Computing Infrastructures - Notes - v1.6.0

260236

May 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. [1]
- Quantitative System Performance: Computer System Analysis Using Queueing Network Models. [3]
- Course slides. [5]

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. [1]

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.

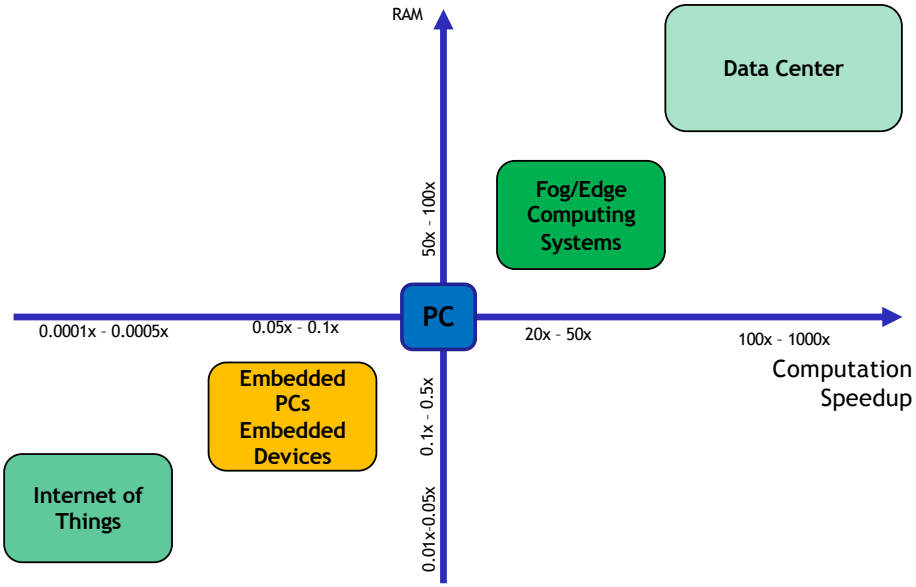
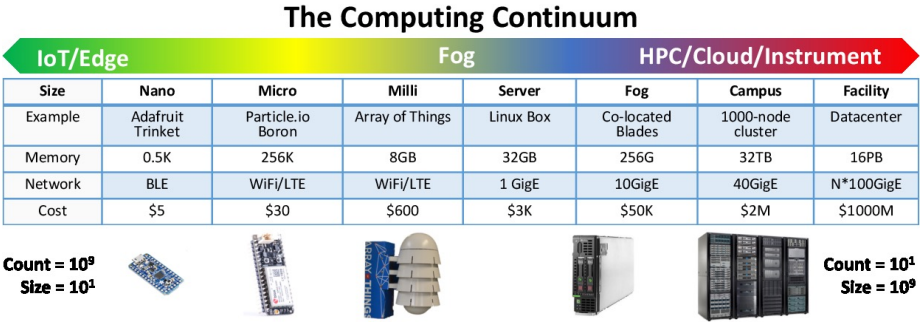


Figure 1: **Examples** of Computing Infrastructures. [6]



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). [6]

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**

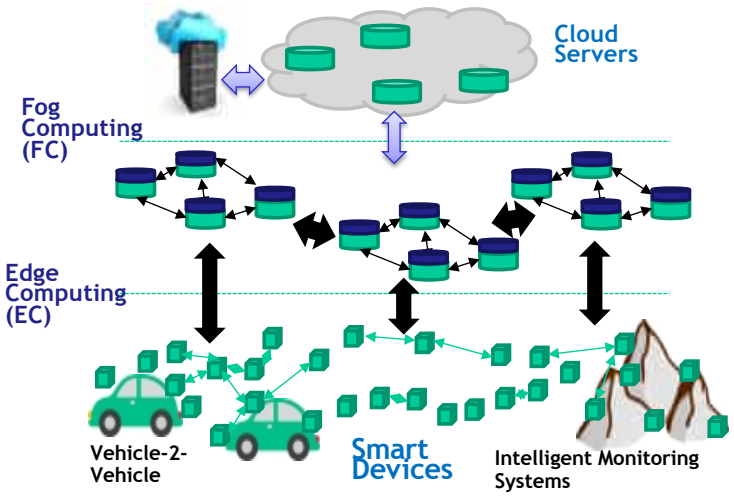
- **Pretty high power consumption**.
- **(Some) Hardware design has to be done**.

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

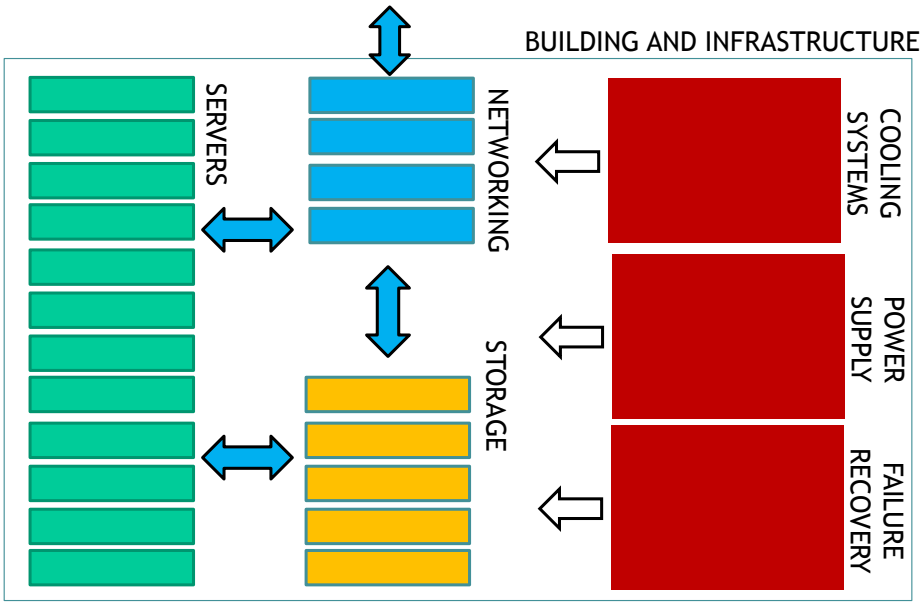


Figure 3: Architectural overview of Warehouse-Scale Computing.

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 71):
    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 74.
  - 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 87.
  - 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 90.
  - **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:

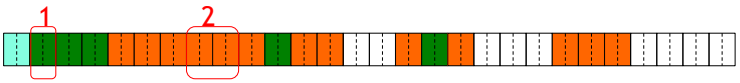
- Metadata:
  - Fixed-position metadata clusters, used to initialize and mount the file system
  - Variable-position metadata clusters, which manage directories and symbolic links
- File data clusters, containing the actual contents of files
- Unused clusters, which are free and available for future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

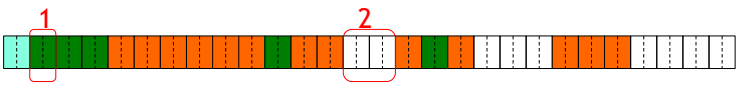
The following explanation introduces some basic operations on the files to see what happens inside the disks.

- **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:

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

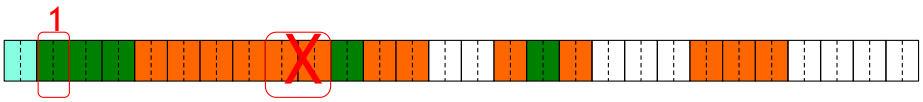
And the internal fragmentation w is:

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

- **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**.

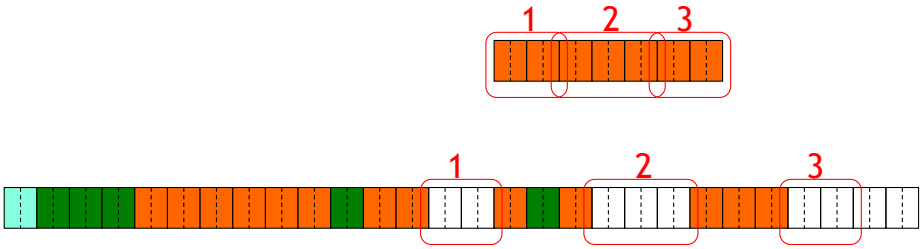


Figure 5: Each number (1, 2, 3) corresponds to a portion (chunk) of a file. A file that was meant to be stored as `[1][2][3]` (contiguously) has instead been broken into parts and stored in a scattered layout across different disk clusters. This situation represents **external fragmentation**, where 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.

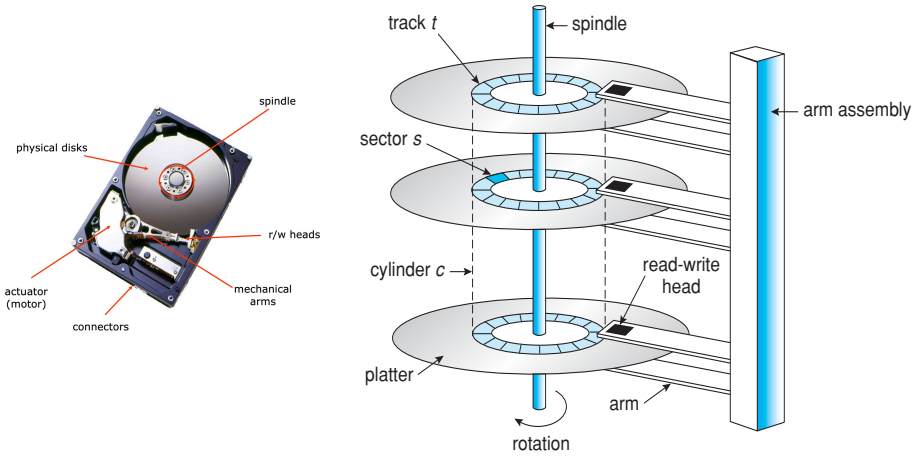


Figure 6: Hard Drive Disk anatomy.

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.

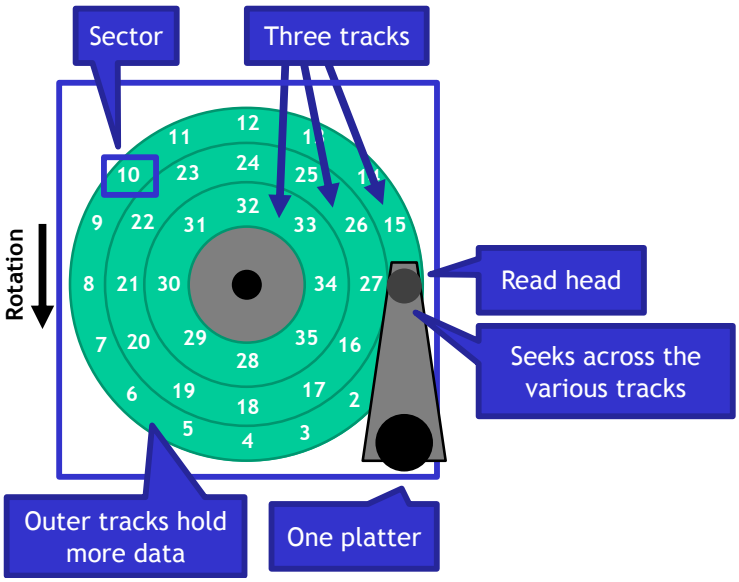


Figure 7: Example of HDD geometry.

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 127.

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**).

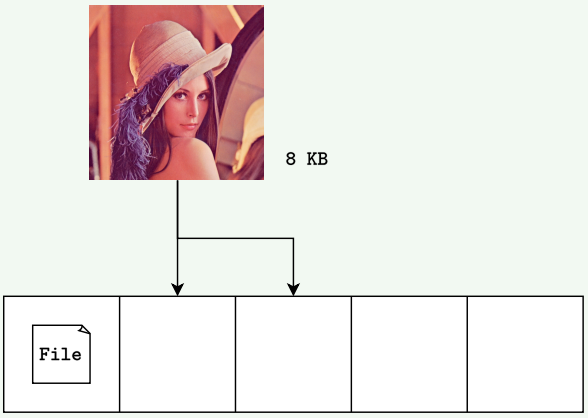
So a 4KB write might cause **hundreds of KBs** to be written internally!

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:

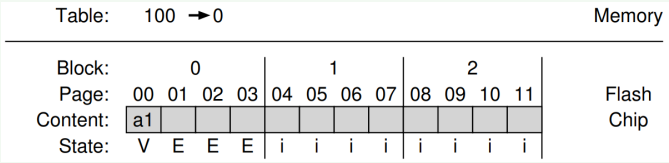
1. The initial state is with all pages marked as `INVALID(i)`:

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

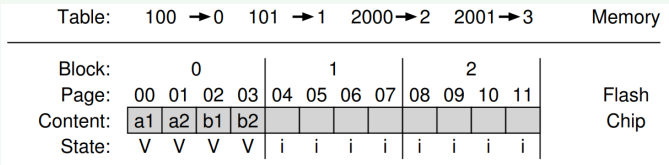
2. Erase block zero:

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

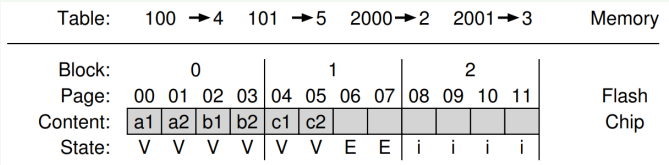
3. Program pages in order and update mapping information (first `Write(100, a1)`):



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*.

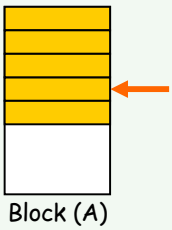
Garbage Collection (GC) is the process of **reclaiming space** by **erasing blocks that contain invalid pages**. It works like this:

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.

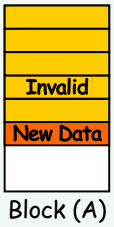
Example 7: how garbage collection works

The steps are:

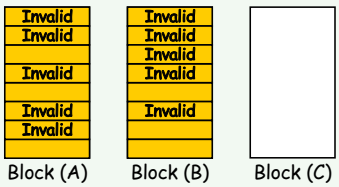
1. Update request for existing data:



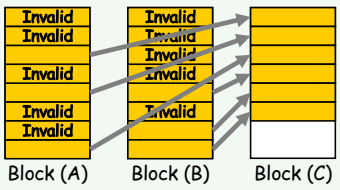
2. Find a free page, and save the new data:



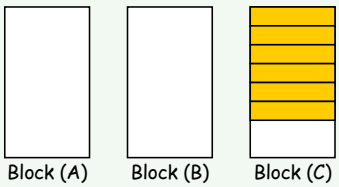
3. This scenario may continue until there are not enough free blocks:



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).

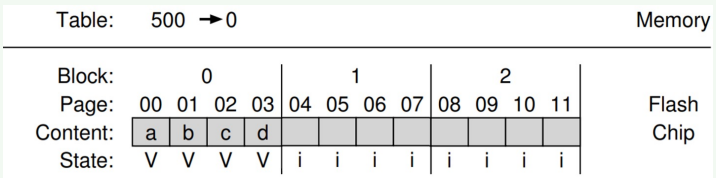
     Cons

     * **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

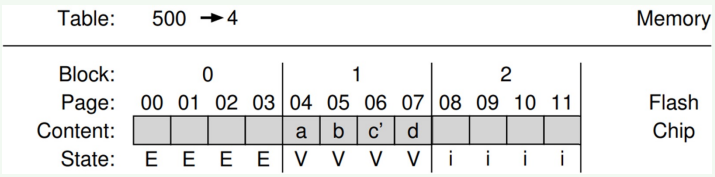
The first four writes:

- `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**.

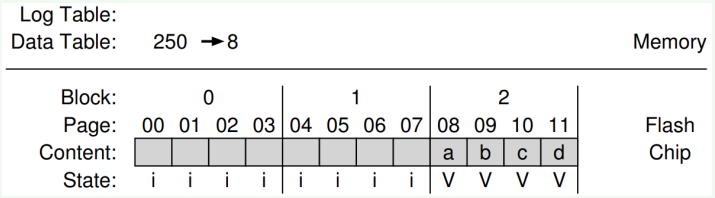
  ✔ **Pros**
  - Lower memory usage than pure page-level
  - Lower write amplification than pure block-level

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

Example 9: Hybrid Mapping

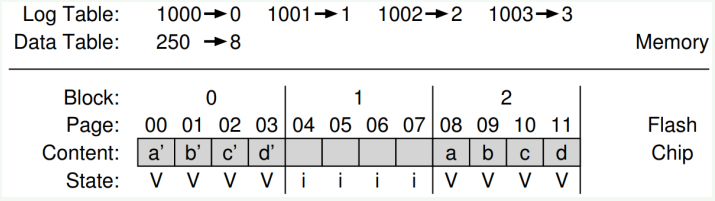
Let's suppose the following sequence:

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

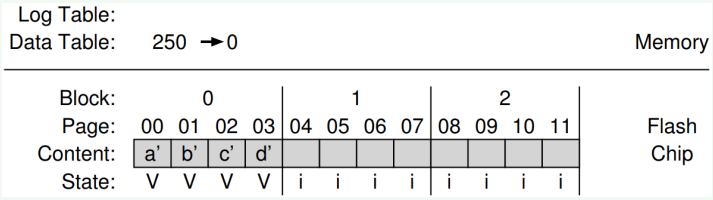


Let's update some pages:

- `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

RAID (Redundant Array of Independent Disks) is a **data storage virtualization technology**[4] that **combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy**, **performance improvement**, or both. This contrasts the previous concept of highly reliable mainframe disk drives, which were referred to as **Single Large Expensive Disks (SLED)**, also called **Just a Bunch of Disks (JBOD)** method where each disk is a separate device with a different mount point.

