# Computing Infrastructures - Notes - v1.0.0

260236

September 2024

# 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. [6]

About:

GitHub repository

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

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

# Contents

# 1 Introduction: definition of Data Center and Computing Infrastructure

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

**Definition 1**

**Data Centers** are buildings where multiple servers and communication gear are co-located because of their common environmental requirements and physical security needs, and for ease of maintenance. [1]

**Definition 2**

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

A number of computing infrastructures exist:

- **Cloud** offers virtualized computing, storage and network resources with highly-elastic capacity.
- **Edge Servers** are on-premises hardware resources that perform more compute-intensive data processing.

  In other words, an edge server is a piece of hardware that performs data computation at the end (or "edge") of a network. Like a regular server, an edge server can provide compute, networking, and storage functions.[1]
- **IoT** and **AI-enabled Edge Sensors** are hardware devices where the data acquisition and partial processing can be performed at the edge of the network.

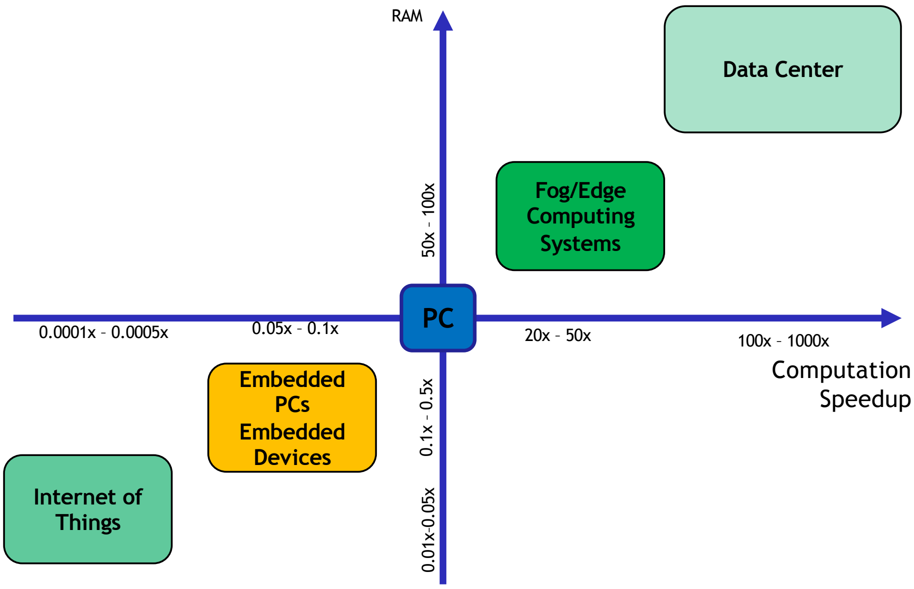
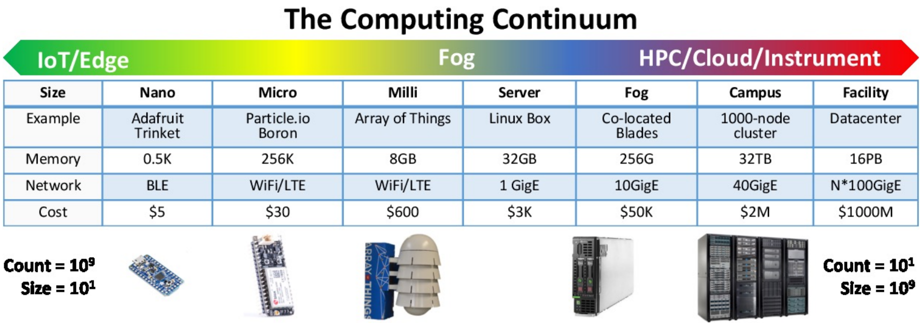



Figure 1: An **example** of Computing Infrastructures. [7]

The **Computing Continuum**, a novel paradigm that extends beyond the current silos of cloud and edge computing, can enable the seamless and dynamic deployment of applications across diverse infrastructures. [4]

[1] More info here.




| 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: The Computing Continuum. [7]

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

---

## Data Centers

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

### ✔ Data Centers Advantages

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

### Data Centers Disadvantages

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

## Internet-of-Things (IoT)

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

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

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

✔ **Internet-of-Things Advantages**

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

**Internet-of-Things Disadvantages**

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

---

## Embedded (System) PCs

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

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

✔ **Embedded System Advantages**

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

**Embedded System Disadvantages**

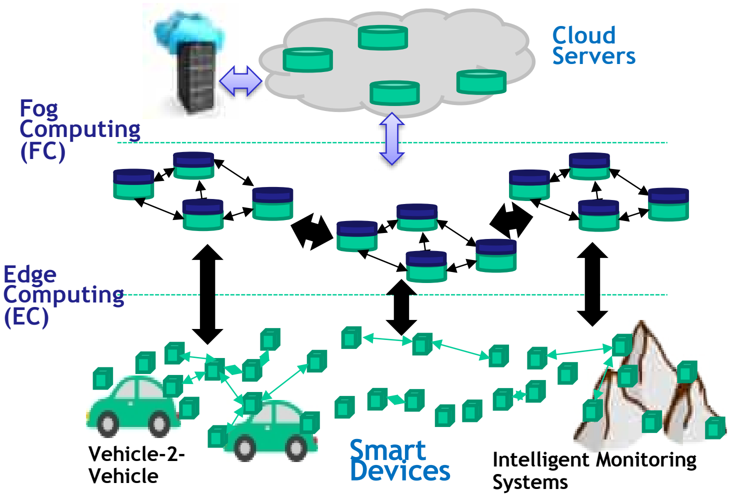
- **Pretty high power consumption**.
- **(Some) Hardware design has to be done**.

## Edge/Fog Computing Systems

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

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

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




### ✔ Fog/Edge Advantages

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

### Fog/Edge Disadvantages

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

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

Table 1: Differences between Edge and Fog Computing Systems.

#### 2.1.1.2 The Datacenter as a Computer

In the last few decades, computing and storage have moved from PC-like clients to smaller, often mobile, devices combined with extensive internet services. Furthermore, traditional enterprises are also shifting to Cloud computing.

The **advantages** of this migration are:

- **User-side**:
  - **Ease of management** (no configuration or backups needed);
  - The **availability of the service is everywhere**, but we need connectivity.
- **Vendors-side**:
  - SaaS (Software-as-a-Service) allows **faster application development** (more accessible to make changes and improvements);
  - **Improvements and fixes** in the software are **more straightforward inside their data centers** (instead of updating many millions of clients with peculiar hardware and software configurations);
  - The **hardware deployment** is restricted to a **few well-tested configurations**.
- **Server-side**:
  - **Faster introduction of new hardware devices** (e.g., HW accelerators or new hardware platforms);
  - Many application **services can run at a low cost per user**.

Finally, another advantage is that **some workloads require so much computing capability that they are a more natural fit in the datacenter** (and not in client-side computing). For example, the search services (web, images, and so on) or the Machine and Deep Learning.

#### 2.1.1.3 Warehouse-Scale Computers

The trends toward server-side computing and widespread internet services created a new class of computing systems: **Warehouse-Scale Computers**.

**Definition 1**

**Warehouse-Scale Computers (WSCs)** is intended to draw attention to the most distinctive feature of these machines: **the massive scale of their software infrastructure, data repositories and hardware platform**.

**What is a *program* at a WSC?**

In Warehouse-Scale Computing **the program is an internet service**, which may **consist of tens or more individual programs that interact to implement complex end-user services** such as *email*, *search*, or *maps*. These programs might be implemented and maintained by different teams of engineers, perhaps even across organizational, geographic, and company boundaries.

**Difference between WSCs and Data Centers**

WSCs currently power the services offered by companies such as Google, Amazon, Microsoft, and others. The main difference from traditional data centers (see more on page 5) is that **WSCs belong to a single organization, use a relatively homogeneous hardware and system software platform, and share a common systems management layer**. In contrast with the typical data center that belongs to multiple organizational units or even different companies, use dedicated HW infrastructure in order to run a large number of applications (more details on page 5).

**How is the WSC organized?**

The **software on WSCs**, such as Gmail, runs on a scale far beyond a single machine or rack: it **runs on clusters of hundreds to thousands of individual servers**. Therefore, the machine, the computer, is itself this **large cluster or aggregation of servers** and must be **considered a single computing unit**.

Most importantly, WSCs run fewer vast applications (internet services). An advantage is that the **shared resource management infrastructure allows significant deployment flexibility**. Finally, the requirements of:

- **Homogeneity**
- **Single-Organization Control**
- **Cost Efficiency**

Motivate designers to take new approaches to constructing and operating these systems.

#### 2.1.1.4 Multiple Data Centers

Sometimes the data centers are **located far apart**. **Multiple data centers** are (often) **replicas of the same service**:

- To *reduce user latency*
- To *improve service throughput*

Typically, a request is fully processed within one data center.

The world is divided into **Geographic Areas (GAs)**. Each Area is defined by Geo-political boundaries (or country borders). Also, there are at least two computing regions in each geographical Area.

The **Computing Regions (CRs)** are the smallest geographic unit of the infrastructure from the customer's perspective. Multiple Data centers within the same region are not exposed to customers.

However, they are defined by a latency-defined perimeter, typically less than 2ms for round-trip latency. Finally, they're located hundreds of miles apart, with considerations for different flood zones, etc. It is too far from synchronous replication but suitable for disaster recovery.

The **Availability Zones (AZs)** are finer-grain **locations within a single computing region**. They allow customers to run mission-critical applications with high availability and fault tolerance to Data Center failures. Because there are fault-isolated locations with redundant power, cooling, and networking (they are different from the concept of the Availability Set).

This hierarchical structure ensures efficient data management and compliance with local data laws while optimizing network performance through strategically placing data centers.

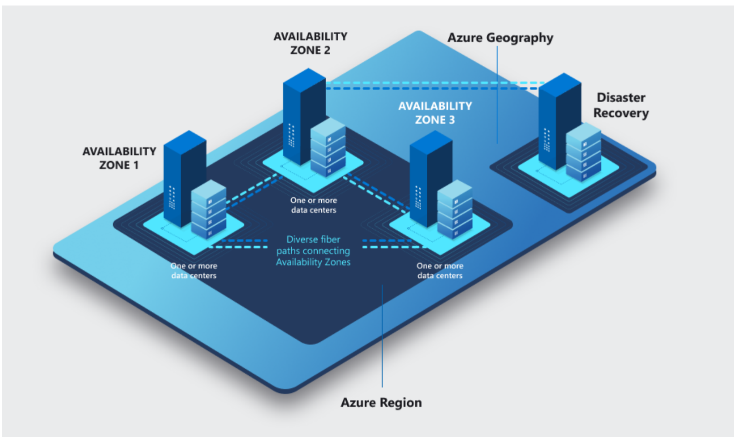



Figure 3: Example of Azure Availability Zones.

#### 2.1.1.5 Warehouse-Scale Computing / Data Centers Availability

The services provided through WSCs (or DCs) **must guarantee high availability**, typically aiming for at least 99.99% uptime (e.g. one hour of downtime per year).

Some **examples**:

- 99,90% on single instance VMs with premium storage for a more accessible lift and shift;
- 99,95% VM uptime SLA for Availability Sets (AS) to protect for failures within a data center;
- 99,99% VM uptime SLA through Availability Zones.

Such fault-free operation is more accessible when an extensive collection of hardware and system software is involved.

**WSC workloads must be designed to gracefully tolerate large numbers of component faults with little or no impact on service level performance and availability!**

---

#### 2.1.1.6 Architectural overview of Warehouse-Scale Computing

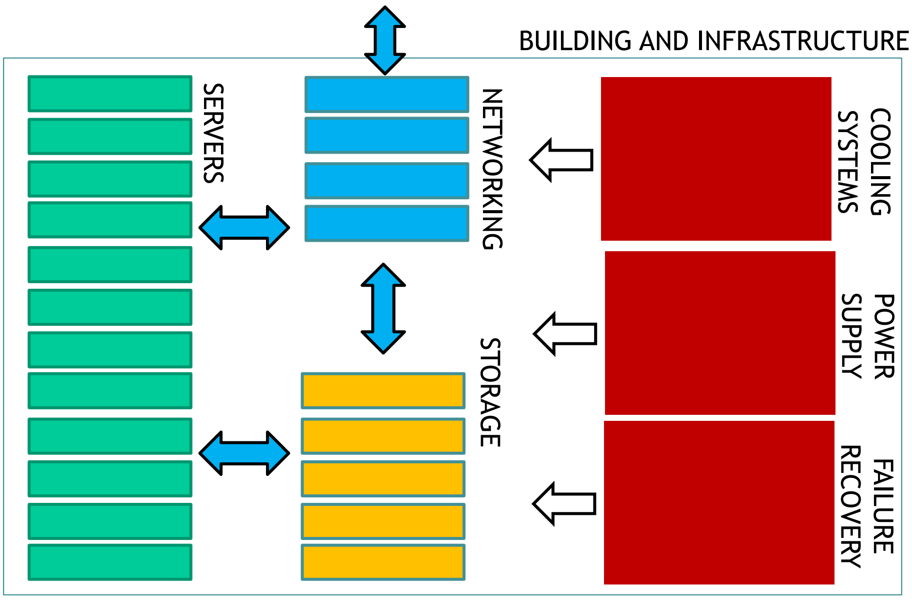



Figure 4: Architectural overview of Warehouse-Scale Computing.

- **Server** (section 2.2.1, page 16). Servers are the **leading processing equipment**: different types according to CPUs, RAM, local storage, accelerators, and form factor. The servers are **hosted on individual shelves** and are the **basic building blocks of Data Centers and Warehouse-Scale Computers**. They are interconnected by hierarchies of networks and supported by the shared power and cooling infrastructure.

- **Storage** (section 2.2.2, page 23). Disks, flash SSDs, and Tapes are the **building blocks** of today's **WSC storage systems**. These devices are **connected to the Data Center network and managed by sophisticated distributed systems**.

  Some **examples**:

  - Direct Attached Storage (DAS)
  - Network Attached Storage (NAS)
  - Storage Area Networks (SAN)
  - RAID controllers

- **Networking** (section 2.2.3, page 55). The **Data Center Network (DCN) enables efficient data transfer and interaction between various components**. The data processing ecosystem within the DCs needs to reach the DC services from outside. Communication equipment includes switches, Routers, cables, DNS or DHCP servers, Load balancers, Firewalls, etc.

- **Building and Infrastructure**. WSC has other essential components related to power delivery, cooling, and building infrastructure that must be considered. Some interesting numbers:

  - Data Centers with up to 110 football-pitch size.
  - 2-100s MW power consumption (100k houses), and the largest in the world is 650 MW.

## 2.2 Node-level

### 2.2.1 Server (computation, HW accelerators)

Servers are like ordinary PCs, usually more powerful, but with a **form factor that allows them to fit into the shelves** (such as rack, blade enclosure format, or tower; the differences are explained later). They are usually built in a tray or blade enclosure format, **housing the motherboard**, the **chipset**, and **additional plug-in components**.

The **motherboard** acts as the central hub, **connecting all the crucial components of the server and enabling them to communicate and work together**.

It provides sockets and plug-in slots to install CPUs, memory modules (DIMMs), local storage (such as Flash SSDs or HDDs), and network interface cards (NICs) to satisfy the range of resource requirements.

The **chipsets** and **additional components** are grouped in the following way:

- Number and type of CPUs:
  - From 1 to 8 CPU socket.
  - Intel Xeon Family, AMD EPYC, etc.
- Available RAM:
  - From 2 to 192 DIMM Slots.
- Locally attached disks:
  - From 1 to 24 Drive Bays.
  - HDD or SSD.
  - SAS (higher performance but more expensive) or SATA (for entry-level servers).
- Other special purpose devices:
  - From 1 to 20 GPUs per node, or TPUs.
  - NVIDIA Pascal, Volta, etc.
- Form factor:
  - From 1 unit to 10 units.
  - Tower.

### Differences between Rack, Blade and Tower

**Tower Server**

A **Tower Server** looks and feels much like a **traditional** tower **PC**.

✔ Advantages

- ✓ **Scalability** and **ease of upgrade**. Customized and upgraded based on necessity.
- ✓ **Cost-effective**. Tower servers are probably the *cheapest of all kinds of servers.*
- ✓ **Cools easily**. Since a tower server has a low overall component density, it cools down easily.

Disadvantages

- ✗ **Consumes a lot of space**. These servers are difficult to manage physically.
- ✗ **Provides a basic level of performance**. A tower server is *ideal for small business that have a limited number of clients.*
- ✗ **Complicated cable management**. Devices aren't easily routed together.

#### 2.2.1.1 Rack Servers

**Rack Servers** are unique **shelves that accommodate all the IT equipment** and allow their interconnection. The racks are used to store these rack servers.

Server racks are measured in **Rack Units**, or "U". One U is approximately 44.45 millimeters. The main advantage of these racks is that they **allow designers to stack up other electronic devices and servers**.

A rack server is designed to be positioned in a bay by vertically stacking servers one over the other along with other devices (storage units, cooling systems, network peripherals, and batteries).

✔ **Advantages**

- ✓ **Failure containment**. Very little effort to identify, remove, and replace a malfunctioning server with another.
- ✓ **Simplified cable management**. Easy and efficient to organize cables.
- ✓ **Cost-effective**. Computing power and efficiency at relatively lower costs.

