~]$ sudo mount -t nfs   nfs-server-host-or-ip:/srv/nfs_share /mnt/nfs_client
[chandan@rocky9 ~]$ sudo mkdir -p /mnt/nfs_client
[chandan@rocky9 ~]$ sudo mount -t nfs 192.168.122.61:/srv/nfs_share /mnt/nfs_client
```

6. To check the NFS shares mounted on the client, use the mount command with type **nfs 4**:

```
[chandan@rocky9 ~]$ mount -t nfs4
192.168.122.61:/srv/nfs_share on /mnt/nfs_client type nfs4 (rw,relatime,vers=4.2,rsize=131072,wsize=131072,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=sys,clientaddr=192.168.122.248,local_lock=none,addr=192.168.122.61)
```

7. Files on the NFS share can be accessed at **/mnt/nfs_client**:

```
[chandan@rocky9 ~]$ ls /mnt/nfs_client
file-on-nfs
```

NFS is a convenient way to share files across Unix/Linux systems, but its design prioritizes ease of use over security. Without proper hardening, NFS can expose sensitive data or become an attack vector. The following are some of the key things to keep in mind to make NFS secure:

- Restrict access with exports by explicitly providing access to subnets:

```
/shared 192.168.1.0/24(ro,sync,no_root_squash)
/data 10.0.0.100(rw,sync)
```

- Use **no_root_squash**; by default, NFS maps root (UID 0) on the client to nobody using the **root_squash** sharing option. This can be disabled using the **no_root_squash** option. This will let remote root users on NFS clients act as local root on the NFS server, leading to privilege escalation. The **all_squash** option maps all remote users to unprivileged users.
- Ensure files have restrictive permissions.

NFS also provides a convenient way to share files across Unix/Linux systems, but to ensure system security, making proper configuration is essential. By restricting exports to trusted clients and disabling insecure options like **no_root_squash**, administrators can mitigate risks such as unauthorized access and data interception. Firewall rules, network segmentation, and consistent UID/GID mappings further enhance security.

Ultimately, a well-hardened NFS deployment balances functionality and security, ensuring reliable file access without exposing systems to unnecessary vulnerabilities. Proper monitoring and access controls are key to maintaining a secure NFS infrastructure.

Hypertext Transfer Protocol

HTTP serves as the foundational communication protocol of the modern web, enabling the exchange of information across the internet. Developed initially in the early 1990s to facilitate hypertext document sharing, HTTP

has evolved into the primary mechanism for data transfer between web browsers and servers, supporting today's rich digital ecosystem of websites, applications, and services.

Operating as a request-response protocol in the application layer of the Internet Protocol Suite, HTTP follows a simple yet powerful client-server model where web browsers or other user agents act as clients that initiate requests, and web servers process these requests and return appropriate responses. The protocol's stateless nature means each request from a client to a server contains all the information necessary to understand and complete that request, without the server needing to retain session information between requests.

HTTP's design emphasizes extensibility and flexibility, allowing it to transport not just hypertext documents, but virtually any type of digital content, including images, videos, and application data. The protocol supports various request methods - such as GET for retrieving resources, POST for submitting data, and PUT for updating resources. Modern implementations typically use HTTP over **Secure Sockets Layer** (**SSL**)/**Transport Layer Security** (**TLS**) encryption, **Hypertext Transfer Protocol Secure** (**HTTPS**) to protect data integrity and confidentiality, making it safe for sensitive communications like online banking and e-commerce.

As the backbone of web communication, HTTP's standardized operation enables interoperability between different systems and technologies, allowing users to seamlessly access information from servers anywhere in the world. Its continued evolution through versions (from HTTP/1.0 to HTTP/2 and HTTP/3) demonstrates its ongoing importance in addressing performance and security requirements of an increasingly complex digital landscape. Understanding HTTP is essential for web developers, system administrators, and anyone working with internet technologies, as it underpins nearly all web-based communication and services we use daily.

Configuring NGINX HTTP szerver

NGINX is a lightweight, scalable web server ideal for hosting static and dynamic content. By configuring server blocks, optimizing static file

delivery, and enabling proper logging, you can efficiently serve web applications:

```
$ sudo apt install nginx
$ sudo systemctl start nginx
$ sudo systemctl enable nginx
```

NGINX configuration files are stored in **/etc/nginx/**, the main configuration file is **/etc/nginx/nginx.conf**. This file includes other configuration files, which are specific to websites hosted on the server. This can be seen in the main configuration file as follows:

```
include /etc/nginx/sites-enabled/*;
```

A web server can host multiple websites through the configuration of virtual hosts. Virtual Hosts (also called **server blocks** in NGINX) allow you to host multiple websites or domains on a single web server. Instead of requiring separate physical machines for each site, NGINX uses the requested domain name (server_name) to determine which content to serve.

There are two types of virtual hosts:

- **Name-based**: Uses the domain name (e.g., example.com vs test.com).
- **IP-based**: Uses different IP addresses (less common).

The websites hosted on the server are configured using files stored at **/etc/nginx/sites-available/**.

For our lab, we will create a site called **example.com**.

1. The configuration for this site is created at **/etc/nginx/sites-available/example.com** with the following configuration:

```
# example.com
server {
    listen 80;
    server_name example.com www.example.com;
    root /var/www/example.com/html;
    index index.html index.htm;
    location / {
        try_files $uri $uri/ =404;
```

```
    }
}
```

The listen 80 tells NGINX to listen on HTTP port 80, **server_name** configures the virtual host name for the website.

2. The following root directive configures the directory where your website files are stored:

```
$ sudo mkdir -p /var/www/example.com/html
$ sudo chown -R www-data:www-data /var/www/example.com/html
$ echo "<h1>Welcome to Example.com</h1>"|sudo tee /var/www/example.com/html/index.html
```

3. To enable the site, the configuration of this website should be available **/etc/nginx/sites-enabled/**. To do this, we will create a symlink as follows:

```
$ sudo ln -s /etc/nginx/sites-available/example.com /etc/nginx/sites-enabled/
```

4. Now we can verify if the configuration of NGINX is valid with **nginx -t**. This can help detect issues with the syntax of the configuration:

```
chandan@ubuntu24:~$ sudo nginx -t
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

5. Once the configuration is validated, the **nginx** server is reloaded with the updated configuration as follows:

```
$ sudo systemctl reload nginx
```

6. To test the website, we will ensure that the domain example.com gets resolved to the IP Address of the web server. This is normally done using a CNAME to point the domain name to the hostname of the web server. For local testing, we can add the mapping of the domain name to the IP of the web server in **/etc/hosts**:

```
chandan@ubuntu24:~$ cat /etc/hosts
127.0.0.1 localhost
```

```
127.0.1.1 ubuntu24
127.0.0.1 example.com www.example.com
```

7. Now, you can use the following **curl** command to check the website as follows:

```
chandan@ubuntu24:~$ curl http://example.com
<h1>Welcome to Example.com</h1>
```

As we have configured the website as a virtual host, you will get a different result if you access the server using its IP Address.

Configuring SSL certificates on NGINX for HTTPS

HTTPS is the secure version of HTTP, encrypting all communication between a user's browser and a website. HTTPS uses TLS/SSL to encrypt data, preventing hackers from intercepting sensitive information like login credentials, personal messages, etc. In this section, we will use self-signed certificates to secure our website. To do this, first, we will create a self-signed certificate for the website with OpenSSL. A self-signed certificate allows you to encrypt traffic between your NGINX server and clients. While browsers do not trust it, it is useful for testing, internal services, or development environments.

1. We will first install the OpenSSL package:

```
$ sudo apt install openssl
```

2. Run the following command to create the certificate:

```
$ sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
-keyout /etc/ssl/private/nginx-selfsigned.key \
-out /etc/ssl/certs/nginx-selfsigned.crt
```

When prompted, please enter the details for the certificates. Here is an example:

```
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:Karnataka
Locality Name (eg, city) []:Bengaluru
Organization Name (eg, company) [Internet Widgits Pty
```

```
Ltd]:Example.com
Organizational Unit Name (eg, section) []:IT
Common Name (e.g. server FQDN or YOUR name)
[]:example.com
Email Address []:admin@example.com
```

3. Now we will update the NGINX configuration for the website at **/etc/nginx/sites-available/example.com** as follows:

```
# example.com
server {
        listen 80;
        server_name example.com www.example.com;
        root /var/www/example.com/html;
         return 301 https://$host$request_uri;
}
server {
        listen 443 ssl;
        server_name example.com www.example.com;
        ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
         ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;

         root /var/www/example.com/html;
         index index.html index.htm;

         location / {
                try_files $uri $uri/ =404;
         }
}
```

Note that we have also added a directive to redirect any access to the HTTP port 80 to the HTTPS website.

4. Now, to test the website, we can use the **curl** commands as follows:

```
chandan@ubuntu24:~$ curl http://example.com
<html>
```

```
<head><title>301 Moved Permanently</title></head>
<body>
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/1.24.0 (Ubuntu)</center>
</body>
</html>
```

5. The access to the website at HTTP port now redirects to the HTTPS URL with a 301 HTTP code; we can use **curl -L** to follow the redirection. We will also use the **-k** option with **curl** to disable the verification of the certificate of the website:

```
chandan@ubuntu24:~$ curl -kL  http://example.com
<h1>Welcome to Example.com</h1>
chandan@ubuntu24:~$
```

6. The **-k** option is used in the preceding code as the host does not trust the self-signed certificate used by the website. This can be fixed by configuring the client system to trust our self-signed certificate. To trust the certificate of the website, we can add it to the client system's certificate trust store. This can be done as shown in the following code:

```
chandan@ubuntu24:~$ sudo cp /etc/ssl/certs/nginx-selfsigned.crt /usr/local/share/ca-certificates/
chandan@ubuntu24:~$ sudo update-ca-certificates
Updating certificates in /etc/ssl/certs...
rehash: warning: skipping ca-certificates.crt,it does not contain exactly one certificate or CRL
rehash: warning: skipping duplicate certificate in nginx-selfsigned.pem
1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
```

7. With the certificate added to the client system’s trust store, the **-k** option can be removed now:

```
chandan@ubuntu24:~$ curl https://example.com
<h1>Welcome to Example.com</h1>
chandan@ubuntu24:~$
```

Using NGINX as a reverse proxy

A reverse proxy is a server that sits between clients (like web browsers) and backend servers [such as application servers or **application programming interfaces** (**APIs**)], acting as an intermediary to manage and forward requests. Unlike a forward proxy, which hides clients from servers, a reverse proxy hides servers from clients, providing a single point of contact for incoming traffic. When a client makes a request, the reverse proxy evaluates it and forwards it to the appropriate backend server. Once the backend processes the request, the reverse proxy sends the response back to the client, making it appear as if the proxy itself is the origin server.

This architecture offers several advantages, including improved security, load balancing, and performance optimization. By shielding backend servers from direct exposure to the internet, a reverse proxy helps prevent attacks like **distributed-denial-of-service** (**DDoS**) and unauthorized access. It can also distribute traffic across multiple servers to ensure high availability, cache static content to reduce latency, and handle SSL/TLS encryption to offload the computational burden from backend systems. Additionally, reverse proxies enable advanced routing, allowing different paths or domains to be directed to specific services, useful for microservices architectures or multi-application deployments. Common use cases include web acceleration, API gateways, and serving as an entry point for containerized applications in cloud environments. Popular software for reverse proxying includes NGINX and Apache HTTP Server (with **mod_proxy**).

Refer to the following steps to configure NGINX as a reverse proxy:

1. To configure NGINX as a reverse proxy, we need to add a location stanza to our configuration and use the **proxy_pass** option to forward the connection to a backend server. We also set some headers in the HTTP request while forwarding the connection to the backend, this is

to preserve the original address of the client making the request:

```
server {
        listen 443 ssl;
        server_name example.com www.example.com;
        ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;

        root /var/www/example.com/html;
        index index.html index.htm;

        location / {
                try_files $uri $uri/ =404;
        }
       location /app1 {
                proxy_pass http://localhost:3001; # Forward to app port
3001
                rewrite ^/app1/(.*)$ /$1 break;
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;

        }
}
```

2. With this configuration, we are now ready to deploy a web app running at localhost and port 3001, which can be accessed by making a request to **https://example.com/app1/**. Note that we have also added a directive to rewrite the request to remove the **/app1/** prefix from the request.

   To test the preceding configuration, we can use the following commands to start a web server on port 3001:

```
chandan@ubuntu24:~$ mkdir app1/
chandan@ubuntu24:~$ cd app1/
chandan@ubuntu24:~/app1$ echo app1 > index.html
chandan@ubuntu24:~/app1$ python3 -m http.server -b localhost
```

```
3001
Serving HTTP on 127.0.0.1 port 3001 (http://127.0.0.1:3001/) ..
```

3. A client connection can now be made as follows:

```
[chandan@rocky9 ~]$ curl -k
https://example.com/app1/index.html
app1
```

4. To view the headers set by the proxy, we can run a listener on port 3001 with **netcat** command. In the following output, you can see the header **X-Real-IP** preserves the IP Address of the client that can be accessed by the backend server to know the original source of the request:

```
chandan@ubuntu24:~/app1$ nc -l 3001
GET /index.html
HTTP/1.0
Host: example.com
X-Real-IP: 192.168.122.248
Connection: close
User-Agent: curl/7.76.1
Accept: */*
```

Configuring multiple backend servers with upstream

In the preceding example, we went through the configuration for a simple deployment of a reverse proxy with only one backend; there can be multiple backend servers behind a reverse proxy. This kind of setup is used for load balancing of requests and high availability. To add multiple backend servers for a reverse proxy location, the configuration can be updated as follows:

```
upstream backend {
    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
}
server {
```

```
        listen 443 ssl;
        server_name example.com www.example.com;
        ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
         ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;

         root /var/www/example.com/html;
         index index.html index.htm;

         location / {
                 try_files $uri $uri/ =404;
         }

         location /app1 {
                 proxy_pass http://backend;  # Forward to local app on port
3000
                 rewrite ^/app1/(.*)$ /$1 break;
                 proxy_set_header Host $host;
                 proxy_set_header X-Real-IP $remote_addr;
         }
}
```

Using health checks for backend servers

NGINX's passive health checks monitor backend servers (upstreams) implicitly by analyzing real client request outcomes, without sending separate health probes. If a server fails to respond correctly to actual traffic, NGINX temporarily stops forwarding requests to it, improving reliability without additional overhead.

Failure tracking is done with **max_fails** and **fail_timeout**. These can be configured in the upstream block; these parameters define when a server is marked unhealthy:

```
upstream backend {
   server 192.168.1.10:3000 max_fails=3 fail_timeout=30s;
   server 192.168.1.11:3000 max_fails=3 fail_timeout=30s;
}
```

The **max_fails=3** indicates that after 3 consecutive failed requests, the server is marked *unhealthy*.

The **fail_timeout=30s** indicates that NGINX stops sending traffic for 30 seconds when a server is marked as unhealthy. After the **fail_timeout** period, NGINX retries the server.

The following example shows a complete example of an upstream passive health check:

```
http {
    upstream backend {
        server 192.168.1.10:3000 max_fails=3 fail_timeout=30s;
        server 192.168.1.11:3000 max_fails=3 fail_timeout=30s;
    }

    server {
        listen 80;
        location / {
            proxy_pass http://backend;
            proxy_next_upstream error timeout http_500 http_502 http_503;
            proxy_connect_timeout 2s;
            proxy_read_timeout 5s;
        }
    }
}
```

In the preceding example:

- **proxy_next_upstream** defines which errors trigger a failover (e.g., timeouts, HTTP 500).
- Timeouts ensure quick detection of unresponsive servers.

You can also define a fallback server with the backup option, which is used only when all primaries are down, as follows:

```
upstream backend {
    server 192.168.1.10:3000;
    server 192.168.1.11:3000;
    server 192.168.1.12:3000 backup;  # Activated only if 10/11 fail
```

```
}
```

Using NGINX cache to improve server performance

NGINX can cache responses from backend servers (like APIs or web apps) to reduce load and improve performance. This can be done with the **proxy_cache_path** to set up the cache and associate it with a location using the **proxy_cache** directive:

```
http {
    proxy_cache_path /var/cache/nginx levels=1:2
keys_zone=my_cache:10m inactive=60m;
    upstream backend {
        server 192.168.1.10:3000;
     }

     server {
        listen 80;
        server_name example.com;

        location / {
            proxy_cache my_cache;
            proxy_pass http://backend;
            proxy_cache_valid 200 302 10m;
            proxy_cache_valid 404 1m;
            add_header X-Cache-Status $upstream_cache_status;
        }

        location ~ /purge(/.*) {
            proxy_cache_purge my_cache
"$scheme$request_method$host$1";
        }
    }
}
```

In the preceding example, the following points should be noted:

- **/var/cache/nginx** is the path for storage of cached files.

- **levels=1:2** refers to the levels parameter in NGINX's **proxy_cache_path** directive, which defines the directory hierarchy structure for storing cached files on disk. It controls how NGINX organizes cached content to optimize file lookup performance and avoid issues with too many files in a single directory (which can slow down filesystems).
- **keys_zone=my_cache:10m** configures a shared memory zone of 10MB for cache keys.
- **inactive=60m** specifies that cached items unused for 60 minutes will be deleted.

Conclusion

This chapter explored essential network services that form the backbone of modern IT infrastructure. We began with DNS, the critical service that translates human-readable domain names into IP addresses, ensuring seamless connectivity across the internet and private networks. We examined its architecture, configuration (using BIND), and troubleshooting techniques, including dynamic updates and secure zone management.

Next, we discussed NTP, which synchronizes system clocks across devices to maintain accuracy for logging, security, and distributed systems. Proper NTP configuration prevents time drift, ensuring reliable authentication and coordination in networked environments.

The chapter then covered SSH, a fundamental protocol for secure remote administration. We reviewed key-based authentication and port forwarding. We also explored NFS, a distributed file-sharing protocol that enables seamless access to remote storage. We covered server/client setup, permissions, and security considerations to ensure efficient and safe file sharing. Finally, we examined HTTP, the foundation of web communication. While focusing on its role in serving web content, we also highlighted secure alternatives like HTTPS and the importance of encryption in modern web services.

Together, these services, DNS, NTP, SSH, NFS, and HTTP, enable reliable,

secure, and efficient network operations. Mastering their configuration and maintenance is crucial for any system administrator or network engineer, ensuring a robust infrastructure capable of supporting both internal and public-facing applications.

In the next chapter, we will look at load balancing on Linux and explore various options available on Linux to scale services by sharing load across multiple servers.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 6

Load Balancing and High Availability

Introduction

In the architecture of modern network services, the ability to efficiently distribute incoming traffic across a group of backend servers is a fundamental requirement for achieving scalability, high availability, and performance. This chapter delves into the practical implementation of this critical function within the Linux operating system, exploring the built-in software capabilities that transform a standard Linux instance into a powerful and versatile load balancer. We will move beyond theoretical concepts to examine the concrete tools and methodologies, from the well-established **IP Virtual Server** (**IPVS**) integrated directly into the kernel to the flexible proxy-based approach of HAProxy. Our focus will be on understanding how these technologies intelligently direct client requests, manage server pools, and ensure resilience against failures, thereby forming the robust, software-defined foundation upon which reliable and responsive applications are built.

Structure

This chapter covers the following topics:

- High availability at Layer 3
- Installation and configuration of Keepalived
- IPVS and iptables load balancing
- High availability at Layer 4
- Using BGP for high availability
- Layer 7 load balancers
- Installation and configuration of HAProxy

Objectives

The primary objective of this chapter is to equip you with a comprehensive understanding of how to architect, implement, and manage a software-based load balancer within the Linux environment. We will achieve this by moving from fundamental concepts to practical execution, first establishing a solid theoretical groundwork on load distribution algorithms and network traffic flow. Building upon this foundation, the chapter will provide a detailed exploration of the two predominant Linux load balancing technologies, such as high-performance, kernel-space IPVS and the feature-rich, application-layer HAProxy. Finally, our aim is to synthesize this knowledge into a practical framework, guiding you through the design considerations, configuration steps, and operational best practices required to deploy a robust, fault-tolerant load balancing solution that enhances the scalability and resilience of your service infrastructure.

High availability at Layer 3

High availability at Layer 3, the network layer, is fundamentally concerned with the preservation of a consistent and reachable **Internet Protocol** (**IP**) address, even in the event of a complete failure of the physical or **virtual machine** (**VM**) currently owning it. In the context of a Linux load balancing infrastructure, this is not about the health of individual backend

servers but about ensuring the continuous availability of the load-balancer itself, which acts as the critical single point of entry for all incoming traffic. The core strategy for achieving this is through a **virtual IP address** (**VIP**) and a protocol that governs which node in a cluster is responsible for announcing and handling traffic for that VIP at any given time. The de facto standard for this purpose on Linux is the **Virtual Router Redundancy Protocol** (**VRRP**), most implemented by the robust Keepalived software.

The principle of operation hinges on a primary-secondary cluster architecture. Two or more Linux servers are configured to participate in a VRRP group. This group is assigned a unique virtual router ID and, most importantly, a shared VIP address. This VIP is the address that clients and upstream networks use to reach the load-balanced service. Within this VRRP group, one node is elected as the primary, while the others remain in a backup state. The primary node has the crucial duty of holding the VIP on its local network interface and responding to **Address Resolution Protocol** (**ARP**) requests for it, thereby assuming responsibility for all traffic destined for that IP address. It actively performs its load balancing duties, distributing connections to the backend pool.

The core of the high-availability mechanism is the continuous communication between cluster members through multicast (or optionally unicast) VRRP advertisements. The primary or master node periodically sends out these advertisements, a signal to all backup nodes that it is still alive and operational. Should these advertisements cease for a period longer than a pre-configured timeout, the backup nodes interpret this silence as a failure of the master. This triggers an election process among the remaining healthy nodes, based on a configured priority value. The node with the highest priority will transition to the master state, immediately claim the VIP for itself by configuring it on its own interface, and send a gratuitous ARP broadcast to inform the entire local network that the MAC address associated with the VIP has now changed. This ARP update is critical as it repoints all layer 2 switching infrastructure to the new master almost instantaneously. The following figure shows the message exchange involved in VIP election:

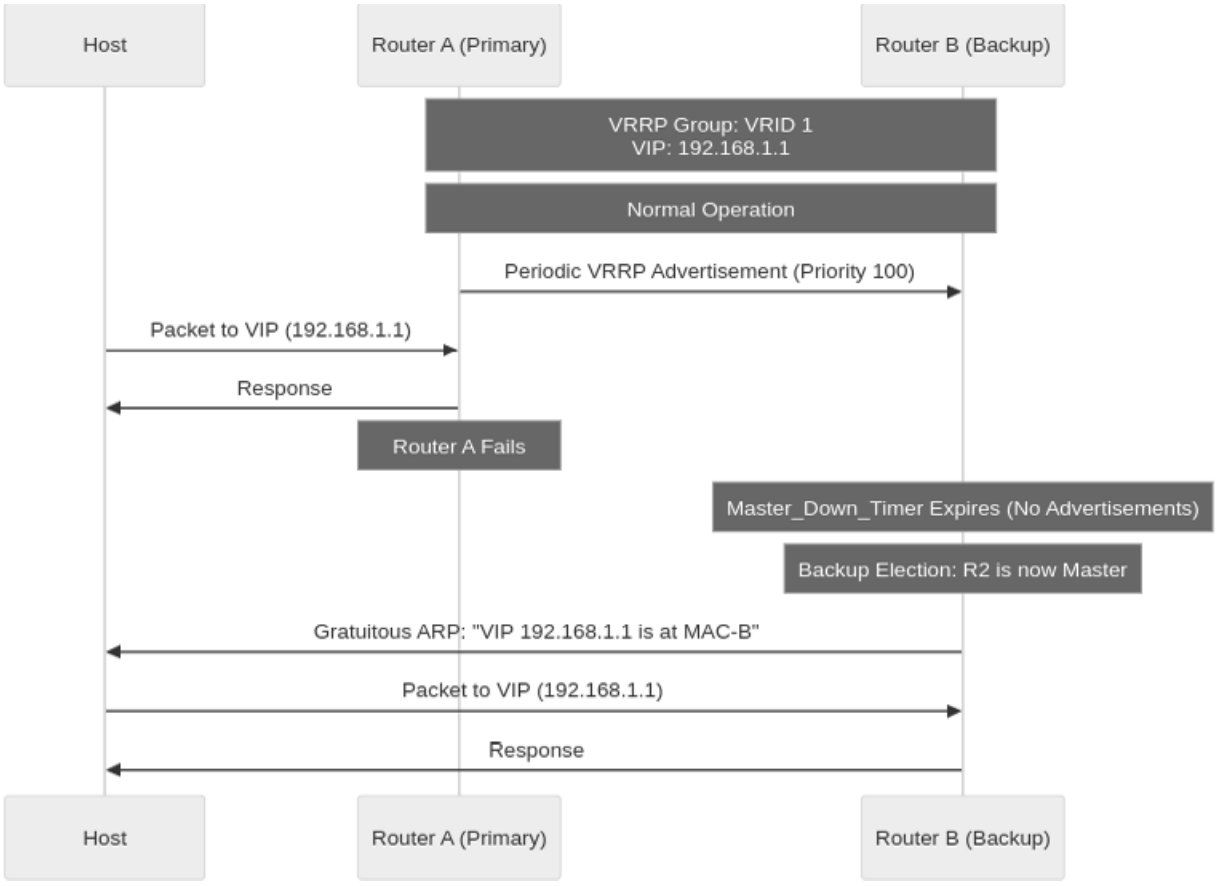


***Figure 6.1**: Working of the VRRP protocol*

From the perspective of the client and the network, the transition is nearly seamless. A brief interruption in established **Transmission Control Protocol** (**TCP**) connections may occur during the failover period, as the new master has no knowledge of the stateful connections that were being handled by the failed node. However, new connection attempts will be immediately picked up by the new master, making the service itself appear continuously available. This stateless failover is perfectly adequate for many web services and protocols where clients are designed to retry failed requests.

Installation and configuration of Keepalived

Keepalived is the most popular application for building robust, high-availability infrastructures on Linux. It is a daemon that provides two

fundamental, complementary sets of features: first, it implements the VRRP to manage the failover of a VIP between nodes; and second, it performs sophisticated health checking of local services and system resources, allowing it to make intelligent decisions about whether a node is fit to hold the master role. It is this combination of network-level redundancy and application-aware monitoring that makes Keepalived so powerful, moving beyond simple heartbeat to a true assessment of a node's health.

Keepalived is available in the standard repositories for both Ubuntu and Rocky Linux, making installation very straightforward.

The following commands can be used to install the package on Ubuntu and Rocky Linux:

```
$ sudo apt install keepalived # On Ubuntu
$ sudo dnf install keepalived # On Rocky Linux
```

Once the package is installed, the service can be started with the following **systemd** command:

```
$ sudo systemctl enable --now keepalived
```

The operation of Keepalived is governed by a single, central configuration file, typically **/etc/keepalived/keepalived.conf**. The following is an example of a Keepalived configuration:

```
# Global definitions for the daemon
global_defs {
   notification_email {
      admin@yourdomain.com  # Email for alerts (requires local mail setup)
   }
   notification_email_from keepalived@localhost
   smtp_server 127.0.0.1
   smtp_connect_timeout 30
   router_id LVS_MASTER_01  # A unique identifier for this node
}

# Define the VRRP instance
vrrp_instance VI_1 {
```

```
    state MASTER            # On the master node. On backup node, use
'BACKUP'
    interface eth0          # The network interface to use     virtual_router_id
51    # ID for group (0-255). MUST be same on both nodes.
    priority 100            # Election priority (1-254). Master must be higher.

    advert_int 1            # Time in seconds between VRRP advertisements

    # Authentication for VRRP packets (simple password)
    authentication {
        auth_type PASS      # Use password authentication
        auth_pass mySecret # max 8 chars. MUST be same on both nodes.
    }

    # The virtual IP address that will be shared and moved between nodes
    virtual_ipaddress {
        192.168.1.100/32 dev eth0  # The VIP with a /32 mask on the
interface
    }
}
```

This file is structured into several key blocks, each defining a different aspect of the daemon's behavior. The entire configuration is built around the concept of a vrrp_instance, which defines a single group of hosts participating in VRRP for a specific VIP. A single Keepalived daemon can manage multiple vrrp_instances, allowing you to manage redundancy for multiple services or networks from one node.

A minimal but functional vrrp_instance configuration begins with the declaration of the instance itself, such as **VI_1**. The state directive, while often configured, is largely symbolic, and the election process determines the true state. The critical parameter is interface, which specifies the network interface (e.g., **eth0**, **ens192**) on which VRRP advertisements will be sent, and the VIP will be hosted. The virtual_router_id is a numeric identifier that must be unique within the same broadcast domain; all nodes that are to be part of the same redundancy group must share this identical ID.

The priority value, typically ranging from 1 to 255, is central to the election process; the node with the highest priority becomes the master, assuming it passes its health checks. The **advert_int** defines the interval, in seconds, between VRRP advertisements sent by the master to signal its liveness.

On the backup node, the configuration is almost the same except for the following changes:

```
# Global definitions for the daemon
global_defs {
   notification_email {
      admin@yourdomain.com  # Email for alerts (requires local mail
setup)
   }
   notification_email_from keepalived@localhost
   smtp_server 127.0.0.1
   smtp_connect_timeout 30
   router_id LVS_BACKUP_01  # A unique identifier for this node
}
vrrp_instance VI_1 {
   state BACKUP          # On the master node. On backup node, use
'BACKUP'
   interface eth0        # The network interface to use     virtual_router_id
51    # ID for group (0-255). MUST be same on both nodes.
   priority 50           # Election priority (1-254). Master must be higher.
```

The heart of the VRRP configuration is the **virtual_ipaddress** block. This section contains one or more lines defining the VIPs that this VRRP instance will manage. The VIP is specified along with its netmask (e.g., 192.168.1.100/24). When a node becomes the master, Keepalived automatically adds this IP address to the specified network interface. Upon transitioning to backup, it just as reliably removes it.

The authentication block within the **vrrp_instance** provides a simple security measure to prevent unauthorized hosts from joining the VRRP group and causing disruption. It can use either PASS, which is a plaintext password, sent in clear text, or an **authentication header** (**AH**), which is an IP authentication header, more secure but rarely supported, alongside a

shared password string.

However, the most critical feature for robust operation is the integration of health checks through the **vrrp_script** directive. This allows you to define a script or command that Keepalived will execute at regular intervals. The exit code of this script determines the health of the node. A return code of 0 indicates success, while any other code indicates failure. A common and powerful use case is to check whether the local load balancer process (like HAProxy or NGINX) is running and functioning correctly, not just that the **operating system** (**OS**) is up.

For example, the following script might check if HAProxy is listening on its port:

```
vrrp_script chk_haproxy {
    script "pidof haproxy"  # Check haproxy process
    interval 2              # Check every 2 seconds
    weight 50               # Value to add to priority on success.
    fall 2                  # Require 2 failures for KO
    rise 2                  # Require 2 successes for OK
}
```

This script block is then linked to the VRRP instance with a **track_script** stanza:

```
track_script {
    chk_haproxy
}
```

This integration is what creates an application-aware cluster. If the **chk_haproxy** script fails (e.g., HAProxy crashes), Keepalived will subtract the weight from the node's priority. If this adjusted priority falls below the priority of a backup node, the local node will voluntarily relinquish the master role, triggering a failover. This ensures that the VIP is always hosted on a node that is not only online but also successfully providing the critical service, thereby guaranteeing true service availability from the client's perspective. Through this elegant yet powerful configuration, Keepalived transforms a pair of Linux servers into a resilient, self-healing gateway for your services.

IPVS and iPTables load balancing

At the heart of high-performance, large-scale load balancing on Linux lies a powerful kernel facility known as IPVS. Operating deep within the kernel's networking stack, IPVS functions as a transport-layer (Layer 4) load balancer, directing incoming network packets to a pool of backend servers with exceptional efficiency and minimal overhead. Unlike application-layer proxies that must terminate and re-establish connections, IPVS simply forwards packets, making decisions based on fundamental information such as IP addresses, port numbers, and a configured scheduling algorithm. This approach allows it to handle massive volumes of traffic with minimal latency, making it the underlying engine for some of the world's most demanding web infrastructures.

The true power of IPVS stems from its integration directly into the Linux kernel, enabling it to process packets at a level close to the raw network hardware. This architectural position provides a significant performance advantage over user-space load balancers, which must constantly copy data between kernel and user space, consuming valuable **central processing unit** (**CPU**) cycles and memory bandwidth. IPVS avoids this cost altogether, resulting in a highly scalable solution capable of managing tens of thousands of concurrent connections and gigabits of traffic on modest hardware.

To manage this kernel-level subsystem, the primary tool is **ipvsadm**, a user-space command-line utility that provides an administrative interface for defining virtual services, configuring pools of real servers, and setting the load balancing algorithms that determine traffic distribution. While IPVS handles the high-speed packet forwarding itself, it is frequently paired with a companion daemon like Keepalived, which assumes the critical roles of providing high availability through VRRP and monitoring the health of the backend servers. Together, they form a complete, robust, and enterprise-grade software load balancing solution that offers a compelling alternative to expensive hardware appliances, providing the foundational layer upon which scalable and resilient cloud-native and on-premises services are built.

Installation and configuration of IPVS

Before leveraging the high-performance load balancing capabilities of IPVS, the necessary software must be installed and the kernel modules loaded. It is important to understand that IPVS is not a standalone application but a kernel feature. The **ipvsadm** package provides the essential user-space administration tool to interact with and configure this kernel subsystem.

The installation process is straightforward and consistent across most modern Linux distributions. The primary requirement is to install the **ipvsadm** package, which will automatically handle loading the necessary kernel modules. Use the following command to install the package for **ipvsadm**:

```
$ sudo yum install ipvsadm   # For older CentOS
$ sudo dnf install ipvsadm   # For newer CentOS, Fedora, Rocky

$ sudo apt install ipvsadm   # For Ubuntu
```

Upon installation, the kernel modules (i**p_vs, ip_vs_rr, ip_vs_wlc**, etc.) are typically loaded automatically. You can verify this with the **lsmod** and **grep** commands:

```
$ sudo lsmod | grep ip_vs
```

This should return a list of loaded modules, with **ip_vs** at the core and others for specific schedulers.

To ensure the modules load on system boot, they can be added to the **/etc/modules** file on Ubuntu systems. On Rocky Linux systems, this is usually handled automatically by the **ipvsadm** service.

Configuring IPVS is a process of defining two core constructs—the virtual service and the pool of real servers that support it. This configuration is primarily accomplished using the **ipvsadm** command-line utility, which serves as the administrative interface to the IPVS kernel subsystem. Unlike configuration managed through a persistent file, **ipvsadm** is typically used to manipulate the IPVS routing table directly in memory, making changes immediately effective but also transient; a system reboot will wipe the configuration. Therefore, the commands are almost always executed from

within a persistent startup script or are managed by a higher-level orchestration tool like Keepalived, which can dynamically update IPVS based on health checks and failover events.

The first step in any IPVS configuration is to define a virtual service, also known as a **virtual server**. This represents the public-facing endpoint to which clients will connect. The command structure for adding a virtual service is:

```
ipvsadm -A -t|u|f <vip:port> -s <scheduler>
```

The **-A** flag signifies the addition of a new virtual service. The protocol is specified with:

- **-t** for TCP
- **-u** for UDP
- **-f** for a firewall mark, an advanced technique used for handling multiple ports as a single service.

This is followed by the VIP address and port number.

The **ipvsadm** command is the primary tool for managing IPVS. Its syntax follows a clear pattern for adding, modifying, listing, and clearing rules.

The **-s** option sets the scheduling algorithm, which is the core logic IPVS uses to distribute incoming connections. Common algorithms include:

- **rr** for round robin.
- **wlc** for weighted least-connections, which is the default.
- **lc** for least-connections.
- **sh** for source hashing, which provides session persistence by always routing the same client IP to the same backend server.

Once the virtual service is established, the next step is to populate it with real servers, the backend nodes that will handle the actual workload. This is done with the following command:

```
ipvsadm -a -t|u|f <vip:port> -r <rip:port> -g|i|m [ -w <weight> ]
```

The **-a** flag appends a real server to an existing virtual service. The **-r** option specifies the real server's IP address and port. The most critical parameter is the forwarding method, selected by:

- **-g** for gateway, or direct routing
- **-i** for IPIP, or tunneling
- **-m** for masquerading, or NAT

Direct routing (**-g**) is the most performant and commonly used method. In this mode, the load balancer forwards incoming packets to the real server with the destination MAC address changed, but the IP header is left intact. The real server must be configured to accept traffic for the VIP on a loopback interface and must not answer ARP requests for that IP on its physical interface. This allows the real server to process the packet and respond directly to the client, bypassing the load balancer on the return path. This preserves the load balancer's outbound bandwidth and minimizes latency.

NAT or **masquerading** (**-m**) is a simpler but less scalable method. The load balancer acts as a full proxy, rewriting both the destination IP address of the incoming packet to that of the real server and the source IP of the return packet from the real server back to its own VIP. This requires the load balancer to process all bidirectional traffic, creating a potential bottleneck. It also typically requires the real servers to have their default gateway pointed to the load balancer's internal IP.

Tunneling (**-i**) encapsulates the forwarded packet within an IPIP tunnel packet. Like direct routing, it allows the real server to respond directly to the client, but it can be useful in cross-network scenarios where direct routing is not feasible due to Layer 2 network constraints.

The **-w** option allows you to assign a static weight to a real server, giving more powerful backend nodes a larger share of the incoming connections when using a weighted scheduler like **wlc**.

A complete example of creating a virtual service for **Hypertext Transfer Protocol** (**HTTP**) and adding two real servers using direct routing would look like the following:

```
# Create a virtual service for TCP port 80 on VIP 192.168.1.100 # using Weighted Least-Connections
ipvsadm -A -t 192.168.1.100:80 -s wlc
```

```
# Add real server 10.0.1.10:80 using Direct Routing (-g) with a weight of 3
ipvsadm -a -t 192.168.1.100:80 -r 10.0.1.10:80 -g -w 3

# Add real server 10.0.1.11:80 using Direct Routing (-g) with a weight of 1
ipvsadm -a -t 192.168.1.100:80 -r 10.0.1.11:80 -g -w 1
```

Refer to the following code, which is the most common command to list the current virtual servers and their real server pools. The **-n** option specifies numeric output (showing IPs instead of trying to resolve hostnames), which is faster and clearer:

```
ipvsadm -Ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
TCP  192.168.1.100:80 wlc
  -> 10.0.1.10:80                 Route   3      0          0
  -> 10.0.1.11:80                 Route   1      0          0
```

To verify the configuration, the command **ipvsadm -Ln** provides a clear, numeric output listing all configured virtual services and their associated real servers, along with current connection counts. It is important to remember that while **ipvsadm** is powerful for direct manipulation and testing, for a production, persistent, and highly available setup, defining these rules within a Keepalived configuration is the standard practice, as it automatically handles the replication of the rules, health checking of the real servers, and failover of the entire IPVS instance between nodes. The following scenario explains the use of IPVS in detail.

Scaling a high-traffic e-commerce API

Company *ShopQine*, an e-commerce platform experiencing rapid growth. Their primary product catalog **application programming interface** (**API**),

running on a single server, is becoming a bottleneck during peak sales and flash events. Users experience slow response times and occasional timeouts, directly impacting revenue.

Aim: Implement a highly scalable, performant, and redundant load balancing solution to distribute traffic across multiple API servers without introducing significant latency or a single point of failure.

Solution: A highly available IPVS load balancing cluster using **direct routing** (**DR**) mode. The solution involves a two-tier architecture:

- **Load balancer tier (IPVS directors)**: A pair of Linux servers configured with IPVS and Keepalived. They share a VIP that is the public endpoint for all API traffic.
- **Application tier (real servers)**: A pool of identical backend servers, each hosting an instance of the ShopQine API application. They are configured to accept traffic for the VIP.

The following figure describes the flow of traffic with **ipvsadm** direct routing and the return path:

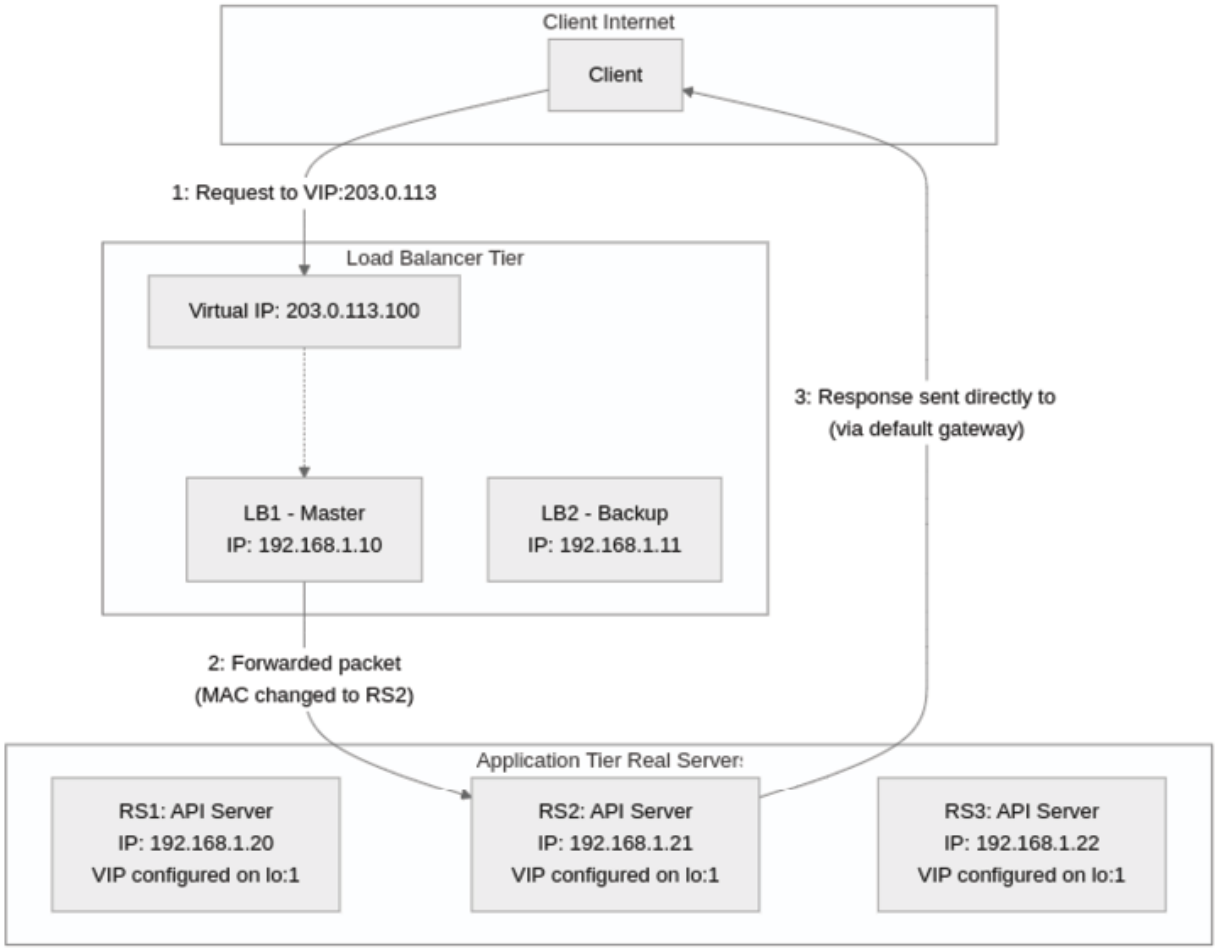


***Figure 6.2**: Traffic flow with IPVS direct routing*

The key to performance is the use of DR. The load balancers only handle incoming requests. Return traffic from the API servers flows directly back to the client, bypassing the load balancers entirely. This prevents the load balancers from becoming a bottleneck for outbound traffic, which is often the larger volume.

The packet flow happens as follows:

- **Incoming request**: A client sends a TCP SYN packet destined for the public VIP (203.0.113.100).
- **Packet forwarding**: The active **load balancer** (**LB)** 1–master receives this packet. Based on its IPVS rules and the configured scheduling algorithm (e.g., weighted least-connections), it selects a healthy real server (e.g., RS2). LB1 then forwards the packet directly to RS2. It

only changes the destination MAC address to that of RS2; the IP headers, including the destination IP, which is still the VIP, remain untouched.

- **Direct response**: RS2 receives the packet, because it has the VIP configured on a loopback interface (lo:1), its kernel recognizes and processes the packet. The application generates a response. This response packet has the VIP as its source IP and the client's IP as its destination IP. RS2 sends this packet directly to its default gateway, which routes it back to the client, completely bypassing the IPVS load balancers. This asymmetric flow is the secret to IPVS DR's high performance.

Configuration details

On the IPVS load balancers (LB1 and LB2), the configuration is managed by Keepalived (**/etc/keepalived/keepalived.conf**). This provides both the VRRP failover for the VIP and the IPVS rule management, as shown in the following code:

```
vrrp_instance VI_1 {
    state MASTER        # 'BACKUP' on LB2
    interface eth0
    virtual_router_id 51
    priority 100        # '90' on LB2
    advert_int 1
    authentication {
        auth_type PASS
        auth_pass securepassword
    }
    virtual_ipaddress {
        203.0.113.100/32 dev eth0  # The virtual IP
    }
}

# IPVS Virtual Service and Real Server Pool
virtual_server 203.0.113.100 80 {
```

```
    delay_loop 15
    lb_algo wlc         # Weighted Least Connections
    lb_kind DR          # DIRECT ROUTING mode
    protocol TCP

    # Health check for each real server
    real_server 192.168.1.20 80 {
        weight 1
        TCP_CHECK {
            connect_port 80
            connect_timeout 3
            retry 2
            delay_before_retry 1
        }
    }
    real_server 192.168.1.21 80 {
        weight 1
        TCP_CHECK {
            connect_port 80
            connect_timeout 3
        }
    }
    real_server 192.168.1.22 80 {
        weight 2        # This server is more powerful, give it more weight
        TCP_CHECK {
            connect_port 80
            connect_timeout 3
        }
    }
}
```

A critical configuration is required on each real server to ensure it can process packets destined for the VIP without conflicting with the load balancers. The following script can be used to configure the real servers:

```
#!/bin/bash
# /etc/keepalived/real_server_config.sh
```

```
# Configure the VIP on the loopback interface
VIP=203.0.113.100
/sbin/ifconfig lo:1 $VIP netmask 255.255.255.255 up

# Prevent the real server from responding to ARP requests for the VIP
echo 1 > /proc/sys/net/ipv4/conf/all/arp_ignore
echo 2 > /proc/sys/net/ipv4/conf/all/arp_announce
```

This script does the following configuration:

- Adds the VIP as a secondary IP on the loopback interface. This interface is not exposed to the network and will not respond to ARP.
- Adjusts kernel ARP parameters so the server will only respond to ARP requests for IPs configured on the interface the request came in on. Since the VIP is on lo:1 and not eth0, the server will never announce itself as the owner of the VIP, leaving that to the current master load balancer.

The preceding script must be deployed and executed on the real servers after boot to persist the configuration. This can be done using a systemd unit.

High availability failover works as follows: if the primary LB1 fails or its health check script fails (e.g., it cannot reach a critical backend service), a VRRP election gets triggered as LB2 (Backup) stops receiving VRRP advertisements from LB1. After a short timeout, LB2 promotes itself to Master. This leads to VIP Takeover, where LB2 claims the VIP by sending a **Gratuitous ARP** (**GARP**) broadcast to the local network. This updates the switch's ARP table, ensuring all future packets for the VIP are sent to LB2's MAC address.

The following figure describes the process of failover of VIP when the primary (master) load balancer fails:

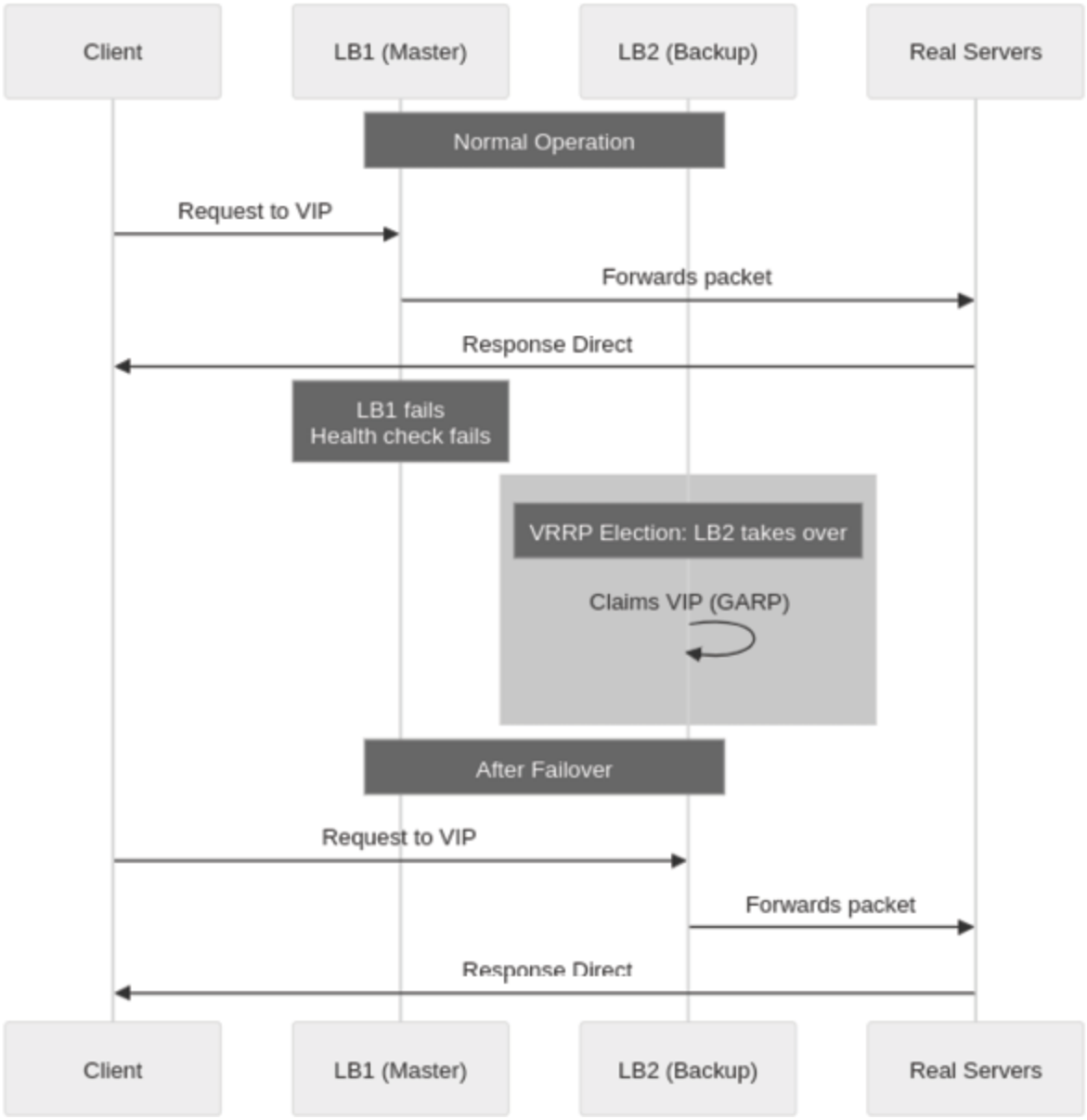


***Figure 6.3**: IPVS failover flow*

Client connections that were in flight during the failover may be dropped and need to retry (stateless failover), but LB2 immediately handles new connections. The pool of real servers remains unchanged and healthy, so service is restored instantly for new requests.

This IPVS DR cluster scenario provides ShopQine with a robust solution with the following benefits:

- Near-line speed due to direct server return.

- Adding more API servers to the pool is trivial.
- The Keepalived+IPVS combination eliminates the load balancer as a single point of failure.
- Built with standard Linux software on commodity hardware or VMs.

High-performance TLS termination cluster

Company *SecureData* needs to offload **Transport Layer Security** (**TLS**)/**Secure Socket Layer** (**SSL**) encryption from its application servers to dedicated hardware for performance and security. Their current solution, a single TLS termination proxy, is a bottleneck and a single point of failure.

Aim: Create a scalable, redundant tier for TLS termination that can handle thousands of new TLS handshakes per second.

Solution: An active-active IPVS DR cluster fronting dedicated TLS termination appliances.

This solution provides the following characteristics:

- **Performance**: TLS handshakes are CPU-intensive. IPVS DR efficiently distributes thousands of new TCP connections (the TLS handshakes) across a pool of dedicated TLS hardware without touching the encrypted data or the much larger response traffic.
- **Session persistence**: The **source hash** (**sh**) scheduler can be used to ensure all packets from a single client session are sent to the same TLS appliance. This is critical for TLS, which requires a stateful session.
- **Efficiency**: The TLS appliances handle the encryption/decryption and then communicate with the backend application servers in plain HTTP, simplifying the internal network.

The following figure describes the setup for the proposed solution with a master and backup load balancer:

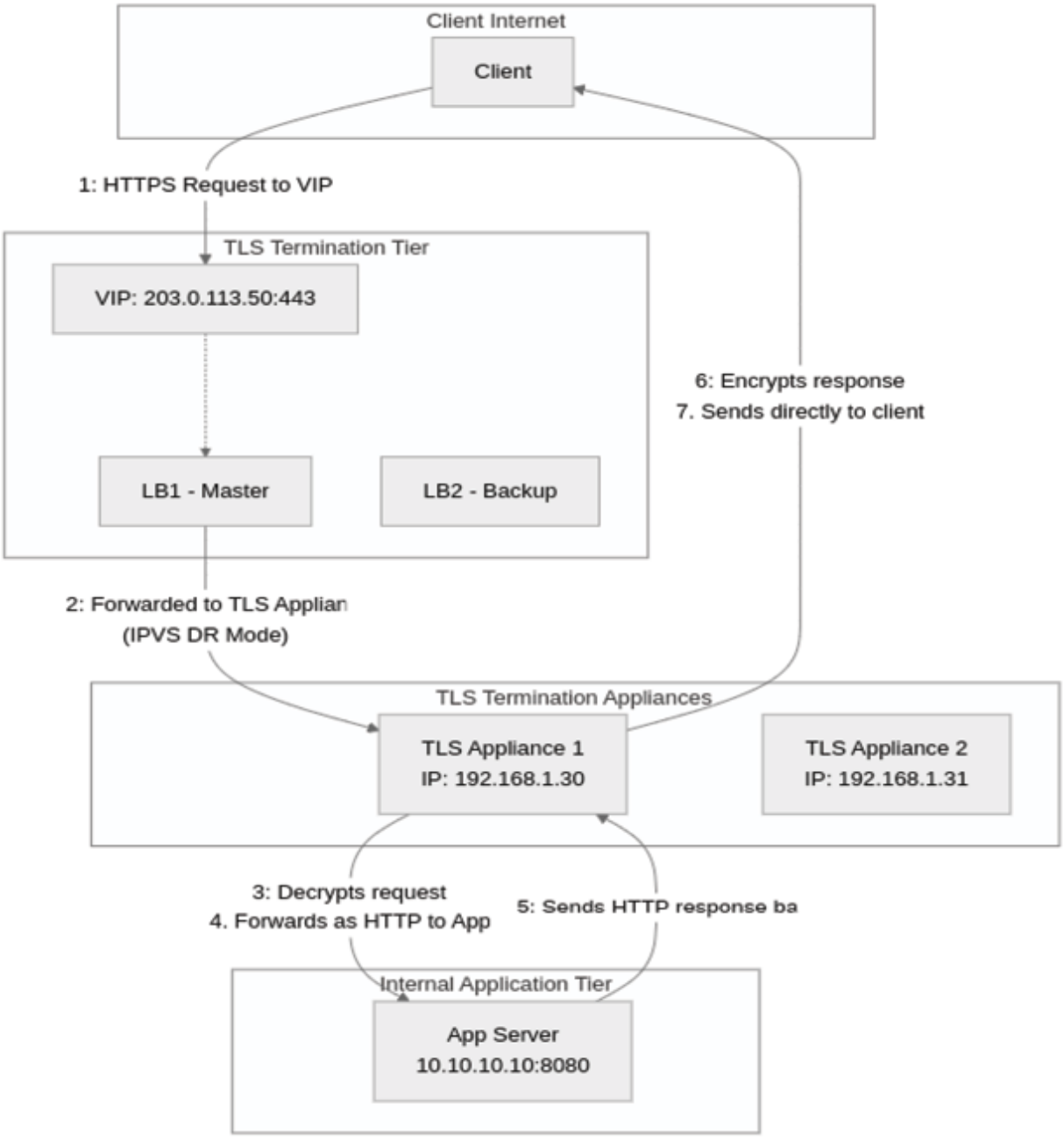


***Figure 6.4**: Layer7 load balancing with session persistence*

The configuration snippet for Keepalived is as follows:

```
virtual_server 203.0.113.50 443 {
    delay_loop 10
    lb_algo sh          # Source Hash for TLS session persistence
    lb_kind DR
```

```
    protocol TCP

    real_server 192.168.1.30 443 {
        weight 1
        TCP_CHECK {
            connect_port 443
            connect_timeout 3
        }
    }
    real_server 192.168.1.31 443 {
        weight 1
        TCP_CHECK {
            connect_port 443
            connect_timeout 3
        }
    }
}
```

Load balancing UDP-based traffic

Company *VoiceFusion*, a VoIP provider, needs a solution that can handle the **User Datagram Protocol** (**UDP**) based SIP traffic and can scale horizontally.

Objective: Distribute UDP SIP traffic across a cluster of SIP proxies without introducing delay or packet loss.

Solution: IPVS in NAT mode for **Session Initiation Protocol** (**SIP**) signaling. The solution has the following characteristics:

- **UDP support**: IPVS has robust support for UDP, making it suitable for SIP, DNS, and other UDP-based protocols.
- **NAT mode required**: For stateful protocols like SIP, the load balancer must see both sides of the conversation (request and response) to manage the connection state correctly. NAT mode ensures all return

traffic flows through the LB, which is necessary for the LB to manage the UDP *connection* state correctly.

- **Health checking**: IPVS can perform UDP health checks, ensuring faulty SIP proxies are taken out of rotation.

The following figure describes the setup for the proposed solution:

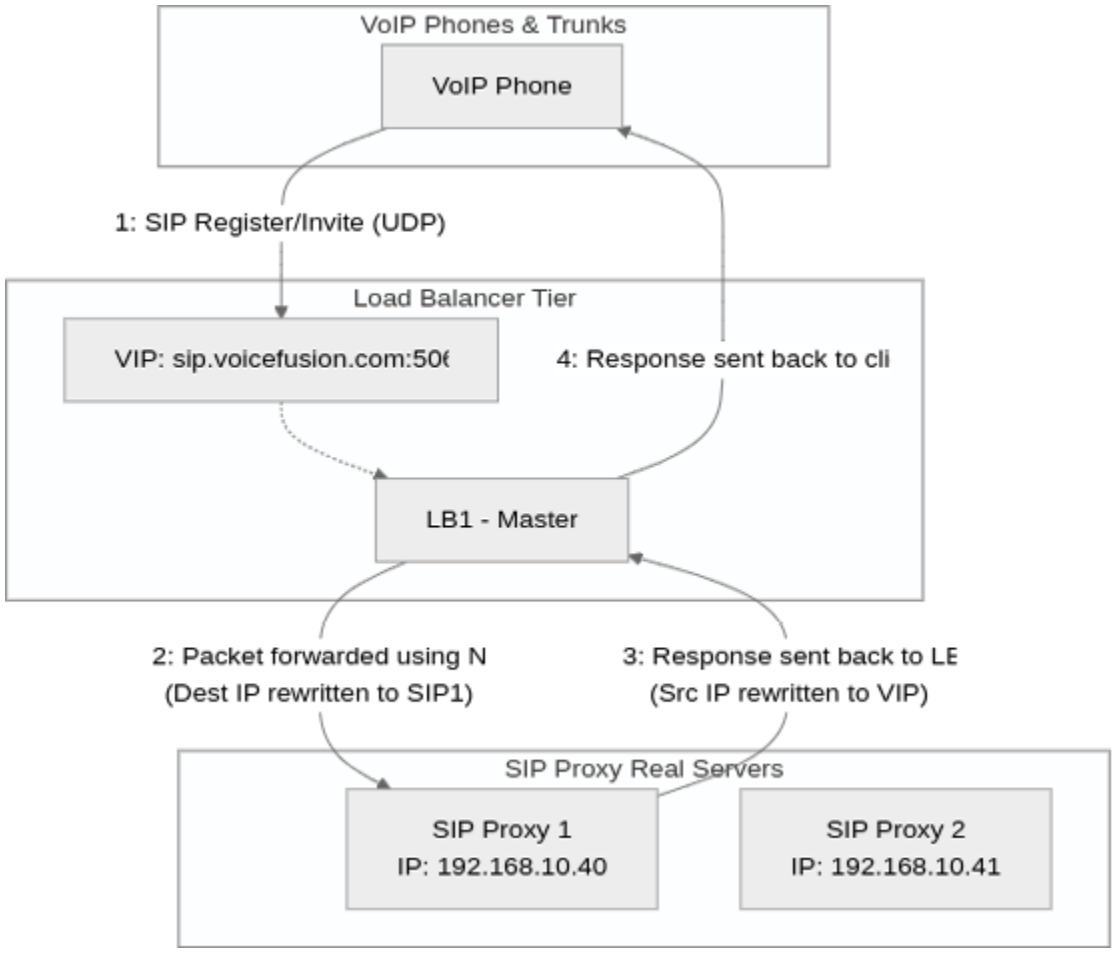


***Figure 6.5**: Layer 7 load balancing UDP traffic*

The following code represents the configuration snippet for Keepalived:

```
virtual_server 203.0.113.75 5060 {
    delay_loop 10
    lb_algo wlc
    lb_kind NAT         # Network Address Translation for UDP
statefulness
```

```
    protocol UDP        # Critical: UDP protocol

    real_server 192.168.10.40 5060 {
        weight 1
        UDP_CHECK {      # UDP specific health check
            retry 2
        }
    }
    real_server 192.168.10.41 5060 {
        weight 1
        UDP_CHECK {
            retry 2
        }
    }
}
```

High availability at Layer 4

High availability at Layer 4 builds upon the foundation of Layer 3 redundancy to address a critical shortcoming: while a VIP can be seamlessly failed over between nodes using VRRP, the state of active network connections—the ongoing conversations between clients and your service—is not automatically preserved. Layer 4, the transport layer of the **Open Systems Interconnect** (**OSI**) model, is where protocols like TCP and UDP operate, managing these connections through state tables that track sequences, acknowledgments, and ports. Standard Layer 3 failover is stateless; the new master node has no knowledge of the connections its predecessor was handling, resulting in a hard break. Clients experience dropped sessions, interrupted downloads, and failed transactions, forcing them to reconnect, often manually. For many modern, resilient applications, this is acceptable behavior, as clients will automatically retry. However, for legacy systems, long-lived transactions, or protocols where connection

setup is exceptionally expensive, this interruption can be detrimental.

