

Computing Infrastructure

Elia Ravella

July 31, 2021

Contents

Part I

Introduction

1 Computing Infrastructure

What is a CI? A CI is a *technological structure* that provides hardware and software for computation to other systems. This means a lot: is a joint (and heterogeneous) system that comprises both HW and SW to deliver several services. With CI is not intended the *application delivered* but the combination of elements that makes the application available and running.

So everything in between a RaspPI that does torrent seeding from a local network to the whole datacenter that runs an AWS service can be considered a computing infrastructure.

Part II

Hardware Infrastructures

2 Data Centers

2.1 Pros and cons

Data centers are big centralized CIs that comprise a lot of servers (to provide computations) a communication infrastructure, and a storage service.

PROS of a DC:

- Lower IT cost: renting a virtual machine is cheaper than buying / building and maintaining a whole system (for some time horizons, of course);
- High performance: virtualized resources make scaling easier, and provides optimized and finely tuned services that an in-house solution cannot provide *so* easily and fast;
- No effort software updates;
- Unlimited (!) storage capacity;
- Increased data reliability;
- Universality of accesses;
- Device indipendence.

The CONS of a DC can be found simply inverting the POV for the PROS aspects: outsourcing resources gives "someone else" the control over some crucial aspects of a CI. This is a good thing when costs (also time costs) must be reduced, but can be a bad thing when a fine grained control over a full system is needed. Also, *latency* pops in, due to the needed connection to the DC.

2.2 DCs ans WSCs

The data center approach has been emerging in the last years. The idea behind it is that we should not "overpack" nodes of a network with all the computing capabilities required, but instead giving them a connection to such CIs. Data centers are the perfect example for this paradigm: the user interact with a client (that can be *any kind of device*) that also interact with a remote structure to provide computations. To a careful watch, SaaS and their spread are a direct consequence of this paradigm shift. Another use of DCs is using their computing capabilities not in a fragmented way to provide miryads of services, but to perform an *extremely costly computation*, like a training of a neural network.

From a DC approach we are moving to Warehouse Scale Computers nowadays. This latter approach consists of NOT "mixing up the pot" in a DC (so having a lot of heterogeneous technologies in order to achieve different tasks) but instead to "homogeneify" the cloud structure in order to

do better optimization of it. Not only in the HW, but also in the SW. To centralize the DC capabilities under a single organization (as often happens) means that clients no more run their application on someone else's hardware, but instead choose from a set of precooked solutions by such vendor/organization. Is this *bad*? WSCs are just a "SaaS - oriented" Data Center Architectures, so *there's no real transition between DCs and WCSs* the only transition is in the number of *virtualized layers the client must go through to access an application*.

We can sum up the difference between DCs and WSCs in this way: where DCs are intended as a powerful collection of different servers, WSCs tries to offer a homogeneous interface that can be used also as a single server to such a collection of hardware. Still, if we consider a larger definition for DCs, WSCs are a type of DCs.

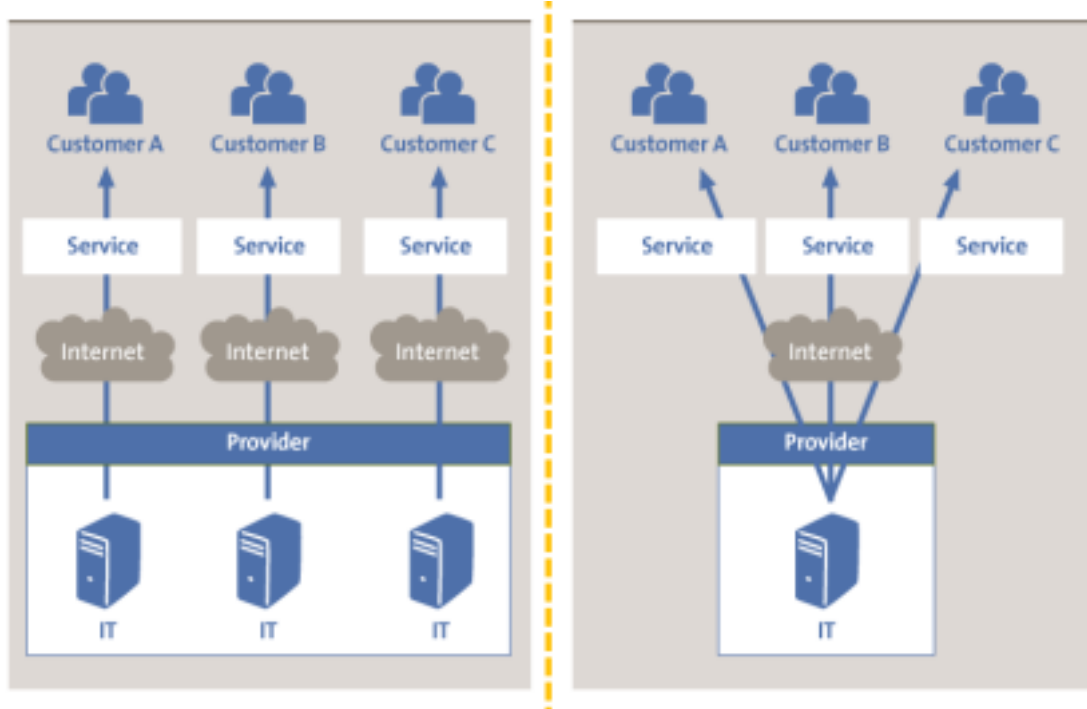


Figure 1: Left, the classic DC approach. Right, the WSC scheme.

2.3 Architectural Overview of DCs and WCSs

A standard data center building is organized in separated modules, every one dedicated to a specific task. The four main modules types are

- Servers: computations
- Storage
- Networking: intended as the whole communication infrastructure
- Building-integrated systems:
 - Cooling system: often as powerful as the server themselves, fundamental for getting rid of excess heat
 - Power supply: integrated in the building and provided with systems to avoid power shortage
 - Failure recovery: physically realized as a building module in order to be as most fault - tolerant

Servers are organized in racks, blades or tower, and are the classical computational unit usually found in server farms. They could also not have memory attached.

Storage is the crucial long term memory for a CI. Usually built with flash SSDs and ferromagnetic

mechanical disks. The memory units must provide high speed I/O capabilities, and also advanced networking power, also at software level (NAS, SAN, DAS).