The data are striped across several disks accessed in parallel:

- **High data transfer rate**: large data accesses (heavy I/O operations).
- **High I/O rate**: small but frequent data accesses (light I/O operations).
- **Load Balancing** across the disks.

Two techniques exist to guarantee these features: **data striping** (improve performance) and **redundancy** (improve reliability).

Data Striping is the technique of **segmenting logically sequential data**, such as a file, so that **consecutive segments are stored on different physical storage devices**. A small quantity of terminology:

- **Striping**: **data are written sequentially** (a vector, a file, a table, etc) in units (stripe units such as bit, byte, and blocks) **on multiple disks** according to a cyclic algorithm (round robin).
- **Stripe unit**: the **dimension of the data unit written on a single disk**.
- **Stripe width**: number of **disks considered by the striping algorithm**:
  1. **Multiple independent I/O requests** will be executed in parallel by several disks, decreasing the disks' queue length (and time).
  2. Multiple disks will execute **single Multiple-Block I/O requests** in parallel, increasing the transfer rate of a single request.

The **redundancy technique is introduced because** the more physical disks in the array, the more significant the size and performance gains, but the **larger the probability of failure of a disk**.

In fact, the *probability of a failure* (assuming independent failures) in an array of 100 disks is 100 higher than the probability of a failure of a single disk! For **example**, if a disk has a **Mean Time To Failure (MTTF)** of 200.000 hours (23 years), an array of 100 disks will show an MTTF of 2000 hours (3 months).

[4] I/O virtualization: data are distributed transparently over the disks, then no action is required of the users.

The **Redundancy** is the **technique of data duplication or error correcting codes** (stored on disks different from the ones with the data) **that are computed to tolerate loss due to disk failures**. Since write operations must also update the redundant information, their **performance is worse than traditional writing**.

Data is distributed across the drives in one of several ways, referred to as **RAID levels**, depending on the required level of redundancy and performance. The different schemes, or data distribution layouts, are named by the word *RAID* followed by a number, for example RAID 0 or RAID 1. Each scheme, or RAID level, provides a different balance among the key goals: reliability, availability, performance, and capacity. The RAID levels are:

- `RAID 0` striping only
- `RAID 1` mirroring only
  - `RAID 0+1` nested levels
  - `RAID 1+0` nested levels
- `RAID 2` bit interleaving (not used)
- `RAID 3` byte interleaving - redundancy (parity disk)
- `RAID 4` block interleaving - redundancy (parity disk)
- `RAID 5` block interleaving - redundancy (parity block distributed)
- `RAID 6` greater redundancy (2 failed disks are tolerated)

Note: these notes do not study the levels `RAID 2` and `RAID 3`.

Table 6: RAID - Table of Contents.

RAID 0

In RAID 0, the data are **written on a single logical disk and split into several blocks distributed across the disks** according to a striping algorithm.

When is it used?

It is used where **performance** and **capacity** are the primary concerns. These mean that a minimum of two drives is required.

Advantages

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

Disadvantages

Single disk failure will result in data loss.

How does it work?

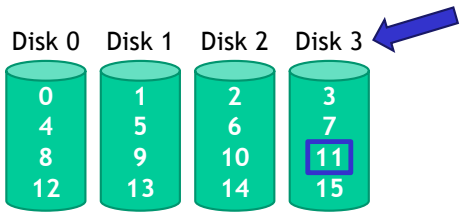
The key idea is to present an **array of disks as a single large disk** and maximize parallelism by striping data across all N disks.

Now, we will give some metrics to understand how it is possible to **calculate access to a specific data block** and compare the **performance** of different RAID technologies.

To **access to a specific data blocks**, the formulas are:

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

For example, given the following schema:



If it is requested, the logical block is 11, and the disks are 4:

$$\begin{aligned} \texttt{Disk} &= 11 \% 4 = 3 \\ \texttt{Offset} &= 11 \div 4 = 2.75 \approx 3, \text{ then physical block 2 starting from 0} \end{aligned}$$

Note that the **chunk size is critical** because it impacts disk array performance. With **smaller chunks**, there is **greater parallelism** than with **big chunks**, which have **reduced seek times**. The typical arrays use 64 KB chunks.

To measure RAID **performance**, we focus on sequential and random workloads. Assume disks in the array have sequential transfer rate S, and the info about the disk is:

- Average seek time: 7 ms
- Average rotational delay: 3 ms
- Transfer rate: 50 MB/s

For a single large transfer (10 MB):

$$S = \frac{\texttt{transfer\_size}}{\texttt{time\_to\_access}}$$

$$S = 10\ \text{MB} \div (7\ \text{ms} + 3\ \text{ms} + 10\ \text{MB} \div 50\ \text{MB/s}) = 47.62\ \text{MB/s}$$

If the disks in the array have a random transfer rate R, and for a set of small files (10 KB):

$$R = \frac{\texttt{transfer\_size}}{\texttt{time\_to\_access}}$$

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

Analysis

- **Capacity**: N, where N is the number of disks. Then, everything can be filled with data.
- **Reliability**: non-existent. If any drive fails, data is permanently lost. Then, the Mean Time To Failure (MTTF) equals the **Mean Time To Data Loss (MTTDL)**:
$$\texttt{MTTF} = \texttt{MTTDL}$$
- **Sequential read and write**: $N \times S$
- **Random read and write**: $N \times R$

Where N is the number of disks, S the sequential transfer rate and R the random transfer rate.

RAID 1

Although RAID 0 offers high performance, it has zero error recovery. For this reason, RAID 1 makes **two copies of all data** (again, a minimum of 2 disk drives are required).

When is it used?

It is used when **zero error recovery is not allowed**.

Advantages

- **High reliability**. When a disk fails, the *second copy is used*.
- **Read of a data**. It can be *retrieved* from the *disk with shorter queueing, seek, and latency delays*.
- **Fast writes** (no error correcting code should be computed). But *still slower than standard disks* (due to duplication).

Disadvantages

- **High costs** because only 50% of the capacity is used.

How does it work?

In principle, a RAID 1 can mirror the content over more than one disk. This feature gives resiliency to errors even if more than one disk breaks. Also, it allows a **voting mechanism to identify errors not reported by the disk controller**. Unfortunately, this is **never used in practice because the overhead and costs are too high**.

However, disks could be coupled if several disks are available (always in an even number). Nevertheless, the total capacity is halved, and each disk has a mirror. Then, the question is simple: *How do we organize this architecture?* The answer is the nested RAIDs: `RAID 0 + 1` and `RAID 1 + 0`.

We define the `RAID x + y` (or `RAID xy`) as:

- $n \times m$ disks in total
- m groups of n disks
- Apply `RAID x` to each group of n disks
- Apply `RAID y` considering the m groups as single disks

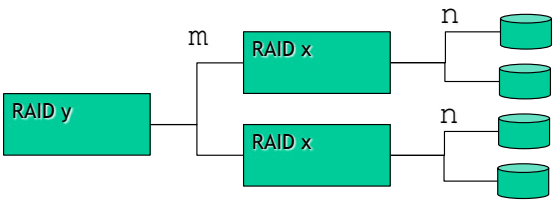
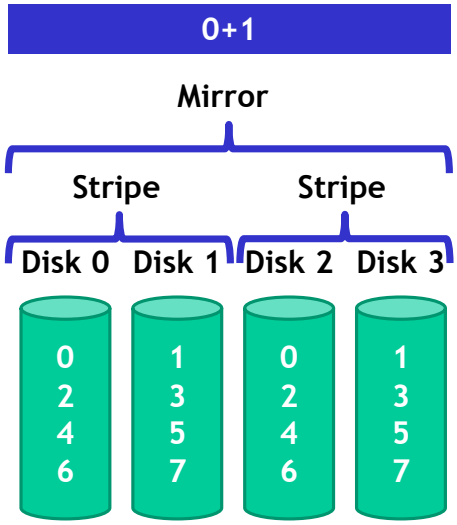


Figure 8: `RAID x + y` general architecture.

The `RAID 0 + 1` is a **group of striped disks (RAID 0) that are then mirrored (RAID 1)**. So:

1. The mirroring first (RAID 1)
2. Then the striping (RAID 0)

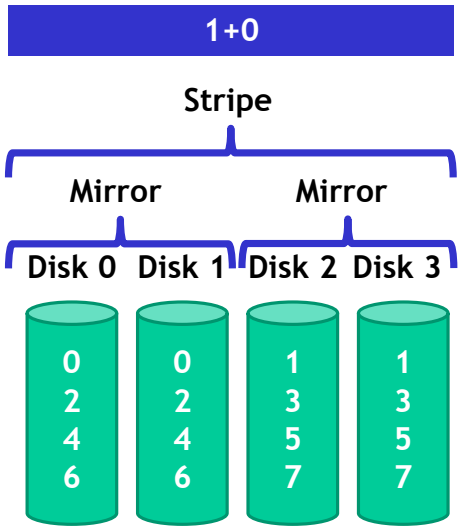
There are necessary almost four drives. Note that after the first failure, the model becomes a RAID 0.



The `RAID 1 + 0` is a **group of mirrored disks (RAID 1) that are then striped (RAID 0)**. So:

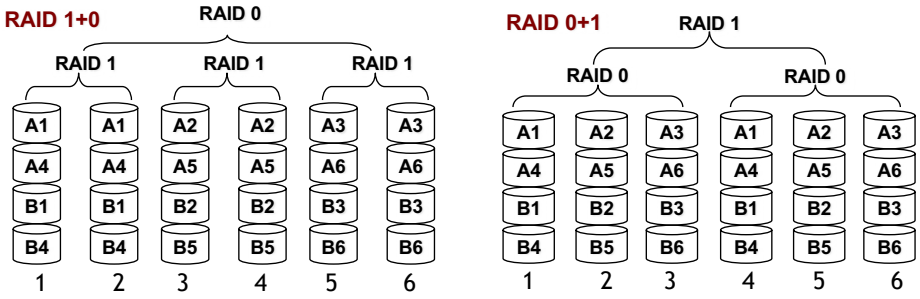
1. The striping first (RAID 0)
2. Then the mirroring (RAID 1)

There are necessary almost four drives. Usually, it is used in databases with very high workloads (fast writes).



≠ Differences between RAID 01 and RAID 10

Look at the following architectures:



What we can say is:

- The performance of RAID 01 and RAID 10 are the same.
- The **main difference is the fault tolerance**[5]!

On most implementations of RAID controllers, **RAID 01 fault tolerance is less**. Since we have only two groups of RAID 0, if two drives (one in each group) fail, the entire RAID 01 will fail. Looking at the architecture above, if Disk 1 and 4 fail, both groups will be down. So, the whole RAID 01 will fail.

On the contrary, RAID 10 since there are many groups (as the individual group is only two disks), even if three disks fail (one in each group), the RAID 10 is still functional. Looking at the architecture above, even if Disk 1, 3, and 5 fail, the RAID 10 will still be functional.

Fault Tolerance: `RAID 01` $\ggg$ `RAID 10`

[5]Fault tolerance is the ability of a system to maintain proper operation despite failures or faults in or more of its components.

So, given a choice between RAID 10 and RAID 01, it should be better to choose RAID 10 to have more fault tolerance.

Analysis

- **Capacity**: $N \div 2$, where N is the number of disks. Then, two copies of all data, thus half capacity.
- **Reliability**: 1 drive can fail, sometimes more! In an optimistic scenario, $N \div 2$ drives can fail without data loss.
- **Sequential write**: $(N \div 2) \times S$; two copies of all data, thus half throughput.
- **Sequential read**: $(N \div 2) \times S$; half of the read blocks are wasted, thus halving throughput.
- **Random read**: $N \times R$; it is the best scenario for RAID 1 because the read can be parallelized across all disks.
- **Random write**: $(N \div 2) \times R$; two copies of all data, thus half throughput.

Where N is the number of disks, S is the sequential transfer rate, and R is the random transfer rate.

It is essential to observe RAID 1. There is a notorious **problem** called the **Consistent Update Problem**.

Mirrored writes should be atomic. Then, all copies are written, or none are written. Unfortunately, this is very **difficult to guarantee** sometimes, for example, in a power failure scenario. To **solve** this problem, many RAID controllers include a **write-ahead log**, a **battery backend**, and **non-volatile storage of pending writes**. With this system, a **recovery procedure ensures the recovery of the out-of-sync mirrored copies**.

RAID 4

RAID 4 consists of **block-level striping with a dedicated parity disk**.

When is it used?

RAID 1 offers highly reliable data storage, but it uses half the space of the array capacity. To achieve the **same level of reliability without wasting capacity**, it is possible to use RAID 4, which uses **information coding techniques to build lightweight error recovery mechanisms**.

Advantages

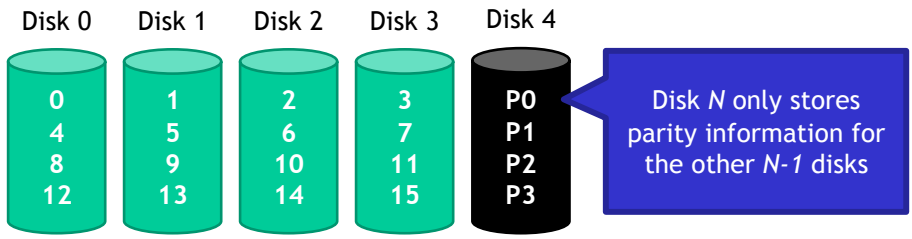
- **Good performance** of random reads.

Disadvantages

- **Random Write performance is terrible** due to being *bottlenecked* by the parity drive.

How does it work?

Disk N only stores parity information for the other $N-1$ disks. The parity is calculated using the `XOR` operation[6].



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

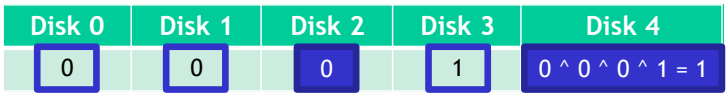
Figure 9: RAID 4 - *How does it work?*

The **serial** or **random read** is not a problem in RAID 4 because there is a **parallelization across all non-parity blocks** in the stripe despite a tiny performance reduction due to the parity disk.

[6] "A or B, but not A and B"

During the parity writing, the blocks are updated. The **random write** performance is affected by the approach used:

- **Additive parity** (as known as reconstruct-writes):
  1. Read all other blocks in a stripe in parallel;
  2. `XOR` those with the new block to form a new parity block;
  3. Write the new data block and new parity block to disks.



- **Subtractive parity** (as known as read-modify-writes):
  1. Read only the old data block to be updated and the old parity block;
  2. Compute the new parity block using:

$$P_{new} = (D_{new} \veebar D_{old}) \veebar P_{old}$$

     Where $\veebar$ is the logical `XOR`.
  3. Write the new data block and new parity block to disks.

Despite the **sequential write** does **not suffer any performance effect** from the parity disk. Because it uses full-stripe write:

1. Buffer all data blocks of a stripe
2. Compute the parity block
3. Write all data and parity blocks in parallel [8]

Analysis

- **Capacity**: $N-1$, where N is the number of disks, and the -1 is because one is dedicated to the parity disk.
- **Reliability**: 1 drive can fail. Massive performance degradation during partial outage.
- **Sequential read/write**: $(N-1)\times S$; parallelization across all non-parity blocks in the stripe (parity disk has no effect).
- **Random read**: $(N-1) \times R$; reads parallelize over all but the parity drive (parity disk has no effect).
- **Random write**: $R \div 2$; writes serialize due to the parity drive, and each write requires one read and one write of the parity drive.

Where N is the number of disks, S is the sequential transfer rate, and R is the random transfer rate.

RAID 5

RAID 5 **rotates parity blocks across stripes**.

When is it used?

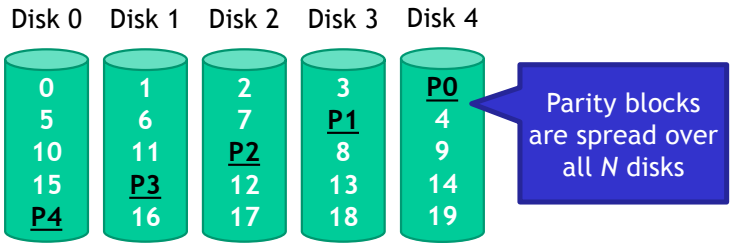
Unlike in RAID 4, parity information is distributed among the drives. This technique is **used to improve significantly the random write throughput** against the RAID 4 system.

Advantages

- Improved random write despite the RAID 4 system.

How does it work?

The writes are spread roughly evenly across all drives.



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

The **random write** in RAID 5 is:

1. Read the target block and the parity block
2. Use subtraction to calculate the new parity block
3. Write the target block and the parity block

Thus, **four total operations** (two reads, two writes) **are distributed across all drives**.

Analysis

- **Capacity**: $N - 1$ (**same as RAID 4**), where N is the number of disks.
- **Reliability**: 1 drive can fail (**same as RAID 4**). Massive performance degradation during partial outage.

- **Sequential read/write**: (N - 1) * S (**same as RAID 4**); parallelization across all non-parity blocks in the stripe (parity disk has no effect).
- **Random read**: $N \times R$; unlike RAID 4, **reads parallelize over all drives**.
- **Random write**: $(N \div 4) \times R$; unlike RAID 4, **writes parallelize over all drives**, and each write requires two reads and two writes, hence $N \div 4$.

Where N is the number of disks, S is the sequential transfer rate, and R is the random transfer rate.