The objective of high availability at Layer 4, therefore, is to create a stateful failover mechanism. This ensures that not only the VIP but also the live session state—the socket tables, sequence numbers, and connection tracking information—is continuously synchronized from the active master to a standby backup node. In the event of a failure, the backup node can immediately assume the VIP and all active connections, picking up precisely where the failed master left off. From the client's perspective, there may be a slight pause or delay in packet flow, but the underlying TCP connection remains intact and valid; no **reset** (**RST**) packet is sent, and the application can continue its dialogue without needing to re-authenticate or re-establish the session.

Achieving this stateful Layer 4 failover requires specialized software and a dedicated, high-speed, private network between the cluster nodes for state synchronization. The most common implementation in the Linux ecosystem is achieved through the combination of the IPVS kernel subsystem and the Keepalived daemon. While Keepalived manages the VRRP failover of the VIP, it can also be configured to leverage IPVS's built-in packet forwarding and, crucially, its support for synchronizing state.

The mechanism works through a primary-backup model where the active master node continuously mirrors its IPVS connection state table to the backup node(s). This is typically configured in Keepalived using the sync interface. A dedicated network interface, often a crossover cable or a separate VLAN, is used for this synchronization traffic to ensure low latency and to avoid congesting the public data network. The VRRP sync group is setup as follows:

```
vrrp_sync_group VG_1 {
   group {
      VI_1
   }
   notify_master "/path/to/master_script.sh"
   notify_backup "/path/to/backup_script.sh"
   notify_fault "/path/to/fault_script.sh"
}
```

Within the IPVS configuration section of **keepalived.conf**, a sync daemon is defined to handle this replication:

```
virtual_server 192.168.1.100 80 {
    delay_loop 6
    lb_algo wlc
    lb_kind DR  # Direct Routing mode is common for performance
    protocol TCP

    # Define the real servers behind the load balancer
    real_server 10.0.1.10 80 {
        weight 1
        TCP_CHECK {
            connect_port 80
            connect_timeout 3
        }
    }
    # ... more real_servers

    # Configuration for the Layer 4 state synchronization
    sorry_server 127.0.0.1 80 # Optional fallback if all realservers are down

    # Sync configuration for stateful failover
    alpha_mode  # This node is a backup until a sync is received (waiting to become master)
    omega_mode  # Run a script on quorum loss or failover

    # The shared secret and sync interface for state replication
    sync {
        syncid 1
        device eth1 # Dedicated sync interface
        # A password can be used for authentication between sync nodes
        # password secret
    }
}
```

On the backup node, an identical IPVS configuration is present, but it remains passive, its IPVS table updated in real-time by the master's sync

daemon. The backup node receives every update to the connection state table. When a failover event occurs, triggered by the VRRP protocol, the new master node activates its IPVS table, which is already fully populated with the state of all active connections. It can then immediately begin responding to packets for those existing flows, acknowledging sequences correctly and forwarding traffic to the appropriate backend servers without any interruption in the logical session.

It is important to recognize the trade-offs. State synchronization generates constant network traffic between nodes and consumes memory and CPU resources to maintain the mirrored state tables. The complexity of the configuration is also significantly higher than that of a stateless Layer 3 setup. Therefore, this approach is reserved for scenarios where maintaining uninterrupted sessions is a strict business or technical requirement, justifying the additional overhead and complexity. For the vast majority of web and API services built to be resilient and stateless, a stateless Layer 3 failover provided by a standard Keepalived and HAProxy setup remains the simpler, more robust, and highly effective choice.

Using BGP for high availability

While VRRP is the de facto standard for providing high availability within a single local network (L2 segment), **Border Gateway Protocol** (**BGP**) offers a more robust, scalable, and network-wide approach. BGP is the routing protocol that glues the entire internet together, and it can be used internally to manage the availability and reachability of services, making it a powerful tool for building fault-tolerant systems that span multiple locations or data centers.

The fundamental shift in thinking with BGP-based high availability is that each server (load balancer, application server, etc.) is treated not just as an endpoint, but as a router. Instead of relying on a virtual IP and ARP broadcasts like VRRP, each server announces its availability to host a service by *advertising a route* to the network. The network infrastructure (top-of-rack switches, core routers) then makes intelligent decisions about where to send traffic based on these dynamic route advertisements.

Imagine two data centers, each with an identical load balancer hosting the same service with the IP 203.0.113.100.

The following steps are involved in setting up high availability with BGP route advertisement:

1. **Route advertisement**: Both load balancers run a BGP daemon. They establish a BGP session with their local network router (the *BGP peer*).
   a. The primary load balancer in **Data Center 1** (**DC1**) advertises a route to 203.0.113.100/32 with a standard BGP path attribute, like an **Autonomous System** (**AS**) path.
   b. The backup load balancer in **Data Center 2** (**DC2**) also advertises a route to 203.0.113.100/32.
2. **Path selection**: The network routers now have two paths to the same IP prefix. They use standard BGP best path selection algorithms to choose the preferred route. Key attributes can be manipulated to control this choice; for example, the following attributes can be adjusted to influence the path selection:
   a. **AS path**: The primary site can prepend its own AS number multiple times to make its path look longer and less preferable.
   b. **Local preference**: A higher local preference value is always preferred. The primary site's router can assign a higher local preference to the route from the primary load balancer.
3. **Graceful failover**: The primary load balancer continuously performs health checks on its local service. If the service fails, the BGP daemon on that server withdraws its route advertisement. It tells the network routers that *I am no longer a valid path to 203.0.113.100.*
4. **Automatic convergence**: Upon receiving the route withdrawal, the network routers immediately remove that path from their routing tables. The only remaining path is through the backup load balancer in DC2. Within seconds, all network traffic destined for 203.0.113.100 is automatically rerouted to DC2. This convergence happens globally across any network that accepts these BGP advertisements.

The following example uses BIRD Internet Routing Daemon, an open-source dynamic routing daemon, to demonstrate how BGP can be used to

advertise the route for a VIP from two data centers, DC1 and DC2, with different AS paths.

The following is a simple BGP configuration for DC1 as the primary and advertising the VIP 203.0.113.100/32:

```
router id 192.168.1.10;
protocol kernel {
    persist; # Don't remove routes on BIRD shutdown
    scan time 20;
    import all; # Learn routes from the kernel
    export filter { # Control which routes we advertise to the kernel
        if net = 203.0.113.100/32 then accept; # Advertise our VIP
        reject; # Block everything else
    };
}
protocol device {
    scan time 10;
}
protocol bgp uplink {
    local as 64512;
    neighbor 192.168.1.1 as 65530;
    # Import rules: what we learn from the router
    import all;
    # Export rules: what we advertise to the router
    export filter {
        # Only advertise our specific VIP
        if net = 203.0.113.100/32 then {
            accept;
        }
        # Reject all other routes from being advertised to the peer
        reject;
    };
```

```
    source address 192.168.1.10; # IP to use for the BGP session
}
```

For DC2 acting as the backup, we also advertise the VIP 203.0.113.100/32, but we prepend our AS twice to the AS path to make it longer and less desirable to BGP's best path selection algorithm, as shown in the following code:

```
router id 192.168.1.11;
protocol kernel {
    persist;
    scan time 20;
    import all;
    export filter {
        if net = 203.0.113.100/32 then accept;
        reject;
    };
}
protocol device {
    scan time 10;
}
protocol bgp uplink {
    local as 64512;
    neighbor 192.168.1.1 as 65530;
    import all;
    export filter {
        if net = 203.0.113.100/32 then {
            # --- THE CRITICAL LINE --- #
            # Prepend our AS number two extra times to the AS_PATH
            bgp_path.prepend(64512); # First prepend
            bgp_path.prepend(64512); # Second prepend
            # Now the path is: 64512, 64512, 64512 -> 65530
            accept;
```

```
        }
        reject;
    };
    source address 192.168.1.11;
}
```

The following figure describes the load balancing setup with BGP route advertisement from two different datacenters:

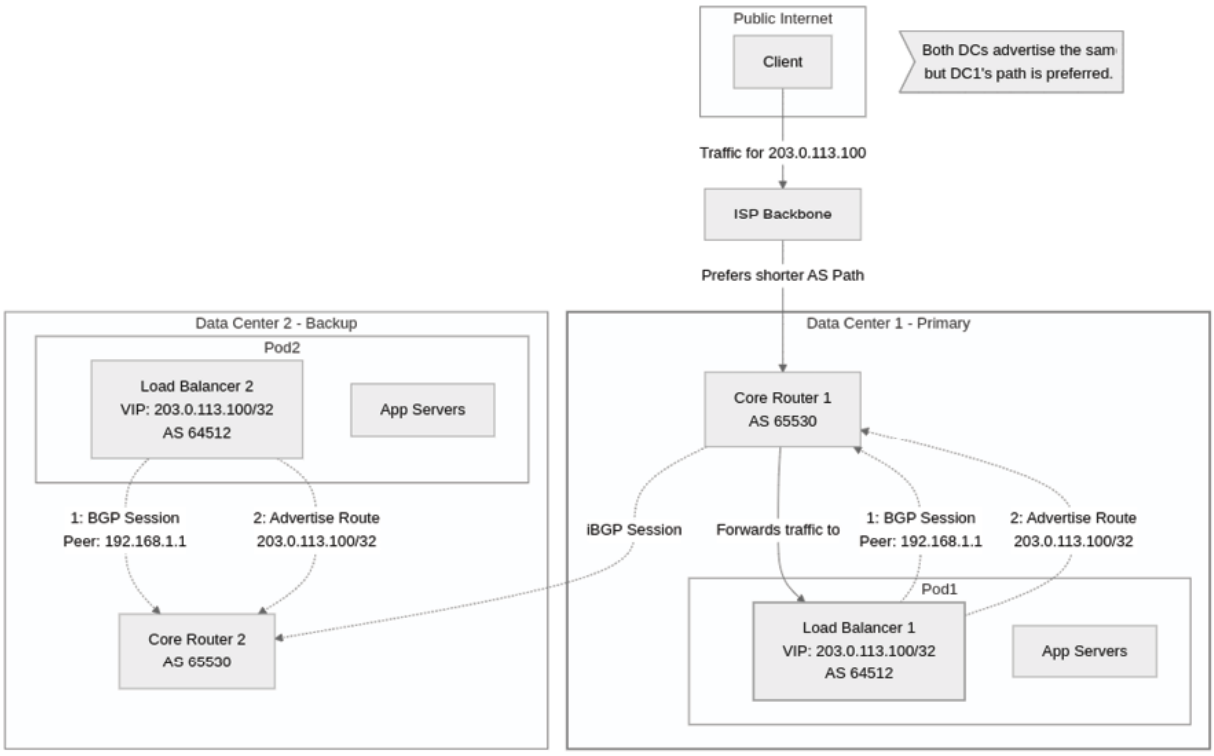


***Figure 6.6**: BGP-based load balancer setup*

The following list outlines the key advantages of BGP for HA:

- **Elimination of Layer 2 constraints**: BGP operates at Layer 3. It does not require servers to be on the same physical network segment, enabling active-active HA across different data centers or availability zones.
- **Incredible scalability**: BGP is designed to handle the Internet's routing table. It can easily scale to manage thousands of IP prefixes and complex network topologies.

- **No single points of failure**: The control plane is distributed. There is no single *master* as in VRRP; the network makes decisions based on the best available path.
- **Seamless integration with network policy**: Traffic engineering becomes possible. You can use BGP communities to tag routes and influence how traffic enters and exits your network on a global scale.
- **No ARP issues**: Since it relies on IP routing and not MAC address flapping, it avoids the potential ARP-related issues that can sometimes complicate VRRP/ARP setups.

A health check script can trigger the failover. The health check script for BIRD is more direct. Instead of adding and removing the network from the BGP process, it simply adds or removes the VIP from the Linux kernel. BIRD's kernel protocol then automatically picks up the change and advertises/withdraws the route. The following script can be used for a health check with BIRD:

```
#!/bin/bash
VIP="203.0.113.100"
HEALTH_CHECK_URL="http://localhost/health"
if curl --silent --max-time 2 --output /dev/null --fail
"$HEALTH_CHECK_URL"; then        # Service is healthy. Add the VIP
if it's not already there.
    if ! ip addr show dev lo | grep -q "$VIP"; then
    echo "Adding VIP $VIP" ip addr add $VIP/32 dev lo
    fi
else
    # Service is unhealthy. Remove the VIP if it is present.
    if ip addr show dev lo | grep -q "$VIP"; then
    echo "Removing VIP $VIP" ip addr del $VIP/32 dev lo
    fi
fi
```

Using BGP for high availability represents a paradigm shift from host-centric tricks like VRRP to a network-centric model. It is inherently more

complex to implement and requires coordination with network engineering. However, for modern, distributed systems spanning multiple failure domains, such as multi-data-center deployments, large-scale cloud environments, or Kubernetes clusters using projects like MetallB, BGP provides the rock-solid, scalable, and standardized foundation necessary for true global load balancing and resilience. It moves the problem of availability from the server room and into the network fabric itself.

Layer 7 load balancers

A Layer 7 load balancer operates at the application layer, the highest level of the OSI model. Unlike its Layer 3/4 counterparts that only see IP addresses and ports, a Layer 7 load balancer has complete visibility into the content and context of client requests. It acts as a reverse proxy, terminating incoming client connections, inspecting the full payload of the application protocol [such as HTTP, **Hypertext Transfer Protocol Secure** (**HTTPS**), or gRPC], and then initiating a new, separate connection to a backend server based on this analysis. This deep packet inspection enables intelligent, fine-grained traffic management decisions that are impossible at lower layers.

The core differentiator of Layer 7 load balancing is its ability to make routing decisions based on application-specific data. For HTTP/S traffic, which is the most common use case, this includes:

- **URL path**: Routing requests for **/api/** to one pool of servers and requests for **/images/** to another.
- **HTTP headers**: Directing traffic based on the host header (virtual hosting), content-type, or custom headers like X-Device-Type to send mobile traffic to different servers.
- **HTTP method**: Potentially routing GET requests to a read-optimized pool and POST requests to a write-optimized pool.
- **Cookies**: Using information in cookies for session persistence (sticky sessions), ensuring a user continues to be directed to the same backend server.

- **Query strings**: Making decisions based on parameters in the URL (e.g., ?user=premium).

This application awareness unlocks powerful features critical for modern, complex architectures. The most significant is SSL/TLS termination. The load balancer decrypts incoming HTTPS traffic, offloading this computationally expensive task from the backend servers. It can then inspect the plaintext HTTP request, make a smart routing decision, and communicate with the backend servers over unencrypted HTTP (or re-encrypt it with a different certificate). This centralizes certificate management and improves backend performance.

Furthermore, Layer 7 load balancers enhance security and resilience. They can act as a **web application firewall** (**WAF**), filtering out malicious requests before they reach the application servers. They can perform rate limiting to protect backends from **denial-of-service** (**DoS**) attacks or API abuse. They can also perform advanced health checks, not just checking if a server's port is open, but by sending a specific request (e.g., GET /health) and verifying the HTTP status code and response body to ensure the application is truly healthy.

However, this intelligence comes at a cost. The process of terminating connections, decrypting SSL, parsing HTTP, and re-establishing connections to the backend is computationally expensive. This introduces higher latency and lower maximum throughput compared to a simpler Layer 4 load balancer. It also becomes a single point of failure for application logic, making its own high availability configuration paramount.

The following figure illustrates the key steps:

1. Connection termination
2. Content inspection
3. Intelligent routing
4. The subsequent new connection to the backend

These highlight the proxy nature of a Layer 7 load balancer:

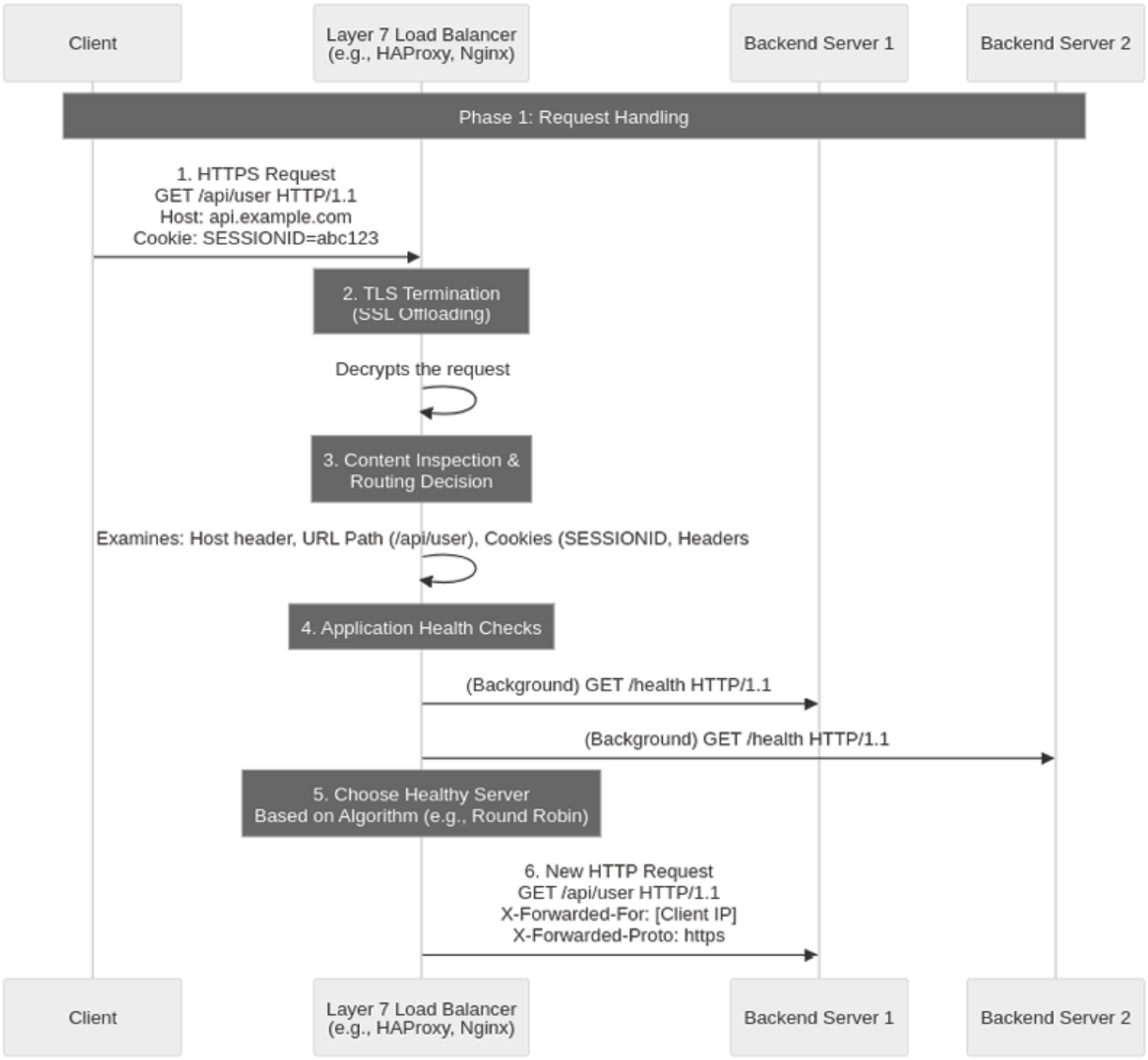


***Figure 6.7**: Layer7 load balancer connection handling*

In practice, Layer 7 load balancing is typically implemented by sophisticated software like HAProxy and cloud-based **application load balancers** (**ALBs**) from providers like **Amazon Web Services** (**AWS**) and Google Cloud. These tools understand the *language* of the traffic they manage to ensure it is delivered efficiently, securely, and based on business logic.

Installation and configuration of HAProxy

HAProxy is a software-based Layer 7 load balancer, renowned for its performance, reliability, and rich feature set. It operates as a reverse proxy, terminating client connections, inspecting application-layer protocols (primarily HTTP/HTTPS), and making intelligent routing decisions based on content before initiating new connections to backend servers.

HAProxy is included in the following default repositories of all major Linux distributions, making installation straightforward:

```
$ sudo apt install haproxy # For Ubuntu
$ sudo dnf install haproxy # For Rocky Linux
$ sudo yum install haproxy # For older versions of Redhat derivatives
```

The following command should output the installed version, confirming the software is ready:

```
$ haproxy –v
```

HAProxy's configuration file is typically located at **/etc/haproxy/haproxy.cfg**. It is divided into several structured sections: global, defaults, frontend, backend, and listen. Understanding these blocks is key to mastering HAProxy.

Global and defaults sections

The following global section contains process-wide settings, and the defaults section provides default parameters for all subsequent sections:

```
global
    log /dev/log local0   # Log to a local socket
    maxconn 4000          # Maximum connections per process
    user haproxy          # Drop privileges to these users
    group haproxy
    daemon                # Run as a background daemon

defaults
    mode http             # Default mode for all sections is HTTP (Layer 7)
    option httplog        # Enable detailed HTTP logging
    option dontlognull    # Don't log empty connections
    timeout connect 5s    # Max time to connect to a backend server
```

```
    timeout client 30s    # Max time client can be idle
    timeout server 30s    # Max time backend server can be idle
    option forwardfor     # Add X-Forwarded-For header with client IP
    retries 3             # Number of retries on a server failure
```

In the preceding configuration, the following options should be noted:

- **mode http**: This is what enables Layer 7 functionality.
- **option forwardfor**: It is crucial for backends to see the original client IP, not the load balancer's IP.
- **timeout directives**: It is essential for stabilizing connections and cleaning up idle sessions.

Frontend section

The following frontend block defines the IP and port that clients connect to. It is the public-facing listener:

```
frontend main_http_frontend
    bind :80              # Listen on port 80 on all interfaces
    bind :443 ssl crt /etc/ssl/example.com.pem # For HTTPS

    # Define an ACL (Access Control List) to match a path
    acl is_static path_beg -i /static /images /javascript /css
    acl is_api path_beg -i /api
    acl host_blog hdr(host) -i blog.example.com

    # Use ACLs to route traffic
    use_backend static_servers if is_static
    use_backend api_servers if is_api
    use_backend blog_servers if host_blog

    # Default backend if no ACLs match
    default_backend web_servers
```

This frontend demonstrates key layer 7 features:

- **SSL termination**: The bind: **443 ssl crt <file>** line tells HAProxy to decrypt HTTPS traffic using the provided certificate bundle.

- **ACLs**: These are the rules for inspecting HTTP content.
- **path_beg**: Checks the beginning of the URL path.
- **hdr(host)**: Inspects the Host header, enabling virtual hosting.
- **use_backend:** Directs traffic to a specific group of servers based on the ACL.

Backend section

The following backend block defines a group of servers (*real servers*) that will receive traffic from the frontend:

```
backend web_servers
    balance roundrobin      # Load balancing algorithm
    option httpchk GET /health HTTP/1.1\r\nHost:example.com # Adv health check
    server web01 192.168.1.20:80 check weight 1
    server web02 192.168.1.21:80 check weight 1
    server web03 192.168.1.22:80 check backup # Marked as a backup server

backend api_servers
    balance leastconn       # Better for stateful or long-lived API connections
    cookie SERVERID insert indirect nocache # Session persistence with cookies
    server api01 192.168.1.30:8080 check cookie s1
    server api02 192.168.1.31:8080 check cookie s2

backend static_servers
    balance source          # Persist based on client IP
    server static01 192.168.1.40:80 check
```

In the preceding configuration, the following option should be noted:

- **balance**: It defines the load balancing algorithm. The following tables

describe the various load balancing algorithms available in HAProxy:

Algorithm	Keyword	Primary use case	Persistence based on
Round robin	roundrobin	General-purpose, stateless	N/A
Static round robin	static-rr	Servers with fixed, unequal capacity	N/A
Least connections	leastconn	Long-lived/stateful connections	Current connection count
Source IP hash	source	IP-based sticky sessions	Client source IP
URI hash	uri	Caching proxy efficiency	Requested URI
URL parameter	url_param	Session ID in URL	URL parameter value
Header hash	hdr	Advanced content routing	HTTP header value
Random	random	Large server farms	Random selection
RDP cookie	rdp-cookie	Microsoft Remote Desktop	RDP cookie

***Table 6.1** : HAProxy load balancing algorithms*

- **option httpchk**: This is a Layer 7 health check. HAProxy will send an HTTP GET /health request to the server. If it returns a 2xx or 3xx response, the server is considered healthy. This is far superior to a simple TCP check.
- **cookie**: It enables session persistence (sticky sessions) by injecting a cookie in the client's response, ensuring they return to the same server.
- **server**: The definition of a backend server. The check option enables health checks. weight allows a weighted distribution. Backup marks a server that will only be used if all non-backup servers are down.

Stats page

HAProxy provides a built-in web-based dashboard for monitoring. The following configuration can be used to set up the admin **user interface** (**UI**):

```
frontend stats_frontend
    bind *:8404
```

```
    stats enable
    stats uri /admin?stats
    stats refresh 10s
    stats auth admin:SecurePassword123 # Authentication for stats page
    # Restrict access to a specific IP range for security
    acl allowed_ips src 192.168.1.0/24
    http-request deny if !allowed_ips
```

Before applying the configuration, you can check the syntax of the configuration file. This can be done using the following command:

```
$ sudo haproxy -f /etc/haproxy/haproxy.cfg -c
```

Before restarting, always validate your configuration to avoid breaking the service.

On systemd systems like Ubuntu and Rocky Linux, use the following command to reload without dropping connections (zero-downtime reload):

```
$ sudo systemctl reload haproxy
```

Now you can navigate to **http://your-haproxy-ip:8404/admin?stats** in a web browser and log in with the credentials you set. This dashboard provides real-time insights into frontend and backend traffic, server status, session rates, and more, making it an invaluable tool for management and troubleshooting.

Conclusion

In this chapter, we discussed the two primary philosophies for distributing traffic: the raw, high-speed packet forwarding of Layer 4, exemplified by IPVS, and the intelligent, application-aware routing of Layer 7, mastered by HAProxy. IPVS operates deep within the kernel, offering unparalleled efficiency for scenarios demanding massive throughput and minimal latency, making it ideal for handling bulk traffic or UDP-based protocols. HAProxy, acting as a sophisticated reverse proxy, provides the context-aware intelligence necessary for modern web applications, enabling advanced routing, SSL termination, and deep health checks based on the

actual content of HTTP requests.

Crucially, we have seen that these data plane solutions are only half the story. The control plane for high availability is deftly handled by Keepalived and the VRRP protocol, which ensures that the critical virtual IP, the single point of entry for all traffic, is never a single point of failure. Keepalived's ability to integrate service health checks with VRRP failover creates a resilient system that can proactively reroute traffic away from failing components, ensuring continuous service availability.

In the next chapter, we will learn about Linux network namespaces and how they can provide isolated network stacks and their application in the real-world.

CHAPTER 7

Network Namespaces and Virtual Networking

Introduction

In the architecture of modern computing, the network is the central nervous system. Applications are no longer monolithic entities running on a single server but distributed systems composed of dozens of interconnected microservices, databases, and caches. This paradigm demands a radical rethinking of network infrastructure, one that must be simultaneously agile, secure, and efficient. *How do we host hundreds of these independent, network-reliant services on a single host without crippling port conflicts, security vulnerabilities, and administrative chaos?* The answer lies not in physical hardware, but in virtualization at the operating system level.

This chapter delves into the powerful kernel primitives that make this software-defined networking revolution possible. We begin by exploring the core concept of network namespaces, the fundamental building block for network isolation in Linux. You will learn how a namespace creates a logically distinct copy of the network stack, complete with its own interfaces, routes, and firewall rules, used for providing an isolated environment for processes.

Moving beyond theory, we will master the essential iproute2 toolkit to

create, manage, and inspect these namespaces firsthand. With this foundation in place, we will progress to advanced virtual device types like **Media Access Control address based Virtual Area Network** (**MACVLAN**) and **Internet Protocol address based Virtual Area Network** (**IPVLAN**), which elegantly bypass the host network stack to connect namespaces directly to physical networks, and **tunnel or tap interfaces** (**TUN/TAP**), which enables user-space networking for **virtual private networks** (**VPNs**) and hypervisors.

Finally, we will bridge the gap between these low-level mechanisms and the technologies that power the industry. You will discover how Docker containers and Kubernetes pods are, at their heart, applied examples of network namespaces. We will deconstruct the **Container Network Interface** (**CNI**) to understand how orchestration platforms automate the plumbing we learn to build manually.

By the end of this chapter, you will possess a deep, practical understanding of the virtual networking layer that underpins containerization and cloud computing. You will be equipped not only to configure isolated networks but also to troubleshoot complex networking issues and truly understand the infrastructure you work with every day.

Structure

This chapter covers the following topics:

- Need for network virtualization
- Network namespace
- IPRoute2 toolkit
- Using MACVLAN for bypassing the host stack
- Scaling with shared MACs using IPVLAN
- Integrating VLANs with namespaces
- User-space networking with TUN and TAP
- Docker container networking deep dive
- Kubernetes networking fundamentals
- Network debugging in an orchestrated world

- Security hardening and considerations

Objectives

This chapter aims to provide a foundational and applied knowledge of Linux network namespaces, the core technology enabling network isolation. You will learn to construct and manage isolated virtual networks using command-line tools and advanced virtual interfaces. Furthermore, we will deconstruct how these underlying mechanisms power modern container orchestration systems. The goal is to empower you to not only configure these environments but also to effectively diagnose connectivity issues and fully understand the networking architecture of containerized applications.

Need for network virtualization

In the traditional computing model, a server operating system provides a single, unified network environment. This global network stack, comprising the physical interfaces, **Internet Protocol** (**IP**) addresses, routing tables, **Address Resolution Protocol** (**ARP**) caches, and firewall rules, is shared by every process and application running on the host. While simple to conceptualize, this architecture becomes a significant constraint in modern environments where density, security, and agility are paramount. The fundamental problem is a lack of isolation, as any change to the network configuration or any problem within it has system-wide repercussions.

This shared model introduces several critical operational challenges that hinder development, deployment, and security. The first and most common is port conflicts. Two independent applications, perhaps one a production web service and another a development build, cannot bind to the same privileged port (e.g., 80 or 443) simultaneously. This forces complex workarounds, such as running services on non-standard ports and implementing user-level proxies, which add unnecessary complexity and potential points of failure. Secondly, this model presents grave security risks. A compromised service on a global stack offers the attacker a

foothold on the host's entire network plane. From there, they can scan other local services, attempt lateral movement, or manipulate the host’s firewall (iptables) and routing rules to exfiltrate data or redirect traffic. The principle of least privilege is violated by design, as every process has implicit trust to at least observe the host's network traffic.

Finally, the monolithic stack offers poor agility and testing capabilities. It is complicated and difficult to create self-contained, reproducible network environments for development or testing on a single machine. For instance, simulating a multi-tier application with separate web, application, and database tiers, each with its own unique network policies and routing requirements, is nearly impossible without resorting to multiple physical or **virtual machines** (**VMs**). This slows down development cycles and makes it hard to accurately test network configurations before deploying to production. These limitations collectively created a strong impetus for a new model of network isolation, one that was more lightweight and integrated than full hardware virtualization, but more robust than simple userspace tricks.

The initial industry solution to isolation problems was hardware virtualization, enabled by hypervisors like VMware ESXi, KVM, and Xen. This approach solves the problem by running entire independent operating systems as guests, each with its own completely isolated virtual network adapter connected to a virtual switch. While effective, this method carries substantial overhead: each guest **operating system** (**OS**) consumes significant **central processing unit** (**CPU**), memory, and storage resources solely to run its own kernel and system processes. The boot time and operational weight of full VMs can be prohibitive when needing to spin up hundreds or thousands of isolated environments.

Network namespaces represent a paradigm shift to a far more lightweight and efficient model: OS-level virtualization. A network namespace is not a virtual machine but a feature of the Linux kernel that provides a logically isolated instance of the global network stack. It is a form of containerization specifically for networking. Processes are attached to a specific namespace and can only see and use the network resources, like interfaces, routes, ports, and rules that exist within that namespace. This isolation is

implemented directly within the kernel, making it not only a userspace emulation but a native, high-performance capability. The overhead of creating a new network namespace is negligible; it involves creating a new set of kernel data structures, allowing for thousands of isolated network environments to run simultaneously on a single host with minimal resource impact. This efficiency is the key enabler for the container revolution.

Use cases for the adoption of network virtualization

The power of network namespaces is harnessed across the entire technology landscape, forming the invisible backbone of modern infrastructure:

- **Container runtime networking (Docker, containerd, podman)**: This is the most ubiquitous application of network namespaces. By default, each container is launched into its own network namespace. This is why a container can bind to port 80 internally without conflicting with another container or the host; each has its own isolated *localhost* and network interface. This provides the fundamental isolation that allows multiple containerized applications to coexist peacefully on a single host.
- **Enhanced security and cloud multi-tenancy**: Network namespaces enable rigorous micro-segmentation. Sensitive services like databases can be placed in namespaces with strict inbound and outbound firewall policies, ensuring they can only communicate with explicitly authorized application namespaces. This dramatically reduces the attack surface and contains potential breaches. Cloud providers leverage this to enforce isolation between different customers (*tenants*) on shared physical hardware, ensuring one customer cannot snoop on another's network traffic.
- **Development, testing, and CI/CD**: Developers and system administrators can use namespaces to create complex, multi-node network topologies on a single laptop or build server. They can simulate entire networks, complete with routers, firewalls, and servers each running in a dedicated namespace. This allows for realistic testing of network configurations, failure scenarios, and distributed applications without requiring a lab full of physical hardware,

significantly accelerating development cycles and improving reliability.

- **Network function virtualization (NFV)**: Networking services that were once physical appliances, such as firewalls, load balancers, routers, and VPN terminators, can now be run as software instances within isolated namespaces. This makes network infrastructure as programmable, scalable, and agile as the applications it supports.

  This shift from hardware-defined to software-defined networking within the OS is the critical foundation that enables the scalability, security, and rapid pace of modern software development and deployment.

Network namespace

To truly understand network namespaces, we must look beyond the commands and examine their implementation within the Linux kernel. A network namespace is not a VM or emulated environment; it is a native kernel feature that creates a logically separate instance of key networking data structures. When created, a new namespace receives its own complete and independent copy of the following components:

- **Network devices**: Both physical and virtual interfaces can be moved into a namespace, making them invisible to other namespaces.
- **Routing tables**: Each namespace maintains its own routing table (ip route), determining how packets are forwarded.
- **ARP cache**: The Address Resolution Protocol cache, which maps IP addresses to MAC addresses, is namespace specific.
- **Netfilter/iptables rules**: Firewall and **Network Address Translation** (**NAT**) rules are completely isolated. A rule in one namespace does not affect traffic in another.
- **Protocol stacks**: Separate instances of the IPv4 and IPv6 protocol stacks, including **Transmission Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**) ports. Port 80 in one namespace is distinct from port 80 in another.

This isolation is implemented through kernel data structures that are tagged

with a namespace identifier. When a process runs within a namespace, the kernel ensures all network-related system calls operate only on resources belonging to that namespace. This implementation makes namespaces incredibly lightweight. Creating a new namespace merely allocates these data structures without duplicating actual code or consuming significant memory. The efficiency of this design is what enables modern systems to run thousands of isolated network environments simultaneously.

Lifecycle of a network namespace

Understanding the lifecycle of a network namespace is crucial for effective management. Unlike processes, namespaces can persist beyond the execution of a command, which is both a powerful feature and a potential source of *namespace litter* if not managed carefully.

Creation is straightforward using the **ip netns add <namespace-name>** command. This command creates a new namespace and, critically, establishes a persistent reference to it in the **/var/run/netns/** directory. This persistence is what allows administrators to attach to namespaces later, even if no processes are currently running inside them. Without this filesystem anchor, a namespace would exist only as long as a process remained within it.

A namespace can be deleted using **ip netns delete <namespace-name>**. This command removes the persistent reference from **/var/run/netns/**.

Note: The namespace itself will only be destroyed when no more processes are using it. This distinction is important; deleting the reference makes the namespace harder to manage, but the kernel will clean it up automatically once all processes exit. This behaviour mirrors how files are deleted: the directory entry is removed immediately, but the data blocks are not freed until the last file descriptor is closed.

Interaction with other namespace types

While powerful on its own, a network namespace rarely operates in isolation. It is part of Linux's comprehensive namespace ecosystem, where different types of namespaces work together to provide complete process isolation. The most essential synergy is with **process identifier** (**PID**) namespaces.

A process's network namespace membership is determined by its perspective on the network stack, while its PID namespace determines what processes it can see. When you combine these namespaces, you create a truly isolated environment where a process can only see and communicate with network resources and processes within its own set of namespaces. This combination is exactly what container runtimes like Docker use. Each container gets its own PID namespace (seeing only its own processes) and its own network namespace (seeing only its own network interfaces).

This interaction becomes significant when debugging. Tools like `ps aux` run from the host will show processes from all namespaces, but their network perspective is that of the host's default namespace. To properly diagnose network issues in a container, you must use tools that operate within the container's network context, either by entering the container's namespace or using namespace-aware utilities like **nsenter**.

User namespaces add another layer of complexity and security. They allow for privilege mapping, where a process can have root privileges within its user namespace but limited privileges in the host namespace. When combined with network namespaces, this allows unprivileged users to create and configure their own network environments without requiring full root access on the host system.

Inspection and debugging network namespaces

The Linux **/proc** filesystem provides a window into the kernel's view of namespaces, offering powerful inspection capabilities for debugging and management. Every process on the system has a **/proc/<pid>/ns/** directory containing symbolic links that represent the namespaces it belongs to. The network namespace is represented by **/proc/<pid>/ns/net**.

These are not ordinary symlinks. Their inode numbers act as unique identifiers for each namespace instance. By comparing the inode numbers of the net symlinks for two different processes, you can definitively determine whether they share the same network namespace. This is invaluable for verifying isolation in complex environments.

The **nsenter** (namespace enter) command is the Swiss Army knife for

namespace debugging. It allows you to run arbitrary commands in the namespaces of another process. The syntax **nsenter -t <pid> -n <command>** executes the specified command in the network namespace of the target process. This is how you can run troubleshooting tools like **tcpdump**, **ip addr**, or **ss** from the perspective of a container without needing to access the container itself. For example, to see the network interfaces visible to a Docker container, you can find its PID on the host and execute **nsenter -t <pid> -n ip link show**.

This approach is far more potent than container-specific tools because it gives you direct access to the underlying kernel primitives. You can inspect iptables rules that are invisible from the host, test network connectivity from the container's perspective, and diagnose routing issues without modifying the container image or entering it through its shell. This deep inspection capability is what separates administrators who merely use containers from those who truly understand them.

IPRoute2 toolkit

For decades, Linux network configuration was managed through a collection of utilities known as **net-tools**, which included familiar commands like **ifconfig**, **route**, **arp**, and **netstat**. While these tools were once the standard, they suffered from significant limitations that became apparent as Linux networking grew more complex. The tools operated in isolation, each with its own syntax and output format, making scripted automation cumbersome and error prone. More critically, they could not interact with many of the advanced networking features added to the Linux kernel, such as network namespaces, policy routing, and traffic control.

The IPRoute2 suite emerged as the modern replacement, designed from the ground up to address these shortcomings. Unlike the disparate net-tools, IPRoute2 provides a unified, consistent interface to the kernel's networking capabilities through a single powerful command: **ip**. This command uses a structured syntax with object-oriented design principles, making it both more logical and more powerful. The **ip** command allows administrators to manipulate network interfaces, IP addresses, routing tables, ARP caches,

and much more using a predictable syntax. Furthermore, IPRoute2 is namespace-aware, meaning it can operate on objects within specific network namespaces, a capability absent in net-tools. This makes it an indispensable tool for managing the virtualized networks that form the core of this chapter. While the old commands may still be present on some systems for backward compatibility, any serious network administration or automation must now be performed with IPRoute2.

Core syntax and structure of ip command

The power of the **ip** command lies in its logical and consistent structure. The general syntax follows a pattern of **ip [OPTIONS] OBJECT COMMAND [ARGUMENTS]**. The OBJECT is the network entity you wish to manage, such as a link (network device), **addr** (IP address), a route (routing table), or a **netns** (network namespace). The **COMMAND** specifies the action to perform on that object, such as add, delete, show, or set.

For example, to view all network interfaces, you would use **ip link show**. To add a new IP address to an interface, the command is **ip addr add 192.168.1.10/24 dev eth0**. This object-command pattern is applied consistently across all objects, drastically reducing the learning curve once the basic structure is understood. The show command is critical as it is the primary tool for inspection and verification. Adding the **-c** (color) option greatly enhances readability by colorizing the output, making it easier to distinguish different fields.

Another critical advantage of IPRoute2 is its detailed and accurate output. The -d (details) option provides more verbose information, while the **-s** (statistics) option provides packet and byte counters for interfaces. This level of detail is essential for troubleshooting. The ability to pipe this structured output to tools like **grep**, **awk**, and **jq** makes it the perfect foundation for automated scripting and monitoring solutions. Mastering this syntax is not just about learning new commands; it is about adopting a more powerful and precise methodology for interacting with the Linux networking stack.

Creating a network namespace and verifying isolation

The theoretical concepts of namespaces become concrete through practical application. This lab guides you through using the IPRoute2 toolkit to create isolated network environments and verify their operation. The details are as follows:

- **Creating namespaces**: We begin by creating two namespaces, **red** and **blue**, to represent two isolated environments. The command is simple:

  ```
  $ sudo ip netns add red
  $ sudo ip netns add blue
  ```

  This creates two new network namespaces and establishes persistent references to them in **/var/run/netns/**.

- **Listing and inspecting namespaces**: To verify the namespaces were created, list them:

  ```
  $ ip netns list
  ```

  - This should output **red** and **blue**. Now, let us inspect the network stack inside the **red** namespace. We use the **exec** command to run shell commands inside the namespace:

    ```
    $ ip netns exec red ip link show
    ```

  - You will see only the loopback interface (**lo**), which is automatically created in every new namespace. Crucially, it is in the **DOWN** state. Compare this to the output of **ip link show** on the host, which shows all physical and virtual interfaces. This demonstrates the isolation.

- **Bringing the loopback interface up**: Even the loopback interface must be explicitly activated within its namespace:

  ```
  $ ip netns exec red ip link set lo up
  $ ip netns exec red ip addr show
  ```

- **Verifying complete isolation**: Finally, we prove that the two namespaces cannot communicate. Attempt to **ping** the loopback address of the **blue** namespace from inside the **red** namespace. This will fail because there is no network path between them:

```
$ ip netns exec red ping -c 2 127.0.0.1 # This will succeed.
$ ip netns exec red ping -c 2 192.168.1.1 # This will fail
```

This exercise confirms that you have successfully created two independent, isolated network stacks using the IPRoute2 toolkit.

Using MACVLAN for bypassing the host stack

The primary methods of connecting network namespaces, like the **veth** pair and **bridge** combination, involve the host kernel acting as a software-based switch or router. While flexible, this processing introduces overhead as packets traverse the host's network stack. MACVLAN offers a radically different and performant approach: it allows you to attach a virtual network interface in a namespace directly to a physical host interface, effectively bypassing the host's network stack for incoming and outgoing traffic.

Think of a MACVLAN interface as a virtual network card that is directly wired into the physical network adapter. Each MACVLAN virtual interface is assigned its own unique **media access control** (**MAC**) address. From the perspective of the external network switch, each MACVLAN interface attached to a physical host port appears as a distinct physical device connected to that switch. This architecture provides several key advantages. First, it offers near-native network performance, as packets are forwarded between the physical interface and the MACVLAN interfaces with minimal processing by the host kernel. Second, it simplifies network design by making namespaces first-class citizens on the physical **Local Area Network** (**LAN**); they obtain IP addresses from the network's **Dynamic Host Configuration Protocol** (**DHCP**) server and communicate directly with other devices without any NAT or explicit routing on the host.

The physical interface to which the MACVLAN interfaces attach is called the **parent interface**. It can be a physical Ethernet adapter (e.g., **eth0**), a bonded interface, or even a **Virtual Local Area Network** (**VLAN**) sub-interface (e.g., **eth0.100**). The parent interface must be placed in promiscuous mode to allow it to receive frames destined for the MAC addresses of its child MACVLAN interfaces. The kernel driver for the

parent interface must support this mode for MACVLAN to function correctly.

The following figure shows a typical MACVLAN setup with the MACVLAN interfaces using **eth0** on the host as the parent interface to connect the network namespaces to the external network:

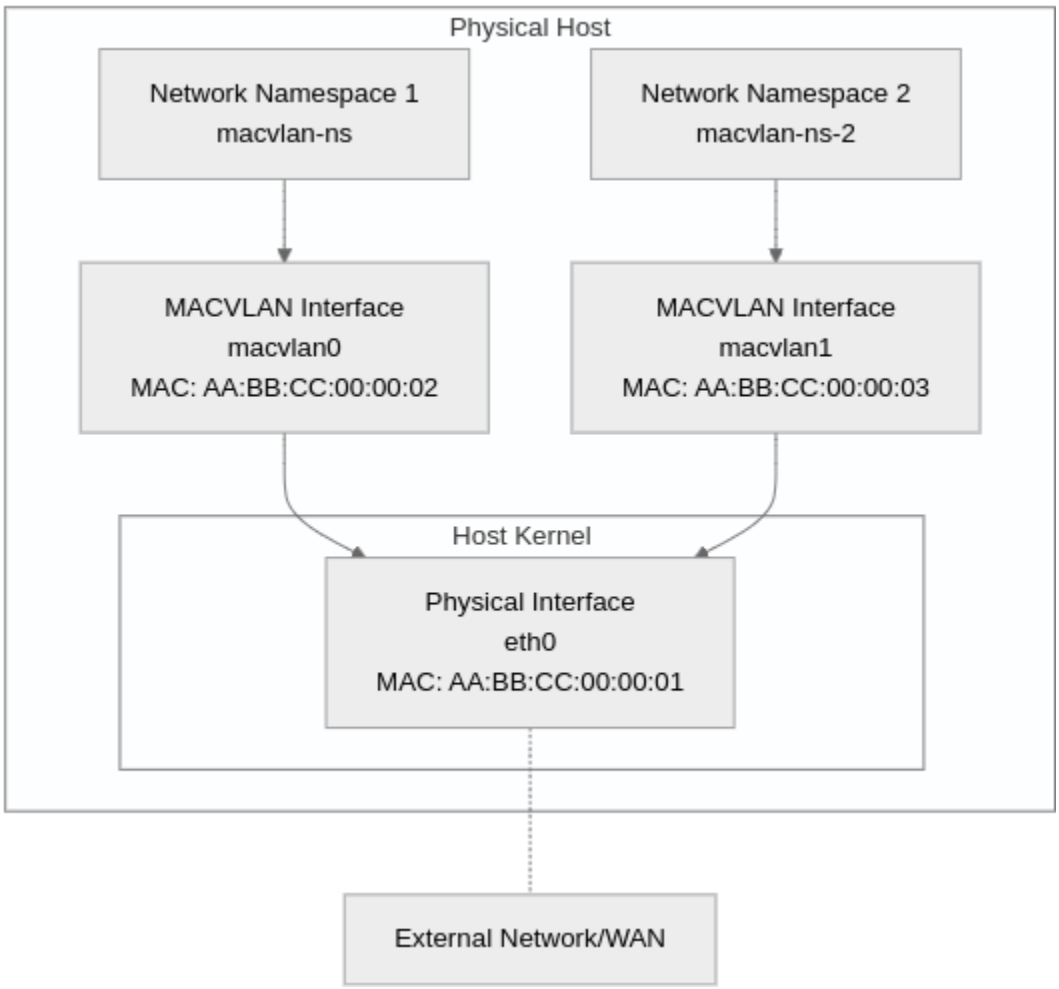


***Figure 7.1**: MACVLAN setup*

MACVLAN modes

The behaviour of a MACVLAN setup is governed by its mode, which determines how traffic between MACVLAN interfaces on the same parent interface is handled. The choice of mode is critical and depends on your network environment and requirements.

- **Bridge mode**: The most common and often default mode is bridge

mode. In this mode, the Linux host kernel acts as a simple switch between all MACVLAN interfaces on the same parent. If two MACVLAN interfaces on the same host need to communicate, the kernel bridges the packets directly between them; the frames are not sent out to the external physical switch only to be reflected back. This is highly efficient for intra-host communication. However, it can create a potential problem: the external switch never sees the MAC addresses of both communicating endpoints on its port, which can confuse its MAC address table and break communication if the switch employs strict port security features.

- **Virtual Ethernet Port Aggregator** (**VEPA**): VEPA mode, defined by the **Institute of Electrical and Electronics Engineers** (**IEEE**) 802.1Qbg standard, is designed to work with smart switches that support VEPA or hairpin mode. In this mode, all traffic, even between MACVLAN interfaces on the same host, is forced out onto the physical wire to the external switch. The switch is responsible for reflecting the frames back to the host if the destination MAC address is on the same port. This mode provides better visibility and control for the network administrator, as all traffic passes through the physical switch, but it requires specific hardware support and doubles the traffic on the physical link for intra-host communication.
- **Private mode**: The private mode is a more restrictive variant of VEPA. It blocks any communication between MACVLAN interfaces on the same host, even if the switch supports hairpin mode. Communication is only possible via the external network. This mode is used for enforcing strict isolation between namespaces.
- **Passthrough mode**: The passthrough mode is a specialized mode that allows a single MACVLAN interface to be attached to the parent, and it grants the virtual interface direct access to the physical interface's features. It is rarely used for general namespace networking and is more common for specialized virtualization passthrough scenarios.

Configuring a MACVLAN interface in bridge mode

This lab guides you through creating a MACVLAN interface inside a network namespace and attaching it to a physical host interface, giving the namespace a direct presence on the physical LAN:

1. **Prerequisite check**: Identify your host's physical Ethernet interface connected to the LAN (e.g., **eth0**, **enp3s0**). Ensure it is up and has a network connection:

   ```
   $ ip link show
   ```

2. **Create the MACVLAN interface in a namespace**: We will create a MACVLAN interface directly inside a new namespace. The **ip link add** command creates the virtual interface, specifies its type as **macvlan**, defines the parent interface, and sets the mode:

   ```
   $ sudo ip netns add macvlan-ns
   $ sudo ip link add macvlan0 link eth0 type macvlan mode bridge
   $ sudo ip link set macvlan0 netns macvlan-ns
   ```

   The second command creates a new interface named **macvlan0** that is a child of **eth0** and operates in bridge mode.

3. **Configure the interface and test connectivity**: Enter the namespace, bring the interface up, and obtain an IP address via DHCP or assign a static IP from your LAN's subnet:

   ```
   $ sudo ip netns exec macvlan-ns ip link set macvlan0 up
   $ sudo ip netns exec macvlan-ns ip addr add 192.168.1.100/24 dev macvlan0
   ```

   - Alternatively, for DHCP:

     ```
     $ sudo ip netns exec macvlan-ns dhclient macvlan0
     ```

4. Now, test connectivity from within the namespace. You should be able to ping other devices on your LAN and the internet, completely bypassing the host's IP stack:

   ```
   $ sudo ip netns exec macvlan-ns ping 192.168.1.1  # Ping your gateway
   $ sudo ip netns exec macvlan-ns ping 8.8.8.8      # Ping a public DNS
   ```

Despite its performance benefits, MACVLAN has several important limitations that can be stumbling blocks in production environments.

The most significant limitation is its dependence on promiscuous mode and

driver support. The parent interface must be placed in promiscuous mode, which is often blocked by default in cloud and virtualized environments (e.g., **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**), Azure) for security reasons. Even on bare metal, some **network interface card** (**NIC**) drivers do not properly support promiscuous mode or the specific kernel calls MACVLAN requires, leading to unreliable behavior or complete failure.

The bridge mode can confuse network switches. Suppose a switch uses a security feature called sticky MAC or port security. In that case, it may learn the MAC address of the first MACVLAN interface and then block traffic from the second, third, or subsequent interfaces on the same physical port, as it interprets them as a MAC spoofing attack. This makes the VEPA mode preferable in managed datacenter environments, but it requires a switch that supports the 802.1Qbg standard, which is not common on cheap consumer-grade hardware.

Another key consideration is that the host itself loses its IP address on the parent interface if that interface is used as a MACVLAN parent. When you create a MACVLAN interface, the parent interface can no longer have an IP address assigned to it directly. All IP communication for the host must be moved to a different interface or to a MACVLAN interface created in the host's default network namespace. This is often the most surprising aspect for newcomers.

Finally, there is a scalability concern. While creating hundreds of MACVLAN interfaces is possible, it can place a strain on the external physical switch, which must now learn and maintain a MAC address table entry for every single MACVLAN interface on that host, treating one physical port as hundreds of individual devices.

MACVLAN is a powerful tool for high-performance namespace networking on bare-metal hosts or in private virtualization environments where you control the hypervisor and network hardware. Its simplicity and performance are unmatched, but its inflexibility regarding promiscuous mode and host IP addressing make it unsuitable for many cloud or restricted environments.

Scaling with shared MACs using IPVLAN

MACVLAN's requirement for unique MAC addresses per interface becomes a significant scaling limitation in large-scale environments. Modern network switches maintain **content addressable memory** (**CAM**) tables that map MAC addresses to physical ports. These tables have finite sizes, often ranging from 8,000 to 32,000 entries on enterprise gear. In dense virtualization hosts running hundreds of containers, each with its own MACVLAN interface, a single physical port can consume thousands of switch table entries. This can lead to MAC table overflow, causing switches to indiscriminately flood unknown unicast traffic or drop frames, severely degrading network performance for all connected devices.

IPVLAN was developed explicitly to solve this scaling problem. Unlike MACVLAN, which assigns a unique MAC address to each virtual interface, IPVLAN interfaces within the same parent interface share a single MAC address. To the external switch, all IPVLAN interfaces attached to a single host port appear as a single network device with one MAC address but multiple IP addresses. This architectural shift provides immense scalability. A host can theoretically create thousands of IPVLAN interfaces without consuming any additional MAC address table entries on the connected network switch, making it ideal for high-density container orchestration platforms like Kubernetes.

The following figure describes a physical host using IPVLAN interfaces with a shared MAC address with the host interface eth0 to connect the network namespaces to the external network:

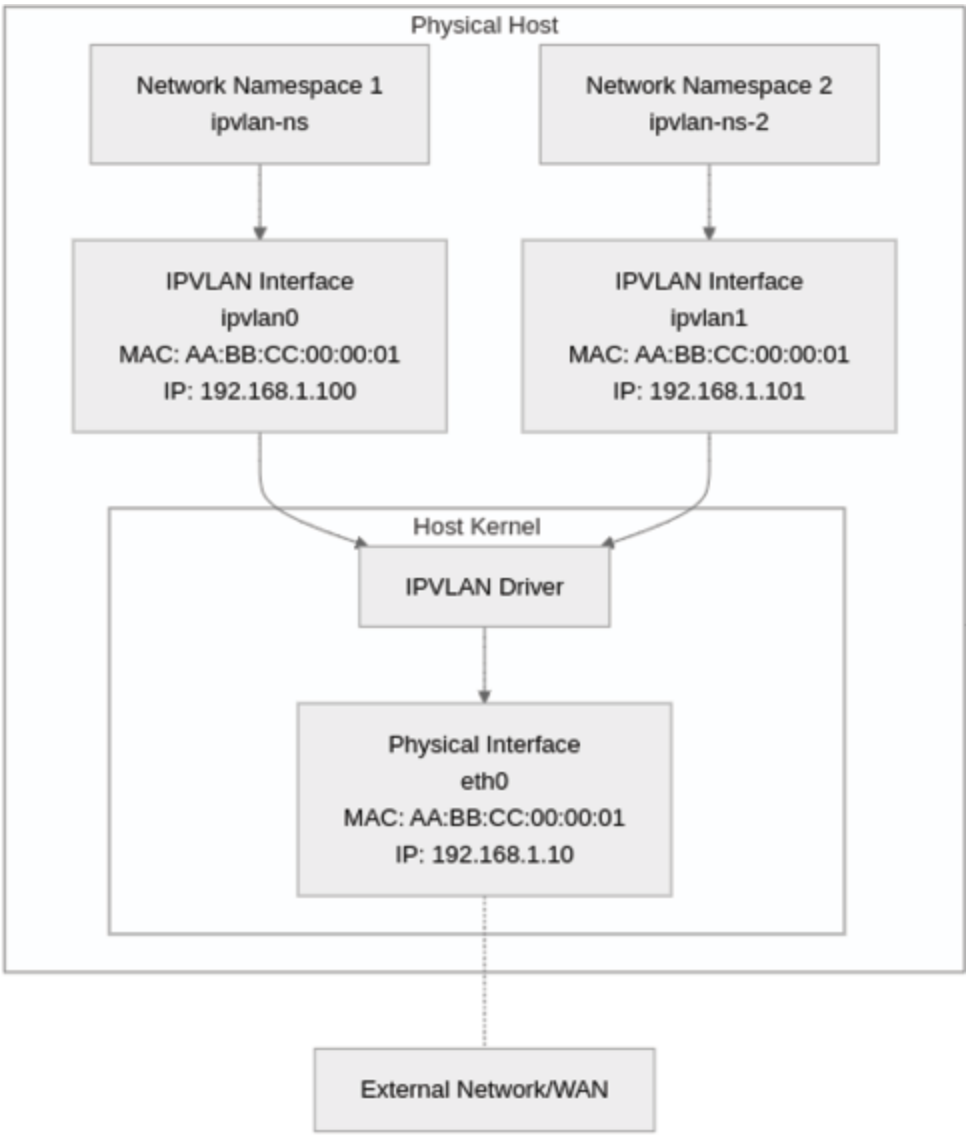


***Figure 7.2**: IPVLAN setup*

The shared MAC model also simplifies network policy enforcement. Since the switch only sees one device, port-level security policies are applied uniformly. Network administrators can manage access control at the IP level without worrying about MAC address spoofing or exhaustion. This makes IPVLAN particularly attractive in managed cloud and enterprise environments where network stability and security are paramount. However, this design also introduces a key operational difference—IPVLAN interfaces are inherently IP Layer or **Layer 3** (**L3**) aware, meaning they operate with a stronger understanding of IP routing compared

to the more L2-focused MACVLAN, which influences their available modes of operation.

Comparing IPVLAN and MACVLAN

Choosing between IPVLAN and MACVLAN requires a thorough understanding of their fundamental differences in operation, performance, and compatibility.

The most obvious difference is the MAC address strategy. MACVLAN assigns a unique MAC address to each virtual interface, making each namespace appear as a distinct physical device on the LAN. IPVLAN shares the parent interface's MAC address, making all virtual interfaces on a host appear as a single multi-homed device. This is the core trade-off: MACVLAN offers traditional L2 semantics at the cost of switch table space, while IPVLAN conserves switch resources but requires the network to operate with pure IP routing semantics.

Their behavior with broadcast and multicast traffic also differs significantly. In MACVLAN bridge mode, broadcast frames are efficiently delivered to all local MACVLAN interfaces without being forwarded to the external wire. In IPVLAN, the handling of broadcast traffic is more nuanced and depends on the chosen mode (L2 or L3). Generally, IPVLAN can be more efficient in suppressing unnecessary broadcast traffic, which is beneficial in very dense environments.

A critical practical difference is host networking. Using a physical interface as a MACVLAN parent requires moving the host's IP address to a different interface or to a MACVLAN interface itself. IPVLAN does not have this restriction. The parent interface can retain its IP address and normal host network connectivity while simultaneously acting as a parent for IPVLAN interfaces. This makes IPVLAN much easier to integrate into existing host configurations without disruptive changes.

Finally, there is a difference in driver support and maturity. MACVLAN has been in the kernel longer and is generally considered very stable. IPVLAN, while mature, can sometimes have driver-specific issues because it relies on a different kernel framework. It is always critical to test with your specific network hardware and kernel version before deploying IPVLAN widely in

production.