The networking infrastructure is the backbone of the communication in (and from/to) a data center. Structured hierarchical approach to networking structures and organization are used to ensure security and performance

The building itself is part of the CI.

Next sections will delve into the details of the single modules.

2.4 Servers

Servers are the computational compartment of a WSC. They're organized in racks (shelves) that host the computational units (pizza box computer). Servers are just computers. That's all. They got a MOBO, local primary memory, a CPU, and I/O capabilities. They can be organized as

- Towers: cheap, cooling is easy, upgrade is easy. They're also big, and they provide low density of computational power.
- Racks: literal racks that host the pizza box shaped computational units, and accomodate all the wiring and additional connection or cooling systems. They can host also heterogeneous components, not only standard computational modules. Rack's dimension and geometry is a standard. Rack organization provides better failure handling and simplified cable management, but they're more power demanding (wrt towers) and maintenance is a mess (multiple devices must be maintained at the same time).
- Blades: also called hybrid racks, the idea is similar to the rack system, but a server is inserted *vertically* instead of *horizontally* in a dedicated place. This (together with the average smaller size of a blade server) enhances computation per volume ratio while keeping all the pros of a rack system. Blade servers have disadvantages: heat management is complex, and enclosures and envelops are more expensive wrt racks.

To sum up:

Format	Pros	Cons
Tower	Cost effective, easy to cool down (low component density), easy to upgrade	bulky, not so powerful, messy cable management
Rack	Easy to replace, straightforward cabling, cost effective	high component density leads to high power consumption, difficult to maintain singularly
Blade	Super compact, support for centralized management, easy cabling	Expensive first configuration, nightmare to cool down

2.4.1 Hardware Accelerator

WSCs architectures and the rise of heavy computation loads (like for intense machine learning and AI training) have caused a spike in the complexity required every computation, even far beyond the Moore's law. To satisfy this need, dedicated hardware (in the form of *hardware accelerators*) are deployed in WSCs, to enhance the performance in determined fields.

GPU Graphical Processing Units are featured in accelerators to enhance *parallel fast computation*. Due to the architecture of them, they're also perfect to do matrix computation.

TPU Tensor Processing Units. These are more specialized GPUs in the machine learning direction. They're (simplifying) custom circuits to support tensor computations (matrix calculus) that are *very fast* at doing so. Google released TPuv3 at the time I'm writing

FPGA Good ol' Field Programmable Gate Array circuits are still used to enhance performance in custom computational oriented architectures. These circuits are somewhat "fully customizable" hardware computers that if programmed in a very specific way can enhance *every kind* of possible computation.

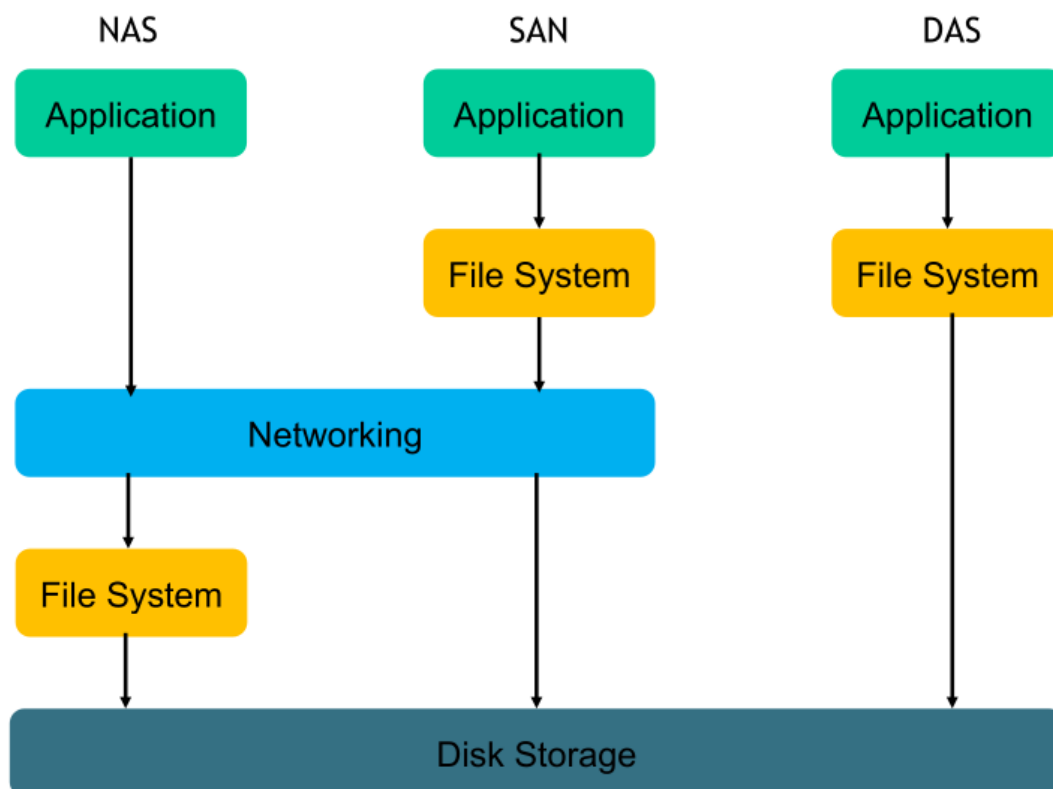
2.5 Storage

Storage. Not much more. Technologies involved: HDDs and SSDs and Flash memory. Performance (and relevant) parameters:

- read / write speed and latency (or seek time)
- memory density
- security: but this is intended as an hardware security, not data security (for now)
- cost

2.5.1 Storage Systems - DAS, NAS, SAN

- Direct Attached Storage is every device that offers storage capabilities and is directly attached to the unit processing such data in the storage. HDDs are DAS for personal computers, for example. They're directly accessible from the OS of the machine accessing it. They're difficult to scale. They are difficult to manage when particular storage solutions are needed. Moreover, the OS's proprietary file sharing model / protocol / software must be used in order to share files with other machines.
- Network Attached Storage is a storage system that lies *on the other end* of a network. A NAS only offers data services accessible from dedicated protocols (as FTP, SAMBA) and it's *an actual computer* that offers services. A NAS is a *full fledged computer* that's *only meant for storage services*, accessible through network protocols. It's located somewhere in the network. It's easily scalable (both augmenting the storage of a single NAS or multiplying the NAS number work).
- Storage Area Network is like NAS, but the remote storage unit are *passive* and are directly accessible by the OS (by means still of a network connection). They're *remotely accessed simple storage devices*. Disks are visible *like local storage, like DAS* to the server machine, but they're network attached. They have specialize hardware that mask the network in between, and the SAN can be accessed *directly through the filesystem*.