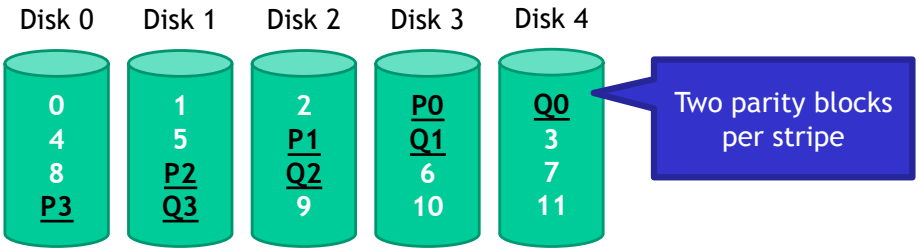
RAID 6

RAID 6 can tolerate multiple disk faults (**more fault tolerance**) concerning RAID 5. It tolerates two concurrent failures.

How does it work?

It uses Solomon-Reeds codes with two redundancy schemes and requires $N + 2$ disks (where N is the number of disks). Note that the **minimum set is 4 data disks**.

Unfortunately, it has a **high overhead for writes** (computation of parities) because each write requires six disk accesses due to the need to update both the P and Q parity blocks (slow writes).



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 7: Comparison of RAID levels.

Where the throughput is:

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

And the latency is:

- Read
- Write

RAID level	Capacity	Reliability	R/W performance	Rebuild performance	Suggested applications
0	100%	N/A	Very good	Good	Non critical data
1	50%	Excellent	Very good / good	good	Critical information
5	(n-1)/n	Good	Good/ fair	Poor	Database, transaction based applications
6	(n-2)/n	Excellent	Very good/ poor	Poor	Critical information, w/minimal
1+0	50%	Excellent	Very good/ good	Good	Critical information, w/better performance

Figure 10: Characteristics of RAID levels.

RAID 0 has the best performance and most capacity.

RAID 1 (10 better than 01) or **RAID 6** have the greatest error recovery.

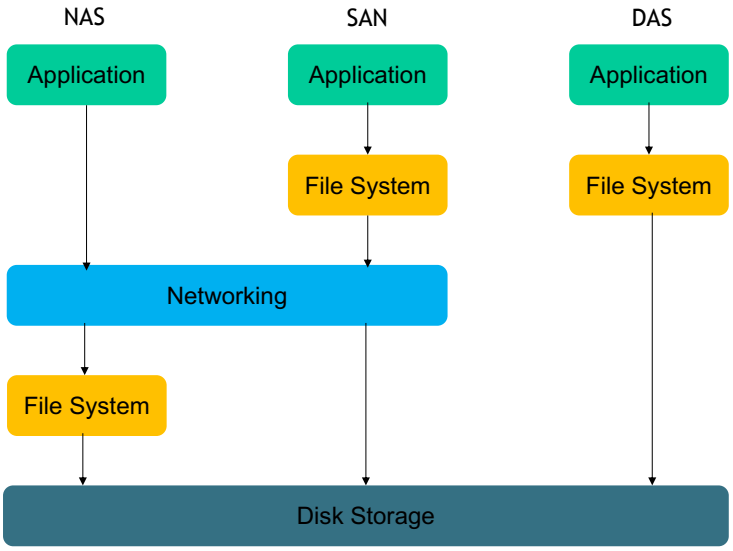
RAID 5 has the better ***balance*** between space, performance, and recoverability.

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).

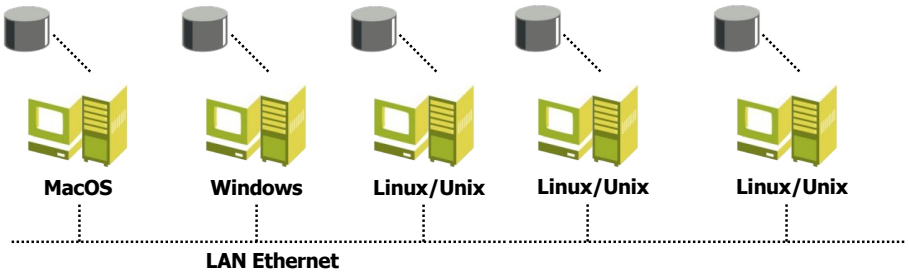


Figure 11: DAS architecture.

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.

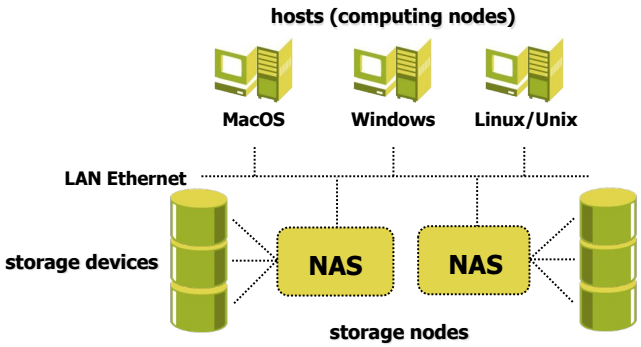


Figure 12: NAS architecture.

✔ 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.

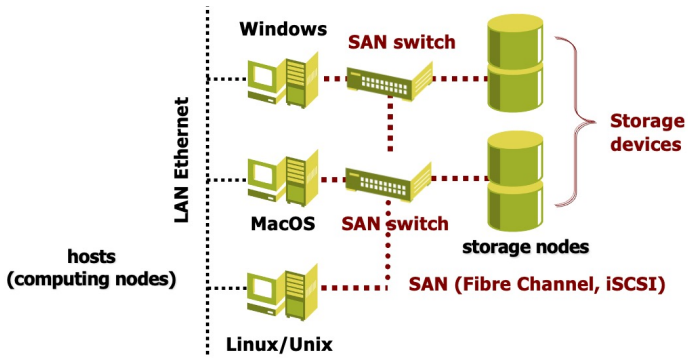


Figure 13: SAN architecture.

	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 14: 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*.

A **solution** can be to double the aggregate compute capacity or the aggregate storage simply by **doubling the number of compute or storage elements**.

The **doubling leaf bandwidth** is used since the networking has no straightforward horizontal scaling solution. Then, with **twice as many servers**, we will have **twice as many network ports and thus twice as much bandwidth**.

What is a bisection bandwidth?

Bisection bandwidth is a **measure of network performance**, **defined as the bandwidth available between two equal-sized partitions when a network is bisected**. This measure accounts for the *bottleneck bandwidth of the entire network*, providing a representation of the actual bandwidth available in the system. The bisection should be done to minimize the bandwidth between the two partitions. It is often used to evaluate and compare networks for parallel architectures, including point-to-point communication systems or on-chip micro-networks.

Assuming that every server needs to talk to every other server, we need to double not just leaf bandwidth but bisection bandwidth.

How to design a data centre network?

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**:

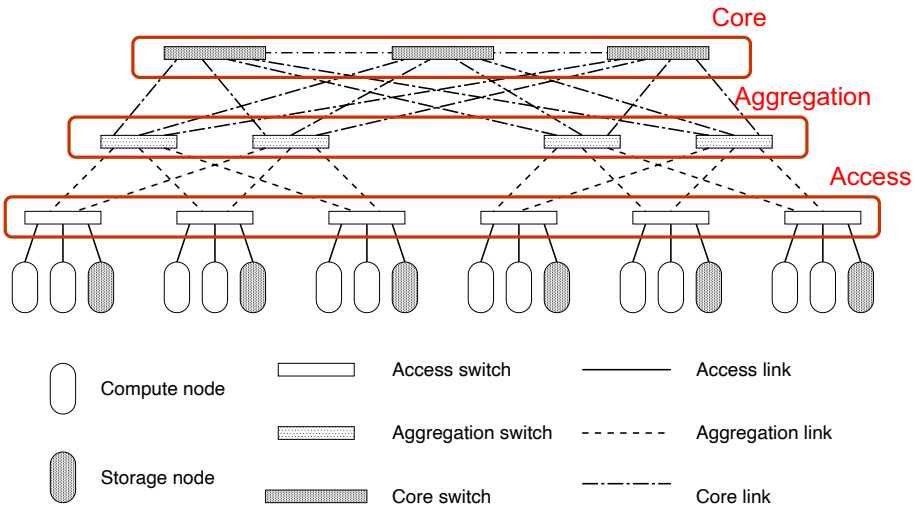
- **DCN Switch-centric architectures**. DCN uses **switches to perform packet forwarding**.[7]
- **DCN Server-centric architectures**. DCN **uses servers with multiple Network Interface Cards (NICs)**[8] **to act as switches** and perform other computational functions.
- **DCN Hybrid architectures**. DCN **combines switches and servers** for packet forwarding.

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

[8] 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**. All servers in a rack are connected to a ToR access switch within the same rack. The aggregation switches are in dedicated racks or shared racks with other ToR switches and servers.

✔ **Advantages**

1. **Simpler cabling** because the number of cables is limited.
2. **Lower costs** because the number of ports per switch is limited.

👎 **Cons**

Higher complexity for switch management.

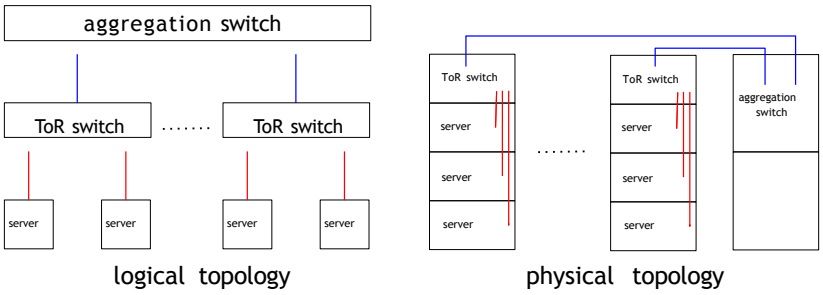


Figure 15: ToR (Top-of-Rack) architecture.

- **EoR (End-of-Row) architecture**. Aggregation switcher are positioned one per corridor, at the end of a line of rack. Servers in a racks are connected directly to the aggregation switch in another rack. Exists a patch panel to connect the servers to the aggregation switch.

✔ **Advantages**

Simpler switch management.

👎 **Cons**

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

1. **Complex cabling**.
2. **Longer cables** then **higher costs**.

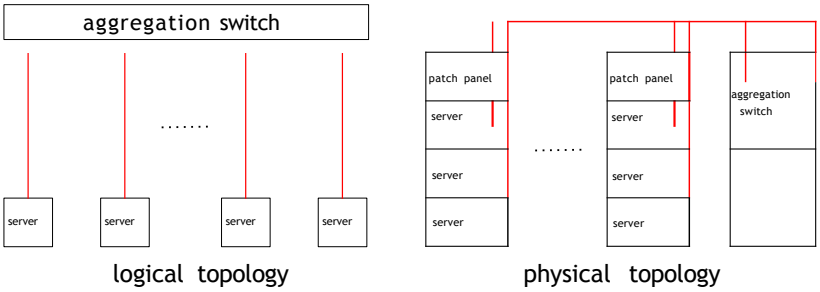


Figure 16: EoR (End-of-Row) architecture.

2.2.3.3 Switch-centric: Leaf-Spine architectures

In the following section we present two Leaf-Spine architectures: the Leaf-Spine model and the Pod-based model (Fat Tree Network).

The **Leaf-Spine architecture** consists of **two levels of interconnection**:

1. The **leaf** (which is a ToR switch);
2. The **spine** (which has dedicated switches, aggregation switches).

In practice, servers have two interfaces connected to two ToR switches to provide fault tolerance.

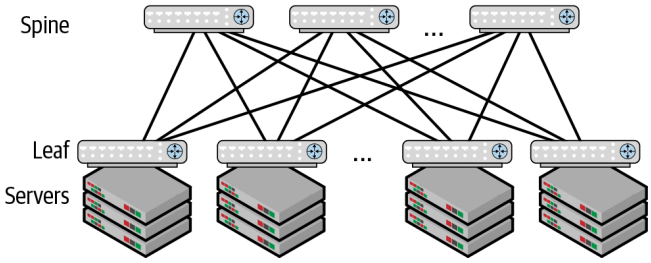


Figure 17: Leaf-Spine architecture.

Now we will explain the Leaf-Spine architecture. If m is the number of **mid-stage switches** and n is the **number of inputs and outputs**, the Leaf-Spine topology is as follows:

- Each switch module is bi-directional.
  - *Leaf* has $2k$ bidirectional ports per module;
  - *Spine* has k bidirectional ports per module.
- Each path traverses either 1 or 3 modules.

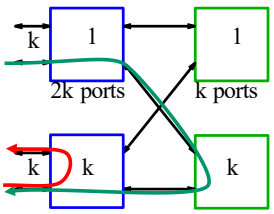


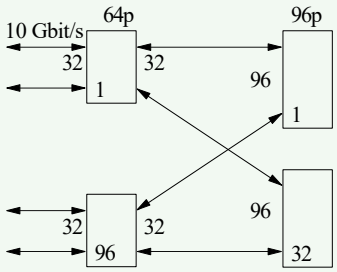
Figure 18: Explanation of Leaf-Spine architecture.

The **advantages** are: use of homogeneous equipment, simple routing, the number of hops is the same for any pair of nodes, small blast radius.

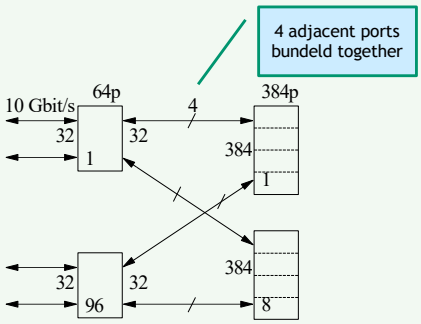
Example 10

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.



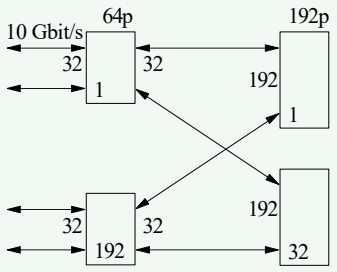
2. The second has only 8 switches but they have more ports: 384 ($8 \times 384 = 3072$).



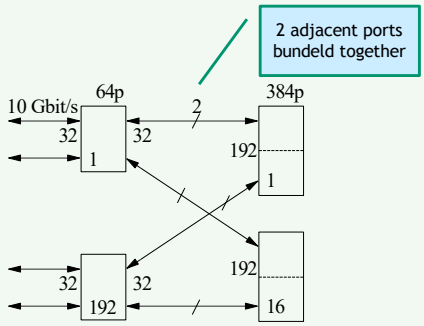
Example 11

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 ($16 \times 384 = 6144$).



The **Pod-based model**, also called **Fat Tree Network**, is another network architecture used to **increase the scaling** feature respecting the leaf-spine.

It transforms each group of spine-leaf into a **PoD (Point of Delivery)**[9] and adds a super spine layer.

It is a **highly scalable** and **cost-effective** DCN architecture designed to **maximise bisection bandwidth**. It can be built using standard Gigabit Ethernet switches with the same number of ports.

It is composed by a ***leaf*** of $2k^2$ bidirectional ports:

- k^2 ports to the servers;
- k^2 ports to the data center network.

In general, let k^2P servers: there are $2kP$ switches with $2k$ ports and k^2 switches with P ports. Using the Fat-Tree model, **the P value is 2k, so for $2k^3$ servers, there are $5k^2$ switches with 2k ports** ($k^2 + 2k \cdot 2k$).

At the **edge layer**, there are $2k$ pods (groups of servers), each with k^2 servers.

- Each edge switch is directly connected to k servers in a pod and k aggregation switches.
- A Fat-Tree network with $2k$-port commodity switches can accomodate $2k^3$ servers in total.
- k^2 core switches with $2k$-port each, each one connected to $2k$ pods.
- Each aggregation switch is connected to k core switches.

[9] A Point Of Delivery is a module or group of network, compute, storage and application components that work together to deliver a network service. The PoD is a repeatable pattern and its components increase the modularity, scalability and manageability of data.

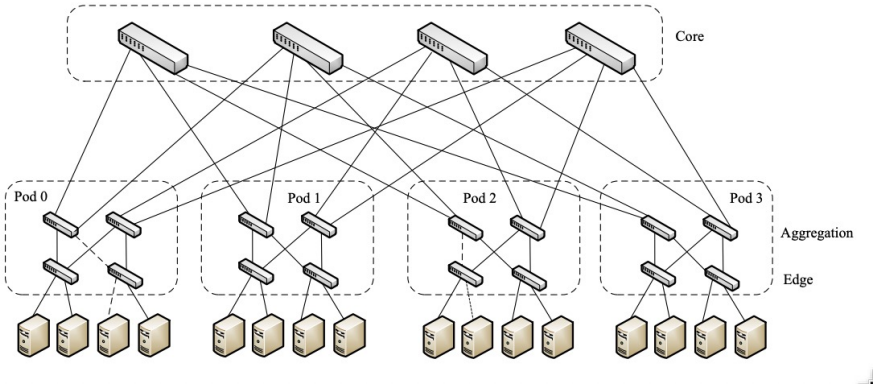


Figure 19: Fat-Tree Network, with $k = 2$, 4 pods, 16 servers, 20 switches.

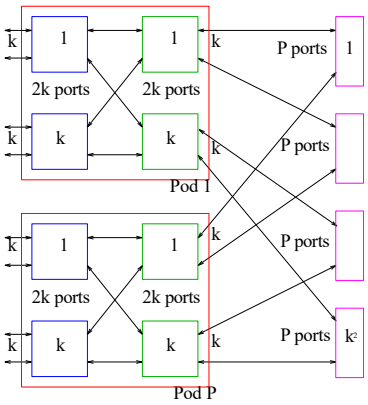


Figure 20: Explanation of Fat-Tree Network.

