Handouts of fog and cloud computing

Leonardo De Faveri

A.A. 2021/2022

Table of contents

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 conforms 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 make 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 unavailable 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 that act indipendently. Groups of nodes refere to an aggregation node that handle them and more aggregation nodes are then connected to a central point that provides, among others, an interface for users.

Note. A *fog node* is an active component that performs some operations and not just 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

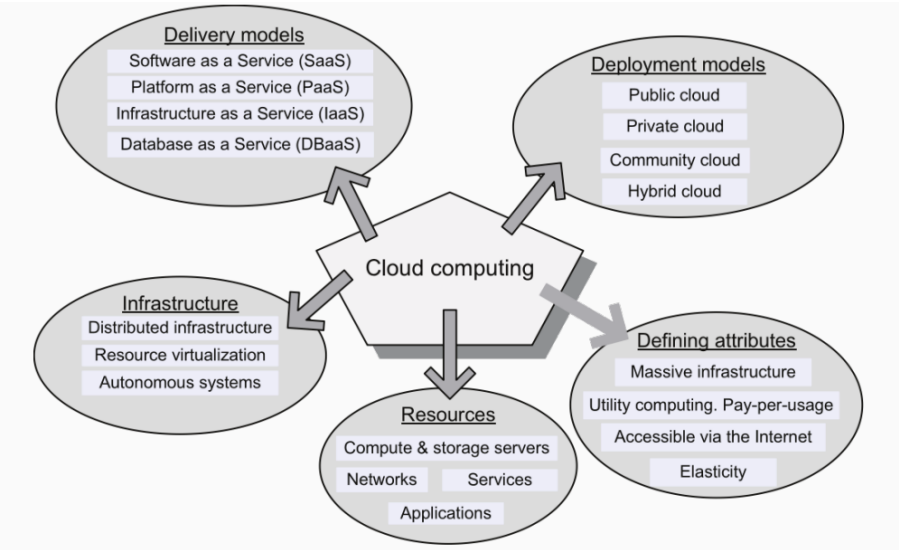


Image 2.1: Five key aspects of *cloud computing*

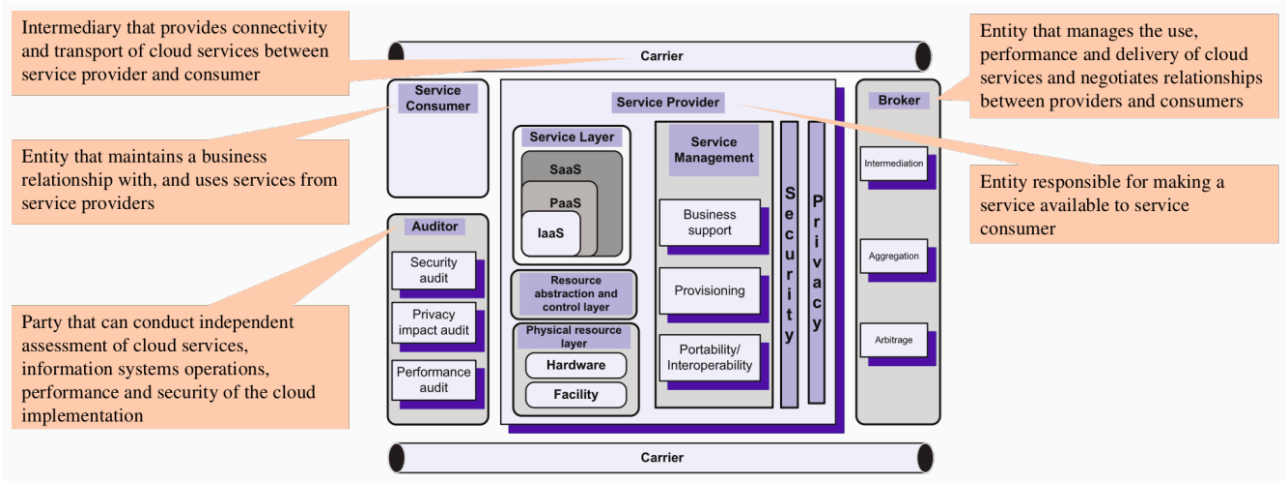


Image 2.2: *NIST* reference model for *cloud computing*

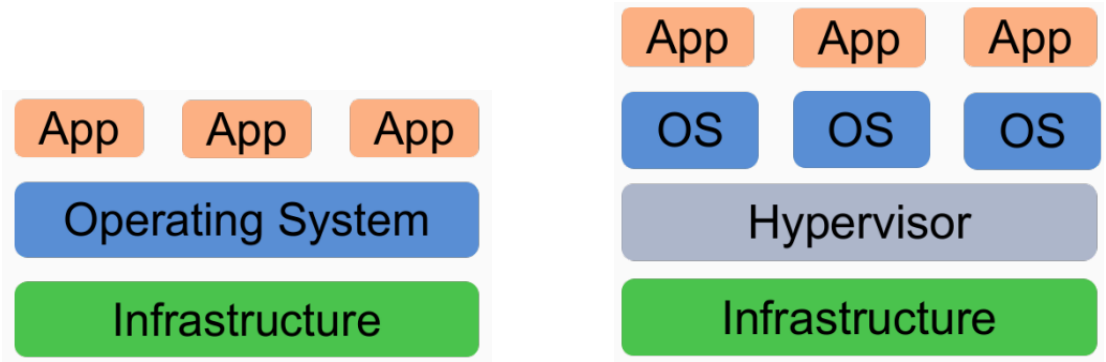
Note. A carrier is someone who provides access to a cloud service, such as Telecom, while a broker is a subject that handles the delivery of cloud services to users, such as a portal to the cloud (e.g. Booking.com).

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 systems, 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.

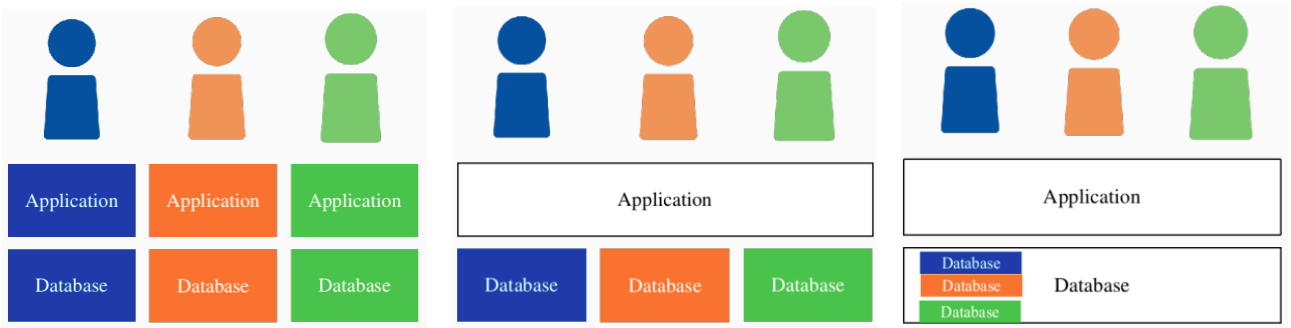
With single-tenancy each user has its own software instance.

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 a software instance, 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 problems resulting from *underprovisioning* and *overprovisioning*.

Overprovisioning happens when current workload is using much less resources than the ones that have been allocated, thus resulting in a waste. Simmetrically, *underprovisioning* means that the available resources are insufficient to serve 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. The main types of model are three:

- *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 allowed 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 control 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 include: 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. They're able to deploy arbitrary software including operating systems, but there's still an underlying instrastructure that can't be accessed.

Examples Amazon EC2

Note. Everything can be deployed as 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 cloud infrastructures may be accessed and by whom and who is responsible for their maintainance. In particular, there are four types of environment.

2.3.1 Public cloud

- *Consumer*: general users or large industrial groups;
- *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 such as their mission, policies and security considerations;
- *Service provider*: either the organizations or a third party;
- *Resource location*: either on-premises or 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 enable data and applications portability.

For example an organization might use a *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 application requiring version 1.0 of some library, created conflicts with another application requiring a different version for the same library;
- *Temporal isolation for performance predictability*: it could happen that one application used a lot of resources causing performance degradation for another application;
- *Strong spacial isolation for security and reliability*: if some application 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 minimising costs and energy consumption;
- *Flexibility and agility*: system administrators have complete control over *virtual machines* execution, and they can pause and restart them. Moreover, they 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 for each by each server;
- *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 desktop 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 minimise the interactions among 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 kernel, libraries and applications. Examples of interfaces between software layers are ISA, ABI and API.

Definition 8 - Virtual machine (VM).

A virtual machine is a software emulation of a physical machine that executes both OS and applications as if they were being executed on a physical machine.

When taking about *virtual machines* 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 environment ;

Definition 9 - Hypervisor.

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

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

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

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

The *hypervisor* is often implemented as a Linux-based stripped-off OS (i.e. an OS with minimum functionalities) to make it more efficient and more easily securable. The *hypervisor* exports 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 *guest OSs* with a "virtual hardware" whose characteristics are 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 environment, CPU, memory and I/O need to be virtualised correctly.

3.2 CPU virtualization

VMMs assign one or more CPU cores to each *VM* so that they can run their OSs. 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 *VMMs*, they must satisfy three characteristics:

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

3.2.1 Some definitions

System based on x86 or x64 architectures are usually modeled 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.

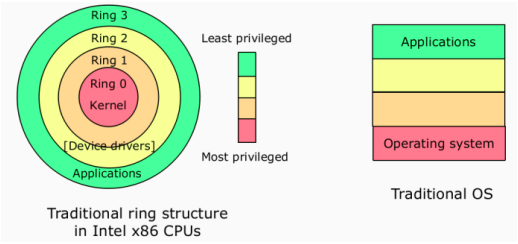


Image 3.1: *Privilege-ring model*

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*. The possible models are two:

- *0/1/3*: the *VMM* runs at *ring 0*, *guest OSs* at *ring 1* and applications at *ring 3*. Since, in x86 architetures, some privileges with respect to memory accesses are granted to *ring 0-2*, *guest OSs* 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 the currect CPU privileg level is stored.

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 that *privileged instruction* in kernel mode.

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

1. *Exceptions*: invoked when an unexpected error or system mulfunction occurs (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 some OS instructions.

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

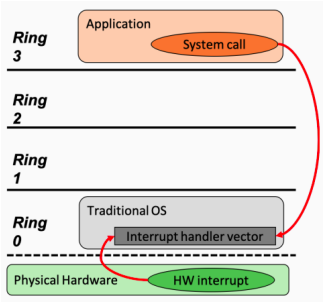


Image 3.2: *Trap handling* in traditional OSs

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 routine 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 routine 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 run in a *virtual machine*, so they need to be modified;
3. *Hardware assisted virtualization*: the *hypervisor* exploits some functionalities provided by modern CPU chips;

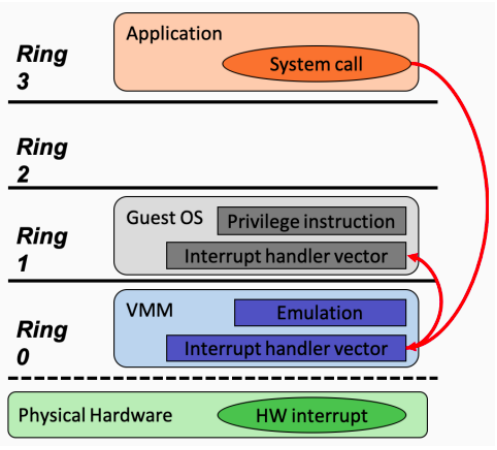
3.2.2 Trap & emulate paradigm

This paradigm allows *full virtualization* and provides that *guest OSs* run in an unprivileged environment, 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 (of course, 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 *VMM* handles it by modifying the state of the *virtual machine*.

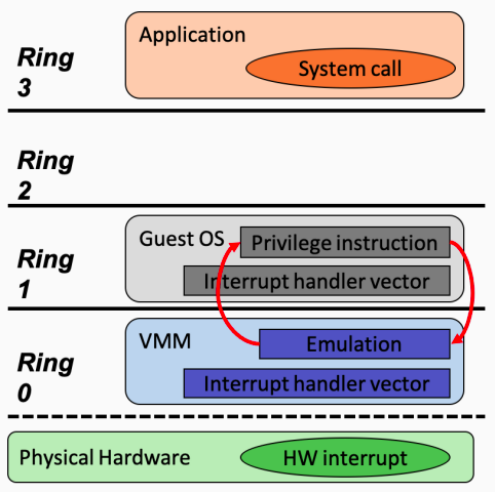
System call handling

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



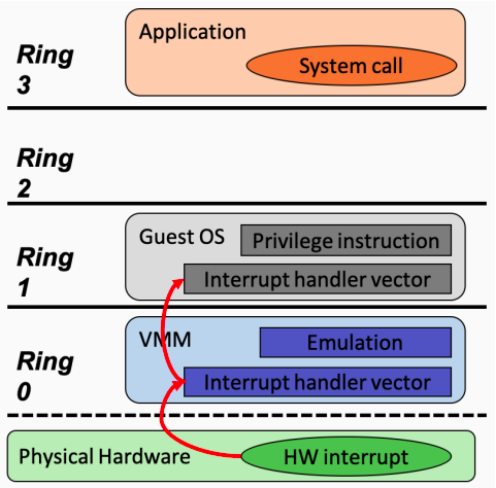
Privileged instruction handling

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*. As we've just seen, these actions are time-consuming.