2.6 Networking

Design a network architecture for a WSC ain't easy job. Moreover, the actual networking technologies haven't a straightforward horizontal scaling solution. This means that the *the number of connections needed increases exponentially with the number of clients*, except for some clever solutions, if we are looking for a capillar connectivity. And, in a WSC, we are. Our aim is to provide additional bisection bandwidth, that's the bandwidth available between two partitions of the network.

A WSC networking system can be divided in 2 separate blocks:

- an internal LAN that connects all the components (the servers) among themselves
- an interface to the internet (DMZ..?)

Obviously, the networking infrastructure in a WSC is crucial because it also connects the functional units (as the server) to the storage infrastructure and so it's *fundamental* that this connections is solid. The need of inter-server bandwidth incrases exponentially with the number of server, obviously.

A WSC internal network is usually divided in 3 levels:

1. Access: this network layer is the one that connects the servers to the network. It's embodied by the network infrastructures embedded in the server racks.
2. Aggregation: this level groups the servers together (VLAN-like) and it's usually implemented as Top Of the Rack switches, that group together all the server in a aisle (for example).
3. Core: furthest level from the single server, aggregates entities of the "aggregation" level.

Many network topologies are out there:

1. Fat tree (classic hierarchical topology, clearly visible access aggregation and core levels)
2. D-Cell, recursive topology. Cells are organized in levels and the network "builds itself" through a discover mechanism
3. Hypercube topologies

2.7 Building and Non-IT Equipment

The way in which the hardware is arranged *inside* a DC is as crucial as the building composing the DC itself, as well as the non-IT instruments (like the cooling system, or the energy system) that completes the DC structure.

2.7.1 Energy System

Powering a DC is difficult on many layers; first of all, computing infrastructures of this scale need a lot of energy, but this energy must be provided steadily and must be always available. Moreover, the power must not be subject to cuts, improvised shortages or outing. So not only the energy system of a DC must handle a lot of power, but should also handle it *with care*. Usually, DCs have a UPS that does all this work.

The UPS The Uninterruptible Power System is the hearth of the energy subsystem of a data-center. It is composed of 3 main parts:

1. A socket to the external provided power. This is usually high voltage power that must be trimmed down (order of 50 kV) in order to be utilized.
2. An internal generators system that is powered up when the external power supply fails. These systems takes usually 10 to 20 seconds to power up.
3. A battery array to bridge the gap between an external power outage and the wind-up time of the backup generators.

The UPS must also "filter" the external power provided, cutting spikes and removing dangerous harmonics in current. This is usually done by a AC to DC to AC converter. Then, electric power is sent to power distribution units (that resembles the one in the houses) that manages the power flow in each row or rack.

2.7.2 Cooling System

The cooling system removes the heat generated by the equipment. It must imply some sort of loop (thermodynamically speaking) that warms up a medium and then transfers the heat somewhere else.

There are several strategies to cool down a data center:

- Open loop: the fresh air is "sucked in" from the outside and then passively heated by the servers themselves. The hot air exhaust is let flow out from the top. In fact, this is a open-the-windows approach. Obviously this approach is very easy to set up, and requires little to no additional hardware (the additional hardware can be needed to cool down the air intake, or speed up the airflow, controlling humidity...) and can perform very well for its cost. Being so simple, it has its drawbacks: it depends from the outside temperature, the air flow is "uncontrolled"...
- Closed loop: this approach is more similar to the personal computer's one. Heat is removed from the hardware and then the medium used to remove it is chilled in another place. Why closed? Because it does not depends on external air / chiller medium to refrigerate a room. Instead, the medium is chilled in a heat exchanger (usually located in a dedicated CRAC room) and then reinsterted in the loop. Multiple loop can be exploited in order to enhance the control over the heat transfer.
- In rack cooling: manufacturers of server racks can add a device that acts as a heat exchanger (in particular, generally it's a air to water exchanger) that sucks out the out directly from the servers.
- In row cooling: as in rack cooling, just done at row level.
- Circuit cooling: as in a personal PC, we can directly cool down circuitry (usually with liquid cooling systems) and this approach offers the most elevated performances. However, liquid circuitry cooling system are hard to manage, other than impractical.

2.7.3 DCs Tiers

Availability of a DC is expressed on a scale, from one to four, that depends on a set of fixed parameters. Here's the table:

Tier Level	Requirements
1	<ul style="list-style-type: none"> • Single non-redundant distribution path serving the IT equipment • Non-redundant capacity components • Basic site infrastructure with expected availability of 99.671%
2	<ul style="list-style-type: none"> • Meets or exceeds all Tier 1 requirements • Redundant site infrastructure capacity components with expected availability of 99.741%
3	<ul style="list-style-type: none"> • Meets or exceeds all Tier 2 requirements • Multiple independent distribution paths serving the IT equipment • All IT equipment must be dual-powered and fully compatible with the topology of a site's architecture • Concurrently maintainable site infrastructure with expected availability of 99.982%
4	<ul style="list-style-type: none"> • Meets or exceeds all Tier 3 requirements • All cooling equipment is independently dual-powered, including chillers and heating, ventilating and air-conditioning (HVAC) systems • Fault-tolerant site infrastructure with electrical power storage and distribution facilities with expected availability of 99.995%

Figure 2: DC tiers

Part III

Software Infrastructure

3 Virtualization

Virtualization is the procedure of "making a resource available through software artifacts". This resource can range from a particular kind of processor to a whole application set or service. Virtualization is the main technology enabling cloud computing, because of the flexibility mainly, the isolation and the security offered. Server VMs are implemented without using a host OSs as the personal computer one, they rely instead on a virtual machine monitor (like HyperVisor) that manages the hardware to run multiple VMs at the same time.

