Handouts of fog and cloud computing

Leonardo De Faveri

A.A. 2021/2022

Indice

Chapter Nr.1

Introduction

Definition 1 - Data science.

Data science is the science of learning from data, and it employs various techniques such as statistical methodologies, machine learning and data mining.

Data science relies on large amount of data that is constantly increasing in quantity, variety and veracity (i.e. data is more and more accurate and conform to the studied reality). Finally, since data is fast in production, it needs to be collected and manipulated as fast.

Because of these characteristics, we are now facing many challenges in storing, sharing, analysing, transfering and securing data. To address these problematics, distributed and scalable systems are required. This resulted in the proliferation of large data centers that, by storing tons of servers, have centraliesed data manipulation and storing. This came in hand with a reduction in plants, IT assets, operating and energy costs.

Cloud computing allowed all of this to be possible and furthermore, transformed what was a product into a service that can suit specific users needs. For example, companies that maintain data centers can provide storage, computational power, network access and many other commodities to their customers as on-demand services for which they pay-as-they-go, meaning that they can pay only for the resources they actually use.

To be considered convinient, a cloud service must satisfy some requirements:

- *Connectivity*: it must be possible to move data through the network;
- *Interactivity*: users need to have an interface through which monitor their products, the resources they're using and made configurations;
- *Reliability*: users mustn't be affected by maintainer's failures (i.e. providers must prevent and handle them);
- *Performance*: services must be better than what customers already have;
- *Pay-as-you-go*: there mustn't be upfront fees and users must only pay for what they use;
- *Programmability*: it must be easy for users to develop and maintain their products;
- *Data management*: providers must be able to handle large amount of data;
- *Efficiency*: the plants must be efficient on costs and power usage;
- *Scalability and elasticity*: providers must be flexible and give rapid response to users needs;

Definition 2 - Cloud computing.

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

So, *cloud computing* relies on 5 key points:

1. *Shared or pooled resources*: resources are retrieved from a common pool;
2. *Broad network access*: it must be available from anywhere through internet connection and must be accessible using any platform;
3. *On-demand automated reservation*: customers can reserve resources as needed without requiring human interaction with cloud service provider;
4. *Rapid elasticity*: resources can be rapidly and automatically scaled up and down to satisfy customers demands;
5. *Pay by use*: services are metered like a utility, so users must pay only for the services they're using, and they must also be able to cancel them at anytime;

Sure, centralizing too much can be a bad idea (e.g. if an entire data center goes down, tons of services may be unable for a long time and for everyone), and many operations that require just a "small" portion of data might be computed outside a data center and nearer to the source of that data. From this idea, originated the concept of *fog computing*.

Definition 3 - Fog computing.

Fog computing is an evoluting of cloud computing in which computation is decentralized by subdividing it into multiple nodes thast act indipendently. Groups of nodes referes to an aggregation node that handle them and more aggregation nodes are then connected to a central point that provides, among others, an interfaced for users.

Note. A *fog node* is an active component that performs some operations and not hust a passive data collector such as a sensor.

This may allow reducing resources required by a single data center, since it might store and handle only the results of manipulations already performed by *fog nodes* or *aggregation nodes*.

