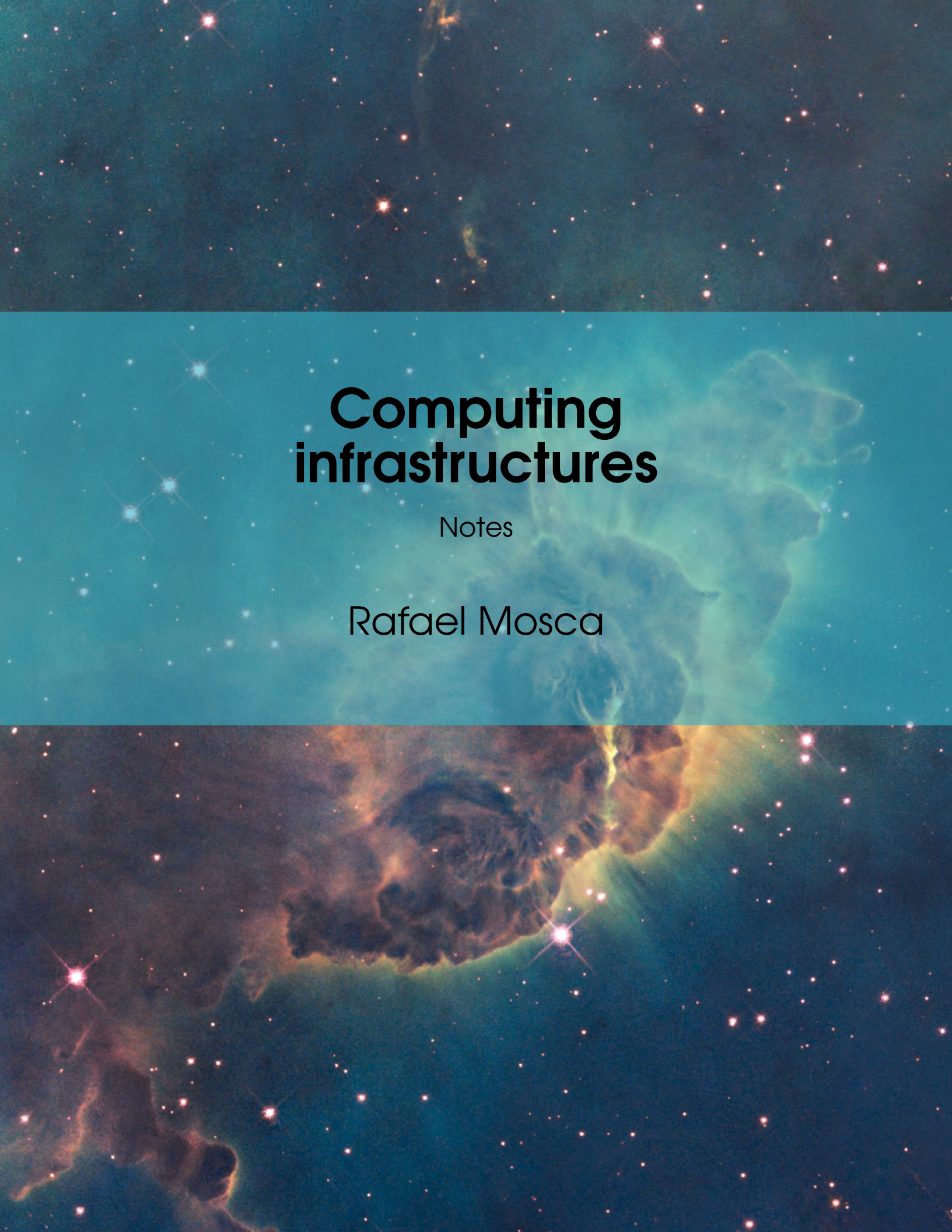


Computing infrastructures

Notes

Rafael Mosca

Written with LaTeX

Disclaimer: This book was done by a student based on his notes of the course Computing Infrastructure held by teachers Paolo Cremonesi, Manuel Roveri and Enrico Barbierato.

First release, July 2019

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione: - Non commerciale - Non opere derivate 4.0 Internazionale.

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

Per leggere una copia della licenza visita il sito web `https://creativecommons.org/licenses/by-nc-nd/4.0/` o spedisci una lettera a Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Contents

I Part One

1 Storage 9

1.1 Operations on a file 10

1.1.1 Reading a file 10

1.1.2 Writing a file 10

1.1.3 Deleting a file 11

1.2 Hard Disk Drives 12

1.2.1 HDD Performance 13

1.3 Solid State Drives 14

1.3.1 Internal organization 14

1.3.2 Performance 14

1.3.3 SSD vs HDD 16

2 RAID 17

2.1 RAID 0 18

2.2 RAID 1 19

2.3 RAID 2 20

2.4 RAID 3 20

2.5 RAID 4 21

2.6 RAID 5 22

2.7 RAID 6 **23**
2.7.1 Update of parity . . . 24
2.7.2 Recovery techniques . . . 24
2.8 Hybrid RAID approaches **26**
2.8.1 RAID 0+1 . . . 26
2.8.2 RAID 1+0 . . . 26
2.8.3 Similarities and differences . . . 26
2.9 Data storage systems **27**
2.9.1 DAS . . . 27
2.9.2 NAS . . . 27
2.9.3 SAN . . . 27
2.9.4 Differences between NAS and SAN . . . 27

3 Dependability . . . 29
3.1 Reliability **29**
3.1.1 Failures . . . 30
3.1.2 Metrics . . . 31
3.1.3 System Reliability . . . 32
3.2 Availability **34**
3.2.1 Difference between Availability and Reliability . . . 34
3.2.2 RAID reliability . . . 35

4 Data Center . . . 39
4.1 Data center components **39**
4.1.1 Rack modules . . . 41
4.2 Network architectures **41**
4.2.1 Three Layers . . . 41
4.2.2 Fat-tree . . . 42
4.2.3 D-Cell . . . 43
4.3 Efficiency metrics **43**
4.3.1 PUE . . . 43
4.3.2 Quality Tier . . . 43

5 Virtualization & Cloud Computing . . . 45
5.1 Machine's layers of abstraction **46**
5.1.1 ISA . . . 47
5.1.2 ABI . . . 47
5.2 Virtual Machines **48**
5.2.1 Process VM . . . 48
5.2.2 System VM . . . 49

5.3 Types of virtual machines **49**
5.3.1 Multiprogrammed systems . . . 50
5.3.2 Dynamic translators . . . 50
5.3.3 Classic VM . . . 50
5.3.4 Whole-System VM . . . 50
5.4 Hypervisors **51**
5.4.1 Type I Hypervisor . . . 51
5.4.2 Type II Hypervisor . . . 52
5.5 Virtualization technologies **52**
5.5.1 Virtualization implementations . . . 52
5.5.2 Properties of virtualization . . . 52
5.5.3 Virtualization techniques . . . 53
5.6 Virtual resources **54**
5.7 CPU Virtualization **54**
5.7.1 Core partitioning . . . 54
5.8 Memory Virtualization **55**
5.8.1 RAM virtualization . . . 55
5.8.2 Virtual disks . . . 56
5.9 I/O Virtualization **56**
5.10 Network Virtualization **56**
5.10.1 Network Address Translation . . . 57
5.10.2 Bridged Networking . . . 57
5.10.3 Host-only Networking . . . 57
5.10.4 Internal networking . . . 57
5.11 Server consolidation **58**
5.12 Cloud Computing **59**
5.12.1 Cloud Computing Ontology . . . 59
5.12.2 Types of clouds . . . 61

II Part Two

6 Capacity Planning . . . 65
6.1 Queue Network Models **65**
6.2 Laws **67**
6.2.1 Utilisation Law . . . 67
6.2.2 Little's Law . . . 67
6.3 Open System **68**
6.4 Closed System **68**
6.4.1 Batch systems . . . 68
6.4.2 Think time systems . . . 69

6.5 Stations **69**
6.5.1 Forced Flow Law 70
6.6 Load Balancing **72**

7 Performance Models 73

7.1 Closed System Asymptotic Bounds **73**
7.1.1 Optimistic Bounds 73
7.1.2 Pessimistic Bounds 74
7.2 Balanced System Bounds **77**
7.2.1 Response Time Balanced Bounds 78
7.2.2 Throughput Balanced Bounds 78

8 Open Models 81

8.1 Bounds **83**

Part One

1 Storage 9

1. Storage

Thanks to the technological advancements in our society, today it is possible to store data in some recording mediums such as a USB, CD, HDD or SDD, just to name a few.
As we will see, not all recording mediums have the same performance, some perform well in certain situations while others show a very poor performance in certain applications. To analyse the performance of the different drives we must first understand how data is stored in such drives.

A **disk** or drive can be seen by the Operative System (OS) as a *collection of data blocks* that can be read or written independently. We can think of data blocks as elements of an array that can store some data. To retrieve data from an array, we need to know the indexes of the elements from which we want to retrieve the data, in a similar way, each data block is characterised by a unique numerical address called **Logical Block Address (LBA)**, that helps to uniquely identify each element. Even though this address helps in both the ordering and management of the data blocks, we normally address data through a better way: **files**.

Typically, to simplify the access to the disk, the OS groups data blocks into what we call **clusters**, these clusters contain:

1. File data, i.e. the actual content of the file.
2. Metadata, i.e. the information required by the file system:
   - File name
   - File size and file type
   - Date of creation, modification and last access
   - Security information (owners, access list, encryption)
   - *Links to the LBAs where the file content can be located on the disk*

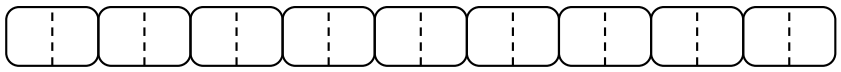


Figure 1.1: Eight clusters made of sixteen data blocks

In UNIX we can access some metadata of a file with the `stat <filename>` command

```
File: "/etc"
ID: 0 Namelen: 255 Type: reiserfs
Block size: 4096 Fundamental block size: 4096
Blocks: Total: 1977922 Free: 1272318 Available: 1272318
Inodes: Total: 0 Free: 0
```

1.1 Operations on a file

1.1.1 Reading a file

To read a file we must:

1. Access the metadata of the file to locate its blocks
2. Access the blocks to read their content

1.1.2 Writing a file

To write a file we must do essentially two things:

1. Access the metadata to locate free blocks
2. Write data in the assigned blocks

Since the file system can only access clusters, the real size in terms of disk space for a file is always a multiple of the cluster size, this organisation can lead to wasted disk space. We call this waste of space **internal fragmentation** of files.
If we call s the file size, c the cluster size, a the actual size on disk, we can write the following formula:

$$a = \lceil \frac{s}{c} \rceil * c \tag{1.1}$$

Where the **wasted space** due to the organisation of the file into clusters is:

$$w = a - s \tag{1.2}$$

R As an attentive reader could suggest, the smaller the cluster size, the smaller the waste. Why not make it equal to one block? The answer lies on the metadata, a cluster size must be big enough as to contain all the needed metadata for a file.

1.1.3 Deleting a file

Deleting a file is a tricky thing, actually when we delete a file we are not actually deleting the file, we are just updating the metadata and marking the blocks occupied by the file as not in use anymore. To be sure of permanently deleting the file we should overwrite all the blocks that held some of its data. If we do not overwrite immediately, when a new file request to be written, the clusters will show up as available and the old data will be replaced by the new one.

As the life of the disk evolves, there might not be enough space to store a file contiguously. In this case, the file is split into smaller chunks that are inserted into the free clusters spread over the disk. The effect of splitting a file into non-contiguous clusters is called **external fragmentation**. As we will see, this phenomenon reduces the performance of an HDD quite a lot, for this reason, operative systems like Windows offer a Disk Fragmenter, a tool that rearranges files stored on the disk so that they occupy contiguous storage locations and minimise the performance loss.

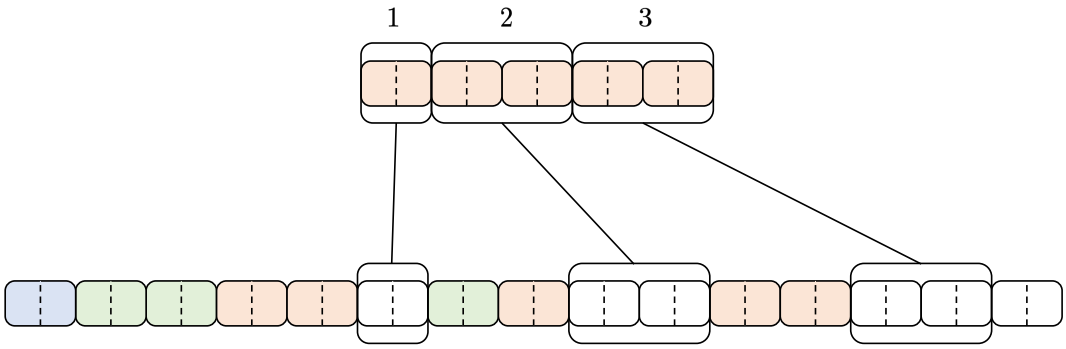


Figure 1.2: Example of external fragmentation as file needs to be stored in non-contiguous clusters.

1.2 Hard Disk Drives

A hard disk drive (HDD) is a non-volatile data storage device that uses rotating disks (platters) coated with magnetic material and moving actuator arms which read and write data to the platter surfaces. Data is read in a random-access manner, meaning individual blocks of data can be stored or retrieved in any order rather than strictly sequentially. HDDs are characterised by the number of platters, rotation speed, track density, number of disk surfaces (heads), data capacity, MTTF, average seek time and bandwidth among other parameters. It is important to define some key concepts: we call a **sector** a read/write atomic unit contained in a track, a **track** is the set of sectors who are at the same distance from the disk's rotation center (same radius) in a single platter, the set of tracks with the same radius in all platters is called a **cylinder**.

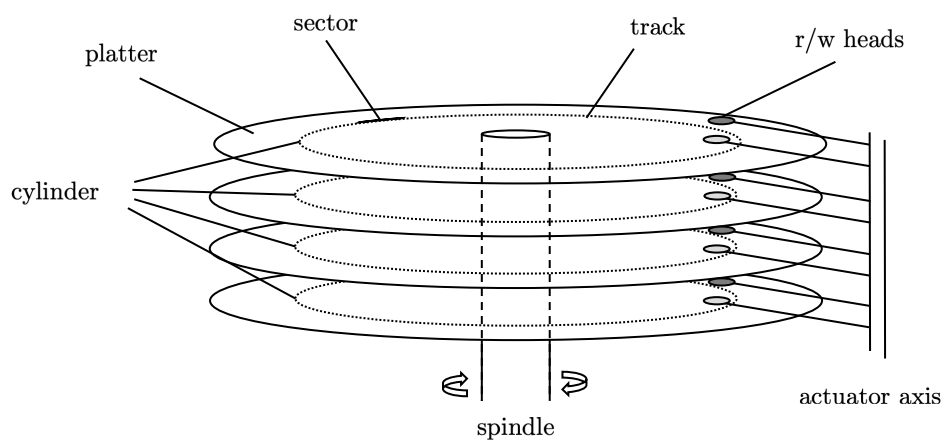


Figure 1.3: Structure of a hard disk drive

Each sector on a HDD is identified by three numbers (CHS):

- C - Cylinder (starting from 0)
- H - Head (from 0 to HPC-1)
- S - Sector (from 1 to SPT)

Where HPC are the heads per cylinder and SPT the sectors per track.

chap1_chsexample.pdf

It is important to highlight that clusters are normally a power-of-two multiple of the sector size and there is a function in the BIOS of PCs that convert LBAs into the three CHS numbers that uniquely identify the sector on the HDD.

$$LBA = (C * HPC + H) * SPT + (S - 1) \tag{1.3}$$

1.2.1 HDD Performance

There are two main concepts to measure performance, service time and response time. **Service time** considers only the execution of the request while **response time** considers both the time spent in queue waiting for the resource and the execution of the request.

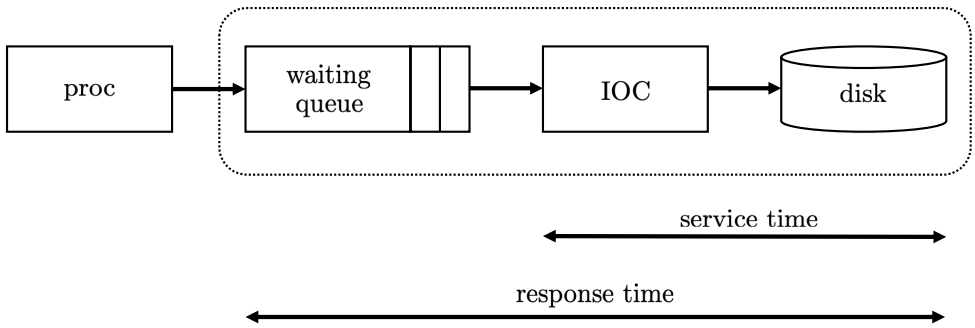


Figure 1.4: Response and service time

The service time can be expressed as:

$$\text{Service time} = \text{seek time} + \text{latency} + \text{transfer time} + \text{controller overhead} \tag{1.4}$$

where the **seek time** is head movement time and is a function of the number of cylinders traversed, **latency** is the time needed to wait for the sector, **transfer time** that depends on rotation speed, storing density and cylinders and the **controller overhead**.