**Disadvantages**

- ✗ **Power usage**. Needs of additional cooling systems due to their high overall component density, thus consuming more power.
- ✗ **Maintenance**. Since multiple devices are placed in racks together, maintaining them gets considerably though with the increasing number of racks.

#### 2.2.1.2 Blade Servers

**Blade Servers** are the latest and the most advanced type of servers in the market. They can be termed hybrid rack servers, where servers are placed inside blade enclosures, forming a blade system.

The **most significant advantage** of blade servers is that these servers are the **most minor types of servers available now** and are **great for conserving space**.

Finally, a blade system also meets the IEEE standard for rack units, and each rack is measured in the units of "U".

**✔ Advantages**

- ✓ **Size and form factor**. They are smallest and the most compact servers, requiring minimal physical space. Blade servers offer *higher space efficiency* compared to traditional rack-mounted servers.
- ✓ **Cabling**. Blade server don't involve the cumbersome tasks of setting up cabling. Although you still might have to deal with the cabling, it is near to negligible when compared to tower and rack servers.
- ✓ **Centralized management**. Blade enclosures typically come with centralized management tools that allow administrators to easily monitor, configure and update all blades from a single interface.
- ✓ **Load balancing, failover, scalability**. Uniform system, shared components (including network), simple addition/removal of servers.

**Disadvantages**

- ✗ **Expensive configuration and Higher initial cost**. Although upgrading the blade server is easy to handle and manage, the initial configuration or the setup requires more effort and higher initial investment.
- ✗ **Vendor Lock-In**. Blade servers typically require the use of the manufacturer's specific blades and enclosures, leading to vendor lock-in. This can limit flexibility and potentially increase costs in the long run.
- ✗ **Cooling**. Blade servers come with high component density. Therefore, special accommodations have to be arranged for these servers to ensure they don't get overheated. Heating, ventilation, and air conditioning systems (HVAC) must be carefully managed and designed.

#### 2.2.1.3 Machine Learning

**Deepening: Machine Learning (supervised learning)**

**Machine learning** (ML) is a branch of artificial intelligence (AI) and computer science that focuses on the using data and algorithms to enable AI to imitate the way that humans learn, gradually improving its accuracy (source).

UC Berkeley breaks out the learning system of a machine learning algorithm into three main parts:

1. A Decision Process: In general, machine learning algorithms are used to make a prediction or classification. Based on some input data, which can be labeled or unlabeled, your algorithm will produce an estimate about a pattern in the data.

2. An Error Function: An error function evaluates the prediction of the model. If there are known examples, an error function can make a comparison to assess the accuracy of the model.

3. A Model Optimization Process: If the model can fit better to the data points in the training set, then weights are adjusted to reduce the discrepancy between the known example and the model estimate. The algorithm will repeat this iterative "evaluate and optimize" process, updating weights autonomously until a threshold of accuracy has been met.

The main goal is to learn a target function that can be used for prediction. Given a training set of labeled examples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, estimate the prediction function $f$ by minimizing the prediction error on the training set:

$$y = f(x)$$

Where $y$ is the output, $f$ is the prediction function and the $x$ is an image feature.

**Deepening: Artificial Neural Network**

The **Artificial Neural Network** is a computational model inspired by the human brain (perceptron). It consists of interconnected nodes (neurons) organized in layers to process and analyze data and used to learn data representation from data (learn features and the classifier/regressor).

The learning process of a Neural Network is as follows: Neurons make decisions (activation functions). There are wights, so the connections between neurons are strengthened or weakened through training- randomly initialized.
The (training data) Neural Networks learn from historical data and ex-

amples. Then, labeled data are provided.

**Deepening: effects of ML and ANN**

Deep learning models began to appear and be widely adopted, enabling specialized hardware to power a broad spectrum of machine learning solutions.
Since 2013, AI learning compute requirements have doubled every 3.5 months (vs. 18-24 months expected from Moore's Law).
To satisfy the growing compute needs for deep learning, **WSCs deploy specialized accelerator hardware**:

- Graphical Processing Units (GPUs) are used for data-parallel computations (the same program is executed on many data elements in parallel). In order to use parallel programming, high-level languages such as CUDA, OpenCL, OPENACC, OpenMP, and SYCL exist. This technique allows up to 1000x faster than CPU.
- Tensor Processing Unit (TPU), where Tensor is a n-dimensional matrix, are used for training and inference.
- Field-Programmable Gate Array (FPGA) are programmable hardware devices. The device user can program an array of logic gates ("configured") in the field instead of the people who designed it. An array of carefully designed and interconnected digital subcircuits that efficiently implement common functions, offering very high levels of flexibility. The digital subcircuits are called configurable logic blocks (CLBs).

  FPGA Applications in Data Centers:

  - Network acceleration: FPGAs can offload specific processing tasks from CPUs, improving overall network performance and reducing CPU workload.
  - Security acceleration: Encryption, decryption, and other security-related tasks can be accelerated using FPGAs, enhancing data centre security while maintaining performance.
  - Data analytics: FPGAs can accelerate specific algorithms in data analytics workloads, leading to faster data processing and analysis.
  - Machine learning: FPGAs can be configured to efficiently implement specific machine learning algorithms, potentially offering performance advantages for specialized tasks.

| | Advantages | Disadvantages |
|---|---|---|
| **CPU** | • Easy to be programmed and support any programming framework.<br>• 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.<br>• 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).<br>• Limited flexibility compared to other platforms. |

### 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. Due to its mechanical movement, the **solid-state drive (SSD)** is the best solution (quality-price) because there are no mechanical or moving parts, and they are built out of transistors (NADN flash-based devices). The **non-volatile memory express (NVMe)** also exists, which is the **latest industry standard** for running PCIe[2] SSDs.

As for price classification, we can see that the NVMe is the most expensive solution:

1. NVMe (between 100€ and 200€ for 1TB)
2. SSD (between 70€ and 100€ for 1TB)
3. HDD (between 40€ and 60€ for 1TB)

For these reasons, it is reasonable to use a **hybrid solution** (HDD + SSD):

- 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

An **OS can see the disks as a collection of data blocks** that can be read or written independently. To allow the ordering/management among them, **each block is characterized by a unique numerical address** called **LBA (Logical Block Address)**. Typically, the **OS groups blocks into clusters**[3] **to simplify the access to the disk**. Typical cluster sizes range from 1 disk sector (512 B, or 4 KB) to 128 sectors (64 KB).

Each *cluster* contains:

- **File data**. The actual content of the files.
- **Metadata**. The information required to support the file system:
  - File names
  - Directory structures and symbolic links
  - Creation, modification, and last access dates
  - Security information (owners, access list, encryption)
  - **Links to the LBA where the file content can be located on the disk**

The disk can thus contain **several types of clusters**:

- Metadata:
  - Fixed position (to bootstrap the entire file system)
  - Variable position (to hold the folder structure)
- File data (the actual content of the files)
- Unused space (available to contain new files and folders)

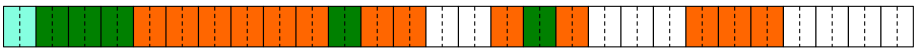
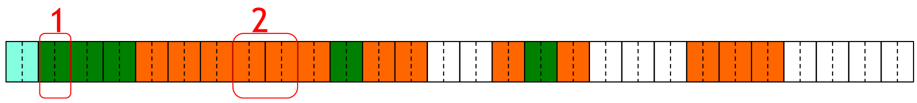


Figure 5: A cluster can be seen visually as an array. In this image, for example, we've shown three types of cluster: metadata fixed position (azure), metadata variable position (green), file data (orange), unused space (white).

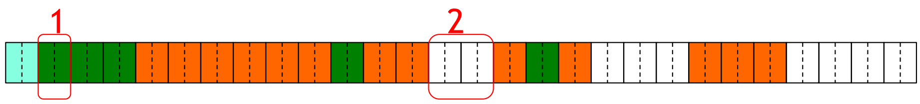
[3]**Clusters** are the minimal units an OS can read from or write to a disk.

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

- **Reading**. To read a file:
  1. Access the metadata, variable position (because it contains the folder structure), to locate its block;
  2. Access the blocks to read the contents of the file.




- **Writing**. To write a file:
  1. Access the metadata, variable position (because it contains the folder structure), to find free space.
  2. Write the data in the allocated blocks (cluster type: unused space).




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:

$$a = \text{ceil}\left(\frac{s}{c}\right) \times c \tag{1}$$

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 1: internal fragmentation**

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

The *actual size* on the disk is:

$$a = \text{ceil}\left(\frac{27}{8}\right) \cdot 8 = \text{ceil}\left(3.375\right) \cdot 8 = 4 \cdot 8 = 32 \text{ byte}$$

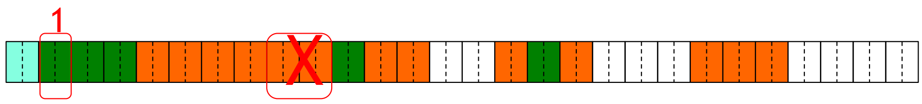
And the internal fragmentation $w$ is:

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

- **Deleting**. To delete a file:

  1. Just update the metadata, variable position (because it contains the folder structure), to say that the blocks where the file was stored are no longer used by the OS.

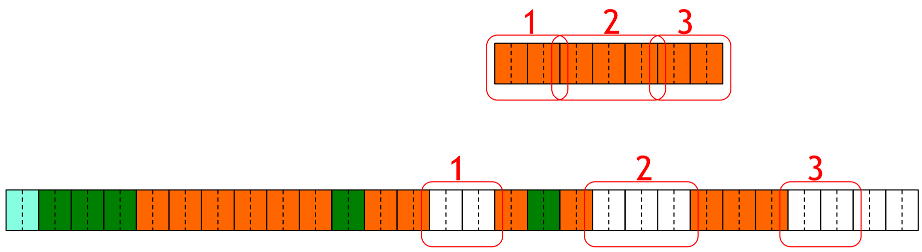
  **Deleting a file never actually deletes the data on the disk**: if a new file is written to the same clusters, the old data is replaced by the new.




- **External fragmentation**. As the disk's life evolves, there might **not be enough space to store a file contiguously**.

  In this case, the file is split into smaller chunks and inserted into the free clusters spread over the disk.

  The effect of **splitting a file into non-contiguous clusters** is called **external fragmentation**.



#### 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.

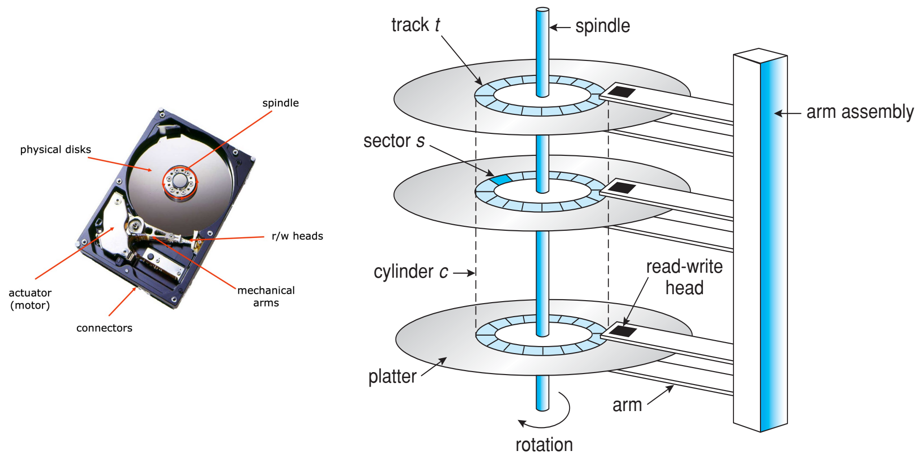



Figure 6: Hard Drive Disk anatomy.

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**[4]).

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)**.

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

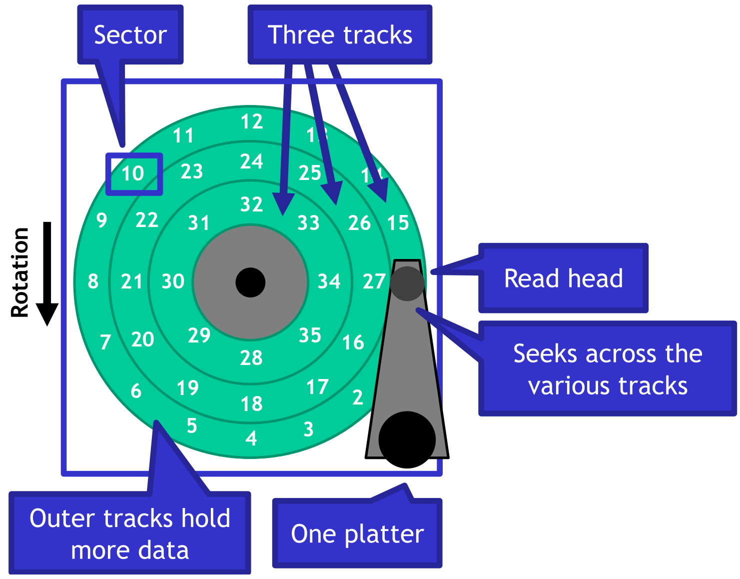



Figure 7: Example of HDD geometry.

Given the architecture of the HDD, there are **four types of delay**:

- **Rotational Delay** is the **time to rotate the desired sector to the read head**, and it's related to RPM.
- **Seek Delay** is the **time to move the read head to a different track**.
- **Transfer time** is the **time to read or write bytes**.
- **Controller Overhead** is the **overhead for the request management**.

In order to reduce the delay in the HDD, the companies use a high-speed and tiny memory (8, 16 or 32 MB) called **cache** (also called **track buffer**). The cache is used when there is a:

- **Read caching**. It reduces read delays due to seeking and rotation.
- **Write caching**. It is divided into two different implementations:
  - **Write Back cache**: the drive reports that writes are complete after they have been cached. The disadvantage is that it has an inconsistent state if the power goes off before the write-back event.
  - **Write Through cache**: the drive reports that writes are complete after they have been written to disk.

So, the **caching helps improve disk performance**. The critical idea under caching is that if there is a **queue of requests to the disk, they can be reordered to improve performance**. The estimation of the request length is feasible, knowing the position of the data on the disk.

There are several **scheduling algorithms**:

- **First Come, First Serve (FCFC)**. It is the most basic scheduler, serving requests in order. The ***disadvantage*** is that there is much time spent seeking.

- **Shortest Seek Time First (SSTF)**. It is primary purpose is to minimize seek time by always selecting the block with the shortest seek time. The ***advantage*** is that it is optimal and can be easily implemented. The main ***disadvantage*** is that it is prone to starvation.

- **SCAN**, otherwise known as the Elevator Algorithm. The head sweeps across the disk, servicing requests in order. The ***advantage*** is that it performs reasonably well and does not suffer starvation. The ***disadvantage*** is that the average access times are higher for requests at high and low addresses.

- **C-SCAN (Circular SCAN)**. It is like the SCAN algorithm, but only service requests in one direction. The ***advantage*** is fairer than SCAN. However, the ***disadvantage*** is that it has worse performance than SCAN.

- **C-LOOK**. It is a C-SCAN variant that peeks at the upcoming addresses in the queue. The head only goes as far as the last request.

#### 2.2.2.3 SSD

The **solid-state drive (SSD)** does not have mechanical or moving parts like an HDD. It is built out of transistors (like memory and processors). It has higher performance than HDD.

It stores bits in cells. Each cell can have:

- Single-Level Cell (SLC), a single bit per cell.
- Multi-Level Cell (MLC), two bits per cell.
- Triple-Level Cell (TLC), three bits per cell.
- And so on... QLC, PLC, etcetera.

Internally, the SSD has a lot of NAND flashes, which are organized into Pages and Blocks. Some terminology:

- A **Page** contains **multiple logical block** (e.g. 512 B - 4 KB) **addresses** (LBAs). It is the **smallest unit that can be read/written**. It is a sub-unit of an erase block and consists of the number of bytes which can be read/written in a single operation. The states of each page are:
  - **Empty** (`ERASED`): it does not contain data.
  - **Dirty** (`INVALID`): it contains data, but this data is no longer in use (or never used).
  - **In use** (`VALID`): the page contains data that can be read.
- A Block (or **Erase Block**) typically consists of **multiple pages** (e.g. 64) with a total capacity of around 128-256 KB. It is the **smallest unit that can be erased**.

When passing from the HDD to SDD, there is a problem known as Write Amplification (WA). **Write amplification (WA)** is an **undesirable phenomenon associated with flash memory and solid-state drives (SSDs)** where the actual amount of information physically written to the storage media is a multiple of the logical amount intended to be written.

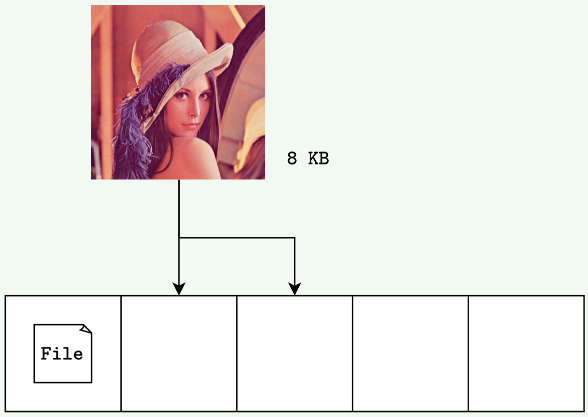
**Example 2**

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

Let us write a 4 KB text file to the brand-new SSD. The overall writing time is 4 seconds (write speed $\times$ file dimension, 1 KB/s $\times$ 4 KB).

