Computing Infrastructures - Notes - v1.9.0

260236

June 2025

Preface

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

- The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. [2]
- Quantitative System Performance: Computer System Analysis Using Queueing Network Models. [4]
- Course slides. [6]

About:

GitHub repository

These notes are an unofficial resource and shouldn't replace the course material or any other book on computing infrastructure. 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 Data Center and Computing Infrastructure

There's no single definition of a Data Center, but it can be summarized as follows.

Definition 1: Data Center

Data Centers are buildings where multiple servers and communications equipment are co-located for common environmental requirements, physical security, and ease of maintenance. [2]

Definition 2: Computing Infrastructure

A **Computing Infrastructure** (or IT Infrastructure) is a technological infrastructure that provides hardware and software for computation to other systems and services.

Traditional data centres have the following characteristics:

- **Host a large number** of relatively small or medium sized **applications**;
- Each **application is running on a dedicated HW infrastructure** that is de-coupled and protected from other systems in the same facility;
- **Applications tend not to communicate each other**.

Those **data centers host** hardware and software **for multiple organizational units** or even **different companies**.

2 Hardware Infrastructures

2.1 System-level

2.1.1 Computing Infrastructures and Data Center Architectures

2.1.1.1 Overview of Computing Infrastructures

The **Computing Continuum** is a **distributed computing environment** that seamlessly **integrates endpoints** (IoT devices), **edge computing**, and **cloud computing** to **optimize data processing across different layers of infrastructure**. Therefore, it consists of three main layer:

1. **Endpoints** (IoT Devices & Sensors) - The **Data Collectors**. At the very edge of the continuum, we find Endpoints. They are **small**, **low-power devices** embedded with sensors that **collect and transmit data**.

   Examples

   - A **temperature sensor** in a smart home, detecting room conditions.
   - A **fitness tracker** measuring heart rate and step count.
   - A **manufacturing robot** equipped with vibration sensors to detect faults.

   Challenges. While IoT devices are **great at collecting data**, they have **limited processing power** and cannot perform complex calculations. As a result, they often send raw data to the next level for further analysis.

2. **Edge & Fog Computing** - **Processing Data Close to the Source**. To **reduce the dependence on cloud processing**, the Edge Computing layer introduces **local computation** at the network's edge. This means that instead of sending all raw data to the cloud, some processing occurs closer to where the data is generated.

   Examples

   - **Smart security cameras** analyzing video footage locally to detect intruders before sending alerts to a cloud-based security system.
   - **Autonomous vehicles** processing sensor data in real-time to make split-second driving decisions without relying on a remote server.

   Fog Computing vs. Edge Computing. **Fog Computing** refers to **processing data at an intermediate level**, such as an IoT gateway or a local network node.

   - **Edge Computing** happens **directly on the device** producing the data.
   - **Fog Computing** moves **processing to a nearby network gateway** or **edge server**.

Advantages

- **Lower latency**: Decisions are made **faster** since they don't rely on a cloud response.
- **Bandwidth savings**: Only important data is sent to the cloud, **reducing network congestion**.
- **Privacy and security**: Some data can remain **local**, avoiding unnecessary cloud exposure.

Challenges. While Edge Computing is beneficial, it **requires additional hardware** and **power** at the network's edge, which can **increase infrastructure complexity and costs**.

3. **Cloud Computing** - The **Powerhouse of Large-Scale Processing**. At the top of the continuum, cloud computing provides **massive storage and computation capabilities**. It is well-suited for applications that require significant computational resources, such as big data analytics, machine learning training, scalable web services (e.g., Netflix, YouTube, Amazon AWS).

   Examples

   - **AI model training** is typically done in cloud data centers because it requires significant GPU/TPU power.
   - **E-commerce platforms** like Amazon use cloud computing to manage global inventory and transactions.
   - **Streaming services** store and distribute high-quality video content from cloud servers.

   Advantages

   - **Scalability**: Computing power can be adjusted based on demand.
   - **Resource efficiency**: Data centers consolidate computing, making operations more cost-effective.
   - **Advanced analytics**: Supports complex workloads like deep learning and data mining.

   Challenges

   - **Latency issues** make real-time decision-making difficult.
   - **Network dependency**: A constant internet connection is required.
   - **High operational costs** for companies relying solely on cloud resources.

To summarize Computing Continuum creates a balanced computing environment by ensuring that workloads are distributed intelligently across IoT, edge, and cloud layers.

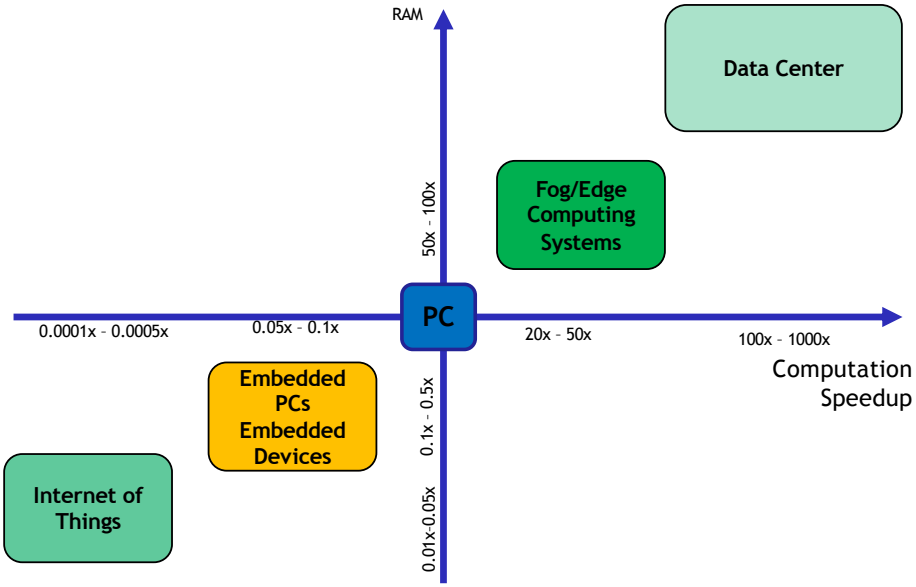
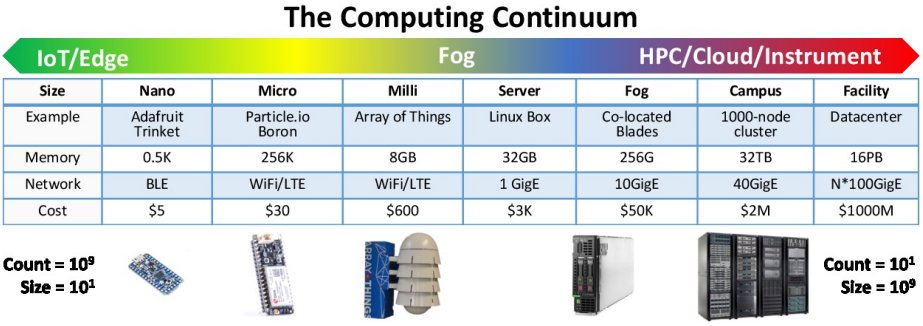


Figure 1: **Examples** of Computing Infrastructures. [7]



Size	Nano	Micro	Milli	Server	Fog	Campus	Facility
Example	Adafruit Trinket	Particle.io Boron	Array of Things	Linux Box	Co-located Blades	1000-node cluster	Datacenter
Memory	0.5K	256K	8GB	32GB	256G	32TB	16PB
Network	BLE	WiFi/LTE	WiFi/LTE	1 GigE	10GigE	40GigE	N*100GigE
Cost	$5	$30	$600	$3K	$50K	$2M	$1000M

Figure 2: This figure represents the Computing Continuum, how different computing layers interact to process data efficiently. It is a hierarchical structure moving from IoT devices at the left (closest to the data source) to cloud computing at the right (handling large-scale processing). [7]

In the following pages, we analyze the computing infrastructures mentioned in the previous example.

Data Centers

The definition of a Data Centers can be found on page 5.

✔ Data Centers Advantages

- **Lower IT costs.**
- **High Performance.**
- **Instant software updates.**
- **"Unlimited" storage capacity.**
- **Increased data reliability.**
- **Universal data access.**
- **Device Independence.**

👎 Data Centers Disadvantages

- **Require a constant internet connection.**
- **Do not work well with low-speed connections.**
- **Hardware Features might be limited.**
- **Privacy and security issues.**
- **High power Consumption.**
- **Latency in taking decision.**

Internet-of-Things (IoT)

An **Internet of Things (IoT) device is any everyday object embedded with sensors, software, and internet connectivity**.

This allows to collect and exchange data with other devices and systems, typically over the internet, with limited need of process and store data.

Some **examples** are Arduino, STM32, ESP32, Particle Argon.

✔ **Internet-of-Things Advantages**

- **Highly Pervasive**.
- **Wireless connection**.
- **Battery Powered**.
- **Low costs**.
- **Sensing and actuating**.

👎 **Internet-of-Things Disadvantages**

- **Low computing ability**.
- **Constraints on energy**.
- **Constraints on memory (RAM/FLASH)**.
- **Difficulties in programming**.

Embedded (System) PCs

An **Embedded System** is a computer system, a combination of a computer processor, computer memory, and input/output peripheral devices, that has a dedicated function within a larger mechanical or electronic system.

A few **examples**: Odroid, Raspberry, jetson nano, Google Coral.

✔ **Embedded System Advantages**

- **Persuasive computing**.
- **High performance unit**.
- **Availability of development boards**.
- **Programmed as PC**.
- **Large community**.

👎 **Embedded System Disadvantages**

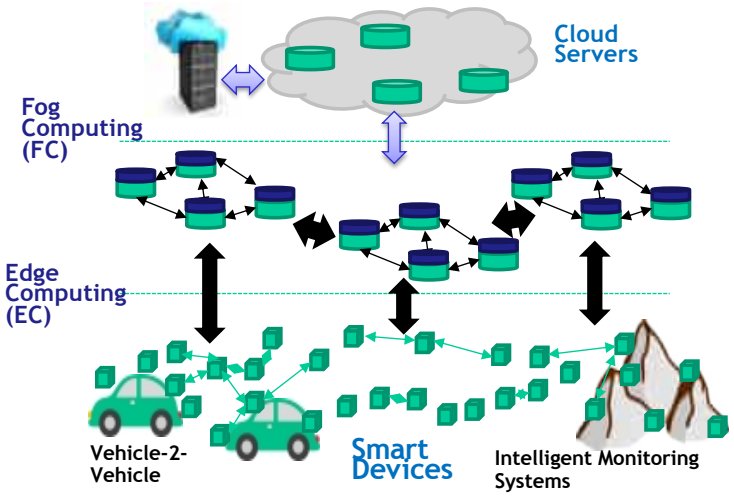
- **Pretty high power consumption**.
- **(Some) Hardware design has to be done**.

Edge/Fog Computing Systems

The key **difference** between **Fog Computing** and **Edge Computing** is associated with the location **where the data is processed**:

- In **edge computing**, the data is processed closest to the sensors.
- In **fog computing**, the computing is moved to processors linked to a local area network (IoT gateway).

Edge computing places the intelligence in the connected devices themselves, whereas, fog computing puts in the local area network.



✔ Fog/Edge Advantages

- **High computational capacity.**
- **Distributed computing.**
- **Privacy and security.**
- **Reduced Latency in making a decision.**

👎 Fog/Edge Disadvantages

- **Require a power connection.**
- **Require connection with the Cloud.**

Feature	Edge Computing	Fog Computing
Location	Directly on device or nearby device.	Intermediary devices between edge and cloud.
Processing Power	Limited due to device constraints, sending data to central server for analysis.	More powerful than edge devices. However, sending data to a central server for analysis.
Primary Function	Real-time decision-making, low latency. However, central server analyzing combined data and sending only relevant information further.	Pre-process and aggregate data, reduce bandwidth usage. However, central server analyzing combined data and sending only relevant information further.
Advantages	Low latency, reduced reliance on cloud, security for sensitive data.	Bandwidth efficiency, lower cloud costs, complex analysis capabilities.
Disadvantages	Limited processing power, single device focus.	Increased complexity, additional infrastructure cost.

Table 1: Differences between Edge and Fog Computing Systems.

2.1.1.2 The Datacenter as a Computer

The concept of treating the datacenter itself as a singular, unified computing system represents a significant evolution in computing infrastructures. Traditionally, computing resources were individually managed and isolated (client-server computing); now, the paradigm shifts towards collective resource management and holistic infrastructure design (datacenter-based computing, particularly Warehouse-Scale Computers, WSCs).

User Experience Improvements

For end-users, the shift to centralized computing in datacenters provides significant advantages:

- **Ease of Management**. **No need for local configuration or backups**.
  - ✔ Users no longer need to manually configure and maintain software or hardware on their personal devices.
  - ✔ Centralized cloud-based systems handle updates, security patches, and maintenance automatically.
- **Ubiquity of Access**
  - ✔ **Users can access** services and data **from any device**, **anywhere**, as long as they have an internet connection.
  - ✔ This allows for seamless synchronization[1] across multiple devices (e.g., accessing Google Drive files from a laptop, tablet, or phone).
- **Reduced Hardware Dependencies**
  - ✔ **Computing power** is **shifted** from end-user devices **to cloud services**.
  - ✔ Even low-powered devices (e.g., smartphones, Chromebooks) can perform complex tasks by offloading computation to datacenters.

  For example, cloud-based applications like Google Docs allow real-time document editing across multiple devices without requiring powerful local hardware.

[1] Seamless synchronization: The automatic and continuous updating of data across multiple devices or platforms without user intervention, ensuring consistency and accessibility in real-time.

✅ Advantages for Service Providers (vendors)

From the perspective of software and hardware vendors, the transition to datacenter computing brings several operational and economic benefits.

- **Software-as-a-Service (SaaS) Enables Faster Development**

  ✗ Traditional software development involved deploying updates individually to millions of heterogeneous client devices.

  ✓ Now, **cloud-based applications** allow:

  * Centralized software deployment.
  * Rapid updates and feature rollouts.
  * Easier bug fixes and security patches.

  For example, instead of releasing a new version of Microsoft Office every few years, Microsoft now provides Microsoft 365, where updates are continuously applied.

- **Simplified Hardware Deployment**

  ✗ Traditional software development required compatibility with millions of different client hardware configurations.

  ✓ Now, with server-side computing, companies **only need to support a few well-tested hardware platforms** in their datacenters.

  For example, instead of optimizing software for thousands of different PC models, Google can optimize Gmail and YouTube to run efficiently in their controlled cloud environment.

- **Better Resource Utilization**

  ✓ Datacenters allow for **efficient resource allocation**, ensuring that computing power is shared dynamically across multiple applications.

  ✓ Cloud providers can achieve **high resource utilization**, reducing costs and improving performance.

  For example, cloud providers like AWS and Google Cloud use containerization (e.g., Kubernetes) to dynamically allocate computing resources to workloads as needed.

Benefits of Server-Side Computing

Beyond user experience and vendor advantages, server-side computing enables several key technological improvements.

- **Faster Introduction of New Hardware**

  ✗ In a traditional computing model, upgrading hardware required end-users to buy new devices.

  ✓ In a datacenter-centric model, **hardware upgrades happen in the cloud**, where service providers can deploy: new processors, AI accelerators (e.g., TPUs, GPUs), more efficient storage solutions.

  For example, Google introduced TPUs (Tensor Processing Units) for machine learning, allowing AI workloads to run more efficiently in their datacenters without requiring users to upgrade their devices.

- **Cost Efficiency for Large-Scale Applications**

  ✗ Many applications require enormous computational resources that are impractical for client devices.

  ✓ Running these applications in datacenters allows for **better scaling** and **lower cost per user**.

  For example, Google Search processes billions of queries daily, requiring petabytes of storage and high-speed computation. Another example is training deep learning models (e.g., ChatGPT, image recognition, autonomous vehicles) is computationally expensive and only feasible in large datacenters.

2.1.1.3 Warehouse-Scale Computers

With the advent of server-side computing and large-scale Internet services, a new class of computing systems emerged: **Warehouse-Scale Computers (WSCs)**. These systems were designed to meet the unique requirements of large-scale applications, which often involve multiple interdependent programs interacting within a unified infrastructure. Characteristics of WSCs are:

- Unlike traditional data centers, **WSCs are** not just collections of servers, but **complete computing environments**.
- A single WSC might consist of **thousands of servers**, all working together as a **single, large computing unit**.
- **WSCs support complex, large-scale services** such as:
  - Search engines (Google Search, Bing)
  - Cloud-based email (Gmail, Outlook)
  - Online maps and navigation services (Google Maps, Waze)
  - Machine learning and artificial intelligence platforms (Google's TensorFlow, OpenAI's models)

  The scale at which these systems operate requires a fundamentally different architectural approach-one optimized for efficiency, scalability, and unified resource management.

Definition 1: Warehouse-Scale Computers (WSCs)

Warehouse-Scale Computers (WSCs) is a computing system designed to operate at an extreme scale, integrating massive software infrastructure, vast data repositories, and a unified hardware platform to function as a single, cohesive computing entity, typically owned and managed by a single organization.

⚖ Warehouse-Scale Computers vs. Traditional Data Centers

- **Traditional Data Centers: A More Fragmented Approach**. A traditional data center is a facility that houses a collection of servers and networking equipment, providing computing resources for various applications and organizations. These environments are characterized by:
  - A **diverse range of applications**, each running on **its own dedicated hardware infrastructure**.
  - **Applications that do not communicate** with one another or share resources.
  - A **multi-tenant model**, where hardware and software are used by multiple organizations or business units.

  This model works well for enterprises hosting multiple independent applications, such as corporate databases, enterprise software, and web hosting services.

- **Warehouse-Scale Computers: A Unified Approach**. In contrast, WSCs are designed to function as a **single, highly integrated computing unit**. Unlike traditional data centers, which support numerous small applications in isolation, WSCs are optimized for **massive, unified workloads** controlled by a **single organization**. Key characteristics of WSCs include:
  - **Homogeneous hardware and software**: The infrastructure is standardized, making it easier to manage and scale.
  - **Centralized resource management**: Instead of treating each server as an independent entity, WSCs manage computing resources dynamically across a large-scale cluster.
  - **Support for large-scale applications**: WSCs are built to run a **small number of massive applications**, such as search engines, AI models, or cloud services.

  By shifting towards WSCs, companies like Google, Amazon, and Microsoft have been able to achieve better efficiency, scalability, and cost-effectiveness, leading to the next evolution of computing infrastructure.

  For example, when we perform a Google Search, the request is processed across multiple servers simultaneously in a WSC, with different servers handling indexing, ranking, and query matching. The user only sees the final result, but behind the scenes, thousands of machines collaborate in milliseconds.

- **Ownership**
  - **Traditional DC**: Multiple organizations/tenants
  - **WSCs**: Single organization
- **Application Scale**
  - **Traditional DC**: Many small-to-medium apps
  - **WSCs**: Few massive applications
- **Hardware**
  - **Traditional DC**: Heterogeneous, diverse setups
  - **WSCs**: Homogeneous architecture
- **Resource Management**
  - **Traditional DC**: Isolated per application
  - **WSCs**: Centralized, dynamic allocation
- **Software Execution**
  - **Traditional DC**: Independent workloads
  - **WSCs**: Interdependent, large-scale services

From Data Centers to WSCs and Back

Initially, WSCs were designed exclusively for handling large-scale, internet-facing applications. However, as cloud computing evolved, the **lines between traditional data centers and warehouse-scale computing began to blur**. Modern cloud computing is a convergence of these two models: traditional Datacenter and WSC.

- Today's public cloud platforms (AWS, Google Cloud, Microsoft Azure) run many small applications, much like traditional data centers.
- These **applications rely on Virtual Machines (VMs) and Containers**, which efficiently distribute workloads across a warehouse-scale infrastructure.

Era	Computing Model	Example
Past	Traditional Data Centers	Enterprise IT infrastructure
Present	Warehouse-Scale Computers	Google Search, AI training
Future	Hybrid WSC-DC Model	Cloud computing (AWS)

2.1.1.4 Multiple Data Centers

The **architecture of modern computing infrastructure** is not limited to single data centers but rather **extends across multiple, geographically distributed facilities**.

Why use multiple datacenters?

A single datacenter can host and process workloads, but distributing computing infrastructure across multiple facilities offers several **advantages**:

- ✔ **Lower Latency for Users**. When **datacenters are closer to users**, request-response times are reduced. Services like Google Search, Netflix streaming, and cloud gaming benefit from regional datacenters.
- ✔ **Improved Throughput & Load Balancing**
  - **Spreading requests across multiple locations** prevents bottlenecks.
  - **Global services handle millions of requests per second**, which would overload a single facility.
- ✔ **Disaster Recovery & Fault Tolerance**
  - **If one datacenter fails** (due to power outages, natural disasters, or cyberattacks), **another can take over**.
  - **Redundancy ensures minimal downtime** for mission-critical applications.

The Hierarchical Organization of Multi-Datacenter Infrastructure

To effectively distribute workloads and ensure resilience, cloud providers divide their infrastructure into a **structured hierarchy**.

1. **Geographic Areas (GAs)**. The **highest level of organization**, defined by geopolitical boundaries or country-specific regulations. The primary factor is data residency laws, ensuring compliance with legal frameworks (e.g., GDPR in Europe).

   Example 1: Geographic Area

   For example, a GA might represent North America, Europe, or Asia-Pacific. A user in France may have their data processed within Europe to comply with EU data protection regulations.

2. **Computing Regions (CRs)**. A **finer granularity withing Geographic Areas**, representing **large-scale infrastructure within a geographic region**. Each GA contains at least two Computing Regions for redundancy. Key characteristics:
   - **Users see computing regions as isolated environments** where they can deploy workloads.
   - **Multiple DCs exist within a region but are not individually exposed to users**.
   - **Latency-defined perimeter**: Ensures that intra-region latency remains within 2ms for round-trip communication.
   - **Distance considerations**:
     - Datacenters within a region are hundreds of miles apart.
     - This protects against localized failures (e.g., natural disasters, power grid failures).

Example 2: Computing Region

AWS has "US-East-1" (Virginia) and "US-West-1" (California) as distinct computing regions.

3. **Availability Zones (AZs)**. Within a Computing Region, **datacenters are** further **divided into Availability Zones (AZs)**.
   - *What are Availability Zones?*
     - **Physically separate datacenters within a region**, each with redundant power, cooling, and networking.
     - **Fault isolation** ensures that failures in one AZ do not affect others.
     - **Designed for mission-critical applications** that require continuous availability.
   - *How they work?*
     - **Application-level synchronous replication** occurs across multiple AZs, **ensuring data consistency**.
     - A **minimum of three AZs** per region is recommended for quorum-based decision-making.
   - *Example Scenarios*
     - Customer 1 deploys services in a single AZ, risking downtime if the AZ fails.
     - Customer 2 uses multi-AZ deployment, ensuring failover in case of failure.
     - Customer 3 distributes workloads across multiple AZs, achieving full redundancy.

With the rise of low-latency applications, modern cloud providers go beyond central data centers by deploying **Edge Locations**. They are **smaller facilities located closer to the user**. They are **used to cache** frequently accessed data and accelerate content delivery. Commonly used in content delivery networks (CDNs) such as Cloudflare, AWS CloudFront, and Akamai.

Example 3: Edge Locations

- Streaming services (Netflix, YouTube): Caching popular videos at Edge Locations reduces buffering.
- Cloud gaming (NVIDIA GeForce Now, Xbox Cloud Gaming): Minimizing latency for real-time responsiveness.
- IoT applications: Autonomous cars and smart cities require ultra-fast data processing near the source.

2.1.1.5 Availability in WSCs and DCs

Availability is a critical factor in datacenter and warehouse-scale computing since these **infrastructures host mission-critical applications that must remain operational with minimal downtime**. Achieving high availability requires fault-tolerant architectures, redundancy, and proactive failure management.

Availability is measured as the **percentage of uptime over a given period** (usually a year). Another important topic is the **Service Level Agreement (SLA)**, which is a **formal contract between a service provider** (e.g., AWS, Google Cloud, Microsoft Azure) **and a customer** that defines the expected level of service reliability, availability, and performance. If the provider fails to meet the SLA, the customer may receive compensation (e.g., service credits or refunds).

Avail. (%)	Downtime/Year	Use Case
99.90%	$\approx$ 8.8 hours	Single-instance VMs with premium storage
99.95%	$\approx$ 4.4 hours	Availability Sets (AS) within a datacenter
99.99%	$\approx$ 1 hour	Availability Zones (AZs) for redundancy

Table 2: Availability Targets and Downtime Limits.

The **higher the availability**, the **less downtime is tolerated**, requiring advanced fault-tolerance strategies.

2.1.1.6 Architectural Overview of WSCs

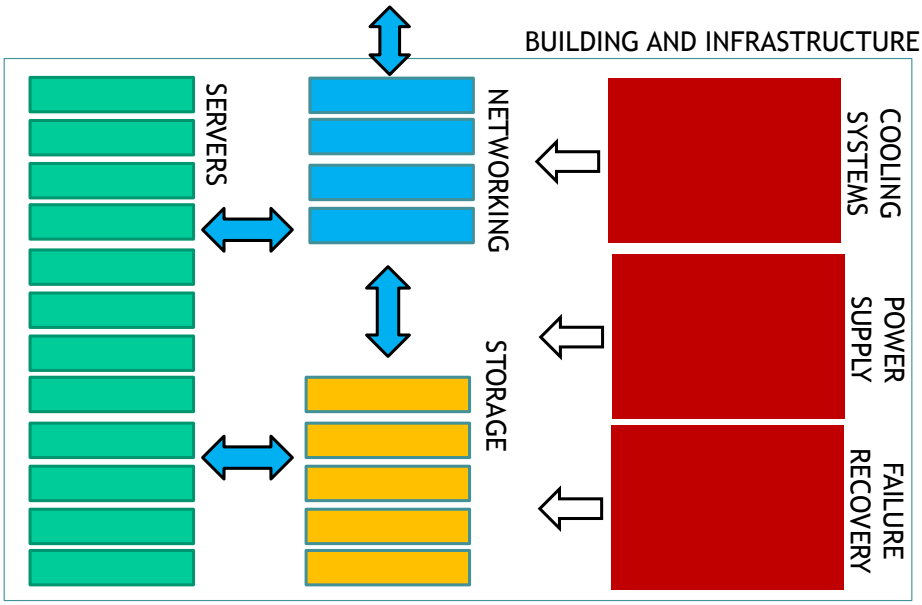


Figure 3: Architectural overview of Warehouse-Scale Computing.

The **architecture** of Warehouse-Scale Computers (WSCs) is **designed to handle massive-scale computing workloads** by integrating computation, storage, networking, power management, and infrastructure into a unified system. While specific implementations may vary, the overall architectural organization remains relatively stable, focusing on scalability, efficiency, and resilience.

A **WSC is more than just a collection of servers**; it is a tightly integrated computing environment that requires specialized hardware, networking, and infrastructure. Its architecture consists of several key components:

- **Servers (Compute Layer)**, section 2.2.1, page 25.
  - **Primary processing units** handling computation.
  - A wide variety of server configurations:
    - **General-purpose CPUs** (Intel, AMD) for standard workloads.
    - **Accelerators** (GPUs, TPUs, FPGAs) for AI, ML, and specialized computations.
    - **Local storage options** (HDD, SSD) for high-speed data access.
  - Servers are arranged in racks, interconnected through a high-speed network.

  Servers, which host application workloads and services, are the **basic building blocks** of WSCs.

- **Storage Systems**, section 2.2.2, page 34.
  - **Data storage is fundamental** to WSCs, as they host large-scale datasets and distributed applications.
  - Three primary types of storage (section 2.2.2.5, page 78):
    1. **Direct Attached Storage (DAS)**, local storage connected to individual servers.
    2. **Network Attached Storage (NAS)**, centralized storage accessible over a network.
    3. **Storage Area Networks (SAN)**, high-performance, block-level storage for large-scale applications.
  - Storage technologies include:
    1. **Hard Disk Drives (HDDs)** (section 2.2.2.2, page 39), used for cost-efficient, high-capacity storage.
    2. **Solid-State Drives (SSDs)** (section 2.2.2.3, page 45), faster, but more expensive, used for high-speed access.
    3. **Tape Storage**, rare but still used for archival data and backups.

  Storage is managed through distributed file systems and RAID configurations for redundancy and fault tolerance.

- **Networking Infrastructure**, section 2.2.3, page 81.
  - The Datacenter Network (DCN) **enables communication** between **servers**, **storage**, and **external users**.
  - Key network components:
    - **Switches & routers**, direct data traffic within and outside the WSC.
    - **Load balancers**, distribute incoming workloads efficiently across servers.
    - **Firewalls & security gateways**, protect against cyber threats.
    - **DNS/DHCP servers**, manage internal IP addressing and service discovery.
  - Low-latency, high-bandwidth connections are crucial for large-scale data processing.

  Networking is the backbone that ensures seamless data exchange between different parts of the WSC.

- **Cooling Systems & Environmental Control**, section 2.3.1, page 102.
  - WSCs **generate massive amounts of heat**, requiring **advanced cooling solutions**:
    - **Air cooling**, traditional method using fans and airflow management.
    - **Liquid cooling**, more efficient, uses liquid-cooled pipes to dissipate heat.
    - **Immersion cooling**, servers are submerged in non-conductive liquid for superior heat dissipation.

Cooling solutions are essential for optimizing energy efficiency and maintaining server performance.

- **Power Supply & Energy Management**, section 2.3.2, page 105.
  - **WSCs consume enormous amounts of power**, requiring dedicated **energy management systems**:
    - **Datacenters can consume up to 650 MW** (equivalent to 100,000 or more households).
    - **Redundant power supplies** ensure uptime in case of power failures.
    - **Renewable energy sources** (solar, wind) are increasingly used to reduce environmental impact.
  - Uninterruptible Power Supplies (UPS) & Backup Generators prevent downtime.

Power infrastructure is designed for high efficiency and reliability, ensuring continuous operation.

- **Failure Recovery & Fault Tolerance**
  - **High availability (99.99%)** is a core requirement for WSCs.
  - Key **failure management strategies**:
    - **Redundant hardware and storage replication** to prevent data loss.
    - **Automated failure detection and mitigation** through AI-driven monitoring.
    - **Multi-region and multi-zone deployments** for disaster recovery.

Resilience is built into every layer to minimize disruptions and downtime.

2.2 Node-level

2.2.1 Server (computation, HW accelerators)

A **Server** is a computing system designed to manage, process, and deliver data or services to other computers (clients) over a network. In the context of datacenters and Warehouse-Scale Computers (WSCs), servers are the **atomic units of computation**, the fundamental building blocks of the entire system architecture.

Though **conceptually similar to a desktop PC**, servers **differ** in critical ways:

- They are **significantly more powerful**, scalable, and modular.
- They are designed for **continuous operation**, high availability, and **dense physical packaging** within a rack structure.

Servers in modern datacenters must balance **performance**, **density**, **power efficiency**, and **maintainability**.

Server Types

Server form factors **define how** servers are **physically organized** and **deployed** in datacenter environments. There are three principal types:

1. **Tower Server**s (section 2.2.1.1, page 27). Resemble traditional desktop PCs. They are ideal for small-scale deployments or low-density use cases.
   - ✔ **Pros**: Easy to upgrade, good cooling, low cost.
   - ✖ **Cons**: Large footprint, not optimized for rack deployment.
2. **Rack Server**s (section 2.2.1.2, page 28). Designed to slide **into a rack** (standardized shelves) in units (U); e.g., 1U = 1.75 inches. It is the most common server format in datacenters.
   - ✔ **Pros**: High compute density, easy to cable and scale.
   - ✖ **Cons**: Requires dedicated infrastructure (rack, cooling, power).
3. **Blade Server**s (section 2.2.1.3, page 30). Extremely compact and multiple blades share power, cooling, and networking through a **blade enclosure**. It is excellent for environments where space and energy are at a premium.
   - ✔ **Pros**: Highest density and modularity, centralized management.
   - ✖ **Cons**: Higher initial cost, vendor lock-in, increased heat density.

Server Architecture

Servers are typically **integrated into a tray or blade enclosure**, which contains:

- **Motherboard**: The central PCB that **interconnects all components**.
- **Chipset**: **Manages data flow** between CPU, RAM, storage, and peripherals.
- **Expansion slots**: For GPUs, network cards, and other **accelerators**.

Servers in WSCs tend to **use homogeneous hardware/software platforms** to simplify large-scale orchestration and maintenance.

Server Architecture

The **Motherboard** acts as a **central nervous system for the server**, it hosts:

- **CPU sockets** (e.g., up to 2 for dual Xeon systems)
- **DIMM slots** for RAM
- **Storage connectors** (e.g., SATA, NVMe)
- **NIC slots** (Network Interface Cards)

This level of configurability allows tailoring servers for compute-heavy, memory-bound, or I/O-intensive applications.

2.2.1.1 Tower Server

A **Tower Server** is a type of server designed in a vertical, standalone chassis that closely resembles a **standard tower desktop computer**. Unlike blade or rack servers, which are designed for high-density environments, tower servers prioritize **simplicity and accessibility**, often at the cost of physical footprint.

- **Structure**: Independent, **vertical** case (not meant for rack mounting).
- **Deployment**: Common in small businesses, branch offices, or settings where only a few servers are needed.
- **Internal layout**: Lots of **space for expansion components** like disks or PCIe cards.

Advantages

- ✔ **Scalability & Ease of Upgrade**. **Easy** to open and **upgrade**, users can add storage, memory, or cards as needed.
- ✔ **Cost-Effective**. Usually the **cheapest server type**, suitable for budget-constrained environments.
- ✔ **Easy Cooling**. Due to **low component density**, natural airflow is often sufficient. Less need for specialized cooling systems.

Limitations

- ✖ **Space Consumption** Tower servers consume **significant physical space** and don't scale well in quantity.
- ✖ **Basic Performance** They usually **offer lower performance and redundancy** compared to enterprise-grade rack or blade servers.
- ✖ **Cable Management** Not ideal for structured environments, cables can become messy and hard to manage.

2.2.1.2 Rack Servers

A **Rack Server** is a server built specifically to be **mounted vertically in standardized racks**, which are metallic shelves designed to hold multiple servers and IT components. Rack servers are the **default choice** in medium to large-scale datacenters, balancing compute density, modularity, and serviceability.

Physical Standardization

- **Servers are stored in racks** which follow a global standard:
  - **1U (Rack Unit)** = 1.75 inches (44.45 mm) in height.
  - Servers may come in 1U, 2U, 4U, up to 10U formats depending on power and component density.
- Racks also house other components: networking switches, storage arrays, power distribution units (PDUs), and cooling units.

This **standardization allows for efficient vertical stacking** of servers, optimizing physical space and simplifying cabling.

Racks as More Than Just Shelves

A rack is not just a mechanical holder, it is **part of the power, networking, and management infrastructure of the datacenter**:

- **Power Infrastructure**:
  - Shared power distribution units.
  - Battery backup (UPS).
  - Power conversion units.
- **Networking**:
  - Top of Rack (ToR) switches connect all servers in the rack to the datacenter network fabric.
  - Simplifies cabling and reduces latency.
- **Cooling**: designed for front-to-back airflow, aligned with datacenter cooling strategy (e.g., cold aisle containment).
- **Dimensions** can vary, but the classic rack is 19 inches wide and up to 48 inches deep.

Advantages

- ✔ **Modularity**: Individual servers can be **hot-swapped**, upgraded, or replaced **without disrupting others**.
- ✔ **Failure Containment**: Easy to **isolate and service a failed node** without bringing down the system.
- ✔ **Cable Management**: Organized by rear/backplanes or Top-of-Rack (ToR) switches.
- ✔ **Cost-Efficient Scaling**: **Scales vertically** at relatively lower incremental cost compared to other formats.

Challenges

- ✖ **High Power Demand**: **Higher component density** requires more energy and advanced cooling systems.
- ✖ **Thermal Hotspots**: Tight stacking can cause **hot zones**, especially with accelerator-heavy nodes.
- ✖ **Maintenance Overhead**: Large racks with tens of servers can become **complex to manage** physically as systems scale.

2.2.1.3 Blade Servers

Blade Servers represent the **most advanced evolution** in server form factors. They are designed to **maximize space efficiency** and **centralized manageability**, making them ideal for **large-scale enterprise datacenters** and **high-performance computing environments**.

A blade server is essentially a **stripped-down**, **ultra-thin server board** (the "blade") that fits into a blade enclosure, a shared chassis providing:

- Power
- Networking
- Cooling
- Centralized management

The enclosure conforms to the same **rack unit standard** (U), allowing it to integrate seamlessly with existing rack infrastructure. We can think of a blade system as a server equivalent of a modular bookshelf, where each "book" is a full server, and the "bookshelf" provides shared power, ventilation, and data connectivity.

Advantages

- ✔ **Compactness & Density**: The **smallest physical form factor** among all servers, allowing high-density deployments within a minimal footprint.
- ✔ **Minimal Cabling**: The **shared backplane** removes the need for complex cabling; power and network connections are centralized.
- ✔ **Centralized Management**: Blade systems typically include **unified management interfaces** (e.g., iLO, iDRAC) to monitor and configure blades collectively.
- ✔ **Scalability & Reliability**: New **blades can be added** with minimal disruption; enclosures support **load balancing** and **failover mechanisms**.
- ✔ **Uniform Infrastructure**: Simplifies deployment with **shared cooling**, **network fabrics**, and **power redundancy**.

Disadvantages

- ✘ **High Initial Cost**: Blade enclosures and vendor-specific blades often demand **significant upfront investment**.
- ✘ **Vendor Lock-In**: Typically, only blades from the **same manufacturer** (e.g., HPE, Dell, Cisco) **are compatible** with a given enclosure.
- ✘ **Thermal Density**: The compact form causes **higher heat output per rack unit**, requiring advanced HVAC design and monitoring.
- ✘ **Limited Flexibility**: While modular, blade systems trade off flexibility for density, upgrades and replacements may be **constrained by the enclosure's architecture**.

2.2.1.4 Machine Learning

While Moore's Law historically predicted that transistor density would double every 18-24 months, the **growth in ML model complexity** has surpassed this pace. Since 2013, compute demand for AI training has doubled approximately **every 3.5 months**. This exponential curve far exceeds the capabilities of general-purpose CPUs, triggering a renaissance in specialized hardware.

What is Machine Learning?

At its core, **Machine Learning (ML)** refers to **computational methods** that enable systems to **learn from data** without being explicitly programmed. Rather than defining rules manually, ML allows a system to build a model from **patterns observed in examples**.