Chapter Nr.2

Could ecosystem

2.1 Some definitions

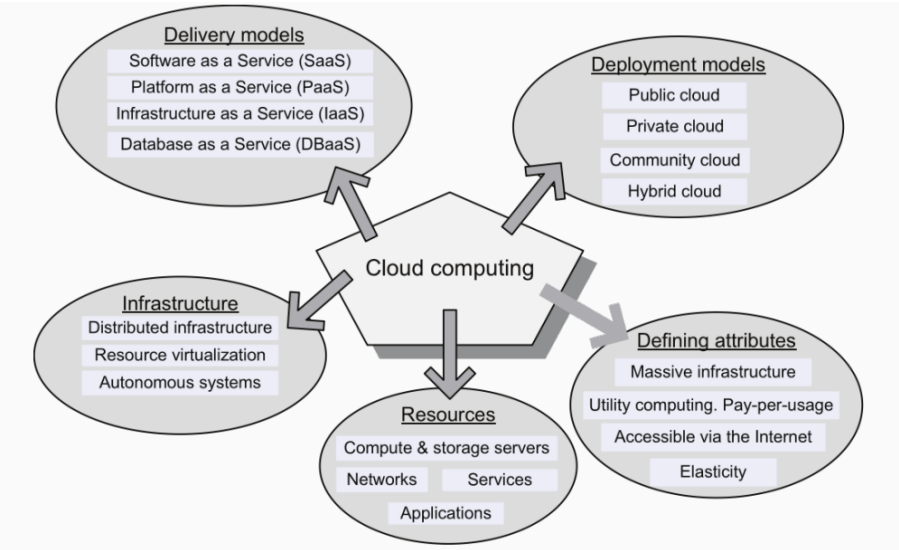


Image 2.1: Five key aspects of *cloud computing*

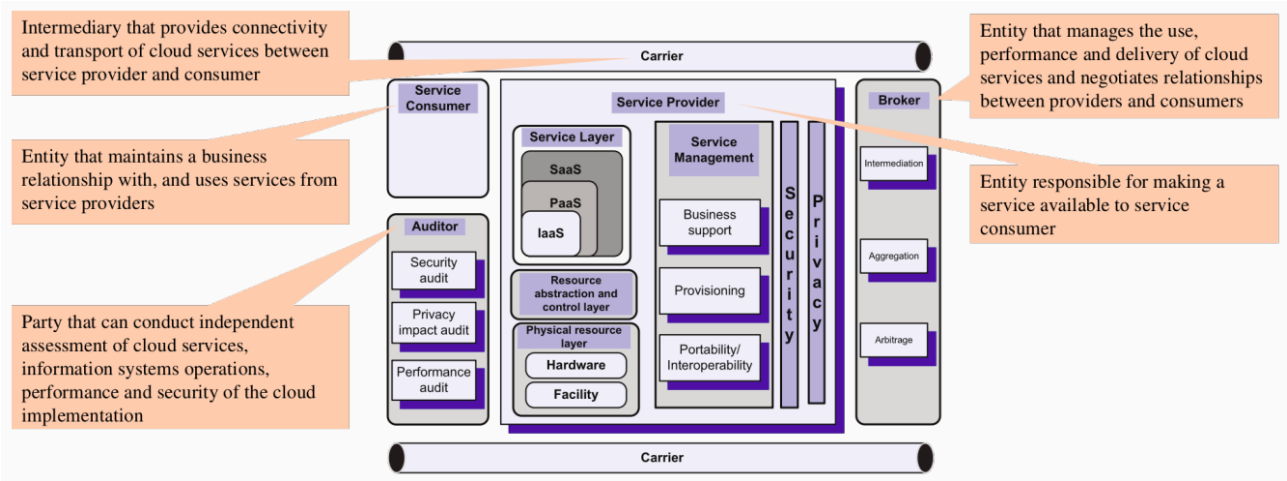


Image 2.2: *NIST* reference model for *cloud computing*

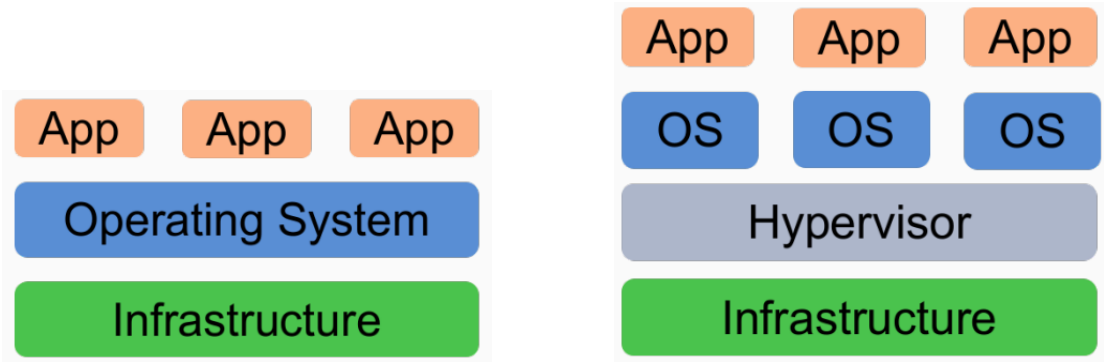
Note. A carrier is someone who provides access to a cloud service, such as Telecom, while a broker is a subject that handles the delivery of cloud services to users, such as a portal to the cloud (e.g. Booking.com).

Before analysing some key aspects of *cloud computing*, some definitions are required.

2.1.1 Virtualization

Definition 4 - Virtualization.

Virtualization allows the abstraction of computing resources by hiding their physical characteristics from the way system, applications and users interact with them.



(a) Non-virtualized system (b) Virtualized system

Image 2.3: General architecture of virtualized systems

2.1.2 Single-tenancy VS multi-tenancy

Definition 5 - Single-tenancy.

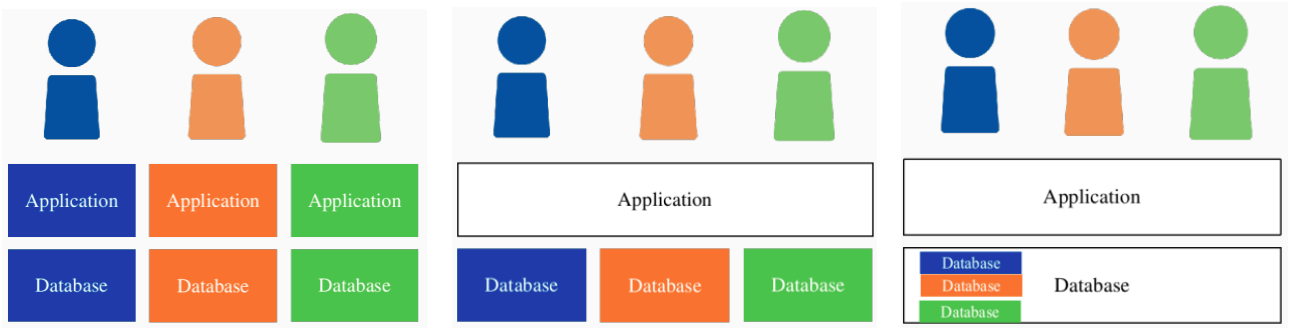
With single-tenancy each user has its own software instance.

Definition 6 - Multi-tenancy.

With multi-tenancy a single isntance of a software can serve multiple users.

As a consequence of these definitions, we can say that with *single-tenancy* each user requires a dedicated set of resources to fulfill its needs, while *multi-tenancy* allows sharing resources managment and costs among all of them.

Actually, in a *multi-tenancy* enviroment, a group of users who share common access with specific privileges to the software isntances, is called *tenent*. An instance includes, among others, data, configurations and users management.



(a) *Single-tenancy* (b) *Single-tenancy* for database and *multi-tenancy* for the application (c) *Multi-tenancy*

Image 2.4: *Single-tenancy* VS *multi-tenancy*

2.1.3 Elasticity and resource provisioning

Resource provisioning is the ability of adding or removing resources at a fine grain (e.g. one server at a time) with a short lead time (e.g. minutes). This allows a close matching of resources and workloads and, together with the *pay-as-you-go* model, brings elasticity to the users who no longer need to worry about sudden spikes of resource usage. The key advantage of elasticity is that it reduces the problems resulting from *underprovisioning* and *overprovisioning*.

Overprovisioning happens when the workload is using much less resources than the ones that have been allocated, thus resulting is a waste. Simmetrically, *underprovisioning* means that the available resources are insufficient to serve the requests, thus resulting in bad performances and possible loss of clients.

When users relies on proprietary resources (e.g. company's private servers) the amount of allocated resources must be determined by the quantity that is required to meet the highest predicted pick. Since it's difficult to predict picks, most of the time there will be redundant resources.

Another advantage of the way *cloud computing* provides resources is on costs, because with a *cloud approch* there isn't any inital cost for buying and setting up the infrastructure.

2.2 Delivery models

Going back to the key aspects of *cloud computing*, delivery models define the kind of product that is provided. There are three main types of models:

- *Software-as-a-Service* (*SaaS*): an application is provided to the users through the web;
- *Platform-as-a-Service* (*PaaS*): APIs and deployment environments are provided to developers;
- *Infrastructure-as-a-Service* (*IaaS*): computing resources are provided to system administrators;

2.2.1 Software as a service

Applications are supplied by service providers and users have no control over their capabilities and underlying cloud infrastructure. This model isn't suitable for real-time applications or applications for which data isn't allows to be stored externally.

