

Network Computing - Notes - v0.8.0

260236

February 2026

Preface

Every theory section in these notes has been taken from the sources:

- Course slides. [4]

About:

 [GitHub repository](#)



These notes are an unofficial resource and shouldn't replace the course material or any other book on network computing. It is not made for commercial purposes. I've made the following notes to help me improve my knowledge and maybe it can be helpful for everyone.

As I have highlighted, a student should choose the teacher's material or a book on the topic. These notes can only be a helpful material.

During the Network Computing for HPC course, I've created a project:

 [TCP Connection Tracker \(eBPF/XDP\)](#)



Contents

1 Datacenters	6
1.1 What is a Datacenter?	6
1.2 Datacenter Applications	11
1.3 Network Architecture	15
1.4 High and Full-Bisection Bandwidth	19
1.5 Fat-Tree Network Architecture	22
2 Software Defined Networking (SDN)	27
2.1 Introduction	27
2.2 Legacy Router & Switch Architecture	28
2.3 SDN Architecture	30
2.4 OpenFlow	32
2.5 OpenFlow limitations	36
3 Programmable Switches	39
3.1 Introduction	39
3.2 Why didn't programmable switches exist before?	41
3.3 Data Plane Programming and P4	43
3.4 PISA and Compiler Pipeline Mapping	46
4 Data Structures	48
4.1 Introduction	48
4.2 Ternary Content Addressable Memory (TCAM)	49
4.3 Deterministic Lookup with Probabilistic Performance	52
4.4 Probabilistic Data Structures	55
4.4.1 1-Hash Bloom Filters	55
4.4.2 Bloom Filters	57
4.4.3 Dimensioning a Bloom Filter	59
4.4.4 Counting Bloom Filters	60
4.4.5 Invertible Bloom Lookup Tables (IBLTs)	61
4.4.6 Count-Min Sketch	65
5 Datacenter Monitoring	67
5.1 Why Datacenter Monitoring Matters	67
5.2 Network Monitoring	69
5.3 Everflow	71
5.3.1 What is Everflow?	71
5.3.2 How it works	72
5.4 FlowRadar	75
5.4.1 Architecture	75
5.4.2 Data Structure used in FlowRadar	76
5.4.3 Collector Decode	78
5.5 In-Band Network Telemetry (INT)	82
5.5.1 What is INT?	82
5.5.2 Modes	83

6 Datacenter Layer 3 Load Balancing	85
6.1 Recap: Datacenters	85
6.2 Introduction	87
6.3 Packet Spraying	89
6.4 Equal Cost Multi Path (ECMP)	91
6.5 Hedera: Dynamic Flow Scheduling	97
6.6 HULA: Load Balancing in P4	101
7 Datacenter Layer 4 Load Balancing	104
7.1 Introduction	104
7.2 Traditional LB Architecture	106
7.3 Real-World Deployments	108
7.4 Design Space	114
7.5 Cheetah (Research Proposal)	121
7.6 Faild (Production Environment)	135
7.6.1 Design Goals and Choices	137
7.6.2 Faild vs Research Proposals	140
7.7 Summary	141
8 End-Host Networking	144
8.1 Why End-Host Networking Matters	144
8.2 Life of a Packet Inside a Server	148
8.3 The Receive Livelock Problem	154
8.4 Interrupt Mitigation Strategies	156
8.4.1 Interrupt Coalescing	156
8.4.2 Polling	158
8.4.3 NAPI (New API)	160
8.5 Multi-Queue NICs	163
8.6 Receive-Side Scaling (RSS)	165
8.7 Advanced Receive Flow Steering (aRFS)	168
8.8 Data Direct I/O (DDIO)	170
8.9 Standard Offloads	172
8.10 PCIe	174
8.11 Compute Express Link (CXL)	179
8.12 NIC Driver	181
8.13 The <code>sk_buff</code> (Socket Buffer)	184
8.14 The Linux Network Stack	186
8.14.1 Generic Receive Offload (GRO)	186
8.14.2 Netfilter	188
8.14.3 TCP/IP Stack	189
8.15 Kernel Bypass	191
8.15.1 Data Plane Development Kit (DPDK)	193
8.15.2 Remote Direct Memory Access (RDMA)	195
9 Laboratories	200
9.1 Introduction to P4 Programming	200
9.1.1 P4 ecosystem and motivation	201
9.1.2 P4 Architecture	203
9.1.3 Control Plane Interaction (<code>P4Runtime</code>)	205
9.1.4 Exercise 1: Packet Reflector	206

9.1.4.1	Build <code>network_topo.py</code>	210
9.1.4.2	Build <code>packet_reflector.p4</code>	213
9.1.4.3	Test P4 program	217
9.1.5	Exercise 2: Packet Repeater	218
9.1.5.1	Build <code>network_topo.py</code>	219
9.1.5.2	Build <code>control_plane.py</code>	221
9.1.5.3	Build <code>packet_repeater.p4</code>	223
9.1.6	Exercise 3: VLAN Handler	227
9.1.6.1	Build <code>network_topo.py</code>	230
9.1.6.2	Build <code>control_plane.py</code>	231
9.1.6.3	Build <code>vlan_handler.p4</code>	233
9.2	Stateful Packet Processing in P4	239
9.2.1	Introduction to Stateful P4 Programs	239
9.2.2	Registers in P4	241
9.2.3	Exercise 1: Heavy Hitter Detector v1	244
9.2.3.1	Build <code>network_topo.py</code>	246
9.2.3.2	Build <code>control_plane.py</code>	247
9.2.3.3	Build <code>hdd_v1.p4</code>	248
9.2.4	Exercise 2: Heavy hitter detector v2	254
9.3	Introduction to eBPF and XDP Programming	263
9.3.1	Why eBPF in Network Computing	263
9.3.2	What is eBPF?	265
9.3.3	eBPF Program Constraints	269
9.3.4	eBPF Core Building Blocks	272
9.3.5	XDP: eXpress Data Path	276
9.3.6	XDP Execution Context	278
9.3.7	XDP Return Codes	280
9.3.8	Tooling: <code>libbpf</code> & <code>bptool</code>	282
9.3.9	Exercise 1: The first eBPF program	286
9.3.10	Exercise 2: Counting with BPF Maps	293
9.3.11	Exercise 3: Packet Parsing	297
9.3.12	Exercise 4: Packet Rewriting	303

1 Datacenters

1.1 What is a Datacenter?

A **Datacenter** is a specialized facility that houses multiple computing resources, including servers, networking equipment, and storage systems. These resources are co-located (placed together in the same physical location) to ensure efficient operations, leverage shared environmental controls (such as cooling and power), and maintain physical security.

So the main characteristics are:

- **Centralized Infrastructure:** Unlike traditional computing models where resources are scattered, datacenters consolidate thousands to millions of machines in a single administrative domain.
- **Full Control over Network and Endpoints:** Datacenters operate under a single administrative entity, allowing customized configurations beyond conventional network standards.
- **Traffic Management:** Unlike the open Internet, datacenter traffic is highly structured, and the organization can define routing, congestion control, and network security policies.

Feature	Datacenter Networks	Traditional Networks
Ownership	Fully controlled by a single organization	Usually spans multiple independent ISPs
Traffic	High-speed internal communication (east-west traffic)	Lower-speed, external client-based traffic (north-south)
Routing	Customizable (non standard protocols)	Uses standard internet protocols (BGP, OSPF, etc.)
Latency	Optimized for ultra-low latency	Variable latency, dependent on ISPs
Redundancy	High redundancy to ensure failover and fault tolerance	Often limited by ISP policies

Table 1: Difference between Datacenters and other networks (e.g., LANs).

❸ Why are datacenters important?

Datacenters are the backbone of modern cloud computing, large-scale data processing, and AI/ML workloads. They provide high computational power and storage for various applications, such as:

1. **Web Search & Content Delivery.** For example, when a user searches for “Albert Einstein” on Google, the request is processed in a datacenter where:

- (a) The query is parsed and sent to multiple servers.
 - (b) Indexed data is retrieved.
 - (c) A ranked list of results is generated and sent back to the user.
2. **Cloud Computing.** Services like Amazon Web Services (AWS), Microsoft Azure, and Google Cloud offer computation, storage, and networking resources on-demand.
 - Infrastructure as a Service (IaaS): Virtual machines, storage, and networking.
 - Platform as a Service (PaaS): Databases, development tools, AI models.
 - Software as a Service (SaaS): Google Drive, Microsoft Office 365.
 3. **AI and Big Data Processing.** Large-scale computations like MapReduce and deep learning training rely on distributed datacenter resources.
 4. **Enterprise Applications.** Datacenters host internal IT infrastructure for businesses, including databases, ERP systems, and virtual desktops.

⌚ Evolution of Datacenters

While the concept of centralized computing dates back to the 1960s, the modern datacenter model emerged with cloud computing in the 2000s. Notable developments include:

- 1970s: IBM mainframes operated in controlled environments similar to early datacenters.
- 1990s: Rise of client-server computing required dedicated server rooms.
- 2000s-Present: Hyperscale datacenters by Google, Microsoft, and Amazon revolutionized networking, storage, and scalability.

🌐 What's new in Datacenters?

Datacenters have been around for decades, but modern datacenters have undergone significant changes in scale, architecture, and service models. The primary factors driving these changes include:

- ✓ The exponential growth of internet services (Google, Facebook, Amazon, etc.).
- ✓ The shift to cloud computing and on-demand services.
- ✓ The need for better network scalability, fault tolerance, and efficiency.

One of the most striking changes in modern data centers is their massive scale:

- Companies like Google, Microsoft, Amazon, and Facebook operate **datacenters with over a million servers at a single site**.
- **Microsoft alone has more than 100,000 switches and routers** in some of its datacenters.
- **Google processes billions of queries per day**, requiring vast computational resources.
- **Facebook and Instagram serve billions of active users**, with every interaction generating requests to datacenters.

Another major change is the **shift from owning dedicated computing infrastructure to renting scalable cloud resources**. Datacenters no longer just host enterprise applications, **they now offer computing, storage, and network infrastructure as a service**. The most common cloud computing models are:

- **Infrastructure as a Service (IaaS)**. User rent virtual machines (VMs), storage, and networking instead of maintaining their own physical servers (e.g., Amazon EC2).
- **Platform as a Service (PaaS)**. Provides a platform with pre-configured environments for software development (databases, frameworks, etc.).
- **Software as a Service (SaaS)**. Full software applications hosted in datacenters and delivered via the internet (e.g., Google Drive).

The move to cloud computing has fundamentally changed datacenters, shifting the focus to resource allocation, security, and performance guarantees. They are also moving from multi-tenancy to single-tenancy:

- **Single-Tenancy**. A client gets **dedicated infrastructure** for their services.
- **Multi-Tenancy**. **Resources are shared among multiple clients while ensuring isolation**.

✖ **Implications**. But this massive scale brings new challenges:

- **Scalability**: The need for **efficient network designs** to handle rapid growth.

Traditional datacenter topologies, such as three-based architectures, are inefficient at scale. New designs, like **Clos-based networks (Fat Tree)** and **Jellyfish (random graphs)**, are being developed to:

- ✓ Ensure **high bisection bandwidth** (allow any-to-any communication efficiently).
- ✓ Provide **scalable and fault-tolerant networking**.

- **Cost management:** More machines mean **higher power, cooling, and hardware costs.**

Datacenters are **expensive to build and maintain**, requiring:

- **Efficient resource utilization** (prevent idle servers from wasting power).
- **Energy-efficient cooling solutions** (cooling accounts for a *huge* portion of operational costs).
- **Automation to reduce human intervention** (e.g., AI-based network optimization).

- **Reliability:** Hardware failures become **common at scale**, requiring **automated fault-tolerant solutions.**

At the scale of modern datacenters, **hardware and software failures are common**. A key principle is: “*In large-scale systems, failures are the norm rather than the exception.*” (Microsoft, ACM SIGCOMM 2015).

Thus, new **automated failover mechanisms** are required to:

- Detect failures **quickly**.
- Redirect traffic **seamlessly**.
- Ensure **minimal service disruption**.

- **Performance & Isolation Guarantees:** In modern datacenters, **customers expect strict performance guarantees** for applications like: low-latency financial transactions, high-bandwidth video streaming, machine learning model training.

To meet these demands, datacenters implement:

- ✓ **Performance Guarantees:** Allocating bandwidth and compute power dynamically.
- ✓ **Isolation Guarantees:** Ensuring one user’s workload does not interfere with another’s.

But this requires **advanced networking techniques**, such as:

- **Traffic engineering** to avoid congestion.
- **Load balancing** to distribute workloads efficiently.
- **Software-defined networking (SDN)** for centralized control over traffic flows.

Key Takeaways: What is a Datacenter?

- **Datacenters centralize** computing resources for performance, security, and scalability.
- **They differ from traditional networks** by offering more control, lower latency, and higher redundancy.
- **Applications include cloud services, AI, and enterprise computing.**
- **Scalability is a key challenge**, with hyperscale datacenters hosting millions of machines.
- **Efficiency and cost containment are major concerns**, requiring innovative architectures.

1.2 Datacenter Applications

Modern datacenters host a variety of applications that range from web services to large-scale data processing. These **applications can be classified based on their traffic patterns and computational needs.**

② Customer-Facing Applications (North-South Traffic)

Customer-facing applications involve direct interaction with users. This type of traffic follows a **North-South communication model**, meaning that **data flows between external users and the datacenter.**

Example 1: North-South Traffic

Examples include:

- **Web Search** (e.g., Google, Bing)
 - A user submits a query (e.g., “Albert Einstein”).
 - The request is routed through the datacenter’s frontend servers.
 - Backend database and indexing servers fetch relevant results.
 - The response is assembled and sent back to the user.
- **Social Media Platforms** (e.g., Facebook, Instagram, X (ex Twitter))
 - Users interact with content hosted in the datacenter (e.g., loading a feed, liking posts).
 - Each interaction requires queries to databases and caching systems.
 - Content delivery is optimized using load balancers.
- **Cloud Services** (e.g., Google Drive, Dropbox, OneDrive)
 - Users upload, store, and retrieve files.
 - Requests must be efficiently distributed across storage nodes.

③ Large-Scale Computation (East-West Traffic)

Unlike customer-facing applications, backend computations do not involve direct interaction with external users. Instead, they focus on **processing massive datasets within the datacenter**. This type of traffic is known as **East-West traffic** because it occurs **between servers inside the datacenter rather than between the datacenter and the external world.**

Example 2: East-West Traffic

Examples include:

- **Big Data Processing** (e.g., MapReduce, Hadoop, Spark)
 - Large datasets are distributed across multiple servers.
 - Each server processes a portion of the data in parallel.
 - Results are combined to generate insights (e.g., web indexing, analytics).
- **Machine Learning & AI Training** (e.g., Deep Learning Models)
 - AI models are trained on massive datasets using clusters of GPUs/TPUs.
 - The process requires high-bandwidth, low-latency communication.
 - Synchronization between nodes is critical (e.g., gradient updates in distributed training).
- **Distributed Storage & Backup Systems** (e.g., Google File System, Amazon S3)
 - Data is replicated across multiple locations for reliability.
 - Servers frequently exchange data to ensure consistency and fault tolerance.

Key differences between North-South and East-West traffic

Feature	N-S traffic	E-W traffic
Direction	External users ↔ Datacenter	Within datacenter
Examples	File downloads	AI training
Bandwidth Needs	Moderate	Very High
Latency Sensitivity	High	Critical
Traffic Type	Query-response	Bulk data transfer

Table 2: Differences between North-South and East-West traffic.

In terms of latency sensitivity, North-South traffic is high because user interactions must be fast. On the other hand, East-West traffic is critical because synchronization delays affect computation.

▣ Traffic Patterns and Their Impact on Networking

The way data moves within a datacenter heavily influences network design. The main goal is to ensure high bandwidth, low latency, and efficient resource utilization.

- **Any-to-Any Communication Model**

- In large-scale distributed applications, any server should be able to communicate with any other server at full bandwidth.
- Network congestion can severely degrade performance, especially for AI/ML workloads and big data processing.

- **High-Bandwidth Requirements**

- Applications like MapReduce and deep learning require high data transfer rates.
- If bandwidth is insufficient, bottlenecks occur, leading to delays.

- **Latency is a Critical Factor**

- Low-latency networking is essential for interactive applications and distributed computing.
- AI training, for example, requires nodes to synchronize frequently; a delay in one node slows down the entire process.

- **Worst-Case (Tail) Latency Matters**

- It's not enough for most requests to be fast; the slowest request can delay the entire computation.
- Minimizing tail latency is crucial for efficient AI model training and database queries.

▲ Challenges in Datacenter Traffic Management

The massive scale and complexity of modern datacenters introduce several networking challenges, including:

- **Network Congestion and Bottlenecks.** When multiple servers communicate simultaneously, some network links become overloaded, leading to congestion.

For example, if many AI training jobs share the same network path, it can become a bottleneck, slowing down training.

This can be a critical issue for applications requiring real-time performance (e.g., financial transactions, cloud gaming).

- **Load Balancing and Traffic Engineering.** How do we distribute traffic efficiently across network links? The solutions are: Equal-Cost Multipath Routing (ECMP, spreads traffic across multiple paths); Dynamic Traffic Engineering (adjusts paths in real time based on congestion levels).

- **Avoiding Link Over-Subscription.** If too many servers send data over a single link, the available **bandwidth is divided**, leading to **slow performance**. Modern datacenters aim for **full-bisection bandwidth**, meaning **any server can talk to any other server at full capacity**.
- **Scaling Challenges.** Traditional datacenter network architectures do not scale well beyond a certain point. **New network topologies** (e.g., Fat Tree, Jellyfish) are being adopted to address these limitations.

Key Takeaways: Datacenter Applications

- Datacenters handle **two major types of applications**:
 1. **Customer-facing applications (North-South traffic)** involve external users.
 2. **Large-scale computations (East-West traffic)** occur within the datacenter.
- **Traffic patterns affect bandwidth, latency, and congestion control.**
- **Managing congestion and ensuring high bandwidth** is critical for performance.
- **New network topologies and routing techniques** help address scaling challenges.

1.3 Network Architecture

The primary goal of a datacenter network is to **interconnect thousands to millions of servers** efficiently. Unlike traditional networks, which focus on wide-area communication, datacenter networks emphasize:

- **High throughput**: Supporting massive data transfers.
- **Low latency**: Ensuring real-time performance for applications.
- **Scalability**: Accommodating rapid growth without performance degradation.
- **Fault tolerance**: Handling hardware failures with minimal disruption.

Datacenter networks physically and logically connect servers through a **multi-tiered architecture**. This hierarchical structure ensures that servers in different racks, pods, or clusters can communicate efficiently.

Traditional Three-Tier Datacenter Network

Most datacenter networks follow a **Three-Tier design**, which is optimized for scalability and efficiency. The three tiers are:

- **Edge Layer (Access Layer)**
 - Located at the **bottom of the hierarchy**, closest to the servers.
 - Consists of **Top-of-Rack (ToR) switches** that connect servers within a rack.
 - ✓ **Purpose**: Aggregates traffic from multiple servers and forwards it to the higher layers.
 - Typically uses **high-speed links (10-100 Gbps per port)** to connect servers.
- **Aggregation Layer (Distribution Layer)**
 - Intermediate layer between the edge and core layers.
 - Connects **multiple ToR switches** within a datacenter pod.
 - ✓ **Purpose**: Helps distribute traffic efficiently **without overwhelming core routers**.
 - Implements **load balancing, redundancy, and failover mechanisms**.
- **Core Layer (Backbone Layer)**
 - The **top layer** of the hierarchy.
 - Composed of **high-capacity, high-speed switches and routers**.
 - ✓ **Purpose**: Responsible for:
 - * **Routing large volumes of traffic** between different aggregation switches.

- * **Connecting the datacenter to external networks** (e.g., the Internet or private backbones).
- Core switches often run at **100 Gbps or higher per port** to support high aggregate bandwidth.

Key characteristics of the Three-Tier model:

- **Position:**
 - **Edge Layer:** Closest to servers.
 - **Aggregation Layer:** Intermediate between edge and core.
 - **Core Layer:** Backbone layer.
- **Primary Function:**
 - **Edge Layer:** Connects servers within racks.
 - **Aggregation Layer:** Aggregates ToR traffic.
 - **Core Layer:** Routes traffic between datacenters or externally.
- **Switch Type:**
 - **Edge Layer:** Top-of-Rack (ToR).
 - **Aggregation Layer:** Aggregation switches.
 - **Core Layer:** Core routers.
- **Speed (per port):**
 - **Edge Layer:** 10-100 Gbps.
 - **Aggregation Layer:** 40-100 Gbps.
 - **Core Layer:** 100 and more Gbps.
- **Fault Tolerance:**
 - **Edge Layer:** Redundant paths to aggregation layer.
 - **Aggregation Layer:** Load balancing across core switches.
 - **Core Layer:** High redundancy & backup links.

A Limitations of the Traditional Three-Based Model

Although widely used, the traditional three-tier model faces **scalability and performance challenges** as datacenters grow.

- **Scalability Issues.** Traditional networks are hierarchical, meaning most communication must pass through the core layer. As datacenters scale, **core switches become bottlenecks** due to increased traffic.
 - **Bandwidth Bottlenecks.** The model assumes that the **most traffic** is North-South (client to server). However, modern workloads involve **high East-West traffic** (server-to-server communication).
- Over-subscription occurs** when the network cannot handle full-bisection bandwidth.

- **Over-Subscription Problem.** Over-Subscription refers to the ratio of worst-case achievable bandwidth to total bisection bandwidth. For example:

- If 40 servers per rack each have a 10 Gbps link, total demand is 400 Gbps.
- If the uplink capacity to the aggregation layer is only 80 Gbps, we have a 5:1 over-subscription.
- This means only 20% of the potential bandwidth is available, causing congestion.

Over-subscription ratios in large-scale networks can reach 50:1 or even 500:1, severely limiting performance.

- **Performance Issues in High-Density Environments.** High latency when traffic must traverse multiple hops to reach other racks. Failures in core routers can impact a large number of servers. Inconsistent network performance due to congestion in aggregation switches.

✓ Modern Datacenter Network Designs

To overcome the scalability and congestion challenges of traditional three-based networks, modern datacenters use alternative architectures.

- ✓ **Fat Tree (Clos Network).** Fat Tree is a multi-stage switching architecture designed to:

- Ensure full-bisection bandwidth: Every server can communicate at full capacity.
- Provide multiple paths between any two servers (high redundancy).
- Balance traffic dynamically to avoid congestion.

It uses K-ary fat tree topology where each pod consists of aggregation and edge switches, and core switches connect multiple pods. The advantages are:

- Scalability: Expands easily by adding more pods.
- Fault Tolerance: Multiple paths prevent failures from disrupting traffic.
- Better Load Balancing: Traffic is evenly distributed.

- ✓ **Jellyfish: Random Graph-Based Topology.** Instead of a strict hierarchical structure, Jellyfish uses a randomized topology. The advantages are:

- Higher network capacity with lower cost.
- More flexible scaling than Fat Tree.
- Better fault tolerance since the network adapts dynamically.

- ✓ **BCube: Datacenter Network for Cloud Computing.** Designed for high-performance cloud computing environments. It is optimized for: multi-path communication, resilience against failures and lowe latency compared to hierarchical models.

Key Takeaways: Network Architecture

- Traditional **three-tier datacenter networks** include **Edge, Aggregation, and Core layers**.
- **Core switches bottlenecks** as datacenters scale.
- **Over-subscription limits bandwidth**, causing congestion.
- Modern topologies like **Fat Tree and Jellyfish** improve **scalability, fault tolerance, and load balancing**.

1.4 High and Full-Bisection Bandwidth

❷ Why is High-Bandwidth important in Datacenters?

Modern datacenters handle **massive amounts of data** due to applications like AI training, cloud services, and big data processing. These workloads *require*:

- **High-bandwidth connections** to support fast data transfers.
- **Low latency** to ensure real-time performance.
- **Scalability** to accommodate increasing workloads.

Unlike traditional networks, where traffic primarily flows between users and servers (North-South), **datacenters experience heavy East-West traffic** (server-to-server communication). This shift **demands high-bandwidth and scalable network designs**.

❸ One step at a time: What a Bisection Bandwidth is and why Full-Bisection Bandwidth is important

Bisection Bandwidth is a key metric that measures the **total bandwidth available between two halves of a network**.

Definition 1: Bisection Bandwidth

If a network is split into two equal halves, the **Bisection Bandwidth** is the **total data transfer rate available between them**.

Definition 2: Full-Bisection Bandwidth

The **Full-Bisection Bandwidth** is when every server can communicate with every other server at **full network speed**.

In other words, bisection bandwidth can be thought of as cutting a data center network in half and measuring the total capacity of the links connecting the two halves. This tells us how much data can flow between the two sections simultaneously.

Example 3: Understand what bisection bandwidth is

Imagine a 1000-server datacenter, where 500 servers are processing data while 500 servers store the results. If the bisection bandwidth is **low**, the **data transfer between processing and storage nodes will be delayed**. This results in slow machine learning model training or delayed database queries.

As we can imagine, the full-bisection bandwidth is a real and critical aspect:

- **Prevents bottlenecks:** Ensures high-throughput communication across racks and clusters.
- **Essential for AI/ML training:** AI models require massive parallel computations with continuous data exchanges.

- **Optimized for cloud computing:** Services like AWS, Google Cloud, and Azure depend on fast, reliable inter-server communication.

A Then try to get high-bandwidth all the time! Yes, but there are some challenges...

Ideally, high-bandwidth should be the ultimate goal, but unfortunately, there are some problems with traditional three-based networks:

X The Problem with Traditional Three-Based Networks. The standard **three-tier (core-aggregation-edge) topology** struggles to scale due to:

1. **Over-subscription** (definition on page 17): The ratio of available bandwidth to required bandwidth is too high.
2. **Core congestion:** Core routers become bottlenecks as traffic grows.
3. **Single points of failure:** A failure in a core switch can affect a large portion of the datacenter.

X Over-Subscription and Its Impact on Network Performance. A naive solution would be to use over-subscription to solve these problems, but this limits performance. **Over-Subscription** happens when the **network is provisioned with less bandwidth than needed** to cut costs.

$$\text{Over-subscription} = \frac{\text{Total server bandwidth demand}}{\text{Available bandwidth at aggregation/core layer}}$$

Common over-subscription ratios are:

- 5:1, only 20% of host bandwidth is available.
- 50:1, only 2% of host bandwidth is available.
- 500:1, only 0.5% of host bandwidth is available.

At 500:1 over subscription, congestion becomes severe, **limiting network efficiency**.

X The cost problem: scaling is expensive!

- Increasing bisection bandwidth requires **more high-performance network hardware**.
- **Scaling traditional networks** (adding more core switches) is extremely costly.
- **Energy consumption rises** with additional hardware.

Thus, **alternative solutions** are needed to achieve high-bandwidth networking **without excessive costs**.

✓ Solutions to Achieve High and Full-Bisection Bandwidth

To overcome these challenges, researchers and engineers have designed **new network architectures**.

- ✓ **Fat Tree (Clos Network) - The Scalable Solution.** Unlike traditional three-based designs, Fat Tree provides **multiple paths** for traffic.

✓ Advantages

- ✓ **Ensure full-bisection bandwidth** by allowing traffic to take alternative routes.
- ✓ **Eliminates single points of failure** using redundant paths.
- ✓ **Load balancing** optimizes network utilization.

- ✓ **Jellyfish - A More Flexible Approach.** Uses a **randomized, non-hierarchical** topology instead of a fixed three structure.

✓ Advantages

- ✓ **Better bandwidth scaling** as new servers are added.
- ✓ **More resilient to failures** (no single critical point of failure).

- ✓ **BCube - Optimized for Cloud Services.** Designed for high-performance cloud environments with **massive inter-server communication**.

✓ Advantages

- ✓ **Fast re-routing** in case of failures.
- ✓ **Low-latency communication for cloud applications.**

Key Takeaways: High and Full-Bisection Bandwidth

- **High-bandwidth networking** is essential for modern datacenters.
- **Full-bisection bandwidth** ensures servers communicate at **full speed**.
- **Over-subscription** creates **bottlenecks**, limiting performance.
- **New network architectures** (Fat Tree, Jellyfish, BCube) solve scalability issues.

1.5 Fat-Tree Network Architecture

A **Fat-Tree** is a **multi-layer, hierarchical network topology** that provides *high scalability, full-bisection bandwidth, and fault tolerance*. It is a **special type of Clos Network**¹, designed to **overcome bandwidth bottlenecks** in traditional three-based networks.

The key idea is: Instead of a traditional tree where higher levels become bottlenecks, Fat-Tree ensures equal bandwidth at every layer by **increasing the number of links as we move higher in the hierarchy**.

❖ Structure of a K-Ary Fat-Tree

A **K-ary Fat-Tree** consists of **three layers**:

1. **Edge Layer (Top-of-Rack, ToR switches):**

- Connects directly to the servers.
- Each edge switch connects $\frac{k}{2}$ servers and $\frac{k}{2}$ aggregation switches.

2. **Aggregation Layer**

- Connects multiple edge switches.
- Ensures **local traffic routing** between racks before sending to the core.
- Each aggregation switch connects $\frac{k}{2}$ edge switches and $\frac{k}{2}$ core switches.

3. **Core Layer**

- The backbone of the Fat-Tree, interconnecting multiple aggregation layers.
- Consists of $(\frac{k}{2})^2$ core switches, where each connects to k pods.

Example 4: Fat-Tree with $k = 4$

- Each pod contains:
 - $(\frac{4}{2})^2 = 4$ servers.
 - 2 layers of 2 2-port switches (Edge and Aggregation).
- Each Edge Switch connects 2 servers and 2 aggregation switches.
- Each Aggregation Switch connects 2 Edge switches and 2 Core switches.
- The Core Layer consists of $(\frac{4}{2})^2 = 4$ core switches.

¹A **Clos Network** is a type of multistage switching topology that enables high-bandwidth and fault-tolerant communication by interconnecting multiple small switches instead of relying on a few large ones. It is commonly used in datacenter networks (e.g., Google Jupiter Fabric) to maximize scalability and minimize congestion.

As a result, multiple paths between servers ensure no single point of failure and full-bisection bandwidth.

✓ Why Use Fat-Tree in Datacenters?

✓ Cost-Effective Scaling

- Can be built using **cheap, commodity switches** instead of expensive core routers.
- All switches operate at **uniform capacity**, simplifying hardware requirements.

✓ Full-Bisection Bandwidth

- Each switch and server has **equal access to bandwidth**, preventing bottlenecks.
- Every packet has **multiple available paths**, ensuring **load balancing**.

✓ High Fault Tolerance

- If one **switch or link fails**, traffic is rerouted through **alternative paths**.
- **No single point of failure**, unlike traditional three-based architectures.

✓ Efficient Load Balancing

- **Multipath Routing** ensures traffic is evenly distributed.
- **No congestion at higher layers**, as each pod has equal bandwidth allocation.

✗ Problems in Fat-Tree Networks

Fat-Tree is a highly scalable and efficient network topology, but **practical challenges exist** when handling real-world workloads.

- **Many flows running simultaneously**. In large datacenters, multiple applications generate concurrent flows. Some flows are **small but latency-sensitive** (mice flows), while others are **large data transfers** (elephant flows). The Fat-Tree must **efficiently balance all these flows** across available paths.
- **Traffic locality is unpredictable**. Some services (e.g., Facebook/Meta workloads) have localized communication within a rack, while others require data exchange across the entire network. Fat-Tree must **dynamically adapt to different workload patterns**.

- **Traffic is bursty.** Some applications generate sudden traffic spikes, leading to temporary congestion. This is problematic for routing since **congestion-aware path selection is difficult**.
- **Too Many Paths Between a Source and Destination.** Unlike traditional network that have a single best route, Fat-Tree networks offer multiple equal-cost paths. *Which path should be used?* Random selection might lead to congestion.
- **Random Path Selection Leads to Collisions.** If routing randomly assigns traffic flows, two large elephant flows may end up on the same link. This creates a congestion hotspot, even though other links remain underutilized.
 - **Ideal case:** Traffic should be spread evenly across all available links.
 - **Reality:** Without congestion awareness, routing **cannot react to traffic conditions dynamically**.
- **Short-Lived vs. Long-Lived Flows Create Conflicts.** An ideal **routing scenario** would be to evenly distribute all flows. However, if a short, latency-sensitive flow suddenly appears on a congested link, its performance suffers. The key problem is that **Fat-Tree does not inherently prioritize latency-sensitive flows**.

⚠ TCP Incast: A Major Issue in Fat-Tree Datacenters

Large-scale parallel requests cause network congestion. In fact, some workloads (e.g., distributed storage systems, AI training) involve a **single client requesting data from multiple servers simultaneously**. This means that all servers respond at once, **overwhelming the switch's buffer capacity**. This results in **packet loss and retransmissions**, significantly increasing latency.

Definition 3: TCP Incast

TCP Incast is a **network congestion issue** that occurs in datacenters when multiple servers send data to a single receiver simultaneously, overwhelming the switch's buffer capacity and causing severe packet loss and performance degradation.

In other words, TCP Incast happens when many-to-one communication causes network congestion, leading to packet loss, TCP retransmissions, and increased latency.

But in this scenario, how does **TCP Incast** happen?

1. A client application requests data from multiple storage servers.
2. All storage servers respond **simultaneously**.
3. The switch **cannot handle all packets at once**, causing **buffer overflow**.

4. **Packet loss triggers TCP retransmissions**, further slowing down performance.

This involves several issues:

- Causes **severe latency spikes**, affecting (AI training and large-scale cloud) workloads.
- Traditional TCP was not designed for this kind of bursty traffic.
- **Fat-Tree cannot solve this issue alone**, it requires transport-layer optimizations.

Google's Approach to Solving Fat-Tree Challenges

Google faced severe scalability, congestion, and failure recovery challenges in its datacenters. Instead of using a traditional Fat-Tree model, they **developed a Clos-based architecture** known as **Google Jupiter Fabric**. The key challenges that Google is addressing are:

- **Scalability**. Traditional networks could not handle Google's exponential growth. Needed a network that scales gracefully by adding more capacity in stages.
- **Failure Tolerance**. A single failure should not impact traffic significantly. Needed path redundancy to ensure seamless operations.
- **Performance and Cost**. High-performance custom-built switching to support full-bisection bandwidth. Used commodity merchant silicon (off-the-shelf networking chips) instead of proprietary network devices, reducing costs.

The solutions adopted by Google are:

- ✓ **Clos Topology for Scalability & Fault Tolerance**. Google moved from traditional Fat-Tree to Clos networks to improve scalability.
 - **Multiple layers of switches**, with multiple paths between every two endpoints.
 - **Graceful fault recovery**: if one switch fails, traffic is rerouted dynamically.
 - **Incremental scalability**: new switching stages can be added without network downtime.

A Clos network was chosen because, unlike Fat-Tree, which suffers from static oversubscription, **Clos networks offer more flexible bandwidth allocation**.

Note that Fat-Tree inherits the scalability and fault tolerance of Clos, but its hierarchical and structured nature leads to congestion, routing complexity, and TCP Incast problems. Google recognized that Fat-Tree had structural limitations, so they modified Clos into the Jupiter Fabric.

✓ **Custom Hardware: Merchant Silicon Instead of Proprietary Switches.** Google avoided vendor lock-in by using commodity hardware (merchant silicon). The reasons are:

- Lower cost than custom ASIC-based routers.
- Faster hardware upgrade cycles.
- More control over network design and software stack.

✓ **Centralized Control for Routing and Network Management.** In traditional datacenters, routing is distributed, meaning each switch makes independent routing decisions. This approach does not scale well in Clos networks with thousands of switches.

The solution is **precomputed routing decisions**. Instead of switches making their own decisions, Google precomputes traffic flows centrally and pushes them to switches.

✓ Advantages

- ✓ **Improves traffic engineering:** Load balancing decisions are optimized globally rather than per switch.
- ✓ **More predictable performance.**
- ✓ **Less congestion:** Can react dynamically to network failures.

Key Takeaways: Fat-Tree Network Architecture

- **Fat-Tree is a special type of Clos Network** that overcomes bottlenecks in traditional tree networks.
- **K-ary Fat-Tree** has three layers (Edge, Aggregation, Core), ensuring equal bandwidth for all nodes.
- **Fat-Tree** provides multiple paths, but **routing is difficult due to unpredictable traffic patterns**.
- **Collisions between large flows create network hotspots.**
- **TCP Incast is a major issue**, where too many responses at once cause packet loss.
- Google's Datacenter Network Strategy:
 - Moved from **Fat-Tree to Clos topology** for better scalability and failure recovery.
 - Used **merchant silicon instead of proprietary hardware** to cut costs and improve flexibility.
 - Implemented **centralized control for routing** to optimize traffic flows.
 - Designed the **Jupiter Fabric** to handle **Google-scale workloads** with incremental scalability.

2 Software Defined Networking (SDN)

2.1 Introduction

Software-Defined Networking (SDN) is an **architectural shift** in networking that **separates** the **control plane** from the **data plane**, allowing for centralized control and programmability. Unlike traditional networks, where control logic is embedded in individual devices, **SDN introduces a centralized software controller that dynamically manages the entire network**.

The **importance of SDN** lies in its ability to:

- Improve **network flexibility** by enabling real-time changes.
- Simplify **network management** through automation.
- Reduce **hardware dependency** by allowing software-driven policies.
- Support **rapid innovation**, making networks more adaptable.

⚠ Traditional Networking vs. SDN

In *traditional networking*, routers and switches **contain both**:

- **Control Plane**, decides how traffic should be forwarded (e.g., routing decisions, firewall rules).
- **Data Plane**, physically forwards packets based on the control plane's decisions.

Challenges of traditional networking:

- a. **✗ Rigid Configuration**: Any changes require manual updates on multiple devices.
- b. **✗ Vendor Lock-in**: Hardware manufacturers impose proprietary limitations.
- c. **✗ Slow Innovation**: Implementing new networking features takes years due to hardware constraints.
- d. **✗ Complex Management**: Network engineers must configure each device individually.

How *SDN differs*:

- a. **✓ Control Plane is centralized** in an SDN controller.
- b. **✓ Data Plane remains distributed** across switches and routers.
- c. **✓ Network logic is programmable**, making updates and changes easier.

This shift makes networks more dynamic, scalable, and easier to manage.

2.2 Legacy Router & Switch Architecture

Legacy network devices, such as routers and switches, are **built with integrated control and data planes**, meaning **each device independently makes forwarding decisions**. These devices consist of:

1. Hardware Components

- **Application-Specific Integrated Circuits (ASICs)**, specialized chips for packet forwarding.
- **Memory (buffers & TCAMs)**, stores forwarding tables and processing queues.
- **Network Interfaces (NICs, Ports)**, physical ports for connecting network cables.

2. Software Components

- **Router OS (Operating System)**, runs network protocols and management interfaces.
- **Routing Protocols (OSPF, BGP, RIP)**, determines paths for packet forwarding.
- **Forwarding Table**, maps destination addresses to outgoing ports.

3. Management and Control Interfaces

- **Command-Line Interface (CLI)**, used for configuring routers manually.
- **SNMP (Simple Network Management Protocol)**, enables monitoring and automation.

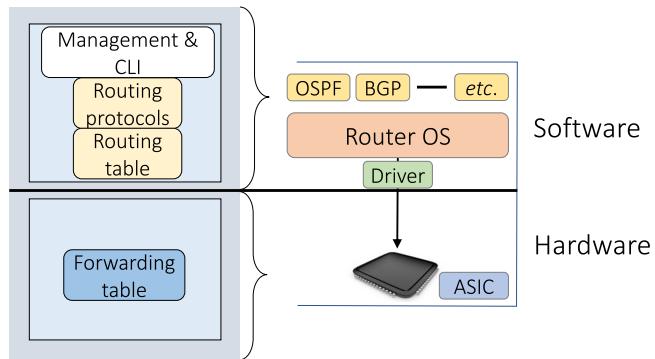


Figure 1: Legacy router and switch architecture.

Traditional network devices have two primary operational planes:

- **Control Plane**: makes forwarding decisions based on routing protocols.
- **Data Plane**: physically forwards packets based on control plane decisions. For example MAC lookup and IP forwarding.

Each router operates autonomously, using routing tables built through protocols like OSPF and BGP. These protocols dynamically learn network paths and update the forwarding tables, ensuring efficient packet delivery.

Packet Processing in a Legacy Router

1. **Lookup Destination IP** → Find matching entry in the forwarding table.
2. **Update Header** → Modify packet headers if needed (e.g., TTL decrement).
3. **Queue Packet** → Send packet to the appropriate output interface.

✖ Since every device handles its own control and forwarding, large-scale changes require individual device updates, making traditional networking complex and inflexible.

Challenges in Traditional Network Management

- ✖ **Complex Configuration & Management.** Each network device has to be configured individually. Protocols like BGP and OSPF require manual tuning for optimal performance. Network engineers must interact with vendor-specific CLIs, which vary by manufacturer.
- ✖ **Limited Innovation & Vendor Lock-In.** New network features require firmware or software updates from vendors. Custom networking solutions are difficult to implement due to proprietary hardware and software.
- ✖ **Slow Response to Failures & Traffic Changes.** Routing adjustments depend on distributed algorithms that can take seconds to minutes to converge. Manual troubleshooting is often needed when failures occur.
- ✖ **Scalability Issues.** Growing networks require more hardware and manual configurations. Updating policies across multiple routers is time-consuming and error-prone.

Key Takeaways: Legacy Router and switch architecture

- Legacy networking relies on autonomous devices with tightly integrated control and data planes.
- Routing is handled by protocols like OSPF and BGP, which operate independently on each device.
- Challenges include manual configuration, vendor lock-in, slow failure response, and scalability issues.
- These limitations paved the way for SDN, which offers centralized, programmable networking.

2.3 SDN Architecture

The core concept of **Software-Defined Networking (SDN)** is the separation of the control plane from the data plane:

- In SDN, **network devices** (switches/routers) become **simple forwarding elements**, executing decisions made by a centralized **controller**.
- The **SDN Controller** is a **software-based system** that **manages, programs, and monitors the entire network**.

Key architecture:

- **Data Plane (Forwarding Engine)** → Located on switches; handles packet forwarding.
- **Control Plane (SDN Controller)** → Runs on external servers; computes forwarding rules.
- **Communication Channel** → Allows the controller to instruct the data plane; typically uses OpenFlow.

But *why decouple?* Enables centralized decision-making, consistent policy enforcement, and simplified management. Facilitates dynamic updates to the network without hardware changes.

▀ The Role of the SDN Controller

The SDN Controller is the **central brain** of the network. It performs:

- ✓ **Network State Monitoring:** Gathers real-time information from all forwarding devices.
- ✓ **Decision-Making:** Calculates the best routes, applies policies, and enforces security.
- ✓ **Rule Installation:** Pushes flow rules to switches, determining how packets should be handled.

Controllers provide a **global, up-to-date view of the entire network**, enabling smarter control than traditional distributed routing.

◀▶ Communication Interfaces

SDN uses two types of APIs to manage communication between layers:

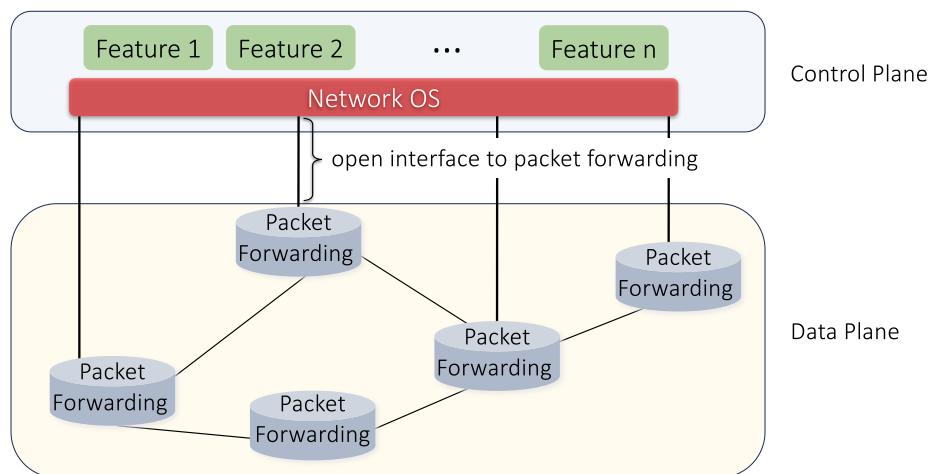
- **Southbound Interface:** Connects the controller to the data plane devices (e.g., **OpenFlow protocol**). It instructs switches via OpenFlow or similar protocols, installing/removing flow rules and collecting stats.
- **Northbound Interface:** Allows **applications to interact with the controller** via APIs (e.g., REST APIs). Applications (e.g., security monitoring, load balancing) query and command the controller to implement network policies.

Network Operating System (Network OS)

The controller runs a Network OS, providing:

- **Abstractions** over the physical network (e.g., topology view, link status).
- **Programmatic Interfaces** for developing control programs.
- **Consistency & Global View**: All decisions are made based on coherent, synchronized data.

The Network OS simplifies the task of writing network control logic by exposing standardized APIs.



Key Takeaways: SDN Architecture

- **Traditional networking** embeds the control plane within each device; SDN **centralizes control** in software.
- The **SDN Controller** dynamically manages the **data plane devices** using a **communication protocol**.
- **OpenFlow** is the primary protocol used to communicate between the controller and switches.
- **Network OS** provides an **abstraction layer** and **programming environment** for writing control logic.

2.4 OpenFlow

OpenFlow is the first and most widely adopted protocol used in Software-Defined Networking (SDN) to enable communication between the **SDN Controller** (control plane) and the **data plane devices** (e.g., switches, routers, they are the forwarding engine). It allows the controller to program flow tables in the switches and control how packets are forwarded, enabling centralized management of traffic.

In other words, OpenFlow is the practical implementation of SDN, standardizing how controllers manage packet forwarding.

❖ How OpenFlow works

Each **OpenFlow switch** contains:

1. **Flow Table**: Contains rules in the form of $Match \rightarrow Action$ pairs.
2. **Communication Interface**: Connects to the SDN controller via the OpenFlow protocol.
3. **Stats Module**: Collects statistics about packet flows.

Example 1: Flow Rules

1. If $Header = p \Rightarrow$ send to port 5.
2. If $Header = q \Rightarrow$ modify header to r , then send to ports 6 and 7.
3. If $Header = p \Rightarrow$ send packet to the controller.

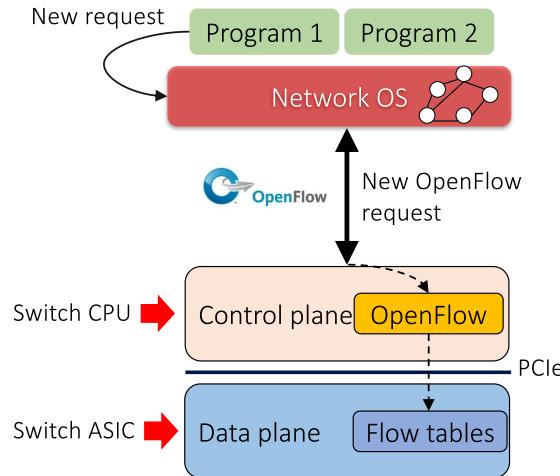
Flow table operation:

1. Packet arrives at the switch.
2. Switch checks for a **matching rule** in its flow table.
3. If matched \rightarrow **apply action** (e.g., forward, modify, drop).
4. If no match \rightarrow **send packet header to controller** for instructions.

Example 2: OpenFlow

1. New packet arrives at switch.
2. Match?
 - Yes \rightarrow forward according to rule.
 - No \rightarrow forward header to controller.
3. Controller analyzes packet and installs a new rule in switch.

4. Next packets of same type → directly processed by switch using newly installed rule.



Actions in OpenFlow

OpenFlow supports many types of actions, such as:

- **Forwarding** to one or multiple ports.
- **Dropping packets**.
- **Modifying headers** (e.g., VLAN tags, IP addresses).
- **Sending packets to controller**.
- **Statistics collection** for flow monitoring.

This **flexibility** allows SDN to implement advanced functions like load balancing, traffic shaping, and security filtering without specialized hardware.

Reactive vs. Proactive Flow Rules

In OpenFlow, the **controller installs flow rules** into switches to determine how packets are processed. There are **two main modes** of operation for rule installation: Reactive and Proactive. These **modes define when and how flow entries are populated in the flow tables** of switches.

- **Reactive Mode**

How it works?

1. When a **new flow** (a new type of packet) arrives at a switch, and **no rule matches it**, the **packet header is sent to the controller**.

2. The **controller analyzes the packet** and **decides what rule** should be installed in the switch to handle it.
3. After the controller sends back a rule, the switch installs it and **forwards the packet** accordingly.

☰ Key Characteristics

- * **Efficient use of flow tables** - Only rules for **active flows** are installed.
- * **Every new flow** incurs **small setup time** (controller interaction delay).
- * **Switch depends on the controller** for flow rule installation.
- * If the **controller connection is lost**, the switch has **limited utility** for new flows.

✓ Advantages

- ✓ **Dynamic** and **adaptive** to real-time network traffic.
- ✓ **Minimizes unused flow entries**.

✗ Disadvantages

- ✗ Adds **latency** for the **first packet** of each flow.
- ✗ **High control plane load** in environments with many short flows.

• Proactive Mode

② How it works?

1. The **controller pre-installs flow rules** in the switches before any packet arrives.
2. The switch **immediately processes packets** using pre-defined rules without contacting the controller.

☰ Key Characteristics

- * **Zero setup delay** for packet processing - packets are forwarded **immediately**.
- * Requires **aggregated or wildcard rules** to efficiently use flow table space.
- * **Independent of controller connectivity** - continues to operate even if controller is unreachable.

✓ Advantages

- ✓ **Fast packet forwarding** with **no initial delay**.
- ✓ **No dependency** on controller for flow rule installation during packet arrival.
- ✓ Ideal for **predictable traffic patterns** or **mission-critical environments**.

✗ Disadvantages

- ✗ Can **waste flow table space** if many pre-installed rules are unused.
- ✗ Requires **good planning** of rules; less flexible to dynamic traffic changes.

In summary, reactive mode is adaptive, but introduces latency and higher controller load; proactive mode is fast and resilient, but requires advance planning of rules.

Key Takeaways

- **OpenFlow** enables the SDN controller to manage **flow tables** in switches.
- **Flow rules** define how packets are handled, allowing **centralized, programmable networking**.
- OpenFlow supports **fine-grained traffic control** via a wide range of **match/action rules**.
- Two operation modes:
 - Reactive (dynamic but with latency).
 - Proactive (fast but needs good planning).

2.5 OpenFlow limitations

OpenFlow, conceived around 2007, introduced centralized control by standardizing how switches expose forwarding behavior to an SDN controller (as we discussed in the previous section, page 32). The **insight** at that time was that **most switches perform similar tasks** (Ethernet switching, IPv4 routing, VLAN tagging, ACL enforcement) **all via fixed, predictable behaviors**.

OpenFlow capitalized on this fixed-function approach. Controllers could install flow rules into switches, dictating how they process known packet headers. However, a **critical limitation** emerged: **we couldn't add new protocols or processing capabilities easily**. Because **OpenFlow assumes a static data plane, hardcoded to process only a predefined set of protocols and headers**.

✓ Expanding OpenFlow: pushing its limits

As networking needs evolved, particularly in **virtualized environments** and **cloud datacenters**, operators needed more **specialized packet processing**. For example, **VXLAN**, used to identify tenants in multi-tenant environments, wasn't supported in early OpenFlow.

✓ To address this, vendors and the OpenFlow community developed **new versions** (1.1, 1.2, 1.3, ...). Each iteration **added support for more header types**, up to 50 different header types, but **the process was slow and cumbersome**. Each new feature needed:

- New OpenFlow specification extensions.
- New ASICs in hardware to support the processing logic.

⚠ Hardware Bottlenecks: The ASIC Development Bottleneck

Here lies the **core problem**: even with updated protocols, **switches couldn't adapt until vendors redesigned and shipped new ASICs** (Application-Specific Integrated Circuits).

This hardware dependency meant:

- ✗ New features took **years to reach production**.
- ✗ Network owners **couldn't simply get a software upgrade**.
- ✗ The result: **slow innovation** in data plane capabilities.

Example 3: VXLAN

Virtual Extensible LAN (VXLAN) was urgently needed by cloud providers and datacenters to enable multi-tenant network virtualization.

Despite this high demand, hardware vendors took ≈ 4 years to support VXLAN in switches due to ASIC development cycles and the fixed-function nature of OpenFlow switches.

Even though vendors delayed its release, once VXLAN was available, it became a standard requirement in data centers.

But attention! In the meantime, network operators used complex software overlays or kludges to simulate VXLAN functionality, increasing network complexity and cost.

👎 The Cost of Delay: Workarounds and Complexity

When vendors take years to deliver a new feature, network engineers often **develop complex workarounds, increasing network complexity and technical debt**. Even when the vendor releases the official feature:

- The workaround may already be deeply integrated.
- The official solution may no longer solve the problem.
- Worse, it may require a forklift upgrade, replacing hardware at high cost.

This inertia locks networks into **suboptimal solutions** and **impedes the agility promised by SDN**.

⌚ The Missing Ingredient: Programmability at the Data Plane

The shift from fixed-function to programmable data planes mirrors other computing domains:

Domain	Hardware	Compiler/SW Stack
General Computing	CPU	Java, C, OS Kernels
Graphics	GPU	OpenCL, CUDA
Signal Processing	DSP	Matlab Compiler
Machine Learning	TPU	TensorFlow Compiler
Networking	PISA Switch	P4 Language, P4 Compiler

Just as CPUs became programmable via compilers, **networking needs flexible data planes programmable via languages** like P4, running on PISA (Protocol Independent Switch Architecture).

Key Takeaways: OpenFlow limitations

- OpenFlow was a **revolution in control plane innovation**, but its **rigid data plane** became a bottleneck. The industry's response, iterative protocol updates and ASIC redesigns, proved **slow and reactive**.
- A true solution lies in programmable data planes, where **software defines packet processing**, and the network evolves **as fast as the application demands**.
- This transition is **not trivial**, it requires new hardware, new abstractions, and operator retraining, but it's essential to **fulfill SDN's promise of rapid, flexible, and scalable networking**.

3 Programmable Switches

3.1 Introduction

In the past, **network switches were designed with fixed-function pipelines**. These switches could process packets extremely fast, but their internal logic was essentially “hardcoded” by hardware vendors. This meant that the functionality they provided, things like Ethernet switching, IP routing, and basic ACLs, was rigid and **difficult to extend or modify**.

However, as networks evolved and application demands grew more complex, the limitations of these fixed-function switches became apparent. There was a **growing need for flexibility at the data plane**, the part of the switch responsible for real-time packet processing. Network operators started to ask: *what if we could program the switch behavior instead of relying on vendors to update the hardware every time we needed new features?* This is where the concept of **programmable switches comes into play**.

💡 Why Programmability?

The **motivation** behind programmable switches stems from the **increasing complexity and dynamism of modern networks**. Today’s infrastructures must support custom protocols for emerging technologies like IoT, 5G, and machine learning. They must also be able to adapt quickly to changing requirements, detect and mitigate threats in real-time, and perform network telemetry and monitoring with high granularity.

With **traditional switches**, making such changes **often meant waiting months** (or even years) for new hardware to be designed and released. In contrast, **programmable switches allow network behavior to be redefined using software**, even after deployment. This ability to program the forwarding logic gives networks a software-like agility that was previously unthinkable at the data plane level.

⚠️ Control Plane vs Data Plane

To understand the significance of programmable switches, it’s useful to recall the basic architecture of a network device. Typically, a **switch is divided into two major components**:

- The **Control Plane**, which is **responsible for**:
 - Computing routing tables;
 - Handling management tasks;
 - Making decisions about where traffic should go.
- The **Data Plane**, which is **responsible for**:
 - **Forwarding packets** at line rate, based on the decisions made by the control plane.

Traditionally, most of the innovation in networking happened in the control plane, for example, with Software-Defined Networking (SDN), which centralized and virtualized control logic (section 2, page 27). But the data plane remained fixed and closed.

Programmable switches shift this dynamic. They open up the data plane to innovation, **allowing developers to express forwarding behavior in a high-level language such as P4**. This means we can now rethink how packets are processed inside the switch itself.

■ The Rise of PISA

A key enabler of this shift is the **Protocol-Independent Switch Architecture (PISA)**. Proposed by Barefoot Networks (later acquired by Intel), **PISA** is a flexible hardware architecture that allows the structure of the switch pipeline to be configured by software. Using PISA, one can define new packet formats, parsing rules, match-action logic, and even custom metadata fields, all **using a high-level language like P4**.

With PISA-based switches, it is no longer necessary to hardcode support for every protocol in silicon. Instead, **developers can define how packets are handled at runtime**. This brings about a level of protocol independence and reconfigurability that was previously reserved for general-purpose processors, but with the performance and parallelism needed to operate at terabit speeds.

3.2 Why didn't programmable switches exist before?

In short, **programmable switches didn't make sense before** because we lacked the technical feasibility and practical justification. But now, due to advances in chip design and network complexity, it's finally possible, and necessary, to build them.

\$ In the past: Programmability was too expensive

In the past, the trade-off between programmability and cost was too high:

1. **Performance was too low.** Programmable hardware, like FPGAs or general-purpose CPUs, was much slower than fixed-function ASICs.

- ✓ A **fixed switch chip** could forward billions of packets per second.
- ✗ A **programmable one?** Too slow for line-rate performance.

So if we wanted programmability, we had to sacrifice speed. That was a deal-breaker for core network equipment.

2. **Chip area and power cost were too high.** Fixed-function logic is compact and power-efficient. Programmable logic, by contrast, used to **take up more silicon and required more power**. Result: vendors and data center operators couldn't justify using programmable switches, they were too big, too hot, and too slow.

✓ What changed?

Three technological trends made programmable switches finally viable:

- ⌚ **Chip speed caught up.** We now have programmable switch chips (like Barefoot Tofino) that can **run at line rate**, just like fixed-function ones. In other words, programmability no longer costs us speed.

- ⚠ **Network complexity exploded.** There are now **too many protocols and features** to hard-code everything into silicon:

- New protocols, encapsulations (VXLAN, GTP, QUIC, etc.)
- Monitoring, load balancing, AI, security; all need custom, real-time logic.

Hard-coding all of this would take years, and would never be flexible enough.

- ✓ **Moore's Law made logic "free".** Thanks to Moore's Law:

- We can double the amount of logic in the same area every 2 years.
- The **cost of programmability** in terms of **chip area and power** has become **negligible**.

Now, the logic that makes a switch programmable barely takes up more space than a fixed-function design.

Factor	Before	Now
Chip speed	Too slow for line-rate	Equal to fixed-function
Logic cost	Too expensive (area + power)	Basically free
Protocols	Few, stable	Too many to hard-code
Urgency	Low	High (cloud, IoT, 5G, ML)

Table 3: Why didn't programmable switches exist before?

3.3 Data Plane Programming and P4

Traditionally, configuring a switch meant **writing static forwarding rules**, usually via vendor-specific commands or protocols like OpenFlow. But this was **not true programmability**. We could **configure behavior**, but we **couldn't change how the switch processes packets internally**. With P4 (Programming Protocol-independent Packet Processors), that changes.

▀ What is P4?

P4 (**Programming Protocol-independent Packet Processors**) is a **high-level, domain-specific programming language** designed to **describe how packets should be processed by the data plane of a network device**.

Unlike general-purpose languages like C or Python, P4 is not Turing-complete. Instead, it is built to:

- Define **how to parse packet headers**
- Specify **how to match on those headers**
- Decide **what actions to take**

The **goal of P4 is to describe the behavior of the switch pipeline**, not to implement general algorithms. Specifically, P4 was designed with four main goals in mind:

1. **Reconfigurability**: We should be able to **change switch behavior after deployment**.
2. **Protocol Independence**: The switch should not be tied to Ethernet/IP / TCP. **We define the packet format**.
3. **Target Independence**: The same **P4 program should run on different hardware** (ASICs, FPGAs, software switches).
4. **Flexibility and Abstraction**: Developers write in P4, and the **compiler maps it to the switch's low-level pipeline architecture**.

⚠ P4 is so cool, but OpenFlow is not the same?

We already discussed what OpenFlow is in Section 2.4, page 32. The short answer is no, P4 is different.

- OpenFlow is a **control protocol** for **configuring predefined forwarding behavior**.
- P4 is a **programming language** for **defining the forwarding behavior itself**.

Let's make an analogy to understand the difference.

- **OpenFlow is like the driver of a regular car.** The driver can:

- ✓ Steer left or right
- ✓ Press the gas or brake
- ✓ Use turn signals, radio, windshield wipers

But the driver can't:

- ✗ Change how the engine works
- ✗ Reprogram how turning the wheel affects the tires
- ✗ Add a new driving mode (e.g., “turbo boost”)

That's OpenFlow. We're in control of what happens (where to drive, how fast), but how the car works internally is fixed. We're controlling pre-built behavior, we're not changing the system.

- **P4 is like the car engineer or mechanic.** The car engineer can:

- ✓ Redefine how the steering works (e.g., make left turn rotate only one wheel)
- ✓ Change how the engine responds to the pedal
- ✓ Add entirely new modules (e.g., self-driving mode, rocket engine, etc.)

That's P4. We're not just driving the car, we're deciding what the car is capable of doing in the first place. We write the “rules” for how the system should behave.

❖ Workflow

1. Before starting to write a P4 program, is **necessary to know the P4 Architecture Model**. The **P4 Architecture Model** is a **logical interface** between:

- The **P4 program** written by the developer.
- The underlying **hardware target** (e.g., ASIC, FPGA, software switch)

This model tells the compiler: “here's what the hardware looks like, these are the building blocks our P4 program can use.”. This abstracts away hardware details and makes P4 programs portable across multiple targets.

It's pretty obvious that the P4 architecture model is defined by the hardware switch we have. Because if our switch doesn't support some feature (e.g. packet cloning, a second pipeline), we can't use it.

2. **Write the P4 Program.** The network operator or developer writes a P4 program to describe:

- Which **packet headers** to parse (e.g., Ethernet, IP, or custom)
- What **tables** to build (match fields, actions)

- How the **control flow** works (pipeline logic)
- What actions to perform (forward, drop, modify, etc.)

This is written in a .p4 file.

3. **Compile the P4 Program.** The P4 program is passed to a **P4 Compiler**, which does two main things:

- (a) **Generates a device-specific binary.** This is tailored to the target hardware (e.g., Tofino, FPGA, software switch like MBv2).
- (b) **Produces a runtime API.** This allows a controller (or CLI) to: install rules (e.g., match on `dstIP=10.0.0.1` forward to port 3), modify tables dynamically.

