Network Computing - Notes - v0.8.0

260236

January 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.

Contents

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).

❓ 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**.

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.

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.

⚠ 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

❓ 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**.

❓ *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.

⚠ 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:

✗ **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 16): 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.

✗ **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**.

✗ **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 $\left(\frac{k}{2}\right)^2$ core switches, where each connects to k pods.

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

- Each pod contains:
  - $\left(\frac{4}{2}\right)^2 = 4$ 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 $\left(\frac{k}{2}\right)^2 = 4$ core switches.

[1] A **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-to-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.

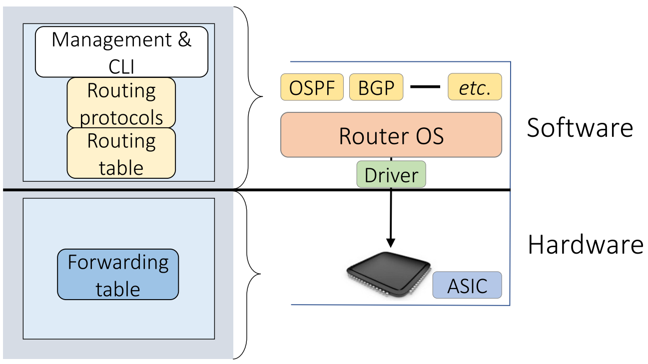


Figure 1: Legacy router and switch architecture.

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 o 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:

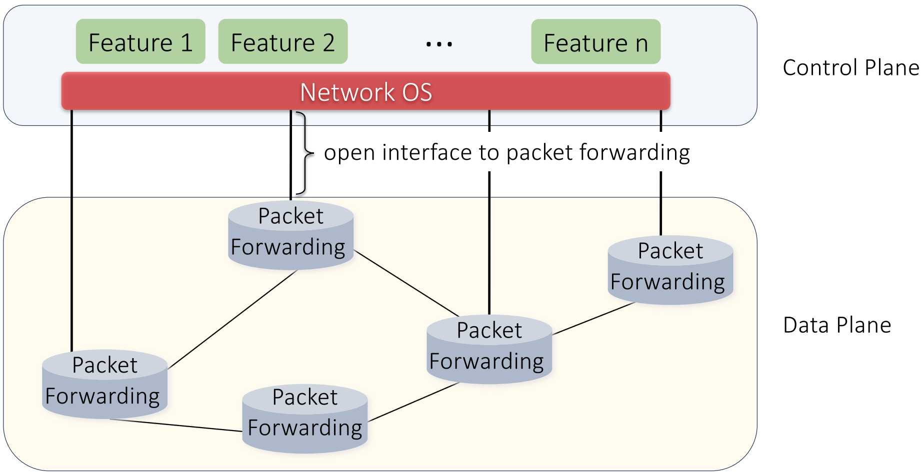
- **Southbound Interface**: Connects the controller to the data plane devices (e.g., **OpenFlow protocol**). It instructs switches via OpenFlow or similar protocols, installing/removing flow rules and collecting stats.
- **Northbound Interface**: Allows **applications to interact with the controller** 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.

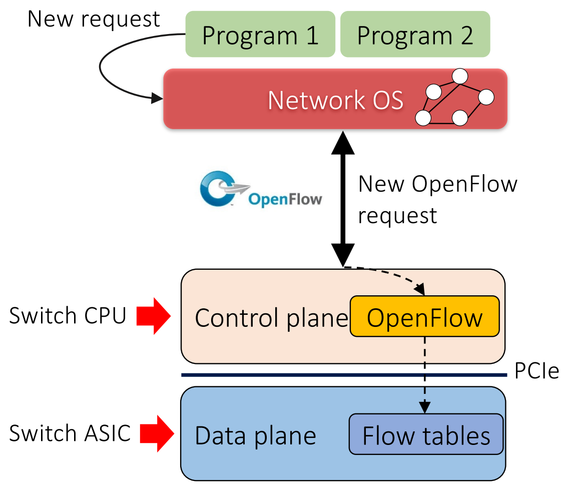
Flow table operation:

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**

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 31). 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 26). 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 31. 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.