In supervised learning:

- A system learns a **target function** $y = f(x)$ that **maps inputs** (features) **to outputs** (labels).
- This is **done using a training dataset** $(x_1, y_1), \ldots, (x_N, y_N)$, and the model is later tested on unseen inputs.

Applications include: classification (e.g., cat vs. dog), regression (e.g., predicting flight delays), image recognition, speech synthesis, fraud detection, etc.

Artificial Neural Networks (ANNs)

Artificial Neural Networks (ANNs) are a **subset of ML models** inspired by the human brain. They consist of **layers of interconnected neurons**, including:

- **Input layer**: receives the data
- **Hidden layers**: transform data using weighted functions and nonlinear activations
- **Output layer**: produces the prediction

The key learning mechanisms are:

- **Backpropagation**: adjusts weights based on the error between prediction and actual target.
- **Gradient descent**: optimizes the model parameters iteratively.

Hardware Acceleration: Why ML Needs More Than CPUs

Modern ML, particularly **deep learning**, is computationally expensive. Training models like GPT or ResNet involves processing **billions of parameters** across massive datasets. To meet these demands, **Warehouse-Scale Computers (WSCs) integrate specialized accelerators such as**:

- **Graphics Processing Units (GPUs)**. GPUs are highly parallel processors originally designed for graphics rendering but are now extensively used for ML because they:
  - Execute the **same operation across many data elements in parallel** (SIMD).
  - Accelerate matrix operations central to deep learning.
  - Support ML frameworks via CUDA, OpenCL, OpenMP, SYCL, etc.

  GPUs are often housed in **PCIe-attached trays**, interconnected via NVLink or NVSwitch for ultra-fast data exchange.

  Distributed training across multiple GPUs requires **low-latency**, **high-bandwidth interconnects**. Performance can also be bottlenecked by slowest learner or network synchronization delays.

- **Tensor Processing Units (TPUs)**. Developed by Google, TPUs are **domain-specific architectures** designed **specifically for ML workloads**. TPU generations:
  - **TPUv1**: Inference-only, connected via PCIe.
  - **TPUv2**: Supports both training and inference; includes MXUs (matrix units) and high-bandwidth memory (HBM).
  - **TPUv3**: Liquid-cooled, supercomputing-class performance. Up to 100 PFLOPS per pod.
  - **TPUv4/TPUv5**:
    - v4 pod: 4096 devices
    - v5e: cost-efficient variant
    - v5p: high-performance variant scalable to 8000+ devices
    - Used in global data centers since 2023

  A **TPU Pod** aggregates **hundreds of TPU cores with shared memory** and custom high-speed networks for massive parallelism.

- **Field-Programmable Gate Arrays (FPGAs)**. FPGAs offer **customizable digital logic** that can be reprogrammed after manufacturing.
  - Flexible hardware, can be reconfigured for different algorithms.
  - Suitable for:
    - Network acceleration
    - Security tasks (e.g., encryption)
    - Data analytics
    - Specialized ML inference

  For example, Microsoft Azure integrates FPGAs for infrastructure efficiency, lowering carbon footprint and improving hardware reuse.

Feature	GPU	TPU	FPGA
Purpose	General-purpose ML compute	ML-specific acceleration (esp. DL)	Flexible, reconfigurable hardware
Programmability	CUDA, OpenCL, etc.	TensorFlow, PyTorch (high-level)	VHDL, Verilog (low-level, HDL)
Flexibility	High	Medium (optimized for tensors)	Very high (reprogrammable)
Efficiency	Good	Excellent (for tensor ops)	Excellent (for specific pipelines)
Use Case	Training + Inference	Training + Inference	Offloading, network, analytics

Table 3: Summary: GPU vs TPU vs FPGA.

	✅ Advantages	❌ Disadvantages
CPU	✔ Easy to be programmed and support any programming framework. ✔ Fast design space exploration and run your applications.	✖ Suited only for simple AI models that do not take long to train and for small models with small training set.
GPU	✔ Ideal for applications in which data need to be processed in parallel like the pixels of images or videos.	✖ Programmed in languages like CUDA and OpenCL and therefore provide limited flexibility compared to CPUs.
TPU	✔ Very fast at performing dense vector and matrix computations and are specialized on running very fast programming based on Tensorflow.	✖ For applications and models based on the TensorFlow. ✖ Lower flexibility compared to CPUs and GPUs.
FPGA	✔ Higher performance, lower cost and lower power consumption compared to other options like CPUs and GPU.	✖ Programmed using OpenCL and High-Level Synthesis (HLS). ✖ Limited flexibility compared to other platforms.

Table 4: Comparison of CPU, GPU, TPU and FPGA.

2.2.2 Storage (type, technology)

Data has significantly grown in the last few years due to sensors, industry 4.0, AI, etc. The growth favours the **centralized storage strategy** that is focused on the following:

- **Limiting redundant data**
- **Automatizing replication and backup**
- **Reducing management costs**

The *storage technologies* are many. One of the oldest but still used is the **Hard Disk Drive (HDD)**, a magnetic disk with mechanical interactions. However, the mechanical nature of HDDs imposes physical limits on access speed and reliability. In contrast, **Solid-State Drive (SSD)**, which lack moving parts and are built using NAND flash memory, offer significantly faster access times and better durability. The **Non-Volatile Memory express (NVMe)** also exists, which is the **latest industry standard** for running PCIe[2] SSDs.

In terms of cost per terabyte, NVMe drives are currently the most expensive (typically €100-200 for 1 TB), followed by SSDs (€70-100), while HDDs remain the most economical option (€40-60). This price-performance hierarchy makes hybrid storage architectures (HDD + SSD) increasingly appealing:

- A speed storage technology (**SSD or NVMe**) as **cache** and **several HDDs for storage**. It is a combination used by some servers: a small SSD with a large HDD to have a faster disk.
- Some HDD manufacturers produce Solid State Hybrid Disks (SSHD) that combine a small SDD with a large HDD in a single unit.

[2] **PCIe (peripheral component interconnect express)**. is an interface standard for connecting high-speed components

2.2.2.1 Files

The operating system views the disk as a **flat collection of independently addressable data blocks**. Each **block is assigned a unique LBA (Logical Block Address)**, enabling efficient data referencing and organization. To streamline access and reduce management overhead, the **OS typically groups these blocks into clusters**, larger units that serve as the minimum granularity for disk I/O operations.

Clusters typically range in size from a single disk sector (512 bytes or 4 KB) up to 128 sectors (64 KB), depending on the file system configuration. Each cluster may store either:

- **File data**. The actual contents of user files.
- **Metadata**. System-level information necessary to support the file system,:
  - File and directory names
  - Folder hierarchies and symbolic links
  - Timestamps (creation, modification, access)
  - Ownership and access control data
  - **Links to the LBA where the file content can be located on the disk**

Consequently, the **disk space is divided into different cluster types** based on their purpose:

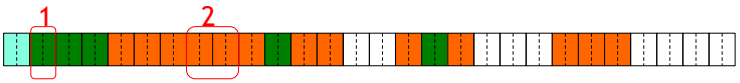
- Metadata:
  - Fixed-position metadata clusters, used to initialize and mount the file system
  - Variable-position metadata clusters, which manage directories and symbolic links
- File data clusters, containing the actual contents of files
- Unused clusters, which are free and available for future allocations

Figure 4: A cluster can be seen visually as an array. In this image, for example, we've shown three types of cluster: metadata fixed position (azure), metadata variable position (green), file data (orange), unused space (white).

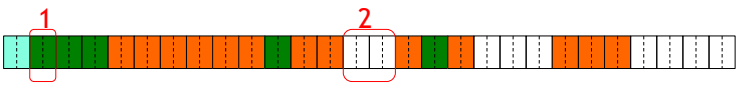
The following explanation introduces some basic operations on the files to see what happens inside the disks.

- **File Reading**
  1. **Access the Metadata**. Before the system knows **where** the file is stored on disk, it must find information about the file, called metadata. This metadata is stored in **variable-position metadata clusters**. These clusters are not always in the same place on disk, they move and grow as the system evolves.
  2. **Locate the File's Data Blocks (clusters)** and **Read the Actual Content**. Once the OS has the metadata, it now knows:
     - Which **Logical Block Addresses (LBAs)** contain the data.
     - **How many clusters need to be read** to get the full content.

     It uses this information to issue **read commands** to the disk controller. Finally, the **disk accesses the physical sectors or clusters** indicated by the LBAs and transfers that data into main memory (RAM).



- **File Writing**
  1. **Access Metadata to Find Free Space**. The operating system **first checks the file system's metadata** to **find a free area of disk space** where it can store our new data.
  2. Once free space is identified, the OS:
     - **Allocates** one or more clusters, depending on the file size
     - **Writes our data** into these clusters on the physical disk



Since the *file system can only access clusters*, the **actual space taken up by a file on a disk is always a multiple of the cluster size**. Given:

- s, the *file size*
- c, the *cluster size*

Then the **actual size on the disk** a can be calculated as:

$$a = \left\lceil \frac{s}{c} \right\rceil \times c \tag{1}$$

Where ceil rounds a number up to the nearest integer. It's also possible to calculate the **amount of disk space wasted by organising the file into clusters** (**wasted disk space** w):

$$w = a - s \tag{2}$$

A formal way to refer to wasted disk space is **internal fragmentation** of files.

Example 4: internal fragmentation

- File size: 27 byte
- Cluster size: 8 byte

The *actual size* on the disk is:

$$a = \left\lceil \frac{27}{8} \right\rceil \cdot 8 = \lceil 3.375 \rceil \cdot 8 = 4 \cdot 8 = 32 \text{ byte}$$

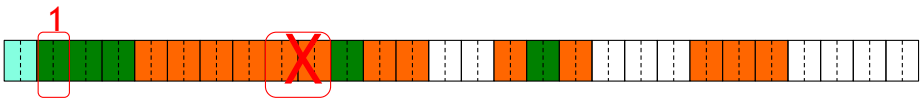
And the internal fragmentation w is:

$$w = 32 - 27 = 5 \text{ byte}$$

- **Deleting**

  1. The file system updates its metadata structures to:
     - **Remove the file name** from the directory
     - **Mark the clusters** where the file was stored as **free or available**
     - Optionally, update timestamps or record deletion events

     Importantly: **The data itself is not erased at this stage.** The actual bytes remain on disk until they are overwritten by another file.



- **External fragmentation**. It happens when there are **enough free clusters on the disk** to store a file, but **not all together (not contiguous)**. So, when the system tries to write a large file, it must **split it** into smaller parts and place them in **different, scattered locations** on the disk.

  Why does this happen? Over time, as files are created, deleted, resized, or moved, the disk becomes **less organized**. Clusters are freed in different spots, and the available space is no longer one big continuous area. So:

  - A new file cannot fit in one continuous chunk.
  - The OS stores it in **multiple non-adjacent clusters**.

  This is called **external fragmentation** because: the **fragmentation is not inside the file itself, but in the way its data is laid out externally across the disk**.

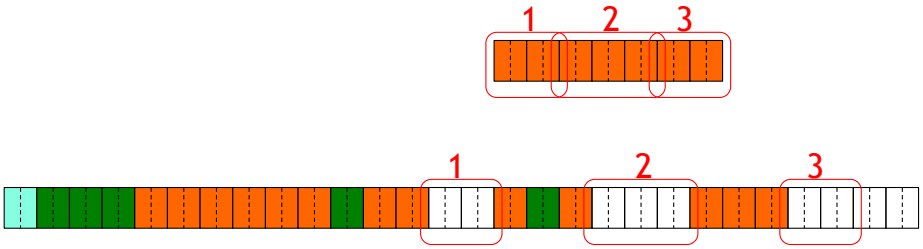


Figure 5: Each number (1, 2, 3) corresponds to a portion (chunk) of a file. A file that was meant to be stored as `[1][2][3]` (contiguously) has instead been broken into parts and stored in a scattered layout across different disk clusters. This situation represents **external fragmentation**, where the file system could not find a large enough continuous block of free space to store the file all together.

2.2.2.2 HDD

A **Hard Disk Drive (HDD)** is a **data storage device that uses rotating disks (platters) coated with magnetic material**.

Data is read randomly, meaning individual data blocks can be stored or retrieved in any order rather than sequentially.

An HDD consists of one or more rigid (*hard*) rotating disks (platters) with magnetic heads arranged on a moving actuator arm to read and write data to the surfaces.

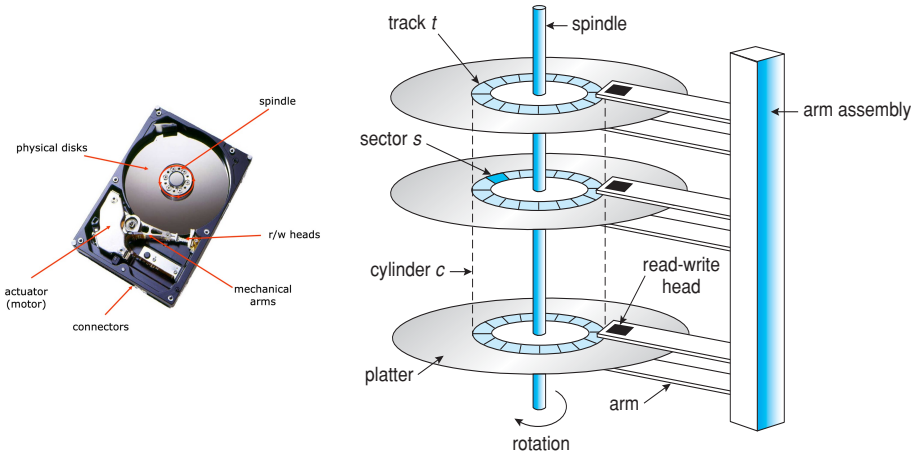


Figure 6: Hard Drive Disk anatomy.

Externally, hard drives expose a large number of **sectors** (blocks):

- Typically, 512 or 4096 bytes.
- Individual **sector writes** are **atomic**.
- Multiple sectors write it may be interrupted (**torn write**[3]).

The geometry of the drive:

- The sectors are arranged into **tracks**.
- A **cylinder** is a particular track on multiple platters.
- Tracks are arranged in concentric circles on **platters**.
- A disk may have multiple double-sided platters.

The **driver motor spins the platters at a constant rate**, measured in **Revolutions Per Minute (RPM)**.

[3]Torn writes happen when only part of a multi-sector update is written successfully to disk.

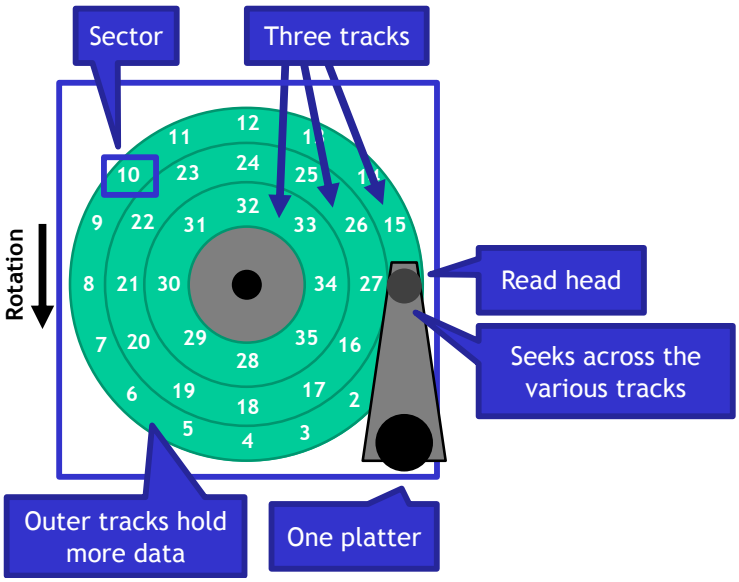


Figure 7: Example of HDD geometry.

The geometry of an HDD refers to the **physical layout** of data on the surface of a spinning disk. Figure 7 shows this:

- **One Platter**. The circular surface shown is a platter, which is a rigid, magnetic-coated disk where data is stored. Most HDDs have multiple platters stacked vertically, but here we focus on one for simplicity.
- **Tracks**. The platter is divided into concentric circles called tracks. Each track is a circular path that the read/write head can follow. In the figure 7, we see three tracks, each larger than the last, moving outward from the center.
- **Sectors**. Each track is divided into pie-like slices called sectors. A sector is the smallest physical unit that can be read or written on a disk (typically 512 bytes or 4 KB). The numbers (0 to 35) represent sector identifiers along the circular tracks. Note how the outer tracks contain more sectors because they have a larger circumference, so they can physically hold more data; this is known as Zone Bit Recording (ZBR), not mentioned in the course.
- **Read/Write head**. The read head is shown floating above the platter. It moves radially across the surface to switch from one track to another (this is called seek). Once on the correct track, the head waits for the desired sector to rotate beneath it (rotational latency), and then it reads/writes data.
- **Rotation and seek behavior**. The platter spins at high speed (e.g., 7200 RPM). As it rotates, the sectors pass under the stationary head. Data is accessed when the correct sector aligns with the head.

⚠ Types of Delay in disk Access

When a file is read from or written to a disk, especially an HDD, the total time taken is influenced by several delays. These are due to the mechanical and electronic processes involved in locating and transferring the data. Exists **four types of delay**:

- **Rotational Delay** (a.k.a. **Rotational Latency**) is the **time needed for the disk to rotate so that the desired sector aligns with the read/write head**. It depends on the RPM (Revolutions Per Minute) of the disk.
- **Seek Delay** is the **delay caused by the mechanical movement of the read/write head as it travels from one track to another** on a spinning disk. It is the dominant mechanical delay in many HDD operations, especially for random access patterns.

  How it works. Moving the head involves a sequence of physical phases:

  1. **Acceleration**: the actuator moves the head out of its resting position.
  2. **Coasting**: the head glides at a constant speed (if the distance is large).
  3. **Deceleration**: the actuator slows down to avoid overshooting.
  4. **Settling**: a short pause to stabilize the head at the desired track.

- **Transfer time** is the **time required to actually move the data**, once the head is correctly positioned over the desired sector. It's the final step in the I/O pipeline, where data is **read from or written to** the magnetic surface.

  What affects transfer time?

  1. **Rotational Speed (RPM)**, determines how quickly sectors pass under the head.
  2. **Data density**, more tightly packed data, more data per second read.
  3. **Size of the request**, reading more data takes more time.

  Even though it's much shorter than seek or rotation delays, **transfer time scales with data size**. For large sequential reads, transfer time becomes more relevant, especially when seek and rotation delays are minimized.

- **Controller Overhead** is the **non-mechanical delay introduced by the disk controller**, which is the hardware interface between the OS and the physical disk. It involves:

  – **Buffer management**: transferring data between disk and system memory (often using DMA).
  – **Interrupt processing**: informing the OS when the I/O operation is complete.
  – **Command translation**: converting OS-level I/O requests into device specific actions (e.g., SATA/NVMe commands).

– **Scheduling and queueing**: organize I/O operations for efficiency.

To see how these delays are calculated, we suggest you refer to the performance section 4.2, about HDD 4.2.1, page 153.

Cache

Hard Disk Drives (HDDs) are mechanical devices, and their performance is often limited by seek times and rotational latency. To partially overcome these limits, **many HDDs integrate a small** (8, 16, 32 MB) **amount of fast memory** (RAM) on the controller board, this is the **cache** or **track buffer**. The key functions of HDD cache are:

1. **Read Caching**. When the disk reads a block from the platter, it may also load **adjacent blocks** into the cache, expecting that they might be requested next (a technique known as **read-ahead**).

   Pros

   - ✔ If the next read request is for cached data, the disk can respond **instantly** from fast RAM.
   - ✔ This avoids mechanical movement and **cuts seek and rotational delays**.
   - ✔ Very effective for **sequential reads**.

2. **Write Caching**. There are two strategies here:

   (a) **Write-Back Cache** (Faster, Riskier). The HDD **reports the completion of the write** operation as soon as the data is in the cache, **before it is actually written to the platter**. Actual writing to disk happens later, in the background.

   Pros

   - ✔ Faster perceived performance
   - ✔ Useful in workloads with many small writes

   Cons

   - ✖ If power is lost before the data is flushed to disk, the **data is lost**. It is a **file system corruption risk**!
   - ✖ This is why it's considered **dangerous in critical systems** without battery backup or UPS.

   (b) **Write-Through Cache** (Safer, Slower). The HDD only reports the completion of the write operation **after the data has been fully written to disk**. Safer, but **slower**, since the OS must wait for the mechanical operation to finish.

3. Hybrid Cache: **Flash-Based Caching**. Some modern HDDs integrate **small flash memory** used for **persistent caching**:

   - Data stays even if the power goes out.
   - Combines the speed of SSD with the capacity of HDD.
   - Great for frequently accessed blocks (e.g., boot files, apps).

Disk Scheduling

While **caching** helps improve disk performance, it **doesn't eliminate the delays caused by seek and rotational latency**, especially during random access patterns. In systems with many I/O requests (like in a database or OS kernel), it's crucial to **choose the right order** to process requests to **minimize head movement**. This is where disk scheduling algorithms come in.

Instead of serving I/O requests in the order they arrive, the **disk controller (or OS)** can render them to improve efficiency. It is possible:

- Because every **disk request** includes the **target position** (i.e., the cylinder/track).
- So we can **estimate the cost (seek time)** of each request and choose the most efficient order.

Common disk scheduling algorithms are:

1. **First Come, First Serve (FCFC)** (the worst)

   How it works? Requests are handled in the **order they arrive**. No optimization, simple queue processing.

   Pros

   ✔ (Naive) Easy to implement

   Cons

   ✘ Can lead to **long seek distances** (i.e., inefficient head movement)

2. **Shortest Seek Time First (SSTF)** (great performance, but watch out for starvation)

   How it works? Always serve the request **closest to the current head position**.

   Pros

   ✔ Minimizes **total head movement**.

   ✔ Efficient in practice.

   Cons

   ✘ Can lead to **starvation**: distant requests may never be served if new close requests keep arriving.

3. **SCAN (Elevator Algorithm)** (good performance as SSTF, but fairer)

   How it works? The head **moves in one direction** (like an elevator), serving all requests in that direction. When it reaches the end, it **reverses direction**.

   Pros

   ✔ Good worst-case behavior.

   ✔ **No starvation**, every request will eventually be served.

Cons

- ✗ Requests at **edges** of the disk may have **longer wait times**.

4. **Circular SCAN (C-SCAN)** (fair, but less efficient than SCAN in some cases)

   How it works? Like SCAN, but head **only moves in one direction**. When it reaches the end, it **jumps back** to the beginning (like a circular elevator).

   Pros

   - ✓ More **uniform wait time** for all requests.

   Cons

   - ✗ Longer total movement (because of the jump).

5. **C-LOOK** (smart compromise between performance and fairness)

   How it works? Like C-SCAN, but instead of going to the physical end of the disk, th **head only goes as far as the last request in that direction**, then **jumps back to the smallest request**.

   Pros

   - ✓ **Saves movement** compared to full C-SCAN.
   - ✓ Still **avoids starvation**.

Algorithm	Fairness	Risk of Starvation	Strategy
FCFS	✓ Yes	✗ None	Serve in arrival order
SSTF	✗ No	⚠ Possible	Serve closest request
SCAN	✓ Yes	✗ None	Elevator (back & forth)
C-SCAN	✓ Yes	✗ None	One-direction sweep (circular)
C-LOOK	✓ Yes	✗ None	One-direction sweep (limited)

Table 5: Scheduling Algorithms.

2.2.2.3 SSD

A **Solid-State Drive (SSD)** is a **non-volatile storage device** that retains data without power. Unlike traditional Hard Disk Drives (HDDs), an SSD has **no mechanical parts**, *no spinning platters*, *no moving heads*. Internally, it's made of **transistors**, similar to those found in CPUs and RAM. It includes a **controller**, which manages read/write operations, wear leveling, garbage collection, and interface emulation. SSDs often adopt HDD-compatible interfaces (e.g., SATA, PCIe/NVMe) and form factors (2.5", M.2) for backward compatibility. Offers **higher performance** than HDDs, especially in **random access latency** and IOPS.

Flash Cell Technologies: Storing Bits in NAND

Modern SSDs use **NAND Flash Memory**, where data is stored in memory **cells**. Each cell can store one or more bits:

Cell Type	Bits per Cell	Characteristics
SLC	1	Fastest, most durable (up to 100k cycles), expensive
MLC	2	Slower than SLC, less durable, more dense
TLC	3	Used in most consumer SSDs; cheaper
QLC	4	High density, lower endurance
PLC	5	Experimental/extremely high density, low endurance

Increasing the number of bits per cell improves **capacity and cost-efficiency**, but **reduces endurance** and **performance**. Each cell type requires more precise voltage thresholds and incurs **more error correction and wear**.

Internal Organization

Solid-State Drives are built on NAND flash memory, which is **internally structured in a hierarchical manner** to optimize storage density and access efficiency.

- **Cell**: Basic storage unit (SLC, MLC, TLC, etc.) storing bits via trapped electrons.
- **Page**: The **smallest unit that can be read or written**. Typically 4-16 KB.

  Pages can be:

  - ✔ **Valid (In-use)**, contain active, readable data.
  - ✖ **Dirty (Invalid)**, hold obsolete or overwritten data.
  - – **Empty (Erased)**, ready to be programmed (written).
- **Block**: The **smallest unit that can be erased**. Usually contains 64-256 pages.

A block might have a capacity of 128-256 KB, composed of many smaller pages. Each **page** maps to a **Logical Block Address (LBA)** visible to the OS.

Deepening: Logical Block Address (LBA)

The **Logical Block Address (LBA)** is a key abstraction used by operating systems and file systems to **refer to storage locations** on a disk (HDD or SSD) in a simple, sequential manner.

LBA is an index number that uniquely identifies a fixed-size block of data (typically 512 bytes or 4 KB) on a storage device. It **hides the physical layout** of the drive (cylinders, heads, sectors) and presents a **flat address space** to the OS.

How does it work with SSDs? The OS issues read/write commands to specific LBA addresses. Internally, the SSD controller translates each LBA to a **physical location** in flash memory using a structured called the **Flash Translation Layer (FTL)**. This mapping is **dynamic** due to wear leveling, garbage collection, and bad block management.

In summary, the LBA is how the operating system sees the disk. Instead, the physical address is where the data actually resides inside the SSD.

Key operations are:

- `READ`: reads data from a **page**.
- `PROGRAM`: writes to a **page** (only if it's empty).
- `ERASE`: wipes an entire **block** (required before rewriting any page in that block).

The **important limitation** is that flash memory **cannot overwrite data in place**, we have to erase the whole block before reusing it. This lead to a **read-modify-erase-write cycle** even for simple updates.

Write Amplification Phenomenon

Write Amplification refers to the phenomenon where **the amount of actual data written to the NAND flash is greater than what the host system requested to write**.

Why does Write Amplification happen in SSDs? It stems from the **erase-before-write constraint** of NAND flash memory:

1. **(Flash) Pages can only be written once**, and to change them, we must erase the **entire block** (which may contain many pages).
2. If we modify even a **small piece of data**, the SSD:
   - Allocates a new page for the updated data.
   - Marks the old page as invalid (dirty).
   - Eventually, **copies all valid pages** from a block, **erases the block**, and **rewrites it** (this is called **garbage collection**).

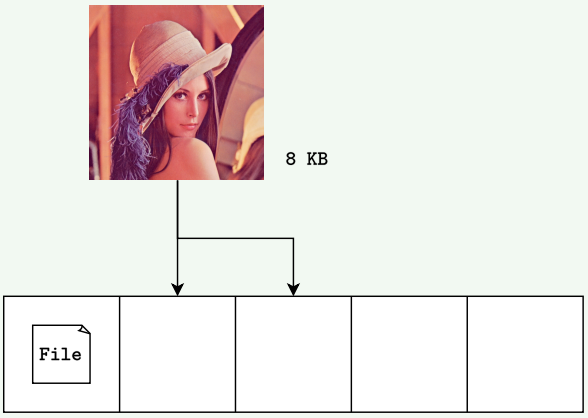
So a 4KB write might cause **hundreds of KBs** to be written internally!

Example 5: Write Amplification

Given a hypothetical SSD:

- Page Size: 4 KB
- Block Size: 5 Pages
- Drive Size: 1 Block
- Read Speed: 2 KB/s
- Write Speed: 1 KB/s

1. Write a 4 KB `.txt` file.
   - One page used (page size ÷ file dimension);
   - Time: 4 seconds (write speed × file dimension, 1 KB/s × 4 KB).
2. Write an 8 KB `.png`
   - Takes 2 pages;
   - There are 3 pages used in total;
   - Time: 8 seconds.



3. Delete the 4 KB `.txt` file
   - The first page is now **invalid** (dirty), but still **physically used**;
   - The SSD cannot reuse it until the whole block is erased.
4. Write a 12 KB image
   - OS sees "3 free pages", but **only 2 are truly empty**;
   - Can't fit 12 KB in 2 empty pages.

What happens internally?

(a) SSD reads the 8 KB of still-valid pages from NAND into a temporary cache.
This step takes 4 seconds (size to read ÷ read speed).

(b) Marks the 1st page (old `.txt`, step 1) as discarded.

(c) Places the new 12 KB into the cache.

(d) **Erases the entire 20 KB block** (this is mandatory before rewriting any page)

(e) **Rewrites**:
- 8 KB old data
- 12 KB new data

This step takes 20 seconds to write to the NAND.

The OS thought it was just a 12 KB write and expected only 12 seconds. But the SSD actually wrote 20 KB and read 8 KB (24 seconds total). This is a case of write amplification caused by limited empty pages, erase-before-write constraint, and the need to preserve valid data.

A direct mapping between Logical and Physical pages is not feasible inside the SSD. Therefore, each SSD has an FTL component that makes the SSD *look like an HDD*.

Flash Translation Layer (FTL)

The **Flash Translation Layer (FTL)** is the hidden "brain" of an SSD, it **makes NAND flash usable in the same way as a traditional hard disk**, despite its very different constraints.

Translates Logical Block Addresses (LBA) (see page 46) from the operating system into actual **physical locations** in the NAND flash. Also, it makes the SSD behave like an HDD to the OS (abstracts away erase-before-write and wear issues).

Why We Need Translation? Direct LBA to physical mapping isn't feasible because:

- ✗ Flash memory **can't overwrite in-place** (must erase first)
- ✗ Pages are grouped into blocks and must be programmed **in order**.
- ✗ Flash memory blocks **wear out over time**, requiring wear balancing.

The FTL responsibilities are:

1. **Address Mapping**. Maintains a **mapping table**, logical to physical page. Supports dynamic remapping when data is updated (e.g., a page becomes dirty and is relocated).

2. **Log-Structured Writes**. Uses log-structured techniques: **writes go to the next available page** in an erased block. It ensures writes are sequential (within a block), reducing write amplification and improving performance.
3. **Garbage Collection**. Identifies blocks full of **invalid/dirty pages**. Reads out remaining valid pages, erases the block, and rewrites valid data + new data elsewhere. Necessary to free up space for future writes.
4. **Wear Leveling**. Flash cells can only endure a limited number of erases. FTL spreads out writes across all blocks to **prevent early death** of heavily used regions.

Example 6: Log-Structured FTL

Setup:

- **Page size**: 4 KB
- **Block size**: 4 pages (total 16 KB per block)
- **Initial condition**: all pages are marked **INVALID**
- **Action**: perform a series of logical writes

Assume that a page size is 4 KB and a block consists of four pages. The write list is (`Write(pageNumber, value)`):

- `Write(100, a1)` → write value `a1` to logical page `100`
- `Write(101, a2)` → write value `a2` to logical page `101`
- `Write(2000, b1)` → write value `b1` to logical page `2000`
- `Write(2001, b2)` → write value `b2` to logical page `2001`
- `Write(100, c1)` → overwrite logical page `100` with value `c1`
- `Write(101, c2)` → overwrite logical page `101` with value `c2`

The steps are:

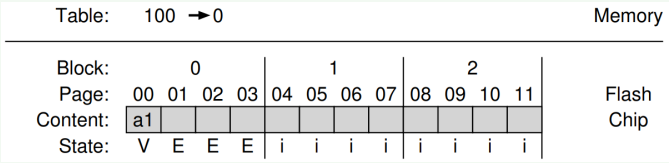
1. The initial state is with all pages marked as `INVALID(i)`:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	i	i	i	i	i	i	i	i	i	i	i	i

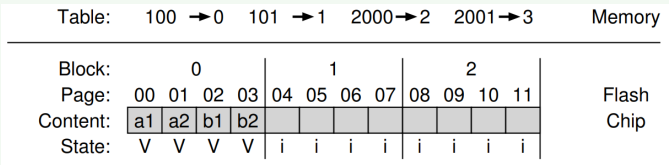
2. Erase block zero:

Block:	0				1				2			
Page:	00	01	02	03	04	05	06	07	08	09	10	11
Content:												
State:	E	E	E	E	i	i	i	i	i	i	i	i

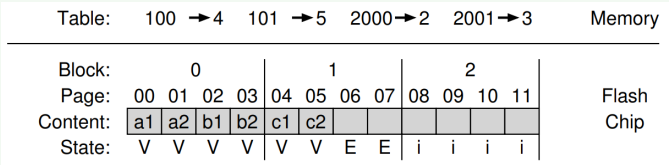
3. Program pages in order and update mapping information (first `Write(100, a1)`):



4. After performing four writes (`Write(100, a1)`, `Write(101, a2)`, `Write(2000, b1)`, `Write(2001, b2)`):



5. After updating 100 and 101:



Why Garbage Collection Exists

In SSDs, data cannot be updated in place, when a page is modified:

- The **old version becomes obsolete** (marked *invalid*).
- The **new version** is written to a **fresh page**.
- Over time, blocks fill with **invalid pages**, this is called *garbage*.

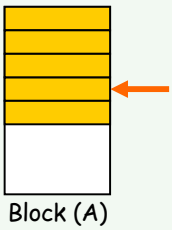
Garbage Collection (GC) is the process of **reclaiming space** by **erasing blocks that contain invalid pages**. It works like this:

1. **Identify a block** that has invalid (dirty) pages.
2. **Copy** any **still-valid pages** into a new block.
3. **Erase** the **original block** completely (erasing works only at block granularity).
4. **Update** the **mapping table** to reflect new page locations.

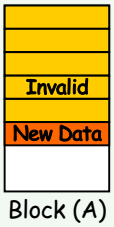
Example 7: how garbage collection works

The steps are:

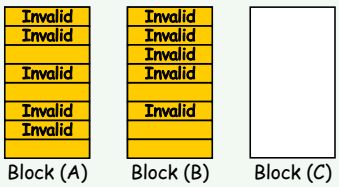
1. Update request for existing data:



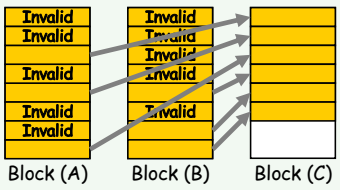
2. Find a free page, and save the new data:



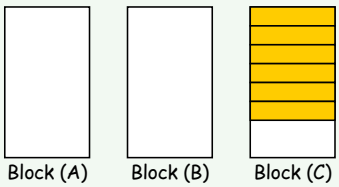
3. This scenario may continue until there are not enough free blocks:



4. Collect valid pages into a free block:



5. Update the map table and erase invalid (obsolete) blocks:



Three Major Problems of SSD Architecture

1. **Garbage Collection Is Expensive**. Garbage collection (GC) in SSDs is **unavoidable** due to the nature of NAND flash memory: flash **cannot overwrite pages**, only erase entire blocks.

   It is so expensive because it requires reading valid data (read cost), rewriting that data (write cost), and erasing entire blocks. If blocks are **partially valid**, even small updates can trigger **large internal data movement**. This leads to write amplification, effective write speed and wear on NAND cells.

   The **more valid data** in a block, the **higher the cost** of GC.

   - **Ideal case**: reclaim blocks with only invalid pages (no migration needed).
   - **Realistic case**: migrate live data, high overhead.

   Mitigation Techniques

   - **Overprovisioning**. Reserve extra flash capacity that the user can't see. More space, then delayed GC.
   - **Background GC**. Perform GC during **idle periods** to avoid disrupting foreground writes.
   - **Write buffering**. Use DRAM/SRAM to accumulate small writes and reduce fragmentation.
   - **Hot/Cold Separation**. Store frequently updated data (hot) separately from rarely updated data (cold).

2. **The Ambiguity of Delete**. In traditional file systems (especially on HDDs), **deleting a file** simply:

   - Removes the file's metadata (e.g., from the file system's directory tree).
   - Does **not erase or inform the disk that the data blocks are invalid**.

   This behavior is fine for HDDs, which can overwrite sectors anytime. But for SSDs, this **creates a serious mismatch**.

   SSDs rely on Garbage Collection (GC) to free space. GC assumes that **only invalid pages** can be discarded. However:

   - The SSD sees no distinction between "old" and "deleted" data unless explicitly told.
   - So even **deleted files look valid** to the SSD.
   - When GC runs, it **copies all pages** it believes to be valid, including junk!

   This causes SSDs to waste time and NAND endurance **preserving deleted data**.

   How to Fix it: TRIM / UNMAP. Modern OSs and SSD interfaces support special commands: TRIM (SATA) and UNMAP (SCSI/NVMe).

These commands allow the **OS to explicitly tell the SSD**: these Logical Block Addresses (LBAs) are no longer valid, feel free to erase them.

3. **Mapping Table Size and FTL Scalability**. In an SSD, the **Flash Translation Layer (FTL)** maps: **Logical Block Addresses (LBAs) from the OS to physical flash pages**. This mapping is essential because:

   - Flash memory can't overwrite in place
   - Pages must be written sequentially
   - Blocks must be erased before reuse

   So the SSD keeps an **internal mapping table** to know **where every logical page actually resides**.

   The mapping table **grows** proportionally with: **drive capacity**, and **granularity of mapping**.

   FTL Strategies to Cope

   - **Block-Level Mapping**. The FTL maps **each logical block number (LBN)** to a **physical block number (PBN)**. All pages within that block are **assumed to map 1:1**.

     Pros

     * **Very small mapping table**
     * Efficient in **sequential-write** workloads (e.g. logging, archiving).

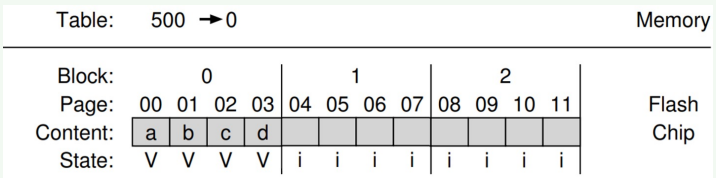
     Cons

     * **Terrible for random writes**: to update just 1 page, the entire block must be read, modified, erased, rewritten.
     * Results in very **high write amplification**.