2.2.3.4 Server-centric and hybrid architectures

CamCube is a **server-centric architecture** typically proposed for building container-sized data centres.

Advantages

It can **reduce implementation and maintenance costs** by using only servers to build the Data Center Network. It also exploits network locality to **increase communication efficiency**.

Disadvantages

It **requires servers with Multiple Network Interface cards** to build a 3D tours network, **long paths** and **high routing complexity**.

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

A **DCell** is a **scalable and cost-effective hybrid architecture** that uses switches and servers for packet forwarding. It is a recursive architecture and uses a basic building block called $DCell_0$ to construct larger DCells.

$DCell_k$ ($k > 0$) denotes a level-k DCell **constructed by combining** $n + 1$ **servers** in $DCell_0$. A $DCell_0$ has n ($n < 8$) **servers directly connected by a commodity switch**.

Disadvantages: **long communication paths**, many required Network Interface Cards and **increased cabling costs**.

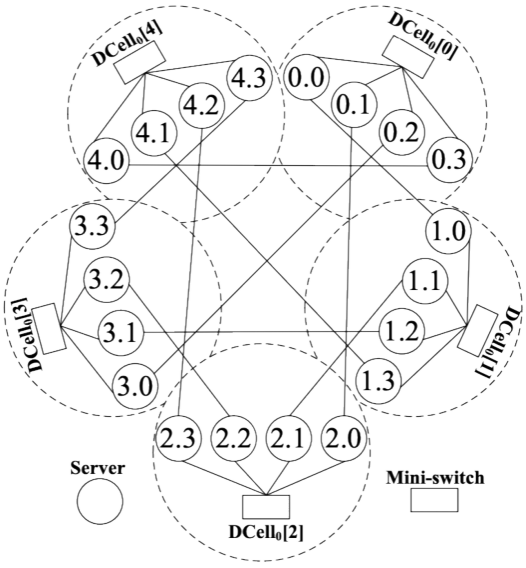


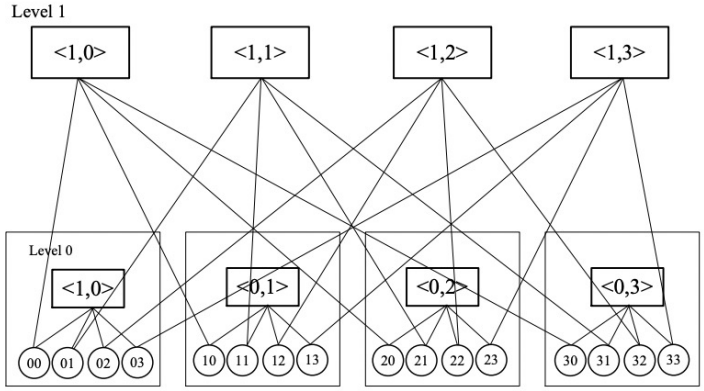
Figure 21: DCell hybrid architecture.

BCube is a **hybrid and cost-effective architecture** that scales through recursion. It provides **high bisection bandwidth** and **graceful throughput degradation**.

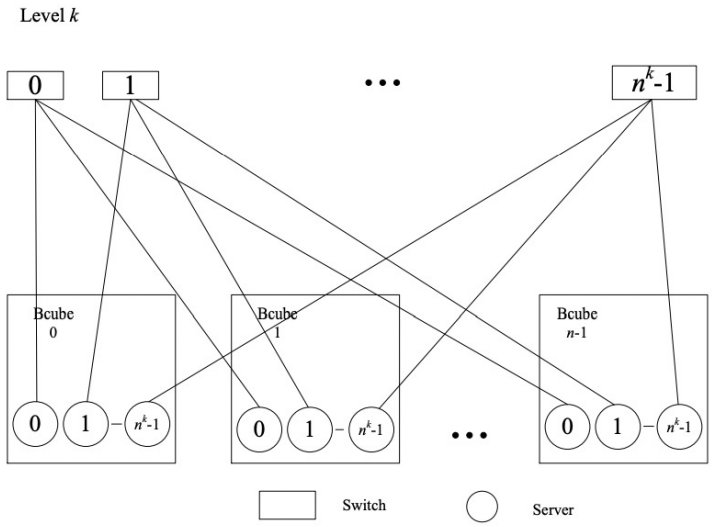
It uses BCube as a building block, consisting of n servers connected to an n-port switch.

A $BCube_k$ $(k > 0)$ is constructed with n $BCube_{k-1}s$ and n^k n-port switches. In a $BCube_k$ there are $n^{(k+1)}$ $k+1$-port servers and $k+1$ layers of switches.

Disadvantages: **limited scalability** and **high cabling costs** (NICs reason).



(a) Request phase.



(b) Grant phase.

Figure 22: BCube hybrid architecture.

MDCube is designed to **reduce the number of cables used to connect containers**.

- Each container has an ID which is mapped to a multidimensional tuple.
- Each container is connected to a neighbouring container with a different tuple in one dimension.
- There are two types of connections: Intra-container links and high-speed inter-container links.

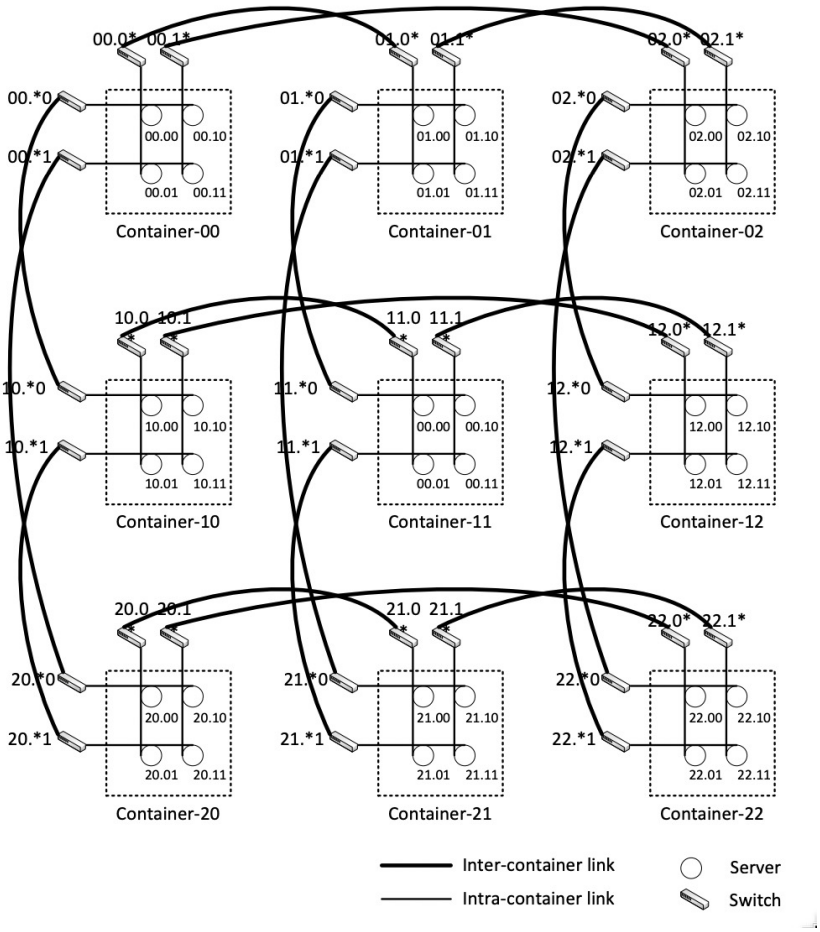


Figure 23: MDCube hybrid architecture.

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

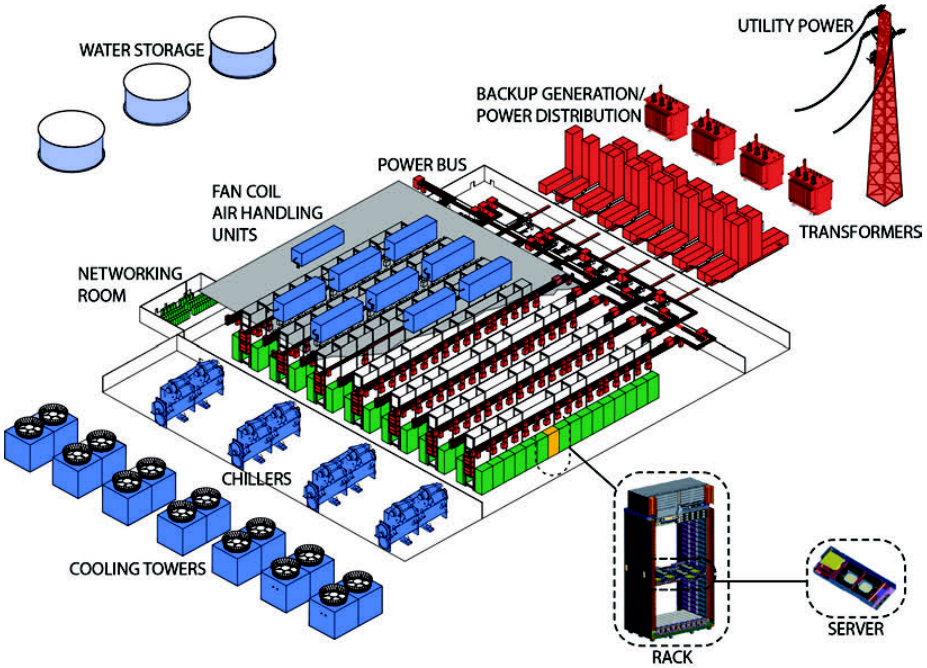


Figure 24: The main components of a typical data center. [1]

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.

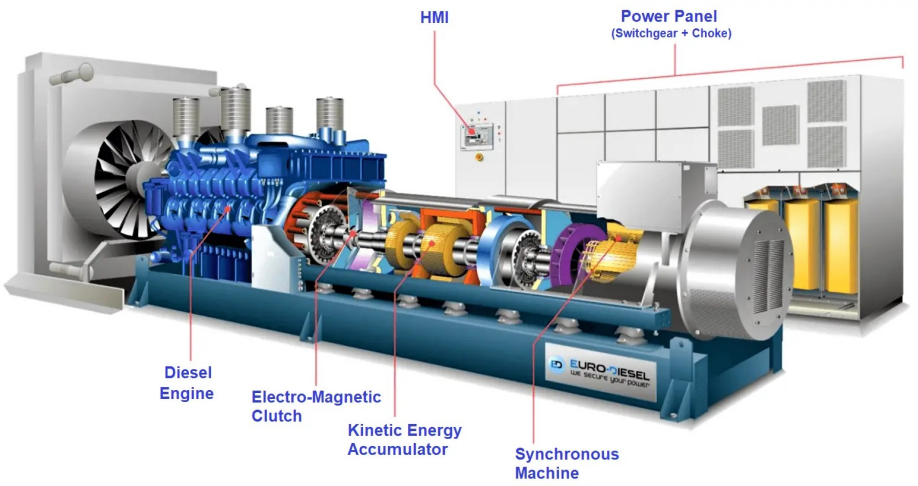


Figure 25: A Rotary UPS system.

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.

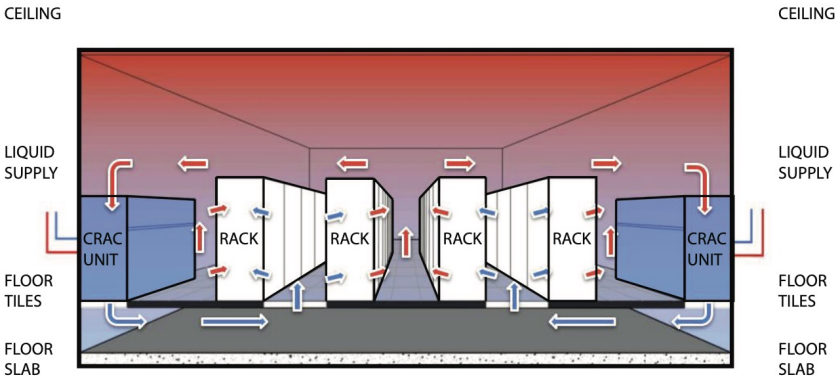
- ***Two loops***. The airflow through the underfloor plenum, the racks, and back to the **CRAC (computer room air conditioning)** defines the primary air circuit as the first loop. The second loop (the liquid supply inside the CRAC units) leads directly from the CRAC to the external heat exchangers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



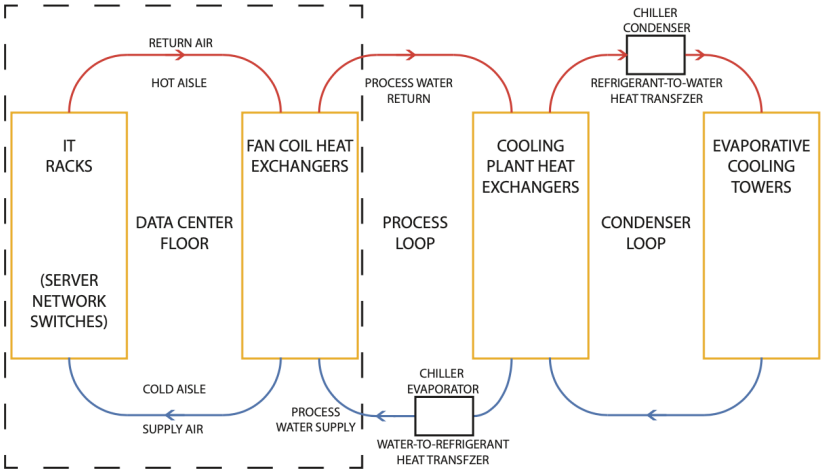
- ***Three loops***. It is used in large-scale data centres. The architecture can be viewed in the following figure.

  ✔ 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**.

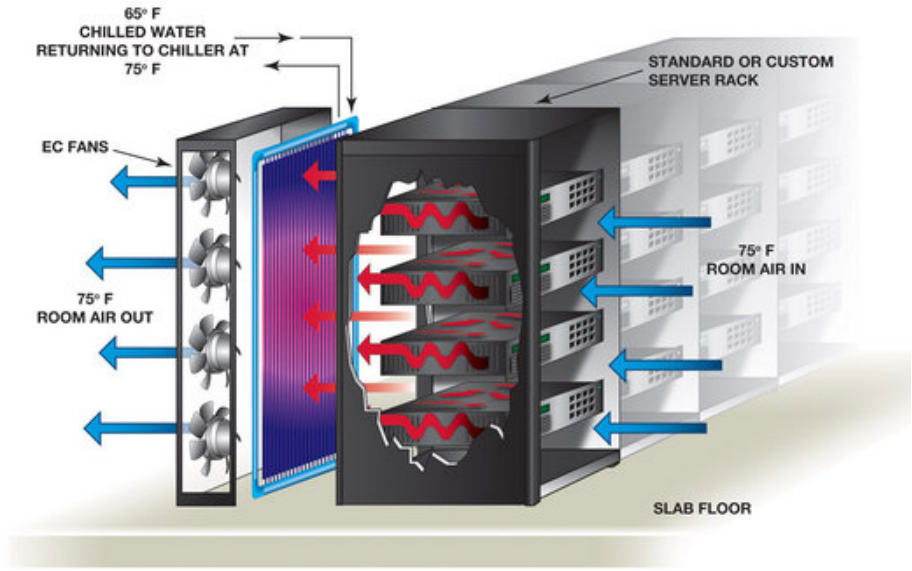


Figure 26: In-Row Cooling Mechanism (source: Energy Start).

- **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).

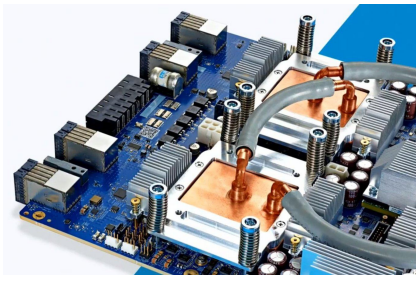


Figure 27: Liquid Cooling Mechanism.

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**:

$$\texttt{PUE} = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}} \tag{7}$$

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

$$\texttt{TFP} = \text{covers IT systems} \; + \; \text{other equipment} \tag{8}$$

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:

$$\texttt{DCiE} = \texttt{PUE}^{-1} = \frac{\text{IT Equipment Power}}{\text{Total Facility Power}} \tag{9}$$

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 8: 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 94) and System VM (page 95).

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 97). 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. [7]

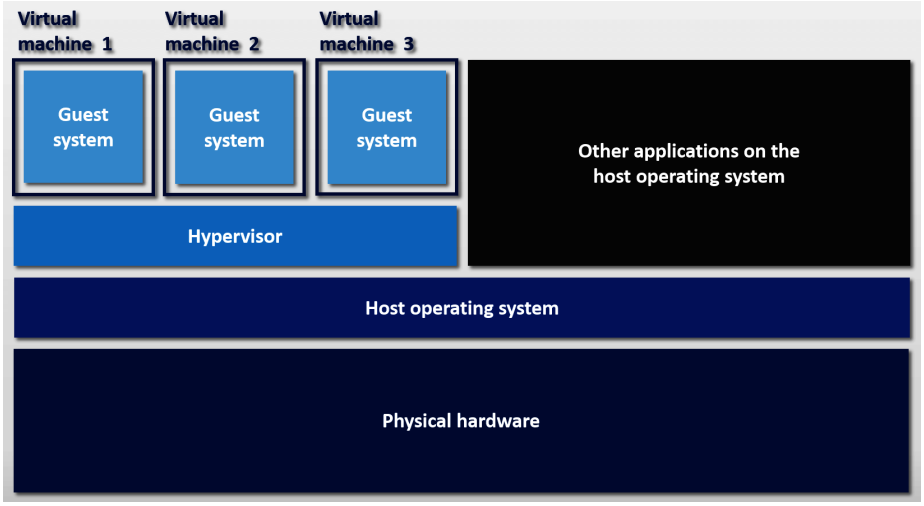


Figure 28: Operating system view if there are VMs in the physical machine (source: ionos).