Now, let us write an 8 KB picture file for the almost brand-new SSD; thankfully, there is space. The overall Writing time is 8 seconds, and the calculation is the same as above.




Now, consider that the first file inserted on the first page is unnecessary.

Finally, let's write a 12 KB pic to the SSD. Theoretically, the image should take 12 seconds. However, it is wrong! The SSD has only two empty pages and one dirty page (invalid). Then, the operations are:

1. Read block into cache.
2. Delete page from cache (set dirty page).
3. Write a new picture into the cache.
4. Erase the old block on the SSD.
5. Write cache to SSD.

The OS only thought it was writing 12 KBs of data when the SSD had to read 8 KBs (2 KB/s) and then write 20 KBs (1 KB/s), the entire block. The writing should have taken 12 seconds but took $4 + 20 = 24$ seconds, resulting in a write speed of 0.5 KB/s, not 1 KB/s.

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*.

The **Flash Translation Layer (FTL)** is placed in the hierarchy between the **File System** and **Flash Memory**. It aims to do **three main actions**:

1. **Data Allocation** and **Address Translation**: It efficiently reduces Write Amplification effects (see page 30); the program pages within an erased block in order (from low to high pages), called **Log-Structured FTL**.
2. **Garbage collection**: reuse of pages with old data (Dirty/Invalid).

3. **Wear levelling**: FTL should spread across the blocks of the flash, ensuring that all of the blocks of the device wear out roughly simultaneously.

**Example 3: Log-Structured FTL**

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(101, a2)`
- `Write(2000, b1)`
- `Write(2001, b2)`
- `Write(100, c1)`
- `Write(101, 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 (`Write(100, a1)`):

Table: 100 → 0 — Memory

| Block: | 0 | | | | 1 | | | | 2 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | Flash |
| Content: | a1 | | | | | | | | | | | | Chip |
| State: | V | E | E | E | i | i | i | i | i | i | i | i | |

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

Table: 100 → 0 101 → 1 2000 → 2 2001 → 3 — Memory

| Block: | 0 | | | | 1 | | | | 2 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | Flash |
| Content: | a1 | a2 | b1 | b2 | | | | | | | | | Chip |
| State: | V | V | V | V | i | i | i | i | i | i | i | i | |

5. After updating 100 and 101:

| Table: | 100 → 4 | 101 → 5 | 2000 → 2 | 2001 → 3 | Memory |
|---|---|---|---|---|---|

| Block: | 0 | | | | 1 | | | | 2 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | Flash |
| Content: | a1 | a2 | b1 | b2 | c1 | c2 | | | | | | | Chip |
| State: | V | V | V | V | V | V | E | E | i | i | i | i | |

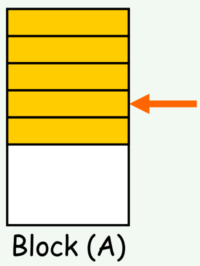
When an existing page is updated, then the old data becomes obsolete. The **old versions of data are called garbage**, and (sooner or later) garbage pages must be reclaimed for new writes to take place.

The **Garbage Collection** is the **process of finding garbage blocks and reclaiming them**. It is a simple process for fully garbage blocks but more complex for partial cases.

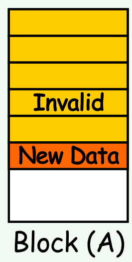
**Example 4: how garbage collection works**

The steps are:

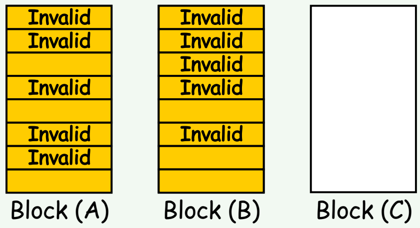
1. Update request for existing data:



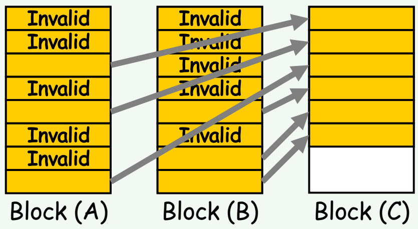

2. Find a free page, and save the new data:



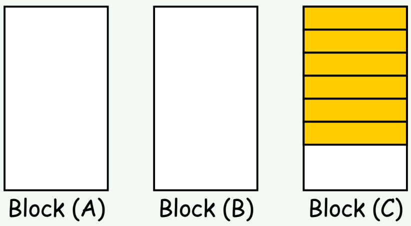

3. This scenario may continue until there are not enough free blocks:




4. Collect valid pages into a free block:




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




### Problem 1: the Garbage Collection is too expensive!

The Garbage Collection is **expensive**. It requires reading and rewriting of live data. Ideal garbage collection is a reclamation of a block that consists of only dead pages.

### Partial solution

Garbage Collection **costs depend on the amount of data blocks that must be migrated**. The <u>**solution**</u> to alleviate the problem is to overprovision the device by **adding extra flash capacity** (cleaning can be delayed) and **running the garbage collection in the background** using less busy periods for the disk.

### Problem 2: the Ambiguity of Delete

When performing background Garbage Collection, the SSD assumes to know which pages are invalid. However, most file systems do not truly delete data. For example, on Linux, the "delete" function is `unlink()`, removing the file meta-data but not the file itself.

1. File is written on SSD
2. File is deleted
3. The Garbage Collection executes:
   - 9 pages look valid to the SSD;
   - BUT the OS knows only 2 pages are valid.

### ✔ Partial solution

New SSD SATA command TRIM (SCSI - UNMAP). The **OS tells the SSD that specific LBAs** (page 24) **are invalid and may be garbaged by the Garbage Collection**.

### ⚠ Problem 3: Mapping Table Size

The **size of the page-level mapping table is too large**. In fact, with a 1 TB SSD with a 4-byte entry per 4 KB page, 1 GB of DRAM is needed for mapping!

### ✔ Partial solution

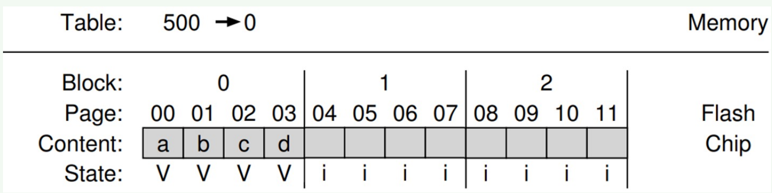
Exists some approaches to reduce the costs of mapping:

- **Block Mapping** (block-based mapping). Mapping at block granularity to reduce the size of a mapping table. With this technique, there is a small writing problem: the FTL must read a large amount of live data from the old block and copy it into a new one.

**Example 5: 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')`

Table: 500 → 4 Memory

| Block: | 0 | | | | 1 | | | | 2 | | | | Flash Chip |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | |
| Content: | | | | | a | b | c’ | d | | | | | |
| State: | E | E | E | E | V | V | V | V | i | i | i | i | |

- **Hybrid Mapping**. FTL maintains two tables: **log blocks** (page mapped) and **data blocks** (block mapped). The FTL will consult the page mapping table and block mapping table when looking for a particular logical block.

**Example 6: Hybrid Mapping**

Let’s suppose the following sequence:

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

Log Table:
Data Table: 250 → 8 Memory

| Block: | 0 | | | | 1 | | | | 2 | | | | Flash Chip |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | |
| Content: | | | | | | | | | a | b | c | d | |
| State: | i | i | i | i | i | i | i | i | V | V | V | V | |

Let’s update some pages:

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

Log Table: 1000 → 0 1001 → 1 1002 → 2 1003 → 3
Data Table: 250 → 8 Memory

| Block: | 0 | | | | 1 | | | | 2 | | | | Flash Chip |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | |
| Content: | a’ | b’ | c’ | d’ | | | | | a | b | c | d | |
| State: | V | V | V | V | i | i | i | i | V | V | V | V | |

When needed, FTL can perform MERGE operations:

Log Table:
Data Table: 250 → 0 Memory

| Block: | 0 | | | | 1 | | | | 2 | | | | Flash Chip |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Page: | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | |
| Content: | a’ | b’ | c’ | d’ | | | | | | | | | |
| State: | V | V | V | V | i | i | i | i | i | i | i | i | |

- **Page Mapping plus Caching**. The basic idea is to **cache the active part of the page-mapped FTL**. If a given workload only accesses a small set of pages, the translations of those pages will be stored in the FTL memory. It will perform well without high memory cost if the cache can contain the necessary working set. Cache miss overhead exists; we need to accept it.

### ✔ The importance of Wear Leveling

As we have mentioned, the wear leveling is essential. The erase/write cycle is limited in Flash Memory. All blocks should wear out roughly at the same time.

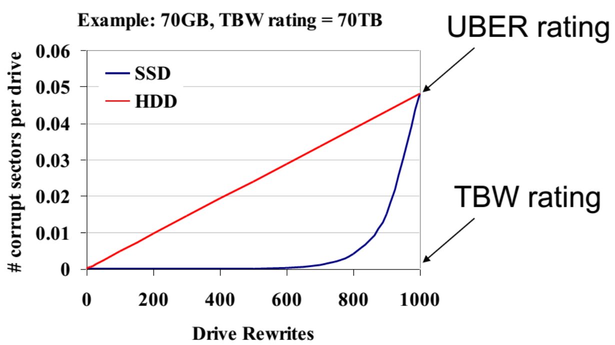
The log-structured approach and garbage collection help spread writing. However, a block may contain cold data: the FTL must periodically read all the live data from such blocks and re-write it elsewhere. A **disadvantage** is that the wear levelling **increases the write amplification** of the SSD and consequently decreases performance. 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:

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

### HDD vs SSD

Exists two metrics:

- **Unrecoverable Bit Error Ratio (UBER)**. A metric for the rate of occurrence of data errors, equal to the **number of data errors per bits read**.
- **Endurance rating: Terabytes Written (TBW)**. It is the total amount of data that can be written into an SSD before it is likely to fail. **The number of terabytes that may be written to the SSD while still meeting the requirements**.



#### 2.2.2.4 RAID

**RAID (Redundant Array of Independent Disks)** is a **data storage virtualization technology**[5] 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).

[5] 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 <u>not</u> study the levels `RAID 2` and `RAID 3`.



Table 2: 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?

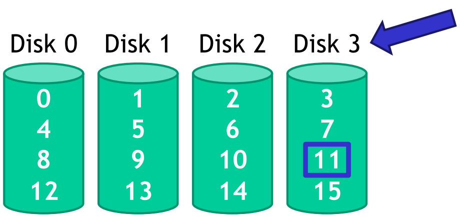
The key idea is to present an **array of disks as a single large disk** and maximize parallelism by striping data across all $N$ disks.

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:

$$\begin{aligned} \texttt{Disk} &= \texttt{logical\_block\_number} \,\%\, \texttt{number\_of\_disks} \\ \texttt{Offset} &= \frac{\texttt{logical\_block\_number}}{\texttt{number\_of\_disks}} \end{aligned} \tag{4}$$

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):

$$\begin{aligned} S &= \frac{\texttt{transfer\_size}}{\texttt{time\_to\_access}} \\ S &= 10 \text{ MB} \div (7 \text{ ms} + 3 \text{ ms} + 10 \text{ MB} \div 50 \text{ MB/s}) = 47.62 \text{ MB/s} \end{aligned}$$

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

$$\begin{aligned} R &= \frac{\texttt{transfer\_size}}{\texttt{time\_to\_access}} \\ R &= 10 \text{ KB} \div (7 \text{ ms} + 3 \text{ ms} + 10 \text{ KB} \div 50 \text{ MB/s}) = 0.98 \text{ MB/s} \end{aligned}$$

### 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

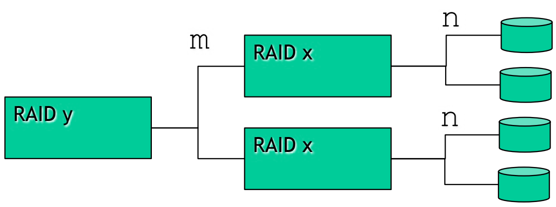
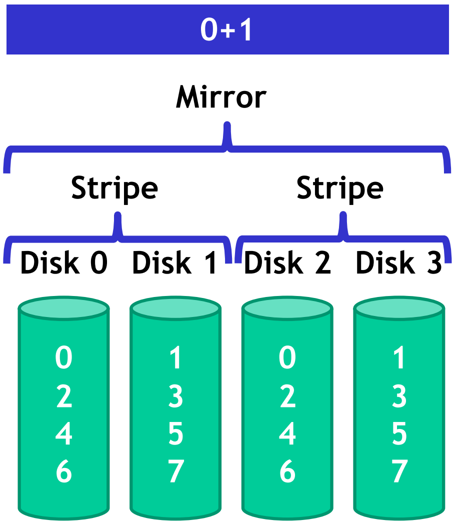



Figure 8: `RAID x + y` general architecture.

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)

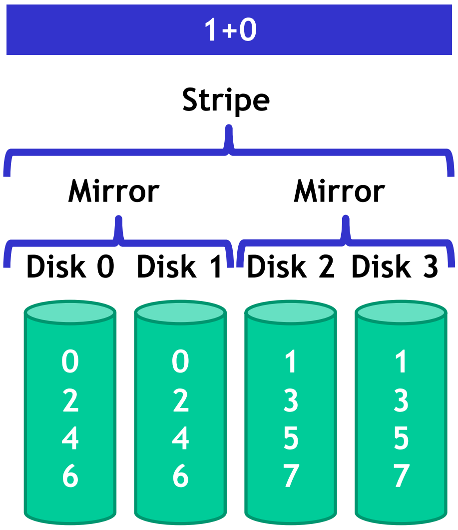
There are necessary almost four drives. Note that after the first failure, the model becomes a RAID 0.




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

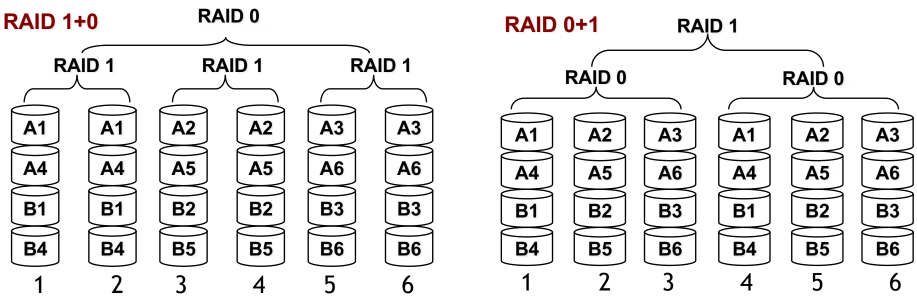
1. The striping first (RAID 0)
2. Then the mirroring (RAID 1)

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**[6]!

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`

[6] 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

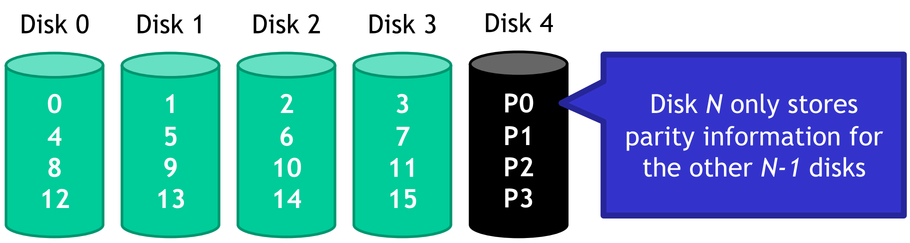
- **Good performance** of random reads.

### 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[7].




| 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 <u>**serial**</u> or <u>**random read**</u> 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.

[7] "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.

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

- **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.

| Disk 0 | Disk 1 | Disk 2 | Disk 3 | Disk 4 |
|---|---|---|---|---|
| 0 | 0 | 1 0 | 1 | 0 ^ 0 ^ 1 ^ 1 1 |

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 [9]

### 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?

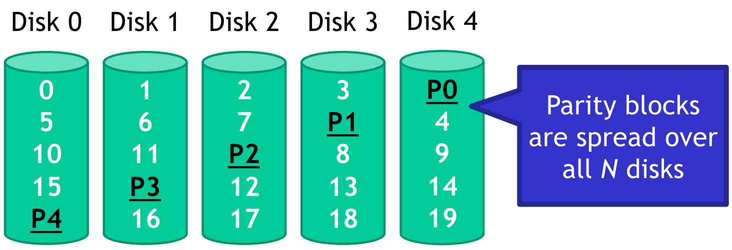
Unlike in RAID 4, parity information is distributed among the drives. This technique is **used to improve significantly the random write throughput** against the RAID 4 system.

### 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.

---

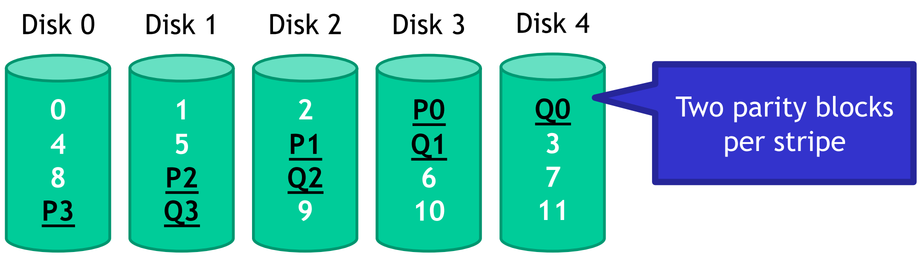
## RAID 6

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 3: 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.