IPVLAN operational modes

IPVLAN operates in two distinct modes, which dictate how traffic is forwarded between interfaces and to the external network.

IPVLAN L2 mode operates most similarly to MACVLAN bridge mode. The driver acts as a virtual switch, but at the IP layer. Communication between IPVLAN interfaces on the same host is direct and does not leave the host. Communication with external hosts is sent out through the parent interface. The key difference from MACVLAN is that all frames share the same source MAC address. From a namespace perspective, L2 mode feels like being directly connected to the LAN. The namespace needs a default gateway configured to route traffic to other subnets, and ARP is used to resolve IP addresses to MAC addresses on the local network.

IPVLAN L3 mode is a unique and powerful feature. In this mode, the IPVLAN driver acts as a virtual router. There is no L2 broadcast domain; the concept of a local LAN segment is eliminated. All communication, even between IPVLAN interfaces on the same host, is routed by the kernel based on its routing tables. This means that namespaces can be placed on entirely different IP subnets even though they share the same physical parent interface. Since it operates at Layer 3, ARP is disabled for IPVLAN L3 interfaces. The external switch only ever sees IP packets from the parent's MAC address, making network presence incredibly simple and efficient. L3 mode is ideal for isolating tenants or applications into different logical subnets while maintaining the performance benefits of bypassing the host's main network stack.

Implementing an IPVLAN L2 network

This lab creates an IPVLAN interface in L2 mode, the most common configuration:

1. Load the kernel module if necessary (usually built-in):

```
$ sudo modprobe ipvlan
```

2. Create a namespace:

```
$ sudo ip netns add ipvlan-ns
```

3. Create an IPVLAN interface in L2 mode, attached to parent eth0:

```
$ sudo ip link add name ipvlan0 type ipvlan link eth0 mode l2
```

4. Move the interface into the namespace:

```
$ sudo ip link set ipvlan0 netns ipvlan-ns
```

5. From within the namespace, bring it up and assign an IP:

```
$ sudo ip netns exec ipvlan-ns ip link set ipvlan0 up
$ sudo ip netns exec ipvlan-ns ip addr add 192.168.1.150/24 dev ipvlan0
$ sudo ip netns exec ipvlan-ns ip route add default via 192.168.1.1
```

6. Test connectivity:

```
$ sudo ip netns exec ipvlan-ns ping 8.8.8.8
```

IPVLAN offers a compelling set of advantages for modern network infrastructure. Its premier advantage is massive scalability; by eliminating MAC address proliferation, it enables thousands of containers to run on a host without impacting network switch stability. Its simplified network view eases the burden on network administrators, as switches see a clean, single MAC address per host. The retention of the host's IP address on the parent interface simplifies configuration and management. Furthermore, the L3 mode provides a unique ability to create multiple routed subnets on a single host interface, offering advanced network segmentation without complex bridging.

However, IPVLAN is not without its limitations. Its primary constraint is driver compatibility. Not all physical network interface drivers support the IPVLAN kernel framework. It is most reliably used with common Intel and Mellanox NICs and their well-maintained Linux drivers. In virtualized environments, the underlying hypervisor's **virtual NIC** (**vNIC**) must also support it. This can make it less universal than MACVLAN.

The shared MAC address can also be a double-edged sword, breaking layer-2 dependent services. Any service that relies on seeing unique MAC addresses will fail. For example, a DHCP server that uses MAC addresses for static IP assignment will assign the same IP to all interfaces unless configured to use another unique identifier like the client ID, which is often

based on the IPVLAN interface's **DHCP Unique Identifier** (**DUID**). Similarly, some legacy license servers or security systems that bind to a MAC address will not function correctly.

Ideal use cases for IPVLAN are clear. It is the superior choice in large-scale Kubernetes clusters where the density of pods per node would otherwise overwhelm top-of-rack switches. It is also excellent for cloud-native applications that are designed to communicate via IP and **Domain Name System** (**DNS**), not L2 broadcasts. The L3 mode is perfect for multi-tenant environments on a single host, where each tenant's namespace requires isolation into a separate IP subnet while maintaining high-performance data paths.

IPVLAN represents an evolution of the network virtualization paradigm, prioritizing scalability and manageability over traditional L2 semantics. While its requirements can make it less plug-and-play than MACVLAN, its benefits in large-scale, IP-based environments are unparalleled, making it a critical technology for the infrastructure underlying modern container platforms.

Integrating VLANs with namespaces

VLANs have long been the cornerstone of network segmentation in enterprise environments, allowing a single physical network infrastructure to be partitioned into multiple logical broadcast domains. The IEEE 802.1Q standard defines the method for tagging Ethernet frames with a **VLAN identifier** (**VID**), enabling switches to logically separate traffic on the same physical wire. Traditionally, this segmentation ended at the server's physical NIC; the server itself would see either untagged (native) traffic or be aware of a single VLAN via a tagged sub-interface.

The integration of 802.1Q VLANs with Linux network namespaces unlocks a powerful paradigm: micro-segmentation at the application level. This allows a single physical host to act as a multi-tenant router, terminating multiple distinct VLANs and directing traffic from each VLAN into isolated namespace environments. For instance, a web server namespace

can be placed on the DMZ VLAN (e.g., VID 100), an application server namespace on a secure application VLAN (VID 200), and a database namespace on a locked-down backend VLAN (VID 300). All three can coexist on the same host, their traffic rigorously separated by VLAN tags until it is routed between them by a controlled, namespaced process (like a firewall or API gateway).

This is achieved by creating VLAN sub-interfaces on a physical parent interface configured as an 802.1Q trunk. A trunk port carries traffic for multiple VLANs and understands VLAN tags. The Linux kernel's networking stack can create virtual interfaces that represent these VLANs. The canonical naming convention is **<interface>.<VID>**, such as eth0.100 for VLAN 100 on physical interface eth0. The critical step is moving these VLAN sub-interfaces into individual network namespaces. Once inside, the namespace has a dedicated interface that is a member of a specific VLAN, and its entire network stack—including its routing table and firewall rules—operates within the context of that single broadcast domain. This provides a clean, standardized method for connecting namespaces to existing, VLAN-segmented physical networks.

Configuring a VLAN interface inside a namespace

This lab guides you through configuring a physical interface as a trunk port, creating a VLAN sub-interface, and placing it into a network namespace, effectively connecting the namespace to a specific VLAN on your physical network. The details are as follows:

1. **Prerequisite for host trunk configuration**: First, ensure the host's physical interface (**eth0** in this example) is up and has no IP address assigned. It will act as a pure trunk:

   ```
   $ sudo ip addr flush dev eth0
   $ sudo ip link set eth0 up
   ```

   The VLAN traffic must be allowed to pass through this interface. The **vlan** kernel module must be loaded, and the switch port connected to eth0 must be configured as a trunk port allowing your desired VLAN (e.g., VLAN 100).

2. **Create the VLAN sub-interface**: The **ip link** command to create the virtual VLAN interface. The type **vlan** specifier and the id parameter are key:

```
$ sudo ip link add link eth0 name eth0.100 type vlan id 10
```

   This command creates a new virtual interface named eth0.100. The link eth0 parameter specifies the parent physical trunk interface. The type **vlan id 100** tells the kernel this interface will strip the tag for VLAN 100, presenting untagged frames to whatever process uses this interface.

3. **Create a namespace and move the VLAN interface**: Now, create a namespace and place the VLAN interface inside it. This disconnects the interface from the host's default network namespace:

```
$ sudo ip netns add vlan100-ns
$ sudo ip link set eth0.100 netns vlan100-ns
```

4. **Configure networking within the namespace**: Enter the namespace to configure the VLAN interface with an IP address appropriate for that VLAN's subnet:

```
$ sudo ip netns exec vlan100-ns ip link set lo up
$ sudo ip netns exec vlan100-ns ip link set eth0.100 up
$ sudo ip netns exec vlan100-ns ip addr add 192.168.100.50/24 dev eth0.100
$ sudo ip netns exec vlan100-ns ip route add default via 192.168.100.1
```

5. **Verification and testing**: Verify the configuration from within the namespace and test connectivity to other devices on the same VLAN:

```
$ sudo ip netns exec vlan100-ns ip addr show eth0.100
$ sudo ip netns exec vlan100-ns ping 192.168.100.1
```

If the gateway responds, congratulations. You have successfully extended your VLAN 100 broadcast domain into an isolated network namespace. The namespace now has direct L2 connectivity to other devices on VLAN 100, and its traffic is neatly segregated from other VLANs by the physical switch infrastructure. This powerful pattern allows for the seamless integration of containerized and namespaced workloads into existing, traditionally segmented enterprise networks.

User-space networking with TUN and TAP

While MACVLAN and IPVLAN optimize the path for standard Ethernet and IP traffic within the kernel, TUN and TAP devices solve a different problem: they provide a mechanism for user-space programs to handle network traffic directly. These virtual network kernel interfaces act as a conduit between the kernel's networking stack and a software application running in user space, enabling the implementation of custom network protocols, VPNs, and other advanced networking functionalities that are impractical to build directly into the kernel.

The key to understanding TUN/TAP is recognizing the distinction between the two types of interfaces based on the **Open Systems Interconnection** (**OSI**) model layer at which they operate. A TAP device operates at the Layer 2 (Data Link Layer), dealing with raw Ethernet frames. When an application reads from a TAP device, it receives complete Ethernet frames, including headers and payload. When it writes to a TAP device, it injects a full Ethernet frame into the kernel's network stack, which then processes it as if it arrived from a physical wire. This makes TAP devices ideal for applications that need to emulate a virtual network adapter, such as hypervisors (QEMU/KVM), creating bridged networks for virtual machines, or for implementing software bridges and Ethernet tunnelling protocols that require frame-level manipulation.

Conversely, a TUN device operates at the Layer 3 (Network Layer), dealing exclusively with raw IP packets (or IPv6 packets). It strips away the Ethernet header, presenting only the IP packet to the user-space application. This abstraction is perfect for applications concerned with routing and encrypting IP traffic, as it removes the complexity of dealing with MAC addresses, ARP, and Ethernet framing. The most common use case for TUN devices is VPNs; the VPN client application uses a TUN device to receive IP packets from the kernel, encrypts them, and then sends the encrypted payload over a physical socket to the VPN server. The kernel routes all traffic destined for the VPN's subnet to the TUN device, which is handled entirely by the user-space VPN process.

Primary use cases of VPNs, hypervisors, and emulation

The ability to hand network traffic to a user-space program unlocks a wide range of powerful applications, making TUN/TAP interfaces a critical tool for advanced networking.

The most ubiquitous use case is VPNs. Applications like OpenVPN, WireGuard, and OpenConnect rely heavily on TUN devices. The workflow is consistent: the VPN software creates a TUN interface and configures the kernel to route a specific subnet (e.g., 10.8.0.0/24) to it. The kernel sends any IP packet destined for that subnet to the TUN device. The VPN application reads these raw IP packets from the TUN device, encapsulates them within a secure encryption protocol (like TLS or WireGuard's crypto), and transmits them over a regular TCP or UDP socket to the VPN gateway. The process is reversed for incoming traffic. This allows the complex cryptography and key exchange to be implemented in a safe, maintainable user-space environment while the kernel handles standard IP routing.

VM hypervisors like QEMU/KVM use TAP devices to provide networking for guest VM. When configured in *bridged* mode, the hypervisor creates a TAP device for each VM and connects it to a software bridge (e.g., **br0**) on the host. The VM's virtual NIC is connected to this TAP device. The hypervisor acts as the user-space application, and it reads Ethernet frames emitted by the guest OS from the TAP device and can forward them onto the physical network (via the bridge), or it can write frames received from the bridge into the TAP device, delivering them to the guest. This provides the VM with direct L2 connectivity to the host's network.

Beyond these, TUN/TAP is used for network emulation (e.g., simulating complex network topologies with tools like GNS3), implementing **software-defined networking** (**SDN**) data planes, and creating custom tunneling protocols for specialized applications. The simplicity of the model, like reading and writing packets from a file descriptor, makes it accessible to any programmer, fostering innovation in network services without requiring changes to the Linux kernel itself.

Creating a TUN device

Creating a TUN device requires the TUN kernel module and typically requires root privileges. The following commands will create device **tun0**:

```
$ sudo ip tuntap add dev tun0 mode tun
$ sudo ip link set tun0 up
$ sudo ip addr add 10.0.0.1/24 dev tun0
```

A user-space program can now open **/dev/net/tun**, claim this device, and begin reading IP packets that the kernel routes to the 10.0.0.0/24 network. The program can process these packets and inject responses back into the kernel via the same interface, enabling full control over the network flow.

Understanding Docker container networking

Docker's networking model is fundamentally an application of Linux network namespaces, virtual interfaces, and bridging, packaged into user-friendly abstractions called **network drivers**. Each driver represents a different connectivity pattern, with the none and host drivers representing the two extremes of the isolation spectrum. The details are as follows:

- **None driver**: The none driver provides maximum isolation by placing the container in its own network namespace but giving it no network interfaces whatsoever. When a container is launched with **--network none**, it has no external network connectivity, no loopback address (until manually configured), and no way to communicate with other containers or the host. This driver is used for specialized security-sensitive workloads where any network access is prohibited, or for containers that function purely as data processors using volume mounts, with no need for network communication. From a namespace perspective, Docker creates a standard network namespace for the container, but simply does not populate it with any virtual Ethernet devices connected to the outside world.
- **Host driver**: The host driver sits at the opposite extreme, providing minimum isolation by skipping network namespace creation entirely. A container launched with **--network host** shares the host's network namespace. This means the container sees and uses the host's network

interfaces, IP addresses, and routing tables directly. A process in the container binding to port 80 will conflict with any other process on the host trying to bind to port 80. While this mode offers performance benefits by eliminating the overhead of virtual interfaces and NAT, it sacrifices the fundamental security and isolation guarantees that containers provide. Its primary use case is for high-performance networking applications, such as load balancers or monitoring agents, that need direct access to the host's network stack and where port conflicts can be carefully managed.

- **Bridge driver**: The default bridge driver is the most common Docker network and perfectly illustrates the application of the Linux networking primitives discussed in this chapter. When the Docker daemon starts, it automatically creates a single software bridge interface named docker0, typically with an IP address in the 172.17.0.0/16 range. This bridge exists in the host's network namespace and acts as a virtual switch for all containers on the default bridge network.

Docker container lifecycle

The container lifecycle reveals the underlying mechanics. When a container is started (without specifying a network), Docker:

- Creates a new network namespace for the container.
- Creates a **virtual Ethernet** (**veth**) pair.
- Places one end of the veth pair (vethX) inside the container's namespace, renaming it eth0.
- Connects the other end of the veth pair to the docker0 bridge.
- Assigns an IP address from the bridge's subnet to the container's eth0 interface.
- Configures iptables rules on the host to provide outbound Internet connectivity via MASQUERADE (SNAT) and to manage port publishing (**-p** flag), which creates DNAT rules forwarding traffic from the host's IP and port to the container's IP and port.

The following figures show two Docker containers on the physical host

connecting to the Docker0 bridge on the host. The traffic from the container to the external network goes through SNAT translation:

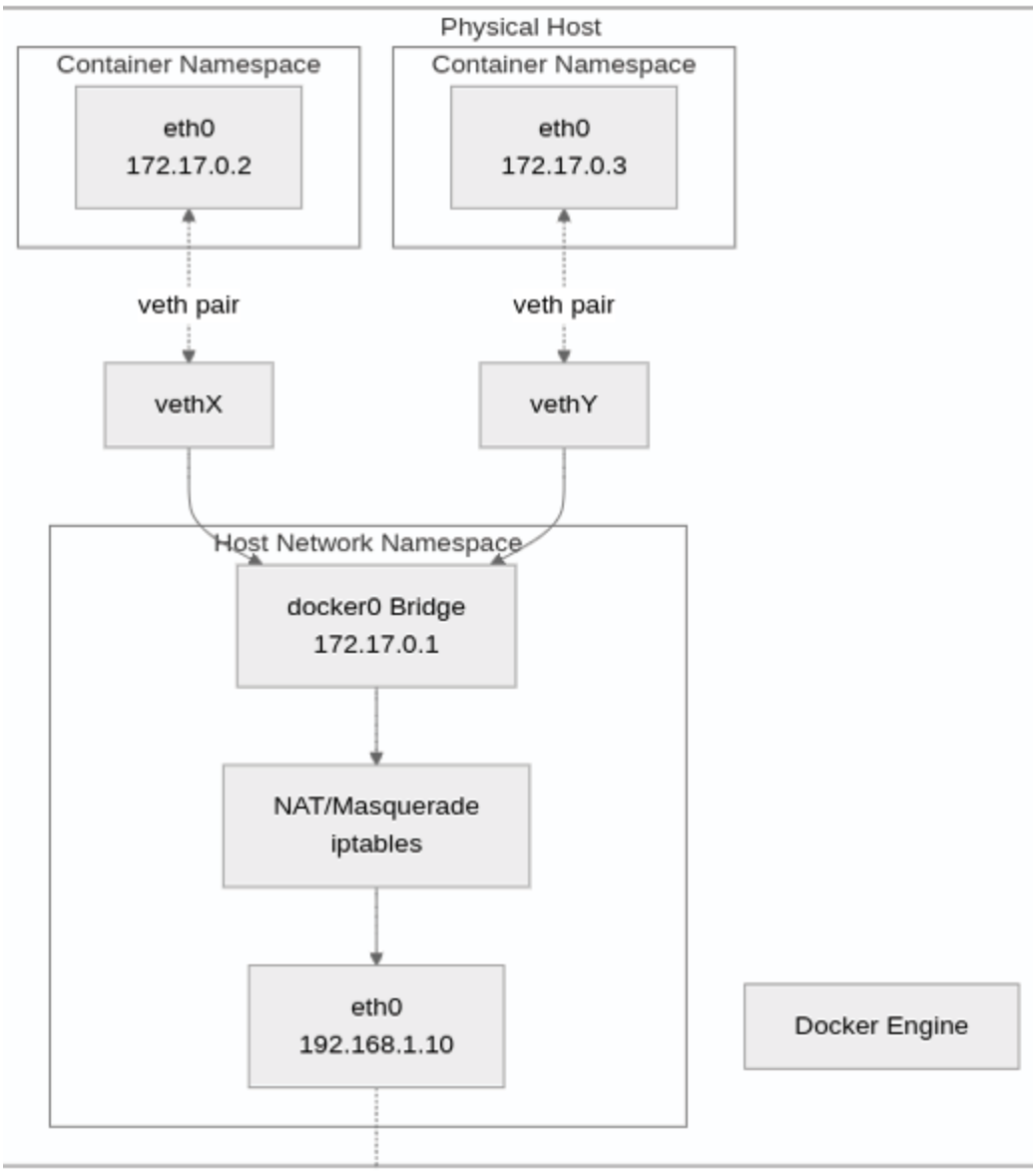


***Figure 7.3**: Docker network setup*

This architecture provides isolated, out-of-the-box connectivity. Containers on the same default bridge can communicate with each other by IP address, and they can access the external internet through the host's NAT mechanism. However, the default bridge has limitations: service discovery is manual (reliant on static IPs or injected environment variables), and the

network is not isolated from other containers on the same host.

Inspecting Docker namespaces and interfaces

Docker conveniently abstracts away the complex **ip** commands, but the underlying structures are still present and inspectable, which is crucial for debugging. The key challenge is that Docker does not create entries in **/var/run/netns/**, making its namespaces invisible to **ip netns** list.

To overcome this, we must find the network namespace of a running container. Every process, including those in containers, has a **/proc/<pid>/ns/**net file. First, find the full container ID and the PID of its init process:

```
$ docker inspect --format '{{ .State.Pid }}' <container-name>
$ docker inspect -f '{{.State.Pid}} {{.Name}}' $(docker ps -q) | grep -v ^0
```

Once you have the PID (e.g., 12345), you can inspect the container's network namespace using one of two methods. You can use the **nsenter** command to execute network tools within the container's context:

```
$ sudo nsenter -t 12345 -n ip addr show
$ sudo nsenter -t 12345 -n iptables –L
```

Alternatively, you can create a temporary mount to make the namespace visible to **ip netns**:

```
$ sudo ln -s /proc/12345/ns/net /var/run/netns/container12345
$ ip netns list  # Will now show 'container12345'
$ ip netns exec container12345 ip route sho
```

On the host, use **ip link show,** and **brctl show docker0** (if bridge-utils is installed) to see the veth interfaces connected to the bridge. This reveals the complete plumbing; for every container, a veth pair connects its isolated network namespace to the shared docker0 bridge on the host.

Docker's container network model

The bridge, none, and host drivers are implementations of Docker's high-

level CNM. This model defines the core components that comprise a Docker network:

- **The sandbox**: This is the isolated network namespace. It contains the configuration for a container's network stack (interfaces, routes, DNS). A sandbox is specific to a single container.
- **The endpoint**: This is the virtual network interface (e.g., the eth0 inside the sandbox). An endpoint acts as a connector, belonging to one sandbox and one network. It is typically one end of a veth pair.
- **The network**: This is the virtual network, implemented as a Linux bridge, VLAN, or overlay. It is a collection of endpoints that can communicate with each other. A network can have many endpoints, and an endpoint can only belong to one network. When you run **docker network create mynet**, you are creating a network object.

  When you run docker run **--network mynet**. Docker creates a new Sandbox (network namespace) for the container, creates an endpoint (veth interface) inside it, and connects that endpoint to the **mynet** network (bridge). This model provides immense flexibility. Users can create multiple networks (e.g., frontend-net, backend-net) and attach containers to them, creating custom, isolated network topologies on a single host. This is a vast improvement over the legacy default bridge, enabling secure multi-tier application architectures where only specified containers can communicate (e.g., web containers on frontend-net can talk to app containers on backend-net, but cannot talk to database containers on a separate **db-net**). The CNM is the abstract blueprint that Docker uses to map user intent onto the specific Linux kernel networking primitives we have explored throughout this chapter.

Kubernetes networking fundamentals

Kubernetes introduces a powerful and strict networking model that fundamentally relies on the concepts of network namespaces and virtual interfaces. Unlike Docker's container-focused model, Kubernetes operates at the pod level, where a pod is the smallest deployable unit and can contain

one or more tightly coupled containers. The foundational principle of Kubernetes networking is that every pod gets its own unique IP address. This IP address is routable within the entire cluster network, meaning any pod can communicate with any other pod without NAT.

This *IP-per-pod* model simplifies application design and operation significantly. It eliminates the need for developers to manually manage port conflicts between pods, as each pod has its own IP and thus its own entire port space. Applications can use well-known ports (e.g., 80 for HTTP) regardless of how many copies of the pod are running on the same node. This model also provides a clean, consistent network identity for each pod, making debugging, tracing, and security policy enforcement (via NetworkPolicy) vastly more straightforward. The model mandates that this IP address is fully routable and that all pods can reach all other pods, nodes, and the internet. It is the responsibility of the underlying container runtime and the networking solution using CNI plugin to implement this model on top of the host's physical network.

Pod's network namespace

A critical implementation detail is how the IP-per-pod model is enforced. Since all containers within a single pod share a single network namespace, this means all containers in a pod see the same network interfaces, IP address, routing table, and port space. If one container in a pod listens on port 8080, another container in the same Pod can connect to that port via localhost:8080.

Kubernetes achieves this shared namespace through a clever architectural pattern using an invisible *pause* container. For every pod scheduled on a node, the kubelet (node agent) first creates a special infrastructure container whose sole purpose is to acquire and hold the pod's network namespace. This container does nothing but pause, hence its name. The image for this container is very small and minimal (e.g., **registry.k8s.io/pause**:**3.9**). All other *user* containers (e.g., your application container) deployed within the pod are then started with the Docker **--network=container**:**<pause-container-id>** option (or its CRI equivalent). This instructs the container runtime to forego creating a new network namespace and instead have the

user container join the network namespace of the pre-existing pause container.
The following figure describes how the kubelet orchestrates the creation of a shared network namespace for all the containers in a pod:

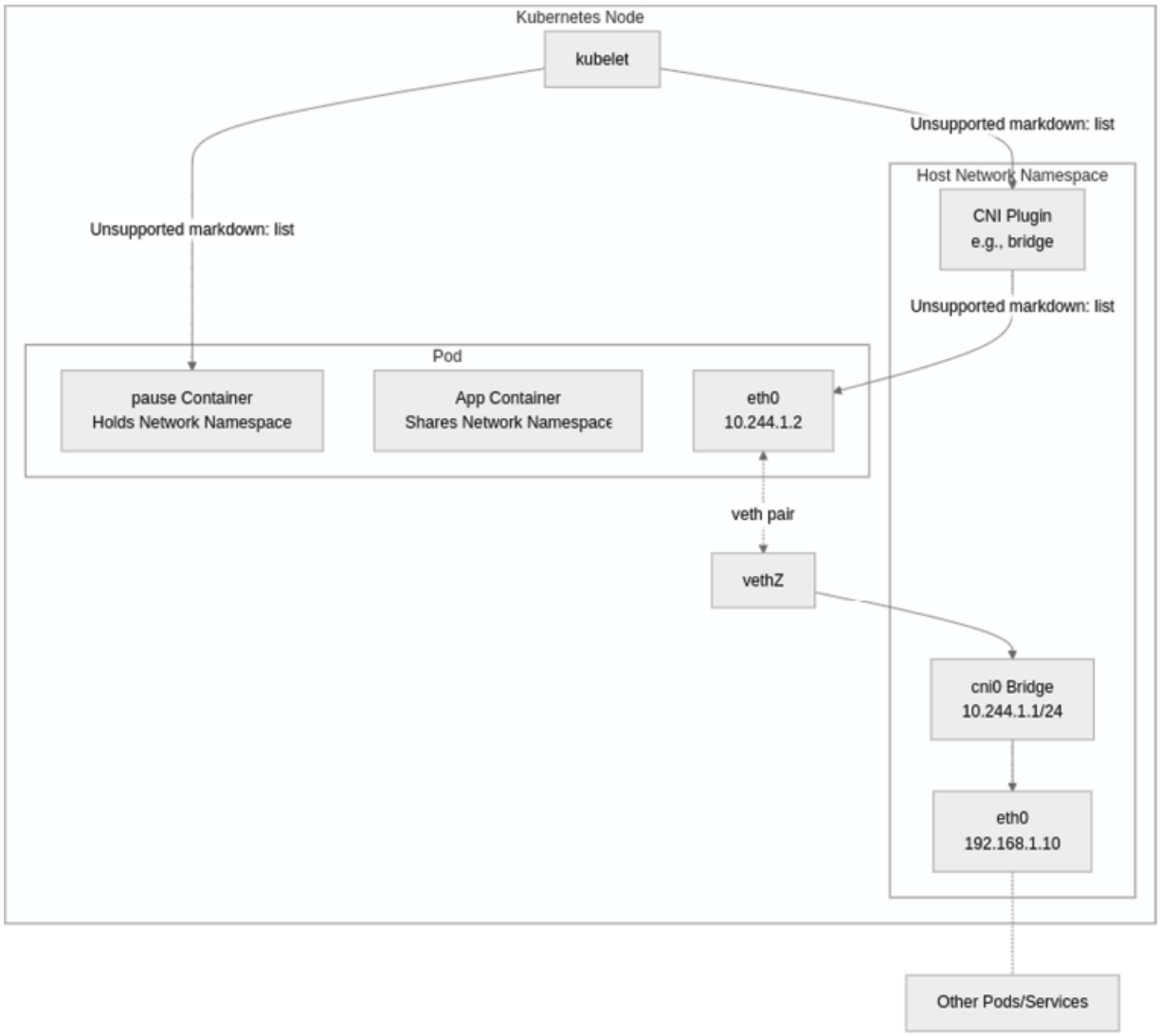


***Figure 7.4**: Kubernetes network setup*

This design ensures the network namespace remains stable and exists for the entire lifetime of the pod, even if the application containers within it crash and restart. The pause container acts as the anchor for the pod's network identity, guaranteeing that the IP address and network configuration persist consistently. This is a prime example of how

Kubernetes builds upon low-level kernel primitives (namespaces) to create a robust, higher-level abstraction.

Container network interface standard

Kubernetes itself is deliberately agnostic about how networking is implemented. It does not contain code to create veth pairs or bridges. Instead, it delegates all networking operations to plugins that adhere to the CNI specification. CNI is a simple, plugin-based framework that defines how a network is attached to a container (or pod's pause container) at runtime.

The CNI specification is minimal. It expects a plugin to be an executable that the kubelet can call with a command (e.g., ADD, DEL) and a set of **JavaScript Object Notation** (**JSON**) formatted configuration data (e.g., the container ID, network namespace path). The plugin's job is to perform the necessary operations to connect the container to the network and return a result (e.g., the assigned IP address) in a standard JSON format. This decoupling is powerful. It allows for a rich ecosystem of CNI plugins to exist, each implementing different network capabilities, from simple layer-2 bridges (e.g., bridge) and overlay networks e.g., Flannel, Calico's **Virtual eXtended Local Area Network** (**VXLAN**) to advanced policy-driven meshes (e.g., Cilium, Calico).

When a pod is scheduled to a node, the kubelet's workflow is:

1. Create the pause container to establish the sandbox and get the network namespace path.
2. Execute the configured CNI plugin with the ADD command, passing it the container ID and the path to the network namespace file (**/proc/<pause-pid>/ns/net**).
3. The CNI plugin performs its work (e.g., creates a veth pair, connects it to a bridge, and assigns an IP).
4. The kubelet reads the IP address returned by the CNI plugin and updates the pod's status. This process happens for every pod on the node, and the chosen CNI plugin is responsible for ensuring the pod IP is routable according to the Kubernetes network model.

CNI plugin workflow

To understand the magic, let us deconstruct what a simple CNI plugin (like the bridge plugin) does when the kubelet calls it with an **ADD** command for a new pod. The details are as follows:

- **Parse configuration**: The plugin reads its CNI configuration JSON. This file defines the plugin's name (**bridge**), the name of the bridge to use on the host (e.g., **cni0**), the IP subnet to use for pods (e.g., **10.244.1.0/24**), and the path to the host's iptables binary.
- **Check or create bridge**: The plugin checks if the bridge (e.g., **cni0**) exists on the host. If not, it creates it and assigns it an IP address (e.g., 10.244.1.1/24). This bridge will be the gateway for all pods on this node.
- **Create veth pair**: The plugin creates a virtual Ethernet (veth) pair. One end will be placed inside the pod's network namespace, and the other will remain on the host.
- **Move interface**: The plugin moves one end of the veth pair into the pod's network namespace (using the namespace path provided by the kubelet) and renames it to eth0.
- **Configure pod interface**: Inside the namespace, the plugin brings the **lo** interface up, brings the **eth0** interface up, and assigns it the next available IP address from the configured subnet (e.g., 10.244.1.2/24). It also adds a route setting the bridge's IP as the default gateway.
- **Connect host interface**: The plugin connects the other end of the veth pair to the bridge (**cni0**).
- **Setup NAT**: The plugin adds iptables rules on the host to perform MASQUERADE (SNAT) for traffic originating from the pod's subnet and destined for the outside world, enabling internet access.
- **Return result**: The plugin returns a JSON result to the kubelet containing the assigned IP address and other information, completing the operation.

This process, repeated by the CNI plugin for every pod, is how the abstract Kubernetes networking model is concretely implemented on each node, weaving together namespaces, virtual interfaces, and routing to create a

unified cluster network.

Network debugging in an orchestrated world

In an orchestrated environment, direct shell access to nodes and containers is often restricted or impractical at scale. Kubernetes provides first-class tools for debugging that respect the abstraction layers of pods and namespaces. The primary tool is **kubectl exec**, which allows you to execute commands directly inside a running container. This is the quickest way to perform basic network checks from a pod's perspective.

Commands like **kubectl exec <pod-name> -- nslookup <service-name>** test service discovery via CoreDNS, while **kubectl exec <pod-name> -- curl -v http://<another-pod-ip>:port** tests basic HTTP connectivity to another pod. However, many production containers are built from minimal images that lack even basic networking tools like ping, ip, nslookup, or curl. This is where the pattern of ephemeral debug containers becomes invaluable. Instead of modifying the application image, you can temporarily run a sidecar container packed with utilities in the pod's network namespace. The **kubectl debug** command facilitates this. For example, launching a troubleshooting container using the popular **nicolaka/netshoot** image provides immediate access to a full network toolkit:

```
$ kubectl debug -it <pod-name> --image=nicolaka/netshoot --target=<container-name>
```

The **--target** flag is crucial; it tells Kubernetes to share the target container's namespaces (including the network namespace) with the new debug container. This allows you to run **tcpdump**, **ip addr**, **ss**, and other diagnostics from the network context of the application pod without altering the application container itself. This is the recommended, non-intrusive method for debugging network issues within a pod.

Host-level node inspection

While **kubectl exec** operates within the Kubernetes API abstraction, sometimes it is necessary to break through and inspect the raw Linux

underpinnings on the node itself. This is essential for diagnosing issues with the CNI plugin, host-level firewall rules (**iptables/nftables**), or routing daemons. The key to this is finding the host-level PID of a pod's *pause* container and using the **nsenter** command.

First, you need to shell into the node hosting the problematic pod. Once on the node, find the PID of the pod's pause container. The **crictl** utility (part of the CRI standard) is the modern tool for this:

```
$ sudo crictl pods # List all pods on the node with their IDs

# List containers in a pod to find the pause container's ID
$ sudo crictl ps --pod <pod-id>

# Inspect the pause container to get its PID
$ sudo crictl inspect <pause-container-id> | grep -i pid
```

With the PID (e.g., 12345), you can now use **nsenter** to explore the pod's network namespace from the host. The command **sudo nsenter -t 12345 -n <command>** executes any network command within the pod's namespace. This is incredibly powerful for deep inspection:

- **sudo nsenter -t 12345 -n ip addr show** reveals the complete interface list from the pod's perspective, showing the **eth0** interface created by the CNI plugin.
- **sudo nsenter -t 12345 -n ip route** show displays the routing table inside the pod, showing the default gateway (typically the CNI bridge's IP).
- **sudo nsenter -t 12345 -n iptables -L -n -v** shows any netfilter rules that apply specifically within the pod's namespace, which is critical for debugging NetworkPolicies implemented by CNI plugins like Calico.
- **sudo nsenter -t 12345 -n tcpdump -i eth0 -w pod_traffic.pcap** performs a packet capture on the pod's eth0 interface, allowing you to analyze raw traffic flows.

This approach provides ultimate visibility. You are no longer limited by the tools installed in the application container; you can use the full suite of host debugging utilities to inspect the network environment that the CNI plugin

constructed for the pod. This deep inspection capability, moving seamlessly between the Kubernetes API and the underlying Linux kernel primitives, is what separates competent administrators from true experts in diagnosing complex orchestrated network failures. It connects the high-level abstractions of pods and Services directly back to the namespaces, veth pairs, and bridges that form their foundation.

Security hardening and considerations

The principle of least privilege is paramount in containerized environments, and Linux Capabilities provide the mechanism to enforce it at the kernel level. Traditionally, a process was either privileged (root, UID 0) or unprivileged. Capabilities break down the all-or-nothing power of root into distinct units, allowing for more granular control. For network namespaces, two capabilities are particularly critical.

CAP_NET_RAW is the capability to create raw and packet sockets. This is required for using utilities like **ping** (which uses ICMP sockets) and **tcpdump** (which uses packet sockets to capture traffic). Without this capability, a process within a namespace cannot generate raw IP traffic or perform network sniffing. **CAP_NET_ADMIN** is a broader and more powerful capability. It grants the privilege to perform various network administration tasks, including modifying routing tables, configuring network interfaces (setting IP addresses, bringing them up/down), managing firewall rules (iptables), and enabling IP forwarding. This capability is essentially required for any process that needs to modify the network stack of its namespace.

In a secure deployment, application containers should almost never be granted these capabilities. **CAP_NET_RAW** should be dropped to prevent network reconnaissance from within a compromised container. **CAP_NET_ADMIN** must be rigorously restricted, as it would allow a malicious actor within a container to completely reconfigure its own network environment, potentially bypassing security controls, creating unauthorized tunnels, or disrupting network connectivity. Kubernetes pod security contexts and Docker **--cap-drop** flags are used to explicitly remove

these capabilities, ensuring containers run with only the minimal privileges they absolutely require.

Firewall between namespaces with network policies

While namespaces provide isolation, they are not a firewall. By default, pods in a Kubernetes cluster can communicate freely, a principle known as flat networking. Network policies are the Kubernetes resource that defines how groups of pods are allowed to communicate with each other and other network endpoints. They act as a distributed, namespace-aware firewall.

A NetworkPolicy is a namespaced resource that selects pods using labels and defines ingress (inbound) and egress (outbound) rules. These rules can specify allowed traffic based on source/destination pod selectors (within the cluster), IP **Classless Inter-Domain Routing** (**CIDR**) blocks (external networks), and specific ports and protocols. Crucially, the enforcement of these policies is not done by a central component but is delegated to the CNI plugin (e.g., Calico, Cilium, Antrea). The plugin is responsible for translating the high-level NetworkPolicy spec into low-level, namespace-specific data plane rules.

For example, a policy can be written to isolate a namespace containing databases, only allowing ingress traffic on port 5432 from pods labeled as **app**: **backend-api** and blocking all other inbound connections. It can also control egress, preventing pods from contacting external malicious IP ranges. The implementation of these rules occurs within the pod's network namespace (via iptables/nftables or **Extended Berkeley Packet Filter** (**eBPF**) programs) or on the virtual switch (e.g., Open vSwitch) to which the pod is connected. This provides a declarative, API-driven method for implementing micro-segmentation and zero-trust networking principles between namespaces, moving beyond simple isolation to enforced, intent-based communication rules.

Performance considerations

The choice of networking virtualization technology has direct performance implications. MACVLAN/IPVLAN in bridge/L2 mode offers near-native

performance by bypassing the host's main network stack, making them ideal for latency-sensitive applications. However, they can be limited by external switch support and host driver compatibility. The veth pair + bridge model used by default in Docker and many CNI plugins introduces a small amount of overhead due to the additional bridging and routing logic in the host kernel, but it is highly flexible and universally supported. Overlay networks (e.g., VXLAN, WireGuard) used in multi-node clusters add encapsulation/decapsulation overhead, increasing CPU usage and slightly reducing throughput, but are necessary for creating a unified cluster network across different physical subnets. Understanding these trade-offs is essential for selecting the right networking solution for a given workload.

Conclusion

This chapter has journeyed from the fundamental kernel primitive of the network namespace to the sophisticated orchestration platforms that rely on it. We began by mastering isolation with ip netns, then built connectivity using virtual interfaces like veth, MACVLAN, and IPVLAN. We explored how these components form the foundation of Docker's networking and are orchestrated at scale in Kubernetes through the CNI framework. Finally, we discussed how to debug and secure these complex environments. The consistent theme is that modern cloud-native networking is not magic; it is the applied integration of these Linux kernel features, providing the agility, scalability, and isolation required by today's applications.

In the next chapter, we will explore traffic shaping and **quality of service** (**QoS**) configuration in Linux.

CHAPTER 8

Traffic Shaping and QoS

Introduction

In the pristine world of the network, bandwidth is often represented as a clean, wide pipe, a seemingly limitless channel for the unimpeded flow of data. The reality of network operation, however, is far closer to a complex and bustling metropolitan highway system during peak hours. On this digital pathway, a sudden large file transfer can resemble a truck merging into traffic, while the delicate, real-time packets of a voice call are like an ambulance needing a clear and immediate path. Without rules, without lanes, and without traffic signals, this system rapidly devolves into a gridlock of collisions, delays, and frustration.

This is the fundamental problem that **quality of service** (**QoS**) seeks to solve. It is the discipline of managing network resources to ensure that critical applications receive the performance they require, even when the network is under heavy load. QoS is not about creating more bandwidth, but rather about intelligently managing the bandwidth that exists. It is the discipline of defining priority, of deciding which traffic must flow immediately and which can afford to wait.

At the heart of practical QoS implementation in the Linux world lies a powerful and versatile tool **traffic control** (**tc**), part of the iproute2 suite. tc tool provides the mechanisms to shape, schedule, and police the packets

flowing through a network interface. While QoS defines the policy, tc provides the knobs to enforce that policy. It allows the network administrator to construct sophisticated traffic control structures, creating virtual lanes, speed limits, and queues to tame the chaos of competing data flows.

This chapter delves into the principles of traffic shaping and QoS, moving from the conceptual framework to the concrete syntax. We will explore how tc can be used to implement the essential QoS paradigms like controlling the rate of data transmission with shapers, ensuring fair access with schedulers, and marking or dropping packets with filters and policers.

Structure

This chapter covers the following topics:

- Understanding the fundamental problem
- Differentiating between QoS and traffic control
- Linux traffic control system
- Differentiated services code point
- First-in first-out qdiscs
- Rate limiting with Token Bucket Filters
- Hierarchical tree for bandwidth allocation
- Stochastic fairness queuing
- Prioritizing traffic flows with priority queuing
- Classifying packets into streams
- Active queue management
- Policing traffic to enforce strict rate limits
- Applying traffic shaping to ensure predictable rates
- Ingress and egress policing and shaping
- Integrating traffic control with netfilter
- Deploying a complete and practical traffic shaping policy

Objectives

Upon completing this chapter, the reader will be able to design and implement Traffic Shaping and QoS policies. You will learn about the Linux traffic control suite and how to use it to effectively manage network bandwidth, prioritize critical applications, and ensure reliable performance under constrained conditions. You will develop the skills to classify network traffic using various criteria such as IP addresses, ports, protocols, and **differentiated services code point** (**DSCP**) markings. You will learn about creating multi-tiered service levels that prioritize latency-sensitive services like video conferencing over bulk data transfers. You will gain the ability to design and implement ingress and egress policies that mitigate network congestion, minimize jitter and packet loss, and enforce bandwidth guarantees and ceilings. You will learn about configuring traffic shapers to smooth bursty traffic, policers to enforce absolute limits, and schedulers to manage packet transmission order. This will help you to maintain reliable performance under constrained or contended network conditions. We will explore rate-limiting traffic by enforcing absolute bandwidth boundaries using mechanisms like token buckets. The chapter will also address the critical challenge of bufferbloat. You will learn about **Active Queue Management** (**AQM**) and how to proactively manage queue lengths rather than allowing buffers to fill uncontrollably, thereby reducing standing queues and minimizing **Transmission Control Protocol** (**TCP**) latency.

Understanding the fundamental problem

The entire discipline of traffic shaping and QoS is predicated on a single, inescapable reality of physical networks: *bandwidth is a finite resource*. This concept of finiteness is the foundational problem that all subsequent techniques attempt to solve. While it is tempting to think of a network link as a vast, empty pipeline, a more accurate analogy is that of a shared, fixed-width conduit that must carry every type of communication simultaneously. This inherent limitation leads to a state of constant contention, where diverse data streams compete for their share of this constrained space.

The consequences of this contention manifest in several ways that directly degrade user experience and application performance. When the total

demand for bandwidth exceeds the available capacity of the link, the network device must buffer the excess packets, a process that introduces latency. As buffers fill, this latency increases, causing noticeable delays in interactive applications like voice calls or online gaming. Should the influx of data persist, these buffers will eventually overflow, leading to packet loss. For reliable protocols like TCP, this loss triggers retransmissions, further congesting the link in a negative feedback loop. For real-time protocols like **User Datagram Protocol** (**UDP**), the loss is permanent, resulting in dropouts, corrupted video, or robotic audio.

While a consistent, predictable delay can often be compensated for, unpredictable jitter is profoundly disruptive to real-time communications. It occurs precisely because of the bursty, uncontrolled nature of data flows competing on a congested link; a large file transfer packet can momentarily *jump the queue* ahead of a voice packet, delaying its arrival and destabilizing the steady stream of data.

Therefore, the fundamental problem is not merely a lack of bandwidth, but the inefficient contention for it. Different applications have vastly different tolerances for these network impairments. A bulk file transfer is bandwidth-hungry but largely insensitive to latency or jitter, whereas a **Voice over Internet Protocol** (**VoIP**) call requires minimal bandwidth but is critically dependent on low latency and jitter. Left unmanaged, they conflict destructively. The goal of traffic shaping and QoS is to manage the contention for it, imposing order upon the chaos to ensure that the right applications get the right resources at the right time.

Differentiating between QoS and traffic control

In the landscape of network management, the terms QoS and traffic control are often used synonymously, yet they represent distinct concepts that operate at different levels of abstraction. Understanding this distinction is crucial for moving from a theoretical goal to a practical implementation. At its core, the difference is one of policy versus mechanism, of the what versus the how.

QoS is the strategic objective. It is a set of requirements and desired outcomes designed to ensure the reliable performance of critical applications on a network. QoS is the policy that defines which types of traffic are important and what level of performance they require. It answers business and application-level questions like, *Should our VoIP phones have priority over web browsing? What is the maximum bandwidth a single user can consume? How do we prevent a backup job from saturating the wide area network (WAN) link during business hours?* QoS is the blueprint for network behaviour, specifying that real-time traffic must have low latency, that mission-critical data must be delivered reliably, and that best-effort traffic should not starve other applications.

Traffic control is the tactical implementation of that policy. It comprises the low-level mechanisms and tools that enforce the rules set forth by the QoS strategy. In the Linux ecosystem, traffic control is synonymous with the **tc** command and its underlying infrastructure. It is the engine that performs the actual work of shaping, scheduling, and policing packets. While QoS decides that voice traffic should be prioritized, traffic control is the system that identifies voice packets, places them in a high-priority queue, and ensures they are transmitted ahead of packets in lower-priority queues. It manipulates the flow of data through queues, classifies packets into streams, and allocates bandwidth. These concepts will be discussed in the following section on QoS metrics.

Exploring the key QoS metrics

To effectively manage network performance, one must first understand the language used to measure it. The quality of any network connection is quantified through four fundamental metrics, each describing a different dimension of how data travels from source to destination. These metrics are as follows:

- Bandwidth
- Latency
- Jitter
- Loss

They form the essential criteria that QoS mechanisms are designed to influence and protect. The details are as follows:

- **Bandwidth**: Often used synonymously with speed, refers to the maximum rate of data transfer across a network path, typically measured in bits per second. It represents the capacity of the digital conduit, like the width of the highway. While a high bandwidth is desirable, it is merely a potential; its effective use is dictated by how this capacity is shared among competing flows. Without management, a single application can consume all available bandwidth, starving others in a phenomenon known as **congestion**, making bandwidth control a primary function of traffic shaping.
- **Latency**: Sometimes called **delay**, is the time it takes for a packet to travel from its source to its destination, measured in milliseconds. This is the propagation time of the journey. Low latency is critical for interactive applications such as video conferencing, online gaming, and remote desktop sessions, where any perceptible delay disrupts the flow of communication. High latency can be caused by physical distance, network congestion, or processing delays within networking equipment.
- **Jitter**: It is the variation in latency over time. If latency is the average travel time, jitter is the inconsistency between those times. Imagine packets as cars leaving at regular intervals; high jitter means some cars arrive bunched up while others arrive with large gaps. This is particularly damaging for real-time streaming protocols like VoIP, which rely on a steady, predictable flow of packets. The receiving end must use a jitter buffer to smooth out these variations, but excessive jitter leads to buffer underruns, resulting in choppy audio or video and corrupted communication.
- **Packet loss**: It is the percentage of packets sent that fail to reach their destination. This is the ultimate failure in data delivery, like a letter lost in the mail. Loss occurs primarily when network buffers become saturated and are forced to discard incoming packets. For reliable protocols like TCP, packet loss triggers retransmissions, which in turn increases latency and reduces effective throughput. For real-time

protocols like UDP, however, there is no retransmission; lost packets are simply gone, manifesting as dropouts, artifacts, or silence in the media stream.

Together, these four metrics provide a complete picture of network health. A successful QoS policy does not merely maximize one at the expense of the others; it seeks an intelligent balance, prioritizing and protecting traffic based on its unique sensitivity to each metric. Voice traffic, for instance, requires minimal bandwidth but is highly sensitive to latency, jitter, and loss. A file transfer, in contrast, is hungry for bandwidth but largely tolerant of delay. Understanding these relationships is the first step in sculpting network traffic to meet application demands.

Linux traffic control system

At the heart of practical traffic management on Linux systems lies the traffic control system, universally accessed through its command-line interface, **tc**. The traffic control system provides a powerful framework where abstract policies of QoS are translated into concrete action, providing the kernel-level mechanisms to shape, schedule, and filter the flow of network packets. Its architecture is sophisticated and flexible, designed to manage the complex problem of network contention directly within the operating system's networking stack.

The Linux traffic control system operates on a fundamental principle that every packet egressing a network interface must pass through a series of structured, configurable components that collectively decide its fate. This journey is not a simple queue but a programmable pipeline, often visualized as a tree-like hierarchy. The primary building blocks of this architecture are:

- Qdiscs
- Classes
- Filters
- Actions

In the following section, we will examine these components of Linux network traffic control.

Examining the core components of traffic control

The architecture of the Linux traffic control system is built upon four fundamental components that work together to manage network traffic. Understanding these building blocks, qdiscs, classes, filters, and actions, is essential for constructing effective traffic control policies. Each component serves a distinct purpose in the packet processing pipeline, creating a flexible and hierarchical system for traffic management. The following figure describes the Linux traffic control system, which classifies packets based on match criteria and applies the configured action to the classified packets:

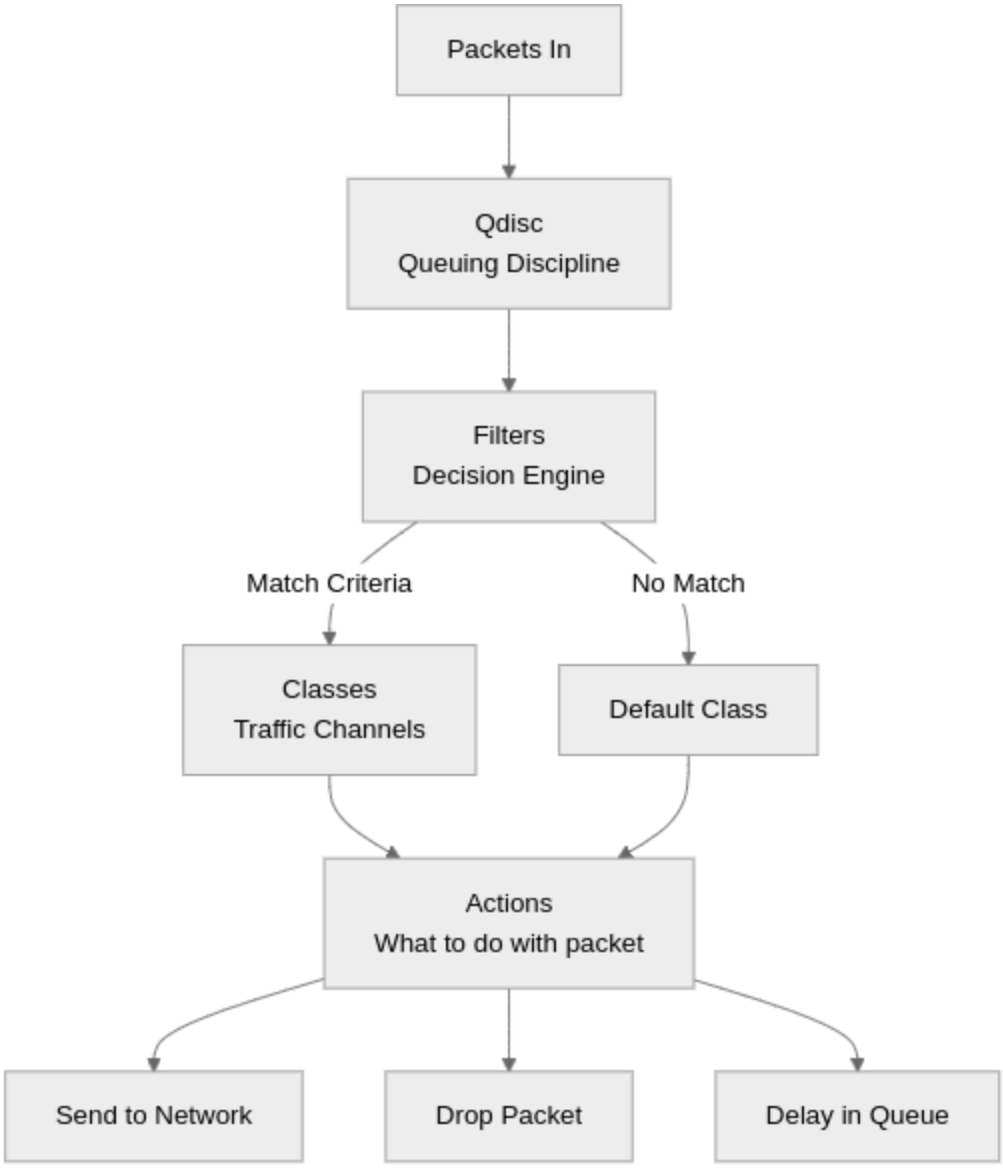


***Figure 8.1**: Components of traffic control*

The main components of the traffic control system on Linux are as follows:

- **Qdisc or queuing discipline:** They form the foundation of the entire traffic control structure. Think of a qdisc as the algorithm that governs how packets are stored and scheduled for transmission on a network interface. Every interface has at least one qdisc, which acts as the root of the traffic control tree. Qdiscs come in two primary varieties:
  - **Classless**: Classless qdiscs apply a single algorithm to all traffic without any internal subdivision, such as the simple **first-in, first-**

out (**FIFO**) or the more sophisticated **Token Bucket Filter** (**TBF**).

- **Classful**: Classful qdiscs categorize traffic into different classes based on certain attributes of the traffic, like services requiring low latency or high bandwidth, etc. Classful policies provide a container that can hold multiple classes and enable complex hierarchical policies between classes of traffic. Examples of classful qdiscs are **Hierarchical Token Bucket** (**HTB**) or **Priority Queuing** (**PRIO**),

  The qdisc represents the location and method by which packets are queued and scheduled. Within classful qdiscs, we find classes, which are the branches of the traffic control hierarchy.

- **Class:** It represents a specific channel or conduit for traffic, each with its own allocated resources and queuing behaviour. In practical terms, classes allow you to partition bandwidth and create virtual sub-interfaces within a single physical interface. For example, you might create one class for voice traffic with a guaranteed minimum bandwidth, another for business applications with a different bandwidth allocation, and a default class for all other traffic. Classes can be nested within parent classes, creating a true hierarchy that mirrors organizational priorities. They represent the specific traffic channels that carry different types of data.
- **Filters**: These are the rules that examine incoming packets and determine which class they should be directed to. The decision-making intelligence of the system resides in filters. Without filters, there would be no mechanism to sort traffic into the appropriate classes. Filters can match packets based on various criteria, including source and destination IP addresses, port numbers, protocol types, DSCP values, or firewall marks set by iptables or nftables. A filter is attached to either a qdisc or a class and instructs the system to take a specified action for packets matching certain criteria. Filters represent the methodology for classifying and routing different types of traffic.
- **Actions:** It defines what happens to packets as they pass through the traffic control system. While the most common action is to simply enqueue a packet into a specific class, the action system provides much

more sophisticated capabilities. Actions can include:

- **Policing**: Dropping packets that exceed a certain rate.
- **Shaping**: Delaying packets to smooth out bursts.
- **Modifying packet headers**: Such as remarking DSCP values for downstream processing.
- **Redirecting packets**: To different network interfaces.
- **Multiple actions**: Multiple actions can be chained together to create complex packet processing pipelines.

Together, these four components form a comprehensive framework for traffic management. The qdisc provides the structural foundation, classes create the organizational hierarchy, filters implement the classification logic, and actions execute the final operations. This modular architecture allows network administrators to build everything from simple rate limiters to complex, multi-level quality of service policies that can meet the diverse needs of modern network environments.

Differentiated services code point

DSCP enables networks to provide differentiated service levels without maintaining per-flow state, making it scalable for large networks while ensuring critical applications get the performance they need. The following are some of the benefits of using DSCP:

- **End-to-end signaling**: DSCP marks travel with packets across network boundaries.
- **Standardized**: Interoperable between different vendors' equipment.
- **Granular control**: 64 possible values for precise traffic classification.
- **Backward compatible**: Works with older TOS-based systems.

DSCP is a 6-bit field in the IP header that enables QoS classification and prioritization on IP networks. It is the modern replacement for the older **type of service** (**TOS**) field and provides a standardized way to mark packets for different treatment levels.

The following table describes the main DSCP forwarding classes:

Class name	DSCP value	Binary	Decimal	Purpose and use cases
Network control	CS7	111000	56	Network routing protocols **Open Shortest Path First** (**OSPF**), **Border Gateway Protocol** (**BGP**)
Internetwork control	CS6	110000	48	Inter-network control
Voice admit	VA	101100	44	Voice call signalling
Expedited forwarding	EF	101110	46	VoIP, video conferencing
Assured forwarding 4	AF41	100010	34	Interactive video, **Video Teleconferencing** (**VTC**)
Assured forwarding 3	AF31	011010	26	Business applications
Assured forwarding 2	AF21	010010	18	Streaming media
Assured forwarding 1	AF11	001010	10	Bulk data transfer
Best effort	CS0	000000	0	Default traffic, web browsing
Background	CS1	001000	8	File transfers, backups

Table 8.1*: DSCP codes*

DSCP marks can be set on an egress packet using iptables. The following are some examples of using **iptables** rules to mark the DSCP for packets:

```
iptables -t mangle -A POSTROUTING -p udp --dport 5060:5070 -j DSCP --set-dscp 46
iptables -t mangle -A POSTROUTING -p tcp --dport 22 -j DSCP --set-dscp 26

iptables -t mangle -A POSTROUTING -p tcp --dport 21 -j DSCP --set-dscp 8
```

These DSCP marks can now be used by traffic control to identify and queue

the packets according to the priority of the traffic.

First-in first-out qdiscs

At the most fundamental level of packet queuing lies the principle of FIFO management, a concept so intuitive that it serves as the default behavior for network interfaces across Linux systems. When traffic flows smoothly and bandwidth is plentiful, this simple approach operates transparently in the background. However, when congestion occurs and packets begin to accumulate, the specific FIFO algorithm employed becomes critically important to network performance and fairness.

The **pure FIFO** (**pfifo**) **qdisc** represents the simplest form of queue management. It operates exactly as its name suggests; packets are transmitted in the exact order they arrive, forming a single linear queue. When the interface is ready to send another packet, it always takes the one that has been waiting the longest. This approach is perfectly fair in a narrow sense; every packet gets its turn, but it makes no distinctions between types of traffic. A large, background file transfer packet receives the same priority as a tiny, latency-sensitive voice packet, provided it arrived first. The primary configuration parameter for **pfifo** is the limit or queue length, which determines how many packets can be buffered before new arrivals are dropped. While this prevents unlimited memory consumption, a very long FIFO queue can lead to bufferbloat, where excessive queuing delays introduce significant latency even when bandwidth is available. The following command adds or replaces a **pfifo** queuing discipline to interface with a limit of 100 packets:

```
tc qdisc add dev eth0 root pfifo limit 100
tc qdisc replace dev eth0 root pfifo limit 100
```

To view the statistics of the queuing discipline, use the following command:

```
tc -s qdisc show dev eth0
```

However, the true default qdisc on most Linux systems is not the simple **pfifo** but **pfifo_fast**, which introduces a crucial layer of intelligence while maintaining FIFO principles within priority bands. This **qdisc** implements

not one but three separate FIFO queues, numbered 0 through 2, with band 0 having the highest priority. The fundamental innovation lies in how packets are assigned to these bands. Rather than relying on complex filters, **pfifo_fast** uses the TOS field in the IP header, specifically, the precedence bits to determine a packet's priority band. This allows the network stack to provide basic quality of service without any administrative configuration. The following command configures a **pfifo** fast queuing discipline for the attached interface:

```
tc qdisc add dev eth0 root pfifo_fast limit 100
```

The scheduling behavior of **pfifo_fast** follows a strict priority scheme. The **qdisc** will always transmit all available packets from band 0 before even looking at band 1, and similarly exhausts band 1 before servicing band 2. This ensures that packets marked with higher precedence values receive immediate attention, making it possible for critical traffic to jump the queue. Within each band, however, pure FIFO ordering is maintained and hence the fast in the name, indicating it is a multi-band FIFO rather than a complex scheduler.

The significance of **pfifo_fast** extends beyond its technical operation. It represents the baseline against which all other queuing disciplines are measured and provides a good enough QoS for many environments without any configuration overhead. Its existence demonstrates that even simple priority mechanisms, when built into the fundamental networking stack, can provide meaningful traffic differentiation. However, when an administrator explicitly attaches any other **qdisc** to an interface, **pfifo_fast** is replaced, making way for more sophisticated traffic control policies. Understanding these fundamental FIFO approaches provides essential context for appreciating the advanced capabilities of classful **qdiscs** and the deliberate management of network backlog.

Rate limiting with token bucket filters

TBF represents one of the most straightforward and intuitive approaches to traffic rate limiting within the Linux traffic control system. As a classless

queuing discipline, TBF operates on a simple but powerful principle that makes it ideal for implementing basic bandwidth constraints and preventing any single traffic flow from monopolizing a network interface. Its elegance lies in its implementation mechanism, which closely mirrors the economics of resource consumption.

