Computing Infrastructures

Niccoló Didoni

February 2022

Contents

Part I

Introduction

Chapter 1

Data centres

With the evolution of computer science and information technology, companies try to reach as many people as possible with their services, thus we need infrastructures that are able to efficiently bear the load of extremely available applications. Data centres have been design to solve this problem. Before moving on with the description of data centres we should try to define them.

> **Definition 1** (Data centre)**.** *A data centre is a building or part of a building whose primary purpose is to house a computer room and its support area.*

Immediately we notice that a data centre is defined as a building or a portion of it. In fact, this definition includes both data centres of big companies that have all computers in a separate building and data centres of small companies in which servers might be in the same building as the offices. This definition also points out that we should care not only of the computer room, but also of the area where support systems are housed, hence we should put emphasis on support systems, too.

Definition 1 highlight the main characteristics of a data centre but it's rather general. A more precise definition could be

> **Definition 2** (Data centre)**.** *A data centre is a facility used to house computer systems and associated components, such as telecommunication and storage systems. It generally includes redundant power backup supplies, redundant communication connection, environmental control systems and various security devices (both physical and virtual).*

This definition adds many details to the previous one, especially with respect to the components associated to the computer systems.

Both previous definitions focus on the technical part of data centres. The following definition focuses more on business and cost.

> **Definition 3** (Data centre)**.** *A data centre is a centralized repository, either physical or virtual, for the storage, management and dissemination of data and information organized around a particular body of knowledge or pertaining to a particular Business.*

As we have seen each definition is given from a different perspective, but they have two concepts in common

- **Computer systems** and their associated components.
- **Business** and cost.

1.1 A general definition

The aim of a company or of a person usually is to offer its services (e.g. an application) to the largest number of people all around the globe. This means that the system that implements the service has to **scale up**. Furthermore, one might also want to protect its system (i.e. its idea, its service) in one place so that it can be protected, maintained and controlled. To achieve this goal we need

- **Power**. The computers that make up the system require power to work, thus we have to ensure to have enough power to make the service available.
- **Cooling systems**. Processor and computer components heat up when working, especially under heavy work loads, thus we should ensure that the system is kept at an optimal temperature. Cooling is vital because if computers work at high temperature, they might break thus reducing the availability of the service.
- **Networking systems**. The service has to be used from all around the world, thus the system has to be connected to the internet with an appropriate networking system that allows to handle all the user's requests.
- **Data management systems**. An high number of users generates a massive amount of data that has to be handled efficiently.

The four components aforementioned and the scale up property define a **data centre**. Notice that from this definition a data centre should run **continuously** and **efficiently**.

Data centre components The components of a data centre can be divided in

- **IT equipment**.
- **Facility equipment**.
- **Server rooms and support areas**.

1.1.1 Enterprise data centres

A data centre build and maintained by a single company and used for the services of the company is called **enterprise data centre**. Having a data centre has many advantages for a company, in fact

- It represents an asset (it's a building after all).
- The company has complete control over the operational activities on the data-centre. This means that the owner can operate more efficiently.

It's not all good, the main drawback of having a proprietary data centre is that it requires a lot of effort (monetary and in terms of knowledge and expertise) to build. This means that we have a big entrance barrier for building enterprise data centre, thus a small company might not have the resources to build one. For this reason a data centre has to be considered on the business level, in fact it represents a big cost for a company.

1.1.2 Third-party data centres

Building a proprietary data centre isn't the only solution. A company can also decide to outsource the data centre, in particular we can distinguish between

- **Co-location data centres**. Co-location data centres are data centres shared among multiple companies to reduce the costs for each company.
- **Third-party data centres** or **Managed Hosting Platforms**. Third-party data centres are completely built and maintained by an external company.

These are the main ways in which a company can outsource the hardware (equipment, facility and related systems) of a data centre (i.e. we have full control over the software). These aren't however the only solutions available, in fact one might want to outsource the software, too. In this case the main solutions available are

- **Infrastructure as a Service**.
- **Platform as a Service**.
- **Software as a Service**.

In this case the hardware is shared among different companies.

Co-location data centres

Co-location data centres allow a company to partially outsourcing a data centre. In particular, a third company rents the building where the data centre is to other companies. This means that the building (and sometimes the equipment, but in general only the building) is rented to other companies that can install their machines and software. In other words the owner of the building and the company that uses the data centre are two different entities (i.e. companies). Basically, who is renting the facility ensures availability for what concerns the building and the associated services (e.g. cooling, power, security).

This type of outsourcing has many **advantages**,

- The company that needs the data centre doesn't have to care about the facility in which it is. This means that the company that rents the facility also cares about the power supply, the cooling system, the security and all other stuff related to the building in which the equipment is.
- The data centre is still quite customisable, in fact the third company only rents the building, thus the customer can choose its equipment.

Co-location data centres also have some **disadvantages**,

- The data centre as a building is not an asset anymore, in fact the facility is owned by a third party. This also means that the cost of building a data centre is initially reduced, in fact a company doesn't have to invest a lot of money in building the facility.
- The equipment is collocated in the same building with other companies, thus a buyer hasn't full access to the facility. In particular the staff can enter the data centre only in specific hours of the day and for a limited time. In other words, a company hasn't 24/7 access to its equipment.

- A company still has to pay for the equipment used for the data centre. Of course, this is in some cases an advantage, because it allows more customisation. In other cases, especially for companies that don't need custom equipment, it's an unnecessary cost.

Third-party data centres

Third-party data centre allow to completely outsource a data centre. In particular, the building and the equipment is completely managed by a third company and we only provide the software. Notice that there is a big difference between this type of outsourcing and the cloud computing or something-as-a-Service, in fact in this case a company is exclusively assigned some resources that are therefore not shared among different entities (like in cloud computing).

Third-party data centres have many **advantages**,

- The initial investment is very low, in fact a company doesn't need the capitals to build the facility and the expertise to design the data centre.

Completely outsourcing isn't all good, in fact

- Completely outsourcing has an high operational cost (getting payed for an operation is the business of the third party company that rents the data centre).
- Companies have a lower control of the operational activities because they are handled by third-party staff.
- Companies can't customise their hardware.
- Once a company enters a third-party data centre, it customises its business to the hardware that the company provides. This means that it's hard to change data centre because it means investing money to optimise the software and the business on a different hardware. This phenomenon is called **customer locking**.

Part II

Hardware infrastructures

Chapter 2

Definitions

2.1 Computing infrastructures

In this section we are going to analyse the main computing infrastructures and their characteristics. Before moving to the description of each type of infrastructure, let us introduce a general definition of computing infrastructure

> **Definition 4** (Computing infrastructure)**.** *A computing infrastructure is a technological infrastructure that provides hardware and software for computation to other systems and services.*

Notice that this definition focuses both on the hardware and the software needed by the computing infrastructure to work. It's also important to highlight that the application that is running on the computing infrastructure is not part of it (i.e. the application uses the infrastructure).

A general definition The definition of computing infrastructure is very general in includes multiple types of systems (not only data centres) with different size and computing power. In particular we can divide the infrastructures in

- **Data centres**. Data centres are the biggest and the fastest infrastructures.
- **Fog or Edge Computing Systems**. Fog (or Edge) computing systems lie in between data centres and personal computers.
- **Personal computers**.
- **Embedded computers**.
- **Internet Of Things systems**.

A visual representation of this types of system with respect to their computation speed and memory usage is shown in Figure 2.1.

Computing continuum Some time ago the market offered few types of devices (i.e. few shapes, sizes and speeds). Nowadays, there are many different solutions that span from very low power, memory and cost to high power and cost. These availability is called **computing continuum** because we can choose among many different devices, thus we can pick the one that fits better the needs of a project.

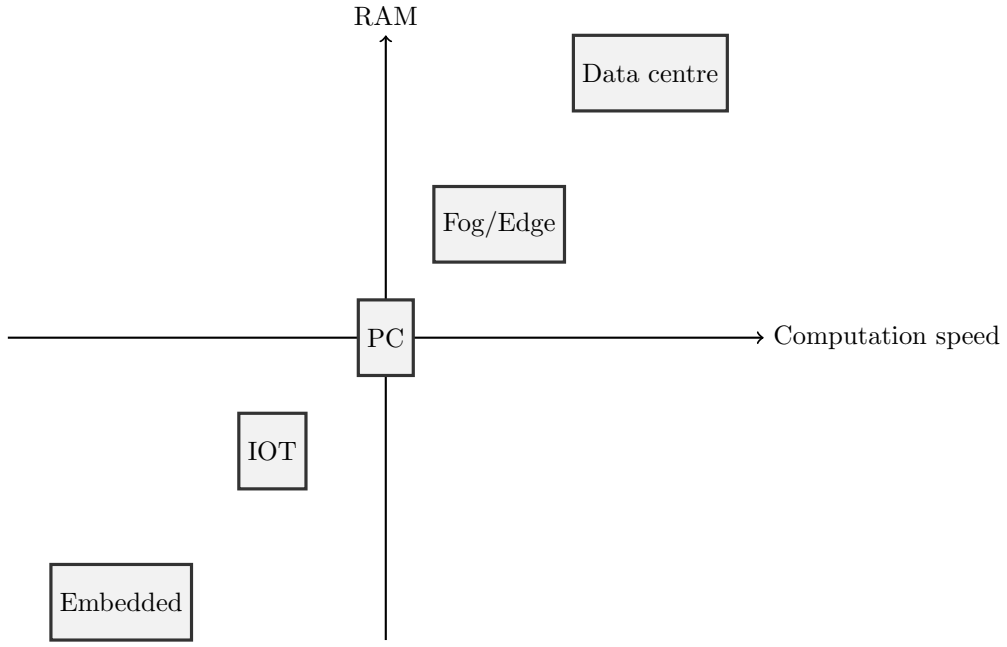


Figure 2.1: Different computing infrastructures plotted with respect to their speed and memory usage.

2.1.1 Data centres

Since we have already analysed data centres, we are only going to list the main advantages and disadvantages of this class of infrastructures.

Advantages The main advantages of data centres are

- Data centres have a **low IT cost**.
- Data centres ensure an **high performance**.
- The **software can be updated as soon as possible** because all machines are in a single building.
- Data centres have an **almost unlimited storage**.
- Data centres have **an high data reliability** because data is always redundant.
- Documents and files in a data centres can be **accessed from everywhere** because the building is connected to the network.
- Data centres provide **independence from the device on which an application runs on**, in fact the main computation is done in the data centre. Basically, clients only read the computation done by the data centre.

Disadvantages The main disadvantages of data centres are

- Data centres **require internet connection** because the equipment (i.e. the component on which the computation is done) is physically far from the entity that needs the computation. Notice that this is also true for the company that owns the data centre, in fact the servers have to be accessed to the network to allow fast and flexible usage.
- Companies have to **take care also of the physical security** of a data centre, because it's in a separate building (that might be far from the company's main facilities).
- Data centres have **high power consumption**. This factor increases the cost of maintaining a data centre.
- Data centres introduce **high latency** because a request has to go back and forth through the network. This happens because the actual computation is done by the data centre but the device that requested the service is in another location. This means that data centres aren't suited for time-critical operations; in these cases it's better to use a more local infrastructure (i.e. a device closer to the user). Consider for instance a car that has to swiftly take a decision (e.g. an action to avoid an incident). In this situation it's better to do the computation on-board (i.e. with an embedded device) because of the low delay. A data centre could be used to store the log of the actions performed by the car to avoid the incident because the data centre has a huge amount of memory and logging isn't a time critical operation.
- Data centres are **not always very customisable**, in fact they usually use low-end components that are quite standard.

2.1.2 Internet Of Things

Let's move on the other side of the computing infrastructures' spectrum. IOT devices are small boards like microcontrollers (e.g. Arduino), FPGAs and ASICs that are mainly used to collect data from sensors but can also do some small computations and store little data. Usually sensors are attached to the boards that can do some primitive analysis on the data (e.g. filtering). An important remark about IOT devices is that they have to be very power efficient because they usually run on batteries.

Advantages The main advantages of IOT devices are mainly related to their size and proximity to the data source. In particular

- IOT devices are **highly pervasive**. This means that they can be moved easily in a system.
- IOT devices can use **wireless communication** (usually the wireless module is integrated in the board). Wireless communication is a key point for these devices, in fact it's a very power hungry task and IOT devices have to reduce power consumption. To improve efficiency, IOT devices usually filter and reduce the sensors' data to send as few data as possible through the network. Usually wireless communication is also not wide-band to ensure minimal power consumption.

Disadvantages The disadvantages of IOT devices are related to the processing power and in general to power consumption. In particular,

- Computation is limited to **low frequencies**.

- **Memory and storage are limited**.
- **Communication is limited** to few hertz.
- IOT devices are difficult to program. This characteristic comes from the fact abstraction increases power consumption, thus it's more efficient to use a different library for each device and for each application.

2.2 Embedded boards

Let's continue our overview over computing infrastructure towards more powerful devices. Embedded boards are, in fact, more powerful edge devices (i.e. devices near the data and the sensors). As IOT devices, embedded devices can be connected to different sensors (or actuators) and can communicate wirelessly. On the other hand embedded devices offer more computation power and sometimes come equipped with accelerators like GPUs or machine learning accelerators. This power comes at a cost, in fact embedded boards usually have to be connect to a power source instead of batteries.

Advantages The advantages of embedded boards are related to the trade-off between the computation power and the reduced size of the devices. In particular,

- Embedded boards are usually **small** and comes equipped with no monitor nor keyboard (even if it's possible to connect some).
- Embedded boards can directly **sense data and do some computations**.
- Embedded boards are **simple to program**, in fact we can use well-known programming languages (e.g. Python, C or C++).

Disadvantages The disadvantages of embedded boards are related to power efficiency. In particular,

- Embedded boards are **less efficient** because of the higher computing capabilities.
- Embedded boards are in a sort of **intermediate computing continuum** between IOT devices and general purpose computers. In this sense a board can be characterised by the modules that come or that can be attached to the board.

2.2.1 Edge and fog computing

Edge and fog computing refer to computing infrastructures that sit between data centres and personal computers. Even if they represent the same power and memory consumption category, edge and fog computing are different concepts. The main difference is that

- **Edge computing** is closer to data and data generation. In other words edge computing focuses on the **intelligence of data**.
- **Fog computing** is closer to the last step before transferring the data on the cloud (i.e. in a data centre). In other words fog computing focuses on the **intelligence of network**.

Edge computing

Edge computing refers to a computing infrastructure that process the data from the edge of a system (e.g. sensors). Basically they are closer to the edge of the system with respect to data centres. It's important to understand that these systems aren't like embedded boards, in fact they are power hungry machines (e.g. servers) that can process a lot of data and do heavy computations (e.g. machine learning).

Fog computing

Fog computing refers to a computing infrastructure that process the data before sending it to the cloud. In other words, fog computing is the last step where we know that the data is handled locally.

Chapter 3

Data centres as a computer

3.1 Warehouse-scale computers

Warehouse-scale computers (WSC) change the classical idea of data centre. Usually, a data centre is shared among different companies or rented by a third party as a service, in any case each customer is assigned with some specified hardware that is not shared with other customers. Warehouse-scale computer offer a different perspective, in fact the data centre is seen as a single computer on which many companies are offered a single service. In other words a warehouse-scale computer is owned by a single entity that provides a service as it were provided by a single computer.

Warehouse-scale computers share some characteristics with data centres but have also some differences. In particular

- WSC are large scale infrastructures.
- WSC have to be connected to the internet to allow the customers to use the services offered.
- WSC offer a service to multiple customers, thus the hardware is in some sense shared among the customers. The customers doesn't actually realise this because the service is offered as it were run by a single computer. This is the main difference with data centres.

Advantages The main advantages warehouse-scale computers are

- WSC usually run a limited number of applications.
- WSC are very efficient because resources are shared, thus can be handled by the provider that can optimise their usage and allocation.

3.2 Connected data-centres

Data centres and warehouse scale computers can be connected among them to provide

- **Latency reduction**. Basically, if there are multiple data-centres (of a company) in the world a computation can be done closer to the client, thus reducing the time needed to send the request back and forth.
- **Parallelism**.

- **More computational power**.
- The possibility to adhere to different **laws and regulations**. In particular different regions of the world have different rules that regulates the data of the clients (how it has to be handled and who can access it). Having a data-centre in each region allows to adhere to the laws of that region.
- **Increased throughput**.

Now we are going to analyse with an increasing level of granularity the zones in which a data-centre can be placed.

3.2.1 Countries and areas

The most general level of geographic division is countries and areas. Some examples are the US, the European Union and China. The main characteristic of this division is the fact that the data-centres have to adhere to some laws (e.g. the GDPR) that regulate how data can be stored, where it can be stored, by whom it can be accessed and how it can be handled. When a client buys a service from a data centre (or WSC) or decides to rent some space in it (i.e. use a co-location data centre), he/she can select the country or geographical area where the centre is.

3.2.2 Regions

Geographical areas can further be divided into regions. Regions allow to reduce latency and cost. Some examples of region are Rome, Virginia, Frankfurt and so on. Notice that when a customer (i.e. the company that rents the data centre) selects a region, it doesn't pick a specific data centre but one among a set of data centres in the region. Furthermore, a region is the smallest deployment unit that a customer can choose (i.e. the customer can only choose in which region to put the equipment).

Latency Latency can be reduced selecting regions closer to the users, in fact if the client is close to the data centre, the round trip time of the client's request is small.

Cost Choosing the right region doesn't only reduce latency but also cost, in fact

- Power costs more in some regions than others.
- The land on which the facility has to be built costs more in some regions.
- The temperature of the environment is different, thus the cooling system may be effected.
- Takes are different from one region to another.
- Labour costs more in some regions.

Distribution of data centres A region can defined as a set of data centres that have a Round Trip Time (RTT) of less than 2 ms. Regions have to be very far away one to another (usually around 160 km), to allow regions to be independent. Independent regions are good for disaster recovery in fact usually when a disaster happens in a region, the other isn't affected, being so far away.

Synchronisation Since data centres are so far away, the data can't be synchronously replicated, thus it's not exactly the same at every time.

3.2.3 Availability zones

A region is made of different Availability Zones (AZs). AZs are areas in a region or part of a data centre that are completely independent with respect to power and cooling. For this reason AZs allow a customer to run critical operations because if the power system of an AZ fails, another AZ can do the same operation because its power supply is still working. In other words every AZ has a different and independent power and cooling system. Availability zones aren't enough for disaster recovery, in fact they are in the same region which, in case of a disaster, is usually completely compromised.

Data replication AZs aren't visible to customers (i.e. those who rent the data centre). A customer can however decide to have multiple AZs in a region. This allow him/her to have synchronous replication (remember that data centres in a region are within 2 ms RTT). In case of replication, a customer is assigned at least 3 AZs because we need quorum to decide which is the correct data when a problem occurs. Notice that a client chooses only to have replication, not which AZs to use.

3.2.4 Edge locations

Some companies use edge locations to do computations that are short enough without going directly to the data centre. In particular an edge location is a smaller data centre accessed before reaching the actual data centre to do some pre-computation. If the computation is small enough the request might even not be forwarded to the data centre.

3.3 Availability

Availability is a fundamental property of a data centre, in fact if a service on a DC isn't available, it makes lose money to the provider of the service (not only because the users can't access the service but also because the provider loses credibility). Usually the money loss isn't limited to the time in which the DC isn't available, either.

Target values A good value for availability is 99.99 %, in fact it means that a service is not online for less than an hour per year. Unfortunately not all services have such an availability, in fact we usually have

- $99,90\%$ on single instance VMs with premium storage for an easier lift and shift.
- $99,95\%$ VM uptime SLA for Availability Sets (AS) to protect for failures within a datacenter.
- $99,99\%$ VM uptime SLA through Availability Zones

Warehouse-scale computers Warehouse-scale computers have an high availability because they have to run a single service, thus if one component fails, the other components can do the same job without problem.

Chapter 4

Architectural overview of warehouse-scale computers

In this section we are going to analyse the components a warehouse-scale computer is made of. In particular we can divide the components needed in a warehouse-scale computer in

- **Servers** used for computation.
- **Networking systems** for communicating with the outside and with the servers inside the WSC.
- **Storage**.
- **Equipment** needed for cooling, power and failure recovery.

The first important thing to remember is that the hardware in a WSC is usually homogeneous (i.e. the same components are used all over the WSC). This choice

- Reduces costs because buying large quantities of a single component is more convenient (scale economy).
- Is better for computing the same service (remember that WSC run just one service) because we can replicate on multiple components the behaviour of the service on a single component.

4.1 Servers

The first macro-component we are going to analyse is the server. A server is the basic building block of a warehouse-scale computer. Servers are made of different components, in particular

- One or more **processing units** (CPU).
- **Storage devices** (usually volatile and non volatile).
- One or more **network interfaces**.
- A **case**, a **power supply** and a **cooling system**.

Servers can be of many shapes and can be organised in different structures. Usually servers have no I/O interfaces and are monitored from a control unit that checks if the server is working properly. Let us now introduce the main components a server is made of.

4.1.1 Motherboard

Every server is build around a motherboard: a PCB on which we can plug all the components needed for the server. Choosing a motherboard with enough slots to accommodate all current and future components is fundamental. As we have said, WSC's components are homogeneous and motherboards are no different in fact even in this case it's convenient to choose a single design and use it all over the WSC.

When choosing a motherboard we have to consider

- The number and type of processors that it can host.
- The number and type of RAM slots available (usually from 2 to 192 DIMM slots).
- The number of locally attached disks. Usually a motherboard can support
  - Drive bays.
  - Hard Disk Drives (HDDs) or Solid State Drives (SSDs).
  - SAS (higher performance but more expensive) or SATA (for entry level servers) slots.

  A server could also be attached to external storage.
- Special purpose devices like Graphical Processing Units (GPUs) and Tensor Processing Units (TPUs) for Machine Learning tasks.

4.1.2 Shape

The shape of a server is fundamental not only for space occupancy but also for cooling. Servers can be organised in many different shapes, some of the most important are

- **Towers**.
- **Racks**.
- **Blades**.

Towers

A tower has the same shape of commercial desktop computers. A tower can host a single server.

Advantages The main advantages of towers are that

- Are easy to customise.
- Are very cheap.
- Have a lot of free space. This is good for cooling but makes it harder to efficiently organise towers.

Disadvantages The main disadvantages of towers are that

- Consume a lot of space.
- Offer only a basic level of performance.
- Cable management is hard.

Racks

Racks are the most common shape for storing multiple servers. Racks are like shelves that develop vertically stacking one component above the other. Each rack can host servers and additional components (like networking or storage components). Each component is of a standardised size, in particular each a component is defined by its height; the unitary height is 44.45 mm (indicated with 1U) and a component can be a multiple of 1U (e.g a 4U component is 177.8 mm tall). As we have said, a rack can host also components different from servers, in particular a rack usually has

- Batteries on the bottom for power redundancy.
- Network components on the top, in what is called the Top Of the Rack (TOR).

Advantages The main advantages of a rack are

- They **create order** in the server room.
- **Cable management is easy**.
- Components can be **organised and interchanged easily** due to their standardised size.
- They are **cost effective**.