R We normally approximate mean latency as the time taken to scan $1/2$ round

The response time is not as important as service time so it will not be covered in detail, it is enough to know that it depends on the number of requests in the waiting queue (queue length), the utilisation level of the resource, arrival rate, and inter-arrival times distribution. Of course, the mean disk service time increases as the variability increases.

Performance of an HDD is highly influenced by where the data is located. As we said before, files are normally stored in a contiguous way to decrease the needs of seek and rotational latency times. We call the **data locality** l the percentage of blocks that do not need seek or rotational latency to be found. The effect of locality can be expressed with the following equation:

$$T = (1 - l) * (T_s + T_l) + T_c + T_t \tag{1.5}$$

Let's suppose that data locality $l = 90\%$, with this equation we can observe that the seek and latency times are only considered for the remaining 10%, as in average, 9 out of 10 blocks do not require that the head moves to read the block as they are the "next-in-line" blocks to read.

1.3 Solid State Drives

Solid state drives or SSDs are non-volatile data storage devices that use integrated circuit assemblies as memory to store data persistently. They normally use the same form factor and communications protocols as HDDs but as opposed to HDDs, SSD have no mechanical parts as platters are replaced by NAND based memories and are therefore more physically robust.

1.3.1 Internal organization

The array of NAND cells of which SSDs are made of can be of various typologies: SLC, MLC, TLC, or QLC. The most common are SLC (Single Level Cell) that can store 1 bit and MLC (Multi Level Cell) that can store 2 bits of information, this twice the capacity is paid by reducing speed and endurance but fortunately it reduces cost too. This technology was actually what allowed SSD to reach the masses.

NAND flash is organised into:

- Pages: smallest unit that can be read or written. They can be in three states:
  empty: do not contain data.
  in use: contains data that can be read.
  dirty: contain data but it is no longer in use.

- Blocks (or Erase Blocks): smallest unit that can be erased, typically consisting of multiple pages.

A page may contain eight 512-byte logical block addresses (LBAs). A block typically consists of 64 pages with a total capacity of 256KB.
There does not exist a formula to convert LBAs to page indexes, as the conversion is done thanks to special **look-up tables** (LUT). By managing the mapping with tables we can dynamically re-map LBA to pages and move their content over by just changing their pointer.

In SSDs:
- Only empty pages can be used
- Only dirty pages can be erased but in order to erase it we must erase the entire block, meaning all the pages in the block must be either empty or dirty.
- If no empty page exists some dirty page needs to be erased.
- If no block contains just dirty or empty pages we need to follow special procedures to gather the empty pages over the disk.

1.3.2 Performance

SSDs have higher theoretical access speed ranging from 15-20 times faster than an HDD, however in some real contexts performance may be a lot worse than theoretical or even slower than HDDs.
Let's suppose we have a block made of 8 pages and the follow operations occur:
1. `File1` is written occupying 2 pages.
2. `File2` is written occupying 4 pages.
3. `File1` is deleted by marking the pages as dirty.
4. We want to write `File3` that need 4 pages to be stored.

The operative system sees that there are no 4 empty pages to save the file so it starts a procedure to free up space:

1. Reads the whole block of 8 pages and copies it into cache
2. Deletes the 2 dirty pages from cache
3. Copies `File3` to the cache.
4. The whole block on cache is written onto the SSD.

This whole operation that theoretically required the writing of 4 pages actually required the OS to read 6 pages and write 8 pages, this is known as *write amplification*.

Important definitions

- **P/E Cycles**, an endurance metric that establishes the number of times the NAND cells can be programmed and erased.

- **Write Amplification Factor (WAF)** are the bytes written to NAND versus bytes written from PC, more formally:

$$\text{write amplification} = \frac{\text{data written to the flash memory}}{\text{data written by the host}}$$

- **Terabytes Written (TBW)**, an endurance metric which describes how much data can be written to the SSD over its lifecycle.

$$\text{TBW} = \frac{\text{capacity GB / 1000 PE Cycles}}{\text{WAF}}$$

- **Unrecoverable Bit Error Ratio (UBER)** A metric for the rate of occurrence of data errors, equal to the number of data errors per bits read

$$\text{UBER} = \frac{\text{number of data errors}}{\text{number of bits read}}$$

TRIM

SSD have a TRIM command that enables the OS to notify the SSD which pages do not contain valid data. After trimming, the SSD will not preserve any contents of the block when writing new data to a page of flash memory. This command achieves fewer writes (resulting in less write amplification thus increasing drive life) and, as there is no need for a read-erase-modify sequence, a higher write throughput.

Overprovisioning

Overprovisioning is a tactic employed by some manufacturers to increase performance, it consists in the allocation of additional storage space to mitigate write amplification issues.
Normally manufacturers have at least a 7.37% overprovisioning due to the difference between binary and decimal conversions: a gigabyte (GB) is precisely 1,000,000,000 bytes, and one gibibyte (GiB) is precisely $2^{30} = 1,073,741,824$ bytes, or about 7.37% more than a GB, despite this extra storage, some manufacturers allocate even more memory and sometimes native 128 GiB drives are sold as 100 GB drives.

Mean read time

Considering that n_p pages need to be read and that T_{readP} is the time to read the page into the cache, $T_{transfP}$ is the time to transfer the page to OS and T_{cmd} is the time of the controller to handle a read command, the mean read time of an SSD is:

$$T_r = T_{cmd} + T_{readP} + T_{transfP} + (n_p - 1) * \max(T_{readP}, T_{transfP}) \tag{1.6}$$

This considering that when reading n_p pages, reading the page and its transfer to the OS can be done in parallel.

1.3.3 SSD vs HDD

There is a point, roughly after a thousand drive rewrites, that the SSD performance becomes worse than the hard disk drive. This is not the only drawback, actually SSDs can be written a limited number of times, and they suffer of the write amplification phenomenon. Therefore there is no answer to which type of drive is better, it depends on the situation of our drive (how many times has it been rewritten, etc) and the value-cost analysis as SSDs have smaller storage size for a significantly higher price.
To balance the advantages of both types of drives, some drive manufacturers produce Solid State Hybrid Disks (SSHD) that combine in a single unit a small SSD and a large HDD.

2. RAID

In the late 1980s, researchers at Berkeley, California published a paper showing that a redundant array of inexpensive disks (RAID) could outperform the top mainframe disks available at the time. In the original paper, authors suggested five RAID configurations that could lead to a significant increase in performance or reliability. Since then, many configurations have evolved, some hybrid variations have been created, some companies have created their own proprietary solutions, and even the acronym has changed to be: redundant array of independent disks, replacing the inexpensive word with independent.

These data techniques rely on mainly three mechanisms:

- **data striping** which is a mechanism that breaks data into pieces and writes such pieces across multiple disks. Where to place the pieces is this is done following an algorithm which is normally *round robin*. This mechanism allows for an increased *performance*.

- **data redundancy** which is a mechanism that keeps additional data in order to reconstruct damaged data. This mechanism allows for an increased *reliability*.

- **I/O virtualization** which is a mechanism that separates logical from physical resources, this allows to apply the previous two mechanisms on an array of disks while at a logical level all operations take place on a single large disk and all the procedures are completely transparent to the user.

The different RAID techniques are classified according to their redundancy technique (if any), how they distribute this redundancy technique among disks and how data is interleaved.
Data can be interleaved at a bit, byte or block level, we call *fine grained* when data is interleaved in small units (bit, byte) and *coarse grained* when it is interleaved in large units (block). Fine Grained is ideal for large files that require fast transfer while coarse grained is ideal for small files that are frequently accessed like a database.

2.1 RAID 0

This RAID configuration is characterised by striping and no redundancy. That means that data are written on a single logical disk and split in several blocks distributed across the disks according to a striping algorithm

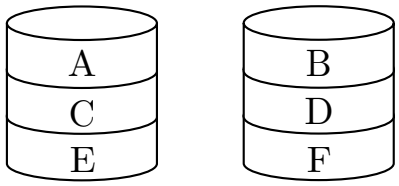


Figure 2.1: RAID 0

Min. number of Disks	2
Space efficiency	1
Usage	used where the primary concerns are performance and capacity rather than reliability
Key aspects	• data striping
Pros	• best write performance • lowest cost
Cons	• single disk failure results in data loss

This is the configuration used for high performance computing applications as in such contexts, speed and capacity are more important than reliability.

2.2 RAID 1

This RAID configuration is characterised by always having two copies of the data. Every time data is written to a logical disk, the data is written on two different physical disks so when a disk fails the second copy can be used. The problem with this approach is that we can save just half the data we could achieve without the mirroring mechanism as the second disk is an exact copy of the first and cannot be used to store different information.

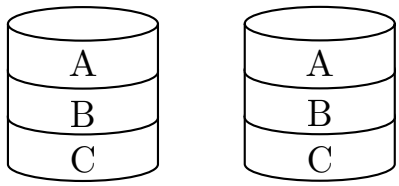


Figure 2.2: RAID 1

Min. number of Disks	2
Space efficiency	$1/n$ normally $n = 2$
Usage	used where the primary concerns are reliability and to a lesser extent performance, like databases
Key aspects	• data redundancy - mirroring
Pros	• good write performance • faster read performance
Cons	• high costs

Theoretically, a RAID 1 configuration can mirror the data to more than one disk to improve its resiliency to errors, but in practice this approach is never used as the overhead and costs are too high and there are better configurations with more disks.
This is the configuration used for database application that have a high transaction rate.

2.3 RAID 2

Is a rarely used configuration that assumes the failed component is unknown, stripes data at the bit (rather than block) level, and uses a Hamming code for error correction. It requires less redundant disks than mirroring, considering n data disks we need $log_2(n+1)$ redundant disks.

2.4 RAID 3

Is a configuration that assumes the failed component is known[1], it stripes data at the byte level, and uses a single redundant disk reserved for the parity bits.

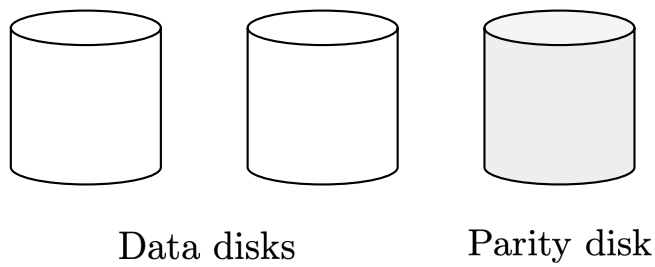


Figure 2.3: RAID 3

Min. number of Disks	3
Potential storage use	$1 - {}^{1}/_{n}$
Usage	used where high bandwidth but not high I/O rates are required
Key aspects	• byte level striping • dedicated parity
Pros Cons	• a single redundant disk • only one request can be served at the time

[1] This assumption is almost always verified as the disk controller is usually able to determine which one is the broken disk.

2.5 RAID 4

Data is interleaved across disks in blocks and there is a parity disk, each write must update the requested data blocks and must compute and update the parity block. However, since all the writes must go to the dedicated parity disk, this causes a performance bottleneck for all write operations.

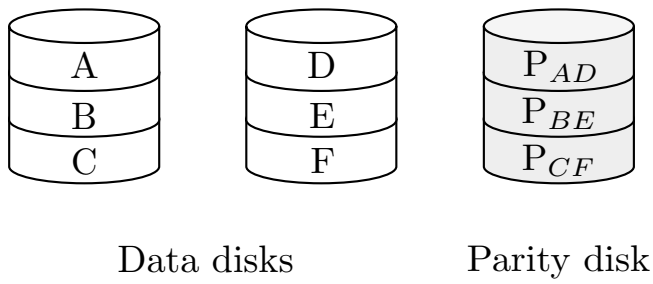


Figure 2.4: RAID 4

Min. number of Disks	3
Potential storage use	$1 - {}^{1}/_{n}$
Usage	works very well in the case of really large files
Key aspects	• block level striping • dedicated parity
Pros Cons	• fast reads • limited write speed

2.6 RAID 5

This RAID configuration interleaves data across disks in blocks just as RAID 4 but instead of having a single parity disk, parity information is distributed among the drives.

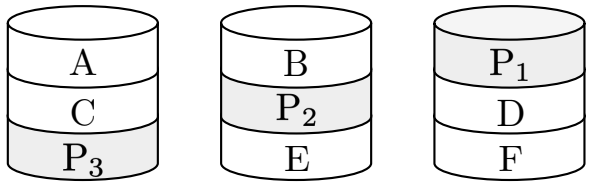


Figure 2.5: RAID 5

Min. number of Disks	3
Potential storage use	$1 - {}^{1}/_{n}$
Usage	where performance and reliability are important, TPS
Key aspects	• block level striping • distributed parity
Pros	• avoids parity disk bottleneck • faster reads • load balancing across the disks
Cons	• limited write speed

2.7 RAID 6

Until this point all the RAID configurations (except RAID 0) guarantee fault tolerance to a single drive failure, however for some applications a higher reliability is needed and we would want to tolerate the failure of not one but two disks. This configuration is like RAID 5 but adds another distributed parity block for a total of two parity blocks distributed across all the disks in the array. The larger the drive capacities and the larger the array size, the more important it becomes to choose RAID 6 instead of RAID 5 as the probability of encountering an URE (Unrecoverable Read Error) becomes more likely.

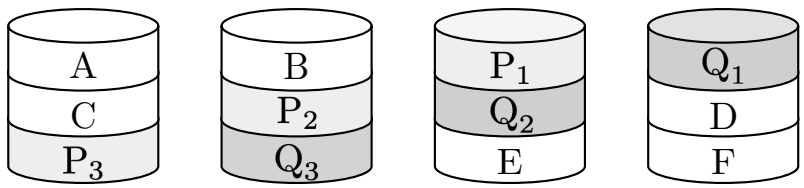


Figure 2.6: RAID 6

Min. number of Disks	4
Potential storage use	$1 - {}^{2}/_{n}$
Usage	used for environments that need long data retention periods of critical information
Key aspects	• block level striping • double distributed parity
Pros	• 2 concurrent failures are tolerated
Cons	• high overhead (slow writes)

RAID 5 uses the XOR function to compute the parity values from the data. For a RAID 6 it is not enough just to add one more XOR function. If two disks in a RAID 6 array fail, it is not possible to determine the location of the data blocks using only XOR function. Therefore in addition to the XOR function, RAID 6 arrays use a more complex function such as the *Reed-Solomon code* that produces different values depending on the location of the data blocks. The parity values are normally called P and Q and are calculated as follows:

$$P = D_0 + D_1 + D_2 + \ldots + D_{n-1} = \sum_{i=0}^{n-1} D_i \tag{2.1}$$

$$Q = D_0 + g \cdot D_1 + g^2 \cdot D_2 + \ldots + g^{n-1} \cdot D_{n-1} = \sum_{i=0}^{n-1} g^i \cdot D_i \qquad g \neq 1 \tag{2.2}$$

The value g is called *generator* and is normally g=2.

2.7.1 Update of parity

When writing a file we must do the following steps:

1. Read the data block D where we will write the file and the current parity values P and Q related to that block. Even if we do not care about the current content of the block as it will be overwritten with the new information, we need it to update the parity values.

2. Update P and Q with the following formulas:

$$\begin{aligned} P^{new} &= P^{old} + D^{new} - D_i^{old} \\ Q^{new} &= Q^{old} + g^i \cdot (D^{new} - D_i^{old}) \end{aligned} \tag{2.3}$$

3. Write the new information and the new parity values

2.7.2 Recovery techniques

When disks fail, we end up with different recovery situations, we now explore the different cases that may arise:

Parity P or/and Q fails

This is the simplest case as what is failing is simply the redundancy disks and they can be easily recomputed with formulas 2.1 and 2.2.

A data disk fails

If a single data disk failed, it can be reconstructed using the P values:

$$D_i = P - \sum_{j=0,\, j \neq i}^{n-1} D_j \tag{2.4}$$

A data disk and Q fail

This is also a simple case as we can recover the failed disk D with the formula 2.4 and after the data disk has been rebuilt we can simply reconstruct Q with its formula.

A data disk and P fail

We can recompute firstly the data disk D that failed using Q:

$$D_i = \frac{Q - \sum_{j=0,\, j \neq i}^{n-1} g^i \cdot D_i}{g^i} \tag{2.5}$$

and after the data disk has been rebuilt we can simply reconstruct P with its formula.

Two data disk fail

This is the more complex case, if two data disks fail we must write a system of equations. Let's call P^* and Q^* the parities computed without the the two failed disks D_i and D_j, we can then write the following system of equations:

$$\begin{cases} P = P^* + D_i + D_j \\ Q = Q^* + g^i \cdot D_i + g^j \cdot D_j \end{cases} \tag{2.6}$$

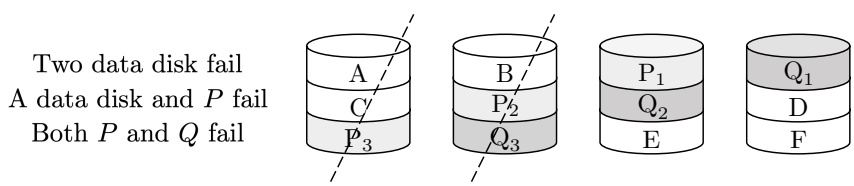
by solving the system we find that disk D_i can be reconstructed as:

$$D_i = \frac{g^{-i}(Q - Q^*) - g^{j-i}(P - P^*)}{1 - g^{j-i}} \tag{2.7}$$

However solving the system at each failure results very computationally inefficient, for this reason a table of two coefficients, A_{ij} and B_{ij} can be precalculated:

$$\begin{aligned} A_{ij} &= \frac{g^{-i}}{1 - g^{j-i}} \qquad B_{ij} = \frac{-g^{j-i}}{1 - g^{j-i}} \\ D_i &= A_{ij}(Q - Q^*) + B_{ij}(P - P^*) \end{aligned} \tag{2.8}$$

R Note that, since RAID 6 has data striped and parity values distributed, several of the previously considered cases can happen at the same time as we consider the cases stripe-wise:



To overcome the problem of having limited precision to store values while solving the previous equations, we use **Galois Fields** algebra, this allows the computations using only integer numbers that can fall inside a byte.