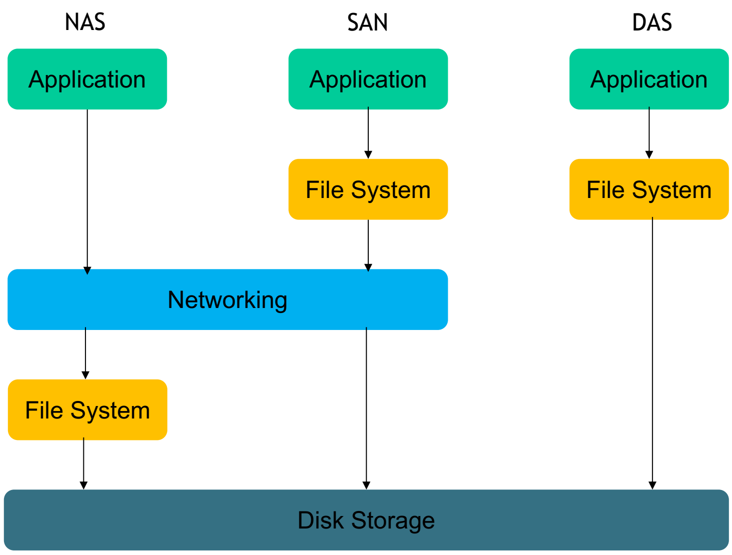
**RAID 5** has the better ***balance*** between space, performance, and recoverability.

#### 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).

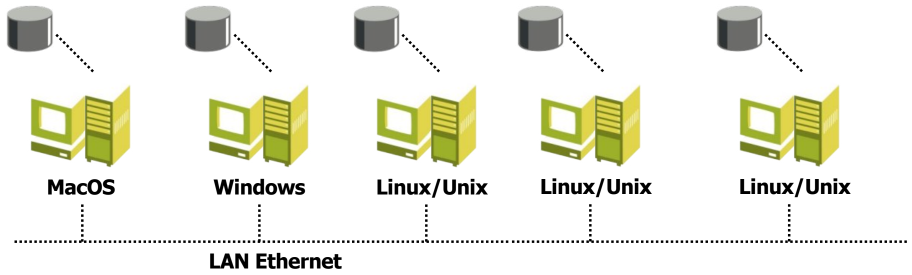



Figure 11: DAS architecture.

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.

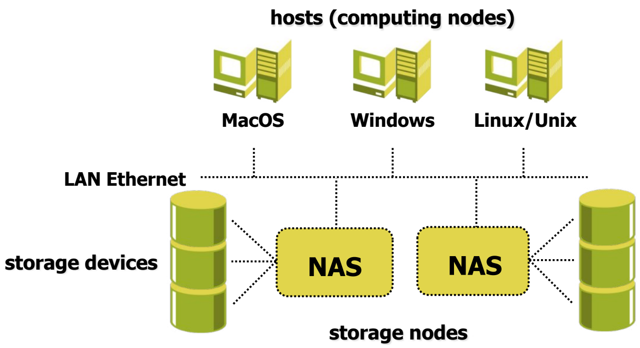



Figure 12: NAS architecture.

### ✔ 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.

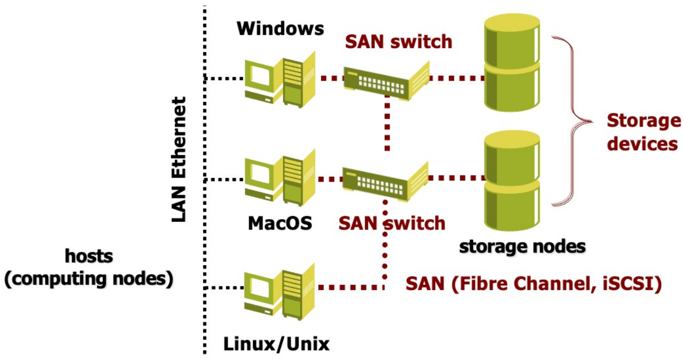



Figure 13: SAN architecture.

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

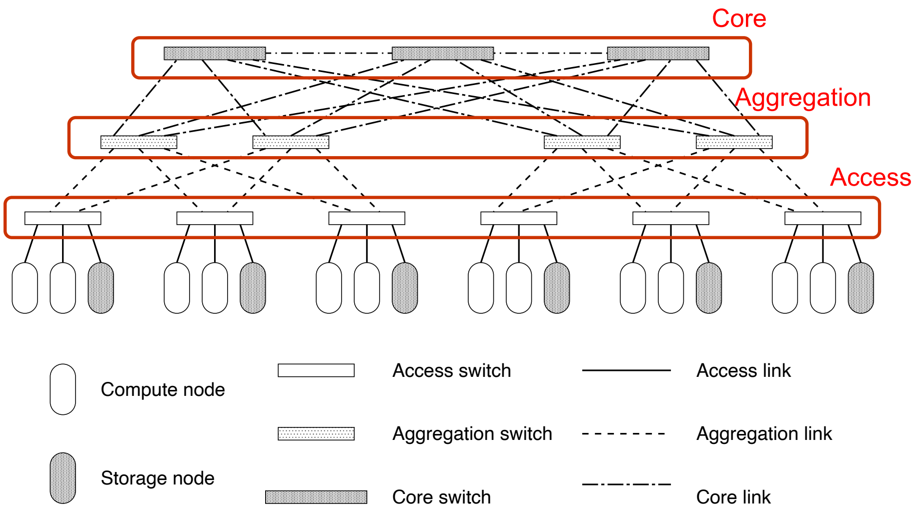
- **DCN Switch-centric architectures**. DCN uses **switches to perform packet forwarding**.[8]
- **DCN Server-centric architectures**. DCN **uses servers with multiple Network Interface Cards (NICs)**[9] **to act as switches** and perform other computational functions.
- **DCN Hybrid architectures**. DCN **combines switches and servers** for packet forwarding.

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

[9] 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.

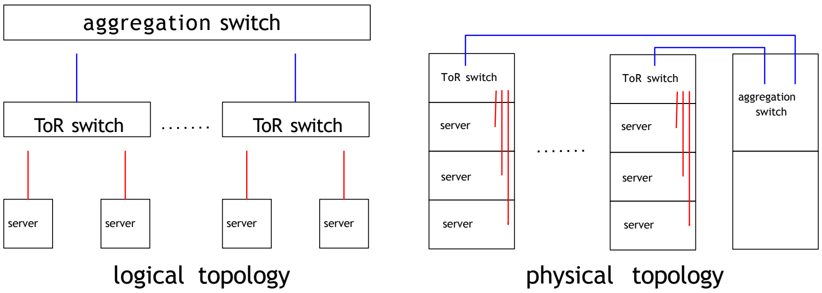



Figure 15: ToR (Top-of-Rack) architecture.

- **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**.

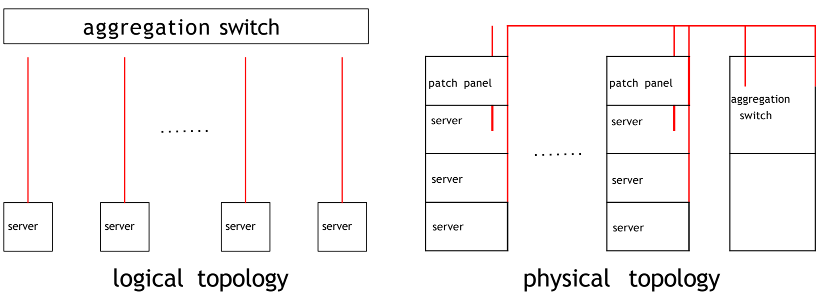



Figure 16: EoR (End-of-Row) architecture.

#### 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.

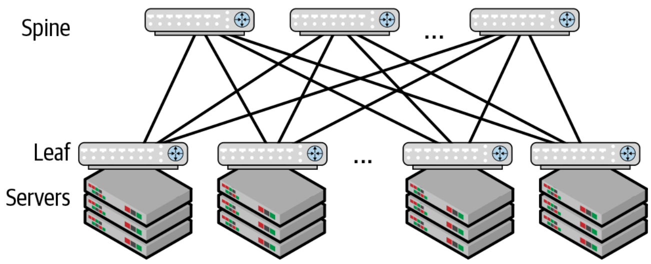



Figure 17: Leaf-Spine architecture.

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.

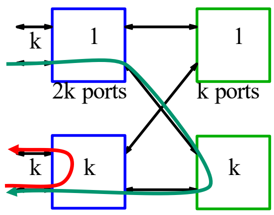


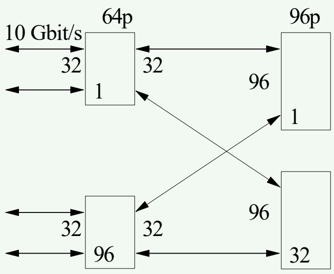

Figure 18: Explanation of Leaf-Spine architecture.

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

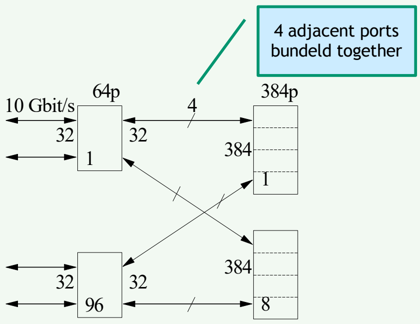
**Example 7**

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.



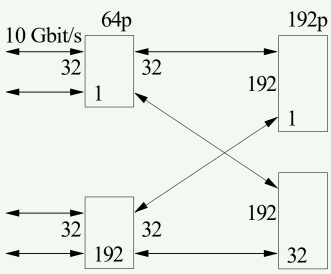

2. The second has only 8 switches but they have more ports: 384 $(8 \times 384 = 3072)$.



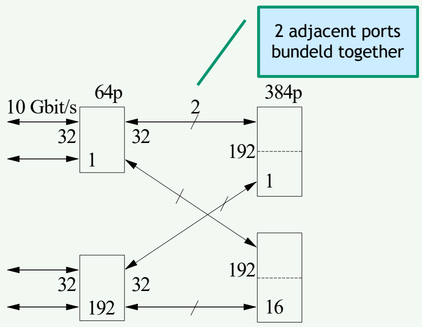

**Example 8**

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)**[10] 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 $\mathbf{2k^3}$ servers, there are $\mathbf{5k^2}$ 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.

[10]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.

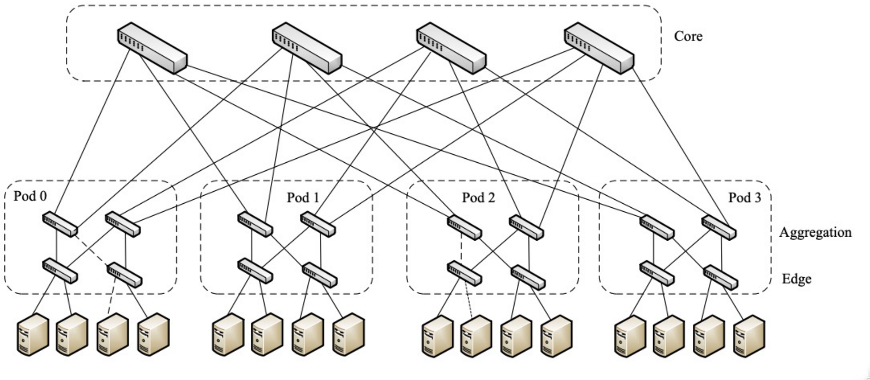



Figure 19: Fat-Tree Network, with $k = 2$, 4 pods, 16 servers, 20 switches.

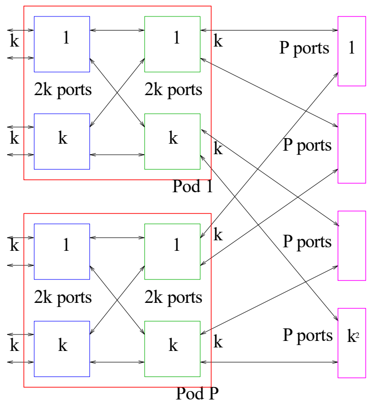



Figure 20: Explanation of Fat-Tree Network.

#### 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**.

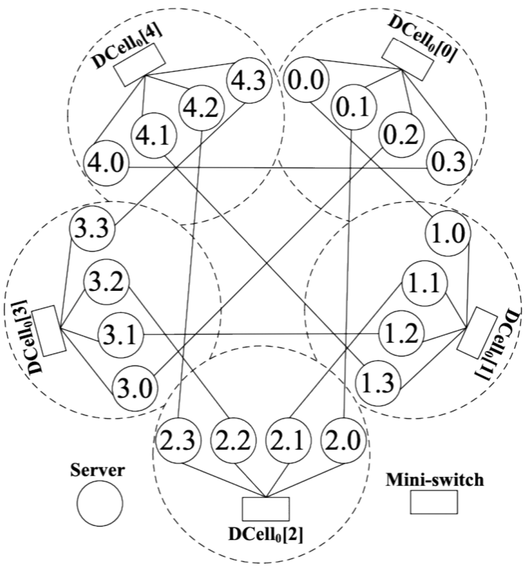


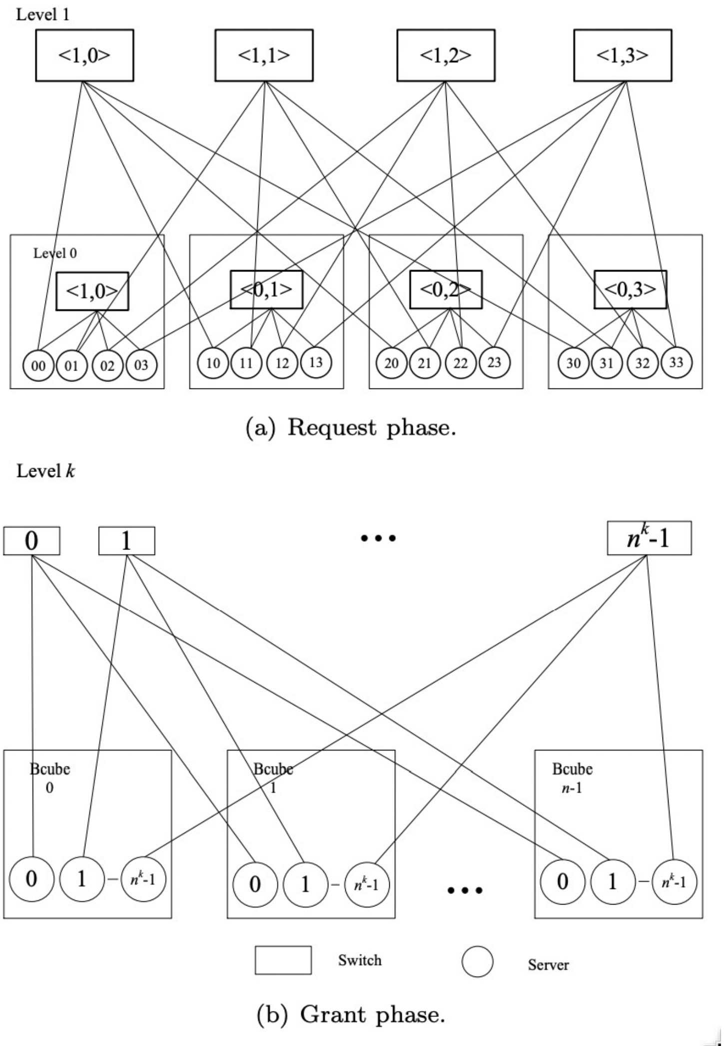

Figure 21: DCell hybrid architecture.

**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.

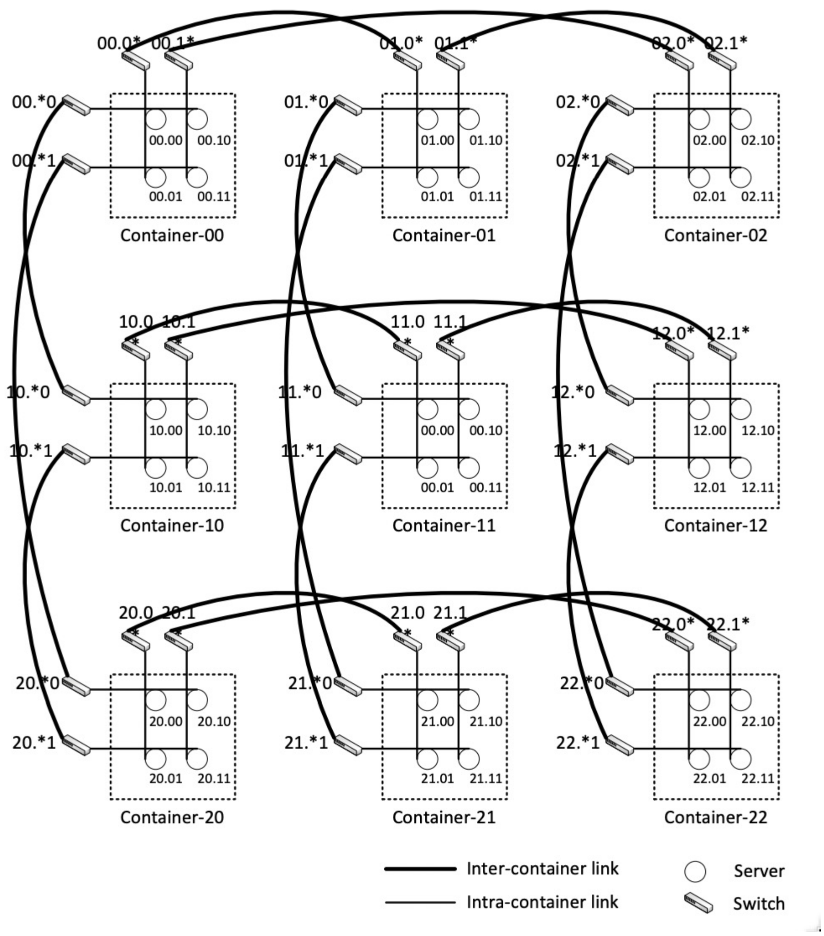



Figure 23: MDCube hybrid architecture.