Disadvantages The main disadvantages of racks are

- Power and cooling management is hard because of the limited space. In particular there is no space for air to cool down the server. Furthermore the vertical disposition creates a difference in temperature between the lower and upper servers (hot air rises).
- Maintenance is hard because of the reduced space.

Blades

Nowadays data centres are organising servers in blades. This shape is much smaller that the others we have studied, thus it allows to obtain an high server density. A blade is made of many vertical slots where servers (which have a rectangular shape) can be slid in. Imagine blades as a cluster of vertical slots in which cassette like servers can be inserted. Blades can be also inserted in racks.

Advantages The main advantages of blades are related to the ease of maintenance and the compactness. In particular

- Since a server can be inserted and removed directly from its location, it's very easy to replace a server.
- The servers used for blades are very compact.
- Cable management is very easy, in fact blades use a simple and unified interface and the number of cables needed is very small.
- Thanks to its much simpler and slimmer infrastructure, load balancing among the servers and failover management tends to be much simpler.

Disadvantages The main disadvantages of blades are

- The compactness of blades is also a disadvantage in terms of cooling. In particular since servers have very little room for air, it's very hard to cool them.
- The initial cost of building a blade is very high.

4.1.3 Hardware accelerators

In modern days, data centres are required to execute computational intensive tasks. In particular, machine learning and deep learning algorithms are becoming more and more used. To cope with such workload we have to use hardware acceleration (i.e. to use some components that can speed up the execution of a specific class of tasks). The most popular accelerators are

- **GPU**s. Graphical Processing Units can be used for computer graphics but also for machine learning and other data intensive operations.
- **TPU**s. Tensor Processing Units can accelerate machine learning.
- **FPGA**s. Field Programmable Gate Arrays allow to design ad-hoc components.

Graphical Processing Units

Graphical Processing Units exploit data-parallel instructions; basically the same program of piece of code is executed, in parallel, on many different data elements. GPUs are therefore good at computing simple operations on different, uncorrelated data elements (e.g. compute the position of points in a space, where each point doesn't depend on the other points). Long story short, GPUs are built for parallel operations on independent data. Notice that GPUs aren't done only for graphics and machine learning, in fact they can accelerate whatever task that requires to execute the same operation on different data (e.g. how a protein interacts with different types of molecules).

To do so, GPUs have a large number of simple cores (opposite to CPUs that have few, yet complex cores) that can execute simple instructions of a simple instruction set. In other words GPUs have an higher core density, in fact in a die of (about) the same dimensions a GPU has an higher number of cores. Furthermore, GPUs have

- Higher throughput.
- Higher latency tolerance.
- Deeper pipelines (hundreds of stages).
- An high computations per memory access ratio.

Even caching is different between GPUs and CPUs. In particular the latter has bigger caches than the former.

Programming GPUs GPUs can't be programmed with the same languages used for CPUs. Some of the most popular GPUs programming languages are CUDA, OpenCL, OPENACC, OPENMP and SYCL.

Multiple GPUs GPUs can be connected to improve learning in machine learning algorithms. In particular each GPU handles a part of the model (i.e. learns some parameters) and finally the data is collected in a single node and broadcasted to all GPUs. An important part of ML acceleration is GPU to GPU communication, thus manufacturers have developed custom made protocols to improve communication performance. An example is NVIDIA's NVLink.

Tensor Processing Units

Tensor Processing Units are pieces of hardware designed ad-hoc by Google for machine learning acceleration. TPUs are able to execute a lot of tensor operations (super common in machine learning) at very high speeds. In general TPUs can be used both for learning and inference (i.e. predict an output from a given input). Notice that the former operation is the one that requires more computational power.

Versions of TPUs Google has developed three types of TPUs

- **TPUv1** can be used only for inference. This has been the first attempt at accelerating machine learning operations but it doesn't focus on the most intensive operation (i.e. learning).
- **TPUv2** added hardware for training with training completely dedicated to tensor operations.
- **TPUv3** improved cooling using liquid cooling, thus improving computational power.

FPGAs

FPGAs can be used for machine learning, too (for both learning and inference). The main advantage of FPGAs is that they are not specific for an operation but can be programmed to accelerated a number of different tasks (basically we can create whatever circuit we want). FPGAs are also quite power efficient. The main drawbacks of FPGAs are that they are very expensive and not as good as TPUs and GPUs at machine learning.

4.1.4 Equipment

Another important thing to keep in mind when building a server room is the equipment needed for cooling. In particular shelves (racks or blades) are organised in rows separated by a corridor (just like in a library). Corridors are fundamental not only to allow people to access the servers but also for cooling. In particular we have to place servers in to allow a continuous flow of cool air.

4.2 Storage

Data intensive applications are becoming more and more popular because of IOT, machine learning, media streaming and industry 4.0 (to say a few), thus data centres have to face a huge increment in storage demand. Having so much data is useful because it can help in taking decisions (the more data, the easier is to decide what to do based on experience) but it also poses problems in storing, communicating and processing such traffic. Furthermore, data has moved a lot from personal computers to cloud-based platforms, hence increasing the load on data-centres. Another important thing to highlight is that there is a difference between data produced and data stored, in fact data can be compressed (especially in case of redundant data) or we can one references to the location

where the actual data is (e.g. instead of sending an entire image in a mail we send the url of the image location on a cloud server).

Storing technologies There exists different storing technologies each suiting better to a specific problem. The most common are

- **Hard Disk Drives** HDD. HDDs are magnetic disks that allow fast access to big blocks of data. This technology is prevalent today.
- **Solid State Drives** SSD. SSDs are storing devices completely based on transistor logic (hence with no mechanical parts). This allows SSDs to reach staggering speeds with respect to HDDs. SSDs are becoming very popular (despite being more expensive than HDDs) because of their speed and reliability.
- **Non Volatile Memory express** NVMe.
- **Tapes**. Tapes are very old and slow storing devices that are still used today for long term access. One might want to store a large amount of data (e.g. logs) without accessing frequently.

HDDs and SSDs share the same interface. These technologies aren't exclusive, in fact they can be combined to achieve better general performances. For instance we can combine SSDs and HDDs to create a device that ensures good performance both on frequent accesses (thanks to the SSD) and large data accesses (thanks to the HDD). SSDs can also behave like caches for HDDs.

4.2.1 Hard Disk Drives

Disk abstraction

Before introducing the technological solutions used to implement HDDs, let us define how data is generally divided and organised. An operating system sees an hard disk as a collection of data blocks, each identified by a Logical Block Address. Data blocks are grouped in clusters that are the smallest unit with which the OS works (i.e. the OS moves data in clusters). Clusters contains both data and meta-data (i.e. data about the data) used to access the data. Typically, meta-data blocks contain

- File names.
- Directory structures and symbolic links.
- File size and file type.
- Creation, modification, last access dates.
- Security information (owners, access list, encryption).

Each cluster can contain either blocks of data or meta-data (not both). In particular a disk is usually made of

- Some meta-data blocks at the beginning (fixed) used by the OS for bootstrapping.
- Some meta-data blocks for describing the disk structure.
- Data and meta-data.

Read and writes When the OS wants to read or write some data, it has to

1. Query the meta-data to know where data is.
2. Access the data using the information obtained querying the meta-data.

Internal fragmentation One of the problem with the division in blocks and clusters is the fact that in some cases, if the data to store is not a multiple of the length of a cluster, then some space in a cluster remains unused. In this case we talk about internal fragmentation because a cluster is internally divided in a used part and in a unused one. The actual size on disk occupied by some data is computed as

$$size_{actual} = \text{ceil}\left(\frac{size_{data}}{size_{cluster}}\right) \cdot size_{cluster}$$

where

- $size_{actual}$ is the actual space occupied on disk, considering the fact that a cluster partially occupied has to be considered as completely occupied.
- $size_{cluster}$ is the size of a cluster.
- $size_{data}$ is the size of the data to store.

Given the actual size we can compute the wasted space as the difference between the actual and true size.

$$wasted = size_{actual} - size_{data}$$

Consider for instance that we want to store 43 bytes $size_{data} = 43$ in a cluster of 10 bytes ($size_{cluster} = 10$). This means that the actual size occupied is

$$size_{actual} = \text{ceil}\left(\frac{size_{data}}{size_{cluster}}\right) \cdot size_{cluster} = \text{ceil}\left(\frac{43}{10}\right) \cdot 10 = 50$$

in fact we occupy 5 clusters of 10 bytes each. This means that the wasted space is 7 bytes.

File deletion Another important thing to keep in mind is that when we delete some data from the disk, we are only removing the meta-data and telling that the location of memory can be written. Basically there is no way for the OS to find the deleted data, but the data is still there. This means that if we scan the whole disk, we can still find deleted data.

External fragmentation Usually we would like data to be stored in contiguous clusters to be accessed faster. When the disk is empty, it's easy to ensure this property but when the disk fills up, and files are removed, we might have to store data in non-contiguous clusters. This problem can reduce the performance when reading. Notice that external fragmentation is a problem only for HDDs.

Structure

Hard disk drives are made of multiple spinning magnetic disks called platters stacked on on top of the other. Each platters is divided in concentric tracks and each track is further divided in sectors. The tracks with the same diameter on all platters are called cylinders. Platters are accessed using a magnetic arm, moved by a motor, that has a magnetic head for each platter. Figure 4.1 shows a platter seen from above. A more complete representation of an hard disk is shown in Figure 4.2.

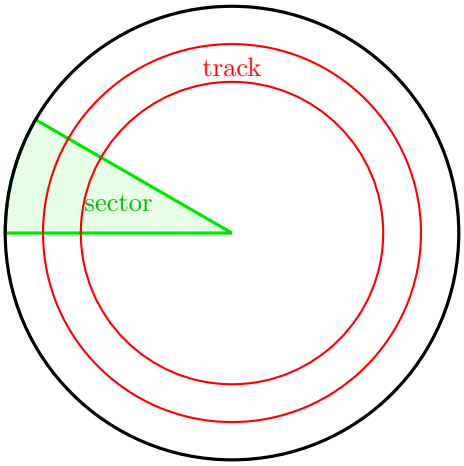


Figure 4.1: Sectors and tracks in a platter.

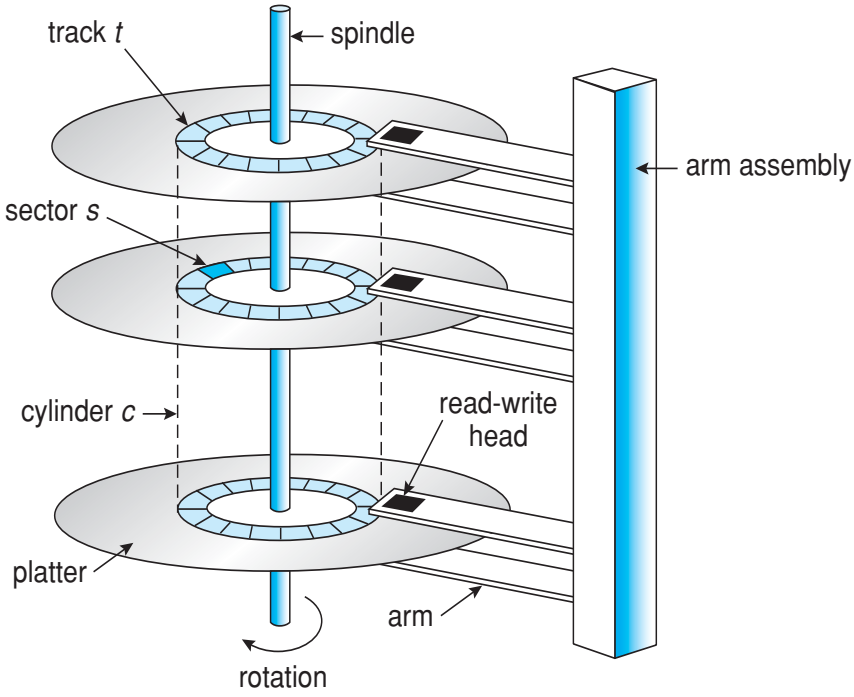


Figure 4.2: The complete structure of an hard disk drive.

Access

Differently from tapes, hard disks can have random access, but the time needed to access the data depends on where the data is (and on where the data is with respect to the previous data retrieved). When we want to access some cluster (either for writing or reading) we have to

1. Wait for the platter to have reached the correct sector. Remember that the disks are continuously rotating (at the same speed, measured in rotations per minutes) in the same direction. The faster the disks spin, the easier is to access data.
2. Move the arm to reach the correct track and sector. The arm's movement is called seek.
3. Access the data using the magnetic head.

As we can see each action increments the time needed to access some data, in particular we can define four delays

- The **rotational delay** (or latency). The rotational delay is the time we have to wait before having the needed sector aligned with the head. This delay mainly depends on the rotational speed of the disks.
- The **seek delay**. The seek delay is the time we have to wait to reach the desired track.
- The **transfer time**. The transfer time is the time needed to magnetically read or write the data.
- The **controller overhead**. The controller overhead is the time needed by the HDDs controller to handle the request and decide which sectors and tracks have to be accessed.

Notice that the first two delays only account for reaching the correct position on the disks and are related to mechanical movements while the last two delays consider the time needed to access the data and have a digital source. An important thing to notice is that subsequent clusters on different tracks aren't adjacent because when we want to reach a cluster C_{i+1} from C_i we have to consider that the disk spins and the head has to move (which takes time). If the clusters were adjacent we would have had to wait a full rotation to reach C_{i+1}. This very concept is shown in Figure 4.3.

In formulas we can write that the total delay is the sum of four components

$$T_{IO} = T_{rot} + T_{seek} + T_{transfer} + T_{overhead}$$

Notice that T_{IO} doesn't consider the delay for multiple requests that are waiting to be executed (i.e. T_{IO} is the time needed from when the operation starts to when it finishes). Let us analyse each component separately.

Rotational delay The rotational delay depends on the rotational speed of the disks (measured in rotations per minute, RPM). Given the RPMs of a HDD we can compute the average rotational delay as half of the time needed to complete a full rotation.

$$T_{rot,avg} = \frac{T_{fullrot}}{2}$$

If we call R_{min} the number of rotations per minutes and $R_{sec} = \frac{R_{min}}{60}$ the number of rotations per second we obtain the average rotational time in seconds as

$$T_{rot,avg} = \frac{T_{fullrot}}{2} = \frac{\frac{1}{R_{sec}}}{2} = \frac{1}{2 \cdot R_{sec}} = \frac{1}{2 \cdot \frac{R_{min}}{60}} = \frac{30}{R_{min}} \quad [s] \tag{4.1}$$

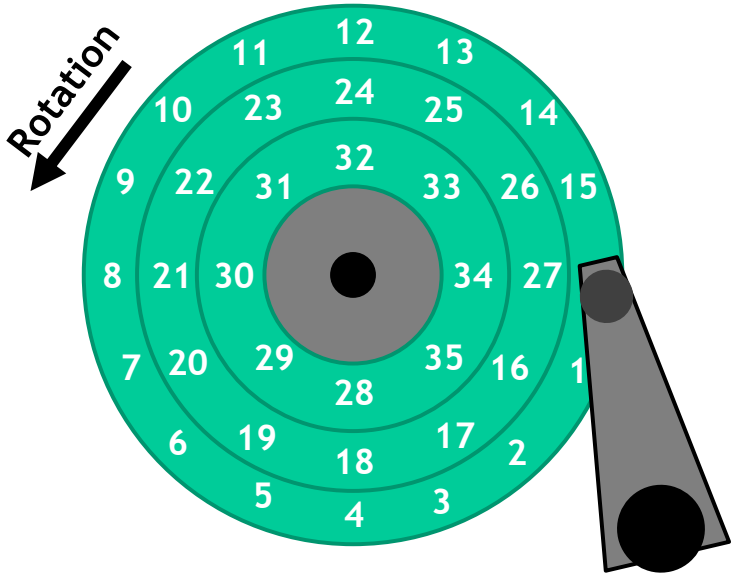


Figure 4.3: Clusters (shown as numbers) organisation on tracks.

This result is justified by the fact that we assume that from a sector we can get to any other sector, thus on average we do half a turn.

Seek delay The seek delay depends on how far the head is from the destination. Notice that the head, differently from the disks, moves only when it's needed.

When we analyse the seek delay we have to consider that the head goes through four phases

1. The **acceleration phase** in which the head starts moving.
2. The **constant speed phase** in which the head reaches its maximum speed.
3. The **deceleration phase** in which the head starts decelerating.
4. The **stop phase** in which the head stops on the correct location of the disk.

This demonstrates that the head doesn't move at constant speed, thus time is not linear. However, we can empirically show that the time can be approximated as linear and that the average seek delay is

$$T_{seek,avg} = \frac{T_{seek,max}}{3} \tag{4.2}$$

where $T_{seek,max}$ is the time needed to go from the outermost to the innermost track. This information can be easily found in the data-sheet of the HDD.

Transfer time and controller overhead Transfer time and controller overhead depend on the bandwidth of the controller and in practice are much smaller than rotational and seek delays because the latter ones have a mechanical origin. This means that when optimising the input/output time of an HDD we should focus on the rotational and seek delays. In formulas we can divide the mechanical

delays (rotational and seek delays) from the digital ones (transfer time and controller overhead) and write

$$T_{IO} = T_{mechanical} + T_{digital}$$

Data locality

An important thing to keep in mind when considering access time is data locality, that is how much data is stored in contiguous clusters (data in contiguous clusters is accessed faster). Given the percentage d of files that are stored in contiguous clusters we can compute the I/O time as

$$T_{IO} = (1 - d)\big(T_{rot} + T_{seek}\big) + T_{transfer} + T_{overhead} \tag{4.3}$$

Notice that only T_{rot} and T_{seek} are effected by data locality, in fact the digital part of T_{IO} doesn't depend on where the data is. In other words, data locality allows to reach a location swiftly but the time to access it doesn't change.

Caching

HDDs exploit caching techniques to improve input/output performance. In particular, HDDs, when the OS reads a sector, start caching the following sector because probably (if data locality is strong) the OS will query the following sector, too. Caches can also be used for writing, consider for instance two modifications of the same file in a short period of time. The HDD can cache the first write request and, when the second request arrives, it can finally write the new doc on disk. In this example, caching has reduced the number of accesses from 2 to 1. Caching write requests doesn't come for free. In particular, the HDD notifies the OS, telling that the write has been completed, when the data is written in cache (not in memory). This is a problem because we have to ensure that, when the OS is notified (i.e. when the transaction is complete), the data is permanently saved to storage (remember that caches are volatile memories). Write caching can be implemented in two ways

- **Write back caches**. Write back caches report a write complete to the OS as soon as the data is written in cache. This behaviour might lead to inconsistent states because if the power goes off before saving the cached data to disk, some updates are lost. To solve this problem we can add backup power (like a small battery or a capacitor) that, in case of power loss, keeps the HDD alive the time needed to save the cached data to disk. Another option is to keep a log of all operations done by the OS to rebuild a consistent state.
- **Write through caches**. Write through caches report a write complete to the OS as soon as the data is written on the disks.

Scheduling

HDDs can access data randomly, but the access time depends on where the head is and where it has to move. This means that we can schedule I/O requests to move the head in a smart way and improve performance. Say for instance that an HDD receives the requests above (where the number represents the number of track on which the head has to move)

```
R 1
R 5
W 2
R 4
```

If the requests are executed in order as soon as they arrive we have to visit tracks 1, 5, 2 and 4. However, if we put the requests in a buffer, we can schedule them and visit 1, 2, 4 and 5. As we can see the second option requires the head to move less because it doesn't have to go back and forth along the disk. Some scheduling techniques are

- **First Come First Served** (FCFS). FCFS is the most basic and unoptimised scheduler. It executes a request as soon as it arrives.
- **Shortest Seek Time First** (SSTF). SSTF searches and executes the request with the shortest seek time with respect to the previous request. This implies that the scheduler has to compute the track distance for every buffered request. The main problem of this technique is that we privilege an area of disk, because we execute all requests that are close one to the other (consider for instance 1, 10, 11, 3, 2, 4, 0; if we start for 1 then we execute 0, 2, 3, 4 and finally 10 and 11). This problem is called starvation.
- **SCAN**. The SCAN algorithm solves starvation by scanning the disk. In particular the head is bounced back and forth from one border to the other. Notice that if the head is on track 4 (moving towards highest tracks) and a request for track 3 arrives, such request is handled only after reaching the border and bouncing back. This means that in the worst case scenario a request is handles after two scans of the disk (request for track 1 and head in track 2 moving towards bigger tracks). Moreover, tracks in the middle of the disk are visited more than the one at the borders.
- **Circular-SCAN**. C-SCAN tries to solve the SCAN problem of having to scan the disk twice when it arrives a request for the track before the head. In particular, when the head reaches the outer border of the disk, it comes back (without accessing intermediate tracks) to the inner border (basically we are simulating a circular scan). One might think that moving the head from one border to the other without accessing the disk is a waste of time, however we have to highlight that the head can move at maximum speed, thus the time lost is very small. On the other side, when the head does little movements, it loses some time during acceleration and deceleration.
- **C-LOOK**. C-LOOK is similar to C-SCAN, in particular it avoids going to the boards of the disk if no request has to be served there. Notice however that this scheduler, differently from C-SCAN, needs to look at the requests to check if there is query on the border.

Scheduling can be done

- By the **operating system**. The OS can try and schedule read and writes to optimise disk access but, since it might not know how sectors are organised on disk, it might be very weak. Moreover, some studies show that OS scheduling, combined with HDD scheduling might cause problems.
- By the **HDD**. Typically the HDD itself schedules the requests because it knows the characteristics of the disk. Usually HDD schedulers use a **Disk Command Queue** that stores and reschedules pending requests.

4.2.2 Solid State Drives

Solid State Drives, differently from HDDs, are storage devices with no mechanical parts and based only on transistors. Removing the mechanical part allows SSDs to be faster and reduce latency.

Moreover, the time needed to obtain a data is almost independent from its location. Another thing to remember is that mechanical components are much more vulnerable to physical damage, thus SSDs are in general more solid that HDDs. The main disadvantage of SSDs is that they have a limited number of writes.

Cell technology

The main difference between SSDs is how data is stored in a cell of memory (i.e. the basic storage unit). In particular,

- A **Single Level Cell** stores one bit.
- A **Multiple Level Cell** stores two bits, i.e. it has four possible states (playing with voltage ranges).
- A **Triple Level Cell** stores three bits, i.e. it has eight possible states.

There exists types of cell that can store even 4 or 5 bits but the ones we have seen are the most common one.

Choosing the SSD with the right type of cell is very important, in fact each technology has it's advantages and disadvantages. In particular

- **Denser cells** (i.e. with more bits per cell) can store **more information** in about the same space but are much **slower**. This happens because the voltage ranges used to differentiate the levels are narrower, thus additional logic (that increases latency and reduces speed) has to be used to precisely execute read and writes. Consider for instance a 5 volts interval. If we have 2 levels only, the threshold is around 2.5 volts and it's easy to measure. On the other hand, with 8 levels, the classification range is 0.625 volts which is harder to detect, thus we need additional logic to precisely detect a level. For this reason TLC based SSD are used on general purpose computers that need a lot of memory but in which speed isn't critical.
- Cells with **few levels** are much **faster** but can store **less data** in the same space. For this reason SLC based SSDs are used in data centres that have to ensure a lot of speed and have a lot of space to store many SSDs.