This process isn't necessary for all architetures, but unfortunatly, it is in x86 and x64 architectures, which are the most common. Moreover, these architectures present 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 So, we have some *sensitive instructions* that don't *trap* and, consequently, the *VMM* cannot emulate their correcct behavior during execution.

To address this problem, we can change virtualization paradigm, or introduce some code into the *VMM* that parses the instruction stream to dynamically detect all *sensitive instructions*. Then, we can both use interpretation and binary translation. Interpretation is 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.

One pro 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 code dynamically 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 environment 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 them 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 can be removed. Of course, 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 to introduce in *guest OS* kernels 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 environment, only modifiable OSs can be used, because it's necessary to access their source code.

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

3.2.4 Hardware assisted virtualization

Up to this point, we still have two unsolved issues: complex implementation of *VMMs* 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 because the *VMM* can dynamically configure which instructions have to be trapped. There are some instructions then, that cause *virtual machines* to exit unconditionally (e.g. `INVD` instruction for CPU internal cache invalidation) and therefore can never be executed in a virtualised non-root environment. 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.

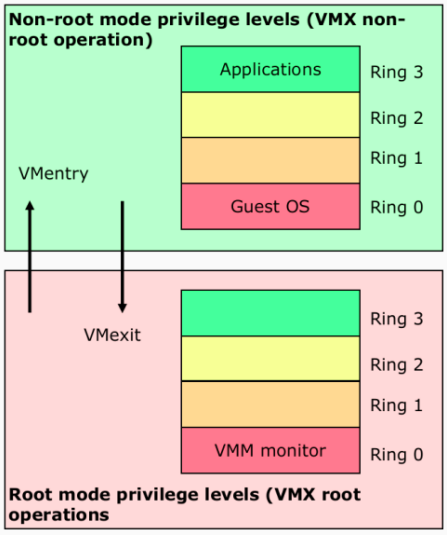


Image 3.3: *Hardware assisted virtualization* modes

The *VMM* runs al *ring 0* in *root-mode*, while *guest OSs* run at *ring 0* in *non-root mode*. Of course 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*, that 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 register modifications needed to set a certain configuration in *guest OSs*. *VMCS* introduced dedicated instructions to modify it: `VMWRITE` and `VMREAD`.

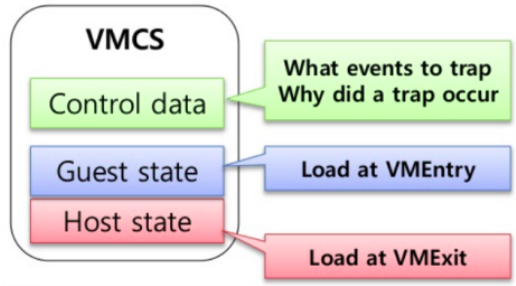


Image 3.4: *Virtual Machine Control Structure*

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 only use virtual addresses, so when they need to access memory, those virtual addresses 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*) of a page to the corresponding 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.

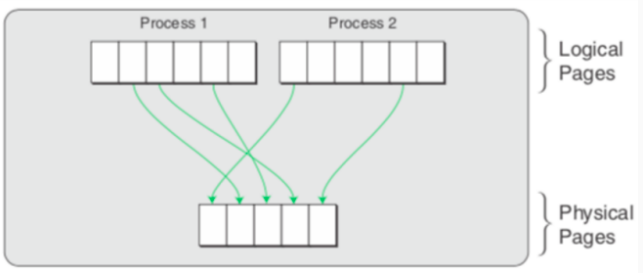


Image 3.5: Memory paging

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 *VMs* associate *Logical Page Numbers* of each 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 it's 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*.

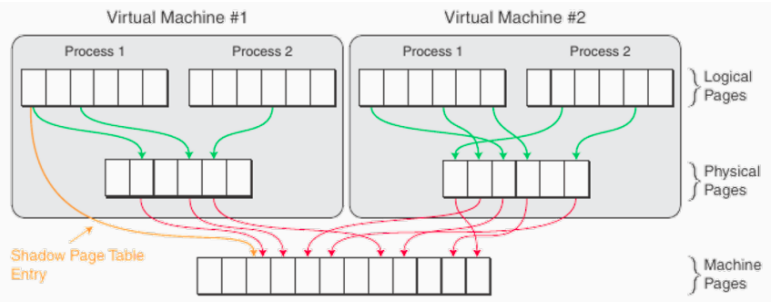


Image 3.6: Memory paging in virtualised environments

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* in the shadow table bounding it to the correct *MPN*.

Problems with shadow page table Of course 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 a *Guest OS* to update them frequently (e.g. some applications might couse many page faults).

3.3.3 Hardware assisted memory virtualization

To avoid that extra overhead, hardware manufacturers 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 *MMU*, so that, when a logical address is accessed, the *MMU* walks guest page tables as in the case of native execution (no virtualization), but for every *PPN* accessed during this process, the *MMU* will also walk nested page tables to determine the corresponding *MPN*.

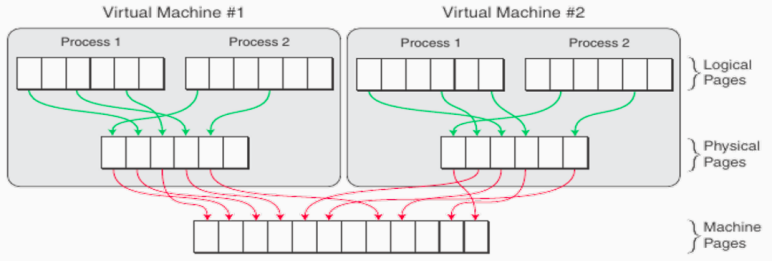


Image 3.7: Hardware assisted memory virtualization

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 *LPNs* and *PPNs*, 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 *Guest OSs* an emulated device for which it implements in software an hardware specification. *Guest OSs* use that device without knowing that it is being emulated and in fact, they use 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. Of course, 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.

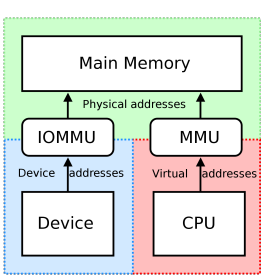
Paravirtualized drivers are a convenient solution that also allows further optimization such as "memory balooning". When creating a virtual machine, the *VMM* defines its memory size and allocates it statically. To obtain a dynamic and more efficient use of memory, the *VMM* can exploit memory balooning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

3.4.3 Direct assignment

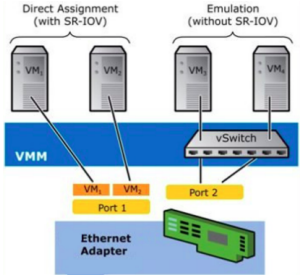
With *direct assignment* a device is exclusively assigned to one *VM* that can directly communicate with it without needing any driver apart from the traditional ones of the device. The device is totally handled by that *Guest OS*; hence it can't be multiplexed over several *virtual machines*.

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 I/O operations and perform the correct translation. The problem is that this is slow and can introduce a significant overhead in for those 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 *MMU*, *IOMMU* remaps addresses accessed by the hardware according to the same tables used to map *PPNs* to *MPNs*, allowing direct memory access cycles to safely access correct memory locations.

As for network cards 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 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 distinct objectives: performance the first and easiness of deployment and utilization the second. Of course 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 because drivers and support come from the mainstream OS. Performance can also be very good.

Examples KVM

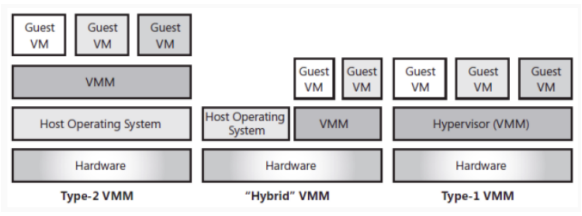


Image 3.9: *Hypervisor* architectures in comparison

3.6 OS-level virtualization

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

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 started to be 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

This is the context in which 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 to 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 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 system admins to partition and control resources among different isoltated environments. Another requirement is on security and isolation, meaning that each environment should be assigned to one application or user, and it should prevent misbehaviours 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 answer the question: 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

Linux cgroups are a kernel feature created to limit, account and isolate or deny resources usage to processes or groups of them. They consist of two components: kernel support and user-space tools that handles the kernel control mechanism.

For instance, commands such as `nice` and `cpulimit` can be used to managed cpu consumption of single processes, while *cgroups* allow for a simpler control over a group of processes.

So, *cgroups* features are:

- *Resouces limiting*: groups can be set to not exceed a configured memory limit, which also includes file system cache;
- *Prioritization*: some groups may get a larger share of CPU utilization, disk I/O throughput or network bandwidth;
- *Accounting*: resources usage of each group can be measured (useful for billing pourposes);
- *Control*: it's possible to freeze and restart groups of processes, and control their checkpoints too;

3.6.3 Linux namespaces

Linux namespaces are another kernel feature, highly related to *cgroups*, although not being part of them. They're ment to prevent groups of processes to "see" some kind of resources of other groups. They do so, by creating distinct virtual environments for a specific class of resources (e.g. different file systems, networking, ...), and for each of them the kernel has to create different and indipendent instances of the data structures it uses to handle those resources. Each object (e.g. process) can be assigned to a *namespace* and can access other objects belonging to the same *namespace*. When the access to a given data structure is requested, the kernel uses the ID of the *namespace* to retrieve data from the proper structure. Up to now, the Linux kernel offers seven type of *namespaces*:

Namespace	**Constant**	**Isolates**
IPC	`CLONE_NEWIPC`	System V IPC, POSIX message queues
Network	`CLONE_NEWNET`	Network devices, stacks, ports, ...
Mount	`CLONE_NEWNS`	Mount points
PID	`CLONE_NEWPID`	Process IDs
User	`CLONE_NEWUSER`	Users and groups IDs

UTS	`CLONE_NEWUTS`	Hostname and NIS domain name
Cgroups	`CLONE_NEWCGROUP`	Control groups

Table 3.1: *Linux namespaces*

PID namespace In Linux, every process originates from the *init* process as it's child (or grandson, or great-grandson, ...). Hence, every processe is placed in a single tree of processes. A process with enough privileges might inspect and kill others, and to avoid that, *PID namespaces* allow creating nested process trees which represent a separated set of resources. Processes in a subtree don't know the existance of other parallel trees, so they cannot inspect nor kill processes in those trees.

So, *PID namespaces* allow processes to create new trees, with their own PID 1 process. The process that creates the subtree remains in the parent tree and knows about its child. In fact, processes in the *parent namespace* have a complete view of processes in the *child namespace*, as if they were any other processes in the parent tree. However, the first child becomes the root of its own process tree, and it doesn't know anything about the originating one.