The result is something the switch can understand and execute.

4. **Deploy to the Switch (Target).** The compiled **output is loaded onto a P4-capable target**, such as: an ASIC (e.g., Barefoot Tofino), an FPGA-based switch, a software simulator (e.g., BMv2). At this point, the switch now knows how to: parse packets, match them in tables, take programmed actions.
5. **Runtime Table Configuration.** Once the program is installed, we still need to:

- **Populate the tables** with actual forwarding rules.
- This is usually done via a **controller**, using a runtime API (e.g., gRPC, Thrift, P4Runtime)

It's like programming the switch with policy, after the logic has been defined.

Finally, the user is only concerned with the P4 program and the controller (to populate the tables). Instead, the P4 compiler, the P4 architecture model, and the switch (e.g., ASIC) are provided by the vendor.

3.4 PISA and Compiler Pipeline Mapping

Protocol-Independent Switch Architecture (PISA) is the **hardware abstraction** used by modern programmable switches (e.g., Barefoot Tofino). The idea behind PISA is simple but powerful: instead of building fixed-function blocks into hardware (e.g., IP routers, firewalls), **expose a generic pipeline of programmable stages**, and let software define what each stage does.

💡 PISA Architecture

A PISA switch consists of the following main components:

- **Parser.** Extracts packet **headers** and creates a structured **representation** (called a **Packet Header Vector**, or PHV). The PHV contains the keys for the match-Action units.
- **Multiple Match-Action Stages.** A pipeline of identical stages. Each stage:
 - Matches on some fields (using SRAM or TCAM)
 - Executes simple actions (via Arithmetic Logic Units - ALUs)
 - Modifies the PHV (e.g., changing a header field, setting a drop flag)
- **Deparser.** Reassembles the packet by combining the (possibly modified) headers and payload. Every packet flows through this pipeline, so the logic must be fully deterministic and parallelizable.

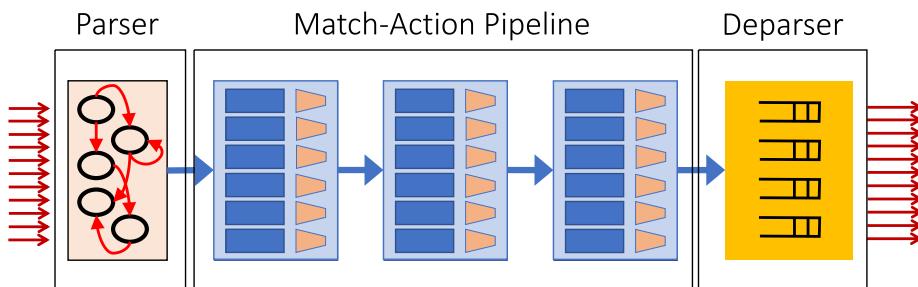


Figure 2: PISA architecture.

❓ Why use a pipelined architecture instead of a single processor?

A naive design would use **one CPU** to handle every packet: perform all lookups (routing, ACLs, NAT, etc.), apply all rules. But this would require an **unrealistically high frequency** to process billions of packets per second.

Just like in CPUs, we divide the processing into **stages**, each with: local memory (tables), local ALU, fixed resources. Each packet moves one stage forward per clock cycle, so we can **process many packets in parallel**.

✓ Protocol Independence

One of **PISA**'s most powerful features is that the chip **knows nothing in advance**.

- It doesn't recognize IP, Ethernet, TCP, or any protocol at all.
- The **programmer defines everything**: what headers to parse, what fields to match, what actions to perform.

This is what makes it **protocol-independent**, and feature-proof.

✖ What does the compiler do?

Here's the key part of PISA and P4: we don't directly **program the pipeline, the compiler does**. We write a logical program in P4, and the P4 compiler:

- Analyzes dependencies between operations:
 - **Match dependency**: A table needs data generated by a previous match.
 - **Action dependency**: An action needs a value produced by a previous action.
- Packs logic into stages without violating resource limits
- Ensure parallelism and no data hazards

4 Data Structures

4.1 Introduction

Modern network devices, particularly programmable switches (PISA, page 46), implement a **packet processing pipeline** composed of three main blocks:

1. **Parser**: Extracts relevant headers from incoming packets.
2. **Match-Action Pipeline**:
 - **Match**: Uses lookup tables to compare extracted headers against known values.
 - **Action**: Applies logic (e.g., modify headers, make routing decisions).
3. **Deparser**: Reassembles the final packet for transmission.

This flow is deterministic and must maintain constant processing latency per stage, as switches are often implemented as hardware pipelines (one packet per stage per clock cycle).

💡 Layer 3 (L3) Router

The L3 router is a classic example used to explain the packet matching process:

- **Input**: IP destination address from packet.
- **Match Logic**: Find the Longest Prefix Match (LPM) in a routing table.
- **Action Logic**: Forward the packet to the correct output port, and adjust MAC address accordingly.

Longest Prefix match (LPM) is a fundamental concept in IP routing, where the **goal is to find the most specific route** (i.e., the one with the longest matching prefix) **for a given IP destination address**.

When a router receives a packet, it checks the **destination IP address** and compares it to entries in its **routing table**, which typically contain IP prefixes like:

- 192.168.0.0/16
- 192.168.1.0/24
- 192.168.1.128/25

The router selects the entry whose prefix **matches the destination address** and has the **longest subnet mask** (i.e., most specific match).

In high-speed routers or programmable switches, **LPM must be done very quickly**, ideally in constant time. The naive solution for LPM is linear search over all routing entries. However, with thousands of entries, this is computationally infeasible at line rate. So the key questions in this section are:

- How do we **efficiently implement LPM**?
- Which data structures allow **fast lookups in a predictable and limited time**?

4.2 Ternary Content Addressable Memory (TCAM)

Ternary Content Addressable Memory (TCAM) is a specialized kind of (hardware) memory that works differently from standard RAM. Instead of accessing data by address, TCAM lets us input data and instantly tells us if and where it's stored. This is called **associative memory** or **content-based lookup**. It is built specifically for fast parallel search.

Unlike binary memories (which store 0s and 1s), **TCAMs can store 0, 1, or a third state** (ternary) called “*don't care*”. This third value allows flexible and partial matching, making TCAMs very effective for operations like Longest Prefix Match (LPM) in IP routing.

RAM vs TCAM

- RAM (Random Access Memory):
 - Ask: “**What** is stored at address X?”.
 - Classic address-value access.
- TCAM:
 - Ask: “**Where** is the value X stored?”.
 - The memory searches all entries in parallel and returns the matching address in constant time. In other words, it **returns the address where the value is stored**.

This associative search is **very fast**, which is why TCAM is often used in **packet classification** and **routing tables** in high-speed switches. Usually these two hardware are **put together** because the TCAM gives the index, we use it to index the RAM, and we get the information.

✓ Pros

- ✓ **Speed**: Lookup happens in constant time, regardless of the number of entries.
- ✓ **Wildcard Matching**: TCAMs handle “*don't care*” bits, allowing prefix and pattern-based lookups.
- ✓ **Ideal for Match-Action Pipelines**: TCAM is a good fit for hardware pipelines like those found in P4-programmable switches.

✗ Cons

- ✗ **High power consumption**: Every lookup checks all entries in parallel.
- ✗ **Expensive**: Due to the hardware complexity and power demands.

Example 1: TCAM in packet routing

Imagine a TCAM storing IP prefixes:

- 0: 192.168.3.0/24
- 1: 192.168.1.0/24
- 2: 192.168.2.0/24

If an incoming packet has destination IP 192.168.2.1, the TCAM instantly finds that it matches entry 2.

However, this match index alone isn't enough to decide what to do. So, we usually pair TCAM with a RAM block that stores the actual action:

1. TCAM gives the index
2. Use it to index RAM
3. Get forwarding info, output port, etc.

⚠ Dealing with Multiple Matches

Sometimes a destination IP can match multiple entries. For example:

- Entry 2: 192.168.2.0/24
- Entry 3: 192.168.2.0/28

Both may match the same address, but **only one result is returned**. Depending on the hardware, this could be:

- The **lowest matching index** (first match)
- The **highest matching index** (last match)

To ensure correct behavior (e.g., always choosing the most specific prefix), **entries need to be carefully ordered**. This introduces extra logic during configuration or compile time.

💡 Extra Hardware: SRAM

Alongside TCAM, **each pipeline stage** may also have **SRAM**. It's used for:

- Storing values linked to TCAM matches.
- Keeping state (e.g., counters, flags).
- Performing fast value retrieval during match-action processing.

SRAM is faster and cheaper than TCAM, but does not support associative lookup, so it **complements TCAM** rather than replacing it.

Key Takeaways

- TCAM is **fast**, parallel, and supports wildcards, great for networking.
- It's **costly and power-hungry**, so it's used sparingly and carefully.
- Works in tandem with **SRAM** for decision and action pipelines.
- Entry **ordering matters** to get correct behavior (e.g., longest prefix match).

4.3 Deterministic Lookup with Probabilistic Performance

② Problem Setup

We want to store a collection of elements (a set) in memory, and be able to:

- **Insert** new elements.
- **Check** if an element exists.
- **Do it fast**, ideally in constant time.

There are two broad strategies:

- **Deterministic** (this section):
 - ✓ **Always gives the correct answer** (*deterministic lookup*, answer is always correct).
 - ✗ **Slower or require more memory** (*probabilistic performance*, time depends on insertion history).

With this type of data structures, we **always get the correct answer** (true if present, false if not), but the **number of steps** (e.g. in Separate Chaining, explained below, the steps are determined by traversing a chain) is **not fixed** because it depends on: collisions, load factor, quality of the hash function.

- **Probabilistic** (section 4.4, page 55):
 - ✓ **Uses less memory or is faster** (*deterministic performance*, always the same number of operations).
 - ✗ **Might give false positives/negatives** (*probabilistic lookup*, result might be wrong with some small probability).

With this type of data structures, the **time is constant**, we have a fixed number of bit checks (usually one or a few), but the **answer can be wrong**. We can get a false positive (return true if the element isn't in the set), or we never get a false negative (if it says false, the element definitely wasn't inserted).

In this section we analyze the deterministic approach, so an output that we know what is, but the number of operations required is unknown (probabilistic). This is not suitable for network computing because it is detached from the PISA idea, but we present it for academic purposes.

■ Hash Table

A **Hash Function** maps **data** of arbitrary size (e.g., strings like "hello") **into a fixed-size integer space**. This **integer** is then **used as an index in an array** called a **Hash Table**.

Pseudo-code:

```

1 index = hash("hello")
2 hash_table[index] = "hello"

```

But **collisions can occur**, multiple inputs may hash to the same index. To handle this, we **use separate chaining**.

✓ Separate Chaining: The Basic Idea

The basic idea of **Separate Chaining** is as follows. If two values hash to the same index, we **chain them together in a list**:

Index 10 → “hello” → “port” → “fire”

So instead of storing just one value per index, we allow **each index to store a linked list** (or vector, or queue).

⌚ Performance Analysis

Let's say we're inserting N elements into a table with M buckets.

- **Average list size:** $\frac{N}{M}$
- **Best case** (uniform distribution): All chains are of similar length → fast lookups.
- **Worst case:** All N elements hash to the same bucket → one long chain → $O(N)$ lookup time.

So the **load factor** $\frac{N}{M}$ is key to understanding performance.

⚠ Collision Probability

How likely is it to avoid collisions at all?

- 1st insertion: no collision.
- 2nd: no collision with probability $1 - \frac{1}{M}$
- 3rd: no collision with probability $1 - \frac{2}{M}$
- ...
- N -th: no collision with probability $1 - \frac{N-1}{M}$

Multiply all together to get the probability of **zero collisions**:

$$P(N, M) = \prod_{i=0}^{N-1} \left(1 - \frac{i}{M}\right) \quad (1)$$

This means that even if $M = 10000$ and $N = 100$, the chance of having at least one collision is about 40%! In other words, **collisions are almost inevitable unless $M \gg N$** .

✓ Pros and ✗ Cons of Separate Chaining

- ✓ **Output deterministic and accurate**, no false positives/negatives.
- ✓ Simple and well-understood.
- ✓ **Performs well if load factor is low.**
- ✗ **Memory usage can grow** if many chains form.
- ✗ **Slower when many elements are inserted and collisions increase.**
- ✗ Not ideal for extremely large-scale or memory-constrained environments.

4.4 Probabilistic Data Structures

4.4.1 1-Hash Bloom Filters

We've just seen **Separate Chaining**, which gives **accurate answers** but has **unpredictable performance**, not ideal for hardware pipelines. Now we flip the perspective.

This section introduces **probabilistic data structures**, where:

- ✓ Insertions and lookup have a fixed, **deterministic number of operations**, typically 1.
- ✗ However, the **lookup result is probabilistic**, so it can produce false positives with a small probability.

Why this trade-off? Because in networking hardware (e.g., PISA architecture), we care more about **fixed latency** than occasional inaccuracies.

✓ A simple bit-based Data Structure

Let a set implemented as a simple bit array:

- An array of M 1-bit cells, all initially set to 0.
- To insert an element:
 1. Compute a hash function $\text{hash}(x)$
 2. Set the bit of the result of the hash function to 1: $\text{bit}[\text{hash}(x)] = 1$
- To check if x is in the set, we simply: $\text{bit}[\text{hash}(x)] == 1$

This data structure is often called a **1-hash Bloom Filter** because it has only **one hash function** and only **one bit per element**.

Example 2: Single-Hash Bloom Filter

Let an array of M 1-bit cells, all initially set to 0, we insert:

1. “Rust” → sets 1 bit of the array to 1
2. “Hello” → sets another bit to 1
3. “Fine” → sets another bit to 1. Now 3 bits are set

Now we will try some lookups.

- “Hello” → $\text{bit}[\text{hash(Hello)}] == 1?$ YES → ✓ return true
- “Bye” → $\text{bit}[\text{hash(Bye)}] == 1?$ NO → ✗ return false
- “P4” → $\text{bit}[\text{hash(P4)}] == 1?$ YES → ✓ return true, but we never inserted it. It is a **false positive**

⌘ Probabilistic Analysis

Let:

- N : number of inserted elements
- M : number of bit cells

Probability that an element maps to a particular bit is:

$$\frac{1}{M}$$

So:

- Probability that an element doesn't map to a bit:

$$1 - \frac{1}{M} \quad (2)$$

- Probability that a bit stays 0 after N insertions:

$$\left(1 - \frac{1}{M}\right)^N \quad (3)$$

- Probability that a bit becomes 1, called **False Positive Rate (FPR)**:

$$\text{FPR} = 1 - \left(1 - \frac{1}{M}\right)^N \quad (4)$$

- Finally, the **False Negative Rate** is 0. Bloom filters (1-hash or multiple hashes) **guarantee that they cannot return a false negative**. Suppose our hash function returns a value of 3 when we put in the string “Rust” (`hash(Rust) = 3`); if we put the word “Rust” into the bit array, we have `bit[hash(Rust)] = 1` \Rightarrow `bit[3] = 1`. Later, when we query “Rust”, the data structure will always return `true`, because `bit[hash(Rust)] = bit[3] == 1 ? true`.

✓ Pros

- ✓ Simple
- ✓ Fast, constant-time insertion and query
- ✓ Deterministic performance, perfect for hardware pipelines

✗ Cons

- ✗ Not always accurate, there can be false positives.
- ✗ To keep FPR low (e.g. 1%), we need 100× more memory than elements.

4.4.2 Bloom Filters

A **Bloom Filter** is a space-efficient probabilistic data structure used for **membership queries**:

- **Fast** insertions lookups.
- **No false negatives**, but may return **false positives**.
- This trade-off is ideal for **fixed-latency, high-speed systems** (like programmable switches).

🔗 Generalization of the 1-hash Bloom filter to k -hash

To **reduce false positives**, we **extend the 1-hash Bloom Filter**:

- Instead of just 1 hash function, we use K different hash functions.
- Each function maps the input element to a different positions in the bit array.

📌 How Insertion Works

Let's say we want to insert "Rust":

1. Compute K hash functions in parallel:

$$h_1(x) \quad h_2(x) \quad \dots \quad h_K(x)$$

2. For each $h_i(x)$, set the bit at position $h_i(x)$ to 1.

- $h_1(x) = 1$
- $h_2(x) = 1$
- \dots
- $h_K(x) = 1$

❓ How Lookup Works

To check whether an element is in the set:

- Compute all K hashes
- If **all corresponding bits are set to 1**, we return **true** (element may be present)
- If **at least one bit is 0**, we return **false** (element is definitely not present)

⚠ False Positives

Let's say "Fire" is not inserted but happens to have all its hash bits already set by "Rust", "Hello", or "Fine". The filter will wrongly return `true`, a false positive. Still, **no false negatives** can occur: if an element was inserted, all bits are set, and it will always return `true`.

% Probability Analysis

Let:

- N : number of **inserted elements**
- M : number of **bits in the filter**
- K : number of **hash functions**

Then:

- **Probability a particular cell is still 0 after inserting N elements:**

$$\left(1 - \frac{1}{M}\right)^{(K \cdot N)} \quad (5)$$

- **Probability of a false positive (all K bits set for a non-inserted element), the **False Positive Rate (FPR)**:**

$$\text{FPR} = \left(1 - \left(1 - \frac{1}{M}\right)^{(K \cdot N)}\right)^K \quad (6)$$

Just as an idea, with 1'000 elements inserted, 10'000 bits in the filter (cells), and 7 hash computations, we get a probability of FPR of only 0.82%. And if we increase the bits in the filter (M) to 100'000, the FPR is about 0%! So, with a **moderate increase in memory and hash computations**, we can get extremely low FPRs.

✓ Pros

- ✓ Very **memory-efficient**, uses up to $10\times$ less memory **than separate chaining**.
- ✓ Lookup and insertions are **predictable and fast**, constant time with K steps.
- ✓ Still **no false negatives**.

✗ Cons

- ✗ Requires **more computation** than the single-hash version (e.g., 7 hash functions).
- ✗ Slightly **more complex to implement in hardware**.

4.4.3 Dimensioning a Bloom Filter

We want to design a Bloom Filter that:

- Stores N elements
- Uses M bits (memory size)
- Applies K hash functions

But we also to **control the False Positive Rate (FPR)** and avoid unnecessary computation.

There are three parameters in play:

1. **Memory M** : more bits \Rightarrow lower FPR
2. **Number of Hashes K** : more hashes \Rightarrow lower FPR, but higher computational cost
3. **False Positive Rate (FPR)**: we want this to be as low as possible.

Improving one usually worsens another. This is the classic **space/time/error trade-off**.

➊ Asymptotic Approximation for FPR

In our case, the Asymptotic Approximation is a simplified mathematical expression that **estimates the False Positive Rate (FPR)** of a Bloom Filter when the number of **cells M is large**. It's derived from the exact expression but uses limits and approximations that hold when $M \gg N$. It's much easier to work with and very accurate in practice.

If we insert N elements into a Bloom filter with M bits and use K hash functions, the **exact False Positive Rate (FPR)**:

$$\text{Exact FPR} = \left(1 - \left(1 - \frac{1}{M}\right)^{(K \cdot N)}\right)^K \quad (7)$$

This expression can be tedious to compute, especially for large values of M , N , and K . By using the approximation:

$$\left(1 - \frac{1}{M}\right)^{K \cdot N} \approx e^{-K \cdot \frac{N}{M}} \quad \text{when } M \gg 1$$

The **Asymptotic Approximation of False Positive Rate (FPR)** is:

$$\text{FPR} \approx \left(1 - e^{-K \cdot \frac{N}{M}}\right)^K \quad (8)$$

This approximation is easier to analyze and is widely used in practice.

➋ Finding the Optimal Number of Hash Functions

The optimal number of hash functions K **minimizes the FPR** for given M and N . We can find it by minimizing the FPR formula:

$$K_{\text{opt}} = \frac{M}{N} \cdot \ln(2) \quad (9)$$

4.4.4 Counting Bloom Filters

In the standard Bloom Filter:

- Inserting an element means setting multiple bits to 1.
- But we **never know which element caused a bit to be 1**, because multiple elements may share the same hash outputs.

⚠ What happens if we try to delete?

Let's say we inserted: "Rust" and "Hello". And now we want to delete "Rust". If "Rust" and "Hello" both caused a bit (say, index 9) to be set to 1, and we reset it to 0 to delete "Rust", now:

- When we query "Hello", it might show a 0 in one of its position.
- This creates a **false negative**, which violates one of the core guarantees of Bloom filters!

So, **manually unsetting bits can remove evidence of other elements**.

✓ Solution: Counting Bloom Filters

To enable deletion, we **upgrade each bit into a counter**, this structure is called a **Counting Bloom Filter**. It works like this:

- Instead of a bit array, we use an **array of small integers**.
- When **inserting**, for each hash $h_i(x)$, increment $\text{counter}[h_i(x)]$.
- When **deleting**, for each hash $h_i(x)$, decrement $\text{counter}[h_i(x)]$.

We can safely decrement counters, knowing that **only when the last element that hashed to that index is deleted will the counter reach zero**. All previous analyses about false positives, FPR formula and K optimal are still valid, but now we **use more memory** and **add increment/decrement logic**.

⚠ Risk: Counter Overflow

Counters must be large enough:

- If they **overflow** (e.g., go above 255 for 8-bit counters), the **filter can become corrupted**.
- Worse, if a counter **underflows** (e.g., we delete too many times), we might **accidentally remove bits** for elements still in the set \Rightarrow **false negatives**.

4.4.5 Invertible Bloom Lookup Tables (IBLTs)

With Count Bloom Filters, we can:

- Insert elements
- Delete them
- But we **can't list what's inside**, or **retrieve keys/values**, the information is “smeared” across the structure.

Now we want something more powerful that can also list all entries or recover a specific key-value pair.

▀ What is an IBLT?

An **Invertible Bloom Lookup Table** is a data structure that:

- Stores **key-value pairs**
- Supports **deletion** and **enumeration (listing)**
- Is inspired by Bloom Filters, but has a richer cell structure.

Each cell contains three values:

1. **Count**: how many key-value pairs map to this cell.
2. **KeySum**: XOR (or sum) of all keys that mapped here.
3. **ValueSum**: XOR (or sum) of all values that mapped here.

We hash the key using multiple hash functions, just like a Bloom filter, and update each corresponding cell.

✚ Insertion

To **insert a key-value pair**:

1. Use K hash functions to map the key to K cells.
2. For each cell:
 - (a) Increment the **count**
 - (b) Add the key to **KeySum**
 - (c) Add the value to **ValueSum**

— Deletion

To **delete** a key-value pair:

1. Use the same K hash functions.
2. For each cell:
 - **Decrement** the **count**
 - **Subtract** the key from **KeySum**
 - **Subtract** the value from **ValueSum**

If the key was inserted, this will perfectly remove it.

Q Lookup and Recovery

To **find** a value for a key:

1. Try to find a cell where `count == 1` and the `KeySum == input key`
2. If found, then `ValueSum` gives the value associated with that key

But:

- If the key is mixed with other keys in all K cells, recovery is hard.
- That's why **some keys may not be recoverable immediately**.

② Enumerate everything stored in it

Once the structure is filled with multiple key-value pairs, we may want to enumerate everything stored in it, not just individual lookups. This process is known as **decoding** or **peeling** the IBLT. This restore operation is often used in real-world scenarios, for example, when we want to compare two sets of two different devices.

The **decoding algorithm** is:

1. Scan the table for a cell where:
 - `count == 1`
 - `KeySum` and `ValueSum` correspond to an actual key-value pair
2. When found:
 - Add the pair to output
 - Simulate deletion: subtract this key and value from all corresponding cells
 - Update the IBLT

For example:

- Initial IBLT contains:

Count	KeySum	ValueSum
1	7	98
2	202	48
3	209	146
2	159	101
1	50	45

- First, a cell with `count = 1` reveals:

- $(7, 98) \Rightarrow$ added to output
- Remove it from the IBLT (as if deleting it)

Count	KeySum	ValueSum
0	0	0
2	202	48
2	202	48
1	152	3
1	50	45

- After update:

- Next, find $(152, 3) \Rightarrow$ decode and remove

Count	KeySum	ValueSum
0	0	0
1	50	45
1	50	45
0	0	0
1	50	45

- Finally, we can easily retrieve the last one which is $50, 45$.

- The final result is:

$$\{(7, 98), (152, 3), (50, 45)\}$$

This process works **only if at least one of the key-value pairs is initially recoverable**, and then the remaining pairs become recoverable as the IBLT gets simplified.

⚠ Decoding problems

Sometimes, the decoding process **gets stuck**:

- All cells have `count > 1`, or are tangled with other keys
- We cannot isolate any key-value pair

When this happens:

- Listing **fails**
- The **IBLT** is said to be in a **non-decodable state**
- This usually happens when the **load factor is too high** (i.e., too many elements for the number of cells)

So IBLTs are powerful because allowing insertion, deletion, lookup and enumeration; but we need to allocate enough space, because if we overloaded, we risk failure to decode.

Feature	Standard Bloom	Counting Bloom	IBLT
Insert	✓	✓	✓ (key-value)
Delete	✗	✓	✓
Membership Test	✓ (yes/no)	✓	✓ (via decoding)
False Negatives	✗	✗	✗ (unless corrupted)
False Positives	✓	✓	✗ (when decoding works)
Listing Elements	✗	✗	✓ (if decodable)
Memory Efficiency	Very high	Moderate	Lower (more fields)

Table 4: IBLT vs Bloom Filters.

4.4.6 Count-Min Sketch

The **Count-Min Sketch** is a **probabilistic data structure** used to estimate the **frequency of elements** in a stream.

- We don't store each element individually.
- Instead, we use a **compat structure to maintain approximate counts**.
- It's designed for efficiency, especially when tracking millions of elements would be too memory-intensive.

❖ How does it work?

We create a 2D array of counters with:

- d rows (one per hash function)
- w columns (size of each hash domain)

This gives a table of size $w \times d$, much smaller than a full hash table for all possible items. **Each row has a different hash function**.

✚ Insertion

To insert an element (e.g. `ip.dest1`):

1. **Hash the element** with each of the d hash functions.
2. **Each hash** gives us a **column index in its row**.
3. **Increment** the corresponding **counters**.

So we increment **1 counter per row**, total d counters updated.

❑ Querying the Frequency

To estimate the count of an element:

1. Hash it again with the same d hash functions.
2. Get the counter values from the same positions.
3. **Return the minimum** of those d counters.

The **minimum** because:

- Collisions with other elements can cause overestimation (counters get inflated).
- But the **minimum is never less than the true count**, so it's a **safe lower bound**.

That's where the name comes from: count, because it estimates the frequency, and min, because it takes the minimum over multiple counters.

✓ Advantages

- Sublinear space: uses **much less memory** than a full table.
- **Fast**: insertions and queries are both $O(d)$ time (constant if d is fixed).
- Suitable for **high-speed data streams** (e.g., network flows, telemetry, monitoring).

Feature	Value
Use case	Approximate frequency counts
Memory	Sublinear ($w \times d$)
Insertion time	$O(d)$
Query time	$O(d)$, returns minimum
Overestimates	Possible
Underestimates	Never
Similar to	Counting Bloom Filter

Table 5: Count-Min Sketch summary.

5 Datacenter Monitoring

5.1 Why Datacenter Monitoring Matters

Imagine we're running a distributed application in a datacenter, and performance suddenly degrades. The possible root causes can be multiple:

- A software bug in the application logic.
- Network congestion between the servers.
- A broken fiber cable disrupting communication.
- A hardware failure, e.g. broken switch.
- A network misconfiguration that reroutes traffic inefficiently.
- A bug in the routing protocol.
- And many more...

There is a huge space of possible issues, and **pinpointing the root cause without visibility is extremely difficult**.

Many papers describe the importance of monitoring in datacenters.

- In the *Pingmesh* [6] article, they point to research that has begun to investigate **how to distinguish network problems from application-level bugs**. They highlight the diagnostic ambiguity in complex systems. Without monitoring, it's extremely hard to tell whether a slowdown is due to:
 - Software bugs;
 - Application overload;
 - Or actual network failures.

Monitoring systems must disambiguate the root cause across layers, application vs network.

- In the “*Understanding and Mitigating Packet Corruption in Data Center Networks*” [13] article, they show how **minor misconfiguration or failures** (e.g., wrong routing entry) can ripple through a system, **creating major outages**. It stresses that even low-level, seemingly unimportant events must be visible to prevent or debug large-scale issues. For example, a single corrupted forwarding rule in a switch might cause traffic loss affecting thousands of users.

Monitoring must include fine-grained data (like per-packet or per-flow telemetry) **to detect these small but critical problems**.

- In the “*Flow Event Telemetry on Programmable Data Plane*” [12] article, they show that **performance degradation often happens silently**, with no clear immediate failures. These “gray failures” don't crash systems but hurt performance. They're invisible without high-resolution monitoring (latency histograms, queue lengths, retransmits, etc.).

Monitoring should detect subtle deviations, not just crashes or time-outs.

- In the *CloudCluster* [10] article, they push toward deep programmability and visibility within the network. This points to the **evolution of monitoring tools**:
 - From passive logs and SNMP stats;
 - To programmable packet tracing and real-time telemetry;
 - That help pinpoint network issues quickly and accurately.

Visibility must be deep, dynamic, and distributed across the system.

5.2 Network Monitoring

Network Monitoring is the **continuous observation of a computer network to detect slowdowns or failures in components**. Its purpose is to detect, localize and respond to faults before they impact users.

⌚ Monitoring Scope

Monitoring spans the **entire network path**. Each router (or switch/server) is a point where failures or slowdowns can occur:

- A switch could drop the packet silently.
- A routing issue could cause the packet to loop.
- A delay could occur due to congestion in queues.

To effectively detect and diagnose problems, the monitoring system must observe not just endpoints, but the entire path, or at least enough of it to detect *where* things go wrong, or understand *why* a packet failed to reach its destination. This is why datacenter monitoring often tries to trace or mirror packets at different points in the network, to reconstruct the packet's journey and find anomalies.

🛠️ Monitoring Techniques

There are many ways to monitor a network. It can be done:

1. From Switches:

- (a) **Built-in Features** (section 5.3, page 71)
 - *NetFlow*: collects IP traffic statistics.
 - *Mirroring*: duplicates selected packets for analysis.
 - *SNMP (Simple Network Management Protocol)*: polls device stats.

(b) Programmable Switches

- Use *data plane programmability* (e.g., P4 language) to define custom monitoring behaviors.
- Enables *custom counters*, tagging, filtering, or tracing at wire speed.

2. From Servers:

(a) Standard Tools

- `netstat`: network connections and stats.
- `tcpdump`: packet capture and inspection.
- `traceroute`: path tracing and latency.

(b) Ad-hoc Monitoring Services

- Lightweight daemons or agents tailored for the datacenter.
- Export performance metrics or send alerts.

There is no single way to monitor, a **mix of passive and active, centralized and distributed methods is used**. Monitoring systems must collect data from multiple vantage points to build a full picture of the network's health.

⚠ Why traditional monitoring isn't enough

In large-scale datacenters many failures are subtle and effect only specific flows of packets:

- **Silent packet drops.** Packets are dropped but not reported by switches. The causes are software bugs or faulty hardware.
- **Silent blackholes.** Traffic is blackholed without showing in forwarding tables. The causes are corrupted TCAM entries.
- **Inflated end-to-end latency.** Packet flow experiences unexpected delays. The causes are congestion or queuing.
- **Loops.** Packets circulate endlessly. The causes are middleboxes modifying headers or breaking routing logic.

These failures are:

- **Not visible** in flow-level stats.
 - **Not logged** by switches.
 - **Hard to localize** with only endpoint observations.
-  We need **per-packet visibility** to detect and understand them.

❓ So can we monitor every packet on the network?

Tracing all packets in large datacenters is not scalable:

- Aggregate traffic can exceed 100 terabit per seconds.
- Microsoft estimated 3200 servers needed just to collect and analyze the data (in 2015).

To make packet-level telemetry practical, some strategies are required:

1. Monitoring must be **selective and smart** (e.g., sample important flows).
2. Diagnosing problems often requires **correlating behaviors across multiple hops**.
3. **Passive tracing alone is insufficient:**
 - It may miss transient problems.
 - It lacks the context to localize root causes effectively.

5.3 Everflow

5.3.1 What is Everflow?

Everflow [5] is Microsoft's system for **packet-level telemetry in production datacenters**, and it is built around three key concepts:

1. **Match and Mirror on the Switch.** Everflow leverages the match-action capability of commodity switches. It **defines rules to match specific packets and then mirror (copy) them to a monitoring collector**. Three matching rules:

- TCP SYN / FIN / RST: to trace connection setup/teardown.
- **Special debug bit:** used to flag packets for tracing.
- **Protocol traffic:** such as BGP or other control plane packets.

This allows the system to **monitor important or suspicious traffic patterns without touching every packet**.

2. **Switch-Based Reshuffler.** Mirroring packets from all switches generates huge data volumes. A single analysis server can't handle this load. The solution is to **use one or more intermediate switches** (reshuffler) that:

- (a) Receive mirrored packets.
- (b) Distribute them intelligently across multiple collectors.

This **balances load and scales the telemetry infrastructure**.

3. **Guided Probing.** The system can **inject specific test packets into the network**. These packets are crafted to **explore or verify behaviors** (e.g., path correctness, loss, latency). They are useful because:

- Helps when match and mirror alone misses packets (e.g., for complete TCP flow analysis).
- Can reproduce or test suspected failures.
- Distinguishes between persistent and transient issues.

Uses DSCP bits (in IP headers) and parts of the IPID field to mark and sample packets.

Idea	Purpose	Key Technique
Match and Mirror	Capture relevant packets	Matching on SYN / FIN / debug bits; mirroring to collectors.
Reshuffler	Scale analysis	Distribute mirrored packets across servers.
Guided Probing	Actively test network behavior	Inject custom packets using special bit fields.

Table 6: Summary of Everflow concepts.

5.3.2 How it works

Everflow isn't just a single-purpose tool, it's an extensible framework that supports different debugging applications, all coordinated through a central controller.

1. **Everflow is Application-Driven.** Operators use Everflow to **run specific troubleshooting tasks**, such as:

- Latency profiling
- Packet drop debugging
- Loop detection

Each task is handled by an Everflow application tailored to that goal.

2. **The Controller as the Central Brain.** The **controller** coordinates the full debugging process:

- It receives:
 - (a) The operator's request (e.g., trace all flows to a web server).
 - (b) The expected network routing.
- It then:
 - (a) **[init] Installs match-and-mirror rules** in selected switches.
 - (b) **[config] Configures the analyzers** to process mirrored traffic.
 - (c) **[debug] Sets the debug bits** (e.g., using DSCP or IPID) in custom probes if needed.

This modular design allows Everflow to adapt to the operator's intent dynamically.

3. **Data Collection via Reshuffler and Analyzers.** Once rules are deployed:

- **Mirrored packets** from the switches **go to a Reshuffler**.
- The **Reshuffler distributes the traffic** to multiple **analyzers** (to balance the load).
- The **analyzers inspect** the packet streams for signs of abnormal behavior.

4. **Smart Storage: Only Save What's Important.** Even with match-and-mirror, the system can generate a **massive amount of trace data**. For optimization, the **analyzers write to memory** only packets with the **debug bit set**, or **packets that show anomalies** (e.g., unusual delays, missing responses). This filtering prevents overload and ensures only useful diagnostic data is saved.

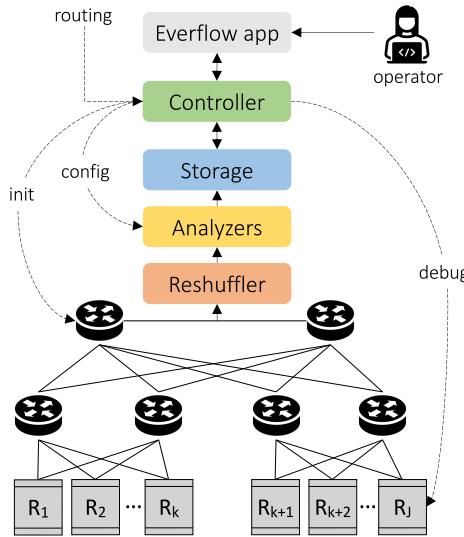


Figure 3: Summary of Everflow’s End-to-End operation:

- **Operator Request.** Chooses a debugging goal (e.g., latency analysis).
- **Controller.** Interprets request, maps it to rules and config.
- **Switch Configuration.** Match and mirror rules installed.
- **Probing (Optional).** Probes injected with debug bits.
- **Data Reshuffling.** Mirrored packets routed to analyzers.
- **Analysis.** Analyzers check for problems.
- **Selective Storage.** Only suspicious packets are saved.

Example 1: Real episode

Internal users reported that **some connections to a web service were timed out**. This violated the service level agreement (SLA). The root cause was suspicious: packet drops were occurring, but *where exactly?*

The service architecture involved multiple components: clients, load balancers, web servers, databases. All interconnected over the datacenter network. But the **datacenter is huge**, with many possible failure points.

The investigation begins.

1. Load Balancers showed no errors in their counters.
2. Some switches were checked manually, no issue found.
3. But the problem persisted, random connection timeouts were still

happening.

This is where Everflow comes in.

1. Everflow was used to **mirror TCP SYN packets** (which initiate connections) across the network.
2. Through its **trace analysis**, it was observed that:
 - Many SYN packets **never reached** the destination web server.
 - This only happened for **one specific web server**.
3. Further analysis revealed:
 - All SYN packets to that web server were **dropped at one switch**.
 - The switch showed **no error counters**, completely silent.

The root cause has been identified. The TCAM (Ternary Content Addressable Memory) on that switch was corrupted (TCAM stores forwarding table entries, used to decide where packets go). Because the corruption was silent:

1. The **switch dropped packets silently**.
2. **No logs, no alarms, no metrics**, traditional monitoring failed.

After a reboot of the switch, the issue disappeared.

5.4 FlowRadar

5.4.1 Architecture

FlowRadar [8] is a scalable, low-overhead solution for **per-flow monitoring** in datacenter networks. Its goal is to track **how much traffic each flow generates**, using **programmable switches** with fixed, minimal resources.

❷ Why FlowRadar?

Monitoring networks at flow-level granularity is **valuable but expensive**:

- ✖ **Packet mirroring.** Every interesting packet is copied and sent to an external analyzer.
- ⚠ **Problem:** way **too much traffic.** In datacenters with 100 terabit per seconds, this would flood our monitoring system.
- ✖ **Per-flow counters at switches.** Maintain one counter per flow inside the switch.
- ⚠ **Problem:** switches have **very limited memory.** With millions of flows, we run out of space fast.

So FlowRadar sits in the sweet spot:

- ✓ It avoids mirroring massive amounts of data.
- ✓ It doesn't require full flow counters in the switches.
- ✓ It **works with fixed, limited operations per packet**, suitable for programmable hardware.

The main idea is to encode information compactly in the switch, then **decode it later at the collector**.

❖ How it works

The FlowRadar works in three different ways:

1. **In the Switch.** Each switch maintains a **compressed data structure** to track flows and their counters. The **structure is similar to an Invertible Bloom Lookup Table** (IBLT, page 61), it records **flow IDs and counters** in a space-efficient way. Operations per packet are fixed and fast (ideal for hardware).
2. **Periodic Reports.** Switches **periodically export** their **encoded flow data to central collectors**.
3. **At the Collector.** Collectors receive the compressed data. Using multiple switch reports, they **correlate and decode** per-flow information. This allows the network operator to recover flow ID, and packet or byte count per flow.

5.4.2 Data Structure used in FlowRadar

Switches have very limited memory, but we want to count how many packets are part of each individual flow. Instead of using a separate counter per flow, which would consume too much space, FlowRadar stores compressed aggregate information in a fixed-size structure.

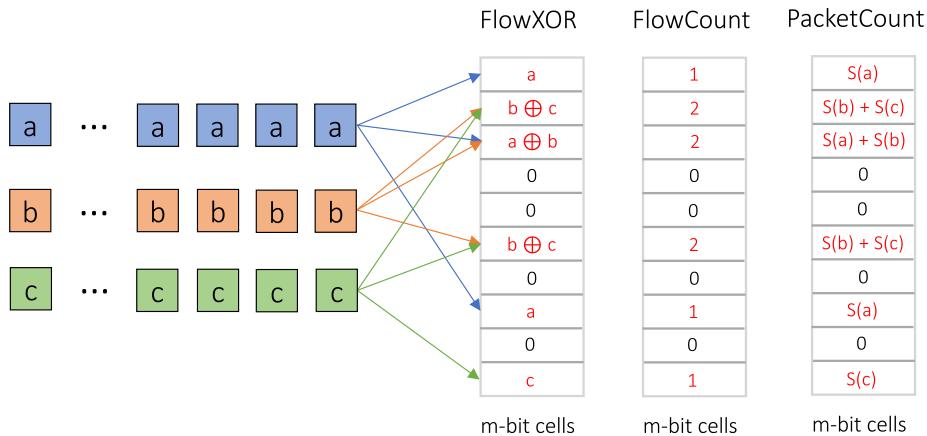
Each switch maintains three tables of m cells:

- **FlowXOR**. XOR² of all flow IDs hashed into each cell.
- **FlowCount**. Number of flows that map to each cell.
- **PacketCount**. Total number of packets from those flows.

For example, assume flows A, B, and C are all seen by the switch. Each flow is hashed into multiple cells. The tables get updated like this:

- FlowXOR: $a \otimes b$, $b \otimes c$, etc.
- FlowCount: how many flows are hashed into each cell (1, 2, etc.)
- PacketCount: total packets seen in each cell (e.g., $S(a) + S(b)$)

So each cell contains a mix of data from different flows.



❸ Why use XOR?

Because the XOR operation is **reversible**. If we know the XOR of two values and one of them, we can recover the other. This **allows the collector to decode the original flows** by:

1. Getting reports from multiple switches.
2. Iteratively solving the system of XOR equations.

²XOR (Exclusive OR) is a binary operation denoted by \otimes , where the result is 1 if the two input bits are different, and 0 if they are the same.

⚠ What is Flow Filter and why do we need it?

The **Flow Filter** is a small data structure (like a Bloom Filter) used inside the switch to remember which flows the switch has already seen.

Let's say flow a sends 10 packets. All those packets will pass through the switch. But we don't want to treat each packet like a new flow, **we only want to register flow a once in the compressed counters**. If we update the XOR and FlowCount on every packet:

- The **FlowXOR** would get corrupted.
- The **FlowCount** would become too high.
- We'd lose the ability to decode the flows correctly later.

So the **Flow Filter** helps us avoid this.

❷ What does Flow Filter actually do?

For each packet:

1. The switch looks at the Flow ID (e.g., source IP + dest IP + ports).
2. It checks the Flow Filter:
 - If the flow is **new** (not in the filter yet):
 - (a) It updates:
 - FlowXOR: add this flow's ID via XOR.
 - FlowCount: increment the count.
 - PacketCount: add 1.
 - (b) It marks this flow as *seen* in the filter.
 - If the flow is **already known**:
 - (a) It updates **only** PacketCount.

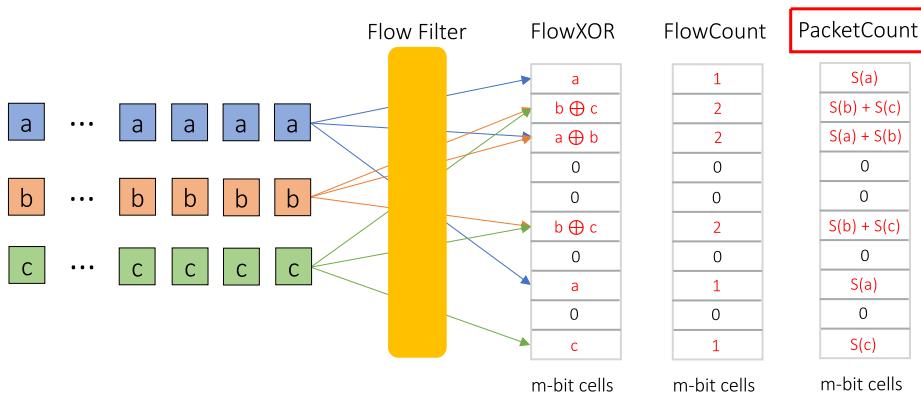


Figure 4: Correct representation of the data structure used in FlowRadar.

5.4.3 Collector Decode

Once the switch sends its tables (FlowXOR, FlowCount, PacketCount) to the collector, there are **two stages of decoding**:

1. **Single Decode** (local), performed at the collector level, recovers flows where the data structure is clean enough.
2. **Network-Wide Decode** (distributed), performed across multiple collectors/switches, combines views from many switches to fully decode the remaining flows.

Step 1 - Single Decode

Single Decode is the **local decoding stage**, it happens at one switch (or collector handling one switch's data). The goal is to **recover as many flows as possible** using **only the local FlowXOR, FlowCount, and PacketCount tables** collected from that switch. The key idea is to find “pure” cells (cells containing information from only one stream) and start the decoding process:

1. **Find a Pure Cell.** A cell is **pure** if **FlowCount = 1**. It means only one flow was hashed to that cell. For example, let the following FlowRadar table, this step identifies the first row:

FlowXOR	FlowCount	PacketCount
a	1	5
$a \otimes b$	2	12
$b \otimes c \otimes d$	3	13
0	0	0
0	0	0
$b \otimes c \otimes d$	3	13
0	0	0
a	1	5
0	0	0
$c \otimes d$	2	6

2. **Remove the Flow's Contribution from Other Cells.** Flow a was hashed to multiple cells. So now we remove a 's effect from all its associated cells:

FlowXOR	FlowCount	PacketCount
0	0	0
b	1	7
$b \otimes c \otimes d$	3	13
0	0	0
0	0	0
$b \otimes c \otimes d$	3	13
0	0	0
0	0	0
0	0	0
$c \otimes d$	2	6

3. **Removal may produce purer cells.** By removing flow a , other cells might now have $\text{FlowCount} = 1$ (pure cell). Repeat the previous steps until everything is decoded.

⚠ **Possible Stall.** Some flows are mixed together in such a way that no cell has $\text{FlowCount} = 1$. The solution here is to **apply** the second decoding stage, called **network-wide decode**.

✓ Step 2 - Network-Wide Decode

In the previous stage, each switch tries to decode as many flows as it can **locally**, by identifying *pure* cells. But sometimes decoding gets stuck because:

- Flow cells contain multiple flow mixed.
- No pure cells remain.

To solve this, we use **network-wide redundancy**: packets of the **same flow** **traverse multiple switches**, and those switches may store **different parts** of the encoded data. By combining these views, we can **solve flows that are undecodable at any single switch**.

For example, image the following situation:

Switch 1			Switch 2		
FlowXOR	FlowCount	PacketCount	FlowXOR	FlowCount	PacketCount
a	1	—	$a \otimes d$	2	—
$a \otimes c \otimes d$	3	—	$a \otimes c$	2	—
$b \otimes c \otimes d$	3	—	$b \otimes c \otimes d$	3	—
$a \otimes b \otimes c$	3	—	$a \otimes b \otimes c$	3	—
$b \otimes d$	2	—	$b \otimes d$	2	—

In both switches, single decode fails. But when **merged**, we have enough constraints to decode all flows. This is similar to solving a system of equations with more equations than unknowns. In other words, we don't need massive memory in each switch, we can use **small, compressed flow encodings per switch** and decode everything later by **combining views across the network**.

⌚ But how do we know which switch saw which flow?

To decode accurately, we must know which switch processed which flow, because otherwise we might combine FlowXORs from switches that didn't see a particular flow (wrong result). The **solution is the Flow Filter**. Each switch has a Flow Filter, so we can:

1. Query the flow filter: "Did you see the flow a ?"
2. If yes, we use that switch's data for decoding flow a .

This **guarantees correctness in multi-switch decoding**.

Step	Description
1.	Each switch reports its compressed counters (FlowXOR, FlowCount, PacketCount)
2.	Some flows decoded via Single Decode
3.	Remaining flows are solved by combining equations from multiple switches
4.	Use Flow Filters to determine which switch saw which flows
5.	Network-wide correlation fully decodes the remaining flows

Table 7: Network-Wide Decode summary.

⚠ What if Switches Disagree Due to Packet Loss?

Imagine that:

- A packet of flow f passes through Switch A and Switch B.
- Due to **transient issue**, one of the switches **misses that packet** (e.g., due to mirroring loss or memory overwrite).

Now, when both switches report their Flow Radar data structure:

- The values for flow f might not match across switches.
- This creates **inconsistencies** when trying to decode.

✓ **Solution: Redundancy**. Even if packet counts differ, the set of flows seen by each switch can still be decoded. And more importantly, once we know which flows each switch saw, we can treat each switch's data as a system of linear equations and solve for the actual packet counts.

1. We use the **Flow Filter** to determine which flows each switch saw.

2. We **decode flow IDs** using the FlowXOR and FlowCount tables.

3. We set up a **linear system of equations** per switch:

- Each **cell** gives us an equation:

$$\text{XOR}(f_1, f_2, \dots) \Rightarrow \text{total packet count} = P$$

- We solve for the **unknown packet counts** of individual flows.

4. If the same flow has **different counts** across switches:

- It signals a possible **packet loss**;
- Or a measurement **inconsistency**

5.5 In-Band Network Telemetry (INT)

5.5.1 What is INT?

In-Band Network Telemetry is a framework where the **data plane itself** collects **telemetry information** as packets traverse the network. Instead of relying on mirrored copies or external probes (like Everflow or FlowRadar), INT embeds **telemetry instructions directly into packets**. This means that the **packet asks switches along its path to record certain metadata** (e.g., delay, queue size, switch ID).

INT demonstrated the power and usefulness of programmable switches. It was one of the first real-world use cases where P4 offered something no traditional switch could do.

❖ How it works?

1. Packets carry **INT headers**.
2. INT-capable devices **read the instructions** in those headers.
3. They **collect specific network state** and **append it to the packet** as it moves.
4. The telemetry data is delivered **in-band**, alongside the normal traffic.

The data collected cloud include: switch ID, input and output ports, queue occupancy, timestamp (arrival and departure), packet latency per hop.

⌚ Why INT?

INT solves key limitations of older monitoring tools:

- ✓ No need for **mirrored** traffic (like Everflow).
- ✓ **Real-time per-packet** information.
- ✓ **High visibility** into what happens at every hop.

This is especially useful for: fine-grained performance monitoring; diagnostic congestion, jitter, and path problems; dynamic traffic engineering.

5.5.2 Modes

Telemetry data can be collected and exported in different ways, depending on:

- Whether packets are modified or untouched
- Whether data is inserted in packets or sent to collectors separately
- How much pressure is put on switches vs collectors.

Each mode offers trade-offs between performance, visibility, and system overhead. There are three different modes:

1. **INT-XD (eXport Data).** Switches do not modify the packet. Instead, they send telemetry data directly to the collector based on local configuration.

Pros

- ✓ No changes to the packet ⇒ avoids MTU issues.
- ✓ Easier for legacy packet flows.

Cons

- ✗ Heavy pressure on collectors (they must gather data from all switches).
- ✗ Telemetry query is based on switch config, not what the packet asks.

2. **INT-MX (eMbed instruct(X)ions).** The packet is marked to indicate it wants telemetry. Switches send telemetry data out-of-band (to a collector), but based on the packet's mark.

Pros

- ✓ Telemetry is packet-driven, more dynamic than INT-XD.

Cons

- ✗ Still puts pressure on collectors.
- ✗ Requires modifying packets, could affect headers, MTU.

3. **INT-MD (eMbed Data).** The packet itself is modified to carry Telemetry metadata in-band. Each switch inserts data into the packet as it passes through.

Pros

- ✓ No extra pressure on collectors.
- ✓ Telemetry query is packet-dependent, enabling full visibility along the path.

Cons

- ✗ Packets are modified, which may:
 - (a) Break some applications;
 - (b) Exceed MTU (Maximum Transmission Unit);
 - (c) Require special handling at end hosts.

- Use INT-XD if we want no packet modifications and can tolerate heavy collector load.
- Use INT-MK for moderate flexibility but still out-of-band.
- Use INT-MD if we want **maximum in-path visibility** and can handle packet growth.

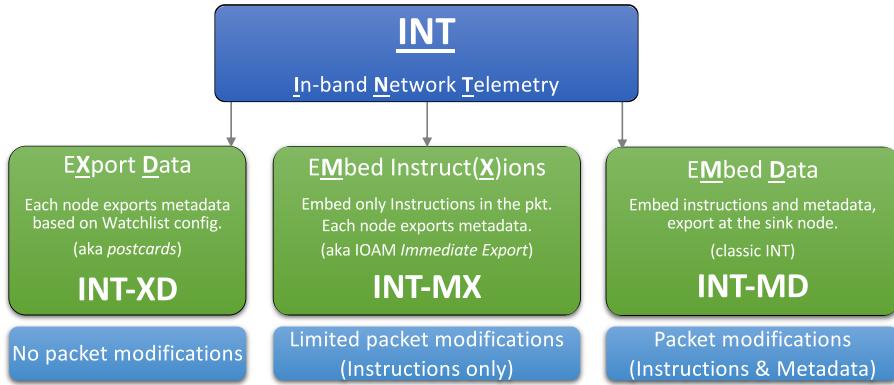


Figure 5: INT modes.

Mode	Packet Modified	Collector Load	Query Driven By	Pros	Cons
INT-XD	✗ No	✗ High	Switch Config	MTU-safe	Inflexible, collector-heavy
INT-MK	✓ Yes	✗ High	Packet Mark	Flexible	MTU risk, collector-heavy
INT-MD	✓ Yes	✓ Low	Packet	No collector pressure	MTU impact, complex parsing

Table 8: Summary of INT modes.

6 Datacenter Layer 3 Load Balancing

6.1 Recap: Datacenters

Characteristics of Workloads

Datacenters support very **different types of applications**, which shape the way networks are designed.

- **HPC (High-Performance Computing)**. Focus on scientific simulations, weather modeling, genome analysis. Workloads: large parallel computations. Require **low latency** and **high throughput** for inter-node communication.
- **Web services**. Think Google Search, Facebook, Instagram. Many short-lived requests/responses. Traffic is bursty, dominated by **small “mice flows”**. Latency-sensitive: users notice if it’s slow.
- **Machine Learning (ML)**. Training large models across GPUs/TPUs. Requires frequent **synchronization** (gradient updates). Workloads dominated by **large “elephant flows”** (bulk transfers). Need predictable, low tail latency (slowest worker delays the whole training).
- **Big Data (MapReduce, Spark, etc.)**. Shuffle phase = massive all-to-all data exchange. Demands **high aggregate bandwidth**. Workloads are throughput-oriented, but also sensitive to stragglers.

Traffic Variability

Datacenter traffic is not uniform. It’s a **mixture of short and long flows**, and this mix complicates network design.

- **Short flows (mice)**. Few KBs to MBs, bursty, latency-sensitive. For example: a web request, RPC, or cache lookup. Critical: the user’s experience depends on their fast completion.
- **Long flows (elephants)**. Hundreds of MBs to GBs, throughput-oriented. For example: dataset shuffling, ML gradient exchange, backups. Can easily congest network links if not managed carefully.
- **Traffic patterns**
 - **Shuffle traffic**: many-to-many, typical in MapReduce or Spark.
 - **Gradient aggregation**: many-to-one or all-to-all, in ML training.
 - **Incast**: one client requests data from many servers at once → bursts that overwhelm buffers.

Networks must serve both mice and elephants efficiently. Prioritizing one can hurt the other.

⌚ Goal: High Bisection Bandwidth

Split the network into two equal halves; the **bisection bandwidth** is the total capacity of the links connecting them. Instead, **full-bisection bandwidth** means that every server can communicate with every other at full NIC speed, without bottlenecks.

⌚ Why it matters?

- HPC: ensures all nodes in a parallel job can exchange data efficiently.
- Web services: prevents hotspots³ when thousands of requests are routed.
- ML/Big Data: allows large-scale shuffles without stragglers.

The main **challenge** is achieving full bisection bandwidth at low cost, but it requires special topologies (Fat-Tree, Clos, Jellyfish).

In conclusion, datacenters run a **mix of workloads** (HPC, web, ML big data), which produce **variable traffic patterns** (mice vs. elephant flows, shuffle, gradient aggregation). The **main networking goal** is to deliver **high bisection bandwidth** so any-to-any communication can happen efficiently.

³A **Hotspot** in a datacenter network is a **point of congestion**, usually a specific link or switch that gets overloaded with too much traffic, while other parts of the network remain underutilized.

6.2 Introduction

Remark: OSI model

The **OSI (Open Systems Interconnection) Model**, developed by the International Organization for Standardization (ISO), is a conceptual framework that standardizes how different systems communicate over a network. It divides network communication into **seven layers**, each with specific responsibilities, enabling interoperability between diverse systems and technologies.

The Seven Layers of the OSI Model:

1. **Physical Layer:** Handles the physical connection between devices, transmitting raw bits over a medium. It defines hardware elements like cables, hubs, and transmission modes (e.g., simplex, half-duplex). Examples include USB and Ethernet.
2. **Data Link Layer:** Ensures error-free data transfer between nodes on the same network. It manages framing, physical addressing (MAC), and error detection. Devices like switches and bridges operate here.
3. **Network Layer:** Responsible for routing and forwarding data across different networks. It uses logical addressing (IP) to determine the best path for data packets. Routers work at this layer.
4. **Transport Layer:** Ensures reliable end-to-end communication. It segments data, manages flow control, and handles error recovery. Protocols like TCP and UDP operate here.
5. **Session Layer:** Manages sessions between devices, including establishing, maintaining, and terminating connections. It also handles synchronization and recovery.
6. **Presentation Layer:** Translates data into a format suitable for the application layer. It handles encryption, compression, and data formatting (e.g., JPEG, MPEG).
7. **Application Layer:** Closest to the user, it provides network services like file transfer, email, and directory services. Protocols include HTTP, FTP, and SMTP.

❷ What is Layer 3 Load Balancing?

Layer 3 (L3) means the **network layer** in the OSI model, where **IP routing** happens. In a datacenter with many redundant paths (like a Fat-Tree), there are usually several routes of the **same cost** between a source and destination. **Layer 3 Load Balancing** is the process of:

- ❸ Choosing **which path** a packet (or a whole flow) takes among those equal-cost routes.
- ❹ With the goal of **distributing traffic** evenly across the network.

Unlike **Layer 4-7 load balancing** (used for applications, web servers, etc.), L3 load balancing doesn't care about *which service* is being accessed. It only cares about **routing packets** across the network fabric efficiently. It's a **network-wide optimization**:

- Not about picking *which server* handles a request,
- But about picking *which path* data takes to reach its destination.

② Why L3 Load Balancing is Needed

Modern datacenter topologies (like **Fat-Tree/Clos**) provide **many equal-cost paths** between any two racks. For example: rack A wants to send data to rack B → there may be 8, 16, or more paths that cost the same.

- ✗ If traffic always follows a single path, some links get congested (hotspots) and other links remain idle (wasted bandwidth).
- ✓ With **load balancing** the traffic is spread across multiple available paths. It ensures higher **bisection bandwidth utilization** and improves both **throughput** and **latency**.

The **goal** is to **maximize the use of the parallel paths** by distributing traffic wisely.

⚠ Challenges

Load balancing at Layer 3 (IP routing) is not trivial because of **traffic dynamics**:

- **Flows come and go quickly.** Millions of short-lived flows (RPCs, queries) coexist with long-lived flows (backups, ML training). If the algorithm reacts too slowly → short flows finish before rebalancing even happens.
- **Mix of short and long flows**
 - **Mice flows:** small, latency-sensitive, numerous.
 - **Elephant flows:** large, bandwidth-hungry, can dominate a link.

Balancing both types is tricky:

- If we spread elephants badly → collisions → hotspots.
- If we treat mice like elephants → too much scheduling overhead.
- **TCP sensitivity.** TCP assumes packets of a flow arrive in order. If packets of the same flow are split across multiple paths → reordering happens → TCP slows down (false congestion signals). This rules out naïve strategies like packet spraying.

So the challenge is balance traffic in real-time while respecting the nature of **mice vs. elephant flows** and avoiding TCP issues.

6.3 Packet Spraying

Packet Spraying is a load balancing technique where **each packet** of a flow is sent over a **different path** in the network, instead of keeping the whole flow on a single path. This technique **balances the load immediately** across all available links and ensures that no path is left idle, thus **optimizing network capacity utilization**. However, there is one **significant issue**: **packets of the same flow may arrive out of order** due to the varying delays of the different paths taken. TCP is confused by out-of-order delivery and thinks packets are lost, resulting in unnecessary retransmissions and reduced throughput.

💡 Idea of Packet Spraying

Instead of assigning an entire flow to **one path**, packet spraying sends **each packet** of the flow independently across different available paths. For example, we have 4 equal-cost paths.

- Packet 1 → Path A
- Packet 2 → Path B
- Packet 3 → Path C
- Packet 4 → Path D

The intuition of Packet Spraying: By spreading packets at the *finest granularity*, all network links are used more evenly, and congestion is less likely to form.

✓ Pros and ✗ Cons

✓ Pros

- ✓ **Great load distribution.** No path stays idle while another is overloaded. Utilizes all network capacity.
- ✓ **Simple logic.** Doesn't need complex flow classification or scheduling. Just a round-robin or randomized assignment of packets.
- ✓ **Fast reaction.** Even short flows (mice) benefit, because their few packets can be split across paths immediately.

✗ Cons

- ✗ **TCP reordering problem.** TCP expects packets of a flow to arrive **in order**. With packet spraying, packets take different paths → different latencies → **arrive out of order**. TCP interprets out-of-order packets as **loss** → triggers retransmissions, reduces congestion window, lowers throughput.
- ✗ **Hardware complexity.** Switches need per-packet decisions at line rate, which can be expensive.
- ✗ **Not suitable for elephants.** Large flows generate so many packets that reordering overhead becomes very high.

⚠ The Reordering Issue (and why Packet Spraying is absolutely avoided in production)

Let's say *flow F* has 3 packets:

- P1 goes on Path 1 (latency 5 ms).
- P2 goes on Path 2 (latency 8 ms).
- P3 goes on Path 3 (latency 6 ms).

They arrive at the receiver as: P1 → P3 → P2. This out-of-order process produces the following errors:

- TCP sees missing sequence numbers (it expected P2 after P1).
- Receiver sends **duplicate ACKs** (saying “I didn’t get P2”).
- Sender wrongly assumes **congestion/loss** → retransmits and slows down.

As a result, throughput drops and latency increases. CPU is wasted on useless retransmissions. That’s why it is **not used in production** as a general solution.