In a nutshell, **denser cells are slower**.

Structure

An SSD is a NAND flash memory organised in flash pages that contains logical blocks and each logical block contains many pages. In particular,

- A logical **block** contains multiple pages and it's identified by a Logical Block Address (LBA). A block is the smallest unit that can be erased.
- A **page** (not the flash page) is a sub-unit of a block and is the smallest unit that can be written or read. A page can be in three states
  - **Empty** or erased. An empty page doesn't contain data.
  - **Dirty** or invalid. A dirty page contains data that is not in use.
  - **In use** or valid. A valid page contains data that can be read.

Interfaces and write amplification problem

SSDs have the same physical interface as HDDs, however they expose a different software interface. In particular an SSD can be

- **Read**.
- **Programmed** (or written).
- **Erased**.

however the OS usually issues write or read commands. This problem comes from the fact that SSDs can't overwrite in-use data, thus it (the block in which the page is) has to be cached, erased and written back. This problem is called write amplification problem. Let us analyse the write amplification with an example. Say we have a small SSD with only one block. The block is made of 5 pages, each of which has a size of 4 KB. Say we want to write a document of 4 KB. Since the SSD is empty we can put it in the first page. After this operation, we want to store another file (say an image), this time occupying 8 KB. Such big of a file doesn't fit in a single page, hence we have to use page 2 and 3. Currently we have occupied pages 1 (with the document) and pages 2 and 3 (with the image). The next operation to execute is a deletion of the document. Remember that, when deleting a file we are only changing its meta-data, hence the first block is still occupied (it's in the dirty state). Now we want to store a video that occupies 12 KB, i.e., three pages. It's true that we have 3 pages not valid (pages 1, 4 and 5), however the first page is dirty and not empty, hence it can't be written. The SSD can't simply erase page 1, since it can only erase blocks, hence it has to:

1. Save the block in cache.
2. Delete page 1 in the cache.
3. Store the video in pages 1, 4 and 5 of the cache.
4. Eraser the block in the SSD.
5. Store back the block from cache to the SSD.

As we can see, this sequence of operations is much more costly than simply writing three pages directly in the SSD.

Flash Translation Layer

The write amplification problem is solved by a component in the SSD called Flash Translation Layer (FTL) that maps OS requests in requests that can be actually executed by the SSD. Basically, the FTL makes the SSD visible to the OS like a normal HDD. An FTL has many tasks, in fact it handles

- **Data allocation** and **address translation** to reduce the write amplification problem.
- **Garbage collection** to reuse pages with old data.
- **Wear levelling**. FTL should try to spread writes across the blocks of the flash ensuring that all of the blocks of the device wear out at roughly the same time.

Log structured structure Mapping physical addresses (i.e., the page addresses used by the SSD) and page addresses (i.e., the page addresses used by the OS) is very hard in SSDs, thus we have to use a log structured system in which pages are written one after the other on the SSD and a table maps the page address to the physical address. Say, for instance, that the OS wants to write a in page 100. The SSD (let us consider it empty) puts a in the first page available (say page 0) and maps 100 to 0. In other words the table maps a logical page address to a physical address.

Notice that this behaviour is relevant when we want to modify the value on a page. Consider the example of before and say we want to write b in page 100. Normally the SSD would have to cache the entire block where a is, delete the block, modify the value of a in the cache and store the block. Using the log structured system we simply have to

1. Store b in a new physical page (say page 4).
2. Change the mapping of 100 in the mapping table. Practically, 100 is mapped to 4.
3. Mark the page where a is as dirty.

This sequence of operations significantly increases the writing speed (especially when the SSD is empty).

Another important thing to analyse is how large the mapping table should be, in fact having a large mapping table can be very expensive (the more space, the more power, the more cost). There exists several techniques to reduce the size of the mapping table

- **Block mapping**. Block mapping maps logical blocks in physical blocks. This means that we know in which block a logical page is, but not in which specific physical page. Put it in another way, the mapping is done at block level (i.e., we map a block into a block) and not at page level. This method allows to save space (because a physical block contains many physical pages) but has some drawbacks. In particular when a logical page is moved from a block to another, all the pages in the same block have to be moved or we have to pay the write amplification.
- **Hybrid mapping**. Hybrid mapping reduces space but increases latency. In particular the FTL uses two tables
  - One with fine grain that addresses the page (i.e. the table we initially introduced).
  - One with gross grain that addresses the block (i.e. like in block mapping).

  When the operating system wants to access some data, the FTL initially checks the page table and then, if the address is not there, queries the block table. On the other hand, when the FTL has to write some data at a certain page, it initially checks if that page is in the block table. If not, a block mapping entry is added. If the page is in the block table, we add (or remap) an entry in the page table.
- **Caching**. Caching allows to save some mapping in a cache to allow better performances. In this case we pay the price of cache misses.

Garbage collection Using a log approach creates many memory areas with dirty values. Such pages should be deleted to allow new values to be written. This job is done, usually asynchronously and periodically, by the garbage collector. In particular it scans the memory looking for dirty pages and

- If an entire block is dirty, it immediately deletes it.
- If a block is half dirty, it saves the pages in-use in another block and deletes the former block. Notice that usually this procedure is done with multiple blocks, namely the in-use pages of multiple blocks are merged in a single blocks so that many blocks can be erased and the blocks are compacted.

Garbage collection is almost transparent to the user if the SSD is connected to power and the user isn't using it. On the other hand, if the garbage collector works during I/O operations, the user might experience latency.

Also notice that an SSD's speed drops when the available storage decreases. This happens because the garbage collector has to collect many blocks to free pages and because of the write amplification problem. A way to reduce this issue is over-provisioning. Basically an SSD manufacturer installs more space than the one declared to allow log table to work efficiently by occupying more blocks.

Wear levelling Cells in SSDs can be written only a limited number of time (and it's not a fixed value, we can only approximate it). This means that the SSD has to be careful where the values are written. In particular if we keep writing an area of memory, this will wear fast hence reducing the total capacity and speed of the SSD. When a block is heavily accessed and another isn't (we need something that monitors the number of accesses), the FTL can decide to swap them and wear the latter block to increase the life of the former one. Also notice that the log structured approach and the garbage collector can improve the duration of blocks. Usually, FTL can also spread the data and make writes along the SSD.

One final remark: SSDs are not affected, as HDDs by data locality, in fact the access time is independent from the fact that two blocks are stored one close to the other. Furthermore we must not defragmentate an SSD, not only because it would be useless, but also because it would increase the wear.

Operating system operations

When the operating system wants to delete a file on a SSD, it simply deletes and rewrites the meta-data about that file. This means that the old meta-data is garbage and can be collected, however, the data itself is still valid for the SSD because the OS only asked to delete the meta-data and not the actual data. This is a big problem because it can swiftly occupy the storage and reduce the speed of an SSD.

To solve this issue, operating systems can use a `trim` operation that notifies the SSD to

- Update the meta-data.
- Invalidate the files so that they can be collected.

4.2.3 Comparison

SSDs are getting very popular for personal computers because of their speed and because the wear effect is limited since the number of writes done by a PC is very small. On the other hand, data centres write data much more often, thus SSDs aren't used that much. Usually they are used as temporary storage (closer to the server), while HDDs are used as main storage (outside the server). To better explain this difference we can introduce two SSD metrics

- The **Unrecoverable Bit Error Ratio** (UBER). The UBER metrics measures the occurrence of data errors per bit read.

- The **TeraBytes Written** (TBW). The TBW is an endurance rating and measures the total amount of data that can be written into an SSD before it's likely to fail. In other words, the TBW measures the number of terabytes that can be written to the SSD while still meeting the requirements. Notice that this measure isn't precise and can only be estimated. For instance a typical TBW for a 250 GB SSD is between 60 and 150 TB, this means that a normal computer might take a very long time to write that volume of data (240 days if we consider a TBW of 60 TB and the SSD is rewritten, i.e. 250 GB are written, every day). Also notice that some studies demonstrated that the lifetime of an SSD depends on too many factors to be precisely estimated.

4.2.4 RAID

Redundant Arrays of Independent (or Inexpensive) Disks is a technique to improve performance size and reliability of a storing device. In particular, instead of using a single disk, RAID disks are made of multiple disks that are seen by the user (i.e. by the OS) as a single disk. Basically the interface of RAID disks is the same of simple disks and the presence of multiple disks is transparent to the user. To allow this behaviour, RAID disks use an adapter (also called controller) that exposes the same interface of a single disk and handles the requests coming from the OS. Data in a RAID system can be organised in different ways to obtain different improvements with respect to a normal disk, in particular we can use

- **Data striping** to improve **performance**.
- **Redundancy** to improve **reliability**.

Data striping In RAID disks, data is striped across several disks that can be accessed in parallel. This allows to obtain

- An **high data transfer rate**. High data transfer rates are obtained through large data accesses.
- An **high input/output rate**. High I/O rates are obtained through small but frequent data accesses.
- **Load balancing** capabilities.

As we can see, data striping improves the performance of a RAID system, yet it worsens the reliability. In particular, having N disks, the time that elapses before the first failure is smaller with respect to a single disk, because the probability of a failure is the sum of the probability of a disk to fail.

Let us now consider striping more in the detail. In striping, data is written sequentially in stripe units on multiple disks according to a cyclic algorithm that visits the disks in a round-robin fashion. We also have to remember that

- The stripe unit is the dimension of the unit of data that are written on a single disk (e.g. bit, byte, block).
- The stripe width is the number of disks considered by the striping algorithm.

Redundancy Redundancy (also called mirroring) allows to replicate data across the disks, thus every block of data has at least two copies in the system. This means that if one disk fails, the OS can still access the data because it's available on another disk. However, redundancy reduces the performance and the storage of a RAID disk because data has to be written twice and some space is occupied by redundant data.

Data access Data can be accessed

- **Sequentially** (i.e. from one block in disk i to the next block in disk $i+1$).
- **Randomly**.

In both cases we can spread data accesses along all the disk to improve performance.

RAID hardware RAIDs are complex computer systems that contain

- A dedicated CPU to control the disks.
- Software.
- Both a RAM and non-volatile memories.

RAID levels RAID is a general technology that can implement different technologies to offer different functionalities. In particular

- **RAID 0** offers striping only.
- **RAID 1** offers mirroring only.
- **RAID 0+1** and **RAID 1+0** offer both mirroring and striping but in different order.
- **RAID 2** offers bit interleaving.
- **RAID 3** offers byte interleaving and redundancy through parity disks.
- **RAID 4** offers block interleaving and redundancy through parity disks.
- **RAID 5** offers block interleaving and redundancy through distributed parity blocks.
- **RAID 6** improves the redundancy of RAID 5. In particular it tolerates two disk failures at the same time.

When we measure the performance of RAID we should compare it with the performance of a single disk when using random access or sequential access. In particular we will consider the random access rate R and the sequential access rate S.

RAID 0

RAID 0 implements only data striping, hence improving performance and reducing reliability. More precisely, the main advantages of RAID 0 are

- Low cost.
- Linear performance improvement.

However this comes at a cost, in fact the time to first failure reduces.

Striping Striping allows to spread the data in multiple disks. Say for instance we want to store 8 blocks of data (from 0 to 7) in $N = 4$ disks. The striping algorithm distributes the data as in Figure 4.4. Basically, block i is stored

- in disk $i \mod N$ ($i \mod 4$ in our example)
- with an offset from the fist block of the disk of i/N ($i/4$ in our example).

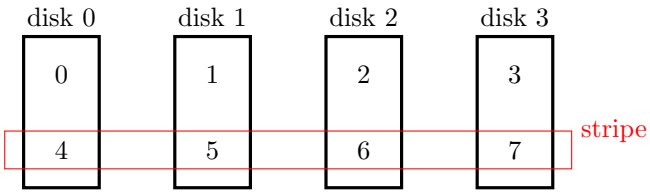


Figure 4.4: An example of striping.

The blocks of data at the same level for each disk (e.g. block 0, 1, 2 and 3) are called stripe.

Chunk sizing A disk can store many chunks of data. The dimension of the chunk has an impact on performance, in particular

- Smaller chunks improve parallelism.
- Larger chunks improve reduce seek time.

For instance, the RAID in Figure 4.4 has a chunk size of 1 block, while the RAID in Figure 4.5 has a chunk size of two blocks. Usually, RAID uses 64 KB chunks.

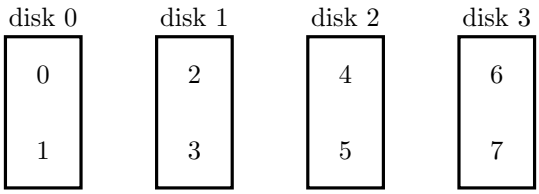


Figure 4.5: An example of striping with chuck size of 2 blocks.

Performance If we consider a RAID of level 0 with N disks,

- The capacity of the system is N times the capacity of a single disk.
- The reliability is 0 because if one disk fails, data is permanently lost.
- The performance on sequential access linearly increases because the data is completely spread along multiple disks.
$$N \cdot S$$
- The performance on random access linearly increases because the data is completely spread along multiple disks.
$$N \cdot R$$

RAID 1

RAID 1 implements only data mirroring (i.e. redundancy). Basically, data is written on multiple disks so that we have at least two copies for each block of data (on different disks). This technique allows to improve reliability because if a disk fails, we can still access the data on another disk, however it reduces the available space (if we have 2 TB of total space, we can only save 1 TB of data because the other 1 TB is used for redundancy). Furthermore, mirroring also increases access time (with respect to RAID 0) because data has to be written on multiple disks. Notice that, even if RAID 1 is slower than RAID 0, it has still good performances, in fact the controller can read the disk with the smallest queue.

Performance If we consider a RAID of level 1 with N disks,

- The capacity of the system is $N/2$ times the capacity of a single disk because we have two copies of each data block.
- The reliability is 1 because if one disk fails, data can be read from another disk.
- The performance of sequential reads is half the one of RAID 0 because we can read in parallel only from $\frac{N}{2}$ disks.
$$\frac{N}{2} \cdot S$$
- The performance of sequential writes is half the one of RAID 0 because we have to write on two different disks.
$$\frac{N}{2} \cdot S$$
- The performance of random reads is, in the best case scenario the same as RAID 0 because we don't access data in a specific order.
$$N \cdot R$$
- The performance of random writes is half the one of RAID 0 because we have to write on two different disks.
$$\frac{N}{2} \cdot R$$

Consistent updates RAID 1 replicates data in different disks, thus we have to be sure that, when the OS wants to write some data, it's actually written on two different disks. Say for instance that, during a write operation, one disk loses power and the block can't be written. We have to ensure that data get correctly replicated, even in these situations. To solve this issue we can use caches that store all pending operations and remove them only when all copies have been written.

RAID 0+1 and RAID 1+0

Striping (offered by RAID 0) and mirroring (offered by RAID 1) can be combined to improve both performance and reliability. In both cases, both functionalities are implemented, but in different order, hence we obtain different level of performance. In both cases the RAID is made of $m \cdot n$ disks, divided in m groups of n disks each. If we consider RAID $x + y$, then RAID x is applied to each group of disks and then RAID y is applied to all groups (considering a group as a single disk).

RAID 0+1 RAID 0+1 applies striping to each group of n disks and then mirroring to the groups. This means that if we consider a group, data is organised in stripes and divided between the disks of the group, but no redundancy is applied. However, if we consider all the groups, the data written in a group is mirrored in another group. Basically we are creating redundant groups of striped data. An example of RAID 0+1 with 6 disks is shown in Figure 4.6. Notice that, after the first failure, RAID 0+1 becomes a simple RAID 0, because one replica can't be accessed anymore. Say for instance disk 3 fails, we can't access disk 3, 4 and 5 but we can still use disk 0, 1 and 2 that are RAID 0.

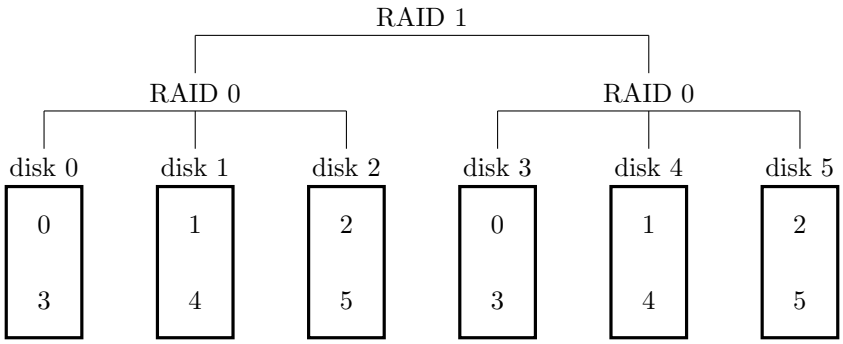


Figure 4.6: An example of RAID 0+1 with 6 disks.

RAID 1+0 RAID 1+0 applies mirroring to each group of n disks (i.e. when we write to a group data is written in two different disks) and striping to all groups. An example of RAID 1+0 is shown in Figure 4.7. RAID 1+0 is used in databases with very high workload where fast writes are required.

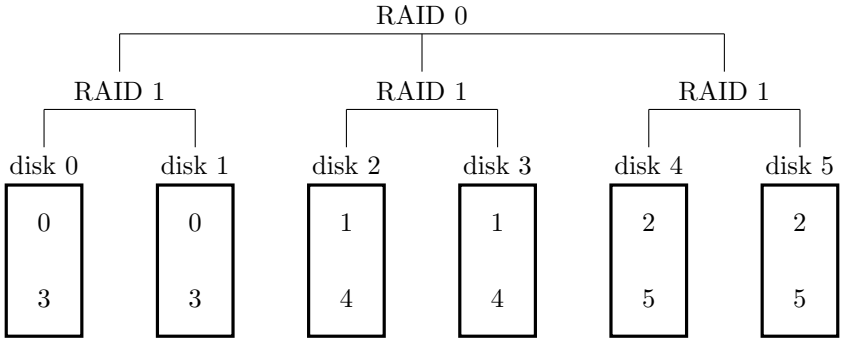


Figure 4.7: An example of RAID 1+0 with 6 disks.

Performance Performance-wise, both implementations have the same speed and storage capacity are the same, however RAID 1+0 is more reliable than RAID 0+1. Let's consider Figures 4.6 and 4.7 to understand this difference in reliability. In the first case, if disk 0 fails, then the whole system fails if either disk 3, 4 or 5 fail (because these disks contain the replicas of the data in the RAID that failed). In other words we have 3 possibilities out of 5 to get a total failure. On the other hand if disk 0 fails in a RAID 1+0, then the whole system fails only if disk 1 fails too, namely we have

a total failure in 1 case out of 5 which is less probable than the previous scenario. An example of RAID 4 is shown in Figure 4.8.

RAID 4

RAID 1 is reliable but wastes a lot of space, in fact, half of the capacity is dedicated to redundancy. RAID 4 tries to solve this issue using information coding techniques to build light-weight error recovery mechanisms (e.g. parity bits). In particular, out of the N disks that are part of the RAID, 1 is used to store the parity bits of the data in the other $N-1$ disks. The parity bits in the parity disk can be used to check the correctness of the data and to restore them in case of errors.

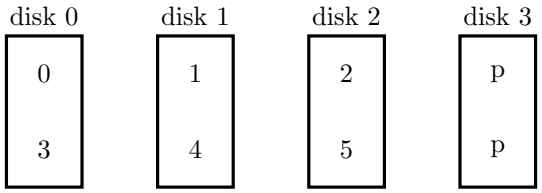


Figure 4.8: An example of RAID 4.

Parity bits Each block of a disk contains the parity bits of the stripe of the same level. In particular the block at level i contains the `xor` of the parity bits at the same level of the other disks. Parity bits in the parity disk can be computed in two ways

- **Addictive**. When some block is overwritten, the controller reads the blocks of the same stripe in the other disks and recomputes the parity bit in the parity disk. Practically, the controller has to read $N-1$ disks and write 1 disk (the parity disk). The addictive way is useful when we have to write all disks.
- **Subtractive**. When some block is overwritten, the controller reads the old value from the parity disk, reads the old value from the overwritten block, and updates the parity bit as

  $$parity_{new} = data_{old} \oplus data_{new} \oplus parity_{old}$$

  The subtractive technique is useful for random accesses on some disks only.

Performance If we consider RAID 4 with N disks

- The **capacity** is $N-1$ because a disk is used for parity bits.
- With respect to **reliability**, RAID 4 allows one drive to fail but, after the failure it requires a lot of time to recover data.
- The **performance on sequential read and writes** is $N-1$ times the one on a single disk because we can exploit striping on $N-1$ disks and use the addictive method to compute and write the parity bits.

  $$(N-1) \cdot S$$

- The **performance on random reads** is $N-1$ times the one on a single disk because the parity bits aren't modified, only accessed with a marginal reduction in access time.

$$(N-1) \cdot R$$

- The **performance on random writes** is worse than the one of a single disk, in particular it's half the one of a disk.

$$\frac{R}{2}$$

  This happens because every time a block is written we have to compute the parity bits, most of the times using the subtractive method. Namely, we have to access the parity disk, hence we create a queue of read and writes requests that can only be executed one at a time.

RAID 5

RAID 5 solves RAID 4's issue of expensive random reads removing the parity disk and distributing the parity bits across the N disks. In practice, for each disk, a block is dedicated to the parity bits of its stripe. A representation of this structure is shown in Figure 4.9 (block p represents a block containing parity bits).

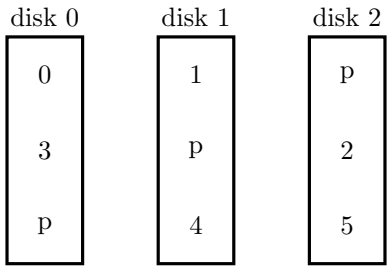


Figure 4.9: An example of RAID 5.

Performance If we consider a RAID 5 with N disks,

- The storage **capacity** is $N-1$ because we are still using the space of one disk to store parity bits, but it's spread along multiple disks.
- The **reliability** is identical to RAID 4, hence only one disk can fail and the recovery process is expensive.
- The **performance on sequential reads and writes** is identical to RAID 4 because we can still leverage striping and addictive parity.

$$(N-1) \cdot S$$

- The **performance on random reads** is a little better than RAID 4 because we can parallelise reads among all N disks.

$$N \cdot R$$

- The **performance on random writes** improves by a lot with respect to RAID 4 in fact we obtain an improvement of a factor $\frac{N}{4}$ over the single disk case.

$$\frac{N}{4} \cdot R$$

This result is given by the fact that, every time we have to compute the parity bit after an operation we have to

1. Read the old data parity.
2. Read the stripe parity.
3. Write the new parity.
4. Write the new data.

hence two reads and two writes are required. Such operations are however spread over N disks (because the parity bits are not on the same disk), hence we obtain the result above.