The TBF algorithm operates around the concept of tokens and a bucket. Imagine a bucket that has a fixed capacity and is continuously being filled with tokens at a specific, constant rate, the token rate. Each token represents permission to send a certain amount of data, typically one byte. Before a packet can be transmitted, it must account for itself by acquiring tokens equal to its size in bytes. If sufficient tokens are available in the bucket, the packet is immediately sent, and the corresponding tokens are removed. If there are insufficient tokens, the packet must wait until enough tokens accumulate in the bucket. This mechanism directly enforces a long-term average transmission rate equal to the token fill rate. The following figure shows the Token Bucket Filter in action; it describes how the traffic burst rate is handled using the bucket capacity:

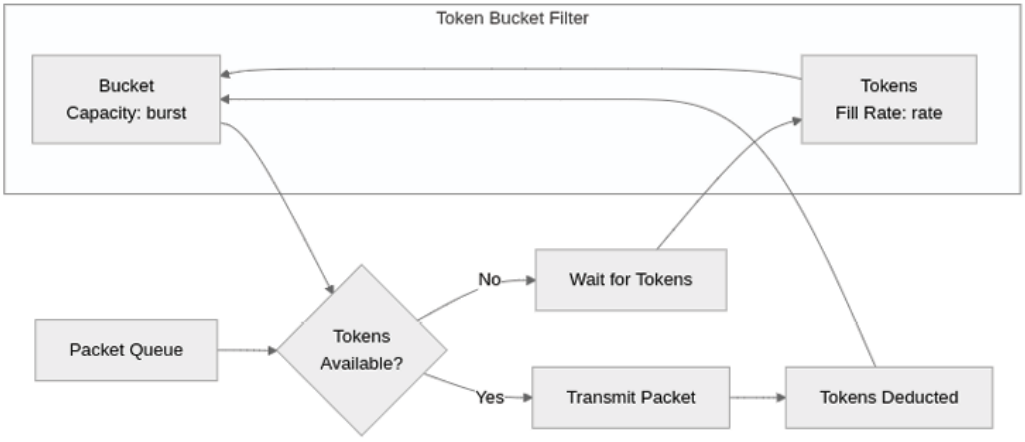


***Figure 8.2**: Token Bucket Filter*

However, real-world network traffic is often bursty, and the TBF accommodates this through the bucket size parameter. The bucket depth determines the maximum number of tokens that can be stored, which in turn defines the maximum burst size that can be handled without delay. A larger bucket allows for greater bursts of traffic, as multiple packets can be sent

back-to-back until the bucket is emptied, after which packets must wait for the bucket to refill at the steady token rate. The bucket size determines the maximum size of burst that can be accommodated before introducing additional latency. A larger bucket can smooth out bursts, but inherently increases the delay data experiences while waiting in line. This two-parameter system rate and burst provides a flexible way to control both the average bandwidth and the short-term traffic peaks.

The practical implementation of TBF using the **tc** command is remarkably concise. To apply a basic rate limit of 1 megabit per second with a standard burst capacity to a network interface **eth0**, the command would be:

```
tc qdisc add dev eth0 root tbf rate 1mbit burst 32kbit latency 400ms
```

This command attaches a TBF **qdisc** to the root of the **eth0** interface. The rate parameter specifies the sustained bandwidth limit, while the burst parameter defines the maximum burst capacity in bytes. The latency parameter determines how long a packet is willing to wait for tokens to become available before it might be dropped, providing an important control mechanism for managing bufferbloat. We will discuss bufferbloat and its mitigation in a later section in the chapter.

The TBF's strength lies in its simplicity and predictability. It is particularly well-suited for scenarios that require straightforward rate limiting without complex prioritization schemes, such as enforcing upload limits on customer connections, preventing backup jobs from saturating WAN links, or ensuring that non-critical services do not consume more than their allocated share of bandwidth. While it lacks the hierarchical capabilities of classful **qdiscs**, the Token Bucket Filter remains an essential tool in traffic control and acts as a fundamental building block for anyone beginning their journey into network traffic management.

Hierarchical tree for bandwidth allocation

When the requirements for traffic management evolve beyond simple rate limiting to encompass complex organizational policies, the HTB emerges as the essential tool for building sophisticated bandwidth allocation structures.

Unlike classless **qdiscs** that apply a single policy to all traffic, HTB enables the creation of a true hierarchy with a tree-like structure that mirrors organizational needs, allowing administrators to divide bandwidth into logical channels and sub-channels with precise guarantees and limits.

The power of HTB lies in its ability to implement both minimum guarantees and maximum constraints in a parent-child relationship model. At the root of the hierarchy sits the main HTB **qdisc**, which represents the total bandwidth available on the interface. From this root, we create parent classes that define major traffic categories, for instance, business applications, user internet, and network management. Each of these parent classes can then be subdivided into child classes, creating a granular structure that ensures fair allocation while maintaining organizational boundaries.

The HTB algorithm operates through two fundamental parameters of rate and ceil. The rate represents the guaranteed minimum bandwidth that a class will receive under any conditions; this is the service level agreement for that traffic category. The ceil defines the maximum bandwidth the class can borrow if other classes are not using their allocated capacity. This borrowing mechanism is what makes HTB exceptionally efficient; unused bandwidth is not wasted but rather redistributed to classes that need extra capacity, up to their respective ceiling limits.

Consider a practical implementation for a company's WAN link:

```
tc qdisc add dev eth0 root handle 1: htb default 20

tc class add dev eth0 parent 1: classid 1:1 htb rate 100mbit ceil 100mbit
tc class add dev eth0 parent 1:1 classid 1:10 htb rate 30mbit ceil 60mbit
tc class add dev eth0 parent 1:1 classid 1:20 htb rate 50mbit ceil 80mbit
tc class add dev eth0 parent 1:1 classid 1:30 htb rate 20mbit ceil 40mbit
```

The following figure shows the traffic classification using the hierarchical

Token Bucket Filter:

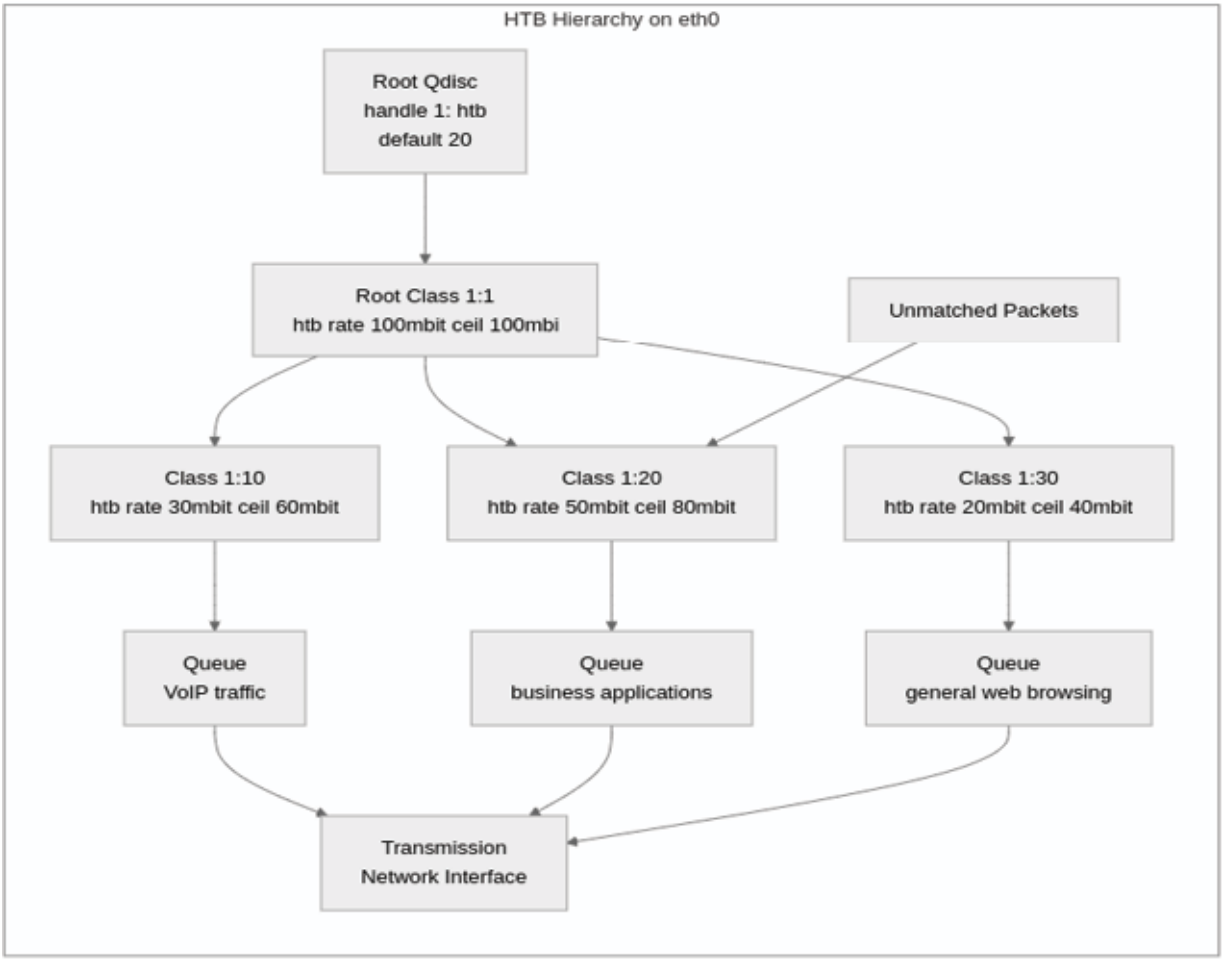


***Figure 8.3**: Hierarchical token bucket*

In this structure, the root class 1:1 represents the entire 100 Mbps link. Three child classes are created: 1:10 for VoIP traffic with a guaranteed 30 Mbps, 1:20 for business applications with 50 Mbps guaranteed, and 1:30 for general web browsing with 20 Mbps guaranteed. The ceiling values allow each class to borrow unused bandwidth. VoIP can burst to 60 Mbps if available, business apps to 80 Mbps, and web browsing to 40 Mbps.

The hierarchy can extend further by creating grandchild classes. The business applications class (1:20) might be subdivided to guarantee specific bandwidth to database replication, file sharing, and email services, each with its own rate and ceiling parameters. This multi-level approach enables extremely precise traffic engineering that reflects business priorities.

What makes HTB particularly valuable is its intuitive behavior under congestion. When all classes are demanding their guaranteed rates, each receives exactly its allocated minimum. When some classes are idle, active classes can grow to their ceiling values. And when previously idle classes become active again, HTB gracefully redistributes bandwidth back to the guaranteed minimums without sudden disruptions. This predictable, hierarchical control makes HTB the foundation for most enterprise-quality traffic shaping policies, providing the structural framework needed to translate business requirements into technical implementation.

Stochastic fairness queuing

Stochastic fairness queuing (**SFQ**) with perturb is a queuing discipline in Linux traffic control that provides fair bandwidth distribution among multiple traffic flows. *SFQ* is a classless qdisc that ensures fair queuing by separating traffic into multiple flows. It follows round-robin scheduling between flows and prevents any single flow from monopolizing bandwidth. The perturb value (in seconds) controls how often SFQ rehashes its flow classification. Following is an example of using SQL **qdisc**:

```
tc qdisc add dev eth0 parent 1:10 handle 110: sfq perturb 10
```

SFQ uses a hash function to identify traffic flows (typically based on source/destination IP+port), Every perturb seconds, SFQ changes its hash function and stops malicious flows from *learning* their hash bucket and dominating it. Without perturb, aggressive flows could determine their hash bucket and could send traffic optimized for that specific hash. This could defeat the fairness mechanism. The following example shows the use of SQF with HTB, with SQF applied as a child of the HTB classes:

```
# HTB with SFQ leaves for internal fairness
tc qdisc add dev eth0 root handle 1: htb default 30
tc class add dev eth0 parent 1: classid 1:10 htb rate 50mbit ceil 80mbit
tc class add dev eth0 parent 1: classid 1:20 htb rate 30mbit ceil 50mbit

# Add SFQ to each class for fair distribution within the class
```

```
tc qdisc add dev eth0 parent 1:10 handle 110: sfq perturb 10
tc qdisc add dev eth0 parent 1:20 handle 120: sfq perturb 10
```

The following figure shows the traffic classification using the SFQ for fair distribution of bandwidth:

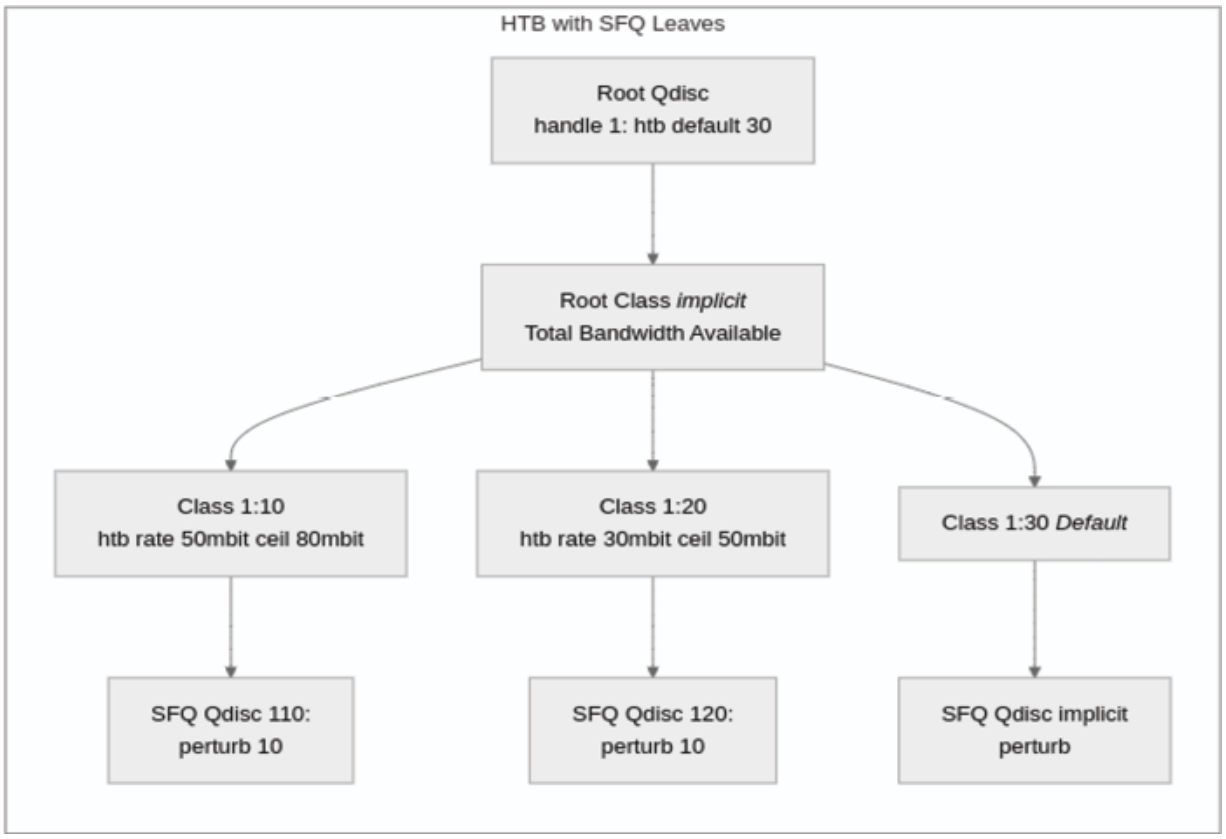


***Figure 8.4**: Stochastic fairness queuing*

A perturb value of 10-30 seconds is typically sufficiently long to avoid excessive overhead, but short enough to prevent manipulation.

Prioritizing traffic flows with priority queuing

There are many network scenarios where strict prioritization, rather than bandwidth guarantees, is the primary requirement. For these situations, the PRIO **qdisc** offers a straightforward yet powerful mechanism to ensure that critical traffic receives immediate forwarding while less important traffic

waits its turn. The PRIO **qdisc** operates on a simple but effective principle, which is to always service higher-priority queues first, making it ideal for environments where latency sensitivity varies dramatically between application types.

The architecture of PRIO is built around multiple bands, typically three by default, though configurable to meet specific needs. Each band represents a priority level, with Band 0 having the highest priority and subsequent bands receiving service only when all higher-priority bands are empty. This strict prioritization means that even a small amount of high-priority traffic can completely starve lower-priority bands if left unchecked, which is both the strength and the potential pitfall of this approach.

What distinguishes PRIO from the default **pfifo_fast** is the administrator's ability to control how traffic is classified into these priority bands explicitly. Rather than relying on the IP header's TOS bits, PRIO uses the same powerful filter mechanism as other classful **qdiscs**, allowing classification based on any combination of source/destination IP addresses, port numbers, protocol types, or firewall marks. This gives the network administrator precise control over what constitutes priority traffic in their specific environment.

A typical PRIO configuration might look like this:

```
tc qdisc add dev eth0 root handle 1: prio bands 3
tc qdisc add dev eth0 parent 1:1 handle 10: pfifo limit 100
tc qdisc add dev eth0 parent 1:2 handle 20: pfifo limit 500
tc qdisc add dev eth0 parent 1:3 handle 30: pfifo limit 1000


tc filter add dev eth0 protocol ip parent 1: prio 1 u32 match ip dport 5060 0xffff flowid 1:1
tc filter add dev eth0 protocol ip parent 1: prio 2 u32 match ip dport 22 0xffff flowid 1:1
tc filter add dev eth0 protocol ip parent 1: prio 3 u32 match ip sport 80 0xffff flowid 1:2
```

In this example, **Session Initiation Protocol** (**SIP**) traffic (port 5060) and **Secure Shell** (**SSH**) sessions (port 22) are directed to the highest priority

band (1:1), ensuring minimal latency for these interactive applications. Web traffic (port 80) is assigned to the middle band (1:2), while all other traffic defaults to the lowest priority band (1:3). Each band contains its own queuing discipline; in this case, simple FIFO queues with different buffer sizes that reflect the relative importance and expected volume of each traffic type.

The true value of PRIO becomes apparent in mixed-workload environments. Voice and video conferencing packets can be given the highest priority to maintain call quality, while bulk file transfers and software updates can be relegated to lower bands where they would not impact time-sensitive applications. This ensures that a large backup job or system update would not degrade the user experience for interactive applications.

However, the simplicity of strict prioritization requires careful design. Without additional rate limiting, high-priority traffic can completely monopolize the link, making PRIO often used in combination with other **qdiscs**. A common pattern places PRIO as the root **qdisc** for prioritization, with HTB or TBF **qdiscs** attached to each band to prevent any single priority level from consuming all available bandwidth. This hybrid approach delivers both the latency benefits of strict prioritization and the fairness of bandwidth allocation, making PRIO an essential component in the traffic control toolkit for managing diverse application requirements.

Classifying packets into streams

The sophisticated traffic management structures built with **qdiscs** and classes remain theoretical constructs without the intelligent machinery to direct individual packets to their appropriate destinations within the hierarchy. Filters and classifiers perform this critical routing function, the decision-making engine of the Linux traffic control system that examines each packet and determines its fate within the configured policy. Classification is the process that breathes life into the traffic control structure, transforming a static tree of classes into a dynamic system that can distinguish between and manage diverse traffic types.

Filters operate as the gatekeepers attached to **qdiscs** or classes, each containing a set of conditions that packets must satisfy and an action to execute when a match occurs. The most common action is to direct the packet to a specific class ID within the hierarchy, but filters can also trigger more complex operations like dropping, remarking, or redirecting packets. The true power of the classification system lies in its flexibility; multiple filters can be applied to a single point in the tree, and packets can be evaluated against these filters in sequence until a match is found or a default action is taken.

The **tc** command provides several classification methods, each suited to different scenarios. The **u32** classifier is one of the most powerful and commonly used, allowing direct filtering based on any field in the IP header using offset-based matching. For example, to classify SSH traffic into a high-priority class, one might use:

```
tc filter add dev eth0 parent 1: protocol ip u32 match ip dport 22 0xffff flowid 1:10
```

This filter examines the destination port field in the TCP header and directs matching packets to class 1:10. The **u32** classifier's strength is its precision; it can match on any combination of source/destination addresses, protocol types, port ranges, DSCP values, or even specific bits within packet headers.

For more complex matching scenarios, the **fw** classifier provides integration with Linux's netfilter framework, allowing classification based on firewall marks set by iptables or nftables. This approach separates the complex matching logic, which can include stateful inspection, connection tracking, and sophisticated pattern matching, from the traffic control system itself.

A typical workflow might involve:

```
iptables -t mangle -A FORWARD -p tcp --dport 80 -j MARK --set-mark 10
tc filter add dev eth0 parent 1: protocol ip handle 10 fw flowid 1:20
```

Here, iptables marks all HTTP traffic with mark 10, and the **fw** classifier simply routes marked packets to the appropriate class based on this pre-determined value.

For modern applications, the flower classifier offers a more intuitive and expressive syntax, supporting complex matches on multiple header fields in a single statement. For instance:

```
tc filter add dev eth0 parent 1: protocol ip flower ip_proto tcp dst_port 443 skb_priority 5 action skbedit priority 6 flowid 1:30
```

This demonstrates the flower's ability to match TCP traffic on port 443 while also modifying packet metadata; a capability essential for complex QoS policies.

The classification process typically follows a waterfall model, i.e., packets are evaluated against filters in order of their priority, and the first matching filter determines the packet's path. This allows for specific exceptions to be processed before general rules. When no filters match, packets are directed to the default class, making proper default configuration a critical aspect of any traffic control policy.

Effective packet classification transforms abstract bandwidth allocations into operational reality. It is the mechanism that ensures VoIP packets find their way to the low-latency voice class, that business applications receive their guaranteed bandwidth, and that non-essential traffic is properly constrained.

Active queue management

Bufferbloat is a phenomenon in packet-switched networks where excessively large buffers in network equipment (routers, switches, modems) cause high latency and jitter, even when the network link is not fully saturated. Modern networking equipment often comes with very large buffers, sometimes holding seconds' worth of data. Packets enter these large buffers and wait their turn for transmission. During congestion, packets accumulate in these deep buffers, causing **queue buildup**. Even though packets are not being dropped, they experience long queuing delays. Bufferbloat creates a counterintuitive situation as packets are not being dropped, the throughput appears normal, packets wait in queues for seconds, causing high Latency, congestion control fails as no packets are

dropped, and senders do not slow down.

The following figure shows the performance of the networking device based on the buffer size. A smaller buffer can cause unintended packet drops and require retransmission, while a large buffer causes queueing delay:

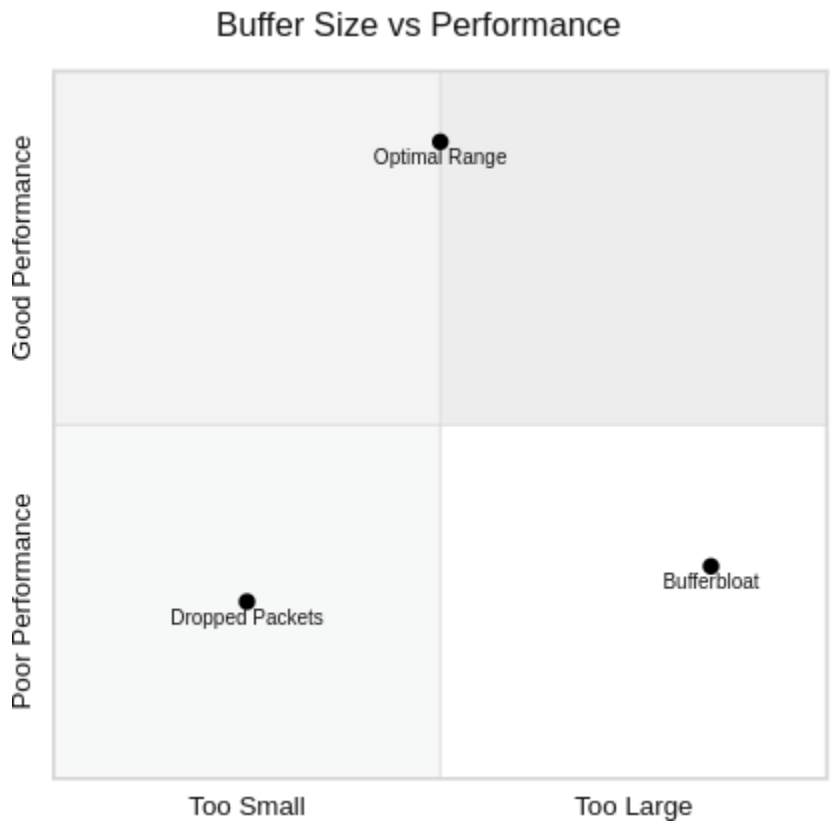


***Figure 8.5**: Buffer bloat in queues*

Bufferbloat can be managed by using **Active Queue Management** (**AQM**) to control latency. **qdisc fq_codel** actively manages queue lengths to control latency and combines fair queuing with delay management:

```
tc qdisc add dev eth0 root fq_codel
tc qdisc add dev eth0 root fq_codel limit 1000 target 5ms interval 100ms
```

Standard traffic shaping without AQM can worsen bufferbloat:

- **HTB with large buffers**: Creates the same problem at a different layer.
- **PRIO without limits**: High-priority traffic still waits behind buffered

low-priority packets.

- **Simple rate limiting**: Does not address the fundamental queuing problem.

FQ-CoDel combines two techniques of **flow queuing** (**FQ**) and **controlled delay** (**CoDel**) to comprehensively address bufferbloat. The details are as follows:

- **FQ**: Flow isolation separates traffic into 1024 individual sub-queues based on source/destination IP and port. Fair scheduling uses deficit round robin to give each flow equal opportunity. New flow protection rate-limits new flows to prevent them from overwhelming existing ones.
- **CoDel**: Per-Flow AQM applies the CoDel algorithm independently to each flow queue. It uses delay monitoring to track queuing delay and only acts when the delay exceeds a certain target, and drops packets only from flows causing excessive delay.

The following figure describes the working of fair queuing with CoDel algorithm:

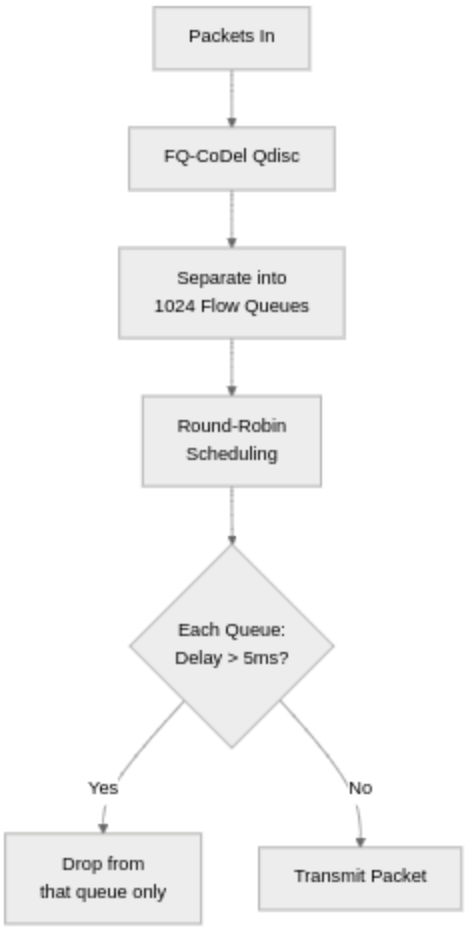


***Figure 8.6**: Active queue management with FQ-CoDel*

FQ-CoDel is particularly effective because it does not just manage the aggregate queue—it provides intelligent per-flow management that naturally enforces fairness and maintains low latency across all applications without requiring complex policy configuration.

Policing traffic to enforce a strict rate limit

While traffic shaping focuses on smoothing and delaying packets to manage congestion, policing operates on a fundamentally different principle—immediate enforcement of strict rate limits with zero tolerance for excess. A traffic policer acts as a gatekeeper at the entrance to a network domain, measuring the rate of incoming traffic streams and taking decisive action the moment they exceed established boundaries. This approach provides the

sharp teeth behind service level agreements and security policies, ensuring that no traffic flow can exceed its allocated share under any circumstances.

The mechanics of policing typically employ token bucket algorithms like those used in shaping, but with a critical behavioral difference. Where a shaper buffers excess packets to transmit them later when tokens become available, a policer simply discards them or marks them as non-compliant. This creates a hard boundary that cannot be crossed, making policing essential for scenarios where absolute rate limits must be maintained, such as enforcing contractual agreements with customers or preventing denial-of-service attacks from overwhelming network resources.

The **tc** framework implements policing through filter actions, allowing administrators to attach rate enforcement to specific traffic classes. A basic policer configuration might appear as:

```
tc filter add dev eth0 parent 1: protocol ip u32 match ip src 192.168.1.100 police rate 1mbit burst 64k drop
```

This command creates a policer that monitors traffic from IP address 192.168.1.100, allowing an average rate of 1 megabit per second with a burst tolerance of 64 kilobits. The critical component is the drop action, which dictates that any packets exceeding this limit will be immediately discarded. This hard enforcement ensures the specified host cannot consume more than its allocated bandwidth, regardless of network conditions.

However, policing strategies have evolved beyond simple drop mechanisms. Modern implementations often employ a two-rate, three-color marker system that provides more nuanced handling of traffic violations. This approach classifies packets as green (within contract), yellow (exceeding committed information rate but within peak rate), or red (exceeding all limits). Each colour can trigger different actions:

```
tc filter add dev eth0 parent 1: protocol ip u32 match ip dport 80 0xffff police rate 5mbit burst 100k peakrate 10mbit mtu 1514 conform-exceed drop/violate drop
```

In this sophisticated example, HTTP traffic is policed with a committed rate of 5 Mbps and a peak rate of 10 Mbps. Packets within the committed rate

are treated as *conform* and transmitted normally. Those exceeding the committed rate but within the peak rate are *exceed* and might be marked with lower priority rather than dropped. Only packets violating the peak rate face immediate discard.

The choice between policing and shaping represents a trade-off in network management. Policing protects network resources from abuse and ensures strict adherence to policies, but does so at the cost of potential packet loss and TCP retransmissions. Shaping provides smoother, more predictable throughput but requires buffering that introduces latency. In practice, many networks employ both strategies: policing at network boundaries to enforce absolute limits and shaping within the network to manage congestion and provide QoS. This dual approach ensures both compliance with policies and efficient utilization of available bandwidth, with policing serving as the essential enforcement mechanism that maintains the integrity of the overall traffic management system.

Applying traffic shaping to ensure predictable rates

Where policing serves as the strict enforcer of rate limits, traffic shaping operates as the thoughtful regulator that transforms erratic, bursty traffic patterns into smooth, predictable flows. The fundamental purpose of shaping is not merely to limit the overall transmission rate but to actively manage the timing of packet departure, introducing deliberate delays to prevent congestion and create consistent, manageable traffic streams. This approach recognizes that while network traffic is inherently bursty, network links perform best with steady, predictable loads.

The mechanics of traffic shaping revolve around the controlled use of buffering and scheduling. When a shaper detects that traffic is arriving faster than the permitted rate, it does not discard the excess packets as a policer would. Instead, it holds them in a queue and releases them at the configured rate, effectively spreading a sudden burst over a longer time period. This smoothing process eliminates the peaks and valleys in network

utilization that can overwhelm downstream equipment and cause packet loss, replacing them with a consistent throughput.

The TBF provides a classic example of simple traffic shaping in action. As discussed earlier, TBF uses a token system where packets can only be transmitted when sufficient tokens are available. When a burst occurs and depletes the token bucket, subsequent packets must wait for tokens to replenish at the steady rate. This creates exactly the smoothing behavior that defines traffic shaping; the burst is not eliminated but rather stretched out to fit within the configured rate limit. The burst parameter in TBF specifically controls how much bursting is allowed before the smoothing mechanism engages.

More sophisticated shaping emerges with classful qdiscs like HTB. While HTB is renowned for its bandwidth allocation capabilities, it also provides advanced shaping through its rate and ceiling parameters. The guaranteed rate ensures minimum service levels, while the ceil controls how much bursting is permitted when excess bandwidth is available. More importantly, HTB's hierarchical nature allows for multi-level shaping—a parent class can shape the aggregate of its child classes, while each child class shapes its own specific traffic type. This enables administrators to create complex shaping policies where, for example, the total VoIP traffic is shaped to 10 Mbps while individual VoIP streams within that aggregate are also shaped to prevent any single stream from dominating.

The practical benefits of traffic shaping become most apparent in scenarios with known bandwidth constraints. Consider a business with a 50 Mbps internet connection that needs to support VoIP, video conferencing, and bulk data transfers. Without shaping, a large file download could instantly consume all available bandwidth, causing jitter and latency in real-time communications. With shaping implemented, the file transfer is smoothed to use no more than 30 Mbps, ensuring that 20 Mbps remains consistently available for time-sensitive applications. The file still transfers, but it takes slightly longer in exchange for preserving voice and video quality.

The configuration of a shaper involves careful consideration of the latency-buffer tradeoff. Larger buffers allow for greater burst absorption and more efficient link utilization but introduce higher latency, a phenomenon known

as bufferbloat. Smaller buffers minimize latency but may lead to underutilization of the link, as traffic cannot burst to fill momentary gaps. The art of traffic shaping lies in finding the optimal balance for each specific application environment.

In essence, traffic shaping is the practice of teaching impatient traffic to be patient. It acknowledges that while bursts are natural, sustained predictability is preferable for overall network health. By smoothing the rough edges off traffic patterns, shaping creates a more stable, predictable network environment where applications can perform consistently, and administrators can plan capacity with greater confidence. It represents the civilizing influence in the often-chaotic world of network traffic management.

Ingress and egress policing and shaping

The direction of traffic flow relative to a network interface fundamentally changes the approach and implementation of traffic control measures. While egress (outgoing) traffic management has been the traditional focus of QoS mechanisms, ingress (incoming) traffic control presents distinct challenges and requires different strategies. Understanding this directional distinction is crucial for implementing comprehensive traffic policies that effectively manage both traffic entering and leaving a network node.

Egress traffic control operates on packets as they leave a network interface, and this is where traditional shaping and policing mechanisms are applied. When we attach a **qdisc** to an interface using commands like:

```
tc qdisc add dev eth0 root htb
```

We are implementing egress control. The Linux kernel has complete authority over when packets are transmitted from the interface, allowing it to delay packets in queues (shaping) or discard them entirely (policing). This control is direct and deterministic; the kernel decides exactly when each packet leaves the interface, making egress the ideal domain for implementing sophisticated scheduling algorithms like HTB and PRIO. Most traffic management policies naturally focus on egress control because

we have absolute control over what we send out and when we send it.

Ingress traffic control, by contrast, deals with packets arriving at an interface, a scenario where our control is inherently limited. We cannot shape incoming traffic in the traditional sense because we cannot tell remote hosts to slow down their transmission; we can only decide how to handle the packets once they arrive. The **tc** system provides an ingress **qdisc** specifically for this purpose, but its capabilities are fundamentally different from egress qdiscs. Ingress control primarily focuses on policing by measuring the rate of incoming traffic and dropping excess packets when they exceed established limits. This makes ingress policing invaluable for protecting against flood attacks and ensuring that incoming traffic complies with contractual agreements.

The practical implementation highlights these differences. For egress shaping, we might use:

```
tc qdisc add dev eth0 root handle 1: htb default 10
tc class add dev eth0 parent 1: classid 1:1 htb rate 10mbit
```

For ingress policing, the approach is necessarily different:

```
tc qdisc add dev eth0 handle ffff: ingress
tc filter add dev eth0 parent ffff: protocol ip u32 match u32 0 0 police rate 5mbit burst 64k drop
```

The special handle **ffff:** identifies the ingress, and the policer acts as a gatekeeper for incoming packets. This configuration would drop any incoming traffic that exceeds 5 Mbps, protecting the system from being overwhelmed by external sources.

The strategic implications of this directional distinction are significant. Egress control is typically used for optimizing outbound traffic quality, ensuring that voice packets get priority, that no single user monopolizes upload bandwidth, and that traffic conforms to upstream service agreements. Ingress control, meanwhile, serves protective functions: preventing denial-of-service attacks, ensuring that incoming traffic does not overwhelm processing capabilities, and enforcing incoming rate limits for peer connections.

In comprehensive network designs, both ingress and egress controls work in

concert. A network edge router might employ ingress policing to protect against external floods while using egress shaping to ensure fair distribution of outgoing bandwidth among internal users. This bidirectional approach recognizes that traffic management is not merely about controlling what we send, but also about managing what we are willing to receive and process. The combination creates a complete traffic management system that addresses both the traffic a node originates and the traffic it must consume.

Integrating traffic control with netfilter

The true power of Linux traffic control emerges when it collaborates with other subsystems, like netfilter, the framework that underpins iptables and nftables. This integration creates a sophisticated division of labour where netfilter handles complex packet classification and marking, while **tc** focuses on the specialized task of traffic shaping and queuing based on those marks. This separation of concerns allows administrators to leverage the strengths of both systems for advanced traffic management scenarios.

Netfilter operates at the packet filtering level, providing extensive capabilities for stateful inspection, connection tracking, and complex pattern matching that far exceed the classification abilities of **tc** filters alone. Where **tc**'s **u32** classifier can examine basic header fields, netfilter can make decisions based on entire connection states, application-layer content, and sophisticated rule sets that consider the context of network conversations rather than just individual packets. This makes netfilter ideally suited for the initial classification work, leaving **tc** to perform the queuing disciplines it excels at executing. The following figure shows how **tc** can integrate with iptables to take advantage of the extensive traffic classification capability of IPTable rules:

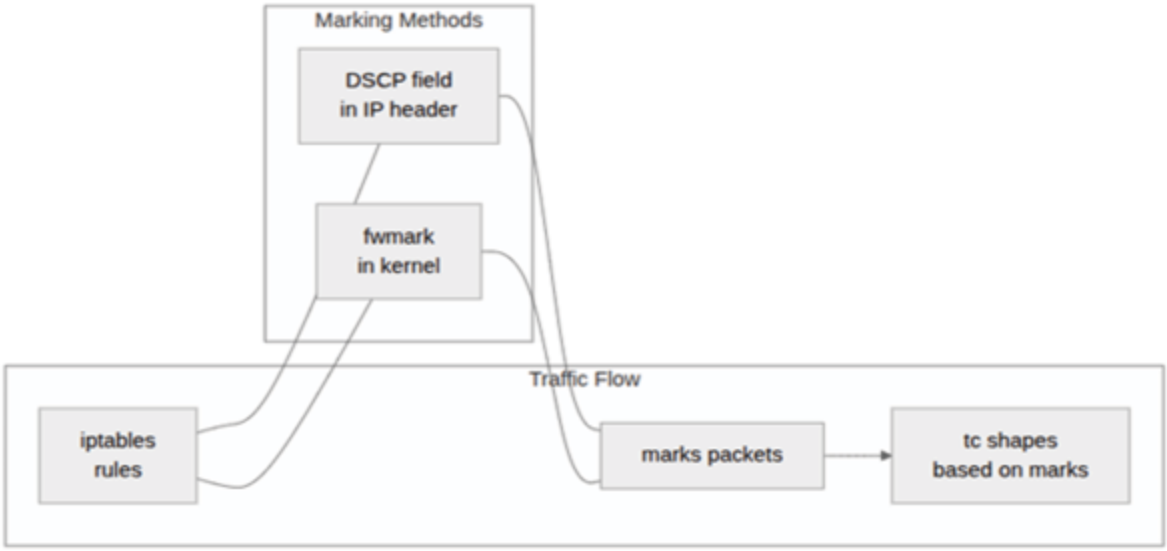


***Figure 8.7**: Integration of iptables and TC*

The integration mechanism centers on packet marking, netfilter tags packets with metadata that **tc** can later recognize and act upon. The workflow typically follows a clear pattern: first, netfilter rules in the mangle table examine packets and set marks using the MARK target or set **DiffServ Code Points** (**DSCP**) using the DSCP target. Then, **tc** filters match these marked packets using the **fw** classifier or DSCP matching and direct them to appropriate classes within the traffic control hierarchy.

Consider a practical example where we prioritize VoIP traffic while limiting peer-to-peer applications:

- Mark packets using **iptables** in the mangle table:

  iptables -t mangle -A FORWARD -p udp --dport 5060:5070 -j MARK --set-mark 10

  iptables -t mangle -A FORWARD -m layer7 --l7proto bittorrent -j MARK --set-mark 30

- Classify marked packets using **tc**:

  tc filter add dev eth0 parent 1: protocol ip handle 10 fw flowid 1:10

  tc filter add dev eth0 parent 1: protocol ip handle 30 fw flowid 1:30

In this configuration, the complex task of identifying SIP traffic and

BitTorrent protocols is handled by netfilter, while **tc** focuses on its core competency, ensuring that marked packets receive the appropriate queuing treatment.

The integration becomes particularly powerful when working with DiffServ Code Points, as this allows end-to-end quality of service signalling:

```
iptables -t mangle -A FORWARD -p tcp --sport 22 -j DSCP --set-dscp-class ef
tc filter add dev eth0 parent 1: protocol ip match ip dsfield 0xb8 0xfc flowid 1:10
```

Here, SSH traffic is marked with the **Expedited Forwarding** (**EF**) DSCP value, which **tc** then recognizes and directs to a low-latency class. This approach maintains QoS markings that can be honoured throughout the network path.

For stateful applications, netfilter's connection tracking capabilities enable sophisticated traffic classification that would be impossible with **tc** alone:

```
iptables -t mangle -A FORWARD -m connbytes --connbytes 1048576: --connbytes-dir both --connbytes-mode bytes -j MARK --set-mark 40
```

This rule marks all packets belonging to connections that have transferred more than 1MB of data, allowing **tc** to identify and potentially rate-limit long-lived flows, a common requirement for fair bandwidth sharing.

The synergy between netfilter and **tc** represents the maturity of Linux's traffic management capabilities. By allowing each subsystem to focus on what it does best, administrators can build incredibly sophisticated traffic policies that combine the packet inspection intelligence of netfilter with the precise traffic engineering capabilities of **tc**. This integration transforms Linux from a simple packet forwarder into a fully featured traffic management platform capable of implementing enterprise-grade quality of service policies.

Deploying a complete and practical traffic shaping policy

Crafting a coherent policy requires moving beyond theoretical understanding to practical implementation, transforming isolated **tc** commands into a structured solution that balances performance, fairness, and administrative overhead. A well-designed policy serves as the operational foundation of an organization's network priorities, ensuring that critical applications receive the resources they need while preventing any single user or service from degrading the overall network experience.

The development process begins with a thorough assessment of network requirements and traffic patterns. This foundational analysis identifies the key applications, their performance sensitivities, and their typical bandwidth consumption. For instance, a business might determine that VoIP requires minimal bandwidth but absolute priority for low latency, video conferencing needs moderate bandwidth with stable jitter characteristics, while file transfers and software updates can tolerate variable performance but should not monopolize available capacity. This assessment directly informs the policy structure, determining how many traffic classes are needed and what resources each should receive.

Consider a practical policy for a company with a 100 Mbps internet connection, implemented using HTB for hierarchical control:

- Create root HTB qdisc with default class:

  ```
  tc qdisc add dev eth0 root handle 1: htb default 60
  ```

- Root class matching link capacity:

  ```
  tc class add dev eth0 parent 1: classid 1:1 htb rate 100mbit ceil 100mbit
  ```

- Create traffic classes with appropriate allocations:

  ```
  # VoIP - highest priority
  tc class add dev eth0 parent 1:1 classid 1:10 htb rate 2mbit ceil 5mbit
  # Video conferencing
  tc class add dev eth0 parent 1:1 classid 1:20 htb rate 20mbit ceil 40mbit
  # Business applications
  tc class add dev eth0 parent 1:1 classid 1:30 htb rate 50mbit ceil
  ```

```
80mbit
# Web browsing
tc class add dev eth0 parent 1:1 classid 1:40 htb rate 10mbit ceil 30mbit
# Email and chat
tc class add dev eth0 parent 1:1 classid 1:50 htb rate 5mbit ceil 15mbit
# Default and miscellaneous
tc class add dev eth0 parent 1:1 classid 1:60 htb rate 13mbit ceil 25mbit
```

- Add appropriate queuing disciplines to each leaf class:

```
tc qdisc add dev eth0 parent 1:10 handle 110: pfifo limit 50
tc qdisc add dev eth0 parent 1:20 handle 120: pfifo limit 100
tc qdisc add dev eth0 parent 1:30 handle 130: sfq perturb 10
tc qdisc add dev eth0 parent 1:40 handle 140: sfq perturb 10
tc qdisc add dev eth0 parent 1:50 handle 150: sfq perturb 10
tc qdisc add dev eth0 parent 1:60 handle 160: sfq perturb 10
```

The classification phase then brings this structure to life, using a combination of netfilter marking and direct tc filtering:

- Mark packets using iptables (or nftables) for complex classification:

```
iptables -t mangle -A OUTPUT -p udp --dport 5060:5070 -j MARK --set-mark 10
iptables -t mangle -A OUTPUT -p udp --dport 16384:32768 -j MARK --set-mark 20
iptables -t mangle -A OUTPUT -p tcp --dport 443 -j MARK --set-mark 30
iptables -t mangle -A OUTPUT -p tcp --dport 80 -j MARK --set-mark 40
iptables -t mangle -A OUTPUT -p tcp --dport 25,587,993,995 -j MARK --set-mark 50
```

- Direct marked traffic to appropriate classes:

```
tc filter add dev eth0 parent 1: protocol ip handle 10 fw flowid 1:10
tc filter add dev eth0 parent 1: protocol ip handle 20 fw flowid 1:20
tc filter add dev eth0 parent 1: protocol ip handle 30 fw flowid 1:30
tc filter add dev eth0 parent 1: protocol ip handle 40 fw flowid 1:40
tc filter add dev eth0 parent 1: protocol ip handle 50 fw flowid 1:50
```

Deployment requires careful planning to minimize disruption. A phased approach often works best, beginning with monitoring mode to validate classification accuracy, followed by conservative rate limits that are gradually tightened based on observed performance. Comprehensive testing should verify that the policy achieves its goals without introducing unexpected side effects, such as bufferbloat or starvation of certain traffic types.

Monitoring and maintenance form the final, ongoing phase of policy management. The **tc** command's **-s** option provides vital statistics about queue lengths, packet drops, and bandwidth utilization:

```
tc -s qdisc show dev eth0
tc -s class show dev eth0
```

These metrics reveal whether the policy is functioning as intended or requires adjustment. Periodic reviews ensure the policy evolves alongside changing network usage patterns and business requirements.

A successful traffic shaping policy represents the artful application of traffic control science. It balances precise technical controls with practical operational considerations, creating a network environment where business objectives dictate traffic treatment rather than leaving application performance to the vagaries of contention and coincidence. Through careful design, implementation, and maintenance, traffic shaping transforms the network from a mere connectivity platform into a strategic asset that actively supports organizational goals.

Conclusion

This chapter has illuminated how the abstract goals of QoS find concrete expression through the mechanisms of traffic control. We recognized that bandwidth is a finite and contended respurce. We have seen how qdiscs provide the structural foundation, classes enable hierarchical organization, filters implement intelligent classification, and actions execute precise operations on packet flows. From the straightforward rate limiting of TBF to the sophisticated hierarchy of HTB and the strict prioritization of PRIO, each tool serves a distinct purpose. The integration with netfilter demonstrates how Linux's networking subsystems collaborate to provide capabilities that exceed what any single component could achieve alone.

Traffic shaping and QoS policies provide crucial control over how packets are scheduled and transmitted from a Linux system. These mechanisms operate on the outward flow of data, managing contention and priority at the egress point, but they do not address the fundamental behaviors of the protocols generating that traffic in the first place. The TCP, the reliable workhorse of Internet communication, comes with a default configuration designed for universal compatibility across diverse and often constrained network environments. These conservative defaults, while ensuring stability on lossy links or across great distances, often fail to unleash the full potential of modern, high-bandwidth, low-latency network infrastructure. The subsequent chapter, therefore, turns inward, moving from the queue management of the network interface to the core of the protocol stack itself, exploring the art and science of TCP/IP tuning. We will delve into the kernel parameters and socket options that govern congestion control algorithms, buffer sizes, and timing behaviors, empowering you to transform a generalized TCP stack into a finely-tuned engine capable of maximizing throughput, minimizing latency, and dynamically adapting to the specific characteristics of your network path.

In the next chapter, we will explore details of the TCP/IP stack. We will go through the configuration and tuning of the network stack using the sysctl and proc filesystem interface provided by Linux for different use cases.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 9

Deep Dive into TCP/IP Stack Tuning

Introduction

The network is the circulatory system of modern computation. From serving a web page to a user across the globe, to replicating data across cloud datacenters, to executing a high-frequency trade, the speed and reliability of data transmission are paramount. At the heart of nearly every one of these interactions on a Linux system lies a powerful, complex **Transmission Control Protocol and Internet Protocol** (**TCP/IP**) stack.

For many, the default configuration of this stack is a set-and-forget-it affair. The kernel's defaults are designed for universal compatibility, a one-size-fits-all solution that ensures your system can communicate on any network. However, *what if your needs are specialized? What if you are running a high-traffic web service, a data-intensive backup server, a latency-sensitive financial application, or a high-performance computing cluster?* The generic settings suddenly become a bottleneck, an invisible ceiling limiting your application's potential. The network, which should be a transparent highway for your data, can become a source of frustrating latency, unpredictable throughput, and needless resource consumption.

This chapter is your guide to lifting that ceiling. It is a deep dive into the art

and science of Linux TCP/IP stack tuning. We will move beyond the defaults and learn how to sculpt the kernel's networking behavior to meet specific performance goals. This is about building a fundamental understanding of why these parameters matter and how they interact with the physics of networking.

Structure

The chapter covers the following topics:

- The proc filesystem and sysctl
- Tuning for maximum throughput
- Tuning connection handling
- Tuning for stability
- TCP congestion control
- Modern web server with QUIC and HTTP/3
- Practical implementation

Objectives

We will begin by exploring the core parameters you can use to control everything from the size of data windows to the handling of thousands of concurrent connections. You will learn to tune for raw throughput by mastering TCP window scaling and buffer sizing to fill high-bandwidth, high-latency pipes. You will discover how to optimize connection handling to reduce handshake overhead and efficiently manage the lifecycle of sockets, preventing your server from being hobbled by TIME_WAIT states. We will dedicate significant attention to TCP congestion control algorithms. You will learn the difference between the default, robust CUBIC and the modern, model-based BBR, which can dramatically reduce latency and improve throughput by proactively understanding the network path rather than just reacting to packet loss. Finally, we will look to the future with an introduction to **Quick UDP Internet Connections** (**QUIC**) and HTTP/3.

By the end of this chapter, you will have a comprehensive and clear methodology to diagnose bottlenecks, implement targeted optimizations, and validate their effectiveness.

The proc filesystem and sysctl

The Linux kernel exposes a vast array of its internal runtime parameters through a special, memory-based filesystem known as **proc** (the process) filesystem. This interface provides a direct window into the kernel's state, allowing for both inspection and modification of system attributes. For network tuning, the specific directory of interest is **/proc/sys/net/**.

This directory is hierarchically organized, mirroring the structure of the kernel's networking subsystems. The key subdirectories for our purposes are:

- **net/ipv4/:** Contains parameters specific to the IPv4 protocol implementation, including core TCP/IP settings for connection management, routing, and addressing.
- **net/ipv6/:** Holds the equivalent parameters for the IPv6 protocol stack.
- **net/core/:** Includes parameters that govern the behavior of the core networking layer, applicable to both IPv4 and IPv6, such as generic buffer sizes and queueing behavior.

Within these directories, plain text files represent individual kernel parameters. For example, the state of TCP window scaling is controlled by the file **/proc/sys/net/ipv4/tcp_window_scaling**. The current value of a parameter can be viewed by reading the file, typically using commands like **cat**:

```
$ cat /proc/sys/net/ipv4/tcp_window_scaling
1
```

A value of **1** indicates that the feature is enabled. It is critical to understand that writing to these files provides a direct mechanism to change the kernel's running configuration. This can be done using the **echo** command with elevated privileges:

```
# echo 1 > /proc/sys/net/ipv4/tcp_window_scaling
```

This command enables TCP window scaling. Changes made this way take effect immediately but are volatile; they will be reverted to their default values upon the next system reboot. This method is ideal for testing and experimentation.

While directly manipulating **/proc** files is functionally effective, the preferred and more robust method for managing kernel parameters is the **sysctl** command. This utility is specifically designed to interact with the **/proc/sys** directory structure, providing a cleaner and more consistent interface.

The **sysctl** command uses dots (**.**) to represent the directory path. For instance, the file **/proc/sys/net/ipv4/tcp_window_scaling** corresponds to the parameter **net.ipv4.tcp_window_scaling**.

To view a specific parameter, use the following code:

```
$ sysctl net.ipv4.tcp_window_scaling
net.ipv4.tcp_window_scaling = 1
```

To view all currently available networking parameters:

```
$ sysctl -a | grep '^net\.'
```

To change a parameter for the current session (equivalent to echoing to the **/proc** file), use the **-w** (write) flag:

```
# sysctl -w net.ipv4.tcp_window_scaling=1
```

This change is, like the direct file write, non-persistent across reboots. To ensure tuning parameters survive a system restart, they must be recorded in a configuration file. The system loads these files at boot time, applying the settings they contain. The primary configuration file is **/etc/sysctl.conf**. For better organization, especially when managing configurations from packages or custom scripts, the **/etc/sysctl.d/** directory is used. Files in this directory with a .conf extension are read and applied in lexicographical order. To make a change permanent, add the parameter and its desired value to one of these files. The syntax is the same as the **sysctl** command's output. The following is an example of setting a **sysctl** parameter through a config file:

```
# Enable TCP Window scaling for large transfer windows
net.ipv4.tcp_window_scaling = 1
```

After editing the configuration file, you can either reboot the system or use the **-p** flag to load the settings immediately. To load all configuration files:

```
# sysctl --system
```

To load a specific file, use the following command:

```
# sysctl -p /etc/sysctl.d/99-my-tuning.conf
```

This combination of the **/proc** filesystem and the **sysctl** utility forms the foundational toolkit for all subsequent Linux kernel networking tuning discussed in this chapter.

Tuning for maximum throughput

To achieve maximum throughput, the goal is to keep the network pipe completely full of data in transit. The capacity of this pipe is not determined by bandwidth alone; it is a function of both the network's speed and its latency. This fundamental concept is captured by the **Bandwidth-Delay Product** (**BDP**). The BDP represents the maximum amount of data (in bytes) that can be in flight between a sender and a receiver at any given time. It is calculated as:

BDP (bits) = Bandwidth (bits/second) × Round-Trip Time (seconds)

BDP (bytes) = [Bandwidth (bits/second) × Round-Trip Time (seconds)] / 8

Consider a network path with 1 Gbps of bandwidth and a 50 ms RTT, which is common in continental or cross-continental links. The formulas will be used as follows:

BDP = (1,000,000,000 bits/s × 0.050 seconds) / 8

BDP = 6,250,000 bytes (or ~6.25 MB)

This means the network pipe can hold 6.25 MB of data. For the sender to fill this pipe and achieve the full 1 Gbps throughput, it must be able to have up to 6.25 MB of unacknowledged data in flight.

This is where the problem with default settings arises. The original TCP header uses a 16-bit field to advertise the receive window size, which limits

the window to a maximum of 65,535 bytes (64 KiB). As our example shows, a 64 KiB window is utterly insufficient for a 1 Gbps, 50 ms link, creating a hard ceiling on performance:

Max throughput = (Window Size / RTT)

Max throughput =~ (65,535 bytes × 8 bits/byte) / 0.050 seconds = ~10.5 Mbps

Despite having a 1 Gbps link, the default TCP window limits throughput to just over 10 Mbps. The following sections detail the mechanisms to overcome this limitation.

To break the 64 KiB barrier, RFC 1323 defines the TCP window scaling option. This mechanism allows a TCP connection to negotiate a scaling factor during the three-way handshake, effectively shifting the 16-bit window field to create a much larger virtual window.

During the SYN handshake, each end advertises its own window scale factor. The scale factor represents a binary shift left operation. A scale factor of N allows the advertised window to be multiplied by 2^N. The true receive window is calculated as:

Advertised Window × 2^(Scale Factor)

This table lists the scale factor and the maximum window size:

Scale factor	Multiplier	Maximum window size
0	1	64KiB
1	2	128KiB
2	4	256KiB
16	16,384	1GiB

***Table 9.1**: Scale factor and window size*

Check the status of window scaling controlled by the following kernel parameter:

```
sysctl net.ipv4.tcp_window_scaling
```

This is enabled by default on modern Linux systems. Window scaling is a negotiated option. Use the **ss** command to verify it is active on a connection. Look for the **wscale**—value in the output:

```
$ ss -it
State      Recv-Q Send-Q  Local Address:Port   Peer Address:Port
ESTAB      0      0       192.168.1.10:443     10.0.0.50:12345
... wscale: 7,7 ...
```

The **wscale: 7,7** indicates the send and receive scale factors are 7, allowing for a window size of *65,535 × 2^7 = ~8 MiB*.

Enabling window scaling is only the first step. The kernel must also be configured with sufficiently large socket buffers to hold the data for these large windows. The Linux kernel auto-tunes these buffers within defined limits, which we must adjust to match our BDP.

The relevant parameters are split into two categories:

- Global maximums
- TCP-specific auto-tuning ranges

Global maximum parameters set the hard ceiling for all socket types (TCP, UDP, raw):

- **net.core.rmem_max:** Maximum receive buffer size for a single socket.
- **net.core.wmem_max:** Maximum send buffer size for a single socket.

TCP auto-tuning memory ranges are buffer sizes that the kernel auto-tunes for each connection based on factors like available memory and observed latency. The **tcp_rmem** and **tcp_wmem** parameters define the bounds of this auto-tuning.

Use the following **sysctl** command to view the current configurations:

```
$ sysctl net.ipv4.tcp_rmem
net.ipv4.tcp_rmem = 4096    131072    6291456
$ sysctl net.ipv4.tcp_wmem
net.ipv4.tcp_wmem = 4096   16384     4194304
```

sysctl prints the values for the **net.ipv4.tcp_rmem** and **net.ipv4.tcp_wmem** as min, default, max values. They define the default value of the parameter and the range for the auto-tuned value. For example, the **tcp_rmem** values can be described as:

- **min**: Minimum receive buffer size for each TCP socket. Used even

under memory pressure.

- **default**: The default receive buffer size. This value overrides the **net.core.rmem_default** setting for TCP.
- **max**: The maximum receive buffer size for auto-tuning. This value cannot exceed **net.core.rmem_max**.

Similarly, **net.ipv4.tcp_wmem** are the equivalent settings for the send buffer.

Additionally, the system-wide memory control parameter **net.ipv4.tcp_mem** defines the total system-wide memory usage for TCP, in 4 KiB pages.

This can be viewed with the following **sysctl** command:

```
$ sysctl net.ipv4.tcp_mem
net.ipv4.tcp_mem = 186255 248343    372510
```

The values for this parameter are printed as low, pressure, and high and are described as follows:

- **low**: TCP is not pressured to moderate memory use.
- **pressure**: When total TCP memory usage exceeds this value, the kernel starts to moderate memory use (e.g., by shrinking buffers). Enter *pressure* mode.
- **high**: The maximum number of pages allowed for queued TCP data. The kernel will aggressively drop packets to keep memory consumption under this limit.

Let us calculate the buffer sizes using our 1 Gbps / 50 ms example with a BDP of ~6.25 MB:

To calculate the target buffer size, a good rule of thumb is to set the maximum buffer size to 2× to 4× the BDP. This allows for bursts and multiple connections. It also accounts for the fact that the BDP can fluctuate. The formula is as follows:

Target Buffer = 2 × 6.25 MB = ~12.5 MB (13,107,200 bytes)

To configure the parameters, we set the **sysctls** for the global maximums to at least the target buffer size, as follows:

```
sysctl -w net.core.rmem_max=13421728  # ~13 MB
```

```
sysctl -w net.core.wmem_max=13421728
```

To configure the TCP auto-tuning limits. The min and default can often be left at system defaults for most use cases, while the max should be set to the target. Use the following sysctl commands to tune the max value:

```
sysctl -w net.ipv4.tcp_rmem=”4096 87380 13421728”
sysctl -w net.ipv4.tcp_wmem=”4096 87380 13421728”
```

Ensure **tcp_mem** is sized appropriately for the total number of concurrent connections you expect. For a server with 8 GB of RAM dedicated to networking with many connections:

Total Pages = (8 GB / 4 KiB) = 2,097,152 pages

A reasonable setting might be to allocate roughly 3 GB, 4 GB, and 6 GB of memory for low, pressure, and high thresholds:

```
sysctl -w net.ipv4.tcp_mem=”786432 1048576 1572864”
```

Other considerations to keep in mind while setting these buffer sizes are bufferbloat and system memory pressure:

- **Buffersbloat**: Excessively large buffers can lead to bufferbloat, a condition where large queues in network buffers introduce high latency, especially when the link becomes congested. BBR congestion control is designed to help mitigate this.
- **Memory pressure**: On a server handling thousands of connections, setting the maximum buffer too high can lead to exhausting system memory. The **tcp_mem** parameter is the final safeguard against this.

Tuning is a balance between per-connection performance and total system capacity.

Tuning connection handling

The TCP three-way handshake (SYN, SYN-ACK, ACK) introduces a fundamental round-trip of latency before any application data can be transmitted. For short-lived connections or high-latency networks, this overhead can significantly impact performance. This section covers kernel parameters that help mitigate this cost.