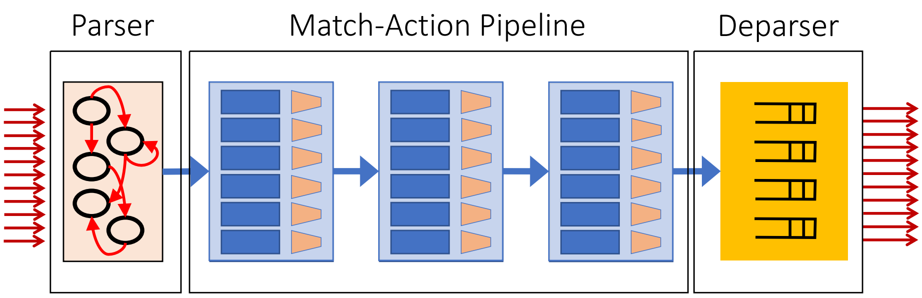


Figure 2: PISA architecture.

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 compile 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 45), 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 we input data and instantly tells we **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 supper 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

❓ 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 factory, quality of the hash function.

- **Probabilistic** (section 4.4, page 54):
  - ✔ **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:

```
index = hash("hello")
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**:

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

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** tan 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 `hash(x)`
  2. Set the bit of the result of the hash function to 1: `bit[hash(x)] = 1`
- To check if `x` is in the set, we simply: `bit[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`" → `bit[hash(Hello)] == 1`? $\xrightarrow{\text{YES}}$ ✔ return `true`
- "`Bye`" → `bit[hash(Bye)] == 1`? $\xrightarrow{\text{NO}}$ ✘ return `false`
- "`P4`" → `bit[hash(P4)] == 1`? $\xrightarrow{\text{YES}}$ ✔ 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**:

$$1 - \frac{1}{M} \tag{2}$$

- **Probability** that a **bit stays <u>0</u> after** N **insertions**:

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

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

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

- 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:

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

2. For each $h_i(x)$, set the bit at position $h_1(x)$ to `1`.
   - $h_1(x) = 1$
   - $h_2(x) = 1$
   - ...
   - $h_K(x) = 1$

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**:

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

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

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

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× 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**.

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)**:

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

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

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

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

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

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

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:

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

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 `counter[`$h_i(x)$`]`.
- When **deleting**, for each hash $h_i(x)$, decrement `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.

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**.

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:

1. Initial IBLT contains:

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

2. First, a cell with `count = 1` reveals:

   - `(7, 98)` ⇒ 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

3. After update:

   (a) Next, find `(152, 3)` ⇒ decode and remove

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

   (b) Finally, we can easily retrieve the last one which is `50, 45`.

4. 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 we 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