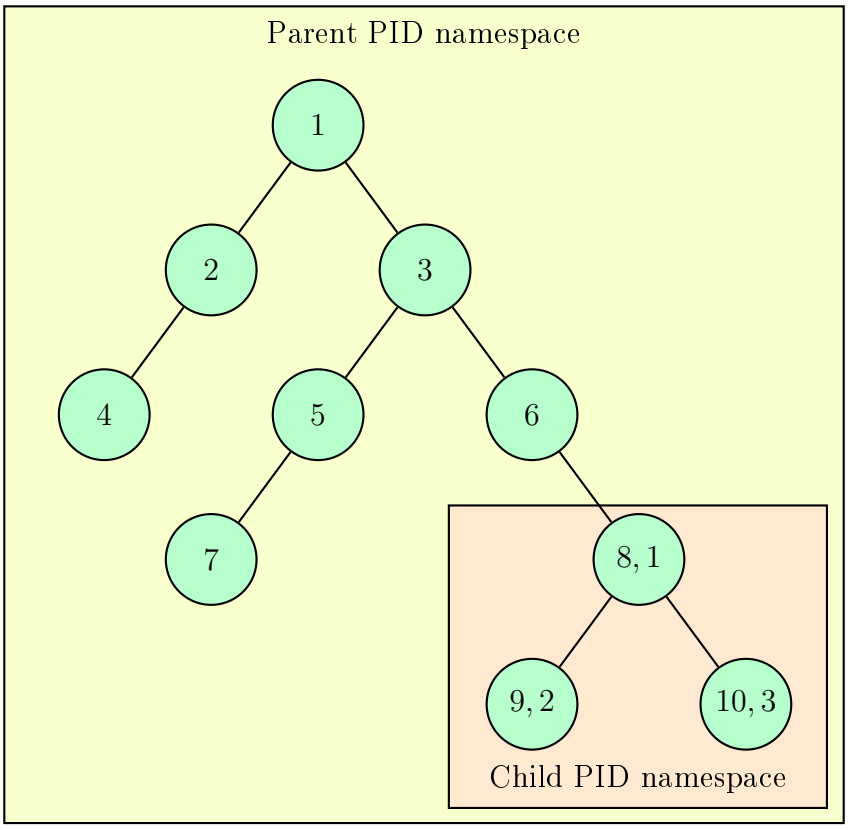


Image 3.10: *PID namespace* example

Note. The image above shows us that each process has one PID for each process tree it belongs to.

Network namespace *Network namespaces* allow processes to perceive a completely different network setup (i.e. loopback interface, general interfaces, routing tables, firewall rules, ...). Once a *network namespace* is created, we should create an additional "virtual" network interface that spans multiple *namespaces*. Virtual interfaces, often called `veth`, are network abstractions of wires with two ends. Each end is "connected" to a *namespace*, thus allowing traffic to move

between them. Finally, routing and bridging protocols in *root namespace* can allow traffic to reach its destination (inside or outiside the machine).

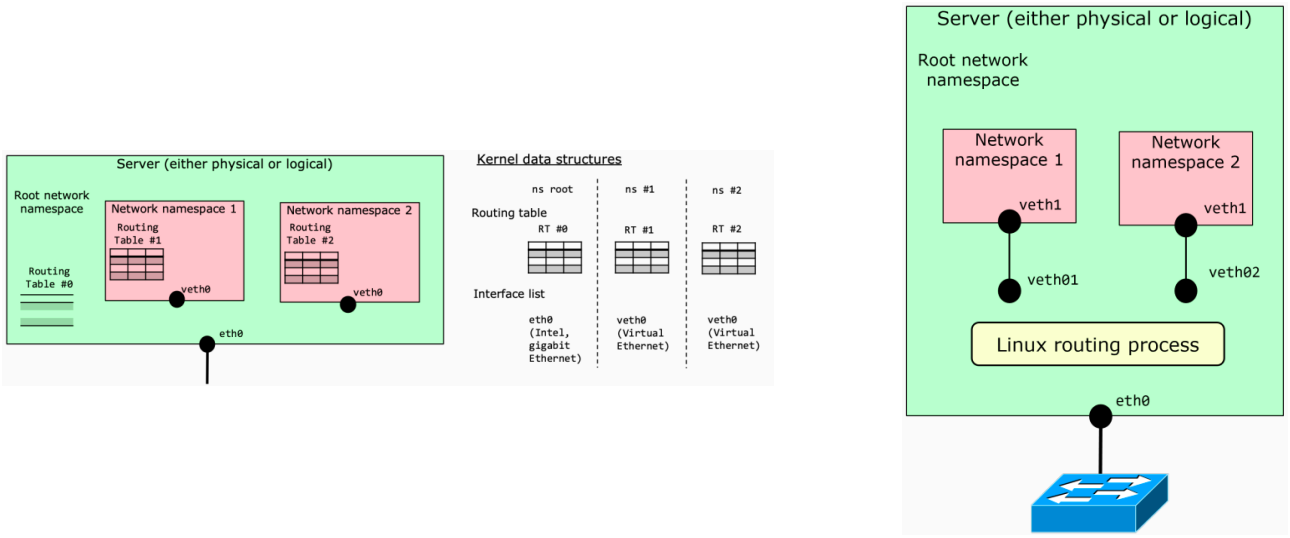


Image 3.11: *Network namespaces* examples

Other namespaces in short

- *IPC*: creates private inter-process communication resources for isolated processes;
- *Mount*: enables the creation of a completely new file system with the desired structure (it's similr to chroot);
- *UTS*: provides isolation of two system identifiers: hostname and NIS domain name;
- *User*: allows a process to have root privileges within the *namespace* without giving the very same access outside it;
- *Cgroups*: provides a mechaism to virtualise the view of the `/proc/$PID/cgroup` file and *cgroup* mounts;

3.6.4 Linux containers

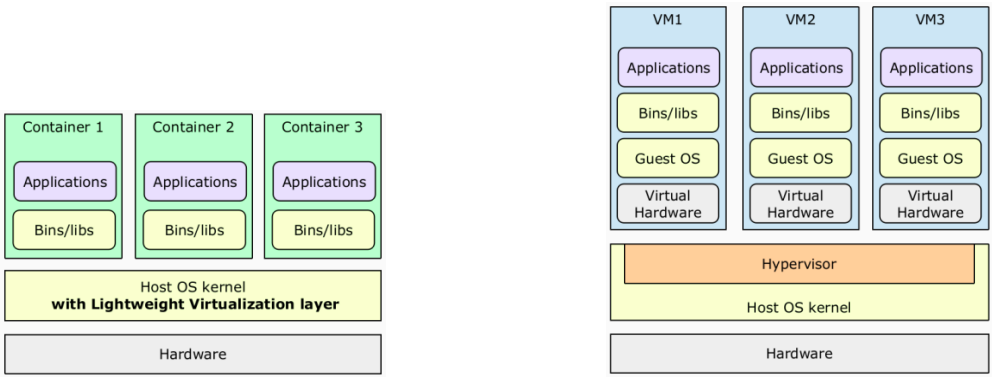
Linux cgroups and *namespaces* show some limitations beyond process isolation. In fact, they provide a way to accomplish virtualization on a single server, but can't be used on an entire datacenter. Then, they're flexible, but really hard to use, because they require a lot of commands to set up a simple isolated environment. Also, they cannot guarantee portability because there isn't an easy way to package and move them to another server; hence, when it comes to portability, *VMs* are still preferable. So, *cgroups* and *namespaces* need to be extended and made easier to use; *Linux containers* go in this direction.

Containers provide a *lightweigth virtualization* that allows processes and resources isolation without the complexity of *full virtualization*. They are an *OS-level virtualization* method for running isolated Linux systems, called containers, on a single control host and over the same shared kernel.

Containers can group processes together inside an isolated environment, to which different resources can be assigned and, as we said earlier, they share the same kernel as the host. From

the inside, each *container* looks like an indipendent machine, while from the outside they're seen as normal processes.

On the other hand, they don't emulate software and can't use a different kernel. Also, security isn't guaranteed by the model itself, although an appropriate level of security can still be achieved.



(a) *Containers architecture* (b) *Hypervisors architecture*

Image 3.12: *Containers VS Hypervisors architecture*

Compared to *VMs*, *containers* are both faster and ligher in terms of resource usage. Hence, a larger number of *containers* can cohexist on a single host. However, *VMs* still provide better isolation (e.g. kernel exploits don't affect other *VMs*) and security, as mentioned before.

Note. *Linux containers* can both be seen as infrastructure primitives (aka "lightwieigth *VMs*") and an application management and configuration system.

LXC - Linux Containers in practical What we've talked about until now are the general characteristics of *containers* in a Linux system. *LXCs* (*LinuX Containers*) are the actual implementation of that technology. They're implemented using *cgroups*, *namespaces* and other technologies such as Apparmor and SELinux profiles that allow them to obtain bettere security performance.

So, *LXCs* make it easier to create and manage isolated environments, but they still lack in resource isolation, because the resource quota of a container might be affected by others, and both checkpointing and migration, because those features are not provided by the Linux kernel and third party tools aren't 100% effective, yet. Another important limititantion is on portability, because they cannot be transfered across servers without recreating them.

3.6.5 Docker

So far, we've been able to obtain some form of *lightweigth virtualization*, but we still struggle when it comes to packaging and deployment of environments and applications in them. *Docker* focuses on applications, simplifying their deployment and execution by creating a lightweigth, portable and self-contained "package" that runs everywhere.

Docker aims to obtain a clean separation of environments, an easy way to manage resources, applications (e.g. a unified "run" command for all apps) and networking (e.g. Docker bridge, transparent NAT, ...). Therefore, *Docker* is not a virtualization engine because it leverages existing primitives such as *cgroups* and *namespaces*.

Summarizing, we can say that *Docker* is ment to make applications run consistently in different machines regardless of kernel version, operating system distribution (it needs to be a Linux distro, tho) and network settings. Also, it makes separation of concerns easy: developers

can focus on what there is inside the container (e.g. libraries, binaries, "manifest", ...), devOPs can focus on the outside (logging, monitoring, remote access, ...).

Docker VS LXC *Docker* is optimized for the deployment of applications, as opposed to *LXCs* that focus more on containers as lightweigth machines. Then, *Dockers* containers are portable across machines, while *LXCs* need to be rebuilt.

3.6.6 Docker usage in general

Docker distinguishes between images and containers. The first are immutable templates for containers that can be pushed and pulled from a registry. The seconds, on the other hand, are instances of an image that can be started, stopped, etc. Thus, they maintain their state within their file system.

As mentioned, images can be stored and retrieved from a registry that can either be public (e.g. Docker hub) or private. Then, given an image, *Docker* offers a unified interface for creating a container via the `docker run` command.

Docker network management By default, all containers are connected to a `docker0` bridge and *Docker* automatically implements a private IP network with its address space and the appropriate routing table and iprules for internet connectivity.

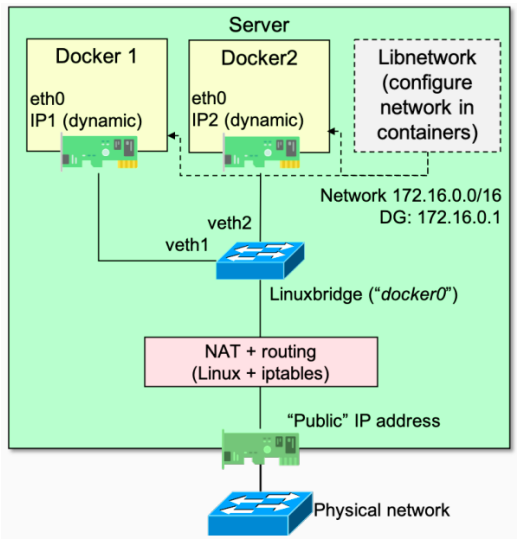


Image 3.13: *Docker* network setup

This way, *Docker* provides automatically inbound connection, meaning that containers running on the same *Docker host* can communicate with each other without additional configurations. To communicate with the *Docker host* itself or with other hosts, one or more container ports have to be exposed.

It's interesting to point out that *Docker* doesn't use a DHCP server to assign IP addresses. Instead, network operations are carried out by a `libnetwork` component that implements the *Container Network Model* (*CNM*). This component decides what IP address to use within the container, and configures the inside endpoint of the virtual interface to which that address has been assigned.

Docker file system management Since containers must replicate the entire file system required for them to run, *Docker* must assure that no host file is accessed from within the

container and that each container holds all that files required to run. Hence, *Docker* must grant isolation and portability.

Note. Whereas containers need to hold all their files, each container might weigth hundreds of megabytes.

To comply with the requirements above, *Docker* uses a *union file system*. A *union file system* works on top of other file systems, and it gives a single, coherent and unified view to file and directories of separate file systems. In other words, it mounts multiple directories to a single root, in fact, it is more of a mounting system rather than a file system. There are different types of union file systems, *Docker* in particular uses *OverlayFS*.