Example 8: Block Mapping

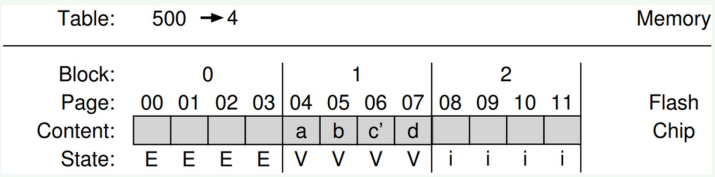
The first four writes:

- `Write(2000, a)`
- `Write(2001, b)`
- `Write(2002, c)`
- `Write(2003, d)`



And finally the last one:

- `Write(2002, c')`



- **Hybrid FTL**. Combine the best of Block-Level Mapping for most pages and use **Page-Level Mapping** (FTL maps each logical page number to a physical page number) for small updates. Often implemented using a **log block buffer**.

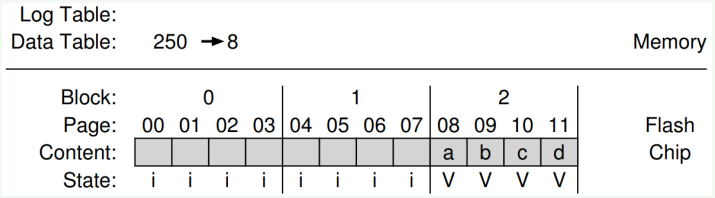
  ✔ **Pros**
  - Lower memory usage than pure page-level
  - Lower write amplification than pure block-level

  ✖ **Cons**
  - More complex logic (copy-back handling, log merging)
  - Still suffers from some write amplification during **log cleaning**

Example 9: Hybrid Mapping

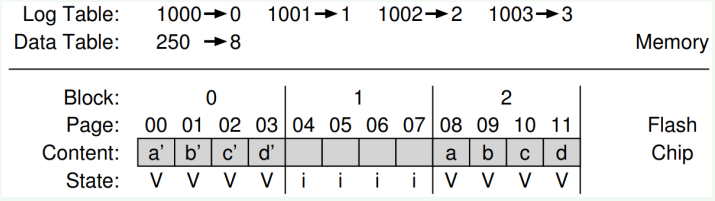
Let's suppose the following sequence:

- `Write(1000, a)`
- `Write(1001, b)`
- `Write(1002, c)`
- `Write(1003, d)`

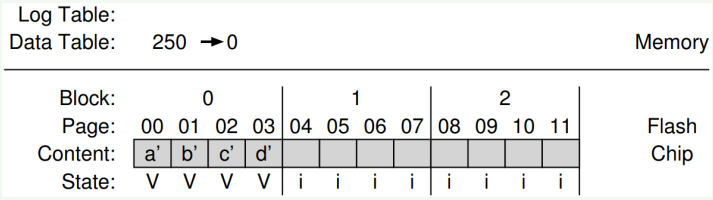


Let's update some pages:

- `Write(1000, a')`
- `Write(1001, b')`
- `Write(1002, c')`
- FTL updates only the page mapping information



When needed, FTL can perform MERGE operations:



- **Page mapping plus caching** (a.k.a. **Demand-based FTL (DFTL)**). Keep **only a portion** of the page-level mapping table in DRAM (a cache). **Store** the full mapping table in **flash itself**. On cache miss, **read the mapping** from flash into RAM (like a page table swap-in).

  ✔ **Pros**

  * **Scales to very large SSDs** with **low DRAM footprint**
  * Maintains page-level flexibility without storing entire table in RAM

  ✖ **Cons**

  * Mapping lookup incurs **read latency** on cache misses
  * More complex metadata management (must protect flash-stored tables)

✔ The importance of Wear Leveling

Flash memory has a **limited number of erase/write (EW) cycles**:

- Each block can only sustain ≈3,000 to 100,000 cycles (depending on type: SLC, MLC, TLC, QLC).
- If some blocks are used more than others (write **skew**), they **wear out faster**.

Without intervention, the SSD's lifespan would be determined by its **most heavily used block**. Ensure that **all blocks** in the SSD **wear out evenly** to: maximize **overall drive lifetime**, avoid **early failure** due to localized hot spots.

How Wear Leveling works? The **Flash Translation Layer (FTL)**:

1. Monitors **erase/write cycles** per block.
2. Identifies **cold blocks** (rarely updated, long-lived data).
3. Periodically:
   - **Reads** valid data from cold blocks
   - **Moves** it to fresher blocks
   - **Erases** and **reuses** the original blocks

   Even clod blocks are "rotated" to ensure wear balance.

There are three types of Wear Leveling:

- **Dynamic**: Spread writes among blocks not currently holding static data
- **Static**: Occasionally move long-lived (cold) data to give its block a rest
- **Hybrid**: Combine both (used in most SSDs)

Wear Leveling Disadvantages

- **Increases Write Amplification**: moving cold data incurs extra writes.
- **Consumes bandwidth**: periodic copying reduces performance.
- **Complexity**: requires tracking per-block wear counts and scheduling background operations.

However, to **partially fix** this, a simple policy to apply is that each flash block has an **Erase/Write Cycle Counter** and maintains the value of:

$$|\text{Max (EW cycle)} - \text{Min (EW cycle)}| < \varepsilon \tag{3}$$

Where ε is a system-defined Wear Leveling threshold.

HDD vs SSD

UBER and TBW are two metrics help quantify how **reliable and durable a storage device is over time and under heavy use**.

- **Unrecoverable Bit Error Ratio (UBER)**. The probability that a bit cannot be recovered correctly by the device, even after error correction.

$$\text{UBER} = \frac{\text{Number of unrecoverable bit errors}}{\text{Total bits read}} \tag{4}$$

  A **lower** UBER means **better data reliability**.

- **Endurance rating: Terabytes Written (TBW)**. The total amount of data we can write to the SSD over its warrantied lifetime before cells are expected to wear out.

$$\text{TBW} = \text{Endurance rating of the SSD (from manufacturer)} \tag{5}$$

  For example, a 250 GB SSD with TBW = 70 TB, we can write: $70 \div 365 = 190$ GB/day.

2.2.2.4 RAID

How is RAID born?

Before 1980, **JBOD (Just a Bunch Of Disks)** was a common setup before the advent of disk arrays and RAID. JBOD simply **connects multiple physical disks to a system**, but each disk operates **independently**. There's **no striping** and **no redundancy**. Each disk has its **own mount point**, and the operating system or user decides where to store data. If one disk fails, **only the data on that disk is lost**, no protection or recovery is provided. **Performance bottlenecks** also occurred as CPUs became faster than disk I/O.

Researchers at UC Berkeley (notably Patterson, Gibson and Katz) proposed using **multiple cheaper disks** together: treat **many small, inexpensive disks** as **one large logical disk**. These were called **Disk Arrays**. The main idea was to achieve higher performance and more memory through parallelism.

- Disk Arrays appear as a **single logical high-performance disk**.
- Use **data striping** and **parallel access**, but these **weren't formalized or optimized** in the early implementation. This is because the core idea was to spread data across multiple disks to improve throughput (multiple disks serving different I/Os) and latency (multiple I/Os in parallel reduce queue wait time). But the **parallelism was often implicit**, not exposed to the software or the user. Some arrays allowed **concurrent I/O by accident**, not design. **I/O scheduling** was hardware-specific, **not standardized**.

Key improvements:

- ✔ **Parallel Access**: Multiple disks could serve I/O requests simultaneously.
- ✔ **Performance Boost**: Higher bandwidth and lower latency via concurrent disk activity.
- ✔ **Scalability**: Easy to increase capacity by adding disks.
- ✔ **Lower Cost per GB**: Used **multiple small, cheap disks** instead of one large expensive disk.
- ✔ **Modular Architecture**: Allowed easier maintenance, replacement, and upgrades.

Limitations:

- ✖ **No Standard Striping**: Striping (if present) was non-standard and not guaranteed.
- ✖ **No Redundancy**: If a disk failed, **data was lost**, arrays were vulnerable.
- ✖ **No RAID Levels**: No formal trade-offs between performance/reliability (e.g. RAID 0-6).
- ✖ **No Fault Tolerance Mechanism**: No automatic reconstruction, hot spares, or parity.

✖ **Limited Write Optimization**: RAID improved performance (esp. for writes) using smart parity layouts.

In 1988, the same researchers published the seminal paper: "A Case for Redundant Arrays of Inexpensive Disks (RAID)" [8]. They made two key contributions:

- They **formally introduced RAID**: not just disk arrays, but structured ones that also **handle reliability** via redundancy.
- They **defined RAID levels** (RAID 0 to RAID 5 in the original paper), each with different trade-offs between: *performance*, *reliability* and *storage efficiency*.

RAID was important because it **solved the reliability problem introduced by disk arrays**. Disk Arrays solved performance and capacity, but RAID solved reliability too, combining **Data Striping** (performance) and **Redundancy** (reliability).

Definition 2: RAID

RAID (Redundant Array of Independent Disks) is a **storage technology** that **combines multiple physical disks into a single logical unit** to improve performance, reliability, or both, by using data striping, redundancy, or both.

RAID is based on two core techniques: **Data Striping** (for performance, splits data across disks), and **Redundancy** (for reliability, adds fault tolerance via mirroring or parity).

Exists different RAID levels. Each level defines a strategy for:

- *How data is striped*
- *How redundancy is added*
- *What failure modes are tolerated*
- *How performance and capacity are affected*

Data Striping

Data Striping is the technique of **splitting data into small chunks** (called **stripe units**) and distributing them **across multiple disks in a round-robin manner**. Used primarily to **improve performance** by enabling parallel disk access. It does **not add redundancy** by itself.

- **Stripe Unit**: The amount of **data written to a single disk** before moving to the next one. Can be in **bits**, **bytes**, **blocks** or **KBs**. Small stripe unit, better parallelism for large files. Large stripe unit, fewer disk operations for small reads.
- **Stripe Width**: The **number of disks** over which the data is stripped. If we have 4 disks participating in stripping, stripe width is 4. **Determines how many disks work in parallel** for a given I/O.
- How are **Multiple Independent I/O Requests** processed? **Small**, random I/O **requests** from **different applications or users**. RAID can **process them in parallel** on different disks. As result: **lower disk queue lengths and faster response times**.
- How is a **Single Multiple-Block I/O Request** processed? A **large read or write operation** on a big file (e.g., a video or a database query). Striping lets **multiple disks work together** to serve this one big request. As result, **much faster data transfer** compared to using one disk.

Redundancy

Redundancy is the method of **adding extra information to protect data from disk failures**. If a disk fails, the system can reconstruct the missing data using the redundant information. Redundancy is *essential* because:

- Disk arrays use many disk, and **more disk**, **higher risk of failure**.
- RAID introduces redundancy to **tolerate and recover from failures**.

There are two types of redundancy:

1. **Data Duplication (Mirroring)**. Every block of data is **copied exactly** to another disk. Used in RAID 1 and RAID 10.
   - ✔ Can survive **complete disk failures**.
   - ✔ Fast reads (can load-balance between mirrors).
   - ✖ **Wastes 50%** of storage (one disk stores only a copy).
2. **Parity-Based Redundancy**. Adds a **calculated parity block** (usually XOR) that allows **rebuilding lost data**. Used in RAID 5 and RAID 6.
   - ✔ More **storage-efficient** than mirroring.
   - ✔ Can tolerate 1 (RAID 5) or 2 (RAID 6) disk failures.
   - ✖ **Writes are slower** (need to read/update parity on each write).

Why is it called "Redundant Array"? Because RAID **stores more than just our data**, it stores the **redundant data necessary to protect it**.

RAID levels

A **RAID level** defines a specific way to organize **striping** and **redundancy** across multiple disks. Each level balances **three key trade-offs**:

- **Performance**: *How fast are reads/writes?*
- **Fault Tolerance**: *Can it survive disk failure?*
- **Storage Efficiency**: *How much usable space is left?*

RAID	Striping	Redundancy	Tolerates Failure?	Main Benefit
0	✔	✖ None	✖ No	Max speed, no protection
1	✖	✔ Mirroring	✔ 1 disk	Reliability via copies
5	✔	✔ Parity (rotated)	✔ 1 disk	Balance of perf + space
6	✔	✔ Dual Parity	✔ 2 disks	Extra protection

Table 6: Standard RAID Levels: RAID 4 is not shown because it is rarely used; RAID 5 has the same concept but eliminates RAID 4's limitations. RAID 2 and 3 are also not shown because they are not covered in this course.

Each RAID level is a **design pattern** that answer: "*How can I spread and protect data across multiple disks to meet my performance, capacity, and reliability goals?*".

Table 7: RAID - Table of Contents.

RAID 0 - Striping Without Redundancy

RAID 0 splits (stripes) data across multiple disks to increase performance. It **does not provide redundancy**: if one disk fails, **all data is lost**.

Structure

- **Data Striping**: ✔ Yes (block-level)
- **Redundancy**: ✖ None
- **Minimum disks**: 2
- **Fault Tolerance**: ✖ 0 disks, a single failure is fatal
- **Storage Efficiency**: ✔ 100% (all capacity is usable)

When is it used?

High performance environments where **data loss is acceptable**, such as temporary data storage, video editing, gaming, scratch disks or caches.

Performance

Operation	Result
Reads	✔ Much faster (parallel access)
Writes	✔ Much faster (split writes)
Failure	✖ One disk dies, **all lost**

✔ Advantages

- **Lower cost** because it does not employ redundancy (no error-correcting codes are computed and stored).
- **Best read/write performance** (it does not need to update redundant data and is parallelized).

Disadvantages

- **Zero fault tolerance**.
- High risk: the failure rate is equal to the sum of all disk failure probabilities.

How does it work?

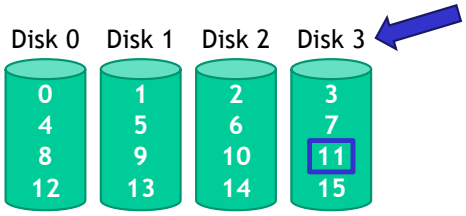
The main idea of RAID 0 is to use several physical disks as **a single logical disk** by **striping** data across them (splitting data into small blocks and writing them in **round-robin fashion** across all N disks). This approach improves performance:

- **Sequential access** is distributed across all disks, increased transfer rate.
- **Random access** naturally spreads I/O load across disks, lower latency and better throughput.

The system uses the following formulas to **determine which disk and offset holds a given logical block**:

$$\begin{aligned} \texttt{Disk index} &= \texttt{logical\_block\_number} \,\%\, \texttt{number\_of\_disks} \\ \texttt{Block Offset} &= \frac{\texttt{logical\_block\_number}}{\texttt{number\_of\_disks}} \end{aligned} \tag{6}$$

For example, the command `read block 11` in a 4-disk array gives disk 3 as the index and block 2 as the offset on that disk.



This formula **allows the RAID controller to locate any block without metadata**, just with arithmetic.

The choice of chunk size is critical because it affects performance at this level. The **Chunk Size** (also called **Stripe Unit** size) is the amount of **data written to each disk before moving to the next one**.

The impact of chunk size is evident:

- **Small** chunks → better **parallelism** (each access touches more disks).
- **Large** chunks → lower **seek overhead** (each file uses fewer disks).

In practice, typical RAID arrays use 64 KB chunks as a **balanced** (tradeoff) default.

Measuring Performance

- **Measuring *Sequential* Transfer Rate**. For **large sequential I/O**, the transfer time includes:

  1. **Seek time** (page 153)
  2. **Rotational latency**
  3. **Actual Data Transfer Time**:

$$S = \frac{\texttt{transfer\_size}}{\texttt{time\_to\_access}} = \frac{\texttt{data\_size}}{\texttt{disk\_rate}} \tag{7}$$

  For example, suppose there is 7 ms of *seek time*, 3 ms of *rotational latency*, and 10 MB of *data to transfer* at a *disk rate* of 50 MB/s. The total time is given by the seek time, rotational latency, and the time taken to transfer the data:

$$\text{Time} = 7\,\text{ms} + 3\,\text{ms} + (10\,\text{MB} \div 50\,\text{MB/s}) = 210\,\text{ms}$$

  Thus, the sequential throughput is:

$$S = 10\,\text{MB} \div 0.21\,\text{s} = 47.62\,\text{MB/s}$$

  In RAID 0 with N disks, the **Total Sequential Throughput** is:

$$\text{Total Sequential Throughput} = N \times S \tag{8}$$

  S is the sequential throughput for a single disk in the system.

- **Measuring *Random* Transfer Rate**. For **small random accesses**, latency dominates over transfer. Using the same example as before, but changing the file size to 10 KB, the time is as follows:

$$\text{Time} = 7\,\text{ms} + 3\,\text{ms} + (10\,\text{KB} \div 50\,\text{MB/s}) = 0.98\,\text{MB/s}$$

  Resulting random throughput:

$$R = 10\,\text{KB} \div 10.2\,\text{ms} = 0.98\,\text{MB/s}$$

  In RAID 0 with N disks, the **Total Random Throughput** is:

$$\text{Total Random Throughput} = N \times R \tag{9}$$

  R is the random throughput for a single disk in the system.

Feature	Description
Capacity	N, all disk space is usable (no redundancy)
Reliability	0, MTTDL (Mean Time To Data Loss) = MTTF (Mean Time To Failure)
Performance	It provides full parallelization for both random and sequential throughput

RAID 1 - Mirroring

RAID 0 offers performance, but **no protection** against failures. **RAID 1 solves this** by maintaining **two identical copies of all data** (mirroring).

- ✔ **High reliability**: if one disk fails, the second copy is used seamlessly.
- ✔ **Fast reads**: can read from either disk (or balance across them).
- ⚠ The **write speed** is slightly slower than that of a single disk because the **data must be written to both disks**.
- ✖ **High cost**: only **50% of total disk space** is usable.

In theory, RAID 1 can use **more than 2 disks** to mirror data, but rarely done due to **very high costs** (only $1 \div N$ capacity used), and too much overhead for typical storage needs. In other words, more than 2-way mirroring is possible but **not practical**.

Why combined RAID levels at all?

RAID 1 is great, but it has its **limits**. It mirrors data, meaning one disk has the data and the other disk has a copy of it. Therefore, we always need pairs of disks. With two disks, RAID 1 is simple and reliable. But what if we want to use more than two disks and want both high reliability and high performance? We want to **combine the reliability that RAID 1 provides with the performance and scalability that RAID 0 provides**. In other words, how do we structure the disks as their number increases to get the best of both RAID 0 and RAID 1? The answer lies in the RAID 10 and RAID 01 techniques.

The goal of combining RAID levels is to achieve both reliability and performance. The combination is **generally written as** RAID $x + y$ (sometimes called RAID xy), meaning we **first** apply RAID x to small groups of disks and **then** apply RAID y on top of those groups.

We divide $n \times m$ total disks into m groups, each containing n disks.

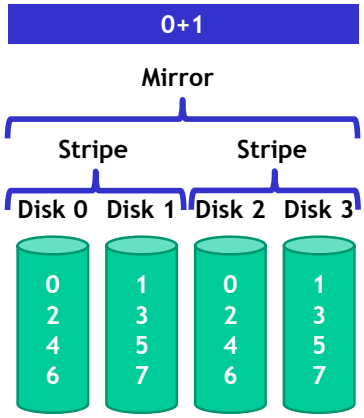
1. Apply RAID x to each group.
2. Apply RAID y to the m groups, treating each group as a "logical disk".

There are two possible combinations at this level:

- **RAID 0 + 1** is a **striped array of mirrored sets**. It combines the performance of RAID 0 and the fault tolerance of RAID 1, but with **some limitations**.

  The structure consists of a **minimum of 4 disks**:

  1. **Start with striping** (RAID 0): split data across 2 or more disks for performance.
  2. **Then mirror the whole striped group** (RAID 1).



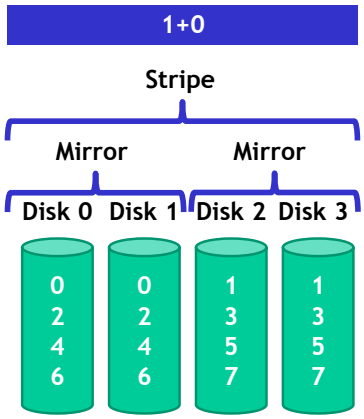
⚠ **RAID 0 + 1: Failure Behavior**. If **any one disk fails**, the system still works using the mirrored group. But if **a disk fails in each stripe**, **all data is lost**. After the **first failure**, the array **degrades into pure RAID 0**, losing redundancy.

Aspect	Value
Usable space	50% (half is mirror)
Read speed	✔ High (parallel reads)
Write speed	⚠ Slower (must write both copies)
Reliability	✔ Tolerates 1 disk failure (in same group)

- **RAID 1 + 0** combines **mirroring (RAID 1)** and **striping (RAID 0)**. First, data is **mirrored**, then it is **striped** across mirrored pairs. It offers **high performance (like RAID 0)**, **high fault tolerance (like RAID 1)**.

  The structure consists of a **minimum of 4 disks**:

  1. **Create mirrored pairs** (RAID 1)
  2. **Stripe data across pairs** (RAID 0)



RAID 1 + 0: Failure Tolerance

✔ Can survive **multiple disk failures**, as long as **only one per mirrored pair**.

✖ If both disks in a mirrored pair fail, data loss.

RAID 10 is **more fault-tolerance than RAID 01**, especially in cases of multiple failures. Because each mirrored pair in RAID 10 is self-contained:

- If one disk in a pair fails, only **that pair** needs to be rebuilt (not the entire array).
- The rest of the array **remains fully operational** and can keep serving requests.

In RAID 01:

- A single disk failure disables **half of the array**.
- A second failure in the other half = total data loss.

Aspect	Value
Usable space	50% of total (because of mirroring)
Read speed	✔ Very fast (parallel reads from mirrors)
Write speed	✔ Faster than RAID 5/6 (no parity calc.)
Reliability	✔ High (survives multiple failures safely)

Widely used in databases, transactional systems, and virtualized environments where both performance and fault tolerance are critical.

In conclusion, RAID 10 is created by **first mirroring disks** and **then striping** across the mirrored pairs. This configuration provides **faster read and write** speeds and **greater fault tolerance** than RAID 01, making it ideal for high-load systems.

Measuring Performance

- **Capacity**. Total usable space is $N \div 2$, since each block is stored twice, only **50% of disk space** is usable.
- **Reliability**. Can **survive 1 disk failure** guaranteed. In bets-case scenarios, can survive **up to** $N \div 2$ **disk failures**, if no two are mirrors of each other. However, if **both disks in a mirrored pair fail**, data is lost.
- ***Sequential* Performance**

  Sequential Write. Writes must go to both copies, then halved throughput:

$$\text{Sequential Write} = \left(\frac{N}{2}\right) \times S \tag{10}$$

Sequential Read. In theory, reads could be optimized, but usually reads are directed to **only one disk per mirror**. So, **half of the disks are idle** during reads:

$$\text{Sequential Read} = \left(\frac{N}{2}\right) \times S \tag{11}$$

- ***Random* Access Performance**

  Random Read. **Best case** for RAID 1. Reads can be **load-balanced** across all N disks. System can choose which mirror is less busy, then fully parallel.

  $$\text{Random Read} = N \times R \tag{12}$$

  Random Write. Writes must go to both disks in each pair. Only $N \div 2$ mirrors, then only $N \div 2$ writes happen in parallel.

  $$\text{Random Write} = \left(\frac{N}{2}\right) \times R \tag{13}$$

RAID 1 provides **excellent reliability** and **great random read performance**. Its main limitations are only 50% of storage is usable, and sequential throughput is limited to half the potential.

The Consistent Update Problem

In RAID 1, every write must be done **twice** (once per mirror). This raises a **consistency issue**: *what if the system crashes or loses power after writing to one disk, but before writing to the other?*

- One mirror has the **updated data**
- The other mirror has **state (old) data**
- The two copies are now **out of sync**

This violates the **atomicity** principle (a write must be **all or nothing**, either all mirrors are updated or none).

How it's handled. Many RAID controllers implement a **Write-Ahead Log (WAL)**:

- A special **non-volatile memory area** (battery-backed cache).
- Writes are first saved into the log **before** writing to disks.
- If power fails, the controller can use the log to:
  - Complete the write
  - Or roll it back (to restore consistency)

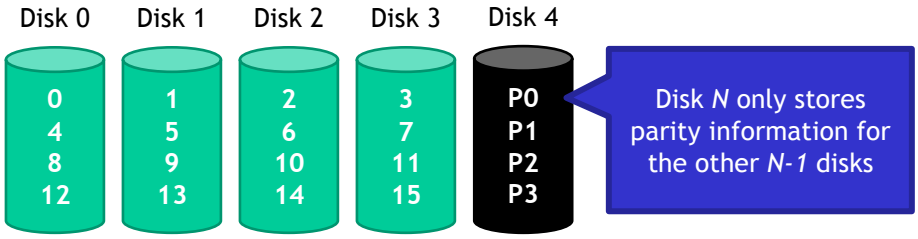
This ensures that mirrored copies stay **synchronized**, even after failure.

RAID 4 - Parity Drive

RAID 4 uses **block-level striping** (like RAID 0), but adds a **dedicated parity disk** to provide **fault tolerance**.

Parity Calculation

Parity is computed using the **XOR (exclusive OR)** operation. For example, 4 blocks per stripe:



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	0 ^ 0 ^ 1 ^ 1 = 0
0	1	0	0	0 ^ 1 ^ 0 ^ 0 = 1
1	1	1	1	1 ^ 1 ^ 1 ^ 1 = 0
0	1	1	1	0 ^ 1 ^ 1 ^ 1 = 1

Parity calculated using XOR

Figure 8: RAID 4 - *How does it work?*

It is used XOR because it has a special property: if we know all but one value, we can **reconstruct the missing one**. For example, if Disk 2 is lost, it can be reconstructed using an XOR operation with Disks 0, 1, and 3 and the Parity unit.

Pros and Cons

- Fault tolerance: can survive **1 disk failure**.
- More **space-efficient** than RAID 1 (only 1 disk used for redundancy).
- **Parity disk is a bottleneck**. Every write must update parity, then a single disk becomes overloaded.
- Write performance is poor compared to RAID 0 or RAID 10.

How does it work?

- **Updating Parity on Write in RAID 4**. When we **modify a block** in RAID 4, we must also update the **parity block** in that stripe. There are two ways to do it:

  1. **Additive Parity**. Recalculate parity using all the blocks in the stripe. Steps:
     (a) **Read all other data blocks** in the stripe.
     (b) XOR them together to get the **new parity**.

     Simple and correct, but **expensive** because must **read all blocks** to recompute.
  2. **Subtractive Parity** (Efficient: modify incrementally). Update the parity by using a **mathematical trick**:

     $$\text{New Parity} = \text{Old Parity} \oplus \text{Old Data} \oplus \text{New Data} \tag{14}$$

     This works thanks to the **reversible** XOR property:

     $$A \oplus A = 0, \quad A \oplus 0 = A$$

     Very efficient because only need to read the block being written (**old value**) and the **old parity block**.

- **Reads in RAID 4**. Reads in RAID 4 are **fast and efficient**, especially compared to writes. Data is **striped across all data disks**, and the **parity disk is not involved** in normal reads. There are two possible read scenarios:

  1. **Sequential Read**. Large, continuous read (e.g., loading big file). Data is read in **parallel** from all data disks. It is **fast**, each disk reads part of the file simultaneously.
  2. **Random Read**. Small, scattered reads (e.g., database lookups). Still efficient because:
     – Each disk can handle **its own read** in parallel.
     – Parity disk is **not touched** during a normal read.

  As we saw at the beginning, if a data **disk fails**, the missing block is **reconstructed using XOR**:

  $$D_{\text{missing}} = P \oplus D_1 \oplus D_2 \oplus D_3 \oplus \dots$$

  Read recovery is still possible, it is just slower.

- **Serial Writes in RAID 4**. Serial writes mean writing **a sequence of blocks** that all belong to the **same stripe**, typically during large, contiguous writes (e.g., streaming, backups).

  In RAID 4, data is **striped** across all data disks. The **parity for each stripe** is stored in a **dedicated parity disk**.

  During serial writes, each data disk receives **one block** from the stripe. The **parity disk** is updated **once per stripe**. Since all writes hit the same stripe, the **parity disk becomes a bottleneck**.

Operation	Description
Data writes	✔ Parallel across data disks
Parity update	✘ Centralized
Bottleneck	Parity disk handles **every stripe's write**
Comparison	Same structure as read, then but **parity slows it**

Even though data writes are parallel, all serial writes in the same stripe **converge on the parity disk**, making it the **main point of contention**.

- **Random Writes in RAID 4**. A **random write** updates **one block** somewhere on disk, not the entire stripe. This is common in databases, file systems, and OS workloads.

  The writing process for each block is as follows:

  1. **Read** the **old data block** and the **old parity block**.
  2. **Compute** the parity block:

     $$P_{\text{new}} = D_{\text{old}} \oplus D_{\text{new}} \oplus P_{\text{old}}$$

     This efficiently updates parity using subtraction (XOR).
  3. **Write** the **new data block** and the **updated parity block**

⚠ The main problem here is the **bottleneck** caused by the parity disk. Even though the data block may change on **any disk**, the **parity block is always on the same disk**. Each write operation must **read from** and **write to** that disk. This creates **serialization**, only one parity update can occur at a time. Even with multiple writes to different data disks, they all **wait for the parity disk**.

Operation	Behavior
Data writes	✔ Fast (per disk)
Parity update	✘ Always goes to the **same disk**
Bottleneck	⚠ Parity disk handles **every write**
Overall	✖ **Terrible performance** on random writes

Measuring Performance

- **Capacity**: $N - 1$. One disk is used exclusively for parity, so it is **not usable for data**.
- **Reliability**: can tolerate **1 disk failure** (any one of the data or parity disks). But if a disk fails, the system can still serve data using parity reconstruction. However, write and read performance becomes **massively degraded** (especially if parity disk is affected).
- **Read/Write Performance**

Operation	Performance	Notes
Sequential Read	$(N-1) \times S$	✔ Full parallelism across all data disks
Sequential Write	$(N-1) \times S$	✔ Parallel as long as writes hit different stripes
Random Read	$(N-1) \times R$	✔ Reads avoid parity disk, then high parallel performance
Random Write	$R \div 2$	✖ Terrible, every write must access parity disk twice

> Parity disk becomes a **point of contention for every write**. Causes **serialization of random writes**, reducing overall throughput drastically.

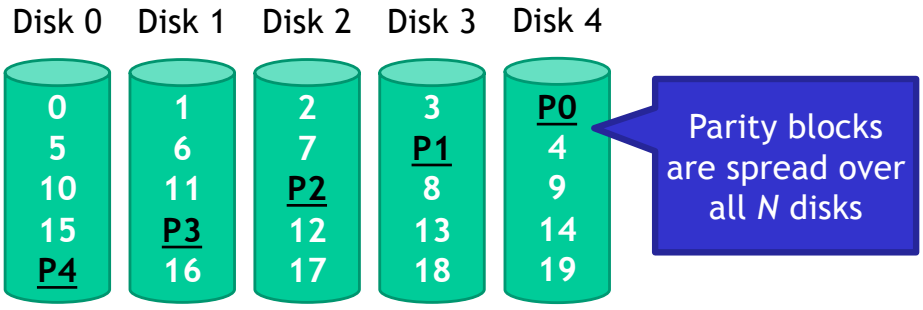
In conclusion, RAID 4 provides ample capacity and quick sequential input/output (I/O) thanks to block-level striping. It uses one dedicated parity disk for fault recovery. However, its **main weakness is its poor random write performance**, as all writes target the same parity disk. This is why RAID 4 is rarely used in practice and **led to the development of RAID 5, which solves this issue by distributing parity**.

RAID 5 - Rotating Parity

RAID 5 extends RAID 4 by using **distributed (rotating) parity**. This removes the **bottleneck of the single parity disk**.

How does it work?

In RAID 4, parity is always stored on a **dedicated parity disk**, causes a bottleneck. In RAID 5, parity is **spread (rotated)** across **all disks**, **one stripe at a time**.



Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	0	1	1	0 ^ 0 ^ 1 ^ 1 = 0
1	0	0	0 ^ 1 ^ 0 ^ 0 = 1	0
1	1	1 ^ 1 ^ 1 ^ 1 = 0	1	1
1	0 ^ 1 ^ 1 ^ 1 = 1	0	1	1

Parity position rotates from one disk to the next. So **each disk holds both data and some parity**; there's no dedicated parity disk. However, parity is still computed using the XOR operation, and the operand depends on the disks that calculate it. For example, Disk 4 will calculate the XOR using Disks 0, 1, 2, and 3. Disk 3 will use disks 0, 1, 2, and 4, and so on for the others.

What happens during a Random Write?

Like in RAID 4, RAID 5 must **preserve parity** whenever data changes. A **random write** modifies only one block in a stripe, so we use an efficient **subtractive parity update**.

1. **Read**
   (a) Read the *old data block*
   (b) Read the *old parity block*
2. **Compute**. Use XOR subtraction:

$$P_{\text{new}} = D_{\text{old}} \oplus D_{\text{new}} \oplus P_{\text{old}}$$

3. **Write**
   (a) Write the *new data block*
   (b) Write the *updated parity block*

In total, there are **4 disk operations**: 2 reads (old data and old parity) and 2 writes (new data and new parity).

RAID 5 improvement over RAID 4

In **RAID 4**, all parity writes go the **same disk**, and this creates a **bottleneck**. **RAID 5 solves** this slowdown by **rotating the parity**. Parity updates are spread across all disks, which distributes the load evenly. This allows **random writes to occur in parallel**, eliminating serialization due to a single disk. This is obviously much better than RAID 4.

Analysis

- **Capacity**: $N - 1$ disks. Same as RAID 4, one disk's worth of capacity is used for **distributed parity**.
- **Reliability**: tolerates 1 **disk failure**. However, during partial outage the degraded mode phenomena is highlighted, because all missing data must be reconstructed on-the-fly using parity, which causes massive performance degradation.
- **Read/Write Performance**

Operation	Performance	Notes
Sequential Read	$(N-1) \times S$	= Same as RAID 4, striping across data disks
Sequential Write	$(N-1) \times S$	✔ Fully parallel if stripes span across disks
Random Read	$N \times R$	Better than RAID 4, all disks can be read in parallel
Random Write	$\frac{N}{4} \times R$	Better than RAID 4, writes are distributed. However, each write requires an additional 2 reads and 2 writes for parity updates.

For each random write, we have 2 reads and 2 writes with N disks and parallelism. The **throughput** is:

$$\text{Throughput} \approx \frac{N}{4} \times R \tag{15}$$

In conclusion, RAID 5 provides a good **trade-off between capacity**, **reliability**, and **performance**. It avoids RAID 4's parity bottleneck by using **rotating parity**, enabling better parallelism for **random writes**. However, in case of a disk failure, **performance degrades** significantly due to **on-the-fly parity reconstruction**.

RAID 6 - Dual Parity

RAID 6 is an extension of RAID 5 that adds **a second independent parity block** per stripe. This allows it to **tolerate two simultaneous disk failures**, a major reliability upgrade over RAID 5.

How does it work?

It can tolerate **up to 2 concurrent disk failures**. Uses **two different types of parity**:

- **P parity**: is the same as in RAID 5, a "*simple*" XOR (exclusive-or) of the data blocks in a stripe.
- **Q parity**: is more complex. It is generated using **Reed-Solomon** coding, a mathematical technique over **Galois Fields (GF)**, which enables the correction of multiple simultaneous failures.

Each parity serves a different purpose:

- **P (XOR parity)** handles **one disk failure**.
- **Q (Reed-Solomon parity)** can handle a second failure that isn't linearly dependent on the first one.

This combination allows RAID 6 to recover from **two simultaneous disk failures**. In simple terms:

- If we lose 1 disk, we can **recover using XOR** (as in RAID 5).
- If we lose 2 disks, the system **solves a set of linear equations** (using P and Q) **to recover the missing data**.

This requires the two parities to be **independent**, so their information isn't redundant.

The **disk configuration requires** $N + 2$ **disks**. N are data disks, and 2 are distributed parity blocks (P and Q) per stripe. The **minimum configuration** requires at least **4 data disks**, for a total of **6 disks minimum**.

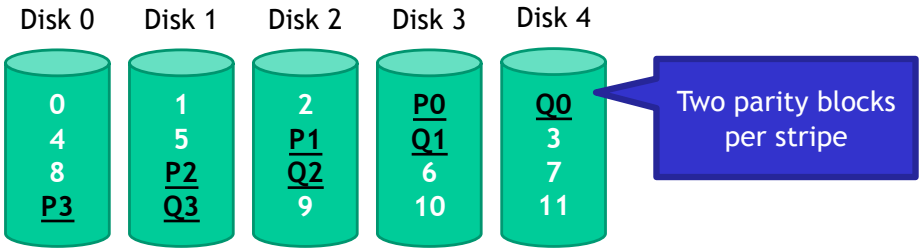
Write Performance

Write performance is **slower than with RAID 5**. Each write requires updating both parity blocks, necessitating at least **6 disk accesses**.

1. Read *old data*, read *parity block P*, and read *parity block Q*
2. Write *new data*, write *new parity block P*, and write *new parity block Q*

High overhead makes RAID 6 **slower on random writes**, but **safer**.

In summary, **RAID 6** adds a **second**, **independent parity block** ($P + Q$) per stripe, enabling **recovery from two simultaneous disk failures**. RAID 6 uses Reed-Solomon coding for the Q parity and distributes both parity blocks across all disks. Although **RAID 6 is more fault-tolerant than RAID 5**, it suffers from higher write overhead, requiring at least 6 I/O operations per write.



Comparison and characteristics of RAID levels

The following table shows eight fundamental properties of the RAID levels.

- N: number of drives
- R: random access speed
- S: sequential access speed
- D: latency to access a single disk

	RAID 0	RAID 1	RAID 4	RAID 5
Capacity	N	$N \div 2$	$N - 1$	$N - 1$
Reliability	0	$1 \vee : N \div 2$	1	1
Sequential Read	$N \times S$	$(N \div 2) \times S$	$(N - 1) \times S$	$(N - 1) \times S$
Sequential Write	$N \times S$	$(N \div 2) \times S$	$(N - 1) \times S$	$(N - 1) \times S$
Random Read	$N \times R$	$N \times R$	$(N - 1) \times R$	$N \times R$
Random Write	$N \times R$	$(N \div 2) \times R$	$R \div 2$	$(N \div 4) \times R$
Read	D	D	D	D
Write	D	D	$2 \times D$	$2 \times D$

Table 8: Comparison of RAID levels.