Image 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 highlights 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*" [12] article, they showing 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 mu 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*" [11] 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 timeouts.

- In the *CloudCluster* [9] article, they push toward deep programmability and visibility withing 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 70)
      - *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 modify 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 th 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.

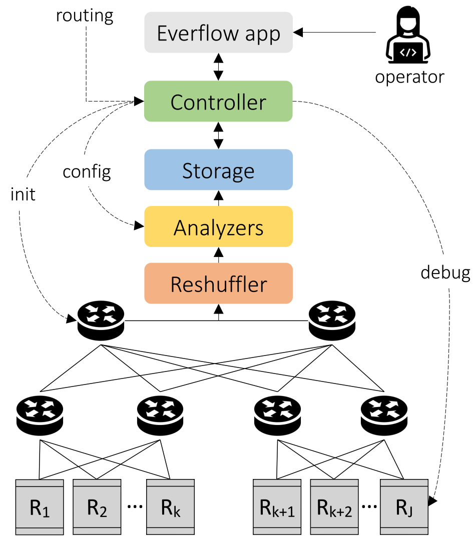


Figure 3: Summary of Everflow's End-to-End operation:

- **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 reveled:
   - 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.

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 our 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 60), 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.

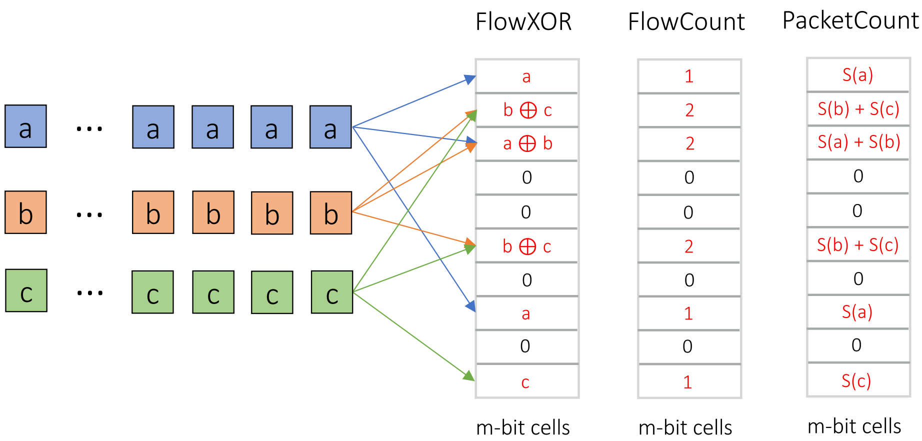
Each switch maintains three tables of m cells:

- **FlowXOR**. XOR[2] 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**.



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.

[2]XOR (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**.

❓ 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**.

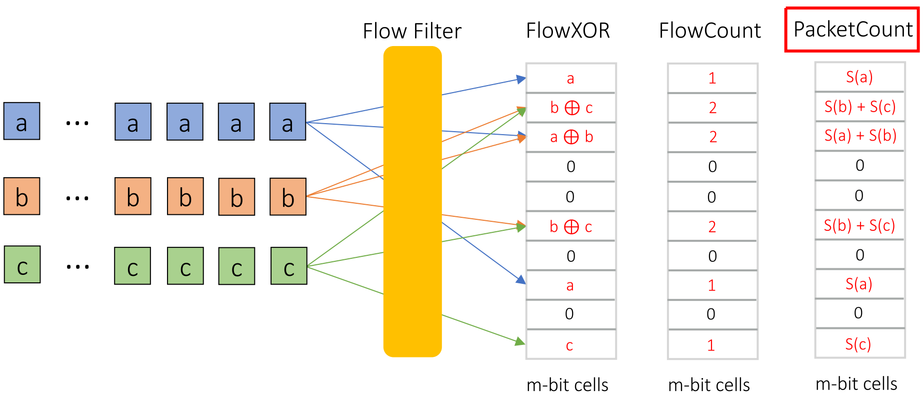


Figure 4: Correct representation of the data structure used in FlowRadar.

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 `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 `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 that 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?

Image 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:

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

   - 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.

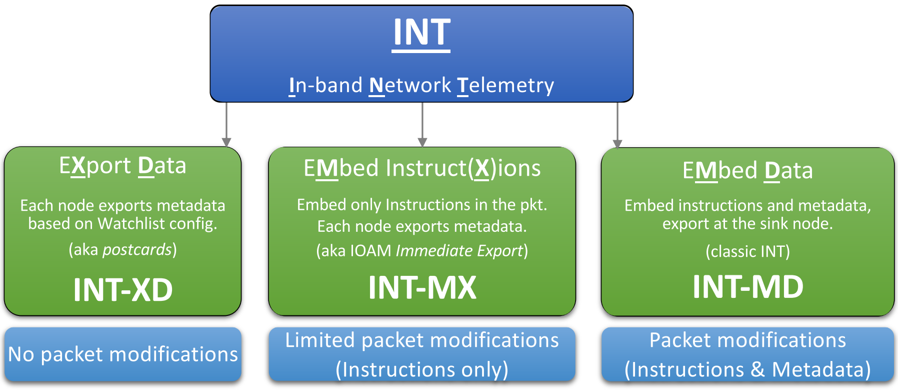


Figure 5: INT modes.

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 hotspots[3] 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.

[3] A **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.

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:

- Choosing **which path** a packet (or a whole flow) takes among those equal-cost routes.
- 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.

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 links. 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 96) and HULA (P4-based, page 100).

✅ 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 $\rightarrow$ 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 $\rightarrow$ 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:

```
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):

```
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., $0 \ldots 2^{32-1}$). But the switch only has a small number of next-hops (k paths). We need a consistent, deterministic way to map "large space $\xrightarrow{to}$ small space".
The Solution: Modulo. Compute:

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

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

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

```

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 90):

- 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 datacenters[4] 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 29) 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 31).