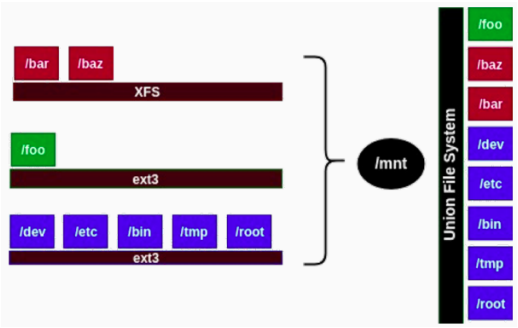
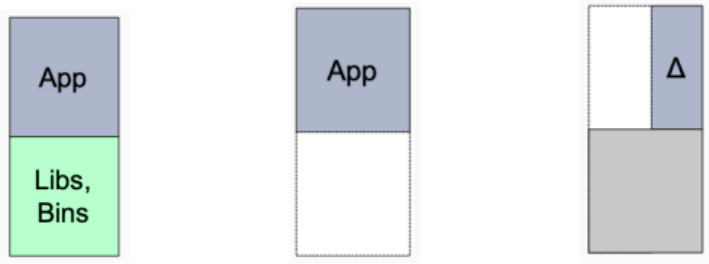


Image 3.14: Behaviour of a *union file system*

A *union file system* can be seen as a logical merge of multiple layers in which only the upper layer is writable; all the lower layers are read only. When a process wants to read from a file, it starts searching it from the upper layer, than moves to the lower ones until it isn't found. When accessing a file in write mode instead, the file system copies that file for that process (i.e. container) only. All the other processes continue to use the original one. When it comes to file (or directory) removal, if it resides in the upper layer, it is removed directly, and it's simulated through a *whiteout* file in lower layers. This kind of file exists only in `union` directory, whitout physically appearing in neither the upper not lower layers.

Such a behaviour of *union file systems* allows multiple containers to share the same files and directories until they need to modify them. This reduces the disk fooprint of containers on *Docker host* and their loading time.



(a) Original app (b) Copy of the app (c) Modified app

Image 3.15: Files sharing into a *union file system*

Docker images are created exploting this behaviour. In fact, any image results from adding information on top of another image.

Automatic build *Docker* allows developers to automatically create a container starting from its composing elements. The recipe used to create the container is stored in a file called

`Dockerfile` in which developers can write all the instructions that have to be executed to set up the container. Using a tool called `docker-compose`, multi-container applications can be configured and deployed using the single command `docker-compose up`. Finally, various *Docker host* can be turned into a single virtual host through *container orchestration tools* such as *Docker swarm* and *Kubernetes*.

Chapter Nr.4

Cloud networking

4.1 Data centers networks

Unquestionably, networking is the core of *cloud computing*, in fact, it has been made feasible by the continuous evolution of internet connectivity. Historically, networks have been classified into three layers:

- *Tier 1*: core part of network infrastructure that allows all other networks on the internet to communicate each other;
- *Tier 2*: portions of *tier 1* infrastructure which are bought by internet service providers that can then sold access to customers;
- *Tier 3*: part of the network infrastructure that allows final users to gain access to other networks;

Note. *Tier 2* networks communicate each other as peers through special nodes called Internet Exchange Points (IXP).

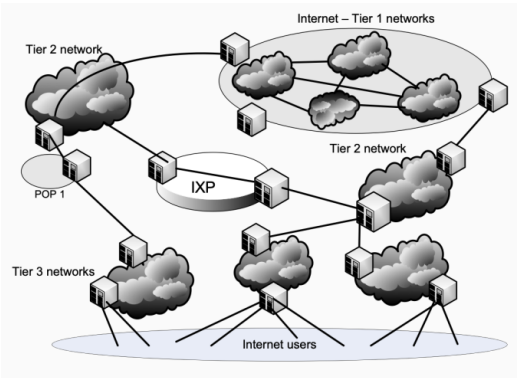
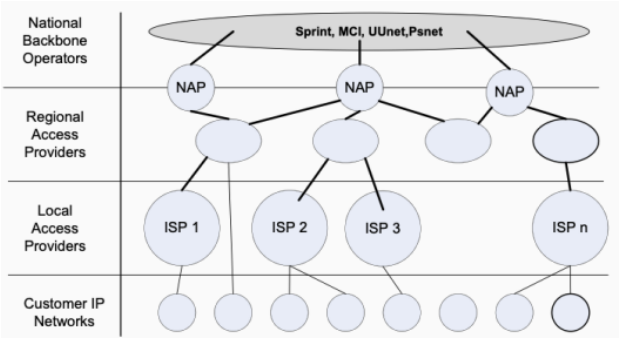
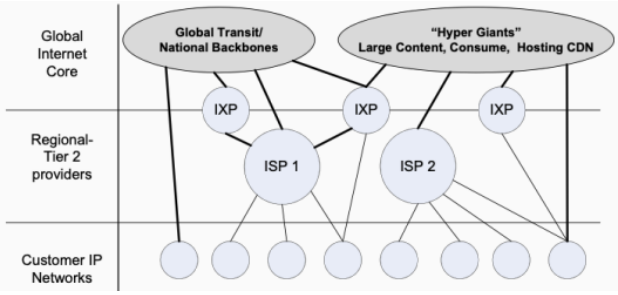


Image 4.1: Internet infrastructure

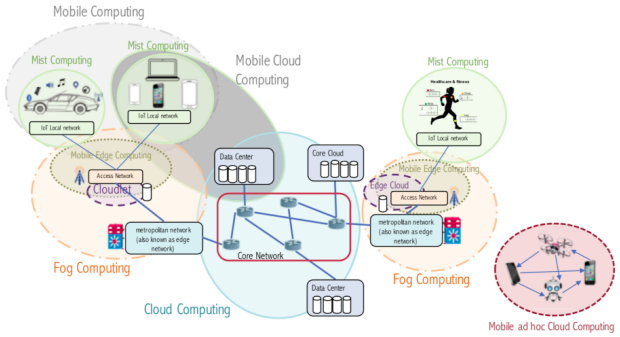
The transformation of the internet experience, with the shift towards web only services (e.g. web applications instead of desktop applications), resulted in a significant increase in volumes of shared data and required bandwidth. To address this increase in bandwidth demand, cloud providers introduced Content Delivery Networks (CDN) whose function is to bring service providers closer to final users. A secondo step was made with the spread of IoT and decentralized computing paradigms such as *fog computing*.



(a) Traditional infrastructure



(b) Modern infrastructure



(c) Today's infrastructure

4.1.1 Characteristics of data center networks

Data centers are huge structures with tons of servers, so they demand high bandwidth with the outside and a very low RTT within their premises. Also, each data center is a single administrative domain, meaning that its administrators have full control over the internal network infrastructure, endpoints and protocols. This allows them to deviate from standards and adopt each kind of custom solution, as long as the services they host are reachable.

This liberty becames relavant as soon as we realize that most of the traffic in a data center is created by machine-to-machine communications, leaving user-to-machine traffic a small percentage of the total amount.

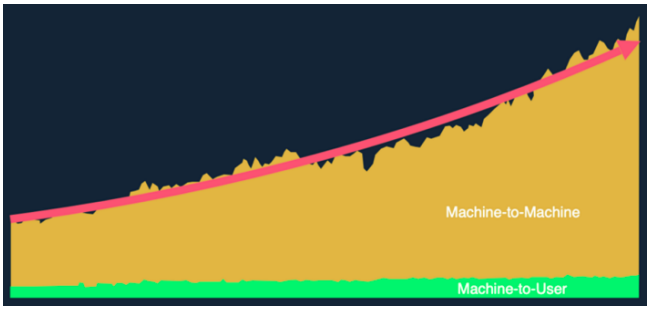


Image 4.2: Traffic within a data center

This means that performances of networks inside the data centers, which are called *interconnection networks*, are critical.

Interconnetion networks An *interconnection network* is composed of nodes and links, or communication channels. Nodes can be servers, memory units or even processors. Then, each node has an interface connecting it to the network, and the number of link the node is connected defines its degree. The interconnection fabric is made of switches[1] and links. Switches receive packets, and look inside them to determine the way toward their final destination. An n-way

switch is a switch with n ports that can be connected to n links. The interconnection fabric determines whether the *interconnection network* is *blocking* or not. It is *non-blocking* if any permutation of source and destination nodes can connect each other at any time. It is *blocking* if this is not true.

Interconnection networks can be distinguished by three parameters:

- *Topology*: defines the way nodes are interconnected;
- *Routing*: defines how a message gets from source to destination;
- *Flow control*: negotiates how the buffer space is allocated;

The *topology* also determines the *network diameter* and *bisection width.*

Definition 13 - Network diameter.

Network diameter is defined as the average distance between pairs of nodes.

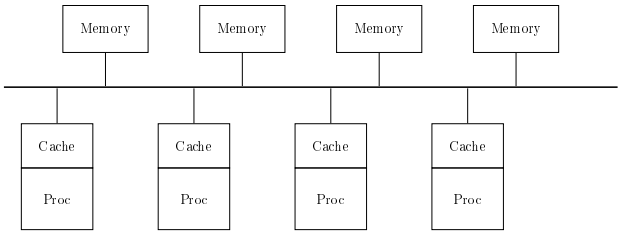
Definition 14 - Bisection width.

Bisection width is defined as the minimum number of links that have to be cut to partition the interconnection network into two halves.

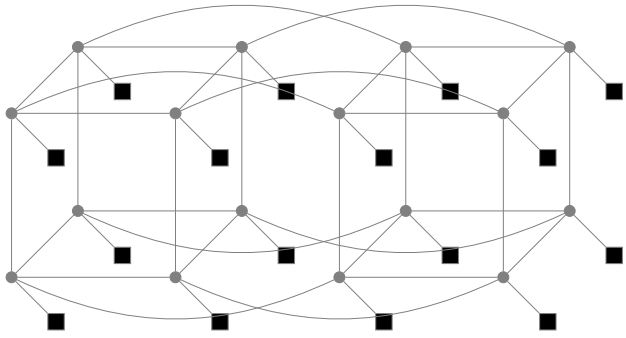
When an *interconnection network* is partitioned into two networks of them same size, the *bisection bandwidth* measures the communication bandwidth between the two. We talk about *full bisection bandwidth* when one half of nodes can communicate simultaneously with the other half.

Topologies There are two main types of *topologies*:

- *Static networks*: servers are connected through direct connections;
- *Switched networks*: servers are connected through switches;