TCP Fast Open

The TCP 3-way handshake is a process where two hosts (a client and a server) exchange three specific TCP packets to set up a connection before any actual data is sent. This process serves several critical purposes; both sides inform the other of their starting sequence numbers. This is crucial for ensuring that all packets are received, are in order, and that no packets are missing or duplicated. It verifies that data can flow in both directions. Using random **Initial Sequence Number** (**ISN**) prevents a delayed or duplicate packet from a previous connection from being mistaken as part of a new, valid connection.

The three steps are as follows:

1. **Synchronize (SYN)**:
   a. The client sends a TCP packet to the server.
   b. This packet has the SYN flag set to 1.
   c. The client also chooses a random ISN and puts it in the sequence number field. Let's call this number x.
2. **Synchronize-acknowledge (SYN-ACK)**:
   a. The server receives the SYN packet and responds with its own TCP packet.
   b. This packet has two flags set, the SYN flag set to 1 and the ACK flag set to 1. The server chooses its own random ISN. Let us call this number y.
   c. The server sets the acknowledgment number field to x + 1 (the client's sequence number plus one).
3. **Acknowledge (ACK)**:
   a. The client receives the SYN-ACK packet and sends a final TCP packet back to the server.
   b. This packet has the ACK flag set to 1.
   c. The sequence number is now set to x + 1 (because the first byte of data it sends will be number x+1).
   d. The acknowledgment number is set to y + 1 (the server's sequence number plus one).

The following figure describes the sequence of steps in the TCP 3-way handshake to setup a new connection:

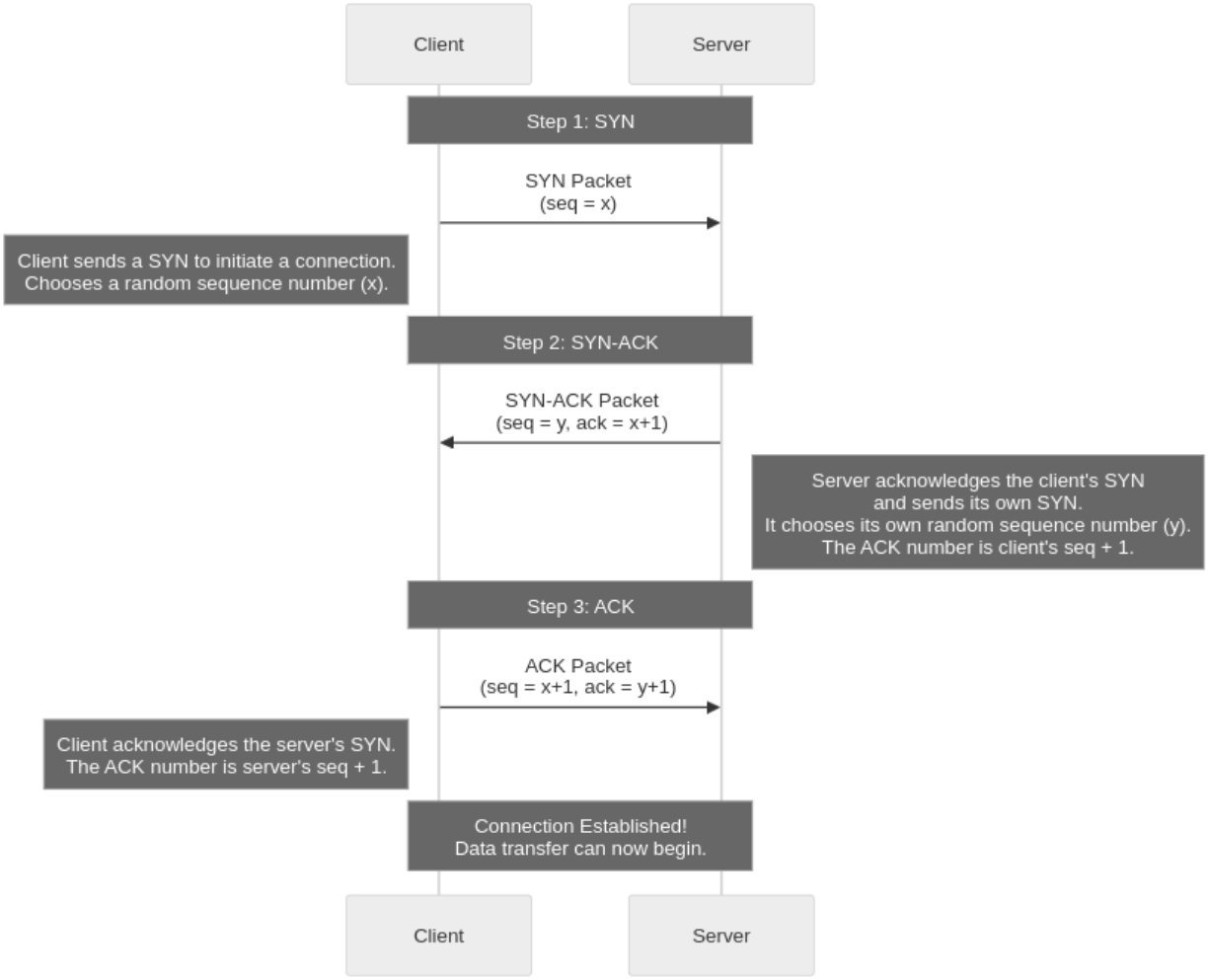


***Figure 9.1**: TCP 3-way handshake*

TCP Fast Open is a mechanism that allows data to be sent in the initial SYN packet and the corresponding response to be sent in the SYN-ACK, effectively reducing the handshake to **one round-trip** (**1-RTT**) for eligible connections.

The first connection sets up the TFO cookie as follows:

- A client performs a normal 3-way handshake but includes a TFO cookie request in the SYN.
- The server responds with a cryptographically secure cookie in the SYN-ACK.

The following figure describes the sequence of steps used to setup the first connection with TCP Fast Open:

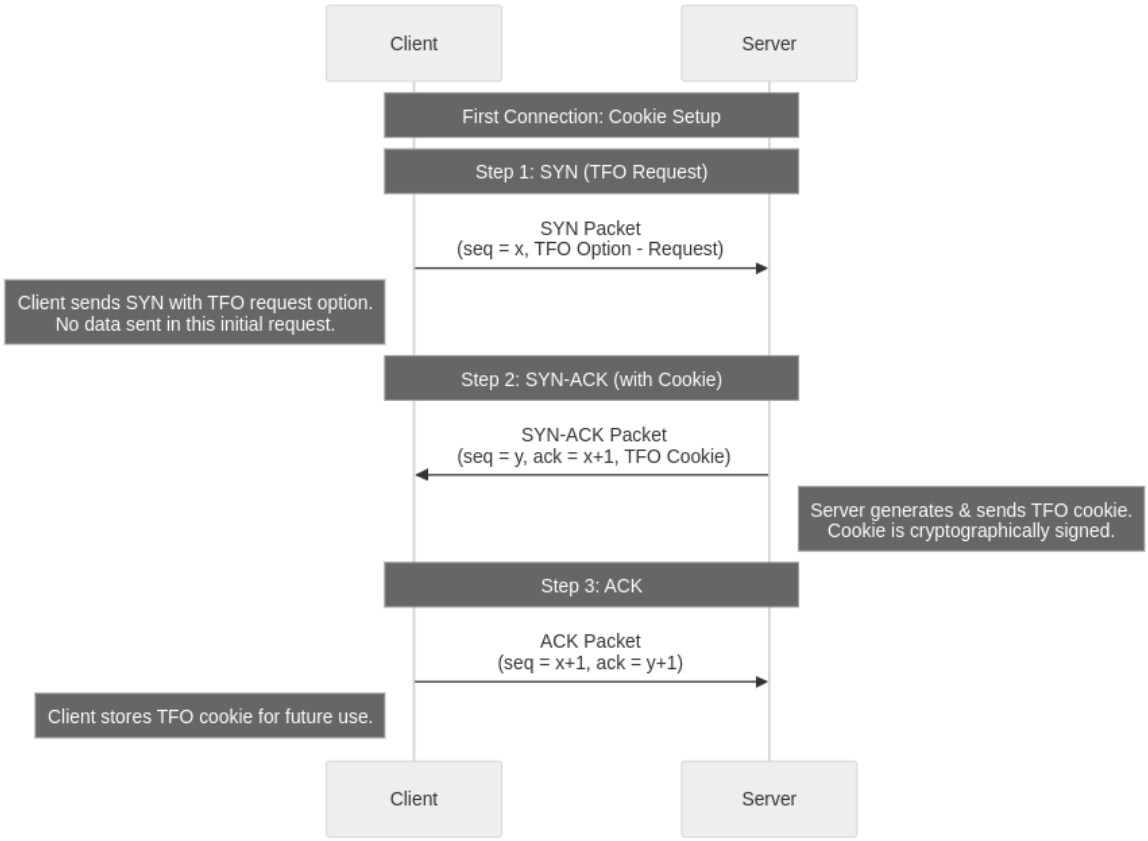


***Figure 9.2**: TCP Fast Open first connection*

The subsequent connections are as follows:

- The client stores the cookie. On its next connection to the same server, it includes both the TFO cookie and the initial data payload in the SYN packet.
- **Validation**: The server validates the cookie. If valid, it can acknowledge the data and deliver it to the application before the 3-way handshake is complete.

The following figure shows the steps in setting up the subsequent connection with TCP Fast Open:

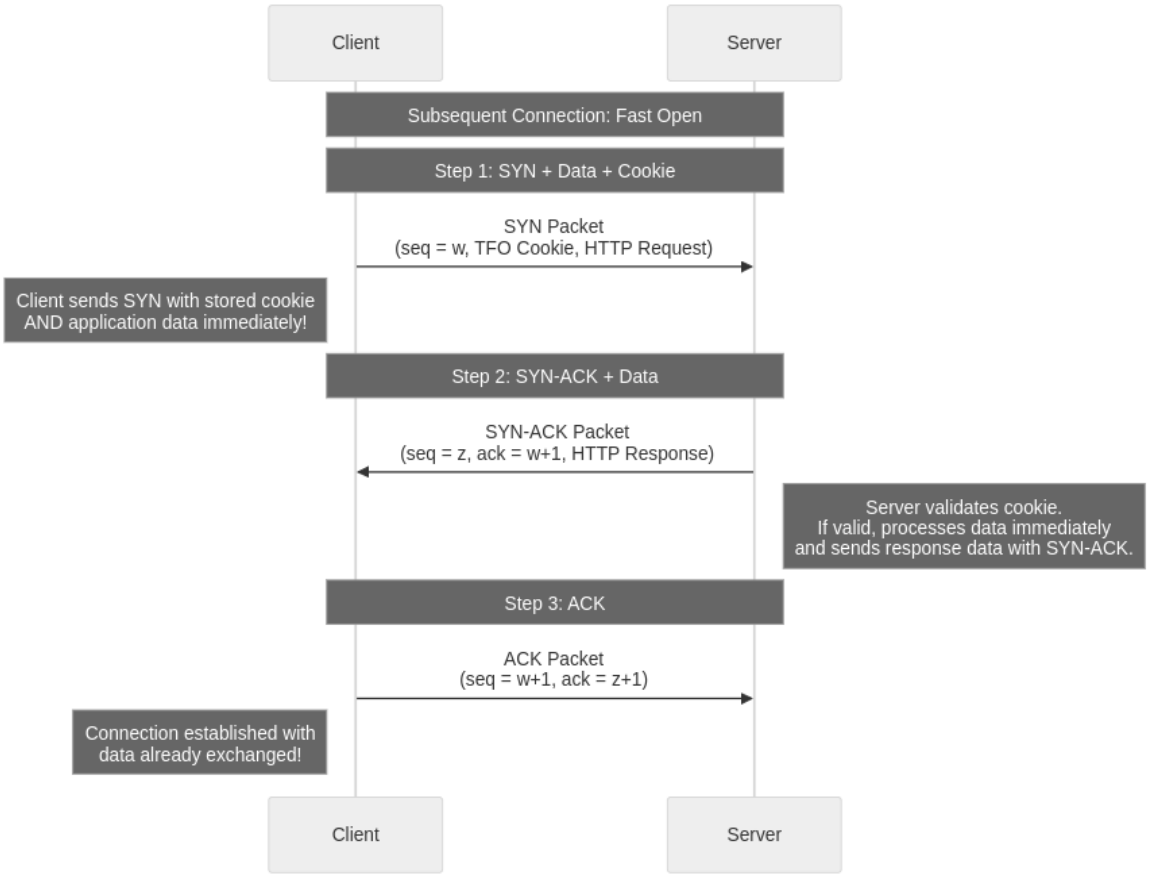


***Figure 9.3**: TCP Fast Open subsequent connection*

To enable TCP Fast Open, the **net.ipv4.tcp_fastopen sysctl** parameters must be set. The value for this parameter is a bitmask that controls TFO behavior:

- **Bit 0 (1)**: Enables TFO for clients (sending TFO cookies).
- **Bit 1 (2)**: Enables TFO for servers (accepting TFO cookies).
- **Bit 2 (4)**: Enables TFO without a cookie on a connection where a previous TFO cookie was already used (experimental/use with caution).

A common production setting is 3 (client and server enabled): **net.ipv4.tcp_fastopen = 3**

SYN backlog tuning

When a server receives a SYN packet, it places the incoming connection

request into a SYN queue (or incomplete connection queue). If this queue is full, new SYNs will be dropped, leading to failed connection attempts.

The following **sysctl** configurations can be used to control the SYN queue behavior:

- **net.ipv4.tcp_max_syn_backlog**: The maximum number of remembered connection requests (SYN packets) which have not yet received an acknowledgment from the connecting client.

  ```
  $ sysctl net.ipv4.tcp_max_syn_backlog
  net.ipv4.tcp_max_syn_backlog = 1024
  ```

- **net.ipv4.tcp_synack_retries**: This parameter controls the number of times the kernel will retransmit the SYN-ACK packet for a half-open connection. The default is typically 5, which, with standard timeouts, can leave a connection in the half-open state for over a minute. Reducing this value helps clear half-open connections from the queue more quickly during an attack. Tuning this to 2 or 3 can significantly reduce the lifetime of a spoofed connection, making the server more resilient.

  ```
  # Reduce SYN-ACK retries to clear half-open connections faster
  $ sysctl net.ipv4.tcp_synack_retries
  net.ipv4.tcp_synack_retries = 2
  ```

- **net.core.somaxconn:** The maximum number of established connections, waiting to be accepted by the application, that can be queued for a single listening socket. If the queue is full, the kernel will drop new SYNs, even if **tcp_max_syn_backlog** has space.

  ```
  $ sysctl  net.core.somaxconn
  net.core.somaxconn = 4096
  ```

- **High-performance server**: For a high-performance server, both values should be increased significantly from their defaults. In the following example, both are set to 65536:

  ```
  sysctl –w net.ipv4.tcp_max_syn_backlog=65536
  sysctl –w net.core.somaxconn=65536
  ```

Managing connection teardown

Unlike the symmetric 3-way opening handshake, closing a connection is a 4-step process. This is because a TCP connection is full-duplex, and data can flow independently in both directions simultaneously. Therefore, each direction must be closed independently.

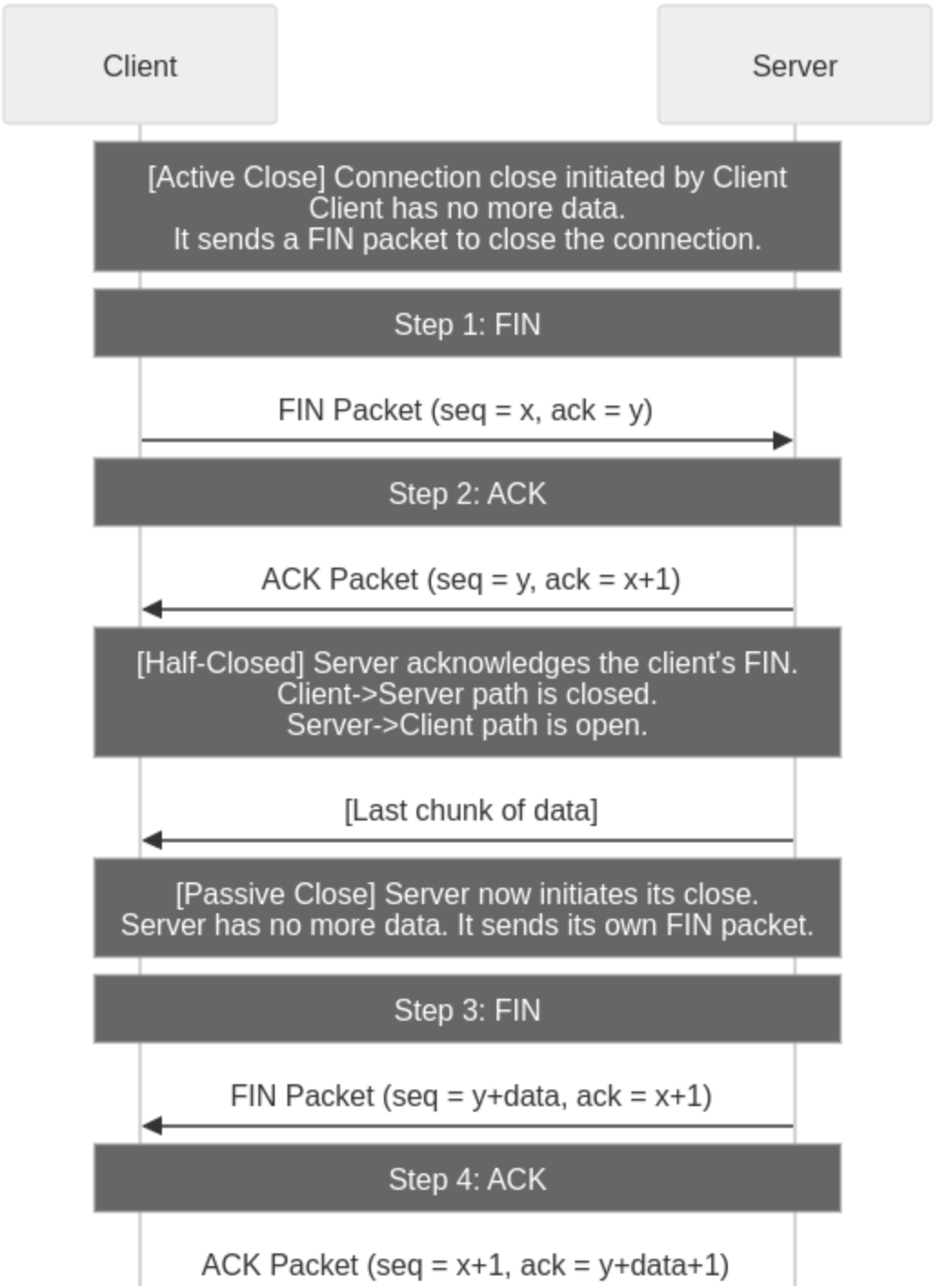
The four steps are as follows:

1. **Finish** (**FIN**) from one host:
    a. The client decides it has no more data to send. It sends a TCP segment with the FIN flag set to 1.
    b. The FIN segment consumes one sequence number. If the last data byte sent was sequence number x-1, the FIN will have sequence number x.
    c. The client enters the FIN-WAIT-1 state.
2. ACK from the other host:
    a. The server receives the FIN and responds with an ACK segment.
    b. The ACK number is set to x + 1 (the client's FIN sequence number plus one).
    c. The server enters the CLOSE-WAIT state. The client, upon receiving this ACK, enters the FIN-WAIT-2 state.
    d. At this point, the connection is half-closed. The client cannot send any more data to the server, but the server can still send data to the client.
3. FIN from the other host:
    a. When the server is also ready to close the connection, it sends its own segment with the FIN flag set.
    b. This FIN will have its own sequence number (e.g., y + data_len if it sent more data after step 2).
    c. The server enters the LAST-ACK state.
4. ACK from the first host:
    a. The client receives the server's FIN and responds with a final ACK segment.

b. The acknowledgment number is set to the server's FIN sequence number plus one.

c. The client enters the TIME_WAIT state.

The detailed sequence diagram of the steps is as follows:



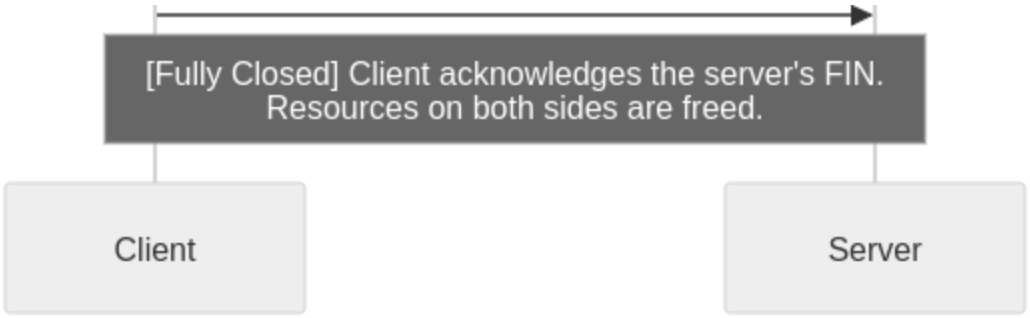


***Figure 9.4**: Closing TCP connection*

The TIME_WAIT state is a normal part of TCP connection termination. After the active closer, typically the client sends the final ACK, it must keep the socket pair in TIME_WAIT for a duration of 2 * **maximum segment lifetime** (**MSL**). This ensures that any stray, delayed packets from the old connection are not mistaken as part of a new connection.

When a server is the one to close the connection (e.g., after serving an HTTP response), it enters TIME_WAIT. Under high load, this can lead to a large number of sockets in this state, consuming kernel memory and, more critically, exhausting the available local port range for new outgoing connections from the server. The kernel parameters for TIME_WAIT management are as follows:

- **net.ipv4.tcp_tw_reuse:** Allows a socket in TIME_WAIT to be reused for a new outgoing connection if it is safe to do so from a protocol perspective. Primarily useful for servers that make many outgoing connections (e.g., a web server acting as a client to backend services). It helps prevent local port exhaustion.
- **net.ipv4.tcp_fin_timeout:** Time in seconds a socket must remain in the FIN-WAIT-2 state if the other end is not closing. This is not the TIME_WAIT duration, but it is related to connection teardown. The default is 60 seconds. Reducing this can help free resources slightly faster in certain edge-case scenarios.
- **net.ipv4.tcp_max_tw_buckets**: The maximum number of TIME_WAIT sockets the system will hold simultaneously. When this limit is exceeded, the kernel will begin to destroy TIME_WAIT

sockets, printing a warning. This can be used as a safety net to prevent a TIME_WAIT explosion from consuming all memory.

- **net.ipv4.ip_local_port_range**: Two numbers defining the range of local ports available for outgoing connections. A wider range provides more possible ports, which can directly mitigate port exhaustion. The ports in this range should not be used as server ports.

Keepalives and connection persistence

TCP Keepalives are a mechanism to detect if the remote peer of a connection has become unreachable (crashed, network partition) without tearing down the connection. A small probe packet is sent after a long period of inactivity.

The following parameters can be used to tune the keepalive probs. These are per-socket defaults that can often be overridden by the application:

- **net.ipv4.tcp_keepalive_time**: The time in seconds a connection must be idle before the first keepalive probe is sent. The default is typically 7200 seconds (2 hours).
- **net.ipv4.tcp_keepalive_intvl**: The time in seconds between successive keepalive probes, if no response is received. The default is 75 seconds.
- **net.ipv4.tcp_keepalive_probes**: The number of unacknowledged keepalive probes to send before considering the connection dead and closing it. Default is 9.

For connection pools in environments like application servers talking to databases or proxies (e.g., NGINX upstreams), the defaults are often too high. A dead connection might not be detected for over 2 hours. A more aggressive configuration can help clean up stale connections faster. The following example detects a dead peer in around 11 minutes: (5 min + 75s * 5):

```
net.ipv4.tcp_keepalive_time = 300
net.ipv4.tcp_keepalive_intvl = 75
net.ipv4.tcp_keepalive_probes = 5
```

Tuning for stability

Tuning the network for stability is about understanding how the system behaves under load. Monitoring packet drops, resisting attacks, and identifying bottlenecks before they cause significant user-facing issues.

Hardening against SYN floods

A SYN flood is a classic **denial-of-service** (**DoS**) attack that aims to exhaust a server's resources by overwhelming it with TCP connection requests. A typical SYN flood attack occurs as follows:

1. The attacker sends a rapid succession of SYN packets to the server, often with a spoofed source IP address.
2. For each SYN, the server allocates kernel memory to track the new connection, sends back a SYN-ACK, and waits for the final ACK from the client to complete the three-way handshake.
3. Since the source IP is spoofed, the final ACK never arrives. The half-open connection remains in the SYN-RECEIVED state until a timeout occurs.
4. By flooding the server with these half-open connections, the attacker fills the SYN queue, causing the server to drop legitimate connection attempts.

SYN Cookies are a defense mechanism that allows a server to handle SYN floods without storing state for half-open connections. When the SYN queue begins to fill up, the kernel activates SYN Cookies.

The following steps describe the working of the SYN Cookie mechanism:

1. Instead of storing connection state, the server encodes a cryptographic hash of the connection details (source/dest IP/ports, timestamp) into the ISN it sends back in the SYN-ACK packet.
2. The server then purges all memory of the connection request.
3. If the client is legitimate, it will respond with an ACK that includes the ISN+1. The server can then recalculate the cryptographic hash from the packet headers. If it matches, the connection is reconstructed and moves to the established state without ever having occupied the SYN

queue.

net.ipv4.tcp_syncookies can be used to enable SYN Cookies. It can have three possible values, namely 0 to disable SYN cookies, 1 to enable cookies only when the SYN queue is about to overflow. This is the default and recommended setting, and 2 is always enabled. Not recommended for normal operation.

Monitoring for packet drops

Monitoring packet drops is a critical diagnostic practice to identify bottlenecks. A high and growing dropped count is of primary concern. Use **netstat -i** or **ip -s** link to check for RX/TX packet drops and errors on network interfaces. This often indicates that the network driver or the hardware itself is overwhelmed.

Use the **ip link show** command to check the per interface packet statistics is as follows:

```
$ ip -s link show enp1s0
2: enp1s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 .
   link/ether 52:54:00:ee:6a:84 brd ff:ff:ff:ff:ff:ff
   RX:  bytes packets errors dropped  missed   mcast
      697171    1431      0       4       0       0
   TX:  bytes packets errors dropped carrier collsns
      136149    1283      0       0       0       0
```

The ss command is invaluable for viewing socket-level memory usage and queue states. The following example shows the usage of **ss** command to list listening and established sockets, respectively:

```
$ ss -ltnp
State  Recv-Q Send-Q          Local Address:Port  Peer Address:PortProcess
LISTEN 0      4096           127.0.0.53%lo:53    0.0.0.0:*
LISTEN 0      511            127.0.0.1:6379      0.0.0.0:*
...
```

```
$ ss -tnp
State Recv-Q Send-Q     Local Address:Port      Peer Address:Port
Process
ESTAB 0      0          127.0.0.1:6379          127.0.0.1:57920
ESTAB 0      0          127.0.0.1:57920         127.0.0.1:6379
```

For a listening socket, the receive and send queue values can be interpreted as follows:

- **Recv-Q** shows the number of unaccepted connections in the somaxconn queue. A non-zero value here indicates the application is not accepting connections fast enough.
- **Send-Q** shows the maximum backlog length (somaxconn).

For an established socket, the receive and send queue values can be interpreted as follows:

- **Recv-Q** shows bytes held by the kernel, waiting to be read by the application. A consistently high value indicates the application is not reading fast enough.
- **Send-Q** shows bytes not yet acknowledged by the remote peer.

The **/proc/net/netstat** file provides detailed counters, including those related to SYN Cookies:

```
$ awk '/^TcpExt/{print $1"\t"$2"\t"$3"\t"$4}' /proc/net/netstat
|column -t
TcpExt:  SyncookiesSent  SyncookiesRecv  SyncookiesFailed
TcpExt:  0               0               0
```

SyncookiesSent is the number of SYN-ACKs sent with SYN Cookies. A non-zero value indicates the SYN queue was overwhelmed. **SyncookiesRecv** is the number of valid cookies received, meaning a connection was re-established. **SyncookiesFailed** is the number of failed cookie validations. It should be zero or very low.

A comprehensive stability monitoring strategy involves correlating these metrics. For example, a spike in interface drops coupled with a high **Recv-Q** on established sockets and a growing number of **SyncookiesSent** paints a clear picture of a server under a high connection load, where the application

is struggling to keep up, leading to packet loss and the activation of flood protections.

TCP congestion control

Congestion control is the feedback mechanism that prevents a TCP sender from overwhelming the network. It is a fundamental part of the protocol's stability and fairness. Without it, network routers would experience buffer overflows, leading to high packet loss and a collapse in useful throughput, a condition known as congestion collapse.

The core principle is for each sender to dynamically adjust its **congestion window** (**cwnd**), which is the maximum amount of data it can have in flight before receiving an acknowledgment. The algorithm's job is to find the optimal cwnd—large enough to fully utilize the available bandwidth, but small enough to avoid causing excessive queueing delay (latency) or packet loss.

The congestion control algorithms operate by reacting to signals from the network, primarily one of these:

- **Packet loss**: The classic signal of congestion. Routers drop packets when their buffers are full.
- **Packet delay (RTT):** A more modern signal. An increase in RTT indicates queueing in router buffers, a precursor to full-blown congestion.

The choice of algorithm profoundly impacts throughput, latency, and fairness when sharing a link with other flows.

CUBIC

CUBIC has been the default congestion control algorithm in Linux for a long time. It is designed to perform well on high-bandwidth, high-latency (LFN) networks, which are common in the modern internet. CUBIC is a loss-based algorithm. Its core innovation is using a cubic function of time (rather than the linear function used by its predecessor, BIC) to grow the congestion window after a loss event. The following steps define the

working of the CUBIC congestion control algorithm:

1. Upon detecting packet loss (via duplicate ACKs), CUBIC reduces its window by a multiplicative factor (default is 30%, so the new window is 70% of the previous maximum).
2. Instead of immediately growing the window aggressively, it enters a steady phase. The window growth is determined by a cubic function whose goal is to return to the pre-loss window size (the W_max) slowly and then exceed it cautiously.
3. For the final approach to W_max, CUBIC switches to a standard TCP-friendly **additive increase/multiplicative decrease** (**AIMD**) mode to ensure fairness with legacy Reno/CUBIC flows.
4. Once past W_max, it grows slowly to probe for new available bandwidth.

CUBIC is excellent at utilizing available bandwidth on long-fat networks. It tends to be fairer to flows with different RTTs than older algorithms like Reno. The cubic function provides smooth and stable window growth, reducing oscillations.

On the other hand, as CUBIC is a loss-based algorithm, it must fill router buffers to find the available bandwidth limit. This creates persistent queueing delay, increasing latency even when the network is not fully congested. It only reacts after packet loss has already occurred, which is an inherently lagging indicator of congestion.

To view or set the default algorithm for new connections, use the following sysctl parameter:

```
$ sysctl net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = cubic
```

Bottleneck bandwidth and round-trip time

Bottleneck bandwidth and round-trip time (**BBR**), developed by *Google*, represents a paradigm shift in congestion control. Instead of reacting to loss, it builds an explicit model of the network path and paces packets to avoid creating queues and loss in the first place.

BBR's model is based on two key metrics:

- **BBR.max_BW**: The maximum delivery rate sampled over recent time windows.
- **BBR.min_RTT**: The minimum RTT sampled over a longer time window (typically tens of seconds to minutes).

BBR operates through a four-phase cycle:

1. **STARTUP**: The initial phase, similar to a slow start, that rapidly grows the sending rate to quickly probe for **BBR.max_BW**.
2. **DRAIN**: A brief phase to drain any queue that was built up during **STARTUP**.
3. **PROBE_BW**: The steady-state phase. BBR cycles through varying pacing gains to periodically probe for more bandwidth (slightly above **BBR.max_BW**) while spending most of its time sending at a rate that matches **BBR.max_BW**, which avoids creating a persistent queue.
4. **PROBE_RTT**: Periodically, BBR enters a phase where it reduces its cwnd to a very small value (4 packets) for at least one **BBR.min_RTT** interval. This allows it to drain all queues and get a fresh, accurate measurement of **BBR.min_RTT**.

By avoiding the constant filling of router buffers, BBR can achieve significantly lower and more consistent latency. This is its most celebrated advantage. On lossy networks (e.g., wireless, intercontinental links), BBR often achieves higher throughput because it does not misinterpret random packet loss as a signal of congestion. BBR is explicitly designed to keep queues small, mitigating the bufferbloat problem common with CUBIC.

To enable BBR system-wide use, the following sysctl parameter:

net.ipv4.tcp_congestion_control = bbr

BBR-specific parameters can be found as modules. Their paths can change, but they are typically in **/sys/module/tcp_bbr/parameters/**.

The Linux kernel includes a variety of congestion control algorithms, each optimized for different environments, listed as follows:

- **Reno**: The classic algorithm that defined TCP congestion control. It uses pure AIMD: it increases the window by one segment per RTT (additive increase) and halves the window on packet loss

(multiplicative decrease). It is inefficient on modern high-speed networks.

- **BBRv2/BBRv3**: These are evolutions of the original BBR algorithm designed to address some of its shortcomings in certain scenarios, particularly, better coexistence with loss-based CUBIC flows and refinements in how the algorithm handles packet loss when it does occur.

  As of this writing, BBRv2 is often available but may require a newer kernel or being enabled as a custom module.

- **Data Center TCP (DCTCP)**: Designed explicitly for the unique environment of data centers, which have high bandwidth, very low RTT, and shallow-buffered switches.

  It uses **Explicit Congestion Notification** (**ECN**) to get early, fine-grained feedback from the network before packets are dropped. It adjusts the congestion window proportionally to the fraction of ECN-marked packets.

  Primarily for intra-data-center traffic. It requires ECN support on both ends and on the network switches.

The kernel compiles a list of available algorithms. You can view them with:

```
$ cat /proc/sys/net/ipv4/tcp_available_congestion_control
cubic reno bbr
$ cat /proc/sys/net/ipv4/tcp_congestion_control
Cubic
```

Choosing an algorithm

For general internet server/client, BBR is often an excellent choice for its superior latency and good throughput. For legacy compatibility and stability, CUBIC remains a robust and well-understood default. In a Controlled Data Center Environment, DCTCP (if ECN is configured) is unmatched for low latency and high throughput. The best approach is to test different algorithms (e.g., using **iperf3**) under your specific network conditions and workload patterns.

Modern web server with QUIC and HTTP/3

For decades, the web has been built upon a foundational duo: TCP for reliable data transport and TLS for security. While robust, this combination has inherent architectural limitations that become pronounced in modern, high-latency, and lossy network environments.

HTTP/2 was a major performance improvement over HTTP/1.1, introducing multiplexing, the ability to send multiple requests and responses concurrently over a single TCP connection, removing **head-of-line** (**HOL**) blocking of requests at the application layer. To understand the issue, let us create a simple web application with two endpoints, one with a fast response and one with a slow response. We will expose these endpoints with HTTP/1.1 and HTTP/2.0 services. The following code implements the fast and the slow endpoints at URI **/** and **/slow** respectively.

```
# main.go
package main
import (
"fmt"
"net/http"
"os"
"time"
)
// Create a fast handler
var handlerCommon = http.HandlerFunc(func(w http.ResponseWriter, r
*http.Request) {
fmt.Println(r.Proto, r.RemoteAddr)
fmt.Fprintf(w, "Hello from fast endpoint, Protocol: %s\n", r.Proto)
})

// Create a slow handler
var handlerCommonSlow = http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
```

```
fmt.Println(r.Proto, r.RemoteAddr)
time.Sleep(5 * time.Second)
fmt.Fprintf(w, "Hello from slow endpoint, Protocol: %s\n", r.Proto)
})

func main() {
mux := http.NewServeMux()
mux.Handle("/", handlerCommon)
mux.Handle("/slow", handlerCommonSlow)

var err error
if os.Args[1] == "http2" {
fmt.Println("Starting HTTP/2 server on https://:8443")
fmt.Println("Note: HTTP/2 requires TLS")

// Start server with HTTP/2 support (enabled by default when using TLS)
err = http.ListenAndServeTLS("0.0.0.0:8443", "cert.pem", "key.pem",
mux)
} else {
fmt.Println("Starting HTTP server on http://:8080")
err = http.ListenAndServe(,0.0.0.0:8080", mux)
}
if err != nil {
panic(err)
}
}
```

Also note you will need SSL certificates to run an HTTP/2.0 service, you can create a self-signed certificate as follows:

```
openssl req -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365
-out cert.pem
```

To test the HOL blocking, we will fast run the web application under an

HTTP/1.1 service. To do this, run the following command:

```
go run main.go http &
```

This command will start the application as HTTP/1.1 service. Now we can use **curl** command to check the HOL blocking as follows:

```
curl --parallel localhost:8080/slow localhost:8080 localhost:8080 localhost:8080
HTTP/1.1 [::1]:47034
Hello from slow endpoint, Protocol: HTTP/1.1
HTTP/1.1 [::1]:32830
HTTP/1.1 [::1]:32828
HTTP/1.1 [::1]:47034
Hello from fast endpoint, Protocol: HTTP/1.1
Hello from fast endpoint, Protocol: HTTP/1.1
Hello from fast endpoint, Protocol: HTTP/1.1
```

We are calling the slow endpoint first, followed by calls to the fast endpoint. When running under HTTP/1.1, all the calls will be blocked until the slow call is completed, as seen in the output above.

To run the application under HTTP/2.0, use the following command:

```
go run main.go http2 &
```

To test the behavior of the application, we can use the following command:

```
curl --parallel --http2 -k https://localhost:8443/slow https://localhost:8443 https://localhost:8443 https://localhost:8443
HTTP/2.0 [::1]:37036
HTTP/2.0 [::1]:37036
HTTP/2.0 [::1]:37036
HTTP/2.0 [::1]:37036
Hello from fast endpoint, Protocol: HTTP/2.0
Hello from fast endpoint, Protocol: HTTP/2.0
Hello from fast endpoint, Protocol: HTTP/2.0
Hello from slow endpoint, Protocol: HTTP/2.0
```

In this case, even though the slow endpoint is called first, the calls to the fast endpoints are not blocked.

HTTP/2 solves the HOL issue at the application layer, but the problem still exists at the network layer. HTTP/2 still uses TCP as the transport protocol, and because TCP is a byte-stream protocol that guarantees in-order delivery, a single lost packet within that connection creates a problem. The receiving kernel's TCP stack must buffer all subsequent packets until the missing one is retransmitted and received. This stalls the delivery of all subsequent HTTP/2 streams, even those unrelated to the lost packet. This is transport-layer HOL blocking.

A secure TCP connection requires at least one full round-trip for the TCP handshake and another for the TLS handshake, totaling two round-trips before the first application data can be sent. **TCP Fast Open** (**TFO**) and TLS 1.3's **zero-round-trip time** (**0-RTT**) mode can help, but they are often hampered by deployment challenges and are not universally applicable.

QUIC, standardized as RFC 9000, is a modern transport protocol designed from the ground up to address these limitations. It is implemented as a user-space library on top of UDP, allowing for rapid evolution and deployment without requiring changes to the operating system kernel. Unlike TCP, which is a plaintext protocol, QUIC integrates TLS 1.3 directly into its core design. Every QUIC packet is authenticated and encrypted. This not only provides mandatory security but also prevents ossification in middleboxes (like routers and firewalls), as they cannot inspect the protocol's inner workings, allowing the protocol to evolve more freely.

QUIC's most significant performance innovation is introducing the concept of independent, lightweight streams within a single connection.

Using multiple streams provides the following benefits:

- Each stream is a logically separate channel for a different HTTP/3 request-response pair.
- Lost packets only affect the specific stream they belong to. The other streams can continue to deliver their data uninterrupted.
- This eliminates transport-layer HOL blocking at the protocol level, a fundamental improvement over TCP.

A TCP connection is identified by the 4-tuple (source IP, source port, destination IP, destination port). If a client's IP address changes (e.g., switching from Wi-Fi to cellular), the TCP connection breaks and must be re-established. QUIC uses a connection ID, a persistent identifier that allows a connection to survive IP address changes seamlessly, providing a more robust experience for mobile users.

By combining the transport and cryptographic handshakes and leveraging TLS 1.3, QUIC can often establish a secure connection in 0-RTT for repeat visitors. The client can send application data (e.g., an HTTP request) in its very first packet, using cryptographic material cached from a previous connection. This dramatically reduces latency for subsequent visits.

Traditional TCP tuning is done via sysctl because the TCP stack lives in the Linux kernel. QUIC, by contrast, is primarily implemented in user-space (e.g., within web servers like NGINX, Caddy, or custom applications). This means parameters like flow control windows, congestion control algorithms (QUIC has its own, often based on BBR), and connection timeouts are now managed in the web server's configuration files, not in **/etc/sysctl.d/**.

While the QUIC logic is in user-space, it still uses UDP sockets to send and receive packets. Therefore, the kernel's UDP buffer limits remain critically important. If the application's UDP socket buffers are too small, the QUIC library cannot queue enough data or handle enough incoming packets, leading to poor performance and packet drops. The same global parameters used for other UDP applications apply:

- **net.core.rmem_max** and **net.core.wmem_max** sets the maximum buffer size for a single socket.
- The application (e.g., NGINX) must be configured to request sufficiently large buffers for its QUIC sockets, but it is ultimately bounded by these kernel maximums.

The rise of QUIC and HTTP/3 does not make TCP/IP tuning obsolete. Billions of connections will continue to rely on TCP for the foreseeable future. Instead, QUIC represents an expansion of the performance landscape. A modern performance engineer must now be adept at both: finely tuning the kernel's TCP stack for traditional services and

understanding how to provide a stable platform (via UDP buffers and application configuration) for the next generation of web protocols.

Practical implementation

This chapter has provided a deep dive into the myriad levers and knobs available for tuning the Linux networking stack. The final and most critical step is to apply this knowledge methodically. Indiscriminate tuning can lead to instability, performance degradation, and difficult-to-diagnose problems. This section outlines a disciplined framework for successful implementation.

Before changing any parameter, you must quantify the current state of your system. This baseline is your reference point for measuring the impact of your changes.

For example, to set a baseline for network performance:

- Use **iperf3** or **netperf** to measure throughput and latency between key endpoints. Use ping to track baseline RTT. Use **ss -it** and **ip -s** link to record socket and interface statistics.
- Document throughput (Mbps), latency (ms), retransmission rates, and packet drop counts under both idle and load conditions.

Define clear objectives, as tuning without a goal is optimization without purpose. Define what *better* means for your specific use case. The following are some good examples of clear objectives:

- Reduce the 95th percentile latency for HTTP responses by 15% during peak traffic.
- Increase average data transfer throughput for large files from 5 Gbps to 8 Gbps over a high-latency WAN link.
- Reduce the number of failed connection attempts during connection storms by 50%.

Resist the urge to apply every tuning parameter at once. Changes should be done incrementally, and everything must be documented. Isolate the variables, change one, or at most two, related parameters at a time. For instance, adjust TCP buffer sizes, test, and then move on to congestion

control.

Maintain a detailed change log. For every adjustment, record the following:

- Date/Time
- **Parameter changed**: e.g., net.ipv4.tcp_rmem
- **Previous value**: 4096 87380 4194304
- **New value**: 4096 87380 16777216
- **Hypothesis**: Increasing max TCP read buffer to ~16MB to better match the 10 Gbps / 1ms BDP.

After each change, re-run your performance tests and compare the results against your baseline based on the following parameters:

- Did the change produce the expected effect on your target metrics?
- Did the change negatively impact any other metric? For example, did increasing buffers also increase memory pressure? Use tools like free, top, and ss -m to monitor system resources.

Ensure performance is validated under realistic load conditions, not just in an idle lab. Once you have validated a change through iterative testing, move it to a permanent configuration.

A typical validation of configuration, followed by deployment to production with continuous monitoring, is as follows:

- **Deploy to production**: Add the tuned parameter to a configuration file in **/etc/sysctl.d/** (e.g., 99-network-tuning.conf).
- **Load the configuration**: Run **sysctl --system** to apply the new settings.
- **Continuous monitoring**: Integrate the key metrics from your baseline (latency, throughput, drops, retransmits) into your long-term monitoring solution (e.g., Prometheus/Grafana). Set alerts for when these metrics deviate significantly from their newly tuned norms.

The following snippets are illustrative starting points for common server roles. They are not prescriptive. You must calculate values based on your own network's BDP and test them in your environment.

- Web server profile (many concurrent, short-lived connections):

```
# /etc/sysctl.d/99-webserver.conf
```

```
# Connection Handling
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_fin_timeout = 30
net.ipv4.tcp_max_tw_buckets = 2000000

# Handshake & Backlog
net.core.somaxconn = 65535
net.ipv4.tcp_max_syn_backlog = 65535
net.ipv4.tcp_synack_retries = 2

# Stability & Memory
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_mem = 786432 1048576 1572864
```

- Data/file server profile (high throughput, long-lived connections):

```
# /etc/sysctl.d/99-data-server.conf
# Throughput (Calculate max based on your BDP!)
net.core.rmem_max = 134217728 # 128 MB
net.core.wmem_max = 134217728
net.ipv4.tcp_rmem = 4096 87380 134217728
net.ipv4.tcp_wmem = 4096 87380 134217728
# Congestion Control (Choose based on network)
net.ipv4.tcp_congestion_control = bbr

# Connection Handling
net.ipv4.tcp_slow_start_after_idle = 0
```

Conclusion

Throughout this chapter, we have journeyed from the fundamental concepts of TCP/IP networking to the intricate details of kernel-level tuning, exploring how each parameter, from buffer sizes and window scaling to

congestion control algorithms, contributes to the overall performance and stability of Linux systems. We have seen that the default kernel configuration, while suitable for general-purpose use, often represents a compromise that falls short for performance-critical applications. By understanding the relationship between bandwidth, latency, and the bandwidth-delay product, you now possess the knowledge to systematically analyze network bottlenecks and implement targeted optimizations.

The transition from reactive, loss-based algorithms like CUBIC to proactive, model-based approaches like BBR demonstrates the evolving nature of network performance optimization. Similarly, the emergence of QUIC and HTTP/3 represents a fundamental shift in application-layer protocols, moving complexity from the kernel to user-space while maintaining the critical need for proper system-level configuration. What remains constant is the importance of a methodical approach: measure first, change incrementally, validate thoroughly, and monitor continuously. The true art of tuning lies not in applying every possible optimization, but in selecting the right combination of parameters that align with your specific workload, network characteristics, and performance objectives. Whether you are optimizing a web server for thousands of concurrent connections, a file server for maximum throughput, or a database cluster for low latency, the principles outlined here will serve as your guide to unlocking the full potential of Linux networking.

In the next chapter, we will look into the Linux VPN ecosystem, comparing solutions like WireGuard's revolutionary simplicity and speed with OpenVPN's proven reliability and feature richness. You will learn to configure Site-to-Site and remote-access VPNs, understand cryptographic fundamentals, and discover how to apply the TCP/IP tuning principles from this chapter to overcome the inherent performance overhead of encrypted tunnels. The next chapter will provide the complete toolkit for building secure, performant network bridges across the internet.

CHAPTER 10

Virtual Private Networks on Linux

Introduction

In an era defined by remote work and global connectivity, the need for secure and private communication over public networks has never been more critical. Every time you access a corporate server from your home, or link two cloud data centers, your data crosses the vast, untrusted landscape of the public internet. Like a postcard sent through the mail, this data is visible to anyone who cares to look, vulnerable to interception, manipulation, and theft.

This is where the **virtual private network** (**VPN**) emerges as a foundational technology for modern computing. A VPN creates a secure, encrypted tunnel between your device and a remote network, shielding your online activities from the world. It transforms the public internet into a private path, allowing you to access resources as if you were physically connected to a remote network, all the while hiding your true location and identity.

This chapter delves deep into two of the most significant and complementary VPN technologies available on Linux today: the sleek, modern WireGuard and the robust, enterprise-standard **Internet Protocol Security** (**IPsec**).

Structure

This chapter covers the following topics:

- Understanding VPN
- Introduction to WireGuard
- Configuring a Point-to-Site VPN
- Introduction to IPsec and strongSwan
- Authentication methods of PSK vs. certificates
- Introduction to PKI and Certificate Authority
- Configuring remote access with certificates
- WireGuard vs. strongSwan and when to use which
- Security best practices

Objectives

By the end of this chapter, you will gain comprehensive knowledge and practical skills required to understand, compare, and implement modern VPN solutions. The journey begins with a foundational grasp of what a VPN is and its essential role in securing data in transit by creating encrypted tunnels across untrusted networks like the internet. You will learn the underlying mechanics of how these secure connections are established and maintained.

Building on this core understanding, the chapter will guide you through the landscape of VPN technologies, enabling you to distinguish between different VPN types, such as remote access and Site-to-Site configurations, and to evaluate the characteristics of various protocols. A significant focus will be placed on two prominent solutions: the lean, modern WireGuard protocol and the robust, feature-rich IPsec framework implemented by strongSwan. You will develop competency with both, learning to install the necessary software, generate and manage cryptographic keys, and successfully configure a basic Point-to-Site connection with WireGuard as well as a Site-to-Site tunnel using strongSwan with pre-shared key

authentication. You will explore critical concepts that underpin enterprise-grade security, including the differences between simple pre-shared keys and certificate-based authentication, and the foundational role of **public key infrastructure** (**PKI**).

You will not only understand the theory behind these technologies but will have the hands-on skill to configure, manage, and troubleshoot both a WireGuard Point-to-Site VPN and a strongSwan IPsec Site-to-Site VPN. You will be equipped to make informed decisions about which tool is right for the job, empowering you to build the secure, private, and connected infrastructure that the modern world demands.

Understanding VPNs

At its core, a VPN is a service that establishes a secure, encrypted connection over a less secure network, primarily the public internet. The name itself defines its purpose:

- **Virtual**: It does not rely on physical, dedicated leased lines like traditional private networks. Instead, it uses a virtual pathway constructed over the existing, shared infrastructure of the internet.
- **Private**: All communication flowing through this pathway is encrypted and secured, making it inaccessible to outsiders. It creates a confidential communication session.
- **Network**: It connects two or more devices together, allowing them to communicate as if they were on the same local network.

The most common and effective way to visualize a VPN is as a secure tunnel. Imagine the public internet as a vast, public highway where anyone can see the cargo inside every vehicle. Without a VPN, your information travels in open trucks, visible to everyone, from internet service providers and advertisers to malicious actors on the same network.

A VPN constructs a protected, private tunnel through this public highway. When your data enters this tunnel, it is:

- **Encapsulated**: Placed inside a new, secure packet.
- **Encrypted**: Scrambled using complex algorithms, making it

unreadable to anyone without the key.

This encrypted packet then travels safely to its destination, where it is decrypted and sent on to its final target on the remote network. This tunnelling mechanism provides three fundamental benefits:

- **Privacy and anonymity**: By routing your traffic through a VPN server, you hide your real IP address, a unique identifier tied to your location and internet connection, from the websites and online services you use. To them, your traffic appears to originate from the VPN server, masking your identity and location.
- **Security**: The encryption provided by the VPN protects your data from eavesdropping, especially on untrusted networks like public **Wireless Fidelity** (**Wi-Fi**) at airports or cafes. Even if an attacker intercepts your data, they cannot decipher it without the encryption keys.
- **Access to remote networks**: A VPN allows you to be part of a remote network logically. This enables you to access resources that are restricted to local clients, such as a corporate file server, a home media library, or internal business applications, from anywhere in the world.

VPN is a technology that leverages encryption and tunnelling to create a private, secure network connection. The following sections will explore how this is achieved and introduce the powerful tools available on Linux to implement it.

Understand how VPN works

When you connect to a website or any other network service without using a VPN, your device sends data in plain-text packets across the internet. Each packet has a header containing the source and destination IP addresses (like a *to* and *from* address on an envelope) and a payload containing the actual data (the letter inside). Anyone with access to the network path, your **internet service provider** (**ISP**), or a hacker on the same Wi-Fi can inspect these packets and read the contents.

The process transforms dramatically when you use a VPN through a technique called **encapsulation**. The following figure shows a typical remote access VPN, like one you would use to connect to your office

network from home:

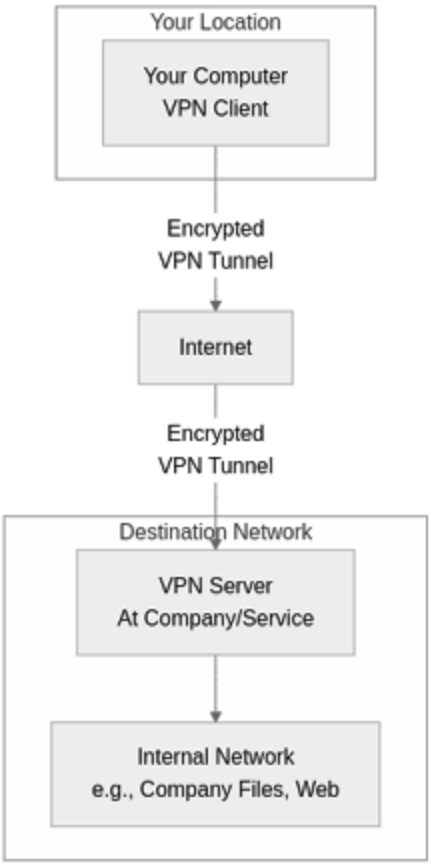


***Figure 10.1**: VPN tunnel setup*

Let us discuss the step-by-step process of the data flow when you connect to a website using a VPN:

1. **Packet creation**: Your device creates a standard IP packet destined for the final target (e.g., example.com at IP 93.184.216.34). The packet contains your real IP address as the source and the destination server's IP.
2. **Encryption and encapsulation**: The VPN client intercepts the packet and encrypts the entire original packet using strong encryption (AES, ChaCha20, etc.). It wraps the encrypted packet inside a new VPN packet. The new outer packet has the VPN server's IP as the destination.
3. **Secure transmission**: The encapsulated packet travels through the

public internet. Internet routers can only see the outer header, which includes your IP as the source address and the VPN server's IP as the destination address. The original encrypted content is unreadable.

4. **Server processing**: The VPN server receives the encapsulated packet. It removes the outer VPN wrapper and decrypts the inner packet using the shared secret keys, restoring the original packet.
5. **Final delivery**: The VPN server forwards the original, restored packet to the destination.
6. **Response**: The destination server sees the packet as coming from the VPN server and responds back to the VPN server. The VPN server encrypts it and sends it back to the VPN client on your computer.

Types of VPNs

VPNs are primarily categorized by their architecture and scale. The most common distinction is between remote access and Site-to-Site configurations. The following are the details:

- A remote access VPN, also known as a **Point-to-Site VPN**, enables a single user device to connect securely to a remote network over the internet. This is the typical setup for employees working from home and will be the focus of our WireGuard configuration guide.

  The following figure describes the Point-to-Site VPN setup where a user connects to the corporate network using a Point-to-Site VPN:

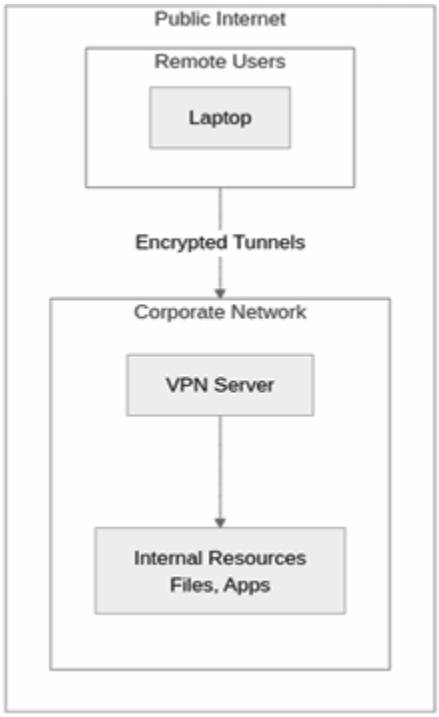


***Figure 10.2**: Point-to-Site VPN*

- In contrast, a Site-to-Site VPN connects two or more entire networks together, such as a corporate headquarters with a branch office, allowing them to communicate as a single, private network. This guide will use strongSwan to demonstrate a Site-to-Site setup.

  The following figure shows the Site-to-Site VPN used to connect two corporate networks:

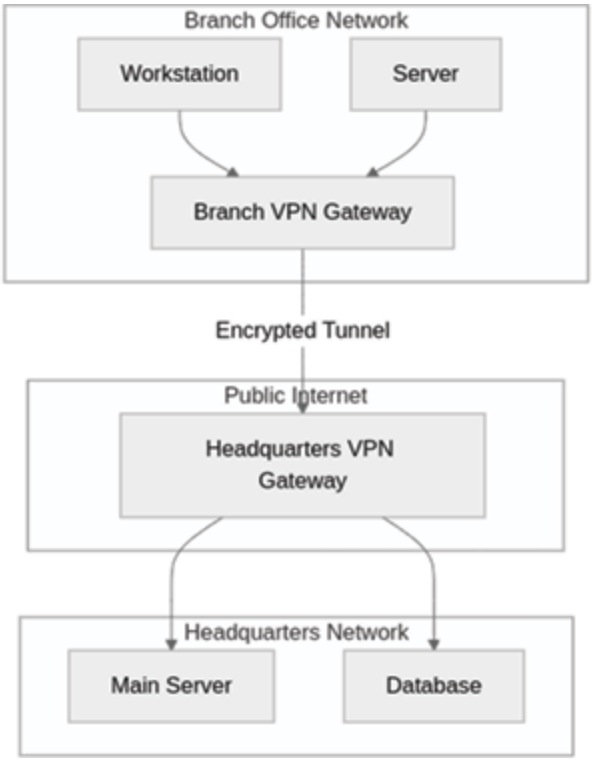


***Figure 10.3**: Site-to-Site VPN*

It is also useful to understand the terminology shift between different VPN technologies. WireGuard, for instance, discards the traditional client and server labels in favor of peers. However, the functional roles of client-to-server (where one peer initiates the connection to another that acts as a permanent endpoint) are still present and map directly to the remote access model.

VPN protocols

While all VPNs serve the same fundamental purpose of creating secure tunnels, they employ different protocols to achieve this goal. Each protocol represents a distinct approach to encryption, authentication, and data transmission, with unique trade-offs in performance, security, and complexity. Understanding these core protocols is essential for selecting the right tool for your specific needs. The following figure describes the VPN protocols mapped to the different networking layers:

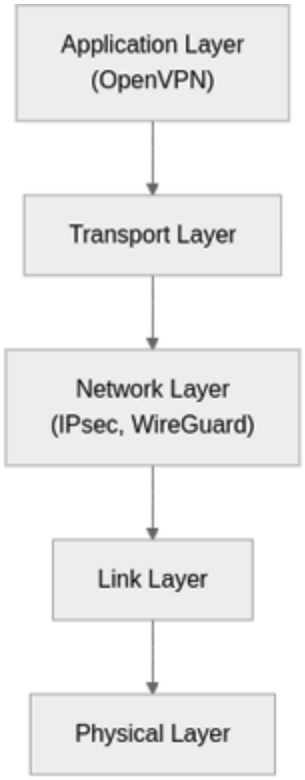


***Figure 10.4**: VPN protocols*

The following is the list of major VPN protocols:

- **OpenVPN**: The established, highly versatile standard. As a user-space protocol, it is renowned for its strong security, stability, and ability to bypass restrictive firewalls by masquerading as standard **Hypertext Transfer Protocol Secure** (**HTTPS**) traffic. Its configurability is a strength, but it can also introduce complexity.
- **IPsec**: A suite of protocols, operating natively at the network layer (Layer 3). This makes it transparent to applications and ideal for Site-to-Site connections and enterprise environments. While powerful, its complexity, with multiple components like **Internet Key Exchange** (**IKE**), **Encapsulating Security Payload** (**ESP**), and **Authentication Header** (**AH**), can be daunting.
- **WireGuard**: The modern contender, designed for simplicity and performance. Its lean, auditable codebase and kernel-level implementation offer blazing-fast speeds and state-of-the-art cryptography by default. It uses a connectionless **User Datagram**

Protocol (**UDP**) design, making it ideal for mobile devices that frequently switch networks

Introduction to WireGuard

WireGuard represents a significant evolution in VPN technology, designed from the ground up to be simpler, faster, and more secure than its predecessors. Unlike older protocols that consist of hundreds of thousands of lines of code, WireGuard is an extremely lean implementation. This minimal codebase makes it easier to audit, debug, and deploy.

The protocol operates at the kernel level on most major operating systems, which provides a substantial performance advantage over user-space implementations. Its design emphasizes modern cryptography, using a curated set of state-of-the-art protocols to ensure robust security without complexity.

A key conceptual shift in WireGuard is its treatment of peers. It moves away from the traditional client-server model, where roles are rigidly defined. Instead, every connected device is simply a peer. This creates a more symmetric network architecture, although in practice, one peer (like a cloud server) is often configured with a stable IP address to act as a central hub for others, forming what is functionally a remote access VPN. This combination of simplicity, high performance, and strong security makes WireGuard an excellent choice for modern networking needs.