6.4 Equal Cost Multi Path (ECMP)

In datacenter networks (e.g., Fat-Tree/Clos topologies), there are **multiple equal-cost paths** between a source and a destination. Traditional IP routing normally picks **one path**, which wastes capacity. **Equal Cost Multi-Path (ECMP)** is the standard mechanism that allows a router/switch to use **all equal-cost paths**.

Main characteristics:

- **Per-flow load balancing:** ECMP does not spray packets individually (like packet spraying). Instead, it ensures that **all packets of the same flow follow the same path** and this avoids TCP reordering.
- **Hashing:** Switches compute a hash of packet header fields (usually 5-tuple: source IP, destination IP, source port, destination port, protocol). The hash value is mapped to one of the available next-hop paths. All packets of the same flow produce the same hash → go on the same path.

In summary, ECMP is a Layer 3 load balancing technique where each flow is assigned to one of the available equal-cost paths using a hash function on packet headers, ensuring packets stay in order and TCP remains happy.

Example 1

Say there are 4 equal-cost paths. A hash function outputs values 0-3.

- Flow A (src, dst, IP and port) hashes to 0 → path 1.
- Flow B hashes to 2 → path 3.
- Flow C hashes to 2 → also path 3.
- Flow D hashes to 1 → path 2.

Each flow is consistently mapped to one path. Packets stay in order.

⌚ Hash Collisions and Inefficiency

In ECMP, the hash function maps each flow to one of the available paths. If two or more **large elephant flows** hash to the same path, that path becomes congested and other paths may stay underutilized. This is called a **Hash Collision**. Collisions are harmless for tiny mice flows, but disastrous when multiple elephants collide.

? **Why Collisions Matter.** **Elephant flows dominate traffic volume.** Even if 90% of flows are small, the few elephants carry most of the bytes. If elephants collide on the same link, throughput is reduced and latency spikes for other flows sharing that link. It creates **hotspots** while parallel links sit idle.

⚠ Inefficiency. Hashing spreads flows *randomly*, not *evenly*. With k paths, the load per path can vary widely, especially when:

- The number of elephant flows is small.
- A few unlucky hashes cluster them together.

So the network's **theoretical capacity** is high, but **effective throughput** is lower due to imbalance.

Example 2: Hash Collisions and Inefficiency

Imagine 4 equal-cost paths and 3 elephant flows.

- Flow A → hashes to path 1.
- Flow B → hashes to path 1.
- Flow C → hashes to path 3.

Path usage:

- Path 1: 2 elephants (overloaded).
- Path 2: empty.
- Path 3: 1 elephant.
- Path 4: empty.

Outcome:

- Path 1 congests → throughput limited.
- 50% of available network capacity wasted (paths 2 and 4 unused).

ECMP's reliance on static hashing leads to **hash collisions**, where multiple large flows land on the same path. This causes **inefficient bandwidth utilization** and **network hotspots**, even though other paths are free.

⌚ There are two problems that ECMP still cannot resolve

Incast and rack skew are two problems that ECMP alone cannot solve. This is one of the reasons that pushes researchers to find a better solution.

- **Bursty Traffic (Incast).** Incast happens when **one receiver asks for data from many servers at the same time**. For example, a storage node requests blocks from 50 servers; all 50 servers respond **at once** and their packets all converge on the **same final link** to the receiver.

⌚ **Why ECMP doesn't help.** ECMP can spread traffic across multiple *upstream* paths. But the **last hop into the receiver** is always the same physical link. That link suddenly gets a burst of packets from 50 sources.

⚠ **The consequence.** The buffer at that last-hop switch **overflows**. Packets are dropped, so TCP retransmits. Latency increases dramatically.

So even if ECMP spreads flows earlier in the path, it **cannot prevent congestion at the final bottleneck** in incast scenarios.

- **Flow Skew Across Racks.** Skew means imbalance. Some racks **generate much more traffic** than others, depending on what services run there.

For example:

- Rack A: runs a database cluster → produces many **elephant flows**.
- Rack B: runs lightweight web servers → produces mostly **mice flows**.

ECMP hashes flows randomly, but Rack A already has more elephants, so its outgoing paths are **more likely to get congested**; Rack B's paths stay underutilized.

⌚ **Why this is a problem.** ECMP doesn't adapt to traffic intensity differences between racks. It treats all flows equally, ignoring that some racks are “heavy hitters”. So **persistent hotspots near busy racks** and wasted capacity elsewhere.

▀ ECMP in Production and Its Limitations

⌚ Why ECMP Was Adopted.

- There are three main reasons:
- **Industry standard:** ECMP is built into traditional routing protocols (OSPF, IS-IS, BGP).
 - **Easy to deploy:** No special hardware or centralized controller needed.
 - **Good enough for mice flows:** in web workloads (many small flows), ECMP spreads traffic fairly evenly. Avoids TCP reordering (a major plus over packet spraying).

⚠️ Observed Problems in Production. But at hyperscale (tens or hundreds of thousands of servers), ECMP inefficiencies become visible:

- **Static & oblivious to congestion.** ECMP only looks at header hashes, not at link utilization. A congested link may still attract new elephant flows while other links remain idle.
- **Flow collisions.** In large topologies, even with thousands of equal-cost paths, collisions between elephants are common. A few unlucky hashes waste a lot of bisection bandwidth.
- **Wasted capacity.** Studies (e.g., on fat-tree topologies with $\approx 27k$ hosts) showed ECMP could waste **over 60% of available bisection bandwidth** on average due to imbalance.
- **Long-lived collisions.** Once a flow is hashed to a path, it stays there. If that assignment is bad, the flow suffers for its entire lifetime.

ECMP became the **default production solution** because it's simple, distributed, and TCP-friendly. But at datacenter scale, its **static, hash-based nature** makes it inefficient, prompting research into **smarter, traffic-aware load balancers** like Hedera (SDN-based, page 97) and HULA (P4-based, page 101).

✓ Pros and ✗ Cons

✓ Advantages

- ✓ **Simplicity.** Uses a straightforward hashing mechanism. No central controller or complex scheduling needed. Easy to implement in commodity switches.
- ✓ **Avoids packet reordering.** All packets of the same flow follow the same path. TCP sees packets in order, so it doesn't mistakenly trigger retransmissions.
- ✓ **Good for many short flows (mice).** With millions of short, random flows, the hashing tends to spread them fairly well. This makes ECMP very effective in web-service workloads where flows are small and numerous.
- ✓ **Scalability.** ECMP is distributed: each switch does hashing locally. No centralized bottleneck, works across large-scale datacenters.
- ✓ **Widely supported.** ECMP is built into IP routing standards (OSPF, IS-IS, BGP). Already deployed in real datacenters today.

✗ Disadvantages

- ✗ **Hash collisions.** Two or more elephant flows (large flows) may hash to the same path. Result: some links get congested while others are idle → creates hotspots.
- ✗ **No congestion awareness.** ECMP assigns paths purely based on hash, not on current load. If one link is already overloaded, ECMP doesn't know → it may keep adding new flows there.

- ✖ **Unfairness.** Mice flows are fine, but a single elephant can dominate a link if unlucky with its hash. Other elephants hashed to that path suffer, while bandwidth on other links is wasted.
- ✖ **Static behavior.** Once a flow is mapped, it stays on that path until it finishes. ECMP doesn't migrate flows if conditions change.

ECMP is simple, scalable, and TCP-friendly, which is why it's the default in datacenters. But it's also **static and oblivious to congestion**, so it can lead to **hotspots** when elephant flows collide.

Deepening: How ECMP Uses Hashing to Pick a Path

Three steps:

1. **Multiple Equal-Cost Paths Exist.** Imagine a datacenter topology (like a Fat-Tree). From server **S1** to server **S2**, the routing protocol (e.g., OSPF, IS-IS) discovers that there are k **different next-hop paths** that all have the **same cost**. For example, 4 paths, so the next-hops are {N1, N2, N3, N4}. The routing table on the switch stores:

Destination S2 → Next-hops: N1, N2, N3, N4 (all cost 10)

So, the switch knows it *can choose* any of them.

2. **Switch Computes a Hash of the Flow.** When a new packet arrives, the switch looks at the **flow identifier** (usually 5-tuple: source IP, destination IP, source port, destination port, protocol). It computes a **hash function**, for example:

```
1 hash = H(srcIP, dstIP, srcPort, dstPort, proto)
```

This gives an integer value.

3. **Map the Hash to a Next-Hop.** Now comes the key: the switch takes the hash value **modulo the number of available next-hops (k)**:

```
1 path_index = hash % k
```

- If `path_index = 0` → send packet to N1.
- If `path_index = 1` → send packet to N2.
- If `path_index = 2` → send packet to N3.
- If `path_index = 3` → send packet to N4.

All packets of the same flow have same 5-tuple, then same hash and same path. Instead, different flows have different hashes and likely different paths.

❓ Why This Works

The **paths themselves aren't "hashable"**. Instead, the switch maintains a *list of possible next-hops* for the destination. The hash selects an **index** into that list. This is why the hash needs to be "uniform" → to spread flows across all next-hops evenly.

In other words, ECMP doesn't hash the paths themselves. It hashes the **flow's header fields** and then uses the hash result to pick an **index** from the list of equal-cost paths in the routing table.

❓ Why ECMP Uses Modulo on the Hash Value

We want to assign each flow to **one of the available next-hops**. Suppose there are k equal-cost paths (say 4). The switch needs a simple way to map the **huge range of hash outputs** (e.g., 32-bit integer) down to just 4 choices.

⚠ The Problem. Hash functions produce large numbers (e.g., $0 \dots 2^{32-1}$). But the switch only has a small number of next-hops (k paths). We need a consistent, deterministic way to map "large space $\xrightarrow{\text{to}}$ small space".

✓ The Solution: Modulo. Compute:

```
1 path_index = hash(flow_id) % k
```

The result is guaranteed to be in the range $0 \dots k - 1$. That matches exactly the **indices of the next-hop list**. For example, with 4 paths:

```
1 hash(flow A) = 57 → 57 % 4 = 1 → path 2
2 hash(flow B) = 134 → 134 % 4 = 2 → path 3
3 hash(flow C) = 29 → 29 % 4 = 1 → path 2
4
```

❓ Why Modulo Works Well. There are three main reasons:

1. **Uniformity:** If the hash function is good, the outputs are "random-looking", so modulo spreads flows fairly evenly across paths.
2. **Deterministic:** Same flow always hashes to the same path.
3. **Simple in hardware:** Modulo is fast and easy for switches to implement.

Modulo is used because it **compresses the large hash space into exactly the number of available paths**. That way, every flow gets assigned to one valid next-hop index.

6.5 Hedera: Dynamic Flow Scheduling

We just saw the weaknesses of **ECMP** (page 91):

- It **hashes flows blindly**, without knowing current congestion.
- Multiple **elephant flows** can collide on the same path, creating hotspots.
- Meanwhile, other links stay idle, wasted capacity.

Hyperscale datacenters⁴ needed a **smarter, traffic-aware solution** to balance flows dynamically.

❓ What is Hedera?

Hedera is a **dynamic flow scheduling system for datacenter networks**, proposed in a research paper at NSDI 2010. [1]

❓ **Key Insight of Hedera.** Most datacenter traffic volume is carried by a **small fraction of flows** (the elephants). Mice flows (small, latency-sensitive) are numerous but consume little bandwidth. If we can **identify and schedule only elephant flows** intelligently, we can **fix most congestion while keeping the system lightweight**.

❓ **The Problem Statement.** Hedera addresses this question: “*how can we schedule large flows in a datacenter network so that they are spread across available paths, avoiding hotspots and using full network capacity?*”. Specifically:

- Input: a set of **elephant flows** in a multi-path datacenter topology (e.g., Fat-Tree).
- Goal: assign each elephant to a path such that: network utilization is balanced; no link is overloaded while others are idle; small flows are not disrupted.

Hedera’s motivation is that ECMP wastes bandwidth by ignoring flow sizes, so it proposes a system that **detects elephant flows and dynamically schedules them across paths** to avoid congestion.

❖ Hedera Architecture (SDN + OpenFlow, Centralized Controller)

Hedera introduces a **centralized SDN controller** (page 30) that has a **global view of the datacenter network**.

- Switches **report flow statistics** (e.g., which flows they see, how much bandwidth each uses).
- The controller runs a **scheduling algorithm** to compute optimal paths for **elephant flows**.

⁴**Hyperscale datacenters** are very large cloud facilities designed to support tens of thousands of servers and millions of virtual machines, built with uniform, modular infrastructure (compute, storage, networking) that can scale out efficiently to meet massive and dynamic workload demands.

- The controller then **installs forwarding rules** in the switches using **OpenFlow** (page 32).

So instead of random per-flow hashing (ECMP), Hedera makes **explicit scheduling decisions**.

The **key components** of the Hedera architecture are:

- **Commodity switches (OpenFlow-enabled)**: forward packets based on flow rules (see below) and export flow-level statistics (e.g., byte counts, duration) to the controller.
- **Centralized Controller (Hedera brain)**: collects network-wide flow information, identifies elephants and assigns paths to elephants to balance load.
- **Flow Rules**: installed dynamically by the controller into switches. They specify that packets of flow F should go through next-hop N .

The detailed workflow is as follows:

1. **Flows arrive**: initially handled by ECMP.

❓ Wait, so ECMP isn't being replaced?

No! Because Hedera needs a **lightweight default mechanism** for small flows, and a **smarter mechanism** only for the big ones. So, **all flows start with ECMP**:

- New flows are hashed to a path immediately → very low latency to start forwarding.
 - No controller intervention required.
2. **Switch counters** reveal that some flows are big (exceed a threshold, e.g., > 10 MB). Switch counters report flow statistics to the Hedera controller.
 3. **Controller detects elephants**. If a flow grows beyond a threshold, it's classified as an elephant. So the **controller computes a better path** for that flow.
 4. **Controller installs new rules** in switches via OpenFlow.
 5. **Elephants are moved** to less congested paths, while mice stay with ECMP.

This was one of the first real examples of **SDN applied to datacenter load balancing**.

☒ Elephant vs. Mice Flow Scheduling

First of all, *why do we need different treatment?*

- **Mice flows (tiny, short-lived)**

- They are **the majority by count** but carry very little total traffic.
- They finish so fast that trying to schedule them centrally would take longer than the flow itself.
- They are **latency-sensitive** (user-facing requests).

- **Elephant flows (large, long-lived)**

- They are **few in number** but carry most of the bytes.
- If badly placed, they can congest links and hurt many other flows.
- They are **throughput-sensitive** (bulk transfers, ML gradient sync, big shuffles).

So Hedera's philosophy: **let mice run free (ECMP), but carefully shepherd elephants.**

- **Scheduling Mice Flows.** Mice flows use ECMP (hash-based, per-flow).

- ✓ Immediate forwarding, no controller involvement.
- ✓ Keeps latency low.
- ✓ Scales to millions of flows without overloading the controller.

- **Scheduling Elephant Flows.** The controller monitors flow statistics from switches. A flow is promoted to *elephant* if it exceeds a threshold (e.g., > 10 MB transferred). The controller computes a less congested path for the elephant and installs OpenFlow rules to move it there.

The main **goal** is spread elephants across available paths, avoid collisions (two elephants on the same path) and increase overall throughput and fairness.

☒ Flow Demand Computation Algorithm

Once Hedera identifies the **set of elephant flows**, it needs to estimate how much **bandwidth each elephant “wants”** (its demand) and assign flows to paths so that no single link is overloaded, and network utilization is balanced.

↳ Estimating Flow Demands. The **controller** collects statistics from switches: byte counters per flow and flow duration. From this, it computes the **demand**: expected bandwidth requirement of the flow. Demand is not just “how much data so far”, it is an estimate of how much the flow *will need* in the near future.

☒ Scheduling Algorithm. Hedera then runs a **demand-aware placement algorithm**:

1. **Construct a demand matrix**, where rows are sources and columns are destinations. Each entry is the total demand (sum of flows) between that source and destination.
2. **Solve a multi-commodity flow problem (approximation)**. A multi-commodity flow occurs when many flows compete for shared resources, such as links. The algorithm tries to maximize utilization while respecting link capacities.
3. **Greedy assignment of flows to paths**. Place the largest-demand flows first; assign them to paths with available capacity; continue with smaller flows, updating remaining link capacity.

✓ Strengths and ✗ Weaknesses

✓ Strengths

- ✓ **Traffic-aware scheduling**. Unlike ECMP, Hedera looks at actual flow sizes. Elephants are spread across paths, it avoids collisions and hotspots.
- ✓ **Better utilization of network capacity**. Reduces wasted bandwidth and improves aggregate throughput in Fat-Tree/Clos topologies.
- ✓ **Hybrid design (ECMP + centralized scheduling)**
 - * Mice flows: stay on ECMP → simple, fast, scalable.
 - * Elephant flows: centrally scheduled → efficient use of resources.
- ✓ **Proof of concept for SDN in datacenters**. Hedera was one of the **first real SDN applications**. Showed that centralized control could improve load balancing.

✗ Weaknesses

- ✗ **Controller scalability**. Collecting flow statistics and computing assignments for many elephants is computationally heavy. Doesn't scale easily to Hyperscale datacenters with millions of flows.
- ✗ **Reaction time**. Detection of elephants takes time (flows must exceed a threshold). By the time scheduling decisions are made, network conditions may already have changed.
- ✗ **Centralization overhead**. All decisions come from one controller. In large networks, this becomes a bottleneck and a single point of failure.
- ✗ **Limited granularity**. Only elephants are scheduled; mice remain random. If mice collectively create congestion, Hedera doesn't help.

Hedera is an **improvement over ECMP** because it solves elephant collisions with centralized, demand-aware scheduling, which improves throughput and fairness. However, **controller overhead and slow reaction times limit its scalability** in real-world, production-scale data centers. This is why subsequent work (e.g., HULA) shifted toward **in-switch, decentralized, and faster load balancing** that leverages programmable data planes instead of heavy, centralized control.

6.6 HULA: Load Balancing in P4

Hedera required a **centralized controller** to monitor flows and compute paths. This created **scalability issues** (too much data, too slow to react). The HULA idea was: “instead of centralizing everything, can the **network itself** quickly share congestion information?”.

Q The Key Idea of HULA

HULA [7] proposes **summarized state propagation in the data plane**:

- Switches exchange **lightweight summaries** of congestion information.
- Each switch only needs to know the **best next hop** for a given destination **based on congestion**.
- This information is updated hop-by-hop, similar to a distance-vector routing protocol; but instead of distance, it propagates **available bandwidth**.

In other words, switches gossip about **which path currently has the most free capacity**.

Q **What “summarized state” means.** Instead of reporting **every flow** (like Hedera), each switch maintains only a **simple summary**:

- The “best next hop” to reach each destination.
- The “bottleneck bandwidth” available along that path.

When a switch hears an update from a neighbor, it compares bottleneck values and updates its local decision.

Q **Why this works.** No need for a central controller; state is compact, only “best path summaries”, not per-flow details. Also, updates are fast and localized, so switches can quickly adapt to changing congestion.

❖ Core Workflow

HULA uses **summarized state** (best path + available bandwidth) to guide forwarding.

1. **Probing Phase.** Special “probe” packets are sent periodically. Each switch forwards probes toward destinations, updating them with the **minimum available bandwidth** seen along the path (the bottleneck). This way, probes carry information like: “to reach Rack X through me, the bottleneck capacity is 8 Gbps”.
2. **State Propagation.** Neighboring switches receive probes and compare them with their own tables. They update their **next-hop choice** if the new probe advertises a better path (higher available bandwidth).
3. **Forwarding Decisions.** For normal data packets, the switch consults its **HULA table** (map destination to best next-hop). Packets are forwarded along the path with the **highest bottleneck bandwidth**.

Example 3: How this looks in practice

Suppose a switch has 3 possible next hops to reach Rack X.

- Probe from Next-Hop A says: bottleneck = 5 Gbps.
- Probe from Next-Hop B says: bottleneck = 9 Gbps.
- Probe from Next-Hop C says: bottleneck = 2 Gbps.

The switch picks **Next-Hop B** as the forwarding choice for Rack X.

 **Why P4 Matters.** HULA was implemented using P4 on programmable switches (page 43). P4 allows parsing probe headers, updating state tables at line rate, and making per-packet forwarding decisions based on congestion summaries. So HULA shows the power of **programmable data planes**, the logic of a load balancer can run directly inside the switch.

▲ HULA vs. ECMP vs. Hedera**• HULA vs. ECMP**

- **ECMP** (page 91): per-flow hashing, static, oblivious to congestion. Works fine for mice flows, but elephants collide and waste capacity.
- **HULA**: uses congestion-aware summarized state (probes). Always tries to send traffic on the **least congested path**. Adapts quickly if conditions change.

In summary, ECMP is simple but blind. In contrast, HULA is slightly more complex, yet smart and adaptive.

• HULA vs. Hedera

- **Hedera** (page 97): centralized controller monitors flows, detect elephants, reschedules them. Works, but controller overhead and slow reaction make it less practical at hyperscale.
- **HULA**: no central controller, switches propagate state locally via probes. Runs entirely in the data plane (thanks to P4). Much faster reaction, lightweight, scalable.

In summary, Hedera is centralized and heavyweight. In contrast, HULA is distributed and lightweight.

Feature	ECMP	Hedera [1]	HULA [7]
Path selection	Hash-based (static)	Centralized scheduling	Distributed, congestion-aware
TCP friendliness	Yes (per-flow)	Yes (per-flow)	Yes (per-flow)
Congestion awareness	No	Yes (elephants only)	Yes (all flows, summarized state)
Reaction speed	Instant (but static)	Slow (controller polling)	Fast (in-switch updates)
Scalability	High, but inefficient	Limited by controller load	High (distributed)

Table 9: ECMP vs. Hedera vs. HULA.

Scalability

- **Local state only.** Each switch keeps just a small table: “for each destination rack, what’s the best next hop + bottleneck bandwidth”. No need for per-flow global state like Hedera.
- **Constant overhead.** Probes are periodic and lightweight. Overhead doesn’t explode with number of flows, it scales to very large datacenters.
- **Line-rate operation.** Implemented in P4, so congestion-aware forwarding happens directly in hardware pipelines, at full switch speed.

Multi-Tier Applicability

Datacenter fabrics (Clos/Fat-Tree) are **multi-tiered**: edge → aggregation → spine. HULA integrates naturally because each tier just propagates summarized state (best path per destination rack), and the gossip spreads across the whole fabric, hop by hop. For example:

- Edge switch learns from its uplinks which spine has the least congested path to a destination rack.
- Aggregation switches forward probes upward, spines propagate summaries downward.
- Over time, all switches converge on good next-hop decisions.

Benefits at Scale & Limitations

- ✓ **Distributed load balancing** across the entire topology, not just within one tier.
- ✓ **Fast reaction** to shifting elephant flows anywhere in the fabric.
- ✓ **Robustness:** no central bottleneck, even if a switch fails, the gossiping continues with others.
- ✗ Probes give only **approximate congestion info** (summarized state).
- ✗ Works well when congestion is stable or slowly changing, but very short-lived bursts may still slip through.
- ✗ Still per-flow forwarding, so extreme incast patterns (see page 93) at the receiver can’t be “solved” by HULA alone (same as ECMP/Hedera).

So HULA scales well to multi-tier datacenter networks because it uses **local**, **summarized state** and **distributed probe-based updates**. This makes it lightweight, fast, and practical for hyperscale fabrics where Hedera’s centralized controller would collapse.

7 Datacenter Layer 4 Load Balancing

7.1 Introduction

A **Load Balancer (LB)** is a device/system that assigns incoming client requests to different servers. At **Layer 4 (transport layer)**, this decision is made **based on transport-level headers** (IP addresses + TCP/UPD ports). For example, packets belonging to the same TCP connection must go to the same server.

The goal is to:

- **Distribute traffic uniformly** across servers.
- **Preserve connection affinity** (all packets of a connection handled by the same backend).
- **Improve utilization** of compute, network, and storage resources.

In simple words, instead of one server handling all client requests, an L4 load balancer spreads them across many servers while keeping each connection consistent.

❓ Why is it needed in datacenters?

1. **User-facing traffic.** Datacenters host applications accessed by millions of users (websites, APIs, services). Incoming requests first hit a **Virtual IP (VIP)** exposed to the Internet. The load balancer decides which backend server should handle each request. Without load balancing, a single server would be overwhelmed.
2. **Scalability.** Traffic volume is enormous: thousands/millions of requests per second. A load balancer enables **horizontal scaling** (adding more servers behind the VIP). Different approaches (TCP termination, NAT, Direct Server Return) improve scalability for asymmetric or heavy workloads.
3. **Reliability & performance.** If one server fails, the LB can redirect traffic to healthy servers. Balancing traffic avoids hotspots (some servers overloaded while others are idle). Helps achieve predictable performance and meet SLAs⁵ (latency, throughput).

❓ Why do SLAs matter for load balancing? If all requests go to one overloaded server, latency spikes and SLA targets are violated. L4 Load Balancers spread traffic across servers to ensure:

- **No single server becomes a bottleneck.**
- **Latency and availability goals in SLAs are respected.**

⁵SLA stands for **Service Level Agreement**. It's a **formal contract** between a service provider (e.g., a cloud/datacenter operator) and a customer. It defines the **expected level of server** in measurable terms.

SLAs drive the **design of resilient load balancers**: failover, redundancy, uniform distribution.

💡 What is a VIP?

A **Virtual IP** is an **IP address not bound to a single physical server**, but instead represents a **service** (e.g. www.example.com). Clients on the Internet only see the VIP when they connect. The **load balancer owns the VIP** and listens for incoming connections. When traffic arrives at the VIP:

1. The load balancer decides **which backend server** should handle the request.
2. It then forwards the packet/connection to that chosen server.
3. From the client's perspective, it's always talking to the VIP, even though multiple servers are working behind it.

For example 8.8.8.8 is Google's DNS VIP. Millions of users query 8.8.8.8. Behind the scenes, Google's load balancers spread requests across many DNS servers worldwide. Users don't need to know which specific server they hit, they always use the same VIP.

Layer 4 Load Balancing is fundamental in datacenters because it **efficiently distributes massive user traffic** across backend servers, ensures **connection consistency**, and supports **scalability and fault tolerance**. It's a key building block for serving large-scale Internet services.

7.2 Traditional LB Architecture

Remark: OSI model

See Remark in Section 6.2, page 87.

Traditionally, load balancers are placed to different layers of the network stack. The most common stack in datacenters is composed by 3 layers:

- **Application-Level LB (Layer 7)**. It works at the **application layer** (e.g., HTTP, HTTPS). It terminates the client request and inspects the **application data** (like HTTP headers, cookies, or URLs). It then decides which backend server will serve the request. For example, a client sends a GET for a mp4 video; the load balancer parses the HTTP request and forwards it to the most appropriate server (e.g., one with available cache or CPU).

This gives very fine-grained control, but also means the LB must parse and understand application protocols (heavier processing).

- **Transport-Level LB (Layer 4)**. It works at the **transport layer** (TCP/UDP). Looks at **five-tuple headers** (src/dst IP, src/dst port, protocol). Ensures **connection affinity**, all packets from the same TCP connection go to the same server. It achieves a balance between **efficiency** and **correctness** (avoid reordering, preserve session state). For example, a client opens a TCP connection to VIP; the L4 load balancer decides “all packets from this TCP connection go to server X”.

This is the sweet spot for datacenter load balancing: scalable, connection-aware, and lightweight compared to L7.

- **Network-Level LB (Layer 3)**. It works at the **IP layer**. Balances traffic based on **IP packets** without looking deeper. The common approach is **ECMP (Equal-Cost Multi-Path)**, hash on IP headers to spread flows across multiple servers (page 91). For example, a client sends traffic to a VIP (Virtual IP); the network load balancer forwards packets of each flow to one backend server’s real IP.

Simpler and more scalable than application Load Balancers, but less flexibility since it doesn’t inspect application data.

Putting it together, we get a Multi-Layer Architecture. In a traditional Cloud Load Balancing design we have (when a client sends a request):

1. **Internet**, traffic enters the datacenter through a **VIP**.
2. **L3 Load Balancer (Network)**, decides which *rack or server group* should handle the packets (IP-based).
3. **L4 Load Balancer (Transport)**, ensures all packets of the same TCP / UDP flow are forwarded to the same server.
4. **L7 Load Balancer (Application)**, inside the server cluster, may further dispatch the request to the correct **application instance** (e.g., web server, cache, microservice).

5. **Servers**, the final application processes the request.

L3 comes first because it's the coarsest, simplest routing decision (based on IP). **L4 refines** by handling connection affinity (ensuring one TCP flow doesn't get split). **L7 is last** because it requires deep packet inspection (HTTP headers, cookies, etc.), which is expensive and usually one done closer to the server.

Example 1: Airport Travel Analogy

Analogy: Airport Passenger Flow as Datacenter Load Balancing.

- **Layer 3 - Network LB (IP Packets) → Terminals.** Imagine the airport has **several terminals**. When a passenger (packet) arrives, the airport system decides *which terminal* they should enter. The decision is simple because it's based on flight destination (like ECMP based on IP hash). The system **spreads the crowd evenly**, but doesn't know anything about the passenger's ticket details (application semantics). So, the role of this LB is coarse-grained distribution.
✗ Limitation: Passengers belonging to the same group could be split into different terminals (just like flows that get split badly).
- **Layer 4 - Transport LB (TCP connections) → Gates.** Inside each terminal, the airport must assign passengers to the **correct gate**. All passengers on the *same flight* (same TCP connection/flow) must go to the **same gate**, otherwise, the flight won't depart correctly. The assignment is more precise than at the terminal level and keeps groups (flows) consistent. So, the role of this LB is ensures all packets of the same TCP connection go to the same server.
✓ Benefit: Prevents packet reordering (like ensuring a family travels together).
- **Layer 7 - Application LB (HTTP requests) → Seats.** Once at the gate, each passenger is directed to their **specific seat** in the airplane. The decision depends on details like *ticket class*, *row number*, or *meal preference* (HTTP headers, cookies, URLs). This level understands the **application semantics** and makes the **most fine-grained decisions**. So, the role of this LB is directs specific requests within the application.
⚠ Trade-off: Very smart but resource-heavy (the LB has to "look into the ticket" for each passenger).

The combination ensures the airport (datacenter) runs **efficiently, fairly, and predictably**, even with millions of passengers (packets) arriving per day.

7.3 Real-World Deployments

In this section, we explore real-world deployments of Layer 4 load balancing in datacenters. We will discuss how major companies (Meta, Alibaba, Microsoft, Google) implement L4 load balancing to enhance the performance and reliability of their services. Although the principles are the same (scaling traffic, connection affinity, high availability), their **architectural choices** differ in how packets are handled and returned.

Meta (Facebook)

Meta uses **multi-tier L4 load balancers** to manage traffic. The architecture consists of:

- **Edge L4 Load Balancers:** near the Internet handle (north-south) traffic.
- **Internal L4 Load Balancers:** handle east-west traffic between services inside the datacenter.

Their system emphasizes **scalability** (tens of millions of concurrent TCP connections) and **low tail latency** (minimizing packet processing delay). Meta architectures is based on **Direct Server Return (DSR)** for efficient packet handling: the load balancer forwards incoming packets to the backend server, which then sends the response directly to the client, bypassing the load balancer on the return path. This allows the LB to handle only *incoming* packets, multiplying throughput capacity.

Definition 1: Direct Server Return (DSR)

Direct Server Return (DSR) is a load balancing technique where the load balancer forwards incoming requests to backend servers, but the responses are sent directly from the backend servers to the clients, bypassing the load balancer on the return path. This approach reduces the load on the load balancer and can improve response times.

Alibaba

Alibaba Cloud employs **hierarchical L4 load balancer** built on programmable switches⁶ and smart NICs⁷. Alibaba's architecture often uses **NAT-style translation**.

⁶Programmable Switches are network devices that can be programmed to perform custom packet processing tasks, allowing for more flexible and efficient handling of network traffic (page 39). An example of a programmable switch is one that supports the P4 programming language (page 43), which enables users to define how packets are processed and routed within the switch.

⁷Smart NICs (Network Interface Cards) are advanced network interface cards that offload processing tasks from the CPU, such as packet filtering, load balancing, and encryption, to improve overall system performance. They often include programmable processors and memory to handle complex networking functions directly on the NIC. They differ from traditional NICs, which primarily handle basic data transmission and reception without additional processing capabilities.

NAT (Network Address Translation) is a mechanism that lets a device (like a router or load balancer) **rewrite the source or destination IP address and port** of packets as they pass through. We can think of it as the “middleman” that changes addresses so packets can move between different network zones. In a datacenter, the **load balancer acts as the NAT device**. When a client connects to the **Virtual IP (VIP)** of a service:

1. The **load balancer receives the packet** with destination set to the VIP.
2. It **rewrites the destination IP and/or port** to point to one of the backend servers (e.g., 10.0.1.5).
3. It **remembers the mapping** between the original connection (client IP and port, VIP) and the chosen backend.
4. When the server replies, the LB reverses the translation, changing the source back to the VIP before sending the packet back to the client.

❓ **Why Alibaba’s technique stand out?** Alibaba’s strength isn’t just “*using NAT*”, it’s **how they scale it**. They use **programmable switches** to handle the NAT translations at line rate, allowing them to manage millions of connections efficiently. The switches can perform the address rewriting directly in hardware, which is much faster than doing it in software on a traditional server-based load balancer. Furthermore, the **hierarchical design** allows them to distribute the load across multiple layers of switches:

- **Edge / Border Switches:** the first programmable switches that packets hit after entering the datacenter from the Internet. They handle **VIP addressing**, perform **DNAT** (Destination NAT, map the VIP to a backend servers’s internal IP), and optionally **SNAT** on replies (Source NAT, rewrite server’s source IP to VIP). These edge switches are typically **programmable ASICs** (like Barefoot Tofino) running **P4 pipelines** that apply NAT rules.
 - ✓ Programmable hardware ensures **line-rate performance** (Terabits per second throughput with microsecond latency).
- **Aggregation / Spine Switches:** sometimes, Alibaba distributes part of the load-balancing logic here for **scalability and redundancy**. These switches **don’t rewrite IPs** but may decide *which edge switch* performs the NAT for a given VIP.
 - ✓ Distributing logic across the spine layer avoids bottlenecks and allows fast failover.
- **Top-of-Rack (ToR) switches or SmartNICs (optional):** for internal east-west traffic, NAT or load-balancing functions might also run **closer to servers**. Some deployments offload L4 NAT to **programmable NICs** (SmartNICs or DPUs). The NIC performs per-flow NAT or flow steering to local microservices.
 - ✓ Improves **locality** and offloads the central fabric for internal traffic between tenants or containers.

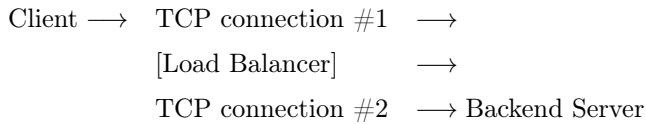
■ Microsoft (Azure)

Azure's technique stands out **because it takes a very different design philosophy** compared to Alibaba, Meta, or Google. While Alibaba and Meta focus on **scaling L4 load balancing in hardware**, **Microsoft prioritizes control, observability, and integration** with the cloud platform, even at the cost of higher per-connection overhead.

Microsoft's Azure load balancers, particularly **Ananta** [11] and its successors, are based on **TCP termination**. That means the **load balancer actually participates in the TCP connection** instead of simply forwarding packets. With “*TCP termination*”, we mean that when a client connects to a service VIP:

1. The **load balancer itself accepts the TCP connection** from the client (acting as the server).
2. Then, it **creates a new TCP connection** from itself to the chosen backend VM (acting as the client).
3. It **relays data** between these two independent connections.

Essentially, the LB “splits” the connection into two halves:



This design gives Azure **full control over every connection**, which brings several major advantages:

- **Fine-grained control and observability.** Since the LB manages both TCP endpoints, it can:
 - Apply **custom congestion control or rate limiting** per connection.
 - Collect **precise telemetry** (latency, throughput, retransmissions, etc.).
 - Perform **real-time health checks** of backend VMs.
 - Easily integrate with monitoring systems and Service Level Agreements (SLAs).

Other architectures (like DSR or NAT) can't easily measure latency or detect slow clients, because replies bypass them or just transit statelessly.

- **Simplified policy enforcement.** With full TCP state:
 - The LB can apply **firewall rules, TLS termination, or security policies** directly.
 - It can perform **connection-based authentication, DDoS protection, or application-layer filtering**.

This makes it ideal for a multi-tenant cloud like Azure, where **isolation and security** are top priorities.

- **Fault tolerance and VM mobility.** Because the LB terminates connections, it can seamlessly:
 - Migrate backend VMs or reroute traffic without breaking client sessions.
 - If a backend VM crashes, the LB can retransmit or buffer data instead of immediately closing the client connection.
 - This ensures connection continuity during failures or scaling events.
- **Software-defined integration.** Azure's L4 LB is part of a software-defined network (SDN) stack:
 - It's implemented in virtualized routers and software agents running across Azure hosts.
 - The system can scale horizontally, multiple LB instances share the same VIP pool, coordinated by a central SDN controller.
 - The architecture is **cloud-native**: it integrates with virtual networks (VNets), security groups, and VM scaling.

Azure deliberately accepts the performance overhead of maintaining per-connection state in exchange for **fine-grained connection management, security, and SLA compliance**, critical in a public cloud. This technique stands out because it **prioritizes control and observability over raw scalability**, integrating the load balancer tightly into the software-defined cloud fabric.

G Google

Google's design is widely studied because it represents one of the **cleanest, most elegant, and most scalable software-defined L4 load balancing architectures ever built: the Maglev system** (published at NSDI 2016) [3].

Maglev is Google's **L4 load balancer**, used for both:

- **External traffic** (from users on the Internet to Google services like Search, Gmail, YouTube).
- **Internal traffic** (service-to-service communication inside Google's datacenters).

It's a **software-based, distributed, stateless load balancer** (data-plane stateless but control-plane stateful) running on commodity servers, yet it handles **millions of requests per second** with near-zero downtime. Maglev's architecture stands out because it combines several key principles:

- **Stateless design.** Unlike Azure's stateful approach, **Maglev is stateless**:
 - It doesn't keep per-flow connection tables.
 - Any Maglev instance can process any packet of any connection.

How? Because it uses **deterministic consistent hashing** to map flows to backends. Each Maglev node computes the same hash:

$$\text{server} = \text{hash}(\text{src_ip}, \text{src_port}, \text{dst_ip}, \text{dst_port}, \text{protocol}) \mod N$$

Where N is the number of backend servers, fixed across all Maglev nodes. This ensures **consistent flow affinity** without storing any state. So, even if packets of the same TCP connection hit different Maglev nodes, they'll all pick the **same backend server** deterministically. The result: no flow state to replicate or synchronize between load balancers, then huge scalability.

- **Distributed horizontally.** Instead of one centralized LB, Google deploys **hundreds of Maglev instances** across datacenters. They share the same **VIP configuration**. The **Anycast routing** mechanism ensures that packets to a VIP are automatically distributed among available Maglev nodes (closest or least loaded). If one **Maglev node fails**, **traffic automatically reroutes to others**, so no DNS change, no service interruption.
- **Software-based on commodity servers.** Maglev runs on standard Linux servers, not on custom ASICs or SmartNICs. This allows rapid updates, integration with Google's SDN control plane, and uniform deployment across global regions. **High elasticity:** spin up or remove Maglev instances dynamically as load changes.
- **Direct Server Return (DSR).** Like Meta, Google uses **DSR (Direct Server Return)**:

- Maglev handles **incoming** traffic only.
- The **reply traffic** (from backend servers to clients) **bypasses the LB** and goes directly out to the Internet.

✓ **Benefit:** Doubles throughput capacity because LBs don't handle response packets.

⚠ **Trade-off:** Harder to monitor replies and handle asymmetric routing, but Google solves this with deep in-network telemetry and strict network control.

➡ A **typical request flow** looks like this:

1. A user on the Internet sends a request to a Google service (e.g., Gmail) using a **VIP (Virtual IP)** address.
2. The request hits the **nearest Maglev instance** (via Anycast routing).
3. The Maglev node computes the hash of the packet's five-tuple and selects a backend server.
4. It **forwards the packet** to the chosen backend server.
5. The backend server processes the request and **sends the response directly** to the user, bypassing Maglev.

✓ Google solved two of the hardest problems in distributed L4 load balancing:

1. **Consistency without state replication**, all nodes make identical decisions locally.
2. **Scalability without single points of failure**, we can add/remove Maglev instances instantly.

Essentially, Maglev turned load balancing from a **hardware bottleneck** into a **stateless, horizontally-scalable software service**.

💡 Key Takeaways

All big players (Meta, Alibaba, Microsoft, Google) rely on **L4 load balancing** as a cornerstone on their datacenter edge. They differ in **where they keep state** and **how they return packets**:

- **Meta and Google** use **Direct Server Return (DSR)**, maximizing throughput by letting backend servers reply directly to clients.
- **Alibaba** employs **NAT-style translation** using **programmable switches** and **SmartNICs** to handle millions of connections at line rate.
- **Microsoft Azure** takes a different approach with **TCP termination**, where the LB fully manages each connection.

Each architecture reflects different priorities: **scalability and performance** (Meta, Google), **hardware efficiency** (Alibaba), or **control and security** (Microsoft). Understanding these trade-offs helps us appreciate the complexity and innovation behind modern datacenter load balancing.

7.4 Design Space

At this point, we know *what* a L4 load balancer does, they distribute traffic across many backend servers while keeping per-connection consistency.

Now we ask the deeper question: *How can we design an L4 load balancer that is efficient, fair, scalable, and persistent?*

- **Efficiency:** handle packets at line rate with minimal latency and CPU cost.
 - ✓ **Why it matters:** The LB must not become a bottleneck for millions of concurrent flows.
- **Fairness:** spread load evenly across servers (avoid hotspots).
 - ✓ **Why it matters:** Prevents some backends from overloading while others sit idle.
- **Scalability:** support a huge number of servers and flows.
 - ✓ **Why it matters:** Enables datacenter-scale operation (10^5 - 10^6 servers).
- **Persistence (Connection Affinity):** packets of the same flow always go to the same backend.
 - ✓ **Why it matters:** Ensures TCP correctness, no packet reordering, no broken sessions.

In short, we want the LB to be *fast, fair, scalable, and consistent*.

⚠ Load Balancing Policies

L4 load balancers can make decisions in different ways, depending on how much state they maintain and what information they use. Some common policies include:

- **Round-Robin (stateful counter):** Maintain a counter i that cycles through the server list. For each **new connection**, choose backend = `server[i]`, then increment i (modulo number of servers).
 - ✓ **Pros:** Simple, easy to implement, ensures even distribution if all connections are similar.
 - ✗ **Cons:** Stateless per-packet, breaks connection affinity if applied to every packet. Also, doesn't consider load variation (some servers may be slow or busy).

This is **good for flow-level dispatch, bad for packet-level dispatch** (because packets of one connection could go to different servers).

- **Hash-based (stateless deterministic):** Compute a **hash** on packet header fields, typically the 5-tuple:

```
h = hash(src_IP, src_port, dst_IP, dst_port, protocol)
backend = h mod N (where N is number of servers)
```

Each packet of a given TCP flow produces the same hash, always same backend.

- ✓ **Pros:** Stateless but preserves connection affinity. Easy to scale across multiple Load Balancers (deterministic decision).
- ✗ **Cons:** Not perfectly fair, hash imbalance can cause some servers to get more flows. When number of backends changes, many flows per remapped (hash churn).
- ❓ **What is hash churn?** When a server is added or removed, the modulo operation changes, causing many existing flows to be remapped to different backends, breaking connection affinity.

This is the method used in **Google's Maglev** and **ECMP routing**.

- **Stateful per-flow mapping:** Maintain an explicit mapping table:

(5-tuple) → backend server

The first packet of a new connection triggers a decision (e.g., round-robin or weighted), and all subsequent packets look up this table. It is a sort of **flow cache**.

- ✓ **Pros:** Perfect persistency and flexibility.
- ✗ **Cons:** Needs huge memory for millions of flows (e.g., NAT tables). Hard to replicate state across multiple LBs. Potential bottleneck if table lookups are slow.

Used by **NAT-based systems** like **Alibaba's** or **Ananta (Azure)**.

❖ Persistence and Connection Affinity

When we connect to a website, for example:

Client → VIP = 203.0.113.10 (www.example.com)

Behind that VIP there might be **hundreds of backend servers**:

10.0.1.1, 10.0.1.2, 10.0.1.3, ...

A **Layer 4 Load Balancer** must decide *which backend* will handle our connection. This decision must be taken carefully, because datacenters are large and dynamic: many load balancers instances (distributed LBs), servers added/removed frequently, VIPs shared across multiple racks. So packets from the same TCP connection may reach **different LBs**. If each LB picks a random backend, we break connection affinity. We need to ensure two properties:

- **Connection Affinity (Local Property):** All packets of the same connection (flow) must go to the same backend server. It is mandatory for TCP correctness, otherwise connections would fail.

Example 2: Connection Affinity

Client opens TCP connection. LB decides “this flow goes to 10.0.1.3 backend”. Every packet in that connection must go to

10.0.1.3 or else the TCP state breaks.

- **Persistency (Global Property)**: The same connection (or flow) is always mapped to the same backend, even if other conditions change (e.g., another LB instance, restart, or network rehash). It ensures that the **same flow always maps to the same backend**, regardless of which LB instance handles it, temporary restarts, scaling events, hash table updates, etc.

Example 3: Persistency

We have two load balancer instances: LB_1 and LB_2 . Due to ECMP routing, different packets of the same TCP flow might hit different LBs:

- LB_1 sees the first SYN packet, and chooses backend 10.0.1.3.
- LB_2 later receives data packets.

To preserve **persistency**, LB_2 must make **the same decision** (10.0.1.3), even though it didn't see the initial SYN packet. So, persistency means determinism across time and devices.

If we have **persistency**, then automatically each connection is mapped deterministically to the same backend, so **connection affinity** is also satisfied. But we can have **affinity without persistency** if only one LB handles the flow (it keeps packets consistent *locally*, but another LB might choose differently).

Persistency \implies Connection Affinity

Deepening: Formalizing Connection Affinity and Persistency

Let F be the **set of flows**, B the **set of backends**, and L the **set of load-balancer instances**. For each $\ell \in L$ define a mapping:

$$f_\ell : F \rightarrow B$$

Which assigns each flow to a backend when processed by instance ℓ .

Connection Affinity (*local* to instance ℓ) means that for **any flow** $\phi \in F$ all packets p of ϕ (flow) **seen** by ℓ (load-balancer) **map to the same backend**:

$$\begin{aligned} \forall \phi \in F : \forall p_1, p_2 \in \text{Packets}(\phi) \cap \text{SeenBy}(\ell), \\ \text{decision}_\ell(p_1) = \text{decision}_\ell(p_2) = f_\ell(\phi) \end{aligned}$$

Persistency (*global*) requires the **mapping to be independent of the LB instance**:

$$\forall \ell_1, \ell_2 \in L, \forall \phi \in F : f_{\ell_1}(\phi) = f_{\ell_2}(\phi)$$

Hence Persistency implies Connection Affinity, since if all f_ℓ are identical then each ℓ trivially maps every flow consistently. Affinity without

persistency corresponds to the case:

$$\exists \ell_1 \neq \ell_2, \exists \phi \in F : f_{\ell_1}(\phi) \neq f_{\ell_2}(\phi)$$

While each individual f_ℓ still satisfies the local affinity property above.

Example 4: Analogy

Think of a **hotel**:

- **Connection Affinity:** Once a guest (flow) checks into room 312, they keep the same room for their stay (same server for all packets).
- **Persistency:** If the hotel has multiple front desks (LBs), no matter which desk the guest checks in at, they should always be assigned room 312 (same server across LBs).

💡 How can we guarantee persistence and connection affinity?

There are two main approaches:

- **Per-flow state (stateful).** On the first packet (SYN), the LB chooses a backend (e.g., using round-robin, weighted). It stores a mapping in a table:

$$(\text{src_IP}, \text{src_port}, \text{dst_IP}, \text{dst_port}, \text{protocol}) \rightarrow \text{backend_id}$$

Every subsequent packet is looked up in that table.

- ✓ Perfect affinity and persistency.
- ✗ Heavy memory usage and synchronization if multiple LBs exist.

Used by **NAT-based** and **TCP-terminating** systems (Alibaba, Azure).

⌚ Is this approach the same as the stateful, per-flow mapping policy? They are **related**, but not the same thing. A **policy** tells us *what choice to make*; a **mapping** tells us *how to remember or reapply that choice*. A **load balancing policy** decides *which backend to pick* for a new flow. A **stateful mapping** remembers that decision to enforce *connection affinity and persistency* across packets.

- **Deterministic function (stateless).** The LB computes:

$$\text{backend} = \text{hash}(\text{5-tuple}) \mod N$$

Where N is the number of backends (servers). Every packet of that connection hashes to the same backend automatically.

- ✓ No state needed.
- ✓ Works across multiple LBs (same hash function, all get same result).

- ✗ Slight imbalance if hash is uneven.
- ✗ When N changes, many flows are remapped (unless consistent hashing is used).

Used by **Google's Maglev**, **ECMP routing**, and **DSR-based systems**.

⚠ Persistence with cluster changes

Even if the load balancer guarantees persistence and connection affinity for ongoing flows, **changes in the cluster**, such as adding or removing servers, can break these guarantees. If we use a simple **hash-based deterministic function** like:

$$\text{backend} = \text{hash(5-tuple)} \mod N$$

Then every time N (the number of servers) changes, the modulo result changes for *all* flows. ⚡ This causes **hash churn**, where many **ongoing flows are suddenly remapped to different backends**, breaking persistence. ✓ To avoid this, we use **consistent hashing**:

- It changes the mapping of only a small subset of flows when servers are added or removed.
- Existing flows stay mapped to their original backends.
- This preserves **persistence** even in a **dynamic cluster**.

Consistent hashing provides *stable, persistent mappings* across server pool changes, avoiding massive remapping of connections.

❖ Consistent Hashing: Deep Dive

We already saw that a **naive hash function** like:

$$\text{backend} = \text{hash(5-tuple)} \mod N$$

Breaks persistence when the number of servers N changes. If one server is added or removed, the modulo value changes for almost **all** hashes. Every existing connection might be remapped to a new backend. This causes **connection churn**, destroying persistency for active flows.

Consistent Hashing is a technique that minimizes remapping when the set of servers changes. It aims to make the system **resilient to membership changes**: when servers are added or removed, **only a small subset of flows** are remapped to different backends, while all others stay on the same one. Instead of directly tying the hash function to the **number of servers** N , we tie it to a **continuous identifier space**: a “hash ring”.

❷ What is a hash ring?

1. Imagine a circular space (0 to $2^{32} - 1$ for a 32-bit hash), representing all possible hash values.

2. Each **server** is assigned one or more positions on this ring, using a hash of its ID or IP address.
3. Each **flow (connection)** also hashes to a point on the same ring.

Now, to find the backend for a flow:

- Move **clockwise** on the ring until we find the first server.
- That server handles the flow.

This guarantees that each server owns a “slice” of the ring (roughly uniform if servers are many). When a server joins or leaves, only its adjacent slice changes, then only flows in that region get remapped.

Example 5: How we find a backend (the rule)

Suppose we have 3 servers: **S1**, **S2**, **S3**. We have a circular space of all possible hash values (0 to 99). Each server is assigned a position on the ring by hashing its ID:

- **S1** → 10
- **S2** → 50
- **S3** → 80

Now, each **flow** also hashes to a number between 0 and 99. For example:

- Flow **F1** hashes to 5 → assigned to backend **S1** (first server clockwise, value 10).
- Flow **F2** hashes to 47 → assigned to backend **S2** (first server clockwise, value 50).
- Flow **F3** hashes to 70 → assigned to backend **S3** (first server clockwise, value 80).
- Flow **F4** hashes to 90 → assigned to backend **S1** (wraps around, first server clockwise is 10).

So, each server handles the interval between its predecessor and itself on the ring:

- **S1** handles [81-10]
- **S2** handles [11-50]
- **S3** handles [51-80]

Example 6: What happens when a server joins or leaves?

Let's add a new server, **S4** at position 60:

- **S1** at 10
- **S2** at 50

- S4 at 60 (new)
- S3 at 80

Now only **flows that hash between 50 and 60** move, those that previously belonged to S3 now go to S4. **All other flows stay exactly where they were.** That's the magic of consistent hashing: the mapping is *consistent* before and after changes, only flows in the new server's interval are remapped.

Why it's “fair” and scalable

If we place servers randomly on the ring, each gets roughly the same share of the space (and thus same amount of traffic). **With many servers, this becomes statically uniform.** To avoid unevenness, we **add virtual nodes**: each physical server is assigned multiple positions on the ring. This smooths out distribution, making it more uniform. For example:

- S1 → 10, 30, 70
- S2 → 50, 90
- S3 → 20, 60, 80

This way, each server gets multiple slices of the ring, balancing load better. Consistent hashing scales well: adding or removing servers only affects a small portion of the ring, so most flows remain stable. It works efficiently even with thousands of servers and millions of flows.

Summary

- L4 load balancers must be **efficient, fair, scalable**, and **persistent**.
- Common **load balancing policies** include round-robin, hash-based, and stateful per-flow mapping.
- **Connection affinity** ensures all **packets of a flow go to the same backend**; **persistency** ensures the **same flow always maps to the same backend**, even across LBs and changes.
- **Consistent hashing** provides a way to **maintain persistency** even when servers are added or removed, minimizing remapping of existing flows.
- By **using a hash ring and virtual nodes**, consistent hashing achieves fair load distribution and scalability.

7.5 Cheetah (Research Proposal)

In the previous topic (page 114), we studied **consistent hashing** and **uniform deterministic load-balancing functions**, which help maintain *persistence* and *connection affinity* when the backend pool changes (servers added/removed). However, **these approaches still require the load balancer to store per-flow state**, especially to ensure persistence for *in-progress* connections.

⚠ The **problem** is that maintaining this **stateful mapping table** (flow → backend) becomes expensive:

- Modern data centers can have **millions of concurrent connections**.
- Each load balancer must handle **tens of millions of packets per second**.
- Replicating or synchronizing this per-flow state across redundant load balancers (for failover or scaling) causes **huge memory and consistency overheads**.

This leads to the **key question** that motivates Cheetah: “*How can we maintain connection persistence and high performance without keeping large per-flow state at the load balancer?*”

② Problem Context

Traditional **stateful L4 load balancers** (e.g., IPVS, Maglev, Ananta) keep a mapping from client connection tuples (5-tuple) to a backend server.

- ✓ **Pros:** Ensure persistency and connection affinity.
- ✗ **Cons:** Large memory footprint, synchronization overhead, and limited scalability.

To address this, some systems use **stateless load balancing** techniques, like **hash-based load balancing**, which computes the backend server for each new connection using a hash function on the connection tuple.

- ✓ **Pros:** Scalable and lightweight, no per-flow state.
- ✗ **Cons:** Cannot ensure persistency when backend membership changes (ongoing flows might break).

Hence, there's a **trade-off** between:

- **Stateful designs:** strong persistency, weak scalability.
- **Stateless designs:** high scalability, but poor persistency on server pool changes.

③ But isn't Maglev already stateless? Yes, but only in the data plane. Maglev is **stateless with respect to connections**, but **not stateless with respect to configuration**. Maglev does **not** store per-flow or per-connection state: no 5-tuple to backend table, nor TCP state tracking. However, Maglev

does maintain state in the control plane, namely the consistent-hashing lookup table and the set of active backends. This state is global (shared by all LBs), static during normal operation and updated only on backend set changes. So “stateless” means **no per-connection state in the data plane**, but Maglev still relies on **global configuration state** to ensure persistency. Cheetah aims to go further by eliminating even this global state dependency, achieving true statelessness while still ensuring per-connection consistency.

⌚ Stateless Cheetah’s Key Idea

Cheetah is a *research prototype* of a **Layer-4 load balancer** proposed at NSDI 2020 [2]. It was developed by researchers from **University of Washington**, **Google**, and **MIT**, as part of the effort to improve scalability and performance of datacenter load balancing. Cheetah’s main goal is to **reduce or eliminate per-flow state** at the load balancer while still ensuring *connection persistency, high performance, and fine-grained load distribution*.

Cheetah proposes a **hybrid design** that:

- Minimizes state at the load balancer (similar to stateless hashing).
- Still **preserves connection persistency**, even if the backend set changes.
- Achieves **fine-grained load balancing** (through flowlet switching).

This makes it a bridge between:

- The **consistent hashing** ideas from the previous section (stateless persistency);
- And **flowlet-based dynamic adaptation** to real-time network conditions (performance and fairness).

So, Cheetah’s goal is to **guarantee per-connection consistency (PCC)**, that is, packets from the same TCP connection always reach the same backend, **without keeping per-flow state** at the load balancer. It achieves this through a **stateless encoding mechanism** and, optionally, a **stateful extension** for advanced visibility.

❖ Core Design Idea

Move the connection-to-server mapping from the load balancer’s memory into the packet itself. Each packet carries a small **cookie** ($\log_2 k$ bits, where k is the number of servers) that encodes the chosen backend server in an **opaque and secure way**. This lets every subsequent packet carry all information needed for correct routing, so the load balancer no longer needs to remember each mapping.

1. **Overview.** The LB stores two small static tables:

- **AllServers table:** maps *server ID*⁸ \rightarrow *DIP (Direct IP)*⁹.

Server ID	DIP (Direct IP)
0	10.0.0.1
1	10.0.0.2
2	10.0.0.3

Table 10: Naïve example of **AllServers** table with 3 backend servers.

VIP (Virtual IP)	Service	Active Servers (Server IDs)
192.0.2.10	api.service-A	{0, 1, 2, 3}
192.0.2.20	web.frontend-B	{4, 5, 6}
192.0.2.30	auth.service-C	{7, 8}
192.0.2.40	db.cache-D	{9, 10, 11, 12, 13}

Table 11: Naïve example of **VIPToServers** table with 4 VIPs, each mapped to a set of backend servers.

- **VIPToServers table:** maps VIP (*Virtual IP*) \rightarrow *set of active servers (Server IDs)*.

When the **first packet of a new connection** arrives at the load balancer:

- The LB looks up the servers for that VIP in the *VIPToServers* table.
- It chooses one backed using **any load balancing policy** (e.g., round-robin, hash-based, page 114).
- The packet is forwarded to the chosen backend server.
- It creates a **cookie** that encodes the chosen server ID in a secure way (explained below).
- The LB **adds this cookie to the packet header** (e.g., as a TCP timestamp or in a custom header).

2. **Cookie Creation.** The cookie is a small piece of data (a few bits) that **encodes the chosen backend server ID**. This cookie is **opaque** to both the client and the backend server, meaning they cannot interpret or modify it. Only the load balancer can decode it, using a **secret hash function**.

When the **first packet** of a new connection is processed by the backend server and the answer is sent back to the LB, the server **creates a cookie** that encodes the chosen server ID in a secure way. The **cookie is added**

⁸The **Server ID** is an **internal identifier** used inside the load balancer. It's not an **IP address**, but simply an **index** or **integer label** that uniquely represents a backend machine. Its purpose is to **compactly identify** each backend server in the load balancer's logic and data structures; small enough to fit efficiently in the per-packet cookie.

⁹**DIP** stands for **Direct IP address**, i.e., the **real IP address of a backend server** inside the datacenter. Each backend machine hosting a server has a unique DIP, but clients never see these DIPs directly; they connect to a VIP (Virtual IP) that represents the service. The LB's job is to map the VIP to one of the backend DIPs.

to the packet header (e.g., as a TCP timestamp or in a custom header) before sending the response back to the client.

The cookie is computed as follows:

$$\text{cookie} = \text{hashS}(\text{connID}) \oplus \text{serverID}$$

Where:

- **connID** is a **unique identifier for the connection** (e.g., 5-tuple).
- **hashS** is a **secure hash** function with a **secret key known only to the LB**.

Deepening: Secure Hash Function

The secure hash function **hashS** ensures that an attacker cannot easily guess or forge valid cookies without knowing the secret key.

❓ **What is a “salt”?** In cryptographic, a “salt” is a random value added to the input of a hash function to ensure that the output (hash) is unique, even for identical inputs. This prevents attackers from using precomputed tables (rainbow tables) to reverse-engineer the hash values.

❓ **Who knows the secret salt “S” (the S in hashes)?** The **secret salt “S”** is known **only to the load balancers**, not to the servers or clients. It is a random value and prevents **clients** from reverse-engineering which backend a connection maps to. It also prevents **malicious users** from targeting a specific server by crafting cookies. So, **only the load balancer** can compute and decode the cookie because it knows both: the secret salt “S” and, the mapping from hash function output to server ID.

- **serverID** is the **ID** of the chosen **backend** server.
- \oplus is the bitwise XOR operation.

When the server receives the packet, it simply ignores the cookie (it doesn’t need to know which server it is). It runs its application logic and sends responses back to the LB. So the **server doesn’t need to decrypt anything**. It just **echoes the same cookie** it received from the client (LB).

3. **Subsequent Packets.** Every following packet of that connection carries the cookie. The LB decodes it, extracts the server ID, and forwards the packet directly; no lookup in per-flow state. The **static tables never change per-connection**, ensuring **persistency** even if VIP-to-server membership changes.

❓ If Cheetah is stateless, how can it decrypt the cookie on every packet without keeping any state or key?

The salt “S” is a **global cryptographic-style secret** shared by all Cheetah load balancers. It isn’t a per-flow key, it’s more like a *seed* for a deterministic pseudorandom mapping function. We can think of it as a 128 or 256-bit random value, generated once and kept secret. So Cheetah does **not store a “per-connection secret”**, it stores **one constant salt “S”** in memory (a few bytes). Each packet carries all the rest of the information needed to restore the mapping. It acts like a “*master key*” shared by all Cheetah load balancers, stored in all LBs in the cluster. Because it’s constant and known only to the LBs, it allows **any LB to decode any packet** just by recomputing a hash.

Property	Description
Per-Connection-Consistency (PCC)	Guaranteed, since the cookie uniquely identifies the backend.
Statelessness	No per-flow memory → scalable.
Flexibility	Supports any LB algorithm, not just hashing.
Security / resilience	Salted cookie prevents clients from targeting a server.
Overhead	Cookie size grows $\approx \log_2 k$ bits (small).

Table 12: Properties of Cheetah’s design.