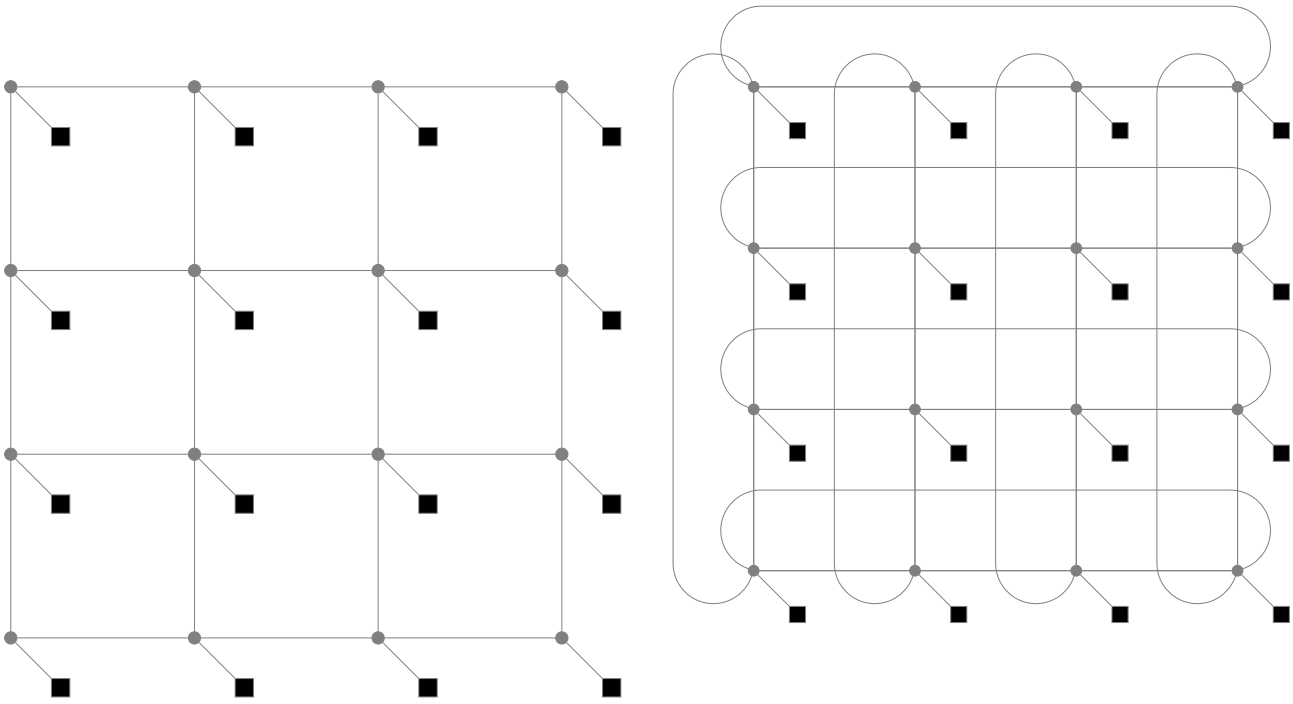


(a) *Bus topology*



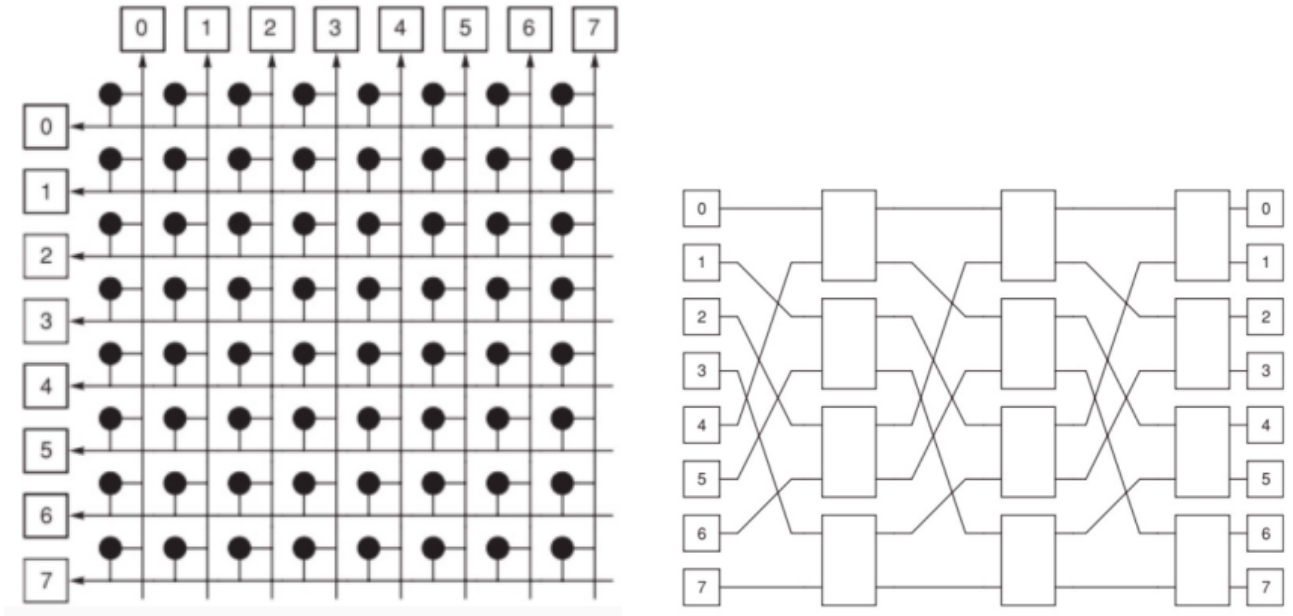
(b) *Hypercube topology*

[1] We're refering to a generic switching device without saying if is a layer 2 or layer 3 one



(c) *2D-mesh topology* (d) *Torus topology*

Image 4.2: *Static networks topologies*



(a) *Crossbar switch topology* (b) *Omega switch topology*

Image 4.3: *Switched network topologies*

Up to this point, we can say that an *interconnection network* must be scalable, provide a high bandwidth with low latency and must guarantee what is called *location transparent communication*, meaning that every server should communicate with the others with similar speed and latency. In simple terms, the position of a server in the data center shouldn't impact its networking performance. This latter requirement translates in the impossibility of adopting a hierarchical organization of servers.

Costs also come in hand when designing an *interconnection network*. Both servers and network apparatus costs are relevant, but talking about networking, the choice of routers and

switches often requires a compromise between latency and costs. For example, based on the number of ports of a router we can have low-radix and high-ridix routers. The former has few ports, while the latter has more; thus bandwidth is divided into a smaller number of wide ports and a larger number of narrow ports respectively.

In general, data center networks are designed around the following schema:

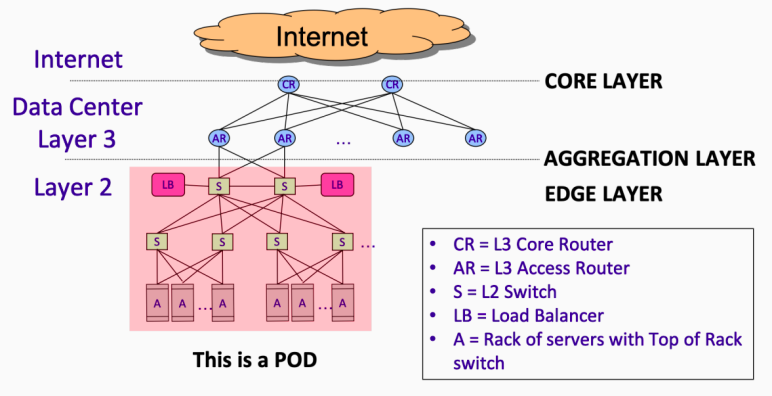


Image 4.4: Generale *interconnetion network* schema

Layer 2 and layer 3 has pros and cons that have to be balanced to guarantee ease of configuration, administration and problem-solving.

Layer 2, that is the ethernet switching part of the infrastructure, is easy to configure, thanks to autoconfiguration protocols such as DHCP, and allows the addition of servers in a plug & play way, but it's important to limit broadcast domains to avoid link saturarion due to broadcast frames, and dealing with the Spanning Tree Protocol might be a struggle.

On the other hand, IP routing on layer 3 makes scalability easy thanks to hierarchical addressing and allows obtaining equal-cost multipath routing[2]. However, its configuration is more complex and migration requires changing the IP addresses.

Note. At the end of the day, the entire *interconnection network* will look like a giant switch that allows all the servers to communicate one another.

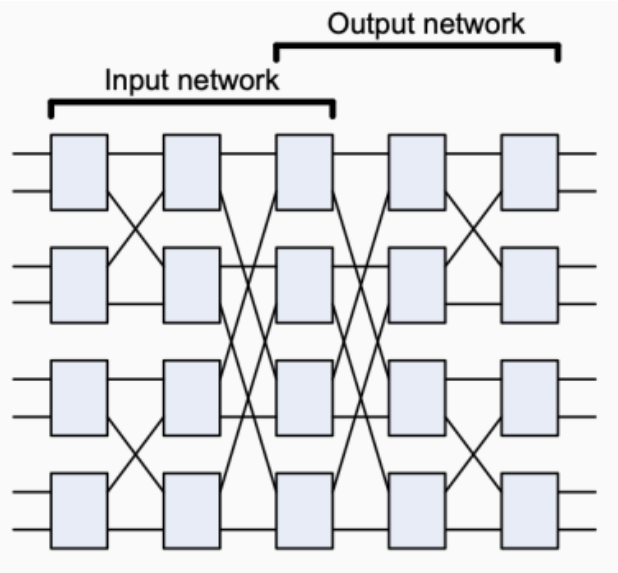
4.1.2 Topologies more in depth

In the *butterfly network topology* data always travels through the most efficient route, but two packets attempting to reach the same port at the same time would couse a collision, so this is a *blocking topology*.

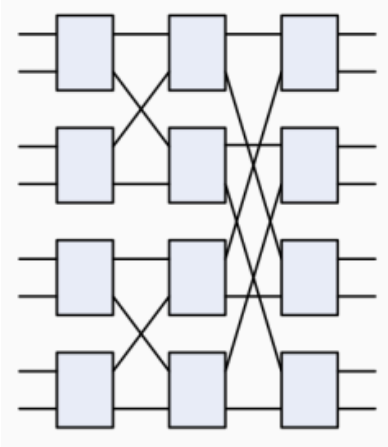
Note. The name comes from the pattern of inverted triangles created by the interconnections, which look like butterfly wings.

Clos networks is a *non-blocking topology* that consists of two *butterfly networks* in which the last stage of the output is fused to the first stage of the output. This way, all packets sent, overshoot their destination and then hop back to it. However, this overshoot is mostly unnecessary and increases the latency because each packet takes twice as many hops as it need. A solution to this is the *folded clos topology* which is obtained by making input and output nodes share the same switch devices. Such networks are also called *fat trees* and the *topology* is also called *fat tree topology*.

[2]We will see more about this later in this section



(a) *Clos topology* made of two *3-stage butterfly networs* and radix-2 routers



(b) *Folded clos topology*

Image 4.5: *Clos* VS *folded clos topology*

Fat tree topology In a *fat tree* network, servers are placed at the leafs, while internal and root nodes are switches. To increase bandwidth, additional links are also added to near-root nodes. The advantage of such *topology* is that all the switching elements are identical and this allows a reduction in costs. Another, already named, characteristic is the presence of multiple equal-cost path that both allow for load splitting and *location trasparent communication.*

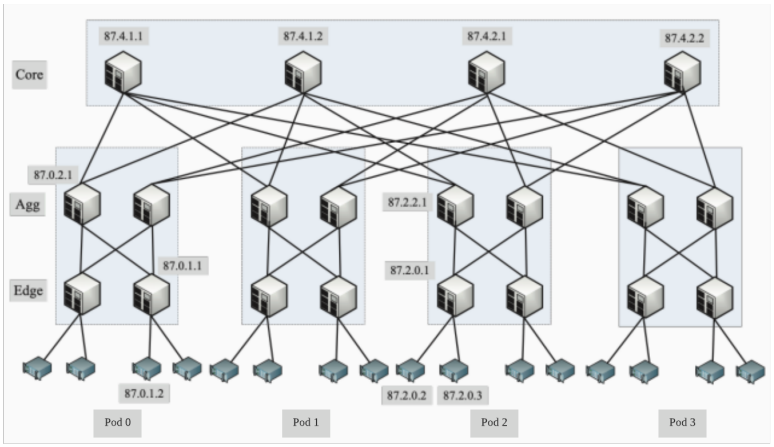


Image 4.6: Tipical *fat tree topology*

A *fat tree* made of k pods has two layers of $k/2$ switches for each pod. Each switch at the lower layer is directly connected to $k/2$ servers and $k/2$ switches of the upper layer.

Note. From the lower to the upper, layers are also called *edge*, *aggregation* and *core* respectively.

Similarly, each *aggregation layer* switch is connected to $k/2$ *core layer* switches which also have $k/2$ more connections to other *aggregation switches*, one for each pod. This configuration results in an infrastructure that has 4 servers, actually, 4 racks with 4 on-top-of-rack switches, $k(k+1)$ switches and a total of $(k/2)^2$ paths connecting each pairs of servers.

When it comes to IP assignment, switches are numbered left-to-right and bottom-to-top as *base.pod.switch*.1 (in the previous image, *base* = 87). *Core layer* switches instead, are numbered as *base.k.i.j* where k is the number of pods and i, j denotates the coordinates of the switch

in the grid, starting from top-left. Finally, servers IPs are in the form *base.pod.switch.serverID* and *serverID* is assigned from left-to-right.

4.1.3 Open issues

Existing *topologies*, including *Fat trees*, still fail to address workload balancing and TCP-related issues.

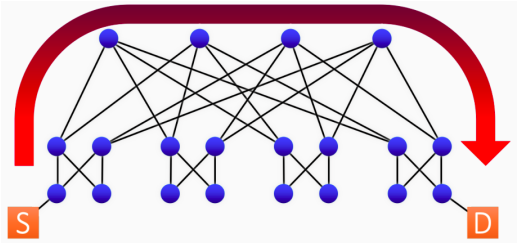
Note. TCP retransmission and congestion avoidance policies causes a degradation of throughput, so new protocols are currently being developed to displace TCP (e.g. QUIC).

When it comes to workload, we distinguish between small and large flows. As the names suggests, the first denotates small flows of data which require low latency, while the second is about bigger flows which should benefit a higher throughput.

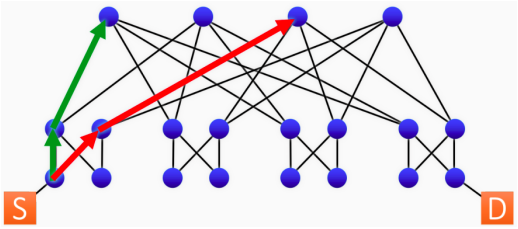
Note. Small and large flows are also called mice and elephants flow respectively.

Example 1 - Traffic balancing problems.