RAID 6

RAID 6 increases the tolerance to faults with respect to RAID 5. In particular, RAID 6 can handle two concurrent failures using Solomon-Reeds codes with two redundancy schemes. Each stripe has two parity blocks that are distributed on different disks (but no disk is completely dedicated to parity blocks).

Notice that increasing the number of parity blocks, we decrease the total capacity and increase the cost of writes since we always have to update the parity, too. More precisely RAID requires 6 disk accesses (1 write for the data, 1 read for the old data, 2 reads for the old parities and 2 writes for the new parities).

Also remember that RAID 6 needs at least four disks.

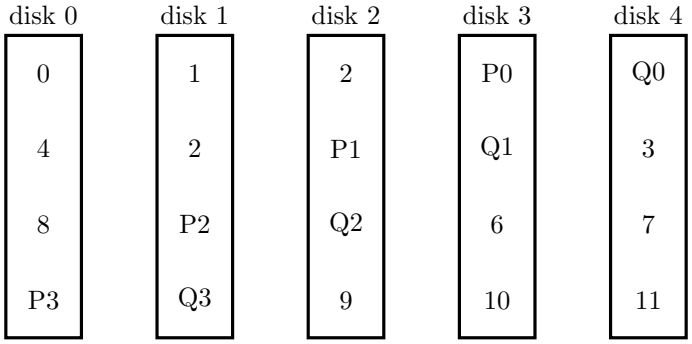


Figure 4.10: An example of RAID 6.

Hot spare

Usually, RAID systems include an idle, unused, spare disk that is used, when a disk fails, to immediately restore the broken disk.

Implementation

RAID systems can be implemented in

- **Hardware**. Hardware implementations are fast and reliable (more than software ones) but migrating a hardware RAID array to a different hardware controller almost never works.
- **Software**. Software arrays are simpler to migrate and cheaper, but have worse performance and less reliability due to the consistent update problem.

Reliability computation

An important information about a RAID system is the time that elapses before the first failure. This value is called Mean Time To Failure $MTTF$. If we assume a constant failure rate, an exponentially distributed time to failure, and independent failures then the $MTTF$ of a RAID with N disks can be computed as

$$MTTF_{RAID} = \frac{MTTF_{disk}}{N}$$

This formula doesn't include fault tolerance mechanisms, i.e. it's the $MTTF$ of a RAID 0 system. However, higher level RAID systems, have mechanisms that allow to restore data and the disk fails only if something fails while restoring the disk that failed. Practically we should include the probability P that a failure happens when the disk is restored and the formula becomes

$$MTTF_{RAID} = \frac{MTTF_{disk}}{N} \cdot \frac{1}{P}$$

Now we should understand what is the value of P for each level of RAID.

RAID 1 In RAID 1 we have a single copy of each disk, hence only one of the two disks can fails. In general, if we have N disks and we are lucky, $N/2$ disks can fail without any problems (we are lucky if among the $N/2$ disks there are no two copies). This means that P is the probability that a copy of the failed disk also fails. In formulas

$$P = \frac{1}{MTTF_{disk}} \cdot MTTR$$

where

- $\frac{1}{MTTF_{disk}}$ is the failure rate for the copy of the failed disk.
- $MTTR$ (Mean Time To Repair) is the time that elapses between failure and restore of the disk.

Putting all together the $MTTF$ of RAID 1 is

$$\begin{aligned} MTTF_{RADI1} &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{P} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{\frac{1}{MTTF_{disk}} \cdot MTTR} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{MTTF_{disk}}{MTTR} \\ &= \frac{(MTTF_{disk})^2}{N \cdot MTTR} \end{aligned}$$

RAID 0+1 In RAID 0+1, the system fails when one disk in the other group (i.e. the group where the failed disk is not) fails, hence P is the probability that one disk in a group fails. In formulas

$$P = \frac{G}{MTTF_{disk}} \cdot MTTR$$

where

- G is the number of disks in a stripe group.
- $MTTR$ (Mean Time To Repair) is the time that elapses between failure and restore of the disk.

Replacing P we obtain

$$\begin{aligned} MTTF_{RADI0+1} &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{P} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{\frac{G}{MTTF_{disk}} \cdot MTTR} \\ &= \frac{(MTTF_{disk})^2}{N \cdot G \cdot MTTR} \end{aligned}$$

RAID 1+0 In RAID 1+0, the system fails if both copies in a group fail, hence P is the probability that the copy of the failed disk fails. In formulas

$$P = \frac{1}{MTTF_{disk}} \cdot MTTR$$

where

- $\frac{1}{MTTF_{disk}}$ is the rate of failure of the copy of the disk that failed.
- $MTTR$ (Mean Time To Repair) is the time that elapses between failure and restore of the disk.

When we replace P we get

$$\begin{aligned} MTTF_{RADI0+1} &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{P} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{\frac{1}{MTTF_{disk}} \cdot MTTR} \\ &= \frac{(MTTF_{disk})^2}{N \cdot MTTR} \end{aligned}$$

RAID 4 and 5 Both RAID 4 and 5 fail if two disks fail before replacement, hence P is the probability that a second disk fails while restoring the failed disk. In formulas

$$P = \frac{N-1}{MTTF_{disk}} \cdot MTTR$$

where

- $\frac{N-1}{MTTF_{disk}}$ is the failure rate for one of the other disks (i.e. the probability that one of the other disks fails).
- $MTTR$ (Mean Time To Repair) is the time that elapses between failure and restore of the disk.

Putting all together the $MTTF$ of RAID 4 and 5 is

$$\begin{aligned} MTTF_{RADI4} = MTTF_{RAID5} &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{P} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{\frac{N-1}{MTTF_{disk}} \cdot MTTR} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{MTTF_{disk}}{(N-1) \cdot MTTR} \\ &= \frac{(MTTF_{disk})^2}{N \cdot (N-1) \cdot MTTR} \end{aligned}$$

RAID 6 RAID 6 tolerates two failures, thus P is the probability that, after a failure, a second and third disk fail. In formulas

$$P = P_{second} \cdot P_{third}$$

Let us analyse the one probability at a time. The probability that a second disk fails while restoring the failed disk is

$$P_{second} = \frac{N-1}{MTTF_{disk}} \cdot MTTR$$

The probability that a third disk fails is

$$P_{third} = \frac{N-2}{MTTF_{disk}} \cdot \frac{MTTR}{2}$$

where

- $\frac{N-2}{MTTF_{disk}}$ is the probability that one among the remaining $N-2$ disks fails.
- $\frac{MTTR}{2}$ is the the average overlapping period between the first and second failure (because we want the three failures to happen while restoring).

If we put all together we obtain

$$\begin{aligned} MTTF_{RAID6} &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{P} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{P_{second} \cdot P_{third}} \\ &= \frac{MTTF_{disk}}{N} \cdot \frac{1}{\frac{N-1}{MTTF_{disk}} \cdot MTTR \cdot \frac{N-2}{MTTF_{disk}} \cdot \frac{MTTR}{2}} \\ &= \frac{2(MTTF_{disk})^3}{N \cdot (N-1) \cdot (N-2) \cdot MTTR^2} \end{aligned}$$

A summary the MTTF for all RAID systems is shown in Figure 4.1.

RAID 0	$\frac{MTTF_{disk}}{N}$
RAID 1	$\frac{(MTTF_{disk})^2}{N \cdot MTTR}$
RAID 1+0	$\frac{(MTTF_{disk})^2}{N \cdot MTTR}$
RAID 0+1	$\frac{(MTTF_{disk})^2}{N \cdot G \cdot MTTR}$
RAID 4	$\frac{(MTTF_{disk})^2}{N \cdot (N-1) \cdot MTTR}$
RAID 5	$\frac{(MTTF_{disk})^2}{N \cdot (N-1) \cdot MTTR}$
RAID 6	$\frac{2(MTTF_{disk})^3}{N \cdot (N-1) \cdot (N-2) \cdot MTTR^2}$

Table 4.1: A summary of the MTTF of some RAID.

4.2.5 Storage systems

In a data centre, the storage can be directly attached to a server or accessed through the network. In particular there exists three types of technologies to connect storage and server

- **Direct Attached Storage** DAR.
- **Network Attached Storage** NAS.
- **Storage Area Network** SAN.

The first type of storage is directly attached to a server while the other two use the network, in different ways, to transfer data to the server. Figure 4.11 shows how storage is connected to the server in the three different configurations.

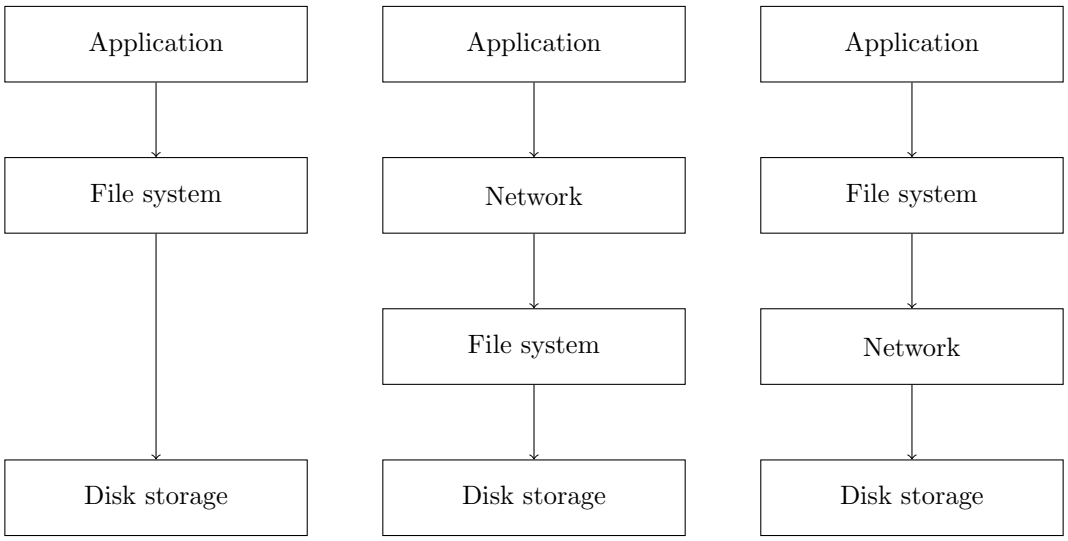


Figure 4.11: How storage is connected to the server(s) in DAS, NAS and SAN.

Direct Attached Storage

A DAS is a storage directly attached to a server. This means that the disk, whatever is the operating system of the server, is not exposed directly to the network and can be accessed exclusively by a machine (i.e. the server to which it's attached). DAS can also share data with other machines, however this operation increases the disk access time because DAS systems haven't been designed to share data between multiple servers.

Notice that even an external disks, connected with a point-to-point protocol, to a server can be considered as DAS.

Disadvantages and use in data centre The main disadvantages of a DAS are

- It has a limited scalability.
- It's hard to use it for sharing files among multiple servers.
- It has poor performance.

This means that DASes aren't well suited for data centre. In particular they are used only as caches that don't need to be shared with other machines. DAS also have some advantages, in fact

- It's very simple.
- It allows to access data quickly.

Network Attached Storage

A NAS is a storage (i.e. a dedicated server for storage) connected to a network. In particular, a NAS is made of multiple disks that can support multiple RAID configurations and exposes its filesystem that the servers can use to get data.

Network Attached Storages use the TCP/IP stack, hence every NAS has an IP address.

Characteristics and differences with DAS The main differences between NAS and DAS are

- The latter is **private storage** while the former **expose data to the network**.
- The latter generates **low or no traffic** while the former **generates a large amount of traffic** because all data passes through the network. Notice that in some case the access to the network and the amount of traffic generated can be a problem for a data centre and can increase the access latency.

Storage Area Network

In a SAN every server handles its filesystem, but the data is on a separate server that is accessed through the network. In this case, differently from NAS, the network on which data is accessed is not the same used for normal communication among the server. In other words, servers use a dedicated network for the storage. This means that a SAN has dedicated protocols, hence an higher bandwidth. A NAS also improves scalability.

SAN solves all the problems of DAS (it allows for file sharing) and NAS (it uses a dedicated network for maximum throughput) but it's very expensive to implement.

Differences with NAS Both NAS and SAN uses the network but in different ways, in particular

- A NAS appears to a server as a file server with it's own filesystem.
- A SAN appears to a server as a disk accessed through the network.

Usages NAS and SAN, given their different properties, are used in different ways,

- NAS is used for low-volume access to a large amount of storage by many users.
- SAN is used for large volumes and multiple, simultaneous access to files, such as audio/video streaming.

4.3 Networking

Networking is a fundamental part in data centres and warehouse scale computers because it connects all the server and storage parts of the DC. We should put focus on networking because when we want to increase the power or the capacity of a DC we can easily increase the number of servers or disks. For instance one can obtain double the capacity by doubling the number of disks or double the processing power by doubling the number of servers. The same isn't true for networking, in fact doubling the bandwidth doesn't doubles the performance of the whole data centre. This is due to the fact that the actual performance is limited by the bottleneck of the network, hence doubling the bandwidth somewhere else isn't going to help because the bandwidth at the bottleneck is still the same.

To handle the bottleneck of a system, we can introduce the concept of **bisection bandwidth**.

Definition 5 (Bisection bandwidth)**.** *The bisection bandwidth is the bandwidth across the narrowest line that equally divides the network into two parts.*

In other words, the bandwidth of the bisection is the sum of the bandwidth of all channels that cross the bisection. The bisection bandwidth characterises the network in terms of bottleneck because if we assume that any device has to talk with all other devices, then the middle of the network (i.e. the bisection) will experience traffic from all nodes.

4.3.1 Architectures and topologies

Three layer architecture

The most basic architecture used to build a data centre's network is called three-layer architecture and defines three layers of data aggregation,

- The **access layer**. The access layer is used by servers in a rack (i.e., the leaf nodes of a network) to access the Top Of the Rack (TOR), which is a switch that collects data coming from the server, aggregates it and sends it to the upper layers.
- The **aggregation layer**. The aggregation layer is handled by switches placed at the end of a corridor (i.e., End Of Line, EOL) and aggregates data coming from the TORs to send them to the upper layer.
- The **core layer**. The core layer represents the highest layer of the architecture and allows the data centre to communicate with the outside. In other words, it routes data coming from the aggregation layer to the outer world.

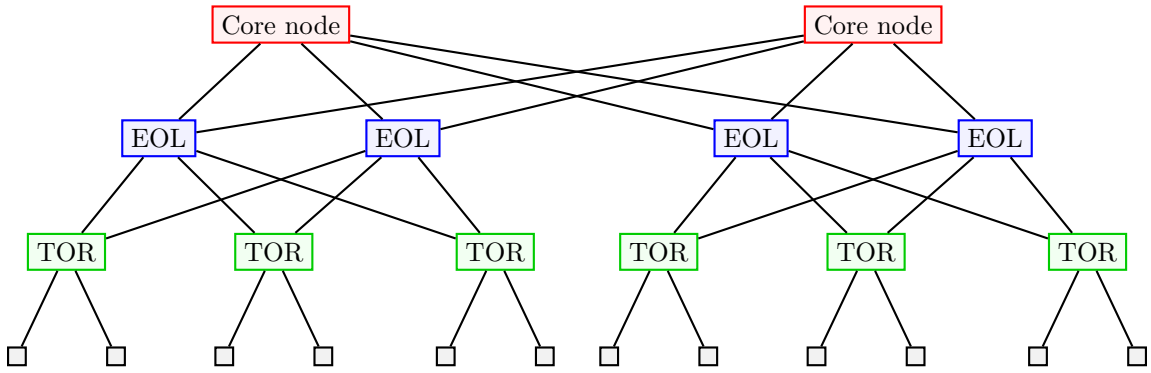


Figure 4.12: A three layer architecture. The core layer is represented in red, the aggregation level in blue and the access layer in green. Servers are represented in black.

A graphical representation of a three-layer architecture is shown in Figure 4.12. As we can see, each node of a layer is connected to more nodes of the layer above. For instance, a TOR isn't connected only to the EOL of its corridor, but also to some other EOLs. This allows the network to have multiple path for routing data. Routing is handled by the Equal Cost Multiple Path (ECMP) protocol.

Advantages and disadvantages This solution is very easy and scalable because it follows the hierarchy of the data centre. However, this architecture becomes more expensive the more nodes we add, because every level has to aggregate more paths and data, hence we need more bandwidth.

Fat three

The fat-tree topology is an evolution of the three layer topology. The network is divided into **pods** each containing the same number of switches of the aggregation and access (also called edge) layer (e.g., 2 aggregation switches and 2 edge switches). Every switch inside a pod is connected to every other switch in the pod and the aggregation layer switches are connected to the core layer. The main difference with three layer topology is that in this case we use a larger number of slower speed switches and connections.

Advantages and disadvantages The fat-tree topology is more complex and requires more complex cabling, however it allows to have an higher throughput.

D-cell topology

The D-cell topology defines the architecture of a network recursively. In particular,

- A server connected to a switch is a D-cell.
- Two or more D-cells connected one to the other are a D-cell.
- Nothing else is a D-cell.

Basically, we are defining levels of D-cells interconnected one to the other.

4.3.2 Interplay of network and storage

The interplay between network and storage is fundamental because the performance of the latter influences the former's and vice versa. In particular we can evaluate and analyse how these parts work together in terms of latency, bandwidth and capacity. For instance, the time to access data in a flash memory on a remote DC (if we consider a network of DCs) is dominated by the latency, throughput and capacity of the network. On the other hand, accessing data on an hard drive depends more on its latency, throughput and capacity. This means that disk locality is no longer relevant in inter-data center computations and that data should be stored where it's needed and where it can be accessed the fastest.

4.4 Building infrastructure

Data centres aren't made of server only, in fact, to work properly, they require other components like cooling systems, power management devices and electrical components. These components can be duplicated, too, to ensure fault tolerance (i.e., the data centre can still work after some failures) and avoid critical damages. In particular, electrical components are equipped with batteries that allow them to stay on for a small period of time after a power loss. Diesel generators can be used in case of longer black-outs, instead.

4.4.1 Cooling systems

Cooling systems are used to reduce the heat generated by the servers in a server room (or in general the data centre). Cooling systems are made of many components,

- Coolers.
- Cold water tanks.
- Heat exchangers.

Cooling systems can be classified in

- The **open-loop** cooling systems.
- The **closed-loop** cooling systems.

Open loop cooling systems

Open loop cooling systems inject cool air from outside the data centre and extract the hot air through the ceiling.

Closed loop cooling systems

In closed loop cooling systems, the hot air taken from the data centre is refrigerated and sent back into the server room to cool the servers down. An example of closed-loop is the **two-loops** cooling system. In this case, we use two loops

- The **inner loop** uses vents to inject fresh air inside the servers. The hot air coming out from the back of the servers is cooled down by a Computer Room Air Conditioning (CRAC) unit. Basically, the inner loop collects heat from the internal server room.

- The **outer loop** replaces the liquid (that has become hot) used by the CRAC units to cool the air in the inner loop. Basically, the second loop exchanges heat with the external environment.

The number of loops can be increased because two loops are in general not enough. In particular we can add a third loop that refrigerates the cooling liquid used by the CRAC units using some chiller condensers that works together with evaporating towers which extract heat from the liquid. Schematically,

- In the innermost loop, fresh air is sent to the server room and hot air is brought in heat exchangers.
- In the middle loop, the cooling liquid used for cooling the air in heat exchangers is sent to chillers that cools it and sent it back to heat exchangers.
- In the outer loop, the liquid used in chillers is sent to cooling towers that cool it down and send it back to the chillers. In particular, cooling towers evaporate part of the liquid to lower its temperature.

Notice that, cooling towers work marginally better when the temperature is low, however they need a specific temperature window (which depends on the temperature of the liquid to cool) to work at their best.

Cooling systems comparison

Each cooling system has its advantages and disadvantages,

- **Open-loop cooling systems** are **very simple** and **cheap** but they **depend on the environment and on seasons** because they need cool temperatures.
- **Two-loops cooling systems** are **simple to implement**, **not very expensive** and **can be used in whatever location**, but their **efficiency is not that good** because it widely depends on the operational efficiency of the machines.
- **Three-loops cooling systems** are **the most expensive** but offer **the best performance** in terms of isolation and efficiency. For this reason, they are widely used in many large data centres.

In rack and in row cooling

In-rack and in-row cooling uses some special equipment to cool down the air inside servers and move it from the back of the sever to the front (i.e., in the opposite direction with respect to two-loops). The cooling system is close to the server but not in it. More precisely:

- **In rack coolers** add 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, essentially reducing the path between server exhaust and CRAC input.
- **In row cooling** works like in-rack cooling except the cooling coils aren't in the rack, but adjacent to the rack.

Liquid cooling

Liquid cooling is an in-server cooling technology that allows to reduce the temperature of a server using a special liquid. The main advantage of this technology is that a liquid is much more efficient at extracting heat (i.e., at cooling down) a server with respect to air. This however comes at a cost, in fact liquid cooling systems are much more expensive than normal air systems and are harder to implement because they need capillary tubes to reach the servers. For this reason this technology is used only in extreme cases (i.e., high performances and high cooling efficiency requirements).

4.4.2 Container based data centres

Container based data centres are DCs placed inside a container that can be used as an additional component to support the operations of a bigger data centre and can be moved easily. In particular, they can be used to

- **Keep the computation close to where the data is generated**.
- **Backup remote facilities**.
- **Confine the server equipment** inside a large data centre.
- **Support the expansion of a data centre**.
- **Temporary expand a data centre** that might require more processing power for a limited period of time.

4.4.3 Availability levels

Data centres are divided in tiers, depending on their availability, each one having different requirements. The availability tiers are shown in Figure 4.13.

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

Figure 4.13: Availability tiers for a data centre.

Part III

Methods

Chapter 5

Dependability

5.1 Introduction

Dependability measures how much we can trust a system, whatever the size of the system we are considering. In particular the concept of trust can be considered under five different aspects,

- **Reliability**. Reliability measure the continuity of a service, i.e., how much the service can run without failing.
- **Availability**. Availability is the probability that a service can be used when it's requested.
- **Maintainability**. Maintainability measures the easiness of fixing a server when it stops working or when it needs some updates.
- **Safety**. Safety checks that the system can't harm humans.
- **Security**. Security controls the confidentiality and integrity of data.

Dependability is complementary to functional verification, in fact

- Functional verification checks that all functional prerequisites and constraints of the system are satisfied.
- Dependability focuses on what happens when something goes wrong.

When a failure happens, the system should be able to keep running, probably with some performance limitation, without exposing the failure to the user (i.e., without the user noticing something went wrong). If the system fails completely, the company that offers the service might lose money, data and time to recover data (if the system is redundant).

5.1.1 When to consider dependability

Dependability should be considered during the whole life of a system, i.e. both at design-time and run-time. In particular,

- A system should be **designed** to be **robust** and always thinking that failures can occur, either in hardware and software. An important thing to keep in mind is that, when designing a system we should assume that the result of a failure is deterministic, namely that we know

what happens when a failure occurs. This allows us to properly recover from a failure and go back to a valid state.