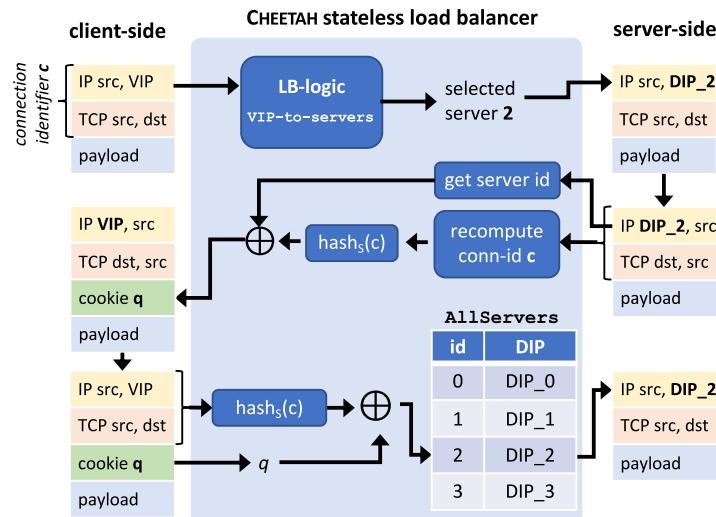


Figure 6: The diagram [2] shows the **complete life-cycle of a connection** through a *stateless Cheetah* load balancer. It illustrates **how per-connection consistency (PCC)** is guaranteed *without* keeping any per-flow state in memory.

⌚ Cheetah Variant: Stateful Version

While **stateless Cheetah** is elegant and fast, some datacenter applications still need **per-connection visibility** for: traffic shaping (rate limiting, NATs), DDoS filtering, per-flow monitoring, selective rerouting of heavy hitters, or other middlebox functions. These features *require* the load balancer to know something about each active connection. So Cheetah extends its design with a **lightweight per-connection table**, but keeps the cookie logic to guarantee PCC.

☒ Core Idea - Index-based connection tables

Traditional stateful LBs (like Maglev or Ananta) use **cuckoo-hash tables** to store the mapping:

$$\text{connID} \rightarrow \text{server}$$

Fast lookups, but large memory footprint (millions of entries) and high synchronization overhead (for redundancy). So, **Stateful Cheetah** replaces this with a **stack-based table architecture** that's *constant-time* even in hardware:

- ConnTable

- ⌚ **What it stores.** Per-connection entries: info about a specific connection (its hash and its server's IP/DIP).
- ⌚ **How it's used.** Directly indexed using the cookie number carried in each packet.

Index	Connection Hash	Backend DIP
0	0xA1B2C3D4	10.0.0.1
1	0xB2C3D4E5	10.0.0.2
2	0xC3D4E5F6	10.0.0.3
3	0xD4E5F607	10.0.0.4
4	-free-	-free-
5	-free-	-free-
6	-free-	-free-

Table 13: Example of ConnTable with 4 active connections. Each entry stores the connection's hash and the backend server's DIP.

- ConnStack

- ⌚ **What it stores.** A simple stack (list) of free positions (indices) in the ConnTable. For example, taking the previous table, the stack would contain:

$$\text{Free Indices} = [6, 5, 4]$$

- ⌚ **How it's used.** When a new connection arrives, the LB “*pushes*” one free slot from this stack.

- **Cookie**

?

What it stores. A small number inserted into the packet header, representing the `ConnTable` index. For example, if the cookie is 2, it means the packet belongs to the connection stored in `ConnTable[2]`, in our case the one with hash 0xC3D4E5F6 and DIP 10.0.0.3.

?

How it's used. Tells the LB exactly *which ConnTable entry* to look at for this connection.

So instead of hashing to *find* the table entry, each packet *tells* the LB its exact slot index via the cookie.

❖ How it works - Stateful Cheetah

✓ New Connection (first packet)

1. A new connection arrives at the LB.
2. The LB pops a free index *i* from the `ConnStack`.
3. It picks a backend server via the load-balancing policy (e.g., round-robin).
4. It stores in `ConnTable[i]`:
 - ID: *i*.
 - hash: `hash_S(connID)`.
 - DIP: chosen backend's DIP.
5. It inserts **index *i*** into the packet's cookie field (e.g., in TCP timestamp bits).
6. Packet is forwarded to the chosen backend.

Now the LB **knows** that all packets with cookie = *i* belong to that connection.

☰ Subsequent Packets

1. When the LB receives another packet with cookie = *i*: it **directly indexes** into `ConnTable[i]`. No hashing, no lookups.
2. It fetches the DIP and forwards the packet.

This gives **constant-time lookups, insertion and deletion** ($O(1)$ complexity).

✗ Connection Close.

- When the flow ends:
1. The LB pushes index *i* back into the `ConnStack`, freeing the slot.
 2. That entry becomes reusable for a future connection.

⌚ Since the hash is not used for lookups in the stateful version, why is it still needed?

In **stateless Cheetah** (page 122), the hash of the connection ID (`hash_S(connID)`) was essential to *decode* the cookie and find the backend. But in **stateful Cheetah**, each packet carries the *index* of its entry directly, so: the LB doesn't recompute hashes to find where the entry is and the table operations are constant-time. So **why keep the hash at all?** The hash on the server serves a **security and validation** purpose:

- **To verify packet integrity and prevent spoofing.** Remember: **any client could try** to forge a cookie (e.g., random index) **to hijack another flow**. That would be disastrous because the LB might forward packets to the wrong backend. So Cheetah stores, in each `ConnTable` entry, the **hash** of that connection's 5-tuple, computed with the **secret salt "S"**.

When a packet arrives:

1. The LB reads the cookie (index).
2. Looks up `ConnTable[index]`.
3. Recomputes `hash_S(connID)` from the packet's header.
4. Compares it with the stored hash.
 - ✓ If they match → it's the correct connection.
 - ✗ If not → drop the packet (forged or mismatched).

That's a simple but powerful **authentication check** using the global secret "S".

- **To protect against attackers reusing indices.** Imagine an attacker guesses cookie 123 and sends packets pretending to belong to that slot. The LB will compute the hash of the fake connection tuple, compare it to the saved hash in `ConnTable[123]`, and immediately see they don't match. Then, the packet is discarded. So the hash keeps **per-connection isolation** secure, even though the index space is small (e.g., 2^{16} slots).
- **To detect table corruption or stale entries.** When a connection closes, the slot is freed. If an old packet (delayed or replayed) arrives with that cookie but a different `connID` the hash check ensures it's not accidentally matched to a new connection that reused the same index. So it's a **safety net** against both external and internal errors.

Role of <code>hash_S(connID)</code> in	<i>Stateless Cheetah</i>	<i>Stateful Cheetah</i>
Used to decode server ID (cookie \oplus hash)	✓ Yes	✗ No
Used to verify packet legitimacy	✗ No	✓ Yes
Used for lookup / routing	✓ Yes	✗ No
Computed per packet	✓ Yes	✓ Yes (for check)

Table 14: Comparison of the role of `hash_S(connID)` in stateless vs stateful Cheetah.

■ Size and Scalability

Each packet carries a small **cookie**, and inside the cookie there's an **index**. If the index part of the cookie is r bits long, then the LB can address up to 2^r **distinct slots** in its ConnTable. So the **maximum number of concurrent connections** that the stateful Cheetah can track is 2^r :

- If the cookie is 8 bits, then $2^8 = 256$ connections.
- If the cookie is 16 bits, then $2^{16} = 65,536$ connections.
- If the cookie is 20 bits, then $2^{20} = 1,048,576$ connections.
- If the cookie is 24 bits, then $2^{24} = 16,777,216$ connections.

So, **more bits in the cookie, more concurrent connections** we can track. But **cookies can't grow forever**, because:

- The cookie must fit inside an existing header field (e.g., part of the TCP timestamp).
- Those fields have limited space, usually around **16 to 32 bits** available.

So one Cheetah load balancer can realistically handle **a few million connections** at most per cookie. To scale beyond that, we need to **partition the connection space**.

✓ **Cheetah's trick: multiple tables.** Due to the limited cookie size, a single Cheetah LB can only track a few million connections. To scale to tens or hundreds of millions of concurrent connections, Cheetah uses **multiple independent connection tables** (ConnTable), each with its own free stack (ConnStack). The cookie now encodes **two pieces of information**:

- **Partition ID** (which table to use): needs $\log_2 m$ bits.
- **Index within that table**: needs r bits.

So the total cookie size is:

$$\text{cookie size} = \log_2 m + r \quad (10)$$

Think of having multiple small tables instead of one giant one.

Example 7: Cookie Size Example

For example, let's say we have:

- $m = 64$ **partitions** (independent tables).
- Each table has $2^r = 2^{16} = 65,536$ **entries**.

Then total capacity is:

$$\text{connections} = m \times 2^r = 64 \times 65,536 = 4,194,304$$

For $m = 64$ and $r = 16$, we get:

$$\text{cookie size} = \log_2 64 + 16 = 6 + 16 = 22 \text{ bits}$$

This fits nicely within a 32-bit field, leaving room for future growth. Also, with 22 bits, we can track over 4 million concurrent connections across all partitions ($64 \times 2^{16} = 4,194,304$).

In practice, each partition can be managed by:

- A different hardware pipeline (in a programmable switch).
- Or a different thread/core (in a software LB).

That way, **load is spread** and **insertions stay constant-time**, even under millions of concurrent flows.

Q And about memory usage? The total capacity is calculated as:

$$\text{connections} = m \times 2^r \quad (11)$$

However, it depends on the size of each **ConnTable** entry.

Example 8: Memory Usage Example

Continuing the previous example with $m = 64$ partitions and $r = 16$ bits per index, we can track 4,194,304 concurrent connections. Now let's estimate the memory usage.

Each **ConnTable** entry stores:

- Index: r bits (e.g., 16 bits).
- Hash: 128 bits (e.g., MD5 hash), or 256 bits (SHA-256).
- DIP: 32 bits (IPv4 address), or 128 bits (IPv6 address).

So each entry is about $16 + 128 + 32 = 176$ bits ≈ 22 bytes (or $16 + 256 + 128 = 400$ bits ≈ 50 bytes for IPv6 and SHA-256). For $2^{16} = 65,536$ entries, each table uses about:

$$65,536 \times 22 \text{ bytes} \approx 1.4 \text{ MB} \quad (\text{IPv4 + MD5})$$

$$65,536 \times 50 \text{ bytes} \approx 3.3 \text{ MB} \quad (\text{IPv6 + SHA-256})$$

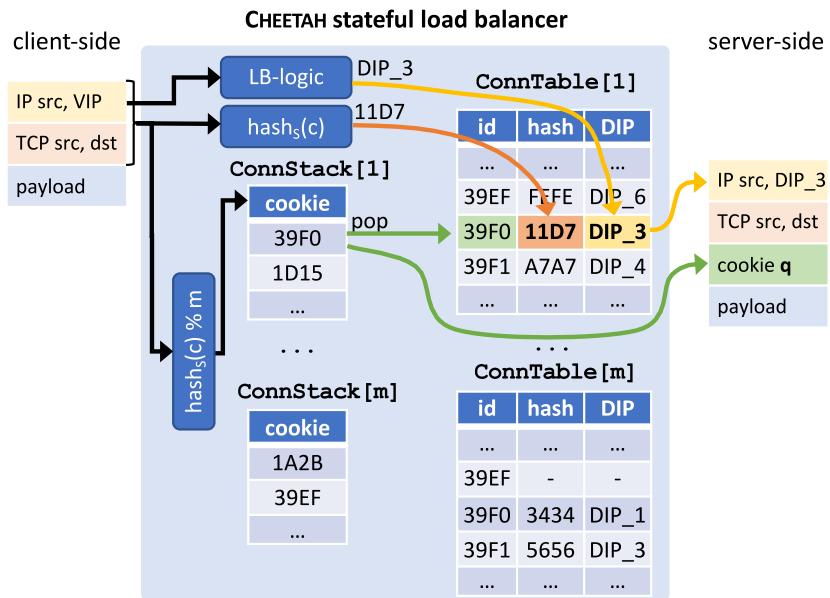
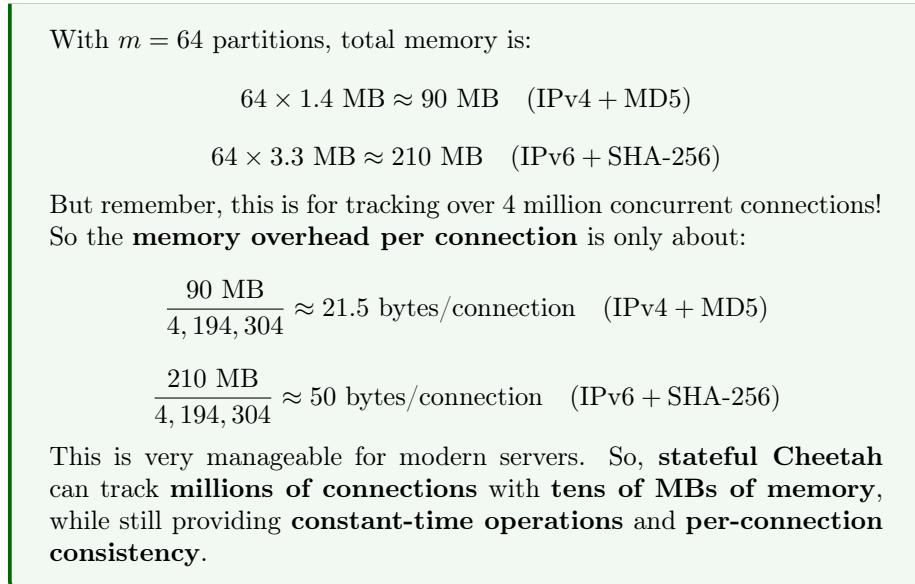


Figure 7: The diagram [2] shows the **Cheetah stateful** LB operations for the first packet of a connection. They do not show the stateless cookie for identifying the stateful LB. The VIP-to-servers is included within the LB-logic and not shown. The server performs Direct Server Return (DSR) so the response packet does not traverse the load balancers. Subsequent packets from the client only access their index in the corresponding ConnTable.

⌚ Hybrid Two-Tier Architecture

Imagine a huge datacenter with millions of incoming connections per second; hundreds of thousands of backend servers (DIPs); tens or hundreds of load balancer nodes. Now, if every LB tried to be fully **stateful** (keeping per-flow state for all connections), it would need:

- Gigabytes of RAM for connection tables;
- Synchronization with peers for redundancy (so PCC isn't broken);
- High complexity and slow updates.

Impossible to scale cleanly. So **Hybrid Cheetah**'s insight is: “*split the job in two (divide and conquer): let the fast, simple LBs do stateless work, and let fewer, powerful LBs handle stateful per-flow logic*”.

✅ Two-tier architecture

- **Tier 1: Stateless Cheetah LB:** Entry point. Chooses which **Tier-2** LB handles the flow and encodes that **choice** into the **cookie**.
- **Tier 2: Stateful Cheetah LB:** Manages actual connection **table** (`ConnTable` + `ConnStack`) and **forwards packets to backends**.

So the **Tier-1 LB** is **stateless**, very fast, and can handle millions of new connections per second. It is placed at the edge of the datacenter, receiving all incoming traffic. Instead, the **Tier-2 LBs** are **stateful**, but there are fewer of them (e.g., one per rack or cluster). They handle the actual connection tracking and backend forwarding.

❖ How it works - Hybrid Cheetah

• Client → Tier 1 (stateless)

1. The packet arrives at the data center edge (Tier-1 LB).
2. The Tier-1 LB computes `hash_S(connID)` and picks **which Tier-2 LB** should handle the flow.
3. It encodes that **Tier-2 LB ID** in the first few bits of the cookie.
4. It forwards the packet to the chosen Tier-2 LB.

Per-Connection Consistency (PCC) across Tier-1 LBs is guaranteed, because they all share the same salt “S” and thus compute the same hash, then the same Tier-2 LB choice.

• Tier 2 → Backend (stateful) (same as stateful Cheetah, page 127)

1. Tier-2 LB receives the packet.
2. It pops a free slot from its `ConnStack`, chooses a backend DIP, and fills one `ConnTable` entry.
3. It appends its local `ConnTable index` (and possibly partition ID) into the cookie.

4. It forwards the packet to the backend.

The cookie now contains:

[Tier-2 LB ID | ConnTable Index | Partition ID (if any)]

- **Server → Client (reply path)**

1. Reply packets from the server carry the same cookie back.
2. Tier-2 LB decodes its own index (ConnTable index) and instantly finds the entry.
3. It forwards the packet to the client through any Tier-1 LB (no state needed).
4. The Tier-1 LB just checks the Tier-2 ID in the cookie and sends it to the correct Tier-2 again if needed.

Thus, every packet, in both directions, finds the right Tier-2 LB and the right ConnTable entry deterministically, **no centralized state or coordination needed**.

It's **hybrid** because it combines the best of both worlds: **stateless speed** at the edge and **stateful control** deeper in the network. The *front tier* behaves like a **stateless hashing-based system** (high speed, low memory), while the *second tier* behaves like a **stateful table-based system** (connection tracking).

➃ The magic: PCC without coordination

Traditionally, if we add or remove load balancers, we risk breaking in-progress connections. Some packets might go to the “*wrong*” LB (which has no state for that flow). Cheetah solves this:

- **Tier-1 stateless LBs** compute the same deterministic mapping using `hash_S(connID)`.
- So even if we add or remove Tier-1s, each connection always lands on the same Tier-2 LB.
- That Tier-2 LB holds the state, no one else needs it.

Result: per-connection consistency is maintained even when the LB cluster changes dynamically.

➂ What happens when we scale up or down?

So, for **Tier-1 LBs, adding or removing nodes is easy**. Each Tier-1 LB uses the same hash function and secret salt “S” to map connections to Tier-2 LBs. So, when a **new Tier-1 LB is added**, it **simply starts receiving a share of new connections based on the hash**. Existing connections continue to be routed to the same Tier-2 LBs as before, **ensuring PCC**.

So the real challenge is **scaling Tier-2 LBs**, because they hold the actual connection state. When we add or remove a Tier-2 LB, we must ensure that existing connections are not disrupted.

+ Adding a Tier-2 LB (easy):

- Tier-1 LBs **update** their internal “VIPToServers”-like table with the **new Tier-2 instance**.
- **Future connections may hash** to the new Tier-2 LB.
- **Existing connections are unaffected** because their cookies encode the old Tier-2 LB ID, but this is not a problem since the old Tier-2 LB still has their state.

- Removing a Tier-2 LB (careful):

- **Active connections** on that LB can be **gracefully drained**. We stop sending *new* connections to that LB, but let the existing ones finish naturally.
- Tier-1 LBs **update** their internal table to **remove** the departed Tier-2 LB.

So the system is **elastic**, scale in/out without breaking flows or losing state.

7.6 Faild (Production Environment)

Faild is a **production-grade Layer-4 load balancer** developed and deployed by **Fastly** to operate at the **edge of the network**, where traffic first enters the infrastructure. Unlike research-oriented designs, Faild is shaped primarily by **operational constraints** rather than theoretical optimality.

Faild is deployed within Fastly's **Points of Presence (PoPs)**, which are best described as **small edge datacenters** located close to end users. Each PoP hosts a limited number of servers, operates largely independently, and is subject to frequent maintenance, upgrades, and failures.

❷ Motivation

Fastly needed a new L4 load balancer because **existing approaches were not well suited for deployment inside PoPs (Points of Presence)**, i.e., **small edge datacenters** where traffic patterns, failure modes, and operational requirements differ significantly from those of centralized datacenters. The key motivations behind Faild's design include:

- **Edge-oriented traffic characteristics.** Traffic at the edge is highly variable, with millions of geographically distributed clients generating short-lived connections. Traditional load balancers often assume more stable traffic patterns typical of datacenter environments. Faild is optimized to handle this bursty, ephemeral traffic efficiently and with low latency.
 - ✖ Traditional L4 load balancers struggle with the **unique traffic patterns at the edge**, leading to inefficiencies and increased latency.¹⁰
 - ✓ Faild is designed to **efficiently manage short-lived, bursty connections** from a large number of clients, minimizing latency and maximizing throughput.
- **Latency constraints.** At the edge, **every microsecond matters**. L7 load balancing is often too expensive due to TCP termination (page 110), application-level parsing, and higher per-connection overhead. Faild allows fast forwarding decisions based solely on transport headers, significantly reducing latency.
 - ✖ L7 load balancers introduce **significant latency** due to TCP termination and application-level processing.¹¹
 - ✓ Faild makes **fast forwarding decisions** based solely on transport headers, minimizing latency.
- **Failure is the norm.** Edge nodes are expected to **fail, restart, and be upgraded frequently**. Faild is designed to be **resilient to partial failures**, with state that is easy to rebuild and cheap to maintain.

¹⁰Because they are designed for more stable traffic, they may not handle the high variability and short-lived connections effectively.

¹¹This is particularly problematic at the edge, where low latency is critical for user experience.

- ✗ Traditional load balancers often assume **stable environments** and struggle with frequent failures at the edge.¹²
- ✓ Faild's state management is designed to be **easy to rebuild** and **resilient to partial failures**, ensuring high availability.
- **Operational simplicity.** Fastly prioritized **predictability and ease of debugging** over perfect load optimality. Faild avoids complex rebalancing logic and large per-flow state tables, favoring a fully stateless design that minimizes disruption during server changes.
 - ✗ Complex load balancing algorithms can lead to **unpredictable behavior** and **difficult debugging** in production environments.¹³
 - ✓ Faild employs a **fully stateless design**, prioritizing **operational simplicity** and minimizing connection disruption during server changes.
- **Gap between research and practice.** Many research proposals for L4 load balancing, such as Cheetah (page 121), assume **stable environments, controlled failure models, and strong consistency guarantees**. In contrast, Faild is built for the messy realities of production systems, where **robustness and debuggability** are more important than theoretical optimality.
 - ✗ Research proposals often fail to account for the **complexities of real-world deployments**, leading to solutions that are difficult to implement in practice.¹⁴
 - ✓ Faild is designed with a focus on **robustness and debuggability**, accepting slight inefficiencies in favor of predictable behavior.
- **PoP-scale constraints.** Each PoP consists of a **small number of servers** and operates with **local decision-making**, without relying on global coordination across datacenters. This makes heavy-weight state replication and centralized control impractical, pushing Faild toward simple, locally recoverable mechanisms.

In summary, Faild was designed to satisfy **real-world edge constraints**: low latency, frequent failures, limited state, and operational simplicity; even if this means giving up some theoretical guarantees offered by purely stateless or perfectly uniform load balancing schemes. In the next sections, we will explore the specific design goals, key choices, and trade-offs that shaped Faild's architecture.

¹²This can lead to complex recovery procedures and increased downtime.

¹³Large per-flow state tables increase memory pressure and complicate recovery after crashes.

¹⁴This includes assumptions about stable environments and controlled failure models that do not hold in production.

7.6.1 Design Goals and Choices

Faild was designed with a **production-first mindset**, where correctness, predictability, and ease of operation take precedence over theoretical optimality. The design goals reflect the realities of operating a large-scale edge infrastructure.

④ Design Goals of Faild

- **Production-first design.** Faild prioritizes **stable behavior under real workloads** and failure scenarios observed in practice. Design choices are validated by deployment experience and operational feedback. The system avoids mechanisms that are difficult to debug, require strong global coordination, or depend on idealized assumptions. So the **production environment drives the design**.
 - ✓ That reduces the likelihood of **bugs** and eases troubleshooting.
- **Simplicity over theoretical optimality.** Faild accepts **slight load imbalances** and **suboptimal resource utilization** if they lead to simpler, more predictable behavior. The system favors **straightforward mechanisms** that are easy to implement, reason about, and maintain over complex algorithms that may achieve marginally better performance *but* introduce operational risks.
 - ✓ This ensures that Faild can maintain service availability and performance even in the presence of frequent failures.
- **Fast failure recovery.** Faild is designed to quickly recover from **failures** common in edge environments, such as load balancer restarts, network partitions, and backend server crashes. The system **minimizes connection disruption and convergence times** by avoiding large state reconstructions and limiting the impact of failures on active connections.
 - ✓ This ensures that Faild can maintain service availability and performance even in the presence of frequent failures.
- **Minimal operational complexity.** Faild emphasizes **ease of deployment, upgrades, and incident response**. The design favors local decisions over global coordination and small, bounded state over large per-flow tables.
 - ✓ This reduces the operational burden on engineers, improves reliability, and enhances maintainability in day-to-day operations.

Faild's design goals reflect a pragmatic approach to building a load balancing system that **meets the demands of real-world production environments**. By prioritizing robustness, predictability, and operational simplicity, Faild aims to deliver reliable performance while minimizing the risks and complexities associated with large-scale deployments.

▣ Key Design Choices in Faild

Faild's architecture reflects a **carefully balanced compromise** between statelessness and operational practicality. Instead of adhering strictly to either extreme, Faild adopts a **controlled and minimal use of state** to satisfy production requirements.

✖ **Why Faild is not purely stateless.** A fully stateless L4 load balancer relies exclusively on hashing. When the backend set changes (failures, scaling), hash functions change, causing large numbers of **existing connections to be remapped**. At the edge, this would cause **massive connection disruption and poor user experience**. Faild avoids this by **not being purely stateless**.

✓ **Controlled use of state.** Faild maintains **small, bounded state** used only where it provides clear benefits, such as preserving connection affinity and limiting disruption during reconfiguration. This **state is easy to rebuild, cheap to maintain, and not required to be globally consistent**. The goal is **damage containment**, not perfect optimality. With “*damage containment*”, we mean that Faild aims to limit the negative impact of changes (like server failures) on existing connections, rather than trying to achieve an ideal distribution of load.

✖ **Practical connection affinity.** Faild implements mechanisms to **preserve connection affinity** for active connections, minimizing disruptions during backend changes. This includes remembering server assignments for active connections and avoiding unnecessary remapping when servers join or leave. The focus is on **real-world effectiveness** rather than theoretical guarantees. This reduces broken TCP connections, retransmissions, and load spikes due to reconnections. In other words, **Faild's affinity mechanisms are designed to degrade gracefully under failures**.

⚠ Engineering trade-offs vs Cheetah

– Cheetah

- * **Research-oriented:** designed to explore theoretical limits and novel algorithms.
- * **Fully stateless:** relies entirely on hashing without maintaining any connection state.
- * **Strong theoretical guarantees:** aims for optimal load distribution and minimal remapping.
- * **Focus on minimizing remapped connections mathematically:** prioritizes theoretical optimality over practical considerations.

– Faild

- * **Production-oriented:** designed for real-world deployment and operational robustness.
- * **Allows limited state:** maintains small, bounded state to preserve connection affinity.
- * **Focus on operational robustness:** prioritizes stability, predictability, and ease of operation.

- * **Optimizes for predictability and ease of recovery:** focuses on minimizing disruption during failures rather than achieving theoretical optimality.

Faild sacrifices some theoretical elegance to gain:

- * Simpler failure handling
- * Better real-world behavior
- * Lower operational risk

Faild deliberately avoids full statelessness, using **minimal, well-sscoped state** to preserve connection affinity and reduce disruption, prioritizing real-world robustness over theoretical optimality.

7.6.2 Faild vs Research Proposals

Faild clearly illustrates the **mismatch between academic assumptions and production realities** in the design of Layer-4 load balancers. Many research proposals are developed under controlled conditions, where failures are rare, backend sets are relatively stable, and infrastructure is homogeneous. In such environments, it is reasonable to optimize for elegant properties such as strict statelessness, perfect uniformity, or mathematically minimal connection remapping.

In production edge systems, however, these assumptions no longer hold. Failures, restarts, and reconfigurations are common events rather than exceptional cases. Traffic patterns are highly variable, and systems must remain operational even when parts of the infrastructure behave unpredictably. Under these conditions, designs that are optimal on paper can become fragile, hard to debug, and costly to operate.

Faild embraces this reality by prioritizing **stability over optimality**. Instead of attempting to perfectly balance load at all times, it **focuses on ensuring that the system behaves consistently under stress and that failures do not lead to widespread disruption**. Minor inefficiencies in load distribution are considered acceptable if they help contain damage and preserve user experience.

Similarly, Faild values **predictability over perfect uniformity**. Deterministic behavior is easier to reason about during incidents, simplifies debugging, and reduces the risk of cascading failures. From an operational perspective, **knowing how the system will react to changes is often more important than achieving ideal theoretical metrics**.

Overall, **Faild demonstrates that effective L4 load balancing in production** is not about achieving perfect theoretical guarantees, but **about making carefully engineered trade-offs**. By sacrificing some elegance and optimality, Faild gains robustness, operational simplicity, and resilience, qualities that are essential in large-scale edge deployments.

7.7 Summary

⌚ Core Problems Addressed

Layer-4 load balancing in datacenter and edge environments is fundamentally shaped by a small set of **core problems** that arise from the interaction between TCP semantics, large-scale traffic, and system failures (page 104).

1. The first and most critical problem is **connection affinity** (page 115). Since user-facing traffic is predominantly carried over TCP, all packets belonging to the same connection must be consistently forwarded to the same backend server. Violating this requirement breaks TCP state, leading to retransmissions or connection resets. Ensuring affinity is therefore a strict correctness constraint rather than an optimization.
2. The second problem is **uniform load distribution**. A load balancer must spread incoming connections across available servers to avoid overloading individual backends and underutilizing others. However, perfect uniformity is difficult to achieve in practice due to heterogeneous traffic patterns, variable request sizes, and differences in server performance. As a result, load balancing mechanisms must balance fairness with practical feasibility.
3. Finally, **failure handling** plays a central role in L4 load balancer design. Backend servers, load balancers, and network components can fail or be reconfigured frequently, especially in edge deployments. The system must handle these events without causing widespread connection disruption or long recovery times. Effective failure handling therefore requires mechanisms that limit the impact of changes and allow the system to recover quickly and predictably.

Layer-4 load balancing is constrained by strict connection semantics, imperfect load distribution, and frequent failures, making **robustness and determinism** **central design concerns**.

⚙️ Design Spectrum

Layer-4 load balancing solutions can be understood as points along a **design spectrum**, defined by **how much state they maintain** and **how they trade off optimality, scalability, and robustness**.

- At one end of the spectrum lie **stateless, hash-based load balancers** (e.g. Stateless Cheetah page 122). These systems map connections to backend servers using a deterministic hash of packet headers, without storing per-flow state.
 - ✓ Stateless designs are attractive because they scale well, are easy to replicate, and recover quickly from failures.
 - ✗ However, they suffer from significant drawbacks when the backend set changes, as even small reconfigurations can cause a large fraction of active connections to be remapped, leading to connection disruption.

- At the opposite end are **stateful load balancers** (e.g., Maglev page 112), which explicitly maintain per-flow tables mapping each active connection to a server.
 - ✓ This approach provides strong *connection affinity* and fine-grained control.
 - ✗ But it comes at the cost of high memory usage, limited scalability, and complex failure recovery. Reconstructing large state tables after failures can be slow and operationally challenging, making purely stateful designs difficult to deploy at scale.
- Between these two extremes lie **hybrid approaches**, such as **Cheetah** (page 121) and **Faild** (page 135), which combine deterministic mapping with a limited and carefully managed use of state. These designs aim to preserve connection affinity and reduce disruption during reconfiguration while avoiding the scalability and recovery issues of fully stateful systems. By selectively introducing state only where it provides clear benefits, hybrid solutions strike a balance between robustness, performance, and operational simplicity.

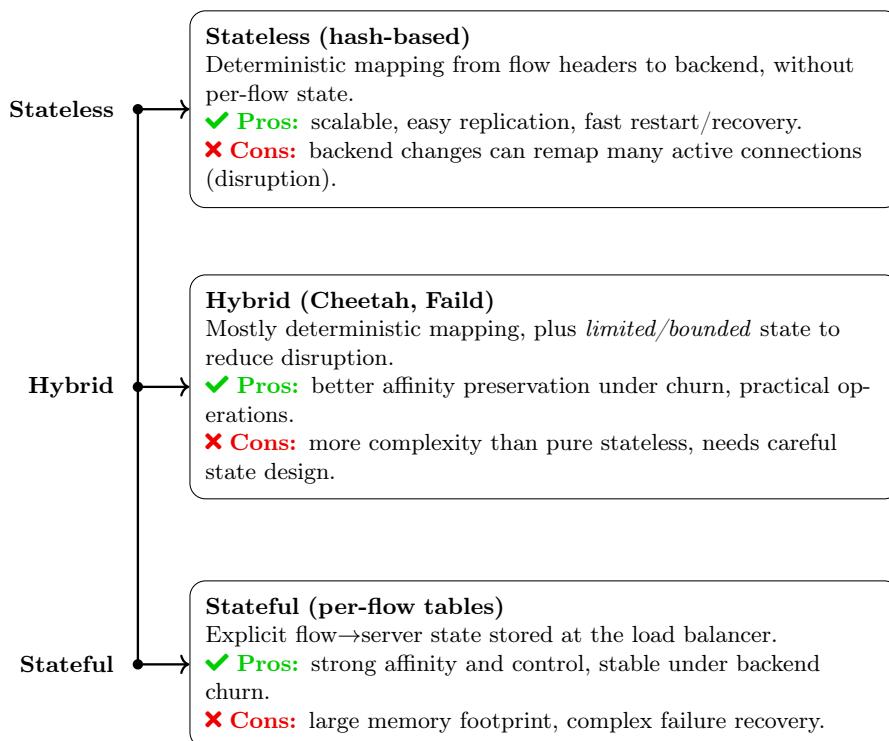


Figure 8: Layer-4 load balancing design spectrum: stateless vs hybrid vs stateful approaches.

🔗 Research vs Production

The comparison between **research proposals** and **production systems** highlights how Layer-4 load balancing design priorities change when moving from idealized models to real-world deployments (page 140).

- Research systems such as **Cheetah** (page 121) focus on achieving **clean theoretical properties**. By remaining stateless, they offer **strong guarantees on determinism and bounded connection remapping when the backend set changes**. These designs are elegant and analytically appealing, as they allow precise reasoning about behavior under controlled assumptions. In stable environments, they can provide **efficient load distribution while minimizing disruption** in a mathematically principled way.
- Production systems like **FaILD** (page 135), on the other hand, are driven by **operational constraints** rather than theoretical optimality. In edge deployments, failures, restarts, and reconfigurations are frequent, and systems must continue operating under imperfect conditions. FaILD therefore adopts a pragmatic approach, allowing a limited amount of state and favoring simple, predictable mechanisms. The goal is not to achieve perfect uniformity or minimal remapping, but to **ensure stable behavior, fast recovery, and ease of operation**.

This contrast shows that while research proposals advance our understanding of what is theoretically possible, production systems must optimize for robustness, debuggability, and long-term maintainability. As a result, production load balancers often sacrifice elegance in favor of reliability and simplicity. In other words, **Cheetah represents the elegance of theory, while FaILD represents the realities of production: both are valuable, but they solve fundamentally different problems** (Cheetah for ideal conditions, FaILD for messy real-world environments).

8 End-Host Networking

8.1 Why End-Host Networking Matters

In a **datacenter network**, traffic does **not** exist on its own. Every packet is:

- **Generated** by an application;
- **Processed** by an application;
- **Terminated** by an application.

And **all applications run on end-hosts** (i.e., servers, VMs, containers). Therefore, **end-hosts are where network traffic begins and ends**. This may sound trivial, but it is the *key insight* behind the **end-host networking** approach.

❓ What is an end-host?

An **End-Host** is simply a **server** (or VM, or container) inside a rack of a datacenter. It typically includes: CPUs, memory, one or more network interfaces (NICs), the operating system kernel and user-space applications. Unlike switches or routers:

- End-hosts **run applications**;
- End-hosts **execute the networking stack** (i.e., implement TCP/IP, UDP, etc.);
- End-hosts **interpret packet payloads** (i.e., they read and write application data).

👤 **Role of End-Hosts in the network.** From the network's perspective, end-hosts are: **traffic sources** and **traffic sinks** (i.e., they generate and consume traffic). However, from the application's perspective, end-hosts are **the only place where application semantics exist**. Indeed, the switches in the network can forward packets, inspect headers, and make routing/load-balancing decisions, but they **cannot create** packets, **consume** packets, or **decide what packets mean** (i.e., application semantics).

❓ **Why does this matter for performance?** Even if the **datacenter network fabric is perfect** (i.e., low latency, high bandwidth, no packet loss), packets still must:

1. Enter the server via the Network Interface Card (NIC);
2. Traverse PCIe (Peripheral Component Interconnect Express) bus to reach the host memory;
3. Be processed by the operating system kernel networking stack (e.g., Linux TCP/IP stack);
4. Be delivered to the application (e.g., web server, database).

If **any of these steps is slow**, then the **overall performance** of the networked application **suffers**, regardless of how good the network fabric is. Therefore, **end-host networking performance is critical** to the overall performance of datacenter applications.

Aspect	End-hosts	Switches / Routers
Run applications	✓	✗
Generate traffic	✓	✗
Interpret payload	✓	✗
Programmability	High	Limited
Deployment speed	Fast	Slow

Table 15: Key differences between end-hosts and in-network devices (switches/routers). The asymmetry in capabilities highlights why end-host networking is crucial for application performance. **End-hosts** are **software-driven** and **network devices** are **infrastructure-driven**.

Remark: What is the PCIe bus?

The **PCIe (Peripheral Component Interconnect Express)** is the **high-speed internal interconnect** that links the **Network Interface Card (NIC)** to the **CPU and memory of a server**. It is *not a networking protocol*, but it is **fundamental to networking performance at the end host**.

In other words, PCIe is the hardware bus that carries packets inside the server, moving data between the NIC and host memory/CPU at very high speed and low latency. The life of a packet at the host is as follows (simplified):

Network → NIC $\xrightarrow{\text{PCIe}}$ Host Memory → Kernel → Application

So PCIe is the **bridge between networking hardware and software**.

⚠ Why improving in-network hardware is hard in practice?

At first glance, improving the network seems like the obvious solution to improving application performance. After all, if the network is faster, then packets should arrive sooner, right? However, in **practice**, this turns out to be **very hard, slow, and expensive**. This is why *end-host networking* becomes attractive.

⚠ Operational complexity

✗ **Networks are shared infrastructure.** Datacenter networks are shared by **thousands of applications**, shared by **many teams**, and

operated under **strict reliability guarantees**. Any change to in-network hardware affects **all tenants, all applications**, potentially **the whole datacenter**. Even a small bug in a switch can cause **network-wide outages**.

- ✖ **Hardware bugs are catastrophic.** In-network devices operate at **line rate** (i.e., they forward packets at full speed), but they are also **hard to debug**, and often fail in **unpredictable ways**. If a switch drops packets silently, corrupts state or misroutes traffic, **everything breaks**, and it is **very hard to trace the root cause**.

⚠ Deployment time

- ✖ **Hardware innovation is slow.** Improving in-network hardware usually means new switch ASICs (Application-Specific Integrated Circuits), NIC firmware updates, drivers, or new control-plane software. All of these take **years** to design, test, manufacture, and deploy at scale. In contrast, end-host software can be updated in **days or weeks**.
- ✖ **Rollouts are painful.** Deploying new network hardware requires staged rollouts, compatibility testing, maintenance windows, and fallback plans. And often these rollouts require **recabling, topology changes, or downtime**, which are all costly and risky. This is the opposite of agile innovation.

⚠ Cost and compatibility

- ✖ **Financial cost.** Network hardware is extremely expensive, tightly budgeted, and amortized over many years. Replacing switches is not done lightly, and often requires capital expenditure approval. In contrast, end-host software changes are **low-cost and iterative**.
- ✖ **Compatibility constraints.** New in-network features must work with **existing protocols**, interoperate with **legacy devices**, and comply with vendor ecosystems. Often the innovation is constrained by **backward compatibility** and operators cannot deploy “experimental” features in production networks (e.g., new congestion control algorithms). End-host software, on the other hand, can be **customized** per application or team.

In summary, while improving in-network hardware seems appealing in theory, **practical challenges** make it **difficult, slow, and costly** in reality. This is why focusing on **end-host networking optimizations** is often the more viable path to improving application performance in datacenter environments.

✓ Advantages of End-Host Innovation

End-host innovation means **moving networking functionality closer to servers**, where applications run. This is not an accident or a workaround; it is a **deliberate design choice** adopted by modern datacenters for several reasons:

✓ Easier Deployment

- ✓ **Software beats hardware.** End-hosts are general-purpose machines, software-driven, and under frequent update cycles. This makes deploying new networking features, like updating a kernel module or a driver, or deploying a user-space service, **much easier** than changing switch hardware or firmware. Because there is **no need to touch the physical network**.
- ✓ **Smaller blast radius.** If something goes wrong, only a **subset of servers** is affected, rollback is easy and failures are **contained**. This makes experimentation and innovation **safe**.

✓ Faster Iteration

- ✓ **End-hosts follow software timelines.** At end-host changes happen in **days or weeks**, while network hardware changes take **years**. Also, CI/CD pipelines, automated testing, and continuous deployment are standard practice for end-host software, enabling rapid iteration and improvement. In contrast, network hardware changes require lengthy testing, staged rollouts, and careful planning. End-host innovation matches the **pace of application development**.
- ✓ **Easier debugging and testing.** At the end-host there is a full OS visibility, tracing tools (eBPF, tcpdump, perf), application-level context. This enables fine-grained performance tuning and rapid diagnosis of regressions.

✓ Closer to applications

- ✓ **Access to semantics.** Only end-hosts know which application owns a packet, what a flow represents, and which latency or throughput matters. This **semantic knowledge** enables smarter optimizations that are **application-aware**, such as prioritizing critical flows, adapting to workload patterns, or implementing custom congestion control algorithms. In contrast, in-network devices operate blindly, without understanding application context, they can only make decisions based on packet headers and statistics (e.g., headers, flow size, etc.).
- ✓ **Better cross-layer optimization.** End-hosts can jointly consider networking, CPU scheduling, memory hierarchy, and application logic. This is **impossible** inside the network fabric.

✓ Less disruption to the network

End-host innovation does not require network-wide coordination, preserves stability of the fabric and avoids vendor lock-in. Operators like this **a lot**.

So kernel bypass, programmable NICs, and eBPF are all examples of **end-host networking innovations** that leverage these advantages to improve application performance in datacenter environments.

8.2 Life of a Packet Inside a Server

When a packet arrives at a server, it does **not** go directly to the application. Instead, it must traverse a **layered processing pipeline**, which includes several steps:

- Hardware boundaries (NIC, bus, memory).
- Protection domains (kernel, user space).
- Software abstractions (network stack, sockets, libraries).

Each step adds **latency**, **CPU overhead**, and **potential bottlenecks**. Understanding this pipeline is crucial for optimizing network performance.

◀ The baseline packet path

At a very high level, an incoming packet follows this path:

1. **Network** (i.e., the wire). It represents the external world where packets are transmitted.
2. **NIC (Network Interface Card)**. The **NIC** is the hardware component connected to the network. Its responsibilities include:
 - Receiving packets from the wire.
 - Storing them temporarily in NIC memory.
 - Performing basic checks (e.g., checksum offload).
 - Initializing data transfer to host memory.The NIC operates **independently of the CPU**, it cannot interpret application data. So it is the **first bottleneck** in the packet path: **if it cannot sustain line rate, everything else is useless.**
3. **PCIe bus** (Peripheral Component Interconnect Express, connecting NIC to memory). The NIC is **not directly connected** to the CPU or main memory. Packets must cross the **PCIe bus** using **DMA (Direct Memory Access)** to transfer data to host memory. The PCIe bus has its own **bandwidth and latency characteristics**, which can impact performance.
4. **NIC Driver** (i.e., software managing the NIC). The **driver** is kernel code that manages the NIC, programs hardware registers, handles interrupts and moves packets into kernel data structures. It acts as the **software interface between hardware and kernel**. It runs in **kernel context**, it is executed frequently and bugs or inefficiencies here can have a large impact, since it is on the **critical path** of packet processing (i.e., every packet must go through it).
5. **Kernel Networking Stack** (operating system's network processing). The **network stack** is a complex software layer that implements network protocols (e.g., TCP/IP). It is responsible for:

- Protocol handling (e.g., TCP state machine).
- Packet reassembly.
- Congestion control.
- Routing.

This is where security checks happen, congestion control lives, but also packet ordering is enforced. The kernel provides **generality and safety**, but at the cost of **performance overhead** due to context switches, data copies, and protocol processing.

6. **User-space Application** (i.e., the server application). Eventually, the packet reaches a socket buffer or a user-space application via a system call (e.g., `recvfrom()`). Crossing from kernel to user space involves **context switches** and **data copies**, which add latency and CPU overhead. The user-kernel boundary crossing is expensive and unavoidable in the baseline model.

This is the **baseline model**, all optimizations later in the notes aim to **reduce the cost of one or more of these steps**.

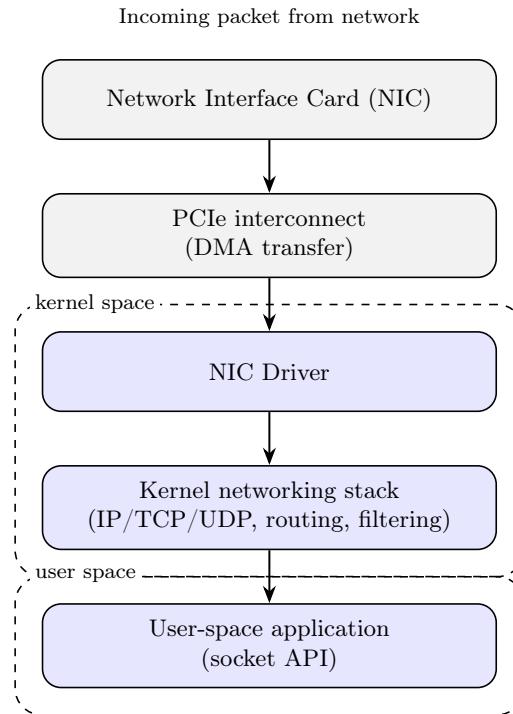


Figure 9: Baseline packet path inside a server.

✖ Kernel packet buffers and descriptor rings

Now that we have a high-level understanding of the packet path, let's look at some important data structures used in the kernel networking stack to manage packets efficiently.

Our focus is on the **interaction between the NIC and the kernel**. Since the **kernel mediates all packet transfers between the hardware and the applications**, its interaction with the NIC determines the performance, scalability, and isolation. This makes the kernel the primary bottleneck and optimization target in end-host networking.

The NIC **cannot write packets wherever it wants** in host memory. Instead, the kernel:

1. **Pre-allocates *packet buffers*** in memory to hold incoming packets.
2. **Tells the NIC where they are** in memory.
3. **Uses *descriptor rings* to coordinate ownership** of these buffers between the NIC and the kernel.

This design avoids CPU involvement in the fast path, enables high throughput, and supports batching and Direct Memory Access (DMA).

? **What are packet buffers?** **Packet Buffers** are regions of host (DRAM) memory allocated by the kernel used to store incoming packets. For example, in Linux, these are typically `sk_buff` structures that hold packet metadata and data, or memory pages managed by the networking subsystem. It is important to note that **buffers are allocated before packets arrive because this avoids dynamic allocation on the fast path**.¹⁵ As anticipated, pre-allocating buffers is crucial for performance because, without it, the kernel would require locks, causing cache misses and severely limiting throughput. **Pre-allocated packet buffers are essential for line-rate reception of packets.**

? **What is a RX descriptor ring?** **RX** stands for *receive* and refers to the direction of traffic:

- ← **RX** path: packet reception (incoming packets).
- **TX** path: packet transmission (outgoing packets).

Obviously, there are **TX descriptor rings** as well, but we focus on RX here because it is more complex and performance-critical.

A **descriptor** is *not* a packet. It is a **small control structure** that describes *where* a packet should go. We can think of it as a **post-it note** attached to a buffer. Usually, a descriptor contains:

¹⁵“*Dynamic allocation on the fast path*” means allocating memory for each incoming packet as it arrives, which would introduce significant latency and overhead. By pre-allocating buffers, the system can quickly place incoming packets into these pre-reserved memory areas, allowing for higher throughput and lower latency. Here, fast path refers to the critical execution path (i.e., the sequence of operations that must be performed quickly to ensure efficient packet processing) that handles incoming packets with minimal delay.

- A **pointer** to a packet buffer in host memory (physical address).
- The **size** of that buffer.
- **Status flags** (empty, full, ownership, errors, etc.).

For example:

```

1 Descriptor:
2     address = 0x1A2B3C4D    # Physical address of packet buffer
3     length  = 2048         # Size of the buffer in bytes
4     status   = EMPTY        # Status flag indicating buffer is empty

```

In simple terms, the descriptor tells the NIC where to DMA-write incoming packets.

Finally, a **ring** is just a **circular queue** with a fixed number of entries. A *circular queue* means that when we reach the end of the queue, we wrap around to the beginning. The circular nature allows for efficient use of memory (no reallocations), constant-time enqueue/dequeue operations, and perfect for hardware-software communication.

Putting it all together, an **RX Descriptor Ring** is a **circular queue of descriptors** that tell the NIC where to place incoming packets in host memory. The RX descriptor ring is:

- Allocated by the **kernel**.
- Shared with the **NIC**.
- Accessed concurrently by both the **kernel** and the **NIC**.

⌚ Relationship between NIC and Kernel memory via RX Descriptor Rings

1. Initialization phase

- The **kernel** allocates packet buffers in DRAM, creates descriptors pointing to empty buffers, marks them as **available** and tells the NIC about them.
- The **driver** programs the NIC with the address of the RX descriptor ring in host memory.
- The **NIC** now knows where buffers are located in host memory for incoming packets.

In this phase, the kernel and NIC set up the necessary data structures to enable efficient packet reception.

2. Packet reception phase. When a packet arrives, the **NIC**:

- Fetches** a descriptor from the RX ring (i.e., gets the address of an empty buffer) via **PCIe reads**.
- Performs a Direct Memory Access (DMA)** to **write** the incoming packet into the specified buffer in host memory.

- (c) **Updates** the descriptor status to **indicate that the buffer is full** and ready for processing by the kernel.

The NIC does **not** allocate memory on the fly, call the CPU, or touch the kernel during this fast path. It **simply uses DMA to write packets into pre-allocated buffers** as indicated by the descriptors.

❓ How does the kernel know when packets have arrived?

So far, the NIC did all the work of receiving packets and writing them into host memory. But the kernel **does not know** a packet arrived until the NIC notify the CPU. In the first naïve design, the NIC raises an **Interrupt Request (IRQ)** to a CPU core. Then, the kernel's NIC driver interrupt handler is invoked. This is the **first moment** the CPU is involved in packet processing.

⚠️ Interrupts can be expensive!

Handling interrupts involves context switches, saving/restoring CPU state, and can lead to cache misses. If packets arrive at a high rate, the **CPU can become overwhelmed with interrupts**, leading to **interrupt livelock** (page 154), where it spends all its time handling interrupts and cannot process packets effectively. To mitigate this, techniques like *interrupt coalescing* (batching multiple packets per interrupt) and *polling* (the kernel periodically checks for new packets instead of relying on interrupts) are often employed, and we will see them later in the notes.

3. **Ownership transfer.** The ownership of a descriptor:

- (a) Start with the **kernel** (buffer is empty, step 1).
- (b) Moves to the **NIC** during DMA write (buffer is being filled, step 2).
- (c) Returns to the **kernel** once packet is written (buffer is full).

This ownership transfer is crucial for synchronization between the NIC and kernel. It ensures that the **NIC only writes to buffers that the kernel has marked as available**, and the kernel only processes buffers that the **NIC has filled with incoming packets**. It prevents data races, avoids locks, and enables zero-copy transfers (i.e., no CPU copies needed).

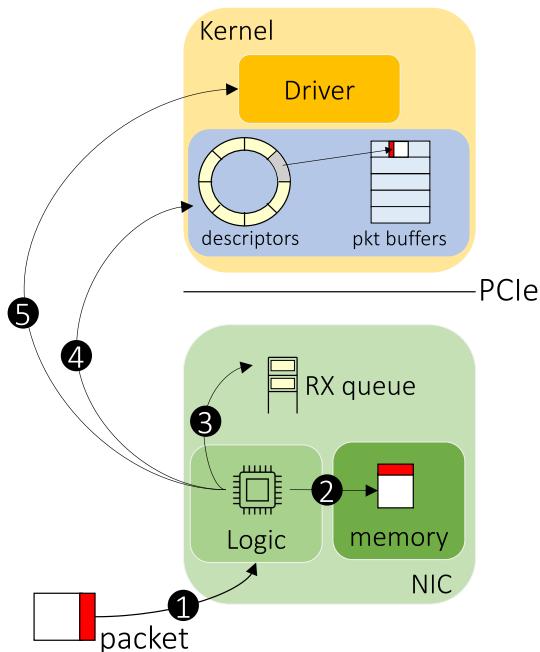
4. **Kernel processing phase.** Upon receiving an interrupt (IRQ) from the NIC, the **kernel**:

- (a) **Checks** the RX descriptor ring to identify descriptors marked as **full** by the NIC.
- (b) **Processes** the packets stored in the corresponding buffers (e.g., protocol handling, socket buffering, statistics).
- (c) **Reclaims** the buffers by resetting the descriptors and marking them as **available** again for the NIC.

This phase involves the kernel regaining ownership of the descriptors after packet reception. Buffer recycling is **essential** to sustain high throughput, as it allows the NIC to continue receiving packets without exhausting available buffers.

Step	NIC	PCIe	CPU (Kernel)
Packet arrival	✓	✗	✗
Descriptor fetch	✓	✓	✗
DMA transfer	✓	✓	✗
Interrupt	✓	✗	⚠ (interrupt)
Kernel processing	✗	✗	✓

Table 16: Summary of which components are involved in each step of the packet reception process using RX descriptor rings.



- The NIC has a **small internal memory** just enough to **hold packets briefly** before DMA transfer. This memory is **only for waiting**, not for storage.
- The NIC RX queue is just a small temporary waiting area inside the NIC that holds packets for a short time after they arrive from the network, until the NIC can copy them into main memory. It is separate from the kernel's descriptor ring and has nothing to do with kernel memory structures.

8.3 The Receive Livelock Problem

In the simplest receive model (page 148):

1. A packet arrives at the Network Interface Card (NIC).
2. The NIC DMA-writes the packet into main memory.
3. The NIC raises an **interrupt (IRQ)**.
4. The CPU stops what it is doing and handles the packet.

This happens **for every packet**.

⚠ Why does this become a problem? Interrupts are **expensive** because they preempt running tasks, flush CPU pipelines, pollute caches, and force context switches. At low packet rates this is fine and the overhead is negligible. However, at high packet rates (e.g., 10 Gbps and beyond), the CPU may spend **most of its time just handling interrupts** and very little time is left for *actual packet processing*. The CPU becomes the bottleneck, not the NIC. **Interrupt cost scales with packet rate, not with packet size**; many small packets are much worse than fewer large ones.

❓ What is Receive Livelock

Receive Livelock is a situation where the CPU spends all its time handling receive interrupts but makes no forward progress in processing packets. The system is busy, active, and consuming CPU, but **not productive**. It is called *livelock* because unlike a *deadlock* where the system is stuck doing nothing, here the system is busy doing something (handling interrupts) but not making progress. In other words, the cpu is *alive*, but **stuck reacting**.

❓ What is the CPU actually doing? In receive livelock, the CPU handles an interrupt, processes *very few* packets, immediately receives another interrupt, and repeats endlessly. It never gets enough uninterrupted time to drain the RX ring buffer and deliver packets to applications.

⚠ Why throughput can drop to zero

The most counterintuitive aspect of receive livelock is that **as the packet rate increases, the throughput can actually drop to zero**. This is counterintuitive because we might expect more incoming packets to lead to more processed packets. However, in receive livelock, the CPU becomes overwhelmed with interrupts and is unable to process packets effectively. The vicious cycle is:

1. High packet arrival rate.
2. NIC generates many interrupts.
3. CPU spends most cycles on interrupt handling.
4. Very little packet processing is completed.
5. RX ring fills up.

6. NIC cannot DMA new packets.
7. Packets get dropped.

As result, despite a high arrival rate, the effective throughput (packets successfully processed) can **plummet to zero** because the CPU is too busy handling interrupts to make any progress on actual packet processing. So **the system is interrupt-bound, not bandwidth-bound**. This is why faster NICs alone do **not** solve the problem.

⌚ Historical Context

Receive livelock was identified in the late 1990s and early 2000s on networks that were much slower than today's 10/40/100 Gbps links. However, modern NICs are 1,000 times faster, yet CPUs did not scale in interrupt efficiency at the same rate. Today, we have 25/40/100+ Gbps NICs, microservices with many small packets, and virtualized, multi-tenant systems. However, the **conditions for livelock are easier to reach than ever before**.

In summary, receive livelock is a critical challenge in high-speed networking where the CPU becomes overwhelmed with interrupts, leading to a situation where it is busy but not productive. In the next sections, we will explore various techniques to mitigate this problem and improve packet processing efficiency.

8.4 Interrupt Mitigation Strategies

8.4.1 Interrupt Coalescing

The core problem we just saw was that **too many interrupts per second overwhelm the CPU** (page 154). Interrupt coalescing fixes this by **reducing the interrupt rate**, not the packet rate. The key idea is very simple: **do not interrupt the CPU for every packet; instead, interrupt it for a group of packets.**

Definition 1: Interrupt Coalescing

Interrupt Coalescing is a mechanism in which a Network Interface Card (NIC) delays and **groups multiple packet** reception events, generating a **single interrupt to notify the CPU** about a batch of packets instead of one interrupt per packet.

⌚ What does “batching” mean?

In interrupt coalescing, **batching** means that the NIC receives **multiple packets**, processes them, and writes them into main memory. Finally, it **generates one interrupt for the entire group of packets**. So instead of interrupting the CPU for every single packet:

packet → interrupt → packet → interrupt → packet → interrupt

The NIC interrupts the CPU only after receiving a batch of packets:

packet packet packet packet → 1 interrupt

🛠 How does the NIC actually do this?

Modern NICs implement interrupt coalescing using **hardware rules**, such as generate an interrupt only:

- After N **packets** have been received, or
- After T **microseconds** have passed since the first packet in the batch was received.

These parameters (N and T) are configurable, allowing system administrators to tune the interrupt coalescing behavior based on their specific workload and performance requirements. In simple terms, the NIC effectively says: “*I'll wait a bit, accumulate work, then notify the CPU once*”.

⌚ Why is this helpful?

Interrupt handling has a **fixed cost**. If we handle:

- ✗ 1 interrupt per packet, we pay the interrupt cost **every time**.
- ✓ 1 interrupt per N packets, we pay the interrupt cost **once for every N packets**.

The CPU now:

- ✓ Spends less time context switching.
- ✓ Spends more time actually processing packets.

This directly reduces the chance of **receive livelock**.

⚠ What are the trade-offs?

The main trade-off with interrupt coalescing is between **latency** and **throughput**.

- ✓ **What improves.** By reducing the interrupt rate, the CPU can handle more packets overall, improving **throughput**, **CPU efficiency** and reducing the likelihood of **receive livelock**. This is why interrupt coalescing is **enabled by default** on most modern NICs.
- ✗ **What gets worse.** Packets are **not delivered immediately**. Because the NIC waits to accumulate packets or waits for a timer to expire before generating an interrupt, this introduces **additional latency** for packet delivery. For applications that require low latency (e.g., real-time communications), this can be a drawback.

Usually, this trade-off is acceptable for most workloads, because they care more about **throughput** than per-packet **latency**. However, for latency-sensitive applications (e.g.. RPCs, trading), careful tuning of the coalescing parameters (N and T) is necessary to strike the right balance.

⚠ What are the limitations?

While interrupt coalescing is effective, it is not a silver bullet. It **does not eliminate interrupts**. Indeed, at **very high packet rates**, even **coalesced interrupts can still overwhelm the CPU**. Additionally, it may not be suitable for all workloads, especially those requiring low latency. Therefore, interrupt coalescing is often used in conjunction with other techniques (like **polling** or **NAPI**, next sections) to further enhance packet processing efficiency.

8.4.2 Polling

Polling replaces this question “*NIC, tell me when a packet arrives*”, with “*CPU, repeatedly check whether packets have arrived*”. So instead of the NIC **pushing** work to the CPU via interrupts, the CPU **pulls** work by checking the RX ring buffer periodically.

Definition 2: Polling

Polling is a **packet reception mechanism** in which the **CPU repeatedly checks the network receive queues for incoming packets** instead of being notified by hardware interrupts.

❷ How can the CPU “actively check” for packets?

There are only **two possible ways** a CPU can check something:

1. Check once, then go do something else;
2. Check repeatedly, in a loop.

The **first option** is useless for polling, because if the CPU checks once and finds no packets, it will go do something else and miss any packets that arrive later. So the only viable option is the **second one** (polling), in which the CPU **spins in a loop**, repeatedly checking the RX ring buffer for new packets.

❸ Why does polling *necessarily imply busy waiting*? To detect packets via polling, the CPU must **continuously check** the RX ring buffer. It executes something like this:

```

1 while (true) {
2     if (rx_ring_buffer_has_packet()) {
3         process_packet();
4     }
5 }
```

Listing 1: Polling loop checking for packets.

This loop does not block, does not sleep, does not wait for an event. Instead, the CPU is **always running** this loop, which is the definition of **Busy Waiting**: the **CPU runs a loop that repeatedly checks the RX descriptors without sleeping or stopping**. So the CPU is always active (busy), never idle and never waiting for an event.

❶ Why does polling avoid livelock, and when is it useful?

With polling there are **no interrupts**, so no interrupt storms can occur, or context switches due to interrupts. The CPU is always in control, and can decide how often to check for packets. This **eliminates livelock** caused by interrupt storms. The receive path becomes **stable and predictable**, because the CPU is not interrupted by the NIC.

The polling is a **good idea** when:

- Packet arrival rate is **very high**.
- There is **always work to do**.
- Dedicating a CPU core to networking is acceptable.

Some typical examples where polling is useful: high-performance servers, packet processing appliances, NFV (Network Function Virtualization) systems, software routers, user-space networking stacks (e.g., DPDK, netmap). In these scenarios, the CPU would be busy anyway, so polling avoids the overhead of interrupts and context switches, leading to better performance and lower latency.

⚠ What are the downsides of polling?

Polling is **not always a good idea**, it's a **trade-off between CPU utilization and latency**. It is wasteful when:

- Traffic is **bursty or low**.
- Packets arrive infrequently.
- CPU cycles are valuable for other tasks.

In these cases, polling can lead to **high CPU usage** even when there are no packets to process, wasting power and resources. Also, if the traffic is low, the CPU may spend a lot of time checking for packets that are not there, leading to inefficiency.

In summary, polling **avoids receive livelock** by eliminating interrupts and letting the CPU actively check for incoming packets, providing **high throughput and low latency**, but at the **cost of increased CPU usage and potential inefficiency under low traffic** conditions.

8.4.3 NAPI (New API)

Although *interrupt coalescing* is a powerful technique for mitigating receive livelock, it has its limitations when the packet rate is extremely high. To address this issue, *polling* was introduced as an alternative to interrupts. However, polling can waste CPU cycles when the packet rate is low.