2.8 Hybrid RAID approaches

Hybrid RAID approaches usually combine two different RAID levels. We call RAID x+y or RAID xy an approach where we have n disks divided into m groups of G disks each and we apply RAID x to each group of G disks and we apply RAID y to the m groups, considering them as single disks.

2.8.1 RAID 0+1

RAID 0+1 or RAID 01 is a hybrid approach also called "mirrored stripes" that performs a mirror of a striped set.In other words, this configuration has a RAID 0 configuration within groups and a RAID 1 configuration between groups.

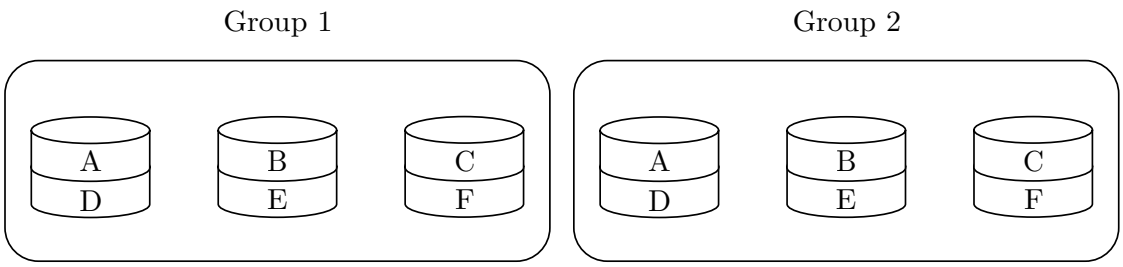


Figure 2.7: On RAID 0+1, no matter the number of disks, the number of groups will always be 2.

2.8.2 RAID 1+0

RAID 1+0 or RAID 10 is a hybrid approach also called "stripe of mirrors" that performs is a striping across a number of mirrored set . In other words, the configuration has a RAID 1 configuration within groups and a RAID 0 configuration between groups.

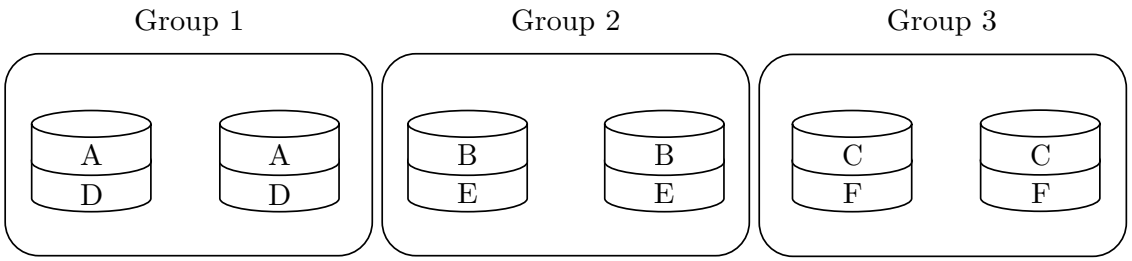


Figure 2.8: On RAID 1+0, no matter the number of disks, the number of disk in each group will always be 2.

2.8.3 Similarities and differences

Both RAID 0+1 and RAID 1+0 require a minimum of four drives to function, both have the same performance but RAID 1+0 has a better fault tolerance as in a RAID 1 group the RAID controller switches to the mirror drive of the disk that has gone down, thus reducing the probability of total loss when a drive has already gone down while a RAID 0 group gets immediately unusable if one of its disks goes down.

2.9 Data storage systems

In the data storage context, there are many technologies that have the goal to improve the persistence and availability of data, examples of such technologies are DAS, NAS and SAN.

2.9.1 DAS

Direct Attached storage (DAS) is simply storage directly attached (i.e. not via a network) to a system, like internal hard drives and external drives connected with a point-to-point protocol such as USB or eSATA. Unless the disk is accessed by block numbers, it needs a file system on top of it to be properly used.

2.9.2 NAS

Network-attached storage (NAS) was born to achieve easy file sharing with other computers on a network. NAS typically provides access to files using network file sharing protocols such as NFS (Network File System), SMB (Server Message Block), or AFP (Apple Filing Protocol). Even though any computer can share files over a network and could be called NAS, we normally call NAS such devices whose main purpose is to share files over a network. NAS units usually contain one or more hard disk drives in a RAID configuration, they have a stripped-down operating system like FreeNAS, and have associated their own IP address.
NAS provides good scalability as we can increment the storage devices in each NAS unit or simply increment the number of NAS units. NAS performance is variable and depends mainly on the speed of and congestion on the network.

2.9.3 SAN

Storage Area Network (SAN) is a computer network which provides access to consolidated, block-level data storage. This means that SAN does not provide file access but only block-level operations, everything related to the file system is left up to the client.

Most storage networks use the SCSI protocol for communication between servers and disk drive devices, it is a low level protocol that is used to send commands to controllers or disks and is encapsulated into secondary or transport protocols (iSCSI - SCSI over Ethernet, FCP - SCSI over Fibre Channel).

SANs have two distinct networks, one TCP/IP and one dedicated network that reduces the overhead and increases the performance (e.g. Fiber Channel). Moreover PCs require special interfaces to connect to the SANs and interfaces are quite expensive.

2.9.4 Differences between NAS and SAN

A disk available through NAS appears as a file server while a disk available through SAN appears to the client OS as a disk, allowing it to be formatted with a file system and access volume utilities.

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

While in a NAS architecture data is transferred using the TCP/IP protocol over Ethernet, SANs often have their own network and storage devices, which have to be bought, installed, and configured. This makes SANs inherently more expensive than NAS architectures.

3. Dependability

According to the IEC, dependability is the term that encompasses several performance attributes such as reliability, maintainability and maintenance support performance to describe the availability of a system.
Dependability cannot be described by deterministic laws and is normally approached with statistical techniques. In the next pages we formally define such concepts.

3.1 Reliability

Reliability can be defined as $1 - Unreliability$, that is the probability that a component does not fail in a certain interval $[0,t]$ given the fact that the component is working properly at time $t = 0$.
Given $f(t)$ the probability density function of the failure, $F(t)$ the CDF of $f(t)$, and T the instant of the first failure occurs, we can define reliability as:

$$R(t) = 1 - F(t) = P(T > t) \tag{3.1}$$

But how is $f(t)$ calculated? What we need to do is to perform **empirical evaluation**, that is, empirically test many identical elements in the same conditions. Suppose we have n_0 elements and $n(t)$ measures how many elements are still working at time t.

If we take $n(t)/n_0$ and as $n_0 \to \infty$ it converges to the value $R(t)$

We need to be careful as we cannot compare reliability values at different time values. For example:

1. $R(10000) = 0.95$
2. $R(20000) = 0.90$

cannot be compared, even if we want the desired lifespan of the system to be 10000 hours, we only know that the second option has a reliability greater than 0.9 at 10000 hours but we do not know if it is lower or higher than 0.95

If we assume that the component is working at time t instead of 0 we call **failure rate** the

probability of a component failure at instant t.

$$\lambda(t) = \frac{f(t)}{R(t)} = -\frac{1}{R(t)}\frac{dR(t)}{dt} \tag{3.2}$$

This failure rate can also be interpreted as the number of faults per unit of time and hence measures the speed of occurrence of a fault.

3.1.1 Failures

The failure of an element can be due to several reasons and they normally appear predominantly in certain periods rather than on the whole lifecycle. We normally represent the variation of failure rate with the lifetime of the element with a curve known as the **bathtub curve**.

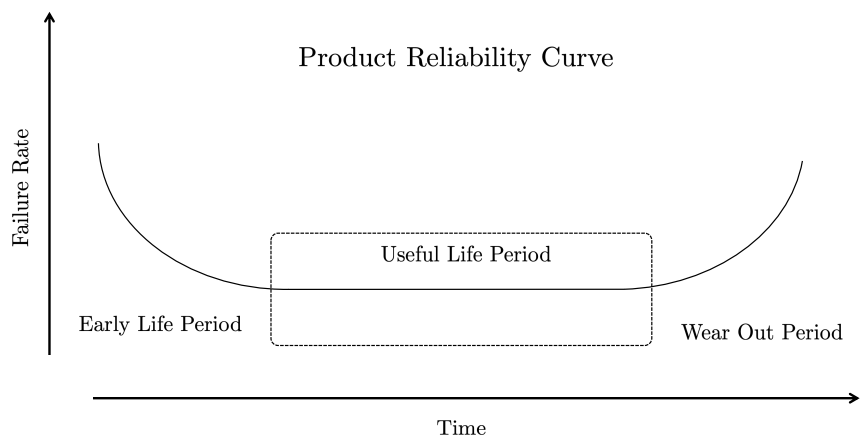


Figure 3.1: Product Reliability Curve or Bathtub Curve

In this curve we can identify 3 phases:

- **Early Life Period** that corresponds to the period at the beginning of the lifecycle of the product, in this phase we can normally attribute failures to *design failures* in hardware or software and to *infant mortality*.
- **Useful Life Period** that corresponds to the period where there is a low constant failure rate and failures are mainly *random failures*.
- **Wear Out Period** that corresponds to the period that ends the lifecycle of the product, as some components start to wear out due to prolonged use and fail eventually making the whole product fail.

To simplify things, from now on we assume a constant failure rate, that is, that as time goes on we are not increasing the probability of failure (we concentrate on the flat region of the bathtub curve) and we assume that failure rate and repair time are independent and exponentially distributed.

3.1.2 Metrics

Reliability can be quantified by mean of some key metrics:

- **Mean Time To Failure** (MTTF), describes the expected time to failure of a system.

$$MTTF = \mathbb{E}[f(t)] = \int_0^\infty t \cdot f(t)dt \qquad \text{but also} \qquad MTTF = \int_0^\infty R(t)dt \tag{3.3}$$

under the hypothesis of exponential distribution we have:

$$\begin{gathered} F(t) = 1 - e^{-\lambda t} \qquad f(t) = \lambda e^{-\lambda t} \qquad R(t) = e^{-\lambda t} \\ MTTF = \mathbb{E}[f(t)] = \frac{1}{\lambda} \end{gathered} \tag{3.4}$$

- **Mean Time To Repair** (MTTR) is a metric that states the mean time between the failure and the recovery from a failure of one component, sometimes also called *downtime*. It includes the time required to discover default, repair it and reset the system.

- **Mean Time Between Failures** (MTBF) is a metric that states the mean time between two failures.

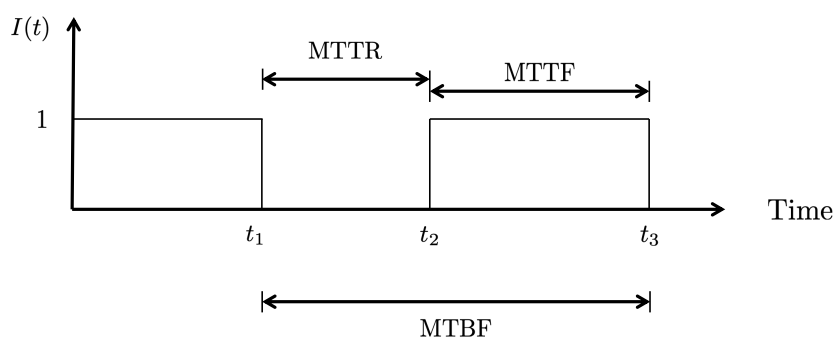


Figure 3.2: Visual representation of reliability metrics

- **Mean Time To Data Loss** (MTTDL) is a metric that states the mean time required to have a number of disk failures that cause a loss of data in the array, i.e., such that it is impossible to reconstruct the lost data from the redundant information (e.g., 1 in a RAID 0, 2 in a RAID 5, 3 in a RAID 6).

R With formulas 3.4 we can compute the probability that a component with MTTF fails at least once in a certain period of time t

$$F(t) = P(T \leq t) = 1 - e^{-\frac{t}{MTTF}} \qquad \text{if t/MTTF} << 1 \text{ with Taylor Expansion} \qquad = \frac{t}{MTTF}$$

3.1.3 System Reliability

We normally care about the reliability of a system rather than a single component, obviously to increase reliability we can introduce multiple redundant components, use only very reliable components with **high MTTF** and components that are repairable in small amount of time **low MTTR**, however the reliability of the system can be greatly improved through an appropriate configuration of the several elements. This configuration can be represented though **reliability block diagrams**.

Reliability block diagrams have two types of connections:

- **Serial connections**, this type of connections require that *all* components are working properly. Suppose we have n components in a serial connection, each one with a reliability $R_i(t)$, to calculate the probability of the system to work properly at time t we need that all components are working properly at time t. Thanks to the independence of components (every component fails independently) we can write:

$$R_{Series}(t) = \prod_{i=1}^{n} R_i(t)$$

or its dual $F(t)$ that represents the probability of failure before t

$$F_{Series}(t) = 1 - \prod_{i=1}^{n} \underbrace{P(x_i > t)}_{\substack{\text{probability that a} \\ \text{component fails after t}}} = 1 - \prod_{i=1}^{n}(1 - P(x_i \leq t)) = 1 - \prod_{i=1}^{n}(1 - F_i(t)) \tag{3.5}$$

- **Parallel connections**, this type of connections require that *at least one* component is working properly. Suppose we have n components in a parallel connection, each one with a reliability $R_i(t)$, to calculate the probability of the system to work properly at time t we need that at least one component works properly at time t.In other words all situations are acceptable except those where all components have failed. Thanks to the independence of components (every components fails independently) we can write:

$$R_{Parallel}(t) = 1 - \underbrace{\prod_{i=1}^{n}(1 - R_i(t))}_{\substack{\text{probability that} \\ \text{all components fail}}} \tag{3.6}$$

or its dual $F(t)$ that represents the probability of failure before t

$$F_{Parallel}(t) = \prod_{i=1}^{n} F_i(t)$$

MTTF

But how is MTTF modified when putting components in series and in parallel?

If we consider the series case we have:

$$F_{Series}(t) = 1 - \prod_{i=1}^{n}(1 - F_i(t)) = 1 - \prod_{i=1}^{n}(1 - (1 - e^{-\lambda_i t})) = 1 - \prod_{i=1}^{n}(e^{-\lambda_i t})$$

$$= 1 - e^{-(\lambda_0 + \lambda_1 + \ldots \lambda_{n-1})t}$$

recalling $\lambda = 1/MTTF$

$$MTTF_{Series} = \frac{1}{\lambda_{series}} = \frac{1}{\lambda_0 + \lambda_1 + \ldots \lambda_{n-1}} = \frac{1}{\frac{1}{MTTF_0} + \frac{1}{MTTF_1} + \ldots \frac{1}{MTTF_{n-1}}} \tag{3.7}$$

and in the case of **identical components** $\forall i \quad \lambda_i = \lambda$:

$$MTTF_{Series} = \frac{MTTF}{n} \tag{3.8}$$

If we consider the parallel case we discover that the distribution of parallel components are non-exponential (even if the components are):

$$F_{Parallel}(t) = \prod_{i=1}^{n} F_i(t) = \prod_{i=1}^{n}(1 - e^{-\lambda_i t})$$

but in the case of **identical components** $\forall i \quad \lambda_i = \lambda$:

$$MTTF_{Parallel} = MTTF\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + 1\right) \tag{3.9}$$

as in $(0, t_1)$ we have n working components, at time (t_1, t_2) we have $n-1$ and so on until we have just one in the interval (t_n, t_{n-1})
in the case of **two different components**:

$$MTTF_{Parallel} = MTTF_1 + MTTF_2 - \frac{1}{\frac{1}{MTTF_1} + \frac{1}{MTTF_2}} \tag{3.10}$$

3.2 Availability

Availability represents the probability that at a given instant of time the system is working and can be computed as the fraction of time the system works properly divided by the total elapsed time.

$$A = \frac{MTTF}{MTTF + MTTR} \tag{3.11}$$

that at a given instant of time the system is computed as the fraction of time during which the system works correctly .

Availability	Uptime	Yearly Downtime	Daily Downtime
1 Nine	90%	36.5 days	2.4 hr
2 Nines	99%	3.65 days	14 min
3 Nines	99.9%	8.76 hr	86 sec
4 Nines	99.99%	52.6 min	8.6 sec
5 Nines	99.999%	5.25 min	0.86 sec

Figure 3.3: The "table of nines", a table normally used to describe the availability of a system

3.2.1 Difference between Availability and Reliability

While availability represents the uptime, reliability represents the probability that a system works properly for a desired period of time without failure.