- Some problems can't be detected at design-time and only verify at **run-time**. For this reason we need to include in the system some detection models to recognise failures and react to them. Moreover, we should also include some mechanisms to understand the source and the cause of the failure.

5.1.2 Where to consider dependability

It's not mandatory to consider dependability for all systems or for all parts of a system. We can say that

- We should always consider dependability in mission and safety-critical systems (more on that later).
- We can care less about dependability in cheap systems since it might be more convenient to buy new devices when one fails.

Mission critical systems

Mission-critical systems are systems that should be working up to the completion of their mission. In other words, in mission-critical systems we don't consider dependability after the mission and we focus on the mission only. An example of mission-critical system is a satellite because it should work, without failures, until its operational time is over.

Safety critical systems

Safety-critical systems are systems in which a failure doesn't only stop the mission, but it's also dangerous for humans.

Data centres

If we consider data centres, dependability can be applied at different levels, depending on the type of data centre. In particular, in third party data centres, the owner of the service that runs in the DC has to consider dependability at the application level because it can't control the lower layer (the company rents some hardware from the DC's owner). On the other hand, in a proprietary data centre, the owner can control the dependability at all levels, since it owns both the hardware and the software.

5.1.3 Tolerance and avoidance

Faults can be both tolerated or avoided, let as understand what are the differences between these two cases.

Fault tolerance

Fault tolerance can be achieved through

- **Error detection** and **error masking** during system operation.

- **Online monitoring**.
- **Diagnostics**.
- **Self-recovery**, **self-repair** and **self-testing** (mainly in software).

Fault avoidance

Fault avoidance can be achieved through

- **A conservative design**.
- **Design validation**.
- **Detailed test of hardware and software**.
- **Infant mortality screen**.
- **Error avoidance**.

Improving fault tolerance and avoidance

To improve fault tolerance and avoidance we can work at three different levels,

- The **technological level**. At the technological level we can improve tolerance and avoidance using more reliable components. Such components are usually older than the brand-new devices, because they have been tested more and their issues have been solved. Working at this level gives the best improvement in term of dependability, however we get **high costs** (because we might use top-level technologies) or **low performance** (because the devices used are older, hence less powerful).
- The **architectural level**. At the architectural level we can use **redundancy** and **majority-vote schemes** to increase dependability. Notice that in this case, the improvement is less than the one obtained at the technological level, and it comes at **high cost** (because we need more devices) or it **worsen the performance of the system** (because voting costs time and slows down the execution).
- The **software level**. At the software level we can deploy a service on multiple servers distributed in different geographical areas to improve dependability. However, even in this case, either the **cost of the system increases** or the **performance decreases**.

Given the analysis above, we can conclude that improving the dependability of a system is a trade-off between costs and performance, hence the best solution for a system depends on the scenario in which it's working.

5.2 Reliability and availability

5.2.1 Reliability

The standard IEEE610 defines reliability as

Definition 6 (Reliability)**.** *The ability of a system or component to perform its required functions under stated conditions for a specified period of time.*

Basically, the idea is that the reliability $R(t)$ is the probability that, given a time instant t, the system doesn't fail in a time interval $[0, t]$ (assuming the system started working at time 0).

$$R(t) = P(\text{system does not fail in } [0, t])$$

Form the definition we can notice that time t plays an important role in reliability and that reliability focuses on the first failure of the system. For this reason, reliability is usually considered in non-reparable systems. Let us consider some values of t to understand how the reliability function changes.

- For $t = 0$ the reliability is 1 because the system is working (by definition) at $t = 0$.

$$R(0) = 1$$

- For $t \simeq 0$ (i.e. time instant close to 0) the reliability is close to 1 because the probability that a system fails just after it started working is very low.

$$R(t \simeq 0) \simeq 1$$

- The reliability decreases as the time increases and it asymptotically reaches the value of 0 when $t = \infty$ because, the more time passes, the more probable is a failure to happen.

$$R(\infty) \to 0$$

From the aforementioned considerations, we say that the reliability function behaves like in Figure 5.1.

Unreliability

The unreliability function $Q(t)$, the complementary of the reliability function, is the probability that at least a failure happens in the interval $[0, t]$.

$$Q(t) = 1 - R(t)$$

5.2.2 Availability

The availability is the characteristic of a component to work when it requires to be used. More precisely,

Definition 7 (Availability)**.** *The availability is the probability to find the system properly working.*

The availability can be computed as the ratio between the up-time t_{up} (i.e., the mean time to failure $MTTF$) and the total working time, namely the sum of up-time t_{up} and down-time t_{down} (i.e., the sum of mean time to failure $MTTF$ and mean time to repair $MTTR$).

$$A = P(\text{find the system properly working}) = \frac{t_{up}}{t_{up} + t_{down}} = \frac{MTTF}{MTTF + MTTR} \tag{5.1}$$

Differently from the reliability, the availability doesn't depend on the time t in which we compute it. For this reason, the availability is used in reparable systems in which multiple failures can happen during the lifetime of the system. More on the $MTTF$ and $MTTR$ will be explained later on.

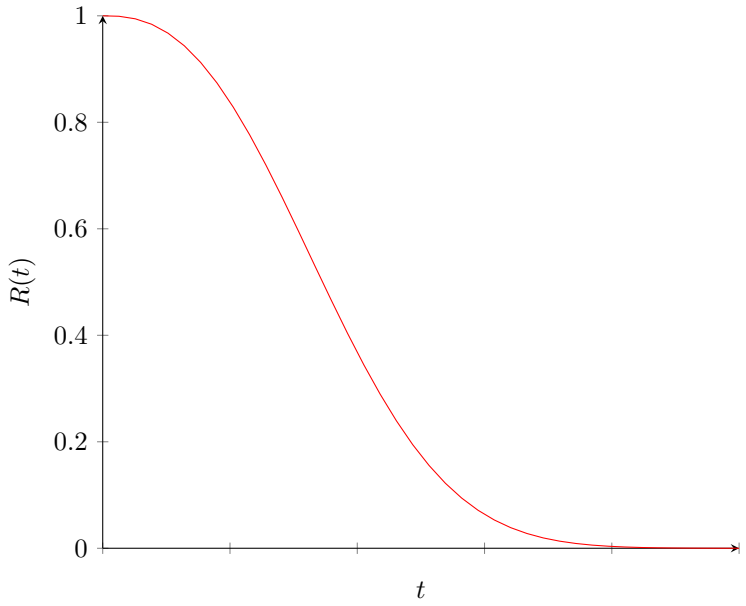


Figure 5.1: A qualitative representation of a reliability function's shape.

Unavailability

As for the reliability, we can define the unavailability as $1 - A(t)$.

Measuring availability

The availability of a system can be measured in number of nines present in A. For instance if $A = 0.9$ we have 1 nine, for $A = 0.99$ we have 2 nines, for $A = 0.999$ we have 3 nines and so on. Usually we consider values from 2 nines (i.e. 0.99, 99% of up-time, which is less than 4 days) to 6 nines (i.e., 0.999999, 99.9999% of up-time, which is around 30 minutes). Values lower than 2 nines are considered unacceptable while values higher than 6 nines are considered impossible to reach. A summary of the down-time for different levels of nines is shown in Table 5.1.

Number of nines	availability	down-time (min/year)
1	0.9 (90 %)	52596.00 (around 5 weeks per year)
2	0.99 (99 %)	5259.60 (around 4 days per year)
3	0.999 (99.9 %)	525.96 (around 9 hours per year)
4	0.9999 (99.99 %)	52.60 (around 1 hour per year)
5	0.99999 (99.999 %)	5.26 (around 5 minutes per year)
6	0.999999 (99.9999 %)	0.53 (around 30 secs per year)
7	0.9999999 (99.99999 %)	0.05 (around 3 secs per year)

Table 5.1: Different levels of availability.

5.2.3 Relation between reliability and availability

Reliability and availability are related.

Non reliable systems

If we consider a not repairable system, since we can have only a failure, the availability corresponds to the reliability.

$$A_{non\ reparable}(t) = R_{non\ reparable}(t)$$

Reparable systems

In reparable systems, the availability is bigger then the reliability.

$$A_{reparable}(t) \geq R_{reparable}(t)$$

In other words we can say that

- There exists systems that have a low reliability but an high availability.
- It's almost impossible to build systems with low availability and high reliability.

Mean Time Between Failures

The Mean Time Between Failures (MTBF) is the time that, on average, elapses between two failures. For this reason, the MTBF is related to the availability. It's computed as the ratio between the total operational time $t_{total\ operational}$ and the number of failures $N_{failures}$

$$MTBF = \frac{t_{total\ operational}}{N_{failures}}$$

The MTBF can also be computed using the failure rate λ, which is the ratio between the number of failures and the total operational time.

$$\lambda = \frac{N_{failures}}{t_{total\ operational}}$$

The MTBF can therefore be computed as 1 over the failure rate λ.

$$MTBF = \frac{1}{\lambda}$$

If we plot the failure rate over time (Figure 5.2) we get a very interesting result. In particular, we can recognise three different areas,

- The area on the left (i.e., the one where t has small values) has an high failure rate that decreases over time. This area is called **infant mortality** because in hardware and software it's easy to spot bugs when the system is at the beginning of its life. Usually, the developer of a system doesn't deploy the system until this phase is over because the failure rate is very high.
- The area in the middle has an almost constant, low-value failure rate. This is the **ideal working zone** because the failure rate is low and the failures are random.

- The area on the right (i.e., for big values of t) has an high failure rate that increases for increasing time instants. This area is called **degradation period** since the failure rate increases because of the degradation of the components (especially mechanical ones). Usually, companies decide to replace a component when it enters the degradation period because it's better to pay for replacing a working component before the end of its life than paying to recover from a failure. This concept is called **predictive maintainment** and it's implemented with artificial intelligence techniques.

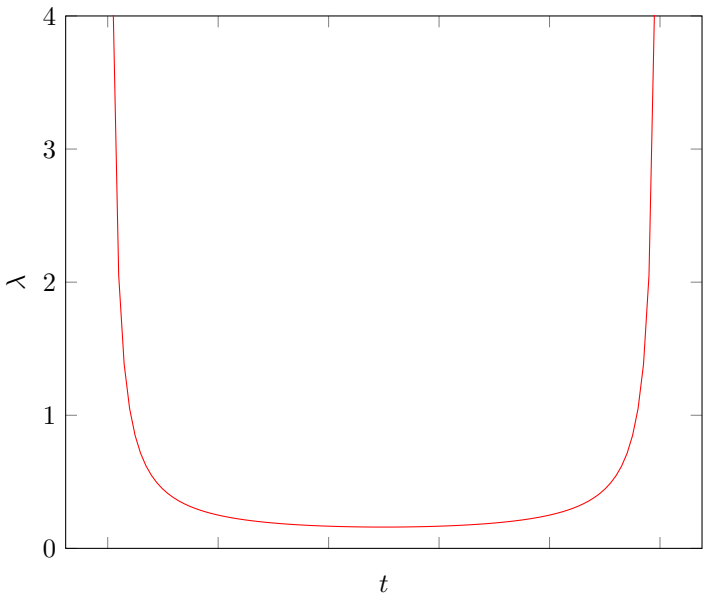


Figure 5.2: Failure rate.

Mean Time To Failure

The Mean Time To Failure (MTTF) is the time that elapses before the first failure of the system. For this reason, the MTTF is related to the reliability and it's computed as the integral of the reliability

$$MTTF = \int R(t)\ dt \tag{5.2}$$

If we consider only the working zone in the failure rate plot, the reliability function can be described by an exponential function

$$R(t) = e^{-\lambda t}$$

then the MTTF is

$$MTTF = \int_0^\infty R(t)\ dt = \int_0^\infty e^{-\lambda t}\ dt = \frac{1}{\lambda}$$

5.2.4 Evaluation of reliability

To compute the reliability of a component we can take many identical instances of that component and make them run in parallel. When a component fails, we have to log the time at which it failed (i.e., the mean time to failure) and after every component has failed we can average the time to failure of each component (because we are computing the mean time to failure).

This technique has a problem, in fact if we have to evaluate the reliability of a component we might have to wait a lot of time, hence we have to find ways to speed up this process. For instance, we use the same procedure aforementioned, putting the components under a lot of stress, to speed-up the component degradation. Basically, we are trying to move the procedure on a new time scale (we can call it speed-up time). Once we get the time to failure of each component (which is smaller than the time to failure measured with the normal time), we have to map the values obtained in the normal time scale. This operation can be done considering some default formulas that depend on the type and level of stress under which the components are working. In other words, these formulas allow to go from $t_{speed-up}$ to t_{normal}.

5.3 Faults

One important thing to clarify is the difference between faults, errors and failures,

- A **fault** is a defect in the system (e.g., a software bug or an electronic malfunction).
- An **error** is a deviation from the required behaviour of the system (e.g., a wrong computation).
- A **failure** is a deviation from the actual behaviour of the system.

A fault can generate and error which in turns can generate a failure, however this sequence can be stopped. In particular a fault might not generate an error if the system is able to handle the fault, for instance, using redundancy. The same is true for failures, in fact, an error might not generate a failure if, for instance, the software can handle it. An important thing to underline is that, an error, even if it generates a failure, might not be critical if it happens in a non critical part of the system. This means that we should put the focus (in the first place) on the critical parts of the system when handling errors.

5.3.1 Classification of faults

. Faults in a data centre (or in a computer) can be divided in,

- **Electronic** faults.
- **Mechanical** faults.
- **Software** faults.

Each type of fault is characterised by a different failure rate curve.

Electronic faults

The failure rate curve of electronic faults is almost identical to the one seen for general faults (Figure 5.2). It's important to underline that, usually a component (e.g., a CPU) becomes obsolete (i.e., it's replaced) before reaching the degradation part. This means that we don't have to apply predictive maintainment.

Mechanical faults

The failure rate curve of mechanical faults has

- A much sorter infant mortality period in which the failure rate is much lower than the general case.
- A constantly increasing failure rate in the working zone because mechanical components usually degrade much faster with respect to electrical components. Notice that in the general case, the working zone is characterised by a constant, low-value failure rate.
- A degradation phase that behaves like in the general case.

For mechanical faults, it's hard to distinguish the working zone from the degradation phase because in the former isn't constant.

Software faults

The failure rate curve for software faults is characterised by

- An infant mortality phase which is much more acute than the general case because software is usually full of bugs when it's released.
- A decreasing failure rate in the working zone because software faults (i.e., bugs) are corrected with time.
- No degradation phase because software faults are bugs in the code, which are corrected hence the failure rate doesn't increase.

This characterisation doesn't take into account the fact that a program is usually upgraded and new features are added, hence new bugs can be introduced. The new characterisation, that takes into account updates, has spikes corresponding to the instants in which a feature is added (i.e., when a bug might have been introduced). Still, the failure rate is still a decreasing function.

5.4 Reliability block diagrams

Reliability blocks diagrams allow to compute the reliability (and the availability) of a system composed of more components. A reliability block diagram represents the logical connections (from the reliability point of view) between the components of the system. Notice that the logical connection is different from the architecture of the system. For instance two components could be not connected at the architectural level but be connected at the logical level because one depend on the other.

Each block in the diagram represents a component of the system and blocks can primarily connected in two ways,

- In **series**.
- In **parallel**.

A system works properly if we can find a path from the beginning of the diagram to the end.

When computing the reliability of a system we will assume that the reliability of a block doesn't depend on the reliability of the other blocks.

5.4.1 Series of blocks

Two blocks are connected in series if they have to work at the same time for the system to work properly.

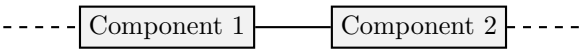


Figure 5.3: A series of two reliability blocks.

Reliability

The reliability of a series of N blocks can be computed as the product of the reliabilities of each block

$$R_{series}(t) = \prod_{i=1}^{N} R_i(t)$$

If we consider components whose reliability has an exponential distribution, then the reliability of the system is

$$R_{series}(t) = \prod_{i=1}^{N} R_i(t) = \prod_{i=1}^{N} e^{-\lambda_i t} = e^{-\sum_{i=0}^{N} \lambda_i t}$$

The reliability can now be used to compute the mean time to failure of the system

$$\begin{aligned} MTTF_{series} &= \int_0^\infty R_{series}(t)\ dt = \int_0^\infty e^{-\sum_{i=0}^{N} \lambda_i t}\ dt = \int_0^\infty e^{-\lambda_{series} t}\ dt \\ &= \frac{1}{\lambda_{series}} = \frac{1}{\sum_{i=0}^{N} \lambda_i} = \frac{1}{\sum_{i=0}^{N} \frac{1}{MTTF_i}} \end{aligned}$$

If we consider the case in which all components have the same failure rate λ, then the failure rate λ_{sys} of the system can be written as $N\lambda$ and the reliability becomes

$$R_{sys}(t) = \prod_{i=1}^{N} e^{-\lambda t} = e^{-N\lambda t}$$

which can be used to compute the MTTF as

$$MTTF_{series} = \int_0^\infty R_{series}(t)\ dt = \int_0^\infty e^{-N\lambda t}\ dt = \frac{1}{N\lambda} = \frac{1}{\frac{N}{MTTF}} = \frac{MTTF}{N}$$

where $MTTF$ is the mean time to failure (equal for each component).

Availability

The availability of a series of N blocks can be computed as the product of the availability of the single blocks (as we have done for the reliability).

$$A_{series}(t) = \prod_{i=1}^{N} A_i(t) = \prod_{i=1}^{N} \frac{MTTF_i}{MTTF_i + MTTR_i}$$

5.4.2 Parallel of blocks

Two blocks are connected in parallel if at least one of the two has to work, for the system to work properly.

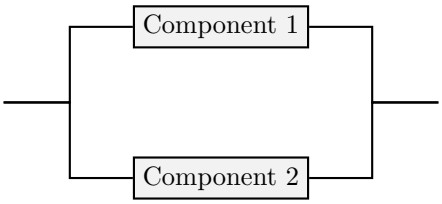


Figure 5.4: A parallel of two reliability blocks.

Reliability

To compute the reliability of a parallel of N blocks we can use the unreliability. In particular, we can say that a parallel of component is not reliable if all blocks fail. The reliability can therefore be computed as the product of the unreliabilities of the single components

$$U_{parallel}(t) = \prod_{i=1}^{N} U_i(t) = \prod_{i=1}^{N} (1 - R_i(t))$$

Now we can say that, the reliability is 1 minus the unreliability, hence we obtain

$$R_{parallel}(t) = 1 - U_{sys}(t) = 1 - \prod_{i=1}^{N} (1 - R_i(t))$$

Availability

The availability of a parallel of N blocks can be computed using the same formula used for the reliability.

$$A_{parallel}(t) = 1 - \prod_{i=1}^{N} \big(1 - A_i(t)\big) = 1 - \prod_{i=1}^{N} \left(1 - \frac{MTTF_i}{MTTF_i + MTTR_i}\right)$$

5.4.3 R out of N

In some cases, more complex blocks are required to model a system. Consider for instance a majority voting scheme. In this case, the majority of components has to agree on a value to increase the reliability of the system. This behaviour can be modelled with a R out of N block (Roo$N$). In particular, a Roo$N$ blocks models a system composed of N identical replicas where at least R replicas have to work fine for the entire system to work properly.

Reliability

The reliability of a Roo$N$ block can be computed as

$$R_{RooN} = R_{voter} \cdot \sum_{i=R}^{N} R^i (1 - R)^{N-i} \cdot \frac{N!}{i!(N-i)!}$$

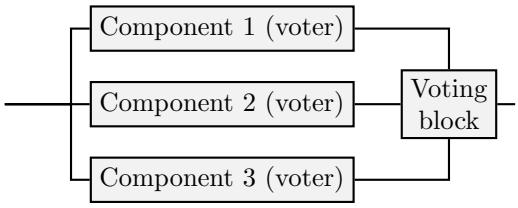


Figure 5.5: A 2oo3 block.

where

- R_{voter} is the reliability of the voter (i.e., of the component that handles voting).
- R is the reliability of the voting components.
- $\frac{N!}{i!(N-i)!}$ is the binomial coefficient that considers all possible combinations in which i components can work.

Let's try to understand where this formula comes from. The reliability is the series between the voter component and the voting components. In particular the second term (the one referring to the voting components) considers all possible ways in which the components can vote starting from the minimum required (i.e., R components did not fail) and going to the best situation in which all components are not faulty $R = N$.

Time to failure-reliability trade-off

If we compute the reliability and the MTTF for a simple 2oo3 block, we notice that the mean time to failure is actually smaller than the MTTF of each single component. However, the reliability of the 2oo3 is higher. In particular, we can notice that the reliability of 2oo3 component is higher until a certain point in time in which, due to the degradation of the components, it's better to have a single component. If we equal the reliability of a 2oo3 block and the reliability of a single component we also get the exact crossover time, which turns out to be $0.7MTTF$. To obtain this result we can equal the reliability of a 2oo3 component and the reliability of a single component.

$$
\begin{aligned}
R_{2oo3} &= R_{single} \\
R_{voter} \cdot \left(R_{single}^3 + 3R_{single}^2(1 - R_{single}) \right) &= R_{single} \\
R_{single}^3 + 3R_{single}^2(1 - R_{single}) &= R_{single} \\
e^{-3\lambda t} + 3e^{-2\lambda t}(1 - e^{-\lambda t}) &= e^{-\lambda t} \\
e^{-3\lambda t} + 3e^{-2\lambda t} - 3e^{-3\lambda t} &= e^{-\lambda t} \\
3e^{-2\lambda t} - 2e^{-3\lambda t} &= e^{-\lambda t} \\
3e^{-2\lambda t} - 2e^{-3\lambda t} - e^{-\lambda t} &= 0 \\
e^{-2\lambda t} - \frac{3}{2}e^{-\lambda t} + \frac{1}{2} &= 0
\end{aligned}
$$

What we get is an equation can be solved replacing $\tau = e^{-\lambda t}$ to obtain

$$t = \frac{\ln 2}{\lambda_{single}} \simeq 0.7MTTF$$

This means that at 70 % of the MTTF, a 2oo3 block becomes less reliable than a single block, hence the majority voting mechanism isn't useful anymore.

5.4.4 Standby block

A standby block is used to represents two components in which one works like the primary component and handles all the workload, while the other component, which is the secondary component only gets activated when the primary component stops working. This architecture requires,

- A component that can detect the failure of the primary component.
- A switching mechanism to reroute traffic from the primary to the secondary component, when the former fails.

Reliability

The reliability of a standby block is always higher than the reliability of a single component of the block.

5.5 Applying redundancy

Given a certain system, we can apply redundancy it in two differ ways,

- We can duplicate the whole system in parallel.
- Duplicate every single component of the system (apart from those which are already duplicated in the original system).

Which solution is best depends on the system and the only way to understand it is to compute the reliability in the two cases and compare the values.