The obvious question then becomes: *can we dynamically switch between the two approaches based on the current load?* This is where **NAPI** (New API) comes into play.

Definition 3: NAPI (New API)

NAPI (New API) is a **hybrid mechanism** that combines the benefits of both **interrupts coalescing** and **polling**. It **dynamically switches** between the two approaches **based on the current load**, allowing for efficient handling of network traffic while minimizing CPU overhead.

② How does NAPI work?

The NAPI process can be divided into three main phases:

- Phase a) Low traffic (interrupt mode).** A packet arrives, and the NIC generates an interrupt. The CPU enters the interrupt handler, which processes the packet and checks the current load. This is identical to the traditional interrupt-driven approach.
- Phase b) High traffic detected (switch to polling mode).** If the interrupt handler sees that many packets are waiting to be processed (indicating high load), or the RX ring (page 151) is not empty after processing a packet, then the kernel **disables further interrupts for that RX queue and switches to polling mode**. In polling mode, the CPU actively polls the RX ring for new packets, processing them in batches without receiving interrupts for each packet. This allows for higher throughput and better performance under heavy load.
- Phase c) Load decreases (return to interrupt mode).** When the RX is drained (or budget, i.e., the maximum number of packets to process in one polling cycle, is reached), the kernel **re-enables interrupts for that RX queue and returns to interrupt mode**. This allows the system to efficiently handle low traffic without wasting CPU cycles on polling.

So **NAPI behaves dinamically** based on the current load, using interrupts for low traffic and polling for high traffic, thus optimizing performance across a wide range of network conditions.

❓ Why NAPI is better than pure polling

Pure **polling always spin** (i.e., it continuously checks for new packets), which can lead to **high CPU usage** even when there are no packets to process. It is really **bad for power consumption** and can **degrade the performance of other applications running on the same system**.

Instead, **NAPI only polls when there is a high load** of incoming packets, and it **reverts to interrupts when the load decreases**. This dynamic switching allows NAPI to efficiently handle varying network traffic while minimizing CPU overhead and power consumption, making it a superior solution compared to pure polling. So NAPI is a **controlled polling model**:

- ✓ **Stability at high load:** NAPI can handle high traffic without overwhelming the CPU, as it switches to polling mode when necessary.
- ✓ **Efficiency at low load:** NAPI minimizes CPU usage when the traffic is low by using interrupts, which allows the CPU to perform other tasks without being unnecessarily occupied with polling.
- ✓ **Avoids livelock:** By dynamically switching between interrupts and polling, NAPI helps prevent the receive livelock problem that can occur with pure interrupt-driven approaches under high load.
- ✓ **Avoids permanent busy waiting:** NAPI does not continuously poll when there are no packets to process, which helps reduce power consumption and allows other applications to run efficiently on the same system.

❖ What actually happens inside Linux?

Each RX queue has:

- A “*poll*” function (i.e., a callback) that is **called when the kernel decides to switch to polling mode**.
- A processing “*budget*” (i.e., the **maximum number of packets to process in one polling cycle**).

And they are used as follows:

- When the kernel is in polling mode, it repeatedly calls the *poll function* until the RX queue is drained or the *processing budget* is reached.
- Once the polling cycle is complete (i.e., either the RX queue is empty or the budget is exhausted), the kernel checks if there are still packets to process.
- If there are, it continues polling.
- Otherwise, it re-enables interrupts and returns to interrupt mode.

So the “*budget*” prevents starvation of other tasks (i.e., it ensures that the CPU does not spend too much time polling and allows other applications to run smoothly). Without the “*budget*”, the CPU could be stuck in a polling loop for an extended period, leading to high latency for other tasks and potential performance degradation.

❖ Why is it called “New API”?

It was called “**New API**” because, at the time it was introduced (early 2000s), it replaced the old Linux network driver interface. It was new relative to the previous interrupt-only driver model and to the old driver callbacks. The name has persisted even though it is now the standard approach for handling network traffic in Linux, and it is no longer considered “new” in the current context.

8.5 Multi-Queue NICs

So far we optimized **how one queue interacts with one single CPU core**. However, modern NICs have multiple queues, and we can use them to further increase the performance of the system. This section describes how to reach parallelism in the NIC and how to use it to further increase the performance of the system.

⚠ Why single RX queue is not enough

The simplest design is:

NIC → **one RX queue** → one interrupt → one CPU core

Even with interrupt coalescing, pooling or NAPI, we still have a fundamental bottleneck: **one RX queue can effectively feed only one core at a time**.

Modern servers have 8/16/32+ CPU cores and 25/40/100 Gbps NICs, and if all packets go through a single RX descriptor ring and a single interrupt source, then only one core does packet processing, and the rest of the cores are idle. This is a **per-core performance ceiling**¹⁶.

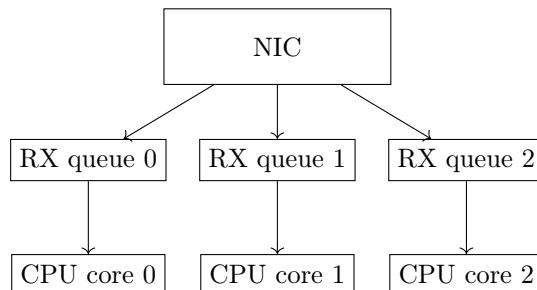
❓ **What breaks first?** At high traffic rates, a single core becomes saturated and the RX ring fills up, causing **packet drops**. So the problem is no longer livelock (page 154), but **lack of parallelism**.

✓ Solution: Hardware RX queues

The solution is to **replicate the RX queue in hardware**. Modern NICs provide multiple *independent* RX queues, each with:

- Its own descriptor ring;
- Its own interrupt source;
- Its own NAPI context.

So the architecture becomes:



Each queue operates independently and can feed a different CPU core, allowing for parallel packet processing.

¹⁶Ceiling means that no matter how much we optimize the system, we cannot go beyond a certain performance level

⌚ **What changes physically?** Instead of one RX descriptor ring in DRAM, we now have **multiple descriptor rings** and **multiple sets of packet buffers**. The NIC can decide which queue to use for each incoming packet, and the CPU cores can process packets from their assigned queues in parallel.

⚙️ Mapping queues to CPU cores

Now comes an important design choice: *which CPU core should handle which RX queue?* The simplest approach is to use a **static mapping**, where each RX queue is assigned to a specific CPU core (i.e., RX queue i is handled by CPU core i). This is done via **CPU affinity** settings in the operating system.

⌚ Concurrency and cache locality

With multiple RX queues processing packets in parallel, there is no need for locking between cores or for contention over shared RX ring resources. Each queue is independent, so there are no shared data structures requiring synchronization. This architecture scales well with the number of CPU cores as long as the NIC has enough RX queues to match the number of cores. This is the **benefit of concurrency**.

Furthermore, if packets from the same flow always go to the same RX queue and that queue is always handled by the same CPU core, we can **benefit from cache locality**. The TCP state stays in the same core's cache, the socket buffers stay warm, and there are fewer cache misses. However, if packets from one flow bounce across different cores, cache lines move between cores, and memory coherence traffic increases. This can **degrade performance**. Therefore, **multi-queue NICs only help if the queue-to-core mapping preserves locality** (i.e., packets from the same flow are processed by the same core).

In summary, multi-queue NICs enable scalable packet processing by providing multiple independent receive queues that can be mapped to different CPU cores, allowing parallelism while preserving cache locality and reducing contention.

8.6 Receive-Side Scaling (RSS)

In the previous section, we saw how multi-queue NICs can provide multiple RX queues to feed multiple CPU cores in parallel. However, we still need **a way to distribute incoming packets across these queues**. This is where **Receive-Side Scaling (RSS)** comes into play.

Receive-Side Scaling (RSS) is a **hardware mechanism in the NIC** that distributes incoming packets across multiple RX queues. It does this using a **hashing algorithm on packet header fields**. So instead of a static assignment of packets to queues (i.e., RX queue i to CPU core i), RSS allows for a more dynamic and flexible distribution of packets based on their content.

When a packet arrives:

1. The **NIC extracts certain header fields from the packet**, such as the source and destination IP addresses, source and destination ports, and protocol type.
2. It **computes a hash value** based on these fields. The specific fields used and the hashing algorithm can be configured, but common choices include the 5-tuple of header fields:
 - IPv4: source and destination IP addresses, source and destination ports, and protocol type.
 - IPv6: source and destination IP addresses, source and destination ports, and next header type.
3. It then **maps the hash value to one of the available RX queues**. This mapping is typically done using a modulo operation, where the hash value is divided by the number of RX queues, and the remainder determines which queue the packet goes to:

$$\text{Queue Index} = \text{Hash Value} \mod \text{Number of RX Queues}$$

❓ **Why use hashing?** Because we want packets from the same flow to go to the same RX queue, but different flows should spread across different queues. Hashing achieves both goals by using the packet header fields that uniquely identify flows.

RSS is important because **distributes flows** across multiple CPU cores, not individual packets randomly. This is critical for performance, because if packets of the same TCP connection went to different cores, TCP state would bounce between caches, locking would increase, and performance would degrade. So **RSS preserves flow affinity** (i.e, packets of the same flow go to the same core) while still allowing for load balancing across cores.

In summary, Receive-Side Scaling (RSS) is a NIC hardware mechanism that distributes incoming network flows across multiple RX queues by hashing selected packet header fields, enabling parallel packet processing across CPU cores.

✓ Benefits of RSS

✓ Load Distribution

✗ **Without RSS.** Even if the NIC has multiple RX queues, if packets are not distributed intelligently:

- * Most traffic may go to a single RX queue;
- * That queue is processed by one CPU core;
- * That core becomes overloaded;
- * Other cores remain underutilized.

So hardware parallelism exists but is **not used effectively**.

✓ **With RSS.** RSS ensures different flows are mapped to different RX queues, where each queue is mapped to a different CPU core. This allows:

- * Multiple cores to process packets in parallel;
- * **Better CPU utilization**;
- * **Higher throughput**.

✓ **Avoiding Single-Core Overload.** This is the most critical benefit. Even with NAPI, interrupt coalescing and polling, if everything lands on one queue (and thus one core), we still **hit a per-core processing limit**. **RSS removes that bottleneck by allowing parallel processing of independent flows**. So now performance scales with:

$$\text{num of CPU cores} \times \text{per-core processing capacity}$$

✗ Limitations of RSS

RSS is powerful, but it is **not perfect**:

✗ **Hash Imbalance.** RSS distributes flows using a **hash function**. But hashing does not guarantee a perfectly even distribution of flows across queues. Some queues may receive more traffic than others, leading to **load imbalance**, also known as **hash imbalance**. This happens because hashing is statistical in nature, and the distribution of flows may not be uniform. For example, if many flows share the same source or destination IP address, they may hash to the same queue, causing that queue to become a bottleneck. Or if there are a few very large flows (e.g., a popular web server), they may dominate one queue while other queues are underutilized.

So RSS **distributes flows, not traffic volume**. It cannot split a single heavy flow across multiple cores. So RSS works best when many flows of similar size are present.

✗ **Cache Inefficiency.** RSS ensures that packets of the same flow go to the same RX queue. But this does not guarantee perfect cache locality.

For example, consider a flow assigned to RX queue 2, which is processed by CPU core 2. But the application using that flow runs on CPU core 5. In this case:

1. Kernel processes packet on core 2
2. Application runs on core 5
3. Data must move across cores (from core 2 to core 5) for the application to access it
4. Cache lines migrate between cores

This creates cache coherence traffic, memory bus pressure and performance loss. So even though RSS preserves **flow affinity at kernel level**, it does not automatically ensure **application-level affinity** to the same core (i.e., the application is not pinned to the same core that processes the flow), which can lead to cache inefficiency.

While RSS enables scalable packet processing by distributing flows across cores, it may suffer from hash imbalance, cannot parallelize single heavy flows, and may cause cache inefficiency when application threads run on different cores.

 RSS Advantage	 RSS Limitation
Distributes flows across cores	Cannot split a single heavy flow
Prevents single-core overload	Hash imbalance possible
Preserves per-flow order	Does not guarantee application affinity
Scales with number of cores	May cause cache inefficiency

Table 17: Summary of RSS advantages and limitations.

8.7 Advanced Receive Flow Steering (aRFS)

The main problem with RSS is that packets of a flow may be processed by kernel on one core, while the application consuming them runs on another core. That creates cache misses, cross-core memory transfers, and latency overhead. So the new question becomes: “*can we steer packets directly to the core where the application runs?*” That’s where **Receive Flow Steering (RFS)** and **Accelerated Receive Flow Steering (aRFS)** come in.

✓ What RFS tries to fix

RFS (Receive Flow Steering), a **software correction mechanism placed in the kernel after RSS**, observes where the application runs. When the application reads from a socket, the kernel recognizes that “*Flow x is being consumed on Core y*”. RFS then **updates an internal mapping that directs packets of flow x to core y**. When future packets arrive, the kernel tries to process them on core y, which improves cache locality and reduces cross-core data movement.

However, it is important to note that **RSS still hashes packets to RX queues** based on the five-tuple. The kernel may internally redirect the processing of packets to the core where the application runs. This improves alignment, but since it is **still inside the kernel**, there is some **overhead**. For example:

1. Packet arrives via RSS to queue 2.
2. Interrupt/NAPI wakes up the kernel thread on core 2 to process the packet.
3. The kernel realizes that the application consuming this flow is running on core 5.
4. Packet processing must be shifted from core 2 to core 5, which involves cross-core data movement and cache misses.

So RFS improves performance by steering packets to the right core, but **it still incurs overhead due to the need to shift processing within the kernel**.

Definition 4: Receive Flow Steering (RFS)

Receive Flow Steering (RFS) is a **kernel-level mechanism** that dynamically steers incoming packets of a flow toward the **CPU core where the consuming application is running**, improving cache locality and reducing cross-core data movement. RFS operates by monitoring where applications read from sockets and updating internal mappings to direct packets of the corresponding flows to the appropriate cores for processing.

What aRFS improves

aRFS (Accelerated Receive Flow Steering) asks: “*why not tell the NIC directly where to send this flow?*”. Rather than relying on the kernel to redirect processing after the packet arrives, aRFS modifies the flow:

1. The application runs on core 5.
2. The kernel detects this.
3. The **kernel programs the NIC** to direct packets from this flow to the RX queue mapped to core 5.
4. Future packets arrive directly at the correct RX queue and are processed on core 5 without shifting processing within the kernel.

This way, **no correction** is needed after the packet arrives and **no cross-core data movement** occurs, which improves performance further.

However, aRFS requires **NIC hardware support** to enable the kernel to program flow steering rules. Not all NICs support aRFS, and enabling this feature may require specific drivers or configurations. Additionally, aRFS may not be beneficial for all workloads, particularly those with highly dynamic flow patterns or unstable core affinities. In such cases, the overhead of programming the NIC may outweigh the performance benefits. Finally, aRFS introduces **additional complexity to the kernel’s bookkeeping** (e.g., tracking which flows are steered to which cores), **flow-to-core mapping management**, and **steering rule updates**. This can add overhead in certain scenarios.

Usually, though, the **overhead is negligible compared to the performance benefits** of improved cache locality and reduced cross-core data movement, especially for high-throughput applications that process large volumes of network traffic.

Definition 5: Accelerated Receive Flow Steering (aRFS)

Accelerated Receive Flow Steering (aRFS) is a **hardware-assisted extension of RFS** in which the kernel **programs the NIC** to steer packets of a flow directly to the RX queue mapped to the CPU core running the application, further reducing cross-core overhead and improving performance.

Mechanism	Who decides?	Goal
RSS	NIC (hash-based)	Spread flows evenly across cores
RFS	Kernel (software)	Align flow with application core
aRFS	Kernel (program NIC)	Hardware steering to application CPU

Table 18: Summary of RSS, RFS, and aRFS.

8.8 Data Direct I/O (DDIO)

To further optimize the data path from the NIC to the CPU, modern CPUs have introduced a feature called **Data Direct I/O (DDIO)**. With DDIO, the traditional data path from the NIC to the application:

NIC → PCIe → DRAM → CPU cache → CPU core

Is optimized to:

NIC → PCIe → Last-Level Cache (LLC) → CPU core

Instead of writing incoming packets into DRAM first, the NIC can **DMA directly into the CPU's L3 cache (Last-Level Cache)**.

Definition 6: Data Direct I/O (DDIO)

Data Direct I/O (DDIO) is a CPU feature that **allows I/O devices**, such as NICs, **to perform DMA writes directly into the processor's last-level cache (LLC)** instead of main memory (DRAM), reducing memory latency and improving cache locality for packet processing.

✓ Benefits of DDIO

- ✗ Without DDIO, when a packet arrives at the NIC, it is written to DRAM via PCIe, and then the CPU must fetch it from DRAM into the cache before processing. This **adds latency** due to the additional memory access.
- ✓ With DDIO, the NIC can write the packet directly into the LLC, allowing the CPU to access it with much lower latency.
 - **Reduced Memory Latency:** By bypassing DRAM, DDIO significantly reduces the time it takes for the CPU to access incoming packets, which is critical for high-performance networking applications.
 - **Reduced Memory Bandwidth Pressure:** Since packets are not written to DRAM, there is less contention for memory bandwidth, allowing other applications to access memory more efficiently.
 - **Better Packet Processing Performance:** With lower latency and improved cache locality, applications can process packets faster, leading to higher throughput and better performance in network-intensive workloads.

⌚ How it relates to aRFS and Affinity

DDIO complements techniques such as aRFS and RSS by ensuring that, once steered to the correct CPU core, **packets can be accessed with minimal latency**. While aRFS and RSS focus on steering packets to the correct core, **DDIO ensures those packets are available in the cache for immediate**

processing. This further enhances the overall performance of end-host networking. Therefore, **cache locality is maximized** because the packet is in the cache, and there are **no cross-core cache transfers** because the packet is already in the correct core's cache.

However, DDIO reinforces the **importance of correct flow steering** because the **benefits of DDIO are only realized if packets are steered to the correct core**, where they can be accessed from the cache. Otherwise, the CPU would still have to fetch the packet from DRAM, which negates the benefits of DDIO.

⚠ Limitations of DDIO: The Leaky DMA Problem

Although DDIO offers significant performance benefits, it also introduces a potential size limitation known as the **Leaky DMA Problem**.

The last-level cache (LLC) is typically much smaller than DRAM and has **limited capacity** (e.g., 20-30 MB). If the amount of traffic becomes too high, the **network interface card (NIC) continues to write new packets into the LLC**. This can lead to **cache evictions**, whereby older packets are removed from the cache to make room for new ones. As a result, **previously received packets may be evicted before they are processed by the CPU**. Thus, rather than improving locality, an **overwhelmed cache can actually degrade performance**. In other words, the cache “leaks” useful packet data before processing it, which can lead to increased latency and reduced throughput.

Advantage	Risk
Lower latency	Cache eviction
Less DRAM traffic	LLC contention
Faster processing	Interference with application cache

8.9 Standard Offloads

We will continue our presentation of techniques to improve end-host networking performance. Before introducing kernel bypass techniques, which allow for very high performance, we will present another class of techniques that leverage CPU workload during packet processing: standard offloads.

Definition 7: Standard Offloads

Standard Offloads are hardware features implemented in modern NICs that offload common packet-processing tasks from the CPU to the NIC, reducing per-packet processing overhead while preserving the traditional kernel networking stack.

❓ **Why do we need standard offloads?** The answer is simple. **Standard Offloads reduce the CPU workload per packet without bypassing or removing the kernel networking stack.** This allows applications to use the familiar socket API and benefit from the kernel networking stack's rich features while offloading certain tasks to the NIC for improved performance.

❓ What are the common types of Standard Offloads?

- **Checksum Offload:** The NIC computes the checksum for outgoing packets and verifies the checksum for incoming packets, reducing CPU overhead for these operations.

Traditionally, the CPU computes TCP/IP checksum for each packet and verifies it for incoming packets. With checksum offload, the NIC takes care of these tasks, allowing the CPU to focus on other processing tasks.

- **TCP Segmentation Offload (TSO):** The NIC handles the segmentation of large TCP packets into smaller segments that fit the Maximum Transmission Unit (MTU) of the network. This allows applications to send larger data chunks, reducing the number of packets and CPU overhead for segmentation. TSO reduces Packets Per Second (PPS) seen by the CPU, improving performance for high-throughput applications.

The problem is that MTU limits the size of packets that can be transmitted over the network. But TCP may generate large data chunks that exceed the MTU. ❌ Without TSO, the CPU must split large buffers into MTU-sized segments, constructs many TCP headers, and sends many packets. ✅ With TSO, the CPU sends a large TCP segment to the NIC, and the NIC takes care of splitting it into MTU-sized segments (and adding the necessary TCP headers), reducing CPU overhead and improving performance.

- **UDP Fragmentation Offload (UFO): Similar to TSO, but for UDP packets.** The NIC handles the fragmentation of large UDP packets into smaller segments that fit the MTU, reducing CPU overhead for fragmentation.

- **Large Receive Offload (LRO):** The NIC aggregates multiple incoming packets into a larger buffer before passing it to the CPU, reducing the number of interrupts and CPU overhead for processing incoming packets.

It is the opposite of TSO. Without LRO, the CPU processes each received TCP segment individually, which can lead to high CPU overhead for high-throughput applications. With LRO, the NIC merges multiple incoming TCP segments into a larger buffer and passes it to the CPU as a single packet, reducing the number of interrupts and CPU overhead for processing incoming packets.

This technique is particularly beneficial for applications that receive a high volume of small packets, as it reduces the number of interrupts and context switches, improving overall performance and reducing Packets Per Second (PPS) seen by the CPU.

These standard offloads are helpful because the packet processing cost is often per-packet, meaning that the CPU overhead is proportional to the number of packets processed. By offloading tasks like checksum computation, segmentation, and aggregation to the NIC, we can significantly reduce the Packets Per Second (PPS) that the CPU needs to handle, improving performance for high-throughput applications while still using the traditional kernel networking stack.

⚠ Limitations. Although standard offloads can reduce CPU overhead for certain tasks, they do **not eliminate kernel overhead** for packet processing, **memory copies**, or the **interrupt-driven processing model**. Therefore, they are optimizations, not architectural shifts.

8.10 PCIe

All the operations discussed so far (RSS, RFS, aRFS, DDIO, and standard offloads) are performed by the CPU, but the CPU needs to exchange data with the network card, and this is done through the PCIe bus.

Definition 8: PCIe

PCIe (Peripheral Component Interconnect Express) is the interconnect that allows the NIC's DMA engine to transfer packet data into host memory. It consists of a root complex and memory controller on the CPU side, and it operates using packetized transactions called TLPs. Because PCIe is itself a packet-based protocol with headers and flow control, it introduces bandwidth and latency overhead that can become a bottleneck at high network speeds.

PCIe is the **de facto standard for connecting high-speed peripherals**, such as network interface controllers (NICs), to the central processing unit (CPU). However, other system components, such as GPUs and storage devices, are also connected through PCIe. This can lead to contention for bandwidth on the PCIe bus.

➲ The Data Path Through PCIe

When a packet arrives at the NIC, the NIC's DMA engine initiates a PCIe transaction to transfer the packet data into host memory. The CPU can then access this data for processing. After processing, if the CPU needs to send a response, it will write the response data back to host memory, and the NIC will perform another PCIe transaction to retrieve this data for transmission. The key components involved in this data path are:

1. **NIC**: The network interface card that receives and transmits packets. We will discuss the NIC architecture in more detail in the next sections.
2. **DMA Engine**: The NIC contains a **Direct Memory Access (DMA) engine** that handles the transfer of packet data between the NIC and host memory without involving the CPU, thus offloading this task from the CPU. The DMA engine takes descriptors from RX queue, writes packet data into memory buffers, and generates PCIe transactions to transfer this data. However, the **DMA engine can only transfer data at the speed of the PCIe bus, which can become a bottleneck at high network speeds.**
3. **PCIe**: The interconnect that allows the NIC's DMA engine to transfer packet data into host memory.
4. **PCIe Root Complex**: The PCIe root complex is a component of the PCIe architecture that connects the PCIe bus to the CPU and memory controller. It is the bridge between the PCIe bus, CPU, and memory controller. It receives PCIe transactions from the NIC's DMA engine and forwards them to the memory controller for access to host memory. Different CPU architectures may have different implementations of the PCIe root complex, which can affect the performance of PCIe transactions.

5. **Memory Controller:** The memory controller manages access to host memory and coordinates with the PCIe root complex. Writes DMA data into DRAM (host memory) and handles memory arbitration and scheduling. The performance of the memory controller can also impact the overall performance of PCIe transactions, especially when multiple devices are contending for memory access.
6. **Host Memory:** The memory where packet data is stored after being transferred by the NIC's DMA engine. The CPU accesses this memory to process packets and generate responses.

Each **basic data unit transferred over PCIe** is called a **Transaction Layer Packet (TLP)**, which consists of a header and payload. The header contains information about the transaction, such as the type of transaction (read/write), the address, and the length of the data. The payload contains the actual data being transferred. The size of TLPs can vary, but they typically range from 128 bytes to 4 KB.

A Effective Bandwidth & Protocol Overhead

- **Physical vs Effective Bandwidth:** While PCIe may have a high raw bandwidth (e.g., PCIe 4.0 $\times 16$ can provide up to 32 GB/s), the **effective bandwidth available for data transfer can be significantly lower due to protocol overhead**, such as TLP headers, flow control, and arbitration. This means that the actual throughput for packet data transfer may be much less than the theoretical maximum.

- **The Sawtooth Pattern:**

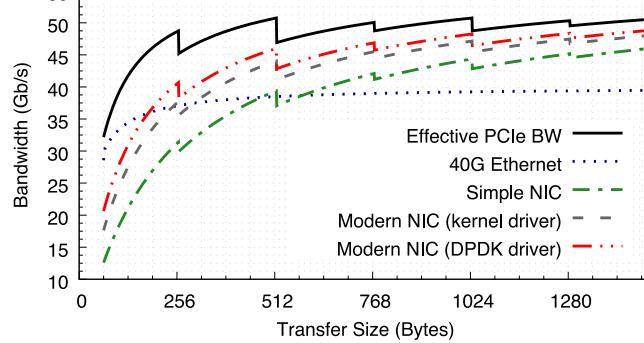


Figure 11: Effective PCIe bandwidth as a function of transfer size. [9] Due to the packetized nature of PCIe transactions (TLPs), small transfers suffer from significant protocol overhead, resulting in reduced effective bandwidth. As the transfer size increases, the relative header overhead decreases, and the achievable bandwidth approaches the physical limit. The characteristic sawtooth pattern arises from PCIe's maximum payload size, which forces large transfers to be segmented into multiple Transaction Layer Packets.

The relationship between transfer size and effective bandwidth can be visualized as a sawtooth pattern.

For **small transfer sizes**, the **overhead of PCIe transactions is high** relative to the amount of data being transferred, resulting in **low effective bandwidth**. As the **transfer size increases**, the **effective bandwidth improves** because the **overhead is amortized** over a larger amount of data. However, **beyond** a certain point, **increasing the transfer size** further may not yield significant improvements in effective bandwidth due to other **bottlenecks in the system** (e.g., memory controller performance, CPU processing time). This is why **optimizing the size of data transfers over PCIe** is crucial for achieving high performance in end-host networking.

❷ **Why does the sawtooth pattern occur?** The **sawtooth pattern occurs** because PCIe has a **maximum payload size** (e.g., 128 bytes, 256 bytes, or 512 bytes depending on the configuration). When a transfer exceeds this maximum payload size, it must be segmented into multiple TLPs, each with its own header and associated overhead. This segmentation leads to periodic drops in effective bandwidth as the transfer size crosses these thresholds, creating the characteristic sawtooth pattern in the effective bandwidth graph.

With smaller transfers, we pay header, protocol and control overhead for each TLP, which significantly reduces effective bandwidth. As transfer size increases, the overhead is amortized over more data, improving effective bandwidth until we hit the maximum payload size, at which point the transfer must be split into multiple TLPs, causing a drop in effective bandwidth and creating the sawtooth pattern.

❸ **Why is it Sawtooth (not smooth)?** The sawtooth pattern arises because **PCIe has a maximum payload size** (MTU-like behavior). **When transfer size exceeds the maximum payload, it must be split into** multiple TLPs. **Each split reintroduces additional overhead** (headers, flow control), causing a drop in effective bandwidth at those points. This results in a non-smooth, sawtooth-like pattern as transfer size increases.

– Small packets:

- ✗ Low PCIe efficiency due to high relative overhead.
- ✗ More TLP overhead per byte of data transferred.
- ✗ More DMA transactions required, increasing latency and reducing throughput.
- ✗ More pressure on PCIe bus and memory controller due to higher transaction rate.

– Large packets:

- ✓ Better efficiency as overhead is amortized over more data.
- ✓ Fewer TLPs needed, reducing protocol overhead.
- ✓ Higher effective bandwidth, approaching PCIe's physical limits.

In this context, using Standard Offloads techniques (e.g., TSO, LRO) to aggregate small packets into larger ones can help improve PCIe efficiency and overall network performance by reducing the number of small transfers and increasing the average transfer size, thus mitigating the impact of PCIe overhead on effective bandwidth.

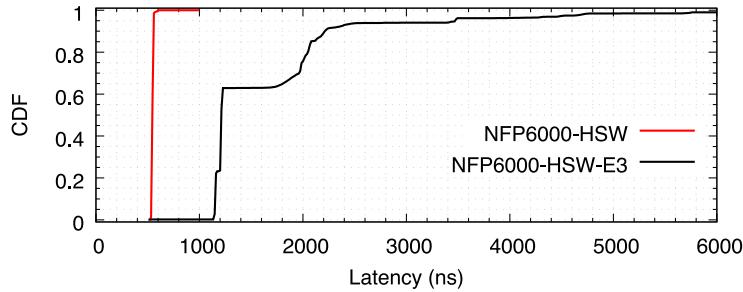


Figure 12: 64B PCIe DMA read latency across different CPU platforms. [9] The cumulative distribution function (CDF) shows significant differences in median and tail latency between systems, highlighting the strong impact of PCIe root complex implementation on end-host performance.

❖ How PCIe works

PCIe lets devices and the CPU **read/write each other's memory**, but the **memories on the two sides are independent**: there is **no cache coherence** between CPU caches and device/accelerator memory. That's the key limitation of PCIe.

1. **CPU does a PCIe read.** The CPU issues a **PCIe read** to fetch some **data** that lives on the device/accelerator side. We can think of this as “CPU asks the device for a cache line / buffer”.
2. **Data is transferred to the CPU side.** The requested **data** comes back over PCIe and is placed in CPU-visible memory and/or cache hierarchy. So now the CPU has a local copy.
3. **CPU reads from cache, but can't know if device data changed.** Now the CPU can read **data** quickly from its **cache**. But critical point is that the **CPU cannot know if, in the meantime, data on the accelerator has changed**. Because PCIe **does not automatically invalidate/update CPU cache lines** when the device updates its memory, the CPU might be working with stale data.
4. **Device memory changes (example: NIC writes into accelerator memory).** The device (e.g., NIC) can write new data into its own memory, but the CPU's cache still holds the old value. This can lead to **data inconsistency** if the CPU continues to read from its cache without being aware of the changes on the device side.

5. **CPU must trigger another PCIe read to be sure.** Because of the lack of cache coherence, the only safe option is for the CPU to perform another **PCIe read** to re-fetch the latest **data** from the device, ensuring that it has the most up-to-date information. This **additional read incurs latency and can degrade performance**, especially if frequent updates are happening on the device side.

In summary, **PCIe is non-coherent**, meaning that the **CPU and device memory are not automatically synchronized**. The CPU must explicitly read from the device to get the latest data, which can lead to performance issues due to stale data and additional PCIe transactions. This is a **fundamental limitation of PCIe** that affects how end-host networking systems are designed and optimized.

8.11 Compute Express Link (CXL)

In general, NIC speeds have been increasing faster than CPU speeds, which has led to the **need for faster interconnects between the CPU and the NIC**. In other words, the interconnect latency is orders of magnitude larger than packet transmission time, which is a **problem** for high-performance applications.

To address this issue, the **Compute Express Link (CXL)** has been developed as a **high-speed interconnect standard** with the goal of **replacing, or evolving from, the PCIe standard**. It uses the **same physical layer and form factor as PCIe**, and provides a **backward compatible** interface (i.e., it can be used with existing PCIe devices).

✓ Improvements over PCIe

CLX offers two main improvements over PCIe:

- ✓ **Lower Latency**: the PCIe minimum latency is around 400 ns, while CXL can achieve latencies as low as 200 ns, which is roughly a **2 \times improvement**. CXL achieves this by simplifying some parts of the PCIe protocol, reducing the transaction overhead and optimizing memory semantics (i.e., allowing for more efficient memory access patterns).
- ✓ **Cache Coherence (Major Conceptual Shift)**: CXL supports **cache coherence**, which allows devices to share memory and maintain a consistent view of data across the system. This is a **significant improvement over PCIe**, which does not support cache coherence and requires software to manage data consistency manually. **With CXL, devices can directly access each other's memory without needing to go through the CPU**, which can significantly reduce latency and improve performance for certain workloads (e.g., those that require frequent data sharing between the CPU and the NIC).

❖ How CXL Works

CLX creates a **cache-coherent domain** between the CPU cache/memory and the device side. So CPU caches and device updates stay consistent.

1. **CPU performs a CXL read**. The CPU issues a **CXL read** to fetch **data**. Conceptually the **same as PCIe** read, but under a coherent protocol.
2. **Data arrives and can be cached safely**. The **data** is returned and stored so the CPU can use it (typically cached). Up to here it **still looks like PCIe**.
3. **CPU uses cached data during processing**. The CPU reads data from its cache while handling an input/request. This is where the **cache coherence** comes into play. Here, the CPU is allowed to rely on the cache **because coherence is enforced by CXL**. This is a **major difference from PCIe**, where the CPU would have to ensure data consistency manually.

4. **If the device/NIC updates the data, CXL triggers cache invalidation.** The critical issue with PCIe is that when a device updates data, the CPU's cache may contain stale information. It would then be up to the software to address this issue. **CXL solves this problem by automatically invalidating the CPU cache when the NIC updates the data.** Specifically, each device participates in the cache coherence protocol. When a device updates the data, the CPU's cached copy is invalidated or updated, depending on the coherence protocol. Thus, the CPU will not use stale data and will be notified of the change.

In summary, **CXL is PCIe-like physically, but adds cache coherence:** CPU can cache device data, and if the device updates it, CXL ensures correctness by triggering cache invalidation, avoiding expensive repeated “re-reads” that PCIe would require.

8.12 NIC Driver

As we know, the NIC driver is responsible for sending and receiving packets on the network, as well as other functions such as interrupt handling and DMA management. In this section, however, we will summarize the NIC driver's main functions and explain how it interacts with the operating system and hardware.

The **NIC Driver** is **software** that runs in the kernel, but it interacts closely with the **hardware** (the NIC) to perform its functions. It abstracts away the hardware details and provides a standardized interface for the kernel to send and receive packets. In other words, it is the **bridge between hardware and the kernel networking stack**. Its job is to:

- Communicate with the NIC hardware to send and receive packets.
- Manage descriptor rings (page 151) for efficient packet processing.
- Convert raw packets into kernel data structures (e.g., `sk_buff`) for the networking stack.
- Inject packets into the Linux networking stack for further processing (e.g., routing, filtering).

When a packet is received:

- **NIC** hardware writes the packet data into a pre-allocated buffer (managed by the driver).
- **DMA** is used to transfer the packet data from the NIC to memory without CPU intervention.
- **Memory** barriers ensure proper ordering of memory operations between the NIC and CPU.
- **Driver** creates an `sk_buff` structure to represent the packet in the kernel and passes it to the networking stack.
- `sk_buff` is a kernel data structure that holds packet data and metadata (e.g., length, protocol information).
- **Networking stack** processes the packet (e.g., routing, filtering) and may generate a response packet.

The driver sits between the NIC (hardware) and the kernel stack (software), ensuring that packets are efficiently transferred and processed. It abstracts away the hardware details, allowing the kernel to focus on higher-level networking functions.

❖ Step-by-Step Receive Process

1. **NIC writes a packet via DMA.** The NIC receives a packet from the network and uses its DMA engine to write the packet into a pre-allocated buffer in memory. This buffer is part of the RX descriptor ring, which the driver has set up to manage incoming packets. The descriptor is updated to indicate that a new packet has arrived and is ready for processing. Note that the NIC does **not notify the application** directly; instead, it only updates the descriptor metadata to indicate that a packet has been received (page 156).
2. **Driver reads RX descriptor ring.** The driver checks the RX descriptor ring (either via interrupt or polling) to see if any new packets have been received. If it finds a descriptor marked as “*packet received*”, it reads the metadata to determine the packet’s location in memory and its length. Each descriptor contains a buffer address, a packet length and a status flag. **This is the first CPU interaction in the receive process.**
3. **Allocate sk_buff (skb).** The driver allocates an `sk_buff` structure to represent the received packet in the kernel. The `sk_buff` is a data structure used by the Linux kernel to manage network packets, containing pointers to the packet data, metadata (e.g., length, protocol information) and other fields needed for processing. The driver initializes the `sk_buff` with the DMA buffer and metadata fields from the RX descriptor. This **operation is not free** because it involves memory allocation, metadata initialization and linking to internal kernel structures. **At high packet rates, this can become a bottleneck** if not optimized.
4. **Clear descriptor entry.** After processing the received packet, the driver marks the descriptor as free and re-adds the buffer to the RX descriptor ring for future packets. From this point on, the NIC can reuse this buffer for new incoming packets. This **step is crucial for continuous packet reception**, as it ensures that the driver maintains a pool of available buffers for the NIC to write into.
5. **Pass skb to upper layers.** Finally, the driver calls `netif_receive_skb` to pass the `sk_buff` up the Linux networking stack for further processing (e.g., routing, filtering). The `netif_receive_skb` function is responsible for delivering the packet to the appropriate protocol handlers and eventually to the application layer. This step involves additional processing overhead as the packet traverses through various layers of the networking stack.

The **NIC driver typically uses NAPI (New API)** for packet reception, combining interrupt-driven and polling mechanisms to improve performance under high load (see page 160). However, even with NAPI, the driver allocates an `sk_buff` for each received packet. The kernel processes the packet, and the CPU must perform context switches to handle it, which can lead to performance issues at high packet rates. Thus, while **NAPI improves scalability, it does not eliminate kernel overhead** or CPU involvement in packet processing.

A Why the Driver is a Bottleneck? The driver performs several operations for each received packet, including:

- Reading the RX descriptor to get packet metadata.
- Allocating an `sk_buff` structure for the packet.
- Initializing metadata fields in the `sk_buff`.
- Managing the RX descriptor ring (marking descriptors as free).
- Calling functions to pass the packet up the networking stack.

At high packet rates (e.g., 10 million packets/sec), even small overhead per packet can add up to significant CPU load. The driver becomes a bottleneck because it **must perform these operations** for every packet, and the **CPU must handle the resulting interrupts and context switches**. This is why optimizing the driver and reducing per-packet overhead is crucial for achieving high performance in network applications.

8.13 The `sk_buff` (Socket Buffer)

The **Socket Buffer** (`sk_buff`), often called `skb`, is the **fundamental kernel data structure used to represent a network packet inside Linux**. In the standard Linux networking stack, every packet is wrapped inside an `sk_buff`. So instead of passing raw memory around, the kernel passes a pointer to an `sk_buff`:

```
struct sk_buff *
```

It contains two main parts:

- **Metadata**: packet length, protocol type, device info, header offsets, timestamp, checksum info, queue pointers, routing info, TCP state references.
- **Packet Data Buffer**: the actual packet bytes.

💡 Memory Layout

The structure of an `sk_buff` is designed to allow flexible manipulation of packet headers without needing to reallocate memory. This is crucial as packets move through the networking stack (Ethernet → IP → TCP → Application), where each layer may need to add or remove headers. Therefore, the memory layout of an `sk_buff` is crucial. Important pointers include:

- `head`: beginning of allocated buffer.
- `data`: start of packet data.
- `tail`: end of packet data.
- `end`: end of allocated memory.

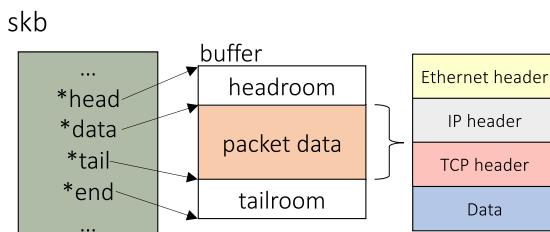


Figure 13: Memory layout of an `sk_buff`. The flexible pointers allow efficient header manipulation. [4]

💡 **Why are there two pointers for the beginning and end of the packet data?** Because the packet may need to grow or shrink as it moves through the stack, the kernel must be able to adjust where the packet data begins and ends without moving or reallocating the underlying buffer. Concretely:

- **On receive**: when a packet is received, the Ethernet header is removed (pull), and the IP header becomes the new start of the packet data.
- **On transmit**: when a packet is sent, a new Ethernet header may be added (push) before the existing data.

The `sk_buff` structure supports these operations by separating the beginning and end of the packet data from the underlying buffer, avoiding expensive reallocations and improving performance.

⚠ Why is it expensive?

At high packet rates, the cost of managing `sk_buff` becomes significant. Each packet requires:

- Allocation of an `sk_buff`.
- Initialization of metadata.
- Multiple memory accesses (cache line writes).
- Linking into queues.
- Eventually freeing it.

Since an `sk_buff` is a relatively **large structure** (hundreds of bytes), this leads to: increased cache pollution; higher memory pressure. This is why kernel bypass frameworks (e.g., DPDK) avoid using `sk_buff` and instead use simple fixed packet buffers without complex metadata, resulting in significant performance gains. But before we get to kernel bypass, we will see how the kernel optimizes packet processing with techniques like GRO (Generic Receive Offload).

8.14 The Linux Network Stack

After the NIC driver creates and initializes the `sk_buff` structure, the packet enters the **Linux network stack**, where it undergoes multiple stages of processing before being delivered to the application. The receive path is not a single function call, but a sequence of subsystems, each with a distinct role:

1. **GRO (Generic Receive Offload)**: performance optimization. Reduces per-packet overhead by merging packets belonging to the same flow.
2. **Netfilter**: policy enforcement and packet filtering. Applies security and policy rules such as firewall filtering and NAT.
3. **TCP/IP Stack**: transport and protocol processing. Ensures reliable transport, ordering, congestion control, and flow control.

Together, these components transform a raw packet received from the NIC into data that can be safely delivered to a socket in user space.

8.14.1 Generic Receive Offload (GRO)

Generic Receive Offload (GRO) is a kernel mechanism that **merges multiple incoming packets belonging to the same flow into a single larger `sk_buff`**, reducing per-packet processing overhead.

It is the **software counterpart** of the hardware-based Large Receive Offload (LRO) found in some NICs (see page 172). While LRO is implemented in the NIC hardware, **GRO is implemented in the Linux kernel** and can be used with any NIC, regardless of hardware support. It is **executed early in the receive path**, before heavy protocol processing.

❷ Why do we need GRO?

The fundamental problem is that the CPU overhead for processing incoming packets is often **proportional to the number of packets** (Packets Per Second, PPS). For high-throughput applications, this can lead to significant CPU overhead.

For example, if 10 TCP segments arrive in quick succession, without GRO, the kernel would process each segment individually, leading to 10 separate `sk_buff` structures and 10 separate processing steps. With GRO, the **kernel can merge these segments into a single `sk_buff`**, reducing the number of packets that need to be processed and thus reducing CPU overhead. So we reduce **packets-per-second (PPS)** seen by upper layers of the network stack, improving CPU efficiency, cache locality, and overall throughput.

✖ How does GRO work?

When a new `sk_buff` arrives to the kernel:

1. Kernel checks if it belongs to an existing flow (same 5-tuple: source/destination IP, source/destination port, protocol).
2. ✓ If it matches an existing flow, the kernel merges the new packet's payload into the existing `sk_buff`, updating headers and metadata accordingly.
3. ✗ If it does not match an existing flow, the kernel creates a new `sk_buff` for the new flow.

GRO maintains temporary flow structures to perform merging, and it is designed to be efficient and transparent to upper layers of the network stack. It **does not change the semantics of packet processing but optimizes it by reducing the number of packets that need to be processed.**

✓ Benefits and ✗ Tradeoffs

- ✓ **Benefits:** the **sawtooth pattern** of effective PCIe bandwidth (see Figure 11) shows that small packets suffer from high overhead, while larger packets achieve higher effective bandwidth. By merging small packets into larger ones, GRO can significantly improve performance for high-throughput applications by increasing effective transfer size **inside the kernel**.
 - Fewer `sk_buff` allocations and deallocations, reducing CPU overhead.
 - Fewer TCP header parses and protocol processing steps, improving CPU efficiency.
 - Fewer function calls and interrupts, improving cache locality and reducing context switches.
- ✗ **Tradeoffs**
 - **Increased latency for individual packets:** since GRO waits to merge packets, it may introduce additional latency for the first packet in a flow, as it waits for subsequent packets to arrive before processing.
 - **Complexity in flow management:** maintaining flow state for merging can add complexity to the kernel's receive path, and may require additional memory for flow tracking.
 - **Possible head-of-line blocking effects:** if one packet in a flow is delayed or lost, it can delay the processing of subsequent packets that are waiting to be merged, potentially impacting performance for that flow.

However, for high-throughput applications, the benefits of reduced CPU overhead and improved performance often outweigh the tradeoffs, making GRO a valuable optimization in the Linux network stack.

8.14.2 Netfilter

Netfilter is the **Linux kernel framework** responsible for packet filtering, Network Address Translation (NAT), connection tracking, and other **packet manipulation operations**.

It is the foundation of **iptables**¹⁷ and **nftables**¹⁸, which are used to define firewall rules, NAT rules, and other packet processing policies.

⌚ **Where does it sit in the receive path?** Netfilter operates after **GRO** and before the TCP/IP stack:

NIC → GRO → Netfilter → TCP/IP Stack → Application

So every packet (or merged packet from GRO) passes through Netfilter before being processed by the TCP/IP stack. Even if our application does not use a firewall, Netfilter hooks are still in the receive path, and the kernel will check for any applicable rules.

⌚ **What does Netfilter do?** It provides **hook points** inside the kernel networking stack where kernel modules can inspect packets and make decisions based on defined rules. Typical operations include:

- **Packet Filtering**: allow/deny packets based on IP, port, protocol, etc. This is the basis of firewall functionality.
- **NAT (Network Address Translation)**: modify source/destination IP addresses and ports for packets, enabling features like masquerading and port forwarding.
- **Connection Tracking**: maintain state of TCP flows, tracking sequence numbers and connection states (NEW, ESTABLISHED, RELATED).

⚠ **Performance Implications**: Netfilter adds rule matching, state lookup, hash table operations, and metadata updates to the receive path. For each packet, Netfilter may traverse rule chains, consult connection tracking tables, and update flow state. Even if no rule matches, the packet still goes through hook logic. At high packet rates, this results in significant CPU overhead.

▣ **Connection Tracking**: Netfilter's connection tracking is essential for stateful firewalling and NAT. It maintains a **hash table of active connections, tracking their state and metadata**. For TCP, it tracks sequence numbers, ACK numbers, and connection states. For UDP, it tracks source/destination IPs and ports. This **allows Netfilter to make informed decisions based on connection state** (e.g., allow established connections while blocking new ones). However, maintaining this state incurs **overhead, especially for high connection rates or long-lived connections**. Connection tracking can also lead to memory pressure if there are many active connections, as each connection entry consumes kernel memory.

¹⁷`iptables` is the user-space utility for configuring Netfilter rules.

¹⁸`nftables` is the newer user-space utility that replaces `iptables` for configuring Netfilter rules.

8.14.3 TCP/IP Stack

The **TCP/IP stack** is the core component of the Linux network stack responsible for processing packets according to the TCP/IP protocol suite. It handles tasks such as reliable transport, ordering, congestion control, and flow control. It is the final stage in the receive path before packets are delivered to the application. After GRO and Netfilter have processed the packet, it enters the TCP/IP stack for protocol processing.

NIC → Driver → GRO → Netfilter → TCP/IP → Socket → Application

At this stage, the packet is an `sk_buff` structure possibly containing merged packets from GRO, and it has passed through any applicable Netfilter rules. Now the kernel must handle **transport-layer logic**.

❓ **What does the TCP/IP stack do?** It performs several critical functions:

- **Generating ACKs:** for TCP packets, the stack generates ACKs to acknowledge received segments and maintain reliable communication.
- **Updating the congestion window:** the stack updates the congestion window based on ACKs and network conditions to control the sending rate.
- **Checking TCP checksum:** the stack verifies the integrity of TCP segments by checking the TCP checksum, ensuring that corrupted packets are detected and discarded.
- **Managing retransmissions:** if a packet is lost or an ACK is not received within a timeout, the stack handles retransmissions to ensure reliable delivery.
- **Maintaining flow state:** the stack keeps track of the state of each TCP connection, including sequence numbers, ACK numbers, and connection states (e.g., `ESTABLISHED`, `CLOSE_WAIT`).
- **Handling out-of-order segments:** when packets arrive out of order, the stack buffers and reorders them before delivering data to the application, ensuring correct in-order delivery.
- **Flow control (receive window):** the stack manages the receive window to control the flow of data from the sender, preventing buffer overflow and ensuring efficient data transfer.
- **Managing timers:** the stack manages various timers (e.g., retransmission timer, delayed ACK timer) to ensure timely processing of packets and efficient communication.

⚠ **Performance Implications:** The TCP/IP stack is reliable, ordered, congestion-controlled, and flow-controlled, which means it must perform complex processing for each packet. This includes hash lookup for socket matching, TCP state management, congestion window adjustments, ACK scheduling, timer management, and buffer reassembly. This is not simple parsing; this is **stateful protocol machinery** (i.e., it must maintain and update state for each

connection). As a result, the TCP/IP stack can be a **significant source of CPU overhead**, especially at high packet rates or with many active connections. The complexity of the TCP/IP stack is one of the main reasons why optimizations like GRO and Netfilter are important to reduce the load on this stage of the receive path.

8.15 Kernel Bypass

The kernel bypass is motivated by one central idea: the traditional kernel networking stack becomes the bottleneck at high packet rates.

⚠ Kernel Overhead

In the traditional Linux networking stack, the data path from the NIC to the application is as follows:

NIC → Driver → Kernel → Socket → Application

Each packet undergoes driver processing, interrupt handling, and a protocol stack (e.g., TCP/IP) before reaching the socket layer. Finally, the data is copied to the user-space buffer. Each of **these steps consumes CPU cycles**. At 10 Gbps, this may be manageable; however, at 100 Gbps, the CPU becomes overwhelmed. This leads to **packet drops and increased latency**. For example, with a minimum packet size of 64 bytes, the packet rate can reach 148.8 million packets per second. This is far beyond what a single CPU core can handle. At these rates, the per-packet overhead must be extremely small; even tens of CPU cycles matter. The classic kernel networking stack, designed for generality and correctness, was not originally designed for this scale.

⚠ Context Switching

When an application receives packets via sockets, it usually makes a system call (e.g., `recv()`) to read data from the kernel. This involves switching (*context switching*) from user space to kernel space, which is costly in terms of CPU cycles. The process is as follows:

Application → System call → Kernel → Return to user

Each packet reception may involve a transition from user space to kernel space and back again. This process is called a **context switch**. Context switches are not free. They can take thousands of CPU cycles because the CPU must save registers, change the privilege level, switch stacks, and flush parts of the CPU pipeline. Although it only costs a few hundred nanoseconds, millions of packets per second results in significant overhead. **Context switches destroy scalability because the CPU spends more time switching contexts than processing packets**. This is especially problematic for high-performance applications, such as web servers, databases, and real-time analytics, which require low latency and high throughput.

⚠ System Call Cost

When using the traditional kernel networking stack socket APIs:

- `recv()`: This system call is used to receive data from a socket. It involves copying data from the kernel buffer to the user-space buffer, which incurs overhead. The kernel must also check permissions, manage buffers, and handle any necessary protocol processing before returning data to the application.

- `send()`: This system call is used to send data through a socket. Similar to `recv()`, it involves copying data from the user-space buffer to the kernel buffer, which also incurs overhead. The kernel must handle protocol processing, manage buffers, and ensure that the data is sent correctly over the network.
- `select()/poll()/epoll()`: These system calls are used for multiplexing I/O operations. They allow an application to monitor multiple file descriptors (sockets) for events such as incoming data or the ability to send data. However, they also involve overhead due to the need to check multiple file descriptors and manage the event loop.

In general, each of these system calls requires a **transition from user space to kernel space**, which is expensive in terms of CPU cycles. **Even when no packet is available**, the cost of the syscall is incurred because the application must check for events. This can lead to **significant overhead**, particularly at high packet rates, when the application may make millions of system calls per second. The **kernel bypass approach aims to eliminate this overhead by enabling applications to access the NIC directly and manage their own buffers**. This avoids the need for system calls and context switches.

⚠ Packet Copy Overhead

In the traditional kernel networking stack:

1. NIC DMA writes packet into kernel memory (kernel buffer).
2. Kernel copies packet from kernel buffer to user-space buffer (application buffer).

We have two copies of each packet: one in kernel memory and one in user space. This equates to **at least one memory copy per packet**, which can be costly at high packet rates. For instance, with a 1500-byte packet and a 100 Gbps link, there can be up to 8.3 million packets per second. This means the CPU must perform 8.3 million memory copies per second. This can **consume a significant amount of CPU cycles** and lead to performance degradation. A **kernel bypass eliminates the kernel-to-user packet copy, allowing the NIC to DMA packets directly into user-space memory**. This reduces memory bandwidth usage and latency.

In summary, the kernel networking stack was optimized for fairness (i.e., sharing the CPU among multiple applications), security (i.e., isolating applications from each other), general-purpose workloads (i.e., supporting a wide range of applications), and multi-tenant systems (i.e., allowing multiple applications to run on the same machine). However, high-performance applications, such as load balancers, firewalls, and trading systems, require maximum throughput, lowest latency, and predictable performance. The kernel bypass approach allows these applications to achieve their performance goals by eliminating the overhead associated with the traditional kernel networking stack.

8.15.1 Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) is a set of user-space libraries and drivers that allow applications to process packets directly from the NIC without using the kernel networking stack.

DPDK does three fundamental things:

1. Uses **user-space drivers** (poll-mode drivers) to bypass the kernel and access the NIC directly.
2. Maps NIC DMA memory directly into user-space. It means that applications can access the NIC's memory directly without copying data between kernel and user space.
3. Uses a **polling model** instead of interrupts to process packets.

>User-Space Drivers (Poll-Mode Drivers)

In a traditional kernel-based networking stack, the NIC driver operates in kernel space and uses interrupts to notify the CPU when packets arrive (see page 156). In contrast, DPDK uses **poll-mode drivers (PMDs)** that run in user space and continuously poll the NIC for incoming packets. This approach eliminates completely the `sk_buff` structure, the kernel socket layer and interrupts. So the application continuously polls the RX ring, directly reads descriptors, and processes packets as they arrive.

Hugepages & Direct Memory Access (DMA)

DPDK uses **hugepages** to allocate large contiguous blocks of memory that can be directly accessed by the NIC via Direct Memory Access (DMA). They are very large memory pages (e.g., 2MB or 1GB) that reduce TLB misses (i.e., the overhead of translating virtual addresses to physical addresses), reduce page table overhead, improve DMA efficiency, and provide physically contiguous memory for the NIC. By mapping this memory directly into user space, DPDK allows applications to read and write packet data directly from/to the NIC without copying data between kernel and user space, significantly improving performance (zero kernel-to-user copying).

Polling Model (No Interrupts)

DPDK uses a **polling model** instead of interrupts to process packets. The application continuously polls the Network Interface Controller (NIC) for new packets, eliminating the overhead associated with handling interrupts and context switches. This technique can also be adopted in kernel-space drivers (e.g., NAPI); however, DPDK's user-space implementation allows for even lower latency and higher throughput by bypassing the kernel entirely. However, in this case, a CPU core must be dedicated to polling the NIC, which can lead to increased CPU usage and power consumption. Therefore, DPDK is typically used in scenarios where high performance is critical and the workload justifies dedicating CPU resources to packet processing, such as in high-frequency trading and network function virtualization.

✓ What DPDK Eliminates

By using DPDK, applications can bypass the entire kernel networking stack. This means that DPDK eliminates:

- `sk_buff` structure and associated overhead.
- Netfilter and other kernel-level packet processing modules.
- Kernel TCP/IP stack and its associated overhead.
- System calls for packet processing (e.g., `recvfrom`, `sendto`).
- Interrupt handling overhead, since DPDK uses polling instead of interrupts.

✗ What DPDK does NOT eliminate.

- DPDK does not remove:
- **PCIe bus communication** between the CPU and the NIC, which still incurs some latency.
 - **DMA operations**, which can still introduce latency, although DPDK optimizes this with hugepages.
 - **Memory bandwidth limitations**, as the CPU and NIC still need to access memory to read/write packet data.
 - **Cache effects**, as the CPU's cache may still be involved in processing packets, although DPDK's design minimizes this overhead.

It removes **software stack overhead**, not hardware communication overhead.

The **main problem** with DPDK is that bypassing the kernel means **giving up all the features it provides**, such as security, isolation, and energy efficiency. This requires implementing those features ourselves in user space. Additionally, DPDK is designed for high-performance applications requiring low latency and high throughput. However, it may not be suitable for general-purpose applications that do not require such performance levels.

Traditional Stack		DPDK
Flexible		Specialized
Secure		Less isolated
Energy-efficient		CPU-hungry
Kernel-managed		App-managed
Easy API		Complex programming

Table 19: Comparison between traditional kernel-based networking stack and DPDK.

8.15.2 Remote Direct Memory Access (RDMA)

Kernel Bypass with a twist

Instead of removing the kernel stack entirely (DPDK), or keeping everything in software (traditional stack), we do something else: we **move transport-layer processing into the NIC**, but we keep the kernel stack for control-plane operations. This is a sort of **twisted kernel bypass**, where we bypass the kernel for data-plane operations, but still use it for control-plane operations.

In other words, instead of only bypassing the kernel stack, **part of the transport-layer functionality such as TCP processing is moved into the NIC**. This reduces CPU involvement and allows higher throughput and lower latency at very high network speeds.

So instead of:

NIC → CPU → TCP → Application

We do:

NIC (with transport logic) → Application

This reduces CPU usage and latency, while still allowing the kernel to manage resources and provide security.

② How can we implement this?

There are two main approaches:

- **TCP Offload Engine (TOE)** is a NIC that **implements a full TCP stack in hardware or firmware**. This allows the NIC to handle all TCP processing, including connection management, flow control, and error handling.

✖ However, it was an idea from the early 2000s that never really took off due to its complexity, cost, and performance issues. The main problems were:

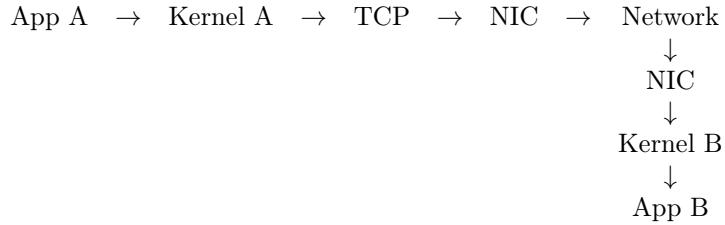
- **TCP Evolves Frequently**: TCP is a complex protocol that has evolved over time with many extensions and features. Implementing a **full TCP stack in hardware is difficult and expensive**, and it can quickly become outdated as TCP evolves.
- **Debugging and Monitoring**: When TCP logic is in hardware, it becomes very difficult to debug or monitor network behavior.
- **Vendor Lock-in**: Each NIC vendor implements TOE differently, leading to compatibility issues and vendor lock-in.
- **Security**: TOE bypasses many kernel security mechanisms, making it harder to enforce policies and maintain security.

For these reasons, TOE was adopted in some proprietary systems but **never became widespread in the industry**.

- **RDMA (Remote Direct Memory Access)** is a more modern approach that allows applications to directly access the memory of remote machines without involving the CPU or kernel for data transfer.

- **Remote:** The memory being accessed is on a different machine across the network.
- **Direct:** The application can directly read/write to the remote memory without going through the CPU or kernel for data transfer.
- **Memory Access:** The application can access the remote memory as if it were local, using load/store operations.

So the traditional networking:



With RDMA, we can do:

App A → RDMA NIC A → Network → RDMA NIC B → Memory B

CPU B is not involved in the data movement at all. The remote CPU is not interrupted, and there is no kernel stack traversal for the data transfer. Precisely, RDMA allows to bypass:

- Kernel networking stack
- TCP/IP stack
- `sk_buff` management
- Netfilter processing
- System calls per packet
- Remote CPU involvement in data transfer

This is **stronger than DPDK**, which still requires the application to manage the NIC and handle some of the processing in software. RDMA allows for **true zero-copy networking** with very low latency and high throughput, making it ideal for high-performance computing, distributed databases, and other latency-sensitive applications. Also, **RDMA removes CPU from the data movement path entirely**, while DPDK still requires the CPU to manage the NIC and handle some of the processing in software.

❖ How does RDMA work?

In RDMA, applications do not “send packets” in the traditional sense. Instead, they issue **verbs**. A **Verb** is a low-level operation submitted to the RDMA NIC (RNIC) to perform communication or memory access. Applications interact with RDMA through a **verbs API**. There are two main categories of verbs:

- **Two-Sided RDMA Verbs:** These verbs require both the sender and receiver to participate in the communication. For example, machine *A* wants to send data to machine *B*.
 1. Machine *B* posts a **Receive Work Request (WR)** to its RNIC, indicating that it is ready to receive data and providing a buffer for the incoming data.
 2. Machine *A* posts a **Send Work Request (WR)**.
 3. The RNIC on machine *A* matches the Send WR with the Receive WR on machine *B* and transfers the data directly into the buffer provided by machine *B* without involving the CPU or kernel on either side.
- **One-Sided RDMA Verbs:** These verbs allow one machine to directly read from or write to the memory of another machine without any involvement from the remote CPU. For example, machine *A* wants to write into machine *B*’s memory.
 1. Machine *B* registers memory and shares the memory addresses and remote keys with machine *A*.
 2. Machine *A* posts an **RDMA Write**.
 3. The RNIC on machine *A* directly writes the data into the specified memory location on machine *B* without any involvement from machine *B*’s CPU or kernel.