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 computers 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?".

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**.

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.

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 receives 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 42). 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 90): 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 96): 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 92) 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 SLAs[5] (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**.

[5] **SLA** 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 86.

Traditionally, load balancers are place 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 90). 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)** $\rightarrow$ **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 is 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)** $\rightarrow$ **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)** $\rightarrow$ **Seats**. Once at the gate, each passenger is direct 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 switches[6] and smart NICs[7]. Alibaba's architecture often uses **NAT-style translation**.

[6] Programmable 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 38). An example of a programmable switch is one that supports the P4 programming language (page 42), which enables users to define how packets are processed and routed within the switch.

[7] **Smart 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** [10] 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:

$$\begin{array}{lll} \text{Client} \longrightarrow & \text{TCP connection \#1} & \longrightarrow \\ & \text{[Load Balancer]} & \longrightarrow \\ & \text{TCP connection \#2} & \longrightarrow \text{Backend Server} \end{array}$$

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.

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:

  $$\text{server} = \text{hash}\left(\texttt{src\_ip}, \texttt{src\_port}, \texttt{dst\_ip}, \texttt{dst\_port}, \texttt{protocol}\right) \mod N$$

  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 **typically 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 switces** 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:

$$\begin{aligned} \text{h} &= \texttt{hash}(\texttt{src\_IP}, \texttt{src\_port}, \texttt{dst\_IP}, \texttt{dst\_port}, \texttt{protocol}) \\ \text{backend} &= \text{h} \mod N \quad (\text{where } N \text{ is number of servers}) \end{aligned}$$

  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:

$$(\text{5-tuple}) \to \text{backend server}$$

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)**.

Persistency and Connection Affinity

When we connect to a website, for example:

$$\text{Client} \to \text{VIP} = \texttt{203.0.113.10} \quad (\texttt{www.example.com})$$

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).

$$\text{Persistency} \implies \text{Connection Affinity}$$

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:

$$f_\ell : F \to B$$

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**:

$$\forall \phi \in F : \forall p_1, p_2 \in \text{Packets}(\phi) \cap \text{SeenBy}(\ell),$$
$$\text{decision}_\ell(p_1) = \text{decision}_\ell(p_2) = f_\ell(\phi)$$

Persistency (*global*) requires the **mapping to be independent of the LB instance**:

$$\forall \ell_1, \ell_2 \in L, \forall \phi \in F : \quad f_{\ell_1}(\phi) = f_{\ell_2}(\phi)$$

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:

  (`src_IP`, `src_port`, `dst_IP`, `dst_port`, `protocol`) → `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:

  $$\texttt{backend} = \texttt{hash}\,(\text{5-tuple}) \mod N$$

  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 flow are remapped (unless consistent hashing is used).

Used by **Google's Maglev**, **ECMP routing**, and **DSR-based systems**.

Persistency with cluster changes

Even if the load balancer guarantees persistency 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:

$$\texttt{backend} = \texttt{hash}\,(\text{5-tuple}) \mod N$$

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:

$$\text{backend} = \text{hash}(\text{5-tuple}) \mod N$$

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".

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 slide 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 113), we studied **consistent hashing** and **uniform deterministic load-balancing functions**, which help maintain *persistency* 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?*"

❓ 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.

❓ **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 th previous section (stateless persistence);
- 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)* → *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 113).
- 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**

[8]The **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; the 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:

$$\texttt{cookie} = \texttt{hashS(connID)} \oplus \texttt{serverID}$$

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 `hashS`)? 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.