Chapter 6

Performance modelling

6.1 Introduction

Performance and scalability are important characteristics of a data centre, because customers can stop using a service due to bad performance, hence leading to a money loss.

When we consider performance it's important to distinguish between

- **Open environments**. In open environments, the number of user connected to the data centre is unknown.
- **Closed environments**. In closed environments, the number of user connected to the data centre is precisely known (e.g., a data centre used by the workers in a company).

Depending on the type on environment we can do different assumptions, especially regarding scalability (i.e., in the former case we have to be more careful). Moreover, the concept of average (used for instance to compute the average number of accesses per seconds) is very different.

6.1.1 Data centre scaling

When building a data centre we should understand how big it should be to handle a certain number of requests in a certain time interval. Basically, we want to dimension the DC in order to successfully handle the service offered to a number (estimated) of clients. The easiest way to solve this problem is to add as many server as possible. Unfortunately, servers cost money, hence we have to consider a trade-off between the number of servers and the impact on the customer of such servers. Knowing the impact of the number of servers, or of the type of servers on the client is fundamental because this information allows us to evaluate the sever-performance trade-off and decide how many servers and of what type should be used.

To evaluate the impact of the number and type of servers on performance, we can use three techniques,

- **Intuition and experience**.
- **Experimental evaluation of alternatives**.
- **Modelling**.

Intuition and experience

Intuition and experience allows us to evaluate the performance of a data centre based on those previously seen. The main advantages of this technique are

- It's **fast, easy to apply and flexible**.
- It **works on a data centre never seen before**.

However, the main disadvantage is that

- It's **not accurate**. In particular, the furthest the data centre is from the ones previously analysed, the less accurate the prediction is. For instance if one is used to working on small data centres, it's hard to predict the behaviour of a large number of servers.

Experimental evaluation of alternatives

Experimental evaluation of alternatives allows us to practically test different configurations on a real data centre to practically evaluate their performance. The basic idea of this technique is to enumerate all the promising configurations of the data centre and run the same set of tests on all configurations to evaluate and compare the performance. The main disadvantages of this technique are,

- It's **laborious**, in fact it requires to practically recreate every configuration to test.
- It's **not flexible**, in fact every data centre has its set of configurations.
- It **requires to have access to the actual machines** to test (even for a long period of time). This also means that this technique can be **quite expensive**.
- The **experimentation is valid only if the application on which the configurations are tested is the one that will be deployed**. For instance, say that we evaluate the configuration on an early-alpha. When the final application is ready, the chosen configuration might have worst performance than another configuration discarded because it didn't perform well with the early-alpha.
- We **can't experiment on hardware that is not ready**. Say, for instance, that we want to evaluate the impact of a processor that will be released in two years (but of which we know the specs). Since the processor hasn't been produced yet, we can't run simulations on it, hence we can't evaluate it.

However, experimental evaluation

- Has a **very high accuracy**.

Modelling

Modelling allows to build abstractions of the systems to evaluate. Models are able to extract some important features (i.e., those needed to model the systems' behaviour) of the systems. The main advantage of modelling is that

- It **works even if the hardware isn't available**.

however, the main disadvantage is that

- Models can be **hard to build**.

Another important characteristic of modelling is that it's very flexible. This is both an advantage and a disadvantage, in fact modelling allows to represents a broad spectrum of situations, which means that we have to be careful in modelling correctly the system, otherwise we will obtain an evaluation which is far from the true one. Models are broadly used at design time because they can represent something which hasn't been built yet.

6.1.2 Quality evaluation techniques

The techniques used to evaluate the quality of a system can be divided into

- **Measurement-based techniques**. Measurement-based techniques can be in turn classified in
  - **Direct measurement**.
  - **Benchmarking**. Benchmarking evaluates the performance of a system whose hardware is ready but the software isn't. In particular, a set of standard programs is run on the hardware to test its capabilities and performance.
  - **Prototyping**. Prototyping evaluates a system in which the hardware is not ready yet. In particular, we design some hardware which has similar characteristics to the final system to approximate the performance of the final hardware. Usually a prototype has lower specs (e.g., a lower processing power or frequency) than the final product. This means that the performance measurement obtained through prototyping should be scaled up to be correctly evaluated.
- **Model-based techniques**. Model-based techniques can be, in turn, classified in
  - **Analytical and numerical techniques**. Analytical and numerical techniques use equations to model the system, where variables are characteristics or parameters of the system. The equations are then used to extrapolate the metrics of the system. These techniques are efficient, precise and fast but are available only in few cases.
  - **Simulation techniques**. Simulation techniques allow to use a software to simulate the actual system we want to analyse to verify its performance. The simulation can be either coarse or fine grained, depending on the level of detail with which the system (and its behaviour) is modelled (i.e., we can model general components or each device in the detail). Remember that, the more detailed the description is, the more time it will take to simulate the execution of a program and the more accurate the result is. Simulations are more general than analytical techniques.
  - **Hybrid techniques**. Hybrid techniques model some components using equations and others using simulations.

6.2 Queuing network modelling

Among all evaluation techniques, we will focus on modelling. In particular, we will consider the **queuing network model**. A queuing network is a model that can be used to model a system and evaluate its performance. In a queuing network, the resources of a system are represented as service centres and the transactions (i.e., a component asks another component to do something) are represented as requests (or customers). A service centre is made of

- A **queue** (also called buffer). A queue contains the requested received by the component but not served yet (i.e., the pending requests).
- A **service station** (also called server). Service stations process a limited number of requests. The requests to execute are taken from the queue.

A graphical representation of a service centre is shown in Figure 6.1.

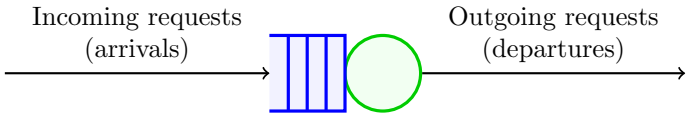


Figure 6.1: A service centre with its service station (in green) and its queue (in blue).

Queuing networks are very simple to use and can be used without many details on how a component is implemented internally, yet they can be used to model even complex systems (like a data centre).

6.2.1 Characterisation of a queuing networks

In queuing networks we have to characterise

- The **arrival**.
- The **service**.
- The **queue**.
- The **population**.

Arrival

The arrival characterisation describes how jobs arrive in the service centre. In particular, the arrival is expressed as the number of jobs that enter the queue in an interval of time and it's called **arrival rate**. In addition to the arrival rate, we should also consider how the arrivals reach a service centre, in fact requests can arrive at a constant rate (easier to handle) or in bursts. An arrival can come

- From the outside.
- From other queues (i.e., it can be the output of a service station).
- From the same component (i.e., the output of a component is reused as input of the same component). In this case we talk about **loop-back arcs**.

Service

The service is the time that a request spends inside a service station, or better said, the time needed by a service station to serve a job. The time needed to serve a job can be

- **Deterministic**. Each request is served in the same amount of time.
- **Probabilistic**. We have a probability distribution over the time needed to serve a request.

Service stations can come in different configurations,

- In the **single server** configuration, a service centre has only one server that can therefor serve one job at a time, hence all requests are executed sequentially. Once a job is finished, the next job to execute is drawn from the queue with a custom policy (i.e., not necessarily from the head of the queue).
- In the **infinite server** configuration, a service centre has as many servers as the maximum number of requests in the system. This means that when a requests arrives, it's executed immediately because there always is a free service station. Namely, the service centre has no queue and all requests can be served in parallel. An infinite server can be seen as a delay component because every request is executed immediately end ends its execution with some delay.
- In the **multiple server** configuration, a service centre has C servers, each of which can serve a request at a time. Since C is fixed and smaller than the maximum number of requests of the system, this configuration needs a single queue where to store the requests that can't be served yet.

Queue

The queue is the part before the server and it's used to store the requests that can't be served by the server (or servers). When describing a queue we have to define which process should be drawn to be served, namely, the **queuing policy** or **service discipline**.

Population

The population is the set of requests that have to be served by the components of the network. The population can be

- **Uniform**. All the request are of the same type.
- **Divided in classes**. A class is a subset of population in which requests require a different type of processing (like some sort of Quality of Service in terms of processing time, paths to follow in the network).

6.2.2 Queuing networks

A complex system can be represented as a queuing network in which each subsystem is represented as a service centre. When designing a system, we can decide the level of detail of each subsystem. For instance one might represent the full architecture of a processor to evaluate how a change in the processor or in the memory impacts on the whole system. On the other hand, one might model a processor as an high level subsystem, without going into its implementation details. The choice depends on the system and on the metric we have to evaluate.

A queuing network isn't only the set of service centres but also the paths that connect the service centres together with the routing direction and the information about the direction that a packet should take in a branch. Queuing networks can be divided in:

- **Open networks**. In open networks, the customers (i.e., the requests) come from the outside and depart from the network.

- **Closed networks**. In closed networks, all customers (i.e., the requests) are already inside the network and never leave the network.
- **Mixed networks**. In mixed networks, some classes of population are kept inside the network (closed network) and others can enter and leave the network (open network).

Routing

When designing a queuing network, we have to consider routing. In particular, a request can sometimes take multiple routes, hence we have to define what path of a branch a request should take. Some routing policies are:

- **Probabilistic**. With the probabilistic policy, each path has a probability of being chosen by a request.
- **Round-robin**. With the round-robin policy, paths are chosen in order one after the other.
- **Join the shortest queue**. With the join-the-shortest-queue policy, a request is sent the path with the smallest number of jobs waiting to be served.

6.2.3 Solving queue networks

Queue networks can be solved:

- In an **analytical way**. Solving a network in an analytical way is harder for non-probabilistic routing.
- Using **simulations**. Simulations make it easier (with respect to the analytical solution) to solve a network with non-probabilistic route policies.

6.3 Operational laws

Operational laws are simple equations that can be used to model a system. Operational laws consider the average case, hence they are

- **Generic rules**.
- **Not based on particular assumptions**.

For these reasons, operational laws give only rules of thumb to model and evaluate the performance of the system.

Operational laws are based on some **observable variables**, whose values are detected observing the system. This means that, to evaluate the performance of a system, we have to continuously monitor it and combine the information obtained to extract knowledge. The variables used in the operational laws (i.e., the observable properties of the system) are

- The **time** t at which the **observation starts**.
- The **time** T **in which the system has been observed**.
- The **number of job arrivals** A.

- The **number of job completions** C.
- The **time** B in which the **system is busy**. B is always smaller than T.
- The **average number** H **of jobs** handled by the system.

With these variables we can write

- The **arrival rate** as the number of arrivals A in the up-time T.

$$\lambda = \frac{A}{T} \tag{6.1}$$

- The **completion rate**, also called **throughput**, as the number of completions C in the up-time T.

$$X = \frac{C}{T} \tag{6.2}$$

- The **utilisation** U as the ratio between time B in which the systems is busy and the up-time T.

$$U = \frac{B}{T} \tag{6.3}$$

- The **mean service time** S **per completed job**.

$$S = \frac{B}{C} \tag{6.4}$$

6.3.1 Job flow balance

The job flow balance imposes that the number of arrivals A has to be equal to the number of completions C.

$$A = C \tag{6.5}$$

When this equation doesn't hold,

- If the number of arrivals is bigger (i.e., $A > C$), the system is accumulating requests (i.e., the system is undersized).
- If the number of completions is bigger (i.e., $C > A$), the system is serving more jobs than it has to. In other words, the system is oversized.

If the job flow balance holds, then the arrival rate λ is equal to the completion rate X

$$\begin{aligned} \lambda &= X \\ \frac{A}{T} &= \frac{C}{T} \\ A &= C \end{aligned} \tag{6.6}$$

To prove this equation correct, we can simply obtain A and C from the arrival rate ($A = \lambda T$) and the completion rate ($C = XT$).

Operational laws in a complex system

The job flow balance (6.5) and the operational laws are valid not only for the system as a whole but also for every component k of the system, hence we can write C_k, B_k, and so on. In particular, we can write

- The **subsystem arrival rate** as
$$\lambda_k = \frac{A_k}{T} \tag{6.7}$$
- The **subsystem completion rate**, also called **throughput**, as
$$X_k = \frac{C_k}{T} \tag{6.8}$$
- The **subsystem utilisation** U as
$$U_k = \frac{B_k}{T} \tag{6.9}$$
- The **subsystem mean service time** S_k **per completed job**.
$$S_k = \frac{B_k}{C_k} \tag{6.10}$$

6.3.2 Utilisation law

Given a component k, we can use the throughput X_k to derive the utilisation U_k using only the throughput and the mean service time S_k (6.4). Starting from

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

we can obtain B_k from the mean service time S_k and replace it in U_k

$$U_k = \frac{B_k}{T}$$
$$U_k = \frac{S_k C_k}{T}$$

Now we can derive the number of completions C_k from the completion rate X_k (6.2) to obtain

$$U_k = \frac{B_k}{T}$$
$$U_k = \frac{S_k \cdot C_k}{T}$$
$$U_k = \frac{S_k \cdot X_k \cdot T}{T}$$
$$U_k = S_k \cdot X_k$$

The equation

$$U_k = S_k \cdot X_k \tag{6.11}$$

just obtained is called **utilisation law** and allows us to compute the utilisation U_k of a component k without monitoring it, because the equation doesn't contain neither the time T not the busy time B_k.

6.3.3 Little law

The little law links the number of requests in a system (or in a subsystem) N and the throughput of the system. The little law states that if the system throughput is X, and each request remains in the system on average for R seconds (i.e., a request requires R seconds to be executed), then for each unit of time, we can observe on average $X \cdot R$ requests in the system

$$N = X \cdot R \tag{6.12}$$

where

- N is the number of requests in the system.
- X is the throughput of the system.
- R is the time to complete a request, also called **residence time**.

To obtain the little law, we can start by saying that the average number N of requests in the system can be written as the ratio

$$N = \frac{W}{T}$$

where

- W is the accumulated time in system, in jobs per seconds (jobs $\cdot\, s$). The accumulated time is the number of requests, times the time window in which we are observing the system (if there are 3 requests in the system during a 2 second period, then W is 6).
- The total time T.

Now we can use the **residence time** R per request (i.e., the time that a request spends in the system),

$$R = \frac{W}{C}$$

for computing $W = R \cdot C$. W can now be replaced in the equation of N to obtain

$$N = \frac{W}{T}$$
$$N = \frac{R \cdot C}{T}$$

Finally, the number of completed requests C can be written as $X \cdot T$ (from $X = \frac{C}{T}$), which replaced in the equation N results in

$$N = \frac{W}{T}$$
$$N = \frac{R \cdot C}{T}$$
$$N = \frac{R \cdot X \cdot T}{T}$$
$$N = R \cdot X$$

Little law at different levels

The little law can be applied at different levels of a system.

Unqueued disk level The little law can be applied to most basic building blocks of a system, like to a disk, without considering it's queue. In this case

- N_{disk} represents the percentage of time in which the disk is busy, so it corresponds to U_{disk}.
- R_{disk} represents the requests average service time.
- X_{disk} represents the rate of serving requests.

Queued disk level If we also consider the queue of a disk,

- $N_{disk,queue}$ is the number of users (waiting or in service) in the service center.
- $R_{disk,queue}$ is the time spent in the service center, namely the sum of waiting time and service time.
- $X_{disk,queue}$ is the throughput of the disk and corresponds to X_{disk}.

Processor level A more more complex subsystem is made of a CPU and some disks. In this context,

- N_{proc} is the total number of users in the subsystem (e.g., requests of web pages per second).
- R_{proc} is the average time spent in the subsystem by each request.
- X_{proc} is the subsystem throughput (e.g., the number of web pages per second).

Whole system Finally, the little law can be applied to the whole system. This time,

- N is the total number of users in the system (which is fixed since we have a closed system).
- R is the total amount of time spent in service, waiting and at the client side (think time, e.g., the time a user spend to read a web page and to elaborate a request).
- X is the rate at which the requests reach the systems from the terminals client and it corresponds to X_{proc}.

6.3.4 Interactive response time law

Interactive systems are usually made of users that submit requests to a computer. This means that a job spends most of its time waiting for the user to submit it. To model this behaviour we can include a new variable Z, which is the **think time**. The think time is the time between processing being completed and the job becoming available as a request again (i.e. the time waiting for the user to submit the request again). If we include the think time in the model, the little law (6.12) can be rewritten as

$$
\begin{aligned}
N &= X \cdot R \\
N &= X_{proc} \cdot R \\
N &= X_{proc} \cdot (R_{proc} + Z) \\
N &= X \cdot (R_{proc} + Z)
\end{aligned}
$$

in fact

- The throughput X of the system is the throughput X_{proc} of the processing unit.
- The response time is the sum of the residence time R_{proc} of the processor and the think time Z.

From the equation above, we can obtain

$$R_{proc} = \frac{N}{X_{proc}} - Z = \frac{N}{X} - Z \tag{6.13}$$

This equation, called **interactive response time law**, describes the response time in an interactive system as the residence time $\frac{N}{X}$ minus the think time Z.

Notice that, if the think time is 0, then the interactive response time law corresponds to the little law.

6.3.5 Linking subsystems

In a system made of multiple connected subsystems, we should be able to define laws to link the operational laws of each subsystem. In particular, if we call C_k the number of completions at resource k, we define the **visit count** V_k of the k-th resource to be the ratio between the number of completions C_k at that resource and the number C of system completions

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

The visit count represents the number of visits to a subsystem k for each completion of the whole system. The value of the visit count can be used to determine how many times a resource has been accessed after a system level completion (i.e., after a generic job ends, we want to know how many times a resource k is visited),

- If $V_k > 1$, resource k is visited multiple times after each completion (from whatever component).
- If $V_k < 1$, resource k is visited only sometimes after each completion (from whatever component).
- If $V_k = 1$, resource k is visited (on average) every time a job is completed (by whatever component).

6.3.6 Forced flow law

The visit count is useful to link the system level throughput to the component-level throughput. This relationship is expressed by the **forced flaw law** that states that the throughput or flows, in all parts of a system must be proportional to one another. More precisely, the throughput X_k at the k-th resource is equal to the product of the throughput X of the system and the visit count V_k at that resource.

$$X_k = X \cdot V_k \tag{6.14}$$

The forced flow law can be derived from the throughput X_k (6.2) of a component replacing $C_k = V_k \cdot C$ and remembering that the throughput X of the system is $X = \frac{C}{T}$ (6.2).

$$X_k = \frac{C_k}{T}$$
$$X_k = \frac{C \cdot V_k}{T}$$
$$X_k = X \cdot V_k$$

6.3.7 Service demand

A job might have to wait for some time before processing begins. To account for this time, we can use the **service demand** D_k, which represents the total amount of service (i.e., the amount of service time) that a system job generates at resource k. The service demand can be computed as

$$D_k = S_k \cdot V_k \tag{6.15}$$

The service demand can be used to compute the utilisation law with respect to the system's throughput. To obtain this result we can start from the utilisation law,

$$U_k = S_k \cdot X_k$$

Now we can use the forced flow law (6.14) to replace X_k and obtain

$$U_k = S_k \cdot X_k$$
$$U_k = S_k \cdot V_k \cdot X$$

Finally we recognise that $S_k \cdot V_k = D_k$ to obtain

$$U_k = S_k \cdot X_k$$
$$U_k = S_k \cdot V_k \cdot X$$
$$U_k = D_k \cdot X$$

The new utilisation law is therefore

$$U_k = D_k \cdot X \tag{6.16}$$

6.3.8 Response and residence time

Response time $\tilde{R_k}$ and residence time R_k are different concepts, in particular

- The **response time** $\tilde{R_k}$ of a subsystem k is the time needed to answer a request, namely, the time from entering the queue to exiting the server.
- The **residence time** R_k of a subsystem k considers loops. In some cases, a job is sent back to the same component that just processed it to create a loop. The residence time considers how many times the job is kept in the loop. In formulas,

  $$R_k = \tilde{R_k} \cdot N_{visits}$$

  where N_{visits} is the number of times that a component is visited in subsequent iterations.

Also note that, the relation between residence time R_k and response time $\tilde{R}_k$

$$R_k = V_k \cdot \tilde{R}_k \tag{6.17}$$

is the same as the one between demand D_k and service time S_k

$$D_k = V_k \cdot S_k \tag{6.18}$$

This means that in systems where $V_k = 1$ (i.e., single queue open system or tandem models):

- Average service time and service demand are equal.

$$D_k = S_k \tag{6.19}$$

- Response time and residence time are identical.

$$R_k = \tilde{R}_k$$

6.3.9 General response time law

The little law (6.12) can be used to compute the response time $\tilde{R}$, however we need to know the number of requests N in the system (remember that $\tilde{R} = \frac{N}{X}$, where R is the residence time). The number of requests might be unknown (e.g., in open systems) hence we have to find an alternative to the little law. We can start applying the little law to the single components to obtain

$$N_k = X_k \cdot \tilde{R}_k \tag{6.20}$$

where

- N_k is the average number of requests in a subsystem. Note that N_k is in general known.
- $\tilde{R}_k$ is the average time spent by a request in a subsystem.

If we replace the forced flaw law $X_k = V_k \cdot X$ (6.14) we obtain

$$N_k = X \cdot V_k \cdot \tilde{R}_k \tag{6.21}$$

$$\frac{N_k}{X} = V_k \cdot \tilde{R}_k \tag{6.22}$$

Now we can notice that the total number of requests in the system is the sum of the mean number of requests at each subsystem

$$N = N_1 + N_2 + \cdots + N_K \tag{6.23}$$

Now we can apply the little law to a single component to obtain

$$R_k = \frac{N_k}{X} \tag{6.24}$$

At this point we can replace 6.21 in 6.24 to obtain

$$R_k = V_k \cdot \tilde{R}_k \tag{6.25}$$

Thanks to 6.17 we can say that the average response time of a job in the system is the sum of the product of the average time for the individual access at each resource and the number of visits it makes to that resource

$$R = \sum_k V_k \tilde{R}_k = \sum_k R_k \tag{6.26}$$

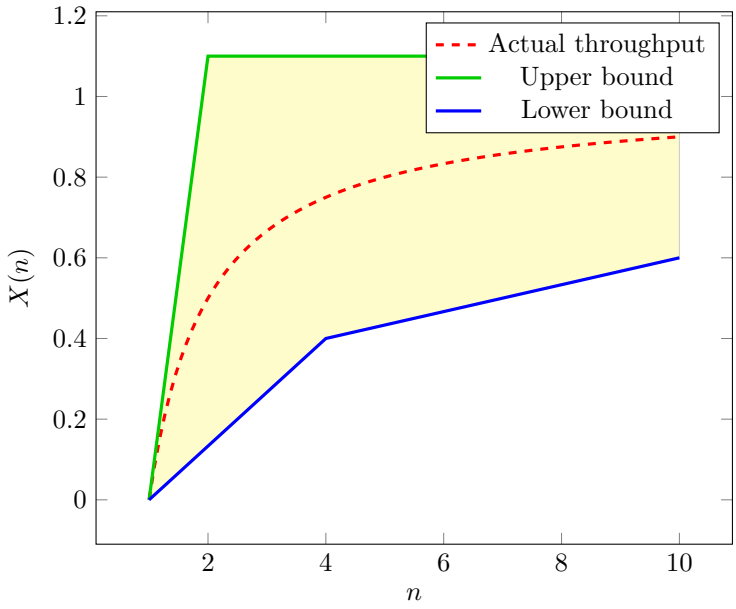


Figure 6.2: Upper and lower bound of a system's throughput $X(N)$.