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

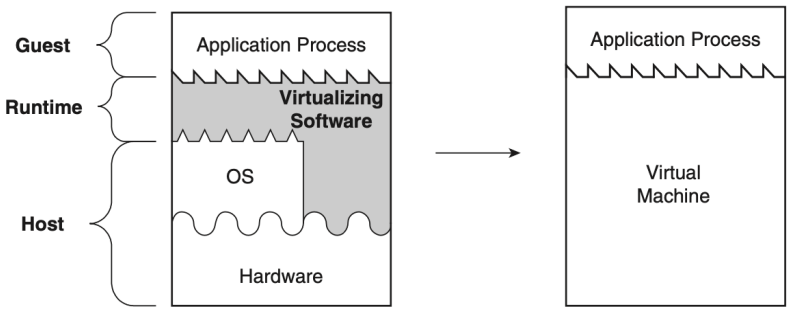


Figure 29: Process VM.

[10] 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)**.

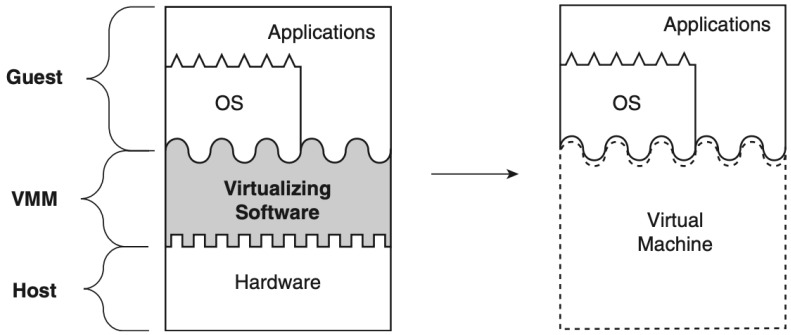


Figure 30: System VM.

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**.

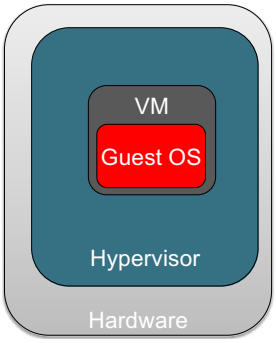


Figure 31: View of the Bare Metal Hypervisor.

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**.

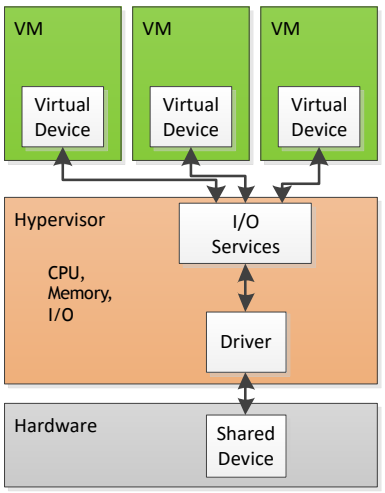


Figure 32: Monolithic architecture.

– **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.

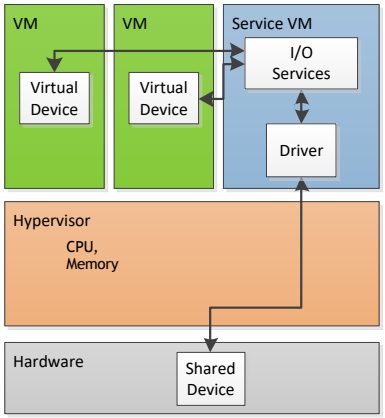


Figure 33: Microkernel architecture.

- **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**.

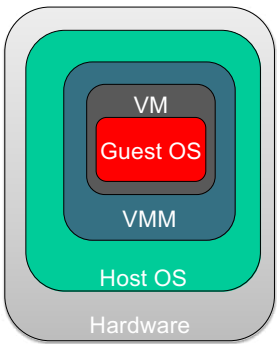


Figure 34: View of the Hosted Hypervisor.

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**.

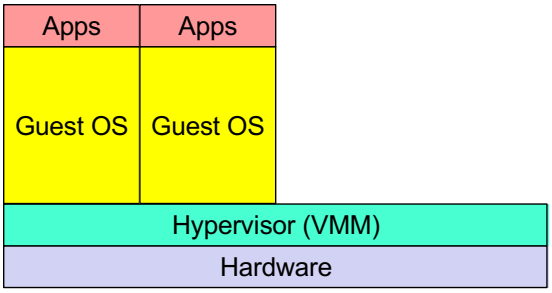


Figure 35: View of the Full Virtualization.

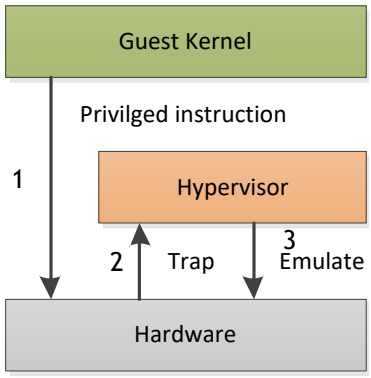


Figure 36: Full Virtualization flow.

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).

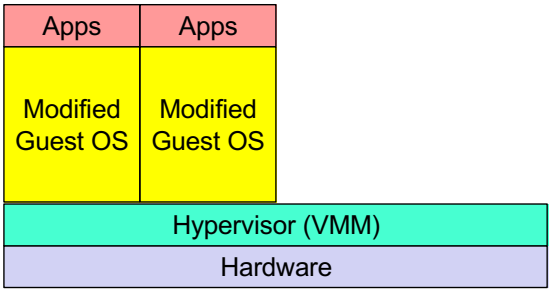


Figure 37: View of the Paravirtualization.

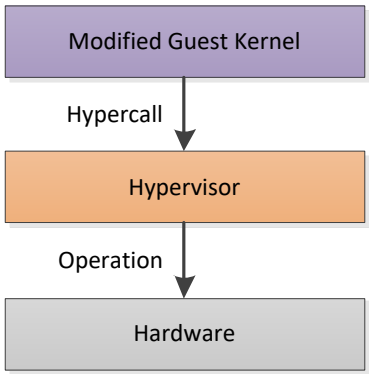


Figure 38: Paravirtualization flow.

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**.

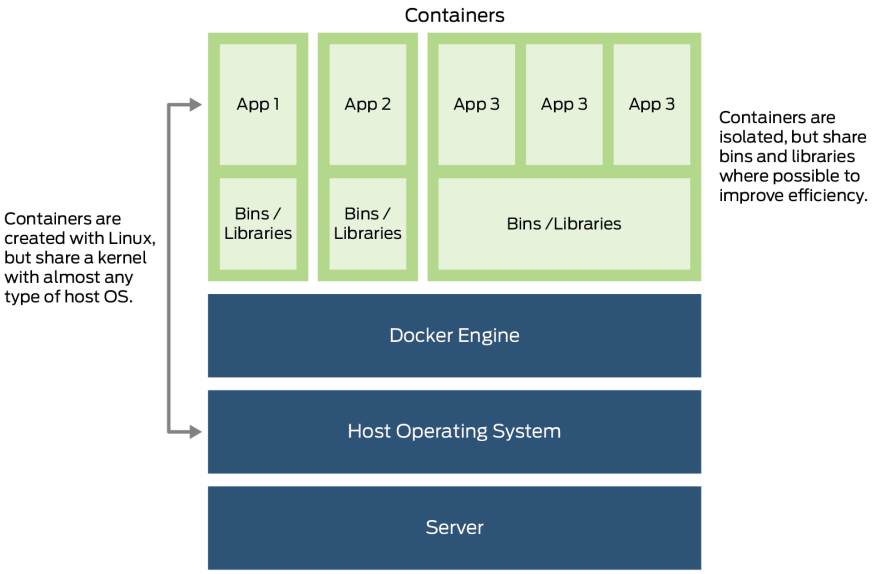


Figure 39: Containers architecture.

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 92).

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

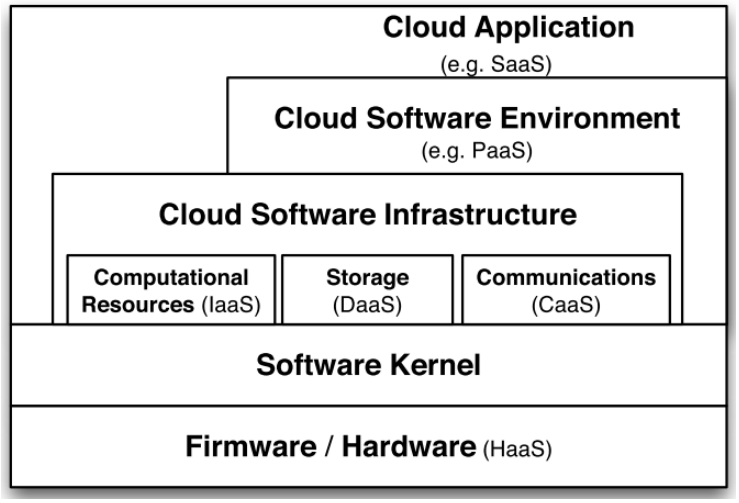
Cloud Computing is a **model for** providing convenient, **on-demand network access to a shared pool of configurable computing resources**, such as networks, servers, storage, applications and services. Anything that can be **quickly provisioned and released with minimal management** or interaction with 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. [2]

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 measures how much we trust a system. More technically, it is the **ability of a system to perform its functionality while exposing**:

- **Reliability**. Continuity of correct service.
- **Availability**. Readiness for correct service.
- **Maintainability**. Ability for easy maintenance.
- **Safety**. Absence of catastrophic consequences.
- **Security**. Confidentiality and integrity of data.

Ok, but why should we be interested in dependability?

During the implementation of a product, there is much effort to make sure that the implementation:

- matches specifications,
- fulfils requirements,
- meets constraints,
- optimizes selected parameters (such as performance, energy, etc.).

Nevertheless, even if all the above aspects are satisfied, the systems fail because something broke! The causes can be multiple: defects, process variation, degraded transistors, radiation, noise, design errors, software bugs, OS bugs, malicious attacks, and human errors.

Then, **dependability is essential** to check how much we can trust a system despite the effects of failure. If we are not convinced, consider that **a failure may have high costs if it impacts economic losses or physical damage**. Not only that, a single system failure may affect a large number of people and may cause information loss with a high consequent recovery cost. For the previous reasons, the systems that are not dependable are likely *not to be used or adopted*.

It seems very important, so when should we think about dependability?

Consistently, both at *design-time* to:

- Analyze the system under design;
- Measure dependability properties;

- Modify the design if required;

And *runtime* to:

- Detect malfunctions;
- Understand causes;
- React.

Furthermore, the failures in development should be avoided, and the design should take failures into account and guarantee that control and safety are achieved when failures occur. The effects of such failures should be predictable and deterministic, not catastrophic!

Always think about dependability, but where should it be applied?

In the past, dependability was relevant only for *safety-critical* and *mission-critical* application environments: space, nuclear, and avionics. Note that:

- **Mission-critical systems** are architectures where a **failure** during operation **can have severe or irreversible effects on the mission the system is carrying out** (for example, satellites, surveillance drones, unmanned vehicles, etc.).
- **Safety-critical systems** are architectures where a **failure** during operation can **directly threaten human life** (for example, aircraft control systems, medical instrumentation, railway signaling, nuclear reactor control systems).

However, in the computing infrastructures, the **downtime is the enemy of every data center!** So it is important to consider the dependability in each scenario in order to guarantee that everything works properly.

Finally, how to provide dependability?

It depends on the paradigm adopted:

- The **Avoidance** paradigm is a **conservative design**; it implements a design validation, has some detailed hardware and software tests, and is an error avoidance-driven approach.
- The **Tolerance** paradigm is an **error detection during system operation**; it implements online monitoring; if there is an error, it gives diagnostic solutions and has a self-recovery and self-repair.

To apply these paradigms, it is necessary to work at the:

- **Technological level** to design and manufacture by employing reliable/robust components.
- **Architectural level** to integrate standard components using solutions that allow to manage the occurrence of failures.

- **Software/Application level** to develop solutions in the algorithms or in the operating systems that mask and recover from the occurrence of failures. This guarantees high dependability, high cost, and reduced performance.

Finally, all of these solutions have a common **cost and reduced performance**.

4.1.2 Reliability and Availability

Dependability contains the properties of reliability and availability (see page 110).

Definition 1: Reliability

The **ability** of a system or component **to perform its required functions under stated conditions for a specified period of time**.

We can also calculate the **probability** that the **system will operate correctly** in a specified operating environment until time t:

$$R(t) = P(\text{not failed during } [0,t]) \tag{10}$$

(assuming it was operating at time $t = 0$). Note that the time t is essential because it is often **used to characterize systems in which even small periods of incorrect behaviour are unacceptable** (e.g. impossibility to repair). For example, if a system needs to work for slots of ten hours at a time, then ten hours is the reliability target.

As a consequence, the **unreliability** $Q(t)$ can be calculated as follows:

$$Q(t) = 1 - R(t) \tag{11}$$

The reliability probability is a **non-increasing function** ranging from 1 to 0 over $[0, +\infty)$.

$$\begin{aligned} R(0) &= 1 \\ \lim_{t \to +\infty} R(t) &= 0 \\ f(x) &= -\frac{\mathrm{d}R(t)}{\mathrm{d}t} \end{aligned} \tag{12}$$

We can observe that the probability of the reliability at the time zero is equal to one because we assume it was operating at time zero. Furthermore, the reliability probability function goes to zero when the time goes to infinity.

Definition 2: Availability

The degree to which a system or component is **operational and accessible when required for use**. It can be calculated as follows:

$$\text{Availability} = \frac{\text{Uptime}}{(\text{Uptime} + \text{Downtime})} \tag{13}$$

The **main difference** between reliability and availability is that **reliability does not break down**, and **availability works when needed**, even if it breaks down.

Finally, we calculate the **probability that the system will be operational at time** t as follows:

$$A(t) = P(\text{not failed at time } t) \tag{14}$$

It is ready for service and admits the possibility of brief outages. Finally, of course, the **unavailability** is:

$$\text{unavailability } = 1 - A(t) \tag{15}$$

What is the relationship between reliability and availability?

The **relationship with the reliability** is that:

- When the **system is not repairable**, the availability and reliability are the same:
$$A(t) = R(t) \tag{16}$$
- In general, the **reparable systems** have
$$A(t) \geq R(t) \tag{17}$$

However, the relationship is more robust because if a **system is unavailable**, **it does not deliver the specified system services**. However, it is possible to have **systems with low reliability that must be available**. Then, the system failures can be repaired quickly and do not damage data, so the low reliability may not be a problem. The opposite is generally more complex.

Metrics

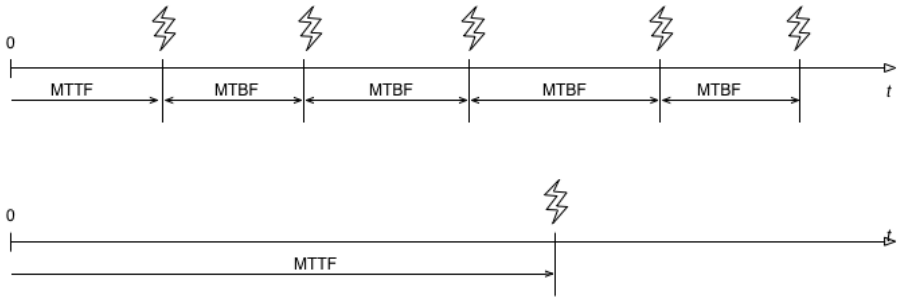
Some metrics exist for reliability and availability.

- The **Mean Time To Failure (MTTF)** is the mean time before any failure will occur. Moreover, it is calculated as the **integral of the reliability probability** (eq. 10, page 113):

$$\texttt{MTTF} = \int_0^\infty R\,(t)\ \mathrm{d}t \tag{18}$$

- The **Mean Time Between Failures (MTBF)** is the **mean time between two failures**. It is the relationship between the total operating time and the number of failures.

$$\texttt{MTBF} = \frac{\text{total operating time}}{\text{number of failures}} \tag{19}$$



- The **Failures In Time (FIT)** is **another way of reporting** `MTBF`. It is the number of **expected failures per one billion hours** (10^9) of operation for a device. Then, the `MTBF` in hours is:

$$\texttt{MTBF} = \frac{10^9}{\texttt{FIT}} \tag{20}$$

- The **Failure Rate** λ is the relationship between the number of failures and the total operating time:

$$\text{Failure Rate } \lambda = \frac{\text{number of failures}}{\text{total operating time}} \tag{21}$$

  If we observe closely, it equals $MTBF^{-1}$, then:

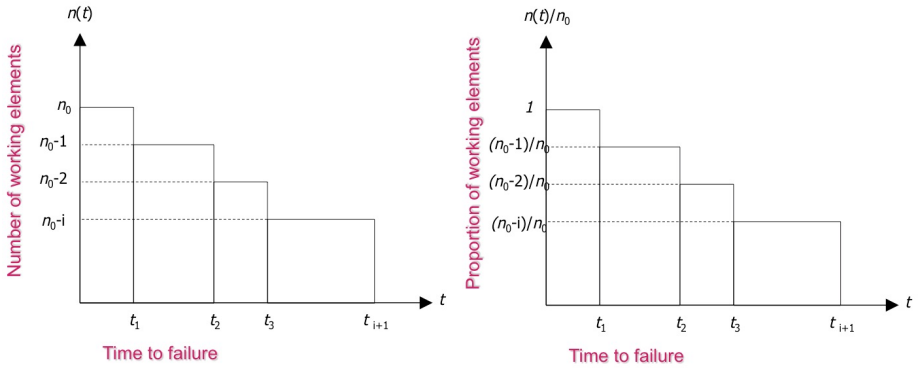
$$\texttt{MTBF} = \frac{1}{\lambda} \tag{22}$$

How to compute reliability? The Empirical Evaluation

In general, Empirical Evaluation is an evaluation method in which results are derived from observation or experiment rather than theory.