Examples Google Drive, Google Docs, Spotify

2.2.2 Platform as a Service

PaaS allows developers to deploy applications (consumer-created or acquired from others) using tools and programming languages supported by the service provider. Developers have controll over the deployed applications and, possibly, over the app hosting environment. However, they still don't have access to the underlying insfrastructure (e.g. network devices, OSs, storage).

This model isn't indicated for portable applications, apps in which proprietary programming languages are used or which require hardware and software customization.

Examples Google App Engine, Heroku

2.2.3 Infrastructure as a Service

Services provided by this model includes: server hosting, storage, computing hardware, operating systems, virtual instances, load balancing, internet access and bandwidth provisioning.

System administrators can manage OSs, storage, deployed applications and may even have little control over network components such as firewalls. So, they're able to deploy arbitrary software including operating systems. However, there is still a below instrastructure that can't be accessed.

Examples Amazon EC2

Note. Everything can be deployed ad a service, for example databases or hardware, thus *Database-as-a-Service* and *Hardware-as-a-Service* may exist.

2.3 Deployment models

Deployment models describe the way the cloud infrastructure may be accessed and by whom and who is the owner. In particular, there are four types of environment.

2.3.1 Public cloud

- *Consumer*: general users or large industrial group;
- *Service provider*: there's an organization that settles down and manages the infrastructure;
- *Resource location*: all resources are within the premises of the cloud provider;
- *Multi-tenancy model*: different consumers are served by the same instances;

2.3.2 Private cloud

- *Consumer*: a specific organization;
- *Service provider*: the same organization that uses it or a third party one;
- *Resource location*: it can either be on-premises if the organization doesn't want to remotely host data, on off-premises if it relyes on a third party private cloud;

2.3.3 Community cloud

- *Consumer*: a community composed by one or more organizations which share common concerns sush as their mission, policies and security considerations;
- *Service provider*: either the organizations or a third party;
- *Resource location*: either on-premises and off-promises;

2.3.4 Hybrid cloud

It's the composition of more deployment models which remain unique entities, but are bound together by standardised or proprietary technologies that enables data and applications portability.

For example an organization might use public cloud for some aspects of its business and a private one for its sensitive data.

Chapter Nr.3

Virtualization

3.1 Introduction

Before virtualization tooked place, companies used to have varius servers but most of the time they were found to be idle. The problem was, that due to OSs failures, they couldn't run flawlessly more than one application at a time. In particular, OSs couldn't provide:

- *Full isolation of configurations and shared components*: for example an app requiring version 1.0 of some library, created conflicts with another app requiring a different version for the same library;
- *Temporal isolation for performance predictability*: it could happen that one app used a lot of resources causing the degradation of performances for another app;
- *Strong spacial isolation for security and reliability*: if some app crashed it might have compromised others;

All of this lead to companies needing to have a lot of different servers running even if they were massively underutilized and were consuming a lot of power.

Computing virtualization established because it offered a flexible way to share hardware resources between different operating systems. This came in hand with both advantages and disadvantages.

Advantages

- *Isolation*: critical applications can run in different and easily isolated OSs. Also, different services can run in the same host, into different *virtual machines* that could even use different CPU cores;
- *Consolidation*: different OSs can run at the same time on the same hardware, thus saving resources and minimysing costs and energy consumption;
- *Fleibility and agility*: the system admin has complete control over *virtual machines* execution, and it can pause and restart them. Moreover, it might migrate one to a different host, or duplicated it to address a workload peak. Finally, it's easy to recover from a disaster (e.g. restarting a VM from a safe snapshot) or spawn a new *virtual machine*;

Disadvantages

- *Additional overhead*: since each *virtual machine* needs it own OS, more hardware resources are required;
- *Increased difficulty in handling different hardware*: it might be difficult for the virtualization manager to grant some application access to special components;

Virtualization can be used for both server and dektop virtualization, but its main usage is in server virtualization. In fact, since more OSs can run on the same physical machine using a configurable amount of resources, it is no longer necessary for system admins to buy machines with specific physical characteristics. Instead, they can just buy *COTS* (Common Off The Shelf) hardware and, on top of which, create different *virtual machines* with their required specifications. This is convinient because companies can buy tons of equivalent servers, put them into a datacenter and virtualise their resources. Also, buying in large volumes often results in a lower individual price.

3.1.1 Some definitions

Before diving into more technical aspects of virtualization, let's give some definitions.

Definition 7 - Layering.

Layering is a common approch to manage system complexity which allows to minimize the interactions among the subsystems of a complex system. The description of those subsystems is also simplified because each of them is abstracted through its interface to the others. Finally, layering allows to manage each subsystem individually.

For example, a computer can be divided into two main layers: hardware and software, and software can then be divided into OS, libraries and applications. The interfaces between software layers are ISA, ABI and API.

Definition 8 - Virtual machine (VM).

Virtual machines are software emulation of a physical machine that executes both OS and applications as if they were executed on a physical machine.

When taking about *virtual machine* we need to distinguish between two actors:

1. *Host OS*: it's the OS that is running on the physical machine and that is handling virtualization;
2. *Guest OS*: it's the OS running on a *virtual machine*, and it shouldn't be aware of being running in a virtualised enviroment;

Definition 9 - Hypervisor.

The hypervisor is the software in charge of the virtualization process, meaning that it has to virtualise the hardware resources and this is done by:

- *Assigning, when possible, a specific set of resources to each virtual machine while garanteeing that each of them doesn't get access outside its boundaries;*

- *Arbitring access to shared resources that cannot be partitioned;*

Note. The *hypervisors* is often refered to also as *Virtual Machine Manager* (*VMM*).

The *hypervisor* is often implemented as a Linux-based stripped-off OS (i.e. an OS with minimum functionality) to make it more efficient and more easily securable. The *hypervisor* exports also a set of "standard" devices to hosted OSs (i.e. the most common pieces of hardware that are supported from most OSs).

The *hypervisor* must provide a "virtual hardware" to *guest OSs* with the exact characteristics specified in a given hardware profile. Also, the real hardware may be different from the virtualised one because it depends on the devices that are exposed by the *hypervisor*.

To allow *guest OSs* to run in a virtualised enviroment, CPU, memory and I/O need to be virtualised correctly.

3.2 CPU virtualization

The *VMM* assigns one or more CPU cores to the VMs so that they can run their OS. The ISA of the virtualised hardware will usually be the same of the physical one, but it is not mandatory. Basically, if they're different there will be an emulation process that will translate messagges between them. However, since the emulation process works by doing a binary translation between the two different ISAs, it is too slow to be generally convinient.