## 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

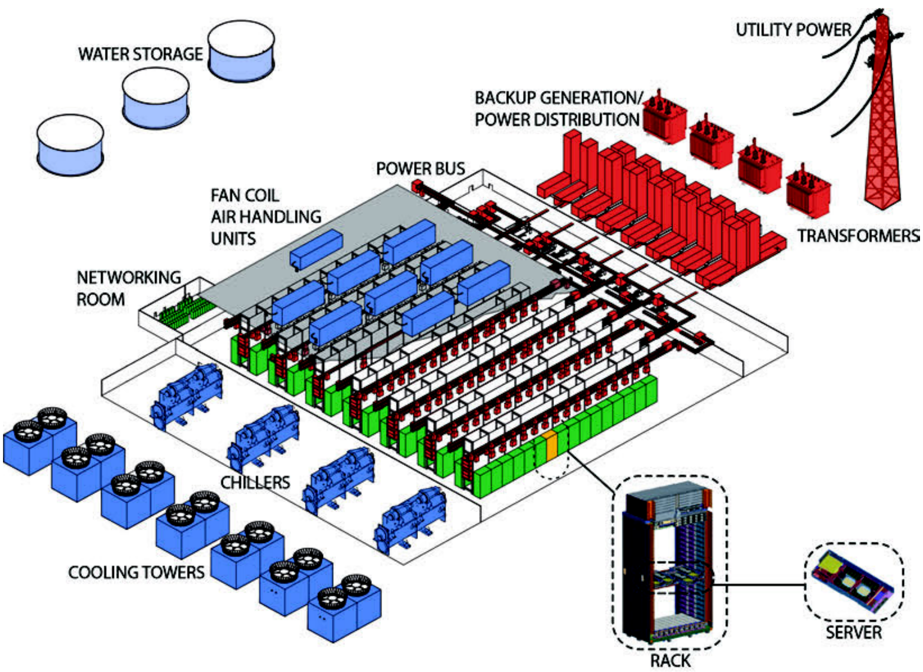



Figure 24: The main components of a typical data center. [1]

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.

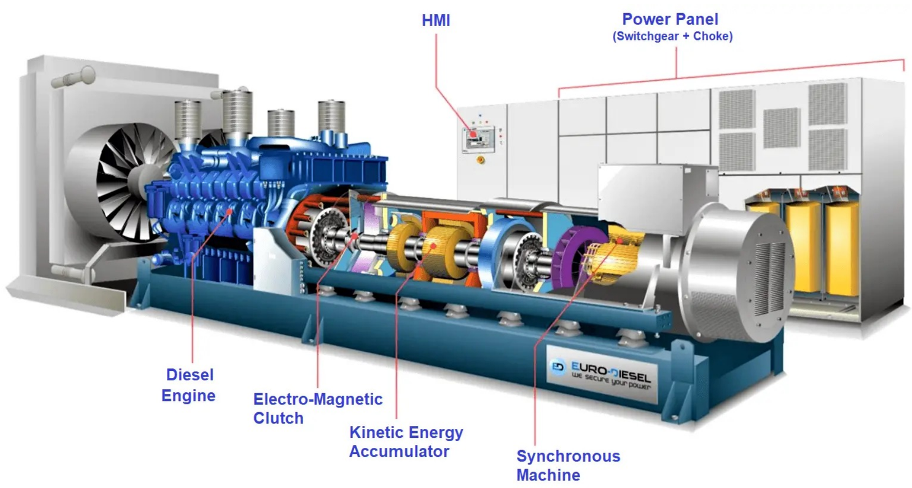



Figure 25: A Rotary UPS system.

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.

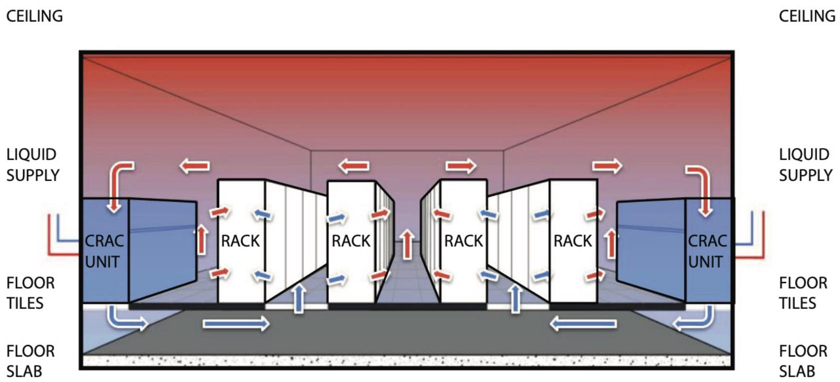
- ***Two loops***. The airflow through the underfloor plenum, the racks, and back to the **CRAC (computer room air conditioning)** defines the primary air circuit as the first loop. The second loop (the liquid supply inside the CRAC units) leads directly from the CRAC to the external heat exchangers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



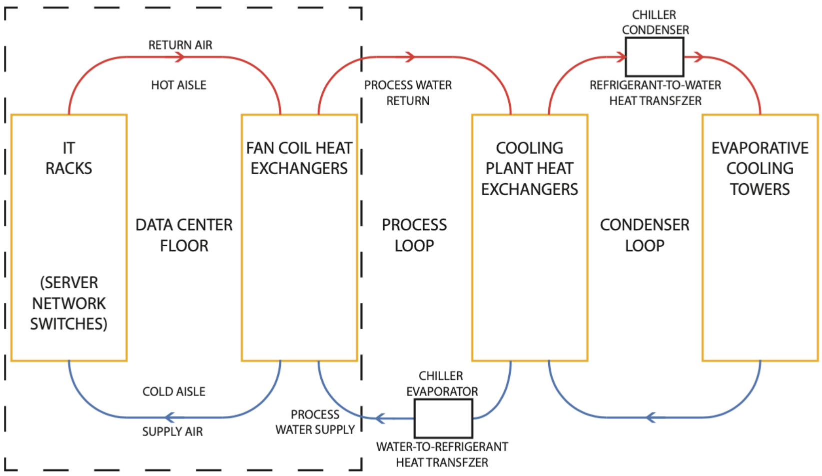

- ***Three loops***. It is used in large-scale data centres. The architecture can be viewed in the following figure.

  ✔ **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**.

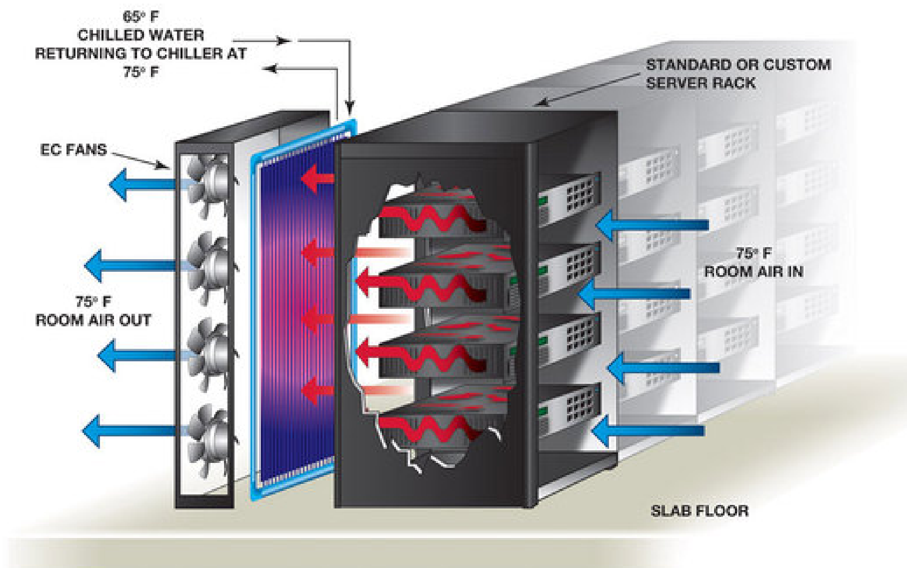



Figure 26: In-Row Cooling Mechanism (source: Energy Start).

- **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).

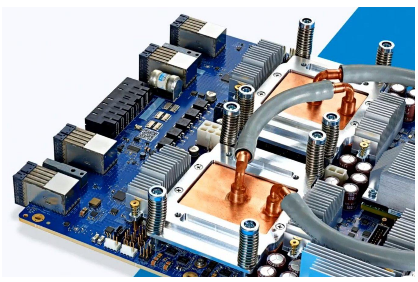


Figure 27: Liquid Cooling Mechanism.

### 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**:

$$\texttt{PUE} = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}} \tag{5}$$

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

$$\texttt{TFP} = \text{covers IT systems } + \text{ other equipment} \tag{6}$$

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:

$$\texttt{DCiE} = \texttt{PUE}^{-1} = \frac{\text{IT Equipment Power}}{\text{Total Facility Power}} \tag{7}$$

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

Table 4: 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 75) and System VM (page 76).

**❓ 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 78). 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. [8]

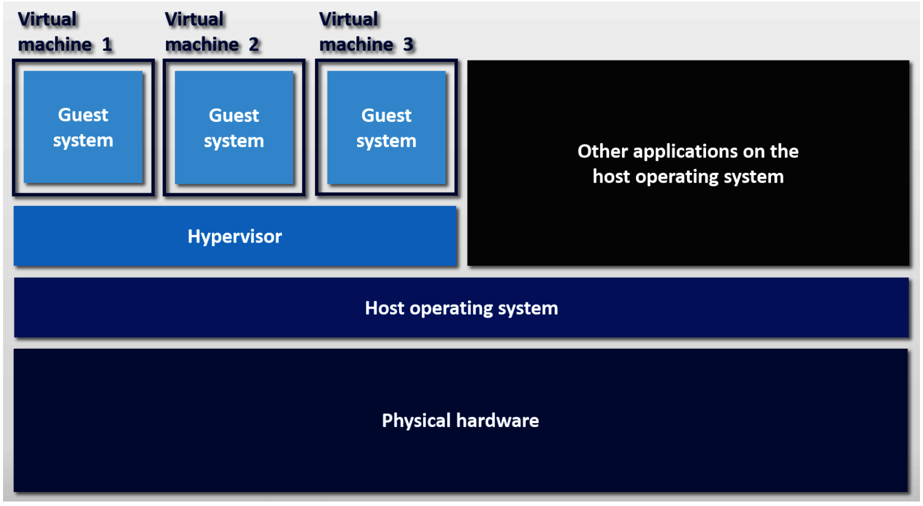



Figure 28: Operating system view if there are VMs in the physical machine (source: ionos).

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

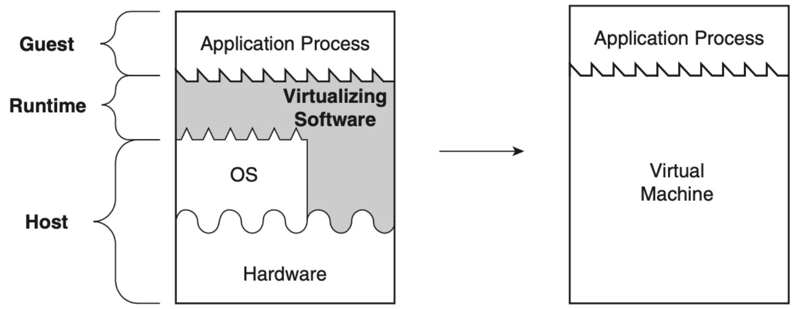



Figure 29: Process VM.

[11] 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)**.

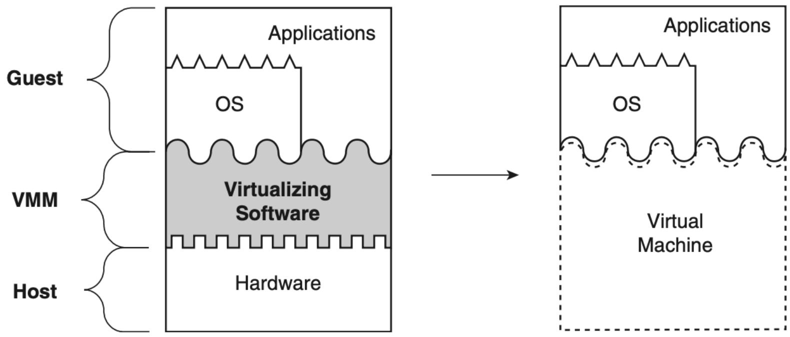



Figure 30: System VM.

### 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**.

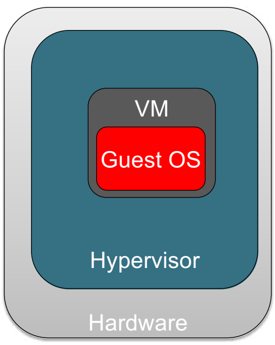



Figure 31: View of the Bare Metal Hypervisor.

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**.

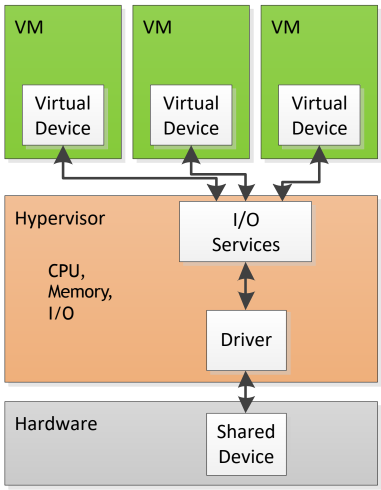



Figure 32: Monolithic architecture.

– 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.

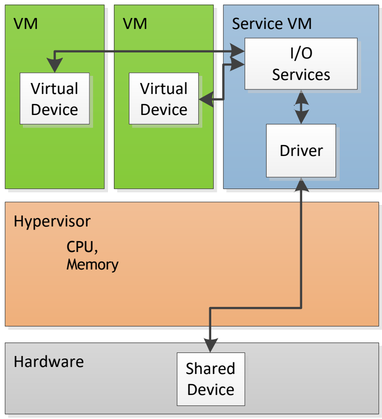



Figure 33: Microkernel architecture.

- **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**.

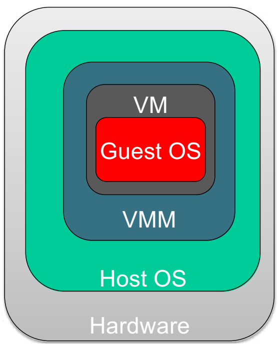



Figure 34: View of the Hosted Hypervisor.

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**.

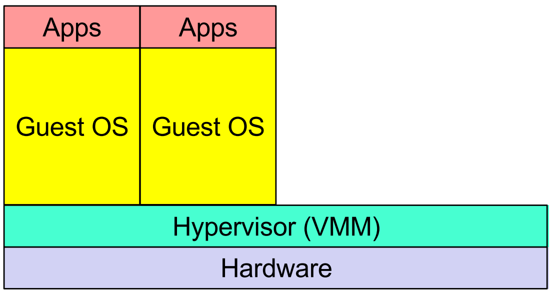



Figure 35: View of the Full Virtualization.

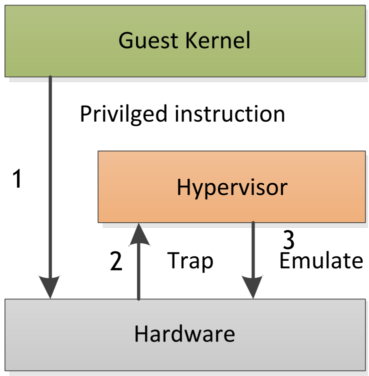



Figure 36: Full Virtualization flow.

---

#### 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).

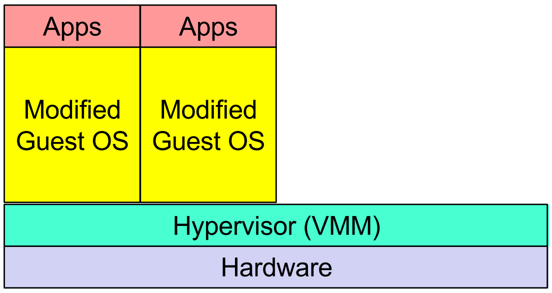



Figure 37: View of the Paravirtualization.