WireGuard core concepts

Understanding WireGuard requires familiarity with its fundamental building blocks, which are notably fewer and simpler than those of traditional VPNs. The entire configuration for a peer typically resides in a single, straightforward interface file.

Each WireGuard interface requires two cryptographic keys that work in tandem: a private key, which must be kept secret, and a corresponding public key, which can be shared freely. The public key acts as that peer's persistent identifier, much like a username, while the private key is used to

sign and decrypt data. For communication to be possible, each peer must know the public keys of its intended partners.

Authorization is managed through the **AllowedIPs** parameter, a powerful and flexible concept that serves a dual purpose. Firstly, it acts as a routing directive, specifying which IP addresses should be sent through the WireGuard tunnel. Secondly, it functions as an access control list; a peer is only allowed to send packets from source IP addresses that match the AllowedIPs range configured on the receiving peer. This elegant design means that knowing a public key alone is insufficient; a peer must also have its IP address explicitly permitted by its counterpart to establish successful communication.

Crypto routing table

Crypto routing is a core concept in WireGuard that maps the network routing decisions with the cryptographic identities of its peers. It is a simple but powerful mechanism that replaces the complex access control and routing tables of traditional VPNs. To understand it, we must first understand the dual role of the AllowedIPs setting for a WireGuard peer.

The AllowedIPs directive in a WireGuard peer configuration acts as a mini, protocol-specific routing table and firewall rule, all in one. It serves two distinct purposes:

- **Outbound traffic filter and router**: When Linux sends a packet to the WireGuard interface (e.g., wg0), WireGuard looks up the destination IP address in its list of peers and finds the peer whose AllowedIPs includes that destination. The packet is then encrypted with that specific peer's public key and sent to their endpoint. If no AllowedIPs match the destination, the packet is discarded.
- **Inbound traffic access control list**: When a packet arrives at the WireGuard interface from a peer, WireGuard decrypts it and checks the packet's source IP address. It then verifies that this source IP is within the AllowedIPs range configured for that specific peer. If it matches, the packet is accepted and passed to the network stack. If not, the packet is silently dropped.

This creates a one-to-one mapping for a specific peer identified by its public key to specific IP ranges for which it is only permitted to send and receive traffic. This is the crypto part of the routing decision.

System’s routing decision for a WireGuard peer can be thought of as a two-step process:

1. The system's routing table first decides which network interface should be used to send traffic for a given destination.
2. Once the system routing decision is made and the WireGuard interface is selected as the outgoing interface for the packet, WireGuard’s crypto routing table comes into play to select the peer to which the packet should be sent.

Installation of WireGuard

The installation of WireGuard on Linux leverages the operating system's native package management system for a streamlined process. However, the specific commands differ between distributions due to their underlying package managers and software repositories. The following procedure outlines the steps for both Rocky Linux and Ubuntu.

On Ubuntu and other Debian-based distributions, the installation is performed using the apt package manager. This is done with the command **sudo apt update**. Following this, you can install the necessary software by running:

```
$ sudo apt install wireguard
```

This command will install the WireGuard kernel module, as well as the essential user-space utilities **wg** and **wg-quick**, which are used for configuration and management.

For Rocky Linux or other Red Hat Enterprise Linux-based distributions, the **dnf** package manager is used. The process is similarly straightforward. You can install the WireGuard package by executing:

```
$ sudo dnf install wireguard-tools
```

It is important to note the different package name; wireguard-tools provides the wg and **wg-quick** utilities. On newer kernel versions, the required module is often included, but if it is not present, you may also need to

install **kmod-wireguard**. You can check the **wireguard** kernel module with the following command:

```
$ modinfo wireguard
```

A critical post-installation step on both operating systems involves enabling IP forwarding if the system is intended to act as a VPN server that routes traffic between clients and other networks. This is configured by editing the **sysctl** configuration file:

```
$ sudo sysctl -w net.ipv4.ip_forward=1
```

Furthermore, the built-in firewall (iptables, nftables, or UFW) must be adjusted to allow UDP traffic on the default WireGuard port, 51820, and to permit forwarding of traffic as needed. Once the software is installed and these system configurations are in place, you are ready to proceed with generating cryptographic keys and creating the WireGuard interface configuration files.

Generating cryptographic keys

The foundation of WireGuard's security lies in its use of public-key cryptography, and every deployment begins with the generation of a key pair for each peer. This process is performed entirely through the command line using the utilities installed with WireGuard. The private key must be guarded with the highest level of confidentiality, as anyone who possesses it can impersonate that peer and decrypt all traffic intended for it. The corresponding public key, however, is designed to be shared freely and is used by other peers to encrypt traffic and verify identity.

Key generation is a two-step procedure:

1. **Generate the private key**: This is done typically by running the command **wg genkey** and saving its output to a secure file. This operation creates a 256-bit cryptographic key that remains on the host and is never transmitted.
2. **Derive the public key**: The public key is derived from the private key using the **wg pubkey** command. The standard method is to pipe the contents of the private key file into this utility, which then computes and outputs the associated public key.

A complete key generation sequence for a peer can be done with a single command as follows:

```
$ wg genkey | tee privatekey | wg pubkey > publickey
```

This single command chain securely generates the private key, saves it to a file named **privatekey**, and simultaneously calculates the public key, saving it to a separate file named **publickey**. The contents of the public key file are what you will provide to other peers in their configuration files, while the **privatekey** is placed in the local peer's configuration and its permissions are restricted to prevent unauthorized access.

Configuring a Point-to-Site VPN

This lab provides a practical walkthrough for establishing a functional Point-to-Site VPN using WireGuard. In this common scenario, a server acts as a stable endpoint (the *Site*), and a client machine (the *Point*) connects to it to access protected resources. You can set up a complete WireGuard lab on your Linux laptop using network namespaces to simulate multiple machines. This is a good way to learn WireGuard concepts without needing multiple physical devices.

The following figure describes the lab setup for Point-to-Site VPN with WireGuard:

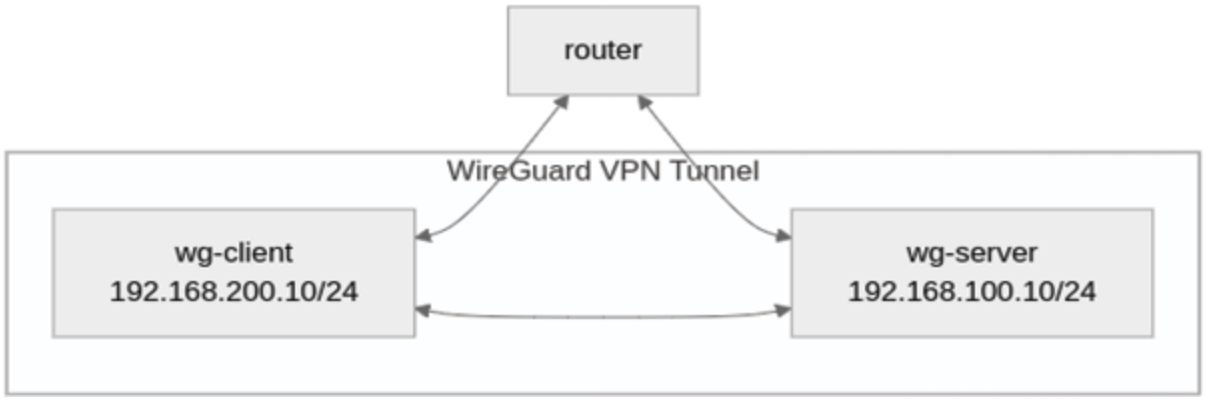


***Figure 10.5**: WireGuard lab setup*

The following steps can be followed to setup WireGuard based Point-to-

Site VPN:

1. **Creating virtual network topology**: We will create three network namespaces to simulate:
   - **wg-server**: The VPN server with a *public* IP.
   - **wg-client**: The client that will connect to the server.
   - **router**: A simulated internet router.
   - **Create network namespaces**:

```
sudo ip netns add wg-server
sudo ip netns add wg-client
sudo ip netns add router
```

   - **Create virtual Ethernet pairs**:

```
sudo ip link add veth-server type veth peer name veth-server-rt
sudo ip link add veth-client type veth peer name veth-client-rt
```

   - **Move interfaces to appropriate namespaces**:

```
sudo ip link set veth-server netns wg-server
sudo ip link set veth-server-rt netns router
sudo ip link set veth-client netns wg-client
sudo ip link set veth-client-rt netns router
```

   - **Configure IP addresses**:

```
sudo ip netns exec wg-server ip addr add 192.168.100.10/24 dev veth-server
sudo ip netns exec router ip addr add 192.168.100.1/24 dev veth-server-rt
sudo ip netns exec wg-client ip addr add 192.168.200.10/24 dev veth-client
sudo ip netns exec router ip addr add 192.168.200.1/24 dev veth-client-rt
```

   - **Bring up all interfaces**:

```
sudo ip netns exec wg-server ip link set veth-server up
sudo ip netns exec wg-server ip link set lo up
```

```
sudo ip netns exec router ip link set veth-server-rt up
sudo ip netns exec wg-client ip link set veth-client up
sudo ip netns exec wg-client ip link set lo up
sudo ip netns exec router ip link set veth-client-rt up
```

- **Enable routing**:

```
sudo ip netns exec router sysctl -w net.ipv4.ip_forward=1
sudo ip netns exec wg-client ip route add 192.168.100.0/24 via 192.168.200.1
sudo ip netns exec wg-server ip route add 192.168.200.0/24 via 192.168.100.1
```

2. **Generate WireGuard keys**:
   - **Generate server keys**:

```
sudo ip netns exec wg-server wg genkey | sudo tee /tmp/server_private.key | wg pubkey | sudo tee /tmp/server_public.key
```

   - **Generate client keys**:

```
sudo ip netns exec wg-client wg genkey | sudo tee /tmp/client_private.key | wg pubkey | sudo tee /tmp/client_public.key
```

3. **Configure WireGuard interfaces**:
   - **Server configuration file** /tmp/wg0-server.conf:

```
[Interface]
PrivateKey = <contents-of-server_private.key>
Address = 10.0.0.1/24
ListenPort = 51820
[Peer]
PublicKey = <contents-of-client_public.key>
Endpoint = 192.168.200.10:51820
AllowedIPs = 10.0.0.2/32
```

   - **Client configuration file** /tmp/wg0-client.conf:

```
[Interface]
```

```
PrivateKey = <contents-of-client_private.key>
Address = 10.0.0.2/32
[Peer]
PublicKey = <contents-of-server_public.key>
Endpoint = 192.168.100.10:51820
AllowedIPs = 10.0.0.0/24
```

4. **Start the WireGuard tunnels**:
   - **Start server**: **sudo ip netns exec wg-server wg-quick up /tmp/wg0-server.conf**
   - **Start client**: **sudo ip netns exec wg-client wg-quick up /tmp/wg0-client.conf**
5. **Test the connection**:
   - **Check WireGuard status with wg show**:

```
sudo ip netns exec wg-server wg show
sudo ip netns exec wg-client wg show
```

   - **Test connectivity using ping**:

```
sudo ip netns exec wg-client ping 10.0.0.1
sudo ip netns exec wg-server ping 10.0.0.2
```

The peers will perform a cryptographic handshake, and the connection should be established immediately. Verify the link with **wg show** on either peer, which should display transferred data and the latest handshake timestamp.

If the connection fails, verify that the server's firewall allows UDP port 51820, ensure all public and private keys are correctly copied, and confirm the server's public IP address in the client's endpoint directive. The **wg show** command provides detailed information about the connection state and any errors.

Introduction to IPsec and strongSwan

IPsec is not a single protocol but rather a comprehensive suite of protocols

designed to secure IP communications by authenticating and encrypting each IP packet in a data stream. A fundamental characteristic of IPsec is its operation at the network layer (Layer 3) of the **Open Systems Interconnect** (**OSI**) model. This layer of operation provides a significant advantage: transparency. Since IPsec secures communication at the IP packet level, it functions independently of the specific applications generating the traffic. This means that no modifications are required for individual applications like web browsers or email clients to benefit from the secured connection, as the encryption and authentication are handled by the operating system's network stack.

This network layer approach also grants IPsec several other key advantages. Its transparency to applications simplifies deployment, while its maturity and standardization have led to native support in a vast array of network infrastructure, including corporate firewalls, routers, and operating systems. Furthermore, the suite employs a robust set of cryptographic algorithms and establishes security associations through a negotiated handshake, providing a very high level of security suitable for enterprise and government use cases.

strongSwan is a leading, full-featured open-source implementation of the IPsec suite for Linux-based systems. It provides a complete toolset for building IPsec VPNs, supporting both the IKEv1 and IKEv2 protocols for key management and the ESP and AH protocols for securing the data itself. As a versatile solution, strongSwan can be used to create both Site-to-Site tunnels, connecting entire networks, and remote access VPNs for individual users, making it a foundational component for complex network security architectures.

Core IPsec concepts

IPsec is not a single protocol, but a framework comprised of several interconnected components that work together to secure IP communications. Understanding these core building blocks, IKE, ESP, and AH, is essential for deploying and troubleshooting IPsec VPNs. IKE.

The IKE protocol is the negotiation engine of IPsec. Before any secure data transfer can occur, two peers must agree on how to protect the

communication. IKE handles this initial negotiation securely over an untrusted network. The modern IKEv2 protocol performs this in two main phases:

- **IKE_SA_INIT (Phase 1)**: This initial exchange negotiates the security parameters for the IKE **Security Association** (**SA**) itself. The two peers agree on encryption and integrity algorithms and, crucially, perform a Diffie-Hellman key exchange to establish a shared secret. This creates a secure channel for the next phase.
- **IKE_AUTH (Phase 2)**: Over the secure channel established in Phase 1, this phase authenticates the identities of the peers using pre-shared keys or digital certificates. Once authenticated, it establishes the first CHILD_SA, which defines the security policies for the actual user data.

The following figure describes the IKE_SA_INIT and IKE_AUTH steps previously involved in establishing the security association:

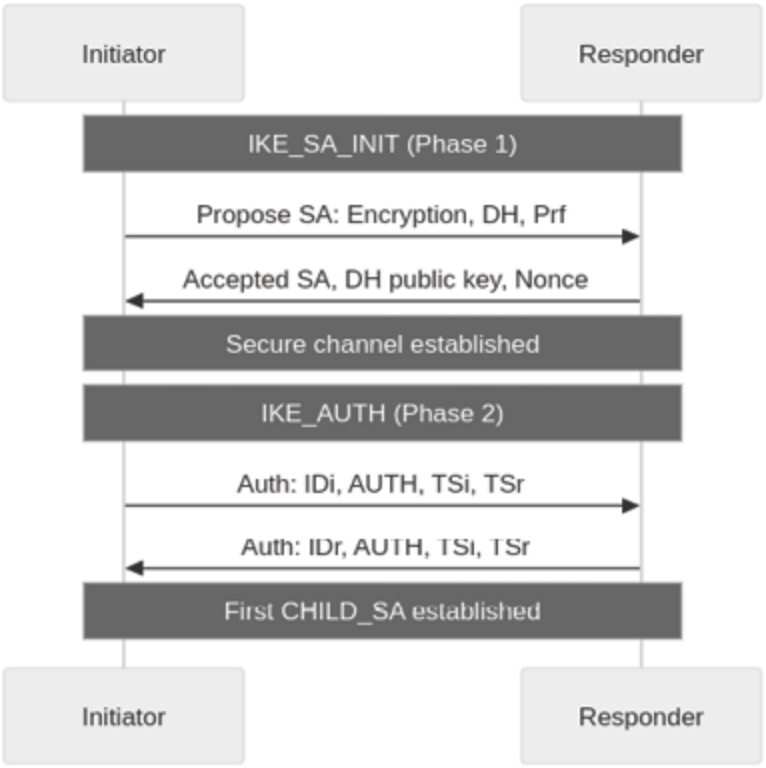


***Figure 10.6**: IPsec initiation*

ESP is the workhorse for data confidentiality in IPsec. It provides three critical security services for the payload of the IP packet:

- **Confidentiality**: Encrypts the packet's payload to prevent eavesdropping.
- **Integrity**: Provides a hash to verify that the data was not modified in transit.
- **Authentication**: Validates that the data originated from the expected peer.

By operating in Tunnel Mode, ESP encapsulates the entire original IP packet, creating a new IP header. This effectively hides the original source and destination addresses, making it ideal for VPNs.

AH provides integrity and authentication, but unlike ESP, it does not provide encryption. Its unique characteristic is that it authenticates the entire IP packet, including the outer IP header. This protects against address spoofing and ensures the packet's path is authentic.

However, because it signs the immutable parts of the IP header, AH is incompatible with **Network Address Translation** (**NAT**). Any change to the IP addresses by a NAT device invalidates the AH authentication hash, causing the packet to be dropped. This limitation is a primary reason ESP is far more commonly used in modern deployments.

IPsec tunnel and transport modes

IPsec can operate in two distinct modes, which define the scope of what gets encrypted and authenticated. The choice between Tunnel Mode and Transport Mode is fundamental and depends entirely on the use case, determining whether IPsec protects only the communication payload or the entire original packet. The details are as follows:

- **Tunnel Mode**: It is the default and most common mode for VPN deployments. In this mode, the entire original IP packet, including both the payload and the original IP header, is encapsulated and protected. The process encrypts the whole original packet and adds a new IP header on top, which contains the source and destination addresses of the VPN gateways.

This mode is essential for Site-to-Site and remote access VPNs because it completely hides the original IP addresses of the communicating hosts. The public internet only sees traffic between the VPN gateways, providing both security and network anonymity. All traffic routed through a gateway-to-gateway VPN tunnel utilizes this mode.

- **Transport Mode**: It provides a more limited scope of protection. In this mode, only the payload of the IP packet, the data containing the TCP or UDP segment, is encrypted and authenticated. The original IP header remains intact and is sent in clear text.

  This mode is primarily used for host-to-host communication where the two endpoints are the direct source and destination of the traffic, and there is no need to hide their IP addresses. It is less common in typical VPN scenarios but can be employed for securing direct connections between servers within a trusted network.

The following figure illustrates the structural difference in how packets are built in each mode:

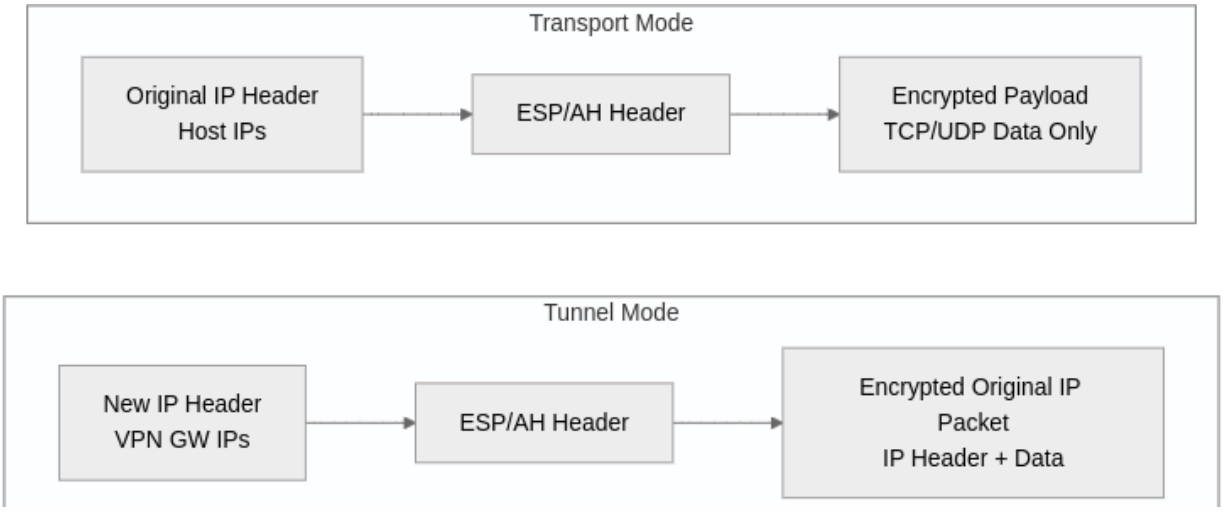


***Figure 10.7**: IPsec modes*

Tunnel Mode is designed for network-level protection between gateways, creating a secure tunnel for all traffic. Transport Mode is designed for endpoint-level protection, securing communication between two specific hosts. For building traditional VPNs that connect networks or allow remote

users to access a corporate network, Tunnel Mode is the appropriate and standard choice.

Installing strongSwan

Installing strongSwan is straightforward thanks to its inclusion in the official repositories of all major Linux distributions. The installation command varies slightly depending on your package manager, but the process remains simple and consistent.

For Ubuntu and Debian Systems, use the following apt package manager to install both the strongSwan core package and the PKI utilities. The latter provides helpful tools for managing certificates, which are essential for enterprise-grade IPsec configurations:

```
sudo apt update
sudo apt install strongswan strongswan-pki
```

On Rocky Linux and Red Hat-based distributions, use the **dnf** package manager. The main strongSwan package contains the essential IPsec functionality needed for most deployments:

```
sudo dnf install strongswan
```

After installation, verify that strongSwan is correctly installed by checking its version. This command confirms that the IPsec command-line utility is available and functioning:

```
ipsec --version
```

The output should display version information for strongSwan and its compiled features, confirming a successful installation. At this point, the strongSwan service is installed but not yet activated. The next section will guide you through the configuration process to establish functional IPsec tunnels.

Authentication methods of PSK vs. certificates

The security of an IPsec VPN hinges on how the participating peers reliably verify each other's identities. strongSwan supports two primary

authentication methods, each with distinct advantages and trade-offs between simplicity and scalability:

- **Pre-shared keys (PSK)**: PSK authentication relies on a single, shared secret string configured identically on both VPN peers. This method is conceptually simple and quick to implement, making it ideal for testing environments, lab setups, or simple Site-to-Site connections between a limited number of gateways. However, its simplicity introduces significant limitations for larger deployments. The same key must be distributed to every peer, creating a serious security risk if any single device is compromised. Furthermore, managing and securely rotating keys becomes increasingly impractical as the number of connections grows, making PSK unsuitable for scalable enterprise environments.
- **Certificates**: The X.509 certificate represents the enterprise standard for authentication, leveraging PKI to provide robust and scalable security. In this model, a trusted **Certificate Authority** (**CA**) issues digital certificates to each peer, which are then used for authentication. This approach offers several critical advantages. It provides strong mutual authentication, as each peer validates the other's certificate against the trusted CA. Scalability is greatly enhanced, as new peers can be added by simply issuing them a unique certificate without reconfiguring existing ones. This model also supports fine-grained access control and non-repudiation. While establishing a PKI requires more initial effort, it is the definitive choice for any production environment requiring security, manageability, and growth.

Configuring a Site-to-Site VPN with PSK

This hands-on lab guides you through establishing a secure Site-to-Site IPsec VPN between two networks using PSK authentication. This is a common scenario for connecting branch offices or cloud networks.

You can create a functional IPsec lab environment on a single Linux machine using Docker containers to simulate multiple hosts. This approach allows you to build and test a complete Site-to-Site VPN without needing multiple physical devices. Two Docker container nodes, node1 and node2, will be used to simulate the sites. The internal network will be configured

on the loopback interfaces of the containers.
The following figure describes the setup for our Site-to-Site VPN lab:

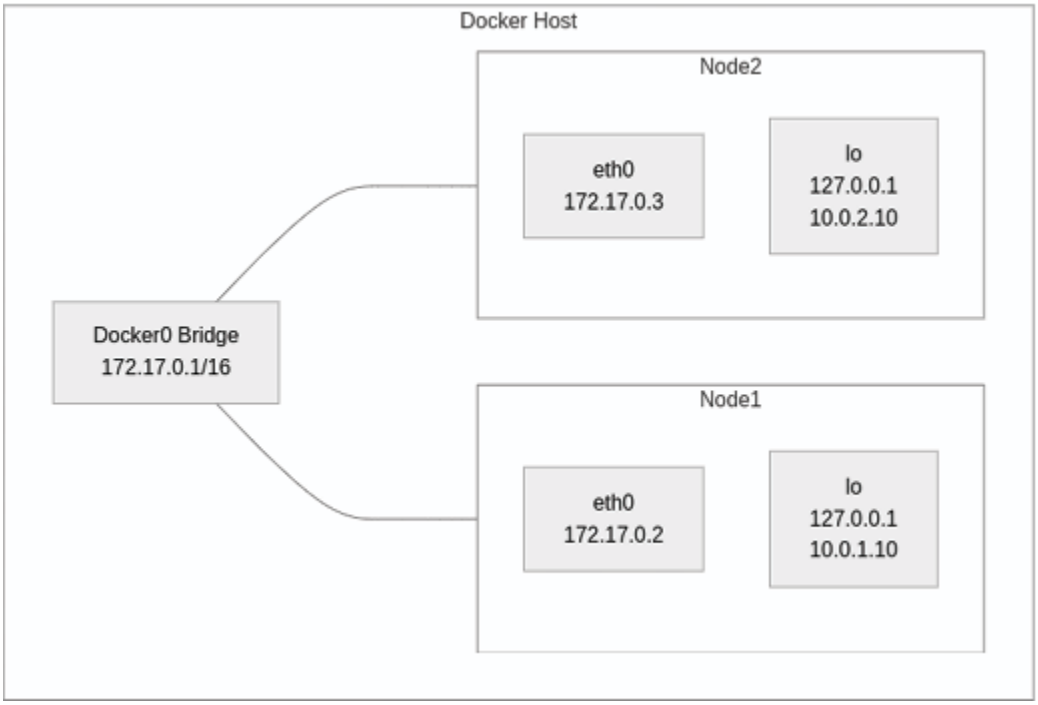


***Figure 10.8**: Lab setup for IPsec with Docker containers*

All the configuration for the nodes will be created in the config directory:

```
├── Dockerfile
└── config
    ├── ipsec.conf
    ├── ipsec.secrets
    ├── node1_ipsec.conf
    ├── node1_ipsec.secret
    ├── node1_ipsec.secrets
    ├── node2_ipsec.conf
    ├── node2_ipsec.secret
    ├── node2_ipsec.secrets
    └── strongswan.conf
```

To set up the IPsec lab, we will proceed with the following steps:

1. We begin by creating the Docker image for the IPsec node for our lab environment.

   Use the following **Dockerfile** to build the image:

```
FROM ubuntu:rolling
ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y \
vim \
iproute2 \
iptables \
iputils-ping \
libcharon-extauth-plugins \
libcharon-extra-plugins \
libstrongswan libstrongswan-extra-plugins \
libstrongswan-standard-plugins \
strongswan \
strongswan-charon \
strongswan-libcharon \
strongswan-pki \
strongswan-starter \
strongswan-swanctl && rm -rf /var/lib/apt/lists/*
```

   To build the Docker image, use the following command:

```
docker build -t ipsecnodepsk:latest .
```

2. Node1 and node2 are the two sites we want to connect using the VPN. We will create the **ipsec.conf** file for the nodes.

   For node1 **config/node1_ipsec.conf**:

```
config setup
    # Uncomment to enable logging to syslog
    # charondebug="ike 2, knl 2, cfg 2"
    uniqueids=no
```

```
conn my-site-to-site
    # Left side (our Docker container's perspective)
    left=172.17.0.2
    leftsubnet=10.0.1.0/24

    # Right side (remote VPN peer's perspective)
    right=172.17.0.3
    rightsubnet=10.0.2.0/24

    # Authentication methods
    authby=secret
    auto=start
    keyexchange=ikev2
    type=tunnel
    compress=no
    ikelifetime=60m
    keylife=20m

    # Use vti for route-based VPN (needs kernel support)
    # vti=vti0
    install_routes=yes # strongswan manages the routing tables
    ike=aes256-sha256-modp2048!
    esp=aes256-sha256!
```

3. The **ipsec.conf** for node2 looks symmetric with the left and the right reversed.

   For node2 **config/node2_ipsec.conf**:

```
config setup
    # Uncomment to enable logging to syslog
    # charondebug="ike 2, knl 2, cfg 2"
    uniqueids=no
```

```
conn my-site-to-site
    # Left side (our Docker container's perspective)
    left=172.17.0.3
    leftsubnet=10.0.2.0/24

    # Right side (remote VPN peer's perspective)
    right=172.17.0.2
    rightsubnet=10.0.1.0/24

    # Authentication methods
    authby=secret
    auto=start
    keyexchange=ikev2
    type=tunnel
    compress=no
    ikelifetime=60m
    keylife=20m

    # Use vti for route-based VPN (needs kernel support)
    # vti=vti0
    install_routes=yes # strongswan manages the routing tables
    ike=aes256-sha256-modp2048!
    esp=aes256-sha256!
```

4. Create the config file for the ipsec shared secret. The files are symmetric for node1 and node2, with the left and the right IP reversed:

   For node1 **config/node1_ipsec.secrets**:

```
# <left-ip> <right-ip>: PSK <secret>
172.17.0.2 172.17.0.3 : PSK "SuperSecretSharedKey123!@#"
```

   For node1 **config/node2_ipsec.secrets**:

```
# <left-ip> <right-ip>: PSK <secret>
172.17.0.3 172.17.0.2 : PSK "SuperSecretSharedKey123!@#"
```

5. Create the configuration file for strongSwan with the following content. This file is common for both the nodes:

```
charon {
  load_modular = yes
  plugins {
    include strongswan.d/charon/*.conf
    # Explicitly ensure sha2 is loaded for the IKE daemon
    sha2 {
      load = yes
    }
  }
}
libstrongswan {
  plugins {
    include strongswan.d/*.conf
    # Explicitly ensure sha2 is loaded for the library functions
    sha2 {
      load = yes
    }
  }
}
```

6. Now you can start the 2 Docker containers with the following command:

```
for i in {1..2}; do
      node=node$i;
      docker run \
      --rm=true -d \
      --name=${node} \
```

```
    --privileged \
    -v ${PWD}/config/${node}_ipsec.conf:/etc/ipsec.conf \
    -v ${PWD}/config/${node}_ipsec.secret:/etc/ipsec.secret
    ipsecnodepsk:latest \
    ipsec start –-nofork
done
```

7. Once the containers for the nodes are up, check the status of **ipsec** tunnel with the following command:

   ipsec status

   You should see a similar output:

   root@f77805420da9:/# ipsec status
   Security Associations (2 up, 0 connecting):
   my-site-to-site[2]: ESTABLISHED 7 minutes ago, 172.17.0.3[172.17.0.3]...172.17.0.2[172.17.0.2]
   my-site-to-site{5}: INSTALLED, TUNNEL, reqid 1, ESP SPIs: c8a0dd75_i ca0c1c02_o
   my-site-to-site{5}: 10.0.2.0/24 === 10.0.1.0/24
   my-site-to-site[1]: ESTABLISHED 7 minutes ago, 172.17.0.3[172.17.0.3]...172.17.0.2[172.17.0.2]
   my-site-to-site{6}: INSTALLED, TUNNEL, reqid 1, ESP SPIs: c08a2acc_i c04b2ea5_o
   my-site-to-site{6}: 10.0.2.0/24 === 10.0.1.0/24
   root@f77805420da9:/#

8. Use the following command to add the local network IPs:

   For node1:

   ip addr add dev lo 10.0.1.10/24

   For node2:

   ip addr add dev lo 10.0.1.20/24

9. Check the connectivity from node1 to node2 using **ping**:

   On node1:

```
ping 10.0.2.10
```

On node2:

```
ping 10.0.1.10
```

10. On the host, run a **tcpdump** on the Docker bridge to ensure that the traffic is encrypted:

```
tcpdump –n –i docker0
tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
listening on docker0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
18:58:36.302772 IP 172.17.0.3 > 172.17.0.2: ESP(spi=0xcf4d7767,seq=0xb), length 136
18:58:36.302844 IP 172.17.0.2 > 172.17.0.3: ESP(spi=0xc0eaea9a,seq=0xb), length 136
18:58:37.326810 IP 172.17.0.3 > 172.17.0.2: ESP(spi=0xcf4d7767,seq=0xc), length 136
18:58:37.326883 IP 172.17.0.2 > 172.17.0.3: ESP(spi=0xc0eaea9a,seq=0xc), length 136
18:58:38.350855 IP 172.17.0.3 > 172.17.0.2: ESP(spi=0xcf4d7767,seq=0xd), length 136
18:58:38.350931 IP 172.17.0.2 > 172.17.0.3: ESP(spi=0xc0eaea9a,seq=0xd), length 136
```

This lab provides a complete, self-contained IPsec testing environment that demonstrates the core concepts of VPN tunneling, encryption, and routing, all within a single Linux machine.

Introduction to PKI and Certificate Authority

While PSK works adequately for simple scenarios, they present significant security and scalability challenges in enterprise environments. As organizations grow, managing shared secrets across multiple sites becomes cumbersome and increases the risk of key compromise. This is where PKI

and X.509 certificates provide a superior authentication framework.

Certificates offer fundamental advantages over PSKs that make them essential for production environments. They provide true mutual authentication where each party cryptographically proves their identity, eliminating the risk of impersonation. Scalability is dramatically improved since adding new peers only requires issuing a new certificate rather than distributing a shared secret to all existing devices. Certificates also enable fine-grained access control and non-repudiation, creating a secure and manageable foundation for large-scale VPN deployments.

PKI hierarchy

A robust PKI operates on a hierarchical trust model that forms a chain of verification. At the top sits the Root CA, a master entity that signs all subordinate certificates and forms the ultimate trust anchor. Below Root CA, Server Certificates are issued to VPN gateways, authenticating them to connecting clients. Finally, Client Certificates are distributed to end-user devices or remote gateways, enabling them to authenticate to the VPN servers. This hierarchical structure means that compromising one client certificate does not affect other connections, unlike PSK compromise, which affects all peers.

The following figure describes the certificate authority setup for VPNs:

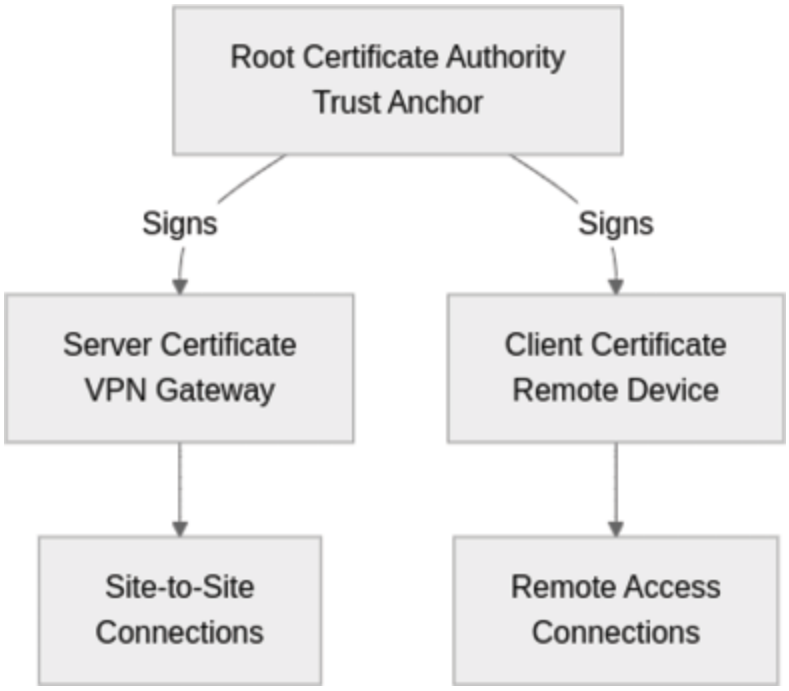


***Figure 10.9**: IPsec certificate setup*

Using strongswan-pki

The strongSwan suite includes the **strongswan-pki** package, which provides command-line tools to establish a complete PKI. These tools allow you to generate a private Root CA key, create a self-signed root certificate, and issue signed certificates for servers and clients. The process involves generating key pairs, creating certificate signing requests, and issuing certificates with appropriate extensions and validity periods. This integrated toolset eliminates the need for external PKI services for many deployments, making certificate-based authentication accessible for organizations of all sizes.

In the next lab, we will put this theory into practice by creating our own CA and issuing certificates for a secure remote access VPN configuration.

Configuring remote access with certificates

In this section, we will set up a certificate-based IPsec lab using strongSwan with a full PKI on a single Linux host using Docker containers to simulate

multiple hosts. Two Docker container nodes, node1 and node2, will be used to simulate the sites. The internal network will be configured on the loopback interfaces of the containers, like the lab on **ipsec** with PSK.

All the configuration files for both nodes will be created in a directory config, and certificates will be stored in certs directory is as follows:

```
├── Dockerfile
├── certs
│   ├── ca-cert.pem
│   ├── ca-key.pem
│   ├── left-cert.pem
│   ├── left-key.pem
│   ├── right-cert.pem
│   └── right-key.pem
└── config
    ├── ipsec.conf
    ├── ipsec.secrets
    ├── node1_ipsec.conf
    ├── node1_ipsec.secrets
    ├── node2_ipsec.conf
    ├── node2_ipsec.secrets
    └── strongswan.conf
```

The steps for setting up IPsec Site-to-Site VPN with certificates are as follows:

1. We will use the same Dockerfile used to setup the ipsec node in the ipsec PSK lab, build an image with the **ipsecnodecerts:latest** tag:

   ```
   docker build  -t ipsecnodecerts:latest
   ```

2. We will create the certificates for the CA and the left and the right site, which will be used to set up node1 and node2, respectively:

   ```
   ipsec pki --gen --type rsa --size 4096 --outform pem > ca-key.pem
   ipsec pki --self --ca --lifetime 3650 --in ca-key.pem \
      --type rsa --dn "C=IN, O=IPsec Lab, CN=IPsec Lab Root CA"
   ```

```
\
    --outform pem > ca-cert.pem
ipsec pki --gen --type rsa --size 4096 --outform pem > left-key.pem
ipsec pki --pub --in left-key.pem --type rsa | \
    ipsec pki --issue --lifetime 1825 \
    --cacert ca-cert.pem \
    --cakey ca-key.pem \
    --dn "C=IN, O=IPsec Lab, CN=left.site.lab" \
    --san left.site.lab \
    --outform pem > left-cert.pem
ipsec pki --gen --type rsa --size 4096 --outform pem > right-key.pem
ipsec pki --pub --in right-key.pem --type rsa | \
    ipsec pki --issue --lifetime 1825 \
    --cacert ca-cert.pem \
    --cakey ca-key.pem \
    --dn "C=IN, O=IPsec Lab, CN=right.site.lab" \
    --san right.site.lab \
    --outform pem > right-cert.pem
```

3. The **ipsec.conf** for the nodes is set up as follows:

   For node1 (left) **config/node1_ipsec.conf**:

```
config setup
    charondebug="ike 2, cfg 2"

conn %default
    keyexchange=ikev2
    ike=aes256-sha2_256-modp2048!
    esp=aes256-sha2_256!
    leftfirewall=yes
    leftsubnet=10.0.1.0/24
```

```
    rightsubnet=10.0.2.0/24

conn left-to-right
    left=172.17.0.2
    leftid="C=IN, O=IPsec Lab, CN=left.site.lab"
    leftcert=left-cert.pem
    leftsendcert=always
    right=172.17.0.3
    rightid="C=IN, O=IPsec Lab, CN=right.site.lab"
    auto=start
```

For node2 (right) **config/node2_ipsec.conf**:

```
config setup
    charondebug="ike 2, cfg 2"

conn %default
    keyexchange=ikev2
    ike=aes256-sha2_256-modp2048!
    esp=aes256-sha2_256!
    leftfirewall=yes
    leftsubnet=10.0.1.0/24
    rightsubnet=10.0.2.0/24

conn right-to-left
    right=172.17.0.3
    rightid="C=IN, O=IPsec Lab, CN=right.site.lab"
    rightcert=right-cert.pem
    rightsendcert=always
    left=172.17.0.2
    leftid="C=IN, O=IPsec Lab, CN=left.site.lab"
    auto=start
```

4. The **ipsec.secrets** is setup with the following configuration:

   For node1 **config/node1_ipsec.secrets**:

   ```
   : RSA "/etc/ipsec.d/private/left-key.pem"
   ```

   For node2 **config/node2_ipsec.secrets**:

   ```
   : RSA "/etc/ipsec.d/private/right-key.pem"
   ```

5. The certificates generated in step 1 should be copied to the respective paths:

   - The CA cert and key should be copied to **/etc/ipsec.d/cacerts**.
   - The certs for the left and the right sites should be copied to **/etc/ipsec.d/certs** on node1 and node2, respectively.
   - The private key for the left and right certs should be copied to **/etc/ipsec.d/private**.

6. Create the configuration file for **strongswan** with the following content. This file is common for both the nodes:

```
charon {
  load_modular = yes
  plugins {
    include strongswan.d/charon/*.conf
    # Explicitly ensure sha2 is loaded for the IKE daemon
    sha2 {
      load = yes
    }
  }
}
libstrongswan {
  plugins {
    include strongswan.d/*.conf
    # Explicitly ensure sha2 is loaded for the library functions
    sha2 {
      load = yes
```

```
    }
  }
}
```

7. Now you can start the 2 Docker containers with the following command:

```
for i in {1..2}; do
   node=node$i
   docker run --rm=true -d --name=${node} --privileged \
          -v ${PWD}/config/${node}_ipsec.conf:/etc/ipsec.conf \
          -v ${PWD}/config/${node}_ipsec.secrets:/etc/ipsec.secrets \
          -v ${PWD}/certs:/etc/ipsec.d/certs \
          -v ${PWD}/certs:/etc/ipsec.d/private \
          -v ${PWD}/certs:/etc/ipsec.d/cacerts \
          ipsecnodecerts:latest ipsec start --nofork
done
```

8. To check the status of **ipsec** tunnel, you can use the following command:

```
ipsec status
```

9. Use the following command to add the local network IPs:

   For node1:

```
ip addr add dev lo 10.0.1.10/24
```

   For node2:

```
ip addr add dev lo 10.0.1.20/24
```

10. Check the connectivity from node1 to node2 using **ping**:

    On node1:

```
ping 10.0.2.10
```

    On node2:

```
ping 10.0.1.10
```

This lab demonstrates a simple way to learn **ipsec** with certificates on a single Linux machine.

WireGuard vs. strongSwan and when to use which

Having explored both WireGuard and strongSwan in depth, a crucial question remains: *which technology should you choose for your specific project?* The decision is not about identifying a universally superior option, but rather about selecting the right tool for your requirements, constraints, and environment.

You should opt for WireGuard when your priority is a lean, high-performance VPN that is remarkably simple to configure and manage. Its minimal codebase and modern cryptographic defaults make it ideal for scenarios demanding maximum throughput and low latency, such as connecting cloud instances or securing high-bandwidth connections. It excels in environments with dynamic peer connections, including laptops and mobile devices that frequently change networks, thanks to its roaming capabilities. WireGuard is the preferred choice when you control both endpoints of the connection and can leverage its straightforward key-based authentication without needing complex negotiation.

Select strongSwan when your project requires interoperability with existing enterprise network infrastructure. If you need to connect to industry-standard firewalls and routers from vendors like Cisco, Juniper, or Palo Alto Networks, IPsec is the established, universal language. It is essential for environments with strict regulatory compliance requirements that mandate specific cryptographic suites or security frameworks. strongSwan is the solution for complex scenarios requiring fine-grained control over negotiation parameters, and it integrates seamlessly with existing PKI, leveraging X.509 certificates for scalable authentication in large organizations.

In general, you can let WireGuard's simplicity and speed drive your greenfield projects and internal connections, while relying on strongSwan's power and interoperability to integrate with enterprise ecosystems and meet complex compliance demands. Both are excellent tools; the context determines which wrench fits the bolt.

Security best practices

Deploying a VPN is a significant step towards securing your network’s traffic, but a functioning tunnel is not automatically a secure one. The configuration choices you make directly impact your resistance to attacks. Adhering to a set of core security practices is essential for building a VPN that is not only functional but also resilient.

For WireGuard, its simplicity is a security feature, but it must be deliberately configured. Beyond using key pairs, you should consider employing pre-shared keys for an additional layer of symmetric encryption, which provides a degree of post-quantum protection. Furthermore, adopt a minimalist approach to peer configuration. The AllowedIPs directive should be as specific as possible, and the list of peers in a server's configuration should only include those that are strictly necessary. This principle of minimal access should be reinforced with strict host-based firewall rules, restricting inbound WireGuard connections to known, trusted source IP addresses where feasible.

When working with strongSwan and IPsec, the attack surface is broader due to the protocol's complexity. The first and most critical decision is to prefer X.509 certificates over pre-shared keys for authentication. Certificates are more scalable, manageable, and provide strong authentication. Always configure the IKE and ESP proposals to use strong, modern cryptographic suites, explicitly disabling weak algorithms and legacy protocols like IKEv1, which is known to have vulnerabilities. As with any critical infrastructure, rigorous patch management is non-negotiable; keep the strongSwan software and the underlying operating system updated to protect against known exploits.

These technology-specific measures should be complemented by general system hardening principles. A stateful firewall is your first line of defence and must be configured to only permit the essential VPN ports. Consistent software updates across the entire system to patch security holes that could be used to compromise the VPN endpoint itself. Finally, the principle of least privilege should govern all access: every user, process, and network connection should operate with the minimal set of permissions required to

function, thereby limiting the potential damage from a breach.

Conclusion

This chapter has guided you through a comprehensive journey from understanding the fundamental principles of VPNs to the practical implementation of two of the most significant VPN technologies on the Linux platform. We began by exploring the core concepts of tunnelling and encryption that are the foundation of all VPNs, established the critical differences between protocols, and progressed to hands-on configuration of both WireGuard and strongSwan IPsec solutions.

In today's interconnected world, VPNs have evolved from a niche tool into a critical component of modern network architecture. They provide the essential secure connectivity for remote work, enable cloud infrastructure integration, protect privacy against surveillance, and ensure business continuity through reliable Site-to-Site connections. As threats to network security grow and privacy concerns intensify, the ability to implement and manage robust VPN solutions remains an indispensable skill for any system administrator or network engineer.

In the next chapter, we will discuss networking in containers with Docker and Kubernetes. We will understand how SDN works and the components of an SDN-based virtual network. We will also discuss the popular cloud environments and explore how the concepts of virtual networks that we have built over the chapters apply in cloud networking.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 11

Virtual Networking and Cloud Integration

Introduction

If you have ever used a cloud service, run a container, or connected to a corporate network, you have relied on virtual networks. A virtual network is a software-based representation of a physical network. It creates isolated network environments on top of a shared physical network infrastructure. Imagine you need to create a private, isolated network for a project, but you must share the same physical wires and routers with everyone else in your building. Technologies like **Virtual eXtensible Local Area Network** (**VXLAN**) solve this by wrapping your private network data inside a standard internet packet, creating a secure tunnel through the shared network. In this chapter, we will move beyond the basic virtual cables and switches to explore the sophisticated systems that power modern cloud applications.

We will see how a programmable virtual switch like **Open vSwitch** (**OVS**) can be instructed to create these tunnels, manage traffic, and enforce rules with incredible flexibility, forming the backbone of many cloud datacenters. We will also explore the **Virtual Private Clouds** (**VPCs**) of major cloud providers like AWS and Google Cloud.

Structure

This chapter covers the following topics:

- Virtual networks and containers
- Overlay networking with VXLAN and GRE
- Open vSwitch as an overlay engine
- Programming Open vSwitch
- Container network interface
- Cloud networking

Objectives

The objective of this chapter is to provide you with a comprehensive understanding of the core technologies that form the backbone of modern virtualized and cloud-native networking. You will be learning the concepts of overlay networking, how protocols like VXLAN create logical networks over a physical underlay to overcome scale and isolation limitations. We will then explore how these principles are implemented in practice through software-defined networking with OVS, giving you insight into a programmable data plane used in many data centers. The chapter will then explore the practical application of these networks in the container world, where you will learn to differentiate between the architectures and trade-offs of major Kubernetes **container network interface** (**CNI**) plugins like Flannel and Calico. Finally, you will learn the fundamental components and services of Amazon VPC, Azure Virtual Network, and Google Cloud VPC. By the end of this chapter, you will be able to connect the dots from a single virtual tunnel to a global cloud network.

Virtual networking and containers

At its heart, containerization is a story about efficiency and agility. It allows developers to package an application with all its dependencies into a single,

standardized unit that can run reliably anywhere. This *build once, run anywhere* paradigm is what has made technologies like Docker and Kubernetes so transformative. But this revolution rests on a critical, often invisible, foundation of isolated virtual networks.

Unlike a **virtual machine** (**VM**), which virtualizes an entire server, a container is just an isolated process on a host operating system. It leverages kernel features like cgroups to manage resources and namespaces to provide isolation. The network namespace is the key piece here. It gives each container its own private network stack, its own network interfaces, **Internet Protocol** (**IP**) addresses, routing tables, and firewall rules. This means *Container A* can listen on port 80, and *Container B* on the same host can also listen on port 80, with no conflict.

This is where isolated virtual networks become essential. Without them, containers would be stuck in their own isolated silos on a single host, unable to communicate to form a distributed application. A web server container could not talk to its database container if they were on different hosts. The solution is to create a virtual network that seamlessly connects these isolated network namespaces across the entire cluster. This virtual network must fulfil several core requirements to be truly effective:

- **Unique identity**: Every container must have its own unique, routable IP address so that other containers can find and talk to it directly.
- **Cross-host communication**: The network must allow a container on node1 to communicate with a container on node2 as if they were on the same local network, regardless of the underlying physical network.
- **Security and isolation**: Just as we isolate the container's filesystem, we must isolate its network. Applications for different projects or tenants must be able to run without interfering with or eavesdropping on each other's traffic.

Therefore, the virtual network is not merely an add-on for containers; it is the central nervous system that allows a group of isolated processes to function as a cohesive, scalable, and resilient application. The technologies we will explore in this chapter, from Docker's own networks to advanced Kubernetes CNI plugins, are all different solutions to this same fundamental problem: how to build a robust, performant, and secure network for a

massively distributed system of isolated processes.

Overlay networking with VXLAN and GRE

As we have established, containers rely on network isolation, but *how do we connect these isolated network namespaces when they are spread across hundreds of different physical machines?* The physical network underneath was not designed for the dynamic, high-density world of containers, where IPs are constantly changing. This is where overlay networks provide a flexible and robust solution.

Think of an overlay network as a network on top of a network. It creates a virtual Layer 2 (Ethernet) or Layer 3 (IP) network that is decoupled from the underlying physical network, called the underlay. The underlay is simply the IP transport that provides reachability between all your physical servers. The overlay is the virtual network where your containers live and communicate.

This decoupling is powerful. It allows you to:

- **Break physical limits**: Create a single, flat virtual network that spans multiple physical subnets or even data centers, overcoming the limitations of VLANs and physical topology.
- **Isolate tenants and applications**: Run multiple, completely isolated virtual networks over the same shared physical infrastructure, each with its own IP scheme.
- **Enable mobility**: A container's IP address can remain the same even if it moves to a different physical host, because its IP belongs to the overlay, not the physical network.

The foundation of this virtual network is built on top of encapsulation, the process of wrapping a container's original network packet inside a new packet to send it across the underlay. Let us examine the two key technologies that provide this feature.

Virtual eXtensible LAN

VXLAN is the modern, industry-standard protocol for building large-scale

overlay networks in data centers. Traditional VLANs use a 12-bit identifier, allowing for only 4,094 unique networks. This is insufficient for large multi-tenant cloud environments. VXLAN solves this with a massive 24-bit **VXLAN Network Identifier** (**VNI**), supporting over 16 million logical networks.

The following figure describes the VXLAN header and packet structure:

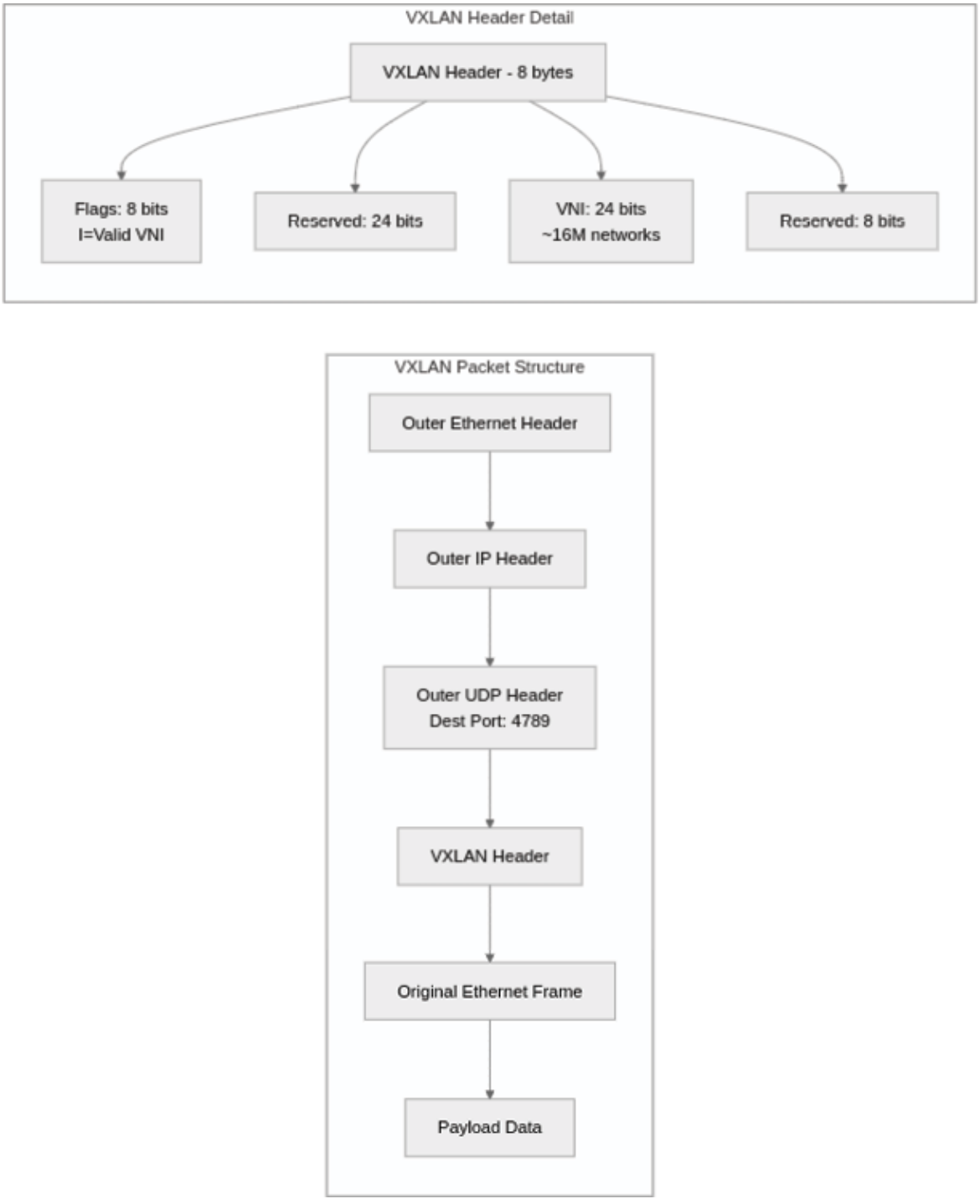


***Figure 11.1**: VXLAN packet structure*

The fundamental difference between VXLAN and VLAN lies in their scale and encapsulation method, which is directly reflected in their header structures. A traditional VLAN header uses a 4-byte tag within the existing

Ethernet frame. This tag contains a 12-bit **VLAN Identifier** (**VID**), a field that limits the number of possible virtual networks to 4,094, a number that is easily exhausted in a large multi-tenant cloud environment.

In contrast, VXLAN employs a robust encapsulation strategy, wrapping the entire original Layer 2 frame within a new **User Datagram Protocol** (**UDP**)/IP packet. The critical VXLAN header is an 8-byte structure that sits between the outer UDP header and the original inner frame. Its layout in bit blocks is as follows:

- 8-bit flags field
- 24 bits of reserved space
- 24-bit VNI
- 8 reserved bits

This 24-bit VNI field is the key to VXLAN's scalability, providing support for over 16 million distinct virtual networks compared to VLAN's limited 4,094.

This structural difference creates a clear divergence in their application. VLAN operates as a simple modification to a frame within a Layer 2 domain, while VXLAN creates a full Layer 3 overlay network, enabling massive multi-tenancy and freeing the virtual network from the constraints of the physical network topology.

When a container on *Host A* sends a packet to a container on *Host B*, the original Ethernet frame (from the container) is entirely wrapped as data inside a new UDP/IP packet. This encapsulation is performed by a software or hardware component called a **VXLAN Tunnel Endpoint** (**VTEP**), which resides on each host. The VTEP is the gateway between the overlay and the underlay. In the following figure, the VTEP can be seen connecting the VXLAN tunnels over the underlay of Layer 3 switches. The VNIs identify the virtual networks to which the VMs or containers are connected.

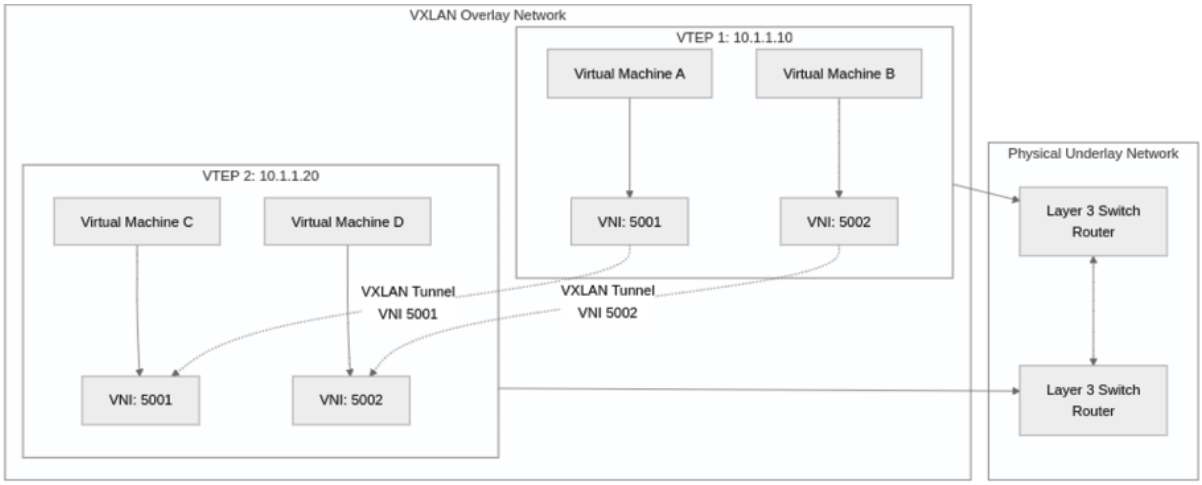


***Figure 11.2**: Underlay and overlay in VXLAN*

VNI is added to the new packet header, identifying which logical network the traffic belongs to. A host can have multiple VTEPs for different VNIs. This new UDP packet is then sent over the standard IP underlay network to the IP address of the destination host's VTEP.

At the destination, VTEP on Host B receives the packet, strips off the outer UDP/IP header, which was used to route the encapsulated packet in the underlay network. The VTEP looks at the VNI to identify the virtual network to which the decapsulated packet should be injected. The decapsulated packet is then injected into the correct virtual network, where the original Ethernet frame is forwarded to the target container.

VXLAN thus creates a virtual Layer 2 network by tunnelling Layer 2 frames over a Layer 3 network, enabling massive scale and tenant isolation.

Generic Routing Encapsulation

Generic Routing Encapsulation (**GRE**) is a simpler, more general-purpose tunnelling protocol that predates VXLAN. GRE is a protocol-agnostic tunnel. It can encapsulate a wide variety of network layer protocols (including IPv4, IPv6, etc.) inside a virtual Point-to-Point link, represented by another IP packet. Like VXLAN, GRE takes the original packet and adds a new header. However, the GRE header is much simpler than the VXLAN header. It lacks a dedicated field like the VNI for large-scale segmentation.

Since the protocol itself does not provide segmentation, this responsibility falls to the systems managing the tunnels. One tunnel per virtual network is the most straightforward approach for providing isolated networks with GRE tunnels. If you need three isolated virtual networks (e.g., for red, blue, and green tenants), you create three separate GRE tunnels between the two endpoints. The details are as follows:

- **Tunnel 1**: **gre_tunnel_red**
- **Tunnel 2**: **gre_tunnel_blue**
- **Tunnel 3**: **gre_tunnel_green**

The isolation is maintained because the operating system or network device treats each tunnel as a separate virtual interface. When a packet for the red network arrives on the **gre_tunnel_red** interface, it is automatically associated with that network. This lack of a native identifier has significant operational consequences:

- **Management overhead**: Scaling to thousands of networks requires managing thousands of GRE tunnels in a full-mesh or complex hub-and-spoke topology. This becomes an administrative overhead very quickly.
- **Resource consumption**: Each GRE tunnel is typically represented as a virtual network interface in the operating system. Creating thousands of virtual interfaces consumes kernel resources and can impact performance.
- **Inefficiency**: Sending traffic for multiple virtual networks over the same physical path requires multiple nearly-identical GRE tunnels, which is redundant and inefficient compared to a single VXLAN tunnel that can carry traffic for millions of networks using the VNI.

While GRE is a versatile tunnelling protocol, its lack of a built-in segmentation identifier like the VNI is its primary architectural limitation in multi-tenant cloud and container environments.