Going back to the *VMM*, it must satisfy three characteristics:

- The execution enviroment exported by it must be identical to the physical one, so that OSs can run unmodified;
- It must have complete control over real system resources, so that any *guest OS* can access only those components it has been grant access to;
- It must run the virtualised systems efficiently;

3.2.1 Some definitions

System based on x86 or x64 architecture are usually model into a *privileg ring* structure. In particular, there are four privileg levels with decreasing privileges as you move away from the center. In fact, *ring 0* is dedicated to the OS kerner, and *ring 3* to generic applications.

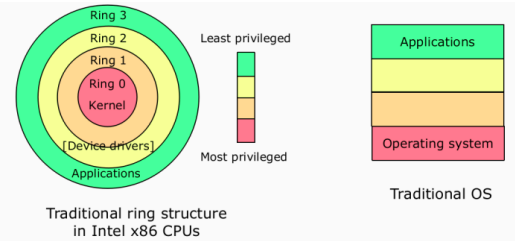


Image 3.1: *Privilege-ring model*

Virtualization can use "ring de-priviledging", a technique that runs *guest OSs* in level greater than *ring 0* so that they have limited privileges and therefore can't interfere with each other or with the *VMM*. However, there are two possible modes:

- *0/1/3*: the *VMM* runs at *ring 0*, *guest OSs* at *ring 1* and applications at *ring 3*. Since, in x86 architeture, some privileges with respect to memory accesses are granted to *ring 0-2*, *guest OS* might still interfere with the *VMM*;

- *0/3/3*: the *VMM* runs at *ring 0*, *guest OSs* and applications at *ring 3*. This solves the previous problem, but *guest OSs* are no longer protected by malicious applications;

Definition 10 - Privileged instruction.

A privileged instruction is a CPU instruction that needs to be executed in a privileged hardware context.

Definition 11 - Sensitive instruction.

A sensitive instruction is a CPU instruction that can leak information about the physical state of the processor.

Note. *Sensitive instructions* are, for example, those that can read the register in which is stored the currect CPU privileg level.

To be virtualizable all CPU's *sensitive instructions* must be *privileged*.

Definition 12 - Trap.

A trap is an event that triggers the switch from an unprivileged context to a priviliged one.

If a *privileged instruction* is called while the CPU isn't running in kernel mode (the mode associated to *ring 0*), a *trap* is generated, so the CPU jumps to the *Hardware Exception Handler Vector* (HEHV) and executes said instruction in kernel mode.

Situations in which a *trap* can occur can be put in one of three buckets:

1. *Exceptions*: invoked when unexpected error or system mulfunction occur (e.g. *privileged instruction* executed in user mode);
2. *System call*: invoked by applications in user mode (e.g. application asking OS for system I/O);
3. *Hardware interrupts*: invoked by hardware events in any mode (e.g. hardware clock timer triggers events);

System call invocation and hardware interrupts In traditional OSs, when an application invokes a *system call*, the CPU will trap to interrupt handler vector in OS, then will switch to kernel mode and execute OS instructions.

Similarly, when an *hardware interrupt* verifies, CPU execution will stop and it will jump to interrupt handler in OS.

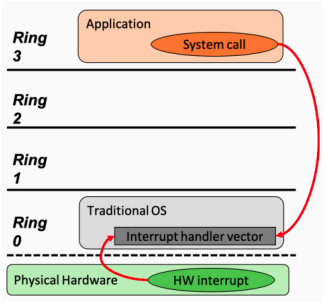


Image 3.2: *Trap handling* in traditional OSs

Diving deeper into *trap handling*, when a *trap* is generated, userland code (i.e. code outside the kernel) generates a *software interrupt* (e.g. thorugh the instruction `INT xx`). Hence, the generic interrupt routing of the OS is started, and it determines where to jump in the OS code to serve that interrupt. Finally, kernel jumps to the identified code, serves the interrupst and then returns control back to the caller (i.e. instruction `IRET`). All of this requires to load and parse the content of several memory locations, so it's rather slow.

A more modern way to serve interrupts uses `SYSENTER` and `SYSEXIT` instructions (`SYSCALL` and `SYSRETURN` in x64 systems) to speed up the process. Practically, userland code writes the address of the targeted kernel routine in a specific register, then `SYSENTER` is called and the kernel jumps to the selected routing reading the address from the register without additional accesses to memory.

Going back to virtualization Said this, we can go back to virtualization and talk about the three types of virtualization that exists:

1. *Full virtualization*: *guest OSs* can run unmodified;
2. *Paravirtualization*: *guest OSs* are aware of being running in a *virtual machine*, so they need to be modified;
3. *Hardware assisted virtualization*: the *hypervisor* exploits some functionalities provided by modern CPU chips;

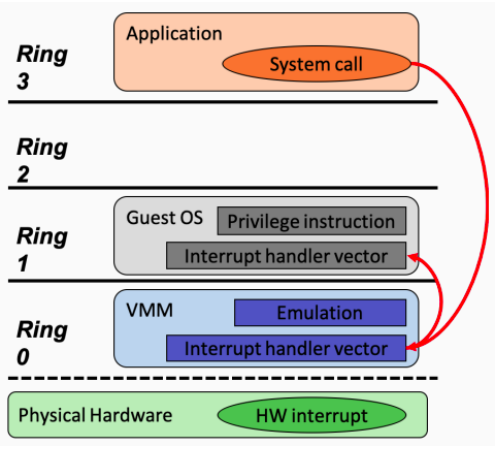
3.2.2 Trap & emulate paradigm

This paradigm allows *full virtualization* and provides that *guest OSs* run in an unprivileged enviroment, hence when a *privileged instruction* has to be executed, a *trap* is launched by the CPU. Then, that *trap* is intercepted by the *VMM* that emulates the effect of the *privileged instruction* for the caller (ofcourse only if it's legitimate) and, at the end, gives control back to *guest OS*.

Actually, when the *VMM* intercepts a *trap* it behaves differently based on the event that caused it. If it was causes by an application, then the *VMM* passes it direcctly to the *guest OS*. On the other hand, if it was caused by the *guest OS* itself, the *MVV* handles it by modifying the state of the *virtual machine*.