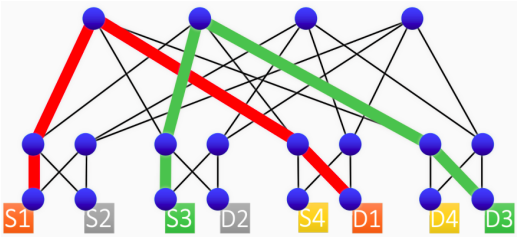
Let's assume to be in the following situation and have many flows going from S to D:



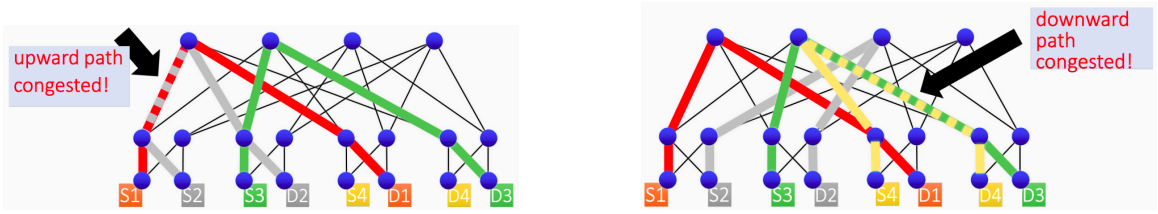
We want to assign each flow to a path in the most efficient way. We could just use the Equal-Cost Multipath routing (ECMP) to randomly assign a path to each flow. This would be a simple approch, but it would also be totally agnostic of available resources; thus we could experience long-lasting collisions between elephant flows.



The problem complicates when we have more source and destination nodes to connect. Ideally, we would like paths to not intersect each other, like in the above picture:



Wrong choices might cause collision in both upward and downward paths:



(a) Upward path congested

(b) Downward path congested

Proposed solutions Up to now, various solutions have been proposed, but none of them solves complety all the issues. For example, an idea was to detect elephant flows by putting a threshold on link capacity (e.g. 10%), compute non-conflicting path for each flow over the limit and using plain ECMP for mice flows.

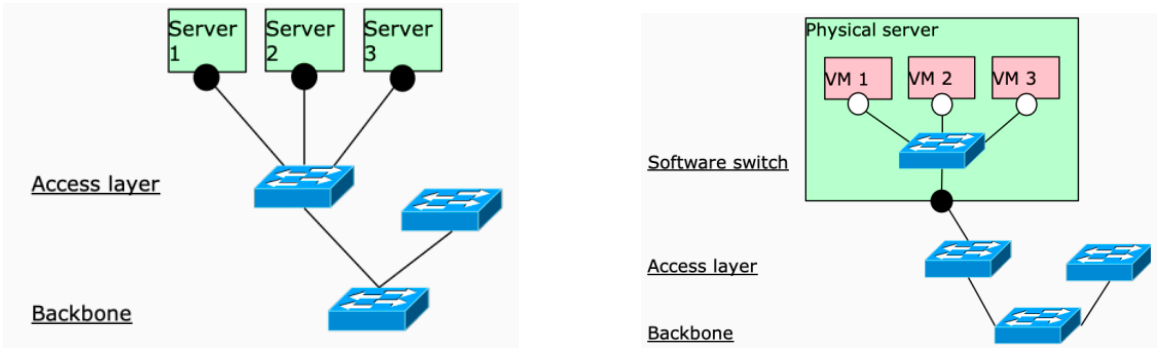
Other proposals where to reroute flows based on link utilization or queue occupancy. However, the first favors mice flows and, vice versa, the second tends to promote elephant flows. We could even try to spread all packets evenly, but that would make congestion control hard to implement.

4.2 Networking in virtualised environments

Virtualization introduces additional complexity for networking, because we need to deliver traffic between VMs or containers and the outer world, and also grant communication between virtualised environments within the same physical machine. We also need to decide how to assign IP addresses to VMs and containers and how to provide advanced features such as load balancing, firewalls, etc.

Note. From a networking perspective there is no difference between VMs and containers.

The first idea would be to imitate what we have in the physical world. So, as we have many servers connected to a switch who's also connected to a backbone, in a virtualised environment, we might connect each environment to a virtual switch and then connect the virtual switch to the physical NIC.



(a) Physical world

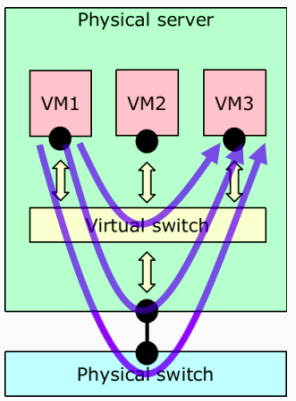
(b) Virtual world

Image 4.7: Copy-paste approch

4.2.1 North/south and east/west communication on a single server

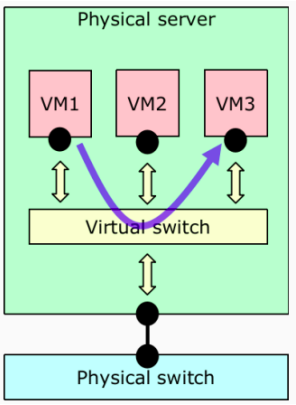
Although we suggested that a virtual switch might be the solution to virtual environments networking, in fact, three main options exist:

1. *Host-based switching*: using a virtual switch in the *Hypervisor* or directly in the *Host OS*;
2. *NIC switching*: using virtual queues in the NIC;
3. *Hairpin switching*: passing through the external switch;



Host-based switching Historically, this is the first solution used and is also called *software bridge* o *softswitch*.

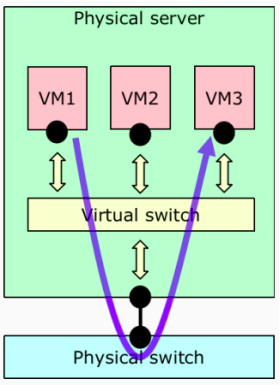
This solution grants both high bandwidth and reduced overhead for inter-VM traffic. Plus, it allows enforcing policies (e.g. firewall rules) directly on the virtual switch as if it was a physical one, so the features are the same. However, this introduces an additional component to configure (i.e. the switch itself has to be configured) and also an additional processing overhead because every message has to pass throught said switch.



It's worth mentioning that this solution requires the physical NIC to operate in promiscuous mode. That's because, since virtual environment MAC addresses are different from the physical NIC one and, by default, each NIC doesn't process packets whose destination MACs are different from its own, those packets would be dropped.

Hairpin switching This solution uses the physical switch, allowing us to leverage its power thus reducing CPU cycles consumption.

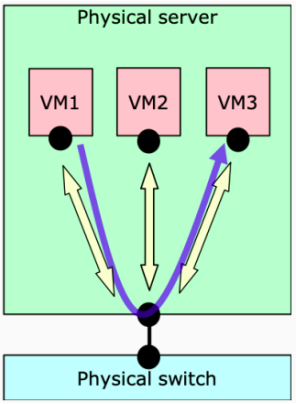
However, this is not necessarily a good thing because PCI bandwidth is smaller than CPU-to-Mem bandwidth, thus it might be saturated quickly. Another problem is that traffic always need to pass through all the intermediate components (e.g. *VMM*, NIC, physical link) and, in case of VM-to-VM communications data crosses those components twice. For all of these reasons, this solution is no longer used.



Again, it's worth mentioning that to use the existing hardware it has to be compatible with *hairpin switching* because, by default, switches don't forward traffic on the same port on which it has arrived.

NIC switching *NIC switching* is an intermediate solution between the previous two, both conceptually and in terms of performance.

Again, passing through the NIC means reducing CPU consumption for networkig related tasks, but we still have to rely on PIC bandwidth. Since data doesn't leave the machine unless it's necessary, we still manage to get improved performance in comparison to *hairpin switching*. However, to be able to enforce complex or advanced networking policies, a standard NIC might not be enought. When that's the case we could opt for a Smart NIC, that is a normal NIC with increased computing power.



Smart NIC are expensive, so their use must be weighed with the characteristics of the actual infrastructure. For example, if the amount of CPU spent due to networking tasks is so large that it leaves little to no time left for applications, it might be worth considering smart NIC adoption. Otherwise, a *host-based* solution could be enough.

Note. For instance, networking may become computationally relevant when using overlay tunneling protocols or DDoS detection and prevention algorithms.

Hardware VS software solutions Software switches provides better performances when local VM-to-VM traffic is predominant, but also when we're dealing with untrusted sources or over-subscribed links (i.e. tipical cloud service provider situations), or when we need a fine-grained control over passing data. To get all of this we pay in terms of CPU consumption reducing resources available to VMs or containers.

Hardware switches instead, are good when most of the traffic is between different servers and when we're applying similar policies to all nodes in the data center. Currently, software solutions are the most used.

4.3 Software bridges in Linux

Linux allows creating a bridge across many interfaces and one unique virtual interface that is valid for the entire host.

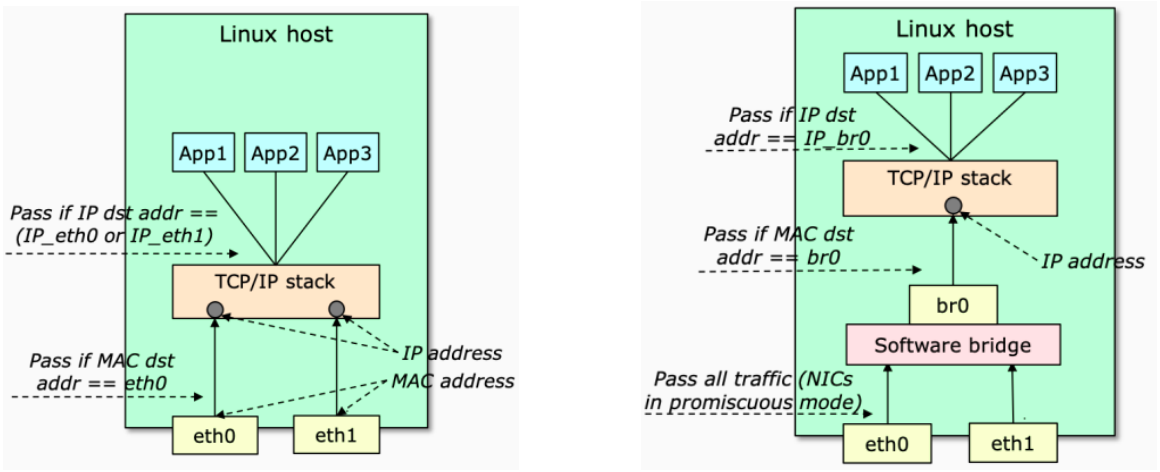


Image 4.8: Traditional VS bridged network

In the past, the setup of a bridging process in Linux was a way to have an 802.1D bridge, running in software, that was part of a bigger network, thus allowing us to modify the bridge code and play with it without having to buy a physical device. Now, it became a way to overcome the limitations of the IP protocol who requires each interface to have a unique address. Linux offers three ways to create a software bridge: *Linuxbridge*, *macvlan* and *Open vSwitch*.

4.3.1 Linuxbridge

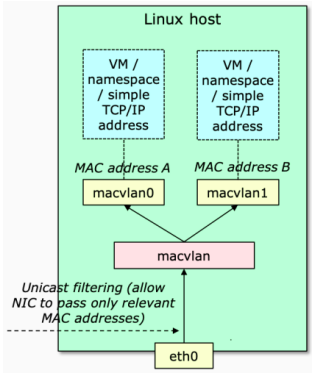
This is the most common kind of bridge, and it behaves exactly like a physical one. To receive packets intended for virtual hosts, it requires NICs to work in promiscuous mode. In fact, packets intended for virtual hosts will have the MAC address of `br0` virtual device as destination. Then, traffic can be sent from both physical and virtual interfaces and the IP address configured in the bridge is reachable from both interfaces (i.e. it works like a sort of loopback address).

4.3.2 Macvlan

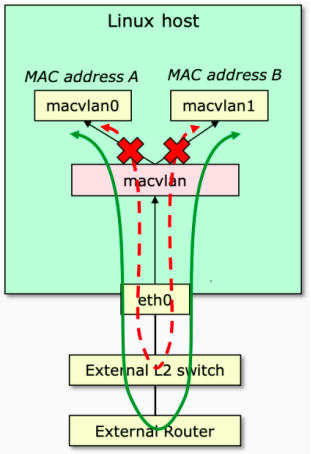
Macvlan implements a VLAN-like behaviour by using MAC addresses instead of 802.1Q tags. Each virtual environment can be assigned to a differenct *macvlan interface*, and each interface has its own MAC address and a distinc IP. *Macvlan* supports four operating modes: *private*, *VEPA*, *bridge* and *pass-through*. For all of these, except for the latter, supporting NIC can be configured to filter unicast packets based on their MAC address instead of running in promiscuous mode.

Private mode *Macvlan* in *private mode* mimics exactly the behaviour of VLANs, meaning that *inter-macvlan* traffic is forbidden and different *macvlans* can communicate only by passing through a router.