Where the throughput is:

- Sequential Read
- Sequential Write
- Random Read
- Random Write

And the latency is:

- Read
- Write

RAID	Capacity	Reliability	Read Perf.	Write Perf.	Rebuild Performance
0	100%	N/A	✔ Very good	✔ Very good	✔ Good
1	50%	✔ Excellent	✔ Very good	✔ Good	✔ Good
5	$\frac{(n-1)}{n}$	✔ Good	✔ Good	✘ Fair	✘ Poor
6	$\frac{(n-2)}{n}$	✔ Excellent	✔ Very good	✘ Poor	✘ Poor
$1+0$	50%	✔ Excellent	✔ Very good	✔ Good	✔ Good

Table 9: Characteristics of RAID levels.

- Best **Performance and Capacity**: `RAID 0`.
- **Maximum Fault Tolerance**: `RAID 1` or `RAID 6`, with RAID $1+0$ **often preferred** over $0+1$.
- **Balanced Solution** (Space, Performance, Recoverability): `RAID 5`.

We report the RAID level comparison here. This is for utility reasons. It is on page 60.

RAID	Striping	Redundancy	Tolerates Failure?	Main Benefit
0	✔	✘ None	✘ No	Max speed, no protection
1	✘	✔ Mirroring	✔ 1 disk	Reliability via copies
5	✔	✔ Parity (rotated)	✔ 1 disk	Balance of perf + space
6	✔	✔ Dual Parity	✔ 2 disks	Extra protection

Final Considerations and Adopted Techniques:

- **Hot Spare Disk**. A *hot spare* is a **physical disk** already installed and powered in the system but **not actively used** to store data or parity. It's **just waiting**.

  This is important because when **a disk** in a RAID array **fails**, **rebuilding** the array, i.e., reconstructing lost data using parity, can **take hours or days**, depending on the disk size and load. However, with a hot spare:

  - ✔ The controller **immediately** starts rebuilding the lost data onto the spare.
  - ✔ No need for human intervention to replace the disk.
  - ✔ It reduces downtime and risk of further data loss during the rebuild period.

  The mechanics of a Formula 1 team are a **great analogy**. The pit crew remains inactive for most of the race. However, the moment the car enters the pit lane, the crew acts immediately and precisely, replacing tires or making adjustments in a matter of seconds. Similarly, the hot spare disk is not part of the active array during normal operations. However, as soon as a failure occurs, the hot spare disk begins the rebuild process, helping to quickly restore full redundancy.

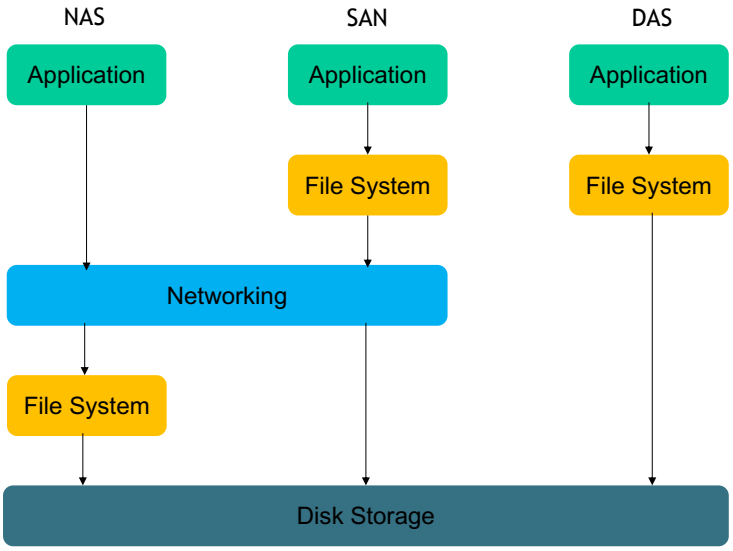
- **Hardware vs Software RAID**
  - **Hardware RAID**. Implemented via a **dedicated RAID controller card** or built into the motherboard.
    - **Pros**
      - ✔ **High performance**: RAID logic is offloaded from the CPU.
      - ✔ Usually includes a battery-backend **cache to improve write speed** and **protect against power loss**.
    - **Cons**
      - ✘ **Portability issue**: if the controller fails and we move the disks to another machine with a different controller, the array might not be recognized.
      - ✘ **Expensive**.
  - **Software RAID**. Managed by the **operating system** (e.g., Linux `mdadm`, Windows Storage Spaces).
    - **Pros**
      - ✔ **Cheap** and **flexible**.
      - ✔ **Easy to migrate**: disks can often be moved to a different machine and reassembled.
    - **Cons**
      - ✘ **Slower performance** because the host CPU handles parity and I/O operations.
      - ✘ **Less reliable** under high load or system crashes.
      - ✘ Vulnerable to the **consistent update problem** (page 67).

2.2.2.5 DAS, NAS and SAN

As the last argument, we introduce **three different typologies** of storage systems:

- **Direct Attached Storage (DAS)** is a **storage system directly attached to a server or workstation**. They are visible as disks/volumes by the client OS.
- **Network Attached Storage (NAS)** is a **computer connected to a network that provides only file-based data storage services** (e.g. FTP, Network File System) to other devices on the network and is visible as File Server to the client OS.
- **Storage Area Networks (SAN)** are **remote storage units connected to a PC using a specific networking technology** (e.g. Fiber Channel) and are visible as disks/volumes by the client OS.

In the following schema, we can see a simple architectural comparison.



✔ **DAS features**

DAS is a **storage system directly attached to a server or workstation**. The term is used to differentiate non-networked storage from SAN and NAS. The **main features** are:

- Limited scalability.
- Complex manageability.
- Limited performance.
- To read files in other machines (the "file sharing" protocol of the OS must be used).

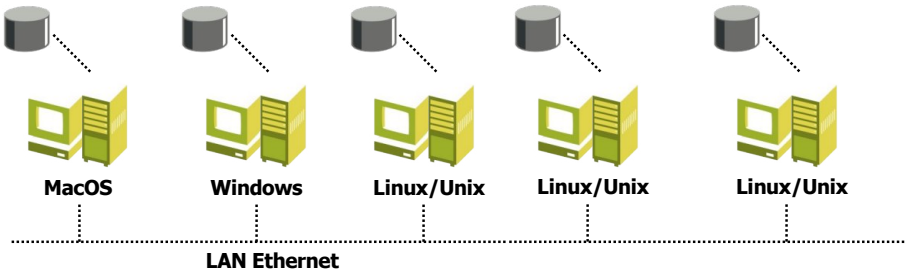


Figure 9: DAS architecture.

Note that all the **external disks connected to a PC with a point-to-point protocol can be considered DAS**.

✔ NAS features

A NAS unit is a **computer connected to a network that provides only file-based data storage services to other devices on the network**. NAS systems contain one or more hard disks, often organized into logical redundant storage containers or RAID. Finally, **NAS provides file-access services to the hosts connected to a TCP/IP network through Networked File Systems/SAMBA**. Each NAS element has its IP address. Furthermore, each NAS has good scalability.

The **main differences between DAS and NAS** are:

- DAS is simply an **extension of an existing server** and is **not necessarily networked**.
- NAS is designed as an easy and self-contained solution for **sharing files over the network**.

Regarding **performance**, NAS depends mainly on the speed and congestion of the network.

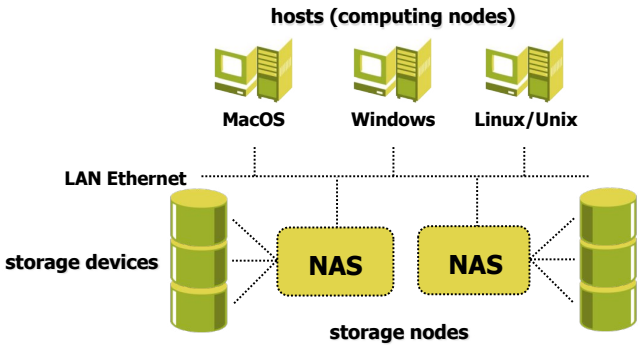


Figure 10: NAS architecture.

✔ SAN features

SANs are remote **storage units connected to servers using a specific networking technology**. SANs have a particular network dedicated to accessing storage devices. It has two distinct networks: one TCP/IP and another dedicated network (e.g. Fiber Channel). It has a high scalability.

The **main difference between a NAS and a SAN** is that:

- **NAS appears to the client OS as a file server**. Then, the client can map network drives to shares on that server.
- **A disk available through a SAN still appears to the client OS as a disk**. It will be visible in the disks and volumes management utilities (along with the client's disks) and available to be formatted with a file system.

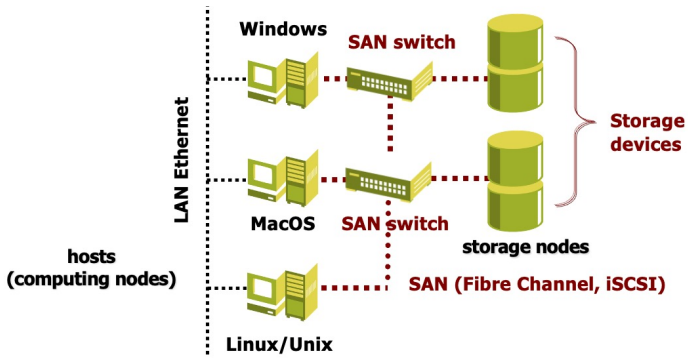


Figure 11: SAN architecture.

	Application Domain	Advantages	Disadvantages
DAS	• Budget constraints • Simple storage solutions	• Easy setup • Low cost • High performance	• Limited accessibility • Limited scalability • No central management and backup
NAS	• File storage and sharing • Big Data	• Scalability • Greater accessibility • Performance	• Increased LAN traffic • Performance limitations • Security and reliability
SAN	• DBMS • Virtualized environments	• Improved performance • Greater scalability • Improved availability	• Costs • Complex setup and maintenance

Figure 12: DAS vs. NAS vs. SAN

2.2.3 Networking (architecture and technology)

2.2.3.1 Fundamental concepts

In the data centre, servers' *performance increases* over time, and the demand for inter-server *bandwidth also increases*.

One **initial solution** is to double the aggregate compute or storage capacity by **doubling the number of compute or storage elements**. The initial solution is horizontal scaling at the edge; we simply add more leaf switches with enough ports to handle the extra servers. However, this only works locally because the fundamental issue is deeper.

Since servers first connect to leaf switches, **adding twice as many servers results in twice as many ports on the leaf switches**. Thus, our **total edge bandwidth doubles**. For instance, if each leaf switch supports 40 servers at 1 Gbps, adding 40 more servers requires either an additional leaf switch or an expansion of the existing switch, doubling our total edge capacity. Thus, the **Leaf Bandwidth** is essentially the **sum of all connections between servers and their closest switches**.

What is the problem with this naïve solution? The critical point is that doubling the leaf bandwidth **only handles server-to-leaf traffic**. However, if **all servers must be able to talk to all other servers**, traffic must cross the entire DCN (Data Center Network). This way, the middle of the network, the **bisection**, **is not scaled**. If every server wants to communicate with every other server, then there must be enough internal network capacity to support all of these connections simultaneously. In other words, doubling the bandwidth of the leaves is trivial (we just add ports), but **doubling the bisection bandwidth is not**.

What is the bisection bandwidth?

Bisection Bandwidth is the total **network bandwidth available** to carry **traffic between two halves** of a network if we split it down the middle.

Imagine our whole datacenter network is a grid of racks and switches. Draw an imaginary **line** that cuts it into two equal sides.

- All the network links **crossing that line** are our **bisection**.
- The **sum of the bandwidth of those links** is the bisection bandwidth.

In practice, this means that if **half of our servers** try to send data to the **other half** *simultaneously*, the bisection bandwidth tells us the **maximum total amount of traffic** that can be pushed across the split **at once**.

Example 10: Bisection Bandwidth

Suppose:

- 100 servers in total, 50 on the left, 50 on the right.
- Between left and right sides, there are 10 uplinks of 10 Gbps each.

So Bisection Bandwidth = 10 links $\times$10 Gbps = 100 Gbps. So:

- Max total traffic left $\rightarrow$ right = 100 Gbps
- Max total traffic right $\rightarrow$ left = 100 Gbps

If the servers try to send more than that across the split, packets get queued, then we get delays or packet drops.

How to design a data center network? Design principles

There are many design principles to follow:

- **Very scalable** in order to support a vast number of servers;
- **Minimum cost** in terms of basic building blocks (e.g. switches);
- **Modular** to reuse simple basic modules;
- **Reliable** and **resilient**;
- It may exploit novel/proprietary technologies and protocols incompatible with legacy Internet.

The **Data Center Network (DCN)** connects a data centre's computing and storage units to achieve optimum performance. It can be **classified** into **three main categories**:

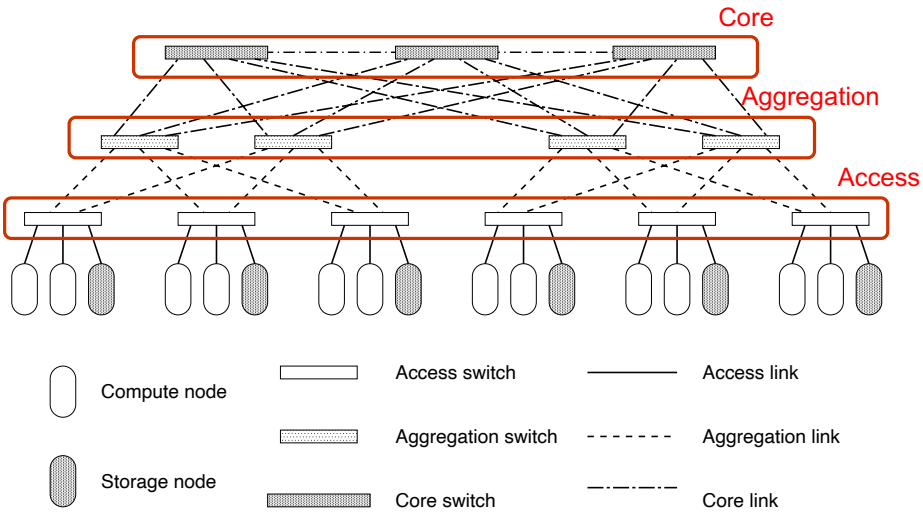
- **DCN Switch-centric architectures**. DCN uses **switches to perform packet forwarding**.[4]
- **DCN Server-centric architectures**. DCN **uses servers with multiple Network Interface Cards (NICs)**[5] **to act as switches** and perform other computational functions.
- **DCN Hybrid architectures**. DCN **combines switches and servers** for packet forwarding.

[4] **Packet forwarding** is the passing of packets from one network segment to another by nodes on a computer network.

[5] A **Network Interface Cards (NICs)** is a computer hardware component that connects a computer to a computer network.

2.2.3.2 Switch-centric: classical Three-Tier architecture

The **Three-Tier architecture**, also called **Three Layer architecture**, **configures the network in three different layers**:



It is a simple Data Center Network topology.

- The **servers** are **connected to the DCN through access switches**.
- **Each access-level switch is connected to at least two aggregation-level switches**.
- **Aggregation-level switches are connected to core-level switches** (gateways).

✔ Advantages

1. **Bandwidth can be increased** by **increasing the switches at the core and aggregation layers**, and by using routing protocols such as Equal Cost Multiple Path (ECMP) that equally shares the traffic among different routes.
2. **Very simple** solution.

👎 Cons

1. **Very expensive** in large data centers because the upper layers require faster network equipments.
2. **Cost very high** in term of acquisition and energy consumption.

In the **access layer**, there are two possible architectures:

- **ToR (Top-of-Rack) architecture**. It is a **switching architecture** for connecting servers **inside a rack** to the rest of the datacenter network. The idea is simple, we place a **network switch at the top of each rack**, this is the **Top-of-Rack switch (ToR switch)**. All servers inside that rack connect **upwards** to that switch. The ToR switch then connects **upstream** to aggregation/core switches that connect the whole datacenter.

  In other words, it **aggregates** all the server links in that rack.

  ✔ **Advantages**

  1. **Simpler cabling** because the number of cables is limited. Short cables run only from each server up to the rack's top.
  2. **Lower costs** because the number of ports per switch is limited.

  👎 **Cons**

  Higher complexity for switch management. Because there are more devices to manage (for example, with 100 racks, there are 100 top-of-rack switches) the control is distributed rather than centralized. This requires a more careful design to prevent loops or misconfigurations.

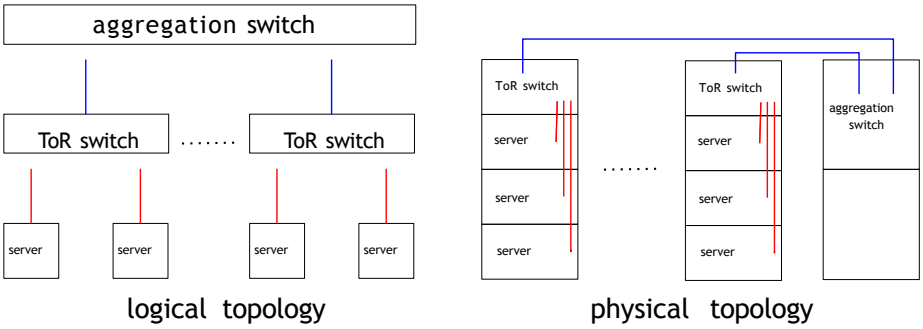


Figure 13: ToR (Top-of-Rack) architecture.

- **EoR (End-of-Row) architecture**. It is a classic **datacenter network architecture** for connecting racks of servers.

  Instead of putting a small switch **inside each rack**, we put **one larger switch at the *end* of each row** of racks. Servers in each rack connect **horizontally** to this shared switch.

  ❓ **How does it work physically?** Imagine a row of 10 racks. Each rack has no ToR switch, instead:

  – Servers in the rack have **longer cables** that run **horizontally** to the big EoR switch.

  – That EoR switch aggregates all server connections from the whole row.

– The EoR switch then uplinks to the aggregation/core layer.

✔ **Advantages**

Simpler switch management.

👎 **Cons**

The aggregation switches must have a larger number of ports, then:

1. **Complex cabling**: Longer cables from every server to the end of the row.
2. **High risk with a fault**: If the EoR switch fails, then whole row loses network.
3. **Longer cables** then **higher costs**.

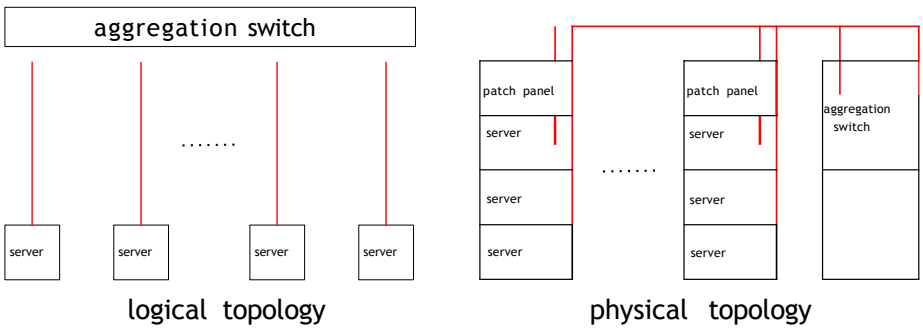


Figure 14: EoR (End-of-Row) architecture.

2.2.3.3 Switch-centric: Leaf-Spine architectures

The **Leaf-Spine Architecture** is a **two-tier network topology** used in modern data centers to **handle massive East-West traffic** and maximize bisection bandwidth.

- **Leaf switches** are the "**bottom layer**". These are basically **ToR (Top-of-Rack)** switches, they connect directly to the servers in each rack.
- **Spine switches** are the "**top layer**". These are high-capacity core switches. They **interconnect all the Leaf switches with each other**.

So **every Leaf switch is connected to every Spine switch**.

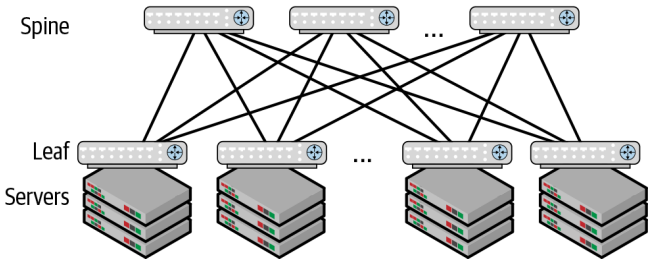


Figure 15: Leaf-Spine architecture.

What problem does it solve? The classic 3-tier DCN designs (page 83) **don't scale bisection bandwidth linearly**. They easily become bottlenecks if many servers talk to each other. Leaf-Spine fixes this by:

- Providing multiple parallel paths between any two racks.
- Ensuring that **bandwidth stays predictable** as we add more racks and more spines.
- Using equal-cost multipath (ECMP) to load-balance flows across all available paths.

What is a Clos network, and how is it linked to leaf-spine architecture?

Originally, the **Clos network** was invented by Charles Clos (1952) for telephone switches. It's a **multi-stage switching fabric** that allows us to build a big non-blocking switch **out of many small cheap switches**. The core idea: instead of one giant switch, we **connect multiple small switches in stages** to get scalable bandwidth **with multiple paths**. In datacenter networking, the **Clos design** maps naturally to modern switch hardware.

Why is Clos good for datacenters? In huge datacenters:

- We want **full bisection bandwidth**: any server can talk to any other at line rate.

- We can't build a single giant switch with thousands of ports, that's impractical.
- Instead, we interconnect many small switches in a **Clos-like fabric** to provide multiple paths and load balancing.

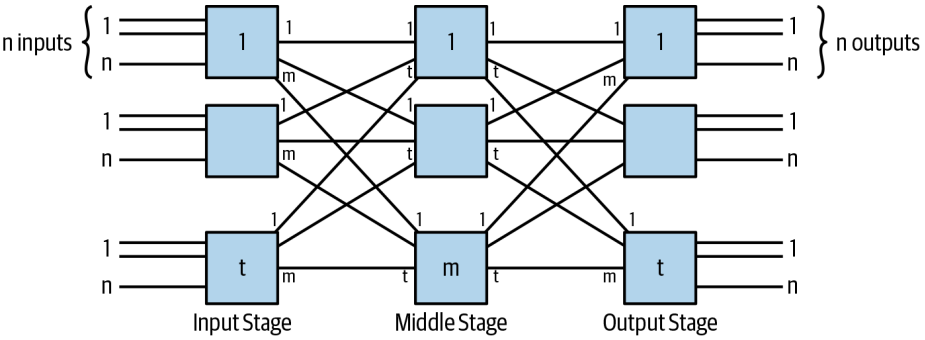
How does the Leaf-Spine topology come from Clos? **Leaf-Spine** is basically the **simplest practical version** of a 2-stage or 3-stage **Clos network**, tailored for Ethernet.

Formal Definition. A **Clos network** is a **multi-stage interconnection network** defined by three integers:

- n: number of inputs per input module.
- m: number of middle-stage switches.
- k: number of output ports per output module.

A classic 3-stage Clos is defined as $C(n, m, k)$:

- Stage 1: k input modules, each with n input ports.
- Stage 2: m middle-stage switches.
- Stage 3: k output modules, each with n output ports.



- Each input module connects to every middle-stage switch.
- Each middle-stage switch connects to every output module.
- So every path crosses: 1 input module, 1 middle-stage switch, 1 output module.

This network has two main non-blocking conditions:

- The Clos network is **rearrangeably non-blocking** if: $m \geq n$ (number of middle-stage switches is greater or equal than number of inputs per input module). So we can rearrange connections to always find a path.

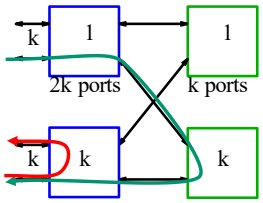
- The Clos network is **strictly non-blocking** if: $m \geq 2n - 1$ (the number of middle-stage switches is greater than or equal to twice the number of inputs per input module minus one).

We **mapped** the **Clos network** in data centers **to the leaf-spine** architecture:

- The **input modules** become the **Leaf switches**.
- The **middle-stage** switches become the **Spine switches**.
- The **output modules** fold back to the same Leafs (because Ethernet is bidirectional).

They often simplify by setting: $n = m = k$. The Leaf-Spine architecture now has:

- Each **Leaf** switch has $2k$ **bidirectional ports**:
  - k ports to servers (down).
  - k ports up to the Spines.
- Each **Spine** switch has k **bidirectional ports**: one link to each Leaf.



In order to achieve **full bandwidth**, each Leaf must be connected to each Spine. So the total number of **uplinks per Leaf** is equal to the number of Spines.

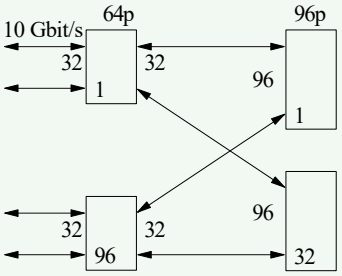
In summary:

$$\begin{cases} \text{Leaves: } 2k \text{ bidirectional ports} \\ \text{Spines: } k \text{ bidirectional ports} \\ \text{Every Leaf connects to every Spine} \\ \text{Full bisection if } m \geq k. \end{cases}$$

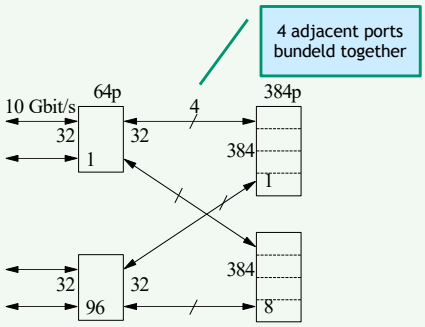
Example 11: DNC design

There are 3072 servers and 3072 ports available at 10 Gbit/s. Provides a leaf-spine design.
There are **two possible designs**.

1. The first consists of 96 switches with 64 ports and 32 switches with 96 ports.



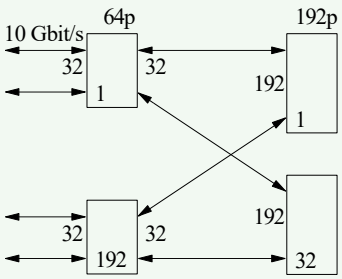
2. The second has only 8 switches but they have more ports: 384 ($8 \times 384 = 3072$).



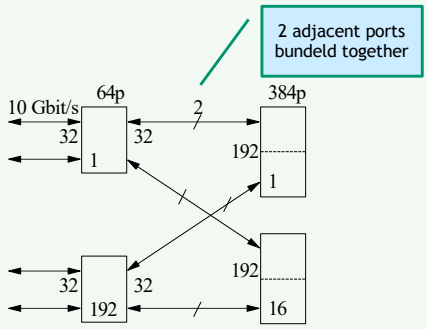
Example 12: DNC design

There are 6144 servers and 6144 ports available at 10 Gbit/s. Provides a leaf-spine design.
There are **two possible designs**.

1. The first consists of 192 switches with 64 ports and 32 switches with 192 ports.



2. The second has only 16 switches but they have more ports: 384 ($16 \times 384 = 6144$).



Why do we scale to a 3-tier Clos?

When our datacenter gets **huge**, even a classic 2-tier Leaf-Spine hits limits:

- The number of **uplinks per Leaf** is fixed by switch port count.
- The number of Spines we can connect to is also limited by the port count on the Spines.

So to connect *thousands* of racks at full bisection bandwidth, we need more stages. This is exactly what the **Fat-Tree (or 3-stage Clos)** solves.

A **Fat-Tree (or 3-stage Clos)** is just a multi-level **Clos network** where link capacity *increases* closer to the core, so the "tree" looks **fat** in the middle. The key idea is:

- Small switches at the **edge**.
- Bigger bandwidth links and more switches at the **aggregation and core layers**.
- So the middle of the network doesn't become a bottleneck.

It is called "fat" because, in a normal tree, the higher up we go, the fewer branches there are, creating bottlenecks. In a Fat-Tree, we keep adding enough parallel links/switches so total bandwidth stays equal across any "slice". So every layer has enough capacity to preserve full bisection bandwidth.

How it works? 3-tier. A classic Fat-Tree has:

- **Edge (Leaf)**: ToR switches in racks, connect to servers and Aggregation.
- **Aggregation (Middle)**: Aggregate Leafs, uplink to Core.
- **Core (Spone/Superspine)**: Connect Aggregations together.

So it's basically a Clos $C(n, m, k)$:

$$\text{Edge stage} \rightarrow \text{Middle stage} \rightarrow \text{Core stage} \rightarrow \text{Middle stage} \rightarrow \text{Edge}$$

Why do we call it Pod-based?

When we have *thousands* of racks, it's unmanageable to think *of one giant switch fabric*. So we **divide** the whole fabric **into repeated building blocks** called Pods. A **PoD (Point of Delivery)**, in datacenter design, is a **modular**, **repeatable unit** of:

- **Racks + Power + Leaf + Aggregation Switches**.
- Often also cooling and structured cabling.

A POD is like a **mini-datacenter inside the datacenter**.

A Scalable, Commodity Data Center Network Architecture [1]

The paper "*A Scalable, Commodity Data Center Network Architecture*" [1] explains a famous datacenter Clos design that shows how we can build a **non-blocking DCN** with commodity switches. The idea is:

- We have a **fixed switch port count**: each switch has k ports.
- The goal is to **connect as many servers as possible** with full bisection bandwidth, no oversubscription[6].
- The Fat-Tree does this by arranging switches in **pods** and a **core layer**.

In the classic Fat-Tree, we choose: $P = 2k$, where P is the number of **pods**, and k is the **number of ports per switch**. All switches are assumed to be the same commodity switch.

Each **pod** has:

- k Edge switches. Each edge switchL
  - Connects **down** to servers: $\frac{k}{2}$ ports.
  - Connects **up** to Aggregations: $\frac{k}{2}$ ports.
- k Aggregation switches. Each Aggregation switch:
  - Connects **down** to Edge: $\frac{k}{2}$ ports.
  - Connects **up** to Core: $\frac{k}{2}$ ports.

[6] Oversubscription is when the total potential demand for bandwidth exceeds the actual available capacity. It's like building a highway with fewer lanes than the number of cars that could be on it, because we're betting not everyone will drive at once.

So each Edge switch connects to:

$$\frac{k}{2}\,\text{servers} + \frac{k}{2}\,\text{uplinks} = k\text{total ports}$$

Total servers per pod:

$$k\,\text{Edge switches} \times \frac{k}{2}\,\text{servers each} = \frac{k^2}{2}$$

But because there are P **pods**, the total servers are:

$$\text{Total servers} = \underbrace{\frac{k^2}{2}}_{\text{per pod}} \times P$$

However, we adopted a generalized variant:

$$\text{Total servers} = k^2 P \tag{16}$$

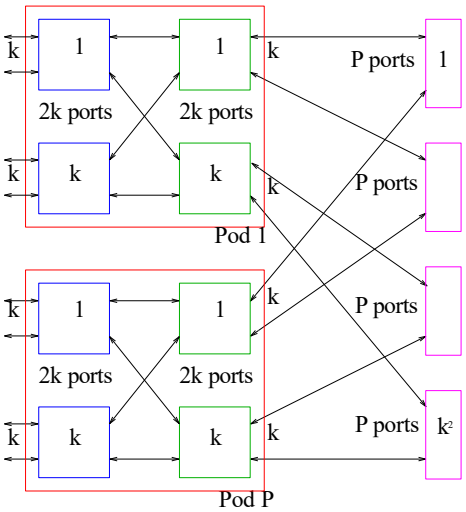


Figure 16: Explanation of Fat-Tree Network.

Example 13: Fat-Tree with $2k$ pods and $k = 2$

Suppose a Fat-Tree has $2k$ pods: $P = 2k$. Each pod has its Edge and Aggregate layers. The core ties all pods together.

- If $P = 2k$, then the total servers are:

  $$\text{Total servers} = k^2 \times P = k^2 \times 2k = 2k^3$$

  So the k-ary Fat-Tree connects $2k^3$ servers.

- If each pod:

  $$k\text{Edge} + k\text{Aggregation} \Rightarrow 2k\text{switches}$$

With P pods, we have $2k \cdot P$ switches in pods. Also, at the Core layer we have k^2 Core switches. So **total switches**:

$$2kP + k^2 \quad \xrightarrow{P=2k} \quad 2k \cdot (2k) + k^2 = 4k^2 + k^2 = 5k^2$$

For $k = 2$, we have:

- $P = 4$ pods.
- Core: $k^2 = 2^2 = 4$ switches.
- Total servers: $2k^3 = 2 \times 2^3 = 16$.
- Total switches: $5k^2 = 5 \times 4 = 20$.

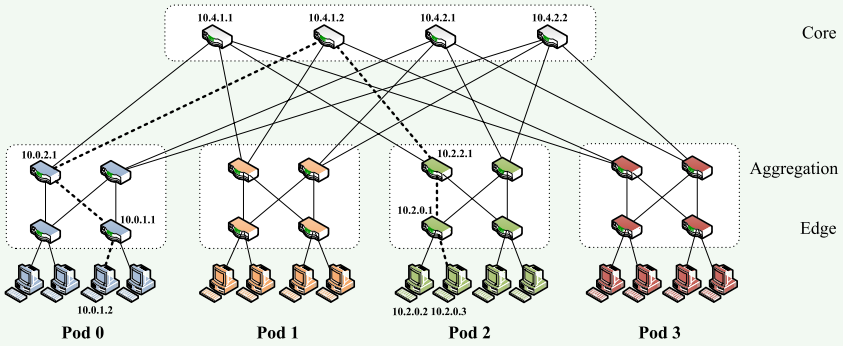


Figure 17: Simple fat-tree topology. Using the two-level routing tables, packets from source `10.0.1.2` to destination `10.2.0.3` would take the dashed path. [1]

Classic k-ary Fat-Tree:

$$\begin{cases} P = 2k, \\ \text{Total servers} = 2k^3, \\ \text{Total switches} = 5k^2, \\ \text{Core switches} = k^2, \\ \text{Pods} = 2k. \end{cases}$$

2.2.3.4 Server-centric and hybrid architectures

In **server-centric architectures**:

- The servers themselves **help do the routing** and **forward traffic**.
- The network topology is built **partly in software**, with direct server-to-server links.
- Switches are minimal or even absent, the servers form the fabric.

So the key idea is that servers are not passive endpoints, but they **actively forward packets** for each other.

CamCube: a concrete example

CamCube was an experimental **server-centric datacenter network** developed at the University of Cambridge around 2010. The key features are:

- Each server has multiple NICs.
- Servers are connected in a **3D torus** or **cube-like mesh** (hence "CamCube").
- There are **no traditional switches**, servers route packets directly to each other.

A CamCube server has **several network ports** (e.g., 6 in the 3D grid). Links go **directly to neighboring servers**. Packets hop from server to server, using custom routing software running **on the servers themselves**. Routing decisions are done in the OS or application layer.

Why do this?

- Reduce or eliminate expensive switches.
- Achieve high bisection bandwidth using cheap NICs and cables.
- Expose the **topology directly to applications**, so apps can optimize data placement and routing.

For example, a key-value store could decide to place replicas on "neighbor" servers in the CamCube coordinate space to minimize hops.

Advantages

- Cheap commodity hardware (just NICs and cables).
- Flexible (easy to experiment with custom routing).
- Tight integration between network and application logic.

Disadvantages

- Adds **routing overhead** on servers.
- Makes the OS and application design more complex.
- Harder to scale wiring neatly, 3D torus cabling is tricky in real racks.
- Server failure impacts routing, so resilience needs extra software.

The hybrid architectures are **DCell**, **BCube** and **MDCube**.

What is DCell?

DCell is a **server-centric datacenter network** proposed by Microsoft Research (Guo et al., 2008). The key idea is build a large-scale datacenter network **without big switches**, using small commodity switches and **servers interconnecting themselves recursively**.

In DCell each **server** is connected to:

- A **mini switch** (commodity, dumb, small, no big core).
- **Other servers directly**, in a clever recursive pattern.

The result is a **highly scalable, fault-tolerant topology** built by **recursively composing many small building blocks**. The core concept is **recursive construction**. We start with a DCell_0, the basic unit, then **combine multiple DCells into bigger DCells**.

1. DCell_0, smallest unit. Contains n servers and one mini switch. Inside:
   - Each server has one port to the switch (for local communication).
   - Each server has **one more port** for direct links to other DCells.
2. DCell_1, we take $k = n + 1$ DCell_0 units. Connect them **directly** using the extra server ports, and each server has direct links to other DCells.
3. DCell_2, we take $k = n + 1$ DCell_1 units. Connect them recursively in the same way.

And so on. At each level, the network expands **exponentially**.

Advantages

- **Scalability**: The recursive structure lets us scale to hundreds of thousands of servers using small switches and extra server NICs.
- **Server-centric**: Servers handle routing logic, they decide how to forward packets to reach other servers.
- **Fault tolerance**: Many redundant paths. If one server or link fails, routing can find alternate paths.

Disadvantages

- Needs extra NICs per server, then **more cabling complexity**.
- Routing at server level adds **CPU overhead**.
- Recursive **cabling** can be operationally **hard** at large physical scales.
- **Performance can degrade if too many servers fail**, the fault tolerance is not free.

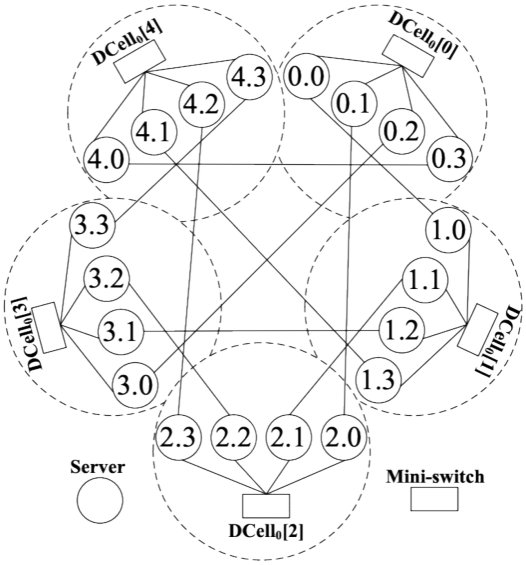


Figure 18: DCell hybrid architecture.

What is BCube?

BCube is a **server-centric datacenter network topology** proposed by Microsoft Research (Guo et al., 2009), same lab that proposed DCell. It was designed specifically for **modular datacenters**, think **container-based clusters** or shipping container datacenters.

Like DCell, BCube:

- Uses **commodity mini-switches** (no big expensive core switches).
- Gives each server **multiple NICs**.
- Builds high bisection bandwidth **by using direct server-switch-server paths**.