Regarding reliability, let us consider:

- n_0 independent and statistically identical elements deployed at time $t = 0$ in identical conditions $n\,(0) = n_0$;
- At time t, the $n\,(t)$ elements do not fail.
- Furthermore, t_1, t_2, ..., t_{n_0} are the times of failure of the n_0 elements. Note that the times to failure are independent occurrences of the random quantity T.



The function:

$$\frac{n\,(t)}{n_0} \tag{23}$$

Is the **empirical function of reliability** that, as $n_0 \to \infty$, **converges to the value**:

$$\frac{n\,(t)}{n_0} \to R\,(t) \tag{24}$$

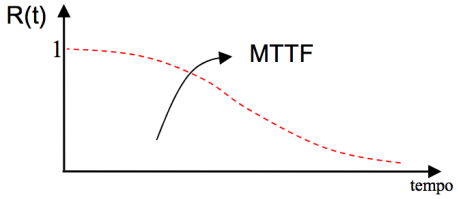
Ok, but what do we do with the reliability probability?

Well, the exploitation of the reliability probability information is **used to compute**, for a complex system, its reliability in time and the **expected lifetime**. Note that the computation of the overall reliability starts from the component one.

Reliability terminology

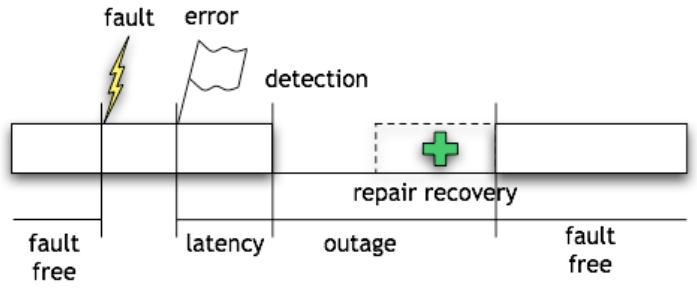
The **Constant Failure rate of the reliability** is:

$$\begin{aligned} R(t) &= e^{-\lambda t} \\ \texttt{MTTF} &= \int_0^\infty R(t)\ \mathrm{d}t = \frac{1}{\lambda} \end{aligned} \tag{25}$$



Then, to refer to it, we use the correct terminology.

- The **Fault** is a **defect within the system**.
- The **Error** is a **deviation from the required operation of the system** or subsystem.
- **Failure** is when the **system fails to perform its required function**.



Example 1

A flying drone with an automatic radar-guided landing system. An example of:

- Fault: the electromagnetic disturbances interfere with a radar measurement.
- Error: the radar-guided landing system calculates a wrong trajectory.
- Failure: the drone crashes to the ground.

Example 2: not always the fault-error-failure chain closes

A tele-surgery system. An example of:

- Fault: the radioactive ions change some memory cells' value (bit-flip).
- Error: some frames of the video stream are corrupted.
- Failure: the surgeon kills the patient.

However, not always the fault-error-failure chain closes:

- Fault: the radioactive ions make some memory cells change value (bitflip), but the corrupted memory does not involve the video stream.
- Error: no frames are corrupted.
- Failure: the surgeon carries out the procedure.

As we can see, there is no activated fault! With the same logic, a flying drone with automatic radar-guided landing:

- Fault: electromagnetic disturbances interfere with a radar measurement.
- Error: the radar-guided landing system calculates a wrong trajectory, but then, based on subsequent correct radar measurements, it can recover the right trajectory.
- Failure: the drone safely lands.

Here, there is no propagated (or absorbed error).

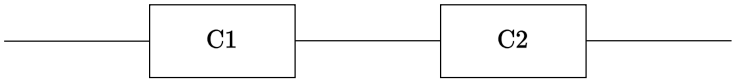
4.1.3 Reliability Block Diagrams

The **Reliability Block Diagram (RBD)**[11] is an **inductive model in which a system is divided into blocks representing distinct elements**, such as components or subsystems. **Each element in the RBD has its reliability** (previously calculated or modelled). All blocks are then combined to model all the possible success paths.

The diagram follows strict rules. To represent:

- **Components in series**: the **system failure** is **determined by the failure of the *first* component**.

$$R_s(t) = \prod_{i=1}^{n} R_i(t) \tag{26}$$



For example, in the previous illustration, reliability is calculated as:

$$R_s(t) = R_{C1}(t) \times R_{C2}(t)$$

In general, if the system S consists of **components with a reliability with an exponential distribution** (the only case considered in this course), the reliability can be calculated as:

$$R_s(t) = e^{-\lambda_s t} \tag{27}$$

Where t is the time and λ_s is the **Failure in time**:

$$\lambda_s = \sum_{i=1}^{n} \lambda_i \tag{28}$$

Note that the λ_i value is explained on page 115 (eq. 21). The Mean Time To Failure of a system is S:

$$\texttt{MTTF}_s = \frac{1}{\lambda_s} = \frac{1}{\sum_{i=1}^{n} \lambda_i} = \frac{1}{\sum_{i=1}^{n} \frac{1}{\texttt{MTTF}_i}} \tag{29}$$

If **all components are identical**:

$$R_s(t) = e^{-n\lambda t} = \exp\left(-\frac{nt}{\texttt{MTTF}_1}\right) \tag{30}$$

$$\texttt{MTTF}_s = \frac{\texttt{MTTF}_1}{n} \tag{31}$$

[11] The RBD argument was already treated in the Software Engineering for HPC notes.

Finally, the **availability** is:

$$A_s = \prod_{i=1}^{n} \frac{\texttt{MTTF}_i}{\texttt{MTTF}_i + \texttt{MTTR}_i} \tag{32}$$

Where MTTR is the **Mean Time To Repair (MTTR)**.

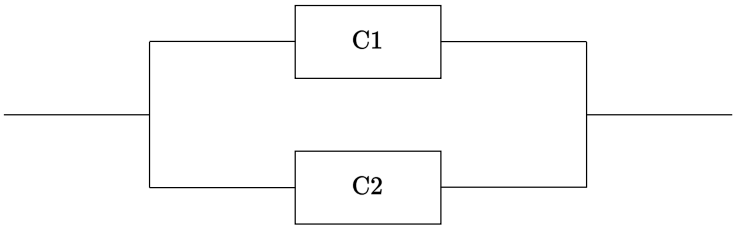
If **all components are identical**:

$$A_s(t) = A_1(t)^n \tag{33}$$

$$A = \left(\frac{\texttt{MTTF}_1}{\texttt{MTTF}_1 + \texttt{MTTR}_1}\right)^n \tag{34}$$

- **Components in parallel**: the **system fails when the *last* component fails**.

$$R_s(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{35}$$



For example, in the previous illustration, reliability is calculated as:

$$R_s(t) = 1 - [(1 - R_{C1}(t)) \times (1 - R_{C2}(t))]$$

Consider a system P composed of n components, the **reliability** is:

$$R_p(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{36}$$

And the **availability** is:

$$\begin{aligned} A_p(t) &= 1 - \prod_{i=1}^{n} (1 - A_i(t)) \\ &= 1 - \prod_{i=1}^{n} \frac{\texttt{MTTR}_i}{\texttt{MTTF}_i + \texttt{MTTR}_i} \end{aligned} \tag{37}$$

The difference between these two representations is that if a **component in the series is unhealthy**, **the whole system is unhealthy**. Instead, in the **parallel architecture**, the **system can work properly if a component is unhealthy**.

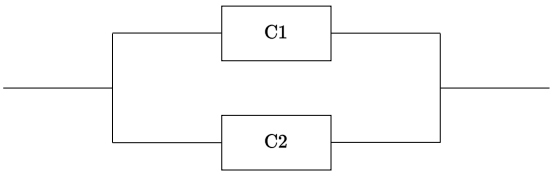
A quick recap

- **Series**.

Reliability:

$$R_s = \prod_i^n R_i \Longrightarrow R_s = R_{C1} \cdot R_{C2}$$

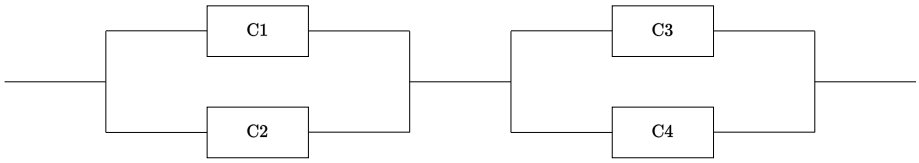
- **Parallel**.



Reliability:

$$R_s = 1 - \prod_i^n (1 - R_i) \Longrightarrow R_s = 1 - [(1 - R_{C1}) \cdot (1 - R_{C2})]$$

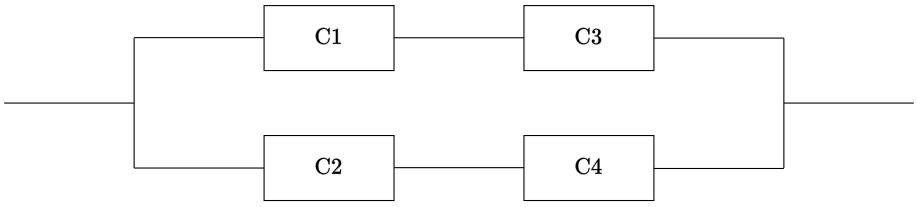
- **Series-Parallel** (component redundancy).



Reliability:

$$R_s = \{1 - [(1 - R_{C1}) \cdot (1 - R_{C2})]\} \cdot \{1 - [(1 - R_{C3}) \cdot (1 - R_{C4})]\}$$

- **Parallel-Series** (system redundancy).



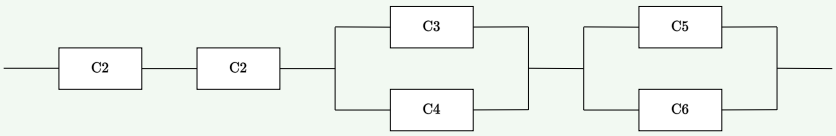
Reliability:

$$R_s = 1 - [(1 - R_{C1} \cdot R_{C3}) \cdot (1 - R_{C2} \cdot R_{C4})]$$

Example 3: 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:

$$\begin{aligned} R_I &= 1 - [(1 - R_{C5}) \cdot (1 - R_{C6})] \\ &= 1 - [(1 - 0.92) \cdot (1 - 0.92)] \\ &= 1 - 0.0064 \\ &= 0.9936 \end{aligned}$$

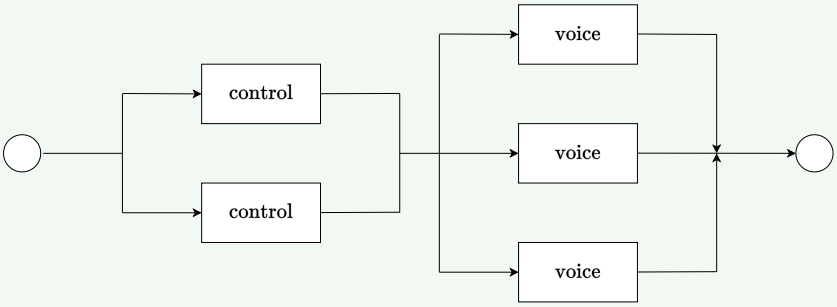
4. Finally, we calculate the reliability of the system by multiplying each calculated component reliability:

$$\begin{aligned} R_s &= R_G \cdot R_H \cdot R_I \\ &= 0.9215 \cdot 0.9999 \cdot 0.9936 \\ &= 0.91551083976 \approx 0.9155 \end{aligned}$$

Example 4: 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. [4]

System failure occurs when the $(n - r + 1)$-th component failure occurs. [4]

But note an interesting observation: [4]

- When $r = n$, the r out of n model reduces to the **series** model.
- When $r = 1$, the r out of n model becomes the **parallel** model.

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:

$$R_s(t) = RV \sum_{i=r}^{n} R_c^i (1 - R_c)^{n-i} \frac{n!}{i!(n-i)!} \tag{38}$$

The last part of the formula can be replaced by the binomial coefficient:

$$\frac{n!}{i!(n-i)!} = \binom{n}{i}$$

The components are:

- R_s: System Reliability
- R_c: Component Reliability
- R_v: Voter Reliability
- n: number of components
- r: minimum number of components which must survive

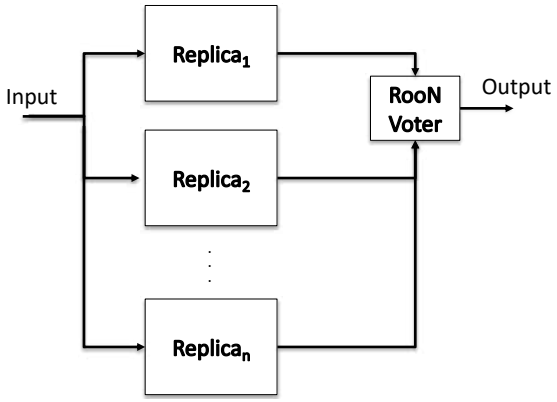


Figure 40: General structure of RooN system.

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**.

The system works properly if 2 out of 3 components work properly **and** the voter works properly.

The **TMR Reliability** R_{TMR} is:

$$R_{TMR} = R_v \left(3 \cdot R_m^2 - 2 \cdot R_m^3\right) \tag{39}$$

And the **TMR MTTF** $MTTF_{TMR}$ is:

$$\texttt{MTTF}_{TMR} = \frac{5}{6} \cdot MTTF_{\text{simplex}} \tag{40}$$

TMR: good or bad?

TMR systems can **tolerate** both **transient**[12] **and permanent faults**[13]. It also has **higher reliability** (for shorter missions).

The **TMR reliability** can be the **same as the series systems** if:

$$R_{TMR}(t) = R_c(t) \Longrightarrow 3e^{-2\lambda_m t} - 2e^{-3\lambda_m t} = e^{-\lambda_m t} \tag{41}$$

The time t is:

$$t = \frac{\ln(2)}{\lambda_m} \approx 0.7\,\texttt{MMTF}_c \tag{42}$$

Note that $R_{TMR}(t) > R_c(t)$ when mission time is less than 70% of $\texttt{MTTF}_c$.

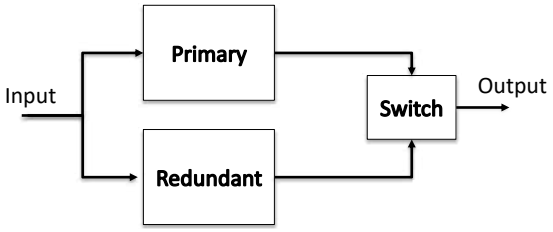
[12] In electrical engineering, a **transient fault** is defined as an error condition that vanishes after the power is disconnected and restored.

[13] In electrical engineering, a persistent or **permanent faults** are a type of fault that is present regardless of the disconnection of the power supply.

4.1.3.3 Standby redundancy

Standby redundancy is a system consisting of two parallel replicas:

- The *primary* replica, which **operates all the time**.
- The *redundant* replica (generally disabled) is **activated when the primary replica fails**.



To be operational, the standby system requires two mechanisms:

1. A mechanism to **determine whether or not the primary replica is functioning properly** (on-line self check);
2. A dynamic switching mechanism to **deactivate the primary replica and activate the redundant replica**.

Standby Parallel Model	System Reliability
Equal failure rates, perfect switching	$R_s = e^{-\lambda t}\,(1+\lambda t)$
Unequal failure rates, perfect switching	$R_s = e^{-\lambda_1 t} + \lambda_1 \dfrac{\left(e^{-\lambda_1 t} - e^{-\lambda_2 t}\right)}{\lambda_2 - \lambda_1}$
Equal failure rates, imperfect switching	$R_s = e^{-\lambda t}\,(1 + R_{\text{switch}}\lambda t)$
Unequal failure rates, imperfect switching	$R_s = e^{-\lambda_1 t} + R_{\text{switch}}\lambda_1 \dfrac{\left(e^{-\lambda_1 t} - e^{-\lambda_2 t}\right)}{\lambda_2 - \lambda_1}$

Table 9: 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:

$$R = \frac{1}{\texttt{DiskRPM}} \tag{43}$$

And in seconds:

$$R_{\text{sec}} = 60 \times R \tag{44}$$

From the R_{sec} we can also calculate the **total rotation average**:

$$T_{\text{rotation AVG}} = \frac{R_{\text{sec}}}{2} \tag{45}$$

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**):

$$T_{\text{seek AVG}} = \frac{T_{\text{seek MAX}}}{3} \tag{46}$$

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**:

$$T_{\text{transfer}} = \frac{\text{R/W of a sector}}{\text{Data transfer rate}} \tag{47}$$

The **total time to complete a disk I/O operation** is called the $T_{\text{I/O}}$ **Service Time**:

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}} \tag{48}$$

If the disk is shared among processes, we must also consider **queueing time**. And the **Response Time** is:

$$T_{\text{response}} = T_{\text{queue}} + T_{\text{I/O}} \tag{49}$$

Where T_{queue} depends on queue length, disk utilization rate, variance in request time, arrival rate of I/O requests.