3.1 Virtual Machines

A machine is defined as "execution environment capable of running a program". This is a very general definition, that ranges from washing machines to computing infrastructures for neural network training. A virtual machine differs from a physical machine by the way in which hardware is managed. A physical machines handles hardware by the OS, while a VM has to interface with a hardware supervisor in order to access it. "Formally", a VM is a *logical abstraction able to provide a virtualized execution environment*.

A VM must provide identical software behaviour (execution on a VM should be transparent). The VM, being composed of a mix of hardware and virtualizing/virtualized software, usually has worse performance wrt his physical counterimplementation. The VM is in charge of the translation of the virtualized software instructions into effective machine instructions.

3.2 Levels of Virtualization

Given the intrinsically layered structure of an operating system, it's easy to discriminate between virtualization approaches basing on the layer they address.

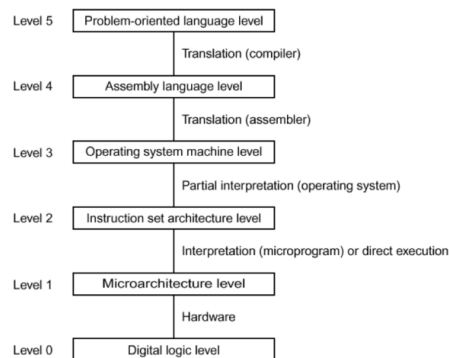


Figure 3: The layered operating system structure

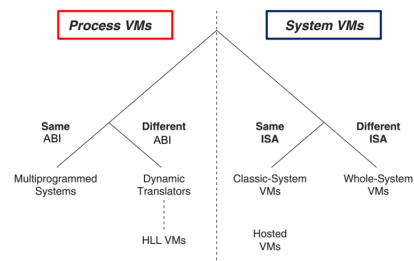


Figure 4: VMs and what they virtualize

3.2.1 Multiprogrammed Systems

Far left of the schema, and it's just a classic parallel OS. It's arguable it is a real VM, because all the traditional hardware-sharing computations nowadays exploit this model.

3.2.2 High Level Language Virtual Machines

Java approach; isolated execution environment for each application. The VM just translates from the proprietary binaries to the actual machine's executable code. "Code once run everywhere".

3.2.3 System VMs

Virtual machine monitor on the bare metal, carries out the sharing. Virtual machine's on top of the VMM, and when a hardware-call is done this is passed through the VMM. The VMM uses the same ISA (the one of the machine). VMM can also be a program installed in another OS: in that case we're talking about *whole system VMs*¹.

3.3 Virtualization Implementation

As already said, in order to virtualize something we have to add some middleware in the software stack that carries out the virtualizing mechanism. We can put this layer *between the hardware and the OS*, or *between the operating system and the processes* as in JVM or even *between the OS and other OSs*, as in virtualbox.

As a remainder, a virtualization software should provide

- Partitioning between softwares
- Isolation, so fault tolerance and security
- Encapsulation: the state of a VM can be easily stored and moved
- HW indpendence. As incapsulation, but for the whole execution

3.4 VMM

We use three terms to define the same thing: Virtual Machine Manager, Virtual Machine Monitor and HyperVisor. They're the same thing, so *a piece of software dedicated to virtualize stuff*.

Hypervisors are the software that acts directly attached to the machine hardware and are capable of hosting an OS.

- Type 1 hypervisors provide an abstraction and a full environment to a host relying directly on the hardware.
- Type 2 hypervisors (virtualbox) provide the virtualized hardware exploiting *a host operating system*.

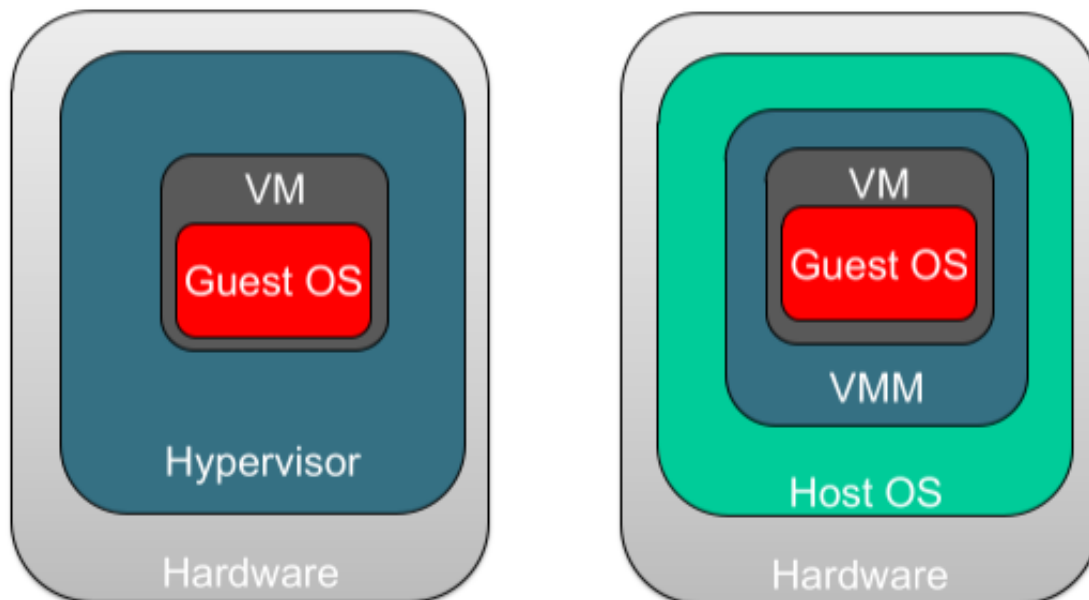


Figure 5: Type 1 and type 2 VMMs

¹Virtualbox approach

Paravirtualization A paravirtualized system *can in certain conditions bypass the hypervisor and directly access the hardware*. This is done to improve performance: certain operations (depending from the OS, the hardware and the hypervisor itself) are infinitely slower if run in the virtualized domain. A paravirtualized solution would execute these operations directly on the hardware, eventually "trapping" some execution paths. Paravirtualized systems offers increased performance and slimmer VMM, but the guest OS must be explicitly ported to the para-API.