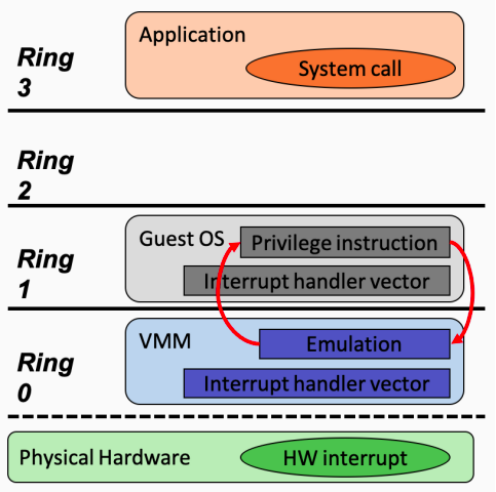
System call handling

When a *system call* happens, CPU traps it to interrupt handler vector of the *VMM*. This then jumps back to the *OS*. All of this, results in extra context switch operations and performance deteriorates further if the *guest OS* isn't able to handle the interrupt routing. So, the time spent to execute a single *system call* might be 10 times greater than what it would have been required by the *host OS*.



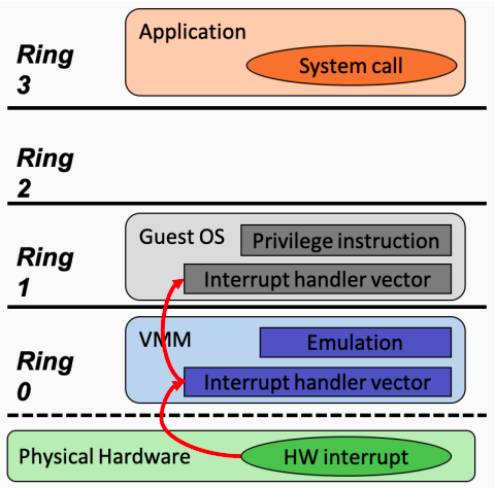
Privileged instruction handling

When a *privileged instruction* is executed it will be trapped to *VMM* who will emulate it. After that, *VMM* will give control back to the caller.



Hardware interrupt handling

When a *hardware interrupt* is launched, CPU will trap it to interrupt handler of *VMM* that then will jump to the corresponding interrupt handler of the *guest OS*.



Problems with systems virtualization In this paradigm, each time a *privileged instruction* is executed in an unprivileged context, a *trap* has to be generated and detected by the *VMM*. These actions are time-consuming, as we have just seen.

However, this process isn't necessary for all architetures, but unfortunatly, it is in x86 and x64 architectures. Moreover, these architectures presents some *sensitive instructions* (e.g. `POPA`, `POPF`) that don't trap when executed in an unprivileged context, hence, these architectures are said to be "non-virtualizable".

Possible solutions Therefore, we have some *sensitive instructions* that don't trap and, consequently, the *VMM* cannot emulate the correcct behavior during the execution.

To address this problem we can change virtualization paradigm or introduce some code into the *VMM* that parses the instruction stream to detect all *sensitive instructions* dynamically. Then, we can both use interpretation and binary translation. Interpretation in an old and slow approch in which emulating a single ASM instruction originates an overhead of one order of magnitude at least. Binary translation, on the other hand, introduces a lower performance overhead.

Dynamic binary translation The idea behind this approch is to dynamically translate "non-virtualizable" ISA to a virtualizable one during run time. In particular, dynamic means that translation is done on-the-fly at execution time and interleaved with normal code execution. Binary means that *VMM* translates binary code instead of source code and this is more efficient.

The pros of this technique is that it still allows *full virtualization* without needing specific hardware support, but virtualization overhead is still too high and several instructions or execution patterns (e.g. *system call*) are significantly slower than real execution.

Note. We could use caching techniques to recognize significant instruction patterns and increase translation speed.

Note. Original VMware *VMM* combined *Trap & emualte* with a system level *Dynamic Binary Translation*. *Guest OSs* run at *ring 1* and *VMM* inspected dynamically code to swap non-trappable portions of code with "safe" instructions.

3.2.3 Paravirtualization

The idea that drives this paradigm is to let *guest OSs* know that they're running in a virtualised enviroment and that, in some case, they'll have to leave control to a *VMM*. So, *guest OSs* are explicitly modified to be virtualizable, changing the interface provided to make it easier to implement.

In particular, *system calls* and "non-virtualizable instructions" are replaced with specific *hypervisor calls* (*hypercalls*). Hence, they won't trap anymore and all the *trap & emulate* process is also removed. Ofcourse, modifications don't affect the *ABI*, so applications can be executed without further changes.

Guest OSs are explicitly deprivileged meaning that they know they're being executed at *ring 1*. This allows the introduction in *guest OSs* kernel of efficient mechanisms that ease the communication with the *hypervisor*:

- *Guesto-to-Hypervisor*: *privileged instructions* are replaced with synchronous paravirtualised equivalent *hypercalls*;
- *Hypervistor-to-Guest*: *hypervisor* can notify certain events asynchronously to the *guest*;

Talking about pros and cons in a paravirtualized enviroment, only modifiable OSs can be used, because it's necessary to access their source code.

Performances are surely higher than those of *Trap & Emulate paradigm* because neither emulation nor translation are required, hence *MVV* implementation is also simpler and faster.

3.2.4 Hardware assisted virtualization

Up to this point, we still have two unsolved issues: complex implementati of *MVV* and necessity to provide full virtualization for most of x86 and x64 systems (most of them weren't still modifiable).

Hardware assisted virtualization aims at providing a solution to those problems by proposing an efficient *Trap & Emulate* approach to virtualization thanks to an additional hardware support.

This is based on the idea of avoiding *sensitive instructions*, either because they can be "promoted" to *privileged* or bacause the *MVV* can dynamically configure which instructions have to be trapped. There are some instruction then, that cause *virtual machines* to exit unconditionally (e.g. `INVD` instraction for CPU internal cache invalidation) and therefore can never be executed in a virtualised non-root enviroment. Finally, all events and some other

instructions can be configured to operate conditionally using *virtual machine* execution control fields.

To do all of this, processors are provided with an additional running mode named *Virtual Machine eXtensions* (*VMX*). When this mode is enabled, CPU will activate two different running modes called *operating levels* that are: *non-root mode* and *root mode*. These modes still works with the usual *privileg ring* structure.

