Network Computing - Notes - v0.4.0

260236

April 2025

Preface

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

- Course slides. [1]

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 15): 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.

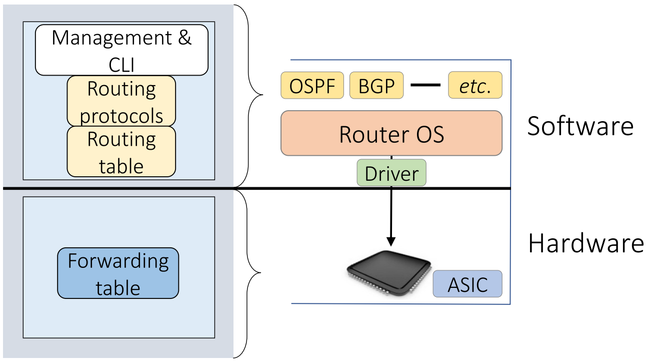


Figure 1: Legacy router and switch architecture.

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:

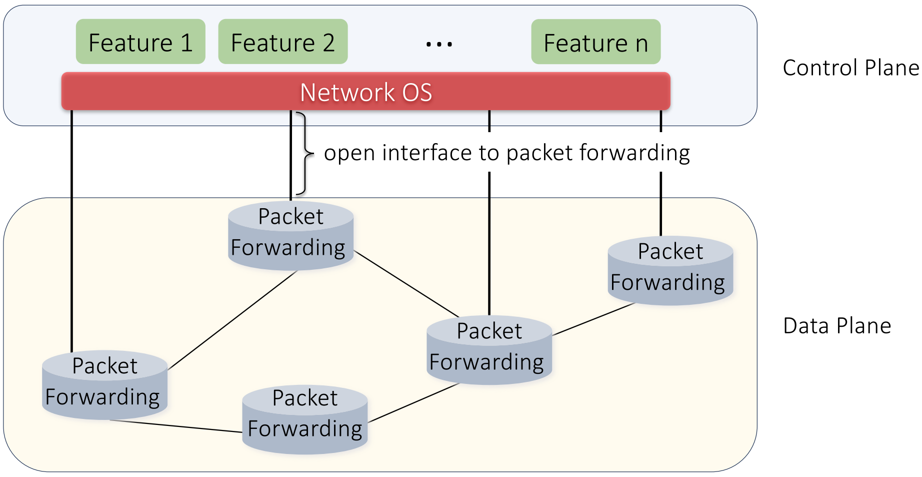
- **Southbound Interface**: Connects the controller to the data plane devices (e.g., **OpenFlow protocol**). It instructs switches via OpenFlow or similar protocols, installing/removing flow rules and collecting stats.
- **Northbound Interface**: Allows **applications to interact with the controller**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

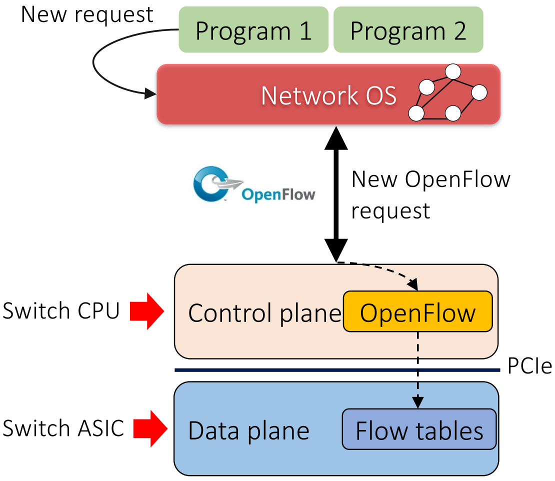
Flow table operation:

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

Example 2: OpenFlow

1. New packet arrives at switch.
2. Match?
   - Yes → forward according to rule.
   - No → 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 30). 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 25). 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 30. 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.

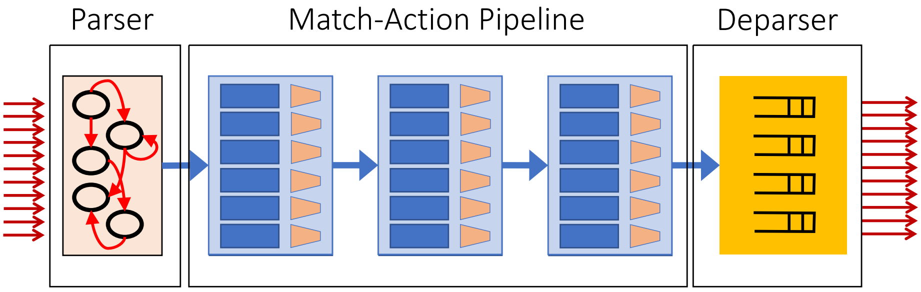


Figure 2: PISA architecture.

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 44), 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 53):

  ✓ **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` ⇒ `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`.
   - $h_1(x) = 1$
   - $h_2(x) = 1$
   - $\dots$
   - $h_K(x) = 1$

How Lookup Works

To check whether an element is in the set:

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

⚠ False Positives

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

% Probability Analysis

Let:

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

Then:

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

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

- **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 ⇒ lower FPR
2. **Number of Hashes** K: more hashes ⇒ 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 $h_i\left(x\right)$, increment `counter[`$h_i\left(x\right)$`]`.
- When **deleting**, for each hash $h_i\left(x\right)$, decrement `counter[`$h_i\left(x\right)$`]`.

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 ⇒ **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.

References

[1] Antichi Gianni. Network Computing. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

Index