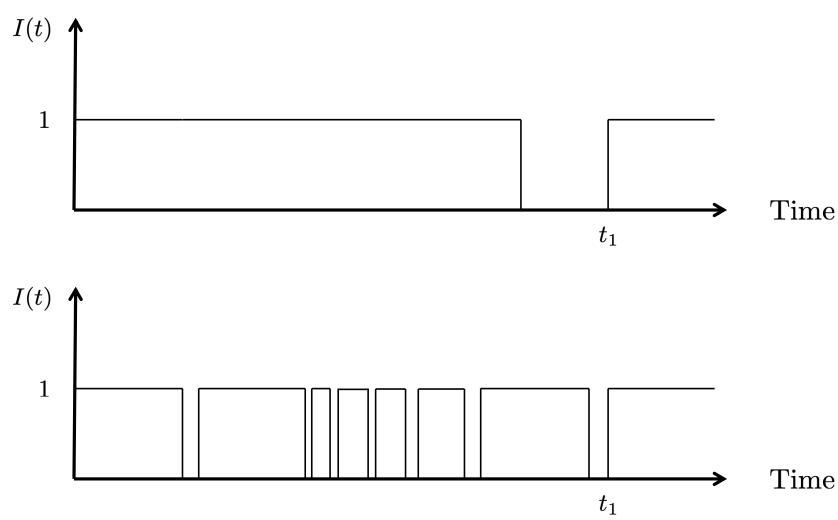


Table 3.1: Both systems have the same availability as they have the same uptime, but they have different reliability, in fact, the system at the topmost part of the image has a better reliability as the time without a failure is bigger

3.2.2 RAID reliability

Assuming that the times to failure are exponentially distributed we can approximate the probability that a set of n disks is still working after t days as:

$$F_{\min(X_1,X_2,\ldots,X_n)}(t) = 1-(1-F(t))^n \simeq \frac{n \cdot t}{MTTF} \tag{3.12}$$

RAID 0

In case of a RAID 0 configuration to cause a data loss a single disk must fail, we can estimate the failure rate of n disks as:

$$MTDL_{RAID0} = E\left[F_{\min(X_1,X_2,\ldots,X_n)}(t)\right] \simeq \frac{MTTF}{n} \tag{3.13}$$

In other words MTTDL does not depend on MTTR for this RAID configuration

RAID 1

In case of a RAID 1 configuration with two disks, for data loss both disks must fail, meaning that the second disk must fail before the repair is performed on the first disk that failed.

$$P(RAID1) = P(1^{st} fail)\, P(2^{nd} fail < MTTR) \tag{3.14}$$

Estimating $P(1^{st} fail)$ as $2/MTTF$ as the number of disks can be two, we can write

$$P(RAID1) = \frac{2}{MTTF} \cdot \frac{MTTR}{MTTF} \tag{3.15}$$

therefore

$$MTDL_{RAID1} = \frac{1}{P(RAID1)} = \frac{MTTF^2}{2 \cdot MTTR} \tag{3.16}$$

R If we consider a RAID 1 configuration with n disks we can write

$$MTDL_{RAID1} = \frac{1}{P(RAID1)} = \frac{MTTF^n}{n \cdot MTTR^{n-1}} \tag{3.17}$$

RAID 5

Suppose a RAID 5 configuration has n disks, we can then write the probability of data loss as:

$$P(RAID5) = P(1^{st} fail)\, P(2^{nd} fail < MTTR) = \frac{n}{MTTF} \cdot \frac{(n-1) \cdot MTTR}{MTTF} \tag{3.18}$$

$$MTDL_{RAID5} = \frac{1}{P(RAID5)} = \frac{MTTF^2}{n \cdot (n-1) \cdot MTTR} \tag{3.19}$$

RAID 6

In a RAID 6 configuration for the system to fail, three disks must fail, considering a configuration with n disks, we can then write the probability of data loss as:

$$\begin{aligned} P(RAID6) &= P(1^{st} fail)\, P(2^{nd} fail < MTTR)\, P(3^{rd} fail < MTTR/2) \\ &= \frac{n}{MTTF} \cdot \frac{(n-1)\cdot MTTR}{MTTF} \cdot \frac{(n-2)\cdot MTTR}{2\cdot MTTF} \end{aligned} \tag{3.20}$$

R We have to use MTTR/2 since during the 3rd failure, there are two disks being repaired, and thus the failure has to happen before the repair of any of the disks, if we assume the second failure happens after MTTR/2 of the first failure, the other disk has to fail before MTTR/2 period of time.

$$MTDL_{RAID6} = \frac{1}{P(RAID6)} = \frac{2\cdot MTTF^3}{n\cdot(n-1)\cdot(n-2)\cdot MTTR^2} \tag{3.21}$$

RAID 0+1

$$P(RAID01) = P(1^{st} fail)\, P(\text{a disk fails in mirror group}) = \frac{n}{MTTF} \cdot \frac{n/2 \cdot MTTR}{MTTF} \tag{3.22}$$

$$MTDL_{RAID01} = \frac{1}{P(RAID10)} = \frac{2\cdot MTTF^2}{n^2\cdot MTTR} \tag{3.23}$$

RAID 0 hybrids

Any RAID configuration having RAID 0 groups has a MTTDL that can be calculated with the following formula:

$$MTDL_{Array} = \frac{MTDL_{Group}}{\text{number of groups}} \tag{3.24}$$

RAID 1+0

$$P(RAID10) = P(1^{st} fail)\, P(\text{mirror disk fails}) = \frac{n}{MTTF} \cdot \frac{MTTR}{MTTF} \tag{3.25}$$

$$MTDL_{RAID10} = \frac{1}{P(RAID10)} = \frac{MTTF^2}{n\cdot MTTR} \tag{3.26}$$

alternatively with the 3.24 and recalling a RAID 10 has always groups with two disks we find the same formula:

$$MTDL_{RAID10} = \frac{\frac{MTTF^2}{2\cdot MTTR}}{n/2} = \frac{MTTF^2}{n\cdot MTTR}$$

RAID 5+0

Let's suppose we have n groups each of G disks, with 3.24 we can write:

$$MTDL_{RAID50} = \frac{MTDL_{RAID5}}{n/G} = \frac{\frac{MTTF^2}{G\cdot(G-1)\cdot MTTR}}{n/G} = \frac{MTTF^2}{n\cdot(G-1)\cdot MTTR} \tag{3.27}$$

RAID 6+0

In a similar way as RAID 5+0 we can calculate the mean time to data loss of RAID 6+0 as:

$$MTDL_{RAID60} = \frac{MTDL_{RAID6}}{n/G} = \frac{2\cdot MTTF^3}{n\cdot(G-1)\cdot(G-2)\cdot MTTR^2} \tag{3.28}$$

4. Data Center

A data center or data centre is nothing more than a facility composed of computers and storage connected by a network that are used by organisations to process and store data. The dimensions of a data center can vary a lot depending on the number of servers in it, its dimensions can range from a single room to those bigger than a football field, indeed the biggest datacenter to date (July 2019) located in Inner Mongolia, China has a surface of 1 km^2, that is roughly 45 city blocks in Manhattan.

4.1 Data center components

Data centers have a complex architecture with several areas and components associated to different purposes:

- **Cooling system** is one of the most important components of the data centre, it is composed by coolers, heat-exchangers and cold water tanks and aims to keep the temperature between 17-28° to guarantee an optimal, hassle-free environment for components that reside in the data centre. Indeed, if components overheat, they are susceptible to a temporary malfunction or permanent failure, and we want to avoid this at all costs. The most effective mean to reduce the temperature are water tanks units, but due to their excessive cost, a data centre just has a few and are used as a backup for the better cost-effective turbo coolers.

- **Power Backup** is made of batteries and diesel generators that protect the data centre against blackouts and power failures, usually batteries can provide half an hour backup while diesel generators can provide backup for a longer period of time but they have a high 20 min startup time, actually batteries are just intended to provide enough energy for the time required to start the diesel generator.

- **Racks** or cabinets are special shelves where we can position several modules of a standard form factor, these modules can be servers, communication equipments, storage units or power distribution units. Each rack has a standard size and their are normally classified according to

their capacity of *rack units* U, the most common ones are $42U$ racks, mainly because their height is nearly 2m which is accessible to most persons.

- **Server corridors** is the location where **racks** are located in, these corridors are organised in a way that fresh air comes from the basement to the *cold aisle* and then is forced to flow through the servers and go to the *warm aisle* after cooling the server. A computer room air conditioning (CRAC) unit positioned near the corridors monitors and maintains the temperature, air distribution and humidity in the room.

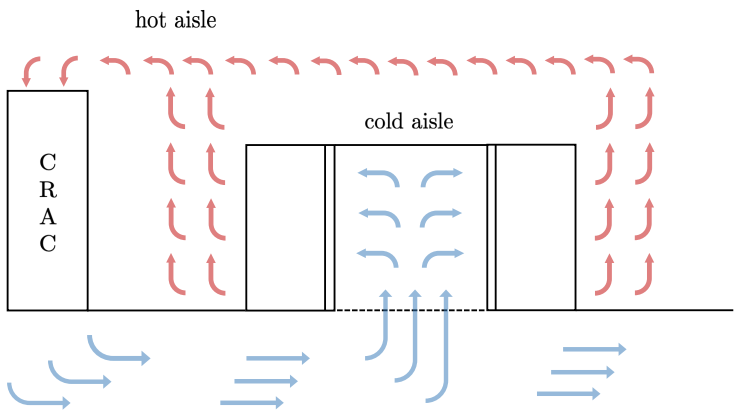


Figure 4.1: Cold Aisle Containment (CAC).

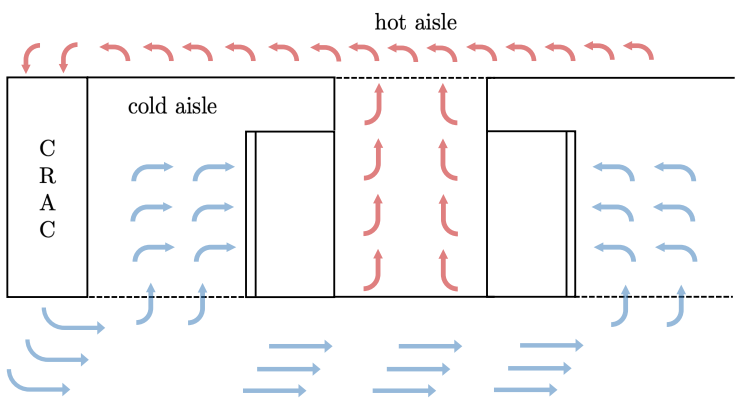


Figure 4.2: Hot Aisle Containment (HAC).

While cold aisle containment is easier to implement and is typically less expensive, hot aisle containment is usually more effective.

4.1.1 Rack modules

Each rack module has not necessarily the size of a rack unit. They can occupy more than one slots and can be half rack units (half the width of a standard unit). As we said before, modules can be:

- Servers, that are the main processing equipment, they can have different CPUs, GPUs, RAM and locally attached disks.
- Communication equipment modules, that allow network interconnections among the devices, they are usually Hubs, Routers, DHCP, Load balancers, Switches, Firewalls, and the like.
- Storage units, that can be JBOD (Just a Bunch Of Disks), RAID controllers or NAS heads.
- A power distribution unit, that is nothing more than a set of plugs that provides energy to the rack and usually has additional features such as energy consumption monitoring and remote power on/off.

4.2 Network architectures

The different servers of the data center are interconnected by a network, how is this network designed in order to guarantee an efficient access to it? There are three main network architectures.

4.2.1 Three Layers

In this configuration we have three layers: **core**, **aggregation**, and **access**. This three layer architecture reflects perfectly the topology of a datacenter.

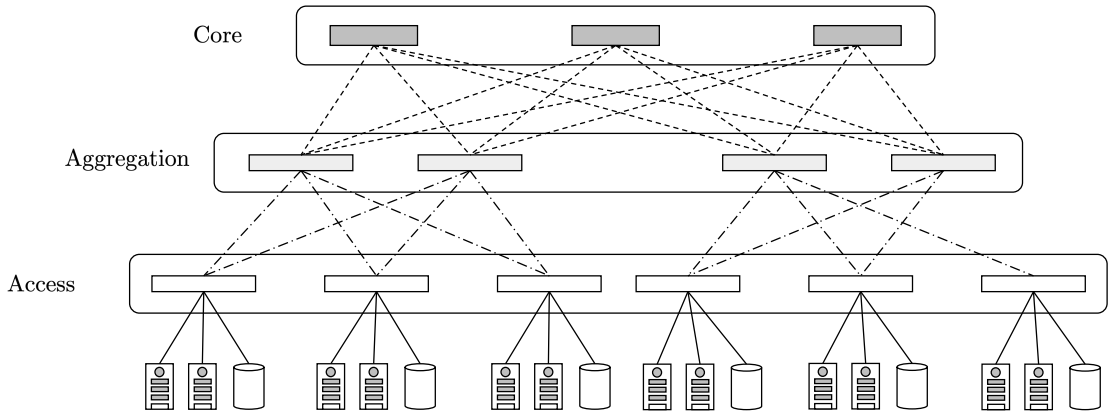


Figure 4.3: Three Layer architecture

Switches in the access layer can be position on two different parts of the rack:

- Top-of-The-Rack (TOR): Access switches are put at the top of each rack. The number of ports per switch are limited, this allows to connect a few compute and data nodes and its scalability is very limited.

- End-of-The-Line (EOL): Switches are positioned one per corridor, at the end of a line. The number of ports per switch are larger thus the system can scale to have a larger number of machines. It has longer cables and as a consequence we incur into higher costs.

In this kind of network, The upper the layer, the faster the network should be and the higher the costs in terms of energy consumption and investment required.
In such networks *bandwidth* can be increased by adding more switches at the core and aggregation layers, and by using routing protocols such as Equal Cost Multiple Path (ECMP) that shares the traffic equally among different routes. Despite all this, three-tier architecture is unable to handle the growing demand of cloud computing.

4.2.2 Fat-tree

Instead of having a smaller number of faster switches, this architecture uses many slow switches and connections (cheaper). The computing nodes are divided into *pods* characterised by the same number of both switches and computing units, each pod contains, $(k/2)2$ servers, $k/2$ access layer switches, and $k/2$ aggregate layer switches that are connected to one or more core switches. Each of the core switches is connected to one aggregate layer switch in each pod.

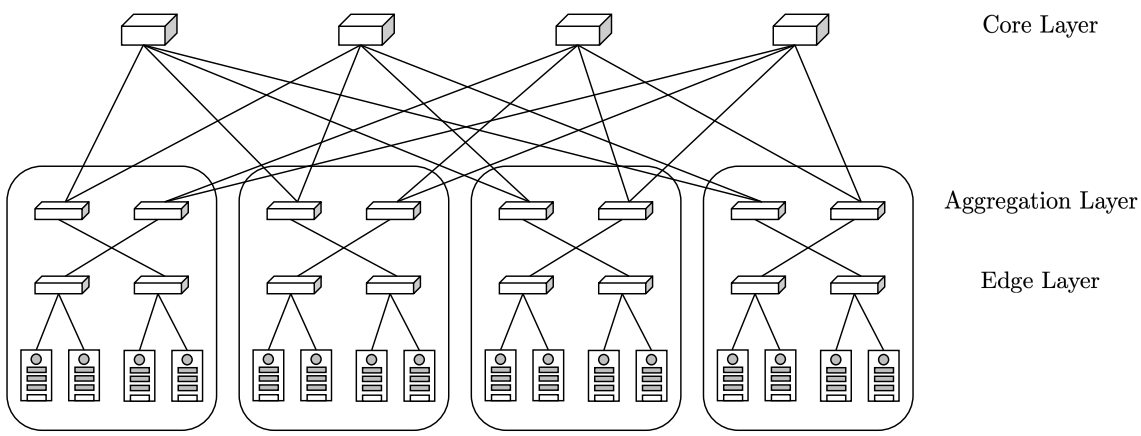


Figure 4.4: Fat-tree architecture

4.2.3 D-Cell

The D-Cell defines the network in recursive way. A $DCell_0$ is the basic unit and building block which contains n servers and one commodity (cheap and slow) network switch. The network switch is only used to connect the server within a cell0. A $DCell_1$ contains k=n+1 $DCell_0$ cells, a $DCell_2$ contains k=n+1 $DCell_1$,... This architecture is highly scalable, a four level DCell with $n = 6$ servers in a $DCell_0$, can host 3.26 million servers and it requires fewer cable and switches than the Fat-tree architecture.