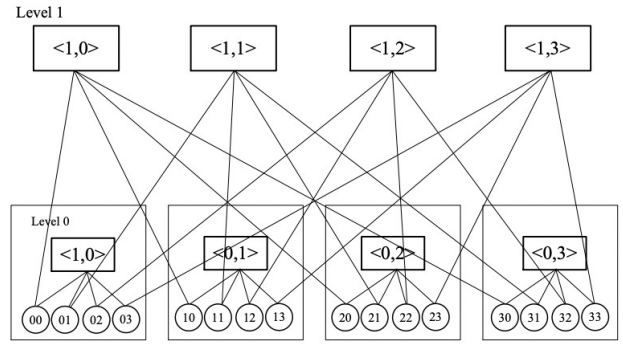
But instead of recursive hierarchy (DCell), BCube uses a **layered, multi-level cube-like structure** with **structured direct links**.

Advantages

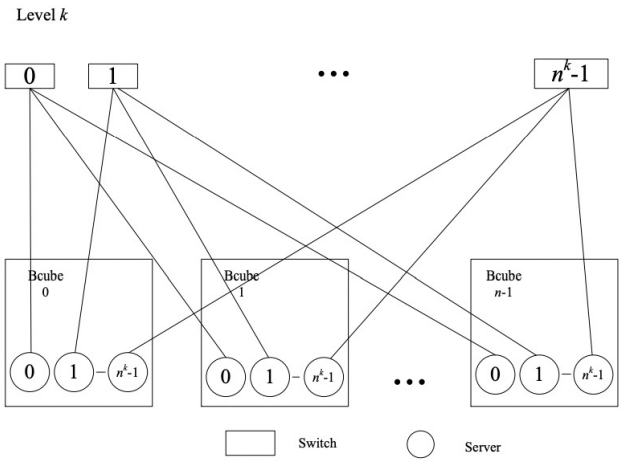
- **Server-centric**: servers handle multi-path routing.
- **High fault tolerance**: if one path fails, many alternative switch levels are available.
- **Great for modular scale**: we can design BCube units as self-contained shipping containers.
- **Full bisection bandwidth**: as we add levels, we add more disjoint paths.

Disadvantages

- **Limited scalability**.
- **High cabling costs**.



(a) Request phase.



(b) Grant phase.

Figure 19: BCube hybrid architecture.

What is MDCube?

MDCube is a **modular datacenter network architecture** that **combines multiple BCubes** into a higher-level *interconnected cube*. It's literally: **Modular Datacenter Cube (MDCube)**. Proposed by the same researchers (Microsoft Research, 2011) to tackle the problem: "*BCube works well for modular, container-sized datacenters; but what if we need to scale to* ***mega-datacenters*** *built from many BCube containers?*".

Each **BCube** is treated as a **building block**. We interconnect multiple BCubes in a **higher-level topology** (like a hypercube or torus). The result is a **multi-level hybrid**:

- Inside: each module is a BCube.
- Outside: the modules are linked by structured interconnects.

MDCube is **hybrid** because:

- It combines **server-centric routing** inside the BCube.
- It uses **structured interconnects** *between* modules, like classic network topologies.
- It blends **switch-centric** commodity switches (inside BCube) with **server centric** direct links (between BCubes).

Advantages

- Keeps switch cost low.
- Good for container-based datacenters.

Disadvantages

Still operationally tricky, then many NICs, custom routing software.

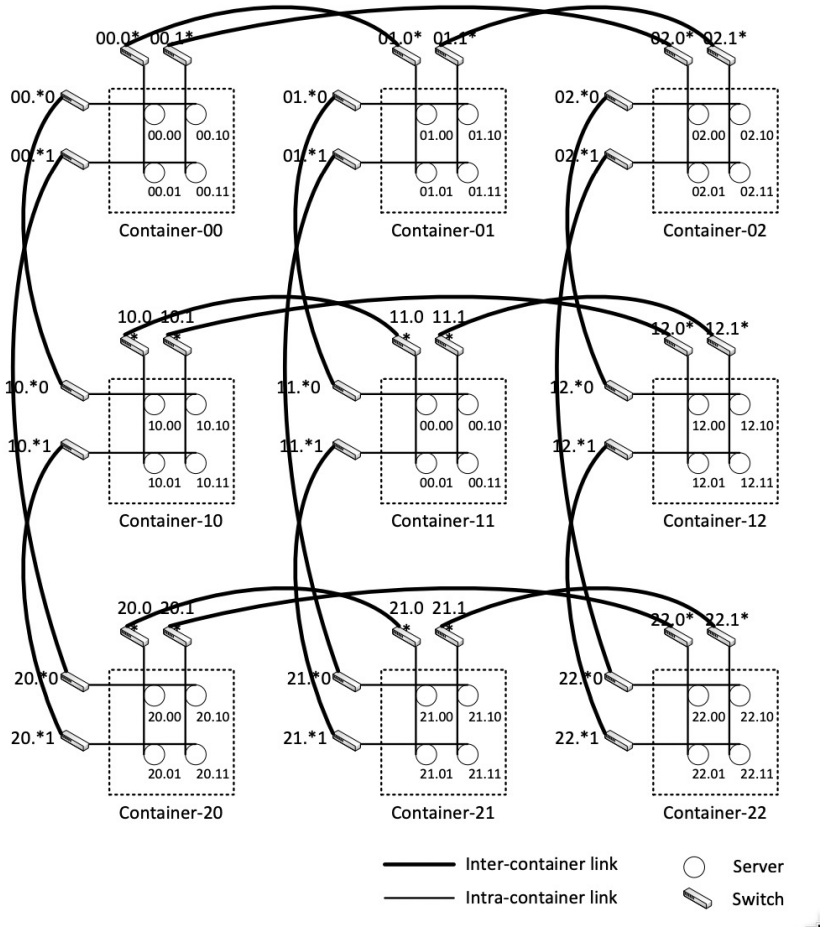


Figure 20: MDCube hybrid architecture.

2.3 Building level

The main components of a typical data center are:

- Cooling system (blue):
  - Water storage
  - Cooling towers
  - Chillers
  - Fan coil air handling units
- Power supply (red):
  - Utility power
  - Transformers
  - Backup generation/power distribution
  - Power bus
- Computation storage networking (green):
  - Networking room

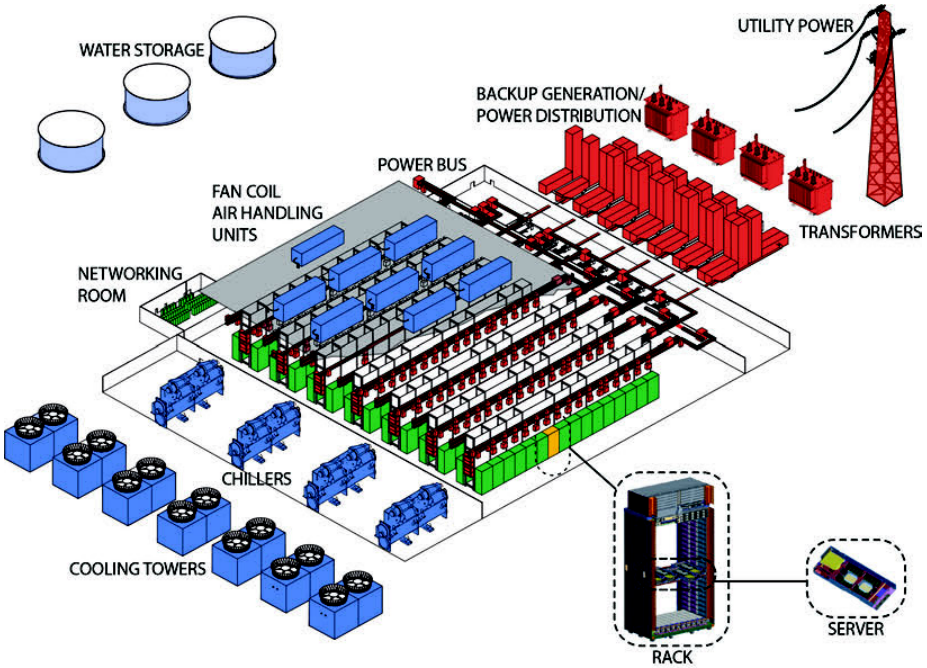


Figure 21: The main components of a typical data center. [2]

The warehouse scale computer or data centre has other important components related to **power delivery**, **cooling** and **building infrastructure** that also need to be considered.

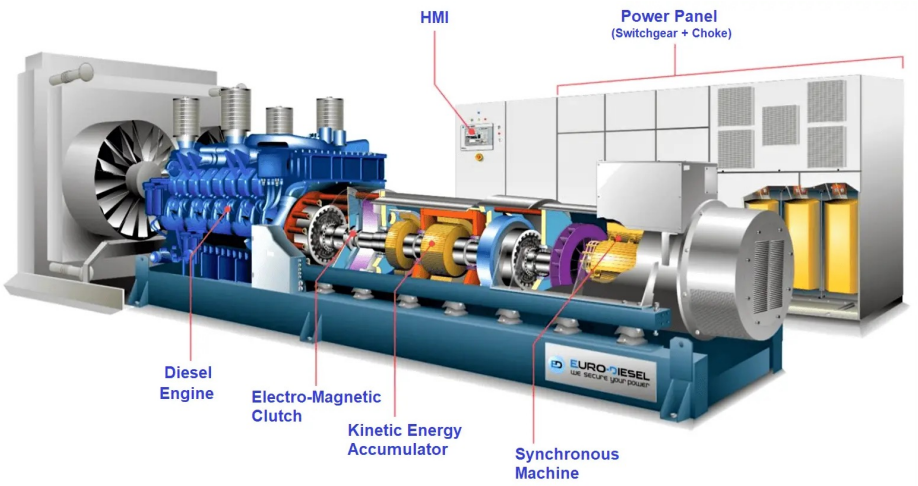


Figure 22: A Rotary UPS system.

In order to protect against power failure, battery and diesel generators are used to back up the external supply.

A **UPS (uninterruptible power supply or source)** is a **continual power system that provides automated backup electric power to a load when the input power source or mains power fails**. There are many types of UPS, but in general, in the DC, the **Rotary UPS system** is used.

A rotary UPS uses the inertia of a high-mass spinning flywheel to provide short-term ride-through in the event of power loss.

2.3.1 Cooling systems

The IT equipment generates a lot of heat. To avoid troubles, cooling systems have been installed. Unfortunately, they are costly components of the data center, and they are composed of **coolers**, **heat exchangers** and **cold water tanks**.

Some techniques exist to improve cooling systems without throwing away too much money.

Open-Loop systems refer to the **use of cold outside air to either help the production of chilled water or directly cool servers**. It is not entirely free in the sense of zero cost, but it involves **very low energy costs** compared to chillers.

Closed-Loop systems come in many forms, the most common being the air circuit on the data centre floor. It is grouped by number of loops:

- ***One loop***. The **main goal is to isolate and remove heat from the servers and transport it to a heat exchanger**. So the cold air flows to the servers, heats up, and eventually reaches a heat exchanger to cool it down again for the next cycle through the servers.

  ✔ **Advantages**

  It can be very efficient.

  ⚠ **Cons**

  - It doesn't work in all climates;
  - It requires filtering of airborne particulates;
  - Can introduce complex control problems.

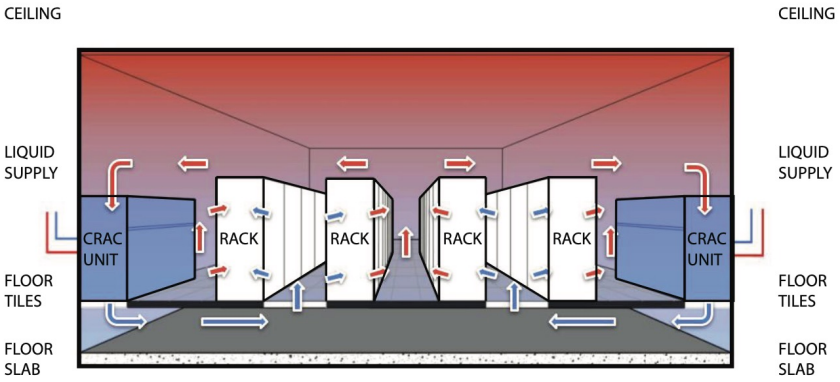
- ***Two loops***. The airflow through the underfloor plenum, the racks, and back to the **CRAC (computer room air conditioning)** defines the primary air circuit as the first loop. The second loop (the liquid supply inside the CRAC units) leads directly from the CRAC to the external heat exchangers (typically placed on the building roof) that discharge the heat to the environment.

  ✔ **Advantages**

  - Easy to implement;
  - Relatively inexpensive to construct;
  - Offers isolation from external contamination.

  ⚠ **Cons**

  Typically have lower operational efficiency.



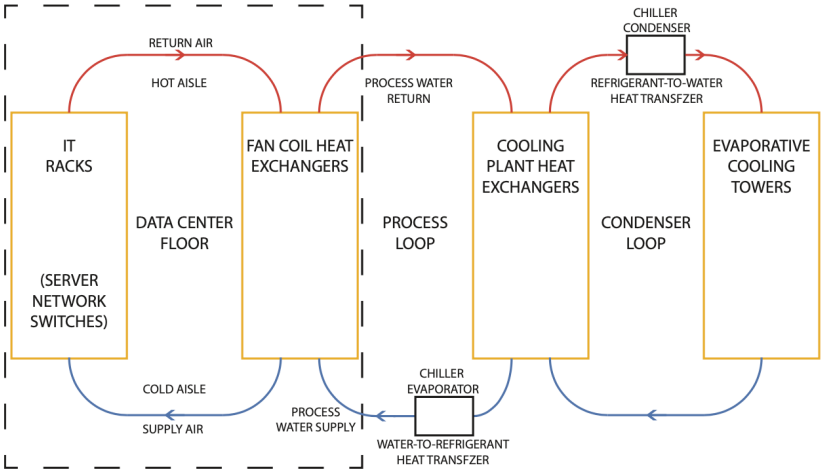
- ***Three loops***. It is used in large-scale data centres. The architecture can be viewed in the following figure.

  ✔ Advantages

  – It offers contaminant protection;
  – It offers good efficiency.

  ⚠ Cons

  – It is the most expensive to construct;
  – It has moderately complex controls.



First loop to the left, second loop in the middle and third loop to the right.

How each rack is cooled?

There are three ways to cool each rack:

- **In-Rack cooler**. It **adds an air-to-water heat exchanger** at the **back of a rack** so the **hot air exiting the servers immediately flows over coils cooled by water**, reducing the path between server exhaust and CRAC input.
- **In-Row cooling**. It works like in-rack cooling, except the **cooling coils** are not in the rack but **adjacent to the rack**.

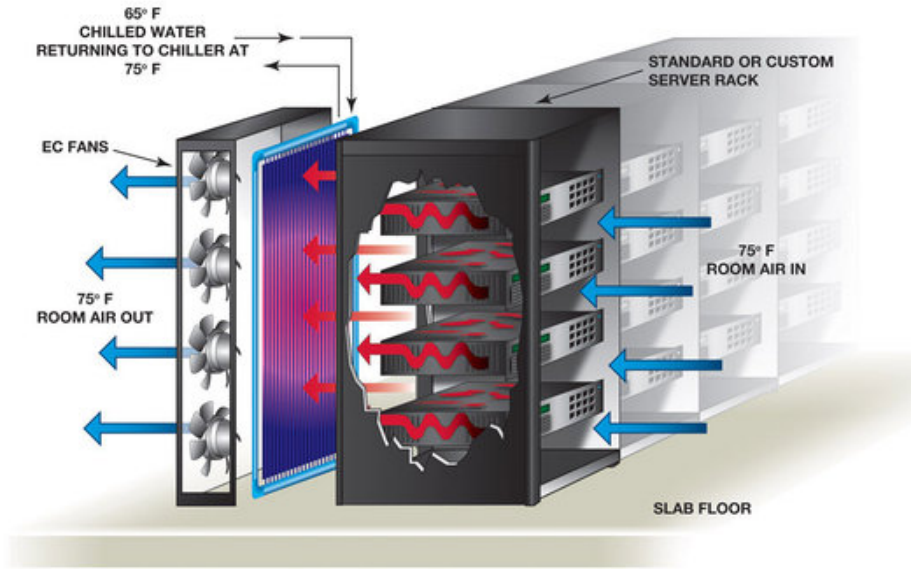


Figure 23: In-Row Cooling Mechanism (source: Energy Start).

- **Liquid cooling**. We can directly **cool server components using cold plates**. The **liquid circulating through the heat skins transports the heat to a liquid-to-air or liquid-to-liquid heat exchanger** that can be placed close to the tray or rack or be part of the data centre building (such as a cooling tower).

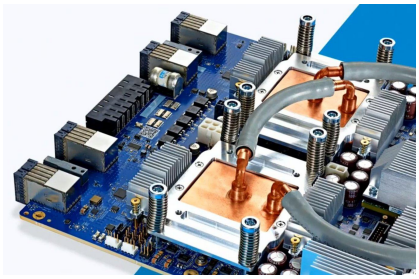


Figure 24: Liquid Cooling Mechanism.

2.3.2 Power supply

What is the problem?

Data centre power consumption is an issue since it can reach several milliwatts. **Cooling usually requires about half the energy the IT equipment requires** (servers + network + disks). Finally, **energy transformation also wastes** much energy when running a data centre.

Is there a metric to measure energy efficiency?

First of all, **no**! Several metrics are helpful to understand how a data centre spends in terms of energy.

One of the most critical metrics is **Power Usage Effectiveness (PUE)**. It is the **ratio of the total amount of energy used by a DC facility to the energy delivered to the computing equipment**:

$$\texttt{PUE} = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}} \tag{17}$$

Where the **Total Facility Power** is calculated as:

$$\texttt{TFP} = \text{covers IT systems} \; + \; \text{other equipment} \tag{18}$$

Where the covers IT systems are servers, network, storage, and other equipment are cooling, UPS, switch gear, generators, lights, fans.

Finally, the **Data Center Infrastructure Efficiency (DCiE)** is the inverse of PUE:

$$\texttt{DCiE} = \texttt{PUE}^{-1} = \frac{\text{IT Equipment Power}}{\text{Total Facility Power}} \tag{19}$$

For **example**, the level of efficiency is shown here:

PUE	DCiE	Level of Efficiency
3.0	33%	Very Inefficient
2.5	40%	Inefficient
2.0	50%	Average
1.5	67%	Efficient
1.2	83%	Very Efficient

2.3.3 Data Center availability

The Data Center availability is defined by in four different tier level. Each one has its own requirements:

Tier Level	Requirements
1	• Single non-redundant distribution path serving the IT equipment. • Non-redundant capacity components. • Basic site infrastructure with expected availability of 99.671%.
2	• Meets or exceeds all Tier 1 requirements. • Redundant site infrastructure capacity components with expected availability of 99.741%.
3	• Meets or exceeds all Tier 2 requirements. • Multiple independent distribution paths serving the IT equipment. • All IT equipment must be dual-powered and fully compatible with the topology of a site's architecture. • Concurrently maintainable site infrastructure with expected availability of 99.982%.
4	• Meets or exceeds all Tier 3 requirements. • All cooling equipment is independently dual-powered, including chillers and heating, ventilating and air conditioning (HVAC) systems. • Fault-tolerant site infrastructure with electrical power storage and distribution facilities with expected availability of 99.995%.

Table 10: Data Center availability.

3 Software Infrastructure

3.1 Virtualization

3.1.1 What is a Virtual Machine?

A **Virtual Machine (VM)** is a **logical abstraction** able to **provide a virtualized execution environment**. More specifically, a VM:

- Provides identical software behavior
- Consists in a combination of physical machine and virtualizing software
- May appear as different resources than physical machine
- May result in different level of performances

Exists two type of Virtual Machine: Process VM (page 109) and System VM (page 110).

What's the difference between a physical machine and a virtual machine?

First of all, the **physical machine** is the computer that can ***host*** n **virtual machines**.

Furthermore, every VM is based on hypervisor software (also known as a virtual machine manager or monitor VMM, page 112). The hypervisor runs as an application on the host operating system (hosted hypervisor) or rests directly on the hardware of the physical machine (bare-metal hypervisor) and manages the hardware resources provided by the host system. The **hypervisor software creates an abstraction layer between physical hardware and virtual machines**. **Each VM runs isolated from the host system and other guest systems on its own virtual environment**. This is referred to as encapsulation.

Processes within a virtual machine do not affect the host or other VMs on the same hardware.

So, to sum up:

1. **Physical machines** are the computers that can ***host*** n **virtual machines**.
2. Each **physical machine has a hypervisor (VMM)** already enabled or asleep on the hardware. **It manages the resources made available by the physical machine**.
3. Each **virtual machine has its own virtual environment**, so they are **encapsulated**, **isolated environments.**

   Obviously, the statement is not true if there is a "*virtual machine escape attack*", but we don't count those extreme cases. [9]

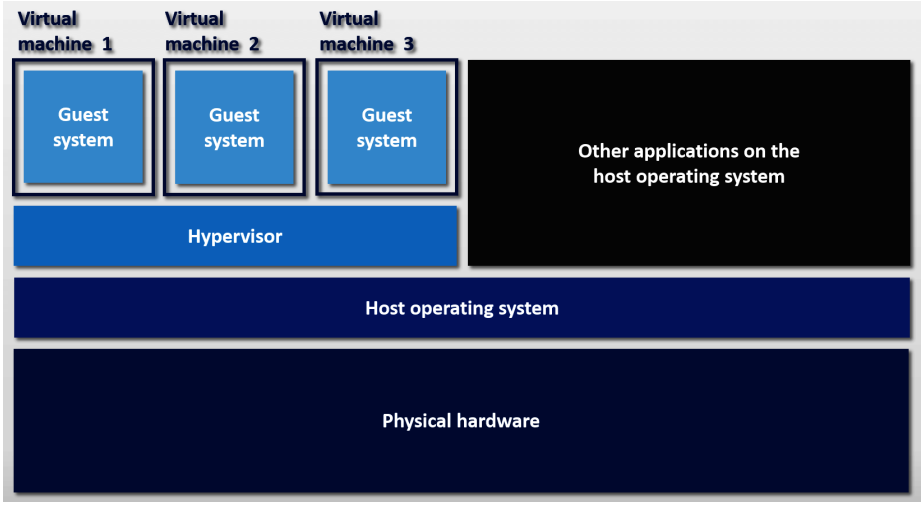


Figure 25: Operating system view if there are VMs in the physical machine (source: ionos).

A little terminology about *host* and *guest*:

- **Host**: the underlying platform supporting the environment/system.
- **Guest**: the software that runs in the Virtual Machine environment as the guest.

3.1.1.1 Process VM

A **Process Virtual Machine**, sometimes called an application virtual machine, or Managed Runtime Environment (MRE), **runs as a normal application inside a host OS and supports a single process**.

The **Virtual Machine is created when that process begins and destroyed when it ends**. A good example is the Java Virtual Machine JVM (see more here).

The purpose of a process VM is to execute a computer program in a platform-independent environment, meaning it can run on a variety of hardware or software.

The virtualizing software:

- is placed at the ABI[7], on top of the OS/hardware combination.
- emulates both user-level instructions and operating system calls.
- is usually called Runtime Software.

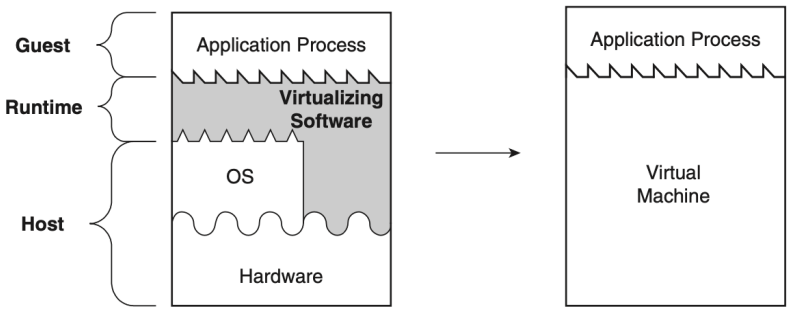


Figure 26: Process VM.

[7] An **Application Binary Interface (ABI)** corresponds to "*Operating system machine level*".

3.1.1.2 System VM

System Virtual Machines are substitutes for real machines and **provide all the functionalities of an actual operating system**. It provides operating system running in it access to underlying hardware resources (networking, I/O, GUI).

With a system VM, the hypervisor will access the underlying machine's resources, giving the user the same capabilities the host device offers.

The **virtualization software is called Virtual Machine Monitor (VMM)**.

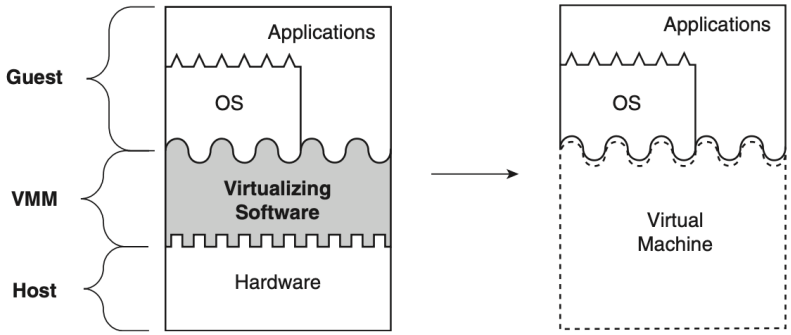


Figure 27: System VM.

3.1.2 Virtualization Implementation

Consider a typical layered architecture of a system by adding layers between layers of the execution stack. Depending on where the new layer is placed, we get different types of virtualization. The virtualization technologies are:

- **Hardware-level virtualization**. The **virtualization layer is placed between hardware and OS**. The interface seen by OS and application might be different from the physical one.
- **Application-level virtualization**. The **virtualization layer is placed between the OS and some application** (e.g. JVM). It provides the same interface to the applications. The applications run in their environment, *independently* from OS.
- **System-level virtualization**. The **virtualization layers provides the interface of a physical machine to a secondary OS and a set of application running in it, allowing them to run on top of an existing OS**. It is placed between the system's OS and other OS (e.g. VMware, VirtualBox). We can enable several OSs to run on a single hardware.

The properties of virtualization technologies are:

- **Partitioning**
  - Execution of multiple OSs on a single physical machine;
  - Partitioning of resources between the different VMs.
- **Isolation**
  - Fault tolerance and security (hardware level);
  - Advanced resource control to guarantee performance (managed by the hypervisor).
- **Encapsulation**
  - The entire state of a VM can be saved in a file (e.g. freeze and restart the execution);
  - Because a VM is a file, can be copied/moved as a file.
- **Hardware independence**
  - Provisioning/migration of a given VM on a given physical server.

3.1.3 Virtual Machine Managers (VMM)

A **Virtual Machine Manager (VMM)** is an application that:

- **Manages** the **VMs**;
- **Mediates access to the hardware resources** on the physical host system;
- Intercepts and **handles** any **privileged** or **protected instructions issued by the VMs**.

This type of virtualization typically runs virtual machines whose operating system, libraries, and utilities have been compiled for the same type of processor and instruction set as the physical machine on which the virtual systems are running.

Note that the Virtual Machine Manager (VMM) can be referred to by different names and also different meanings:

- **Virtual Machine Manager (VMM)**. The virtualization software.
- **Virtual Machine Monitor**. A virtualization software that runs directly on the hardware.
- **Hypervisor**. A VMM or Hypervisor that is also used to create, configure and maintain virtualized resources. It provides a user-friendly interface to the underlying virtualization software.

There are two types of hypervisor:

- **Type 1 Hypervisor** or **Bare Metal Hypervisor** (or **Native Hypervisor**). The term ***bare metal*** refers to the fact that **there is no operating system between the virtualization software and the hardware**. The virtualization software resides on the "bare metal" or the hard disk of the hardware, where the operating system is usually installed.

  Then, the **hypervisor takes direct control of the hardware**.

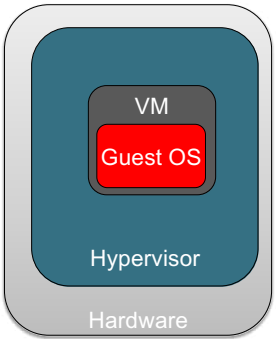


Figure 28: View of the Bare Metal Hypervisor.

Bare Metal Hypervisors can also be **built in two ways**:

– **Monolithic architecture**. Device **drivers run within the hypervisor**.

Advantages

* Better **performance**.
* Better **isolation**.

Disadvantages

* Can run only on **hardware** for which the **hypervisor has drivers**.

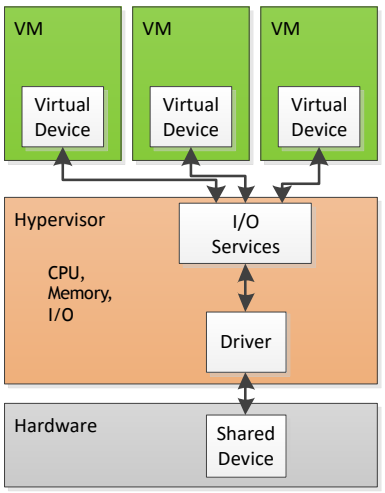


Figure 29: Monolithic architecture.

– **Microkernel architecture**. Device **drivers run within a service virtual machine**.

Advantages

* **Smaller hypervisor**.
* **Leverages driver ecosystem** of an existing OS.
* Can **use third party driver** even if not always easy, recompiling might be required.

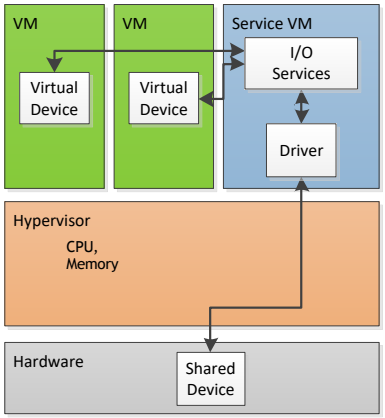


Figure 30: Microkernel architecture.

- **Type 2 Hypervisor** or **Hosted Hypervisor**. Hosted Hypervisors **run within the operating system of the host machine**. Although hosted hypervisors run within the OS, additional operating systems can be installed on top of it. Hosted hypervisors have higher latency than bare metal hypervisors because requests between the hardware and the hypervisor must pass through the extra layer of the OS.

  This type of hypervisor is also called *hosted hypervisor*. Furthermore, the *Guest OS* is the one that runs in the VM, while applications run in the *Guest 0S*. The ***Host OS* controls the hardware of the system**.

  Advantages

  – **More flexible** in terms of underlying hardware.
  – **Simpler** to manage and configure.

  Disadvantages

  – The *Host OS* might **consume** a non-negligible set of physical **resources**.

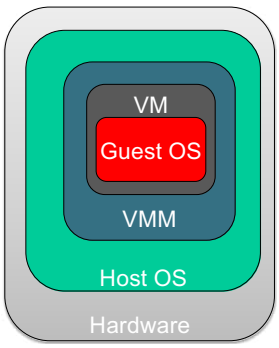


Figure 31: View of the Hosted Hypervisor.

The data is taken from VMware.

3.1.3.1 Full virtualization

Full virtualization is a virtualization technique that **provides a complete simulation of the underlying hardware**.

In full virtualization, the **original operating system runs without knowing it's virtualized**, using translation to handle system calls.

In full virtualization, the **virtual machine completely isolates the guest OS from the virtualization layer and hardware**.

Advantages

- **Running unmodified OS**.
- **Complete isolation**.

Disadvantages

- **Performance**.
- **Hypervisor mediation** to allow the guest (or guests) and host to request and acknowledge tasks which would otherwise be executed in the virtual domain.
- **Not allowed on all architectures**.

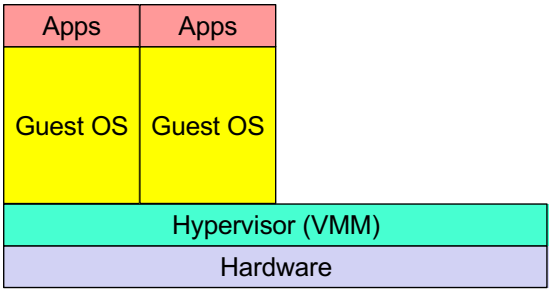


Figure 32: View of the Full Virtualization.

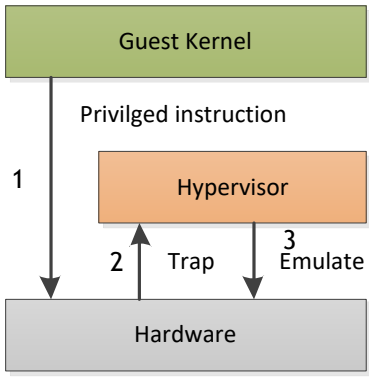


Figure 33: Full Virtualization flow.

3.1.3.2 Paravirtualization

Paravirtualization modifies the OS to use hypercalls instead of certain instructions, making the process more efficient but requiring changes before compiling.

Paravirtualization is a virtualization technique that uses hypercalls for operations to handle instructions at compile time. In paravirtualization, *guest OS* is not completely isolated but it is partially isolated by the virtual machine from the virtualization layer and hardware. VMware and Xen are some examples of paravirtualization.

Guest OS and VMM collaborates:

- VMM present to VMs an interface similar but not identical to that of the underlying hardware.
- To reduce guest's executions of tasks too expensive for the virtualized environment (by means of **"hooks" to allow the guest(s) and host to request and acknowledge tasks** which would otherwise be executed in the virtual domain, where execution performance is worse).

Advantages

- **Simpler VMM**.
- **High Performance**.

Disadvantages

- **Modified Guest OS** (cannot be used with traditional OSs).

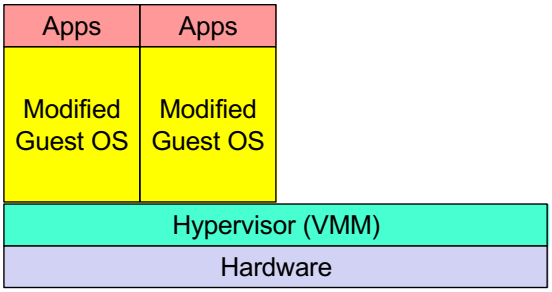


Figure 34: View of the Paravirtualization.

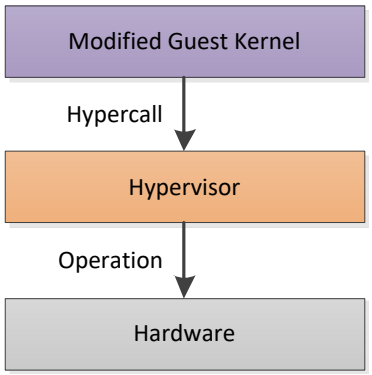


Figure 35: Paravirtualization flow.

3.1.3.3 Containers

Containers are **pre-configured packages**, with everything we need to execute the code (code, libraries, variables and configurations) in the target machine. Some well-known containers are Docker and Kubernetes.

The **main advantage** of containers is that their **behavior is predictable**, **repeatable** and **immutable**. When we create a "*master*" container and duplicate it on another server, we know exactly how it will be executed. There are **no unexpected errors when moving it to a new machine or between environments**.

Example 1

If we have a container for a website, we do not need to export/import the dev/testing/production environments. We just create a container containing the site and move it to the target environment.

Virtual machine provides hardware virtualization, while **containers provide virtualization at the operating system level**. The main difference is that the **containers share the host system kernel with other containers**.

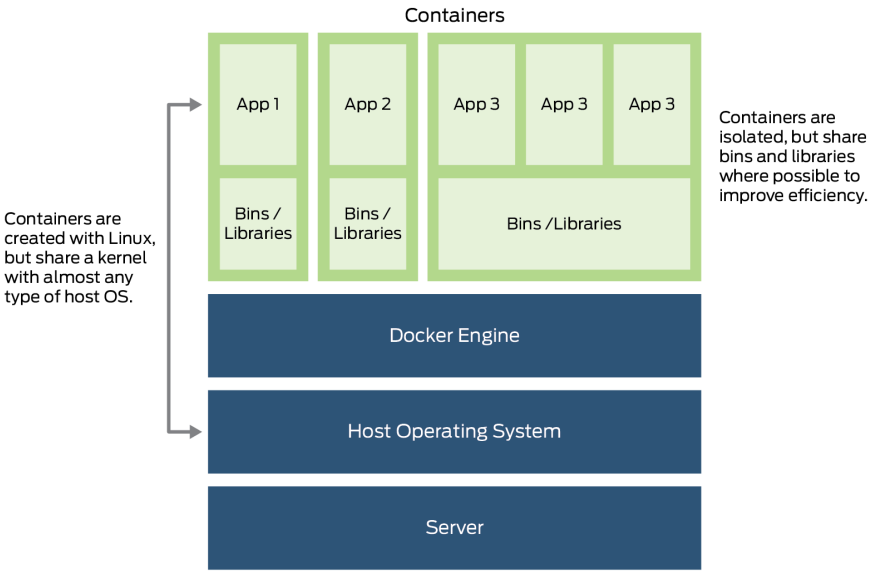


Figure 36: Containers architecture.

Some characteristics of containers:

- **Flexible**. Even the most complex application can be containerized.
- **Light**. The containers exploit and share the host kernel.
- **Interchangeable**. Updates can be distributed on the fly.
- **Portable**. We can create locally, deploy in the cloud and run anywhere.

- **Scalable**. It is possible to automatically increase and distribute replicas of the container.
- **Stackable**. Containers can be stacked vertically and on the fly.

Containers ease the deployment of applications and increase the scalability but they also impose a **modular application development where the modules are independent and uncoupled**.

Container Use Cases

- Make our **local development** and **build workflow faster**, more **efficient** and **lighter**.
- Run **standalone services and applications** consistently **across multiple environments**.
- Use **containers to create isolated instances to run tests**. For example, to create a db server SQL already configured to run tests.
- **Build and test complex applications and architectures on a local host** before deploying to a production environment.
- **Build a multi-user Platform-as-a-Service (PaaS) infrastructure**.
- Provide lightweight, stand-alone sandbox environments for developing, testing and teaching technologies such as the Unix shell or a programming language.
- **Software as a Service (SaaS) applications**.

3.2 Computing Architectures

3.2.1 Cloud Computing

Cloud Computing is a coherent, large-scale, publicly accessible **collection of computing, storage and networking resources**. It is usually available via web service calls over the Internet. The business is based on short or long term access on a pay-per-use basis.

The **cloud is implemented through virtualization**. This involves **partitioning hardware resources** (CPU, RAM, etc.) **and sharing them between multiple virtual machines** (VMs). This model ensures performance isolation and security. The Virtual Machine Monitor (VMM) manages access to physical resources between running VMs. The pros and cons of virtualization are explained in the dedicated chapter 3.1 (page 107).