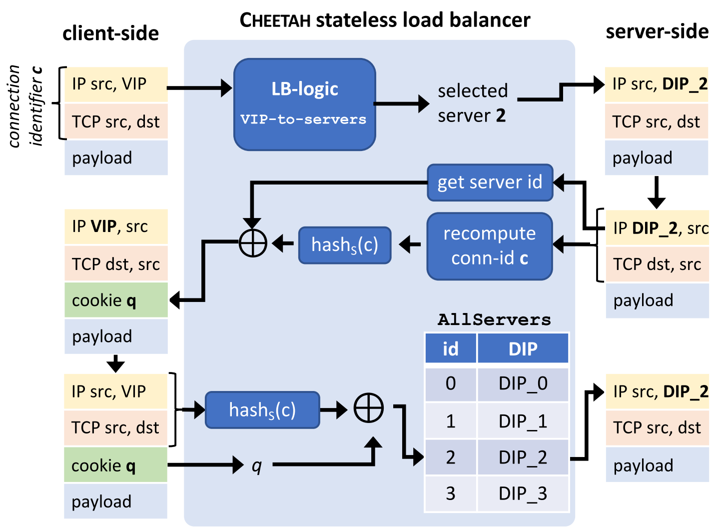


Figure 6: The diagram [2] shows the **complete life-cycle of a connection** through a ***stateless Cheetah* load balancer**. It illustrates **how per-connection consistency (PCC)** is guaranteed *without* keeping any per-flow state in memory.

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:

$$\texttt{connID} \rightarrow \texttt{server}$$

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 sores.** 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 sores.** 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 "*pops*" one free slot from this stack.

- Cookie
  - **What it sores.** 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 slow 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 filed (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 121), 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 `hash_S(connID)` in	***Stateless* Cheetah**	***Stateful* Cheetah**
Used to decode server ID (cookie ⊕ 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 filed (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:

$$\text{cookie size} = \log_2 m + r \tag{10}$$

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.

And about memory usage? The total capacity is calculated as:

$$\text{connections} = m \times 2^r \tag{11}$$

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:

$$65,536 \times 22 \text{ bytes} \approx 1.4 \text{ MB} \quad (\text{IPv4} + \text{MD5})$$

$$65,536 \times 50 \text{ bytes} \approx 3.3 \text{ MB} \quad (\text{IPv6} + \text{SHA-256})$$

With $m = 64$ partitions, total memory is:

$$64 \times 1.4 \text{ MB} \approx 90 \text{ MB} \quad (\text{IPv4} + \text{MD5})$$

$$64 \times 3.3 \text{ MB} \approx 210 \text{ MB} \quad (\text{IPv6} + \text{SHA-256})$$

But remember, this is for tracking over 4 million concurrent connections! So the **memory overhead per connection** is only about:

$$\frac{90 \text{ MB}}{4,194,304} \approx 21.5 \text{ bytes/connection} \quad (\text{IPv4} + \text{MD5})$$

$$\frac{210 \text{ MB}}{4,194,304} \approx 50 \text{ bytes/connection} \quad (\text{IPv6} + \text{SHA-256})$$

This is very manageable for modern servers. So, **stateful Cheetah** can track **millions of connections** with **tens of MBs of memory**, while still providing **constant-time operations** and **per-connection consistency**.