■ **Setting up RDMA Data Channels.** The communication happens through a structure called a **Queue Pair (QP)**, which is the fundamental RDMA communication endpoint. Each QP consists of a **Send Queue (SQ)** and a **Receive Queue (RQ)**. They are created as follow:

1. **Create RDMA Resources:** each side creates a **Protection Domain (PD)**, a **Completion Queue (CQ)**, a **Queue Pair (QP)**, and **Memory Regions (MR)** for the buffers they want to use.
2. **Exchange Connection Info:** each side exchanges the QP number, LIG/GID (address info), PSN (packet sequence number), and memory keys (remote keys for one-sided operations) with the other side. This exchange happens through TCP control channel or some out-of-band mechanism. In other words, **RDMA does not magically discover the remote memory; the applications must explicitly exchange the necessary information to set up the RDMA communication.**
3. **Move QP Through States:** the QP goes through several states (INIT, RTR, RTS) to establish the connection and be ready for communication.

Work Queues. Once the QP is established, we move to the execution phase. First of all, RDMA is **asynchronous**, so when an application posts a verb, it does not block waiting for the operation to complete.

- **Send Queue (SQ):** The application posts work requests (WRs) to the SQ. Each WR describes an operation (e.g., Send, RDMA Write, RDMA Read) and the associated buffers and memory keys. The RNIC consumes entries from the SQ and executes them asynchronously.
- **Receive Queue (RQ):** The application posts receive WRs to the RQ, indicating that it is ready to receive data and providing buffers for incoming data. This is only needed for two-sided operations. For one-sided operations, the remote side does not need to post receive WRs, as the RNIC can directly access the remote memory.

Queue Elements. Now we define the elements inside those queues. There are two important types of elements:

- **Work Queue Elements (WQE)** represents a request submitted by the application to the RNIC. For example, a WQE could represent a Send operation, an RDMA Write, or an RDMA Read. Each WQE contains the necessary information for the RNIC to execute the operation, such as buffer addresses, memory keys, and operation type. The RNIC processes WQEs and executes the corresponding operations on the network.
- **Completion Queue Elements (CQE)** represents a completion notification generated by the RNIC when a WQE is processed. When the RNIC completes a WQE, it generates a CQE and places it in the Completion Queue (CQ). The application can poll or wait on the CQ to receive notifications about completed operations, allowing it to manage resources and handle completions efficiently.

Complete RDMA Execution Flow

1. The application creates RDMA resources (PD, CQ, QP, MR).
2. The application exchanges connection information (QP number, LID /GID, PSN, memory keys) with the remote side.
3. The application moves the QP through states (INIT, RTR, RTS) to establish the connection.
4. The application posts work requests (WRs) to the Send Queue (SQ).
5. The RNIC consumes WQEs from the SQ and executes them asynchronously.
6. When a WQE is completed, the RNIC generates a CQE and places it in the Completion Queue (CQ).
7. The application polls or waits on the CQ to receive notifications about completed operations.

Limitations of RDMA. However, RDMA requires special networks. Common transport protocols for RDMA include:

- **InfiniBand**: A high-speed, low-latency network technology designed for RDMA.
- **RoCE (RDMA over Converged Ethernet)**: An extension of RDMA that runs over standard Ethernet networks.
- **iWARP (Internet Wide Area RDMA Protocol)**: An RDMA protocol that runs over TCP/IP networks, allowing RDMA to be used over existing Ethernet infrastructure without requiring special hardware.

So while RDMA provides significant performance benefits, it also requires specialized hardware and network infrastructure, which can be a barrier to adoption in some environments. Also, RDMA requires lossless networks (or congestion control mechanisms) and specialized NICs, which can increase cost and complexity. Additionally, programming with RDMA can be more complex than traditional socket programming, as it requires managing memory registration, queue pairs, and other low-level details.

9 Laboratories

9.1 Introduction to P4 Programming

The main goal of this laboratory is to **introduce the P4 programming language** and make us familiar with:

- How a **programmable switch** is structured,
- How packets are **parsed, processed, and reconstructed**,
- How to implement **simple packet-processing behaviors** in the data plane.

This is **not** about performance or optimization yet, but about **understanding the model**.

❓ **Conceptual goal.** Until now, from theory side, we learned that networks are no longer just “*dumb pipes*” (i.e., simple forwarding devices), but rather *programmable* entities that can be **customized** to implement a variety of functionalities. This lab makes that idea *concrete*.

✖ **Practical goal.** By the end of this lab, we should be able to:

- Read a **P4 program** and understand its structure,
- Identify:
 - Where headers are parsed,
 - Where decisions are taken,
 - Where packets are emitted,
- Write **basic P4 programs** that reflect packets, repeat packets, and handle VLAN tags.

These exercises are intentionally simple so that **the focus is on the language and architecture**, not on complex networking logic.

9.1.1 P4 ecosystem and motivation

Traditional networks are built with **fixed-function devices**, such as routers and switches, which have predefined functionalities determined by their hardware design. This rigidity limits the ability to adapt to new protocols or requirements without replacing the hardware. Supporting new protocols requires new hardware *or* slow firmware updates, which can be costly and time-consuming. Network operators have **little control** over the internal packet-processing logic of network devices, as parsing, matching, and forwarding behaviors are hardcoded by vendors. This lack of programmability hinders innovation and limits the ability to quickly adapt the network to new protocols or application requirements.

💡 Core motivation behind P4

P4 (Programming Protocol-Independent Packet Processors, see page 43) was designed to answer one question: “*how should packets be processed?*” instead of “*which protocol does this switch support?*”. So the motivation is **protocol independence, programmability of the data plane, and fast innovation without changing hardware**.

⌚ Where P4 fits in the ecosystem

Historically, control plane became **very programmable** (e.g., SDN with OpenFlow), while data plane remained **fixed-function**. So we had this mismatch: *very smart control plane vs. very dumb (but fast) data plane*. The control plane could compute complex policies (e.g., routing decisions) but the data plane could only execute a **fixed set of behaviors** (like matching on predefined headers and forwarding accordingly).

P4 is part of a broader shift toward **network programmability**. It lives **entirely in the data plane**, allowing operators (i.e., the control plane) to define how packets are parsed, what fields can be matched, and which actions are possible.

- **Control Plane** → computes policies and installs rules in the network.
- **Data Plane** → executes packet processing at line rate; P4 is used to define its capabilities.

P4 does **not** replace routing protocols, controllers, or orchestration tools. Instead, it gives them **more expressive power**. In other words, P4 only defines **what the data plane is capable of doing**, while the control plane still decides **how to use those capabilities**.

For example, without P4, the control plane can say “*forward packets based on IP prefix*” because that’s all the hardware supports. In contrast, with P4, the control plane can say “*forward packets based on custom headers, application IDs, telemetry metadata, congestion signals, and flow state*”, because **P4 made those operations possible**. So P4 **extends the vocabulary** of the data plane, enabling more sophisticated policies.

✖ Typical P4 ecosystem components

A **P4 ecosystem** is the toolchain and runtime environment that allows developers to describe packet processing, compile it for specific targets, run it on a switch, and control it at runtime. A P4-based system typically consists of the following components:

1. **P4 program**: A high-level description of the data plane, defining packet parsing, match-action tables, and supported actions.

More specifically, a **P4 program** is a **static description of the data plane**. It defines which headers exist, how packets are parsed, which tables exist, and which actions can be executed. However, it **does not contain forwarding rules**; those are installed at runtime by the control plane.

2. **P4 compiler**: Translates the P4 program into a target-specific representation.

? **Why is a compiler needed?** P4 is high-level and target-independent, but switches are hardware- or software-specific. So we need a **compiler** that translates the P4 source code into a target-specific representation. Usually, a compiler checks correctness (e.g., type checking), enforces architectural constraints, and produces artifacts usable by the target and control plane.

3. **P4 target**: A hardware or software switch that executes the compiled P4 program and processes packets.

In other words, the **P4 target** is the **thing that processes packets**. It can be a programmable switch (e.g., Barefoot Tofino), a software switch (e.g., BMv2, used in this lab), or even a network interface card (NIC) with P4 capabilities. The target executes packet processing at line rate and exposes tables to the control plane. However, it does **not decide policies** or does **not compute routes**; it just executes rules installed by the control plane.

4. **Control plane**: An external entity that installs table entries and policies at runtime, typically using **P4Runtime** (page 205).

More specifically, the control plane runs as a **separate program**, communicates with the switch, and installs **table entries** (e.g., match-action rules) at runtime.

The key idea is that the data plane becomes programmable, while the control plane remains responsible for policy decisions.

9.1.2 P4 Architecture

P4 Target vs P4 Architecture

In P4, we **do not program a specific switch directly**. Instead, P4 separates **what can be programmed** from **where it is executed**. This avoids vendor lock-in (i.e., write code for a specific switch, can only run it on that switch) and allows the same P4 program to run on different hardware or software targets. So, a P4 architecture is **target-independent** and defines *capabilities*, not implementations.

 **What is a P4 architecture?** A **P4 architecture** is an **abstract model** that defines:

1. The structure of the packet-processing pipeline (e.g., how many stages, what operations can be performed at each stage).
2. Which **metadata fields** exist (e.g., packet headers, internal state).
3. Which **externs** (i.e., built-in functions or objects) are available (e.g., counters, registers, hash functions).
4. How packets flow through the pipeline (e.g., how packets are parsed, processed, and emitted).

In simple terms, it answers the question: “*what does a programmable switch look like?*”.

 **What is a P4 target?** A **P4 target** is the **actual device or software** that runs a P4 program. For example, a P4 target could be a specific hardware switch (e.g., Barefoot Tofino) or a software switch (e.g., BMv2). In simple terms, it answers the question: “*where does the P4 program actually run?*”.

A P4 target implements a specific architecture, has hardware/software constraints, and executes the compiled P4 program. Different targets may support different architectures, and may have different performance characteristics. So, an **architecture is the contract** and the **target is the implementation**. We write **one P4 program** for an architecture, then compile it for **different targets** that support it.

Architecture of a P4 program

A P4 program describes **how a packet flows through a switch**. This flow is always structured as a **pipeline** with three main stages: parser, match-action pipeline, and deparser. This structure is **not optional**: every P4 program follows it, regardless of target or architecture. So a P4 program is a declarative description of a packet-processing pipeline.

1. **Parser**, *understand the packet*. It is implemented as a **state machine** that reads packet bytes sequentially, extracts headers, and transitions based on header values. This is the first stage of the pipeline, where the raw packet is transformed into a structured format.

The parsing is explicit and programmable in P4. This feature is important because if tomorrow we invent a new custom header, we can define, parse and use it without waiting for hardware support.

The parser defines the **input** and **output** of the pipeline:

- ➔ Input: raw bytes of the packet from the wire (e.g., Ethernet frame).
- ⬅ Output: structured headers and metadata (e.g., Ethernet header, IP header, TCP header).

The purpose is to identify protocol headers and extract fields so they can be used later. If a field is **not parsed**, it **cannot be matched or modified** later.

2. **Match-Action Pipeline**, *decide what to do*. This is the core of the P4 program. It uses extracted fields to applies programmable logic (e.g., look up a routing table, modify headers, update metadata). Its purpose is to match packet fields against tables and execute actions based on those matches (e.g., forward, drop, modify, clone). This is where **all decisions happen**.

This stage defines the **tables**, the **actions** and the **control logic**.

- **Tables**. A **table** is a data structure that stores rules for matching packet fields and specifies actions to execute when a match occurs. Tables are **not hard-coded** in the P4 program; they are **populated at runtime** by the control plane. This separation allows for dynamic updates without changing the P4 program.
- **Actions**. An **action** is a set of operations that can be performed on a packet (e.g., modify a header field, forward to a port, drop the packet). They describe what happens to the packet when a match occurs. Actions are part of the **data plane logic**, not control plane logic.
- **Control logic**. This defines which tables are applied, in what order, and under which conditions. This gives structure to the pipeline.

3. **Deparser**, *rebuild the packet*. It takes (possibly modified) headers and serializes them back to bytes to send out on the wire. It ensures the packet is correctly formatted before transmission.

9.1.3 Control Plane Interaction (`P4Runtime`)

❷ Why do we need a control plane at all?

After compiling and loading a P4 program, the switch knows **how** to process packets, but it does **not know what rules to apply**. In other words, actions and tables are defined, but they are **empty** until the control plane installs entries. So we need a way for the control plane to interact with the switch at runtime, to install rules, update policies, and manage the network.

▀ What is `P4Runtime`?

P4Runtime is a **standard API** that allows a control plane to communicate with a P4 target, configure its table at runtime, and read or update its state. It is target-independent, architecture-aware, and it is designed specifically for P4-programmable devices. It is like a **bridge** between the high-level network logic and low-level packet processing.

✓ **What it does.** Using `P4Runtime`, the control plane can:

- Insert table entries (e.g., match-action rules) into the switch.
- Modify or delete entries as needed (e.g., for dynamic policies).
- Read counters and registers to get telemetry or state information from the switch.
- React to events (e.g., packet-in messages) generated by the switch.

✗ **What it does NOT do.** `P4Runtime` does **not**:

- Change the P4 program logic (e.g., add new headers or tables); that requires recompilation and reloading.
- Modify the parser or the overall data plane architecture; those are fixed by the P4 program.
- Redefine tables or actions; it can only populate them with entries defined by the P4 program.

Those are **compile-time decisions**.

9.1.4 Exercise 1: Packet Reflector

In this first exercise, we will implement a simple packet reflector in P4. The network topology consists of two hosts connected to a single P4-programmable switch.

The goal of the network topology is to create the **simplest possible network** that lets us observe packet behavior **inside the switch**. Conceptually, the topology is:

Host A \leftrightarrow Switch \leftrightarrow Host B

- Two hosts connected through a single P4 switch.
- No routing or complex forwarding logic is needed; the switch will simply reflect packets back to the sender.
- Just packet ingress and egress processing to observe how packets are handled within the switch.

This simple setup is intentional to allow us to focus on **packet processing** and **not on complex network behavior**. It allows us to send a packet from a host, observe how the switch processes it, and see the packet come back to the sender, effectively reflecting it. If the packet is correctly reflected, so that Host A sends a packet and receives the same packet back, we can confirm that our P4 program is correctly processing packets at the switch level.

❓ What is a “*packet reflector*”?

A **packet reflector** is a switch that receives a packet and sends it back to the ingress port. So a packet comes in on port p , and the switch sends it back out on the same port p . No routing, no learning, no forwarding tables, just a simple reflection of the packet back to the sender. This allows us to observe how packets are processed within the switch without any additional complexity.

❓ What is the purpose of the P4 program?

The P4 program:

1. Parses the packet.
2. Stores the ingress port in metadata.
3. Sets the egress port equal to the ingress port (to reflect the packet back).
4. Emits the packet.

That's it! But conceptually, this uses **all three pipeline stages**:

1. **Parser**: extracts ethernet header and makes packet fields available (even if unused).
2. **Match-Action pipeline**: executes an action, for example:

```
egress_port = ingress_port
```

3. **Deparser:** emits the packet back out.

② What we need for this exercise?

Inside `lab_1/01-PacketReflector` directory, we have:

- `packet_reflector.p4`: is the **data plane program**. It is the incomplete P4 program that we need to fill in. The **P4 target** is the **BMv2 software switch**, which will run our P4 program and process packets according to the logic we define. So this file defines the **capabilities** of the switch, not the dynamic rules (which are defined in the control plane).

Deepening: What is BMv2?

BMv2 (Behavioral Model v2) is a **software implementation of a P4 switch**, used mainly for **testing, teaching, and prototyping**.

③ **Why BMv2 exists?** Real P4 switches (e.g., Tofino) are expensive, vendor-specific, not easy to experiment with, and not suitable for learning. So the P4 community needed a switch that behaves like a real P4 switch, but runs on a normal computer. **That's BMv2:** it allows us to write P4 programs and test them in a software environment before deploying them on real hardware.

④ **What BMv2 actually is?** BMv2 is a **software switch** that runs as a **user-space program** on a normal computer and **executes a P4 program packet by packet**. It implements a P4 **architecture model** (in our laboratory, we use the **V1Model architecture**) and processes packets according to the logic defined in the P4 program. It is not a real switch, but it behaves like one for the purposes of testing and learning P4 programming. So BMv2 is a **P4 target**, not an architecture.

⑤ **What BMv2 does when a packet arrives?** When a packet reaches the BMv2 switch:

1. BMv2 calls the **parser** defined in our P4 program to extract packet headers and fields.
2. Executes the **match-action pipeline** defined in our P4 program, which can modify packet metadata and determine the egress port.
3. Runs the **deparser** to emit the packet back out according to the logic we defined.
4. And finally, BMv2 sends the packet out on the specified egress port.

Conceptually identical to how a real P4 switch would process packets, but all in software.

Deepening: What is the V1Model architecture?

V1Model is a **P4 architecture** that defines the abstract structure, metadata, and interfaces of a simple programmable switch. It is a **reference architecture** used for learning and prototyping P4 programs.

② **Why do we need something like V1Model?** P4 is **architecture-independent** by design. Itself does **not** define how many pipeline stages exist, which metadata fields exist, and how packets enter/exit the switch. So we need a **contract** that says: “if we write a P4 program for *this* architecture, the switch will provide *these* components”. That contract is an **architecture model**.

③ **What does V1Model define?** V1Model defines:

- The **packet-processing pipeline structure** (parser, match-action tables, deparser).
- The **standard metadata** fields. It defines a struct called `standard_metadata_t` containing fields like: `ingress_port`, `egress_spec`, `egress_port`, `packet_length`, `drop`. These fields **exist only because v1model defines them**. If we were using a different architecture, these fields might not exist or might have different names.
- The **externs** available. V1Model defines extern objects such as registers, counters, meters and checksum helpers. These are **black-box primitives** provided by the target (BMv2 in our case) that we can use in our P4 program.

For example, the `ingress_port` field is part of the standard metadata defined by v1model, and BMv2 will automatically populate it with the port number where the packet arrived.

❖ **Relationship: v1model vs BMv2.** The v1model is the **architecture** and defines *what exists* in the switch, while BMv2 is the **target** that implements that architecture. So when we write a P4 program for v1model, we know that BMv2 will provide the components defined by v1model, and we can use them in our program. The architecture defines the **capabilities** of the switch, and the target provides the **implementation** of those capabilities.

- `network_topo.py`: the incomplete topology definition that we need to fill in. It is a Python script that uses Mininet and P4-Utils to:
 - Create hosts and switches,
 - Connect them with links,
 - Load the P4 program into the switch,
 - Enable logging, pcap dumps, and CLI.

We run this script to set up the network topology and start the BMv2 switch with our P4 program loaded. It will create the hosts, connect them to the switch, and prepare the environment for testing.

Deepening: What is Mininet?

Mininet is a **network emulator** that lets us **create virtual networks** (hosts, switches, links) **on a single machine**.

❓ **What does “network emulator” mean?** Mininet creates **real network elements**, but virtualized. It uses Linux namespaces and virtual Ethernet interfaces to create isolated hosts and switches that can communicate with each other as if they were on a real network. So when we create a host in Mininet, it is a real Linux process with its own network stack, and when we create a switch, it is a real software switch (like BMv2) that processes packets.

❓ **What Mininet actually gives us?** With Mininet, we can create: hosts, switches, links between them, IP/MAC configurations, terminals, packet capture and logging. All **without physical hardware**. It is important to note that Mininet does **not** process packets itself; it only provides the **environment**.

- `test.py`: the test script that sends packets and checks if they are reflected back correctly. It is a Python script that runs on the host, sends packets, and checks if packets are reflected. It uses Scapy library to craft and send packets, and to sniff for incoming packets. We run this script after the topology is set up to verify that our P4 program is working correctly. It will send a packet from Host A, wait for the reflected packet, and check if it matches the expected behavior.

We'll implement the P4 program so it performs the 4 required steps (Ethernet header, parsing, swap MAC addresses, reflect to ingress port).



[GitHub Repository](#)

9.1.4.1 Build network_topo.py

Before writing code, let's understand what `network_topo.py` is supposed to do. It is a Python script that doesn't implement packet logic, but rather it **creates the virtual network and loads our P4 program into the switch**. So its responsibilities are only:

1. Create hosts and switches,
2. Connect them with links,
3. Tell the switch **which P4 program to run**,
4. Start the network.

The actual packet processing logic is defined in the P4 program, not in this script. So we will write `network_topo.py` to set up the environment, and then we will write the P4 program to define how packets are processed within the switch.

1. **Import the right API.** We use **P4-Utils**, which wraps Mininet for P4 experiments.

```
1 from p4utils.mininetlib.network_API import NetworkAPI
```

The class `NetworkAPI` is a **high-level interface** that abstracts away the details of Mininet and BMv2, allowing us to create hosts, switches, links, and load P4 programs with simple method calls.

2. **Create the network object.** We create an instance of `NetworkAPI` to manage our network.

```
1 network = NetworkAPI()
```

Now we have a **network object** that we can use to create hosts, switches, and links, and to load our P4 program. At this point, we have an empty network with no hosts or switches.

3. **Set log level.** We set the log level to `info` to see more detailed output when we run the script.

```
1 network.setLogLevel('info')
```

This will allow us to see more information about what the script is doing, which is helpful for debugging and understanding the flow of the program.

4. **Enable debugging helpers.** We enable debugging helpers to get more insights into the network behavior.

```
1 net.enablePcapDumpAll()
2 net.enableLogAll()
```

They give us the dump of all packets in pcap format (which we can analyze with Wireshark) and detailed logs of all events in the network. This is crucial for understanding how packets are processed and for debugging our P4 program.

5. **Enable Mininet CLI.** We enable the Mininet CLI to interact with the network after it is set up.

```
1 network.enableCLI()
```

This allows us to run commands on the hosts and switches, inspect the network state, and test our P4 program interactively. Without this, the network would start and exit immediately.

6. **Add the P4 switch.** We add a switch to the network and specify the P4 program it should run.

```
1 switch = network.addSwitch('s1')
```

This creates a **virtual switch node** named **s1** in our network. This switch will later run BMv2 (the software switch) with the P4 program we will define. At this point, we have a switch in our network (P4-capable), but it is not yet connected to any hosts.

7. **Tell the switch which P4 program to use.** We specify the P4 program that the switch should run.

```
1 switch.setP4Source('s1', 'packet_reflector.p4')
```

This line is crucial because it says “switch **s1**, when you start, load and run the P4 program defined in **packet_reflector.p4**”. What happens under the hood:

- (a) P4 compiler is invoked,
- (b) BMv2 is started,
- (c) Compiled program is loaded.

If this line is missing or wrong, the switch runs with **no logic** and packets go nowhere. So this is how we tell the switch what to do with packets.

8. **Add the host.** We create a host and connect it to the switch.

```
1 host = network.addHost('h1')
```

This creates a **virtual Linux host** named **h1** in our network. This host will be able to send and receive packets. At this point, we have a host in our network, but it is not yet connected to the switch.

9. **Connect host and switch.** We create a link between the host and the switch.

```
1 network.addLink(host, switch)
```

This creates a **virtual Ethernet link** between the host **h1** and the switch **s1**. Now the host can send packets to the switch, and the switch can send packets back to the host. This is essential for our packet reflector to work, as we need the host to be able to send packets to the switch and receive the reflected packets back.

10. **Set L2 behavior.** We set the switch to operate in Layer 2 mode.

```
1 switch.setL2()
```

This configures the switch to operate at Layer 2 (Ethernet level), which is appropriate for our packet reflector. It is needed because we are working with Ethernet frames and we want the switch to process packets based on their Ethernet headers. This is important for our P4 program, which will parse Ethernet headers and reflect packets based on that.

11. **Start the network.** Finally, we start the network to run the switch and hosts.

```
1 network.startNetwork()
```

This line starts the Mininet network, which in turn starts the BMv2 switch with our P4 program loaded and the host ready to send packets. After this line, the network is up and running, and we can interact with it using the CLI or run our test script to verify that the packet reflector is working correctly.

The complete `network_topo.py` script will look like this:

```
1 from p4utils.mininetlib.network_API import NetworkAPI
2
3 network = NetworkAPI()
4
5 # Network general options
6 network.setLevel('info')
7 network.enablePcapDumpAll()
8 network.enableLogAll()
9 network.enableCli()
10
11 # Network definition
12 switch = network.addP4Switch('s1')
13 network.setP4Source('s1', 'packet_reflector.p4')
14 host = network.addHost('h1')
15
16 # Assignment strategy
17 network.addLink(switch, host)
18
19 # Nodes general options
20 network.l2()
21 network.startNetwork()
```

9.1.4.2 Build packet_reflector.p4

Now that we have the topology ready, we can start writing the P4 program for our packet reflector. We will complete the implementation of P4 program step by step, following the TODOs outlined in the exercise. Each TODO will guide us through the necessary components of the P4 program, starting with defining the headers and moving on to parsing, processing, and finally emitting packets.

1. TODO 1: Define headers (Ethernet)

```
1 /* -*- P4_16 -*- */
2 #include <core.p4>
3 #include <v1model.p4>
4
5 /*
6 * H E A D E R S
7 */
8
9 struct metadata {
10     /* empty */
11 }
12
13 /* TODO 1: define headers */
```

Our switch receives a packet as **raw bytes**. To do anything meaningful (swap MACs, check EtherType, forward), P4 needs a **typed view** of those bytes. So TODO 1 is where we define:

- (a) **Which headers exist;**
- (b) **Which fields they contain;**
- (c) How the program will refer to them later.

In P4, a **header** is a special type that represents a **protocol header** that **may or may not be present** in a packet. It has an **implicit validity bit** (valid/invalid). The parser will “extract” it, which makes it **valid** and fills its fields from the packet. Later stages (ingress/egress) can safely check/use it.

For our packet reflector, we only need to define the Ethernet header:

```
1 header ethernet_t { ... }
```

It means that our program will only be able to parse and process the Ethernet header. If we receive a packet with an IP header, for example, we won’t be able to parse it and won’t be able to access its fields.

❓ **Which fields should we define in the Ethernet header?** The exercise provides a hint: “the format of the Ethernet header is 6 bytes for the destination MAC, 6 bytes for the source MAC, and 2 bytes for the EtherType”. So we need to define three fields: **dstAddr**, **srcAddr**, and **etherType**. The first two are 48 bits (6 bytes) long, while the last one is 16 bits (2 bytes) long. We can use the built-in type **bit** to define them:

```
1 header ethernet_t {
2     bit<48> dstAddr;
3     bit<48> srcAddr;
4     bit<16> etherType;
5 }
```

We also need to define a `headers` struct that contains all the headers we want to parse. In our case, we only have one header (Ethernet), so it will be simple:

```
1 struct headers {
2     ethernet_t ethernet;
3 }
```

2. TODO 2: parse ethernet header

```
1 /*
2  * P A R S E R
3  */
4 parser MyParser(packet_in packet,
5                 out headers hdr,
6                 inout metadata meta,
7                 inout standard_metadata_t standard_metadata) {
8
9     /* TODO 2: parse ethernet header */
10
11 }
```

In v1model, the parser is a **state machine** that:

- Reads from `packet_in` packet
- “Extracts” headers into `hdr`
- Decides what state to go to next

For this exercise, we only need **one header** (Ethernet), so the parser can be minimal. We can start in a state called `start`, where we will try to extract the Ethernet header. If the extraction is successful, we will transition to a state called `accept`, which means that we have successfully parsed the packet and can move on to processing it. If the extraction fails (e.g., if the packet is too short), we can transition to a state called `reject`, which means that we will drop the packet.

```
1 parser MyParser(packet_in packet,
2                  out headers hdr,
3                  inout metadata meta,
4                  inout standard_metadata_t standard_metadata) {
5     state start {
6         packet.extract(hdr.ethernet);
7         transition accept;
8     }
9 }
```

Since we only have one header, we don’t need to check the EtherType or anything else to decide what to parse next. We can simply try to extract the Ethernet header and move on.

After parsing, we will have a valid `hdr.ethernet` header and we can access its fields in the ingress/egress processing stages.

Note: in v1model, the “end of parsing” is called `accept` state, and the “parsing failed” state is called `reject`. We don’t need to explicitly define them if we don’t want to do anything special in those cases. If we want to drop packets that fail parsing, we can simply transition to `reject` and the switch will automatically drop them.

3. TODO 3: define packet reflector logic

```

1  /*
2   * I N G R E S S   P R O C E S S I N G
3   */
4  control MyIngress(inout headers hdr,
5                      inout metadata meta,
6                      inout standard_metadata_t standard_metadata)
7  {
8      /* TODO 3: define packet reflector logic */
9 }

```

The ingress control is where we will implement the main logic of our packet reflector. We will:

- Read parsed headers from `hdr.ethernet.*`,
- Swap the source and destination MAC addresses.
- Set the output port to the same port where the packet came from (i.e., reflect it back).

In `v1model`, the final output decision is typically done via:

`standard_metadata.egress_spec`

Which specifies the output port. The code to swap the MAC addresses would look like this:

```

1 bit<48> originalAddr;
2 originalAddr = hdr.ethernet.srcAddr;
3 hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
4 hdr.ethernet.dstAddr = originalAddr;

```

And to reflect the packet back to the input port, we can use:

```

1 standard_metadata.egress_spec = standard_metadata.ingress_port
;
```

We put it all together in the `MyIngress` control:

```

1 control MyIngress(inout headers hdr,
2                     inout metadata meta,
3                     inout standard_metadata_t standard_metadata)
4 {
5     apply {
6         bit<48> originalAddr;
7         originalAddr = hdr.ethernet.srcAddr;
8         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
9         hdr.ethernet.dstAddr = originalAddr;
10        standard_metadata.egress_spec = standard_metadata.
11            ingress_port;
12    }
13 }

```

The `apply` block is where we put the actual processing logic. In this case, we simply swap the MAC addresses and set the output port to reflect the packet back.

4. TODO 4: deparse header

```
1 /*  
2  * D E P A R S E R  
3  */  
4 control MyDeparser(packet_out packet, in headers hdr) {  
5     apply {  
6         /* TODO 4: deparse header */  
7     }  
8 }
```

The deparser is responsible for taking the processed headers and emitting them back as raw bytes to be sent out of the switch. In our case, we only have one header (Ethernet), so we just need to emit it in the correct order. The code would look like this:

```
1 control MyDeparser(packet_out packet, in headers hdr) {  
2     apply {  
3         packet.emit(hdr.ethernet);  
4     }  
5 }
```

The `packet.emit()` function takes care of serializing the header fields back into raw bytes. Since we only have one header, we just emit it directly. If we had multiple headers, we would need to emit them in the correct order (e.g., Ethernet first, then IP, etc.).

The final P4 program will be a combination of all the components we defined in the TODOs, and it will implement a simple packet reflector that swaps the source and destination MAC addresses and reflects the packet back to the input port.



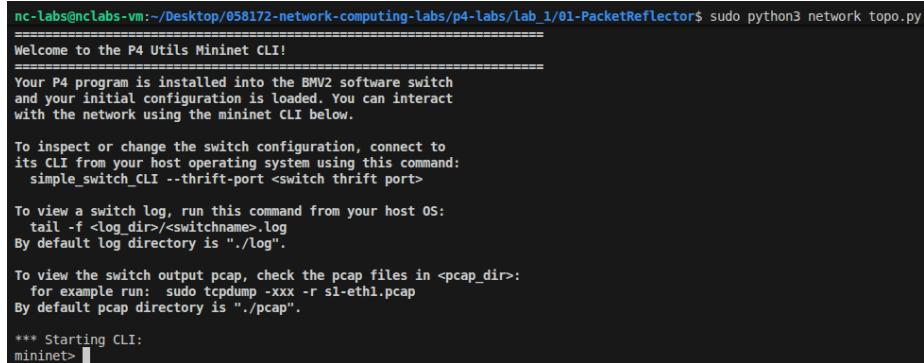
[View the code on GitHub](#)

9.1.4.3 Test P4 program

To test the P4 program, we can type the following command in the terminal:

```
1 sudo python3 network_topo.py
```

This command will start the Mininet network topology defined in the `network_topo.py` file, which includes the P4 switch running the packet reflector program. Once the topology is up and running, we can use Mininet's CLI to interact with the hosts and test the packet reflection functionality.



```
nc-labs@nclabs-vm:~/Desktop/058172-network-computing-labs/p4-labs/lab_1/01-PacketReflector$ sudo python3 network_topo.py
=====
Welcome to the P4 Utils Mininet CLI!
=====
Your P4 program is installed into the BMV2 software switch
and your initial configuration is loaded. You can interact
with the network using the mininet CLI below.

To inspect or change the switch configuration, connect to
its CLI from your host operating system using this command:
  simple_switch_CLI --thrift-port <switch thrift port>

To view a switch log, run this command from your host OS:
  tail -f <log dir>/<switchname>.log
By default log directory is "./log".

To view the switch output pcap, check the pcap files in <pcap_dir>:
  for example run: sudo tcpdump -xxx -r s1-eth1.pcap
By default pcap directory is "./pcap".

*** Starting CLI:
mininet> 
```

Figure 14: Mininet CLI for testing the P4 program.

For example, we can use the following commands in the Mininet CLI to test the connectivity between the host `h1` and the switch `s1`:

```
1 mininet> h1 ping s1
```

However, after running the network topology, we will write `xterm h1` to open a terminal for host `h1`. In this terminal, we can run the following command to test the packet reflection functionality:

```
1 python3 test.py
```

This command will execute the `test.py` script, which sends a packet from host `h1` to the switch `s1` and checks if the packet is correctly reflected back to the host. If the test is successful, we should see a message indicating that the packet was received and reflected correctly. If there are any issues with the P4 program, we may see error messages or the test may fail, indicating that the packet was not reflected as expected. A correct output will look like this:

```
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs/lab_1/01-PacketReflector# python3 test.py
Press the return key to send a packet:
Sending on interface h1-eth0 to 10.0.0.2

[!] A packet was reflected from the switch:
[!] Info: 00:01:02:03:04:05 -> 00:00:0a:00:00:01
```

Figure 15: Output of the test script for the packet reflector P4 program.

9.1.5 Exercise 2: Packet Repeater

The goal of **Exercise 2** is to build a small network with:

- Two **hosts** (**h1**, **h2**).
- One **P4 switch** (**s1**).
- A **control switch** that programs the P4 switch at runtime using **P4Runtime**.

The P4 switch must behave as a **packet repeater**:

- When a packet enters **port 1**, it must be forwarded to **port 2**.
- When a packet enters **port 2**, it must be forwarded to **port 1**.

So, unlike Exercise 1 (page 206):

- Forwarding is **not hard-coded** in the P4 program
- Forwarding decisions are installed **by the controller** using **P4Runtime**

❷ What files are given?

The exercise provides the following files:

- **network_topo.py** (topology and deployment). This file is responsible for **creating and starting the network**. It will contain 1 P4 switch (**s1**), 2 hosts (**h1**, **h2**), and the necessary links.
- **packet_repeater.p4** (data plane program). This file defines **what the switch is capable of doing**, not the actual forwarding rules.
- **control_plane.py** (runtime control logic). This file is the **control plane** of the network. It connects to the P4 switch using **P4Runtime** and installs rules. Here is where you specify the actual forwarding policy (e.g., “packets entering port 1 should be forwarded to port 2”).
- **send.py** (traffic generator). This file is used to **send packets from a host**. It runs on **h1**, crafts an Ethernet plus IP packet, and sends it to a destination IP (e.g., **h2**) using Scapy. It does **not** interact with P4 directly. Generates traffic to test the network. This file is **complete** and should not be modified.
- **receive.py** (traffic observer). This file is used to **observe packets arriving at a host**. It runs on **h2**, sniffs packets on **eth0**, and prints Ethernet, IP, and payload fields. It filters out packets sent by itself. Verifies whether forwarding worked correctly. This file is **complete** and should not be modified.

9.1.5.1 Build network_topo.py

As we have seen in Exercise 1 (page 206), the `network_topo.py` file is responsible for creating and starting the network. In Exercise 2, we will build a slightly more complex topology with:

- Two **hosts** (`h1`, `h2`).
- One P4 **switch** (`s1`).
- A **control switch** that programs the P4 switch at runtime using `P4Runtime`.

So, the code will be similar to Exercise 1, but with an additional host and the necessary links to connect everything together. The final topology will look like this:

```

1 from p4utils.mininetlib.network_API import NetworkAPI
2
3 network = NetworkAPI()
4
5 # Network general options
6 network.setLogLevel('info')
7 network.setCompiler(p4rt=True)
8 network.execScript('python control_plane.py', reboot=True)
9
10 # Network definition
11 switch = network.addP4RuntimeSwitch('s1')
12 host1 = network.addHost('h1')
13 host2 = network.addHost('h2')
14 network.setP4Source('s1', 'packet_repeater.p4')
15
16 # Assignment strategy
17 network.addLink(host1, switch)
18 network.addLink(host2, switch)
19
20 # Nodes general options
21 network.l2()
22 network.enablePcapDumpAll()
23 network.enableLogAll()
24 network.enableCli()
25 network.startNetwork()
```

The code is self-explanatory, but there are a few things to note:

② **What is `network.setCompiler(...)`?** In this exercise, the switch must understand `P4Runtime` messages from the control plane, so we need to add **extra compilation artifacts** to the switch. This is done by setting the compiler to `p4rt=True`, which tells the framework to generate the necessary files for `P4Runtime` support. In practice, the compiler generates a simple JSON pipeline description and a `P4Runtime` (protobuf) description of the switch, which are used by the control plane to interact with the switch at runtime.

⚠ **What happens without this line?** The P4 program is compiled only for **data plane execution**, so no `P4Runtime` metadata is generated. This means that a controller **cannot** talk to the switch, and the execution of `control_plane.py` will fail to connect or install rules, resulting in a non-functional network.

② **What is `network.execScript(...)`?** Even if the switch supports `P4Runtime`, the tables start `empty` and no forwarding rules are installed. So someone must connect to the switch and install forwarding rules at runtime. This is the role of the **control plane**, which is implemented in `control_plane.py`.

This line tells the framework to execute the control plane script after the network is started. More precisely:

- The script is executed **after Mininet starts**;
- It runs in the host environment;
- It connects to the switch via `P4Runtime`;
- It installs forwarding rules on the switch to enable packet forwarding between the hosts.
- With `reboot=True`, the script is re-executed every time the network is restarted, ensuring that the control plane is always up and running.

② **Why do we use `addP4RuntimeSwitch` instead of `addP4Switch`?** The main difference is that `addP4RuntimeSwitch` creates a switch that is compatible with `P4Runtime`, meaning it can be controlled at runtime by a control plane. On the other hand, `addP4Switch` creates a switch that is only meant for data plane execution, without any runtime control capabilities. Since we need to interact with the switch at runtime in this exercise, we must use `addP4RuntimeSwitch`.

9.1.5.2 Build control_plane.py

The `control_plane.py` file is responsible for installing the forwarding rules on the switch. It uses the P4Runtime API to communicate with the switch and configure it according to the P4 program we have written.

The skeleton code provided in `control_plane.py` is as follows:

```

1 #!/usr/bin/env python3
2
3 from p4utils.utils.helper import load_topo
4 from p4utils.utils.sswitch_p4runtime_API import
5     SimpleSwitchP4RuntimeAPI
6
7 topo = load_topo('topology.json')
8 controllers = {}
9
10 for switch, data in topo.get_p4rtswitches().items():
11     controllers[switch] = SimpleSwitchP4RuntimeAPI(
12         data['device_id'],
13         data['grpc_port'],
14         p4rt_path=data['p4rt_path'],
15         json_path=data['json_path']
16     )
17
18 controller = controllers['s1']
19
20 # TODO: write the forwarding rules for the switch
21 controller.table_add('repeater', 'forward', ['1'], ['2'])

```

- The code starts by importing the necessary modules and loading the topology from the `topology.json` file. The JSON file is generated by the P4-Utils when the topology is started, and it contains all the information about the switch names, their device IDs, gRPC ports, and paths to the compiled artifacts.
- We then create a dictionary called `controllers` to store the P4Runtime API instances for each switch. In other words, it maps each switch name to an instance of `SimpleSwitchP4RuntimeAPI`, which is initialized with the corresponding device ID, gRPC port, and paths to the P4Runtime and JSON files. This allows us to easily interact with each switch later on when we need to install rules.
- We connect to each P4Runtime switch by creating an instance of `SimpleSwitchP4RuntimeAPI` for each switch defined in the topology. The function `get_p4rtswitches` returns all switches that were created with `addP4RuntimeSwitch` (so they speak P4Runtime); for each one, we create a connection object that knows how to reach the switch (gRPC port) and what pipeline it is running (paths to the compiled P4 artifacts).

So after this loop, if we have 3 switches, we will have 3 entries in the `controllers` dictionary, each one ready to send commands to its respective switch.

- Finally, we pick the switch controller for `s1` and store it in the variable `controller`. This is the switch we will be configuring in this exercise, as it is the one that will be responsible for forwarding packets between the hosts.

About the TODO, it asks us to write the forwarding rules for the switch. This means that we need to use the P4Runtime API to install the appropriate entries in the match-action tables defined in our P4 program. It can be done by calling methods on the `controller` object, called `controller.table_add` to add an entry to the table. In our case, we will need to add rules to the `repeater` table that we defined in the P4 program, which matches on the ingress port and decides whether to forward or drop the packet. For example, if we want to forward packets coming from port 1 to port 2, we would add a rule like this:

```
1 controller.table_add('repeater', 'forward', ['1'], ['2'])
```

This command tells the switch that if a packet arrives on port 1, it should execute the `forward` action with the parameter 2, which means to forward the packet to port 2. Now we write the P4 program that defines the `repeater` table and the corresponding actions.

9.1.5.3 Build packet_repeater.p4

The `packet_repeater.p4` file is the P4 program that defines the behavior of the switch. It specifies how packets are processed, what headers are parsed, and what actions are taken based on the packet contents.

- The skeleton code provided in `packet_repeater.p4` can be found in the GitHub repository:



[View on GitHub](#)

- **Define headers, parsers and deparser**

```

1  /* -- P4_16 -*- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  /*
6   * H E A D E R S
7   */
8
9  struct metadata {
10 }
11
12 header ethernet_t {
13     bit<48> dstAddr;
14     bit<48> srcAddr;
15     bit<16> etherType;
16 }
17
18 struct headers {
19     ethernet_t ethernet;
20 }
21
22 /*
23 * P A R S E R
24 */
25
26 parser MyParser(packet_in packet,
27                  out headers hdr,
28                  inout metadata meta,
29                  inout standard_metadata_t standard_metadata) {
30     state start {
31         packet.extract(hdr.ethernet);
32         transition accept;
33     }
34 }
```

The header and the parser are the same as in the previous exercise. We define an Ethernet header with the standard fields (destination MAC, source MAC, and EtherType), and the parser simply extracts this header from the incoming packet. The parser is very simple: it just extracts the Ethernet header and then transitions to the accept state, which means that the packet is ready to be processed by the ingress control.

Also the deparser is the same as before, it just emits the Ethernet header back out:

```

1  /*
2   * D E P A R S E R
3   */
4
5 control MyDeparser(packet_out packet, in headers hdr) {
6     apply {
7       packet.emit(hdr.ethernet);
8     }
9 }
```

The deparser takes the processed headers and emits them back as raw bytes. Since we only have one header (Ethernet), we just emit it directly.

• Ingress Processing

```

1  /*
2   * I N G R E S S   P R O C E S S I N G
3   */
4 control MyIngress(inout headers hdr,
5                     inout metadata meta,
6                     inout standard_metadata_t standard_metadata)
7 {
8
9  **** Actions ****
10
11    action forward(bit<9> port) {
12      standard_metadata.egress_spec = port;
13    }
14
15    action drop() {
16      mark_to_drop(standard_metadata);
17    }
18
19  **** Table ****
20
21  table repeater {
22    key = {
23      standard_metadata.ingress_port : exact;
24    }
25    actions = {
26      forward;
27      drop;
28    }
29    // If no rule matches, drop (safe default)
30    default_action = drop();
31    size = 16;
32  }
33
34  apply {
35    repeater.apply();
36  }
}
```

The ingress control is where the main logic of our packet repeater resides. We define two actions: `forward`, which sets the egress port to forward the packet, and `drop`, which marks the packet to be dropped.

We also define a table called `repeater` that matches on the ingress port of the packet. The table has two possible actions: `forward` and `drop`. If

no rule matches, the default action is to drop the packet, which is a safe default to prevent unintended forwarding.

In the apply block, we simply apply the `repeater` table, which means that for each incoming packet, we will look up the ingress port in the table and execute the corresponding action (either forward or drop).

The complete code for the `packet_repeater.p4` file can be found in the GitHub repository:



[View on GitHub](#)

To test the P4 program, we can use the same commands as in the previous exercise:

- Run the network topology with the command:

```
1 sudo python3 network_topo.py
```

- After running the network, we open two terminals for the hosts and run the control plane script in the main terminal. So in the mininet CLI, we run:

```
1 mininet> xterm h1 h2
```

This will open two xterm windows, one for each host.

- In the host `h2` terminal, we run the command to listen for incoming packets:

```
1 python3 receive.py
```

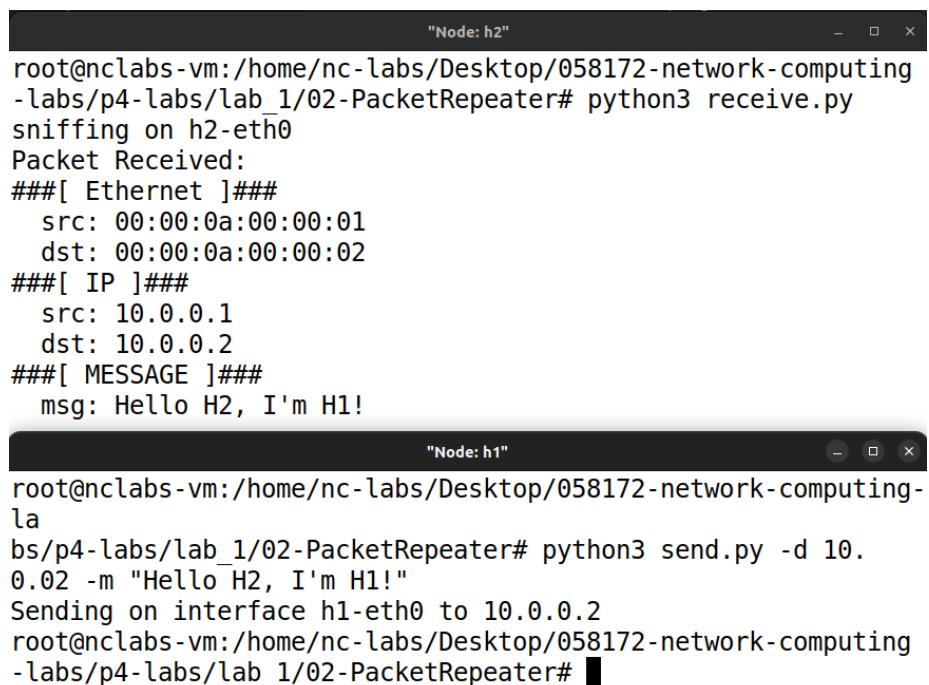
This will start a simple Python script that listens for packets on the host `h2`.

- In the host `h1` terminal, we run the command to send a packet:

```
1 python3 send.py -d 10.0.0.2 -m "Hello H2, I'm H1!"
```

This will send a packet from `h1` to `h2` with the specified destination IP and message.

If everything is set up correctly, we should see the message “*Hello H2, I’m H1!*” appear in the terminal of `h2`, indicating that the packet was successfully forwarded by the switch according to the rules we installed in the control plane.



The image shows two terminal windows. The top window, titled "Node: h2", displays the output of a Python script named "receive.py". It shows a packet being received from node h1, with details like source MAC address (00:00:0a:00:00:01), destination MAC address (00:00:0a:00:00:02), source IP address (10.0.0.1), destination IP address (10.0.0.2), and the message "Hello H2, I'm H1!". The bottom window, titled "Node: h1", displays the output of a Python script named "send.py". It shows a packet being sent to node h2, with details like destination MAC address (00:00:0a:00:00:01), source IP address (10.0.0.2), and the message "Hello H2, I'm H1!". Both windows are running on a Linux system with root privileges.

```
"Node: h2"
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs/lab_1/02-PacketRepeater# python3 receive.py
sniffing on h2-eth0
Packet Received:
###[ Ethernet ]###
    src: 00:00:0a:00:00:01
    dst: 00:00:0a:00:00:02
###[ IP ]###
    src: 10.0.0.1
    dst: 10.0.0.2
###[ MESSAGE ]###
msg: Hello H2, I'm H1!

"Node: h1"
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs/lab_1/02-PacketRepeater# python3 send.py -d 10.0.0.2 -m "Hello H2, I'm H1!"
Sending on interface h1-eth0 to 10.0.0.2
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs/lab_1/02-PacketRepeater#
```

Figure 16: Testing the packet repeater: h1 sends a packet to h2, which receives it successfully.

9.1.6 Exercise 3: VLAN Handler

In this exercise, we will build a VLAN handler. In general, a **VLAN handler** is a piece of switch logic that **adds, removes, checks, and uses VLAN tags to control where packets go**. In other words, it is the part of a switch that understands **Virtual LANs (VLANs)** and enforces VLAN-based isolation and forwarding rules.

Remark: What is a VLAN?

A **VLAN (Virtual LAN)** is a way to split one physical network into multiple **logical networks**, using a small tag inside Ethernet frames. That tag is the **802.1Q VLAN header**, which contains:

- A **VLAN ID (VID)** (12 bits) that identifies which VLAN the frame belongs to.
- A **DEI (Drop Eligible Indicator)** bit that indicates if the frame can be dropped under congestion.
- A **PRI (Priority)** field (3 bits) that indicates the priority of the frame for Quality of Service (QoS) purposes.
- A **TPID (Tag Protocol Identifier)** field (16 bits) that indicates the presence of a VLAN tag (usually set to 0x8100, which is the ethernet type for VLAN-tagged frames).

So a packet can be **untagged** (no VLAN header, just Ethernet + payload) or **tagged** (Ethernet + VLAN header + payload).

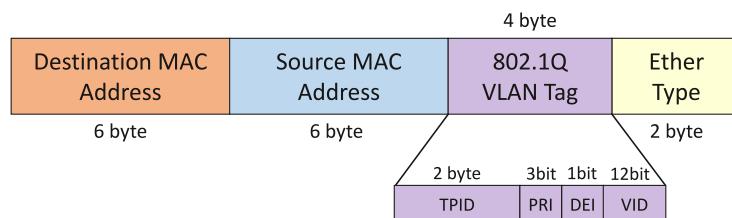


Figure 17: Structure of an Ethernet frame with an 802.1Q VLAN tag. [4]

❷ What does a VLAN handler do?

A VLAN handler sits in a switch and performs **four kinds of tasks**:

1. **Detect whether a packet is VLAN-tagged.** When a packet arrives:
 - Look at the Ethernet type field (**EtherType**).
 - If it is 0x8100, the packet is VLAN-tagged.
 - Otherwise, it is a normal Ethernet frame.

And this is done in the **parser**, because it needs to know how to parse the packet correctly (where the headers are, etc).

2. **Enforce VLAN rules (policy).** Typical rules looks like:

- “Packets from this port **must** be tagged”.
- “Packets from that port **must not** be tagged”.
- “Drop packets that violate the rule”.

This is **validation logic**.

3. **Use the VLAN ID to make forwarding decisions.** Once a packet is known to be VLAN-tagged:

- Extract the VLAN ID from the tag.
- Use it as a **key in a table**.
- Decide which port(s) the packet should go to.

This is VLAN-based forwarding, which is done in the **control plane** (because it is a forwarding decision based on packet metadata).

4. **Add or remove VLAN tags (encapsulation / decapsulation).** A VLAN handler must:

- **Decapsulation:** remove the VLAN tag when sending packets to end hosts.
- **Encapsulation:** add a VLAN tag when sending packets back toward the tagged side.

These tasks explain why it is called *handler*: it doesn’t just forward packets, but it **handles VLAN semantics**.

⌚ What is the behavior we want to implement?

The exercise asks us to build a VLAN handler with the following behavior:

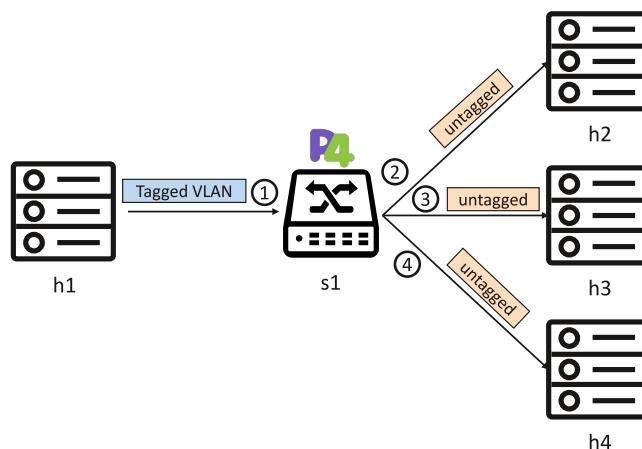


Figure 18: Behavior of the VLAN handler we want to implement. [4]

- The topology is a simple 4-host, 1-switch network (**s1**). Host **h1** is connected to the switch on port 1, and hosts **h2**, **h3**, **h4** are connected on ports 2, 3, 4.
- The switch must enforce these constraints:
 1. **Ingress on port 1 (from h1): must be VLAN-tagged**, Otherwise **drop**.
 2. **Ingress on ports 2/3/4 (from h2/h3/h4): must be untagged**, otherwise **drop**.
 3. For tagged packets from port 1:
 - Read VLAN **VID** from the tag.
 - Consult **control-plane rules**.
 - Forward to the correct host port.
 - **Remove VLAN tag before delivering** (so **h2/h3/h4** see *untagged* packets).
 - If VID not found, **default forward to h2**.
 4. For return traffic (from **h2/h3/h4** to **h1**):
 - Those packets arrive untagged on ports 2/3/4.
 - The switch **adds a VLAN tag** according to the same rules (based on the control-plane rules).
 - And sends them back to **h1** on port 1, where **h1** should receive them **tagged**.

Similar to Exercise 2, we have three files to build: `network_topo.py`, `control_plane.py`, and `vlan_handler.p4`. The first two are similar to Exercise 2, but with different logic. The P4 program is more complex because it needs to handle both tagged and untagged packets, and perform parsing, validation, forwarding, and encapsulation/decapsulation logic. We will go through each file in detail in the next sections.

9.1.6.1 Build network_topo.py

As we have seen in Exercise 2 (page 218), the `network_topo.py` file is responsible for creating and starting the network. In Exercise 3, we will build a slightly more complex topology with:

- Four **hosts** (`h1`, `h2`, `h3`, `h4`).
- One P4 **switch** (`s1`).
- A **control switch** that programs the P4 switch at runtime using `P4Runtime`.

So, the code will be similar to Exercise 2, but with two additional hosts and the necessary links to connect everything together. The final topology will look like this:

```

1 from p4utils.mininetlib.network_API import NetworkAPI
2
3 network = NetworkAPI()
4
5 # Network general options
6 network.setLogLevel('info')
7 network.setCompiler(p4rt=True)
8 network.execScript('python control_plane.py', reboot=True)
9
10 # Network definition
11 switch = network.addP4RuntimeSwitch('s1')
12 host1 = network.addHost('h1')
13 host2 = network.addHost('h2')
14 host3 = network.addHost('h3')
15 host4 = network.addHost('h4')
16 network.setP4Source('s1', 'vlan_handler.p4')
17
18 # Assignment strategy
19 network.addLink(host1, switch)
20 network.addLink(switch, host2)
21 network.addLink(switch, host3)
22 network.addLink(switch, host4)
23
24 # Nodes general options
25 network.12()
26 network.enablePcapDumpAll()
27 network.enableLogAll()
28 network.enableCli()
29 network.startNetwork()
```

The code is self-explanatory, and all the questions about the specific lines are already answered in Exercise 2. The only difference is that we have two more hosts and links, and we set a different P4 source file for the switch. The behavior of the network will be defined by the P4 program and the control plane, which we will see in the next sections.

9.1.6.2 Build control_plane.py

The `control_plane.py` file is responsible for implementing the control plane logic of the VLAN handler. It is more complex than Exercise 2 because it needs to handle both tagged and untagged packets, and perform more complex logic based on the VLAN ID. The code is structured as follows:

```

1 from p4utils.utils.helper import load_topo
2 from p4utils.utils.sswitch_p4runtime_API import
3     SimpleSwitchP4RuntimeAPI
4
5 topo = load_topo('topology.json')
6 controllers = {}
7
8 for switch, data in topo.get_p4rtswitches().items():
9     controllers[switch] = SimpleSwitchP4RuntimeAPI(data['device_id'],
10         data['grpc_port'],
11         p4rt_path=data['p4rt_path'],
12         json_path=data['json_path'])
13
14 controller = controllers['s1']
15
16 controller.table_clear('vlan_table')
17 controller.table_add('vlan_table', 'forward', ['2'], ['2'])
18 controller.table_add('vlan_table', 'forward', ['20'], ['2'])
19 controller.table_add('vlan_table', 'forward', ['3'], ['3'])
20 controller.table_add('vlan_table', 'forward', ['30'], ['3'])
21 controller.table_add('vlan_table', 'forward', ['4'], ['4'])
22 controller.table_add('vlan_table', 'forward', ['40'], ['4'])
23
24 controller.table_clear('port_to_vlan')
25 controller.table_add('port_to_vlan', 'add_vlan_hdr', ['2'], ['2'])
26 controller.table_add('port_to_vlan', 'add_vlan_hdr', ['3'], ['3'])
27 controller.table_add('port_to_vlan', 'add_vlan_hdr', ['4'], ['4'])

```

This code does the following:

- We skip the topology loading part, which is similar to Exercise 2. We load the topology from `topology.json` and create a controller for the switch `s1`.
- For this exercise, we have two tables:
 - `vlan_table`. This table is for traffic **from h1 (tagged side)** to **hosts h2/h3/h4 (untagged side)**. So it maps something (a match key) to the output port. In a VLAN handler, the natural match key is usually the **VLAN ID (VID)** extracted from the VLAN tag, and the action parameter is **egress port**. So conceptually VID → output port.
 - `port_to_vlan`. This is for the **return direction**. Traffic from `h2/h3/h4` to `h1` arrives **untagged**, but the exercise requires that when it goes back to `h1` it is **tagged** with the VLAN corresponding to the source side.
- The rules we add to the tables are as follows:

- For `vlan_table`, we add rules that map VLAN IDs 2, 20 to port 2; VLAN IDs 3, 30 to port 3; and VLAN IDs 4, 40 to port 4. This means that if a packet arrives on port 1 with VLAN ID 2 or 20, it will be forwarded to port 2; if it has VLAN ID 3 or 30, it will be forwarded to port 3; and if it has VLAN ID 4 or 40, it will be forwarded to port 4. If the VLAN ID is not in the table, it will be forwarded to port 2 by default (as per the P4 program logic).
- For `port_to_vlan`, we add rules that say: if a packet arrives on port 2, add VLAN header with VID 2; if it arrives on port 3, add VLAN header with VID 3; if it arrives on port 4, add VLAN header with VID 4. This means that when traffic from `h2` (port 2) goes back to `h1`, it will be tagged with VLAN ID 2; traffic from `h3` (port 3) will be tagged with VLAN ID 3; and traffic from `h4` (port 4) will be tagged with VLAN ID 4.

VLAN IDs (20, 30, 40) represent logical networks used to classify traffic, while port numbers (2, 3, 4) represent physical switch outputs; the control plane maps VLAN IDs to ports to implement VLAN-based forwarding.

9.1.6.3 Build `vlan_handler.p4`

Now that we have defined the behavior we want to implement, we can start building the P4 program. The main file is `vlan_handler.p4`, which will contain the logic for parsing, validating, forwarding, and encapsulating/decapsulating packets based on VLAN tags. We study each section of the P4 program in detail:

- **Headers.** We need to define two headers: the Ethernet header (which is always present) and the VLAN header 802.1Q (which is only present for tagged packets). The Ethernet header contains source and destination MAC addresses, while the VLAN header contains the VLAN ID and other fields.

```

1  /* -- P4_16 -- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  /*
6   * H E A D E R S
7  */
8
9  struct metadata {
10 }
11
12 header ethernet_t {
13     bit<48> dstAddr;
14     bit<48> srcAddr;
15     bit<16> etherType;
16 }
17
18 header vlan_tag_t {
19     bit<3> pcp;
20     bit<1> dei;
21     bit<12> vid;
22     bit<16> etherType;
23 }
24
25 struct headers {
26     ethernet_t ethernet;
27     vlan_tag_t vlan;
28 }
```

- **Parser: detect “tagged vs untagged”.** The parser does exactly the VLAN detection logic:

1. Always extract Ethernet;
2. And if Ethernet type (`etherType`) is 0x8100, extract the VLAN tag as well.

```

1 /*
2  * P A R S E R
3  */
4
5 parser MyParser(packet_in packet,
6                  out headers hdr,
7                  inout metadata meta,
8                  inout standard_metadata_t standard_metadata) {
```

```

9   state start {
10    packet.extract(hdr.ethernet);
11    transition select(hdr.ethernet.etherType) {
12      0x8100: parse_vlan;
13      default: accept;
14    }
15  }
16
17  state parse_vlan {
18    packet.extract(hdr.vlan);
19    transition accept;
20  }
21 }
```

- **Ingress: enforce policy + do forwarding + add/remove tags.**

```

1  /*
2  *  I N G R E S S     P R O C E S S I N G
3  */
4
5 control MyIngress(inout headers hdr,
6                     inout metadata meta,
7                     inout standard_metadata_t standard_metadata)
8 {
9
10 // ----- Actions -----
11 action drop() {
12   mark_to_drop(standard_metadata);
13 }
14
15 action forward(bit<9> port) {
16   standard_metadata.egress_spec = port;
17 }
18
19 // Encapsulate: add VLAN header and send to port 1
20 action add_vlan_hdr(bit<12> vid) {
21   // VLAN header becomes present
22   hdr.vlan.setValid();
23
24   // Set VLAN tag fields
25   hdr.vlan.pcp = 0;
26   hdr.vlan.dei = 0;
27   hdr.vlan.vid = vid;
28
29   // Preserve the "inner" EtherType
30   // inside the VLAN header
31   hdr.vlan.etherType = hdr.ethernet.etherType;
32
33   // Ethernet EtherType becomes TPID for VLAN
34   hdr.ethernet.etherType = 0x8100;
35
36   // Always return to h1 on port 1
37   standard_metadata.egress_spec = 1;
38 }
39
40 // ----- Tables -----
41
42 // From h1 (tagged): VID -> output port (2/3/4).
43 // Default -> 2.
44 table vlan_table {
45   key = {
```

```

46         hdr.vlan.vid : exact;
47     }
48     actions = {
49         forward;
50         drop;
51     }
52     // Exercise requirement: unknown VID defaults to h2
53     // => port 2
54     default_action = forward(2);
55     size = 128;
56 }
57
58 // From h2/h3/h4 (untagged):
59 // ingress port -> VLAN to add + send to port 1
60 table port_to_vlan {
61     key = {
62         standard_metadata.ingress_port : exact;
63     }
64     actions = {
65         add_vlan_hdr;
66         drop;
67     }
68     default_action = drop();
69     size = 16;
70 }
71
72 apply {
73     bit<9> in_port = standard_metadata.ingress_port;
74     bool tagged = hdr.vlan.isValid();
75
76     // -----
77     // Enforce tagging policy
78     // -----
79
80     // Port 1 MUST be tagged
81     if (in_port == 1 && !tagged) {
82         drop();
83         return;
84     }
85
86     // Ports 2/3/4 MUST be untagged
87     if (in_port != 1 && tagged) {
88         drop();
89         return;
90     }
91
92     // -----
93     // Direction A: from port 1
94     // -----
95     if (in_port == 1) {
96         // Decide output port based on VLAN ID
97         vlan_table.apply();
98
99         // If we're sending to ports 2/3/4,
100        // we must strip VLAN before delivery
101        // (h2/h3/h4 should receive untagged frames)
102        if (
103            standard_metadata.egress_spec != 1 &&
104            hdr.vlan.isValid()
105        ) {
106            // Restore Ethernet EtherType
107            // to the inner value

```

```

108         hdr.ethernet.etherType = hdr.vlan.etherType;
109
110         // Remove VLAN header
111         hdr.vlan.setInvalid();
112     }
113     return;
114 }
115
116 // -----
117 // Direction B: from ports 2/3/4
118 // -----
119 // Packet is untagged here (enforced above).
120 // Add VLAN header based on ingress port and forward
121 to port 1.
122     port_to_vlan.apply();
123 }
```

The ingress pipeline is where the main logic of the VLAN handler lives.