6.4 Performance bounds

Operational laws link the main observable quantities of the system, however we are also interested to evaluate where the limit of a system are. In particular, we want to understand what are the constraints, or the bounds, of the system in terms of performance. This evaluation can help us understand the critical areas of a system where we should put our attention. Moreover, defining the performance bounds of a system, allows us to understand if the requirements can be satisfied or not. Performance bound doesn't give a precise description of a system's performance but only an estimate of its upper and lower bound. To better explain this concept, let us consider the throughput X of a system in relation to the number n of devices in the system. The function $X(n)$ can't be defined precisely, but we can bound it. A graphical representation of this idea is shown in Figure 6.2. Performance boundaries also allow us to tell why we can't reach a specific performance, and therefore how to fix the system to achieve such performance.

Bottlenecks The boundaries is also also related to the concept of bottleneck, i.e., the part of the system where we have the worst performance. In particular, the bottleneck of a system is usually the component in which we have the highest demand D_{max}, hence the highest utilisation (because $U_k = D_k \cdot X$). This makes sense, in fact in the extreme case in which U_k is 1 (which is the maximum value for the utilisation) and k is the bottleneck, the component k is always processing jobs, hence some other jobs are always waiting and the queue enlarges. If the queue grows, the system delays because some requests can't be served immediately. Finding bottlenecks isn't important only during production, but also when we have to decide which part of the system to upgrade. In particular, we should try to improve the performance of the bottlenecks.

Performance metrics To define the performance bounds of a system we will use

- The **operational laws**.
- The number K of components in the system. Notice that K can be used to indicate the set of components, too.
- The demand D_k of each component k. The maximum demand among all components is represented as $D_{max} = \max_k\{D_k\}$.
- The demand D of the whole system. The demand D can be computed as the sum of the single components' demands.
$$D = \sum_{k \in K} D_k$$
- The think time Z. Remember that, the think time is used only in closed systems.

The ingredients above will be used to compute the bounds of

- The system's **throughput** X.
- The system's **response time** R.

To compute the bounds we have to put ourselves in two scenario:

- The **best-case (or optimistic) scenario**. This scenario considers light load situations and is used to compute
  - The upper-bound X_u of the throughput (because in the best scenario, the throughput is maximum).
  - The lower-bound R_l of the response time (because in the best scenario, the system need less time to process the requests).
- The **worst-case (or pessimistic) scenario**. This scenario considers heavy load situations and is used to compute
  - The lower-bound X_l of the throughput (because in the worst scenario, the throughput is minimum).
  - The upper-bound R_u of the response time (because in the worst scenario, the system need more time to process the requests).

Finally, we have to evaluate the performance boundaries for open and closed systems separately, because in open systems the number of requests in the system is unknown. To sum thinks up, for every type of system (open and close) we have to

1. Consider the best-case scenario to compute the throughput's upper-bound X_u and the response time's lower-bound R_l.
2. Consider the worst-case scenario to compute the throughput's lower-bound X_l and the response time's upper-bound R_u.

6.4.1 Open models

Open models are easier to analyse then closed models, however the results obtained are less precise (i.e., the bounds are very loose).

Throughput

Best-case scenario Let's start by computing the throughput in the best-case scenario. The best scenario for the throughput is when the throughput X is equal to the arrival rate in input λ.

$$X = \lambda$$

In fact if the arrival rate is bigger than X, we have an accumulation of requests that can't be server immediately. Now that we have sorted out what the best scenario is, we can start computing the throughput. We can start by writing the utilisation U as

$$U = X \cdot D$$

The utilisation should be not greater than 1, otherwise a component could be used more than once. If we impose this condition we obtain

$$U = X \cdot D \leq 1$$

Now we can apply the fact that we are in the best case scenario, hence $X = \lambda$ and write

$$\begin{aligned} U = X \cdot D &\leq 1 \\ \lambda \cdot D &\leq 1 \end{aligned}$$

Form this inequation we obtain

$$X = \lambda \leq \frac{1}{D} \tag{6.27}$$

This means that the maximum arrival rate that the system can handle is $\frac{1}{D}$, otherwise the response time of the system grows.

$$X_{u,open} = \frac{1}{D} \tag{6.28}$$

Worst-case scenario In the worst-case scenario, N requests arrive in parallel and are enqueued to the same component. However, since we are in open system, the number of requests N is unbounded and unknown, hence we can't define a lower bound for the throughput.

$$X_{l,open} = \text{undefined} \tag{6.29}$$

Response time

Best-case scenario In the best case scenario, no request is queued, hence the response time of the system is simply the sum of the demand of the components.

$$R_{l,open} = \sum_{k \in K} D_k \tag{6.30}$$

Since the response time depends on D_k (and $D_k = S_k \cdot V_k$), to optimise it we can

- Decrease the service time, for instance, using faster CPUs.
- Reduce the number of visits V_k. For instance, we can change the topology of the network, in fact, if we put two components in parallel, we halve the number of visits of each of the two components because the requests are shared between the two components.

Worst-case scenario As for the throughput, the worst case scenario depends on the number of requests in the system, which is unknown, hence we can't compute the upper bound for the response time.

$$R_{u,open} = \text{undefined} \tag{6.31}$$

	Throughput	Response time
Best case	$X_{u,open} = \frac{1}{D}$	$R_{l,open} = \sum_{k \in K} D_k$
Worst case	$X_{l,open} = \text{undefined}$	$R_{u,open} = \text{undefined}$

Table 6.1: A summary of the performance bounds for open systems.

6.4.2 Closed models

In closed models, we know the number of requests N in the system, hence we can compute the performance in the best and worst-case scenarios. Moreover, since we are talking about closed systems, we have to remember that we can use the interactive law

$$N = X \cdot (R + Z)$$

Throughput

Single request scenario In the lightest-weight scenario, we have only one request in the system, hence we can write the interactive law as

$$1 = X \cdot (R + Z)$$

Since we have only one request, we can replace $R = D$ and obtain

$$1 = X \cdot (D + Z)$$

from which we can derive the maximum throughput $X_{u,closed}$

$$X_{light,closed} = \frac{1}{D + Z}$$

Worst-case scenario To consider the worst case scenario, we have to increase the number of requests in the system. Before going on, notice that in this case, the throughput can be written as a function of the number of requests (i.e., $X(N)$), however we will use simply X. In this case, the worst situation possible is when every time a job arrives at a station, it finds $N - 1$ jobs (i.e., all the others) in the queue. This means that the last job has to wait $(N - 1) \cdot D$ time units before being processed. In total, we have to wait $N \cdot D$ time units, hence the interactive law is

$$N = X \cdot (R + Z)$$
$$N = X \cdot (N \cdot D + Z)$$

From this equation we can derive the minimum value of X

$$X_{l,closed} = \frac{N}{ND + Z} \tag{6.32}$$

The lower bound for the throughput depends on N, hence we can compute its limit for $N \to \infty$

$$\begin{aligned} \lim_{N\to\infty} X_{l,closed} &= \lim_{N\to\infty} \frac{N}{ND + Z} \\ &= \lim_{N\to\infty} \frac{N}{ND} \\ &= \frac{1}{D} \end{aligned}$$

Best-case scenario In the best case scenario, a job is served immediately (i.e., it's never queued), hence the response time is the sum of the service demands of the single components k (which happens to be the demand of the system)

$$R = \sum_{k \in K} D_k = D$$

If we replace this value in the interactive law we obtain

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

hence the upper bound for the throughput is

$$X_{u,closed} = \frac{N}{D + Z} \tag{6.33}$$

As we can see, the value of X increases linearly with the number of requests N, hence it grows towards infinity. However, each component can't have an utilisation greater than 1, hence we have to impose that

$$U_k \leq 1 \quad \forall k \in K$$

From the utilisation law (the version with the service demand), we know that

$$U_k = D_k \cdot X$$

hence we have to impose that

$$U_k = D_k \cdot X \leq 1$$

which can be rewritten as

$$X \leq \frac{1}{D_k}$$

This inequation has to be true for every component, hence it also has to be true for the component with the highest demand, which is the bottleneck.

$$X \leq \frac{1}{D_{max}}$$

Note that, we consider the component with the highest demand because it's the one that limits the performance of the system, hence if its utilisation is fine (i.e., smaller than 1), then the utilisation

of the other components is fine, too. Thanks to the inequation $X \leq \frac{1}{D_{max}}$ we can put a limit on the value of the throughput, and rewrite it as

$$X_{u,closed} = \min\left(\frac{N}{D+Z}, \frac{1}{D_{max}}\right) \tag{6.34}$$

The cross point N^* (i.e., the number of requests after which the throughput stops growing) between the first (i.e., the line that grows to infinity) and second function can be found putting $\frac{N}{D+Z} = \frac{1}{D_{max}}$

$$\frac{N^*}{D+Z} = \frac{1}{D_{max}}$$
$$N^* = \frac{D+Z}{D_{max}}$$

A graphical representation of the throughput's upper-bound is shown in Figure 6.3.

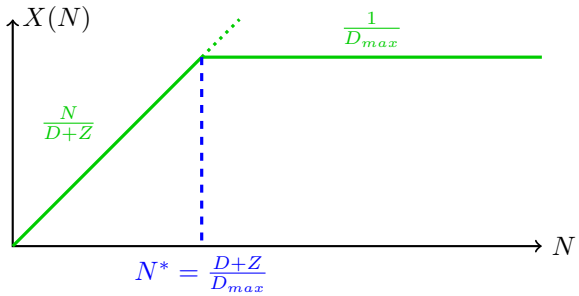


Figure 6.3: The qualitative representation of the throughput's upper-bound $X_{u,closed}$ of a closed system.

Response time

The response time in the best and worst-case scenarios can be obtained from the interactive law.

Best-case scenario The request time can be obtained from the interactive law

$$N = X \cdot (R+Z)$$
$$N = \frac{N}{D+Z} \cdot (R+Z)$$
$$N = \frac{N}{D+Z} \cdot R + \frac{N}{D+Z} \cdot Z$$

We can invert the last equation to obtain

$$\begin{aligned} R \cdot \frac{N}{D+Z} &= N - \frac{N}{D+Z} \cdot Z \\ R &= \left(N - \frac{N}{D+Z} \cdot Z\right) \frac{D+Z}{N} \\ R &= \left(1 - \frac{1}{D+Z} \cdot Z\right) D + Z \\ R &= \left(D + Z - \frac{D+Z}{D+Z} \cdot Z\right) \\ R &= \left(D + Z - Z\right) \\ R &= D \end{aligned}$$

Hence the upper-bound for the request time is

$$R_{l,closed} = D \tag{6.35}$$

Worst-case scenario The request time can be obtained from the interactive law

$$\begin{aligned} N &= X \cdot (R + Z) \\ N &= \frac{N}{ND+Z} \cdot (R + Z) \\ N &= \frac{N}{ND+Z} \cdot R + \frac{N}{ND+Z} \cdot Z \end{aligned}$$

We can invert the last equation to obtain

$$\begin{aligned} R \cdot \frac{N}{ND+Z} &= N - \frac{N}{ND+Z} \cdot Z \\ R &= \left(N - \frac{N}{ND+Z} \cdot Z\right) \frac{ND+Z}{N} \\ R &= \left(1 - \frac{1}{ND+Z} \cdot Z\right) ND + Z \\ R &= \left(ND + Z - \frac{ND+Z}{ND+Z} \cdot Z\right) \\ R &= \left(ND + Z - Z\right) \\ R &= ND \end{aligned}$$

Hence the upper-bound for the request time is

$$R_{u,closed} = ND \tag{6.36}$$

Notice that, the upper bound for the response time becomes more coarse grained and less accurate when N grows because with many requests, the probability that all jobs are always in the same station (which is our worst-case) is low.

	Throughput	Response time
Best case	$X_{u,closed} = \min\left(\frac{N}{D+Z}, \frac{1}{D_{max}}\right)$	$R_{l,closed} = D$
Worst case	$X_{l,closed} = \frac{N}{ND+Z}$	$R_{u,closed} = ND$

Table 6.2: A summary of the performance bounds for closed systems.

Part IV

Software infrastructure

Chapter 7

Cloud computing

Let start our journey in cloud computing by defining it.

> **Definition 8** (Cloud computing)**.** *Cloud computing is a coherent, large-scale, publicly accessible (via Web service calls through the Internet) collection of computing, storage, and networking resources.*

This means that cloud computing allows users to remotely access some resources (e.g., a server or a service) without knowing the physical location of such resources. This approach is both an advantage, since it allows users to access resources located around the globe, and a limitation since the access to the server is strongly bounded to the network's accessibility (if we can't access the network, we can't access the remote resources). Moreover, we should also consider the latency of the infrastructure itself. The companies that offer cloud services usually require users to pay a periodic fee to use their services. This is a big advantage for costumers that can choose the services they need and pay only such services. Basically, cloud computing is elastic and it can scale up and down depending on the user's needs (i.e., on the requests it's receiving from the customer). This is a big advantage and it comes from the fact that the resources are accessed remotely.

Despite it's advantages, cloud computing has several limits:

- When a cloud infrastructure scales up, it remains the same for some time (i.e., it can't scale down immediately).
- The cost of the IT involves also some fixed costs.
- The customer load is neither fixed nor predictable.

7.1 Dynamic load balancing

One of the most important properties of cloud computing is the ability to scale up and down dynamically depending on the load (i.e., the number of requests). To analyse how does dynamic load-balancing work, let us consider a situation in we predict that the load on the server will increase linearly with time. In these situation, one might want to periodically increase the IT resources so that they are always enough to serve all the predicted requests. If we were to plot the IT capacity with respect to the time, the predicted capacity (i.e. the capacity required to serve the predicted load) is a linear function, while the allocated capacity is a step function. The actual load can however

behave very differently and in particular it is quite low at the beginning and increases exponentially with time. This means that we have

- A region in which we are wasting resources because the predicted load is higher than the actual load. This condition isn't a big issue in terms of performance since we can still serve all users. On the other hand, we are wasting a lot of resources (for which we payed).
- A region in which we haven't allocated enough resources since the actual load is higher than the predicted load (because the actual load increases exponentially, or in general faster than a line). This condition is a problem for us since we can't serve all customers, hence we are losing some of them (they won't use our service anymore since it's not available).

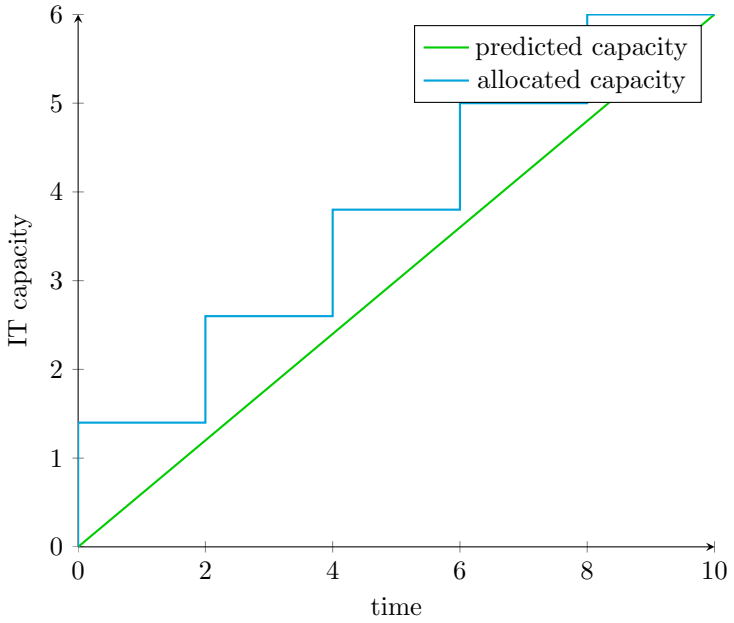


Figure 7.1: The IT capacity of a cloud infrastructure with respect to time.

To solve the problems related with the previous technique for allocating resources we have to allocate resourced dynamically, following the behaviour of the actual load and not of the predicted load. Luckily, since we are not in a physical database, we can allocate and release resources dynamically depending on the load, without losing resources (like in the under-load zone) or requests (like in the over-load zone). That being said, the predicted load has to be always bigger than the actual load, otherwise we can't serve some requests. Ideally, the predicted load should follow closely the actual load without being too far from it. In this situation we allow all used to be served without allocating too many resources.

7.1.1 Virtual machines and server consolidation

To achieve proper dynamic scheduling (i.e., that allows to allocate enough resources to follow the actual load) we can use virtual machines. A virtual machine simulates a computer and runs, with other virtual machines on the same hardware. In other words, many virtual machines share the same hardware without interacting. Virtual machines allow to

- **Decouple software and hardware**. In particular the same VM can run on different hardware, hence it's independent from the machine on which it's placed.
- Have **no unique correspondence between software and hardware usage**. In particular, since we can have multiple machines on the same hardware, having a big load on a VM doesn't mean having the same high load on the hardware on which it runs.
- **Treat OS and applications as a single entity**, independently from the other entities that run on the same hardware.

Thanks to virtual machines we can duplicate application and run every instance in parallel (i.e., on a VM) to serve all requests at the same time. Having parallel and independent applications, we can spawn new VMs or shut some down to follow the load requirements of the infrastructure. Moreover, since the hardware on which an application is placed isn't important, we can allocate the VM on all the devices of the infrastructure so that there aren't some devices with high load and some that have to serve few requests. Basically, we can balance the load on the devices of the infrastructure. This phenomenon is called scalability.

Moreover, since we can replicate the same application on different pieces of hardware, we can improve the availability of a service because if a server fails, other can serve its requests (since an application can be deployed on whatever computer).

Other advantages of virtual machines are

- We can run different operating systems on the same hardware.
- The hardware utilisation increases. This might looks like a disadvantage, but it's actually an advantage because power consumption isn't linear with respect to the utilisation.
- Costumers can access an high available system at a reduced cost, without realising that the services they are using are served on a VM.

7.2 As a Service paradigm

Thanks to virtual machines we can offer different resources as a service (e.g., an application, a computer, storage).

7.2.1 Software as a Service

Software as a Service (SaaS), also called Cloud applications, are applications that can be accessed through the Internet. Users usually access the services provided by this layer through web-portals and are sometimes required to pay fees to use them. Cloud applications can be developed on the cloud software environments or on infrastructure components.

7.2.2 Platform as a Service

Platform as a Service (PaaS), also called Cloud Software Environments, offer software environments remotely accessible (i.e., through the Internet) by developers. The remote development environment offers APIs that allow the developer to remotely develop applications. Some examples of Cloud Software Environments are Amazon Lambda and Tensorflow. The big advantage of this type of systems is that the computationally expensive part of development is handled by the Cloud provider which also offers the API used in the Software Environment. Moreover, the developer doesn't even

have to bother about scalability since it's handled by the provider. Notice that, all these features are payed by the customer since the provider needs a lot of resources to handle load balancing and scalability.

7.2.3 Cloud Software Infrastructure

Cloud Software Infrastructures allow users to access a virtual hardware remotely. Cloud Software Infrastructure are divided in

- **Infrastructures as a Service** (IaaS).
- **Data as a Service** (DaaS).
- **Communications as a Service** (CaaS).

Infrastructure as a Service

An infrastructure as a Service focuses on computation, hence computational power is offered to a customer. In particular, a customer can decide how many VMs (an in general resources) that have to be allocated. In other words, IaaS offers, as a service, the environment on which applications are run.

Infrastructures as a Service are very flexible in terms of computational power that the customers can require thanks to Virtual Machines and virtualisation. Remember however that, VMs and hardware sharing introduce some a small overhead.

Data as a Service

Data as a Service allow customers to store data on a remote disk and access it remotely. DaaS allows to obtain duplication, availability and data consistency without bothering about how to physically solve such problems (the remote Cloud provider handles such problems for us).

Communications as a Service

Communication as a Service allows to remotely handle the communication within a node.

7.3 Cloud classification

Cloud services can be divided in

- **Public Clouds**. Public Clouds are large platforms available to everyone. Public Clouds are based on Service Level Agreements (SLAs) and require users to pay for the performance they require (the higher the performance, the higher the cost). In public Clouds requests are sent via the Internet and the resources are granted via web services.
- **Private Clouds**. In private Clouds, everything is managed internally by the company that uses its own Cloud service (i.e., VMs are run on top of the company servers). The main advantage of this solution is that a company can optimise the resources since it controls them (the infrastructure is not outsourced). On the flip side, the company has to manage all aspects of the infrastructure (which might be expensive) and has to deal directly with scalability (i.e., it has to allocate new resources).

- **Community Clouds**. Community Clouds are consolidated and managed by some confederated companies that share their resources to go beyond the limit of physical resources. The idea is to obtain the same result as for private Clouds ensuring also good scalability (since many companies share their resources), confidentiality (since the service is still private) and peak load handling.
- **Hybrid Clouds**. Hybrid Clouds mix the solutions of the previous types to obtain the best Cloud for a specific context. For instance, companies that hold their private cloud, might have to deal with unpredictable peaks of load, hence they can rent some public resources (or in general resources from other types of Cloud) to serve all requests at peak time.

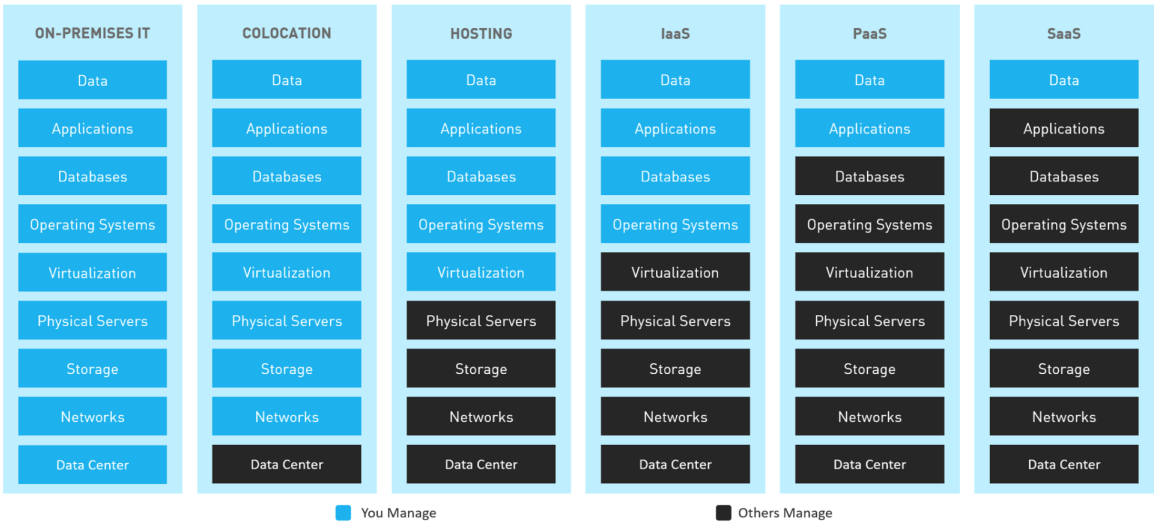


Figure 7.2: A summary of the different ways we can deploy a service, from fully controlled by us (on-premises IT) to mostly controlled by third-parties (SaaS).