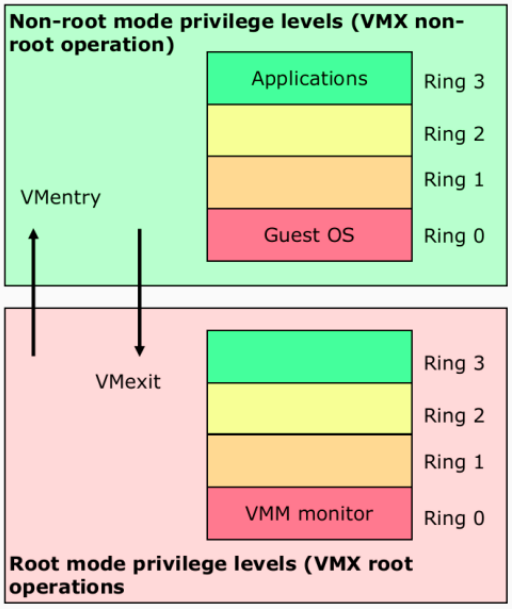


Image 3.3: *Hardware assisted virtualization* modes

The *VMM* runs al *ring 0* in *root-mode*, while *guest OSs* run at *ring 0* in *non-root mode*. Ofcourse applications still run at *ring 3* in *non-root mode*.

VMX instructions If sytem code tries to execute instructions violating isolation of the *VMM* or that must be emulated via software, hardware traps it and switches back to the *VMM*. CPU enters *non-root mode* via the new `VMLAUNCH` and `VMRESUME` instructions, and it returns to *root mode* for a number of reasons, collectively called *VM exits*.

VM exits should return control to the *VMM*, which should complete the emulation of the action that the guest code was trying to execute, then give control back to the *guest* by re-entering *non-root mode*. All the new *virtual machine* instructions are only allowed in *root mode*.

For example, while in *non-root mode*, `INT xx` instruction may cause a swich from *non-root user mode* to *non-root kernel mode*, and `IRET` may return from *non-root kernel mode* back to *non-root user mode*.

So, when a trapping condition is triggered, *VMM* takes control of the execution and emulates the correct behaviour. Transition between *root* and *non-root mode* is realized through:

- *VM entry*: from *VMM* to *guest*;
- *VM exit*: from *guest* to *VMM*;

When this happens, registers and address spaces are swapped in a single atomic operation and, as we would expect, this remains the main source of overhead.

Virtual machine control structure To mantain *virtual machines* state and control information *VMM* uses a particular structure called *Virtual Machine Control Structure* (*VMCS* or *VMCB*). It concretely represents the control panel of the *virtual machine*, storing information about *guest* state, *host* processor and control data (e.g. trapping condition). It also mirrors all registers modifications needed to set a certain configuration in *guest OSs*. *VMCS* introduced dedicated instructions to modify it: `VMWRITE` and `VMREAD`.

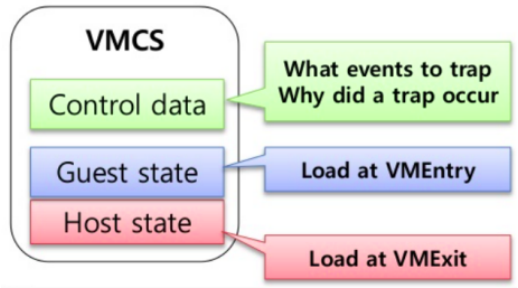


Image 3.4: *Virtual Machine Control Structure*

3.3 Memory virtualization

3.3.1 Memory management in general

Modern operating systems use a *memory paging* technique to access, as contiguous, dispersed locations in the physical memory. In particular, the main memory (RAM) is divided into frames of fixed size. The OS assigns each process one or more pages that are the same size as the frames, so the address space of processes spans across multiple frames which aren't necessarily contiguous, but pages are, so processes can behave as if their address space were unitary.

Therefore, there is a difference between virtual or logical address that referes to pages, and physical addresses that referes to physical memory. Processes use only virtual addresses, so when they need to access memory, logical address have to be translated into physical ones. This translation is done by the *Memory Management Unit* (*MMU*), a unit that resides in the CPU.

Operating systems maintain a page table for each process in which every line holds information about one page. In particular, each row associates the *Logical Page Number* (*LPN*) to the physical one called *PPN*. When a logical address is accessed the *MMU* walks all these page tables to determine the corresponding *PPN*, and thus, determining the frame physical addres too.

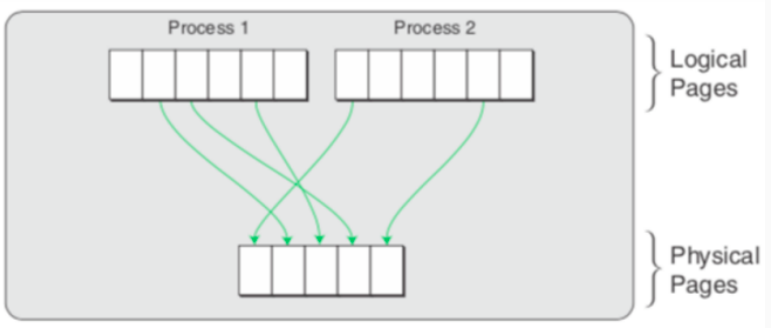


Image 3.5: Memory paging

In the case of big page tables, the *MMU* can use a *Translation Lookaside Buffer* (*TLB*) that works as a cache for recently used page translations. The *TLB* works as a fully associative memory in which *LPNs* are used as keys to get the corresponding *PPNs*.

Note. *TLB* works similarly to a hash map.

There are mainly three reasons for which modern operating systems choose to use *memory paging*:

1. *Simplicity*: every process gets the illusion af a whole address space;
2. *Isolation*: every process address space is strictly separated from others;
3. *Optimization*: it is possible to exploit *swapping* to allow operating systems to handle more pages than what the physical memory alone could hold;

3.3.2 Memory management in virtualised environments

In a virtualised environment, *Guest OSs* don't have direct access to memory, so what they percive as physical addresses are in fact virtual ones. This means that page tables of processes running in the VM associates *Logical Page Numbers* of the processs to *Physical Page Numbers* of the virtual machine, which in turn are associated to *Physical Page Numbers* of the physical machine. For this reason, translating a logical address of a VM's process would require two steps:

1. *Guest Logical Address* $\rightarrow$ *Guest Physical Address*
2. *Guest Physical Address* $\rightarrow$ *Machine Physical Address*

To avoid this, a "shadow page table" is introduced. This table stores and keeps track of the mapping between *Guest Logical Addresses* and *Machine Physical Addresses*. It is invisible from the *guest* point of view because is maintained by the *VMM*, who also exposes it to the *MMU*.