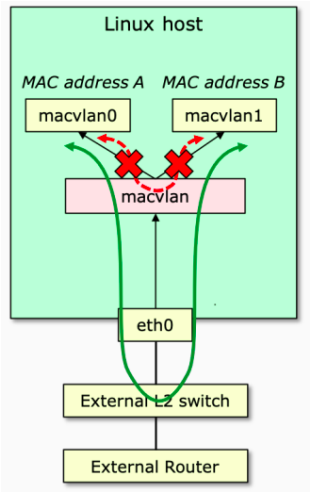
VEPA mode In the case of *VEPA mode*, which stands for *Virtual Ethernet Port Aggregator*, *A* and *B* can't communicate directly each other, instead, they need to send their frames to a physical switch out of the server (actually, it may also be a layer 2 switch on the NIC), and when the frame is sent back into the server, the *macvlan* drivers can deliver the data correctly.



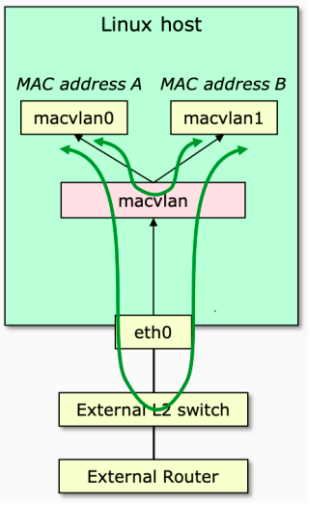
(a) *Macvlan* overview



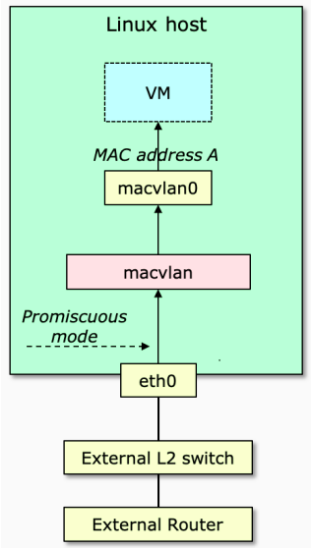
(b) *Macvlan private mode*



(c) *Macvlan VEPA mode*



(d) *Macvlan bridge mode*



(e) *Macvlan pass-through mode*

Image 4.8: *Macvlan* modes

The problem is that the external switch can't be a traditional 802.1D bridge, for the same reason we've already talked about: switches don't sent data back to the same port on which it was received. However, *VEPA mode* is the default one.

Bridge mode *Bridge mode* emulates a simple bridge, that is a bridge with just one uplink, no filtering database (i.e. no MAC learning) and no Spanning Tree Protocol. By means of that, it can forward all kind of traffic, without necessarily going out of the server and then back into it.

Pass-through mode This mode supports only one *macvlan* device, that is to say it can handle a single VM, for instance. This way, all traffic can be sent to the upstream component (e.g. the traffic can be sent directly to a VM), and said component can use any MAC address because, nor the *macvlan* drivers, nor the NIC do any MAC filtering; it will be the VM virtual NIC who will take charge of that. In fact, MAC *A* is no longer used. Of cource the physical NIC needs to work in promiscuous mode.

Note. *Pass-through mode* behaves like if the physical NIC were directly assigned to the virtual environment.

Linuxbridge VS macvlan *Linuxbridge* drivers behaves as a traditional hardware bridge, so they can handle complex situations in which, for instance, hosts on the same physical domain (layer 2 network) resides both on the same host and outside it. They're also useful when more VMs or containers on the same host need to communicate, or when advanced functions (e.g. flood control) are needed.

Macvlan drivers instead, are both simpler and less CPU consuming, because they emulate a trivial bridge with basic functionalities. Therefore, they're recommended when you just need to provide virtual machines or containers an egress connection to the physical network.

4.3.3 Open vSwitch

Open vSwitch (*OVS*) is a software implementation of a virtual multilayer network switch. It's designed to enable effective network automation through programmatic extensions, and it

can operate both as a software-based network switch, running within a *VMM*, and a control stack for dedicated switching hardware. Talking about its implementation, it's implemented into the Linux Kernel, and it's designed to support both standard management interfaces and protocols (e.g. NetFlow, sFlow, ...). It allows for a transparent distribution across multiple physical servers by enabling the creation of cross-server switches in a way that abstracts out the underlying server architecture.

Note. *Open vSwitch* is often implemented into cloud computing software platforms and virtualization management systems (e.g. OpenStack, OpenNebula).

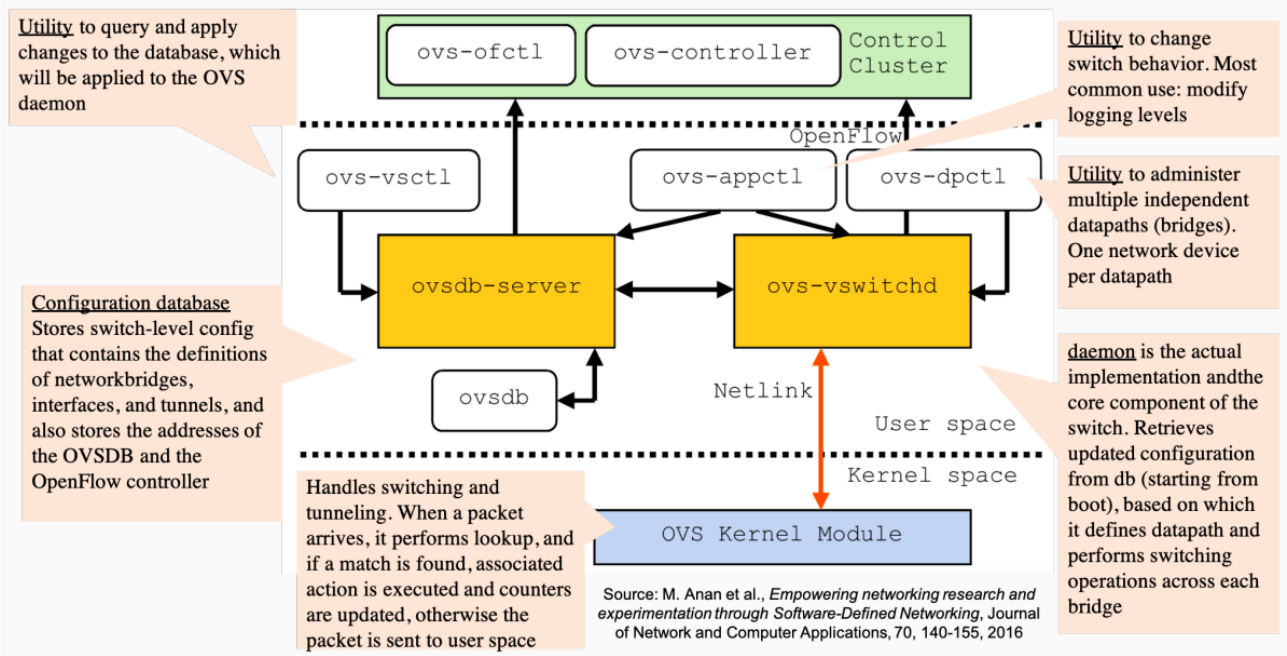


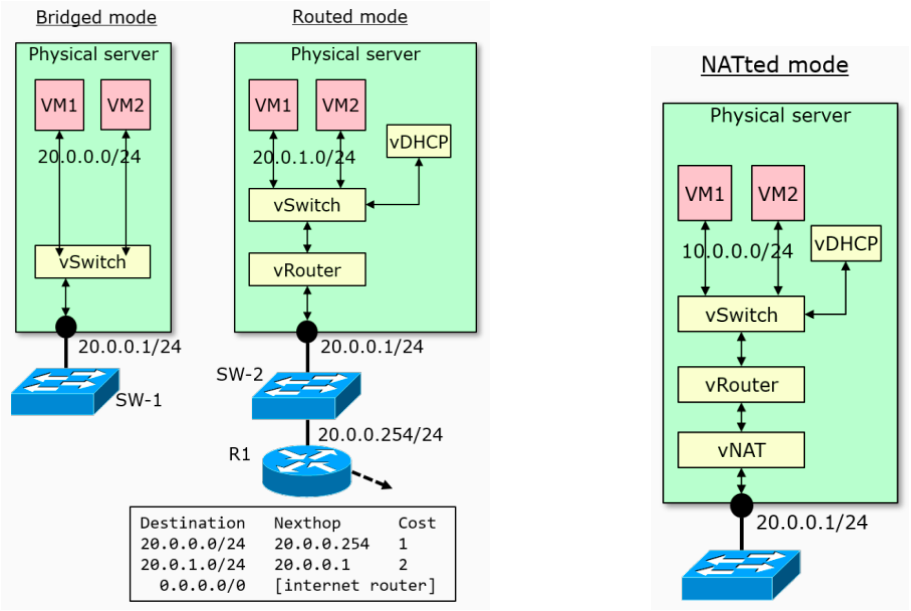
Image 4.9: *Open vSwitch* general implementation schema

4.4 Single server networking services

IP assignment The first problem is about deciding who should assign IP addresses to VMs and containers and also which addresses should be assigned. The first option could be to use the same addresses of the physical (meaning non virtualised) network so that inboud and outbound connections toward VMs or containers don't differe. However, this approch would require the coordination of the physical network manager, thus limiting virtualization agility.

The alternative, would be to use private addresses and then connect to the internet through a NAT. This would be good because IP assignment wouldn't be limited by public IP limits, and it wouldn't require coordination with the physical network manager. Of course, we would have to define static rules to support inbound connectivity and virtual environment would be reached by different IP addresses depending on wheter the sender is inside or outside the private network.

Note. The first solution proposed is also called *direct routing*.



(a) *Direct routing* (b) *NATted routing*

Image 4.10: Solutions to IP assignent

Providing feature-rich network connectivity Up to this point, it should be clear that providing additional network services may require connecting VMs or containers to the physical infrastructure. For example, if private IPs are used for VMs, the *hypervisor* must also provide a router, NAT and a DHCP service. To do this, we can exploit functionalities provided either by the kernel or by another software such as Docker engine o similar. For example, we can handle NAT tables using the `iptables` tool.

Multi-tenancy Accomplishing *multi-tenancy*, that is to say, accomplishing a strong network isolation, is a must in virtualised environments.

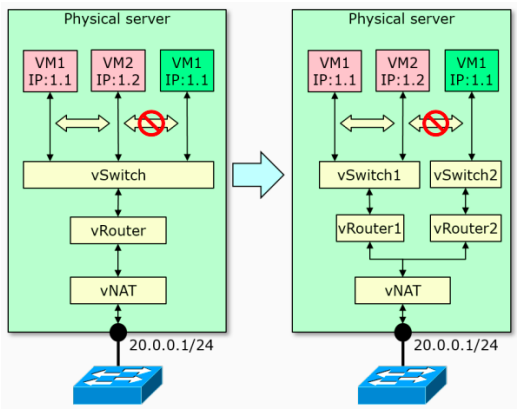


Image 4.11: *Multi-tenancy*

In the physical world this can be achieved through VLANs. However, in a server it's much more convenient to just create multiple virtual switches and routers.

Tenant-defined network services Each tenant might require a complex topology. For example a tenent might deploy 3 VMs who can communicate each other, but just one of which can access the internet. Again, we can achieve this by simply using pre-defined functionalities, provided either by the kernel or by other pieces of software.

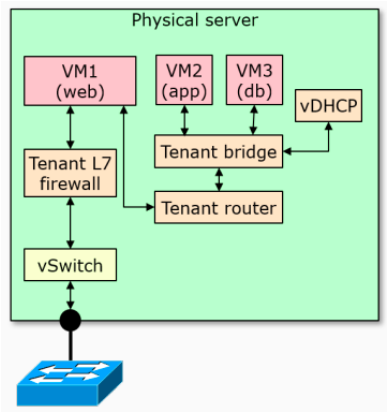


Image 4.12: *Tenant-defined network services*

Co-existence of virtual services and host applications Up to now we didn't mention that host servers have their own TCP/IP stacks, and they need to co-exist together with all the other virtualised services.

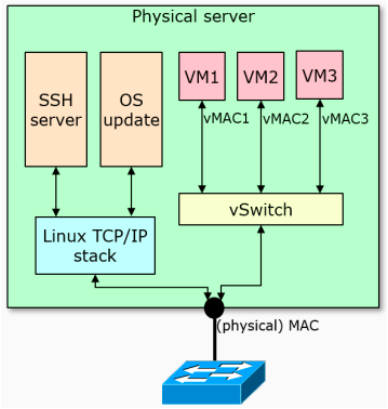


Image 4.13: Virtual and host services co-existence

In particular, there should be a way to distinguish between traffic directed towards host applications and VMs or containers. To do so we can exploit the Linux networking stack.