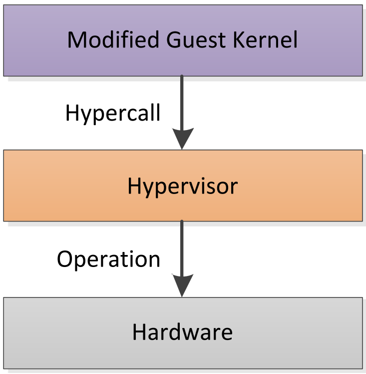



Figure 38: Paravirtualization flow.

#### 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**.

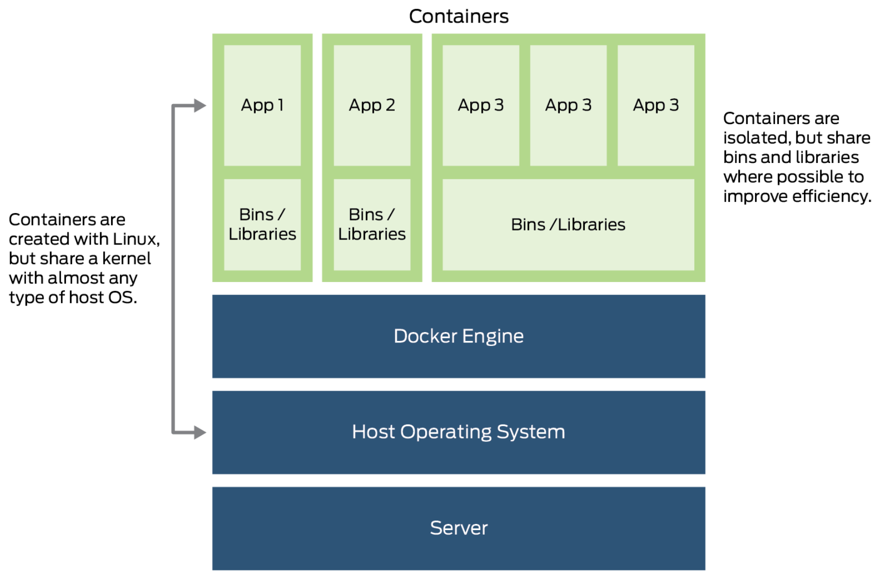



Figure 39: Containers architecture.

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 73).

---

#### 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

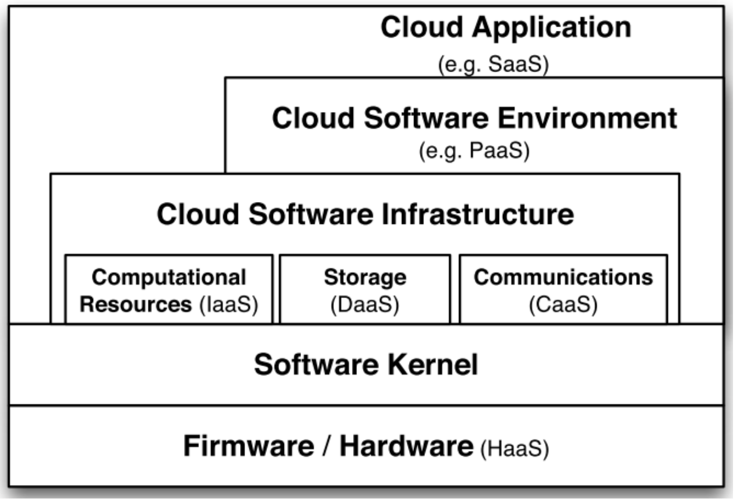
**Cloud Computing** is a **model for** providing convenient, **on-demand network access to a shared pool of configurable computing resources**, such as networks, servers, storage, applications and services. Anything that can be **quickly provisioned and released with minimal management** or interaction 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, planable, 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 91).

**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 $t$:

$$R(t) = P(\text{not failed during } [0,t]) \tag{8}$$

(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:

$$Q(t) = 1 - R(t) \tag{9}$$

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

$$\begin{aligned} R(0) &= 1 \\ \lim_{t \to +\infty} R(t) &= 0 \\ f(x) &= -\frac{\mathrm{d}R(t)}{\mathrm{d}t} \end{aligned} \tag{10}$$

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:

$$\text{Availability} = \frac{\text{Uptime}}{(\text{Uptime} + \text{Downtime})} \tag{11}$$

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:

$$A(t) = P(\text{not failed at time } t) \tag{12}$$

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

$$\text{unavailability } = 1 - A(t) \tag{13}$$

### ❓ 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:

$$A(t) = R(t) \tag{14}$$

- In general, the **reparable systems** have

$$A(t) \geq R(t) \tag{15}$$

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

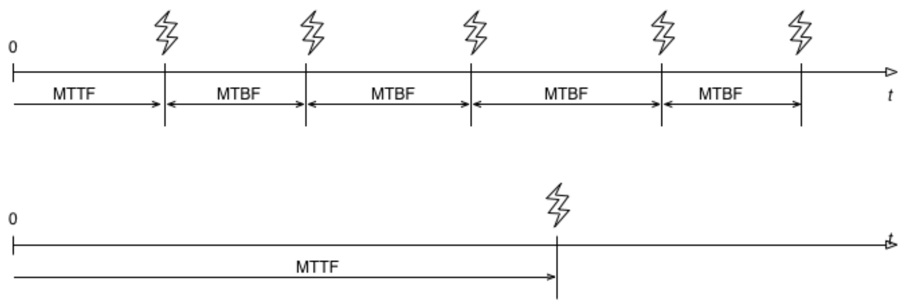
Some metrics exist for reliability and availability.

- 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. 8, page 94):

$$\texttt{MTTF} = \int_0^{\infty} R(t)\, \mathrm{d}t \tag{16}$$

- 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.

$$\texttt{MTBF} = \frac{\text{total operating time}}{\text{number of failures}} \tag{17}$$




- 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:

$$\texttt{MTBF} = \frac{10^9}{\texttt{FIT}} \tag{18}$$

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

$$\text{Failure Rate } \lambda = \frac{\text{number of failures}}{\text{total operating time}} \tag{19}$$

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

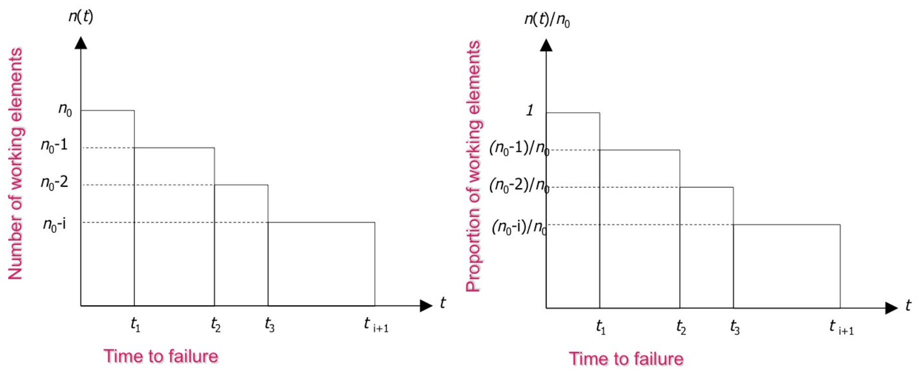
$$\texttt{MTBF} = \frac{1}{\lambda} \tag{20}$$

### ❓ 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:

$$\frac{n(t)}{n_0} \tag{21}$$

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

$$\frac{n(t)}{n_0} \to R(t) \tag{22}$$

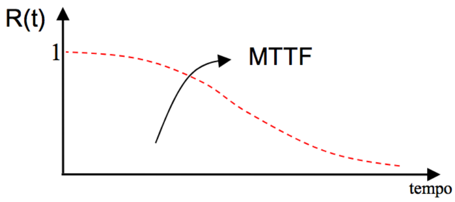
### ❓ Ok, but what do we do with the reliability probability?

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

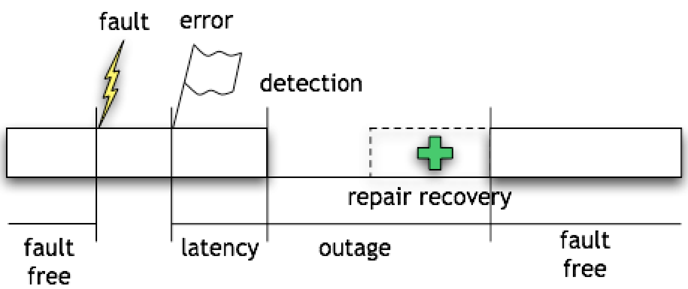
The **Constant Failure rate of the reliability** is:

$$\begin{aligned} R\,(t) &= e^{-\lambda\, t} \\ \mathtt{MTTF} &= \int_0^{\infty} R\,(t)\ \mathrm{d}t = \frac{1}{\lambda} \end{aligned} \tag{23}$$




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).

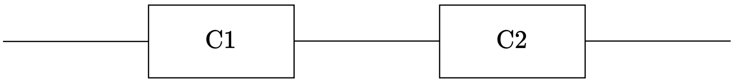
### 4.1.3 Reliability Block Diagrams

The **Reliability Block Diagram (RBD)**[12] 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**.

$$R_s(t) = \prod_{i=1}^{n} R_i(t) \tag{24}$$




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:

$$R_s(t) = e^{-\lambda_s t} \tag{25}$$

Where $t$ is the time and $\lambda_s$ is the **Failure in time**:

$$\lambda_s = \sum_{i=1}^{n} \lambda_i \tag{26}$$

Note that the $\lambda_i$ value is explained on page 96 (eq. 19). The Mean Time To Failure of a system is $S$:

$$\mathtt{MTTF}_s = \frac{1}{\lambda_s} = \frac{1}{\displaystyle\sum_{i=1}^{n} \lambda_i} = \frac{1}{\displaystyle\sum_{i=1}^{n} \frac{1}{\mathtt{MTTF}_i}} \tag{27}$$

If **all components are identical**:

$$R_s(t) = e^{-n\lambda t} = \exp\left(-\frac{nt}{\mathtt{MTTF}_1}\right) \tag{28}$$

$$\mathtt{MTTF}_s = \frac{\mathtt{MTTF}_1}{n} \tag{29}$$

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

Finally, the **availability** is:

$$A_s = \prod_{i=1}^{n} \frac{\texttt{MTTF}_i}{\texttt{MTTF}_i + \texttt{MTTR}_i} \tag{30}$$

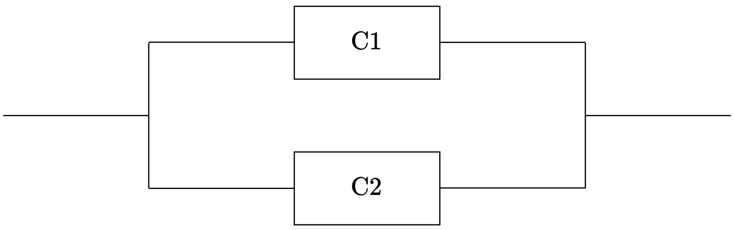
Where `MTTR` is the **Mean Time To Repair (MTTR)**.
If **all components are identical**:

$$A_s(t) = A_1(t)^n \tag{31}$$

$$A = \left(\frac{\texttt{MTTF}_1}{\texttt{MTTF}_1 + \texttt{MTTR}_1}\right)^n \tag{32}$$

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

$$R_s(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{33}$$




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:

$$R_p(t) = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{34}$$

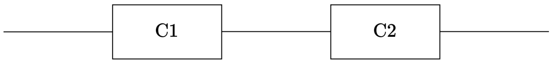
And the **availability** is:

$$\begin{aligned} A_p(t) &= 1 - \prod_{i=1}^{n} (1 - A_i(t)) \\ &= 1 - \prod_{i=1}^{n} \frac{\texttt{MTTR}_i}{\texttt{MTTF}_i + \texttt{MTTR}_i} \end{aligned} \tag{35}$$

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

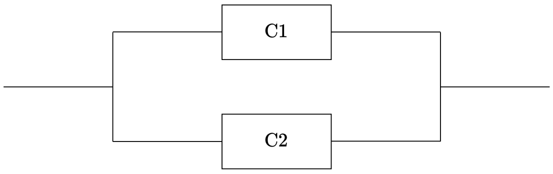
- **Series**.




Reliability:

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

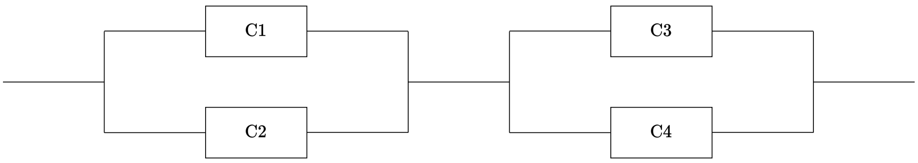
- **Parallel**.




Reliability:

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

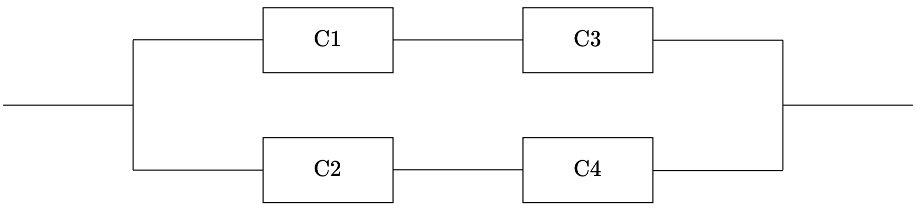
- **Series-Parallel** (component redundancy).




Reliability:

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

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



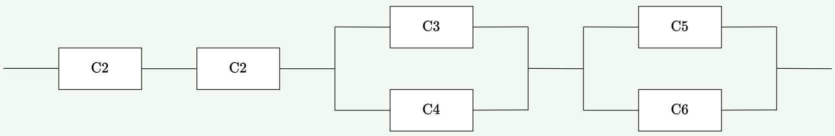

Reliability:

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

**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:

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

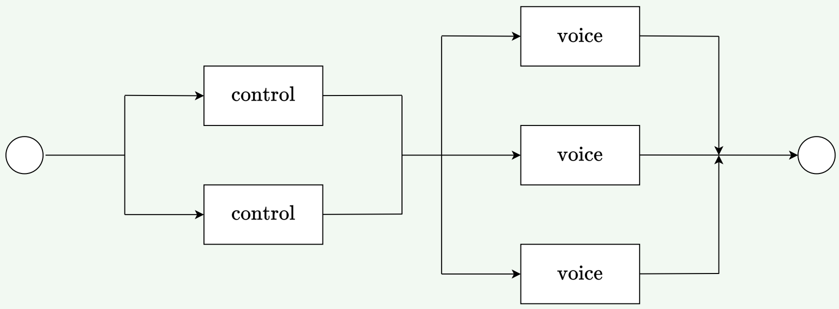
4. Finally, we calculate the reliability of the system by multiplying each calculated component reliability:

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

**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. [5]

**System failure occurs when** the $(n - r + 1)$-th component failure occurs. [5]

But note an interesting observation: [5]

- 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:

$$R_s\left(t\right) = RV \sum_{i=r}^{n} R_c^i \left(1 - R_c\right)^{n-i} \frac{n!}{i!\left(n-i\right)!} \tag{36}$$

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

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

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

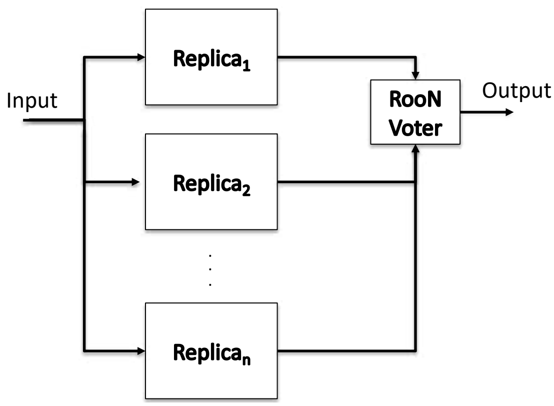



Figure 40: General structure of RooN system.

#### 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:

$$R_{TMR} = R_v \left(3 \cdot R_m^2 - 2 \cdot R_m^3\right) \tag{37}$$

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

$$\texttt{MTTF}_{TMR} = \frac{5}{6} \cdot MTTF_{\text{simplex}} \tag{38}$$

**❓ TMR: good or bad?**

TMR systems can **tolerate** both **transient**[13] **and permanent faults**[14]. It also has **higher reliability** (for shorter missions).

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

$$R_{TMR}(t) = R_c(t) \Longrightarrow 3e^{-2\lambda_m t} - 2e^{-3\lambda_m t} = e^{-\lambda_m t} \tag{39}$$

The time $t$ is:

$$t = \frac{\ln(2)}{\lambda_m} \approx 0.7\,\texttt{MMTF}_c \tag{40}$$

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

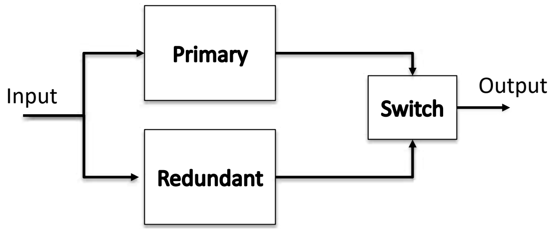
[13]In electrical engineering, a **transient fault** is defined as an error condition that vanishes after the power is disconnected and restored.