3.2.1.1 Server Consolidation

Server Consolidation in cloud computing refers to the **process of combining multiple servers into a single, more powerful server or cluster of servers**.

This can be done to **improve the efficiency and cost effectiveness of the cloud computing environment**.

Server Consolidation is typically **achieved through the use of virtualization technology**, which allows **multiple virtual servers to run on a single physical server**. This enables better utilization of resources, as well as improved scalability and flexibility. It also allows organizations to reduce the number of physical servers they need to maintain, which can lead to cost savings on hardware, power and cooling.

In the context of server consolidation, **Consolidation Management** is the **migration from physical to virtual machines**.

Let us just say that the **characteristics** of this technique are:

- **Scalability**. It is possible to **move VMs without disrupting the applications running in them**.
- **Automatic scalability**. In addition to scalability, it is also possible to **automatically balance workloads according to set limits and guarantees**.
- **High availability**. Servers and applications are **protected against component and system failures**.

Server Consolidation Advantages

The benefits of server consolidation are:

- **Different operating systems can run on the same hardware**.
- Higher hardware utilization means **less hardware is needed** (**lower acquisition** and **maintenance costs**).
- Continued use of legacy software.
- **Application independent of hardware**.

3.2.1.2 Services provided by cloud

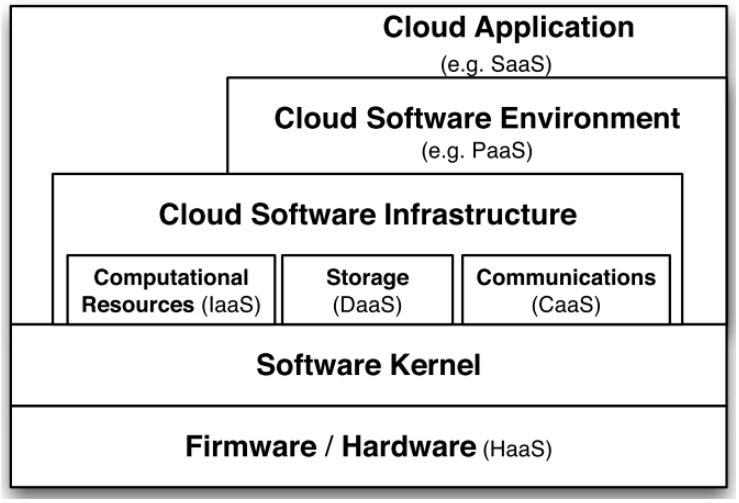
Cloud Computing is a **model for** providing convenient, **on-demand network access to a shared pool of configurable computing resources**, such as networks, servers, storage, applications and services. Anything that can be **quickly provisioned and released with minimal management** or interaction with the service provider.

"**X as a service**" (rendered as `*aaS` in acronyms) is a phrasal template for any business model in which a product use is offered as a subscription-based service rather than as an artifact owned and maintained by the customer. [3]

There is a wide variety of "**as-a-Service**" terms used to describe services offered in clouds.

- AaaS - Architecture as a Service
- BaaS - Business as a Service
- CaaS - Communication as a Service
- CRMaaS - CRM as a Service
- DaaS - Data as a Service
- DBaaS - Database as a Service
- EaaS - Ethernet as a Service
- FaaS - Frameworks/Function as a Service
- GaaS - Globalization/Governance as a Service
- HaaS - Hardware as a Service
- IaaS - Infrastructure/Integration as a Service
- IDaaS - Identity as a Service
- ITaaS - IT as a Service
- LaaS - Lending as a Service
- MaaS - Mashups as a Service
- OaaS - Organization/Operations as a Service
- SaaS - Software as a Service
- StaaS - Storage as a Service
- PaaS - Platform as a Service
- TaaS - Technology/Testing as a Service
- VaaS - Voice as a Service

The **main services** provided by the cloud are shown in the image below.



- **Cloud Application Layer**: SaaS (Software as a Service). **Users access the services provided by this layer through web portals and may be charged for using them**.

  Cloud applications can be developed on the cloud software environments or infrastructure components.

  Some **examples** include Gmail, Webex, Google Docs.

- **Cloud Software Environment Layer**: PaaS (Platform as a Service). Users are **application developers**.

  Vendors provide developers with a **programming language level environment with a well-defined API**. It has many advantages:

  - Ease the interaction between the environment and applications.
  - Accelerate deployment.
  - Support scalability.

  Some **examples** are Amazon Lambda and Microsoft Azure.

- **Cloud Software Infrastructure Layer**: provides resources to the higher-level layers.

  - **Computational Resources Layer**: IaaS (Infrastructure as a Service). The vendor provides **virtual machines to developers**. Then the pros and cons are related to the virtual machines.

    Advantages

    * Flexibility.
    * Root access to the virtual machine to fine-tune settings and customize installed software.

Disadvantages

* Performance impact.
* Inability to provide strong SLA guarantees.

Some examples are Amazon EC2, Google Compute Engine, IBM Cloud.

– **Storage Layer**: DaaS (Data as a Service). This layer allows users to:

1. **Store** their **data** on remote disks.
2. **Access data from anywhere at any time**.

It allows cloud applications to scale beyond their limited server requirements:

* High Reliability: Availability, Reliability, Performance.
* Replication.
* Data consistency.

Some examples are DropBox or GoogleDrive.

– **Communications Layer**: CaaS (Communication as a Service). This layer guarantees:

* **Communication capability**: service-oriented, configurable, plannable, predictable, and reliable.
* Network security, dynamic provisioning of virtual overlays for traffic isolation or dedicated bandwidth, guaranteed message delay, communication encryption, and network monitoring.

3.2.1.3 Types of clouds

There are 4 types of clouds:

- A **Public Cloud** is one that is made available to the public by a specific organization that also hosts the service.

  A third-party provider maintains the hardware, relevant software, and licenses in a globally distributed network of data centers. You can access exactly what you need on demand, at any scale, from any device you choose.

  The providers have a large infrastructure available on a rental basis and provide full customer self-service. Accountability is based on e-commerce.

- A **Private Cloud** is used for a single organization and can be hosted internally or externally.

  Internally managed data centers. The **organization sets up a virtualization environment on its own servers**: in its own data center, in the data center of a managed service provider.

  Advantages

  - We have **total control over every aspect** of the infrastructure.
  - We get the benefits of virtualization.

  Disadvantages

  - It lacks the freedom from capital investment and flexibility.

- A **Community Cloud** is a type of cloud shared by multiple organizations; typically hosted externally, but can also be hosted internally by one of the organizations.

  A single cloud managed by multiple federated organizations that combine multiple organizations allows for economies of scale, and resources can be shared and used by one organization while the others are not using them.

  Technically, it is similar to a private cloud because they share the same software and the same problems, but it requires a more complex accounting system.

  Hosted locally or externally. Typically, community clouds share infrastructure between participants. However, they may be hosted by a separate, dedicated organization or by only a small subset of partners.

- A **Hybrid Cloud** is a type of cloud that is a composition of two or more clouds (private, community, or public) that remain unique entities but are interconnected to provide the benefits of multiple deployment models; hosted internally and externally.

  Hybrid clouds are a combination of any of the previous types. Typically, companies are keeping their private cloud, but that they may be subject to unpredictable peaks of load. In this case, the company rents resources from other types of cloud.

4 Methods

4.1 Reliability and availability of data centers

4.1.1 Introduction

Dependability is a **comprehensive measure** of how much we can **trust a system** to perform its expected function correctly and consistently over time.

Dependability is not a single property but a **composite of multiple interrelated concerns**, some more related to system correctness (reliability, availability), others more to protection (safety, security). In large-scale systems like datacenters, **dependability is not optimal**, it is integral to Service Level Agreement (SLAs)[8], customer trust, and legal compliance (e.g., data protection laws)[9].

Core Attributes of Dependability

Attribute	Meaning	Analogy / Data Center View
Reliability	Continuity of correct service	A web server not crashing or corrupting data for long periods.
Availability	Readiness for correct service	Can the system serve requests *right now*?
Maintainability	Ability for easy maintenance	Hot-swappable disks, modular servers.
Safety	Absence of catastrophic consequences	E.g., fail-safes in a power system to avoid fire/electrical hazard.
Security	Confidentiality and integrity of data	Access control, data encryption, intrusion prevention.

Example 1: Cloud Service Provider

Image Google Cloud or AWS:

- **Reliability**: Our VMs shouldn't crash without reason.
- **Availability**: We must access our cloud database anytime.
- **Maintainability**: They update the hypervisor with zero downtime.
- **Safety**: If a cooling system fails, automatic shutdown prevents hardware fire.

[8] A **Service Level Agreement (SLA)** is a contract between a service provider and a customer that defines the expected level of service. It typically includes metrics like availability, performance, and response times, along with remedies if the provider fails to meet those standards.

[9] **Legal compliance** in data protection laws ensures that organizations handle personal data responsibly and in accordance with regulations. In the EU, the *General Data Protection Regulation (GDPR)* is the primary framework, setting strict rules on data collection, processing, and storage. Globally, different countries have their own laws, such as the *California Consumer Privacy Act (CCPA)* in the U.S., and similar regulations emerging worldwide.

- **Security**: Data encryption at rest and in transit, isolation across tenants.

Do we really need to care about Dependability?

First, we'll explore intuition. Even when a system is *well-designed*, passes *functional verification*, meets *performance and energy constraints*, it can still **fail**! Why? Because real-world conditions and **non-functional factors** can break even perfectly specified systems.

So dependability is necessary because **failures are inevitable**. They stem from **multiple layers of system stack**, including:

Source	Example
Hardware aging	Transistor degradation, wear-out mechanisms.
Manufacturing defects	Process variation, cosmic rays.
Design flaws	Logic errors, incorrect specifications.
Software bugs	Memory leaks, race conditions, kernel panics.
System misconfiguration	Wrong firewall rule, VM setting.
External disturbances	Electromagnetic interference, overheating.
Human errors	Mistakes by admins, developers.
Malicious attacks	DoS, ransomware, data exfiltration.

Even a tiny misbehavior in one layer can cascade through the entire system.

Consequences of Undependability

Let's categorize **why lack of dependability is dangerous**, especially in data centers:

1. **High Economic Cost**: Downtime may cause lost transactions, SLA violations, brand damage.
2. **Information Loss**: Permanent deletion or corruption of critical data (e.g., backups overwritten, logs lost).
3. **Cascade Failures**: One component failure affects dependent services (e.g., cloud outages).
4. **User Distrust**: If users don't trust the service, they abandon it.
5. **Security & Safety Hazards**: leaked credentials, incorrect data processing and in physical systems, risk to human life (think healthcare or industrial systems).

When to Think About Dependability?

Short answer: **Always**. Dependability is **not an optional**, it must be **considered** throughout the **entire system lifecycle**: *design-time*, *deployment*, *run-time*, and even *decommissioning*. We think about dependability:

- At **Design-Time**, the main goal is to **prevent as many failures as possible before the system is deployed**. This is when we:
  - **Analyze**: Model the system and predict how it behaves under faults.
  - **Measure**: Evaluate expected MTTF, availability, error rates.
  - **Optimize**: Adjust design to improve weak points (e.g., redundancy, replication).
  - **Verify**: Perform fault-injection, static analysis, formal verification.
- At **Run-Time**, no matter how good our design is, failures will still happen in the field. So **we must**:
  - Detect: **Monitor system** state to spot abnormal behavior.
  - Diagnose: **Understand** root **causes** (e.g., logs, telemetry).
  - React: **Automatically correct** or **isolate** the problem.

  For example rebooting a crashed VM, replacing a failed disk in a RAID array or routing traffic away from an unhealthy load balancer.

Why Distinguish Development-Time vs Operation-Time Failures?

These two types of failures have completely different roots, so we must use different strategies to handle them.

- **Development-Time Failures**. These are **errors that are introduced during system creation**, in the design, coding or configuration phase.

  Examples
  - A bug in the file system code that causes data corruption.
  - A wrong assumption in system specifications.
  - An incorrect server configuration (e.g., firewall blocking valid ports).
  - A logic error in redundancy management (e.g., mirrored disk not syncing).

  Solutions
  - These **should be avoided** before deployment.
  - They're addressed using:
    - **Testing** (unit, integration, fault-injection)
    - **Verification** (formal methods, code reviews)
    - **Validation** (does it meet real-world expectations?)

  If a system fails due to a development-time error, it means there was something **wrong in how it was built**.

- **Operation-Time Failures**. These are **failures that happen while the system is running**, even if the system was perfectly designed.

  Examples

  * A hard disk physically fails after 3 years.
  * A memory bit flips due to cosmic radiation (soft error).
  * The power supply overheats due to dust.
  * A data center operator accidentally disconnects a cable.
  * A hacker exploits an unknown vulnerability.

  Solutions

  * These **cannot be entirely avoided**, because hardware ages, users make mistakes, environments are unpredictable.
  * They **must be handled at runtime** with:
    - **Monitoring**
    - **Redundancy / Fault tolerance**
    - **Error correction & self-repair**

  If a system fails at runtime, it's not necessarily badly designed, **reality is imperfect**, and we need to plan for that.

Where to Apply Dependability?

Originally, dependability was essential only in a few extreme environments, systems where a failure could endanger lives, waste millions or billions, be unacceptable by definition (space, nuclear, aviation). The types of Critical Systems:

- **Mission-Critical Systems**. The **mission** is **not completed if a failure occurs**. Some examples are satellites (loss = mission failure), unmanned vehicles (e.g., drones, crash = loss of platform and data). **Dependability ensures the system fulfills its intended goal**.
- **Safety-Critical Systems**. If a failure occurs, human life is at risk. Some examples are aircraft control (system crash = airplane crash), nuclear plant controls (meltdown). Here, the **cost of failure is catastrophic**.

Dependability today is no longer limited to space and nuclear systems. Now it's **central to computing infrastructures**, including data centers, cloud platforms, edge devices, etc. Why? Because a failure today in these systems can disrupt global services (e.g., Gmail outage), leak personal data, etc.

How to Provide Dependability?

Provide Dependability means designing our system (hardware and software) in a way that it **keeps working**, even when **things go wrong**, or at least, it fails in a **safe and recoverable way**. There are **two main strategies** to achieve this:

1. **Failure Avoidance** (try not to fail). This approach is like saying: "*let me build the system so well that it doesn't break*". For example, use very reliable hardware, run a lot of testing before deploying our software, do quality control in factories to catch early hardware faults, and use formally verified algorithms in critical parts.

   This approach is **good**, **but expensive**, and we **can't predict everything**.

   Example 2: Failure Avoidance Analogy

   Like buying a car and checking *everything* before leaving the factory: brakes, oil, tires, engine. So it's unlikely to break.

2. **Failure Tolerance** (prepare for when it does fail). This approach accepts the fact that something **will** go wrong. So let us prepare the system to **detect**, **contain**, and **recover** from the problem.

   For example, use RAID for disks, use ECC memory (it can correct small memory errors automatically), monitor servers, and in the cloud, if one data center goes down, reroute to another.

   This is **essential in data centers**, because with thousands of machines, failures are guaranteed.

   Example 3: Failure Tolerance Analogy

   Like having a spare tire in our car, a seatbelt, and road assistance. We're ready when things go wrong.

Where Can We Add Dependability? We can "*add dependability*" at **different layers**:

- **Hardware Level**. Make components themselves very reliable.
  - ✔ Use robust materials.
  - ✔ Use parts that are known to last a long time.
  - ✘ Expensive, and maybe slow (old, stable tech).
- **Architecture/System Design Level**. Make the **structure of the system** fault-tolerant.
  - ✔ Duplicate components (e.g., 2 power supplies, RAID).
  - ✔ Create **clusters**: if one server fails, others can take over
  - ✘ More machines, more space, higher cost.

- **Software Level**. Add dependability using **code**.
  - ✔ Detect problems: logs, alerts, health checks.
  - ✔ Retry or restart automatically.
  - ✔ Recover from failures (e.g., checkpoint/restart in HPC).
  - ✔ Cheaper and flexible.
  - ✖ Needs smart design and might slow down performance.

⚠ There ain't no such thing as a free lunch (TANSTAAFL)

It is impossible to get something for nothing. We always **pay** for dependability:

We Pay In...	Why?
Money	Redundant servers, better hardware.
Time	Development of testing and recovery logic.
Performance	Extra steps for safety and monitoring.

4.1.2 Reliability and Availability

Definition 1: Reliability

Reliability is the probability that a system or component performs its **required functions**, under **stated conditions**, for a **specified period of time** without failure.

According to IEEE Std 610.12-1990: "*The ability of a system or component to perform its required functions under stated conditions for a specified period of time.*"

Mathematical Definition

Let $R(t)$ be the reliability function, representing the **probability that the system has not failed during the time interval** $[0, t]$:

$$R(t) = P\left(\text{No failure during } [0,t]\right) \quad \text{given system was operational at } t = 0 \tag{20}$$

- t is the time. Reliability always **depends on the target time window** t. For example, a system may be reliable over 10 minutes, but not over 10 hours. In critical systems (like avionics or nuclear control), **even a short failure** is unacceptable, this makes **reliability extremely stringent**.
- It is a **non-increasing** function over time.
- $R(0) = 1$: at time 0, the system is assumed to be *working*.
- $R(t) = 0$ as $t \to \infty$: over infinite time, failure becomes certain.

We can also determine **Unreliability** given the reliability function. Let $Q(t)$ be the unreliability function:

$$Q(t) = 1 - R(t) \tag{21}$$

This express the probability that the system **has failed** by time t.

Typical Behavior

The failure rate $\lambda(t)$ may vary over the life of a component:

- **Early stage**: high initial failure due to manufacturing defects.
- **Mid-life**: constant or random failure rate, this is the region of interest in most reliability modeling.
- **End-of-life**: failure rate increases due to wear-out (aging, stress).

Definition 2: Availability

Availability is the probability that a system is **operational and accessible** when it is required to be used.
It represents how much of the time a system is **actually up and running**, accounting for both its reliability and its ability to **recover from failures**.

According to IEEE Std 610.12-1990: "*The degree to which a system or component is operational and accessible when required for use.*"

While **reliability** focuses on *continuous correct operation*, **availability** emphasizes the ***readiness for use*** at a given moment, even after failures occur and are repaired.

√× Mathematical Definition

For **steady-state availability**:

$$\text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}} \tag{22}$$

Formally, $A(t)$ is the probability that the system is **operational** (not failed) at a given time t:

$$A(t) = P(\text{System is working at time } t) \tag{23}$$

In other words, it's the probability that the system is **ready for service** at time t.

- **Non-Repairable Systems**. If the system **cannot be repaired**, then:

  $$A(t) = R(t)$$

  The availability equals the reliability (it will eventually fail and stay failed).

- **Repairable**. For **repairable systems**:

  $$A(t) \geq R(t)$$

  Because even if the system fails, it may be restored and become available again.

We can also determine **Unavailability** given the availability function:

$$\text{Probability that the system is not operational at time } t = 1 - A(t) \tag{24}$$

⚖ Difference from Reliability

Concept	Reliability	Availability
Meaning	No failure in a given time	System is up when needed
Failure OK?	✖ No, failure terminates reliability	✔ Yes, if system recovers quickly
Focus	Time-to-failure	Service continuity
Measurement	Probability of survival	Fraction of time operational
Example	Pacemaker, airplane engine	Cloud server, ATM, database system

A common shorthand for availability levels:

Number of 9s	Avail. (%)	Max Downtime	Typical Applications
2 nines	99.00%	≈ 3.65 days	Home internet, non-critical sites
3 nines	99.90%	≈ 8.76 hours	Web services, small business apps
4 nines	99.99%	≈ 52.6 minutes	Enterprise systems
5 nines	99.999%	≈ 5.26 minutes	Telco switches, high-end data centers
6 nines	99.9999%	≈ 31.5 seconds	Mission-critical infra (e.g. finance)

Reliability & Availability: Related Indices

These indices are used to **quantify** how reliable and/or available a system is over time.

1. **Mean Time To Failure (MTTF)**

   Definition. The **average time** that a system operates before experiencing the **first failure**.

   $$\text{MTTF} = \int_0^\infty R(t)\,\mathrm{d}t \tag{25}$$

   Used for: Non-repairable systems or to model **time to first failure** in repairable systems.

2. **Mean Time Between Failures (MTBF)**

   Definition. The **average time** between **two consecutive failures**, including the time to repair.

   $$\text{MTBF} = \text{MTTF} + \text{MTTR} \tag{26}$$

   Where MTTR is the Mean Time To Repair. Sometimes, **repair time is negligible**, so MTBF $\approx$ MTTF.

   Used for: Repairable systems.

3. **Failures In Time (FIT)**

   Definition. The number of expected failures **per billion** (10^9) **hours of operation**.

   $$\text{MTBF (in hours)} = \frac{10^9}{\text{FIT}} \tag{27}$$

   Used for: Common in hardware reliability (e.g., CPU/SSD specs).

4. **Failure Rate λ**. For exponential failure distributions:

   $$R(t) = e^{-\lambda t} \quad \Rightarrow \quad \lambda = \frac{1}{\text{MTTF}} \tag{28}$$

   It gives the **instantaneous likelihood** of failure per unit time.

How to Compute Reliability? Empirical Evaluation

Let's suppose we want to estimate reliability based on **real-world observations** (e.g. testing 1000 SSDs in a lab). Here's how:

1. **Initial Setup**. Deploy n_0 **independent and statistically identical elements** at time $t = 0$. All start in identical operating conditions: $n(0) = n_0$.
2. **Failure Observation**. Over time, observe the **number of elements still working** at time t, denoted as $n(t)$. For each unit i, record its **time to failure**: $t_1, t_2, \ldots, t_{n_0}$.
3. **Assumption**. Times to failure τ_i are **independent realizations** of a random variable τ.
4. **Empirical Reliability Function**

$$R(t) \approx \frac{n(t)}{n_0} \tag{29}$$

It estimates the **probability** that a system is still operational at time t. As the number of samples $n_0 \to \infty$, this ratio converges to the true reliability function:

$$\frac{n(t)}{n_0} \longrightarrow R(t) \tag{30}$$

This is useful for **empirical validation** of theoretical models (like exponential reliability). Often used in *burn-in testing*, *field reports*, and *accelerated life tests*.

Example 4

Suppose we start testing 1000 servers at time $t = 0$. After 3000 hours, only 980 are still working:

$$R(3000) \approx \frac{980}{1000} = 0.98$$

We repeat this for multiple t-values to plot the empirical reliability curve.

What to do with $R(t)$? And why compute $R(t)$?

The **goal** is to determine the **expected reliability** (or failure behavior) of **complex systems**, such as a data center or a warehouse-scale machine, **over time**. This includes:

1. **Computing MTTF (Mean Time To Failure)**. Once we know the reliability function $R(t)$, we can compute:

$$\text{MTTF} = \int_0^\infty R(t)\,\mathrm{d}t$$

This gives the **expected lifetime** of a component or system before it fails. It is essential for predictive maintenance, capacity planning and warranty.

2. **Composing System Reliability from Components**. Computation of the overall reliability starting from the components' one. Systems are usually composed of multiple **subsystems** (e.g. disks, servers, switches). Each component has its own reliability $R_i(t)$. Using **Reliability Block Diagrams** or other composition models, we derive:

$$R_{\text{system}}(t) = f\left(R_1(t), R_2(t), \ldots, R_n(t)\right)$$

3. **Exponential Reliability Assumption**. We consider only a reliability with an exponential distribution: constant failure rate. This simplifies all analysis:

$$R(t) = e^{-\lambda t}$$

Constant failure rate λ, then memoryless behavior.

What does "memoryless" mean? In probability theory, a **memoryless distribution** is one where: *the probability that a system* ***survives an additional time*** *s does* ***not depend*** *on how long it has already survived.* Mathematically, for the **exponential distribution**:

$$\mathbb{P}\left(T > t + s \mid T > t\right) = \mathbb{P}\left(T > s\right)$$

This is what makes it *memoryless*: **the past does not matter**.

Why does constant λ imply memorylessness? If $R(t) = e^{-\lambda t}$, then:

- The **failure rate** λ is constant.
- The distribution of time-to-failure T is exponential.

This means that a component that has been running for 1000 hours is **just as likely to fail in the next 100 hours** as a brand-new one.

Analogy Image a light bulb modeled with exponential failure. We don't get closer to failure as time passes. Whether it's 10 hours old or 1000 hours old, its probability of lasting the next hour is always the same.

Common for modeling random failure phase (middle or bathtub curve). Makes MTTF easy to compute:

$$\text{MTTF} = \frac{1}{\lambda} \tag{31}$$

Reliability Terminology

1. **Fault**: a detect of flaw in te system, either in hardware or software. It is **passive** and may or may not lead to an error. For example, a fault occurs when an electromagnetic disturbance corrupts a radar signal of a drone.
2. **Error**: a deviation from the correct internal state of the system caused by a fault. An **error is active**, it is manifestation of the fault. For example, an error occurs when the wrong trajectory is computed for a drone based on faulty sensor input.
3. **Failure**: the **external deviation** from the expected service or behavior. It is **observable** by the user or system and may cause damage or interruption. A failure occurs, for example, when a drone crashes to the ground.

These stages form a chain:

$$\text{Fault} \rightarrow \text{Error} \rightarrow \text{Failure}$$

But the chain doesn't **always complete**. There are **two special cases**:

- ✔ **Non-Activated Fault**: a **fault exists** but never triggers an error.

  Example 5: Non-Activated Fault

  A server in a cloud data center has a memory module with a **latent hardware fault**: a single DRAM cell is defective due to a manufacturing imperfection.

  - ⚠ **Fault**: A memory cell in a non-critical area of RAM has a physical defect (e.g., stuck-at-1).
  - ✔ **Error**: No error occurs because that specific memory address has **never been written or read**. Or, the OS allocates memory dynamically and that region is **never touched**.
  - ✔ **Failure**: No incorrect behavior is ever observed by any application or user. The system continues operating normally, and the fault remains **inactive**, never triggered.

  The **fault exists physically**, but it **never becomes active**, and thus causes **no error** or **failure**. This is a classic non-activated fault: it's a dormant effect that could lead to problems only under specific usage or workloads, but never does in the observed execution.

- ✔ **Non-Propagated (Absorbed) Error**: an error occurs but is **detected or corrected** before causing a failure. For example, if a drone calculates the wrong trajectory, but this is later corrected using radar data, this constitutes a safe landing[10] behavior and an instance of absorbed error.

[10] A **safe landing** is the **correct expected behavior** of the drone, the absence of failure, despite a temporary internal error.

Fault Hierarchy

The **Fault-Error-Failure** chain is **not flat or isolated**, it can cascade across components and abstraction levels, and grow into **critical or life-threatening failures**.

A **Fault Hierarchy** is a conceptual model that captures how **faults can propagate** and **escalate** through layers of a system, for example:

- **Low-level hardware fault**: a transistor in a CPU flips and causes a memory bitflip.
- **Mid-level software error**: the corrupted value leads to an incorrect instruction.
- **High-level system failure**: the OS kernel crashes and the entire system reboot.
- **Service-level failure**: web service becomes unavailable to users.
- **Mission-level hazard**: in critical systems, loss of control, safety violations, or even physical harm.

Example 6: Fault Hierarchy

Let's say we're dealing with an autonomous surgical robot:

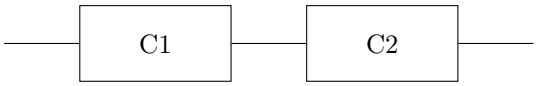
- **Fault**: a memory bitflip due to cosmic radiation (hardware).
- **Error**: the control software computes the wrong joint angle (software logic).
- **Failure**: the robot arm moves incorrectly (physical layer).
- **Hazard**: causes injury to the patient (mission/life-threatening).

Why it matters? **Critical systems** (aviation, medical, autonomous driving) must **block this chain early**, ideally at the fault or error stage. This is the basis for fault tolerance, error detection and correction, and failure containment and recovery.

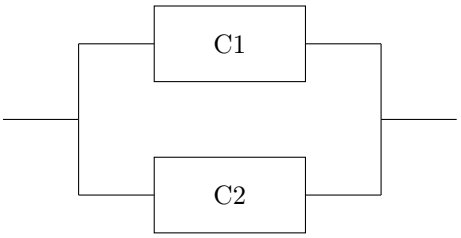
4.1.3 Reliability Block Diagrams (RBDs)

As we saw in the Software Engineering for HPC course, a **Reliability Block Diagram (RBD)** is a *graphical and mathematical model* used to analyze the **reliability of a system** by representing how the success or failure of each component affects the overall system operation.

- **Inductive Model**: Starts from the reliability of individual components and builds up to the system.
- **Blocks**: Each block represents a **component or subsystem**, each with its own reliability function $R_i(t)$.
- **Connections**
  - **Series**: All components must function for the system to work.



  - **Parallel**: The system can work as long as at least one component works.



- **Model Topology**: The RBD structure **may not match the physical system layout**, as it's focused on *functional success paths* rather than actual wiring or data flow.

Every element in the RBD has its **own reliability**. Blocks are combined together to model all possible success paths.

Mathematical Concepts

- **Series Configuration**. In a **series system**, **all components must work** for the system to function. If **any single component fails**, the **whole system fails**. In other words, the system failure is determined by the failure of the *first* component.

  The formula is:

$$R_S(t) = R_{C1}(t) \cdot R_{C2}(t) \cdot \cdots \cdot R_{Cn}(t) = \prod_{i=1}^{n} R_i(t) \tag{32}$$

  Where:

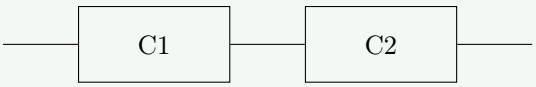
  - $R_S(t)$: system reliability at time t.
  - $R_i(t)$: reliability of the i^{th} component.

– This assumes **independent** component failures.

If each component has a **probability** $R_i(t)$ of working, then the **joint probability** that **all work** is the **product**.

Example 7: Series Configuration

Consider, for example, the following model:



The reliability is: $R_S(t) = R_{C1}(t) \cdot R_{C2}(t)$.

- **Parallel Configuration**. In a **parallel system**, the system functions as long as **at least one component works**. This configuration is used for **redundancy**. In other words, the system fails when the *last* component fails.

  The formula is:

$$R_S(t) = 1 - \prod_{i=1}^{n} [1 - R_i(t)] \tag{33}$$

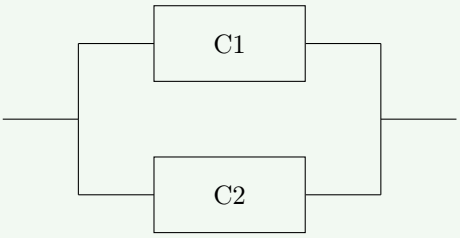
  Where:

  * $1 - R_i(t)$: failure probability of component i.
  * $\prod_{i=1}^{n} [1 - R_i(t)]$: probability that **all fail**.
  * $1 -$ (all fail): probability that **at least one works**.

  In summary, the system works in a parallel configuration **as long as at least one component is functioning**. This **design increases fault tolerance and improves reliability**.

Example 8: Parallel Configuration

Consider, for example, the following model:



The reliability is: $R_S(t) = R_{C1}(t) + R_{C2}(t) - R_{C1}(t) \cdot R_{C2}(t)$.

Other Series Formulas Related to Reliability

- **Reliability Function of Series Systems as one Exponential Component λ_S.** In a **series system**, all components must function for the system to work. If one fails, the system fails. For n components with exponential reliability:

$$R_S(t) = \prod_{i=1}^{n} R_i(t) = \prod_{i=1}^{n} e^{-\lambda_i t} = \exp\left(-t \sum_{i=1}^{n} \lambda_i\right) \tag{34}$$

We define:

$$\lambda_S = \sum_{i=1}^{n} \lambda_i \quad \Longrightarrow \quad R_S(t) = e^{-\lambda_S t} \tag{35}$$

This models the **system as one exponential component** with rate λ_S.

- **Failure in Time (FIT).** The **failure rate** of the whole system in series is:

$$\lambda_S = \sum_{i=1}^{n} \lambda_i \tag{36}$$

This tells us that **each new component added in series increases the failure rate**, reducing overall reliability.

- **Mean Time To Failure (MTTF).** For an exponential distribution:

$$\text{MTTF}_S = \frac{1}{\lambda_S} = \frac{1}{\sum_{i=1}^{n} \lambda_i} = \frac{1}{\sum_{i=1}^{n} \frac{1}{\text{MTTF}_i}} \tag{37}$$

Thus, the more components in series, the lower the systems's MTTF. If even one component is fragile, it significantly impacts the whole system.

- **Special Case: Identical Components.** Assume n components and **same failure rate** λ, then:

$$\lambda_S = n \cdot \lambda \quad \Longrightarrow \quad R_S(t) = e^{-n\lambda t} \tag{38}$$

And:

$$\text{MTTF}_S = \frac{1}{n \cdot \lambda} \tag{39}$$

System becomes n **time less reliable** than a single component.

- **Availability of Series System.** Let $A_i = \frac{\text{MTTF}_i}{\text{MTTF}_i + \text{MTTR}_i}$ as the availability of component i, then:

$$A_S = \prod_{i=1}^{n} A_i \tag{40}$$

And if all components are identical:

$$A_S = A^n$$

Where:

$$A = \left(\frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}\right)^n$$

As with reliability, **availability degrades multiplicatively** in series configurations.

Other Parallel Formulas Related to Reliability

- **Reliability Function of Parallel Systems**. Let $R_i(t)$ be the reliability of component i. Then the system reliability is:

$$R_S(t) = 1 - \prod_{i=1}^{n} [1 - R_i(t)]$$

  - $1 - R_i(t)$: probability that component i has failed by time t.
  - $\prod(1 - R_i(t))$: probability that **all components** have failed.
  - So, 1 – (all failed): probability that **at least one** is working.

- **Special Case: Identical Components**. If all n components have the **same reliability** $R(t)$, then:

$$R_S(t) = 1 - (1 - R(t))^n \tag{41}$$

  As n increases, $R_S(t) \to 1$: very high system reliability.

- **Mean Time To Failure (MTTF)**. For parallel systems with exponential reliabilities $R_i(t) = e^{-\lambda_i t}$, there is **no simple closed-form MTTF** in the general case, but for identical components, if $R(t) = e^{-\lambda t}$, then:

$$R_S(t) = 1 - \left(1 - e^{-\lambda t}\right)^n \tag{42}$$

  MTTF can be found via integration:

$$\text{MTTF}_S = \int_0^\infty R_S(t)\, \mathrm{d}t \tag{43}$$

  This yields increasingly large values as n increases (more redundancy, then more robust).

- **Availability of Parallel System**. Let A_i be the availability of component i, then:

$$A_S = 1 - \prod_{i=1}^{n} (1 - A_i) \tag{44}$$

  If all components are the same:

$$A_S = 1 - (1 - A)^n \tag{45}$$

  Where:

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{46}$$

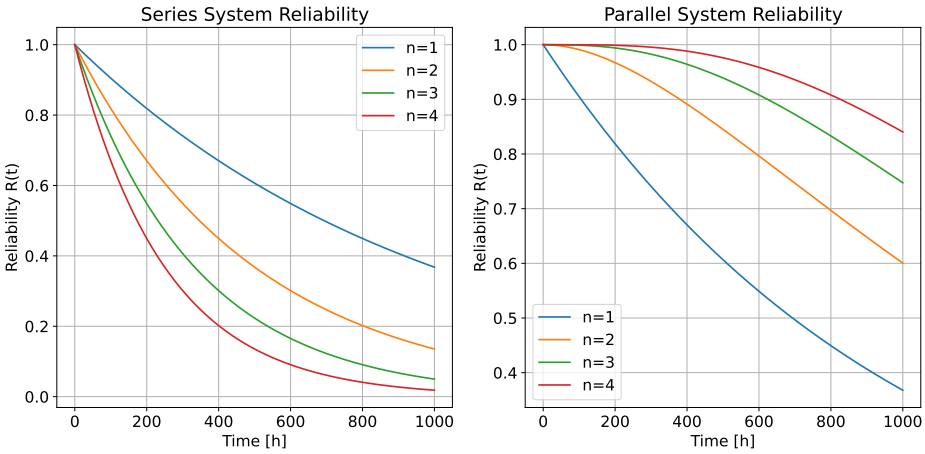


Figure 37: The **left plot** shows the series system reliability:

- Each line corresponds to a system composed of $n = 1, 2, 3, 4$ components in **series**.
- The x-axis is time (0 to 1000 hours).
- The y-axis is the system reliability $R_S(t)$, which **decreases over time**.

For $n = 1$, the system is just a single component, standard exponential decay. As n increases, reliability drops **faster** because the **probability of at least one failure increases**. Mathematically:

$$R_S(t) = e^{-n\lambda t}$$

For example, with $n = 4$, the system becomes much less reliable in shorter timeframes. This means that series systems are fragile; the more components there are, the greater the chance of failure. It's like a chain: the system is only as strong as its weakest link.
The **right plot** shows the parallel system reliability:

- Each curve is a system with $n = 1$ to $n = 4$ components in **parallel**.
- The **system works as long as at least one component works**.

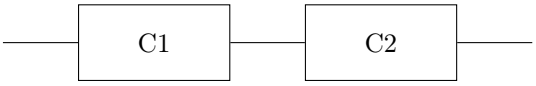
For $n = 1$, it's the same as the single-component series case. As n increases, reliability increases and curves stay higher for longer. Formula used:

$$R_S(t) = 1 - \left(1 - e^{-\lambda t}\right)^n$$

The system becomes more fault-tolerant because it tolerates some component failures. This means that parallelism boosts reliability. With just 3 or 4 redundant units, reliability stays close to 1 for long durations.