Chapter 8

Virtualisation

Virtualisation allows to share some resources (e.g., hardware) among multiple users or applications (e.g., virtual machines). This technology guarantees

- **Isolation**, in fact, different applications run without knowing that other applications are sharing the same hardware and without interfering.
- **Security**, in fact, if an application is compromised, the others aren't because the applications don't interact (even if they share the same hardware).
- **Performance**, in fact, we can balance the number of application running on a piece of hardware to have maximum performance.

8.1 Physical machines

Virtual machines are different from physical machines. To understand what are the main differences between these technologies, we have to analyse a computer's architecture. In particular, a computer architecture is usually built with a layer abstraction in which

- Each **layer** defines a set of functionalities and instructions.
- Layers are connected through **interfaces**.

This type of abstraction allows to decouple functionalities and develop them independently.

8.1.1 Instruction Set Architecture

Among interfaces, one of the most important is the Instruction Set Architecture (ISA), in fact this interface is the one that allows the software (i.e., a program in assembly) to communicate with the hardware. The ISA should be defined before the actual hardware. Moreover, the ISA says how the hardware should be implemented and what we can do at the software level. An high level representation of a physical machine with its layers and interfaces is shown in Figure 8.1. The ISA is the separation point between hardware and software for a physical machine. and can be divided in:

- **User ISA**. The user ISA is the set of instructions that can be executed directly by the user (i.e., applications). This part of the ISA is the interface number 7 in Figure 8.1.

- **System ISA**. The system ISA is the set of instructions that can be executed only by a supervisor software (i.e., the Operating System). The system ISA is used by the OS to manage hardware resources. This part of the ISA is the interface number 8 in Figure 8.1.

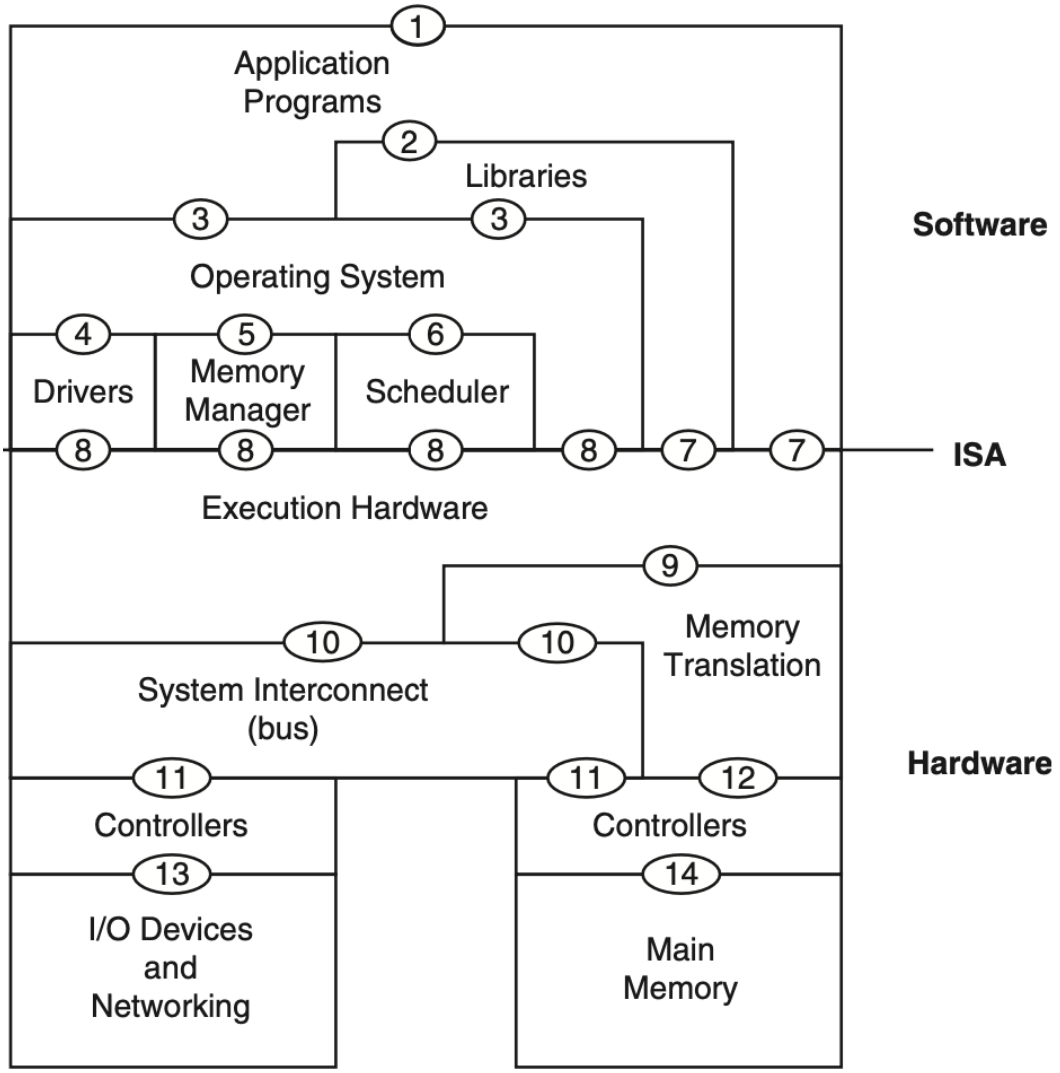


Figure 8.1: Layers in which a machine can be logically organised and the interfaces (numbered) that connects them.

8.1.2 Application Binary Interface

Other important interfaces are the ones on top of the operating system (the ones with number 3 in Figure 8.1). This set of interfaces allows an application to run on top of the operating system. This means that an application can

- Execute instructions directly on the hardware using the **user ISA**.
- Execute instructions on the operating system through system calls (`syscalls`) that ask the operating system to execute some instructions in privileged mode (i.e., using the system ISA).

These two interfaces together (number 3 and 7) are called Application Binary Interface (ABI) and represent the interfaces that an application can use to execute instructions on the hardware. To wrap things up, the ABI is the sum of

- The **user ISA**.
- **Syscalls**.

It's important to remember that, an application can't run on a machine that uses a different ABI because the two systems use different system calls (i.e., the interfaces are different).

8.2 Virtual machines

Virtual machines are an abstraction that provides the possibility to run on a virtualised execution environment. In other words, a virtual machine provides an abstraction of some behaviour of a physical machine from a software level. Namely, a virtual machine offers a different view of the hardware resources physically available. A virtual machine maps the virtual resources exposed into the physical ones physically available on the machine on which the VM is running. Note that, the physical resources exposed by the VM should be physically available, unless the VM can simulate them. From this description we can understand that, a virtual machine is able to replace one interface and map the instructions coming from the user of the interface to the layer above. Schematically, a virtual machine

- Provides identical software behavior with respect to the layer it replaces.
- Consists in a combination of physical machine and virtualising software.
- May appear as different resources than physical machine.
- May result in different level of performances.

Virtual machines can be divided in two categories, depending on the interface they replace (or abstract):

- **System virtual machines**. A system virtual machine exposes the ISA to the user of the VM. Basically, the virtualisation software replaces what's below interfaces 7 and 8.
- **Process virtual machines**. A process virtual machine exposes the ABI to user of the VM. Basically, the virtual machine software replaces what is below interfaces 3 and 7.

Before analysing these two types of virtual machines, let us describe how a system that uses a VM can be divided. A system can be divided into

- A **host**. The host is the set of software and physical hardware components.
- A **virtualisation software**. The virtualisation software is a program that replaces one or more interfaces of the host.
- A **guest**. The guest is the software that runs on top of the virtual machine.

The conjunction of host and virtualisation software is what we call **virtual machine**.

8.2.1 Process virtual machines

In a process virtual machine, everything below interface 3 is virtual, hence the virtual machine is placed at the ABI interface. This means that, the virtual machine

- Supports a **single application process**.
- **Provides the ABI** to the process and translates application instructions to the lower layers (i.e., to the OS and the the hardware).

The software that separates layer 3 from the upper layers is called **runtime software** or virtualisation software. From this description we can say that

- The **host** of the system is the physical hardware, the OS that runs directly on the physical environment and some additional software.
- The **virtualisation software** is the runtime software.
- The **guest** is the application program that runs on top of the runtime software.

This means that, for process virtual machines, the virtual machine is made of

- The physical hardware.
- The software that runs directly on top of the hardware (without virtualisation), like the OS.
- The runtime software.

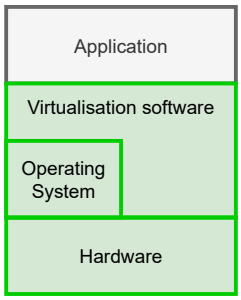


Figure 8.2: The different layers of a process virtual machine. The virtual machine is shown in green.

8.2.2 System virtual machines

System virtual machines place the virtualisation software at the ISA interface, hence they virtualise the hardware and the environment on which the operating system can run. The main tasks of system virtual machines are

- Managing the access to and the sharing of physical resources (e.g., memory, CPUs).
- Intercepting instructions sent by the host OS or by an application and map them to the physical environment.

For system virtual machines, the virtualisation software is called **Virtual Machine Monitor** (VMM) and the system is divided as follows:

- The **host** is the physical hardware (or in general the environment) on which the VMM runs.
- The **virtualisation software** is the virtual machine software.
- The **guest** is the environment on top of the VMM, which includes a guest OS and some processes.

This means that, for system VMs, the virtual machine is made of

- The **virtual machine monitor**.
- The **physical hardware**.

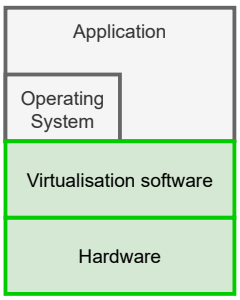


Figure 8.3: The different layers of a system virtual machine. The virtual machine is shown in green.

Management

System virtual machines can be managed in different ways, in particular

- The virtualisation software (i.e., the VMM) can be put directly on the hardware.
- The virtualisation software can be build on top of an host operating system, yet the VM still exposes the ISA. In this case, the system calls of done on the guest OS are mapped to the system calls of the host OS, which in turn will execute them on the physical hardware. This means that, the hardware is running two operating systems (host and guest) which aren't exponsed one to the other.

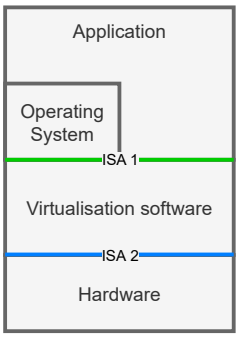
Usage

System virtual machines can also be used in different ways, in fact we can

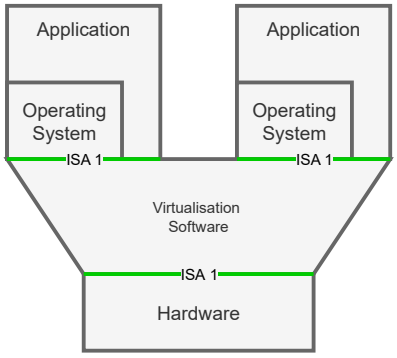
- Use an application designed for an ISA ISA_1 on some hardware that uses a different instruction set ISA_2. In this case we are using the VM for **translation** (or emulation).
- **Share and consolidate the resources**, namely different and independent applications that have the same ISA, can be executed on the same resource. Note that each application has its own operating system and they only share the physical resources virtualised by the VMM.

- Mix the two solutions above. This means running multiple applications on the same hardware, some of which might use a different ISA from the one of the physical resource.

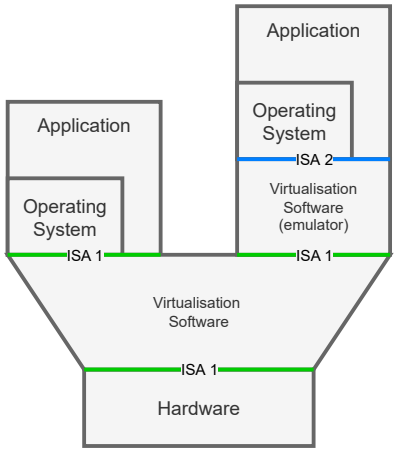
The functionalities above might require different virtualisation software, however there exist programs that allow to do both.



(a) Virtualisation software for emulation.



(b) Virtualisation software for resource management.



(c) Virtualisation software used for both memory management and emulation.

Figure 8.4: A summary of all the ways a Virtual Memory Management software can be used.

8.3 Virtual machines classification

In general, virtual machines can be classified as

- Process VMs.
  - Process VMs in which each application that shares the same hardware has different ABIs. In this case we talk about **Multi-programmed systems**.
  - Process VMs in which each application that shares the same hardware has the same ABI. In this case we talk about **Dynamic translators**.

- System VMs.
  - System VMs in which each guest has different ISAs. In this case we talk about **Whole-system virtual machines**.
  - System VMs in which each guest has the same ISAs. In this case we talk about **Classic-system virtual machines**.

8.3.1 Multi-programmed systems

Multi-programmed systems are process virtual machines that use the same ABI. This is basically the approach that some OS use to implement multi user support since multiple processes can run on the same hardware and OS. In particular, each user process is given:

- The illusion of having a complete machine to itself.
- Its own address space and access to a file structure.

8.3.2 Emulation

Emulation refers to those software technologies developed to allow an application (or OS) to run in an environment different from that originally intended. Emulation works both for process and system virtual machines mapping the ABI or the ISA, respectively.

An emulator reads all the bytes included in the memory of the system it is going to reproduce. Then, the interpreter program fetches, decodes and emulates the execution of each source instruction.

8.3.3 High level language virtual machines

High level language virtual machines are examples of **dynamic translators** and allow to execute each application (or each instance of an application) in an isolated execution environment. An example of high level language VM is the Java Virtual Machine. In high level language VM, the virtual machine has to

- Translate application byte code to OS-specific executable.
- Minimize hardware or OS-specific features for platform independence.

8.3.4 Whole-system virtual machine

A whole-system virtual machine virtualises all software, hence the ISAs are different and both application and OS code require emulation, e.g., via binary translation. Usually, whole-system virtual machines implement the VMM and guest software on top of a conventional host OS running on the hardware.

8.4 Implementation of virtualisation

Virtualisation is implemented using additional layer that can be added in between two layers of the classic layered architecture. In particular we can have:

- **Hardware level virtualisation**. The virtualisation layer is right on top of the hardware and below the OS and the application. In this case, the interface seen by guest OS and application might be different from the physical one.
- **Application level virtualisation**. The virtualisation layer is on top of the system that is already running on the physical machine and allows to build applications on top of it. In this case, the virtualisation layer provides the same interface to the applications. Moreover, applications run in their environment, independently from the host OS. An example of application level virtualisation is the Java Virtual Machine.
- **System level virtualisation**. The virtualisation is on top of the OS (which runs on the physical hardware) and allows to run multiple operating systems on the same hardware. An example of system level virtualisation is VirtualBox.

8.5 Virtual Machine Manager

An important part of a virtual machine is the Virtual Machine Manager, which handles virtualisation and ISA translation in system VMs. The VMM is a piece of software that allows to have control on the VM by mediating user's instructions and mapping them to the instruction of the host ISA. Usually, Virtual Machine Managers can be called in three different ways which in some case are used interchangeably and in other have a more specific meaning. More precisely we have

- **Virtual Machine Monitors**. A Virtual Machine Monitor is specifically the virtualisation software.
- **Hypervisors**. An hypervisor is the virtualisation software that runs immediately on the hardware without needing an operating system (that's why they are also called bare-metal or native). An hypervisor has total and direct control over the underlying physical hardware.
- **Virtual Machine Managers**. A Virtual Machine Manager is a VMM or an hypervisor that is also used to create, configure and maintain virtualised resources and that provides a user-friendly interface to the underlying virtualization software.

To be more clear let us use a less ambiguous classification and in particular let us divide VMMs in

- **Type 1** hypervisors.
- **Type 2** hypervisors.

8.5.1 Type 1 hypervisor

A type 1 hypervisor, also called bare-metal or native hypervisor, has total and direct control over the underlying physical hardware. Type 1 hypervisor can use two different architectures:

- The **monolithic architecture**.
- The **microkernel architecture**.

Monolithic architecture

In the monolithic architecture the device drivers run within the hypervisor. More precisely, the guest machines passes I/O requests through the hypervisor's drivers which execute the requests on the hardware. This means that the physical hardware is directly exposed to the hypervisor. A schematic representation of this architecture is shown in Figure 8.5. This architecture has it's own advantages and disadvantages, in particular the main advantages are that the system can achieve

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

However, this comes at a cost, in fact

- This architecture **can run only on hardware for which the hypervisor has drivers**.

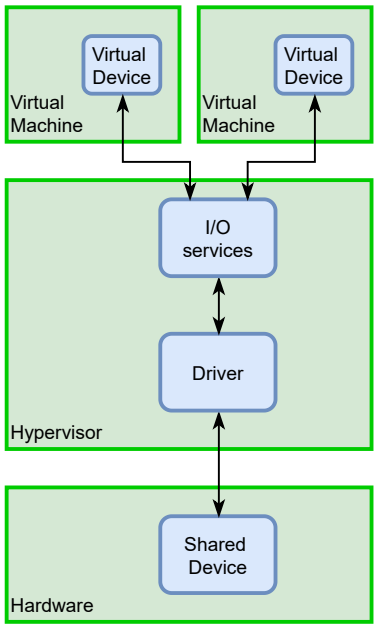


Figure 8.5: The monolithic architecture.

Microkernel architecture

In the microkernel architecture we have a service virtual machine that contains the device drivers. In this case the virtual machines send their commands to the service VM. The main advantages of this architecture are

- We can abstract the access to the shared device.
- There is no need for drivers in the hypervisor, hence the hypervisor is smaller.
- We leverages the driver ecosystem of an existing OS (i.e., the one of the service VM).

- We can use third party drivers.

However this comes at a cost, in fact

- Performance is much worst with respect to monolithic hypervisors.

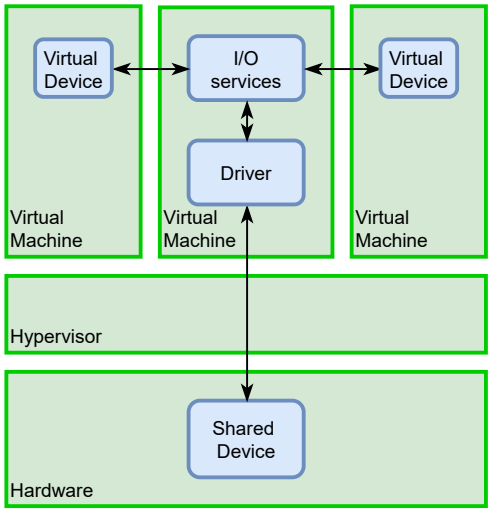


Figure 8.6: The microkernel architecture.

8.5.2 Type 2 hypervisor

A type 2 hypervisor, also called hosted hypervisor, resides within a host operating system and leverages the code of the host operating system. This means that when using a type 2 hypervisor, at least two operating systems are running on the same piece of hardware:

- A host OS, on which the hypervisor runs.
- One or more guest OSs that run on top of the hypervisor. The applications run on the guest OS.

The main advantages of type 2 hypervisors are:

- They are more flexible in terms of underlying hardware.
- They are easier to manage and configure (VMMs can use the host OS to provide GUI, not only BIOS).

It's not all good, in fact the main disadvantages of type 2 hypervisors are

- We must take special care must to avoid conflict between host and guest OS (e.g., Virtual Memory).
- The host OS might consume a non-negligible set of physical resources (e.g., 1 core for the host OS), hence the performance could decrease.

8.6 Virtualisation techniques

System level virtualisation can be implemented with different techniques:

- **Paravirtualisation**.
- **Full virtualisation**.

8.6.1 Paravirtualisation

When using paravirtualisation, the VMM exposes a an interface similar to the one of the hardware (but not exactly the same). The guest OS and the VMM collaborate and communicate using **hooks**. More precisely, the guest OS requests to the VMM to execute some tasks and which will be asynchronously executed by the VMM on the physical hardware. When the task has been executed, the guest OS is acknowledged. Paravirtualisation allows to

- Have better performance since operations are executed by the VMM and not in the virtual domain.
- Have a simpler VMM.

The main drawback is that

- We have to use a modified guest operating system since it has to collaborate with the VMM.

8.6.2 Full virtualisation

Full virtualisation provides a complete simulation of the underlying hardware. This means that we can use normal guest operating systems that will run in a fully isolated environment. Each OS can't realise that the hardware is shared since the VMM shows all its resource that appear to be at full disposal of the guest OS (i.e., the OS thinks it can use all the resources offered by the VMM). In particular, guests OS have full access to

- The full instruction set.
- Input/output operations.
- Interrupts.
- Memory.

Only dome protected instructions are trapped and handled by the VMM. The main advantage of this technology is that

- We can **use unmodified operating systems**.

On the other hand,

- The guests need the VMM mediation to allow the guests and host to request and acknowledge tasks which would otherwise be executed in the virtual domain.
- We can't use this technology on every machine since we need high performance hardware with some specific functionalities.
- Performance is worst with respect to paravirtualisation.

8.7 Containers

Containers are pre-configured packages that help executing some code in a target machine (i.e., a machine with some specific characteristics) despite of the environment in which we are. Basically, the container fixes the environment that we are using to adapt to the target environment. A container holds all the libraries, functions and syscalls needed to run an application in a target environment. A container doesn't replace the operative system but it virtualises it for a specific app it's running. This means that the host system kernel is always shared, also with other containers (differently from VMs).

Using container has many advantages, in particular

- **Flexibility**.
- **Lightness of the application** since containers can share the host kernel.
- **interchangeability** since updates and updates can be distributed on the fly.
- **Portability** since a container can be created locally, deployed in the Cloud and ran anywhere.
- **Scalability** since a container can be duplicated and the different copies can be ran in parallel.
- It's **stackable** since containers can be stacked vertically and on the fly.

Also note that, 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.

Thanks to their versatility, containers are widely used, especially for testing and developing since they allow us to pack everything in a container and simulate the actual functioning of the application with a specific configuration. Here's a list of possible tasks for which it makes sense to use a container:

- Helping make your local development and build workflow faster, more efficient, and more lightweight.
- Running stand-alone services and applications consistently across multiple environments.
- Using container to create isolated instances to run tests.
- Building and testing complex applications and architectures on a local host prior to deployment into a production environment.
- Building a multi-user Platform-as-a-Service (PAAS) infrastructure.
- Providing lightweight stand-alone sandbox environments for developing, testing, and teaching technologies, such as the Unix shell or a programming language.
- Software as a Service applications.

In general, containers are used to provide easy portability and flexibility for software and applications.

Definitions

Equations