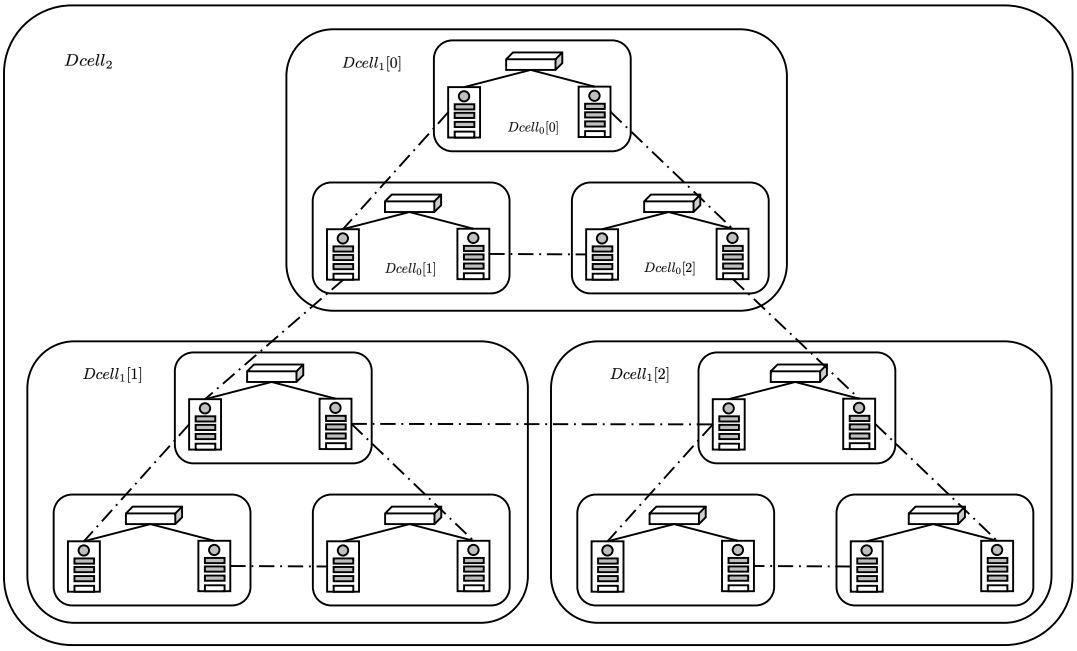


Figure 4.5: D-cell

4.3 Efficiency metrics

4.3.1 PUE

The Power Usage Effectiveness is a ratio that describes how efficiently a computer data center uses energy.

$$\text{PUE} = \frac{\text{Total Facility Energy}}{\text{IT Equipment Energy}} = 1 + \frac{\text{Non IT Facility Energy}}{\text{IT Equipment Energy}} \tag{4.1}$$

4.3.2 Quality Tier

The Uptime Institute defines four tiers according to the level of availability and redundancy of a data center:

- Tier I - Basic site infrastructure (non-redundant)
  99.671% Uptime and no redundancy

- Tier II - Redundant capacity components site infrastructure (redundant)

Tier I + 99.749% Uptime and partial redundancy in power and cooling.

- Tier III - Concurrently maintainable site infrastructure
  Tier II + 99.982% uptime, N+1 fault tolerance and provides at least 72-hour power outage protection.

- Tier IV - Fault tolerant site infrastructure
  Tier III + 99.995% uptime and all cooling equipment is independently dual-powered.

5. Virtualization & Cloud Computing

Cloud computing's popularity has skyrocketed in the last few years, it consists on having computing resources, mainly data storage and processing speed, offered over a network like the internet rather than physically having all the resources by yourself. Cloud providers exploit the benefits of economies of scale, and allow customers to minimise up-front infrastructure costs and maintenance, in addition, it also allows customers to handle and adapt to fluctuating and unpredictable peaks of demand as resources are dynamically allocated. Examples of cloud providers are AWS, Microsoft Azure and Google Cloud.

Cloud computing has several service models, mainly:

- PaaS (Platform as a Service) where vendors offer a development environment to application developers.
- SaaS (Software as a Service) where the vendor provides applications running on a cloud infrastructure.
- IaaS (Infrastructure as a Service) where the vendor provides storage, networks, and other fundamental computing resources where arbitrary software like operating systems and applications can be deployed and ran.

Virtualization is the fundamental technology that powers cloud computing. This technology separates the computing environments from the underlying physical infrastructures.

In this chapter we will study how virtualization is achieved though virtual machines. Before starting to study in depth this technique we must first understand what a machine is and what are the machine levels, once done this we will introduce virtualization and the two different types of virtualization.

5.1 Machine's layers of abstraction

A *machine* is an execution environment capable of running a program, a car, a washing machine, a computer, and the like. To fully understand a machine such as a computer, we must identify some levels of abstraction.
In a computing system we can identify 6 levels of abstraction:

- Level 5 - Problem Oriented Language - A high-level programming language such as C, C++, or Java.
- Level 4 - Assembly Language - Assembly is a very detailed language, very fast and powerful in accessing various features of the hardware.
- Level 3 - Operating System Machine - Provides a convenient interface between assembly language and the abstraction of the hardware architecture's instruction set. Operating systems generally contain many libraries that help a software developer or programmer connect to lower-level system functions (e.g., I/O, memory allocation, etc.) in an organized way.
- Level 2 - Instruction Set Architecture (ISA) - The ISA includes anything programmers use to make a binary machine language program work correctly, including instructions, I/O, etc. The ISA facilitates design of functionality independent of the hardware.
- Level 1 - Microarchitectural Level - Microinstructions are low-level control instructions that define the set of datapath control signals which apply to a given state of a computing machine. The microinstructions, together with their sequencing, comprise the microarchitecture, whose purpose is to rigorously and consistently express the control of logic circuits that comprise the computer hardware. Designing this control in terms of a program that implements machine instructions in terms of simpler microinstructions is called microprogramming.
- Level 0 - Digital Logic - Everything is expressed in terms of functions of ones and zeros with `and`, `or`, and `not` functions.

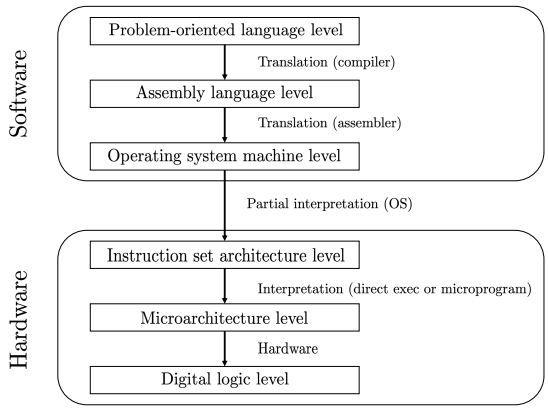


Figure 5.1: Six Layers of abstraction

5.1.1 ISA

The **instructions set architecture (ISA)** is the level that marks the division between hardware and software. As a consequence we consider levels 0-2 as hardware and levels 3-5 as software. The ISA is composed by:

- **System ISA** that comprises all the instructions related to the interactions between the supervisor software (the OS) through Drivers, Memory Management, and Scheduler with the hardware.
- **User ISA** that comprises all the instructions that allow interaction from the application with the hardware (read one data from memory, sum, logical operations, branches, etc.).

5.1.2 ABI

The **application binary interface (ABI)** is the interface between the application that provides a program access to hardware resources through the OS, it corresponds to level 3 in the layered model. The ABI is composed by:

- **System calls** that allow programs to interact with shared hardware resources in an indirect way through the OS
- **User ISA** that comprises all the aspects of interaction from the application with the hardware (read one data from memory, sum, logical operations, branches, etc.).

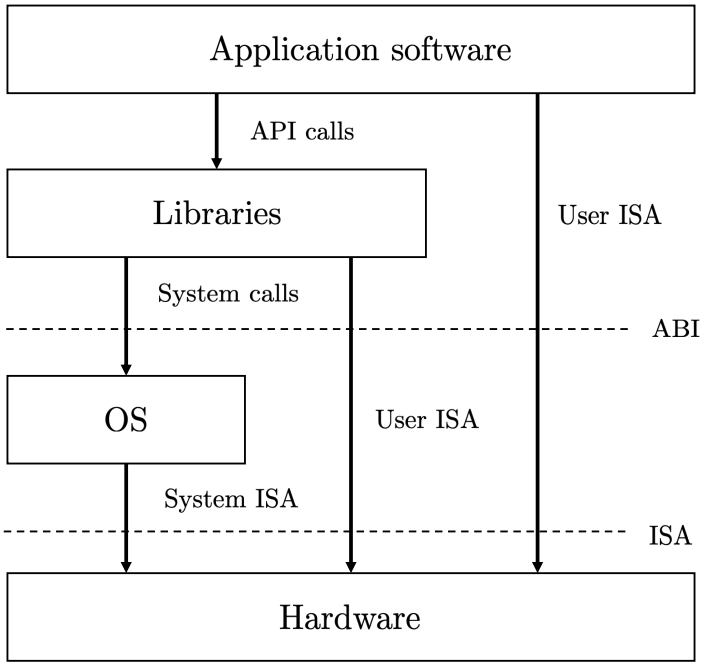


Figure 5.2: API and ABI

5.2 Virtual Machines

A Virtual Machine (VM) is a logical abstraction able to provide a virtual execution environment or a full system environment that emulates the behaviour of another machine. In the VM; terminology we call the actual hardware running the VM the **'host'**, and the virtual machine emulated on that machine the **'guest'**.

JRE

There are several technologies such as Java that are able to run a program on whatever architecture is able to support it, in the case of Java it can run on every architecture that has an interpreter called the Java Runtime Environment (JRE), also addressed as the *Java Virtual Machine*, this virtual machine interprets and executes the Bytecode instructions, a new machine language that does not use specific architecture instructions. The specific JRE implementation for a particular OS such as MacOS, Windows, Linux and Android makes possible the execution within that OS.

Emulator

Thanks to the JRE, Java can run on multiple OS, but what about other languages such as C? Is there any way of executing a program on different hardware from the one it was written for?
According to the Church-Turing thesis, under the assumption that enough memory is available, any operating environment can be emulated within any other environment.
There are special pieces of software called *emulators* that provide the ability to emulate another program or device by reproducing the behaviour of the system it was created for. It understands the behaviour of the CPU, the RAM and all the I/O devices connected to the emulated system and **reproduces not only the ABI but the whole ISA (system and user)**.

The examples given are actually two different solutions for a similar problem, let's formally classify this solutions as a specific type of virtual machines.

5.2.1 Process VM

A process VM, usually called *runtime environment* or *runtime software*, runs as a normal application and supports a single process. This kind of VM **emulates the ABI**: the user-level instructions and the operating system calls, for this reason it is placed on top of the OS/hardware combination. It models the way the machine as seen by processes.

It consists of:

- a logical memory address space assigned to the process
- user-level registers
- an interpreter that allows the execution of the application code.

The Java JRE is an example of process VM.

5.2.2 System VM

A system VM, usually called *full execution environment* or *Virtual Machine Monitor (VMM)*, supports simultaneously different processes and an operating system. This kind of VM **emulates the ISA**: both the user-level instructions and the system level instructions, it can be placed on top on the underlying hardware alone, or on top of another OS. It models the way the machine as seen by the OS.

It allocates physical memory (all the memory available) and I/O resources to the processes that interact with their resources via an OS that is part of the system.
A Space Invaders Emulator is an example of system VM.

5.3 Types of virtual machines

The execution of instructions through emulation can be a slow process and its performance is usually worse compared to an equivalent physical machine. This is due to the fact that instructions of the software need to be mapped to those of the real physical system. However, in some cases, the language executed by the VM might be the same as the one used by the underlying system, in this case, the runtime software or the VMM can exploit this opportunity to speed-up the execution or emulation task.
Windows can be run on a Mac very efficiently because both have the same hardware architecture (Intel x86) and thus share the same ISA.

We can further subdivide the two types of VM according to the compatibility of interface between the host and the guest:

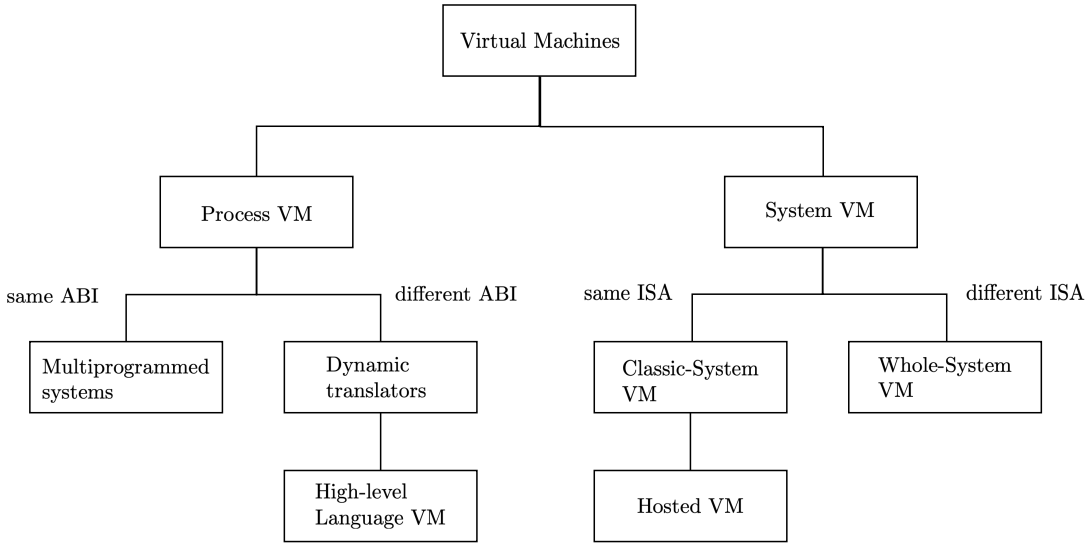


Figure 5.3: Taxonomy of VM

We talk about *emulation* when we have to deal with a different ABI or ISA.

Process VM

5.3.1 Multiprogrammed systems

It is a virtual machine formed by OS call interface + user ISA (ABI) that is commonly used by all modern OS for multi user support.

Each user process is given the illusion of having a complete machine to itself, it is the OS that timeshares and manages the hardware resources to allow this. Actually the process is just given its own address space and is granted access to a file structure. One example is the Task Manager.

5.3.2 Dynamic translators

Are emulators developed to allow an application (or OS) to run in an environment different from that originally intended. Given the different ABI, to emulate one instruction set designed for another hardware we have to make use of:

- **Interpretation** the interpreter program fetches, decodes, emulate the execution of individual source instruction, but it is normally a slow process.
- **Binary Translation** is a process that achieves a better performance by using code profiling and converting at runtime blocks of source instructions with *Just-in-time compilers*. Once a block of instructions is translated, the translated instructions can be cached and repeatedly executed much faster than interpreted.

High-level Language VM

The goal of this virtual machine is to have an isolated execution environment for each application (or instances of the application). It translates application byte code to OS-specific executable minimizing HW/OS-specific features. This technique makes applications run almost normally, except they are not "installed" in a strict sense, its execution is sandboxed, they can "migrate" from one runtime environment to another runtime environment and they are not conflicting one another.
The main example of these types of VM is the Java Virtual Machine.

System VM

5.3.3 Classic VM

A virtual machine which is running directly on top of the hardware (bare machine). This is the most efficient type of virtualization. The VMM must be able to catch all the guest OS's interactions. This is the most popular type of virtualization for cloud computing purposes.

Hosted VM

In a hosted virtual machine, the virtualizing software is on top of an existing host operating system. This is the most popular type of virtualization for consumer purposes.

5.3.4 Whole-System VM

As the ISA is different, we need to emulate the whole execution environment, and need to rely on techniques such as binary translation. This was a popular type of virtualization during the 1990s-2000s where Mac computers relied on the PowerPC architecture instead of the Intel x86 family architecture and there was the need of running Windows applications on such systems.

5.4 Hypervisors

System virtual machines dealing with the same ISA are the most important kind of virtual machines as they are the type of virtual machines that are used for cloud computing. These kind of virtual machines have an application called Virtual Machine Manager (VMM), Hypervisor or Virtual Machine Monitor are all that manages the virtual machine by mediating access to the hardware resources on the physical host system and intercepting and handling any privileged or protected instructions issued by the virtual machines.

5.4.1 Type I Hypervisor

Type I, or *bare-metal hypervisors* are hypervisors that takes direct control of the hardware.

In the **monolithic architecture** the device drivers and I/O Services are handled **within the hypervisor**. It is less efficient in terms of distribution and maintenance as it can only run on hardware for which drivers exist, but it has better performance and isolation.

In the **microkernel architecture** we use a special virtual machine running on top of the hypervisor that provides the access to all the shared I/O devices, a **service VM**. Being drivers an external service, we have a smaller hypervisor and we can use third party drivers without recompiling the whole hypervisor.

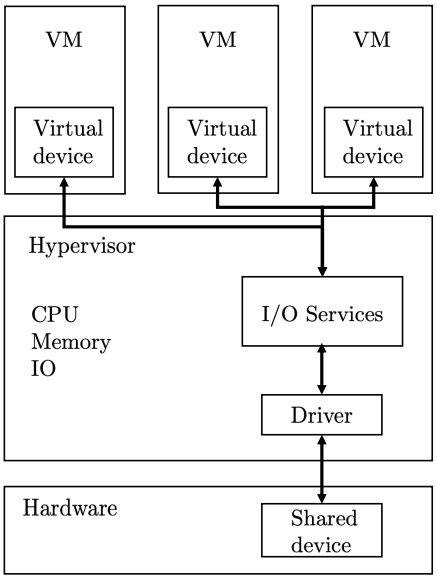


Figure 5.4: Monolithic Architecture

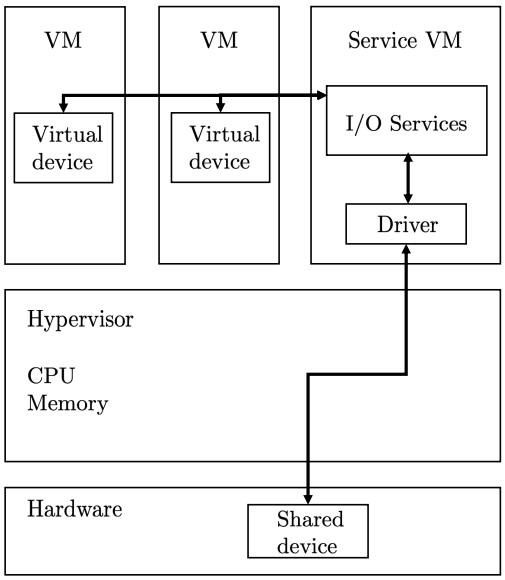


Figure 5.5: Microkernel Architecture

Sometimes Type I Hypervisor can be used to provide hacks to some OS. An hypervisor may mimic vendor specific hardware that allows the installation of pre-activated versions of Windows or may allow MacOS on non-Apple hardware by emulating the EFI of an Apple computer on a standard PC.