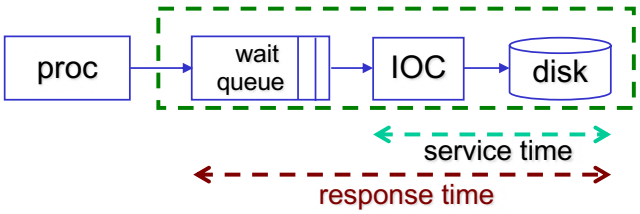


Figure 41: Service and response time.

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:

$$DL = \% \text{ of blocks that can be accessed without new seek or rotation} \quad (50)$$

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 127):

$$T_{\text{I/O AVG}} = (1 - DL) \cdot (T_{\text{seek}} + T_{\text{rotation}}) + T_{\text{transfer}} + T_{\text{controller}} \quad (51)$$

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 128 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).

- Let's assume:
  - A constant Failure Rate;
  - An exponentially distributed time to failure;
  - The case of independent failures.

  (conditions usually used to determine the disk MTTF).

  The **Mean Time To Failure of a disk array** $\texttt{MTTF}_{\textbf{diskArray}}$ is equal to the relationship between the `MTTF` of a single disk and the number of disks:

$$\texttt{MTTF}_{\text{diskArray}} = \frac{\texttt{MTTF}_{\text{singleDisk}}}{\#\,\texttt{Disks}} \tag{52}$$

  Large disk arrays are **too unstable to be used without any fault tolerance approach**. Disks do not have huge `MTTF` since it is highly probable they will be replaced in a "short time". Note that the **RAID 0 has no redundancy**!

$$\texttt{MTTF}_{\texttt{RAID 0}} = \texttt{MTTF}_{\text{diskArray}} = \frac{\texttt{MTTF}_{\text{singleDisk}}}{\#\,\texttt{Disks}} \tag{53}$$

- RAID levels greater than level zero use redundancy to improve reliability. Then, when a disk fails, it should be replaced, and the information should be reconstructed on the new disk using the redundant information. The `MTTR` is the **time needed for this action**! As always, the N value is the number of disks in the array. The **Mean Time To Failure of a RAID** $\texttt{MTTF}_{\texttt{RAID}}$ (except the level zero!) is:

$$\texttt{MTTF}_{\texttt{RAID}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{additionalCriticalFailuresInMTTR}}}\right) \tag{54}$$

  Where:
  - $\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}$ is the **MTTF for the array of N disks**.
  - $\frac{1}{\text{Probability}_{\text{additionalCriticalFailuresInMTTR}}}$ is the **probability of other critical failures in the array before repairing the failed disk**. The RAID level and type of redundancy determine it.

  In detail, the **Mean Time To Failure of each RAID level** (except the zero) is:

  - `RAID 1` - With a single copy of each disk, one drive can fail, and if we are lucky, $N \div 2$ drives can fail without data loss. Then the **MTTF of** $\texttt{RAID 1}$ $\texttt{MTTF}_{\textbf{RAID 1}}$ is:

$$\texttt{MTTF}_{\text{RAID 1}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \tag{55}$$

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{1}{\text{MTTF}_{\text{singleDisk}}}\right) \times \text{MTTR} \quad (56)$$

Where:

* $\frac{1}{\text{MTTF}_{\text{singleDisk}}}$ is the **failure rate for the copy of the failed disk**.
* MTTR is the **period of interest before replacement**.

– RAID 0 + 1 - When one disk in a stripe group fails, the entire group goes off. Then the **MTTF of RAID 01 $\text{MTTF}_{\text{RAID 0 + 1}}$** is:

$$\text{MTTF}_{\text{RAID 01}} = \left(\frac{\text{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \quad (57)$$

It is not the same as RAID 1 because the probability is:

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{G}{\text{MTTF}_{\text{singleDisk}}}\right) \times \text{MTTR} \quad (58)$$

Where:

* G is the **number of disks in a stripe group**.
* $\frac{G}{\text{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the disks in the other group**.
* MTTR is the **period of interest before replacement**.

– RAID 1 + 0 - To fail, the same copy in both groups has to fail, but multiple failure can be tolerated. Then the **MTTF of RAID 10 $\text{MTTF}_{\text{RAID 1 + 0}}$** is:

$$\text{MTTF}_{\text{RAID 10}} = \left(\frac{\text{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \quad (59)$$

It is not the same as RAID 1 because the probability is:

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{1}{\text{MTTF}_{\text{singleDisk}}}\right) \times \text{MTTR} \quad (60)$$

Where:

* $\frac{1}{\text{MTTF}_{\text{singleDisk}}}$ is the **failure rate for the copy of the failed disk**.
* MTTR is the **period of interest before replacement**.

– RAID 4 and RAID 5 - To fail, two disks have to fail before replacement. Then the **MTTF of RAID 4 $\text{MTTF}_{\text{RAID 4}}$** and the **MTTF of RAID 5 $\text{MTTF}_{\text{RAID 5}}$** is:

$$\text{MTTF}_{\text{RAID 4}} = \text{MTTF}_{\text{RAID 5}} = \left(\frac{\text{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndFailureInMTTR}}}\right) \quad (61)$$

And the probability is:

$$\text{Probability}_{\text{2ndFailureInMTTR}} = \left(\frac{(N-1)}{\text{MTTF}_{\text{singleDisk}}}\right) \times \text{MTTR} \quad (62)$$

Where:

* $\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the other disks**.
* `MTTR` is the **period of interest before replacement**.

– `RAID 6` - Two disks failures at the same time are tolerated. Then the **MTTF of `RAID 6` $\texttt{MTTF}_{\text{RAID 6}}$** is:

$$\texttt{MTTF}_{\text{RAID 6}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndAnd3rdFailureInMTTR}}}\right) \tag{63}$$

And the probability is:

$$\text{Probability}_{\text{2ndAnd3rdFailureInMTTR}} = \text{Probability}_{\text{2ndFailure}} \times \text{Probability}_{\text{3ndFailure}} \tag{64}$$

Where:

* $\text{Probability}_{\text{2ndFailure}}$:

$$\text{Probability}_{\text{2ndFailure}} = \left(\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \tag{65}$$

  · $\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the other disks**.
  · `MTTR` is the **period of interest before the replacement**.

* $\text{Probability}_{\text{3ndFailure}}$:

$$\text{Probability}_{\text{3ndFailure}} = \left(\frac{(N-2)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \frac{\texttt{MTTR}}{2} \tag{66}$$

  · $\frac{(N-2)}{\texttt{MTTF}_{\text{singleDisk}}}$ is the **failure rate for one of the remaining disks**.
  · $\frac{\texttt{MTTR}}{2}$ is the **average overlapping period between first and second disk replacement** (both disk not yet replaced).

RAID level	**Metric**
RAID 0	$\texttt{MTTF}_{\texttt{RAID 0}} = \frac{\texttt{MTTF}_{\text{singleDisk}}}{N}$
RAID 1 + 0	$\texttt{MTTF}_{\texttt{RAID 10}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times \texttt{MTTR})}$
RAID 0 + 1	$\texttt{MTTF}_{\texttt{RAID 01}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times G \times \texttt{MTTR})}$
RAID 4	$\texttt{MTTF}_{\texttt{RAID 4}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times N-1 \times \texttt{MTTR})}$
RAID 5	$\texttt{MTTF}_{\texttt{RAID 5}} = \frac{\left(\texttt{MTTF}_{\text{singleDisk}}\right)^2}{(N \times N-1 \times \texttt{MTTR})}$
RAID 6	$\texttt{MTTF}_{\texttt{RAID 6}} = \frac{2 \times \left(\texttt{MTTF}_{\text{singleDisk}}\right)^3}{\left(N \times N-1 \times N-2 \times \texttt{MTTR}^2\right)}$

Table 10: MTTF summary RAID levels.

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.

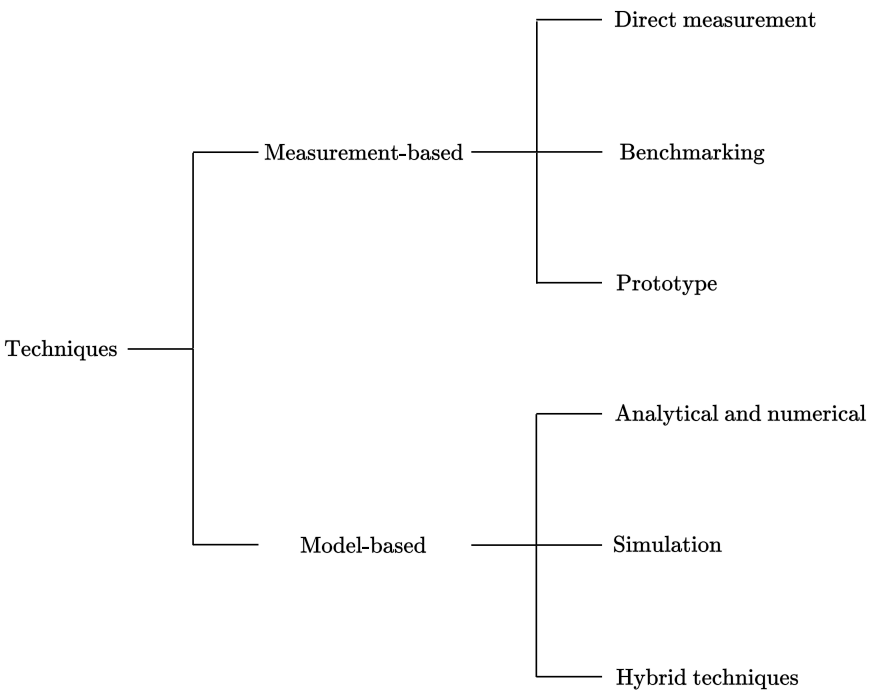


Figure 42: Quality Evaluation techniques.

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**. [3]

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.

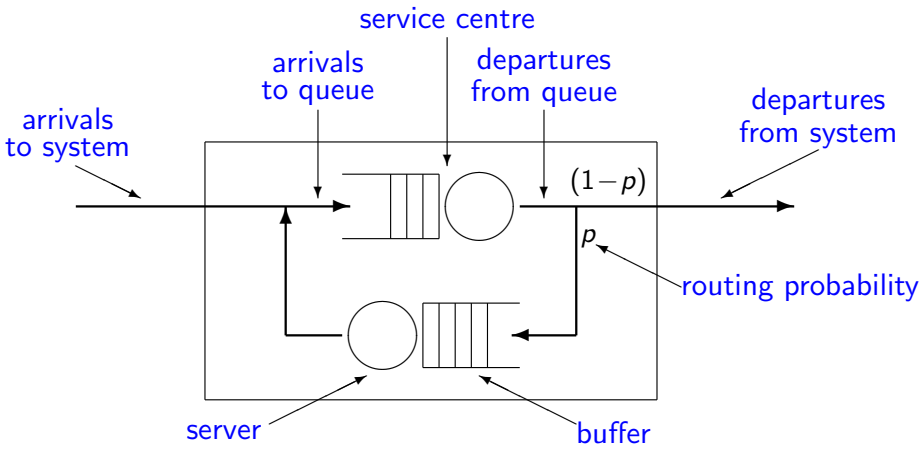


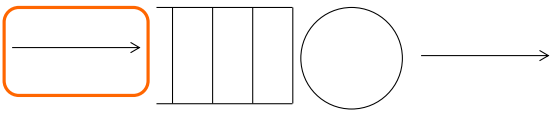
Figure 43: Queueing Network graphical representation.

4.3.2.2 Characteristics

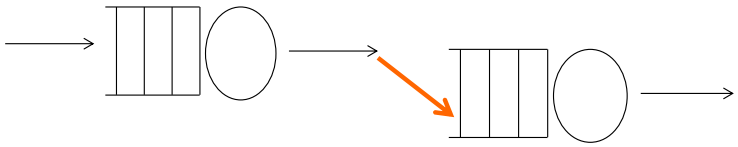
Queueing models are characterized by several aspects:

- **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.

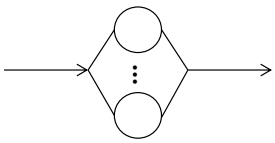
  

- **Service**. Service **represents the time a job spends being served**. The server does the job, but the number of servers can be different:

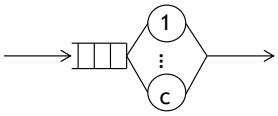
  – **Single server**. It has the ability to serve **one client at a time**. **Waiting customers remain in the buffer** until they are selected for service. Finally, the **next customer is selected depending on the service discipline**.

  – **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:

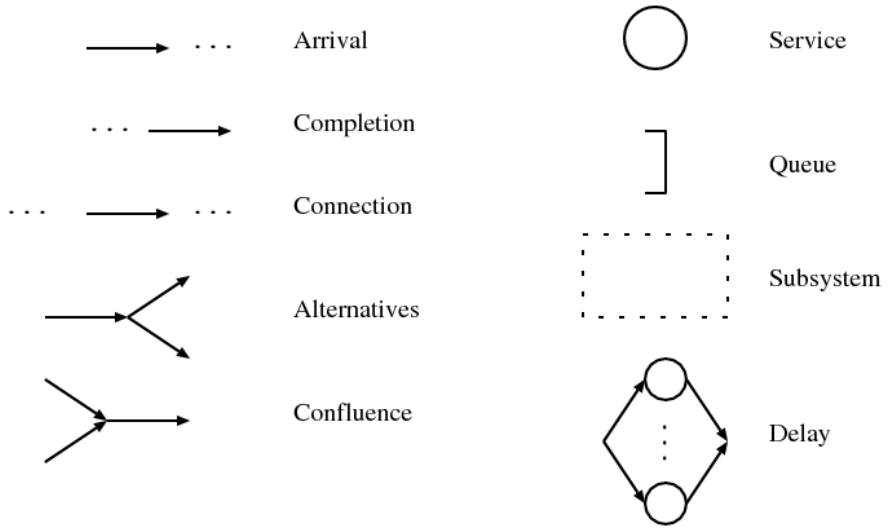
  - **Probabilistic Routing Algorithm**. **Each path is assigned a probability of being chosen by the job that left the station in question**.
  - **Round Robin Routing Algorithm**. The **destination chosen by the job rotates among all possible existing destinations**.

– **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.

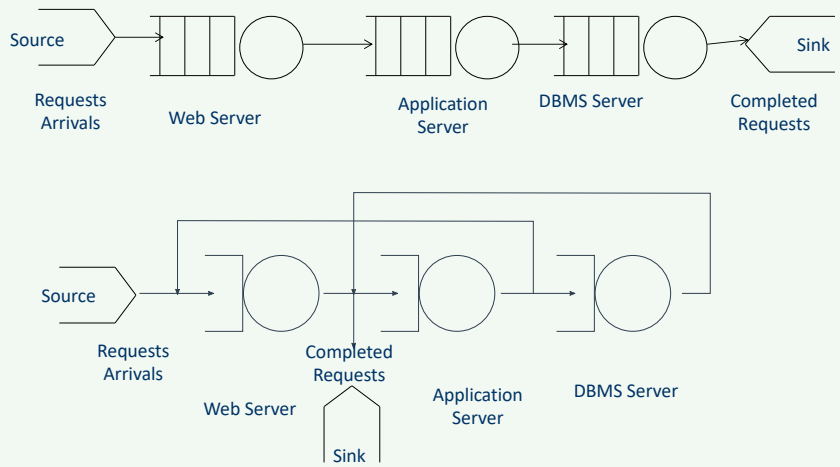
An additional graphical notation is:



Example 5: 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:

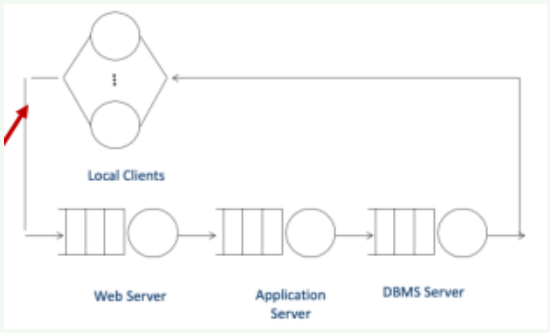
1. The only thing we know is that each server should be visited by the application.
2. In the second case we know that the application **after visiting the web server** requires some operations at **the application server** and then **can go back** to **the web server** and **leave** the system **or** can require service at the **DMBS** and then **go back** to the **application server**.



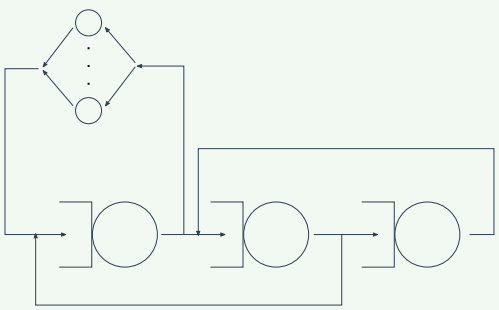
Example 6: Closed Networks

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 138).