3.5 Containers

A container is a pre configured package that comprises all the software pieces needed to run a particular type of software (like the database, the support for certain languages and services). All the software comes *configured*. This approach does not fall entirely under the "virtualization" hat, because they do not depend on a OS (the container does not provide one). Instead, they rely on a container engine (as the Docker engine).

4 Cloud Computing and Edge/Fog Computing

Virtualization paved the way to intensive cloud computing. Being able to virtualize a resource means to *have it available and accessible all the time*². It also decouples the hardware and the software.

One of the most used configuration aims to consolidate the usage of a single server, providing a dedicated environment *for each app* and they can all run in the same machine. This is a huge improvement wrt "an app for each server". On top of the hypervisor software we can also build a *distributed VM Monitor* that handles multiple VMs *across a cluster of machines*: this improves load balancing, performance and *resource usage*.

4.1 Cloud Computing and Services

Cloud computing is a model for enabling on demand network access to a shared pool of computing resources. The cloud approach pushed the IaaS paradigm to his limits: it started as "outsourcing to someone in the internet some computations" and it's "giving someone in the internet full control over our technologic stack".

4.1.1 Software as a Service

Software as a Service is the mechanism that allows users to access an entire software (an entire application) without installing it, just exploiting cloud hardware and systems. GMail, GDocs are examples of SaaS

4.1.2 Platform as a Service

PaaS is aimed towards developers. Cloud providers make available *developing frameworks and platforms* to ease developing applications (like Colab, the ML engine by Google) and the cloud machine "just" have to provide libraries, hardware and APIs that are necessary in the developing process.

4.1.3 Infrastructure, Data and Communication as a Service

IaaS, DaaS and CaaS are three approaches to "simplify your design through services" possibilities. A company can decide to use a DaaS system in order to organize data (this is a full *outsource decision*) to not have the burden of manage a fuckton³ of data. IaaS provides the *computational resources*, maybe to make computational-intensive processes easier (without having to develop the entire app in the outsourced system though). DaaS provides online data: as the aforementioned example, a company that outsource the data to a DaaS provider gains full management of data (so backup, security, integrity, availability) through a web application. CaaS represents a bunch

²That's a bold statement. But with the right approximations, seems legit.

³Yes it's a technical term.

of systems *provided usually as a web service* that provides the communication mechanism for companies.

4.2 Cloud Systems Taxonomy

- Public cloud are the classic cloud systems available to the users. They're large scale computing infrastructure available on a rental basis, that can be pay-as-you-go or fixed costs.
- Private cloud services are cloud systems *that are not shared with other users* so that are fully enclosed in the boundaries of an organization/company.
- Community clouds: a network of private clouds. This approach differentiates from the pure private cloud approach in the complexity of the system managed. The idea is still a single cloud platform for a set of organizations.
- Hybrid cloud: "private clouds that makes available some services".

4.3 Edge and Fog Computing

Edge and fog computing are two "emerging" paradigms that exploits intelligence at the edge of the network (of sensors, of data hotspots ecc) in order to provide faster communications and services.

Part IV

Methods

5 HDD and RAID Technologies

Memory and files in disks are organized in clusters to simplify the management of the data. We have:

$$\begin{cases} a = \text{actual size of a file on disk} \\ c = \text{cluster size} \\ s = \text{file size} \end{cases} \quad (1)$$

and it holds that $a = \text{ceiling}(\frac{s}{c}) \times c$.

5.1 Data and Metadata

Data is the content stored in a mass storage device. Metadata is (as the name suggests) additional data stored in the storage device *that not contains actual content*. Metadata contains indexes, addresses, cached data that is used to manage *the actual data stored*.

5.2 Magnetic Hard Drives

Magnetic disks have an internal structure and an external interface to manage files and data. The internal structure is often based in c/h/s coordinates, that are cylinder, head and sector locations. The external interface is usually composed of clusters. The traslation is carried out directly by the electronics on the disk.

5.2.1 Delays

The principles on which the HDD storage is based relies on some phisical objects and their movement. The intrinsic limits of this system translates into delays in the reading of data. Four type of delays can be identified:

- Rotational: it's the time occurred to move the portion of plate under the desired head. It's related to RPM, of course
- Seek: it's the time needed to a head to move from a track to another
- Transfer: actual time to read the bytes
- Control: overhead time, related to the circuitery that manages the hardware

So, in order to calculate the actual time that passes between the issue of the read command to the presence of the desidered file on the I/O bus we have to sum all the delays.

$$T_{read} = T_{rotate} + T_{seek} + T_{transfer} + T_{overhead} \quad (2)$$

Usually, the transfer speed is usually given as transfer ratio $\frac{MB}{sec}$.

Remember: "data locality" (that can be seen as the opposite of fragmentation, in a sense) impact heavily on the total IO time: in fact, the seek time and rotational delay should be calculated only for the blocks that actually need those. The new formula become:

$$T_{read} = (1 - \text{locality fraction}) * (T_{rotate} + T_{seek}) + T_{transfer} + T_{overhead} \quad (3)$$

Another plausible assumption is to nullify the total overhead needed.

Reducing Latency Caching is the most used mechanism to make data available *faster* in order to reduce latency. Usually, at hardware level, the cache is implemented by means of a phisycal additional RAM memory built directly in the HDD.

Read Cache Often accessed data access can be reduced if there's no need to go down to the plate in order to read it all the times.

Write Back Cache Write buffer: writes are cached before accessing the disk, and then flushed at the end. N.B.: the "write finish" signal is issued at the end of the *caching* of the data.

Write Through Cache No write buffer: the "write finish" signal is issued after the write is actual written on the disk.

Scheduling As always, reordering operation in order to maximize efficiency is a solid approach to enhance performance, in this case reducing latency in accessing the disk. These kind of approaches (that usually prefers "near" sector) are prone to starvation, usually. Workaround to starvation problem usually are based on linearizing the traversing of the cylinders, like the elevator mechanism.

5.3 RAID

Redundant Array of Inexpensive Disks is a technology that exploits the parallelization of I/O bus to multiple disks in order to enhance security, write and read speed, and fault tolerance. Data are copied / distributed on multiple disks, that are presented to the OS as a single one. There are 6 (or 8) different raid configurations:

5.3.1 RAID 0

Data is just striped across disks. This is achieved by splitting the request to all the disks in the array. No data is replicated. RAID 0 provides high read/write speed, but worsen the reliability of the whole system: multiple disks + no backup (redundancy) options jeopardize reliability. It's by far the fastest configuration though.