[14]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 5: Standby redundancy - Quick Formulas.

In the previous table we have:

- $R_s$: System Reliability
- $\lambda$: Failure Rate
- $t$: Operating Time
- $R_{\text{switch}}$: 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 28).

- **Full Rotation Delay** $R$ is:

$$R = \frac{1}{\texttt{DiskRPM}} \tag{41}$$

  And in seconds:

$$R_{\text{sec}} = 60 \times R \tag{42}$$

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

$$T_{\text{rotation AVG}} = \frac{R_{\text{sec}}}{2} \tag{43}$$

- **Seek Time**, the **time to move the head to a different track**, which is divided into several phases:
  - Acceleration
  - Coasting (constant speed)
  - Deceleration
  - Settling

  The $T_{\text{seek}}$ modelling considers a linear dependency with the distance. Also, the **seek average** is:

$$T_{\text{seek AVG}} = \frac{T_{\text{seek MAX}}}{3} \tag{44}$$

- **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**:

$$T_{\text{transfer}} = \frac{\text{R/W of a sector}}{\text{Data transfer rate}} \tag{45}$$

  The **Controller Overhead** is the **buffer management** (data transfer) and **interrupt sending time**.

  Transfer time and Controller Overhead are together because they are required to calculate some interesting metrics.

  - **Service Time** $T_{\text{I/O}}$

$$T_{\text{I/O}} = T_{\text{seek}} + T_{\text{rotation}} + T_{\text{transfer}} + T_{\text{overhead}} \tag{46}$$

  - **Response Time**

$$T_{\text{queue}} + T_{\text{I/O}} \tag{47}$$

    Where $T_{\text{queue}}$ depends on queue-length, resource utilization, mean and variance of disk service time and request arrival distribution.

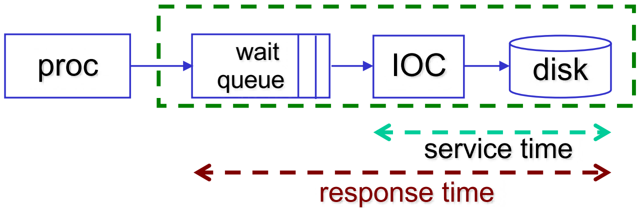



Figure 41: Service and response time.

**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

To calculate the *service time* $T_{\text{I/O}}$, we need the following information:

- $T_{\text{seek}}$, which we already have, and it is 6 ms.
- $T_{\text{rotation}}$
- $T_{\text{transfer}}$
- $T_{\text{overhead}}$, 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 previous service time is supposed to be only for very **pessimistic cases** where **sectors are fragmented on the disk in the worst possible way**. This can happen because the files are tiny (each file is contained in one block) or the disk is externally fragmented.

Thus, each access to a sector requires to pay rotational latency and seek time. This is not the case in many circumstances because the files are larger than one block and stored contiguously.

We can measure the **Data Locality** $DL$ of a disk as the **percentage of blocks that do not need seek or rotational latency to be found**.

Thanks to the Data Locality, it is possible to calculate the **Average Service Time**:

$$T_{\text{I/O AVG}} = (1 - DL) \cdot (T_{\text{seek}} + T_{\text{rotation}}) + T_{\text{transfer}} + T_{\text{controller}} \tag{48}$$

**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 109 to understand the values of $T_{\text{seek}}$, $T_{\text{rotation}}$, $T_{\text{transfer}}$ and $T_{\text{overhead}}$:

- $T_{\text{seek}} = 6$
- $T_{\text{rotation}} = 3$
- $T_{\text{transfer}} = 0.01$
- $T_{\text{overhead}} = 0.2$

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 38).

- 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** $\mathtt{MTTF}_{\mathbf{diskArray}}$ is equal to the relationship between the `MTTF` of a single disk and the number of disks:

$$\mathtt{MTTF}_{\text{diskArray}} = \frac{\mathtt{MTTF}_{\text{singleDisk}}}{\#\,\mathtt{Disks}} \tag{49}$$

  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**!

$$\mathtt{MTTF}_{\mathtt{RAID\ 0}} = \mathtt{MTTF}_{\text{diskArray}} = \frac{\mathtt{MTTF}_{\text{singleDisk}}}{\#\,\mathtt{Disks}} \tag{50}$$

- 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** $\mathtt{MTTF}_{\mathtt{RAID}}$ (except the level zero!) is:

$$\mathtt{MTTF}_{\mathtt{RAID}} = \left(\frac{\mathtt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{additionalCriticalFailuresInMTTR}}}\right) \tag{51}$$

  Where:

  - $\frac{\mathtt{MTTF}_{\text{singleDisk}}}{N}$ 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:

  - <u>`RAID 1`</u> - 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 `RAID 1`** $\mathtt{MTTF}_{\mathtt{RAID\ 1}}$ is:

$$\mathtt{MTTF}_{\text{RAID 1}} = \left(\frac{\mathtt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \tag{52}$$

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (53)$$

Where:

* $\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}$ 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` $\texttt{MTTF}_{\textbf{RAID 0 + 1}}$** is:

$$\texttt{MTTF}_{\text{RAID 01}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \quad (54)$$

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

$$\text{Probability}_{\text{2ndCriticalFailureInMTTR}} = \left(\frac{G}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (55)$$

Where:

* $G$ is the **number of disks in a stripe group**.
* $\frac{G}{\texttt{MTTF}_{\text{singleDisk}}}$ 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` $\texttt{MTTF}_{\textbf{RAID 1 + 0}}$** is:

$$\texttt{MTTF}_{\text{RAID 10}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndCriticalFailureInMTTR}}}\right) \quad (56)$$

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

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

Where:

* $\frac{1}{\texttt{MTTF}_{\text{singleDisk}}}$ 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` $\texttt{MTTF}_{\textbf{RAID 4}}$** and the **MTTF of `RAID 5` $\texttt{MTTF}_{\textbf{RAID 5}}$** is:

$$\texttt{MTTF}_{\text{RAID 4}} = \texttt{MTTF}_{\text{RAID 5}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndFailureInMTTR}}}\right) \quad (58)$$

And the probability is:

$$\text{Probability}_{\text{2ndFailureInMTTR}} = \left(\frac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \texttt{MTTR} \quad (59)$$

Where:

  * $\dfrac{(N-1)}{\texttt{MTTF}_{\text{singleDisk}}}$ 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 $\texttt{MTTF}_{\textbf{RAID 6}}$** is:

$$\texttt{MTTF}_{\text{RAID 6}} = \left(\frac{\texttt{MTTF}_{\text{singleDisk}}}{N}\right) \times \left(\frac{1}{\text{Probability}_{\text{2ndAnd3rdFailureInMTTR}}}\right) \tag{60}$$

And the probability is:

$$\text{Probability}_{\text{2ndAnd3rdFailureInMTTR}} = \text{Probability}_{\text{2ndFailure}} \times \text{Probability}_{\text{3ndFailure}} \tag{61}$$

Where:

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

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

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

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

$$\text{Probability}_{\text{3ndFailure}} = \left(\frac{(N-2)}{\texttt{MTTF}_{\text{singleDisk}}}\right) \times \frac{\texttt{MTTR}}{2} \tag{63}$$

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

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

Table 6: 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.

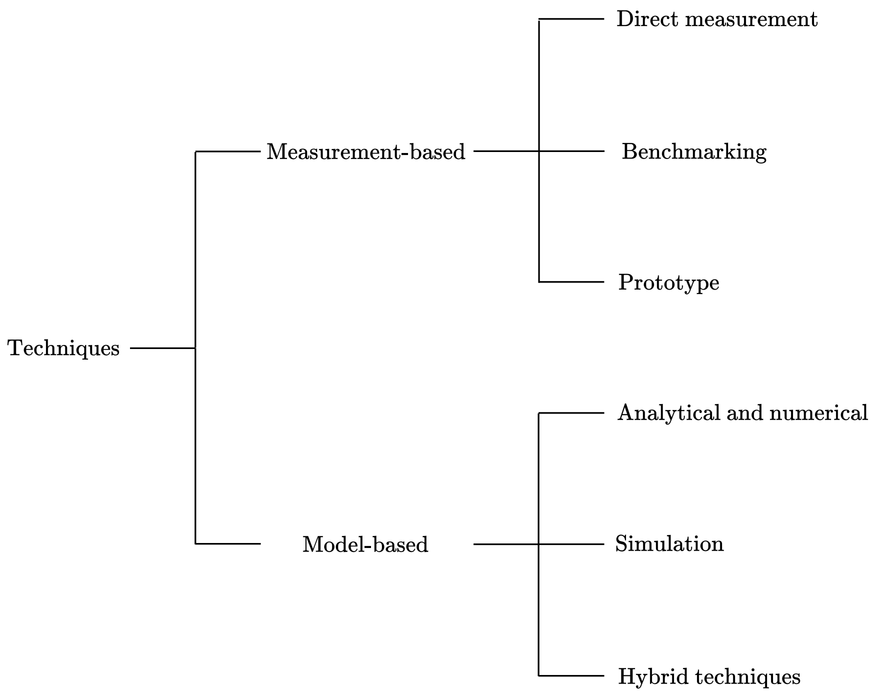



Figure 42: Quality Evaluation techniques.

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.

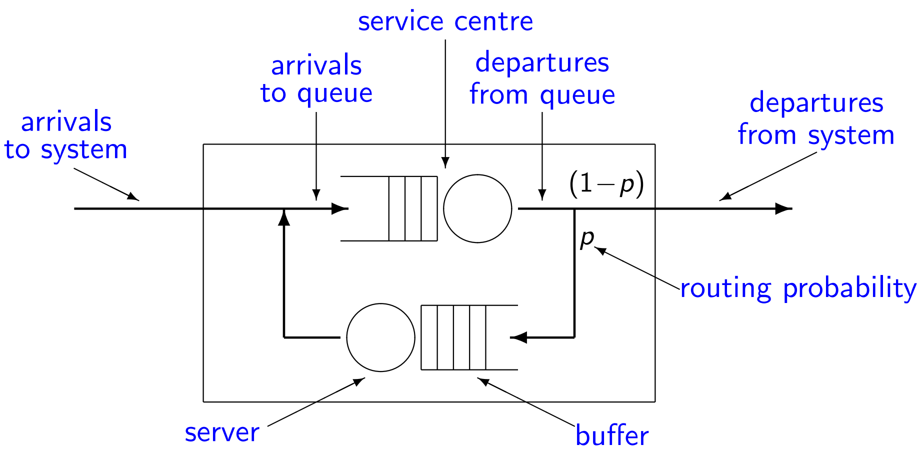


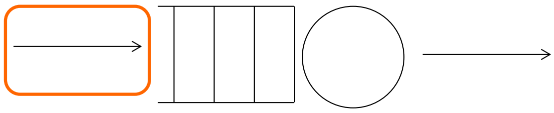

Figure 43: Queueing Network graphical representation.

#### 4.3.2.2 Characteristics

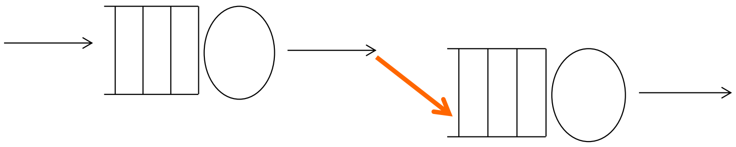
Queueing models are characterized by several aspects:

- **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**:

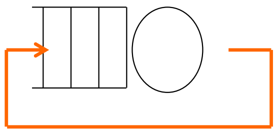
  1. An external source.

  2. Another queue.

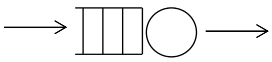
     

  3. The same queue, through a loopback arc.

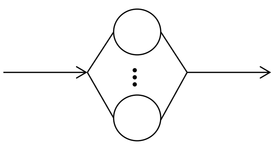
     

- **Service**. Service **represents the time a job spends being served**. The server does the job, but the number of servers can be different:

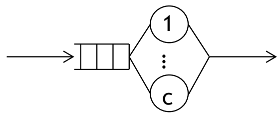
  - **Single server**. It has the ability to serve **one client at a time**. **Waiting customers remain in the buffer** until they are selected for service. Finally, the **next customer is selected depending on the service discipline**.

  - **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:

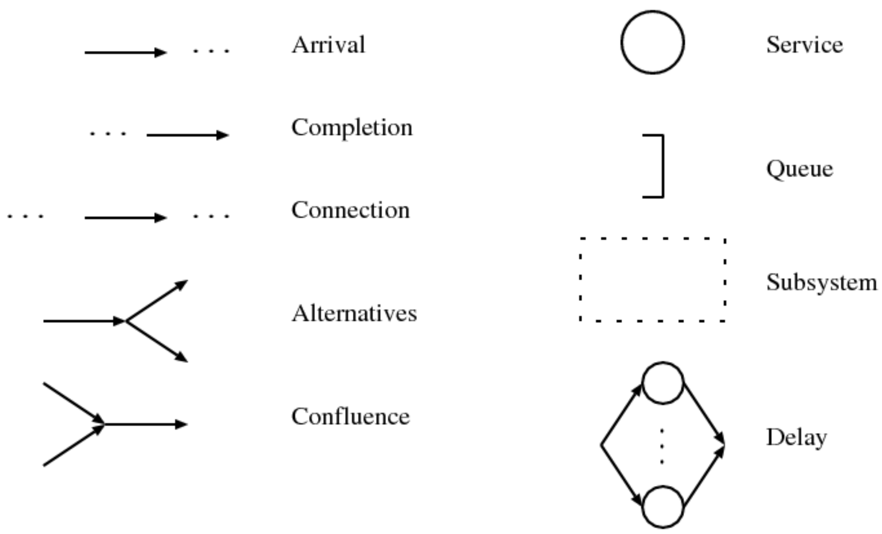
  - **Probabilistic Routing Algorithm**. **Each path is assigned a probability of being chosen by the job that left the station in question**.
  - **Round Robin Routing Algorithm**. The **destination chosen by the job rotates among all possible existing destinations**.

- **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.

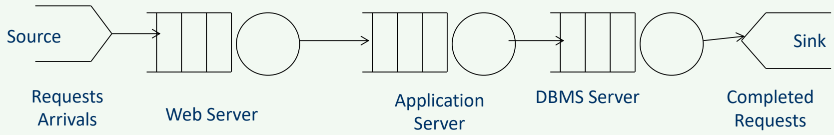
An additional graphical notation is:

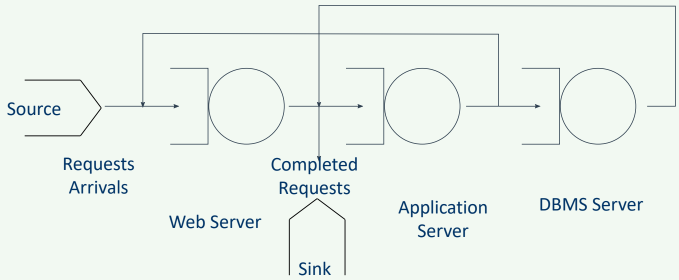



**Example 5: Open Networks**

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**.

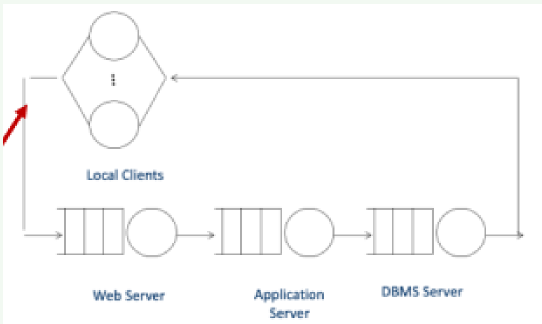


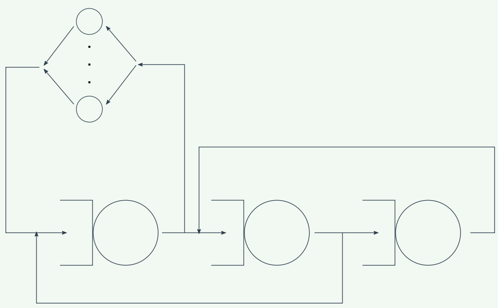

**Example 6: Closed Networks**

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 119).

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** ($B \le T$)
- **N**, the average **number of jobs** in the system

From these values, we can derive the following four important **quantities**:

- **Arrival rate**:

$$\lambda = \frac{\text{number of request arrivals}}{\text{length of time we observe the system}} = \frac{A}{T} \tag{64}$$

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

$$X = \frac{\text{number of request completions}}{\text{length of time we observe the system}} = \frac{C}{T} \tag{65}$$

- **Utilization**:

$$U = \frac{\text{total amount of time during which the system is busy}}{\text{length of time we observe the system}} = \frac{B}{T} \tag{66}$$

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

$$S = \frac{\text{total amount of time during which the system is busy}}{\text{number of request completions}} = \frac{B}{C} \quad (67)$$

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**:

$$\lambda_k = \frac{A_k}{T} \quad (68)$$

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

$$X_k = \frac{C_k}{T} \quad (69)$$

- **Utilization**:

$$U_k = \frac{B_k}{T} \quad (70)$$

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

$$S_k = \frac{B_k}{C_k} \quad (71)$$

#### 4.3.3.2 Utilization Law

Using the formulas:

- Throughput: $X_k = \dfrac{C_k}{T}$
- Mean service time: $S_k = \dfrac{B_k}{C_k}$
- Utilization: $U_k = \dfrac{B_k}{T}$

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**:

$$U_k = X_k \cdot S_k \tag{72}$$

---

#### 4.3.3.3 Little's Law

The **Little's Law** is:

$$N = X \cdot R \tag{73}$$

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.