Going into more details, the association between *Physical Page Numbers* and *Machine Page Numbers* (*MPNs*) is maintained by the *VMM* in internal data structures, while the association between *LPNs* and *MPNs* is stored by the *VMM* in the "shadow page table" that is exposed to the *MMU*.

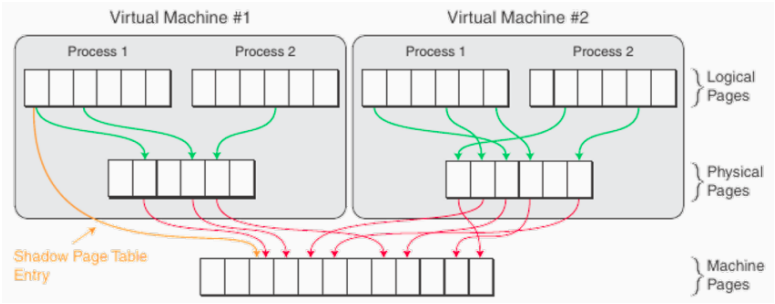


Image 3.6: Memory paging in virtualised environments

It is still possible to cache most recently used translation between *LPN* and *MPN* in a *TLB*.

Shadow page table creation Each *Guest OS* maintains the associations between *LPNs* and *PPNs* as seen before, but when it tries to access a physical address, since it isn't running directly on the hardware, the request is trapped by the *VMM*. Then, the *VMM*, who already knows what *MPN* is bound to that *PPN*, saves the original *LPN* is the shadow table bounding it to the correct *MPN*.

Problems with shadow page table Ofcource the *VMM* is in charge of keeping the shadow page table synchronized with the *Guest OS* page tables. So, an extra overhead is introduced, and it becomes a problem if some applications force *Guest OS* to update them frequently (e.g. some apps might couse many page faults).

3.3.3 Hardware assisted memory virtualization

To avoid that extra overhead, hardware manufacturer implemented a type of hardware that allows the mapping between *LPNs* and *PPNs*, maintained by *Guest OS*, to coexist together with the mapping between *PPNs* and *MPNs*, created by the *VMM*, in the same page table.

In particular, the translation to *MPNs* is put in an additional nested level of page tables. Both the traditional and the nested tables are exposed to the hardware, so that, when a logical address is accessed, the hardware walks the guest page tables as in the case of native execution (no virtualization), but for every *PPN* accessed during the process, the hardware also walks the nested page tables to determine the corresponding *MPN*.

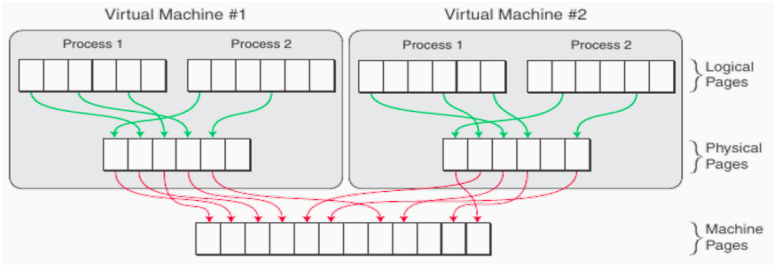


Image 3.7: Hardware assisted memory virtualization

This approch removes the need for the *VMM* to keep synchronized additional tables, thus removing the previously discussed overhead. However, since the hardware has to waltk through two tables to translate every address, the cost of every translation is increased. For this reason, *TLB* becomes critical to guarantee good performance and, for memory intensive tasks, having larger pages might increase the *TLB* "hit ratio".

Tagged TLBs An additional optimization that can be implemented to increase *TLB* hit ratio is represented by *tagged TLBs*. Adding a tag means adding to each *TLB* line a Virtual Processor ID, that is an identifier for each virtual processor. This prevents wrong access to other virtual processors cache lines, thus allowing multiple virtual processors to coexist on the *TLB* at the same time.

Previously in fact, *TLB* needed to be flushed on each *VM exit* and *VM entry* beacuse virtual machines addresses, both *LPN* and *PPN*, aren't globally unique and keeping them in the *TLB* would have created conflicts and accesses to memory areas of other VMs.

3.4 I/O virtualization

There are various techniques to implement I/O virtualization: *device emulation*, *paravirtualization* and *direct assignment*. Unlike CPU and memory, I/O devices might be assigned to just one or some VMs.

3.4.1 Device emulation

With *device emulation* the *VMM* proposes to the *Guest OS* an emulated device which implements in software an hardware specification. *Guest OS* uses that device without knowing that

it is being emulated and to do so, it uses the same drivers used with an equivalent physical device.

This is a simple approch that doesn't require *Guest OSs* to install dedicated drivers, and a single physical device could be multiplexed into multiple emulated devices. Ofcourse, the *VMM* has to remap the communication with the physical device. Then, I/O operations are generally slower than the physical ones and with higher latency, especially in case of devices with high I/O (e.g. NIC, disk). Also, since CPU has to emulate each request, its workload might increase substantially.

3.4.2 Paravirtualized devices

Unlike *CPU paravirtualization* that required kernel modifications, to paravirtualize I/O we just need to write new drivers that can than be added as external modules to the OS.

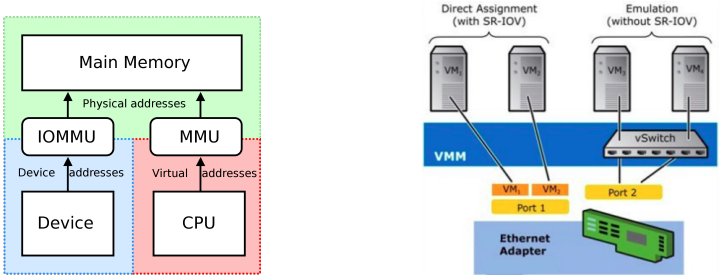
Paravirtualized drivers are a convenient solution that also allows further optimization such as "memory balooning". When creating a virtual machine, the *VMM* defines its memory size and allocates it statically. To obtain a dynamic and more efficient use of memory, the *VMM* can exploit memory balooning paravirtualized drivers installed by *Guest OSs*. Those drivers proive the *VMM* with information about current memory occupation of the guests, allowing it to change the amont of memory allocated to those VMs and providing it to others.