5.4.2 Type II Hypervisor

Type II or *hosted hypervisors* are hypervisors that reside within a host operating system and leverage code of the host OS such as scheduling, memory management. The host OS is the OS that is really able to control the hardware of the system, while the guest OS is the one that runs in the VM. This type of hypervisor are easier to configure as normally the host OS provides a GUI to configure the guest OS. Nevertheless, there might be conflicts between the Host OS and the Guest OS and the Host OS might consume non-negligible physical resources.

To understand what type of hypervisor you should use, you must ask yourself, will the hardware be used only to support virtualization? If the response is positive (usually when dealing when Data centers), we should go for Type I Hypervisor, otherwise for a laptop and other consumer/non-demanding application we should go for Type II.

5.5 Virtualization technologies

5.5.1 Virtualization implementations

Virtualization is implemented by by adding layers between execution stack layers, depending on where the new layer is placed, we obtain different types of virtualization.

- **Hardware-level virtualizatio**n
  A virtualization layer is placed between the hardware and the operative system in order to make the Operative System perceive a different interface from the real, physical one.

- **Application-level virtualization**
  A virtualization layer is placed between the operative system and the applications. Applications run in their environment, independently from OS.

- **System-level virtualization**
  The virtualization layer allows a secondary OS to run on top of an existing OS. This type of virtualization allows multiple OS on a single physical machine. E.g. VirtualBox, VMware. The virtualization layer is a software! but it is a special software as it has access to privileged instructions.

5.5.2 Properties of virtualization

- **Partitioning** we can partition the resources and allow the execution of multiple OS on a single physical machine.
- **Isolation**, it has an advanced resource control to guarantee performance managed by the hypervisor and fault tolerance and security at the hardware level.
- **Encapsulation** it allows to save the entire state of a virtual machine in a file that can be copied or moved.
- **Hardware independence** the VM does not depend on the hardware it can be moved from one machine to another. This is normally used to migrate VM on servers.

5.5.3 Virtualization techniques

Paravirtualization

Is a technique that presents VMs an interface which is similar, yet not identical to the one of the underlying hardware, it provides some *hooks* to allow the guest and host to request and acknowledge tasks which would otherwise be executed in the virtual domain (where the execution performance is worse). Paravirtualization requires the guest operating system to be explicitly ported for the para-API, so it cannot be used with a traditional OS. However it has a simple VMM and a higher performance.

Full Virtualization

In full virtualization VMs are provided with the full instruction set, input/output operations, interrupts and memory accesses. With this technique, traditional OS can be run, however there is the need of hypervisor mediation for the interception and simulation of privileged operations, such as I/O instructions. In addition, full virtualization is possible only with the right combination of hardware and software elements.

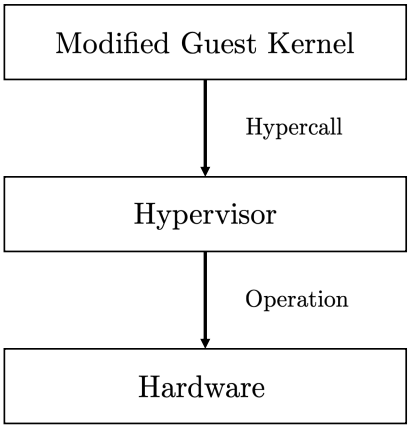


Figure 5.6: Paravirtualization

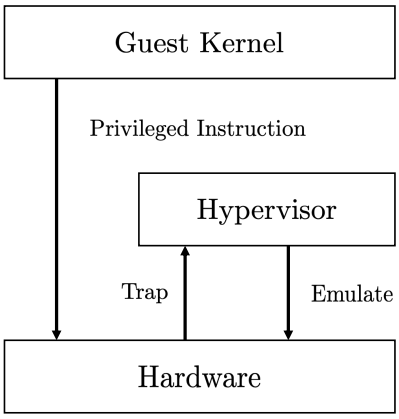


Figure 5.7: Full Virtualization

Kernel-level virtualization

Virtualization happens at Operating System-level, sometimes called OS Virtualization (very light VMs). There is no VMM or Hypervisor, no guest OS, just private Servers virtualized on top of the operating system. For example KVM (Kernel-based Virtual Machine) is a virtualization module in the Linux kernel that allows the kernel to function as if it were an hypervisor.

PROs: Native OS performances and Native hardware support through host

CONs: Single OS and host kernel modifications required

5.6 Virtual resources

The main contribution of a VMM is handling virtual resources. It must provide the Guest OS the interface of a real resource (e.g. a network card) understand the requests issued by the Guest OS to the resource (e.g. when it is trying to access the internet) and emulate the requests on the host hardware (physically sending the request on the network, remembering that such packet was issued by the corresponding VM).

What allows virtualization is a virtualization layer that maps virtual resources to a set of managed resources that may be physical or in the case of stacked virtualization layers, virtual resources. This mapping can be *One to One*, meaning that one virtual resource corresponds to a single physical resource, *Many to One (Sharing)* when multiple virtual resources correspond to one physical resource or *One to Many (Aggregation)* when one virtual resource corresponds to multiple physical resources.

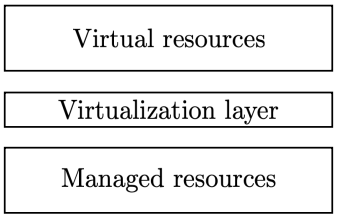


Figure 5.8: The Virtualization layer

Virtual resources can provide virtual machines two types of interfaces to the physical resources: the same one (virtualization) or a different one (emulation). Providing the same interface very convenient as the VMM has a simpler task and a higher performance can be achieved, however it is not always possible. On the other hand emulation achieves compatibility, interoperability and flexibility at the cost of having a limited performance.

5.7 CPU Virtualization

Processors must be able to execute the instructions (system and user-level) sent by the guest OS. The execution must be emulated (binary translation or interpretation) if the ISA are different or can perform Direct Native Execution if the ISA is the same with a considerable boost in performance.

5.7.1 Core partitioning

Each VM can be assigned a number of physical cores of the physical machine. Although in principle a VMM can simulate more cores than the ones actually available, this is a bad practice as it can be only achieved by time-sharing a core and it represents a reduction in the performance. However this doesn't mean that a VMM cannot run simultaneously several VMs using a total number of cores greater than the one available on the physical machine, this is just usually done when we have idling VMs and cores can be appropriately time-shared.
We can limit the time a core is assigned to a VM by setting an *execution cap* that specifies the percentage of time the VM can run. This can be used to limit the CPU resources assigned to a VM, allowing to runa larger number of VMs on the same physical machine. It can however create timing problems on the Guest OS.

5.8 Memory Virtualization

5.8.1 RAM virtualization

The VMM can assign a portion of the physical memory to each VM, it is good practice to make sure that the sum of the memory allocated for VMs is less than the total memory available on the system, however due to poor memory utilisation we normally incur on, it is possible to assign VMs more memory than the one physically available. This can be done through:

Memory Overcommit
Is a technique that can be applied on any Guest OS. In this technique the VMM tries to determine the pages that are less used, to use their space (guessing how much memory the OS is using)

Memory Swapping
Is a technique that is not unique to virtual environments can be applied on any Guest OS. In this technique **virtual memory** is created on an attached disk, which emulates physical memory, frequently accessed pages are held in physical memory, while infrequently accessed pages reside mostly in storage. When the CPU identifies through a trap technique that is trying to access virtual memory, the OS determines which page to free in exchange of the needed memory page (we call it is *swap*), so the page can be copied into physical memory so the CPU can access it.

Memory Balooning
Is a technique that can be applied on Guest OS that support it (has installed the 'balloon driver'- paravirttualization).
It works like follows:

1. Virtual Machine A needs memory, and the hypervisor has no more physical memory available.
2. the VMM interrogates the Guest OSs of other VMs and finds out Virtual Machine B has some unused allocated memory
3. The Balloon driver on VM B 'inflates' and makes some unused memory available to the VMM.
4. The VMM makes this memory balloon available to VM A.
5. Once there is more physical memory available, the balloon on VM B 'deflates'

Memory swapping is supported by the CPU that uses a **trap technique** that identifies when a process tries to access a portion of memory that is not in RAM, however trap alone is not enough to support virtual memory. The CPU must also have a special unit called Memory Management Unit (MMU) that separates the process address space from the physical address space. The MMU has a special data structure that is called the Page Table where it contains for each page index the physical address of the page, a flag that tells whether the page is in RAM or disk another one that tells if the page has been changed from the last saving and yet another flag that tells if the page is free or if it is being used.

The addition of another level in the memory hierarchy (virtualization) forces us to adopt additional mechanisms for memory management. There are two main approaches:

Shadow pages
Is a software solution to the problem, in it the VMM maintains a shadow page table for each of the guest VMs, that maps Virtual Memory Pages of VMs to physical memory of the host. It eliminates one level of indirection, from virtual → real → physical to simply virtual → physical. VMM needs to keep Shadow Pages Table in sync with the Guest OS Page Table and this sync usually introduces a lot of overhead.

Nested pages
Is a hardware solution to the problem (thus requires hardware that supports it), in it there is no overhead like the one on shadow paging. It works by using an additional table, the Nested Page Table (NPT) that is used to translate guest real addresses to system physical addresses: a TLB in a MMU.

5.8.2 Virtual disks

Virtual disks save the entire content of an HDD into a file. *Fixed size formats* immediately create a file whose size corresponds to the size of the HDD while *variable size formats* create disks that start with a reduced size, and can grow up to the total HDD size. In this last format compression is obtained by dividing the space into blocks, and by storing only the used ones. All formats allow to create snapshots by storing the differences with the previous versions.

5.9 I/O Virtualization

There are a number of different types of I/O devices, to better understand how they are handled we classify the devices in:

- Dedicated devices such as a keyboard and mouse. Requests are usually routed through the VMM.
- Partitioned devices such as disks. A big disk can be partitioned into smaller virtual disks.
- Audio devices. The Legacy audio card is emulated, in addition register, DMA access, and I/O are trapped by the VMM.

5.10 Network Virtualization

In virtualisation environments, each virtual machine has normally a *virtual network adapter* (VNA) for communication between other virtual machines or with the network itself, the virtual network adapter emulates a network card and can be of several types. The virtual machine manager can also attach more than one VNA and it can perform routing among the cards, and accept connections through different interfaces.
Virtual network adapters have a critical role in cloud computing where we normally have multiple virtual machines running on a single physical machine as virtual network adapters allow virtual machines and the host to share the same connection yet operating as separate entities. Since all VMs are operated and controlled though the network, this increases the importance of these virtualised components.The different VNAs can operate in several modes, we mainly identify: Network Address Translation (NAT), Bridged networking, Host networking and Internal networking. Each one of them has a different way of communicating with the host and with the external world. We will now analyse in depth each of these modes.

5.10.1 Network Address Translation

Network Address Translation or NAT for short is the simplest way a virtual machine can access an external network. The VM acts pretty much like a normal computer that connects to the Internet through a router but in this case the router is in the networking engine of the virtual machine manager.

From the point of view of the *host*, the VMM is treated as any other application that can act either as a client or as a server whilst from the point of view of the *guest*, the OS of the virtual machine sees a virtual network with a gateway, a DNS and a given subnet mask.

One downside of this approach is that the virtual machine remains invisible and unreachable from the outside unless port-forwarding is done, in other words, if a computer wants to communicate with a particular application of a virtual machine, there must be a correspondence between the host port and the guest port of a virtual machine (*forwarding table*) so when someone sends information to a particular host port, all the packets entering that port will be forwarded to a specific port of a specific virtual machine as specified on the forwarding table. Obviously as the correspondence between host OS port and the tuple (VM, guest OS port) must be unique, the same port on two different VMs must then be mapped to two different ports on the guest OS as a consequence **NAT might use a lot of ports** (we only have 65,535!).

5.10.2 Bridged Networking

Bridged Networking allows the guest OS (i.e. the VM) to send and receive packets directly from a physical network adapter on the host. When using bridged networking, the virtual machine is a full participant in the network, it can contact or be contacted by other machines on the network as if it were a physical computer on the network.

A virtual machine must have its own identity on a bridged network, on a TCP/IP network, the virtual machine needs its own IP address. For this reason a DHCP should be present, or static addresses should be configured, as a consequence bridge networking **consumes IP addresses** and is assigned an IP in the same network of the host's IP.

Bridge filtering also uses a special network feature of the Host OS called *Net Filter* that blocks or allows packets at a network interface based on source and destination addresses. Traffic produced by a VM is sent both to the network where the adapter is connected, and to the Host machine thanks to the Net Filter interface, this makes the Host machine perceive the traffic produced from inside a VM as traffic coming from the associated network adapter.

5.10.3 Host-only Networking

Host-only Networking allows exclusively the connection between the host and the guest virtual machine and subsequently Virtual Machines are not directly connected to the external network. The communication between the host and the guest is made possible thanks to a special virtual network card. Although this characteristic of being unreachable from an external network has the advantage of being more secure, if we want to allow guests to connect to the external network, routing mechanisms should be configured manually on the host.

5.10.4 Internal networking

Internal networking creates a network that connects only the VMs, they cannot communicate with the host or the outside world. Even though this is the most secure configuration, it is also the most limited as the VMs cannot contact a real server. They are normally used for testing complex network configurations.

Summary

Suppose we have the following IP addresses: Host1 = x.y.z.w, Host2= s.t.u.v, VM1 = a.b.c.d, VM2 = e.f.g.h. and we want to communicate to port p, how does the communication takes place between the virtual machines and hosts?

VN Type	H1 to VM1	VM1 to H1	VM1 to VM2 ($H_1 = H_2$)	VM1 to VM2 ($H_1 \neq H_2$)
NAT	x.y.z.w :q *	x.y.z.w :p	x.y.z.w :q *	s.t.u.v :q *
Bridged	a.b.c.d :p	x.y.z.w :p	e.f.g.h :p	e.f.g.h :p
Host-only	a.b.c.d :p	x.y.z.w :p	e.f.g.h :p	not possible
Internal	not possible	not possible	e.f.g.h :p	not possible

Table 5.1: In the table * indicates the need of port forwarding to the appropriate port mapped to the virtual machine

5.11 Server consolidation

Consolidation means to run several virtual machines on a single hardware. In a data centre we have many servers, instead of having a single virtual machine manager we gave a single VM framework that allows to manage independent physical machines (that is exactly what we want for cloud computing!). This allows us to be scalable! Given the fact that virtual machines can be moved as files, we can transfer them to another physical machine without interrupting the execution of the applications inside (live migration). We can do some load balancing on the servers and find the optimal allocation of the virtual machines on physical machines to maximise the performance. We can even introduce the concept of redundancy as VMs are protected against component and system failures.

Server consolidation also other advantages, higher hardware utilisation is achieved meaning less hardware is needed due to the increased efficiency and better management costs, apart from this, we can still continue to use legacy software thanks to virtual machines and the application is independent of the hardware.

Suppose you have a service and you allocate 2 servers and 1 database for the initial phase. As it gets popular the number of users that require the service is greater than the available users we can serve with the expected level of service that we want to provide. As a result we will need to introduce new servers. Cloud computing makes the introduction of new hardware pretty easy as the hardware is already available and the servers can be allocated dynamically. Even though cloud computing has some advantages, it clearly some disadvantages, it usually is very expensive and it has to be thoroughly analysed in terms of cost-benefits, maybe buying your own servers may be more convenient on the long run, and even if it may not seem so, cloud computing alternatives can lead to unexpected costs as they are paid on a pay-per-usage basis.

5.12 Cloud Computing

The current meaning of the term was coined in 2006 by Google's CEO Eric Schmidt. The term however has been used as a marketing buzzword and there is no unique definition, however we can define cloud computing as a paradigm, a computer model where resources like CPU, GPU, storage, network and other services are provided as a general utility that is paid on a consumption-basis just like water, electricity and gas.

According to the NIST definition cloud computing is a model for enabling convenient and on-demand network access to a shared pool of configurable computing resources, like for example networks, servers, storage, applications, services that can be rapidly provisioned and released *with minimal management effort* or service provider interaction. In the NIST definition there are 3 main services provided by a Cloud: PaaS (Platform as a Service), SaaS (Software as a Service), IaaS (Infrastructure as a Service). However there have been a myriad of things offered as a service (aaS) by companies like MLaaS (Machine Learning), IDaaS (Identity), VaaS (Voice), and many others.

5.12.1 Cloud Computing Ontology

We will now analyse each one of the ontologies of cloud computing that aims to highlight through a layered approach the inter-dependency between the different layers and its components.
As we will see, even if from the point of view of the user the interaction is with a single cloud computing service provider, the service provider might be itself a customer of another service provider of a lower service layer.