A quick recap

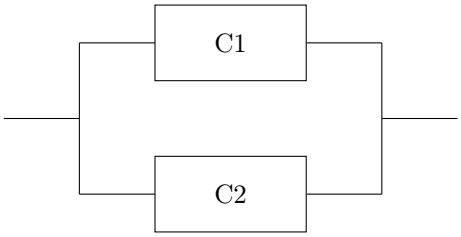
- **Series**



Reliability:

$$R_s = \prod_i^n R_i \Longrightarrow R_s = R_{C1} \cdot R_{C2}$$

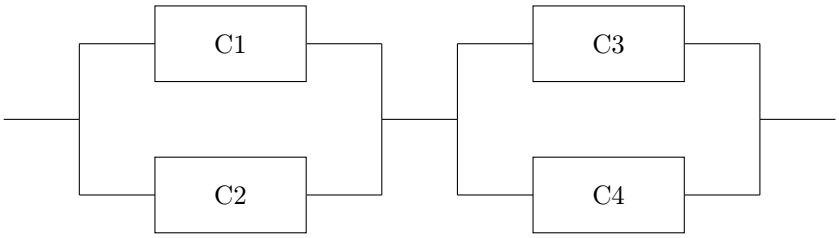
- **Parallel**



Reliability:

$$R_S(t) = R_{C1}(t) + R_{C2}(t) - R_{C1}(t) \cdot R_{C2}(t)$$

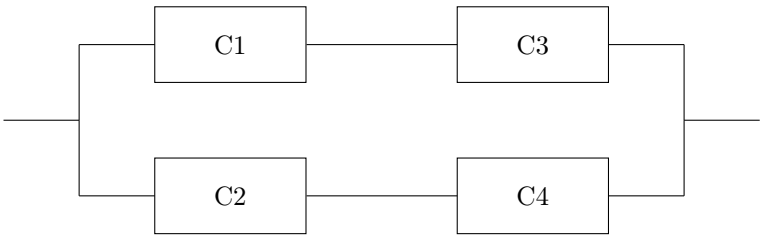
- **Series-Parallel** (**Component Redundancy**)



Reliability:

$$R_S(t) = (R_{C1} + R_{C2} - R_{C1} \cdot R_{C2}) \cdot (R_{C3} + R_{C4} - R_{C3} \cdot R_{C4})$$

- **Parallel-Series** (**System Redundancy**)



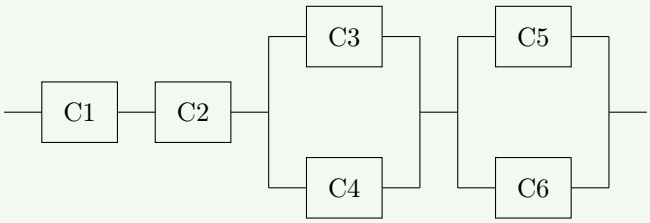
Reliability:

$$R_s = 1 - [(1 - R_{C1} \cdot R_{C3}) \cdot (1 - R_{C2} \cdot R_{C4})]$$

Example 9: calculate the reliability of the system

Question

What is the Reliability of the entire system knowing the reliability of each component?



- $R_{C1} = 0.95$
- $R_{C2} = 0.97$
- $R_{C3} = 0.99$
- $R_{C4} = 0.99$
- $R_{C5} = 0.92$
- $R_{C6} = 0.92$

Solution

1. Consider components $C1$ and $C2$. The reliability, which we will call R_G, is then calculated as a *series*:

$$R_G = R_{C1} \cdot R_{C2} = 0.95 \cdot 0.97 = 0.9215$$

2. Consider components $C3$ and $C4$. The reliability, which we will call R_H, is then calculated as a *parallel*:

$$\begin{aligned} R_H &= 1 - [(1 - R_{C3}) \cdot (1 - R_{C4})] \\ &= 1 - [(1 - 0.99) \cdot (1 - 0.99)] \\ &= 1 - 0.0001 \\ &= 0.9999 \end{aligned}$$

3. Consider components $C5$ and $C6$. The reliability, which we will call R_I, is then calculated as in the previous step:

$$\begin{aligned} R_I &= 1 - [(1 - R_{C5}) \cdot (1 - R_{C6})] \\ &= 1 - [(1 - 0.92) \cdot (1 - 0.92)] \\ &= 1 - 0.0064 \\ &= 0.9936 \end{aligned}$$

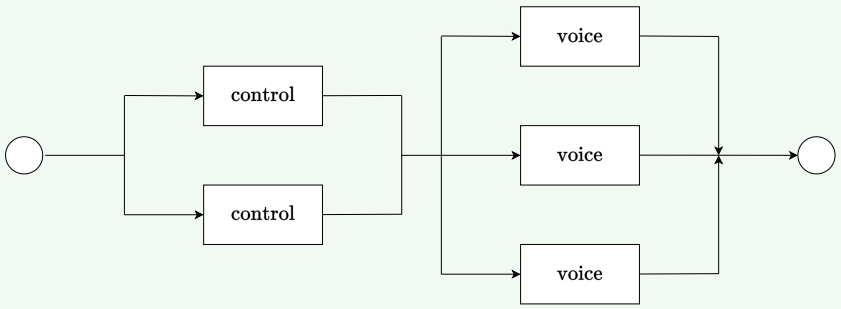
4. Finally, we calculate the reliability of the system by multiplying each calculated component reliability:

$$\begin{aligned} R_s &= R_G \cdot R_H \cdot R_I \\ &= 0.9215 \cdot 0.9999 \cdot 0.9936 \\ &= 0.91551083976 \approx 0.9155 \end{aligned}$$

Example 10: calculate reliability without numbers

Question

The system consists of 2 control blocks and 3 voice channels. The system is up when at least 1 control channel and at least 1 voice channel are up.



Solution

Reliability can be calculated in parallel, as it takes almost a component to work properly. Each control channel has reliability R_c and each voice channel has reliability R_v:

$$R = \left[1 - (1 - R_c)^2\right] \cdot \left[1 - (1 - R_v)^3\right]$$

4.1.3.1 R out of N redundancy (RooN)

An **RooN (r out of n)** redundancy system **contains** both the **series system model and the parallel system model** as special cases. The system has n components that operate or fail independently of one another and as long as at least r of these components (any r) survive, the system survives. [5]

A system with:

- n: total number of identical components.
- r: minimum number of components that must **work correctly**.

The system function **if at least r out of the n components are operational**. **System failure occurs when** the $(n - r + 1)$-th component failure occurs. [5]

But note an interesting observation:

- When $r = n$, the r out of n model reduces to the **series** model. [5]
- When $r = 1$, the r out of n model becomes the **parallel** model. [5]

In simple terms, `RooN` is a system made up of n identical replicas, where at least r replicas have to work well for the whole system to work well.

The reliability formula for the RooN system is:

$$R_s(t) = R_V \sum_{i=r}^{n} R_c^i (1 - R_c)^{n-i} \frac{n!}{i!(n-i)!} \tag{47}$$

Where:

- n: total number of identical components.
- $R_c(t)$: reliability of each component.
- $R_V(t)$: reliability of the **voter** (which combines outputs).
- $R_S(t)$: system reliability.
- $(1 - R_c)^{n-i}$: failure probability of the remaining $n - i$.

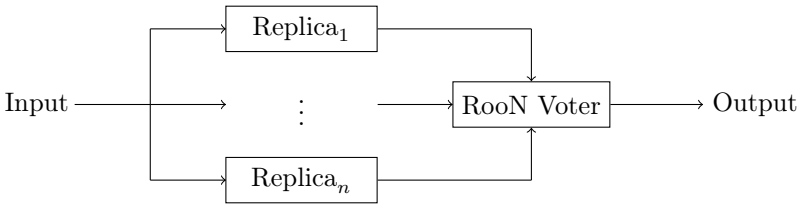


Figure 38: General structure of RooN system.

The reliability of the **central voter** $R_V(t)$ is crucial. If the voter fails, the system fails, even if enough components are still working. In practice, voter reliability is either assumed perfect ($R_V = 1$) or modeled separately.

4.1.3.2 Triple Modular Redundancy (TMR)

Triple Modular Redundancy (TMR) is a fault-tolerant form of N-modular redundancy, in which **three systems perform a process and the result is processed by a majority-voting system to produce a single output**. If any **one of the three systems fails**, the **other two systems can correct and mask the fault**. In other words, it is a special case of the r-out-of-n redundancy model:

$$r = 2, \quad n = 3$$

- TMR consists of **three identical components** performing the **same task in parallel**.
- A **voter** ***compares the outputs*** and ***selects the majority result***.
- The **system works correctly** as long as at **least two components** function correctly and the **voter does not fail**.

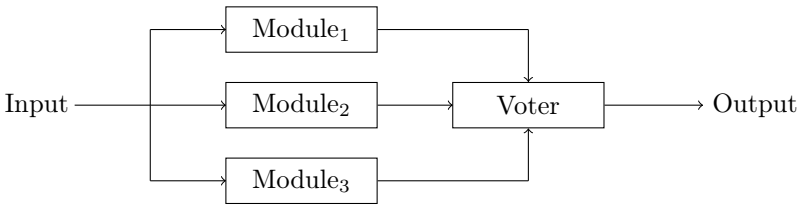


Figure 39: General structure of TMR system.

The **TMR Reliability** $R_{\mathbf{TMR}}$ is:

$$\begin{aligned} R_{\text{TMR}} &= R_v \left(3 \cdot R_m^2 - 2 \cdot R_m^3\right) \\ &= R_v \left[(\text{2-out-of-3 OK}) + (\text{3-out-of-3 OK})\right] \end{aligned} \tag{48}$$

Where:

- R_m: reliability of a **module**.
- R_v: reliability of the **voter**.
- $3 \cdot R_m^2$: probability of exactly 2 modules working.
- R_m^3: probability of all 3 modules working.

To compute the Mean Time To Failure (MTTF) of a Triple Modular Redundancy (TMR) system, assuming:

- **Identical components**: each with exponential reliability $R_m(t) = e^{-\lambda t}$.
- **Voter** has **perfect reliability** $R_v = 1$ (or it's included later as a separate element).

The MTTF of a single component, for exponential reliability, is:

$$R(t) = e^{-\lambda t} \quad \Rightarrow \quad \text{MTTF} = \frac{1}{\lambda}$$

And the **MTTF of TMR system** is:

1. Using the TMR reliability:

$$R_{\text{TMR}}(t) = 3R_m^2(t) - 2R_m^3(t) = 3e^{-2(\lambda t)} - 2e^{-3(\lambda t)}$$

2. To find MTTF:

$$\text{MTTF}_{\text{TMR}} = \int_0^\infty R_{\text{TMR}}(t)\,\mathrm{d}t$$

3. Compute:

$$\text{MTTF}_{\text{TMR}} = \int_0^\infty \left(3e^{-2\lambda t} - 2e^{-3\lambda t}\right)\,\mathrm{d}t = \frac{3}{2\lambda} - \frac{2}{3\lambda}$$

4. Get common denominator:

$$\text{MTTF}_{\text{TMR}} = \left(\frac{9}{6\lambda} - \frac{4}{6\lambda}\right) = \frac{5}{6\lambda}$$

The final formula is:

$$\text{MTTF}_{\text{TMR}} = \frac{5}{6} \cdot \text{MTTF}_{\text{module}} \tag{49}$$

Where $\text{MTTF}_{\text{module}}$ is the Mean Time To Failure of a single component in the system.

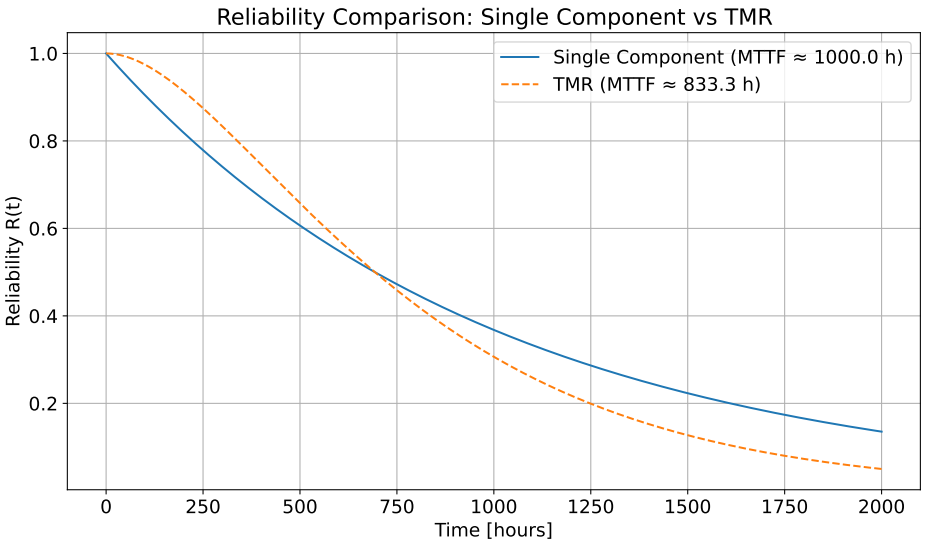


Figure 40: Plot comparing the reliability over time of a single component vs a TMR system.

TMR: good or bad?

Figure 149 shows the benchmark of a TMR system. At **early mission times** (before 700-800), **TMR has higher reliability**, meaning it's better at masking early failures. **Later in time** (> 1000 hours), the **single component** eventually **becomes more reliable** because the three-module TMR setup is exposed to more cumulative risk.

However, Triple Modular Redundancy (TMR) is neither universally good nor bad. Like many engineering trade-offs, its value depends on the mission profile and system requirements.

When TMR is Good

- **Short mission durations**: TMR boosts reliability in early hours. For $t < 0.7 \cdot \text{MTTF}_{\text{module}}$, TMR outperforms a single module.
- **Safety-critical systems**: TMR can tolerate 1 permanent fault and mask transient faults, like radiation bit flips or noise.
- **Spacecraft / Avionics**: TMR is widely used in flight computers, satellites, and nuclear systems where failures are unacceptable.
- **High electromagnetic interference**: TMR can suppress soft errors (temporary glitches).
- **Voting logic is reliable**: If the voter is highly reliable (or replicated), the system benefits greatly from TMR.

When TMR is Bad. For **long mission durations**, TMR has a lower MTTF than a single module. So TMR is bad when long-term availability is key.

What is the MTTF Threshold? There exists a specific time t^* where:

$$R_{\text{TMR}}(t^*) = R_{\text{module}}(t^*)$$

- Before that time ($t < t^*$), TMR has **higher reliability**.
- After that time ($t > t^*$), TMR becomes **less reliable**.

Using numerical or symbolic solving, one finds that:

$$t^* \approx 0.693 \cdot \text{MTTF}_{\text{module}} = \frac{0.693}{\lambda}$$

This threshold corresponds to the time where:

$$R_{\text{TMR}}(t) = R_{\text{module}}(t)$$

Hence, we commonly approximate:

$$R_{\text{TMR}}(t) > R_{\text{module}}(t) \quad \text{for } t < 0.7 \cdot \text{MTTF}_{\text{module}} \tag{50}$$

4.1.3.3 Standby redundancy

Standby redundancy is a fault-tolerance strategy where **redundant components are kept inactive** (in "standby mode") until the **primary component fails**, at which point a **backup** is activated.

Unlike parallel redundancy, where all components run simultaneously, standby redundancy has **only one active component at a time**. It has two main roles:

1. **Primary replica**: operates under normal conditions.
2. **Standby (backup) replica(s)**: activated **only upon failure** of the primary.

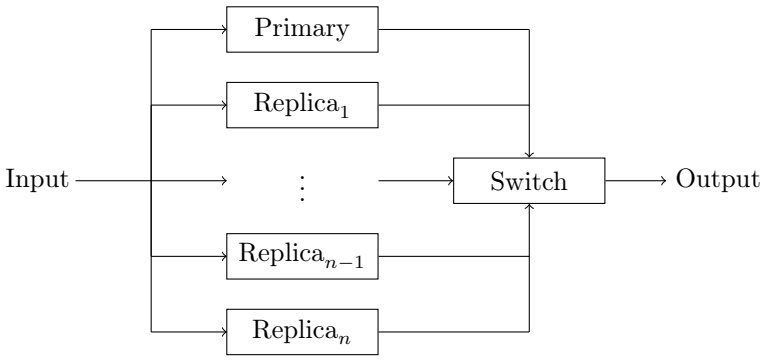


Figure 41: General standby redundancy strategy.

How it works?

It works as follows:

1. The **primary** performs all tasks.
2. A **monitoring mechanism** detects failure.
3. A **switching mechanism** disables the primary and **activates the standby**.

Requirements for operation

Standby redundancy only works if the system includes:

- **Self-check mechanism**: Detects when the primary has failed.
- **Switch mechanism**: Transfers operation to the standby.

Standby Parallel Model	**System Reliability**
Equal failure rates, perfect switching	$R_s = e^{-\lambda t}\left(1 + \lambda t\right)$
Unequal failure rates, perfect switching	$R_s = e^{-\lambda_1 t} + \lambda_1 \frac{\left(e^{-\lambda_1 t} - e^{-\lambda_2 t}\right)}{\lambda_2 - \lambda_1}$
Equal failure rates, imperfect switching	$R_s = e^{-\lambda t}\left(1 + R_{\text{switch}}\lambda t\right)$
Unequal failure rates, imperfect switching	$R_s = e^{-\lambda_1 t} + R_{\text{switch}}\lambda_1 \frac{\left(e^{-\lambda_1 t} - e^{-\lambda_2 t}\right)}{\lambda_2 - \lambda_1}$

Table 11: Standby redundancy - Quick Formulas.

In the previous table we have:

- R_s: System Reliability
- λ: Failure Rate
- t: Operating Time
- R_{switch}: Switching Reliability

4.2 Disk performance

4.2.1 HDD

We can calculate some performance metrics related to the types of delay of HDD (page 41).

- **Full Rotation Delay** R is:

$$R = \frac{1}{\texttt{DiskRPM}} \tag{51}$$

And in seconds:

$$R_{\text{sec}} = 60 \times R \tag{52}$$

From the R_{sec} we can also calculate the **total rotation average**:

$$T_{\text{rotation AVG}} = \frac{R_{\text{sec}}}{2} \tag{53}$$

It is half of a full rotation, because on average, the sector will be halfway around the platter from the current head position.

- **Seek Time**. The time to seek from one track to another depends on the distance moved. In real systems, this relation isn't perfectly linear, but it's often approximated as (**seek average**):

$$T_{\text{seek AVG}} = \frac{T_{\text{seek MAX}}}{3} \tag{54}$$

Where $T_{\text{seek MAX}}$ is the **time for the longest possible seek** (from *outermost to innermost* track) and the division by 3 assumes a **uniform random distribution of seeks** across the disk.

- **Transfer time**. It is the **time that data is either read from or written to the surface**. It **includes** the time the **head needs to pass on the sectors** and the **I/O transfer**:

$$T_{\text{transfer}} = \frac{\text{R/W of a sector}}{\text{Data transfer rate}} \tag{55}$$

The **total time to complete a disk I/O operation** is called the $T_{\text{I/O}}$ **Service Time**:

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}} \tag{56}$$

If the disk is shared among processes, we must also consider **queueing time**. And the **Response Time** is:

$$T_{\text{response}} = T_{\text{queue}} + T_{\text{I/O}} \tag{57}$$

Where T_{queue} depends on queue length, disk utilization rate, variance in request time, arrival rate of I/O requests.

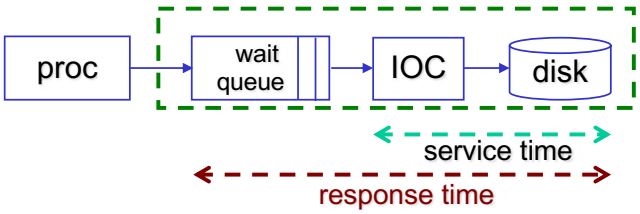


Figure 42: Service and response time.

Exercise 1: Mean Service Time of an I/O operation

The data of the exercise are:

- Read/Write of a sector of 512 bytes (0.5 KB)
- Data transfer rate: 50 MB/sec
- Rotation speed: 10000 RPM (Round Per Minute)
- Mean seek time: 6 ms
- Overhead Controller: 0.2 ms

The goal is to calculate the average I/O service time. To calculate the *service time* $T_{\text{I/O}}$, we need the following information:

- ✔ T_{seek}, which we already have, and it is 6 ms.
- ✘ T_{rotation}
- ✘ T_{transfer}
- ✔ T_{overhead}, which we already have, and it is 0.2 ms.

We also know the rotation and transfer information, but we want to know the *mean* service time. Then we calculate the total rotation average $T_{\text{rotation AVG}}$:

$$\begin{aligned} R &= \frac{1}{\texttt{DiskRPM}} = \frac{1}{10000} = 0.0001 \\ R_{\text{sec}} &= 60 \cdot R = 60 \cdot 0.0001 = 0.006 \text{ seconds} \\ T_{\text{rotation AVG}} &= \frac{R_{\text{sec}}}{2} = \frac{0.006}{2} = 0.003 \text{ seconds} = 3 \text{ ms} \end{aligned}$$

Finally, the transfer time is easy to calculate because we have the R/W of a sector and the data transfer rate. First we do a conversions from

megabytes to kilobytes:

$$\begin{aligned} \text{Data transfer rate:} \quad & 50 \text{ MB/sec} \\ &= 50 \cdot 1024 \text{ KB/sec} \\ &= 51200 \text{ KB/sec} \\ T_{\text{transfer}} &= \frac{0.5 \text{ KB/sec}}{51200 \text{ KB/sec}} \\ &= 0.000009765625 \text{ sec} \cdot 1000 \\ &= 0.009765625 \text{ ms} \approx 0.01 \text{ ms} \end{aligned}$$

The exercise can be completed by calculating the mean I/O service time required:

$$\begin{aligned} T_{\text{I/O}} &= T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}} \\ T_{\text{I/O}} &= 6 + 3 + 0.01 + 0.2 = 9.21 \text{ ms} \end{aligned}$$

The I/O service time computed in the previous exercise (9.21 ms) assumes a **worst-case scenario**. This is very useful for understanding disk behavior, but it **doesn't always reflect what happens in real workloads**. It assumes:

1. **Every time** we read a small file or sector,
2. The **read must seek** to a new track,
3. And then **wait for rotation** to bring the right sector under the head

This is worst-case behavior, and happens when **files are very small**, so each one is just one block (e.g., 512 bytes); or disk is **heavily externally fragmented**, so even larger files are broken into scattered blocks. In such cases, **every access pays both seek and rotational delay**, making the access time slow and constant.

We can introduce a new metric that comes from the idea of measuring locality: **how often we can avoid seek and rotation delays**. We define **Data Locality** DL as:

$$DL = \% \text{ of blocks that can be accessed without new seek or rotation} \quad (58)$$

If **locality is high**, then most of our **data** is laid out in a nice, **sequential way**, therefore performance improves significantly. So, **locality determines whether our performance is close to best-case or worst-case**.

Thanks to the Data Locality, it is possible to calculate the **Average Service Time** by modifying the terms of *seek* and *rotation* of the Service Time equation (page 153):

$$T_{\text{I/O AVG}} = (1 - DL) \cdot (T_{\text{seek}} + T_{\text{rotation}}) + T_{\text{transfer}} + T_{\text{controller}} \quad (59)$$

Exercise 2: Data Locality

The data of the exercise are:

- Read/Write of a sector of 512 bytes (0.5 KB)
- Data Locality: $DL = 75\%$
- Data transfer rate: 50 MB/sec
- Rotation speed: 10000 RPM (Round Per Minute)
- Mean seek time: 6 ms
- Overhead Controller: 0.2 ms

Since the Data Locality is 75%, only 25% of the operations are affected by the DL:

$$(1 - DL) = (1 - 0.75) = 0.25$$

See the exercise on page 154 to understand the values of T_{seek}, T_{rotation}, T_{transfer} and T_{overhead}:

- $T_{\text{seek}} = 6$
- $T_{\text{rotation}} = 3$
- $T_{\text{transfer}} = 0.01$
- $T_{\text{overhead}} = 0.2$

Finally the average time for read/write a sector of 0.5 KB with a DL of 75% is:

$$\begin{aligned} T_{\text{I/O AVG}} &= 0.25 \cdot (6 + 3) + 0.01 + 0.2 \\ &= 0.25 \cdot 9 + 0.21 \\ &= 2.46 \text{ ms} \end{aligned}$$

Exercise 3: Influence of "Not Optimal" Data Allocation

The data of the exercise are:

- 10 blocks of 1/10 MB for each block (10 blocks of 1/10 MB "not well" distributed on disk)
- $T_{\text{seek}} = 6$ ms
- $T_{\text{rotation AVG}} = 3$ ms
- Data transfer rate: 50 MB/sec

In the exercise you were asked to calculate the time taken to transfer a 1 MB file with 100% and 0% data locality:

- Data Locality equals to 100%:
  - An initial seek (6 ms)
  - A total rotation average (3 ms)
  - Now it's possible to do the 1MB global transfer directly because there are no blocks to seek or rotation latency:

$$1 \text{ MB of } 50 \text{ MB} = \frac{1}{50} = 0.02 \text{ seconds } \cdot 1000 = 20 \text{ ms}$$

  - The total time is:

$$T = 6 + 3 + 20 = 29 \text{ ms}$$

- Data Locality equals to 0%:
  - An initial seek (6 ms)
  - A total rotation average (3 ms)
  - In this case, it's not possible to do a global transfer directly, because each block is affected by the seek or rotation latency. Then we have to transfer block by block and calculate the delay:

$$1 \text{ MB of } 10 \text{ MB} = \frac{1}{10} = 0.1 \text{ seconds } \cdot 1000 = 100 \text{ ms}$$

  - The total time is:

$$T = (6 + 3 + 2) \cdot 10 = 110 \text{ ms}$$

    Where 10 is the number of blocks.

Note: the controller times is not considered.

4.2.2 RAID

We can calculate some performance metrics related to the RAID technology (page 57).

- **MTTF (Mean Time To Failure)** is the **expected time before a disk fails**. It is usually provided by the manufacturer, based on statistical modeling (often assuming exponential failure distribution).

  To calculate the MTTF for a Disk Array without redundancy, we must make the following assumptions:

  – Constant Failure Rate;
  – Exponentially distributed time to failure;
  – Independent disk failures.

  Then the formula is:

$$\texttt{MTTF}_{\text{array}} = \frac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}} \tag{60}$$

  This is because the array fails if **any one** of its disks fails; so the **failure rate of the system increases linearly with the number of disks**.

  ⚠ **RAID 0 and MTTF**. RAID 0 offers **no redundancy**, so it behaves like a pure array:

$$\texttt{MTTF}_{\text{RAID 0}} = \texttt{MTTF}_{\text{array}} = \frac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}} \tag{61}$$

  this is why RAID 0 is **high-risk** despite its performance: one disk failure means total data loss.

Example 11: MTTF

If:

– Each disk has an MTTF of $1,000,000$ hours.
– We have an array of 100 disks.

Then:

$$\texttt{MTTF}_{\text{array}} = \frac{1,000,000}{100} = 10,000 \text{ hours}$$

Without fault tolerance, our system is 100× more likely to fail than a single disk.

With RAID 5 or 6, the system can still work if one disk fails. However, **if another disk fails during the rebuild** process, **data is lost** in RAID 5 or **further degraded** in RAID 6. Therefore, the probability of data loss is approximately equal to the probability of a second failure occurring within the mean time to recovery (MTTR) window.

$$\texttt{MTTF}_{\text{RAID}} = \texttt{MTTF}_{\text{array}} \cdot \left(\frac{1}{\% \text{ of additional critical failures during MTTR}} \right) \tag{62}$$

The first part $\texttt{MTTF}_{\text{array}}$ captures the **rate of first failures**. The second part **increases the effective** `MTTF` **depending** on **how** likely we are to **survive the rebuild window** (i.e., how unlikely a second failure is during `MTTR`). In other words, the **probability term** models the likelihood of **another failure occurring during a vulnerable period when the system is degraded** (e.g., during the process of rebuilding a RAID 5 array with one failed disk).

- The **Mean Time To Failure of each RAID level** (except the zero) is:
  - `RAID 1` (page 64) employs **mirroring**: every disk has an exact replica. This means each piece of data is stored **twice**, on two separate drives. The system can tolerate **one drive failure per mirrored pair** without data loss. In theory:
    - ✔ **Best case**: up to $\frac{N}{2}$ drives can fail without affecting data (if each failed drive is from a different mirror).
    - ✖ **Worst case**: system fails if both disks in any mirror pair fail before the array is rebuilt.

    The formula used is:

    $$\texttt{MTTF}_{\texttt{RAID 1}} = \texttt{MTTF}_{\text{array}} \cdot \left(\frac{1}{\%\text{ of 2nd critical failure during MTTR}} \right) \tag{63}$$

    Where:
    - $\texttt{MTTF}_{\text{array}} = \frac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}}$
    - MTTR: Mean Time To Repair.
    - % of 2nd critical failure during MTTR:

      $$\%\text{ of 2nd critical failure during MTTR} = \frac{1}{\texttt{MTTF}_{\text{disk}}} \cdot \texttt{MTTR} \tag{64}$$

      Where:
      - $\frac{1}{\texttt{MTTF}_{\text{single disk}}}$ is the **failure rate** of a single disk.
      - `MTTR` is the the **duration in which the system is in a degraded state**, i.e., vulnerable to a second failure.

    This probability models the **chance that the mirror of a failed disk also fails during the rebuild window**.
  - `RAID 0 + 1` (page 64) means **first striping** (RAID 0), then **mirroring** (RAID 1). The disks are divided into **stripe groups**, and then each group is mirrored. If any disk in a stripe group fails, **that whole group is lost**. The array can survive **a single group failure**, but not a second.

    $$\texttt{MTTF}_{\texttt{RAID 0+1}} = \texttt{MTTF}_{\text{array}} \cdot \left(\frac{1}{\%\text{ of 2nd critical failure during MTTR}} \right) \tag{65}$$

    Where:
    - $\texttt{MTTF}_{\text{array}} = \frac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}}$

* `MTTR` is the mean time to repair (the vulnerable window).
* Despite the RAID 1, the **probability of a second critical failure during the mean time to repair** (MTTR) is:

$$\%\text{ of 2nd critical failure during MTTR} = \left(\frac{G}{\texttt{MTTF}_{\text{disk}}}\right) \cdot \texttt{MTTR} \tag{66}$$

Where G is the number of **disks in each stripe group**, and $\frac{G}{\texttt{MTTF}_{\text{disk}}}$ is the **combined failure rate** of the G disks in the remaining group. Multiplying this by the MTTR gives the probability that the second group will fail during the rebuild.
Once a **disk** in one stripe group **fails**, the **entire group becomes inoperative**. To avoid total data loss, the **mirrored stripe group must not fail before rebuild completes**.

– RAID 1 + 0 (page 65) combines **mirroring first** (RAID 1) then **striping** (RAID 0). Disks are organized into mirrored pairs. Then data is striped across the pairs. A disk can fail in any mirrored pair, the system remains operational. To cause data loss, **both disks in a mirrored pair must fail**. Thus, it **tolerates multiple disk failures** as long as no pair is entirely lost.

$$\texttt{MTTF}_{\texttt{RAID 1+0}} = \texttt{MTTF}_{\text{array}} \cdot \left(\frac{1}{\%\text{ of 2nd critical failure during MTTR}}\right) \tag{67}$$

Where:

* $\texttt{MTTF}_{\text{array}} = \frac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}}$
* `MTTR` is the mean time to repair (the vulnerable window).
* The probability of a second critical failure during the Mean Time To Repair (MTTR) is the same as for RAID 1 (see formula 64):

$$\%\text{ of 2nd critical failure during MTTR} = \frac{1}{\texttt{MTTF}_{\text{disk}}} \cdot \texttt{MTTR} \tag{68}$$

The system is only at risk if the **exact mirror** of the failed disk also fails during the rebuild. So, unlike `RAID 0+1`, the failure risk localized, not dependent on an entire group.

– RAID 4 (page 68) is a block-level striping with a **dedicated parity disk**. All parity information is stored on **one disk**, creating a bottleneck on writes.
RAID 5 (page 72) is a block-level striping with **distributed parity**. Parity is **spread across all disks**, improving write performance and eliminating the RAID 4 bottleneck.
Both RAID 4 and RAID 5 can **tolerate a single disk failure**. Data is reconstructed using parity. Failure occurs only if **a second disk** fails **before** the first one is rebuilt.

$$\texttt{MTTF}_{\texttt{RAID 4, 5}} = \texttt{MTTF}_{\text{array}} \cdot \left(\frac{1}{\%\text{ of 2nd failure during MTTR}}\right) \tag{69}$$

Where:

* $\texttt{MTTF}_{\text{array}} = \dfrac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}}$
* `MTTR` is the mean time to repair (the vulnerable window).
* The probability of a second critical failure during the Mean Time To Repair (MTTR) is:

$$\%\text{ of 2nd failure during MTTR} = \frac{N-1}{\texttt{MTTF}_{\text{disk}}} \cdot \texttt{MTTR} \tag{70}$$

Where N is the number of disks in the array. The **first failure** does not cause data loss (covered by parity). The **second failure** is only dangerous **if it occurs during rebuild**. The probability considers all $N-1$ remaining disks and their individual failure rates.

They have the same formula because they both rely on **single-parity protection**. The **main difference** between them is their **performance** and **load balancing capabilities**, not their reliability!

– RAID 6 (page 74) is an extension of RAID 5 with **double parity**. It uses two independent parity blocks per stripe (commonly P and Q). These allow the array to tolerate **two simultaneous disk failures**. The system only fails if **a third disk fails** during the time in which the two failed disks are being rebuilt.

$$\texttt{MTTF}_{\texttt{RAID 6}} = \texttt{MTTF}_{\text{array}} \cdot \frac{1}{\%\text{ 2nd failure} \cdot \%\text{ 3rd failure}} \tag{71}$$

Where:

* $\texttt{MTTF}_{\text{array}} = \dfrac{\texttt{MTTF}_{\text{single disk}}}{\texttt{Number of Disks}}$
* `MTTR` is the mean time to repair (the vulnerable window).
* The probability of the second failure during MTTR:

$$\%\text{ 2nd failure} = \frac{N-1}{\texttt{MTTF}_{\text{single disk}}} \cdot \texttt{MTTR} \tag{72}$$

It is the risk of a second failure while the first is being repaired.

* The probability of the third failure during MTTR:

$$\%\text{ 3nd failure} = \frac{N-2}{\texttt{MTTF}_{\text{single disk}}} \cdot \frac{\texttt{MTTR}}{2} \tag{73}$$

It is the risk of a third failure during the overlapping time when **both first and second disks** are being rebuilt. The `MTTR` is divided by two because it **assumes** that both **disks are under recovery simultaneously** (**average time**).

Unlike RAID 5, RAID 6 can still recover after a second disk fails. But during this dual-failure state, the system becomes vulnerable to a **third-failure**. That third failure is what RAID 6 models as catastrophic.

Metric	Tolerates
$\texttt{MTTF}_{\texttt{RAID 0}} = \frac{\texttt{MTTF}_{\text{single disk}}}{N}$	0 disk failures
$\texttt{MTTF}_{\texttt{RAID 1+0}} = \frac{\left(\texttt{MTTF}_{\text{single disk}}\right)^2}{(N \times \texttt{MTTR})}$	Multiple failures (1 per mirror)
$\texttt{MTTF}_{\texttt{RAID 0+1}} = \frac{\left(\texttt{MTTF}_{\text{single disk}}\right)^2}{(N \times G \times \texttt{MTTR})}$	1 full group
$\texttt{MTTF}_{\texttt{RAID 4}} = \frac{\left(\texttt{MTTF}_{\text{single disk}}\right)^2}{(N \times N-1 \times \texttt{MTTR})}$	1 disk
$\texttt{MTTF}_{\texttt{RAID 5}} = \frac{\left(\texttt{MTTF}_{\text{single disk}}\right)^2}{(N \times N-1 \times \texttt{MTTR})}$	1 disk
$\texttt{MTTF}_{\texttt{RAID 6}} = \frac{2 \times \left(\texttt{MTTF}_{\text{single disk}}\right)^3}{\left(N \times N-1 \times N-2 \times \texttt{MTTR}^2\right)}$	2 disks

Table 12: MTTF summary RAID levels.

- `RAID 0`: No redundancy. Failure of any disk leads to total data loss.
- `RAID 1+0`: High reliability; localized failure risk; faster rebuilds.
- `RAID 0+1`: Less reliable than `1+0`; entire group fails if any disk in it fails.
- `RAID 4/5`: Identical `MTTF`; `RAID 5` distributes parity to avoid bottlenecks of `RAID 4`.
- `RAID 6`: Extremely high reliability; more complex failure modeling.

Last remarks:

- The increase in **exponential of** $\texttt{MTTF}_{\text{disk}}$ reflects higher redundancy (`RAID 6` uses $\texttt{MTTF}^3$).
- More complex RAID levels (`1+0` or `6`) handle multiple simultaneous failures much more gracefully.
- **Mean Time To Repair (MTTR)** plays a critical role: the long the rebuild takes, the more likely a second or third failure occurs.

4.3 Scalability and performance of data centers

4.3.1 Evaluate system quality

System quality information is critical from a cost and performance perspective. In this context, "*performance*" means the **overall effectiveness of a computer system in terms of throughput**, **response time**, and **availability**.

So how do we evaluate system quality?

There are generally two approaches:

- **Intuition and trend extrapolation**. Obviously, those who possess these qualities in sufficient quantity are rare. The pros are speed and flexibility, but the cons are accuracy.
- **Experimental evaluation of alternatives**. As pro has excellent accuracy, but as con has laborious and flexible.

The techniques are represented in the following figure.

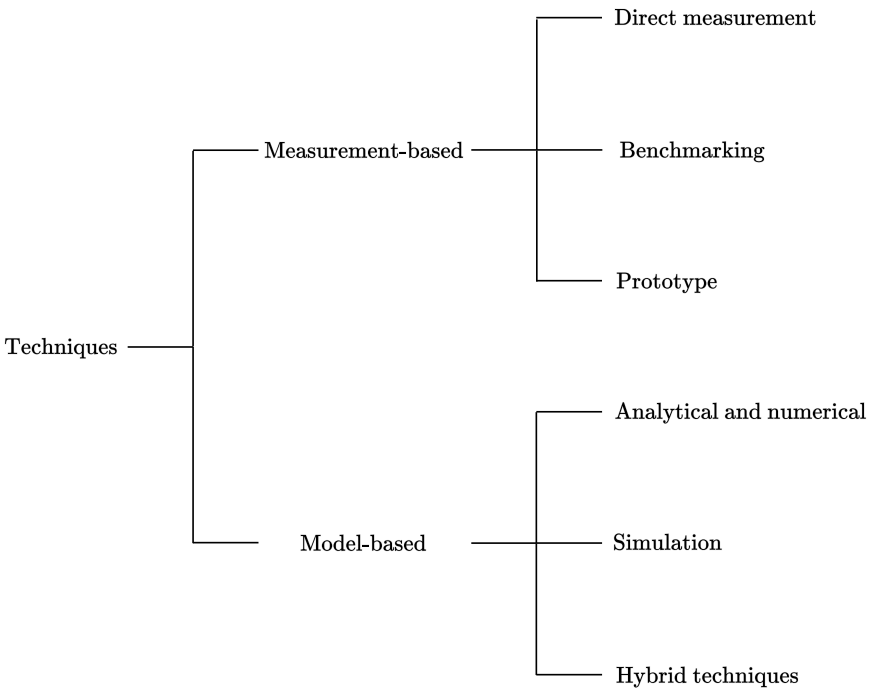


Figure 43: Quality Evaluation techniques.