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**:

$$\lambda = \frac{\text{number of request arrivals}}{\text{length of time we observe the system}} = \frac{A}{T} \tag{67}$$

- **Throughput** or **Completion rate**:

$$X = \frac{\text{number of request completions}}{\text{length of time we observe the system}} = \frac{C}{T} \tag{68}$$

- **Utilization**:

$$U = \frac{\text{total amount of time during which the system is busy}}{\text{length of time we observe the system}} = \frac{B}{T} \tag{69}$$

- **Mean service time** per completed job:

$$S = \frac{\text{total amount of time during which the system is busy}}{\text{number of request completions}} = \frac{B}{C} \tag{70}$$

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:

- **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**:

$$\lambda_k = \frac{A_k}{T} \tag{71}$$

- **Throughput** or **Completion rate**:

$$X_k = \frac{C_k}{T} \tag{72}$$

- **Utilization**:

$$U_k = \frac{B_k}{T} \tag{73}$$

- **Mean service time** per completed job:

$$S_k = \frac{B_k}{C_k} \tag{74}$$

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**:

$$U_k = X_k \cdot S_k \tag{75}$$

4.3.3.3 Little's Law

The **Little's Law** is:

$$N = X \cdot R \tag{76}$$

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.

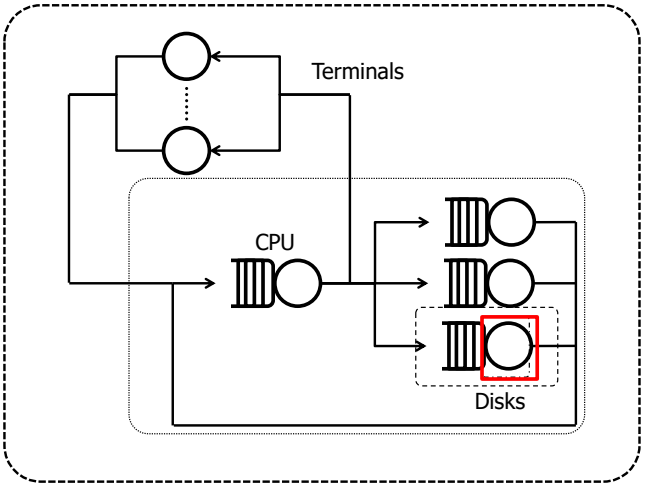
If the system throughput is X requests/sec, and each request remains in the system on average for R seconds, then for each unit of time, we can observe on average XR requests in the system.

Example 7

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:
    $$N = X \cdot R = X \cdot S = U$$

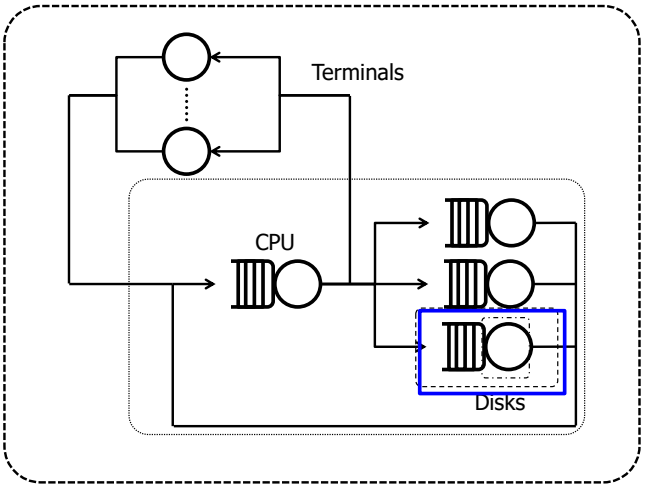


- **Service and Queue level**.
  - The service time is:

$$R = S$$

    Where R is the average of each request remaining in the system.
  - The utilization is:

$$N = \text{Flying requests} \Rightarrow \frac{N}{X} = R = (S + T_{\text{queue}})$$



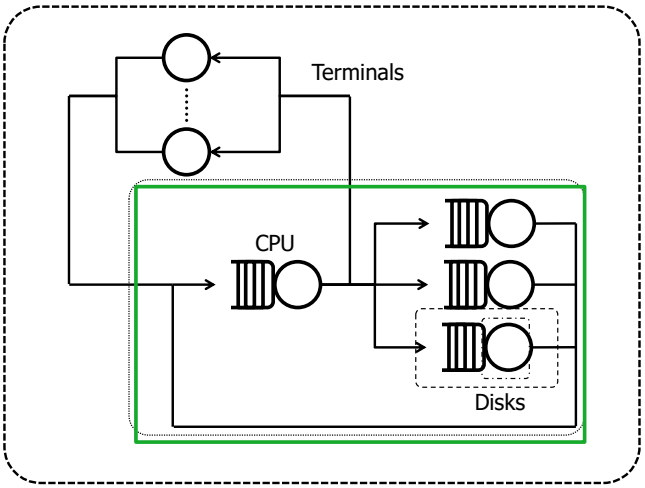
- **Subsystem level**.
  - The service time is:

$$R = \text{Residence Time}$$

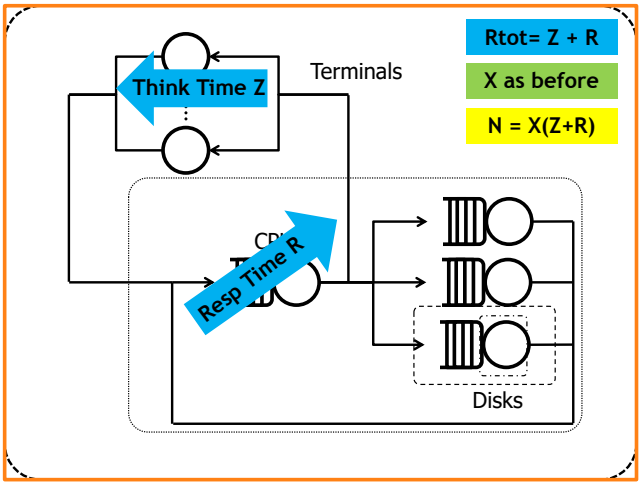
    Residence time corresponds to our conventional notion of response time: the period of time from when a user submits a request until that user's response is returned.

– 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:

$$R = \frac{N}{X} - Z \tag{77}$$

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 8

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**:

$$V_k = \frac{C_k}{C} \tag{78}$$

It is the ratio of the number of completions at the k-th resource to the number of system completions.

Example 9

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.

$$X_k = V_k \cdot X \tag{79}$$

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**:

$$D_k = S_k \cdot V_k \tag{80}$$

Using the service demand, we can rewrite the **Utilization Law**:

$$U_k = X_k \cdot S_k = (X \cdot V_k) \cdot S_k = D_k \cdot X \tag{81}$$

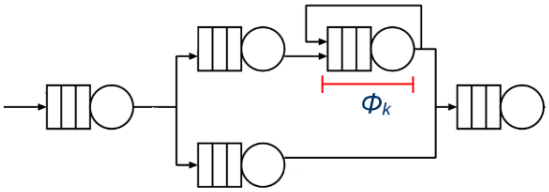
The utilization of a resource is denoted U_k and it is the percentage of time that the k-th resource is in use processing a job. It is also equal to the product of:

- 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.

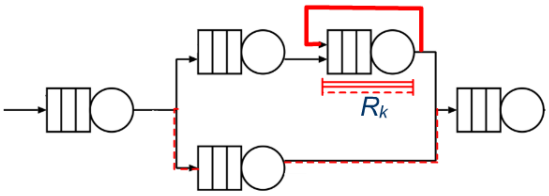
4.3.3.8 Response and Residence Times

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**:

$$\begin{aligned} D_k &= v_k \cdot S_k \\ R_k &= v_k \cdot \tilde{R}_k \end{aligned} \tag{82}$$

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**[14].
- 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:

$$D = \sum_k D_k \tag{83}$$

- $\mathbf{D}_{\max}$, 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**.

[14] **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 143) 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:

$$\lambda_{\text{sat}} = \frac{1}{D_{\max}} \tag{84}$$

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.

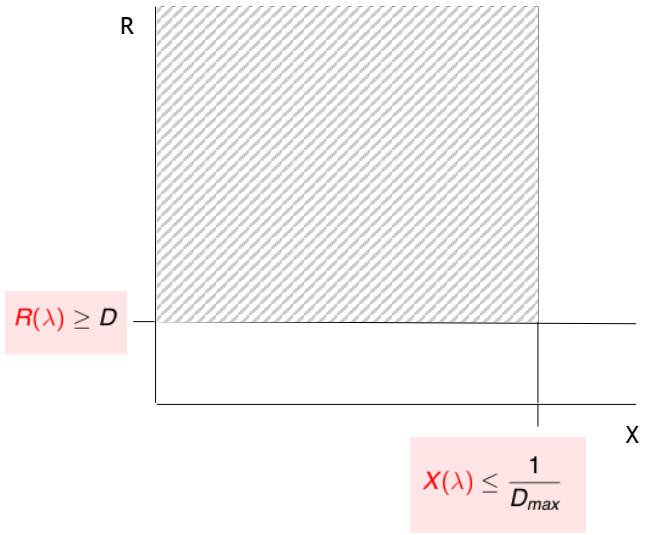
The bounding analysis for open models gives the following formulas:

- Bound for $X(\lambda)$:

$$X(\lambda) \leq \frac{1}{D_{\max}} \tag{85}$$

- Bound for $R(\lambda)$:

$$R(\lambda) \geq D_{\max} \tag{86}$$



Closed Models

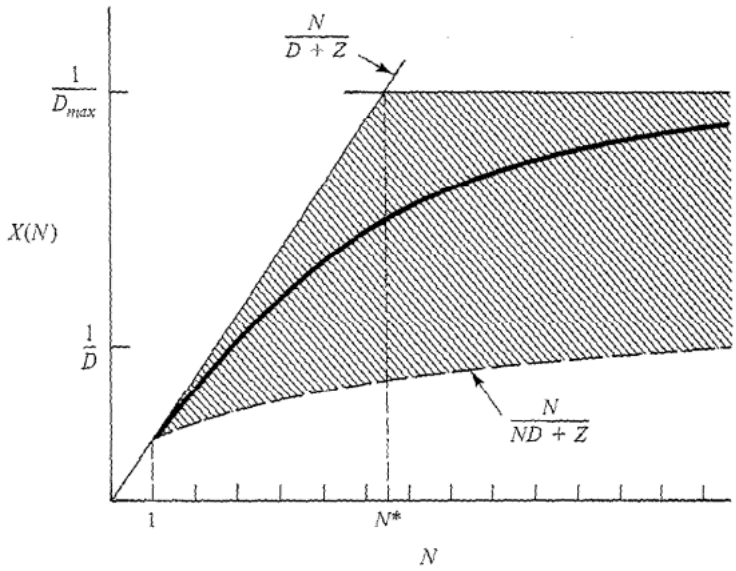
Bounding Analysis depends on the situation:

- **Light Load** situation.
  - **Lower bounds**:
    - **1 customer** case:

$$\begin{aligned} N &= X \cdot (R + Z) \\ 1 &= X \cdot (D + Z) \\ X &= \frac{1}{(D + Z)} \end{aligned} \tag{87}$$

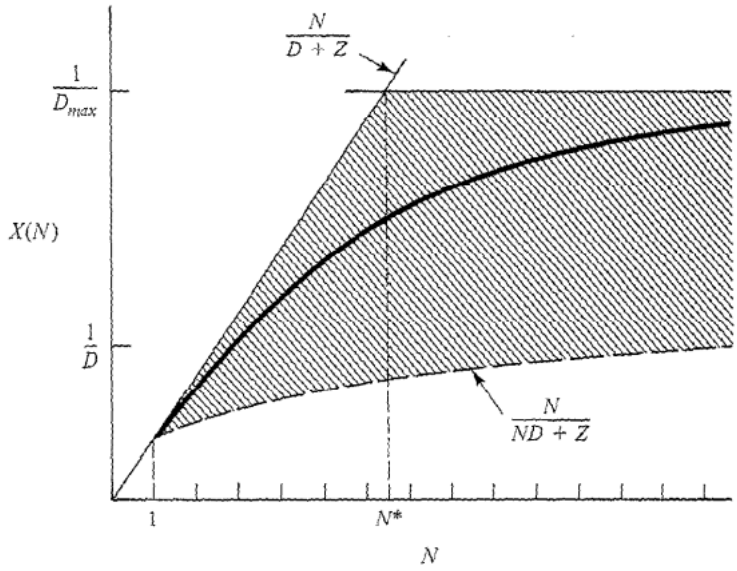
    - **Adding customers**: smallest X (*system throughput*) obtained with largest R (*system response time*), i.e., new jobs queue behind others already in the system.
      Remember: in closed models, the highest possible system response time occurs when each job, at each station, found all the other $N-1$ costumers in front of it, then $R = N \cdot D$.
      In this case $R = N \cdot D$ and X is:

$$\begin{aligned} X &= \frac{N}{(N \cdot D + Z)} \\ \lim_{N \to \infty} \frac{N}{(N \cdot D + Z)} &= \frac{1}{D} \end{aligned} \tag{88}$$



- **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:

$$X = \frac{N}{(D + Z)} \tag{89}$$



- **Heavy Load** situation.
  - **Upper bound**:

$$U_k(N) = X(N) D_k \leq 1 \tag{90}$$

Since the first to saturate is the **bottleneck** (max):

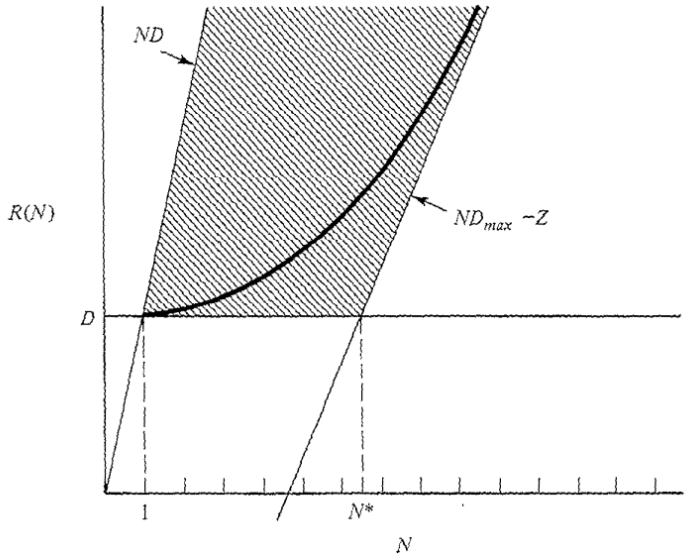
$$X(N) \leq \frac{1}{D_{\max}} \tag{91}$$

The $X(N)$ bounds is:

$$\frac{N}{N \cdot D + Z} \leq X(N) \leq \min\left(\frac{1}{D_{\max}}, \frac{N}{D + Z}\right) \tag{92}$$

The $R(N)$ bounds is:

$$\max(D, N \cdot D_{\max} - Z) \leq R(N) \leq N \cdot D \tag{93}$$

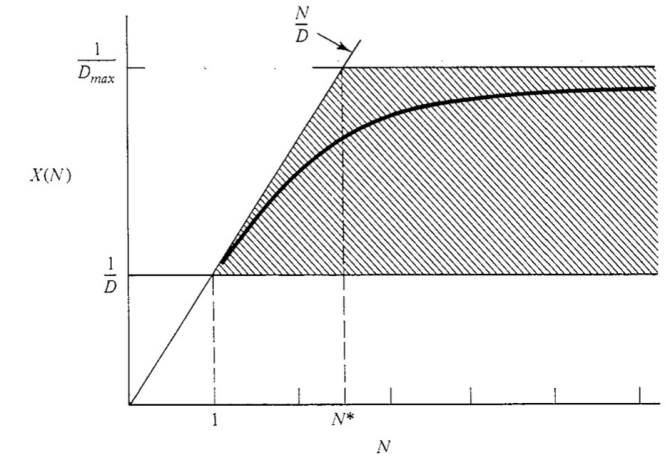


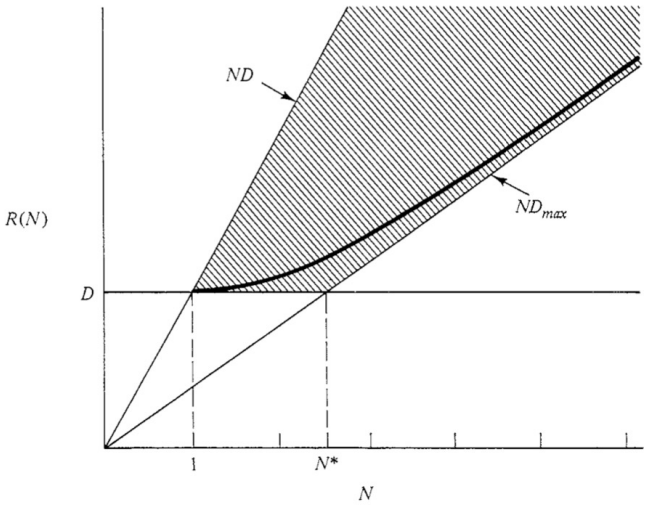
$R(N)$ bounds.

* **N*** case: particular population size determining if the light or heavy load optimistic bound is to be applied:

$$N^* = \frac{D + Z}{D_{\max}} \tag{94}$$

Without thinking time we have the following limits:





References

[1] 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.

[2] Shantanu Bhattacharya and Lipika Bhattacharya. *Xaas: Everything-As-A-Service: the lean and agile approach to business growth*. World Scientific, 2022.

[3] E.D. Lazowska. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, 1984.

[4] 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].

[5] Gianluca Palermo. Computing infrastructures. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[6] Gianluca Palermo. Lesson 1, computing infrastructures. Slides from the HPC-E master's degree course on Politecnico di Milano, 2024.

[7] 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.

[8] Ming-Chang Yang. Lesson 2, raid and data integrity. Slides from the CSCI5550 Advanced File and Storage Systems course on the Chinese University of Hong Kong, 2020.

Index

O

P

Q

R

S

T

U

V

W

X