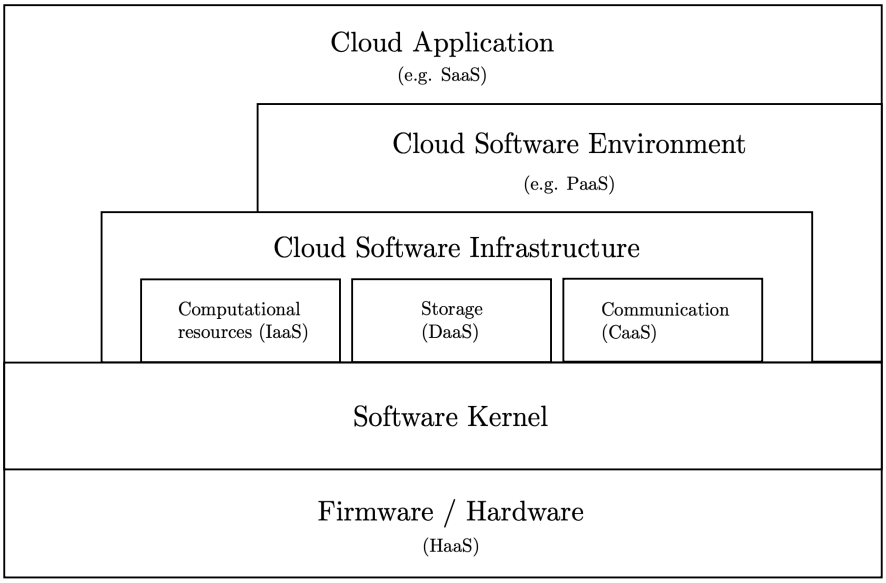


Figure 5.9: A proposed cloud computing ontology by researchers of University of California, Santa Barbara and IBM Watson Research Center.

Cloud Application Layer

The cloud application layer is the highest abstraction of the cloud computing model, users of this kind of services are common civilians and the access normally happens through web-portals or phone applications. Sometimes the service is free* or sometimes they are required to pay fees for their usage. Examples of services of this layer are Gmail and Google Docs.

Cloud Software Environment Layer

In this kind of layer the users are not civilians but developers who are supplied with a programming-language-level environment with well-defined APIs that facilitate interaction between environment and apps, accelerate the deployment and support scalability. Examples in the domain of Machine and Deep Learning are: Amazon Machine Learning (AML), Microsoft Azure Machine Learning and Google AI: TensorFlow among others.

Cloud Software Infrastructure Layer

This layer aims to provides resources to the higher-level layers made up of three main components:

- **IaaS** or Infrastructure as a Service, is computation as a service though VMs. Its advantages lies on the flexibility and fine granularity adjustment and customization for installed software thanks to super-root access. However it also has some disadvantages like performance interference and inability to provide strong guarantees. (e.g. Amazon Elastic Cloud, Windows Azure and Open Stack)

- **DaaS** or Data as a Service, is a paradigm that offers users the possibility to store data on remote disks and access it whenever and wherever the user desires. It facilitates the scalability of cloud applications as it guarantees high dependability: availability, reliability and performance, replication and data consistency. (e.g. DropBox, iCloud, GoogleDrive)

- **CaaS** or Communication as a Service is a vital component for cloud computing, it provides a configurable, schedulable and predictable network while guaranteeing a reliable, secure and monitorable communication.

It is possible that Cloud Application and Cloud Software Environment bypass this layer but doing so would reduce simplicity.

Software Kernel Layer

This layer provides basic software for the management of physical servers: OS kernel, Hypervisor, VMM and clustering middleware.

Hardware Layer

This layer provides, operates, manages, and upgrades the hardware on behalf of its customers. For the life-time of the sublease, it is responsible of everything related to the hardware it provides: maintenance, cooling, repair...

5.12.2 Types of clouds

- **Public** is large scale infrastructure available on a rental basis. However when data is sensitive public cloud is not the answer...

- **Private** the company sets up a virtualisation environment on its own data center with the objective of gaining some of the advantages of virtualisation while still having total control over every aspect of the infrastructure. It obviously is not flexible as in other types of cloud as there are not “almost infinite” resources.

- **Community** a single cloud managed by a community or several federated organisations. This idea is based on the principle of scale economy: combining together several organisations to achieve its advantages. Resources can be shared and used by one organisation, while the others are not using them in these cases a more complex accounting system is required.

- **Hybrid** clouds are the combination of any of the previous types. Usually such clouds are of companies that hold their private cloud, but due to unpredictable peaks of load, the company rents resources from other types of cloud.

Part Two

6 Capacity Planning 65

6. Capacity Planning

In this second part we will study capacity planning. Capacity planning is a set of activities that are performed in order to predict whether there will be enough power to manage the future load of a system. By power we normally mean computational power as this is the domain that we will consider but it also can be done considering electrical power and other types of power.

A possible approach to tackle the problem of whether there will be enough power or not is to do *overkill*, that is, buy a lot of servers to guarantee there will be enough power. Even though the overkill approach is effective it is possible to achieve a similar performance while saving a lot of money through a different approach: *modelling*. Through modelling it is possible to build a model that is capable of forecasting the behaviour of the system in an approximate manner, even if the model has some differences with respect to the real behaviour of the system, it is 'good enough'. There are a lot of models that can be used to forecast the behaviour, in the following pages we will see only a subset of *queue network models*: separable queuing networks.

6.1 Queue Network Models

Queue network models are a type of model that can be used to forecast the behaviour of a system, we can mainly distinguish two types of models: *open models* and *closed models*.

Before the description of the models we give first some definitions that help describe the phenomena of the system:

Definition 6.1.1 We define T as the period of time the system is monitored.

Definition 6.1.2 We define A as the number of requests arrived into the system during the monitored time.

Definition 6.1.3 We define C as the number of completed requests during the monitored time.

Obviously by definition $A < C$ is impossible, but it can happen that $A > C$, this means that there are some requests that are still unfulfilled (maybe the request is still on queue).

Assumption 6.1.1 STEADY-STATE ASSUMPTION

$$A = C$$

we assume that in average there is a kind of equilibrium between the received and served requests. In a real model this is only true for very large T.

Definition 6.1.4 We define the *arrival rate* λ as the speed the requests are arriving into the system during the monitored time.

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

dimensionally the arrival rate is measured in requests/second.

Definition 6.1.5 We define the *throughput* X as the speed the system of completes requests during the monitored time.

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

Thanks to the previous definitions we can reformulate the steady state assumption 6.1.1 as:

$$\begin{aligned} Arrival\ rate &= Throughput \\ \lambda &= X \end{aligned} \tag{6.1}$$

In addition during the observation of the system there are periods of time in which the system is idle. For this reason we introduce another couple of definitions.

Definition 6.1.6 We define the *busy time* B the amount of time the system worked during the observation.

Definition 6.1.7 We define the *utilisation* U as the percentage of the time the system worked over the total time.

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

Definition 6.1.8 We define N as the average number of requests within the system.

Definition 6.1.9 We define the *service time* S as the average time the system took to execute a request, without considering how much time it took to begin execution.

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

Definition 6.1.10 We define the *response time* R as the average time a request takes to be fulfilled from the instant it enters the system to the moment it leaves the system.

Definition 6.1.11 We define the *waiting time* W as the average time a request waits before the moment it begins its execution

With the previous definitions we can write:

$$R = W + S \tag{6.2}$$

the previous equation is not true in all cases but for the models that will be considered in the following pages it is always true.

Definition 6.1.12 We define the *customer population* as:

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

Now that we have defined several quantities, which one do we want to optimise? We normally want to optimise both R and X, however optimising these two quantities at the same time is not possible, we normally have to face a trade off between the two. From the business point of view, what we want to do is to maximise X to guarantee we can serve the maximum amount of customers while balancing R for the satisfaction of the customer itself. However this statement is not always true and depends on the purpose and goal of the application.

6.2 Laws

6.2.1 Utilisation Law

The utilisation law establishes that the utilisation of a system is equal to the throughput times the service time.

$$U = S \cdot X \tag{6.3}$$

This equation is really important as for any model one of the most important quantities is S and normally it can't be measure directly. With the utilisation law we can overcome the limitations and estimate the value of S based on U and X. Its demonstration is quite simple, it is enough to substitute each quantity with their definition.

6.2.2 Little's Law

Little's law establishes that the average number of requests within a system is equal to the response time times the thoughput.

$$N = R \cdot X \tag{6.4}$$

The demonstration of this law is quite complex and involves integrals, however we can prove it through an intuitive yet informal demonstration.

In some sense Little's law can be seen as a law from 'outside' while the utilisation law can be seen as a a special case of Little's law when looked from the 'inside'.

6.3 Open System

In an open system we don't care what happens outside the system. It is the model used for all the online applications.

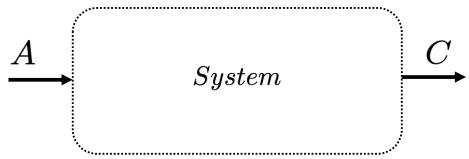


Figure 6.1: The visual representation of an open system

6.4 Closed System

Closed systems are particular systems in which after a request has been completed the request goes back into the system.

Definition 6.4.1 We define the *think time* Z as the average delay of a request before entering the system once again.

For these kind of systems the arrival rate makes no sense.

According to the value of the think time we distinguish two types of closed systems: batch systems and think time systems.

6.4.1 Batch systems

In batch systems we have that the think time $Z = 0$ and as soon as a request is completed a new one enters the system (i.e a for loop). For these kinds of systems N is always deterministic (fixed) and has the meaning of the degree of parallelism, in other words the number of threads supported by the system.

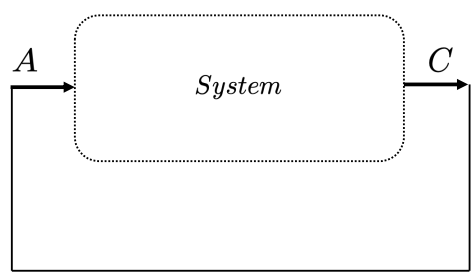


Figure 6.2: The visual representation of a batch system

6.4.2 Think time systems

In think time systems we have that the think time $Z > 0$. These systems were very common in the past when computers were slow a not connected in a network.

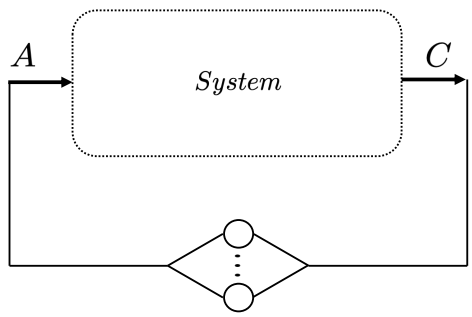


Figure 6.3: The visual representation of an closed system with think time

When dealing with closed systems one must be very careful as we have to consider the think time. We have that $N = N_A + N_B$ and Little's law can be rewritten as

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

If we isolate R we call this the RESPONSE TIME LAW:

$$R = \frac{N}{X} - Z \tag{6.5}$$

6.5 Stations

It is now time to clear what do we mean by system. The most simple type of system is a system with only one component. Each component, normally called *station*, is made up by a *queue* where requests wait to be executed and a *service center* that executes the request.

Figure 6.4: The visual representation of a component

In a real situation we have to deal not with a single component but with k components, in such cases, we say we deal with a *queue network*. It is called a network as the components are interconnected and interact in various ways.

When we deal with several components it is appropriate to define some quantities for each one of the components: $X_i, \lambda_i, R_i, U_i, S_i, W_i$ and we define Q_i the average number of requests in each component. Using Little's law we can write:

$$Q_i = R_i X_i \tag{6.6}$$

Now we ask ourselves, how can we mathematically relate global quantities and quantities defined for each component?

First of all it is important to highlight the fact that **the utilisation, service time and waiting time are defined only for each component, and not for the whole system**, it makes no sense to define the utilisation of the system as a request can visit several components before leaving. We notice that:

$$N = \sum_i Q_i \tag{6.7}$$

and using Little's law and using definition 6.6 we can write:

$$R = \sum_i \frac{N}{X} = \frac{\sum Q_i}{X} = \frac{\sum_i R_i X_i}{X} \tag{6.8}$$

In order to get the arrival rate λ from λ_i, we need to know the topology of the system in order to understand how several components are connected and how many requests may arrive to each component. When we have small systems a simple drawing may be enough but in large system an *incidence matrix* is used. This matrix expresses the connections of the system through its elements P_{ij} that represent the probability of a request to go from station i to station j.

This matrix has an interesting property, the sum of all rows is always 1 as a request should always go somewhere. It is therefore crucial that the incidence matrix contains not only all components but also a *virtual station* that represents the 'outside', in a way that requests can arrive and go out from the boundaries of the system.

This virtual station is conventionally labeled as the $0-th$ component making P_{0j} the probability of a request entering station j from the outside and P_{i0} the probability of a request leaving station i to the outside.

6.5.1 Forced Flow Law

Definition 6.5.1 We define the *number of visits* V_i as:

$$V_i = \frac{X_i}{X}$$

thanks to the previous definition we can rewrite it as:

$$V_i = \frac{X_i}{X} = \frac{C_i}{C} \overset{SSA}{=} \frac{A_i}{A} \Rightarrow \lambda = \frac{\lambda_i}{V_i}$$

and

$$V_i = \sum_j V_j P_{ji} \tag{6.9}$$

It is important for the solution of this linear set of equations we set $V_0 = 1$, in other words we must set the number of visits for the virtual station equal to 1 as the throughput for the virtual component is equal to the throughput of the overall system.

We can rewrite 6.8 using the visit definition as:

$$R = \sum_i R_i Vi \tag{6.10}$$

where we call the term $R_i Vi$ the *residence time*. rewriting V_i we can obtain the FORCED FLOW LAW:

$$X_i = X\, V_i \tag{6.11}$$

Definition 6.5.2 We define the service demand of a component i the total service required by a request to component i:

$$D_i = V_i\, S_i$$

If a request makes an average of V_i visits to a disk and requires an average of S_i milliseconds of service per visit, then that request requires a total of $V_i\, S_i$ milliseconds of disk service, that is by definition its service demand at that disk.

Little's law is usually combined with the forced flow law and the utilisation law.

■ **Example 6.1** Let's suppose that we want to determine the average system response time R for a system with the following known characteristics:

- 50 terminals ($N = 50$)
- 20 seconds average think time ($Z = 20$ secs)
- 8 visits to a specific disk per interaction ($V_{disk} = 8$ visits/request)
- 20% utilisation of that disk ($U_{disk} = 0.2$)
- 75 millisecond average service requirement per visit to that disk ($S_{disk} = 0.075$ secs/visit)

We would like to apply the response time law but we don't have the disk throughput. We can, however, apply the forced flow law to obtain system-level throughput and then use the utilisation law:

$$U_{disk} = S_{disk} X_{disk} \quad , \quad X_{disk} = V_{disk} X \quad \Rightarrow \quad U_{disk} = S_{disk} V_{disk} X = D_{disk} X$$

as a consequence the relationship between the demand time and the throughput is given by:

$$X = \frac{U_i}{S_i V_i} = \frac{U_i}{D_i} \tag{6.12}$$

in the example we obtain $X = 1/3 = 0.3\bar{3}$ requests/sec. Now that we have obtained X, we can calculate R with the response law:

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

giving as a final result $R = 130$ seconds.

6.6 Load Balancing

Suppose we have two disks, a fast one and a slow one, if the load is unbalanced what happens is that we don't make an efficient use of resources as we could achieve a higher total demand (and therefore throughput) by balancing the load. Balancing the load *means to have the same demand on functionally equivalent resources*.

Suppose we have a slow disk with V_{SD} visits and S_{SD} service time and a fast disk with V_{FD} visits and S_{FD} service time. How can we balance the load?

$$V_{SD} + V_{FD} = k$$

$$V_{SD}\frac{S_{SD}}{S_{SD}} + V_{FD}\frac{S_{FD}}{S_{FD}} = k$$

$$\frac{D_{SD}}{S_{SD}} + \frac{D_{FD}}{S_{FD}} = k$$

as we want to balance the load we impose $D_{SD} = D_{FD} = D_{Bal}$

$$D_{Bal}\left(\frac{1}{S_{SD}} + \frac{1}{S_{FD}}\right) = k$$

finally we obtain:

$$D_{Bal} = \frac{k}{\frac{1}{S_{SD}} + \frac{1}{S_{FD}}} \tag{6.13}$$

7. Performance Models

In the previous chapter we learned how to calculate several quantities such as throughput and response time. However all the data we had was empirical and derived from observations. How can we estimate the throughput and the response time for a model we can't observe? We will learn how to give an estimation of such models.

For open models the computation of the throughput is very easy, while for closed models it is computationally expensive. Despite the fact that solving complex models is very complicated, with very little computation it is possible to determine upper and lower bounds on the real system throughput X and real response time R as functions of the system workload intensity (number of customers N). This approximate solution technique is called *bounding analysis*. These bounding techniques can be used to estimate how much can we improve the performance if we perform an update and to determine how to properly size a system (system sizing studies).

We will study a family of bounds called *asymptotic bounds*, that is, bounds that become exact when the number of requests goes to infinite and *balanced system bounds* that hold for a smaller set of systems but have the advantage of being tighter, and thus they provide more precise than asymptotic bounds.

7.1 Closed System Asymptotic Bounds

7.1.1 Optimistic Bounds

We call the throughput upper bound and and the response time lower bound as optimistic bounds since they indicate the performance on the best possible scenario.
Let's suppose we that we arrive in a service centre and there is no queue and we move to the next station and still there is no queue, and we are so lucky that we never have to be on a queue. Which is the response time in this situation?