1. Actions: the “operations” the switch can perform on packets.

We need to define actions for:

- **Dropping packets:** the `drop()` action marks the packet to be dropped by setting a flag in the standard metadata.
- **Forwarding packets:** the `forward(port)` action sets the egress port in the standard metadata, which tells the switch where to send the packet.
- **Adding a VLAN header:** the `add_vlan_hdr(vid)` action constructs a VLAN header with the given VLAN ID, sets the appropriate fields, and prepares the packet to be sent to port 1.

2. Tables: what the control plane can program to make decisions. We need two tables:

- **vlan_table** ($\text{VID} \rightarrow \text{output port}$): this table is used for packets coming from port 1 (tagged). It looks at the VLAN ID and decides which port to forward to (2/3/4). If the VLAN ID is not found, it defaults to forwarding to port 2 (`h2`).
- **port_to_vlan** ($\text{ingress port} \rightarrow \text{VLAN ID to add}$): this table is used for packets coming from ports 2/3/4 (untagged). It looks at the ingress port and decides which VLAN ID to add before sending to port 1. If the ingress port is not found, it drops the packet.

3. Apply block: the logic that processes each packet. The `apply` block contains the actual logic that is executed for each packet:

- First, it checks the ingress port and whether the packet is tagged to enforce the VLAN tagging policy (drop if port 1 is untagged, or if ports 2/3/4 are tagged).
- Then, if the packet is from port 1 (tagged), it applies the `vlan_table` to decide where to forward it, and if it’s going to ports 2/3/4, it removes the VLAN tag before delivery.
- If the packet is from ports 2/3/4 (untagged), it applies the `port_to_vlan` table to add the appropriate VLAN tag and forward it to port 1.

- **Deparser: how to emit the packet out.** The deparser is responsible for emitting the packet out of the switch. In our case, we need to emit the Ethernet header, and if the VLAN header is valid, we emit it as well. The order of emission is important: if the VLAN header is present, it must come after the Ethernet header, otherwise the packet would be malformed.

```

1  /*
2  * D E P A R S E R
3  */
4
5 control MyDeparser(packet_out packet, in headers hdr) {
6     apply {
7         packet.emit(hdr.ethernet);
8         packet.emit(hdr.vlan);
9     }
10 }
```

The code is available here:



[View Code on GitHub](#)

To test the P4 program, we can use the same commands as in the previous exercise:

- Run the network topology with the command:

```
1 sudo python3 network_topo.py
```

- After running the network, we open two terminals for the hosts and run the control plane script in the main terminal. So in the mininet CLI, we run:

```
1 mininet> xterm h1 h2 h3 h4
```

This will open four xterm windows, one for each host.

- In the host **h2** terminal, we run the command to listen for incoming packets:

```
1 python3 receive.py
```

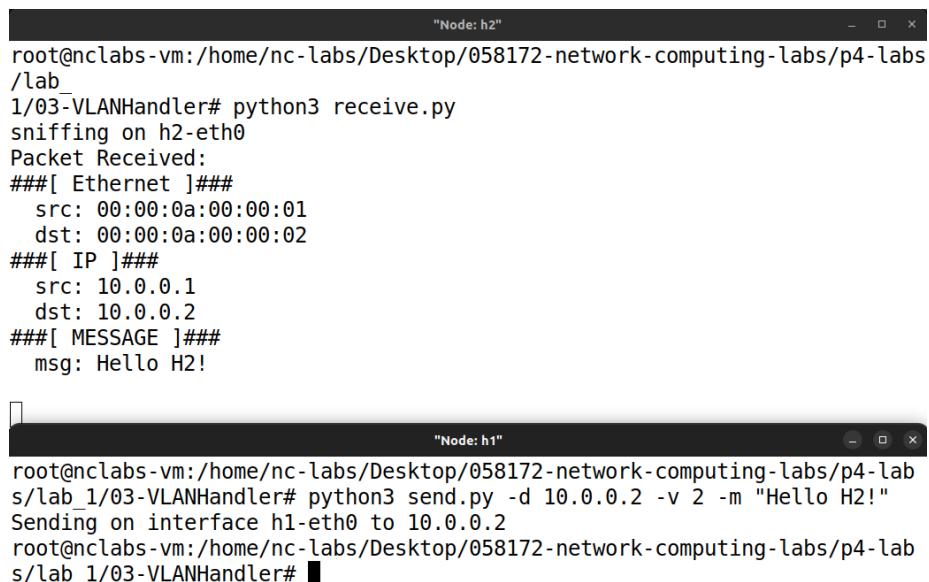
This will start a simple Python script that listens for packets on the host **h2**.

- In the host **h1** terminal, we run the command to send a packet:

```
1 python3 send.py -d 10.0.0.2 -v 2 -m "Hello H2, I'm H1!"
```

This will send a packet from **h1** to **h2** with the specified destination IP, VLAN ID, and message.

If everything is set up correctly, we should see the message “*Hello H2!*” appear in the terminal of **h2**, indicating that the packet was successfully forwarded by the switch according to the rules we installed in the control plane.



The figure consists of two terminal windows. The top window, titled "Node: h2", shows the output of a packet capture on interface h2-eth0. It displays a received Ethernet frame with source MAC 00:00:0a:00:00:01 and destination MAC 00:00:0a:00:00:02. This is followed by an IP header from 10.0.0.1 to 10.0.0.2, and finally a MESSAGE payload "Hello H2!". The bottom window, titled "Node: h1", shows the output of a packet transmission. The command "python3 send.py -d 10.0.0.2 -v 2 -m \"Hello H2!\" " is run, indicating a message is being sent to host 10.0.0.2 via interface h1-eth0.

```
"Node: h2"
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs
/lab_
1/03-VLANHandler# python3 receive.py
sniffing on h2-eth0
Packet Received:
###[ Ethernet ]###
    src: 00:00:0a:00:00:01
    dst: 00:00:0a:00:00:02
###[ IP ]###
    src: 10.0.0.1
    dst: 10.0.0.2
###[ MESSAGE ]###
msg: Hello H2!

"Node: h1"
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-lab
s/lab_1/03-VLANHandler# python3 send.py -d 10.0.0.2 -v 2 -m "Hello H2!"
Sending on interface h1-eth0 to 10.0.0.2
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-lab
s/lab_1/03-VLANHandler#
```

Figure 19: Testing the VLAN handler: h1 sends a tagged packet to h2, which receives it successfully after the switch processes the VLAN tag.

9.2 Stateful Packet Processing in P4

9.2.1 Introduction to Stateful P4 Programs

Most basic P4 programs are **stateless**:

- Each packet is processed **independently** of other packets.
- Decisions depend only on:
 - Packet headers (e.g., Ethernet, IP, TCP).
 - Metadata computed in the current pipeline (e.g., ingress port, packet length).
 - Table entries installed by the control plane (e.g., forwarding rules).

This model is fast and simple, but **not sufficient** for many real network functions. Many network tasks require **memory across packets**, i.e., they need to **Maintain state** about the traffic they see. Examples include:

- Firewalls that track active connections.
- Load balancers that distribute traffic based on server load.

② What is a *stateful* P4 program?

A **Stateful P4 program** is a program that **stores information derived from previous packets** and **uses that stored information** to process future packets. In other words, the data plane *remembers* something about the traffic it has seen, and uses that memory to forward packets differently based on that history.

? **What kind of state do we need to maintain?** Typical examples of state in the data plane include:

- Packet or byte counters (e.g., for traffic monitoring).
- Timestamp of previous packets (e.g., for rate limiting).
- Flow-specific information (e.g., 5-tuple of active connections).
- Thresholds, flags, or temporary values used for decision making.
- Statistics used for monitoring or detection (e.g., number of packets from a source IP).

This state is **persistent across packets, stored inside the switch, and accessed at line rate**.

■ Why stateful programs are important?

Stateful P4 programs enable **advanced network functions** that **cannot be implemented with stateless processing**. For example:

- **Traffic monitoring**: heavy hitter detection, flow counting, latency measurement.

- **Load balancing:** distributing traffic based on server load or connection state.
- **Security functions:** stateful firewalls, DDoS detection, connection tracking.
- **Protocol support:** handling protocols that require state (e.g., TCP connection tracking).

Where is the state stored in a P4 program?

This question is the core of the matter. P4 provides **stateful objects**, which live in the **data plane**, not in the control plane. These objects include:

- **Registers:** arrays of memory cells that can store arbitrary values.
- **Counters:** special registers that can only be incremented or decremented.
- **Meters:** special registers that can be used for rate limiting (not explored in this course).
- **Stateful tables:** tables that can be modified by the data plane itself (e.g., learning MAC addresses, not explored in this course).

These objects are **accessed directly by the data plane** during packet processing, can be read and written by the P4 program, and persist across packets. So they are like **memory** that the data plane can use to store information about the traffic it sees.

Performance considerations

Stateful objects are designed to work at **very high speed**:

- Multiple packets may access registers or counters simultaneously.
- Hardware ensures correctness within architectural limits (e.g., atomic updates, consistency).
- Operations are simple and constrained (e.g., counters can only be incremented, no loops or complex data structures).

These objects are intentionally limited in functionality because the goal is to provide **fast, line-rate access to state** without sacrificing performance, not general-purpose programming capabilities.

9.2.2 Registers in P4

A **register** in P4 is a **stateful memory object** stored in the **data plane**. We can think of it as an **array of hardware**, where each element stores a value of a fixed bit-width, and the array **persists across packets**. Unlike **meta-data**, registers are **not reset per packet** and they survive between pipeline invocations.

Concept	Lifetime	Who writes it?	Purpose
Metadata	Per packet	Data plane	Temporary computation
Header fields	Per packet	Data plane	Packet content
Tables	Per packet	Control plane	Match → action mapping
Registers	Persistent	Data plane	Memory across packets

Table 20: Comparison of different stateful objects in P4. Registers are **the only way** for packets to *directly* update shared state in P4.

② How registers are defined in P4?

Registers are defined **globally**, outside controls:

```
1 register<bit <32> >(1024) pkt_counter;
```

It defines a register array named `pkt_counter` with 1024 entries, each entry stores a 32-bit value `bit<32>`. The entries are indexed from 0 to 1023 using an integer index.

③ Register interface (API) in P4?

Registers are accessed through **extern methods**. The two main methods are:

```
1 register.read(out value, in index);
2 register.write(in index, in value);
```

- `read` reads the value at the specified `index` and stores it in `value` (variable passed by reference).
- `write` updates the value at the specified `index` with the provided `value`.

The operations are **explicit**, meaning that the programmer must call these methods to read or write registers. So there is no `r[i]++` syntax for incrementing a register value; instead, we would read the value, modify it in the control plane, and then write it back. This explicitness is required for hardware mapping and predictable execution.

④ Indirect addressing

Registers support **indirect addressing**, which means that the index used to access the register can be computed at runtime based on packet fields or meta-data:

```

1 bit<32> idx = hash(flow_key);
2 register.read(counter, idx);

```

This is extremely **powerful** because it allows us to **maintain state for different flows** or entities without needing a separate register for each one. For example, we can use a hash of the 5-tuple to index into a register array that counts packets per flow. This flexibility is one of the key advantages of registers in P4, enabling a wide range of stateful applications such as flow counting, rate limiting, and more complex state machines. However, it also requires **careful design to avoid issues like hash collisions** and to ensure that the register size is sufficient for the expected number of flows.

Example 1: Remembering a flow

Suppose we want to count the number of packets for each flow defined by the 5-tuple (source IP, destination IP, source port, destination port, protocol). We can use a register array to store the packet count for each flow. The index into the register can be computed using a hash function on the 5-tuple:

```

1 bit<32> idx;
2 bit<32> count;
3
4 idx = hash(src_ip, dst_ip, src_port, dst_port, proto);
5 flow_reg.read(count, idx);
6 count = count + 1;
7 flow_reg.write(idx, count);

```

In this example, `flow_reg` is a register array defined to hold the packet counts for different flows. We compute the index using a hash of the flow key, read the current count, increment it, and write it back to the register. This is **exactly** what heavy hitters like NetFlow and sFlow do to maintain flow statistics in hardware. However, we must be mindful of hash collisions, which can cause different flows to overwrite each other's counts. To mitigate this, we can use a larger register array or more sophisticated hashing techniques.

⚠ Parallelism and Concurrency

Multiple packets can be processed **in parallel** in the data plane. Two packets may access the same register entry in the same clock cycle. P4 **does not guarantee atomic read-modify-write operations**, unless explicitly supported by the architecture. Therefore, registers are safe for counters and approximate statistics (i.e., they can be incremented without reading the current value), but they may **not** be suitable for complex state machines that require precise updates. For example, if two packets from the same flow arrive simultaneously and both try to update the same register entry, we may end up with a race condition. This is the main reason why P4 is used for *measurement*, not transactional logic.

? Why no loops, pointers, or dynamic memory allocation?

Because registers must map to **hardware SRAM or register files** (i.e., fixed-size arrays), P4 does not support dynamic memory allocation, pointers, or loops that could lead to unbounded execution time. The register size and access patterns must be known at compile time to ensure that the program can be executed at line rate. This design choice allows P4 to achieve high performance while still providing powerful stateful capabilities.

Registers	Counters
General-purpose memory	Specialized objects
Read/Write access	Increment/decrement only
Full control logic	Simpler but safer for concurrency

Table 21: Comparison between registers and counters in P4. Counters are a special type of register that can only be incremented or decremented, which makes them safe for concurrent updates without needing atomic operations.

9.2.3 Exercise 1: Heavy Hitter Detector v1

In this first exercise, we will build a **stateful P4 switch** that:

- Counts how many packets each **source IP** sends,
- And **drops all packets** from that source once a **threshold** is exceeded.

This is a very simple **heavy hitter detector**.

Definition 1: Heavy Hitter

A **Heavy Hitter** is a traffic source (e.g., a host, flow, or application) that contributes a **disproportionately large amount of traffic compared to others**, typically exceeding a predefined threshold in terms of packet count or byte volume over a given observation period.

The network is composed of a single switch **s1** that enforces the heavy hitter detection policy, and three hosts **h1**, **h2**, and **h3** that generate traffic.

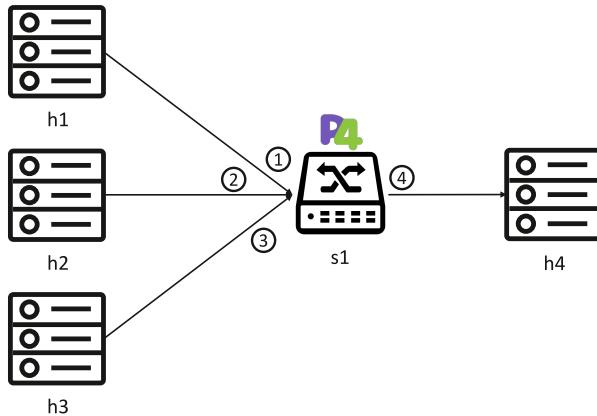


Figure 20: Network topology for Exercise 1.

Before heavy hitter logic, the switch must forward **all packets from h1, h2 and h3 to h4**. This part is **stateless** and can be implemented with a simple **table** that matches on the destination IP address and forwards packets to the correct port (like in the previous sections).

In **v1**, a heavy hitter is defined as a **source IP address** that sends **more than N packets**, where **N is a threshold set by the controller** (i.e., the control plane). This first version of the heavy hitter detector is very simple and has some limitations:

- The **classification key** is only the IPv4 source address. In real scenarios, we might want to consider other fields (e.g., destination IP, ports, protocol) or even combinations of fields (e.g., 5-tuples).
- The **metric** used to identify heavy hitters is the **number of packets**. In practice, we might want to consider other metrics (e.g., number of bytes, flow duration) or even more complex ones (e.g., rate of traffic).

- The threshold is static and set by the control plane. In real scenarios, we might want to have dynamic thresholds that adapt to traffic patterns or use more sophisticated algorithms (e.g., machine learning) to identify heavy hitters.
- The switch only drops packets from heavy hitters and will never accept them again, even if they stop sending traffic for a while. In practice, we might want to have a more flexible policy that allows heavy hitters to recover after some time or based on certain conditions.

❸ **Stateful logic (what memory do we need?)**. This is the first question we should ask ourselves when designing a stateful P4 program: **what state do we need to maintain to implement the desired behavior?** In this case, for each source IP, we must remember:

1. *How many packets have arrived so far?* This is necessary to determine if the source IP is a heavy hitter or not. We can store this information in a register array, where each entry corresponds to a source IP address and contains the packet count.
2. *Whether the threshold has already been exceeded or not?* This is necessary to decide whether to drop packets from that source IP or not. We can store this information in another register array, where each entry corresponds to a source IP address and contains a boolean value indicating if the threshold has been exceeded.

So for **each incoming IPv4 packet**, we need to:

1. Extract source IP address from the packet header.
2. Compute an index from source IP address (e.g., using a hash function) to access the corresponding entries in the register arrays.
3. Read packet counter from register array: `register[index]`.
4. If packet counter is greater than threshold, drop the packet and set the boolean register to true: `register[index] = true`. Otherwise, forward the packet and increment the packet counter: `register[index]++`.

9.2.3.1 Build network_topo.py

The syntax of `network_topo.py` is always the same, and the only thing that changes is the number of hosts, switches, and links. In Exercise 1, we will build a simple topology with:

- Four **hosts** (`h1`, `h2`, `h3`, `h4`).
- One P4 **switch** (`s1`).
- A **control switch** that programs the P4 switch at runtime using `P4RuntimeSwitch`.

```

1 from p4utils.mininetlib.network_API import NetworkAPI
2
3 network = NetworkAPI()
4
5 # Network general options
6 network.setLogLevel('info')
7 network.setCompiler(p4rt=True)
8 network.execScript('python control_plane.py', reboot=True)
9
10 # Network definition
11 switch = network.addP4RuntimeSwitch('s1')
12 host1 = network.addHost('h1')
13 host2 = network.addHost('h2')
14 host3 = network.addHost('h3')
15 host4 = network.addHost('h4')
16 network.setP4Source('s1', 'hdd_v1.p4')
17
18 # Assignment strategy
19 network.addLink(host1, switch)
20 network.addLink(host2, switch)
21 network.addLink(host3, switch)
22 network.addLink(switch, host4)
23
24 # Nodes general options
25 network.l2()
26 network.enablePcapDumpAll()
27 network.enableLogAll()
28 network.enableCli()
29 network.startNetwork()
```

The code is self-explanatory, and all the questions about the specific lines are already answered in the previous exercises. The only difference is that we have more hosts and links, and we set a different P4 source file for the switch. The behavior of the network will be defined by the P4 program and the control plane, which we will see in the next sections.

9.2.3.2 Build control_plane.py

The `control_plane.py` file is responsible for programming the P4 switch at runtime using `P4Runtime`. In Exercise 1, we will implement a simple control plane that programs the switch to detect heavy hitters.

```

1 from p4utils.utils.helper import load_topo
2 from p4utils.utils.sswitch_p4runtime_API import
3     SimpleSwitchP4RuntimeAPI
4
5 topo = load_topo('topology.json')
6 controllers = []
7
8 for switch, data in topo.get_p4rtswitches().items():
9     controllers[switch] = SimpleSwitchP4RuntimeAPI(data['device_id'],
10         data['grpc_port'],
11             p4rt_path=data['p4rt_path'],
12             json_path=data['json_path'])
13 controller = controllers['s1']
14
15 # TODO: write the forwarding rules for the switch
16 controller.table_clear('hhd_threshold')
17 controller.table_add('hhd_threshold', 'set_hhd_threshold', [
18     '10.0.0.1/32'], ['100'])
19 controller.table_add('hhd_threshold', 'set_hhd_threshold', [
20     '10.0.0.2/32'], ['1000'])
21 controller.table_add('hhd_threshold', 'set_hhd_threshold', [
22     '10.0.0.3/32'], ['300'])

```

The code is straightforward: we load the topology, create a controller for each P4 switch, and then program the forwarding rules for the switch. In this case, we are programming the `hhd_threshold` table to set different thresholds for different IP addresses. The behavior of the switch will be defined by the P4 program, which we will see in the next section. The values 100, 1000, and 300 indicate the thresholds for the heavy hitter detection for the respective IP addresses.

9.2.3.3 Build hdd_v1.p4

The `hdd_v1.p4` file is the P4 program that defines the behavior of the switch. In Exercise 1, we will implement a simple heavy hitter detector that uses a register to count the number of packets for each IP address and a table to set the threshold for each IP address.

- **Constants and types**

```

1  /* -- P4_16 -- */
2  #include <core.p4>
3  #include <v1model.p4>
4
5  /*
6   * H E A D E R S
7  */
8
9  #define H4_PORT 4
10 #define HDD_FILTER_ENTRIES 65535
11 const bit<16> TYPE_IPV4 = 0x0800;
12 typedef bit<48> macAddr_t;
```

We define some constants and types that we will use in the program.

- `H4_PORT` is the port number of the host that will receive the heavy hitter notifications.
- `HDD_FILTER_ENTRIES` is the maximum number of entries in the heavy hitter detection table.
- `TYPE_IPV4` is the Ethernet type for IPv4 packets.
- `macAddr_t` is a type for MAC addresses, which are 48 bits long.

- **Header definitions**

```

1 header ethernet_t {
2     bit<48> dstAddr;
3     bit<48> srcAddr;
4     bit<16> etherType;
5 }
6
7 header ipv4_t {
8     bit<4> version;
9     bit<4> ihl;
10    bit<8> diffserv;
11    bit<16> totalLen;
12    bit<16> identification;
13    bit<3> flags;
14    bit<13> fragOffset;
15    bit<8> ttl;
16    bit<8> protocol;
17    bit<16> hdrChecksum;
18    bit<32> srcAddr;
19    bit<32> dstAddr;
20 }
21
22 struct metadata {
23     bit<32> hhd_threshold;
24 }
25
26 struct headers {
```

```

27     ethernet_t ethernet;
28     ipv4_t ipv4;
29 }
```

We define the headers that we will parse from the packets. In this case, we have an Ethernet header and an IPv4 header. We also define a metadata structure that will hold the threshold for the heavy hitter detection; it is **per-packet** metadata (temporary memory for this packet only). It stores the threshold that is fetched from the table action:

- The controller inserts a rule: “for this src IP address, set the threshold to this value”.
- Action `set_hhd_threshold(X)` writes it into the metadata field `hhd_threshold`.

- **Parser: extracting Ethernet and IPv4 headers**

```

1  /*
2  * P A R S E R
3  */
4
5 parser MyParser(packet_in packet,
6             out headers hdr,
7             inout metadata meta,
8             inout standard_metadata_t standard_metadata) {
9     state start {
10         transition parse_ether;
11     }
12
13     state parse_ether {
14         packet.extract(hdr.ethernet);
15         transition select(hdr.ethernet.etherType) {
16             TYPE_IPV4: parse_ip4;
17             default: accept;
18         }
19     }
20
21     state parse_ip4 {
22         packet.extract(hdr.ipv4);
23         transition accept;
24     }
25 }
```

The parser is responsible for extracting the headers from the incoming packets. In this case, we extract the Ethernet header first and then, if the Ethernet type is IPv4, we extract the IPv4 header.

- **Ingress: where the actual heavy hitter detection logic is implemented**

```

1 /*
2 * I N G R E S S   P R O C E S S I N G
3 */
4
5 control MyIngress(inout headers hdr,
6                  inout metadata meta,
7                  inout standard_metadata_t standard_metadata)
8 {
9     register<bit<32>>(HDD_FILTER_ENTRIES) hhd_reg;
```

```

9
10    action drop() {
11        mark_to_drop(standard_metadata);
12    }
13
14    action set_hhd_threshold(bit<32> threshold) {
15        meta.hhd_threshold = threshold;
16    }
17
18    action forward(bit<9> port) {
19        standard_metadata.egress_spec = port;
20    }
21
22    table hhd_threshold {
23        key = {
24            hdr.ipv4.srcAddr: lpm;
25        }
26        actions = {
27            drop;
28            set_hhd_threshold;
29        }
30        default_action = drop();
31    }
32
33    apply {
34        // Check if the packet is IPv4 and
35        // if it comes from one of the first three ports
36        // (the hosts)
37        if (hdr.ipv4.isValid()) {
38            if (
39                standard_metadata.ingress_port > 0 &&
40                standard_metadata.ingress_port < 4 &&
41                hhd_threshold.apply().hit
42            ) {
43                bit<32> hhd_pkts;
44                bit<32> output_hash_ipSrc;
45
46                // Instead of calculating the hash,
47                // we could also emulate the modulo
48                // operation using the following code:
49                // hash(
50                //     output_hash_ipSrc,
51                //     HashAlgorithm.identity,
52                //     0,
53                //     {hdr.ipv4.srcAddr},
54                //     (bit<32>)HDD_FILTER_ENTRIES
55                // );
56                hash(
57                    output_hash_ipSrc,
58                    HashAlgorithm.crc32,
59                    (bit<16>)0,
60                    {hdr.ipv4.srcAddr},
61                    (bit<32>)HDD_FILTER_ENTRIES
62                );
63
64                hhd_reg.read(hhd_pkts, output_hash_ipSrc);
65
66                if (
67                    (meta.hhd_threshold > 0) &&
68                    (hhd_pkts >= meta.hhd_threshold)
69                ) {
70                    drop();

```

```

71         } else {
72             forward(H4_PORT);
73             hhd_reg.write(
74                 output_hash_ipSrc,
75                 hhd_pkts + 1
76             );
77         }
78     }
79 }
80
81 }
82 }
```

- **Register (state)**: we define a register array `hhd_reg` that will hold the packet counts for each IP address. The size of the register is defined by the constant `HDD_FILTER_ENTRIES`.
- **Actions**
 - * **Drop**: the `drop()` action marks the packet to be dropped.
 - * **Set threshold**: the `set_hhd_threshold(bit<32> threshold)` action sets the threshold for the heavy hitter detection in the metadata. This is **how the control plane communicates a parameter** (`threshold`) to the packet processing logic in the data plane.
 - * **Forward**: the `forward(bit<9> port)` action sets the egress port for the packet.
- **Table `hhd_threshold`**: this table matches on the source IP address of the packet and has two possible actions: `drop` and `set_hhd_threshold`. The default action is to drop the packet, which means that if there is no entry for a given source IP address, the packet will be dropped. The control plane will be responsible for inserting entries into this table to set the thresholds for the heavy hitter detection. The key is matched using longest prefix match (LPM), which allows us to set thresholds for individual IP addresses or for subnets.

About the `apply` block:

- We first check if the packet is an IPv4 packet and if it comes from one of the first three ports (the hosts). If these conditions are not met, we do nothing and the packet will be forwarded to the controller by default (as we will see in the egress processing).
- If the conditions are met, we apply the `hhd_threshold` table. If there is a hit, we proceed with the heavy hitter detection logic.
- We calculate a hash of the source IP address to get an index for the register array. This is a common technique to implement a simple counting mechanism without needing to store an entry for each possible IP address. The hash function maps the source IP address to an index in the register array, and we use the register to count the number of packets for that IP address.
- We read the current packet count from the register using the calculated index.

- We compare the packet count with the threshold stored in the metadata. If the packet count is greater than or equal to the threshold, we drop the packet. Otherwise, we forward the packet to the host connected to port H4_PORT and increment the packet count in the register.

The code is available here:



[hdd_v1.p4 on GitHub](#)

To test the P4 program, we can use the same commands as in the previous exercise:

- Run the network topology with the command:

```
1 sudo python3 network_topo.py
```

- After running the network, we open two terminals for the hosts and run the control plane script in the main terminal. So in the mininet CLI, we run:

```
1 mininet> xterm h1 h2 h3 h4
```

This will open four xterm windows, one for each host.

- In the host h2 terminal, we run the command to listen for incoming packets:

```
1 python3 receive.py
```

This will start a simple Python script that listens for packets on the host h2.

- In the host h1 terminal, we run the command to send a packet:

```
1 python3 send.py -d 10.0.0.4 -p 200 -m "Hello H4!"
```

This will send a packet from h1 to h4 with the specified destination IP, number of packets, and message.

If everything is set up correctly, after 100 packets, we should see the packets being dropped by the switch and not reaching h4, which is the host that listens for incoming packets. This is because the threshold for the source IP address of h1 is set to 100 in the control plane, and after 100 packets, the heavy hitter detection logic will drop any additional packets from that source IP address. If we change the threshold in the control plane, we can see how it affects the behavior of the switch and the number of packets that are forwarded to h4.



The image shows two terminal windows side-by-side. The left window, titled "Node: h1", contains the command: "root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs/lab_2/04-HHDv1# python3 send.py -d 10.0.0.4 -p 200 -m \"Hello H2!\"". Below this, the message "Sending on interface h1-eth0 to 10.0.0.4" is displayed. The right window, titled "Node: h4", displays a series of received packets. The output shows multiple "Received from" entries, each followed by a 'Raw' dump and a total count. The counts are 95, 96, 97, 98, 99, and 100. This indicates that 100 packets were sent from h1 to h4, but only 99 reached h4, while one was dropped by the switch.

```
root@nclabs-vm:/home/nc-labs/Desktop/058172-network-computing-labs/p4-labs/lab_2/04-HHDv1# python3 send.py -d 10.0.0.4 -p 200 -m "Hello H2!"
Sending on interface h1-eth0 to 10.0.0.4

"Node: h4"
'Raw') total: 95
Received from ('00:00:0a:00:00:01', '00:00:0a:00:00:04', '10.0.0.1', '10.0.0.4',
'Raw') total: 96
Received from ('00:00:0a:00:00:01', '00:00:0a:00:00:04', '10.0.0.1', '10.0.0.4',
'Raw') total: 97
Received from ('00:00:0a:00:00:01', '00:00:0a:00:00:04', '10.0.0.1', '10.0.0.4',
'Raw') total: 98
Received from ('00:00:0a:00:00:01', '00:00:0a:00:00:04', '10.0.0.1', '10.0.0.4',
'Raw') total: 99
Received from ('00:00:0a:00:00:01', '00:00:0a:00:00:04', '10.0.0.1', '10.0.0.4',
'Raw') total: 100
```

Figure 21: Testing the heavy hitter detector. After sending 100 packets from h1 to h4, the packets are dropped by the switch and do not reach h4, which is the host that listens for incoming packets.

9.2.4 Exercise 2: Heavy hitter detector v2

The second version of the heavy hitter detector is more complex than the first one, but it's also more efficient. The main characteristics of this version are:

- Key: **flow 5-tuple**

```
src IP, dst IP, src port, dst port, protocol
```

- Data structure: **counting Bloom filter** (page 60)
- Algorithm: **probabilistic**
- Threshold: **hard-coded in P4** (1000 packets)
- Two switches and IPv4 forwarding (i.e., the heavy hitter detector is implemented in one switch, and the other switch is used to forward packets to the controller)

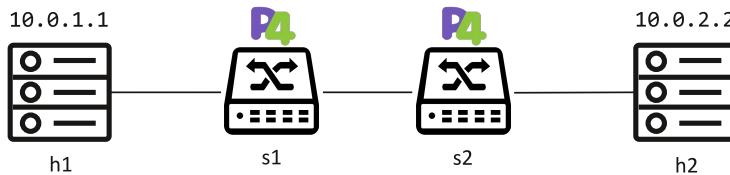


Figure 22: Network topology for Exercise 2.

- Must support **TCP** and **UDP** traffic (i.e., the algorithm must be able to detect heavy hitters for both TCP and UDP flows)

These requirements are more complex than the ones of Exercise 1, but they also allow us to implement a more efficient algorithm.

➊ **What problem v2 is solving?** The main problem that v2 is solving is the scalability of the heavy hitter detector. In Exercise 1, we had to maintain a counter for each flow, which is not scalable. In Exercise 2, we use a counting Bloom filter, which allows us to maintain a compact data structure that can efficiently detect heavy hitters without needing to maintain a counter for each flow.

In real networks, the number of flows is enormous and keeping one counter per flow is impossible. So v2 asks us to detect heavy hitters **approximately**, using **bounded memory**, directly in the data plane. That's why we must use a **counting Bloom filter**.

➋ **Network behavior (what the switches must do)?** Each switch:

1. Forwards packets using an **IPv4 LPM table** (normal routing)
2. Extracts the **5-tuple**
3. Updates a **counting Bloom filter**

4. If the flow **exceeds the threshold** (1000 packets), **drops** the packet
5. Otherwise, decrement TTL, forward packet and update checksum.

This happens **on both switches**.

About the structure of the counting Bloom filter, we have k hash functions, where $k = 2$. So we have 2 independent counters per element. For a flow:

1. Hash the 5-tuple with hash #1 to get index i
2. Hash the 5-tuple with hash #2 to get index j
3. Read counters at index i and j
4. If the flow counter (i.e., the minimum of the two counters) exceeds the threshold, drop the packet
5. Otherwise, increment both counters and forward the packet

P4 implementation

Let the network topology code:

```

1 from p4utils.mininetlib.network_API import NetworkAPI
2
3 net = NetworkAPI()
4
5 # Network general options
6 net.setLogLevel('info')
7 net.setCompiler(p4rt=True)
8 net.execScript('python control_plane.py', reboot=True)
9
10 # Network definition
11 net.addP4RuntimeSwitch('s1')
12 net.addP4RuntimeSwitch('s2')
13 net.setP4SourceAll('./hdd_v2.p4')
14
15 net.addHost('h1')
16 net.addHost('h2')
17
18 net.addLink("h1", "s1", port2=1)
19 net.addLink("s1", "s2", port1=2, port2=2)
20 net.addLink("s2", "h2", port1=1)
21
22 # Assignment strategy
23 net.mixed()
24
25 # Nodes general options
26 net.enablePcapDumpAll()
27 net.enableLogAll()
28 net.enableCli()
29 net.startNetwork()
```

And the control plane code:

```

1#!/usr/bin/env python3
2
3 from p4utils.utils.helper import load_topo
```

```

4 from p4utils.utils.sswitch_p4runtime_API import
    SimpleSwitchP4RuntimeAPI
5
6
7 topo = load_topo('topology.json')
8 controllers = {}
9
10 for switch, data in topo.get_p4rtswitches().items():
11     controllers[switch] = SimpleSwitchP4RuntimeAPI(
12         data['device_id'],
13         data['grpc_port'],
14         p4rt_path=data['p4rt_path'],
15         json_path=data['json_path']
16     )
17
18 controller = controllers['s1']
19
20 controller.table_clear('ipv4_lpm')
21
22 controller.table_add(
23     'ipv4_lpm',
24     'ipv4_forward',
25     ['10.0.1.1/32'],
26     ['00:00:0a:00:01:01', '1']
27 )
28 controller.table_add(
29     'ipv4_lpm',
30     'ipv4_forward',
31     ['10.0.2.2/32'],
32     ['00:00:00:02:01:00', '2']
33 )
34
35 controller.table_set_default('ipv4_lpm', 'drop')
36
37 controller = controllers['s2']
38
39 controller.table_clear('ipv4_lpm')
40
41 controller.table_add(
42     'ipv4_lpm',
43     'ipv4_forward',
44     ['10.0.2.2/32'],
45     ['00:00:0a:00:02:02', '1']
46 )
47 controller.table_add(
48     'ipv4_lpm',
49     'ipv4_forward',
50     ['10.0.1.1/32'],
51     ['00:00:00:02:01:00', '2']
52 )
53
54 controller.table_set_default('ipv4_lpm', 'drop')

```

We can explain the P4 implementation of the heavy hitter detector as follows:

- **Constants**

```

1 /* -*- P4_16 -*- */
2 #include <core.p4>
3 #include <v1model.p4>
4
5 /* CONSTANTS */
6 const bit<16> TYPE_IPV4 = 0x800;

```

```

7 const bit<8> TYPE_TCP = 6;
8 const bit<8> TYPE_UDP = 17;
9
10 #define BLOOM_FILTER_ENTRIES 4096
11 #define BLOOM_FILTER_BIT_WIDTH 32
12 #define PACKET_THRESHOLD 1000

```

The constants are used to define the Ethernet type for IPv4, the protocol numbers for TCP and UDP, the number of entries in the Bloom filter, the bit width of the counters in the Bloom filter, and the packet threshold for detecting heavy hitters.

- **Headers**

```

1 typedef bit<9> egressSpec_t;
2 typedef bit<48> macAddr_t;
3 typedef bit<32> ip4Addr_t;
4
5 header ethernet_t {
6     macAddr_t dstAddr;
7     macAddr_t srcAddr;
8     bit<16> etherType;
9 }
10
11 header ipv4_t {
12     bit<4> version;
13     bit<4> ihl;
14     bit<8> diffserv;
15     bit<16> totalLen;
16     bit<16> identification;
17     bit<3> flags;
18     bit<13> fragOffset;
19     bit<8> ttl;
20     bit<8> protocol;
21     bit<16> hdrChecksum;
22     ip4Addr_t srcAddr;
23     ip4Addr_t dstAddr;
24 }
25
26 header tcp_t{
27     bit<16> srcPort;
28     bit<16> dstPort;
29     bit<32> seqNo;
30     bit<32> ackNo;
31     bit<4> dataOffset;
32     bit<4> res;
33     bit<1> cwr;
34     bit<1> ece;
35     bit<1> urg;
36     bit<1> ack;
37     bit<1> psh;
38     bit<1> rst;
39     bit<1> syn;
40     bit<1> fin;
41     bit<16> window;
42     bit<16> checksum;
43     bit<16> urgentPtr;
44 }
45
46 header udp_t {
47     bit<16> srcPort;
48     bit<16> dstPort;

```

```

49     bit<16> pktLength;
50     bit<16> checksum;
51 }
52
53 struct metadata {
54     bit<32> output_hash_one;
55     bit<32> output_hash_two;
56     bit<32> counter_one;
57     bit<32> counter_two;
58 }
59
60 struct headers {
61     ethernet_t ethernet;
62     ipv4_t ipv4;
63     tcp_t tcp;
64     udp_t udp;
65 }
```

The headers define the structure of the Ethernet, IPv4, TCP, and UDP headers, as well as the metadata that we will use to store the hash values and counters for the Bloom filter.

- **Parser**

```

1 parser MyParser(packet_in packet,
2                     out headers hdr,
3                     inout metadata meta,
4                     inout standard_metadata_t standard_metadata) {
5
6     state start {
7         packet.extract(hdr.ethernet);
8         transition select(hdr.ethernet.etherType){
9
10            TYPE_IPV4: parse_ipv4;
11            default: accept;
12        }
13    }
14
15    state parse_ipv4 {
16        packet.extract(hdr.ipv4);
17
18        transition select(hdr.ipv4.protocol){
19            TYPE_TCP: parse_tcp;
20            TYPE_UDP: parse_udp;
21            default: accept;
22        }
23    }
24
25    state parse_tcp {
26        packet.extract(hdr.tcp);
27        transition accept;
28    }
29
30    state parse_udp {
31        packet.extract(hdr.udp);
32        transition accept;
33    }
34}
```

The parser extracts the Ethernet header, checks if the EtherType is IPv4, and if so, extracts the IPv4 header. Then it checks the protocol field

of the IPv4 header to determine if it is TCP or UDP, and extracts the corresponding header. If the packet is not IPv4, or if the protocol is not TCP or UDP, the parser accepts the packet without extracting any further headers.

- **Ingress processing**

```

1 control MyIngress(inout headers hdr,
2                     inout metadata meta,
3                     inout standard_metadata_t standard_metadata)
4 {
5
6     register<bit<BLOOM_FILTER_BIT_WIDTH>>(BLOOM_FILTER_ENTRIES
7     ) bloom_filter;
8
9     action drop() {
10         mark_to_drop(standard_metadata);
11     }
12
13     action update_bloom_filter(in bit<16> srcPort, in bit<16>
14 dstPort){
15         // get register position
16         hash(
17             meta.output_hash_one,
18             HashAlgorithm.crc16,
19             (bit<16>)0,
20             {
21                 hdr.ipv4.srcAddr,
22                 hdr.ipv4.dstAddr,
23                 srcPort,
24                 dstPort,
25                 hdr.ipv4.protocol
26             },
27             (bit<32>)BLOOM_FILTER_ENTRIES
28         );
29
30         hash(
31             meta.output_hash_two,
32             HashAlgorithm.crc32,
33             (bit<16>)0,
34             {
35                 hdr.ipv4.srcAddr,
36                 hdr.ipv4.dstAddr,
37                 srcPort,
38                 dstPort,
39                 hdr.ipv4.protocol
40             },
41             (bit<32>)BLOOM_FILTER_ENTRIES
42         );
43
44         // read counters
45         bloom_filter.read(
46             meta.counter_one,
47             meta.output_hash_one
48         );
49         bloom_filter.read(
50             meta.counter_two,
51             meta.output_hash_two
52         );
53
54         meta.counter_one = meta.counter_one + 1;
55
56     }
57 }
```

```

53     meta.counter_two = meta.counter_two + 1;
54
55     // write counters
56     bloom_filter.write(
57         meta.output_hash_one,
58         meta.counter_one
59     );
60     bloom_filter.write(
61         meta.output_hash_two,
62         meta.counter_two
63     );
64 }
65
66 action ipv4_forward(macAddr_t dstAddr, egressSpec_t port)
67 {
68     hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
69
70     // set the destination mac address that
71     // we got from the match in the table
72     hdr.ethernet.dstAddr = dstAddr;
73
74     // set the output port that we also get from the table
75     standard_metadata.egress_spec = port;
76
77     // decrease ttl by 1
78     hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
79 }
80
81
82 table ipv4_lpm {
83     key = {
84         hdr.ipv4.dstAddr: lpm;
85     }
86     actions = {
87         ipv4_forward;
88         drop;
89         NoAction;
90     }
91     size = 1024;
92     default_action = NoAction();
93 }
94
95 bit<16> srcPort;
96 bit<16> dstPort;
97
98 apply {
99     if (hdr.ipv4.isValid()) {
100         if (hdr.tcp.isValid() || hdr.udp.isValid()) {
101             if (hdr.tcp.isValid()) {
102                 srcPort = hdr.tcp.srcPort;
103                 dstPort = hdr.tcp.dstPort;
104             } else {
105                 srcPort = hdr.udp.srcPort;
106                 dstPort = hdr.udp.dstPort;
107             }
108             update_bloom_filter(srcPort, dstPort);
109
110         if (
111             meta.counter_one > PACKET_THRESHOLD &&
112             meta.counter_two > PACKET_THRESHOLD
113         ) {

```

```

114         drop();
115         return;
116     }
117 }
118 ipv4_lpm.apply();
119 }
120 }
121 }
```

The ingress processing is where the main logic of the heavy hitter detector is implemented.

– **Actions:**

- * `drop()`: marks the packet to be dropped.
- * `update_bloom_filter()`: updates the Bloom filter counters for the given source and destination ports. It computes two hash values for the flow, reads the corresponding counters from the Bloom filter, increments them, and writes them back to the Bloom filter.
- * `ipv4_forward()`: forwards the packet to the next hop based on the destination IP address. It updates the Ethernet source and destination addresses, sets the egress port, and decrements the TTL.
- **Table ipv4_lpm**: an IPv4 longest prefix match table that matches the destination IP address and forwards the packet accordingly. If there is no match, the default action is to do nothing (i.e., the packet will be dropped later if it is not forwarded).
- **Apply block**: Checks if the packet is IPv4. If it is, it checks if it is TCP or UDP. If it is, it extracts the source and destination ports, updates the Bloom filter, and checks if the flow is a heavy hitter by comparing the counters to the threshold. If the flow is a heavy hitter, it drops the packet. Otherwise, it applies the IPv4 forwarding table.

• **Checksum computation**

```

1 control MyComputeChecksum(inout headers hdr, inout metadata
2   meta) {
3   apply {
4     update_checksum(
5       hdr.ipv4.isValid(),
6       { hdr.ipv4.version,
7         hdr.ipv4.ihl,
8         hdr.ipv4.dsfield,
9         hdr.ipv4.identification,
10        hdr.ipv4.flags,
11        hdr.ipv4.fragOffset,
12        hdr.ipv4.ttl,
13        hdr.ipv4.protocol,
14        hdr.ipv4.srcAddr,
15        hdr.ipv4.dstAddr },
16        hdr.ipv4.hdrChecksum,
17        HashAlgorithm.csum16);
18   }
19 }
```

In the previous exercise, we did not need to compute the checksum because we were not modifying any header fields. In this exercise, we are modifying the TTL field of the IPv4 header, so we need to update the checksum accordingly. The `update_checksum()` function computes the new checksum based on the modified IPv4 header fields. The checksum indicates if the packet has been modified in transit, so it must be updated whenever we modify any of the fields that are included in the checksum computation (in this case, the TTL field).

- **Deparser:** The deparser is responsible for emitting the headers back into the packet after processing. In this case, it emits the Ethernet, IPv4, TCP, and UDP headers in the correct order.

```
1 control MyDeparser(packet_out packet, in headers hdr) {
2     apply {
3         packet.emit(hdr.ethernet);
4         packet.emit(hdr.ipv4);
5         packet.emit(hdr.tcp);
6         packet.emit(hdr.udp);
7     }
8 }
```

The source code of the P4 program is available in the file `hdd_v2.p4` in the repository:



[hdd_v2.p4](#)

9.3 Introduction to eBPF and XDP Programming

9.3.1 Why eBPF in Network Computing

Before touching **syntax**, **maps**, or **XDP**, we need to answer a simple question: *why does eBPF even exist, and why is it relevant for modern networks?*

For a long time, networking research focused on **the network itself**: routers, switches, and the middleboxes (e.g., firewalls, load balancers) that process packets as they traverse the network. But in **datacenters and clouds**, something changed: we control the **end hosts** (servers), the **kernel**, the **NIC**, and we deploy software **much faster** than the traditional hardware upgrade cycles. So the end-host becomes the **most flexible place to innovate**.

For example, think about what happens to a packet when it reaches a server. Every packet must cross:

1. NIC
2. Driver
3. Kernel networking stack
4. Userspace application

If we can intercept packets early, we can drop unwanted traffic, measure flows, rewrite headers or enforce policies. And we can do it **without touching the network fabric**. That's end-host programmability: the **ability to run custom code on the server to process packets as they arrive, directly at the end host**, instead of relying only on fixed kernel or network behavior.

❓ Why eBPF vs kernel modules vs user-space?

Now we have a good motivation for end-host programmability, but why eBPF? Why not just write a kernel module or a user-space program?

- **User-space networking** (e.g., DPDK, netmap) means processing network packets in user-space applications instead of inside the kernel networking stack. In other words, the packet is delivered **to a user program**, and *that program* decides what to do with it.

✓ **Pros:** easy to write and debug, no risk of crashing the kernel, and can leverage user-space libraries and tools.

✗ **Cons:** Packets must cross the kernel boundary to reach the user-space program, which adds latency and overhead. Also, it requires context switching between kernel and user-space, copies of packet data, and may not be suitable for high-performance applications.

It is too slow for high-rate packet processing.

- **Kernel modules** are pieces of code that can be loaded into the kernel to extend its functionality. They run in kernel space and have direct access to kernel data structures and functions.

✓ **Pros:** can achieve high performance since they run in kernel space, and can directly manipulate kernel data structures.

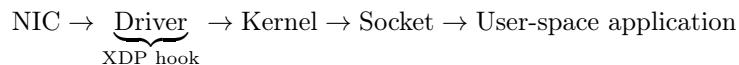
✗ **Cons:** writing kernel modules is complex and error-prone, and a bug can crash the entire system. They also require recompilation and loading into the kernel, which is less flexible than eBPF.

- **eBPF** (extended Berkeley Packet Filter) is a technology that allows **running sandboxed programs in the kernel without the need for kernel modules**. eBPF programs are verified for safety before execution, and they can be attached to various hooks in the kernel, including network events. It was designed to sit exactly in the middle of the spectrum between user-space and kernel modules:

Property	eBPF
Runs in kernel	✓
Safe by design	✓
Dynamically loaded	✓
No kernel patches	✓
Near line-rate performance	✓

❓ So, what does XDP have to do with it?

We will talk about XDP in the next sections, but the short answer is that **XDP (eXpress Data Path)** is a specific use case of eBPF that allows us to run eBPF programs at the earliest point in the packet processing pipeline, right after the NIC receives a packet.



This means we can achieve **line-rate packet processing** and make decisions on packets before they even enter the kernel networking stack, which is crucial for high-performance applications like DDoS mitigation, load balancing, and traffic filtering.

🚧 Performance & Safety Trade-offs

eBPF makes a *deal* with the kernel: it allows us to **run custom code in the kernel**, but it **enforces strict safety checks** to prevent crashes and security issues. This means that while we can achieve near line-rate performance, we also have to work within the constraints of the eBPF verifier, which may reject programs that are too complex or unsafe. However, this trade-off is what makes eBPF a powerful tool for network computing: it gives us the flexibility to innovate at the end host while maintaining the stability and security of the system.

9.3.2 What is eBPF?

Definition 2: eBPF (extended Berkeley Packet Filter)

eBPF (extended Berkeley Packet Filter) is a mechanism that allows user-defined programs to run **safely and efficiently inside the Linux kernel**, in response to specific kernel events.

The previous definition contains **four key ideas** that we will unpack in this section:

1. User-defined programs
2. Inside the Linux kernel
3. Event-driven
4. Safe and fast execution

Now, let's break down each of these ideas to understand what eBPF is and why it is designed the way it is.

From cBPF to eBPF

Originally, the **Berkeley Packet Filter (BPF)** was designed in the early 1990s to solve one simple problem: “*how can we filter packets efficiently inside the kernel for tools like `tcpdump`?*”¹⁹. This was **Classic BPF (Classic Berkeley Packet Filter)**, which was a simple, limited instruction set that could only be used for packet filtering. It was designed to be fast and safe, but it had very limited capabilities. It could only operate on packet data and had a very small instruction set, which made it difficult to use for anything other than simple packet filtering.

Over time, people realized that the packet filtering alone wasn't enough. They wanted to be able to run more complex programs inside the kernel, for a variety of use cases beyond just packet filtering. In particular, they wanted features like: **state, logic, reuse and flexibility**.

In response to this demand, the Linux kernel developers created **eBPF (extended Berkeley Packet Filter)**, which is a much more powerful and flexible mechanism that allows user-defined programs to run safely and efficiently inside the kernel in response to specific events. eBPF is not just for packet filtering anymore; it can be used for a wide range of applications, including performance monitoring, security, and even custom networking features.

¹⁹`tcpdump` is a packet capture and analysis tool that relies on the Berkeley Packet Filter (BPF) to efficiently filter packets inside the kernel before delivering them to user space.

Classic BPF	eBPF
Simple filter	General-purpose
Accumulator-based	Register-based
No state	Persistent maps
Packet filtering only	Generic kernel events
Limited instruction set	Rich instruction set

Table 22: Comparison between Classic BPF and eBPF.

Today, cBPF is still used for simple packet filtering tasks, but eBPF has become the go-to mechanism for more complex and powerful kernel programming. It has opened up a whole new world of possibilities for developers to extend the functionality of the Linux kernel in a safe and efficient way.

✖ eBPF as a sandboxed in-kernel VM

From a technical perspective, what is eBPF? In other words, how does it work? eBPF is **not** native C code, a kernel module, or an arbitrary assembly language. Instead, it is a **virtual machine embedded inside the Linux kernel**, with strict rules and constraints. The key properties of the eBPF virtual machine are:

- **Fixed number of registers:** eBPF programs execute on a register-based virtual machine with a limited set of registers (10 general-purpose registers plus a few special-purpose ones), simplifying verification and preventing unsafe behavior.
- **Bounded stack:** Each eBPF program has access to a fixed-size stack (512 bytes), which eliminates the possibility of stack overflows and enables static analysis by the verifier.
- **No arbitrary memory access:** eBPF programs cannot access arbitrary memory locations; all memory accesses must be performed through verifier-approved pointers (e.g., packet data, maps, or context structures).
- **No system calls:** eBPF programs are not allowed to invoke system calls, preventing them from executing unsafe or blocking operations within the kernel.
- **Controlled interaction via helper functions:** Interaction with kernel subsystems is restricted to a predefined set of helper functions, which provide safe and controlled access to kernel functionality.

This is why eBPF can run **safely inside the kernel** without risking system stability or security.

⌚ Event-driven execution model

Unlike normal programs, eBPF programs **do not run on their own**. They only run **when something happens** (i.e., in response to specific kernel events). This is what we mean by an **event-driven execution model**. Typical events that can trigger eBPF programs include:

- A packet arrives at a network interface (XDP)
- A socket receives data (socket filters)
- A system call is executed (`kprobes`, `uprobes`)
- A tracepoint is hit (tracepoints)

So the model is:

1. Event happens (e.g., packet arrives)
2. Kernel invokes eBPF program associated with that event
3. Program runs
4. Returns a result (e.g., accept/drop packet, log data, update a map)

So there is no `main()` function, infinite loop or background thread. The eBPF program is just a piece of code that gets executed in response to specific events, and then it finishes. This allows eBPF to be very efficient and responsive, since it only runs when needed.

🛡 Safe and efficient execution

The eBPF virtual machine is designed to be **safe and efficient**. The safety comes from the fact that eBPF programs are verified before they are allowed to run. The **Verifier** is a component of the Linux kernel that performs static analysis on the eBPF bytecode to ensure that it adheres to all safety constraints (e.g., no out-of-bounds memory access, no infinite loops, no unsafe helper calls). In other words, the verifier answers the question: “*can this program ever crash or cause instability in the kernel?*” If the **Verifier** determines that the program is **unsafe**, or could potentially cause harm, it will **reject the program** and prevent it from being loaded into the kernel.

Some of the key safety checks performed by the verifier include:

- Every memory access must be within bounds
- All loops must have a bounded number of iterations (or be unrolled)
- Stack usage must be within limits (512 bytes)
- Pointers are tracked precisely to ensure they are valid and safe to dereference
- Execution paths are analyzed to ensure they terminate and do not cause infinite loops

⚠ Despite all these safety checks, eBPF programs can run at near-native speed. How is this possible?

Despite all these restrictions, eBPF is **fast**. But, why?

1. **No syscalls.** The code runs entirely in kernel space, so there is no overhead of crossing the user-kernel boundary. The packet data is already in the kernel, so the eBPF program can access it directly without copying it to user space.
2. **JIT compilation.** The eBPF bytecode is just an intermediate representation. When an eBPF program is loaded into the kernel, it is **Just-In-Time (JIT) compiled** into native machine code for the specific CPU architecture. This means that the eBPF program can run at near-native speed, since it is executing native code instead of interpreted bytecode.

Remark: JIT compilation

JIT (Just-In-Time compilation) is a technique where code is **compiled to native machine instructions at runtime**, instead of being interpreted instruction by instruction.

In the context of eBPF, they are first loaded as **bytecode**, then **translated into native CPU instructions by the kernel**, just before execution.

3. **Runs early.** eBPF programs can be attached to very early points in the kernel's processing pipeline (e.g., XDP runs at the earliest point of packet processing), which allows them to operate on data before it has been fully processed by the kernel, reducing overhead and latency.

In summary, eBPF is a sandboxed, event-driven virtual machine inside the Linux kernel that allows safe, high-performance execution of user-defined programs, verified at load time and often JIT-compiled for efficiency.

9.3.3 eBPF Program Constraints

eBPF programs are powerful, but **not free-form**. They are intentionally constrained so the kernel can **prove** they are safe *before* running them. We can think of these constraints as a **contract**: the kernel agrees to run our code in kernel space **if** we agree to write code that can be statically analyzed and verified to be safe.

⚠ Restricted C

We write eBPF programs in C, but not all C features are allowed. What's missing:

- **No dynamic memory allocation** (e.g., no malloc/free).
- **No recursion** (functions cannot call themselves).
- **No function pointers** (cannot call functions through pointers).
- **No floating point** (e.g., no float/double types).
- **No arbitrary pointer arithmetic** (cannot manipulate pointers like in regular C).
- **Limited standard library** (only a subset of C standard library functions are available).

These restrictions ensure that the eBPF verifier can analyze **all possible execution paths**, track **all memory accesses** and guarantee **termination**. Full C is simply **too expressive** to verify safely, so eBPF uses **C as a syntax**, not as a full programming language.

✖ No unbounded loops

This is one of the most important constraints. **eBPF programs cannot contain loops that the verifier cannot guarantee will terminate.** This means:

- No while:

```
1 while (condition) {
2     // code
3 }
```

- No for:

```
1 for (int i = 0; i < n; i++) {
2     // code
3 }
```

But we can still write loops that the verifier can analyze, such as:

```
1 for (int i = 0; i < 10; i++) {
2     // code
3 }
```

The reason for this restriction is that the **program must be guaranteed to terminate**. If a loop's number of iterations cannot be proven at load time, the verifier rejects the program. This prevents infinite loops, kernel hangs, and CPU starvation. Recent kernels allow **bounded loops**, but only when the bound is statically provable (e.g., a loop that iterates over a fixed-size array). The key point is that the verifier needs to be able to **analyze all possible execution paths** and ensure they all terminate.

⚠ Limited stack size

Each eBPF program has a **fixed stack size of 512 bytes**. This means that there is no heap, the stack cannot grow dynamically, and there is no dynamic memory allocation (i.e., no malloc/free). It is so small because the stack usage must be known at verification time, and deep stacks complicate verification. Additionally, a stack overflow in kernel space would be disastrous, so the kernel enforces this limit to ensure safety and reliability.

☒ Pointer rules

In eBPF **not all pointers are equal**. The verifier tracks pointer *types* and *ranges* (e.g., which memory region they point to). So every pointer access must be bounds-checked and type-checked. For example:

```

1 int *ptr = map_lookup_elem(&my_map, &key);
2 if (ptr) {
3     // Access *ptr safely, knowing it's valid and of the correct
4     // type
5 }
```

Pointer arithmetic must be provably safe and cannot be mixed arbitrarily. This is why we often see patterns like:

```

1 if (ptr + sizeof(struct hdr) > data_end) {
2     return XDP_PASS;
3 }
```

This is not boilerplate; it is **proof** that the pointer access is safe and will not cause a kernel crash. The verifier needs to be able to track all pointer accesses and ensure they are valid, which is why these rules exist.

⌚ How the verifier reasons

The verifier does **not** run our program, it symbolically executes it. It analyzes the code as a **control flow graph** (CFG), tracking all possible paths and states. It uses **abstract interpretation** to reason about variable values, pointer states, and memory accesses. The constraints we discussed are designed to make this analysis tractable and sound, ensuring that the verifier can guarantee safety without needing to execute the code. The verifier checks for:

- **Termination:** All paths must lead to a return statement.
- **Memory safety:** All pointer accesses must be within bounds and of the correct type.

- **Resource limits:** Stack usage and program size must be within limits.

By adhering to these constraints, we can write powerful eBPF programs that run safely in kernel space, enabling us to do things that were previously impossible or unsafe.

9.3.4 eBPF Core Building Blocks

In this section, we will explore the core building blocks of eBPF programs, which include maps, helper functions, and the eBPF instruction set. Understanding these components is crucial for developing efficient and effective eBPF programs.

eBPF Maps

eBPF maps are kernel-resident data structures that allow eBPF programs to **store and retrieve state across multiple executions**.

The problem with eBPF programs is that they are **event-driven** and run **only when an event happens**; once they return, **their stack is gone**. So without maps, every packet would be processed in isolation, no counters, no flow tracking, no connection state, etc. Maps solve this by providing a **persistent state** stored **inside the kernel** and shared across program invocations.

Conceptually, every eBPF map is a **key-value store**, where the key and value have **fixed sizes** defined at load time (i.e., when the eBPF program is loaded into the kernel). Even though maps feel like normal hash tables or arrays, they are actually **kernel objects** accessed only through **helper functions**. The characteristics of maps are as follows:

- **Lifetime:** maps live **independently** of program execution. They exist as long as they are not explicitly destroyed.
- **Sharing:** multiple eBPF programs can share the same map. So user-space programs can access maps created by eBPF programs. This enables a clean split between **data plane** (eBPF program) and **control plane** (user-space application).

 **How do eBPF programs access maps?** eBPF programs **cannot directly deference maps** (i.e., they cannot use normal pointer operations to access map data). Instead, they must use **helper functions** provided by the kernel. This design ensures safety, because helper calls are well-defined in the kernel, which controls synchronization and access patterns. Therefore, the verifier can guarantee that map accesses are safe and do not lead to undefined behavior.

 **Common map types.** Now let's look at the map types we will be using in this course:

- **Array maps** are **fixed-size array where keys are integers from 0 to $N - 1$** . They are very fast, simple and the memory is allocated upfront (i.e., when the map is created). They are ideal for global counters (e.g., total packets processed), configuration values (e.g., sampling rate), or small lookup tables (e.g., protocol numbers to names). It is like a global static array inside the kernel.
- **Hash maps** are key-value hash tables where the keys can be structs (e.g., flow identifiers). The insertions and removals are dynamic, but at the cost of slightly higher overhead than arrays. However, they are extremely

flexible and often used for flow tracking (e.g., counting packets per flow), connection state (e.g., TCP connection tracking), or per-client statistics (e.g., bytes sent per IP address). It is like a dynamic hash table inside the kernel.