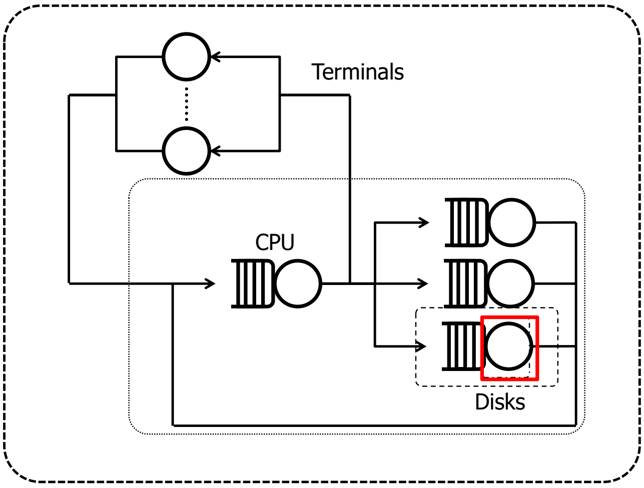
If the system throughput is $X$ requests/sec, and each request remains in the system on average for $R$ seconds, then for each unit of time, we can observe on average $XR$ requests in the system.

**Example 7**

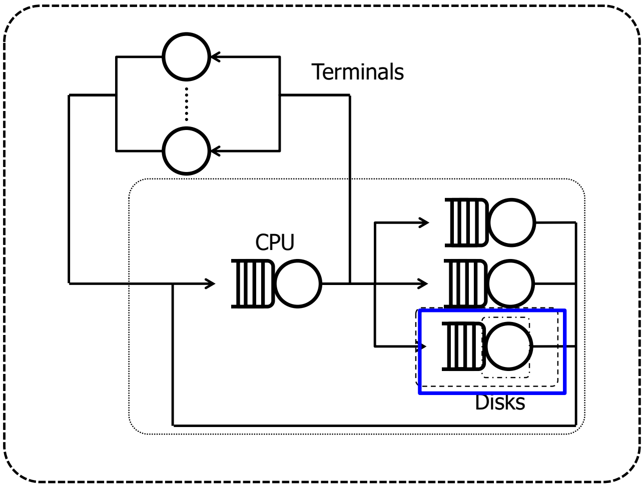
Consider a disk that serves 40 requests/seconds ($X = 40$ req/s) and suppose that on average there are 4 requests ($N = 4$) present in the disk system (waiting to be served or in service).
Little's Law tell that $N = XR \Rightarrow R = \frac{N}{X}$, so the average time spent at the disk by a request 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:

- **<u>Service level</u>**.
  - The service time is:
    $$R = S$$
    Where $R$ is the average of each request remaining in the system.
  - The utilization is:
    $$N = X \cdot R = X \cdot S = U$$



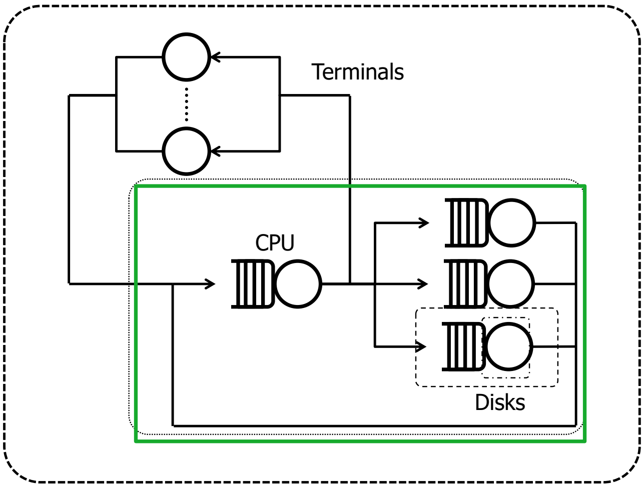

- **Service and Queue level**.
  - The service time is:
  $$R = S$$
  Where $R$ is the average of each request remaining in the system.
  - The utilization is:
  $$N = \text{Flying requests} \Rightarrow \frac{N}{X} = R = (S + T_{\text{queue}})$$



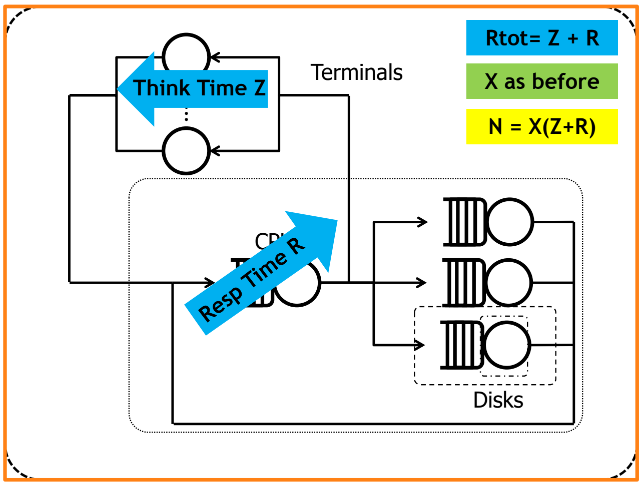

- **Subsystem level**.
  - The service time is:
  $$R = \text{Residence Time}$$
  Residence time corresponds to our conventional notion of response time: the period of time from when a user submits a request until that user's response is returned.

– 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:

$$R = \frac{N}{X} - Z \tag{74}$$

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**:

$$V_k = \frac{C_k}{C} \tag{75}$$

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 $\underline{C_k > C}$, resource $k$ is **visited several times** (on average) during each system level request. This happens when there are **loops in the model**.
- If $\underline{C_k < C}$, resource $k$ **might not be visited** during each system level request. This can happen if there are **alternatives** (e.g. caching of disks).
- If $\underline{C_k = C}$, 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.

$$X_k = V_k \cdot X \tag{76}$$

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 $D_k$**:

$$D_k = S_k \cdot V_k \tag{77}$$

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

$$U_k = X_k \cdot S_k = (X \cdot V_k) \cdot S_k = D_k \cdot X \tag{78}$$

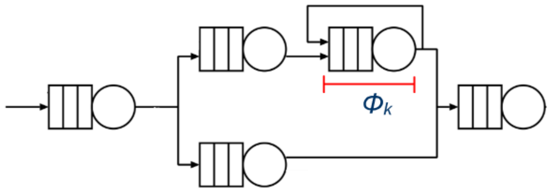
The utilization of a resource is denoted $U_k$ and it is the percentage of time that the $k$-th resource is in use processing a job. It is also equal to the product of:

- 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.

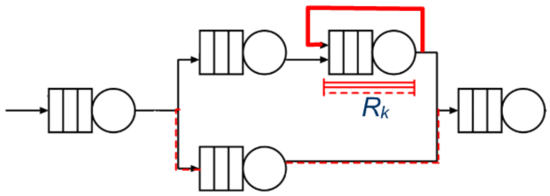
#### 4.3.3.8 Response and Residence Times

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 $\Phi_k$) 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**:

$$\begin{aligned} D_k &= v_k \cdot S_k \\ R_k &= v_k \cdot \tilde{R}_k \end{aligned} \tag{79}$$

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**[15].
- 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:

$$D = \sum_{k} D_k \tag{80}$$

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

[15] **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 $\lambda$ (arrival rate, page 124) 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:

$$\lambda_{\text{sat}} = \frac{1}{D_{\text{max}}} \tag{81}$$

The $R$ (*system response time*) bound is equal to the largest and smallest possible system response time experienced at a given arrival rate ($\lambda$), 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 $D = \sum_k D_k$.

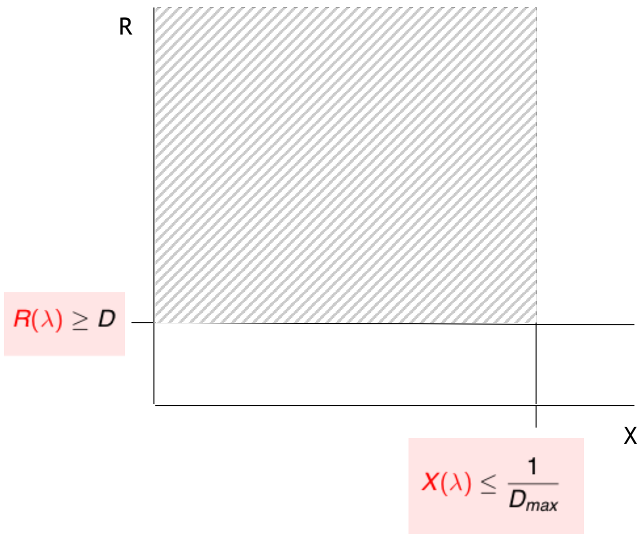
- If $n$ **customers arrive together** every $\frac{n}{\lambda}$ time units, there is **no pessimistic bound** on $R$ (system response time).

  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** $\lambda$ **might be**.

The bounding analysis for open models gives the following formulas:

- Bound for $X(\lambda)$:
$$X(\lambda) \leq \frac{1}{D_{\max}} \tag{82}$$
- Bound for $R(\lambda)$:
$$R(\lambda) \geq D_{\max} \tag{83}$$



## Closed Models

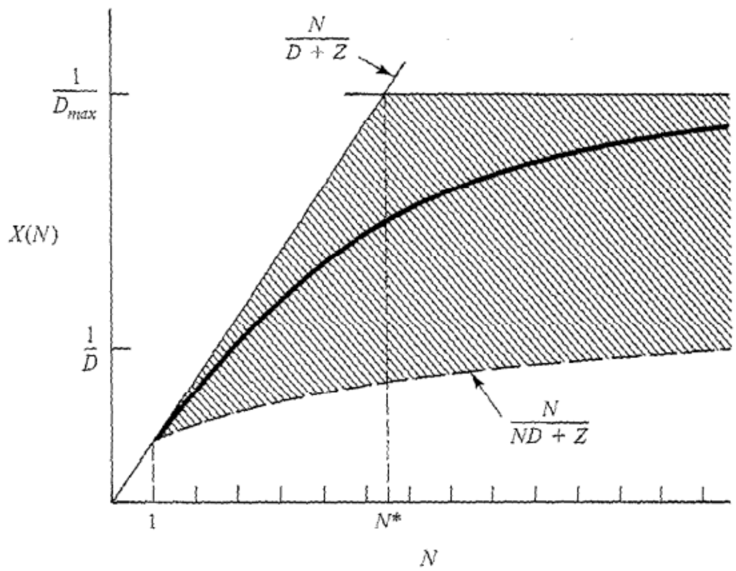
Bounding Analysis depends on the situation:

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

$$\begin{aligned} N &= X \cdot (R+Z) \\ 1 &= X \cdot (D+Z) \\ X &= \frac{1}{(D+Z)} \end{aligned} \tag{84}$$

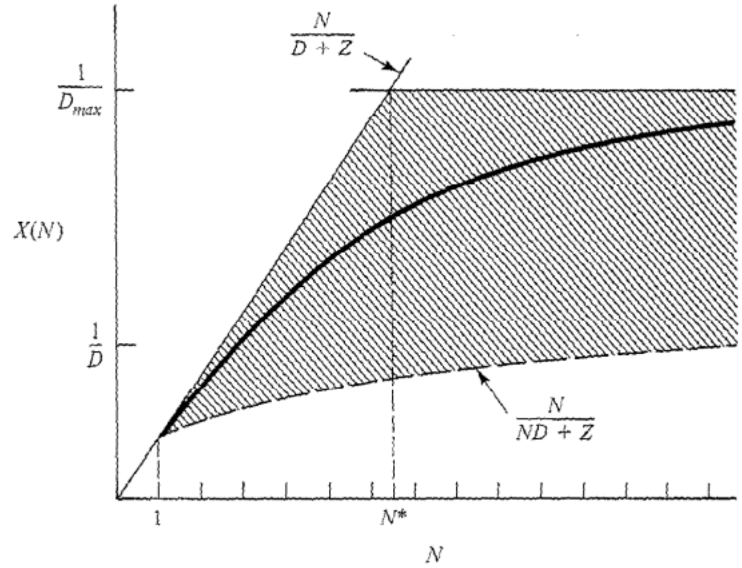
    - **Adding customers**: smallest $X$ (*system throughput*) obtained with largest $R$ (*system response time*), i.e., new jobs queue behind others already in the system.
      Remember: in closed models, the highest possible system response time occurs when each job, at each station, found all the other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

$$\begin{aligned} X &= \frac{N}{(N \cdot D + Z)} \\ \lim_{N\to\infty} \frac{N}{(N \cdot D + Z)} &= \frac{1}{D} \end{aligned} \tag{85}$$



- **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:

$$X = \frac{N}{(D + Z)} \tag{86}$$




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

$$U_k(N) = X(N)\, D_k \leq 1 \tag{87}$$

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

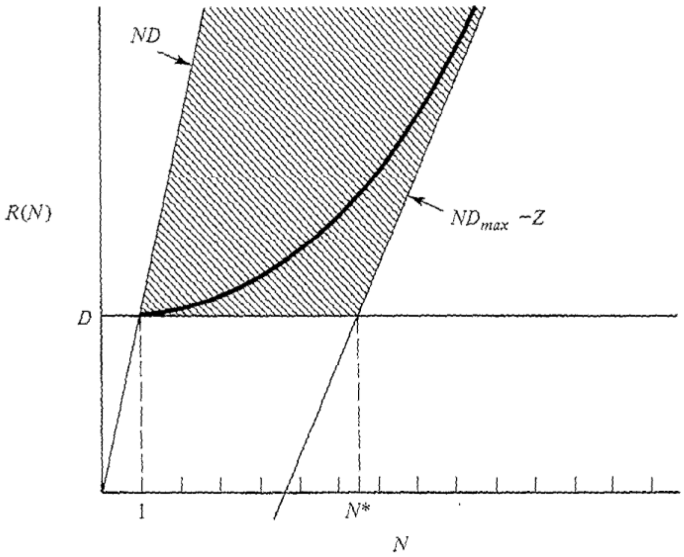
$$X(N) \leq \frac{1}{D_{\max}} \tag{88}$$

The $X(N)$ bounds is:

$$\frac{N}{N \cdot D + Z} \leq X(N) \leq \min\left(\frac{1}{D_{\max}}, \frac{N}{D + Z}\right) \tag{89}$$

The $R(N)$ bounds is:

$$\max(D, N \cdot D_{\max} - Z) \leq R(N) \leq N \cdot D \tag{90}$$

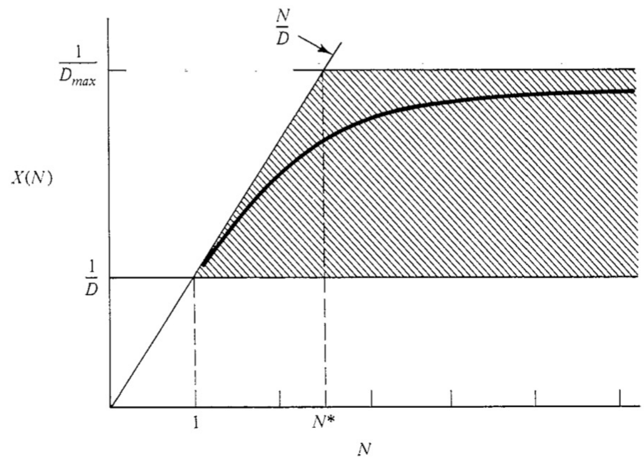


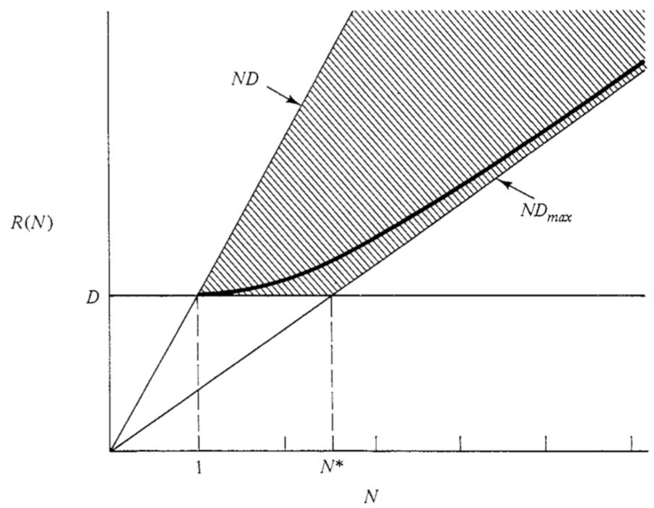

$R\,(N)$ bounds.

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

$$N^* = \frac{D+Z}{D_{\max}} \tag{91}$$

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] Jacopo Marino and Fulvio Risso. Is the computing continuum already here?, 2023.

[5] NIST. 8.1.8.4. R out of N model - itl.nist.gov. `https://www.itl.nist.gov/div898/handbook/apr/section1/apr184.htm`. [Accessed 14-08-2024].

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

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

[8] 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.

[9] 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

## D



## E



## F



## G



## H

## W



## X