When a request enters a system it can visit a station multiple times, to be precise, a request visits a station i V_i times and at each visit it requires a service time of S_i, requiring to component i a total service time equal to $D_i = S_i V_i$. We can write that the total response time is:

$$R = W + D$$

where the waiting time W is zero as we do not wait on any queue and where $D = \sum_i D_i$ is the sum of the total service time required to all components. In the best case scenario we can write $R = D$ which expresses the idea that the minimum amount of time for a request to be completed is exactly D, thus we can affirm that for any system:

$$R \geq \sum_i D_i \qquad \Rightarrow \qquad R \geq D \tag{7.1}$$

We have a lower bound on the response time, in order to transform it to a bound on the response time we can use the response law (Little's Law for closed systems):

$$R = \frac{N}{X} - Z \qquad \Rightarrow \qquad \frac{N}{X} - Z \geq D$$

therefore we can write:

$$X \leq \frac{N}{D+Z} \tag{7.2}$$

7.1.2 Pessimistic Bounds

We call the throughput lower bound and and the response time upper bound as pessimistic bounds since they indicate the performance on the worst case scenario.
On the worst case scenario we have to wait that $N-1$ requests are be fulfilled at every resource plus the time to fulfil our request for all the visits to all resources, in other words, in this unlucky situation we have that the response time is:

$$R = \sum_i N D_i = N \sum_i D_i = N D$$

As a consequence we can write an upper bound for a closed system:

$$R \leq N D \tag{7.3}$$

To write the lower bound for the throughput we use once again the response time law (Little's Law for closed systems) and we write the bound as:

$$X \geq \frac{N}{N \cdot D + Z} \tag{7.4}$$

Can we define other bounds?
For now we've only considered the response law. In order to write other types of bounds we must consider a model with one component. Since the utilisation of a component can't be greater than 1 we can write using the utilisation law $U = S \cdot X$ the expression :

$$X_i \leq \frac{1}{S_i} \tag{7.5}$$

But we're interested on the system throughput and not the throughput of a single component.We can transform the bound by using the forced flow law ($X_i = X \cdot V_i$), so the bound can be rewritten as:

$$X \leq \frac{1}{S_i \cdot V_i} \qquad \Rightarrow \qquad X \leq \frac{1}{D_i} \tag{7.6}$$

this equation has to be true for any value of D_i, therefore we have that:

$$X \leq \min_i \left\{ \frac{1}{D_i} \right\} \qquad \text{equivalently} \qquad X \leq \frac{1}{D_{max}} \tag{7.7}$$

The station with the maximum demand is called the *bottleneck station* and is the component that limits the throughput of the system, so in order to increase the throughput of the system we have to improve the slowest of the components.

If we add the bound on the plot we find that there is an intersection between the throughput upper bound and the bound just found. The value of N in which the two bounds intersect is:

$$N^* = \frac{D+Z}{D_{max}} \tag{7.8}$$

When there is only one station, $D_{max} = D$ and the solution becomes exact.
The real throughput of the system is always closer to the upper bound than to the lower bound and when N tends to infinite, the throughput becomes the upper bound.

We can say that when $N > N^*$ the throughput is flattened and when $N < N^*$ the throughput grows with N.

If we recall that N is the degree of parallelism for batch applications, we can conclude that it doesn't make sense to run more than N^* parallel processes.

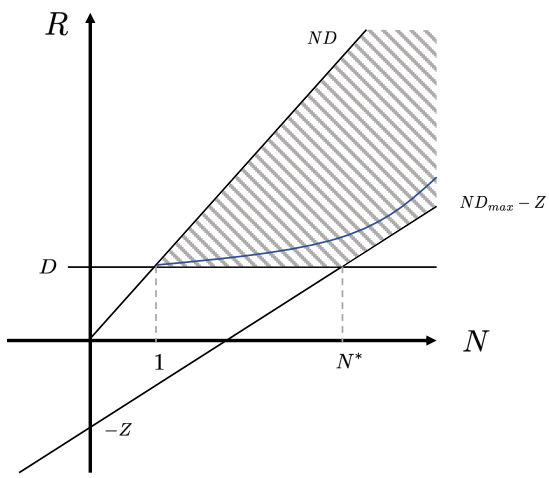
In order to write the same bound for the response time we use the response time law and we can rewrite the bound as:

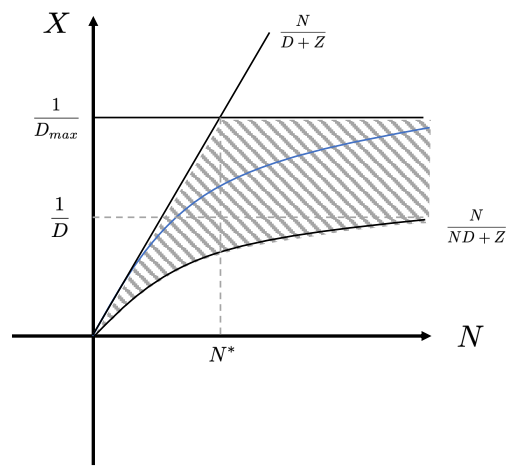
$$R \geq N \cdot D_{max} - Z \tag{7.9}$$

If we add the bound on the plot we find that there is an intersection between the response time lower bound and the above bound. The value of N in which the two bounds intersect is N^*.

The real response time of the system is always closer to the lower bound than to the upper bound and when N tends to infinite, the throughput becomes the lower bound.

We can summarise the all the bounds as:





$$\max\{\, ND_{max} - Z,\, D \,\} \quad \leq \quad R \quad \leq \quad ND \tag{7.10}$$

$$\frac{N}{ND+Z} \quad \leq \quad X \quad \leq \quad \min_i \left\{ \frac{1}{D_{max}}, \frac{N}{D+Z} \right\} \tag{7.11}$$

From the plots we can observe that when $N \simeq N^*$ the system will have the best tradeoff between throughput and response time.

When we have a real system we're not able to measure the demand without first finding the number of visits with the routing probabilities.

7.2 Balanced System Bounds

With a slight increase in the computation required to obtain the bounds we can obtain bounds that are tighter and give us a more precise estimation of the true value. These bounds are called balanced system bounds because they are based upon "balanced systems" in which the service demand at every centre is the same:

$$D_i = D_2 = D_3 = \ldots = D_K \tag{7.12}$$

Even if the "balanced systems" assumption is not always true, the bounds are valid in any situation of a closed system. It is also important to highlight that these kind of bounds aren't usually used as a standalone tool and **must be used together with asymptotic bounds**. Balanced bounds are computed differently depending on the value of Z, now we will now focus only on the computation of balanced bounds for batch systems (closed systems with $Z = 0$).

Before we compute the bounds let's recall that when $V_i = 1$ the response time is defined as:

$$R_i = D_i + W_i$$

we can write the term W_i differently and write:

$$R_i = D_i + D_i A_i \tag{7.13}$$

For a balanced batch system of systems the average queue length is:

$$Q_i = N/K \tag{7.14}$$

R It is important to remark that the number of average number of requests Q_i and the number of requests you see when you arrive A_i can be totally different.

as all the service stations centre are equals. To compute the average number of requests in the system

$$A_i = \frac{N-1}{K} \tag{7.15}$$

The response time in the case in which all stations are equal can be rewritten as:

$$R_i = D_i + D_i \frac{N-1}{K} \tag{7.16}$$

$$R = \sum_i^K R_i V_i = K R_i \tag{7.17}$$

mixing both equations above we can write:

$$R = K D_i + D_i (N-1) \tag{7.18}$$

where the term $K D_i$ is equal to D as $D = \sum D_i$ and given the balanced system assumption, the sum of the demand of all components is K times the demand of a single component.

7.2.1 Response Time Balanced Bounds

Remembering that the balanced system assumption is not always true as components usually have a different demand, we can write the following bound:

$$R \leq D + D_{max}(N-1) \tag{7.19}$$

as in the worst possible case, even if $D_i = D_{max}$ the response time is surely lower.

In addition, supposing the best possible case in which the balanced system assumption holds, we can write the following bound :

$$R \geq D + D_{avg}(N-1) \tag{7.20}$$

where we define the D_{avg} on the basis of the balanced system assumption as:

$$D_{avg} = \frac{D}{K} \tag{7.21}$$

7.2.2 Throughput Balanced Bounds

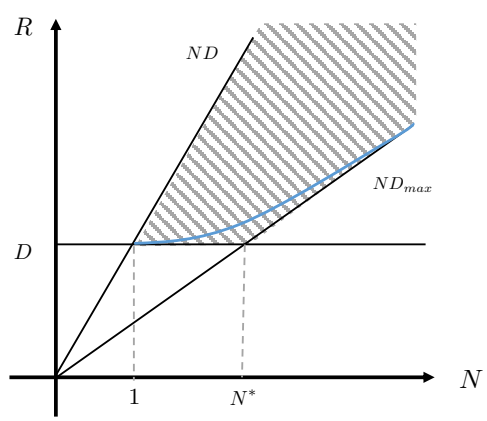
We can transform the balanced response time bounds on balanced throughput bounds by using Little's Law ($N = RX$) as the think-time is zero by hypothesis.

$$X \geq \frac{N}{D + D_{max}(N-1)} \tag{7.22}$$

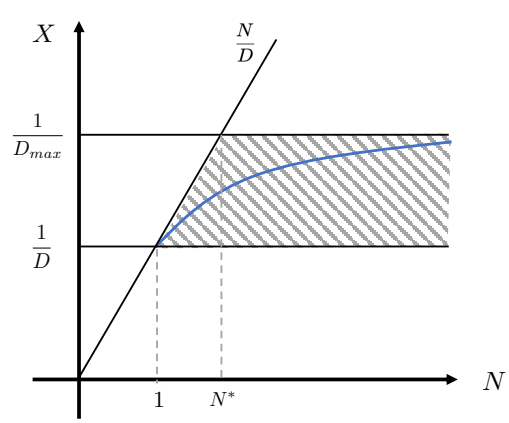
and

$$X \leq \frac{N}{D + D_{avg}(N-1)} \tag{7.23}$$

If we plot the asymptotic bounds for batch systems we obtain the following graphs:



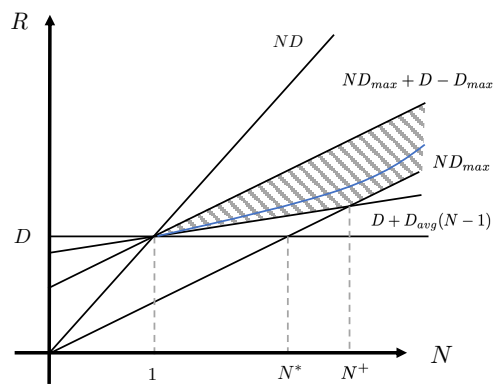
$$\max\{\, ND_{max},\ D \,\} \leq R \leq ND$$

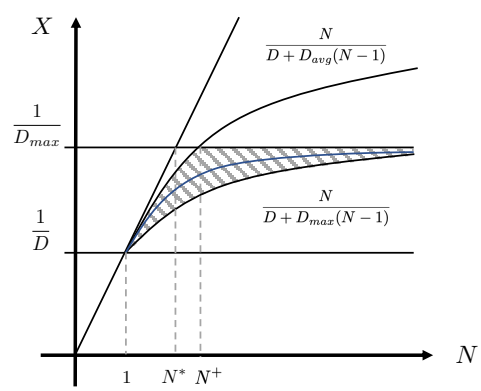


$$\frac{N}{ND} \leq X \leq \min_i \left\{ \frac{1}{D_{max}}, \frac{N}{D} \right\}$$

If we carefully analyse the upper balanced bound for the response time, we notice that it has the same slope of the lower asymptotic bound, the only difference is that the intercept is not zero anymore but $D - D_{max}$.

Also noticing that $D_{avg} < D_{max}$ we can see that the lower balanced bound has a slope which is lower than ND_{max}, keeping in mind the considerations done we can plot the graphs as:





$$\max\{\, ND_{max} \,,\, D + D_{avg}(N-1) \,\} \quad \leq \quad R \quad \leq \quad D + D_{max}(N-1) \tag{7.24}$$

$$\frac{N}{D + D_{max}(N-1)} \quad \leq \quad X \quad \leq \quad \min_i \left\{ \frac{1}{D_{max}} \,,\, \frac{N}{D + D_{avg}(N-1)} \right\} \tag{7.25}$$

We can identify a new point that is the intersection of the lower asymptotic and balanced bounds for the response time and the intersection of the upper asymptotic and balanced bounds for the throughput:

$$N^+ = \frac{D - D_{avg}}{D_{max} - D_{avg}} \tag{7.26}$$

8. Open Models

Let's recall that in an open model, there is no loop, requests just arrive from the outside and the load of the system is given by the arrival rate λ.
If we evidence the dependency of the response time on λ we can rewrite the decomposition of the response time as

$$R_i(\lambda) = S_i + W_i(\lambda) \tag{8.1}$$

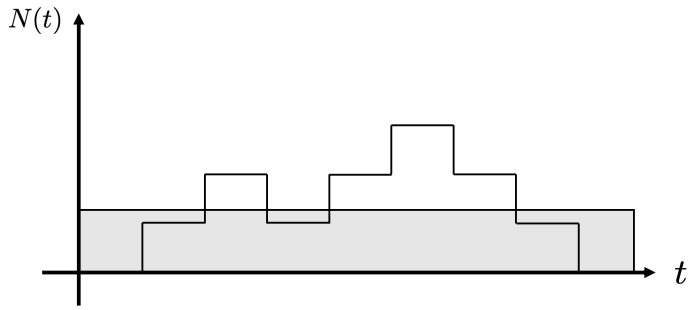
If we know how many requests are "in front" of us (A_i) we can define the waiting time as:

$$W_i = S_i \, A_i(\lambda) \tag{8.2}$$

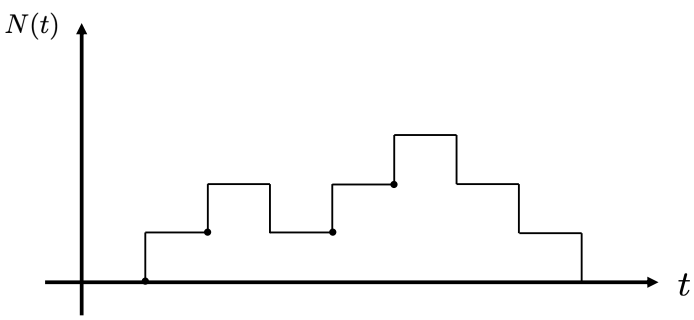
The most difficult part is finding $A_i(\lambda)$. Our goal is to state that the number of requests in front of us $A_i(\lambda)$ is equal to the average number of requests as seen from the outside $Q_i(\lambda)$.

$$A_i(\lambda) = Q_i(\lambda) \tag{8.3}$$

If we can demonstrate that this equation holds, then solving an open model becomes an easy task. First let's clear what we mean by $Q_i(\lambda)$ and $A_i(\lambda)$ by showing their meaning intuitively :



Q_i is the average number of requests in a component, and can be thought of as the height of a rectangle that contains the same area of the discrete graph that plots the number of requests over time



A_i is the average number of requests in front of us, which can be thought of as the average of the points in the graph where there is a step that goes up.

Now that we have cleared the meaning of these two quantities, to demonstrate the previous equality we now have to do some assumptions:

Assumption 8.0.1 The service time has an exponential distribution. This hypothesis introduces a nice property of the exponential distribution: the *memoryless property* that states that what happened in the past does not affect the future.

EXAMPLE In other words the memoryless property implies that if a request has not been served in the past that does not mean the request will be served in less time.

Assumption 8.0.2 The *interrarrival time* $1/\lambda$, that is defined as the time between each arrival into the system and the next, has an *exponential distribution*.

Assumption 8.0.3 FIRST COME FIRST SERVED. The first request that arrives into the system is the first one that is served, in other words, requests will be dealt in the order in which they arrive.

Assumption 8.0.4 PROCESSOR SHARING. If N multiple requests are in the system, they are all processed by the system but with a service time that is N time longer.

If we put together all the previous assumptions the equality 8.3 has been shown to be valid (demonstration is too hard and long), and under such conditions we can write the response time of a single component as:

$$R_i = S_i + S_i Q_i$$

We can apply Little's Law $Q_i = X_i R_i$ and write the equation as:

$$R_i = S_i + S_i X_i R_i \tag{8.4}$$

Which can be rewritten as:

$$R_i = \frac{S_i}{1 - S_i X_i} = \frac{S_i}{1 - U_i} \tag{8.5}$$

Or if we recall the steady state assumption $\lambda = X$

$$R_i = \frac{S_i}{1 - S_i \lambda_i} \tag{8.6}$$

If we look carefully the equation we realise that λ_i is bounded and it cannot be greater than $1/S_i$:

$$\lambda_i < \frac{1}{S_i}$$

We can rewrite the equation once again recalling the Forced Flow Law ($XV_i = X_i$)

$$R_i = \frac{S_i}{1 - S_i X V_i} = \frac{S_i}{1 - D_i X} \tag{8.7}$$

Now that we have expressed the response time of a single component we can we can rewrite the formula of the system response time $R = \sum R_i V_i$ as:

$$R = \sum_i \frac{S_i V_i}{1 - D_i \lambda} = \sum_i \frac{D_i}{1 - D_i \lambda} \tag{8.8}$$

8.1 Bounds

In the last chapter we saw asymptotic and balanced bounds for the response time of closed systems, for open models we can write several bounds combining the ideas seen in the last chapter and the equation found for the response time in the last section.

$$\frac{D}{1 - \lambda D_{avg}} < R < \frac{D}{1 - \lambda D_{max}} \tag{8.9}$$