- **Per-CPU maps** deserve a special mention. **Each CPU gets its own copy of the map**, which eliminates contention between CPUs²⁰ and the needs for locks. They exist because eBPF programs often run on many CPUs simultaneously, and without per-CPU maps, they would have to contend for access to shared data structures, which can lead to performance degradation. However, the **trade-off of this design is that the values must be aggregated**²¹ later (usually in user-space) to get a global view of the data. They are ideal for high-performance counters (e.g., packets processed per CPU), latency measurements (e.g., per-CPU histograms), or any scenario where contention is a concern. It is like having a separate copy of the map for each CPU, which can be accessed without locks, but requires aggregation to get global results.

Helper Functions

Helper Functions are **predefined kernel functions** that **eBPF programs are allowed to call in order to interact with the kernel**. We remarked that eBPF programs cannot call arbitrary kernel functions, so instead of full kernel access, they get a **restricted set of approved helper functions** that provide specific functionalities.

Typical helpers we will use. Some of the most common helper functions we will use in this course include:

- **Map helpers** are the most common. They allow eBPF programs to **lookup**, **update**, and **delete** an element.

`key → helper chosen → value`

These helpers are essential for stateful programs.

- **Read semantics.** Lookup helpers return a pointer to the value associated with the key, or NULL if the key is not found. The eBPF program can read the value through this pointer, but it cannot modify it directly (i.e., it cannot write to the memory location). This design allows multiple eBPF programs to read the same value concurrently without conflicts, while ensuring that updates are controlled and safe.
- **Update semantics.** Update helpers allow the eBPF program to modify the value associated with a key, or insert a new key-value pair if the key does not exist. This operation is controlled and safe, ensuring that concurrent updates do not lead to inconsistencies.

²⁰Multiple CPUs trying to access the same memory location at the same time, which can lead to performance degradation

²¹Combining the values from all CPU copies to get a single result, such as summing counters across CPUs

This distinction between read and update semantics is crucial because lookup gives us a pointer with tracked bounds (i.e., the verifier knows the size of the value and can ensure that the program does not read or write out of bounds), while update allows us to modify the map's contents in a controlled manner.

- **Packet redirection helpers** are used when forwarding packets, for load balancing, or when sending packets to another interface.
- **Time helpers** are used to get kernel time, implement timeouts, or measure interval durations. They are crucial for connection expiration, rate limiting, or latency measurements.
- **Checksum helpers** are used for rewriting packet headers and maintaining protocol correctness.

How to create modular eBPF programs? Tail Calls

eBPF programs have limits: the stack is small (512 bytes), the number of instructions is limited (usually 4096), and the verifier checks for safety, which can make complex programs difficult to write. To overcome these limitations, eBPF provides a mechanism called **tail calls** to split large logic into multiple programs.

Imagine we are implementing a firewall, or a load balancer, or a TCP connection tracker, or a multi-protocol parser. Soon we will hit the instruction limit, or stack limit, or just have a very complex program that is hard to verify. Without tail calls, we would have to cram everything into a single program, which increases risk of verifier rejections and makes the code harder to maintain. **Tail calls allow an eBPF program to transfer execution to another eBPF program without returning.**

Definition 3: Tail Call

A **Tail Call** is a controlled jump from one eBPF program to another, using a special map called a **program array**.

It works like this:

1. Program *A* runs.
2. *A* decides to delegate some work (e.g., because it has reached the instruction limit, or wants to separate logic).
3. *A* performs a tail call to program *B*.
4. Program *B* executes immediately, without returning to *A*.
5. Program *A* never resumes, so it must not rely on any state after the tail call.

This is different from a normal function call, where the caller expects

to return and continue execution. In a tail call, the caller effectively transfers control to the callee and does not expect to return.

Tail calls allow us to create **program chains**, where a sequence of eBPF programs can be executed in order, each handling a specific part of the logic. It's similar to a **pipeline inside the kernel**. For example:

1. A packet arrives.
2. Program *A* parses the Ethernet header and decides to tail call program *B*.
3. Program *B* parses the IP header and decides to tail call program *C*.
4. Program *C* parses the TCP header and makes a decision (e.g., allow, drop, redirect).

This modular approach is powerful because:

- **Instruction limit workaround.** Instead of one huge program, we can have multiple smaller programs that together implement the full logic.
- **Modular architecture.** Each program can focus on a specific task (e.g., parsing a specific protocol), making the code easier to understand and maintain.
- **Dynamic behavior.** We can change the program chain at runtime by updating the program array map, allowing for dynamic updates to the processing logic without needing to reload the entire program.

⚠ **Important constraints.** Tail calls are not free. There are limits:

- **Maximum number of tail calls per packet is usually 32.** This means that a packet can only be passed through a chain of 32 programs before it must be processed or dropped.
- **No return to the caller.** Once a tail call is made, the caller program cannot resume execution. This means that any state or logic in the caller after the tail call will never be executed, so it must be designed accordingly.
- **Stack is not shared between programs.** Each program has its own stack, so data that needs to be shared across programs must be stored in maps or passed through registers (with limitations).
- **Must use a special map type (program array).** Tail calls require a program array map, which is a special type of map that holds references to eBPF programs. The caller program uses this map to specify which program to tail call based on an index.

Despite these constraints, tail calls are a powerful tool for building complex eBPF applications while keeping the code modular and maintainable. However, simple programs that do not require modularity or do not hit instruction limits may not need to use tail calls at all.

In summary, tail calls allow an eBPF program to transfer execution to another eBPF program without returning, enabling modular design and scalable program chaining while preserving verifier constraints.

9.3.5 XDP: eXpress Data Path

XDP (eXpress Data Path) is an eBPF hook²² that allows programs to run at the earliest possible point in the Linux networking receive path, directly in the network driver. This definition contains the most important word: *earliest*.

❷ **Hook point location.** To understand XDP, we must understand where it is located in the Linux networking stack. The receive path of a network packet in Linux can be divided into several stages (simplified for clarity):

1. **NIC (Network Interface Card) receives the packet:** The packet arrives at the network interface card, which is responsible for handling the physical layer of the network communication.
2. **Driver processing:** The NIC's driver processes the packet, which may involve tasks such as DMA (Direct Memory Access) to transfer the packet data into memory.
Here is where XDP operates. It allows eBPF programs to be attached directly to the driver, enabling them to process packets immediately after they are received by the NIC, before any further processing occurs in the kernel.
3. **SKB (Socket Buffer) allocation:** After the driver processes the packet, it typically allocates a socket buffer (SKB) to hold the packet data for further processing in the kernel.
4. **Kernel networking stack:** The packet is then processed by the kernel's networking stack, which includes various layers such as IP, TCP/UDP, and application-level processing.
5. **Socket layer:** If the packet is destined for a local application, it is delivered to the socket layer, where it can be read by user-space applications.
6. **User space applications:** Finally, the packet is delivered to user-space applications that are listening for incoming network traffic.

XDP runs **before** the kernel builds the SKB (`sk_buff`), which makes it very fast (since it avoids the overhead of SKB allocation and processing), very low overhead (since it operates at the driver level), and very powerful (since it can make decisions about the packet before any kernel processing occurs).

❸ **Comparison with TC and Netfilter hooks.** To further understand the significance of XDP, let's compare it with other eBPF hook points in the Linux networking stack, such as TC (Traffic Control) and Netfilter hooks.

- **Netfilter** is a framework in the Linux kernel that provides hooks for packet filtering and manipulation. Netfilter operates at a higher level in the networking stack, **after the SKB has been allocated**. This means that while Netfilter allows for powerful packet processing capabilities, it incurs more overhead compared to XDP, as it requires the kernel to allocate and manage SKBs.

²²A predefined point inside the Linux kernel where an eBPF program can be attached and executed when a specific event occurs.

- **TC (Traffic Control)** is another eBPF hook point that operates at the ingress and egress points of network interfaces. TC allows for more advanced traffic shaping and policing capabilities compared to XDP, but it also operates at a higher level in the networking stack, **after the SKB has been allocated**. This means that while TC provides more flexibility in terms of traffic management, it also incurs more overhead compared to XDP.
- **XDP** operates at the earliest point in the receive path, directly in the driver, **before any SKB allocation**. This allows XDP to process packets with minimal overhead, making it ideal for high-performance use cases such as DDoS mitigation, load balancing, and packet filtering.

Execution Modes

XDP can operate in different execution modes, which determine how the eBPF program is executed and what capabilities it has:

1. **Native Mode (Driver Mode)** is the **preferred execution mode** for XDP:
 - **Runs directly in the NIC driver**, providing the lowest latency and highest performance.
 - Requires support from the NIC driver, which may not be available for all hardware.
 - Offers the most powerful capabilities, as it can access the packet data directly and make decisions before any kernel processing occurs.
 - Uses XDP memory modeling, which allows for efficient packet processing without the overhead of SKB allocation.
2. **Generic Mode** is the **fallback execution mode** for XDP:
 - Works even if the driver does not support native XDP, by running the eBPF program in a more generic context.
 - Implemented **inside the kernel stack** (after SKB allocation), which means it incurs more overhead compared to native mode.
 - It is useful for testing and development purposes, but it is not recommended for production use due to its higher latency and lower performance compared to native mode.
3. **Offloaded Mode** is an **advanced execution mode** for XDP:
 - Program **runs directly on the NIC hardware**.
 - Requires special hardware SmartNIC support, which is not widely available.
 - Highest performance, as it offloads packet processing to the NIC, freeing up CPU resources for other tasks.

This is closer to P4-style hardware programming, where the packet processing logic is implemented directly on the NIC hardware, providing the best performance and lowest latency. However, it requires specialized hardware support and usually is used in high-performance environments such as data centers and cloud providers.

9.3.6 XDP Execution Context

When an XDP program runs, it does **not** receive a packet object as an argument. Instead, it receives a **pointer to a structure** called `xdp_md`, which is the **execution context**. We can think of it as the information the kernel gives our program when a packet arrives, and it contains all the information about the packet and the environment in which the program is running.

A simplified version looks like this:

```

1 struct xdp_md {
2     __u32 data;
3     __u32 data_end;
4     __u32 data_meta;
5     __u32 ingress_ifindex;
6     __u32 rx_queue_index;
7 };

```

This structure contains several fields, but the most important ones for us are:

- `data` is a pointer to the beginning of the packet in memory. It usually points to the Ethernet header of the packet. It is the first byte of the received frame.
- `data_end` is a pointer to the first byte *after* the end of the packet. It is extremely important because if we access memory beyond this pointer, we will cause an out-of-bounds access, which the kernel will detect and reject our program.
- `data_meta` is less commonly used, but it points to a region before `data` that can be used to store metadata about the packet. This is useful for certain advanced use cases, like storing information that can be accessed by other eBPF programs later in the processing pipeline.

? **Why both pointers?** The kernel does not give us a packet length or a safe buffer abstraction. Instead, it gives us raw memory boundaries, and we must always prove to the verifier that we are accessing memory within these boundaries. In practice, we will often write:

```

1 void *data = (void *)(long)ctx->data;
2 void *data_end = (void *)(long)ctx->data_end;
3
4 if (data + sizeof(struct ethhdr) > data_end) {
5     // Not enough data for Ethernet header, drop the packet
6     return XDP_PASS;
7 }

```

This way, we ensure that we only access memory that is within the packet's boundaries, and the verifier will allow our program to run.

? **If `data` and `data_end` are pointers, why are they defined as `__u32`?** This is largely a UAPI/ABI quirk (i.e., the way the kernel's user-space API is defined): in `struct xdp_md`, the fields `data` and `data_end` are declared as `__u32`, so in C they look like integers rather than pointers. In XDP programs we therefore cast them (e.g., `(void *)(long)ctx->data`) to obtain pointers that can be used for packet parsing. Crucially, the eBPF verifier treats these values

as special packet pointers and rewrites/tracks their use at load time, enforcing bounds checks via `data_end`. This design is maintained for compatibility, but programmers should think of `data` and `data_end` as defining the valid packet memory range `[data, data_end]`.

In other words, in the actual kernel, these fields represent memory addresses, but they are exposed as `_u32` for ABI and verifier reasons. The verifier rewrites them into proper pointers during program loading, so what happens is:

1. We cast them to pointers.
2. The verifier checks all pointer arithmetic.
3. The kernel internally treats them as valid packet memory pointers.

The kernel cannot just give us pointers directly (e.g., `struct ethhdr *eth`) because the packet layout and the size are not known at compile time, and the verifier needs to ensure that all memory accesses are safe. By giving us raw pointers and enforcing bounds checks, the kernel allows us to write flexible packet processing code while maintaining safety guarantees.

In summary, if we do something like:

```
1 struct ethhdr *eth = data;
2 eth->h_proto
```

Without checking bounds first, the **verifier will reject our program** because we might be accessing memory beyond `data_end`. We must always check that we have enough data before accessing any fields of the packet, which is a fundamental aspect of writing safe eBPF/XDP programs.

9.3.7 XDP Return Codes

An XDP program must return an integer value that tells the kernel how to handle the packet: “*what should we do with this packet?*”. These return values are predefined constants:

- **XDP_PASS**: let the packet continue up the normal Linux networking stack. So the flow of the packet is not changed, and the **packet is processed as if no XDP program was attached to the interface**. We use this return code when we want to let the packet be processed by the kernel as usual, without any modification.
- **XDP_DROP**: **drop the packet immediately**. The packet will not be processed further, and it will not be forwarded to any other interface. We use this return code when we want to block certain packets, for example, based on their source IP address or other criteria.

This is useful for implementing simple firewall rules or mitigating DDoS attacks. It is important to note that when a packet is dropped, it is **not sent back to the sender**, so the sender will not receive any indication that the packet was dropped. Also, dropping packets can **save CPU resources**, as the kernel does not have to spend time processing the packet further.

However, it is important to use this return code with caution, as dropping packets can lead to unintended consequences, such as disrupting legitimate traffic or causing network instability if used excessively.

- **XDP_TX**: **transmit the packet back out of the same interface it was received on**. This is useful for implementing simple echo servers or for testing purposes. When a packet is transmitted back, it will be sent back to the sender, and the sender will receive a response. This can be used to verify that the XDP program is working correctly and that packets are being processed as expected.

In simple terms, this return code allows us to send the packet back to the sender immediately, bypassing kernel stack and socket layers, which can be useful for certain applications like load balancing or DDoS mitigation.

It is important to note that when a packet is sent back, it must remain valid and properly formatted; otherwise, it may be dropped by the network or cause issues for the sender. If we want to modify the packet before sending it back, we need to update the checksums to ensure the packet remains valid according to the relevant protocols (e.g., Ethernet, IP, TCP/UDP).

- **XDP_REDIRECT**: **redirect the packet to another interface or to a user-space socket**. This is more flexible than XDP_TX because it allows us to send the packet to a different destination, such as another network interface or a user-space application that is listening for packets. This can be useful for implementing load balancing, traffic steering, or for processing packets in user space for more complex logic.

When using XDP_REDIRECT, we need to specify the target interface or socket to which the packet should be redirected. This can be done using helper functions provided by the eBPF API, such as `bpf_redirect()` for

redirecting to another interface or `bpf_redirect_map()` for redirecting to a user-space socket. However, these low-level details are beyond the scope of this introduction, and we will cover them in more depth in later sections when we discuss how to implement XDP programs in practice.

Return Code	What Happens	Typical Use Case
XDP_PASS	Continue normal stack processing	Monitoring
XDP_DROP	Drop packet immediately	Filtering
XDP_TX	Send packet back on same interface	Simple reply
XDP_REDIRECT	Send packet to another interface/destination	Forwarding

Table 23: Summary of XDP return codes and their effects on packet processing.

In summary, the return code is the **only way for an XDP program to communicate with the kernel about how to handle the packet**. By choosing the appropriate return code, we can control the flow of packets through the network stack and implement various functionalities such as filtering, load balancing, or traffic redirection.

9.3.8 Tooling: libbpf & bpftool

So far, we have learned how XDP works and what maps, helpers, and return codes are. We also know the constraints imposed by the verifier. But how do we actually load and manage eBPF programs from user space? This is where `libbpf` and `bpftool` come into play.

■ Role of libbpf

`libbpf` is the **official user-space library** (i.e., a C library) used to **load**, **verify**, **attach**, and **manage eBPF programs**. It acts as a **bridge** between our user-space application and the kernel's eBPF subsystem. Without `libbpf`, we would have to interact with the kernel's eBPF API directly, which is complex and error-prone. `libbpf` abstracts away these complexities and provides a convenient interface for developers.

When we load an eBPF program using `libbpf`, it performs several tasks:

1. **Parses the compiled ELF file** (i.e., the output of our eBPF C code compilation).
2. **Creates maps in the kernel** as defined in our eBPF program.
3. **Loads programs via the bpf syscall**, which involves passing the program's bytecode and metadata to the kernel.
4. **Invokes the kernel's verifier** to ensure the program is safe to run.
5. **Attaches programs to the specified hooks**. In our case, it attaches to the XDP hook, but it can also attach to other hooks, such as `kprobes` and `uprobes`.
6. **Provides file descriptors for maps and programs**, allowing us to interact with them from user space (e.g., reading map values).

So conceptually:

User-space app → `libbpf` → `bpf()` syscall → Kernel verifier and loader

■ Do not use libbpf directly! Use bpftool instead

When writing an XDP program, we actually write **two separate programs**:

- A **kernel program**, which is restricted C code compiled into eBPF bytecode and executed inside the kernel (e.g., attached to the XDP hook). It implements the fast-path data-plane logic under strict safety and resource constraints enforced by the verifier.

For example, we write the kernel program in a file named `xdp_prog.bpf.c`, which contains the eBPF code that will be loaded into the kernel. Then, we compile it using `clang` with specific flags to generate the eBPF bytecode:

`clang → xdp_prog.bpf.o`

- A **user-space program**, which loads and manages the eBPF program using `libbpf`. It interacts with the kernel program indirectly through shared eBPF maps, handling configuration, statistics collection, and control-plane logic that cannot be efficiently or safely implemented inside the kernel.

For example, we write the user-space program in a file named `xdp_loader.c`, which uses `libbpf` to load the compiled eBPF program and manage its lifecycle.

When we got the bytecode from the kernel program (`xdp_prog.bpf.o`), we cannot just execute it directly. We need to load it into the kernel, create maps, attach program to the interface (i.e., the XDP hook), and get file descriptors to interact with maps. That is complicated, so **Linux provides a tool called `bpftool` that uses `libbpf` under the hood to perform all these tasks for us.**

`bpftool` is a **command-line utility** that allows us to manage eBPF programs and maps without writing any C code:

```
1 bpftool gen skeleton xdp_prog.bpf.o > xdp_prog.skel.h
```

This command generates a C header file (`xdp_prog.skel.h`) that contains the necessary code to load and manage the eBPF program defined in `xdp_prog.bpf.o`. The generated **skeleton** is automatically generated C code that wraps our eBPF program using `libbpf` (i.e., it contains the code to load the program, create maps, and attach it to the XDP hook). It provides a struct like this:

```
1 struct xdp_prog_bpf {
2     struct bpf_object *obj;
3     struct bpf_program *prog;
4     struct bpf_map *my_map;
5 };
```

And some helper functions:

```
1 xdp_prog_bpf__open();
2 xdp_prog_bpf__load();
3 xdp_prog_bpf__attach();
```

So instead of manually writing 200 lines of boilerplate code to load and manage the eBPF program, we can just include the generated skeleton header and call these helper functions to do everything for us:

```
1 #include "xdp_prog.skel.h"
2
3 // ...
4
5 skel = xdp_prog_bpf__open();
6 xdp_prog_bpf__load(skel);
7 xdp_prog_bpf__attach(skel);
```

In other words, it is an auto-generated `libbpf` wrapper for our specific eBPF program.

With `bpftool`, our workflow becomes much simpler:

1. Write kernel program in `xdp_prog.bpf.c`.
2. Compile it to eBPF bytecode using `clang`.

3. Generate the skeleton header using `bpftool`.
4. Write a small user-space program that includes the skeleton header and calls the helper functions to load and attach the eBPF program.

Instead of manually parsing ELF (i.e., the output of `clang`), calling the `bpf` syscall, handling map file descriptors, and attaching the program, we can just use the generated skeleton to do all of that for us. In summary, **the skeleton reduces complexity and boilerplate code drastically, allowing us to focus on the logic of our eBPF program rather than the intricacies of loading and managing it.**

Kernel Program vs User-space Loader

This is architectural, not tooling.

- **Kernel Program = Data Plane.** The C file `xdp_prog.bpf.c` runs in the kernel, is attached to the XDP hook, works on every packet, and is subject to the verifier's constraints. Its purpose is to process packets quickly with minimal state handling. It should be small, deterministic, and efficient.
- **User-space Loader = Control Plane.** The C file `xdp_loader.c` runs in user space with full C features *without* verifier constraints. It is responsible for loading the kernel program, attaching it to the XDP hook, read/update maps, print statistics, handle timeouts, and parse command-line arguments. It can be as complex as needed since it does not run in the kernel and is not subject to the same constraints.

In summary, the **kernel program quickly** counts packets per flow, checks the threshold, and drops or passes them. In contrast, the **user-space** sets the threshold value in the map, reads the statistics, and prints the heavy hitters. It is **smart**.

Some useful bpftool commands

`bpftool` is not only useful for generating skeletons but also for inspecting and managing eBPF programs and maps at runtime. Here are some common commands:

- **Inspecting eBPF Objects.** Once an eBPF program is loaded into the kernel, it becomes a kernel object. We can use `bpftool` to inspect these objects:

```
1 bpftool prog list
```

This command shows all loaded eBPF programs, their IDs, types (e.g., XDP), attached interfaces, and other metadata. We can also **inspect maps** with:

```
1 bpftool map list
```

This shows all eBPF maps, their types (e.g., hash, array), sizes, and other details. This is useful for debugging and monitoring our eBPF programs. Finally, we can even **dump map contents** with:

```
1 bpftool map dump id <map_id>
```

This command allows us to see the key-value pairs stored in a specific map, which is invaluable for understanding the state of our eBPF program and verifying that it is working as expected.

- **Generating Skeletons.** As mentioned earlier, we can generate skeleton headers for our eBPF programs with:

```
1 bpftool gen skeleton xdp_prog.bpf.o > xdp_prog.skel.h
```

This command :

- Parses the compiled eBPF bytecode from `xdp_prog.bpf.o`.
- Detect the maps and programs defined in the bytecode.
- Generates a C header file (`xdp_prog.skel.h`) that contains the necessary code to load, manage, and interact with the eBPF program using `libbpf`.

- **Debugging Maps and Programs.** If our eBPF program is not working as expected, we can use `bpftool` to debug it. For example, if we want to **check if our XDP program is attached to the correct interface**, we can run:

```
1 bpftool net
```

This command shows which interfaces have XDP programs attached and their IDs.

If we want to **show program details**, we can run:

```
1 bpftool prog dump xlated id <id>
```

This command shows the eBPF bytecode of the program with the specified ID, which can help us understand how the kernel is executing our program and identify any issues.

9.3.9 Exercise 1: The first eBPF program

A minimal XDP program is just a function that:

- Receives `struct xdp_md *ctx` as an argument, which is the execution context of the XDP program (see page 278).
- Optionally inspects packet bytes via `ctx->data` and `ctx->data_end` pointers, which point to the start and end of the packet data, respectively.
- **Returns an XDP return code** (see page 280), which is an integer that indicates how the kernel should process the packet after the XDP program finishes executing.

At the beginning of the laboratory, we don't even need parsing the packet bytes, just returning a constant already proves the whole pipeline works.

Our kernel program `hello_world.bpf.c` is compiled by clang into an ELF object file (commonly named `hello_world.bpf.o`) that contains: eBPF bytecode, map definitions (if any) and program section metadata. We don't run this file, we **load** it into the kernel, which verifies the eBPF bytecode and, if it is valid, creates a new eBPF program in the kernel with the same logic as our C code.

Finally, the program is attached to an interface's **XDP hook**. Once attached, it runs on every packet received by that interface, and our return code decides the fate of the packet.

In this first exercise, we will write the classic “*Hello, World!*” program of the eBPF world: an XDP program that prints “*Hello World from BPF!*” for every packet received by the interface, then returns `XDP_PASS` to let the packet continue its normal processing in the kernel. Let's start:

1. **Topology** (*what is happening in the network?*). The exercise gives us a simple bash script:

```

1 #!/bin/bash
2
3 # include helper.bash file: used to provide some common
4 #                         function across testing scripts
4 source "${BASH_SOURCE%/*}/../../libs/helpers.bash"
5
6 # function cleanup: is invoked each time script exit (with or
7 #                   without errors)
7 function cleanup {
8     set +e
9     delete_veth 1
10 }
11 trap cleanup ERR
12
13 # Enable verbose output
14 set -x
15
16 cleanup
17 # Makes the script exit, at first error
18 # Errors are thrown by commands returning not 0 value
19 set -e
20

```

```
21 # Create a network namespace and a veth pair
22 create_veth 1
23 sudo ifconfig veth1 10.0.0.254/24 up
```

This script creates a new network namespace and a `veth` pair, then assigns an IP address to one of the `veth` interfaces. The other interface is left without an IP address and is used to load our XDP program. To execute the script, we can run:

```
1 chmod +x create-topo.sh
2 ./create-topo.sh
```

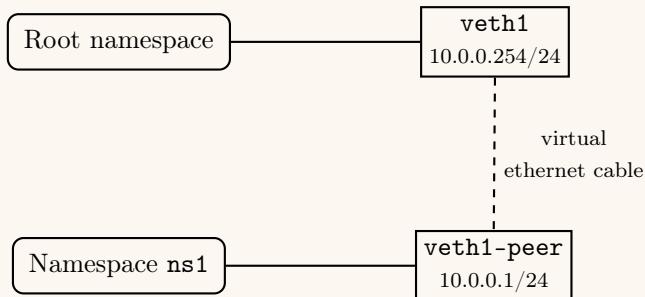
Deepening: What is a veth?

`veth` (virtual Ethernet) is a *pair* of virtual network interfaces that behave like a cable. When a packet is sent to one end of the pair, it immediately appears on the other end. It's like a patch cable but entirely inside the Linux kernel, no physical NIC is involved.

❸ **Why does Linux have veth pairs?** Because Linux supports **network namespaces**. A namespace is basically a completely isolated network stack. Each namespace has its own interfaces, routing tables, firewall rules, ARP tables, etc. But namespaces cannot communicate unless we connect them. And how do we connect two namespaces? With a `veth` pair!

❹ **What does the script above do?** The line `create_veth 1` creates a new namespace (let's call it `ns1`) and a `veth` pair (`veth1` and `veth1-peer`). It puts `veth1` in the default namespace and `veth1-peer` in `ns1`. Then, it assigns the IP address `10.0.0.254/24` to `veth1`. This means that `veth1` can be used to send packets to `ns1` and vice versa, but only if we use the correct IP address and routing rules. In our case, we will use `veth1` to load our XDP program and test it by sending packets from `ns1`.

So the topology becomes:



And we can do:

```
1 ip netns exec ns1 ping 10.0.0.254
```

To send packets from `ns1` to `veth1`, which will be processed by our XDP program, since it is attached to `veth1`.

❓ Why do we need this topology? We want:

- A packet to be **generated somewhere**;
- To travel through a **real interface** (not loopback) so that it can be processed by our XDP program;

Without touching our real network interfaces (like `eth0`), which could cause problems if we mess up with the XDP program. So we need a packet sender, a real interface, a packet receiver and all isolated from our real network. That's exactly what the `veth` pair and the network namespace give us.

So a `veth` pair simulates a **cable between two machines**. We can think of two physical machines connected by a cable:



If Host 1 sends a packet, it physically travels through the cable and arrives at Host 2. The `veth` pair simulates this behavior, but instead of a physical cable, it's a virtual one inside the kernel. So when Host 1 sends a packet, it goes through the virtual cable and arrives at Host 2, just like in the physical case.

Finally, XDP runs on packet **receives**, inside the driver. It does **NOT** run when a packet is sent. So we need a packet to be sent from one namespace (`ns1`), arrive at an interface (`veth1`), trigger the receive path, and execute our XDP program. That's why we need this topology.

2. **eBPF program** (*what does the program do?*). We will write a simple XDP program in C that prints a message for every packet received.

```

1 #include <stdio.h>
2 #include <linux/bpf.h>
3 #include <bpf/bpf_helpers.h>
4
5 SEC("xdp")
6 int xdp_prog_simple(struct xdp_md *ctx) {
7     //TODO: Implement the BPF program
8     bpf_printk("Hello World from BPF!");
9     return XDP_PASS;
10 }
11
12 char LICENSE[] SEC("license") = "Dual BSD/GPL";
  
```

Listing 2: `hello_world.bpf.c`

The `SEC("xdp")` macro tells the compiler that this function must be attached to the XDP hook. Without this, the program won't be loaded as XDP.

The `bpf_printk` is a helper function that allows us to print debug messages from the kernel. The message will be visible in the kernel logs, which we can read with `dmesg` or:

```
1 sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Finally, we return `XDP_PASS` to let the packet continue its normal processing in the kernel. If we returned `XDP_DROP`, the packet would be dropped immediately and never reach the network stack.

3. **Compile the Program** (*is it valid?*). We will compile our C code into an eBPF object file using clang. The command is:

```
1 make
```

Clang will compile our C code into eBPF bytecode and create an ELF object file named `hello_world.bpf.o` inside the `.output` directory. Then, `bpftrace` automatically will generate a skeleton file named `hello_world.skel.h` that contains the necessary code to load and attach our XDP program from user space.

4. **Loading the Program into the Kernel**. Inside `hello_world.c`, we have:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/resource.h>
4 #include <bpf/bpf.h>
5 #include <bpf/btf.h>
6 #include <bpf/libbpf.h>
7 #include <fcntl.h>
8 #include <assert.h>
9 #include <linux/if_link.h>
10
11 #include <argparse.h>
12 #include <net/if.h>
13
14 #ifndef __USE_POSIX
15 #define __USE_POSIX
16 #endif
17 #include <signal.h>
18
19 #include "log.h"
20
21 // Include skeleton file
22 #include "hello_world.skel.h"
23
24 // ...
25
26 int main(int argc, const char **argv) {
27     // ...
28     /* Open BPF application */
29     skel = hello_world_bpf__open();
30     if (!skel) {
31         log_fatal("Error while opening BPF skeleton");
```

```

32         exit(1);
33     }
34
35     /* Set program type to XDP */
36     bpf_program__set_type(skel->progs.xdp_prog_simple,
37                           BPF_PROG_TYPE_XDP);
38
39     /* Load and verify BPF programs */
40     if (hello_world_bpf__load(skel)) {
41         log_fatal("Error while loading BPF skeleton");
42         exit(1);
43     }
44
45     struct sigaction action;
46     memset(&action, 0, sizeof(action));
47     action.sa_handler = &sigint_handler;
48
49     if (sigaction(SIGINT, &action, NULL) == -1) {
50         log_error("sigaction failed");
51         goto cleanup;
52     }
53
54     if (sigaction(SIGTERM, &action, NULL) == -1) {
55         log_error("sigaction failed");
56         goto cleanup;
57     }
58
59     xdp_flags = 0;
60     xdp_flags |= XDP_FLAGS_DRV_MODE;
61
62     /* Attach the XDP program to the interface */
63     int err = bpf_xdp_attach(
64         ifindex_iface,
65         bpf_program__fd(skel->progs.xdp_prog_simple),
66         xdp_flags,
67         NULL
68     );
69
70     if (err) {
71         log_fatal(
72             "Error while attaching the XDP program to the
73             interface"
74         );
75         goto cleanup;
76     }
77
78     // Sleep forever
79     while (1) {
80         sleep(1);
81     }
82 }
```

Listing 3: hello_world.c

This could look intimidating at first, but it's just a lot of boilerplate code to load and attach our XDP program. The important part is:

- **Open skeleton:** `hello_world_bpf__open()` reads the ELF object file generated by clang and prepares the eBPF program for loading.

- **Load program (verifier runs here):** `hello_world_bpf__load()` loads the eBPF bytecode into the kernel. During this step, the kernel's eBPF verifier checks the program for safety and correctness. If the program is invalid, it will be rejected and an error will be returned.

- **Attach to interface:**

```

1 int err = bpf_xdp_attach(
2     ifindex_iface,
3     bpf_program__fd(skel->progs.xdp_prog_simple),
4     xdp_flags,
5     NULL
6 );

```

This function attaches our XDP program to the specified network interface. The first argument is the interface index (we can get it with `if_nametoindex("veth1")`), the second is the file descriptor of our XDP program, the third is a set of flags (we use `XDP_FLAGS_DRV_MODE` to specify that we want to use driver mode), and the last one is for error handling (we pass `NULL` for now).

5. **Testing.** We run the topology, if not already running, and then execute our user space program:

```

1 # Run the topology script if not already running
2 chmod +x create-topo.sh
3 ./create-topo.sh
4 # Run the user space program to load and attach the XDP
      program
5 sudo ./hello_world -i veth1

1 $ sudo ./hello_world -i veth1
2 18:28:29 INFO hello_world.c:72: XDP program will be attached
      to veth1 interface
3 18:28:29 INFO hello_world.c:78: Got ifindex for iface: veth1,
      which is 3
4 18:28:29 INFO hello_world.c:126: Successfully attached!

```

Now we watch the kernel logs:

```

1 sudo su
2 cat /sys/kernel/debug/tracing/trace_pipe

```

And in another terminal, we send packets from ns1 to veth1:

```

1 $ ping 10.0.0.1 -c 1
2 PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
3 64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=1.18 ms
4
5 --- 10.0.0.1 ping statistics ---
6 1 packets transmitted, 1 received, 0% packet loss, time 0ms
7 rtt min/avg/max/mdev = 1.175/1.175/1.175/0.000 ms

```

In the kernel logs, we should see:

```

1 ping-9618      [001] ..s2.  7268.496773: bpf_trace_printk:
      Hello World from BPF!

```

```
2 ping -9618      [001] ..s2.  7268.496996: bpf_trace_printk:  
    Hello World from BPF!  
3 ksoftirqd/1-24 [001] ..s1.  7273.572710: bpf_trace_printk:  
    Hello World from BPF!
```

This means that our XDP program is correctly attached to `veth1` and is executing for every packet received by that interface, printing our message in the kernel logs.

The full code of the exercise is available here:



[Full code on GitHub](#)

9.3.10 Exercise 2: Counting with BPF Maps

In this exercise, we want to:

- Count **number of packets**.
- Count **number of bytes**.
- Store both as 64-bit counters.
- Save them in a **BPF map**.
- Read them from user-space with `bpftool`.
- Print them continuously.

This is our first **kernel \leftrightarrow user-space** communication exercise, and it will be the basis for all the next ones.

Unlike the previous exercise, now state survives across packets and function calls, and user-space can read it. This is the main purpose of BPF maps: they are a key-value store that can be accessed both from kernel and user-space, and they are used to store state across function calls and packets.

- **Define the structure.** We need to define a structure that will hold our counters. We can define it as follows:

```

1 struct datarec {
2     __u64 rx_packets;
3     __u64 rx_bytes;
4 };

```

We use `__u64` to ensure that our counters are 64-bit, which is important to avoid overflow.

- **Define the map.** We need to define a BPF map that will hold our counters. We can define it as follows:

```

1 struct {
2     __uint(type, BPF_MAP_TYPE_ARRAY);
3     __type(key, int);
4     __type(value, struct datarec);
5     __uint(max_entries, 1);
6 } xdp_stats_map SEC(".maps");

```

We use an array map with a single entry (key 0) to store our counters. The value is of type `struct datarec`, which we defined earlier. We set the map type to `BPF_MAP_TYPE_ARRAY` and the maximum number of entries to 1, since we only need one entry to store our counters.

- **Lookup the Map.** We need to lookup the map to get the current counters, update them, and then save them back to the map. We can do this as follows:

```

1 rec = bpf_map_lookup_elem(&xdp_stats_map, &key);
2 if (!rec) {
3     return XDP_ABORTED;
4 }

```

We use the `bpf_map_lookup_elem` helper function to lookup the map and get a pointer to our counters. If the lookup fails, we return `XDP_ABORTED` to indicate an error.

- **Accessing Packet Length.** We can access the packet length using the `data_end` and `data` pointers. The packet length can be calculated as follows:

```
1 void *data_end = (void *)(long)ctx->data_end;
2 void *data = (void *)(long)ctx->data;
```

The packet length is then `data_end - data` and we can save it in our counters:

```
1 __u64 bytes = data_end - data;
```

- **Update the Counters Atomically.** We need to update the counters atomically to avoid race conditions. We can use the `_sync_fetch_and_add` function to atomically update our counters as follows:

```
1 __sync_fetch_and_add(&rec->rx_packets, 1);
2 __sync_fetch_and_add(&rec->rx_bytes, bytes);
```

This will increment the packet counter by 1 and the byte counter by the length of the packet atomically.

Now, we can compile the XDP program using the same command as before.

On the user-space side, we can use `bpftool` to read the counters from the map:

- **Redefine the Structure.** We need to redefine the structure in user-space to match the one we defined in kernel-space. We can do this as follows:

```
1 struct datarec {
2     __u64 rx_packets;
3     __u64 rx_bytes;
4 };
```

- **Read the Map.** We can use the following command to read the map:

```
1 void poll_stats(struct counting_with_maps_bpf *skel) {
2     /* TODO 1: get the map file descriptor for the skeleton */
3     int map_fd = 0;
4
5     map_fd = bpf_map__fd(skel->maps.xdp_stats_map);
6     if (map_fd < 0) {
7         log_fatal(
8             "Error while retrieving the map file descriptor"
9         );
10        exit(1);
11    }
12
13    while(true) {
14        /* TODO 2: define the value type (struct datarec) */
15        struct datarec value;
16        int key = 0;
17        int err = 0;
```

```

19         /* TODO 4: get the value of the map for the key 0 */
20         err = bpf_map_lookup_elem(map_fd, &key, &value);
21         if (err != 0) {
22             log_fatal(
23                 "Error while retrieving the value from map"
24             );
25             exit(1);
26         }
27
28         if (value.rx_packets == 0 && value.rx_bytes == 0) {
29             continue;
30         }
31
32         /* TODO 5: print the number of packets received */
33         log_info(
34             "Number of packets received: %llu",
35             value.rx_packets
36         );
37         /* TODO 6: print the number of bytes received */
38         log_info(
39             "Number of bytes received: %llu",
40             value.rx_bytes
41         );
42         sleep(1);
43     }
44 }
```

1. Get the map file descriptor for the skeleton using the `bpf_map__fd` function.
2. Define the value type (`struct datarec`) to hold the counters.
3. Define the key (0) to access the first entry of the map.
4. Get the value of the map for the key 0 using the `bpf_map_lookup_elem` function.
5. Print the number of packets received using the `rx_packets` field of the value.
6. Print the number of bytes received using the `rx_bytes` field of the value.

Now, we can compile the user-space program and run it. We should see the number of packets and bytes received printed continuously:

```

1 $ sudo ./counting_with_maps -i veth1
2 19:25:10 INFO  counting_with_maps.c:113: XDP program will be
   attached to veth1 interface
3 19:25:10 INFO  counting_with_maps.c:119: Got ifindex for iface:
   veth1, which is 9
4 19:25:10 INFO  counting_with_maps.c:167: Successfully attached!
5 19:25:11 INFO  counting_with_maps.c:87: Number of packets received:
   1
6 19:25:11 INFO  counting_with_maps.c:89: Number of bytes received:
   70
```

All the packets received on the `veth1` interface will be counted and their length will be summed in the byte counter. We can generate traffic on the `veth1` interface using tools like `ping` to see the counters update in real-time.

The full code of the exercise is available here:



[Full code on GitHub](#)

9.3.11 Exercise 3: Packet Parsing

In this exercise, we will write an XDP program that parses the incoming packets. The pipeline of the program will be as follows:

1. Get `data` and `data_end` pointers from the context.
2. Parse the Ethernet header and check if the packet is an IPv4 packet. If not, return `XDP_PASS`.
3. Parse the IPv4 header and find the payload offset (variable header length).
4. If protocol is ICMP, parse the ICMP header and read `sequence number`.
5. If the sequence number is even, return `XDP_DROP`, otherwise update map counters and return `XDP_PASS`.

Let's start the implementation:

- **Define the map to store the counters**

```

1 struct datarec {
2     __u64 rx_packets;
3     __u64 rx_bytes;
4 };
5
6 struct {
7     __uint(type, BPF_MAP_TYPE_ARRAY);
8     __type(key, int);
9     __type(value, struct datarec);
10    __uint(max_entries, 1024);
11 } xdp_stats_map SEC(".maps");

```

We define a BPF map of type `BPF_MAP_TYPE_ARRAY` to store the counters for received packets and bytes. The key is an integer, and the value is a structure containing two 64-bit unsigned integers: `rx_packets` and `rx_bytes`. The map can hold up to 1024 entries.

- **Fix the bounds checking bug in `parse_ethhdr()`**

```

1 static __always_inline int parse_ethhdr(
2     void *data,
3     void *data_end,
4     __u16 *nh_off,
5     struct ethhdr **ethhdr
6 ) {
7     struct ethhdr *eth = (struct ethhdr *)data;
8     int hdr_size = sizeof(*eth);
9
10    /* Byte-count bounds check;
11     * check if current pointer + size of header
12     * is after data_end.
13     */
14    /* TODO 1: Fix bound checking errors */
15    // was: if (data + 1 > data_end)
16    if ((void *)eth + hdr_size > data_end)
17        return -1;
18
19    *nh_off += hdr_size;
20    *ethhdr = eth;

```

```

21     return eth->h_proto; /* network-byte-order */
22 }

```

The function `parse_ethhdr()` is used to parse the Ethernet header. It takes the data and `data_end` pointers, a pointer to the offset of the next header, and a pointer to store the parsed Ethernet header. The function checks if the current pointer plus the size of the Ethernet header is greater than `data_end`, which would indicate that we are trying to access memory beyond the packet data. If this check fails, it returns -1. Otherwise, it updates the offset and stores the parsed Ethernet header, returning the protocol field in network byte order.

- **Implement the function `parse_iphdr()`**

```

1 static __always_inline int parse_iphdr(
2     void *data,
3     void *data_end,
4     __u16 *nh_off,
5     struct iphdr **iphdr
6 ) {
7     struct iphdr *ip = data + *nh_off;
8     int hdr_size;
9
10    if ((void *)ip + sizeof(*ip) > data_end)
11        return -1;
12
13    hdr_size = ip->ihl * 4;
14
15    /* Sanity check packet field is valid */
16    if(hdr_size < sizeof(*ip))
17        return -1;
18
19    /* Variable-length IPv4 header,
20       need to use byte-based arithmetic */
21    if ((void *)ip + hdr_size > data_end)
22        return -1;
23
24    // It can also be written as:
25    // if (data + *nh_off + hdr_size > data_end)
26    //     return -1;
27
28    *nh_off += hdr_size;
29    *iphdr = ip;
30
31    return ip->protocol;
32 }

```

The function `parse_iphdr()` is used to parse the IPv4 header. It takes the data and `data_end` pointers, a pointer to the offset of the next header, and a pointer to store the parsed IPv4 header. The function first checks if the current pointer plus the size of the IPv4 header is greater than `data_end`. Then it calculates the actual header size using the IHL (Internet Header Length) field, which is in 32-bit words, so we multiply it by 4 to get the size in bytes. It also checks if the calculated header size is valid (at least the size of the standard IPv4 header). Finally, it checks if the current pointer plus the calculated header size is greater than `data_end`. If all checks

pass, it updates the offset and stores the parsed IPv4 header, returning the protocol field.

- **Implement the function `parse_icmphdr()`**

```

1 static __always_inline int parse_icmphdr(
2     void *data,
3     void *data_end,
4     __u16 *nh_off,
5     struct icmphdr **icmphdr
6 ) {
7     struct icmphdr *icmp = data + *nh_off;
8     int hdr_size = sizeof(*icmp);
9
10    if ((void *)icmp + hdr_size > data_end)
11        return -1;
12
13    *nh_off += hdr_size;
14    *icmphdr = icmp;
15
16    return icmp->type;
17 }
```

The function `parse_icmphdr()` is used to parse the ICMP header. It takes the data and `data_end` pointers, a pointer to the offset of the next header, and a pointer to store the parsed ICMP header. The function checks if the current pointer plus the size of the ICMP header is greater than `data_end`. If this check fails, it returns -1. Otherwise, it updates the offset and stores the parsed ICMP header, returning the type field of the ICMP header.

- **Implement the main XDP program**

```

1 SEC("xdp")
2 int xdp_packet_parsing(struct xdp_md *ctx) {
3     void *data_end = (void *) (long) ctx->data_end;
4     void *data = (void *) (long) ctx->data;
5
6     __u16 nf_off = 0;
7     struct ethhdr *eth;
8     int eth_type;
9     struct datarec *rec;
10    int key = 0;
11
12    bpf_printk("Packet received");
13
14    eth_type = parse_ethhdr(data, data_end, &nf_off, &eth);
15
16    if (eth_type != bpf_ntohs(ETH_P_IP))
17        goto pass;
18
19    bpf_printk("Packet is IPv4");
20
21    // Handle IPv4 and parse ICMP
22    int ip_type;
23    struct iphdr *iphdr;
24    ip_type = parse_iphdr(data, data_end, &nf_off, &iphdr);
25
26    if (ip_type != IPPROTO_ICMP)
27        goto pass;
28
```

```

29     bpf_printk("Packet is ICMP");
30
31     int icmp_type;
32     struct icmphdr *icmphdr;
33
34     icmp_type = parse_icmphdr(
35         data,
36         data_end,
37         &nf_off,
38         &icmphdr
39     );
40
41     bpf_printk("Packet is ICMP type: %d", icmp_type);
42     if (icmp_type != ICMP_ECHO)
43         goto out;
44
45     // Now let's check the sequence number
46     __u16 seq = bpf_ntohs(icmphdr->un.echo.sequence);
47
48     bpf_printk(
49         "Packet is ICMP ECHO with sequence number: %d",
50         seq
51     );
52
53     // Check if sequence number is even
54     if (seq % 2 == 0) {
55         bpf_printk(
56             "Dropping packet with even sequence number: %d",
57             seq
58         );
59         return XDP_DROP;
60     }
61
62 out:
63     bpf_printk("Packet passed");
64     rec = bpf_map_lookup_elem(&xdp_stats_map, &key);
65     if (!rec) {
66         return XDP_ABORTED;
67     }
68
69     __u64 bytes = data_end - data;
70     __sync_fetch_and_add(&rec->rx_packets, 1);
71     __sync_fetch_and_add(&rec->rx_bytes, bytes);
72
73 pass:
74     return XDP_PASS;
75 }
```

The main XDP program is defined in the function `xdp_packet_parsing`. It starts by getting the `data` and `data_end` pointers from the context. It then initializes the offset for parsing and defines pointers for the Ethernet header and the data record.

The program first calls `parse_ethhdr()` to parse the Ethernet header. If the packet is not an IPv4 packet, it jumps to the `pass` label to allow the packet to pass through.

If the packet is IPv4, it calls `parse_iphdr()` to parse the IPv4 header. If the protocol is not ICMP, it again jumps to the `pass` label.

If the packet is ICMP, it calls `parse_icmphdr()` to parse the ICMP header. If the ICMP type is not ECHO, it jumps to the `out` label.

Finally, it checks if the sequence number of the ICMP ECHO request is even. If it is even, it drops the packet. Otherwise, it updates the counters in the map and allows the packet to pass through.

About the user-space part:

- **Redefine the datarec structure**

```
1 struct datarec {
2     __u64 rx_packets;
3     __u64 rx_bytes;
4 };
```

The user-space program also needs to define the same `datarec` structure to read the counters from the BPF map.

- **Define the function to print the stats** (same as the previous exercise)

```
1 void poll_stats(struct packet_parsing_bpf *skel) {
2     /* TODO 1: get the map file descriptor for the skeleton */
3     int map_fd = 0;
4
5     map_fd = bpf_map__fd(skel->maps.xdp_stats_map);
6     if (map_fd < 0) {
7         log_fatal(
8             "Error while retrieving the map file descriptor"
9         );
10        exit(1);
11    }
12
13    while(true) {
14        /* TODO 2: define the value type (struct datarec) */
15        struct datarec value;
16        int key = 0;
17        int err = 0;
18
19        /* TODO 4: get the value of the map for the key 0 */
20        err = bpf_map_lookup_elem(map_fd, &key, &value);
21        if (err != 0) {
22            log_fatal(
23                "Error while retrieving the value from map"
24            );
25            exit(1);
26        }
27
28        if (value.rx_packets == 0 && value.rx_bytes == 0) {
29            continue;
30        }
31
32        /* TODO 5: print the number of packets received */
33        log_info(
34            "Number of packets received: %llu",
35            value.rx_packets
36        );
37        /* TODO 6: print the number of bytes received */
38        log_info(
39            "Number of bytes received: %llu",
40            value.rx_bytes
41        );
42        sleep(1);
43    }
44 }
```

The full code of the exercise is available here:



[Full code on GitHub](#)

9.3.12 Exercise 4: Packet Rewriting

In this exercise, we will implement a simple XDP program that rewrites the destination port number of TCP and UDP packets to be one less than its original value. For example, if a packet is destined to port 80, we will rewrite it to be destined to port 79. This is a simple example of how XDP can be used to modify packets in-flight, without the need for a full TCP/IP stack.

- Define map and structure to hold packet and byte counters

```

1 struct datarec {
2     __u64 rx_packets;
3     __u64 rx_bytes;
4 };
5
6 struct {
7     __uint(type, BPF_MAP_TYPE_ARRAY);
8     __type(key, int);
9     __type(value, struct datarec);
10    __uint(max_entries, 1024);
11 } xdp_stats_map SEC(".maps");

```

- Implement `parse_ethhdr` function to parse Ethernet header and `parse_iphdr` function to parse IP header (same as in Exercise 3)

```

1 static __always_inline int parse_ethhdr(
2     void *data,
3     void *data_end,
4     __u16 *nh_off,
5     struct ethhdr **ethhdr
6 ) {
7     struct ethhdr *eth = (struct ethhdr *)data;
8     int hdr_size = sizeof(*eth);
9
10    /* Byte-count bounds check; check if current pointer +
11       size of header
12       * is after data_end.
13       */
14    if ((void *)eth + hdr_size > data_end)
15        return -1;
16
17    *nh_off += hdr_size;
18    *ethhdr = eth;
19
20    return eth->h_proto; /* network-byte-order */
21 }
22
23 static __always_inline int parse_iphdr(
24     void *data,
25     void *data_end,
26     __u16 *nh_off,
27     struct iphdr **iphdr
28 ) {
29     struct iphdr *ip = data + *nh_off;
30     int hdr_size;
31
32     if ((void *)ip + sizeof(*ip) > data_end)
33         return -1;
34
35     hdr_size = ip->ihl * 4;
36

```

```

36  /* Sanity check packet field is valid */
37  if(hdr_size < sizeof(*ip))
38      return -1;
39
40  /* Variable-length IPv4 header, need to use byte-based
   arithmetic */
41  if ((void *)ip + hdr_size > data_end)
42      return -1;
43
44  // It can also be written as:
45  // if (data + *nh_off + hdr_size > data_end)
46  //     return -1;
47
48  *nh_off += hdr_size;
49  *iphdr = ip;
50
51  return ip->protocol;
52 }
```

- Implement `parse_udphdr` function to parse UDP header and `parse_tcphdr` function to parse TCP header

```

1 static __always_inline int parse_udphdr(
2     void *data,
3     void *data_end,
4     __u16 *nh_off,
5     struct udphdr **udphdr
6 ) {
7     struct udphdr *udp = data + *nh_off;
8     int hdr_size = sizeof(*udp);
9
10    if ((void *)udp + hdr_size > data_end)
11        return -1;
12
13    *nh_off += hdr_size;
14    *udphdr = udp;
15
16    int len = bpf_ntohs(udp->len) - sizeof(struct udphdr);
17    if (len < 0)
18        return -1;
19
20    return len;
21 }
22
23 static __always_inline int parse_tcphdr(
24     void *data,
25     void *data_end,
26     __u16 *nh_off,
27     struct tcphdr **tcphdr
28 ) {
29     struct tcphdr *tcp = data + *nh_off;
30     int hdr_size = sizeof(*tcp);
31     int len;
32
33     if ((void *)tcp + hdr_size > data_end)
34         return -1;
35
36     len = tcp->doff * 4;
37     if (len < hdr_size)
38         return -1;
39
40     /* Variable-length TCP header,
```

```

41      need to use byte-based arithmetic */
42  if ((void *)tcp + len > data_end)
43      return -1;
44
45  *nh_off += len;
46  *tcphdr = tcp;
47
48  return len;
49 }
```

This parsing code is new, but it is very similar to the parsing code we have seen in Exercise 3. The main difference is that we need to check the length of the UDP and TCP headers, which can be variable-length. For UDP, the length is specified in the `len` field of the header, while for TCP, the length is specified in the `doff` field of the header.

- **Implement the main XDP program to rewrite the destination port number of TCP and UDP packets**

```

1 SEC("xdp")
2 int xdp_packet_rewriting(struct xdp_md *ctx) {
3     void *data_end = (void *)(long)ctx->data_end;
4     void *data = (void *)(long)ctx->data;
5
6     __u16 nf_off = 0;
7     struct ethhdr *eth;
8     struct udphdr *udphdr;
9     struct tcphdr *tcphdr;
10    int eth_type;
11    struct datarec *rec;
12    int key = 0;
13    int action = XDP_PASS;
14
15    bpf_printk("Packet received");
16
17    eth_type = parse_ethhdr(data, data_end, &nf_off, &eth);
18
19    /* If the packet is ARP we should let him pass */
20    if (eth_type == bpf_ntohs(ETH_P_ARP)) {
21        action = XDP_PASS;
22        goto end;
23    }
24
25    if (eth_type != bpf_ntohs(ETH_P_IP)) {
26        action = XDP_ABORTED;
27        goto end;
28    }
29
30    bpf_printk("Packet is IPv4");
31
32    // Handle IPv4 and parse TCP and UDP headers
33    int ip_type;
34    struct iphdr *iphdr;
35    ip_type = parse_iphdr(data, data_end, &nf_off, &iphdr);
36
37    if (ip_type == IPPROTO_UDP) {
38        bpf_printk("Packet is UDP");
39        if (
40            parse_udphdr(data, data_end, &nf_off, &udphdr) < 0
41        ) {
42            action = XDP_ABORTED;
```

```

43         goto end;
44     }
45     __u16 port = bpf_htons(bpf_ntohs(udphdr->dest) - 1);
46     if (port > 0)
47         udphdr->dest = port;
48 } else if (ip_type == IPPROTO_TCP) {
49     bpf_printk("Packet is TCP");
50     if (
51         parse_tcphdr(data, data_end, &nf_off, &tcphdr) < 0
52     ) {
53         action = XDP_ABORTED;
54         goto end;
55     }
56     __u16 port = bpf_htons(bpf_ntohs(tcphdr->dest) - 1);
57     if (port > 0)
58         tcphdr->dest = port;
59 } else {
60     bpf_printk("Packet is not TCP or UDP");
61     action = XDP_ABORTED;
62     goto end;
63 }
64
65 out:
66     bpf_printk("Packet passed");
67     rec = bpf_map_lookup_elem(&xdp_stats_map, &key);
68     if (!rec) {
69         return XDP_ABORTED;
70     }
71
72     __u64 bytes = data_end - data;
73     __sync_fetch_and_add(&rec->rx_packets, 1);
74     __sync_fetch_and_add(&rec->rx_bytes, bytes);
75
76 end:
77     return action;
78 }
```

Here, we first parse the Ethernet header to check if the packet is an ARP packet. If it is, we let it pass without modification. If it is not an ARP packet, we check if it is an IPv4 packet. If it is not an IPv4 packet, we abort the packet. If it is an IPv4 packet, we parse the IP header to check if it is a TCP or UDP packet. If it is a TCP or UDP packet, we rewrite the destination port number to be one less than its original value. Finally, we update the packet and byte counters in the map and return the appropriate action.

The user-space program to load this XDP program and read the counters from the map is similar to the one we have seen in Exercise 2/3. The full code of the exercise is available here:



[Full code on GitHub](#)

References

- [1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 89–92. San Jose, USA, 2010.
- [2] Tom Barbette, Chen Tang, Haoran Yao, Dejan Kostić, Gerald Q Maguire Jr, Panagiotis Papadimitratos, and Marco Chiesa. A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 667–683, 2020.
- [3] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Huelscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A fast and reliable software network load balancer. In *Nsdi*, volume 16, pages 523–535, 2016.
- [4] Antichi Gianni. Network Computing. Slides from the HPC-E master’s degree course on Politecnico di Milano, 2024.
- [5] Albert Greenberg, Dave Maltz, Guohan Lu, Jiaxin Cao, Ratul Mahajan, and Yibo Zhu. Packet-level telemetry in large datacenter networks. In *SIGCOMM’15*, August 2015.
- [6] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. *SIGCOMM Comput. Commun. Rev.*, 45(4):139–152, August 2015.
- [7] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.
- [8] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. Flowradar: a better netflow for data centers. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI’16, pages 311–324, USA, 2016. USENIX Association.
- [9] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. Understanding pcie performance for end host networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 327–341, 2018.
- [10] Weiwu Pang, Sourav Panda, Jehangir Amjad, Christophe Diot, and Ramesh Govindan. CloudCluster: Unearthing the functional structure of a cloud service. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1213–1230, Renton, WA, April 2022. USENIX Association.

- [11] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, et al. Ananta: Cloud scale load balancing. *ACM SIGCOMM Computer Communication Review*, 43(4):207–218, 2013.
- [12] Yu Zhou, Chen Sun, Hongqiang Harry Liu, Rui Miao, Shi Bai, Bo Li, Zhilong Zheng, Lingjun Zhu, Zhen Shen, Yongqing Xi, Pengcheng Zhang, Dennis Cai, Ming Zhang, and Mingwei Xu. Flow event telemetry on programmable data plane. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM ’20, pages 76–89, New York, NY, USA, 2020. Association for Computing Machinery.
- [13] Danyang Zhuo, Monia Ghobadi, Ratul Mahajan, Klaus-Tycho Förster, Arvind Krishnamurthy, and Thomas Anderson. Understanding and mitigating packet corruption in data center networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM ’17, pages 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

Index

Symbols

1-hash Bloom Filter	55
---------------------	----

A

Accelerated Receive Flow Steering (aRFS)	169
Aggregation Layer (Distribution Layer)	15
Asymptotic Approximation of False Positive Rate (FPR)	59

B

Bisection Bandwidth	19
Bloom Filter	57
BMv2 (Behavioral Model v2)	207
Bursty Traffic (Incast)	93
Busy Waiting	158

C

Checksum Offload	172
Classic BPF (Classic Berkeley Packet Filter)	265
Clos Network	22
Completion Queue Elements (CQE)	198
Compute Express Link (CXL)	179
Connection Affinity	115
Consistent Hashing	118
Control Plane	39
Core Layer (Backbone Layer)	15
Count-Min Sketch	65
Counting Bloom Filter	60

D

Data Direct I/O (DDIO)	170
Data Plane	39
Data Plane Development Kit (DPDK)	193
Datacenter	6
Direct Server Return (DSR)	108

E

East-West traffic	11
eBPF (extended Berkeley Packet Filter)	265
Edge Layer (Access Layer)	15
End-Host	144
Equal Cost Multi-Path (ECMP)	91
Overflow	71

F

Faild	135
False Negative Rate	56
False Positive Rate (FPR)	56, 58
Fat-Tree	22
Flow Filter	77

Flow Skew Across Racks	93
FlowCount	76
FlowRadar	75
FlowXOR	76
Forwarding Engine	30
Full-Bisection Bandwidth	19
G	
Generic Receive Offload (GRO)	186
Google Jupiter Fabric	25
H	
Hash Collision	91
Hash Function	52
Hash Table	52
Heavy Hitter	244
Hedera	97
Hotspot	86
HULA	101
Hybrid Cheetah	132
Hyperscale datacenters	97
I	
In-Band Network Telemetry	82
INT-MD (eMbed Data)	83
INT-MX (eMbed instruct(X)ions)	83
INT-XD (eXport Data)	83
Interrupt Coalescing	156
Interrupt Request (IRQ)	152
Invertible Bloom Lookup Table	61
J	
JIT (Just-In-Time compilation)	268
L	
Large Receive Offload (LRO)	173
Layer 3 (L3)	87
Layer 3 Load Balancing	87
Layer 4 Load Balancing	104
Leaky DMA Problem	171
Load Balancer (LB)	104
Longest Prefix match (LPM)	48
M	
Mininet	209
Multi-Tenancy	8
N	
NAPI (New API)	160
NAT (Network Address Translation)	109
Netfilter	188

Network Monitoring	69
NIC Driver	181
North-South traffic	11
O	
One-Sided RDMA Verbs	197
OpenFlow	32
OSI (Open Systems Interconnection) Model	87
Over-Subscription	17, 20
P	
P4 (Programming Protocol-independent Packet Processors)	43
P4 Architecture Model	44
P4 Compiler	45
P4Runtime	205
Packet Buffers	150
Packet Header Vector	46
Packet Spraying	89
PacketCount	76
PCIe (Peripheral Component Interconnect Express)	145, 174
Persistency	116
Points of Presence (PoPs)	135
Polling	158
Proactive Mode	34
Protocol-Independent Switch Architecture (PISA)	40, 46
Q	
Queue Pair (QP)	197
R	
RDMA (Remote Direct Memory Access)	196
Reactive Mode	33
Receive Flow Steering (RFS)	168
Receive Livelock	154
Receive-Side Scaling (RSS)	165
RX Descriptor Ring	151
S	
SDN Controller	30
Separate Chaining	53
Single Decode	78
Single-Tenancy	8
Smart NICs (Network Interface Cards)	108
Socket Buffer	184
Software-Defined Networking (SDN)	27, 30
Standard Offloads	172
Stateful Cheetah	126
Stateful P4 program	239
Stateless Cheetah	122

T

Tail Call	274
TCP Incast	24
TCP Offload Engine (TOE)	195
TCP Segmentation Offload (TSO)	172
Ternary Content Addressable Memory (TCAM)	49
Three-Tier design	15
Transaction Layer Packet (TLP)	175
Two-Sided RDMA Verbs	197

U

UDP Fragmentation Offload (UFO)	172
---------------------------------	-----

V

V1Model	208
Verb	197
Virtual IP	105
VLAN (Virtual LAN)	227

W

Work Queue Elements (WQE)	198
---------------------------	-----

X

XDP (eXpress Data Path)	276
-------------------------	-----