3.4.3 Direct assignment

With *direct assignment* a device is exclusively assigned to one VM that can directly communicate with it without needing any driver apart from the traditional ones of the device. The device is totally handled by the *Guest OS*; hence it can be multiplexed over several virtual machines.

Despite seeming very simple, this approch is very complex indeed, because it raises critical issues on memory usage. Direct memory access (DMA) has to be performed on the physical address space of the *Guest OS*, but the device doesn't know the mapping between *Guest* and *Host* physical addresses. This could potentially lead to memory corruption and to avoid it the *VMM* has to intercept the I/O operations and perform the correct translation. The problem is that this is slow and can introduce a significant overhead in I/O operations.

Hardware assisted direct assignment As seen before, hardware manufacturer can implement technologies to ease the virtualization process.



(a) *IOMMU* extension

(b) *SR-IOV* implementation model

Image 3.8: *Hardware assisted direct assignment* possibilities

One solution to the memory problem discussed above is the introduction of an *Input Output Memory Management Unit* (*IOMMU*), an extension that can boost and make direct access to

memory easier to be implemented in *VMMs*. Like to *MMU*, the *IOMMU* remaps the addresses accessed by the hardware according to the same table used to map *PPNs* to *MPNs*, allowing direct memory access cycles to safely access the correct memory locations.

As for the networks card instead, the *PCI-e* standard defines *Single Root Input/Output Virtualization* (*SR-IOV*) as a mechanism to allow several directly assigned devices to be shared among VMs. *SR-IOV* defines the possibility for the devices to present several virtual devices, "virtual functions" to the OS. The *VMM* will directly assign each virtual function to each VM and the hardware will handle multiplexing by itself.

3.5 Hypervisors architectures

Hypervisors can be based upon two architectures that pursue two distinc objectives: performance the first and easiness of deployment and utilization the second. Ofcourse hybrid implementations exist.

3.5.1 Type 1 architecture

The *hypervisor* runs directly on bare metal, so there isn't any extra layer between the hardware and it. Normally, it's able to provide the best performace. However, the *hypervisor* needs to be implemented as a stripped-off OS with basic functionalities and, thus, there might be problems with drivers. As we already said, *hypervisors* need to have basic functionatilties to be less prone to bugs and attacks.

Examples Microsoft Hyper-V

3.5.2 Type 2 architecture

The *hypervisor* runs on top of an OS as a privileged process, so it's easier to install but less performing.

Examples VirtualBox

Note. Systems with dual boot where a normal OS resides together with a *type 1 hypervisor* are an example of how an hybrid approch works.

3.5.3 Hybrid architecture

Hybrid hypervisors are implemented as a component of the OS kernel. So, the *Host OS* is itself the *hypervisor*, but also works as a normal OS. This makes this kind of *hypervisors* easy to install and deploy bacause drivers and support comes from the mainstream OS. Performance can also be very good.

Examples KVM

3.6 OS-level virtualization

Going back to *full virtualization* we can summarize its pros and cons as it follow:

Advantages

- Compatible with existing applications;
- Supports different OSs;
- Each VM can have its own execution environment;
- The isolation backed by hardware is execellent;

Disadvantages

- Running each *Guest OS* requires additional overhead;
- It's necessary to configure and keep updated each instance of *Guest OS*;
- OS booting time (e.g. seconds or more) might not be acceptable;

Before cloud took place, datacenters stored lots of servers which were meant to run different OSs to meet different requirements (e.g. desktop environments for real users, support for specific peripherals). However, with the spreading of cloud computing, hardware became a commodity and interaction with users is provided by web applications instead of desktop environment. Hence, we could achive great operational efficiency if we reduced the number of OSs to just one: Linux.

Note. From now on, we will only discuss mechanism and approches used by Linux-based operating systems.

3.6.1 Lightweigth virtualization

In this context the idea of *lightweight virtualization* was born. It aims to the creation of a system in which all the advantages of *full virtualization* are guaranteed, but resource consumption is much less concerning.

Lightweigth virtualization is therefore appropriate when there is no need for a classical VM or when its overhead is unacceptable. Also, when we'd like to have an isolated environment that is quick to deploy, migrate and dispose with little or no overhead, or when we want to scale both vertically (i.e. many isolated environments on the same machine) and horizontally (i.e. deploy the same environment on many machines), *lightweight virtualization* can be a good solution.

Going deeper into *lightweigth virtualization* characteristics, with it, we use *OS-level* or *application level virtualization* instead of *full virtualization*. In particular, with *OS-level virtualization*, the *hypervisor* is the Linux kernel itself.

As already mentioned, classical VMs are replaced by isolated environments (i.e. virtual private servers, jails, containers) and each one of them features a given extent of resources management and isolation that usually is less than what can be guaranteed by classical VMs. Finally, applications can be executed inside these environments.

A good *lightwiegth virtualization* implementation must provide a fine-grained control of resources of the physical machine, allowing sysadmins to partition and control resources among different isoltated environments. Another requirement is on security and isolation, meaning that

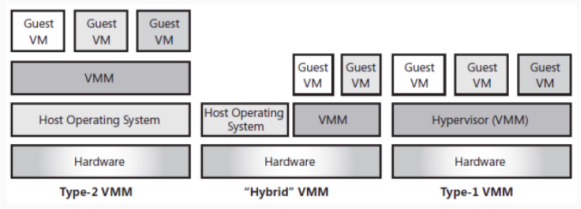


Image 3.9: *Hypervisor* architectures in comparison

each environment should be assigned to one app or user, and it should prevent a misbehaviour in one environment from affecting others. Finally, it should be possible to manage an entire datacenter as a unique entity, such as with cloud toolkits; even better, the capability to integrate *lightweigth virtualization* with a cloud toolkit in order to have the flexibility to deploy VMs, containers and such upon requests, should be provided.

Why is process isolation so important? Community recognized the need to implement strong process isolation in Linux kernel because servers running multiple services want to be sure that possible intrusions on some services don't affect others. Also, it must be safe to run arbitrary or unknown software on a server (e.g. students code, hakaton, testing environment).

How can all of this be done without adopting techniques such as hardware virtualization that generates too much overhead?

In theory many possibilities exists, but practically only a few answers the questions: Linux containers (LXC) and LXC-based software. Other technologies used are Linux *cgroups* and *namespaces*, that were created to strengthen processes isolation without thinking to virtualization, but can be leveraged to create a form of *lightweigth virtualization* with minimun overhead.

3.6.2 Linux cgroups