5.3.2 RAID 1

Data is mirrored. On two disks, the same copy of data is stored. This improves read speed (you can split the read of blocks) but worsen capacity and write speed. Obviously, reliability is increased. On more than 2 disks (where the mirroring is quite straightforward) raid 1 and 0 can be combined:

- RAID 0 + 1: strip then mirror. Data is striped on the first set of disks and then *mirrored striped on the second set*.
- RAID 1 + 0: mirror then strip. The data is first mirrored in two clusters and then striped in the individual clusters. This configuration is more fault tolerant than the 0+1, because the more fault tolerant controller (the RAID 1 one) is closer to the disks. This statement can be formally proven.

Raid 0+1 and 1+0 can be imagined as a series of raid controllers.

5.3.3 RAID 2 and 3

RAID two and three use interleaving to store chunks of data and to generate redundancy. These two technologies are not used nor of interest now.

5.3.4 RAID 4

RAID 4 delegates an entire disk of the array to storing parity bits (calculated by XORRING all the bits in the other disks in that position). The parity bits allows to *reconstruct data* when a single disk fails, because we can XOR back the parity configuration and rebuild the data. This comes at a cost in writes, especially in random ones: updating the parity bits (that are all on the same disk \Rightarrow they cannot be parallelized) costs a lot if the writes are not sequential or not in the same XORRED stripe. The rest of the data is striped across the other disks. This configuration works better on a *odd* number of disks, due to the "extra" one to dedicate to the parity bits.

5.3.5 RAID 5

Why should I accumulate all the parity bits in a single disk? RAID 5 overcomes the limits of RAID 4 by *striping* across the array the parity blocks. This still reduces the capacity of the array *by one entire disk*, but the random writes speed up (you can now parallelize parity writes).

5.3.6 RAID 6

RAID 6 further improves RAID 5 idea, doubling the parity blocks across disks. The parity bit generation algorithm is different: Solomon-Reed's encoding is used instead of simple XOR. RAID 6 "throws away" two disks in the array, but the entire configuration can now tolerate the failure of two disks. Despite the super high fault tolerance of the system though, the write speed is worsened by a lot.

5.3.7 Math Stuff

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

		RAID 0	RAID 1	RAID 4	RAID 5
	Capacity	N	$N / 2$	$N - 1$	$N - 1$
	Reliability	0	1 (maybe $N / 2$)	1	1
Throughput	Sequential Read	$N * S$	$(N / 2) * S$	$(N - 1) * S$	$(N - 1) * S$
	Sequential Write	$N * S$	$(N / 2) * S$	$(N - 1) * S$	$(N - 1) * S$
	Random Read	$N * R$	$N * R$	$(N - 1) * R$	$N * R$
	Random Write	$N * R$	$(N / 2) * R$	$R / 2$	$(N / 4) * R$
Latency	Read	D	D	D	D
	Write	D	D	$2 * D$	$2 * D$

Where, more specifically:

$$\begin{cases} \text{Random transfer rate } R = \frac{\text{transfer size}}{\text{time to access}} \\ \text{Sequential transfer rate } S = \frac{\text{transfer size}}{\text{time to access}} \end{cases} \quad (4)$$

The difference between the two is the way the operations are actually issued on the disk.

6 Performance

6.1 What is Performance?

Classic engineering question: how much did I do well? How do I measure, evaluate the goodness of my job? What's the most adapt figure of merit? How do I abstract enough from the reality in order to design the right solution (modeling problem)?

6.2 Approaches to Evaluation

Two main schools of thought:

Measurements Based Techniques Techniques as directly measuring the performances on the system, benchmarking it still "in beta" or prototyping approaches are based on the direct testing of the physical model or a copy of it.

Model Based Techniques Opposed to using the system itself to test performances, we can model it (build an abstract copy of it) and test that. Among these techniques are

1. Analytical, numerical tests
2. Simulation
3. Hybrid approaches

These techniques are more "broad": they can test cases that it's possible that a brute force approach as benchmarking never reach.

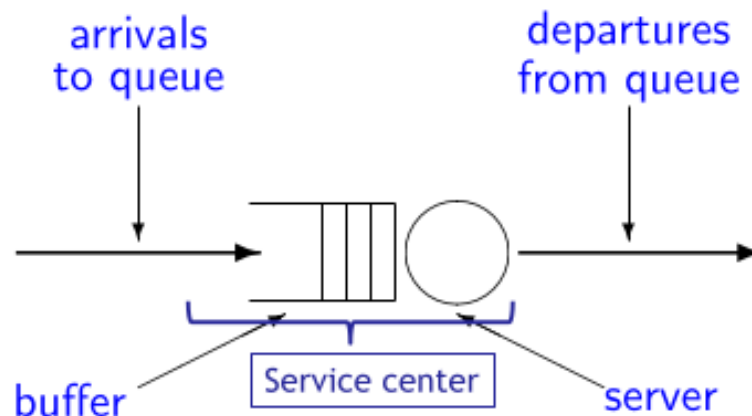
6.3 Evaluating Performance through Queuing Networks

We'll use a model based approach to evaluate performance indexes. The model for us is queuing networks, classic distributed racing systems with scarce resources. These are usually represented as a set of services correlated with a queue to store clients. A client is served and then leaves the service.

6.3.1 The Model

As said, a queuing network is composed by

1. Customers, or *clients*, that need to use a service
2. Service centers, that provide services. A service center is itself composed of
 - (a) A buffer, the queue of arrivals
 - (b) A "server station" that provide the very server.



6.3.2 Parameters of the Model

To characterize our model, we usually quantify some of its aspects, such as

Arrivals Arrivals represent the clients of a system. In a network, they can come from either an external source, from another service station, or from the service station that they've just used (loop back arc).

Services Servers provide services. They can "hold" a client until the job is done, then they pass to the next element in the queue.

Queue The queues represent the buffer of clients before servers. When a client leaves the server, another one from the queue is selected *according to the queuing policy*.

Population The population is the set of clients in the network. Ideally, each member is indistinguishable from the others; usually we can divide members in classes of members that all show the same behaviour. The behaviour of a member is characterized by several factors, as arrival rate, service demand, post service behaviour...