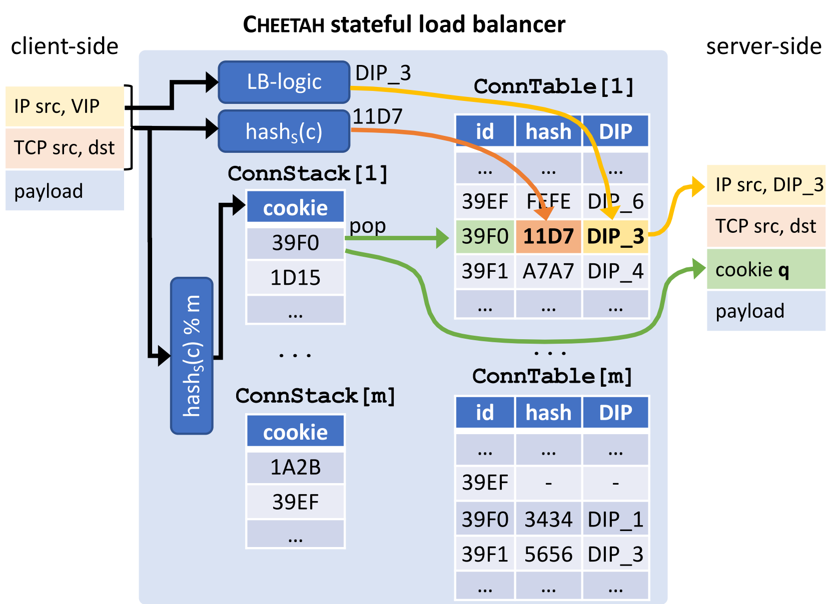


Figure 7: The diagram [2] shows the ***Cheetah stateful*** LB operations for the first packet of a connection. They do not show the stateless cookie for identifying the stateful LB. The VIP-to-servers is included within the LB-logic and not shown. The server performs Direct Server Return (DSR) so the 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 126)
  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).

✔ 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.

? 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.

❓ 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 109), 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.

[10] Because they are designed for more stable traffic, they may not handle the high variability and short-lived connections effectively.

[11] This 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 120), 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.

[12] This can lead to complex recovery procedures and increased downtime.
[13] Large per-flow state tables increase memory pressure and complicate recovery after crashes.
[14] This 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.

◎ 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**.

- **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.

  ✔ That **reduces** the likelihood of **bugs** and **eases troubleshooting**.

- **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-scoped 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 103).

1. The first and most critical problem is **connection affinity** (page 114). 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 121). 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 111), 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 120) and **Faild** (page 134), 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.

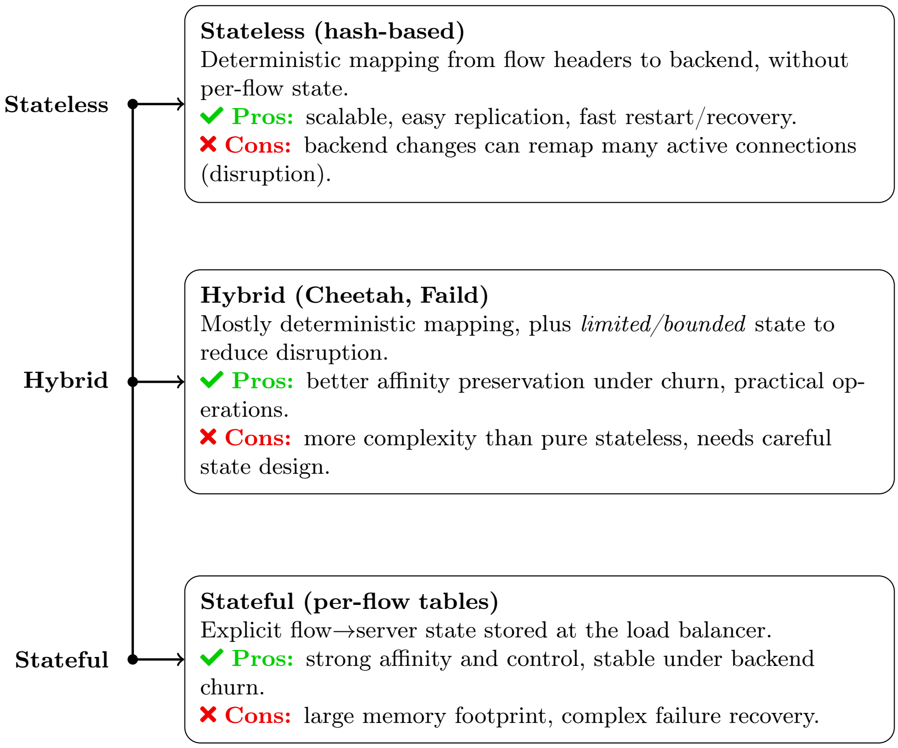


Figure 8: Layer-4 load balancing design spectrum: stateless vs hybrid vs stateful approaches.

⚖ 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 139).

- Research systems such as **Cheetah** (page 120) 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 134), 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).

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-Hielscher, 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, page 311–324, USA, 2016. USENIX Association.

[9] 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.

[10] 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.

[11] 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, page 76–89, New York, NY, USA, 2020. Association for Computing Machinery.

[12] 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, page 362–375, New York, NY, USA, 2017. Association for Computing Machinery.

Index