The most common and useful solution to evaluate system quality is **model-based approach**. The systems are complex, so it is useful to create an **abstraction of the systems called models**. The model-based are divided into three groups:

- **Analytical and numerical techniques** are based on the **application of mathematical techniques**, which usually exploit results coming from the theory of probability and stochastic process.

  Advantages

  - Most **efficient**.
  - **Accurate**.

  Disadvantages

  - **Available** only in very **limited cases**.

- **Simulation techniques** are based on the **reproduction of traces of the model**.

  Advantages

  - Most **general**.

  Disadvantages

  - May be **less accurate**, especially when considering cases where rare events may occur.
  - **Solution time** can also be **very long** if high accuracy is desired.

- **Hybrid techniques combine analytical/numerical methods with simulation**.

4.3.2 Queueing Networks

4.3.2.1 Definition

Queueing Network Modeling is a particular approach to computer system modeling in which the **computer system is represented as a network of queues**. A *network of queues* is a collection of **service centers**, which represent system **resources**, and **customers**, which represent **users** or **transactions**. [4]

Some **examples** of queues in computer systems are:

- CPU uses a time-sharing scheduler.
- A disk serves a queue of requests waiting to read or write blocks.
- A router on a network serves a queue of packets waiting to be routed.
- Databases have lock queues where transactions wait to acquire the lock on a record.

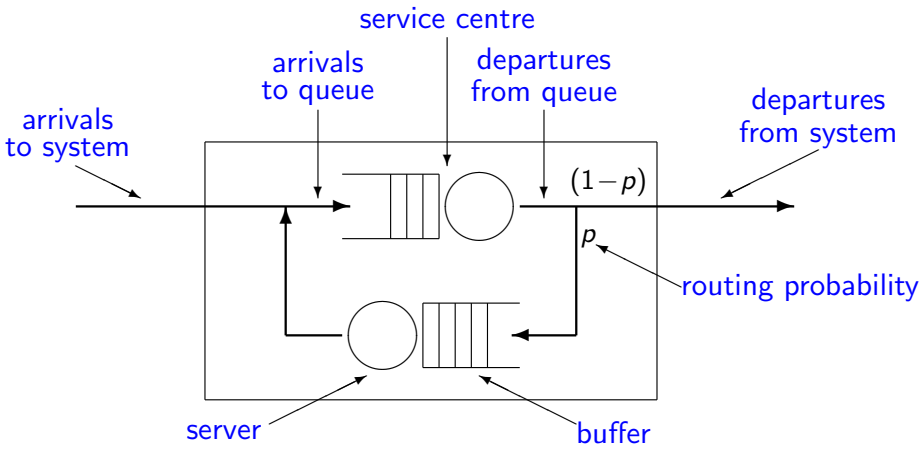


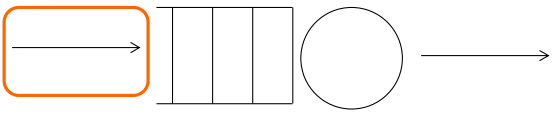
Figure 44: Queueing Network graphical representation.

4.3.2.2 Characteristics

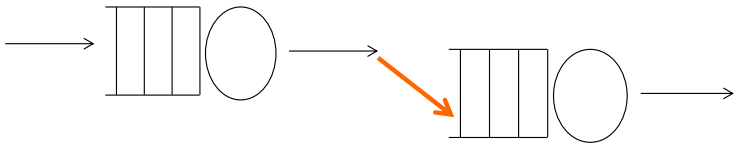
Queueing models are characterized by several aspects:

- **Arrival**. Arrivals **represent orders coming into the system**. They specify *how fast*, *how often*, and *what types of jobs* the **station will service**. **Arrivals can come from**:

  1. An external source.

  2. Another queue.

  3. The same queue, through a loopback arc.

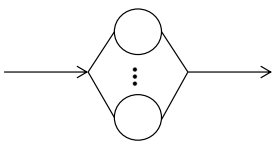
  

- **Service**. Service **represents the time a job spends being served**. The server does the job, but the number of servers can be different:

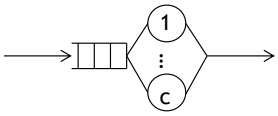
  – **Single server**. It has the ability to serve **one client at a time**. **Waiting customers remain in the buffer** until they are selected for service. Finally, the **next customer is selected depending on the service discipline**.

  – **Infinite servers**. There are always at least **as many servers as there are customers**, and each **customer can have a dedicated server**. As a consequence, there is no queuing (and no buffer).

  - **Multiple servers**. There is a **fixed number of servers** (c in the figure below), each of which can **serve one customer at a time**.
    * `Number of customers in the system` $\leq$ `number of servers` $\Rightarrow$ **no queuing**.
    * `Number of customers in the system` $>$ `number of servers` $\Rightarrow$ the additional **customers must wait in the buffer**.



- **Queue**. If **jobs exceed the parallel processing capacity of the system**, they are **forced to wait in a buffer**.

  When the job currently in service leaves the system, one of the jobs in the queue can now enter the free service center. **Service Discipline/Queuing Policy determines which of the jobs in the queue will be selected to start its service**.

- **Population**. Ideally, members of the population are indistinguishable from one another. When this is not the case, we divide the population into **classes whose members all exhibit the same behavior**. Different classes differ in one or more characteristics, e.g. arrival rate, service demand. Identifying different classes is a task of workload characterization.

- **Routing**. For many systems, we can view the system as a collection of resources and devices, with customers or jobs circulating between them.

  We can associate a service center with each resource in the system and then route customers between the service centers.

  After being serviced at one service center, a customer can move on to other service centers, following a pre-defined pattern of behavior according to the customer's needs.

  A queueing network can then be represented as a graph where the nodes represent the service centers k and the arcs represent the possible transitions of users from one service center to another. Nodes and arcs together define the network topology.

  Whenever **a job has several possible alternative routes after completing service at a station**, an appropriate selection policy must be defined.

  The policy is also called the **Routing Algorithm**. The most important routing algorithms are:

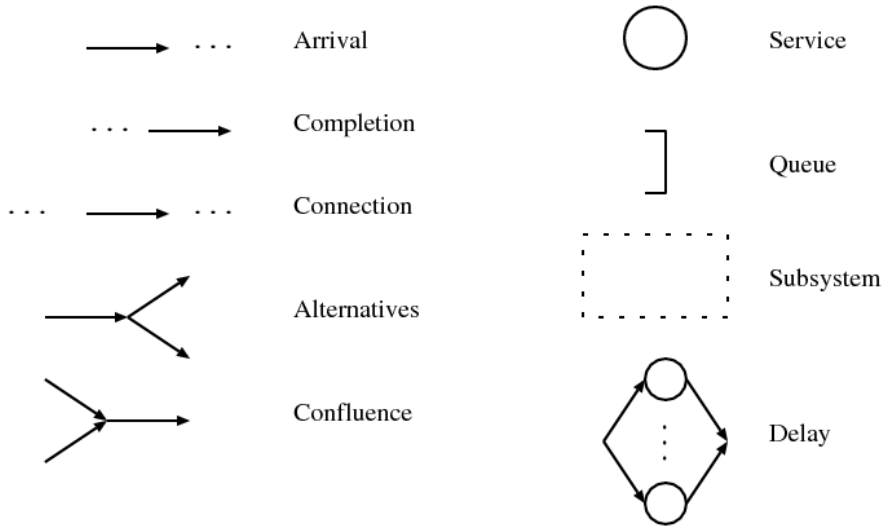
  - **Probabilistic Routing Algorithm**. **Each path is assigned a probability of being chosen by the job that left the station in question**.
  - **Round Robin Routing Algorithm**. The **destination chosen by the job rotates among all possible existing destinations**.

– **Join the shortest queue Routing Algorithm. Jobs can query the queue length of the possible destinations and choose the one with the least number of jobs waiting to be served.**

With important definition of routing, we can say that a **network** can be:

- **Open**. Customers can come from or go to any external environment.
- **Closed**. A fixed population of customers remains within the system.
- **Mixed**. There are classes of customers within the system that exhibit both open and closed patterns of behavior.

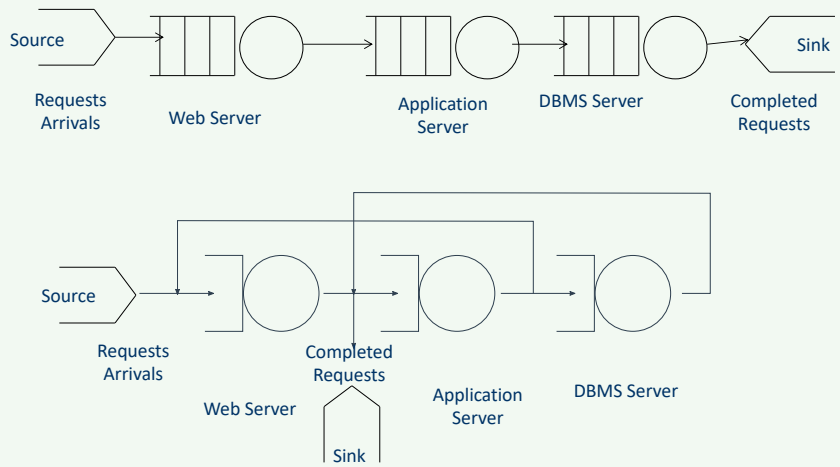
An additional graphical notation is:



Example 12: Open Networks

A client server system, dealing with external arrivals (classical three tier architecture).
Provide a QN model of the system and evaluate the overall throughput considering that the network delay is negligible with respect to the other devices and two different cases:

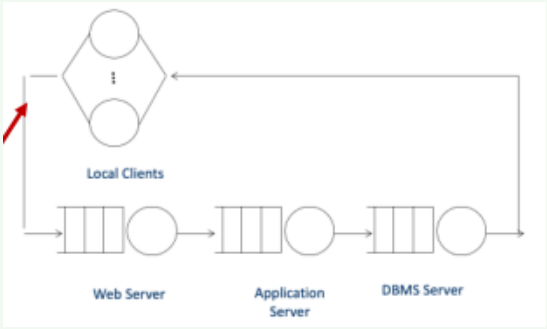
1. The only thing we know is that each server should be visited by the application.
2. In the second case we know that the application **after visiting the web server** requires some operations at **the application server** and then **can go back** to **the web server** and **leave** the system **or** can require service at the **DMBS** and then **go back** to the **application server**.



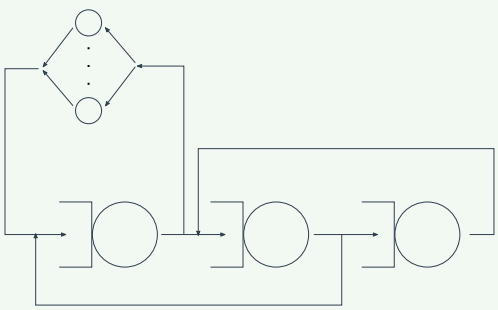
Example 13: Closed Networks

A client server system, **with a finite number of customers** (classical three tier architecture and not accessible from outside).
Provide a QN model of the system and evaluate the system throughput considering that Network delay is negligible with respect to the other devices. Model the two different cases previously described.

- First scenario



- Second scenario



4.3.3 Operational Laws

Operational Laws are **simple equations** that can be **used as an abstract representation** or **model** of the average **behavior** of almost any **system**.

Advantages

- The laws are very **general** and make almost **no assumptions about the behavior of the random variables** that characterize the system.
- **Simplicity**: they can be applied quickly and easily.

In the Computing Infrastructure course, then in this note, the operational laws are applied to the Queueing Network model (4.3.2, page 165).

Operational laws are based on **observable variables**, **values** that we can **derive by observing a system** over a finite period of time.

In general, we assume that the **system receives requests from its environment**. Each **request creates a job** or **customer** within the system. Finally, when a job has been processed, the **system responds to the environment by completing the corresponding request**.

4.3.3.1 Basic measurements

From an abstract system we can derive the following quantities:

- **T**, the **length of time** we observe the system
- **A**, the number of request **arrivals** we observe
- **C**, the number of request **completions** we observe
- **B**, the total amount of time during which the system is **busy** ($B \leq T$)
- **N**, the average **number of jobs** in the system

From these values, we can derive the following four important **quantities**:

- **Arrival rate**:

$$\lambda = \frac{\text{number of request arrivals}}{\text{length of time we observe the system}} = \frac{A}{T} \tag{74}$$

- **Throughput** or **Completion rate**:

$$X = \frac{\text{number of request completions}}{\text{length of time we observe the system}} = \frac{C}{T} \tag{75}$$

- **Utilization**:

$$U = \frac{\text{total amount of time during which the system is busy}}{\text{length of time we observe the system}} = \frac{B}{T} \tag{76}$$

- **Mean service time** per completed job:

$$S = \frac{\text{total amount of time during which the system is busy}}{\text{number of request completions}} = \frac{B}{C} \quad (77)$$

We will assume that the **system is job-flow balanced**. Then the **number of arrivals is equal to the number of completions** during an observation period.

Note that if the system is job flow balanced, then the **arrival rate is equal to the completion rate (throughput)**:

$$\lambda = X$$

A **system** can be thought of as **consisting** of a number of devices or **resources**. Each of these can be treated as a **separate system** from the perspective of operational laws.

An **external request generates a job** within the system; this job may **then circulate among the resources** until all the necessary processing has been done; as it arrives at each resource, it is treated as a request, generating a job internal to that resource.

In this case, we have the following quantities:

- $\mathbf{T}$, the **length of time** we observe the system
- $\mathbf{A}_k$, the number of request **arrivals** we observe for resource k
- $\mathbf{C}_k$, the number of request **completions** we observe at resource k
- $\mathbf{B}_k$, the total amount of time during which the resource k is **busy** ($B_k \leq T$)
- $\mathbf{N}_k$, the average **number of jobs** in the resource k

And we can derive the following four quantities for resource k:

- **Arrival rate**:

$$\lambda_k = \frac{A_k}{T} \quad (78)$$

- **Throughput** or **Completion rate**:

$$X_k = \frac{C_k}{T} \quad (79)$$

- **Utilization**:

$$U_k = \frac{B_k}{T} \quad (80)$$

- **Mean service time** per completed job:

$$S_k = \frac{B_k}{C_k} \quad (81)$$

4.3.3.2 Utilization Law

Using the formulas:

- Throughput: $X_k = \frac{C_k}{T}$
- Mean service time: $S_k = \frac{B_k}{C_k}$
- Utilization: $U_k = \frac{B_k}{T}$

From:

$$X_k \cdot S_k = \frac{C_k}{T} \cdot \frac{B_k}{C_k} = \frac{B_k}{T} = U_k$$

We can derive the **Utilization Law**:

$$U_k = X_k \cdot S_k \tag{82}$$

4.3.3.3 Little's Law

The **Little's Law** is:

$$N = X \cdot R \tag{83}$$

In other words, N is equal to the **number of requests in the system**. Little's Law can be applied to the entire system as well as to some subsystems.

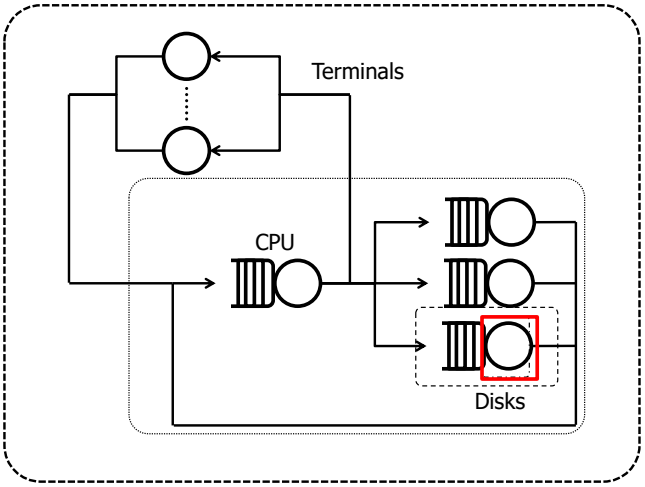
If the system throughput is X requests/sec, and each request remains in the system on average for R seconds, then for each unit of time, we can observe on average XR requests in the system.

Example 14

Consider a disk that serves 40 requests/seconds ($X = 40$ req/s) and suppose that on average there are 4 requests ($N = 4$) present in the disk system (waiting to be served or in service).
Little's Law tell that $N = XR \Rightarrow R = \frac{N}{X}$, so the average time spent at the disk by a request must be $\frac{4}{40} = 0.1$.
If we know S (e.g.) each request requires 0.0225 seconds of disk service and we can then deduce that the average waiting time (time in the queue) is 0.0775 seconds.

The value of Little's Law changes depending on the application:

- **Service level**.
  - The service time is:
    $$R = S$$
    Where R is the average of each request remaining in the system.
  - The utilization is:
    $$N = X \cdot R = X \cdot S = U$$

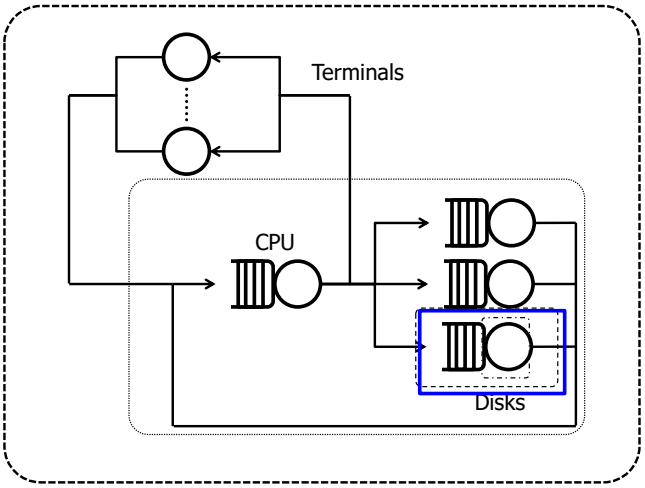


- **Service and Queue level**.
  - The service time is:

$$R = S$$

    Where R is the average of each request remaining in the system.
  - The utilization is:

$$N = \text{Flying requests} \Rightarrow \frac{N}{X} = R = (S + T_{\text{queue}})$$



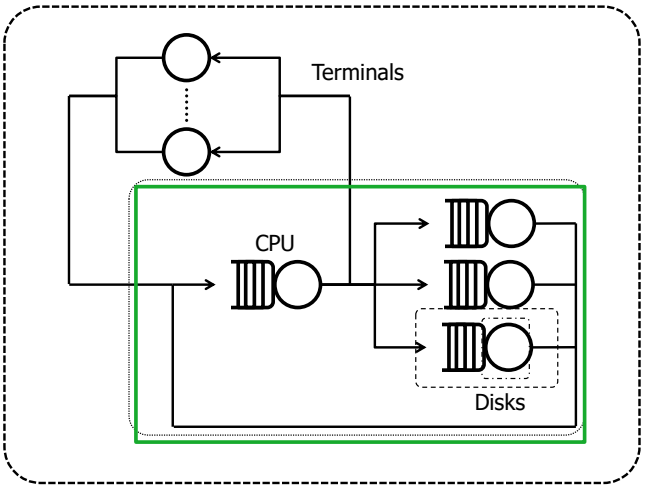
- **Subsystem level**.
  - The service time is:

$$R = \text{Residence Time}$$

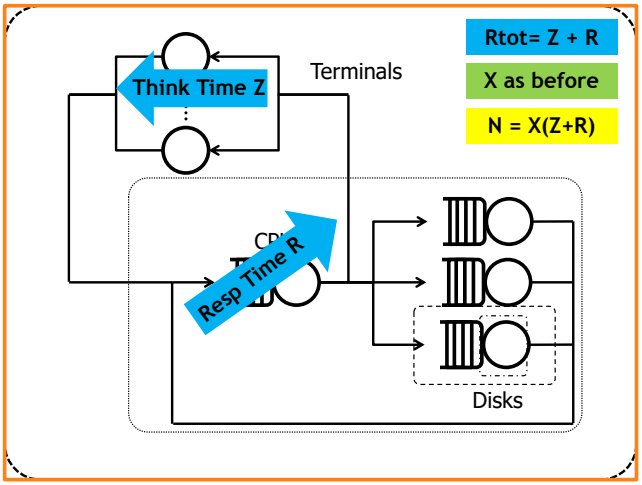
    Residence time corresponds to our conventional notion of response time: the period of time from when a user submits a request until that user's response is returned.

– The utilization is:

$$N = \text{Flying requests} \Rightarrow \frac{N}{X} = R$$



- **System level**.



4.3.3.4 Interactive Response Time Law

The **Interactive Response Time Law** is:

$$R = \frac{N}{X} - Z \tag{84}$$

The response time in an interactive system is the **residence time minus the think time**. Note that **if the think time is zero** ($Z = 0$) and $R = \frac{N}{X}$, then the **interactive response time law simply becomes Little's Law**.

Example 15

Suppose that the library catalogue system has **64 interactive users** connected via Browsers, the average **think time is 30 seconds**, and that system **throughput is 2 interactions/second**. What is the response time?
The interactive response time law tell us that the response time must be $\frac{64}{2} - 30 = 2$ seconds.

4.3.3.5 Visit count

In an observation interval we can count not only completions external to the system, but also the number of completions at each resource within the system. We denote C_k by the **number of completions at resource** k. We define the **Visit Count**:

$$V_k = \frac{C_k}{C} \tag{85}$$

It is the ratio of the number of completions at the k-th resource to the number of system completions.

Example 16

If, during an observation interval, we measure **10 system completions** and **150 completions at a specific disk**, then on average each system-level request requires **15 disk operations**.

Note that:

- If $C_k > C$, resource k is **visited several times** (on average) during each system level request. This happens when there are **loops in the model**.
- If $C_k < C$, resource k **might not be visited** during each system level request. This can happen if there are **alternatives** (e.g. caching of disks).
- If $C_k = C$, resource k is **visited** (on average) **exactly once every request**.

4.3.3.6 Forced Flow Law

The **Forced Flow Law** captures the **relationship between the different components within a system**. It states that the throughputs, or flows, in all parts of a system must be proportional to each other.

$$X_k = V_k \cdot X \tag{86}$$

The throughput at the k-th resource is equal to the product of the throughput of the system and the visit count at that resource.

Rewriting $C_k = V_k \cdot C$ and applying $X_k = \frac{C_k}{T}$, we can derive the forced flow law:

$$C_k = V_k \cdot C \Rightarrow \frac{C_k}{T} = \frac{V_k \cdot C}{T} \Rightarrow X_k = V_k \cdot X$$

4.3.3.7 Utilization Law with Service Demand

If we know the amount of processing each job requires at a resource then we can calculate the utilization of the resource.

Let us assume that each time a job visits the k-th resource, the amount of processing or service time it requires is S_k.

Note that **service time is not the same as the residence time** of the job at that resource. In general a job might have to **wait** for some time **before processing** being.

The **total amount of service that a system job generates at the k-th resource** is called the **Service Demand D_k**:

$$D_k = S_k \cdot V_k \tag{87}$$

Using the service demand, we can rewrite the **Utilization Law**:

$$U_k = X_k \cdot S_k = (X \cdot V_k) \cdot S_k = D_k \cdot X \tag{88}$$

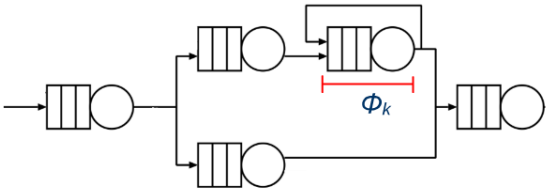
The utilization of a resource is denoted U_k and it is the percentage of time that the k-th resource is in use processing a job. It is also equal to the product of:

- The throughput of that resource and the average service time at that resource;
- The throughput at system level and the average service demand at that resource.

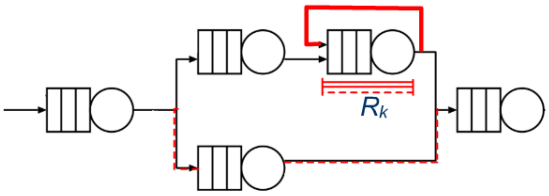
4.3.3.8 Response and Residence Times

When considering nodes characterized by visits different from one, we can define two permanence times: Response Time and Residence Time.

The **Response Time** $\tilde{R}_k$ (or Φ_k) accounts for the **average time spent in station** k, when the **job enters the corresponding node** (i.e. time for the single interaction, disk request):



The **Residence Time** R_k accounts instead for the **average time spent by a job at station** k **during the staying in the system**: it can be greater or smaller than the response time depending on the number of visits.



Note that there is the same relation between Residence Time and Response Time as the one between **Demand Time** and **Service Time**:

$$\begin{aligned} D_k &= v_k \cdot S_k \\ R_k &= v_k \cdot \tilde{R}_k \end{aligned} \tag{89}$$

Also note that for **single queue open system**, or *tandem models*, $v_k = 1$. This implies that **average service time and service demand are equal, and response time and residence time are identical**.

$$v_k = 1 \Rightarrow \begin{aligned} D_k &= S_k \\ R_k &= \tilde{R}_k \end{aligned}$$

4.3.4 Bounding Analysis

4.3.4.1 Introduction

The simplest useful approach to computer system analysis using queueing network models is **Bounding Analysis**. With very little computation it is possible to **determine upper and lower bounds on system throughput and response time as functions of the system workload intensity** (number of arrival rate of customers).

Advantages

- **Highlight** and **quantify** the critical influence of the system **bottleneck**[11].
- Can be **computed quickly**, even by hand.
- Useful in **System Sizing**.
- Useful for **System Upgrades**.

The notation used is:

- **K**, the **number of service centers**.
- **D**, the **sum of the service demands at the centers**, so:

$$D = \sum_k D_k \tag{90}$$

- $\mathbf{D_{max}}$, the **largest service demand at any single center**.
- **Z**, the **average think time**, for interactive systems.

And the following **performance quantities** are considered:

- **X**, the **system throughput**.
- **R**, the **system response time**.

[11] **The resource within a system which has the greatest service demand is known as the bottleneck resource** or **bottleneck device**, and its service demand is $\max_k \{D_k\}$, denoted $D_{\max}$.

The bottleneck resource is important because it limits the possible performance of the system. This will be the resource which has the highest utilization in the system.

4.3.4.2 Asymptotic bounds

The **Asymptotic Bounds** are derived by considering the (asymptotically) extreme conditions of light and heavy loads. There are two possible views:

- **Optimistic**
  - Upper bound: *system throughput*
  - Lower bound: *system response time*
- **Pessimistic**
  - Upper bound: *system response time*
  - Lower bound: *system throughput*

The **extreme conditions** used are:

- Light load.
- Heavy load.

The bounding analysis **assumes** that a customer's service **demand** at a center **does not depend on how many other customers are currently in the system** or in **which service centers they are located**.

Open Models

For the **open models**, the X (*system throughput*) bound is equal to the **maximum arrival rate** that the system can handle.

If the λ (arrival rate, page 170) is greater than the X (system throughput) $\lambda > X$, then the **system is saturated**. This situation **causes new jobs to wait indefinitely**.

The X (*system throughput*) bound is calculated as:

$$\lambda_{\text{sat}} = \frac{1}{D_{\max}} \tag{91}$$

The R (*system response time*) bound is equal to the largest and smallest possible system response time experienced at a given arrival rate (λ), which is only examined if $\lambda < \lambda_{\text{sat}}$. If the condition is **not satisfied**, then the **system is unstable**.

With the open models, there are two **extreme situations** to consider:

- If **no customers interferes with any other**, so **no queue time**, then the system response time is equal to the sum of the service demands at the centers $\mathbf{R} = \mathbf{D}$, with $D = \sum_k D_k$.

- If n **customers arrive together** every $\frac{n}{\lambda}$ time units, there is **no pessimistic bound** on R (system response time).

  Customers at the end of the batch are forced to queue for customers at the front of the batch, and thus experience large response times. The **batch can be extremely long** $N \to \infty$.

  There is no pessimistic bound on response times, regardless of how small the arrival rate λ might be.

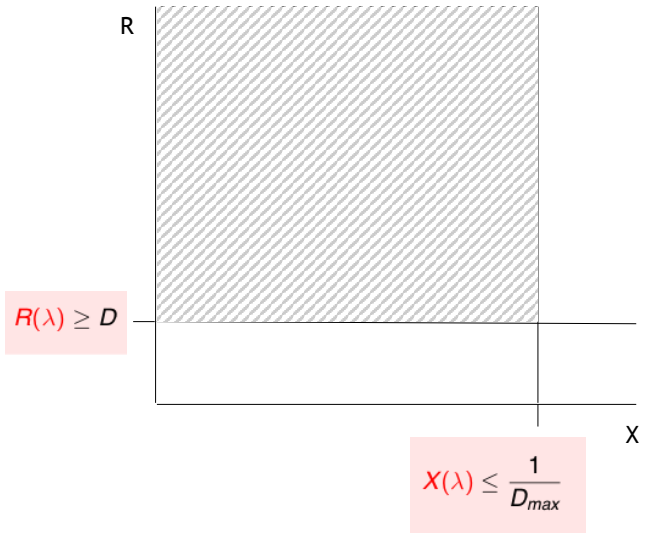
The bounding analysis for open models gives the following formulas:

- Bound for $X(\lambda)$:

$$X(\lambda) \leq \frac{1}{D_{\max}} \tag{92}$$

- Bound for $R(\lambda)$:

$$R(\lambda) \geq D_{\max} \tag{93}$$



Closed Models

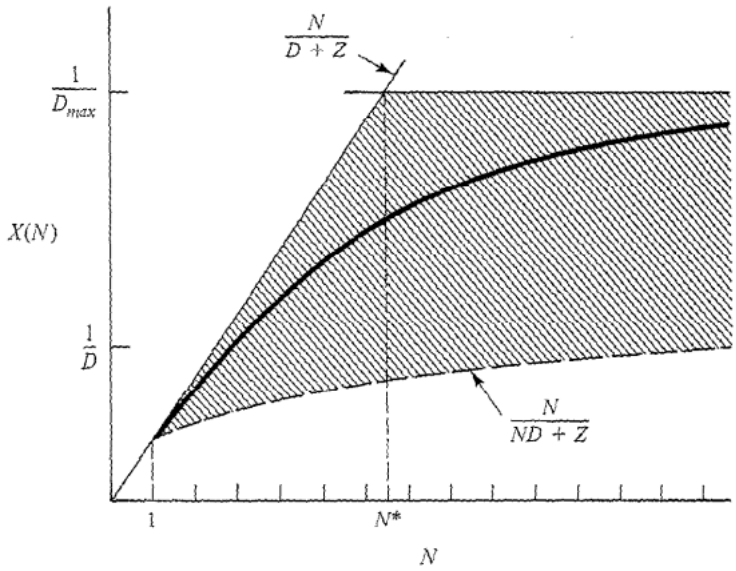
Bounding Analysis depends on the situation:

- **Light Load** situation.
  - **Lower bounds**:
    - **1 customer** case:

$$\begin{aligned} N &= X \cdot (R + Z) \\ 1 &= X \cdot (D + Z) \\ X &= \frac{1}{(D + Z)} \end{aligned} \tag{94}$$

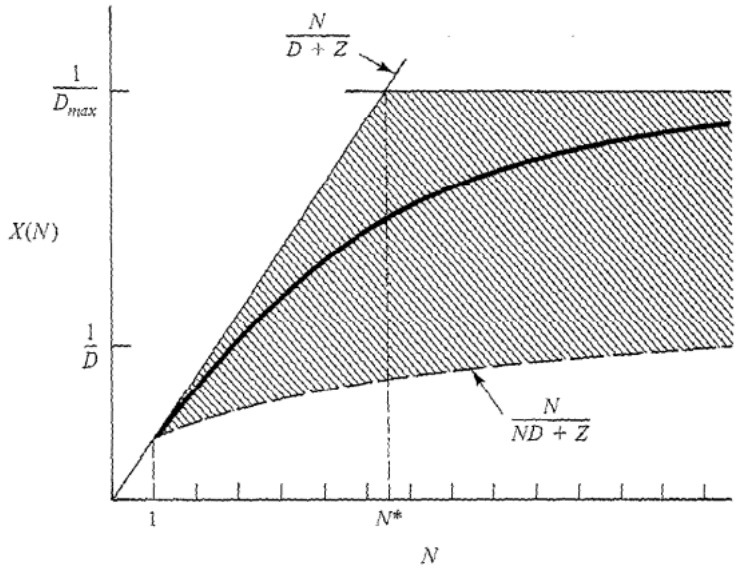
    - **Adding customers**: smallest X (*system throughput*) obtained with largest R (*system response time*), i.e., new jobs queue behind others already in the system.
      Remember: in closed models, the highest possible system response time occurs when each job, at each station, found all the other $N-1$ costumers in front of it, then $R = N \cdot D$.
      In this case $R = N \cdot D$ and X is:

$$\begin{aligned} X &= \frac{N}{(N \cdot D + Z)} \\ \lim_{N \to \infty} \frac{N}{(N \cdot D + Z)} &= \frac{1}{D} \end{aligned} \tag{95}$$



- **Upper bounds**:
  - **Adding customers**: largest X (*system throughput*) obtained with the lowest response time R (*system response time*, i.e. no conflicts).
    Remember: in closed models, the lowest response time can be obtained if a job always finds the queue empty and always starts being served immediately.
    In this case $R = D$ and X is:

$$X = \frac{N}{(D + Z)} \tag{96}$$



- **Heavy Load** situation.
  - **Upper bound**:

$$U_k(N) = X(N) D_k \leq 1 \tag{97}$$

Since the first to saturate is the **bottleneck** (max):

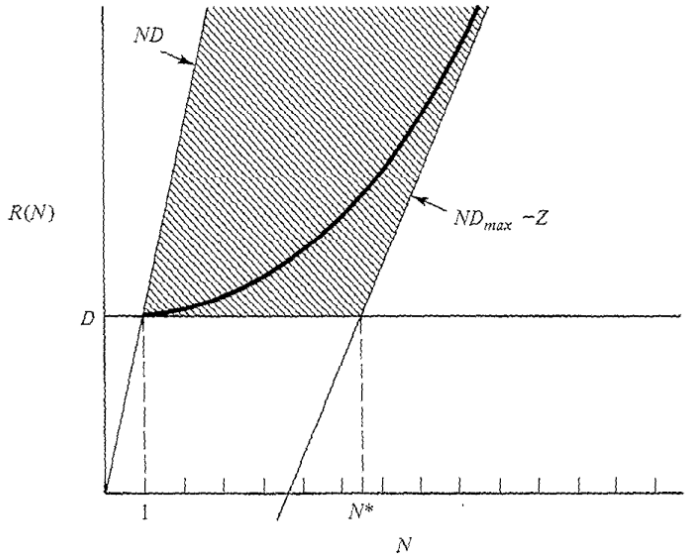
$$X(N) \leq \frac{1}{D_{\max}} \tag{98}$$

The $X(N)$ bounds is:

$$\frac{N}{N \cdot D + Z} \leq X(N) \leq \min\left(\frac{1}{D_{\max}}, \frac{N}{D + Z}\right) \tag{99}$$

The $R(N)$ bounds is:

$$\max(D, N \cdot D_{\max} - Z) \leq R(N) \leq N \cdot D \tag{100}$$

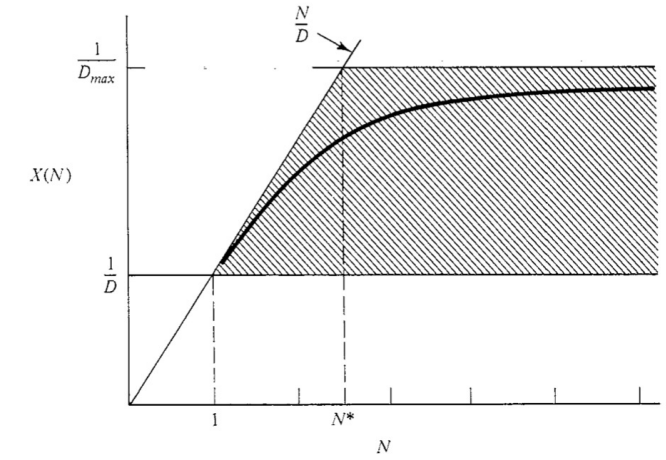


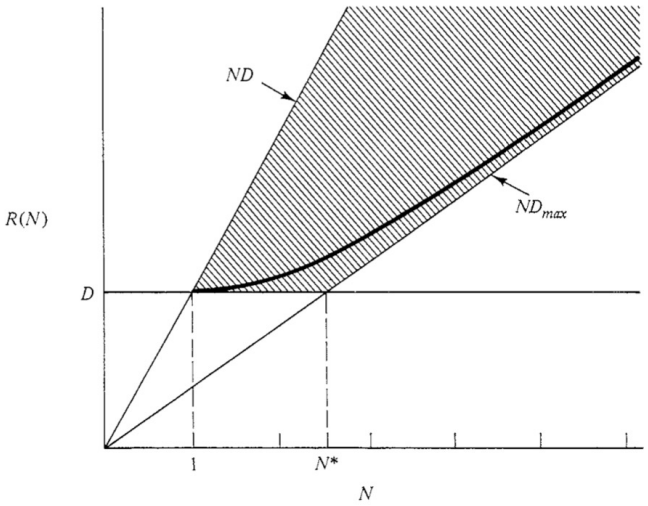
$R(N)$ bounds.

- **N*** case: particular population size determining if the light or heavy load optimistic bound is to be applied:

$$N^* = \frac{D + Z}{D_{\max}} \tag{101}$$

Without thinking time we have the following limits:





References

[1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM computer communication review*, 38(4):63–74, 2008.

[2] L.A. Barroso, U. Hölzle, and P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Springer International Publishing, 2022.

[3] Shantanu Bhattacharya and Lipika Bhattacharya. *Xaas: Everything-As-A-Service: the lean and agile approach to business growth*. World Scientific, 2022.

[4] E.D. Lazowska. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[5] NIST. 8.1.8.4. R out of N model - itl.nist.gov. https://www.itl.nist.gov/div898/handbook/apr/section1/apr184.htm. [Accessed 14-08-2024].

[6] Gianluca Palermo. Computing infrastructures. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[7] Gianluca Palermo. Lesson 1, computing infrastructures. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[8] David A. Patterson, Garth A. Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (raid). Technical Report UCB/CSD-87-391, Berkeley, Dec 1987.

[9] Jiang Wu, Zhou Lei, Shengbo Chen, and Wenfeng Shen. An access control model for preventing virtual machine escape attack. *Future Internet*, 9(2):20, 2017.

Index

S

X