Routing The "place a client goes when is job has been served" is a parameter of the system, in particular is defined by the *routing policy* of that system. Routing policies can be

- Probabilistic
- Channel based (like round robin)
- Queue based (like join-the-shortest-queue)

6.4 The Performance Estimation Process

Once a model has been built (so in our case, a network of queuing processes) in order to evaluate the performance we have to parametrize the model itself and then estimate the performance indices. To do so, we must mathematically formalize how the model behaves: we can do such through the use of Operational Laws.

Operational laws are basically equations that usually represents the averaged behaviour of each system. Operational laws are based on observable variables, that can be "read" from the system. For example, given our model, we can without effort observe the variables

- The number of arrivals A
- The number of completions C
- The amount of time the system is busy B
- The average number of jobs in the system N
- The time span we used to observe the system T
- The visits to each station V

We can already derive some operational laws:

$\lambda = \frac{A}{T}$	Arrival rate
$X = \frac{C}{T}$	Throughput
$U = \frac{B}{T}$	Utilization time
$S = \frac{B}{C}$	Mean service time

Notice that these laws can refer the whole system or the single service station. We direct our experiments on the model in order to have $\lambda = X$, also called "job flow balance" condition. From now on, if an property has a subscript, it's indicating that property (or operational law) of a particular service station or subsystem. Also, we need to introduce a new property if we're talking about single service station: this is N , the number of jobs in the *whole service station*⁴.

Utilization Law From literally one substitution we can derive the so called utilization law:

$$U_k = X_k \times S_k \quad (5)$$

This law expresses the "occupational fraction" of a station, so how much a station is busy. Expressed in words, it's the rate of completion of requests times the time needed to carry out one of them. It's easy to notice how, if this value is greater than one, the queue will grow in time.

⁴so queuing + served

Little Law The Little's law (or result, or lemma) states

$$N = X \times R \quad (6)$$

So the number of requests in the system is equal to the throughput multiplied by the average time R a request spends in the system.

Interactive Response Time When measuring performance in an interactive system, we must also model in some way how the user behave in front of the system itself. To do so, we introduce *think time* Z and we redefine N as $N = X \times (R + Z)$.

Forced Flow Law The FFL is another operational law that binds the throughput of the entire system to the throughput of the single service stations.

$$X_k = V_k \times X \quad (7)$$

Where V_k is the visit count for the particular service station k , defined as $\frac{C_k}{C}$ so the fractions of completions of a single service station. Visit count is important when related to total completions: when V_k is greater than one, the number of completions at system level corresponds to a higher number of visits to the single service station.

Service Demand The service demand represents the amount (in time) of service asked *to a specific service station*. It's computed as

$$D_k = S_k \times V_k \quad (8)$$

Remember: Service Time \neq Residence Time.

Residence and Response Time The response time \tilde{R} is the time spent by a job in a service station, counting the queue time. The residence time instead is the time spent by a job in a service station *during all system time*: so accounting for the loops and reiteration. The pretty straightforward relation is $R_k = V_k \times \tilde{R}$, that's the same relationship between Demand and Service Time.

General Response Time Law The GRTL calculates the average response time for a job:

$$R = \sum_k (V_k \times \tilde{R}_k) = \sum_k (R_k) \quad (9)$$

6.4.1 SUMMARY

Ok so, we've seen some confusing stuff so far. Let's put all together in a nice table.

Name	Letter	Formula
Number of arrivals	A	\
Number of completions	C	\
Busy time	B	\
Average number of jobs in the system	N	\
Observation time	T	\
Visit to each station	V	\
Think time	Z	\
Arrival rate	λ	$\frac{A}{T}$
Throughput	X	$\frac{C}{T}$
Utilization time	U	$\frac{B}{T}$
Mean service time	S	$\frac{B}{C}$

Utilization law	\	$U_k = X_k \times S_k$ or $U_k = D_k \times X^5$
Little's law	\	$N = X \times R$
Little's law, interactive response time	\	$N = X \times (R + Z)$
Forced flow law	\	$X_k = V_k \times X$
Service demand	\	$D_k = S_k \times V_k$
General Response Time Law	\	$R = \sum_k (R_k)$

7 Performance Asymptotic Bounds

Performance bounds are the asymptotic values that our figures of merit can reach in a system. These can be useful in the modeling process to find the element that impact the most the performances, among the others the bottlenecks. Asymptotic analysis provides optimistic and pessimistic bounds on two major performance indexes: throughput X and response time R .

Notation and Quantities We'll focus on expressing X and R as functions of

- $K \rightarrow$ number of service centers
- $D \rightarrow$ sum of all the service demands at the centers
- $D_{max} \rightarrow$ largest service demand at a single center
- $Z \rightarrow$ average think time for interactive systems

We'll analyze

- Optimistic scenarios, where we search X 's upper bound and R 's lower bound
- Pessimistic scenarios, viceversa: throughput lower bound and maximum values of response time

7.1 Tables of Formulae

Bounds for throughput

	Open Systems	Closed Systems
Light Load	$X(\lambda) \leq \frac{1}{D_{max}} \quad (10)$	$\frac{N}{ND + Z} \leq X \leq \frac{N}{D + Z} \quad (11)$
Heavy Load		$\frac{N}{ND + Z} \leq X \leq \min(\frac{N}{D + Z}, \frac{1}{D_{max}}) \quad (12)$

Bounds for response time:

	Open Systems	Closed Systems
Light Load	$R \geq D \quad (13)$	$\max(D, ND_{max} - Z) \leq R \leq ND \quad (14)$
Heavy Load		

⁵it actually makes sense, look next formulae

7.2 SUMMARY

Open Models

$$\emptyset$$
$$X(\lambda) \leq 1/D_{\max}$$
$$D \leq R(\lambda)$$

Closed Models

$$\frac{N}{ND+Z} \leq X(N) \leq \min\left(\frac{N}{D+Z}, \frac{1}{D_{\max}}\right)$$
$$\max(D, ND_{\max}-Z) \leq R(N) \leq ND$$

Part V

Dependability

8 Probabilistic approach

Dependability represents the *availability performance* of a system. It encompasses

- Reliability
- Availability
- Maintainability

Dependability is approached with a statistical/probabilistic POV due to the high human component in it.