Open vSwitch as an overlay engine

While the Linux bridge is a capable virtual switch for basic connectivity, it

lacks the programmability and advanced features required for large-scale, dynamic environments. OVS was designed to fill this gap. It is a production-quality, multilayer virtual switch that acts as a sophisticated overlay engine, enabling complex network topologies to be built in software. The true power of Open vSwitch lies in its programmability and ability to terminate and manage overlay tunnels like VXLAN and GRE, transforming a host from a simple endpoint into an intelligent gateway for virtual network traffic.

To understand how OVS operates as an overlay engine, it is essential to understand its core components:

- **ovsdb-server**: The **ovsdb-server** is the configuration database of the switch. It stores virtual switch definitions, port configurations, and most importantly, tunnel endpoints (e.g., remote IP addresses for VXLAN). This is the long-term memory of the switch. The external tools like **ovs-vsctl** and **ovs-ofctl** communicate with **ovsdb-server** using the **ovsdb** protocol to program the switch. It also provides an interface for management applications to interact with the switch using **Representational State Transfer Application Programming Interface** (**REST APIs**).
- **ovs-vswitchd**: The **ovs-vswitchd** is the core switch daemon. It implements the data plane, communicating with the **ovsdb-server** for configuration and with the kernel module for fast packet forwarding. This is the component that makes forwarding decisions. The **ovsdb-server** installs the switch configuration in **ovs-vswitchd** as flow table entries. The **ovs-vswitchd** is also the component that implements the OpenFlow protocol, which SDN controllers use to program the switch.
- **Kernel datapath**: For performance, a kernel module handles the *fast path* forwarding of packets. When a packet enters the datapath, and there is no matching flow found to forward the packet, the datapath sends the packet to the **ovs-vswitchd** in the user space to determine its forwarding path. This is the slow path that the first packet of a flow goes through to establish the flow. Once a flow is established, it is cached in the datapath for any packets matching the flow.

The following figure shows the components of OVS and how it

communicates with components, including external tools and the **software-defined networking** (**SDN**) controller, to provide programmability:

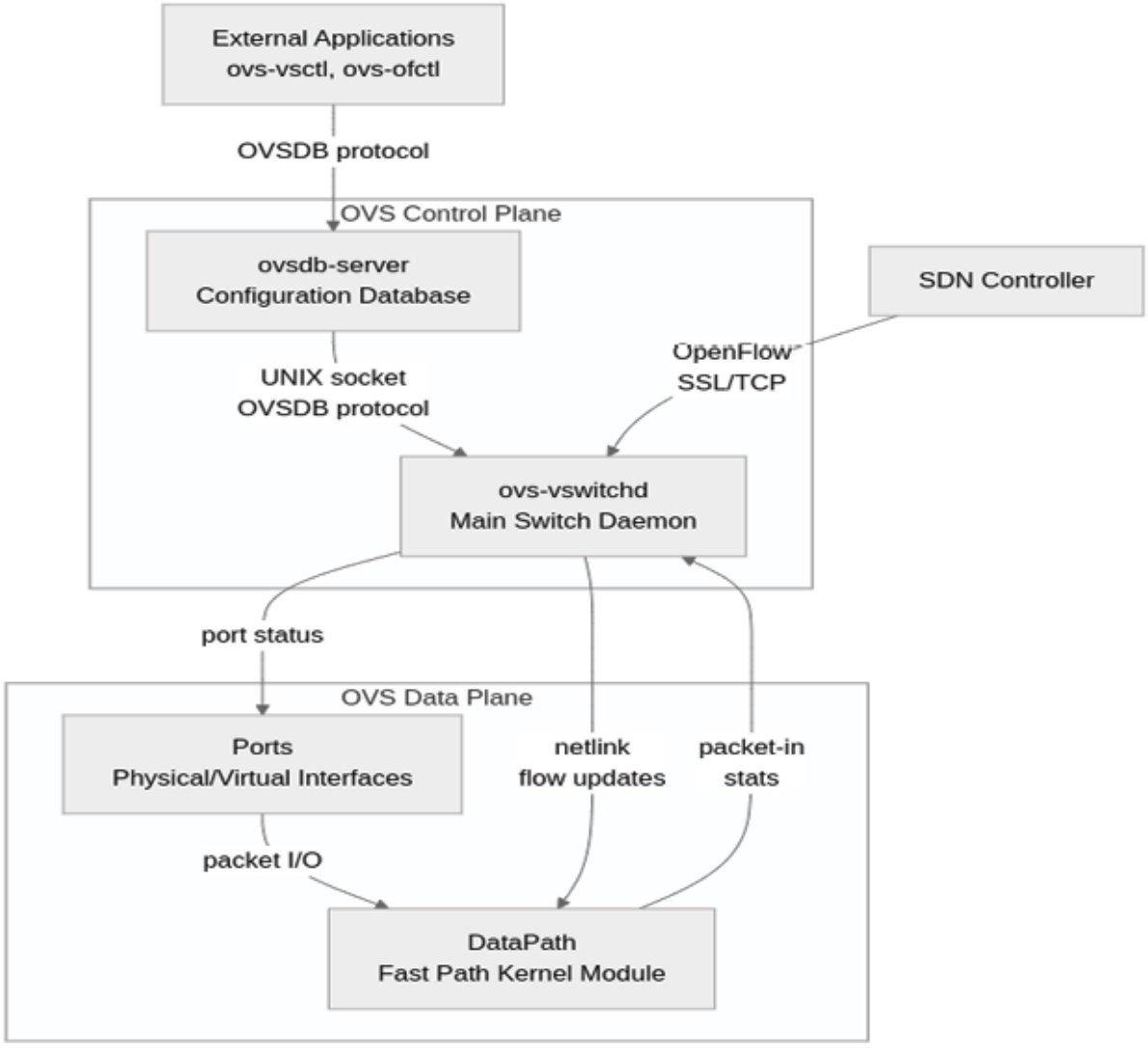


***Figure 11.3**: Components of Open vSwitch*

Creating and managing overlay tunnels

Let us consider a VXLAN overlay between two hosts, A and B. A bridge br0 is created on the switch. Physical interface **eth0** and virtual interfaces **veth1** are then added as ports to this bridge. A VXLAN interface, **vxlan0**, is added to the bridge, which acts as the VTEP. The following commands are used to configure the bridge and its ports:

```
ovs-vsctl add-br br0
```

```
ovs-vsctl add-port br0 eth0
ovs-vsctl add-port br0 vxlan0 -- set interface vxlan0 type=vxlan \
options:remote_ip=192.168.100.20 options:key=100

ovs-vsctl add-port br0 veth1
```

In this example, **vxlan0** is the VTEP. The **key=100** option sets the VNI, logically segregating this overlay network from others.

When a container on *Host A* sends a packet to a container on *Host B*, the following steps are taken by the **ovs-vswitchd** to determine its forwarding path and transmission to the destination container:

1. The packet from the container, with its MAC and IP, arrives on the OVS bridge via veth1.
2. **ovs-vswitchd** consults its flow tables. It determines that the destination MAC is reachable via the **vxlan0** port, which is associated with VNI 100 and remote IP 192.168.100.20.
3. The kernel datapath performs the encapsulation, where the entire original Ethernet frame is wrapped in a new VXLAN header (with VNI 100), which is then placed inside a new UDP/IP packet. The source IP is Host A's underlay IP, and the destination IP is Host B's underlay IP (192.168.100.20).
4. This new packet is sent out via the physical eth0 interface, across the underlay network.
5. On Host B, the packet is received by eth0 and passed to the OVS kernel module.
6. OVS recognizes it as a VXLAN packet, decapsulates it, and examines the VNI. It confirms that the VNI 100 corresponds to its local bridge br0.
7. The original Ethernet frame is injected into the **br0** and forwarded to the correct container's interface based on the inner destination MAC address.

Open vSwitch and software-defined networking

Traditional network switches are monolithic devices. The decision-making

logic of the switch, known as the control plane, is responsible for running routing protocols like **Open Shortest Path First** (**OSPF**), learning MAC addresses, and building routing tables, and the packet-forwarding hardware, known as the data plane, is responsible for pushing bits out of ports as efficiently as possible, are tightly integrated inside the same box.

SDN fundamentally breaks this model by separating the two planes:

- **Control plane**: It is extracted from the individual switches and centralized into a software-based SDN controller.
- **Data plane**: It remains in the network devices (physical or virtual), which become simple, fast packet-forwarding engines.

SDN can be best understood as a hierarchy of intelligent layers, each with a distinct responsibility, working together to manage the network. At the very top sits the Application Layer, which encompasses the various programs and services that require specific network behavior, such as cloud orchestration platforms like OpenStack or Kubernetes, security firewalls, and load balancers. These applications do not communicate with the network hardware directly. Instead, they express their high-level requirements, such as creating a new isolated network or applying a security policy through a Northbound API.

This Northbound API is an interface typically using modern standards like RESTful APIs that feeds these application requirements down to the SDN controller. The controller is the brain of the SDN architecture, a software stack that maintains a global, real-time view of the entire network. It translates the application's business logic into concrete network rules and policies. It is this controller that possesses the intelligence traditionally locked away inside individual network devices.

The controller then communicates its decisions downward using a Southbound API.

The following figure describes how applications like the cloud orchestrator communicate with the SDN controller using the Northbound API. The SDN controller then uses the Southbound API to program the network elements, like the OVS, to realise the virtual network configuration requested by the cloud orchestrator.

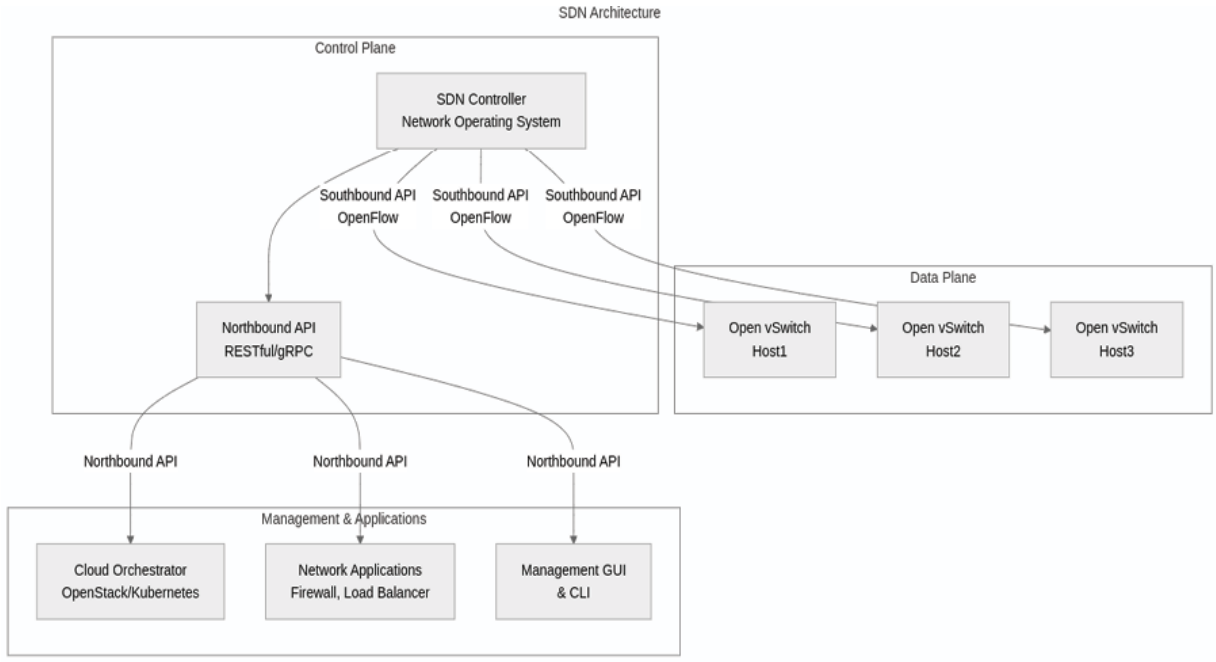


***Figure 11.4**: SDN controller and data plane*

The most prominent of which is the OpenFlow protocol. This southbound interface is the language the controller uses to program the data plane. The data plane consists of the physical and virtual networking devices themselves, such as OVS, physical switches, and routers that support OpenFlow. These devices shed their independent intelligence and become efficient packet-forwarding engines, containing programmable flow.

When a packet arrives at such a device, it is no longer processed by built-in protocols but is matched against flow tables, which dictate precise actions like forwarding, modifying, or dropping the packet. Tables that are populated with rules from the controller. This clear separation of the control plane's intelligence from the data plane's forwarding action is the fundamental revolution that defines SDN.

OpenFlow is a communications protocol that allows an SDN Controller to remotely configure and manage the forwarding behavior of a network device like Open vSwitch. It provides a standardized way for the controller to:

- Add, modify, and delete flow entries in the switch's flow tables.
- Receive notifications about network events (e.g., when a new host

connects).

- Collect statistics from the switch.

A flow table is the fundamental programmable forwarding unit within an SDN data plane, acting as the switch's new set of instructions. The flow table can be visualized as a series of *if-then* rules that have replaced the traditional self-learning logic and protocols, like the spanning tree. Each entry in this table, known as a flow, consists of three primary parts: match fields, priority, and action, described in the following list:

- **Match fields**: The match fields define a specific pattern to look for in a network packet, which can be as broad as all traffic from a subnet or as precise as a single **Transmission Control Protocol** (**TCP**) connection between two applications, based on criteria like source IP address, destination TCP port, or even the virtual network identifier in an overlay packet. This can be a combination of many headers, including:
    - Ingress Port
    - Ethernet Source/Destination MAC address
    - VLAN ID
    - IP Source/Destination address
    - IP Protocol (e.g., TCP, UDP)
    - TCP/UDP Source/Destination Port
    - The VNI in a VXLAN header
- **Priority**: Determines the order in which flow entries are matched. A higher-priority flow is checked first.
- **Actions**: A list of instructions to execute if the packet matches the criteria. Common actions include:
    - **output:PORT**: Forward the packet out a specific port (e.g., a physical NIC or a tunnel).
    - **mod_dl_src**: Modify the source MAC address.
    - **push_vlan / pop_vlan**: Add or remove a VLAN tag.
    - **load:value->NXM_OF_VLAN_TCI[]**: Set a specific VLAN ID.
    - **resubmit**: Send the packet to another flow table for further

processing.

When a packet arrives at the switch, its headers are parsed and compared against the match fields of every flow in the table, starting with the highest priority entry. Upon finding a match, the switch immediately executes the corresponding set of instructions, which are the actions dictating the packet's fate. These actions go beyond simple forwarding and can include modifying the packet's content, sending it to a different table for further processing, or simply dropping it for security purposes. The first packet of a new conversation might be sent to the SDN controller for a decision, but once a flow is installed, all subsequent packets in that conversation are handled at line speed directly in the data plane. In essence, the flow table transforms a generic network switch from an autonomous appliance into a programmable entity that forwards traffic exactly as directed by the central controller's policy. The following steps describe how a new flow is added to the flow table by the SDN controller:

1. When the first packet from a new flow arrives at OVS, there is no matching flow entry in the flow table. This triggers a Table-Miss.
2. OVS encapsulates this unknown packet and sends it to the SDN controller via an OpenFlow packet-in message.
3. The controller's logic analyzes the packet. It might consult its global network view, apply security policies, and determine the optimal path for this flow.
4. The controller calculates the necessary actions and sends an OpenFlow flow-mod message back to OVS to install a new, specific flow entry in its flow tables.
5. All subsequent packets belonging to the same flow now match this newly installed entry and are processed at line-rate in the kernel datapath without involving the controller. This provides high performance.

SDN controller could install a flow like this in OVS:

```
match: ip, nw_dst=10.0.1.5, actions: load:5000->tun_id, output:vxlan0.
```

This rule states: *For any IP packet destined for 10.0.1.5, set the VNI (tunnel ID) to 5000 and send it out the VXLAN tunnel port vxlan0.*

OVS provides the programmable data plane, OpenFlow is the protocol that programs it, flow tables are where the instructions are stored, and the SDN controller is the intelligence that writes those instructions. Together, they transform static network infrastructure into a dynamic, application-aware fabric.

In a multi-tenant cloud, OVS is the foundational technology. Each tenant is assigned a unique VNI, e.g., all traffic from *Tenant A's* VMs is tagged with VNI 5000, while *Tenant B's* traffic uses VNI 5001. OVS on every hypervisor ensures that packets are only delivered to interfaces belonging to the correct VNI, providing strong isolation even though all tenants share the same physical network underlay. This software-defined approach makes the cloud provider's network agile, scalable, and efficient.

Programming Open vSwitch

To understand the flow tables in OVS let us use the following script to setup an Open vSwitch and use the **ovs-ofctl** commands to program the flow rules. In the following script, we will first clear all the flow rules, then add rules for handling:

- ARP packet with normal action. The normal action tells the switch to process the packet using its built-in MAC learning and forwarding logic, like a traditional Ethernet switch.
- Drop ICMP packets from 10.0.0.0/24 network.
- Sent packet for TCP port 80 out through the bridge port 2:

Finally, we will dump the flow rules and statistics of the bridge ports:

```
#!/bin/bash
BRIDGE="of-br0"
echo "=== Setting up OpenFlow Bridge: $BRIDGE ==="

#Create bridge and ports
ovs-vsctl add-br $BRIDGE
for i in 1 2 3; do
```

```
 ovs-vsctl add-port $BRIDGE of-port$i \
   -- set interface of-port$i type=internal
 ip link set dev of-port$i up
done

#Clear existing flows
ovs-ofctl del-flows $BRIDGE
echo "=== Adding OpenFlow Rules ==="

#Add flow rules
ovs-ofctl add-flow $BRIDGE "priority=100,arp,actions=normal"
ovs-ofctl add-flow $BRIDGE \
"priority=200,ip,nw_src=10.0.0.0/24,icmp,actions=drop"
ovs-ofctl add-flow $BRIDGE
"priority=150,tcp,tp_dst=80,actions=output:2"

#Display the flows and port statistics
echo "=== Current Flows ==="
ovs-ofctl dump-flows $BRIDGE
echo "=== Port Statistics ==="
ovs-ofctl dump-ports $BRIDGE
```

On running the script, you should get an output like the following:

```
Cleared all flows
 === Adding OpenFlow Rules ===
Added flow: priority=100,arp,actions=normal
Added flow: priority=200,ip,nw_src=10.0.0.0/24,icmp,actions=drop
Added flow: priority=150,tcp,tp_dst=80,actions=output:2
 === Current Flows ===
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=0.043s, table=0, n_packets=0, n_bytes=0,
idle_age=0, priority=200,icmp,nw_src=10.0.0.0/24 actions=drop
```

cookie=0x0, duration=0.033s, table=0, n_packets=0, n_bytes=0, idle_age=0, priority=150,tcp,tp_dst=80 actions=output:2

cookie=0x0, duration=0.052s, table=0, n_packets=0, n_bytes=0, idle_age=0, priority=100,arp actions=NORMAL

=== Flow Statistics ===

OFPST_PORT reply (xid=0x2): 4 ports

port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=0, bytes=0, drop=0, errs=0, coll=0

port 1: rx pkts=2, bytes=152, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=1, bytes=90, drop=0, errs=0, coll=0

port 2: rx pkts=1, bytes=76, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=1, bytes=90, drop=0, errs=0, coll=0 port 3: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0, crc=0 tx pkts=1, bytes=90, drop=0, errs=0, coll=0

This lab provides a look at how the flow rules use the match condition, priority, and actions to handle packet forwarding.

Docker networking models

Before the dominance of Kubernetes, Docker established the foundational models for container networking. It uses its own architecture, the **Container Network Model** (**CNM**), which shares conceptual similarities with the CNI specification in kubernetes which we will discuss in the next section. The default and most common driver in Docker networking is the bridge network. When the Docker daemon starts, it creates a virtual bridge, typically called **docker0**. Every container launched without specific network options is connected to this bridge via a veth pair. Docker automatically manages IP address assignment from a private subnet and configures iptables rules to provide **Network Address Translation** (**NAT**), allowing containers to reach external networks while appearing to the outside world as the host's IP address. This provides a good balance of isolation and convenience for single-host scenarios.

For situations where maximum performance is critical, and network isolation can be sacrificed, Docker offers the host network driver. This

driver bypasses Docker's network namespace isolation completely. A container using the host network shares the host's networking stack, directly using its IP address and ports. While this eliminates the overhead of NAT and virtual interfaces, it introduces the significant risk of port conflicts and removes a critical layer of security.

To connect containers across multiple Docker hosts, the overlay network driver is used. This driver creates a distributed virtual network that spans across the nodes in a Docker Swarm cluster. Like the CNI plugins we will discuss, it uses VXLAN to encapsulate traffic. Docker Swarm includes a built-in key-value store that synchronizes the network state, such as which services are on which nodes, allowing the overlay network to automatically route packets to the correct destination host.

Container network interface

In the early days of Kubernetes, a significant challenge was the lack of a standard way to connect pods to the network. Each orchestrator and platform had its own method, creating fragmentation and complexity. CNI was created to solve this problem. It is a cloud-native computing foundation project that defines a universal contract between a container runtime, like Kubernetes, and network plugins.

CNI provides a common interface by specifying a very simple, plugin-based model. Its core principle is that networking is a distinct concern that should be decoupled from the core container runtime logic. The interface itself is defined by a set of simple operations, primarily ADD and DEL, which correspond to adding a container to a network or deleting it from one. When Kubernetes creates a pod, the kubelet invokes the designated CNI plugin, passing it a **JavaScript Object Notation** (**JSON**) configuration that describes the network.

CNI provides a standardized protocol for connecting containers to external networks. It defines a simple contract between container runtimes and network plugins that allows for consistent network configuration across different environments. The specification focuses on two primary

operations:

- Adding a container to a network.
- Removing it from a network.

CNI also specifies optional check and version operations for validation and compatibility reporting. The following figure describes the interaction of Kubernetes and the CNI plugin:

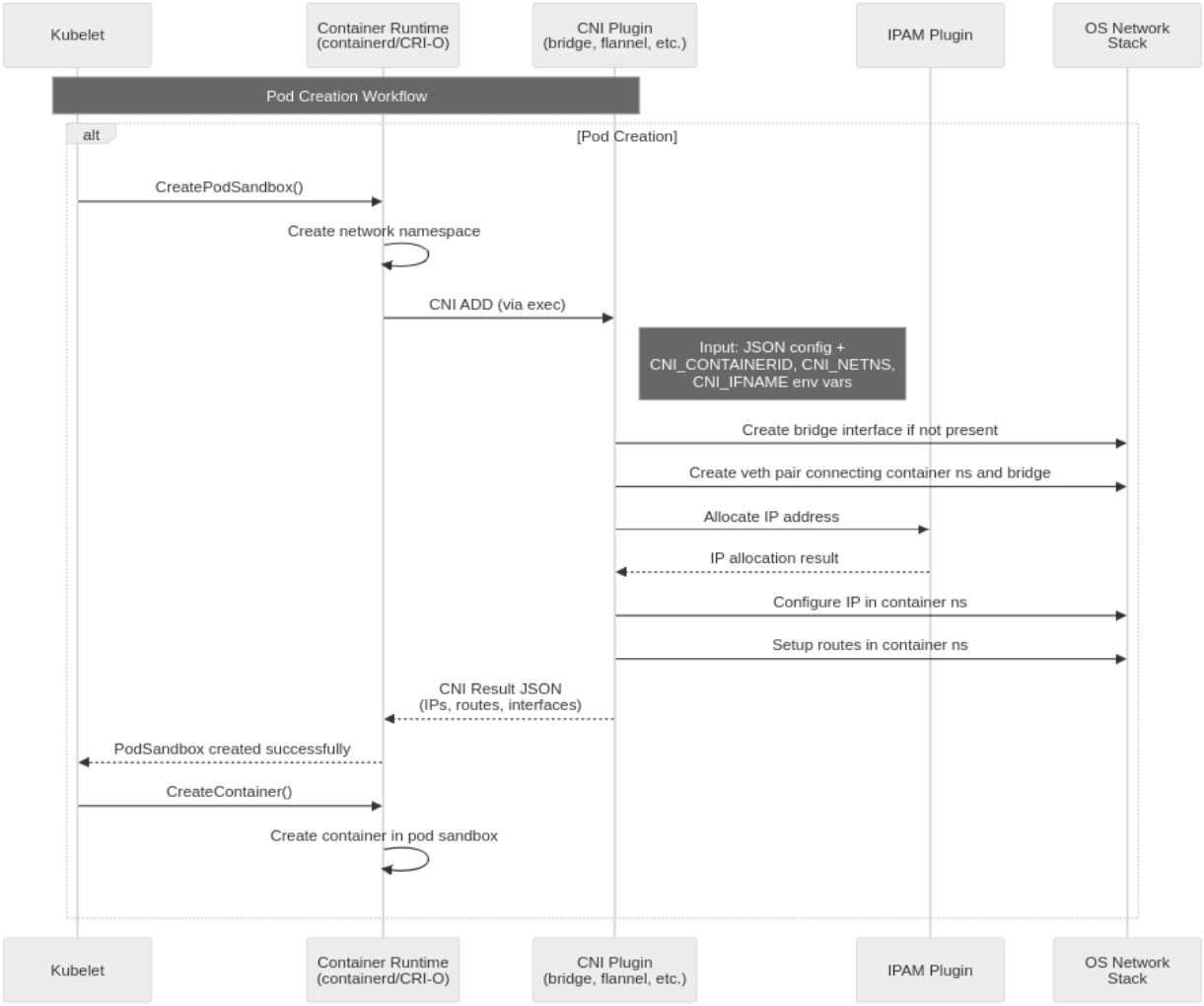


***Figure 11.5**: Pod creation with CNI*

When a container runtime creates a network namespace, it invokes a CNI plugin with a JSON configuration passed via stdin and critical parameters like container ID, network namespace path, and interface name provided as environment variables. The plugin executes within this context and must

return a structured result indicating success or failure. Successful network addition returns interface details, IP addresses, routes, and **Domain Name System** (**DNS**) information in a standardized JSON format. CNI supports three plugin types:

- Main plugins that handle interface configuration, like bridge or macvlan.
- **IP Address Management** (**IPAM**) plugins that manage IP address assignment, such as host-local or dhcp.
- Meta plugins that chain multiple plugins together.

Plugins can be combined in execution chains where each plugin executes sequentially, with the output of one becoming input to the next.

The specification delegates IP address management to specialized IPAM plugins, separating connectivity concerns from address allocation. Network configurations declare both the main plugin and the IPAM plugin separately. Error handling follows consistent patterns with required error codes and messages, plus optional details for debugging.

The plugin's sole responsibility is to execute the necessary commands to connect the container's network namespace to the wider network and then return a result indicating success or failure. This abstraction is what allows Kubernetes to remain network-agnostic. The orchestrator does not need to know how to create a bridge, configure a VXLAN tunnel, or propagate **Border Gateway Protocol** (**BGP**) routes. It only needs to know how to call the CNI plugin and trust that the plugin will perform the required magic. This common interface enables the rich ecosystem of CNI plugins to co-exist, each implementing the same simple contract but using wildly different underlying technologies to fulfil it. In the next section, we will explore the common CNI plugins used with Kubernetes.

Flannel CNI plugin

Flannel is one of the oldest and simplest CNI plugins, designed for ease of use and reliability. Its architecture is straightforward, focused on a per-node agent called **flanneld**. This agent is responsible for managing a flat, overlay network for the entire cluster. Flannel’s operation begins with a central

source of truth, which was traditionally etcd but now primarily uses the Kubernetes API. This central store allocates to each node in the cluster a unique, non-overlapping subnet from a larger, configured pod CIDR block.

The **flanneld** agent on each node watches the API for these allocations. Once it knows its own subnet and the subnets of all other nodes, it programs the local node's networking stack accordingly.

The magic of Flannel lies in its pluggable backends, which determine the data plane technology used. The default and most common backend is VXLAN. Flannel creates a virtual tunnel interface, typically named **flannel.1**, which acts as a VTEP. When a pod on one node needs to communicate with a pod on another node, Flannel encapsulates the traffic in a VXLAN packet, using the destination node's IP as the tunnel endpoint. For environments where a pure Layer 2 network connects all nodes, Flannel offers the more performant **host gateway** (**host-gw**) backend. This backend forgoes encapsulation entirely and instead programs the host's routing table with a direct route, stating that to reach a remote pod subnet, the next hop is the IP address of the node hosting that subnet. This is highly efficient but requires a flat network topology.

The primary advantage of Flannel is its simplicity and robustness; it works out of the box reliably with minimal configuration. The trade-off for this simplicity is a lack of advanced features. Flannel provides basic connectivity but does not implement any form of network policies for security, leaving pods open to unrestricted communication by default. It is an excellent choice for getting a cluster up and running quickly or for environments where advanced networking features are not a requirement.

Flannel with Docker

In this section, we will set up a lab to test Flannel with Docker containers. To set up the lab, we will create an **etcd** node and two **flannel** nodes. The artifacts for this lab are stored in the following directory structure:

```
├── etcd-v3.6.6-linux-amd64
│   ├── Dockerfile
│   ├── Documentation
```

```
│   │   ├── OWNERS
│   │   ├── README.md
│   │   ├── contributor-guide
│   │   ├── dev-guide
│   │   ├── etcd-internals
│   │   └── postmortems
│   ├── README-etcdctl.md
│   ├── README-etcdutl.md
│   ├── README.md
│   ├── README-v2-etcdctl.md
│   ├── default.etcd
│   │   └── member
│   ├── etcd
│   ├── etcdctl
│   ├── etcdutl
│   └── start-etcd.sh
├── flannelnode
│   ├── Dockerfile
│   ├── flanneld-amd64
│   └── start-flannel.sh
└── run.sh
```

The lab setup with containers is described in the following figure:

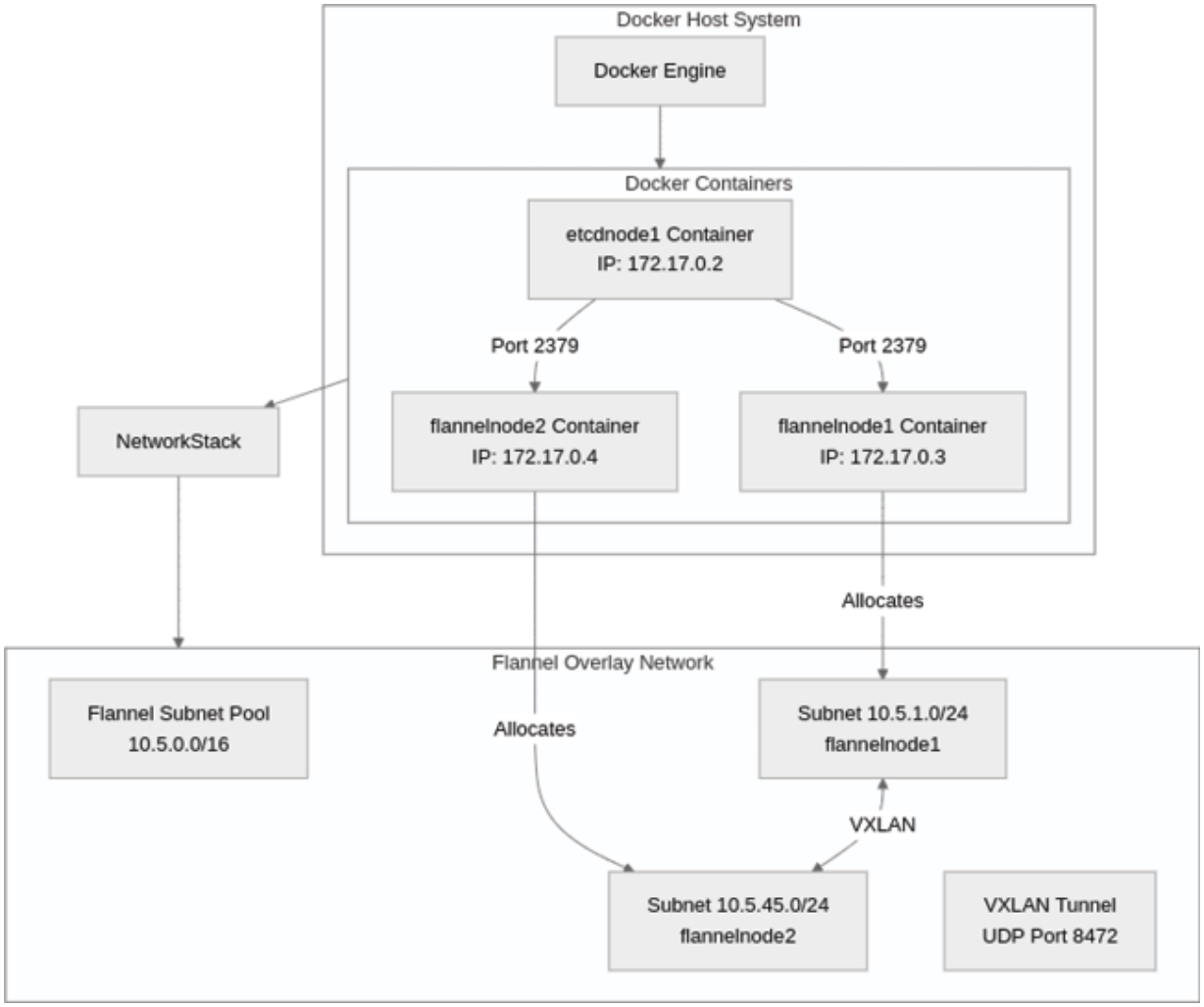


***Figure 11.6**: Flannel setup with Docker containers*

The following steps describe the process of setting up and testing the flannel network:

1. Build the flannel node images from the Dockerfile:
   a. The flannel binary can be downloaded from the flannel project's GitHub page:
   https://github.com/flannel-io/flannel/releases.
   b. The start **flannel.sh** file has the following contents:

```
#!/bin/bash
flanneld -etcd-endpoints 172.17.0.2:2379 > /tmp/log 2>&1
```

   c. We will use the following Docker file for the flannel node:

```
FROM ubuntu:rolling
RUN apt update && apt install -y iproute2 iputils-ping bind9-dnsutils
RUN apt install -y netcat-openbsd
ADD ./flanneld-amd64 /usr/bin/flanneld
ADD ./start-flannel.sh /root/start-flannel.sh
CMD ["sleep", "infinity"]
```

d. Create the **etcd** node image:

i. **etcd** binary can be downloaded from the **etcd** project's GitHub page:
https://github.com/etcd-io/etcd/releases

ii. Start **etcd.sh** file has the following content:

```
#!/bin/bash

etcd --listen-client-urls http://0.0.0.0:2379 --advertise-client-urls http://0.0.0.0:2379 > /tmp/log 2>&1 &
sleep 5
export ETCDCTL_API=3
etcdctl put /coreos.com/network/config '{ "Network": "10.5.0.0/16", "Backend": {"Type": "vxlan"}}'
```

iii. We will use the following Docker file for **etcd** node:

```
FROM ubuntu:rolling
RUN apt update && apt install -y iproute2 iputils-ping bind9-dnsutils
ADD  ./etcd /usr/bin/etcd
ADD  ./etcdctl /usr/bin/etcdctl
ADD  ./etcdutl /usr/bin/etcdutl
ADD  ./start-etcd.sh /root/start-etcd.sh

CMD ["sleep", "infinity"]
```

iv. Create the Docker images with the following commands in their

respective directories:

```
docker build –t etcdnode:1.0 .
docker build –t flannelnode:1.0
```

2. Create a script to start the Docker nodes as follows:

```
#!/bin/bash

docker run --rm=true --name=etcdnode1 -d etcdnode:1.0
docker run --rm=true --privileged --sysctl net.bridge.bridge-nf-call-iptables=1 --name=flannelnode1 -d flannelnode:1.0
docker run --rm=true --privileged --sysctl net.bridge.bridge-nf-call-iptables=1 --name=flannelnode2 -d flannelnode:1.0
```

Use the preceding script to start the **etcd** node and two **flannel** nodes.

3. Start the **etcd** and **flannel**:

a. Once the containers have started, log in to each of the containers and start the service:

```
docker exec -it etcdnode1 bash
/root/start-etcd.sh
```

b. For **flannelnode** containers **flannelnode1** and **flannelnode2**:

```
docker exec -it flannelnode1 bash
/root/start-flannel.sh
```

4. Check the **flannel** interfaces and network configuration:

a. The network interfaces should list the **flannel** interface like the following:

```
# ip -o link sh
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000\    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0@if91: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default \    link/ether 02:cc:e9:13:8e:7d brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

```
3: flannel.1: <BROADCAST,MULTICAST,UP,LOWER_UP>
mtu 1450 qdisc noqueue state UNKNOWN mode DEFAULT
group default \    link/ether ea:d0:c6:df:53:1d brd ff:ff:ff:ff:ff:ff
```

b. The IP address configuration should be like the following:

```
# ip -o addr sh
1: lo inet 127.0.0.1/8 scope host lo\ valid_lft forever preferred_lft
forever
1: lo inet6 ::1/128 scope host proto kernel_lo \ valid_lft forever
preferred_lft forever
2: eth0 inet 172.17.0.3/16 brd 172.17.255.255 scope global eth0\
valid_lft forever preferred_lft forever
3: flannel.1 inet 10.5.96.0/32 scope global flannel.1\ valid_lft
forever preferred_lft forever
3: flannel.1 inet6 fe80::e8d0:c6ff:fedf:531d/64 scope link proto
kernel_ll \ valid_lft forever preferred_lft forever
```

c. The routes are set up:

```
# ip -o route sh
default via 172.17.0.1 dev eth0
10.5.45.0/24 via 10.5.45.0 dev flannel.1 onlink
172.17.0.0/16 dev eth0 proto kernel scope link src 172.17.0.3
```

d. To test the connectivity, we will first change the default route to use Flannel.1 interface:

```
ip route delete default
ip route add default via 10.5.45.0
```

This step should be performed on both the flannel nodes.

5. Check connectivity with **ping**:

a. On the second **flannel** container, check the **ip** address:

```
# ip -o addr sh
1: lo inet 127.0.0.1/8 scope host lo\ valid_lft forever preferred_lft
forever
1: lo inet6 ::1/128 scope host proto kernel_lo \ valid_lft forever
```

```
preferred_lft forever
2: eth0 inet 172.17.0.4/16 brd 172.17.255.255 scope global eth0\
valid_lft forever preferred_lft forever
3: flannel.1 inet 10.5.45.0/32 scope global flannel.1\ valid_lft
forever preferred_lft forever
3: flannel.1 inet6 fe80::78e4:eff:fecc:9efd/64 scope link proto
kernel_ll \ valid_lft forever preferred_lft forever
```

b. In this case, the IP address of the second node is 10.5.96.0.

c. Check the connectivity from the second node to the first node with ping is as shown here:

```
# ping -c3 10.5.96.0
PING 10.5.96.0 (10.5.96.0) 56(84) bytes of data.
64 bytes from 10.5.96.0: icmp_seq=1 ttl=64 time=0.910 ms
64 bytes from 10.5.96.0: icmp_seq=2 ttl=64 time=0.271 ms
64 bytes from 10.5.96.0: icmp_seq=3 ttl=64 time=0.202 ms

--- 10.5.96.0 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2027ms
rtt min/avg/max/mdev = 0.202/0.461/0.910/0.318 ms
```

6. You can check the logs in the **flannel** container to check the **flannel** backend:

```
Subnet:ip.IP6Net{IP:(*ip.IP6)(nil), PrefixLen:0x0},
Attrs:lease.LeaseAttrs{PublicIP:0xac110003, PublicIPv6:(*ip.IP6)
(nil), BackendType:"vxlan",
BackendData:json.RawMessage{0x7b, 0x22, 0x56, 0x4e, 0x49,
0x22, 0x3a, 0x31, 0x2c, 0x22, 0x56, 0x74, 0x65, 0x70, 0x4d, 0x41,
0x43, 0x22, 0x3a, 0x22, 0x65, 0x61, 0x3a, 0x64, 0x30, 0x3a, 0x63,
0x36, 0x3a, 0x64, 0x66, 0x3a, 0x35, 0x33, 0x3a, 0x31, 0x64, 0x22,
0x7d}, BackendV6Data:json.RawMessage(nil)},
Expiration:time.Date(2025, time.November, 24, 9, 47, 5,
339561730, time.Local), Asof:4}} } I1123 09:47:09.339659 15
vxlan_network.go:265] Received Subnet Event with VxLan:
```

BackendType: vxlan, PublicIP: 172.17.0.3, PublicIPv6: (nil), BackendData: {"VNI":1,"VtepMAC":"ea:d0:c6:df:53:1d"}, BackendV6Data: (nil)

This should show the VNI for the VXLAN.

You can test other flannel backends like **host-gw** or WireGuard by changing the configuration in the **start-etcd.sh**.

Calico BGP-based CNI

The Calico CNI plugin takes a fundamentally different approach, favouring high performance and robust network policy over simplicity. It operates on a pure Layer 3 principle, treating the entire data center as one large IP network and each Kubernetes node as an internet router. Calico's architecture is more complex, comprising several components. The Calico node DaemonSet runs on each node and contains two key daemons:

- **Felix**: It is responsible for programming routes, ARP entries, and iptables rules for policy enforcement on the local node.
- **BIRD**: BGP client that is responsible for distributing routing information.

The core of Calico's networking magic is the BGP, the same protocol that glues the entire internet together. Each calico node uses BIRD to advertise the pod IP subnets it hosts to the rest of the cluster. In a full-mesh configuration, every node tells every other node, *The pod subnet 192.168.1.0/24 is on me; send packets for it my way*. This means there is no encapsulation overhead by default. When a pod sends a packet to a pod on another node, the host's kernel routing table, populated by these BGP-learned routes, directly forwards the packet to the correct destination node. The receiving node then delivers it to the target pod. This pure IP routing provides near-native network performance.

Beyond its high-performance data plane, Calico's other major strength is its powerful network policy engine. While Kubernetes has a basic NetworkPolicy API, Calico extends it with its own CRDs, providing capabilities like global network policies that apply across namespaces, egress gateway controls, and more sophisticated rule matching. Calico can

also operate in an overlay mode using IP-in-IP encapsulation if the underlying network is not routable, but its default and preferred mode is non-overlay BGP. It is the preferred choice for performance-sensitive workloads, on-premise deployments where administrators control the network, and environments where granular security policy is a top priority.

Cilium eBPF-based CNI

Cilium represents the next evolutionary step in Kubernetes networking, leveraging the Linux kernel's **Extended Berkeley Packet Filter** (**eBPF**) technology to inject its logic directly into the kernel. eBPF allows for running sandboxed programs inside the kernel without requiring kernel changes or loading modules. For networking, this means that packet filtering, forwarding, and policy enforcement can be implemented with eBPF programs that are more efficient and flexible than traditional methods like iptables.

Cilium's eBPF-powered data plane replaces kube-proxy for service load balancing, leading to significant performance gains, and enables deep observability into network and application traffic. Its most significant advantage is the ability to enforce security policies at the API level. While Calico and Kubernetes NetworkPolicy operate at Layers 3 and 4, Cilium can understand and enforce policies based on Layer 7 protocols like **Hypertext Transfer Protocol** (**HTTP**), gRPC, and Kafka. This means it can create rules that allow GET requests to a **/api** endpoint but deny POST requests, providing security granularity for microservices communication.

When choosing a CNI plugin, one must weigh the trade-offs. Flannel offers simplicity and reliability through its overlay model but lacks advanced features. Calico provides high performance and strong security policies through its BGP-based routing, but requires a more complex setup. Cilium, leveraging eBPF, delivers superior performance, deep observability, and API-aware security, though it introduces a newer and more complex technology stack. The choice ultimately depends on the specific needs of the cluster, balancing factors like performance requirements, security posture, and operational complexity.

Cloud networking

A cloud network is a network infrastructure and services that are hosted, operated, and delivered through cloud computing platforms over the internet. It represents the evolution of traditional networking into a fully virtualized, software-defined, network-as-a-service model. Cloud networking refers to the entire ecosystem of networking capabilities that cloud providers deliver as managed services. The principles of virtual networking we have explored, like isolation, routing, and software-defined overlays, etc., are the foundation for the VPC. A VPC is a logically isolated section of a cloud provider's network, a secure and customizable private network where you can launch your computing resources. It is the cloud-scale equivalent of a traditional corporate data center network, but with the agility of software. Within a VPC, you have complete control over your virtual networking environment, including the selection of your own IP address range, the creation of subnets, and the configuration of route tables and network gateways. This model allows you to extend your own on-premises network into the cloud using industry-standard technologies, or to build a completely new, global network infrastructure. In the following sections, we explore the network offerings of popular cloud service providers.

AWS VPC Cloud Networking

Amazon Web Services (**AWS**) established the VPC model as the standard for cloud networking. An AWS VPC allows you to define a private IP address space for your resources using **Classless Inter-Domain Routing** (**CIDR**) blocks, such as 10.0.0.0/16. Within this space, you segment the network into subnets, which are ranges of IP addresses in a single availability zone. A critical distinction is made between public subnets, which have a route to an Internet Gateway for direct public access, and private subnets, which do not.

For security, AWS provides a two-tiered firewall. Security groups act as a stateful, virtual firewall for individual resources like EC2 instances, controlling traffic based on protocol, port, and source IP. They are typically

used to enforce rules that allow specific application traffic. Complementing these are **Network Access Control Lists** (**NACLs**), which are stateless firewalls that operate at the subnet level, providing a coarse-grained layer of security for all traffic entering or leaving a subnet.

A key innovation in AWS networking is the **Elastic Network Interface** (**ENI**), a virtual network card that can be attached to an EC2 instance. Each ENI has a primary private IPv4 address, one or more secondary private IPv4 addresses, and can have an elastic IP address associated with it. This abstraction is fundamental, as it is the ENI that security groups are attached to, and it is this model that allows the AWS VPC CNI plugin for Kubernetes to assign a true VPC IP address directly to every pod, providing native performance and simplified networking.

To enable resources in private subnets to access the internet for updates or external APIs without being exposed to inbound connections from the internet, AWS offers the NAT Gateway. This managed service allows outbound connections to be initiated from within a private subnet while blocking any unsolicited inbound traffic, a pattern essential for both security and operational maintenance.

The following figure illustrates a typical AWS VPC architecture with public and private subnets. The **Internet Gateway** (**IGW**) serves as the main entrance for internet traffic, connecting to the VPC's route tables. In the public subnet, the NAT Gateway enables outbound internet connectivity for resources in private subnets, while the EC2 web server can directly communicate with the internet. The private subnet contains protected resources like application servers and databases that cannot be directly accessed from the internet, but can initiate outbound connections through the NAT Gateway.

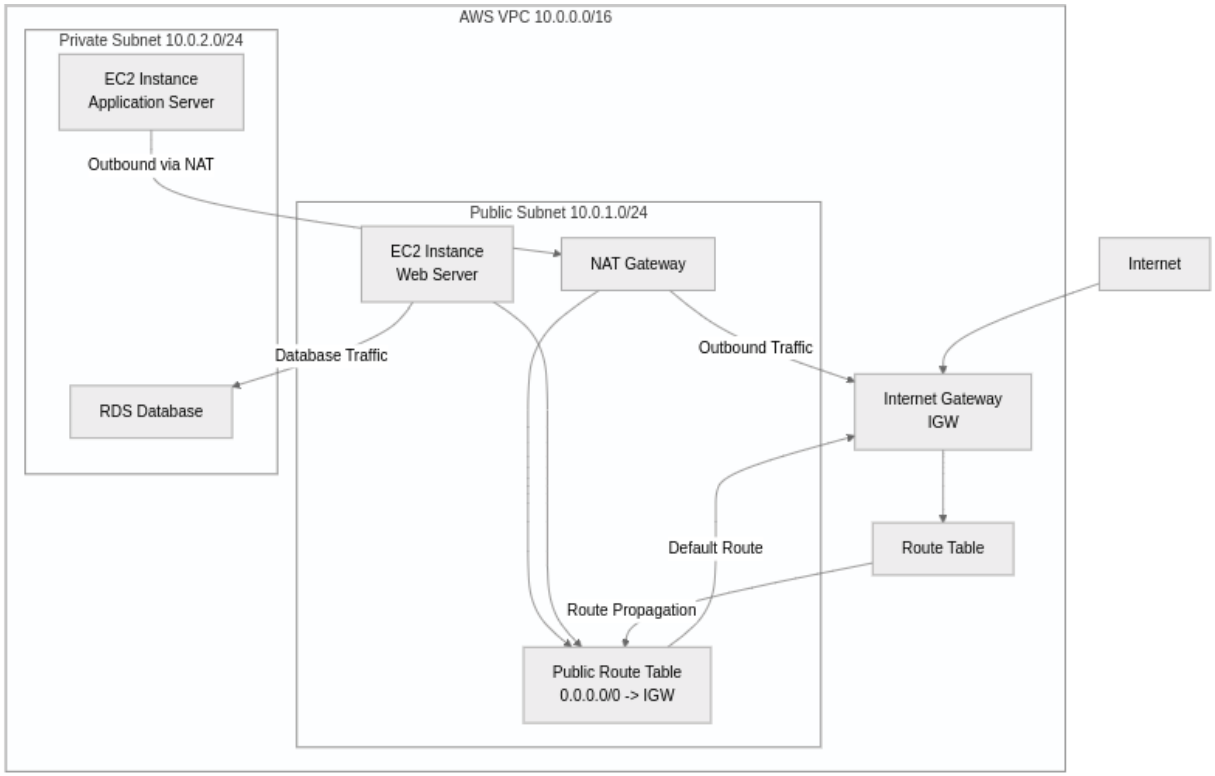


***Figure 11.7**: Components of AWS cloud networking*

Security in this architecture is enforced through multiple layers. Security groups act as virtual firewalls at the instance level, controlling inbound and outbound traffic, while network ACLs provide stateless filtering at the subnet level. Route tables direct traffic between subnets and to external networks, ensuring proper network segmentation and security isolation between public-facing and internal resources.

You can use the AWS CLI to inspect the components of the cloud network. The following commands can be used to explore the virtual network components:

- Lists all VPCs in the region, showing their IP ranges and tenancy using the following command:

  aws ec2 describe-vpcs

- Use the following command to get details for a specific VPC, filter by its ID:

  aws ec2 describe-vpcs --vpc-ids vpc-12345678

- To list all subnets, including their VPC association, availability zone, and CIDR block, use the following command:

  aws ec2 describe-subnets

- To check all route tables, use the following command:

  aws ec2 describe-route-tables

You can explore other options of the AWS CLI to inspect NAT Gateways, Internet Gateways, and security groups, etc. The documentation for the AWS CLI can be found at the given link: **https://docs.aws.amazon.com/cli/latest/reference/**

Microsoft Azure Virtual Network

Microsoft's Azure cloud platform implements its networking fabric through the Azure **Virtual Network** (**VNet**). The core concepts are analogous to an AWS VPC, providing an isolated environment for Azure resources like VMs. You define your own private IP address space and segment it into subnets within Azure regions. **Network Security Groups** (**NSGs**) are the primary tool for traffic filtering, functioning as stateful firewalls that can be applied to either a subnet or the network interface of a virtual machine, allowing for granular control.

A key difference for Azure is its deep and seamless integration with the broader Azure ecosystem. Services like Azure Load Balancer and Application Gateway provide robust traffic distribution, while features like user-defined routes offer fine-grained control over packet flow within a VNet. For secure access to **platform-as-a-service** (**PaaS**) offerings like Azure SQL Database, Azure provides private endpoints. This technology injects a private IP address from your VNet into the service, allowing you to connect to it securely over the Microsoft backbone network without any traffic ever traversing the public internet, thereby significantly reducing the attack surface.

This following figure shows isolation and IP address space for Azure resources. VNets are regional but can be connected globally through peering. Subnets are segments within the VNet that enable network segmentation. Stateful firewall rules that control traffic flow to and from

subnets and individual network interfaces. NSGs can be applied at both the subnet and the NIC level for granular security. Azure Bastions provide secure **Remote Desktop Protocol** (**RDP**) or **Secure Shell** (**SSH**) access to VMs without public IPs. VPN Gateway connects the VNet to on-premises networks securely. Private endpoints enable private connectivity to Azure services like Storage and **structured query language** (**SQL**).

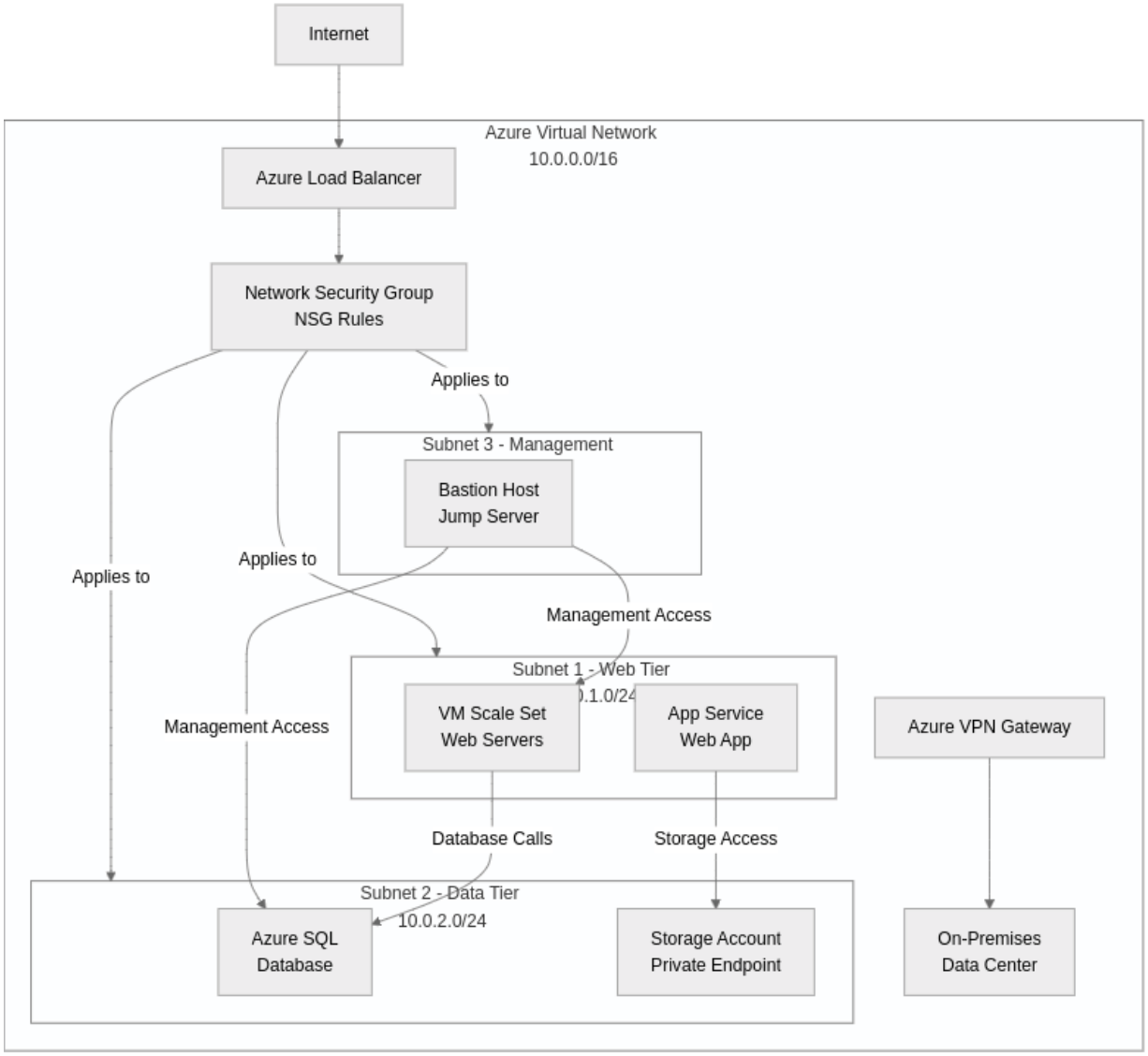


***Figure 11.8**: Components of Azure cloud networking*

This architecture demonstrates Azure's approach to software-defined networking in the cloud, emphasizing security through segmentation and

providing multiple layers of access control and monitoring.

The **az** CLI can be used to explore the cloud network in Azure. The following command can be used to explore the virtual network in Azure:

- To list VNET and subnets in GCP:

```
az network vnet list
az network vnet subnet list
```

- Use the following commands to list the firewall rules:

```
az network nsg list
az network nsg rule list --nsg-name myNSG
```

The **az** CLI provides sub-commands to explore other aspects of the cloud network. The documentation for **az** command can be found at the following link:

https://learn.microsoft.com/en-us/cli/azure/network?view=azure-cli-latest

Google Cloud VPC

Google Cloud Platform (**GCP**) introduces a uniquely simplified and global approach to cloud networking with its GCP VPC. Unlike the regional VPCs of other providers, a GCP VPC is a global resource. This means a single VPC network can span all of Google Cloud's regions around the world without any requirement for configuring inter-region VPC peering. Subnets within this global VPC are still regional, but the routes are global by default. A VM in a subnet in Iowa can communicate with a virtual machine in a subnet in Belgium using their internal IP addresses, and the routing is handled automatically by Google's global fabric.

Firewall rules in GCP VPC are also global and stateful, defined by specifying the source and destination of traffic using IP ranges, tags, or service accounts. This model simplifies policy management across a distributed application. A standout feature of Google's networking offering is private service connect. This allows you to publish and consume services securely across VPC networks, whether your own or those of a service provider. You can access Google-managed services like BigQuery or third-

party SaaS applications through a private, internal IP address from within your VPC, ensuring that service traffic remains on Google's secure network and is never exposed to the public internet. This global, service-centric design reflects Google's extensive experience running its own worldwide services. The following figure shows an example of a cloud network on GCP:

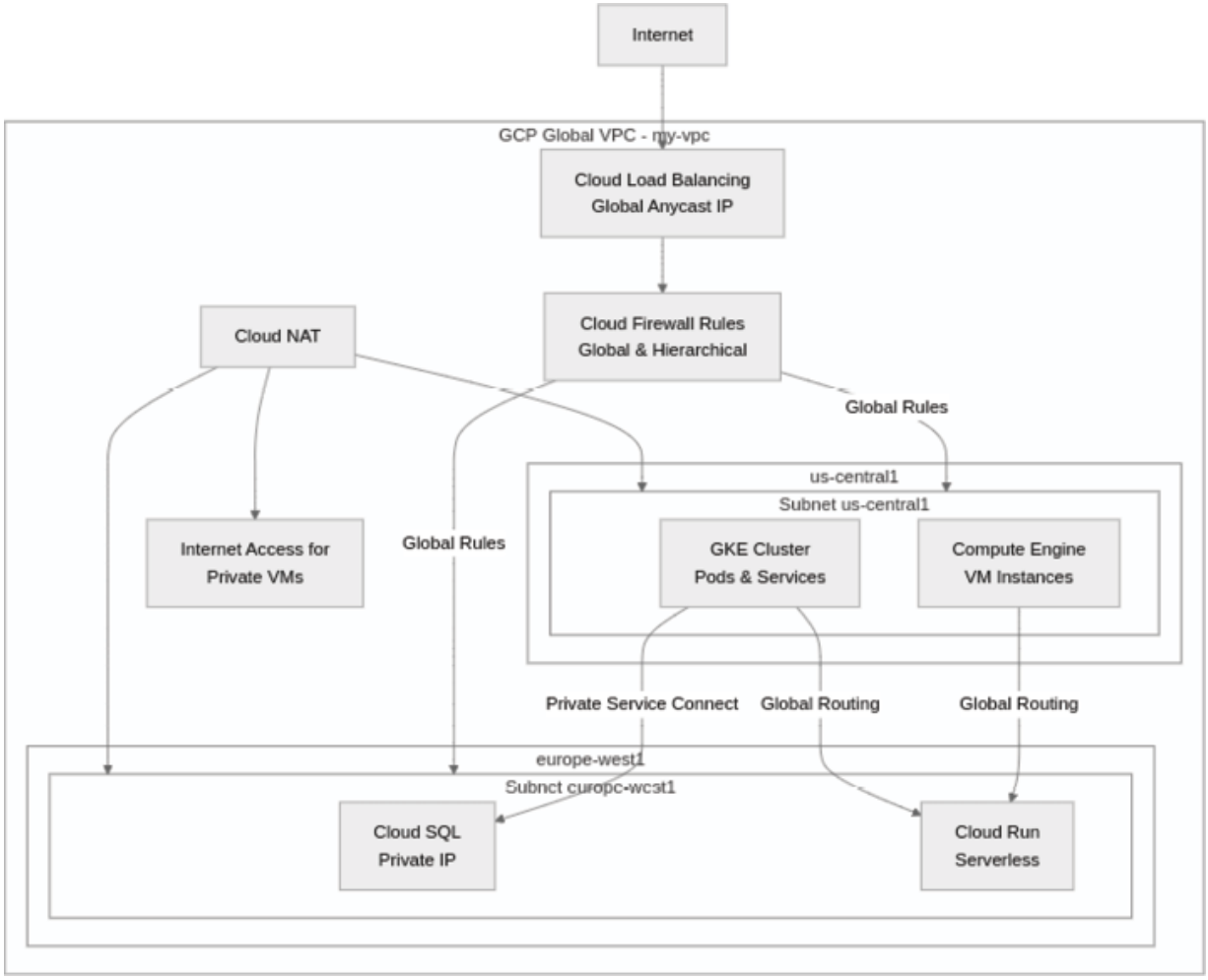


***Figure 11.9**: Components of GCP cloud networking*

Unlike regional VPCs in other clouds, GCP VPC is global by default. A single VPC can span all regions worldwide, with subnets defined regionally but routing happening globally. This means a VM in us-central1 can communicate with a Cloud Run service in europe-west1 using internal IPs without any peering configuration.

Each subnet is tied to a specific region (us-central1, europe-west1) but can route globally. Instances in different regions and subnets can communicate privately without additional setup. Subnets use CIDR blocks that can be expanded as needed. GCP's firewall rules are global and stateful, applying to all instances in the VPC regardless of region. Rules can be defined based on source IP ranges, instance tags, service accounts, and target tags for granular control.

The **gcloud** CLI can be used to explore and inspect the components of the GCP cloud network. The following commands can be used to explore the virtual network in GCP:

- To list the PVC and subnets:

```
gcloud compute networks list
gcloud compute networks subnets list
gcloud compute networks subnets list --regions us-central1
```

- The following command can be used to inspect routers and routing rules:

```
gcloud compute routers list
gcloud compute routes list
```

- To check the firewall configurations, use the following commands:

```
gcloud compute firewall-rules list
gcloud compute firewall-rules describe allow-ssh
```

The preceding commands just provide an overview of the **gcloud** CLI. You can find the **gcloud** documentation for network at: **https://cloud.google.com/sdk/gcloud/reference/compute/networks**

Conclusion

In this chapter, we covered the fundamental problem of connection in a fragmented world, discovering how overlay protocols like VXLAN and GRE provide the essential tunnels that allow virtual networks to be built freely upon a physical underlay. This capability is powered by programmable data planes like OVS, which, through the language of

OpenFlow and flow tables, transform static hardware into a dynamic, software-defined fabric.

We then witnessed how these abstractions are consumed by the world of containers. The CNI provides the crucial pluggable interface that allows Kubernetes to remain elegantly simple while supporting a diverse ecosystem of networking solutions. From the straightforward overlay of Flannel to the high-performance BGP routing of Calico and the eBPF-powered future embodied by Cilium, the core mission remains the same: to provide seamless, secure, and scalable connectivity for distributed applications.

Finally, we scaled these very same principles to their logical conclusion in the cloud. The VPC is the ultimate expression of the software-defined network—a fully managed, globally scalable system that applies the concepts of isolation, subnetting, routing, and firewalling at an immense scale. The journey from a veth pair to an AWS VPC, an Azure VNet, or a global GCP VPC is not a series of disconnected technologies, but a continuous story of abstraction and programmability. In the next chapter, we will explore advanced topics of eBPF and network monitoring.

Join our Discord space

Join our Discord workspace for latest updates, offers, tech happenings around the world, new releases, and sessions with the authors:

https://discord.bpbonline.com



CHAPTER 12

Network Monitoring with eBPF

Introduction

In the dynamic landscape of modern networking, visibility has become both a paramount necessity and a formidable challenge. Traditional network monitoring tools reliant on packet sampling, flow-based approximations, or static kernel instrumentation often fall short in providing the granular, real-time, and low-overhead observability required by today's distributed systems, cloud-native environments, and high-performance applications. They either lack the depth to diagnose subtle performance issues or introduce prohibitive overhead that can itself disrupt the systems they aim to observe.

The **Extended Berkeley Packet Filter** (**eBPF**), a revolutionary technology redefines what is possible within the Linux kernel. This chapter explores the powerful synergy between eBPF and network monitoring. We will explore how eBPF moves beyond the limitations of traditional tools like tcpdump or NetFlow, enabling the creation of sophisticated, kernel-resident monitors that provide continuous insight into network behavior.

Structure

This chapter covers the following topics:

- Limitations of traditional network monitoring
- Networking and ePBF
- Using eBPF tools
- Types of ePBF program
- Writing an eBPF program
- Tracing network functions
- Monitoring socket-level activity
- Security considerations and best practices
- Network monitoring with Tetragon

Objectives

This chapter aims to equip you with the foundational knowledge and practical understanding required to leverage eBPF for modern, high-performance network monitoring. We will begin by understanding the inherent limitations of traditional monitoring methodologies, such as the sampling inaccuracies of flow-based tools and the performance cost of full packet capture, to establish a clear rationale for the eBPF paradigm. From this foundation, the chapter builds the core architectural principles of eBPF that enable its revolutionary approach, focusing on its secure, sandboxed execution model within the kernel and its efficient just-in-time compilation. We will understand the map of critical eBPF hooking points within the Linux networking stack, including the **eXpress Data Path** (**XDP**) for ultra-fast packet processing. The objective is to translate this architectural knowledge into practical capability. We will learn to design, write, and deploy basic eBPF programs that go beyond passive observation.

Limitations of traditional network monitoring

Before exploring the power of eBPF, it is essential to understand the constraints and fundamental trade-offs in conventional network monitoring

approaches. These established methods, while valuable for decades, struggle to meet the observability demands of modern, high-performance, and dynamic infrastructure.

Packet capture with utilities like **tcpdump** or Wireshark operates on a principle of full sampling. While invaluable for deep-dive analysis, this approach suffers from significant drawbacks. Capturing every packet to the user space for analysis imposes a substantial performance penalty, consuming considerable **central processing unit** (**CPU**) and memory resources, which can itself alter the behavior of the system under observation. Furthermore, the volume of data generated is often overwhelming, making continuous, real-time operation impractical for high-throughput environments. This typically makes packet capture reactive, used intermittently rather than as a source of continuous insight.

To mitigate the overhead of full packet capture, flow-based monitoring systems like NetFlow and sFlow were developed. These tools aggregate traffic into summaries or flows, recording key attributes such as source, destination, ports, and byte counts. While efficient and scalable, this aggregation comes at the cost of granularity. The rich, application-layer context of individual packets and transactions is lost. Flow data can indicate that communication occurred and its volume, but it cannot reveal what was transferred, the exact timing of specific requests and responses, or diagnose complex application-layer issues like **Hypertext Transfer Protocol** (**HTTP**) errors or database query performance.

Another class of tools relies on kernel instrumentation via static probes for packet filtering. While these tools are more efficient than pure user-space capture, this methodology is limited in its expressiveness and flexibility. The kernel’s observation points are fixed and predefined, offering a static view into a dynamic system. If a needed metric is not exposed by the kernel's existing counters or if a new protocol requires monitoring, the only recourse is to wait for and upgrade the entire kernel, which might be an impractical and slow process. This inflexibility creates a persistent visibility gap.

Architecture of an eBPF program

The power of eBPF does not come from a single feature, but from a programmable, safety-first architecture that transforms the Linux kernel into an extensible runtime. At its core, eBPF is a **virtual machine** (**VM**) embedded within the kernel, capable of executing user-defined, sandboxed programs in a privileged context without compromising system stability. This architecture rests on several foundational pillars that work in concert to enable safe, efficient, and powerful in-kernel computation.

The life of an eBPF program begins in user space, where it is typically written in a restricted subset of C or via higher-level toolchains. This source code is then compiled into eBPF bytecode, which is a compact, reduced instruction set designed for the kernel's virtual machine. Before this bytecode can be executed, it must pass through the eBPF Verifier, which acts as the gatekeeper of system security and stability. The verifier performs a rigorous static analysis of the program, enforcing a comprehensive set of safety rules. It ensures the program will terminate by:

- Rejecting loops that are not bounded at compile time.
- Validating all memory accesses to prevent corruption.
- Checking that the program uses a finite amount of stack space.
- Confirming that the program only calls a whitelist of approved helper functions.

This verification step is critical, and only programs that are proven safe are allowed to proceed. The following figure maps the flow of an eBPF program from creation to execution, highlighting the key architectural components:

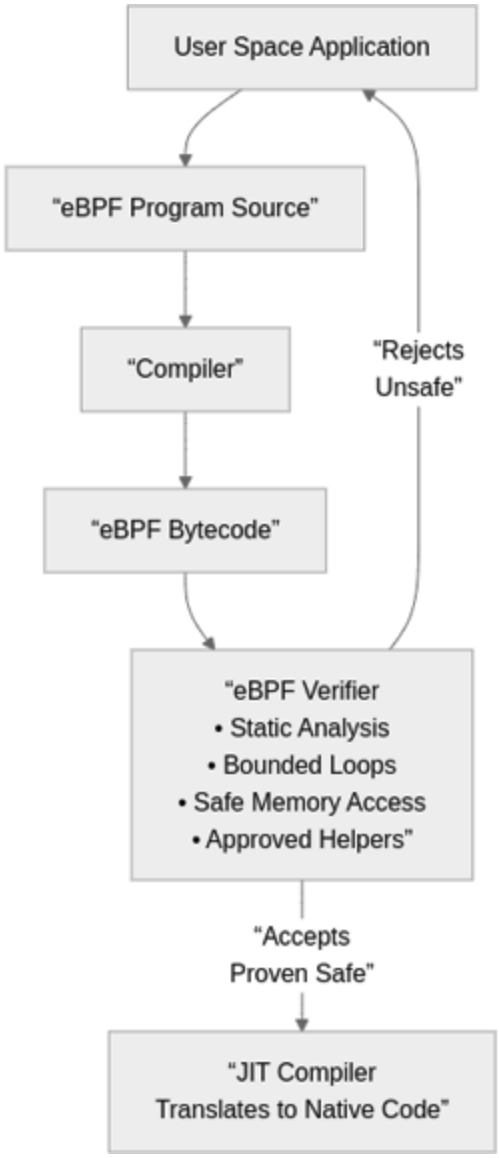


***Figure 12.1**: Verification of eBPF program*

Once validated, the bytecode is **just-in-time** (**JIT**) compiled into the host processor's native machine code. This compilation step eliminates the overhead of interpretation, allowing eBPF programs to execute with near-native performance. This is a prerequisite for their use in high-frequency data paths like networking and performance monitoring.

The compiled program is then attached to a designated hook point within the kernel's execution graph. These hooks are predefined locations, such as network interface arrivals, system call entries, kernel function boundaries, or tracepoint events, where the kernel can invoke external logic. When the

corresponding event occurs at that hook, the kernel schedules the execution of the attached eBPF program. While executing, programs are restricted from performing arbitrary **input/output** (**I/O**) or maintaining state across invocations on their own.

This is where eBPF Maps become essential. Maps are generic, key-value data structures such as hash tables, arrays, or ring buffers that are allocated in kernel memory but shared between the eBPF program and user-space applications. They serve a dual purpose:

- As a communication channel to pass data, metrics configuration to and from the user space.
- As a persistent state store that allows a program to maintain context, aggregate statistics, or coordinate actions across multiple events and program invocations.

The combination of verifier-enforced safety, JIT-compiled speed, strategic hook points, and flexible shared state via maps forms the complete architectural picture that makes eBPF a uniquely powerful platform for system introspection and manipulation.

Networking and eBPF

Integration of eBPF's programmable architecture with the Linux networking stack represents one of its most impactful applications, fundamentally transforming how networks are observed, secured, and optimized. Unlike traditional tools that view the network from the outside, eBPF embeds intelligence directly into the kernel's data plane, enabling real-time processing and decision-making at the speed of the network itself. This integration allows for a new class of monitoring that is continuous, granular, and imposes negligible overhead, moving from statistical sampling and aggregated flows to a world of per-packet, per-connection insight. eBPF achieves this deep visibility by instrumenting the network stack at multiple, strategic layers, providing a choice of observability and control needs. Each hook point offers a unique combination of context and performance characteristics.

XDP is the earliest and fastest hook, executing eBPF programs at the network driver level, immediately as a packet is received and often before it even enters the kernel's main networking stack. This position makes XDP ideal for ultra-high-performance operations like **distributed denial-of-service** (**DDoS**) mitigation, load balancing, and packet filtering, where decisions must be made at line rate with the lowest possible latency. For monitoring, XDP provides a perfect location for precise packet counting, early-drop analysis, and timestamping with nanosecond accuracy.

Traffic control (**TC**) Ingress/Egress hooks operate further up the stack, after initial kernel processing. The TC subsystem provides a richer context, including access to the full kernel socket buffer sk_buff structure, which contains packet metadata and the ability to modify or redirect packets. This makes TC eBPF programs exceptionally powerful for sophisticated traffic shaping, advanced packet filtering, protocol classification, and detailed connection-level accounting that requires understanding packet relationships and state.

Socket-level operations represent the highest layer of instrumentation, where eBPF programs can attach to socket events for both **Transmission Control Protocol** (**TCP**) and **User Datagram Protocol** (**UDP**). Here, programs can trace socket lifecycle events like create, bind, connect, and close to inspect and filter application-layer data, and measure crucial performance metrics like connection latency, retransmissions, and **round-trip time** (**RTT**). This layer bridges the gap between pure network data and application behavior, enabling correlation of network conditions with user-visible performance issues.

Beyond these primary hooks, eBPF also leverages kprobes and tracepoints to instrument internal kernel networking functions. This allows for deep tracing, answering questions about why a packet was dropped, which firewall rule matched, or how a routing decision was made. By combining these instrumentation points, eBPF constructs a complete picture of network activity from the raw bits on the wire to the application's system call.

This multi-layered approach is coordinated through eBPF maps, which allow data collected at a low-level XDP hook to be correlated with socket

events observed higher up the stack. The result is a monitoring paradigm that is not only deep and performant but also rich in context, providing a unified view of network performance, security posture, and application health from a single, programmable framework within the kernel.

Using eBPF tools

While eBPF programs execute within the kernel, their true power is unlocked through userspace tools that facilitate their creation, loading, management, and data visualization. The eBPF ecosystem has evolved a layered toolchain, offering options that range from interactive one-liners for quick diagnostics to comprehensive frameworks for building production monitoring systems. This spectrum is primarily defined by the trade-off between ease of use and flexibility, with bpftrace and **BPF Compiler Collection** (**BCC**) representing the two most prominent paradigms.

The bpftrace tool stands at the high-productivity end of the spectrum. It is a high-level tracing language and runtime. The bpftrace language is structurally similar to the D language used by DTrace. It provides a concise, domain-specific language that allows administrators and developers to write powerful one-liner scripts or short programs without dealing with the lower-level complexities of C, **low level virtual machine** (**LLVM**) compilation, or direct map management. Under the hood, bpftrace scripts are compiled to eBPF bytecode, but this process is entirely abstracted from the user. Its syntax is built around:

- **Probes**, which are the attachment points, like kprobe or tracepoint.
- **Predicates**, which are conditions that must be met to trigger an action.
- **Actions**, which are the code to execute when the probe fires.

The bpftrace's strength lies in its ability to provide immediate, deep insight into network behavior with minimal code. Here are practical examples showcasing its utility for network diagnostics. Let us look at some examples of short bpftrace programs:

- To instantly trace all TCP retransmissions with a single command:

```
$ sudo bpftrace -e 'tracepoint:tcp:tcp_retransmit_skb {
```

```
printf("TCP Retransmit by %s: %s\n", comm, ntop(args->saddr)); }'
```

- This example traces TCP connections:

```
$ sudo bpftrace -e 'kprobe:tcp_v4_connect { printf("TCP connect attempt: %s (PID %d)\n", comm, pid); }'
INFO: node count: 8 Attaching 1 probe...
Program ID: 64
The verifier log: processed 36 insns (limit 1000000) max_states_per_insn 0 total_states 2 peak_states 2 mark_read 1
Attaching kprobe:tcp_v4_connect
```

  - Now, from another window, try to connect to this machine using **Secure Shell** (**SSH**). You should see the output of the probe like the following:

```
TCP connect attempt: ssh (PID 1863)
TCP connect attempt: ssh (PID 1866)
```

- Here is a trace program for counting dropped packets:

```
sudo bpftrace -e '
tracepoint:skb:kfree_skb {
   @drops[args->location] = count();
   printf("Packet dropped at location: %d (reason: %s)\n",
        args->location, args->reason);
}'
Attaching 1 probe...
Packet dropped at location: -1057712609 (reason: 2)
Packet dropped at location: -1057712609 (reason: 2)
Packet dropped at location: -1057712609 (reason: 2)
Packet dropped at location: -1057712609 (reason: 2)
```

- This example generates a histogram of TCP round-trip time:

```
$ cat rtt.bt
tracepoint:tcp:tcp_probe {
```

```
    // This tracepoint has RTT information
    $srtt = args->srtt;
    $rtt = $srtt >> 3;  // Convert from 8*us to us

    if ($rtt > 0 && $rtt < 1000000) {
        @rtt_hist = hist($rtt);
        @rtt_stats = stats($rtt);
    }
}
interval:s:10 {
    printf("\n=== TCP RTT Statistics (μs) ===\n”);
    printf("Histogram:\n”);
    print(@rtt_hist);
    printf("\nStats: ");
    print(@rtt_stats);
    clear(@rtt_hist);
    clear(@rtt_stats);
}
```

- To run the program, execute it with the **bpftrace** command:

```
$ sudo bpftrace rtt.bt
```

- You should see the histogram and statistics generated every 10 seconds:

```
Attaching 2 probes...
=== TCP RTT Statistics (μs) ===
Histogram:
@rtt_hist:
[2K, 4K)      70
|@@@
```

=== TCP RTT Statistics (µs) ===

Histogram:

@rtt_hist:

[2K, 4K) 2
**|@@@

and storing the counter in the **pkt_count** hash. The Python code loads the BCC program by creating an object of BPF and passing the BCC program as **b = BPF(text=bpf_text)**. The handle **b** in the Python program can be used to access the **pkt_count** hash created by the BPF program:

```
#!/usr/bin/env python3
from bcc import BPF
bpf_text = """
#include <uapi/linux/ptrace.h>
// Count packets received
BPF_HASH(pkt_count); // Default key/u64 map
// Trace network receive events
TRACEPOINT_PROBE(net, netif_receive_skb) {
 u64 key = 0;
 u64 *count = pkt_count.lookup_or_init(&key, &key);
 if (count) {
 (*count)++;
 }
 return 0;
}
"""
b = BPF(text=bpf_text) # Load BPF program
print("Counting network packets received...Press Ctrl+C to stop\n")
counters = b["pkt_count"] # Get reference to our counter map
try:
 last_count = 0
 while True:
 # Use b.trace_print() for simple output
 b.trace_print()
# Also show total count
 total = 0
 for k, v in counters.items():
```

```
total = v.value
if total > last_count:
print(f"\nTotal packets: {total}")
last_count = total
except KeyboardInterrupt:
print("\n\nExiting...")
total = 0
for k, v in counters.items():
total = v.value
print(f"Final count: {total} packets")
```

To run the program, simply execute the Python program:

```
$ sudo python3 simple.py
Counting network packets received...Press Ctrl+C to stop
^C

Exiting...
Final count: 101 packets
```

The **libbpf** library represents the core for the modern eBPF ecosystem. It is a C library that provides the lowest-level, most control-oriented API for managing the eBPF lifecycle. Tools built with **libbpf** often use the BPF **Compile Once, Run Everywhere** (**CO-RE**) methodology compile the kernel BPF C code into a single, portable binary object file. This object is then dynamically relocated and loaded by the userspace program at runtime, adapting to the host kernel's specifics. This approach offers significant advantages for production deployment:

- Minimal runtime dependencies
- Smaller binary sizes
- Predictable performance
- Cleaner separation between kernel and userspace code

While more complex to use initially, libbpf is increasingly the recommended path for building production-grade eBPF applications,

including complex network functions and enterprise monitoring agents.

Beyond these core tools, the ecosystem includes specialized utilities. The **bpftool** is a versatile tool for inspecting and manipulating the eBPF subsystem; it can list loaded programs and maps, dump map contents, pin objects to the filesystem, and generate program skeletons. Perf can record and display eBPF-powered trace events. For example, we can use the **bpftool** to dump the map we created in the previous example:

```
$ sudo bpftool map list | grep  "pkt_count"
59: hash  name pkt_count  flags 0x0
$ sudo bpftool map dump id 59
[{
        "key": 0,
        "value": 7356
    }
]
$ sudo bpftool map dump id 59
[{
        "key": 0,
        "value": 7484
    }
]
```

We can see the value of the map changing as we continue to update the map from the previous program, as more packets are received on the host.

Types of eBPF programs

The flexibility of eBPF is realized through a variety of program types, each explicitly designed for a specific class of kernel hook points and operational contexts. This specialization is not arbitrary; the kernel defines these types to enforce strict safety and compatibility guarantees. The program type dictates the signature of the program's main function, its expected input arguments, the subset of helper functions it is permitted to call, and the

specific locations within the kernel where it can be attached. Understanding this classification is essential for selecting the right tool for a given networking, observability, or security task.

At the foundation are program types dedicated to packet processing within the networking data path. The XDP program type is the archetype for high-speed operations, with its main function receiving a direct pointer to the raw packet data as it arrives from the network driver. TC programs, specifically of type **sch_cls** (classifier) and **sch_act** (action), operate on the full **sk_buff** kernel structure, providing the context needed for complex filtering, forwarding, and policy enforcement. For more granular socket control, cgroup/skb programs manage traffic entering or leaving sockets associated with a specific cgroup, enabling advanced container networking and bandwidth management, while socket filter programs (the direct descendant of classic BPF) allow traditional, efficient filtering of packets at the socket level before they are delivered to a user-space application.

Another branch of program types is designed for system observability and tracing. These types grant access to profound internal kernel events rather than just packet flows. kprobe and kretprobe programs can be attached to virtually any kernel function entry and return, respectively, providing the ultimate tool for dynamic instrumentation and debugging of kernel networking code. For more stable and structured tracing, tracepoint programs attach to well-defined, statically instrumented kernel events, offering a reliable interface for monitoring system calls, scheduler decisions, or file operations. The raw_tracepoint programs provide a lower-overhead alternative with direct access to tracepoint arguments. Furthermore, perf_event programs can sample hardware and software performance counters, enabling detailed CPU profiling and correlation of network activity with system-wide performance metrics.

Finally, a critical category exists for control and system integration. cgroup/sysctl programs can intercept and filter system control operations, and **Linux Security Module** (**lsm**) programs hook into the kernel's security framework to implement dynamic security policies. For networking, the ability to attach programs to **Light Weight Tunnel** (**lwt**) in and out points enables programmable encapsulation and routing logic.

The kernel's loading mechanism validates that a program's type matches its intended attach point, creating a secure and predictable environment. This deliberate typing system ensures that a packet-filtering program cannot accidentally be attached to a function tracepoint, preserving both system integrity and programmer sanity. By leveraging the appropriate program type, developers can write focused, efficient, and safe programs that interact with the kernel in precisely the intended manner, unlocking everything from line-rate packet filtering to deep behavioural profiling of distributed applications.

Writing an eBPF program

eBPF programs are small pieces of sandboxed code that run in the Linux kernel. Writing effective eBPF programs requires understanding both the programming model and the various tools available for development. We will begin by first installing the tools and libraries required for building the eBPF programs. This can be done using the following command on Ubuntu:

```
$ sudo apt update
$ sudo apt install clang llvm libbpf-dev iproute2 build-essential linux-tools-common linux-headers-`uname -r`
$ sudo ln -s /usr/include/x86_64-linux-gnu/asm /usr/include/asm
```

Program for inspecting packets with XDP

An XDP program is a specialized eBPF program that runs at the earliest possible point in the Linux networking stack, directly in the network driver, to make high-speed decisions about network packets. The main purpose of an XDP program is to attach a verdict to a packet, which can be used to pass or drop a packet.

Let us now understand the following XDP program, which will count the number of packets received by the interface where the XDP program is attached. It maintains 3 per CPU counters to track Ethernet, IP, and ICMP packets. If an ICMP packet is detected, the XDP program will drop the

packet. The following code should be written in file **xdp_icmp_filter.c**:

```
/* xdp_icmp_filter.c*/
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/in.h>
#include <bpf/bpf_helpers.h>

struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_ARRAY);
    __uint(max_entries, 3);
    __type(key, __u32);
    __type(value, __u64);
} pkt_count SEC(".maps");

/* Helper function for byte order conversion */
static __always_inline __u16 bpf_htons(__u16 hostshort) {
    return ((hostshort & 0xFF) << 8) | ((hostshort & 0xFF00) >> 8);
}

SEC("xdp")
int drop_icmp(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    // Check Ethernet header
    struct ethhdr *eth = data;
    if ((void *)eth + sizeof(*eth) > data_end)
        return XDP_PASS;

    // Filter only IPv4 traffic
    if (eth->h_proto != bpf_htons(ETH_P_IP))
        return XDP_PASS;
```

```
    // Check IP header
    struct iphdr *ip = data + sizeof(*eth);
    if ((void *)ip + sizeof(*ip) > data_end)
        return XDP_PASS;

    __u32 key = 2;
    // Drop ICMP packets (Protocol 1)
    if (ip->protocol == IPPROTO_ICMP) {
        __u64 *value = bpf_map_lookup_elem(&pkt_count, &key);
        if (value) *value += 1;
        return XDP_DROP;
    }

    return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

The following steps describe how to compile and attach the XDP program:

1. To compile the program, use the following command:

```
$ clang -O2 -g -Wall -target bpf -D__TARGET_ARCH_x86 -c
xdp_icmp_filter.c -o xdp_icmp_filter.o
```

2. You can dump the object file created by the compiler to check the sections:

```
$ llvm-objdump -h xdp_icmp_filter.o

xdp_icmp_filter.o: file format elf64-bpf

Sections:
Idx Name          Size     VMA              Type
 0                00000000 0000000000000000  1 .strtab       0000012a
0000000000000000  2 .text          00000000 0000000000000000
TEXT
 3 xdp            000001b8 0000000000000000 TEXT
 4 .relxdp        00000030 0000000000000000  5 .maps
```

```
00000020 0000000000000000 DATA
 6 license        00000004 0000000000000000 DATA
...
Truncated output
```

3. To attach the program to a network interface (**lo** in this case), use the following command:

```
$ sudo ip link set dev lo xdp obj xdp_icmp_filter.o sec xdp
```

4. Use the following command to verify that the **xdp** program is attached to the interface:

```
$ ip link sh dev lo
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 xdpgeneric
qdisc noqueue state UNKNOWN mode DEFAULT group default
qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   prog/xdp id 205
$ sudo bpftool net list
xdp:
enp1s0(2) driver id 185
```

5. To test the program, you can try pinging the localhost, which should fail as the **xdp** program attached to the **lo** interface drops the packets:

```
$ ping -c9 localhost
PING localhost (127.0.0.1) 56(84) bytes of data.


--- localhost ping statistics ---
9 packets transmitted, 0 received, 100% packet loss, time 8232ms
```

6. You can find the id of the map **pkt_count** defined in the program:

```
$ sudo bpftool map list |grep
 pkt_count 543: percpu_array name pkt_count flags 0x0
```

7. Use the following command to view the value of the packet counter, using the **id** of the map from the previous step:

```
$ sudo bpftool map dump id 543
```

```
[{
        "key": 0,
        "values": [{
                "cpu": 0,
                "value": 0
            },{
                "cpu": 1,
                "value": 9
            }
        ]
    },{
        "key": 1,
        "values": [{
                "cpu": 0,
                "value": 0
            },{
                "cpu": 1,
                "value": 9
            }
        ]
    },{
        "key": 2,
        "values": [{
                "cpu": 0,
                "value": 0
            },{
                "cpu": 1,
                "value": 9
            }
        ]
```

```
    }
]
```

This sample program demonstrates the core concepts of an eBPF program.

Tracing network functions

Tracing network functions represents one of the most powerful applications of eBPF, enabling deep introspection into the Linux kernel's networking stack without modifying kernel source code or rebooting. Unlike packet-level monitoring that observes what flows through the network, function tracing reveals how and why the kernel processes network traffic, exposing the internal decision-making logic, performance bottlenecks, and error conditions that remain invisible at higher abstraction levels. This capability transforms network behavior into a transparent, observable system where every packet's journey through kernel space can be traced, measured, and analyzed.

The foundation of network function tracing lies in eBPF's ability to attach to kernel functions via kprobes for dynamic tracing of any kernel function and tracepoints for static, stable instrumentation points.

Tracepoints vs kprobes

Tracepoints are static instrumentation points built into the kernel source code. They are predefined by kernel developers at specific locations and are part of the linux tracepoints subsystem. Tracepoints provide more stability across kernel versions, with well-defined arguments and semantics for major networking events. The basic format of using tracepoint with bpftrace is as follows:

```
bpftrace -e 'tracepoint:subsystem:event { commands }'
```

The following example uses tracepoint to trace the socket state:

```
sudo bpftrace -e 'tracepoint:sock:inet_sock_set_state { printf("TCP state change: %d\n", args->newstate); }'
```

To get a complete list of tracepoints, use the following command:

```
$ sudo bpftrace -l tracepoint:*
```

Kprobes are dynamic instrumentation of arbitrary kernel functions. They can probe any function. This includes kprobes, which can trace entry into a function, and kretprobes, which can trace return from a function. The following example shows the use of kprobe and kretproce with the bpftrace tool:

```
sudo bpftrace -e 'kprobe:tcp_v4_connect {
 printf("TCP connect by %s (PID: %d)\n", comm, pid);
}'

sudo bpftrace -e 'kretprobe:tcp_v4_connect {
 printf("TCP connect returned: %d\n", retval);
}'
```

Kprobes offer unparalleled flexibility, allowing instrumentation of thousands of internal networking functions from the high-level socket **application programming interface** (**API**) down to driver-level packet handling. Together, kprobe and tracepoints create a comprehensive observability framework that can answer questions about network performance and behavior.

The Linux networking stack comprises several layers, each with distinct functions that can be traced. The following table provides a list of some of the commonly used functions for tracing with kprobe at different layers of network packet processing:

Socket layer functions	
sock_sendmsg, sock_recvmsg	TCP-specific send/receive handling
inet_bind, inet_connect, inet_accept	Connection lifecycle management
sock_alloc, sock_init_data	Socket object creation and initialization
TCP protocol processing	
tcp_v4_connect, tcp_v6_connect	TCP connection establishment
tcp_rcv_state_process	TCP state machine transitions
tcp_ack, tcp_clean_rtx_queue	Acknowledgment processing

tcp_retransmit_skb, tcp_retransmit_timer	Retransmission handling
tcp_cong_avoid, tcp_slow_start	Congestion control algorithms
IP layer functions	
ip_rcv, ip_output	IP packet reception and transmission
ip_forward	Packet forwarding decisions
ip_local_deliver	Delivery to local sockets
ip_route_input, ip_route_output	Routing decisions
Queueing discipline (qdisc) functions	
enqueue_to_backlog, __netif_schedule:	Packet queuing
qdisc_run	Queue processing
Device driver level	
netif_receive_skb, netif_rx	Packet reception into stack
dev_queue_xmit	Packet transmission to device

***Table 12.1**: Network tracing functions*

Network function tracing with eBPF provides unprecedented visibility into the Linux kernel's networking implementation, transforming opaque network behavior into a richly instrumented, observable system. By strategically placing probes at critical junctions in the packet processing pipeline, developers and operators can diagnose complex issues, optimize performance, and understand system behavior in ways previously requiring custom kernel builds or extensive static instrumentation. The combination of low overhead, safety guarantees, and programmability makes eBPF-based tracing an essential tool for anyone responsible for the reliability and performance of networked systems.

Monitoring socket-level activity

Socket-level monitoring provides the critical link between raw network packets and application behavior, offering visibility into how programs

communicate over the network. While packet-level hooks like XDP and TC see the raw data, socket-level instrumentation reveals the intent and context of network operations, which process is communicating, which system calls it is using, and how efficiently the operating system is managing connections. eBPF transforms socket monitoring from coarse-grained statistics into a real-time, per-connection observability platform.

The socket abstraction in Linux represents the endpoint of network communication, and eBPF can instrument this layer at multiple points. At the system call interface, eBPF programs can attach to socket-related calls like **socket()**, **bind()**, **connect()**, **accept()**, **send()**, **recv()**, and their variants. This reveals application intent, such as which ports a service binds to, which remote hosts it connects to, and the volume of data exchanged. More importantly, eBPF can trace internal kernel functions within the socket layer through kprobes, exposing details like TCP state transitions, retransmission events, buffer management, and congestion control decisions that remain invisible at the system call level.

Several eBPF program types are specifically designed for socket monitoring. Socket filter programs (the original BPF type, now extended as eBPF) attach directly to sockets, enabling efficient filtering and analysis of traffic for specific applications, similar to tcpdump, but programmable and without copying packets to userspace. Cgroup socket programs operate on traffic entering or leaving control groups, making them ideal for containerized environments where monitoring needs to follow application boundaries. These programs can allow, deny, or redirect socket traffic based on sophisticated policies. Socket operations programs trace the lifecycle of socket objects themselves, tracking creation, binding, connection establishment, and teardown with detailed context about the owning process and connection parameters.

The power of socket-level eBPF monitoring becomes evident in practical observability scenarios. Connection tracking moves beyond simple counters to capture the complete lifecycle of each socket, recording timestamps for state transitions from **SYN_SENT** to **ESTABLISHED** to **CLOSE_WAIT**. Following is an example of tracking socket state with the bpftrace tool:

```
$ sudo bpftrace -e '
```

```
tracepoint:sock:inet_sock_set_state {
 $state_str = "UNKNOWN";

 // Use actual numeric values
 if (args->newstate == 1) { $state_str = "ESTABLISHED"; }
 else if (args->newstate == 2) { $state_str = "SYN_SENT"; }
 else if (args->newstate == 3) { $state_str = "SYN_RECV"; }
 else if (args->newstate == 4) { $state_str = "FIN_WAIT1"; }
 else if (args->newstate == 5) { $state_str = "FIN_WAIT2"; }
 else if (args->newstate == 6) { $state_str = "TIME_WAIT"; }
 else if (args->newstate == 7) { $state_str = "CLOSE"; }
 else if (args->newstate == 8) { $state_str = "CLOSE_WAIT"; }
 else if (args->newstate == 9) { $state_str = "LAST_ACK"; }
 else if (args->newstate == 10) { $state_str = "LISTEN"; }
 else if (args->newstate == 11) { $state_str = "CLOSING"; }

 printf("TCP %s:%d → %s:%d state: %s (PID: %d)\n",
 ntop(args->saddr), args->sport,
 ntop(args->daddr), args->dport,
 $state_str, pid);
}'
```

On running the preceding code and connecting the host with **nc $host_ip 22** you should see output like the following:

```
Attaching 1 probe...
TCP 0.0.0.0:22 → 0.0.0.0:0 state: SYN_RECV (PID: 0)
TCP 192.168.122.155:22 → 192.168.122.1:39284 state:
ESTABLISHED (PID: 0)
TCP 192.168.122.155:22 → 192.168.122.1:39284 state: FIN_WAIT1
(PID: 6009)
TCP 192.168.122.155:22 → 192.168.122.1:39284 state: FIN_WAIT2
(PID: 0)
TCP 192.168.122.155:22 → 192.168.122.1:39284 state: CLOSE (PID:
0)
```

This enables precise latency measurements, not just round-trip times, but connection establishment delays, application think times, and graceful

closure periods. Throughput and buffer analysis reveal how efficiently applications are utilizing their sockets; eBPF can measure send and receive queue depths, monitor for zero-window conditions indicating stalled receivers, and detect when applications are not reading data promptly, causing buffers to fill. Application protocol decoding, while more advanced, becomes possible by inspecting the data flowing through sockets, allowing for HTTP request tracing, **Domain Name System** (**DNS**) query analysis, or database protocol monitoring without modifying the applications themselves.

Perhaps the most valuable application is correlating socket activity with system resources. eBPF maps can associate socket operations with process metadata, CPU usage, memory allocation patterns, and file descriptor consumption. This holistic view answers critical operational questions like *Which process is causing a spike in connections? Is network latency due to application delays or kernel queuing? Why is connection establishment slowing down under load?* The kernel's socket structures contain rich metadata, peer addresses, ports, queue lengths, timestamps, and congestion window sizes, all accessible to eBPF programs.

Deploying socket-level monitoring requires careful consideration of attachment points. System call probes offer portability but may miss internal kernel events. Kprobes provide deeper visibility but require more maintenance across kernel versions. The choice depends on whether the monitoring goal is understanding application behavior, favoring system calls, or diagnosing kernel networking issues, favoring kprobes. Performance overhead is generally low, as eBPF programs run synchronously with the operations they monitor, but programs that perform complex parsing or maintain extensive per-connection state should be benchmarked under production loads.

In modern architectures, socket monitoring extends to security enforcement. eBPF programs can implement dynamic allow/deny policies based on process identity, connection patterns, or data content, providing a programmable alternative to traditional firewall rules that operate at the socket layer with full process context. Container orchestrators increasingly leverage eBPF for network policy enforcement, where cgroup-based socket

programs can isolate container traffic without network namespace manipulation.

The socket-level eBPF monitoring closes the observability gap by instrumenting the interface where applications meet the network stack. It provides the context needed to diagnose performance issues, enforce policies, and understand communication patterns in increasingly complex distributed systems.

Security considerations and best practice

The deployment of eBPF for network monitoring introduces powerful capabilities but also significant security implications that must be carefully managed. eBPF programs are executed within the kernel context, operating with the same privilege level as the operating system itself. This privileged position means that while eBPF enables unprecedented observability and control, it also creates new attack surfaces and potential risks that traditional userspace monitoring tools do not present. Adopting eBPF requires a security-first mindset, balancing the benefits of deep visibility against the risks of kernel-level code execution.

The following list provides an insight into security risks and attack vectors when eBPF is enabled:

- **Kernel integrity threats**:
  - **Malicious eBPF programs**: A compromised eBPF program could manipulate kernel data structures, bypass security controls, or create backdoors. While the verifier prevents obvious malicious code, sophisticated attacks might exploit verification limitations.
  - **Kernel pointer exposure**: eBPF programs can read kernel pointers; while they cannot directly dereference arbitrary pointers, information leaks about kernel memory layout could aid other attacks.
  - **Map data tampering**: Shared maps between kernel and userspace could be manipulated if not properly secured, leading to incorrect monitoring data or denial of service.

- **Privacy and data exposure**
  - **Packet content inspection**: eBPF programs can examine full packet payloads, potentially exposing sensitive data like credentials, personal information, and encryption keys to monitoring systems.
  - **Metadata collection**: Connection tracking and socket monitoring can reveal communication patterns, timing information, and system internals that might be considered sensitive.
  - **Side-channel information**: Precise timing measurements and resource usage data could be used in side-channel attacks against co-located workloads.
- **Denial of service risks**:
  - **Resource exhaustion**: Poorly written eBPF programs could consume excessive CPU, memory, or map storage, affecting system stability.
  - **Kernel lock contention**: Incorrect map usage (non-per-CPU maps in hot paths) can cause lock contention that slows the entire system.
  - **Verifier resource consumption**: Complex programs might require extensive verifier computation, potentially enabling DoS attacks through specially crafted BPF bytecode.
- **Privilege escalation vectors**:
  - **Verifier bypass**: Historical vulnerabilities have shown that verifier bugs can be exploited to execute arbitrary kernel code.
  - **Helper function abuse**: eBPF helper functions, while carefully vetted, have had security issues that could be exploited.
  - **JIT compiler vulnerabilities**: The JIT compilation process itself could be targeted, though modern implementations have extensive hardening.

Security in eBPF network monitoring is not an optional add-on but a fundamental requirement that must be integrated into every phase of the development and deployment lifecycle. Here are some steps that can help mitigate the security risks of enabling eBPF:

1. **Principle of least privilege**: The eBPF programs can be run with the least capability:

```
# Run eBPF loaders with only necessary capabilities

sudo setcap cap_bpf,cap_perfmon,cap_net_raw+eip
/usr/local/bin/ebpf-monitor

# Better: Use Linux Security Modules (LSMs) for finer control
# apparmor profile for eBPF loader
/usr/local/bin/ebpf-monitor {
 capability bpf,
 capability perfmon,
 capability net_raw,
 deny network inet raw,
 /sys/fs/bpf/** rw,
 /sys/kernel/debug/tracing/** r,
}
```

2. **Program verification and validation**: Use static analysis tools specific to BPF, like **bpflint**, **coccinelle**, and ensure security review before deployment.
3. **Configure audit logging for eBPF operations**: Enable kernel security features to monitor and contain the security risk:

```
echo 1 > /proc/sys/kernel/unprivileged_bpf_disabled
echo 1 > /proc/sys/kernel/bpf_stats_enabled # Monitor BPF resource
usage
echo 1 > /proc/sys/kernel/kptr_restrict # Restrict kernel pointer
exposure

# Configure audit rules for eBPF
auditctl -a always,exit -F arch=b64 -S bpf

# Use kernel lockdown in production
echo integrity > /sys/kernel/security/lockdown
```

In the event of a security incident, these steps can help contain the exposure

caused by eBPF programs:

1. Check all eBPF programs loaded into the kernel:

```
sudo bpftool prog list
```

2. Check audit logs of eBPF programs:

```
sudo ausearch -k bpf
```

3. Monitor system calls related to BPF:

```
sudo strace -e bpf -p $(pidof ebpf_loader)
```

4. Unload all BPF programs:

```
sudo bpftool prog list | grep -o '[0-9]*:' | cut -d: -f1 | \
  xargs -I{} sudo bpftool prog unload id {}
```

The unique position of eBPF programs executing within the kernel, yet dynamically loadable, creates both unprecedented opportunities and significant risks. By adopting a defense-in-depth approach that combines kernel security features, careful program design, runtime monitoring, and organizational controls, organizations can safely harness the power of eBPF for network observability while maintaining system integrity and data protection. As the eBPF ecosystem continues to evolve, so must security practices, requiring ongoing vigilance, education, and adaptation to emerging threats and best practices.

Network monitoring with Tetragon

In the previous sections, we have explored the foundations of tracing and filtering with eBPF. In this section, we will look at Tetragon. Tetragon is a powerful, eBPF-based security observability and runtime enforcement platform that provides deep, real-time visibility into network activity at the process and system call level. Unlike traditional network monitoring tools that operate at the packet or flow level, Tetragon focuses on the critical intersection between applications and the network, tracking which processes are making network connections, what system calls they are using, and how they are interacting with the network stack. This process-aware network monitoring bridges the gap between network observability

and security, answering not just what is happening on the network, but which process is responsible and why.

Tetragon's architecture is built on several key eBPF components that work together to provide comprehensive network monitoring:

1. **Process-centric network tracing**: Tetragon instruments the kernel at multiple layers to correlate network events with process execution:
    - System call tracing (connect, bind, accept, send, recv).
    - Socket lifecycle tracking from creation to destruction.
    - Network namespace awareness for containerized environments.
    - cgroup-based filtering for Kubernetes pod-level visibility.
2. **Network monitoring and policy enforcement**: Tetragon extends beyond monitoring to active enforcement through eBPF-powered network policies:
    - Real-time connection tracking with process context.
    - Monitor DNS queries with full process context.
    - Tetragon can perform deep packet inspection for specific protocols.
    - Tetragon excels in containerized environments with built-in Kubernetes awareness.

Let us examine a few of the network policies supported by Tetragon by setting up a lab. Tetragon supports deployment as a systemd service, docker container, and as a kubernetes deployment. For this lab, we will use the Docker container approach.

Use the following command to run the tetragon container with Docker:

```
docker run --name tetragon --rm -d                \
  --pid=host --cgroupns=host --privileged         \
  -v /sys/kernel/btf/vmlinux:/var/lib/tetragon/btf \
  quay.io/cilium/tetragon:v1.6.0
```

This command will start the Docker container for Tetragon in privileged mode, and the container will share the host's PID namespace. The container

also mounts **/sys/kernel/btf/vmlinux** from the host to the container. This is a special file in modern Linux systems that contains the **BPF Type Format** (**BTF**) information for the kernel. It serves as a complete, self-describing blueprint of the kernel's data structures, function prototypes, and type information, enabling eBPF programs to be portable across different kernel versions through a technology called CO-RE.

Once the container is running, the events from the **tetragon** service can be retrieved with the Tetragon client CLI tetra using the following command:

```
docker exec -ti tetragon tetra getevents -o compact
```

You should see output like the following:

```
🚀 process 8fe6a94f593f /usr/bin/curl google.com
💥 exit 8fe6a94f593f /usr/bin/curl google.com 0
🚀 process 8fe6a94f593f /usr/bin/curl youtube.com
💥 exit 8fe6a94f593f /usr/bin/curl youtube.com 0
💥 exit 8fe6a94f593f [kthreadd] 0
```

To configure the **tetragon** service with a network policy, we can restart the Docker container using the following command:

```
docker run --name tetragon --rm -d                \
   --pid=host --cgroupns=host --privileged        \
   -v /sys/kernel/btf/vmlinux:/var/lib/tetragon/btf \
   -v
$PWD/network_egress_cluster_subst.yaml:/etc/tetragon/tetragon.con
f.d/network_egress_cluster.yaml \
   quay.io/cilium/tetragon:v1.6.0
```

This will mount your custom network policy **network_egress_cluster_subst.yaml** as a config file for the Tetragon service. The following are some examples of network policies we can use with Tetragon to:

- **Detect unauthorized network access**: The following tracing policy detects network connections from IP addresses not in the subnet **"192.168.1.0/24"**, **"10.0.0.0/8"**:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
 name: "detect-unauthorized-connections"
spec:
 kprobes:
 - call: "__x64_sys_connect"
 syscall: true
 selectors:
 - matchArgs:
 - index: 1 # sockaddr
 operator: "NotIn"
 values: ["192.168.1.0/24", "10.0.0.0/8"]
 matchActions:
 - action: "Alert"
 argAlert: "Unauthorized external connection attempt"
```

- **Monitor sensitive data exfiltration**: The following policy will detect transfer of data of size greater than 1 MB:

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
 name: "data-exfiltration-monitoring"
spec:
 kprobes:
 - call: "__x64_sys_send"
 syscall: true
 args:
 - index: 0
 type: "int"
 - index: 1
 type: "const void*"
 - index: 2
 type: "size_t"
 selectors:
 - matchArgs:
```

```
 - index: 2 # size
 operator: "GreaterThan"
 values: ["1048576"] # 1MB
 matchActions:
 - action: "Post"
 argAlert: "Large outbound data transfer detected"
```

- **Network performance analysis**: The following example shows how to collect observability for latency in network connections.

```
apiVersion: cilium.io/v1alpha1
kind: TracingPolicy
metadata:
 name: "network-latency-monitoring"
spec:
 kprobes:
 - call: "__x64_sys_connect"
 syscall: true
 return: true
 selectors:
 - matchActions:
 - action: "Metric"
 argMetric: "connect_latency"
 argOp: "Observe"
```

Tetragon represents a significant evolution in network monitoring by bridging the gap between network observability and process-aware security. Its eBPF-based architecture enables real-time monitoring of network activity with full process context, container awareness, and enforcement capabilities. Unlike traditional network monitoring tools that see only packets and flows, Tetragon reveals the complete story, which process initiated a connection, what system calls were used, what data was transferred, and whether the behavior matches expected patterns.

For organizations running modern, containerized workloads, Tetragon provides a unified platform for both observability and security, eliminating the need to correlate data from multiple monitoring systems. Its integration with Kubernetes, minimal performance overhead, and powerful policy

engine make it an essential tool for securing and understanding network activity in complex distributed systems.

As networks continue to evolve with increasing complexity and security challenges, Tetragon's process-aware network monitoring offers a fundamentally better approach that understands not just what is happening on the wire, but why, and provides the controls to ensure it happens securely and correctly.

Conclusion

Network monitoring has undergone a fundamental transformation with the advent of eBPF. By moving the monitoring intelligence into the kernel itself safely, efficiently, and programmatically, eBPF enables a paradigm where every packet, every socket, and every network function can be observed in real time with minimal performance impact.

Throughout this chapter, we have explored how eBPF redefines network monitoring through its unique architectural pillars: the safety guarantees of the verifier, the performance of JIT compilation, the strategic placement of hook points throughout the networking stack, and the flexible data sharing via maps. Network engineers can now move beyond aggregated flow statistics and sampled packet capture to continuous, per-connection monitoring that reveals not just what is happening, but why. As emphasized, eBPF programs execute within the kernel's privileged context, making security considerations paramount. The verification system, while sophisticated, is not infallible; program complexity must be balanced against safety, and deployment must follow rigorous best practices. Organizations must establish clear policies around data collection, retention, and access, particularly when monitoring encrypted traffic or sensitive systems.

Index

A

Address Resolution 24
Address Resolution, illustrating 24-26
Amazon Web Service (AWS) 330
AWS, ensuring 331-333
Azure Virtual Network (VNet) 333
Azure VNet, configuring 333-335

B

Bandwidth-Delay Product (BDP) 256
BBR 270
BBR, configuring 270, 271
BDP, integrating 256-259
Berkeley Internet Name Domain (BIND) 134
BGP, advantages 187, 188
BGP, configuring 185-187
BGP, steps 184, 185
BIND, ensuring 134-137
Border Gateway Protocol (BGP) 184
Bridge Network Interface 15
Bridge Network Interface, illustrating 15, 16
Bufferbloat 241, 242
Bufferbloat, techniques
  CoDel 243
  FQ 242

C

Calico BGP-Based 329
Chrony 143
Chrony, configuring 143-145
CIDR 40
CIDR, configuring 40, 41
Cilium eBPF-Based 329, 330
Classful Addressing 39, 40
Cloud Network 330
Cloud Network, technologies
  Amazon Web Service (AWS) 330
  Azure Virtual Network (VNet) 333

Google Cloud Platform (GCP) 335
CNI, configuring 321-323
CNI Plugin Workflow 219, 220
CNI, sources
Docker Containers 324
Flannel Plugin 323
CNI, types
Calico BGP-Based 329
Cilium eBPF-Based 329
Congestion Control 269
Congestion Control, signals
Packet Delay 269
Packet Loss 269
Congestion Control, terms
BBR 270
CUBIC 269
Connection Tracking 72
Connection Tracking, configuring 72, 73
Container Network Interface (CNI) 219, 321
Container Network Model (CNM) 320, 321
Crypto Routing 286
CUBIC 269
CUBIC, steps 269, 270

D

Destination NAT (DNAT) 75
DHCP 41
DHCP, configuring 41, 42
DHCP Wtih Linux, ensuring 42-45
Differentiated Services Code Point (DSCP) 232
DNAT, configuring 75, 76
DNS, configuring 132-134
DNS, steps 131, 132
Docker Container 214
Docker Container, components
Endpoint 217
Network 217
Sandbox 216
Docker Container, configuring 215
Docker Container, drivers
Bridge 214
Host 214
None 214
Docker Container, inspecting 216
Docker Containers 324
Docker Containers, steps 325-328
Domain Name System (DNS) 130
DSCP, configuring 232, 233

DShield 81
DShield, architecture 81
DShield, steps 82, 83
Dynamic Routing 55
Dynamic Routing, illustrating 55-61

E

eBPF Program 340, 341
eBPF Program, architecture 343, 344
eBPF Program, configuring 341-343
eBPF Program, insights
  Deniel of Service Risks 359
  Kernal Integrity Threats 358
  Privacy/Data Exposure 359
  Privilege Escalation Vectors 359
eBPF Program, integrating 344-348
eBPF Program, steps 359, 360
ECMP, integrating 48-51
E-Commerce API 174
E-Commerce API, integrating 174, 175
Equal-Cost Multi-Path (ECMP) 48

F

Fault Tolerance 27
Fault Tolerance, integrating 27-31
FIFO, configuring 233, 234
File Transfer 109
First-In First-Out (FIFO) 233
Flannel Plugin 323
Flannel Plugin, configuring 323
Fundamental Problem 227

G

GCP, configuring 335-337
Generic Routing Encapsulation (GRE) 312
Google Cloud Platform (GCP) 335
GRE, consequences
  Inefficiency 312
  Management Overhead 312
  Resource Consumption 312

H

HA, ensuring 182-184
HAProxy 190
HAProxy, blocks
  Backend 192

Frontend 191
Global/Default 191
hash-based ipsets 78
Hierarchical Tree Bandwidth (HTB) 236
High Availability (HA) 166
Host-Level Firewall 221
Hostname 5
HTB, configuring 236, 237
HTTP, architecture 153, 154
HTTP, technologies
NGINX 154
Reverse Proxy 158
SSL Certificates 156
Upstream 160
Hypertext Transfer Protocol (HTTP) 153

I

Ingress/Egress Policing 246
IP Address 6
IP Address, configuring 36, 37
Ip Command 201
Ip Command, configuring 201, 202
IPRoute2 201
IPsec 290
IPsec Authentication, methods
Certificates 294
Pre-Shared Keys (PSK) 294
IPsec, configuring 291, 292
IPsec, mode
Transport 292
Tunnel 292
ipsets 76
ipsets, architecture 77
ipsets, purpose 78
ipsets, testing 79-81
ipsets, types
hash-based 78
list-based 78
timeout/metadata-enabled 79
iptables 64
iptables, configuring 71
iptables, ensuring 67-70
iptables, fundamentals
Firewall Chains 66
Firewall Rules 66
Firewall Tables 66
iptables, reasons
Fundamental Knowledge 65

Granular Control 65
Scriptability 65
Widespred Adoption 65
iptables, structure 65
IPVLAN 206
IPVLAN, architecture 208, 209
IPVLAN, configuring 207
IPVLAN, implementing 209, 210
IPVLAN/MACVLAN, comparing 208
IPVS 170, 171
IPVS, ensuring 171-174
IPVS, roles
E-Commerce API 174
Transport Layer Security (TLS) Cluster 178
UDP-Based Traffic 180

K

Keepalived 167
Keepalived, configuring 167-170
Key-Based Authentication 147
Key-Based Authentication, steps 147-150
Kprobes 354
Kubernetes 217
Kubernetes, configuring 218
Kubernetes, foundations
CNI Plugin Workflow 219
Container Network Interface (CNI) 219

L

LACP, illustrating 22-24
Layer 2 Networks 24
Layer 2 Networks, protocols
Address Resolution 24
Fault Tolerance 27
Neighbour Discovery 31
Layer 3 Routing 45
Layer 3 Routing, integrating 45-48
Layer 7 Load Balancer 188
Layer 7 Load Balancer, configuring 188-190
Link Aggregation 18
Link Aggregation Control Protocol (LACP) 22
Link Aggregation, ensuring 18-21
Linux Distributions 5
Linux Distributions, sections
Hostname 5
IP Address 6
Rocky Linux 7

Ubuntu 9
Linux Firewall Management 89
Linux Firewall Management, configuring 89-91
List-based ipsets 78
List open files (Lsof) 110
Lsof, integrating 110-112

M

MACVLAN 203
MACVLAN, configuring 205, 206
MACVLAN, mode
Bridge 204
Passthrough 205
Private 205
VEPA 204
Modern Web Server 272
Modern Web Server, configuring 272-274
Modern Web Server, implementing 275-277
MTR, advantages
Continuous Monitoring 103
Flexible Probe 103
Statistical Insights 103
User-Friendly Interface 103
MTR, configuring 101
MTR, issues
Latency Spikes, analysing 102
Packet Loss, identifying 102
Reverse Paths, comparing 102
Routing Fluctuations, detecting 102
My Traceroute (MTR) 101

N

Neighbour Discovery 31
Neighbour Discovery, illustrating 32, 33
Netcat 107
Netcat, configuring 107
Netcat, sources
File Transfer 109
Socket Statistics (SS) 109
Temporary Server/Client 108
Netfilter 247
Netfilter, ensuring 247-249
NetFlow 124
NetFlow Data, exporting 127
NetFlow, ensuring 124-126
Netmask/Subnetting 38
Netmask/Subnetting, configuring 38, 39

NetStat, integrating 103-107
Network Address Translation (NAT) 73
Network Debugging 220
Network File System (NFS) 151
Network Namespace 196
Network Namespace, adoptions
  CI/CD 198
  Cloud Multi-Tenancy 198
  Container Runtime 198
  Network Function Virtualization (NFV) 198
Network Namespace, components
  ARP Cache 199
  Netfilter Rules 199
  Network Devices 199
  Protocol Stacks 199
  Routing Tables 199
Network Namespace, configuring 197
Network Namespace, isolating 202
Network Namespace, lifecycles
  Inspection/Debugging 200
  Namespace Interaction 200
NetworkPolicy 222
Network Statistics (NetStat) 103
Network Time Protocol (NTP) 139
Network Tuning 266
Network Tuning, terms
  Packet Drops 267
  SYN Floods 266
NFS Client 152
NFS Client, steps 152, 153
NFS, foundations
  Client 152
  Server 151
NFS Server 151
NFS Server, steps 151, 152
nftables 84
nftables, foundamentals
  Chains 85
  Rules 86, 87
  Tables 85
  User Families 84
NGINX 154
NGINX, configuring 162, 163
NGINX, ensuring 154-156
NTP, configuring 142, 143
NTP, integrating 139-141

O

Open vSwitch (OVS) 312
OSI Model 2
OSI Model, architecture 5
OSI Model, layers
  Application 4
  Datalink 3
  Network 3
  Physical 3
  Presentation 4
  Session 4
  Transport 4
OVS, configuring 312, 313
OVS, ensuring 318-320
OVS, steps 314, 315

P

Packet Capturing 112
Packet Capturing, patterns
  Combine Filter Rule 116
  Host-Based Filtering Rule 114
  Network-Based Filtering Rule 114
  Network Port-Based Filtering Rule 115
  Packet Size-Based Rule 115
  Protocol Based Filtering Rule 115
  TCP Flag-Based Filtering Rule 115
Packet Drops 267
Packet Drops, configuring 268
PBR, illustrating 52-55
PCAP 117
PCAP, integrating 117-119
Ping 94
Ping, ensuring 94-97
PKI 299
PKI, architecture 300
Point-to-Site VPN 288
Point-to-Site VPN, steps 288-290
Policy-Based Routing (PBR) 52
Port Forwarding 150
Port Forwarding, types
  Dynamic 150
  Local 150
  Remote 150
PRIO 239
PRIO, configuring 239, 240

Q

QoS 227, 228

QoS, mechanisms
  Bandwidth 228
  Jitter 229
  Latency 228
  Packet Loss 229

R

Remote Access Certificates 301
Remote Access Certificates, steps 301-304
Reverse Proxy 158
Reverse Proxy, steps 158, 159
RNDC 137
RNDC, configuring 137-139
Rocky Linux 7, 8

S

SDN, configuring 315-317
SDN, parts
  Actions 317
  Match Fields 317
  Priority 317
SDN, planes
  Control 315
  Data 315
Secure Shell (SSH) 145
Security Hardening 222
SFQ, configuring 238
Site-to-Site VPN 294
Site-to-Site VPN, steps 295-299
SNAT, configuring 74
Socket Inspection 103
Socket-Level Monitoring 356
Socket-Level Monitoring, configuring 356-358
Socket Statistics (SS) 109
Software-Defined Networking (SDN) 315
Source NAT (SNAT) 74
SSH, architecture 145, 146
SSH, sources
  Key-Based Authentication 147
  Port Forwarding 150
SSH, steps 146
SSL Certificates 156
SSL Certificates, ensuring 156-158
Stochastic Fairness Queuing (SFQ) 237
StrongSwan 293, 300
StrongSwan/WireGuard, comparing 304, 305
SYN Floods 266

SYN Floods, steps 267
Sysctl 254
Sysctl, ensuring 254-256
systemd-timesyncd 142

T

TBF, configuring 234, 235
TCP 259
TCP, configuring 260-262
tcpdump 113
TCP, foundation
  Connection Persistence 266
  Connection Teardown 264
  SYN Backlog Tuning 263
TCP, steps 260
Temporary Server/Client 108
Tetragon 361
Tetragon, illustrating 361-364
timeout/metadata-enabled 79
TLS Cluster, integrating 179
Token Bucket Filters (TBF) 234
Tracepoints 354
Traceroute 97, 98
Traceroute, configuring 99, 100
Traceroute, insights
  Asymmetric Paths 99
  Network Congestion, detecting 99
  Network Loop 99
  Saturated Link 99
Traceroute, limitations 100
Tracing Network Functions 354
Traffic Control System 229
Traffic Control System, architecture 230
Traffic Control System, components
  Actions 231
  Class 231
  Filters 231
  Qdisc/Queuing 230
Traffic Shaping 245
Traffic Shaping, configuring 245, 246
Transport Layer Security (TLS) Cluster 178
TShark 123
TShark, configuring 123
TUN/TAP 212
TUN/TAP, configuring 213

U

Ubuntu 9
UDP-Based Traffic 180
UDP-Based Traffic, integrating 180, 181
Upstream 160

V

VETH Interfaces 13
VETH Interfaces, illustrating 13, 14
Virtual eXtensible LAN (VXLAN) 309
Virtual Interface 10
Virtual Interface, components
  Bonded 13
  Bridge 13
  VETH 13
  VLAN 13
Virtual Interface, concepts
  Bridge Network Interface 15
  VETH Interfaces 13
  VLANs 16
Virtual Interface, configuring 11
Virtual Networking 308
Virtual Networking, configuring 308
Virtual Private Networks (VPNs) 280
Virtual Router Redundancy Protocol (VRRP) 166
VLANs 16, 210, 211
VLANs, configuring 211, 212
VLANs, illustrating 16-18
VPNs, architecture 281
VPNs, configuring 282, 283
VPNs, fundamentals
  Privacy/Anonymity 281
  Remote Networks 281
  Security 281
VPNs, protocols
  IPsec 284
  OpenVPN 284
  WireGuard 284
VPNs, steps 282
VRF 51
VRF, configuring 51
VRF, ensuring 52
VRF/VLAN, comparing 52
VRRP, configuring 166, 167
VXLAN, configuring 309-311

W

WireGuard 285

WireGuard, architecture 285
WireGuard, configuring 286, 287
Wireshark 120
Wireshark, executing 120, 121
Wireshark Packets, filtering 121, 122

X

XDP Program 350
XDP Program, integrating 350-353