Two functions describes the system: $F(t)$ and $f(t)$, respectively the cumulative function and the fault probabilistic distribution. The former represents the *unreliability* of the system analyzed. So, we can define $R(t) = 1 - F(t)$ as the reliability function.

Probabilistics recall:

$$f(t) \rightarrow \text{probabilistic distribution of failure} \quad (15)$$

$$F(t) = \int_0^t f(t)dt \rightarrow \text{cumulative function, unreliability} \quad (16)$$

$$R(t) = 1 - F(t) \rightarrow \text{reliability function} \quad (17)$$

To be noticed: the function $F(t_i)$ represents the probability that component i is working at time t_i *knowing it was working at time 0*. This is to be taken in mind to make a comparison with the failure rate.

From this we can define the Mean Time To Failure for a component (that's the expected value):

$$\begin{cases} MTTF = \int_0^\infty t \cdot f(t)dt \\ MTTF = \int_0^\infty R(t)dt \end{cases} \quad (18)$$

We can also define the failure rate (mathematically, the conditioned probability):

$$\lambda(t)dt = F(t < T \leq t + dt | T > t) \quad (19)$$

So the failure rate represents the probability of failure *assuming the component was working the instant before*. This function can be seen as the number of failures in a given interval.

8.1 Failure Rate

Type of malfunctioning:

- Fault: physical defect or software bugs.
- Error: program incorrectness that can result from a fault.
- Failure: nonperformance of some actions that were expected. They can be result of an error.

Properties of the failure rate Probabilistically, failure rate is

$$\lambda(t)dt = F(t < T \leq t + dt | T > t) \quad (20)$$

So we can derive

$$\lambda(t)dt = \frac{P(t < T < t + dt \cap T > t)}{P(T > t)} \quad (21)$$

From the definition of combined probability, with P as probability. Given

$$P(t < T < t + dt) \cap P(T > t) = P(t < T < t + dt) \quad (22)$$

We obtain

$$\lambda(t)dt = \frac{f(t)dt}{R(t)} = \frac{dF(t)}{R(t)} = -\frac{dR(t)}{R(t)} \quad (23)$$

Reliability as Weibull Failure rate $\lambda(t)$ function has a peculiar function shape: a *bathub* (or long U) shape. The failure rate is high in the starting period and decreases (infant mortality effect) to the constant level during the useful lifetime (constant probability of failures) and it raises again at the end (wear out period).

In fact, reliability follows the *Weibull distribution* so defined: $y(x) = e^{-(\frac{x}{\alpha})^\beta}$. So (due to the $\lambda(t) = f(t)/R(t)$ relation) we obtain

$$\lambda(t) = \frac{\beta}{\alpha} \left(\frac{t}{\alpha}\right)^{\beta-1} \quad (24)$$

Reliability as exponential We can also model the reliability as an exponential function (so $R(t) = e^{-\lambda t}$) we than obtain

$$\begin{cases} F(t) = 1 - e^{-\lambda t} \\ f(t) = \lambda e^{-\lambda t} \\ MTTF = \int_0^\infty t \cdot \lambda e^{-\lambda t} dt = \frac{1}{\lambda} \Rightarrow \lambda = \frac{1}{MTTF} \end{cases} \quad (25)$$

So we can express the reliability as function of MTTF: $R(t) = e^{-\frac{t}{MTTF}}$. And we all know that in certain conditions (like $\frac{t}{MTTF} \ll 1$) an exponential function can be approximated to a linear function.

8.2 System level reliability

We can model the system reliability mainly with RBDs: Reliability Block Diagram. These simply outline the operational dependency between components. The main assumption of the RBD is that the failures are *independent*, that implies that are no "cascading failures".

General formulae for the Reliability of systems:

$$\begin{cases} \text{Serial components: } R_s(t) = \prod_{i=1}^n R_i(t) \\ \text{Parallel components: } R_p(t) = 1 - \prod_{i=1}^n (1 - R_i(t)) \end{cases} \quad (26)$$

System's MTTF We can redefine the mean time to failure to cover the complex systems, keeping in mind the two configurations we use: serial or parallel components.

$$\begin{cases} \text{Serial components: } MTTF_s = \frac{1}{\sum_{i=1}^n (\frac{1}{MTTF_i})} \\ \text{Serial identical components: } MTTF_{ss} = \frac{MTTF}{n} \\ \text{Parallel components: } MTTF_p = \sum_{i=1}^n (MTTF_i) - MTTF_s \\ \text{Parallel identical components: } MTTF_{pp} = MTTF_i * (\sum_{i=1}^{n-1} (\frac{1}{n-i})) \end{cases} \quad (27)$$

Disclaimer MTTF is an *integral measured value*, it must be handled with care when the number of components and links grow. The perfect workflow is to fix unreliability, then calculate reliability⁶, derive to obtain $f(t)$, then *at the end* derive MTTF as expected value of $f(t)$.

8.3 Availability

Introducing MTTR, Mean Time To Repair. This represents the average time required to replace a failed component and "bring the system back up" after a failure. It can be the restart time (for software module) or the replace time (for an hardware component).

Another interval to be taken into consideration when talking about availability is the Mean Time Between Failure, that's just the sum of $MTTF + MTTR$. Availability is defined as the probability of a system to be up and running at a given instant. So $Av = \frac{MTTF}{MTBF}$ that corresponds to

$$Availability = \frac{MTTF}{MTTF + MTTR} \quad (28)$$

⁶additional step, not always needed

The slight difference between availability and reliability takes into consideration the repairing of a system in the case it goes down. The availability function is calculated with *the same identical formulas* used to calculate the reliability for parallel and serial components:

$$\begin{cases} \text{Serial components: } A_s = \prod_{i=1}^n A_i \\ \text{Parallel components: } A_p = 1 - \prod_{i=1}^n (1 - A_i) \end{cases} \quad (29)$$

Talking about MTTF when taking into consideration that components can be *repaired* must be carefully handled: for example, if a parallel system has *interleaving* failures, it never fails entirely. A way of calculating MTTF for repairable system consists in inverting formula (9), obtaining

$$MTTF = \frac{Av * MTTR}{1 - Av} \quad (30)$$