Putting all together Putting all together what we've said so far, we obtain the following general schema:

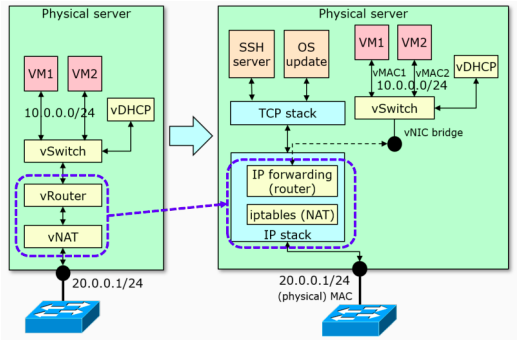


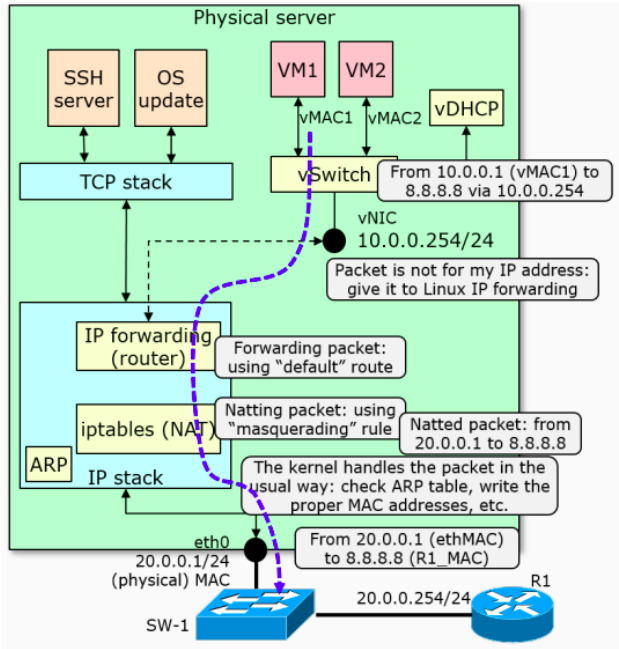
Image 4.14: Putting all together

So, virtual routers and virtual NAT services are in fact implemented into the Linux networking stack. Each cluster of VMs or containers is associated to a virtual NIC. Each virtual environment has its own MAC address and so the virtual NIC. In what we've called *NATted mode*,

all frames sent by hosts with a virtual MAC, pass through the kernel IP stack who translates those addresses according to pre-defined NAT rules. Then, inbound traffic follows the same route backward up to its destination.

Example 2 - Example of communication.

Let's consider a packet sent by VM1 to the internet:




```
$ sudo ip route
default via 20.0.0.254 dev eth0 proto dhcp
20.0.0.0/24 dev eth0 proto kernel scope link src 20.0.0.1
10.0.0.0/24 dev vNIC0 proto kernel scope link src 10.0.0.254
```

```
$ sudo iptables -t nat -L
target      prot  source          destination
MASQUERADE  tcp   10.0.0.0/24   !10.0.0.0/24   masq ports: 1024-65535
MASQUERADE  udp   10.0.0.0/24   !10.0.0.0/24   masq ports: 1024-65535
MASQUERADE  all   10.0.0.0/24   !10.0.0.0/24
```

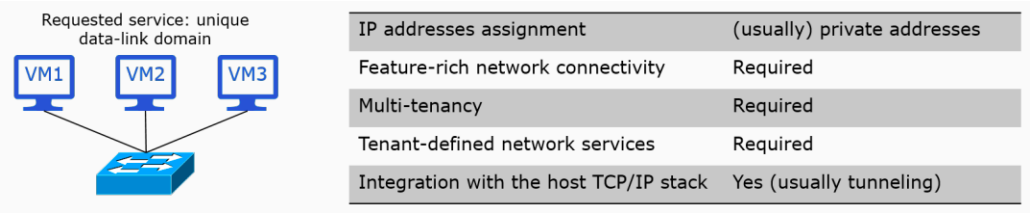
So, VM1 sends the packet to its default gateway, which is `vNIC`. The vNIC receives the packet, it detects that it is not the final recipient, so the packet is transferred to the Linux IP forwarding module. Finally, the Linux IP forwarding module forwards the packet to the next hop based on host routing and NAT tables as usual

4.5 Data center-wide networking services

When moving from a single server to a data center we need to solve the same problem as before. First of all, we have to distinguish between *tenant* and *cloud manager view*. The first wants to deploy some services without caring about the underlying physical infrastructure. The cloud manager instead has to manage the physical infrastructure so that it's able to provide the requested logical view.

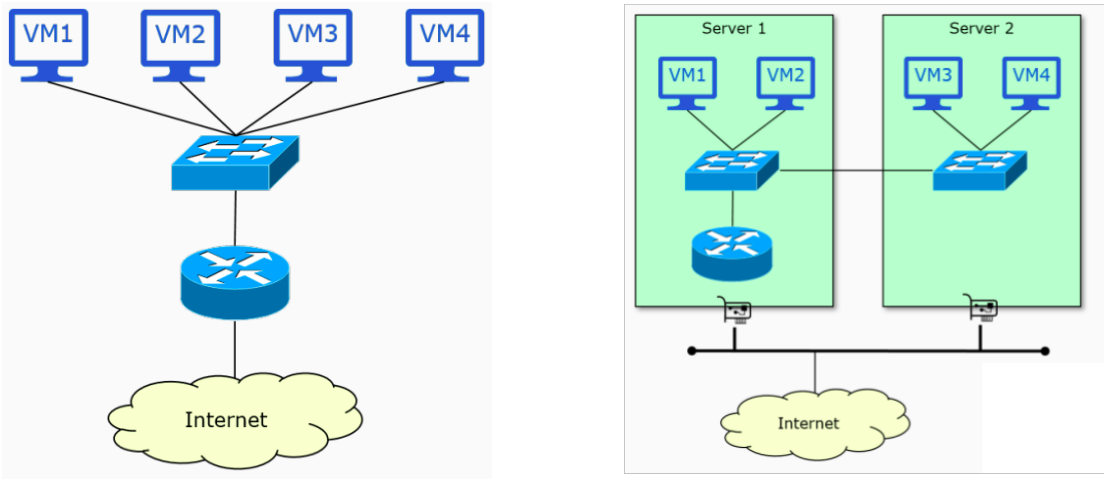
4.5.1 Providing layer 2 connectivity

Tenants may want to have a vannilla layer 2 connectivity that spans amongs all of their services across the data center. This is the case in which tenants want to take IP addressing under their own responsibility. The solution to this is the creation of tunnels between servers. What we'll see are tunnels managed by GRE (General Router Encapsulation).



IP addresses assignment	(usually) private addresses
Feature-rich network connectivity	Required
Multi-tenancy	Required
Tenant-defined network services	Required
Integration with the host TCP/IP stack	Yes (usually tunneling)

Image 4.15: Characteristics required by layer 2 connectivity



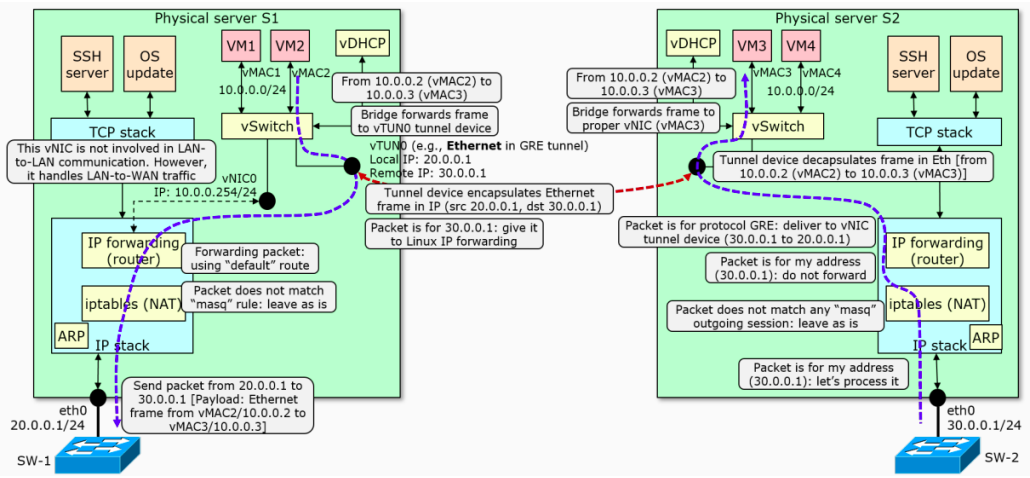
(a) *Tenent view*

(b) *Cloud manager view*

Image 4.16: *Tenenat* VS *cloud manager view*

Example 3 - Example of communication.

Let's consider a packet sent by VM2 to VM3:



To provide layer 2 connectivity between virtual machines on two different physical servers we have created a tunnel (i.e. the red dotted line between two virtual NICs). Essentially, what happens is that, when the packet sent by VM2 reaches the virtual switch, it forwards the packed to `vTUN0` *device.* `vTUN0` *is the interface associated with the tunnel and its remote IP defines the destination address for that packet. In this case it is* `3.0.0.1`*, which is the address of the second physical server, beacause VM3 resides in it. So, the tunneling protocol wraps the original*

packet with a new IP packet that has destination address `3.0.0.1` *and source* `2.0.0.1` *which is the address of physical server 1. The packet is finally taken by the Linux networking stack which sends the packet.*

The destination server, receives the packet, sends it to the tunnel device, which decapsulates the additional header and finally forwars the original frame to the correct destination.

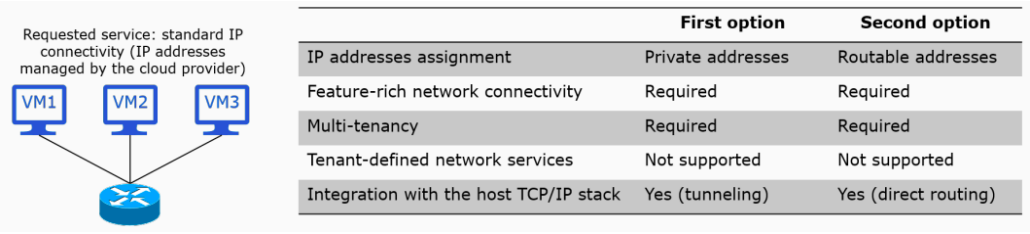
Note. In this situation NAT isn't necessary because the resulting packet looks like a packet sent by an host application.

Could we have used VLAN instead of tunnels?

Short answer: no. The reason is that it would require cooperation with the physical network manager. Also, VLANs are limited to 4096 and setting up everything would be trickier than what we've done for tunnels.

4.5.2 Providing layer 3 connectivity

With layer 3 connectivity, tenants rely on a cloud service provider for IP assignment and management, and this can be achieved in two ways: *tunneling* and *direct routing*.

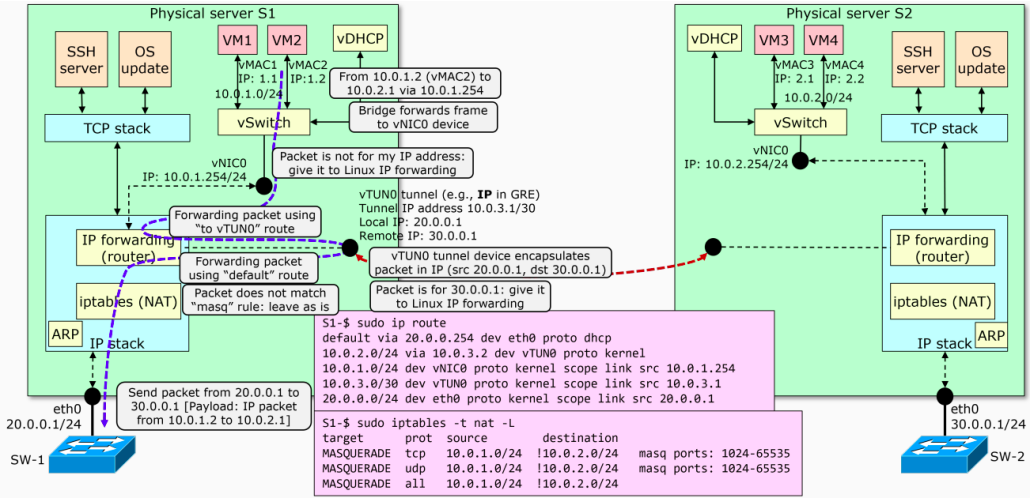


	First option	Second option
IP addresses assignment	Private addresses	Routable addresses
Feature-rich network connectivity	Required	Required
Multi-tenancy	Required	Required
Tenant-defined network services	Not supported	Not supported
Integration with the host TCP/IP stack	Yes (tunneling)	Yes (direct routing)

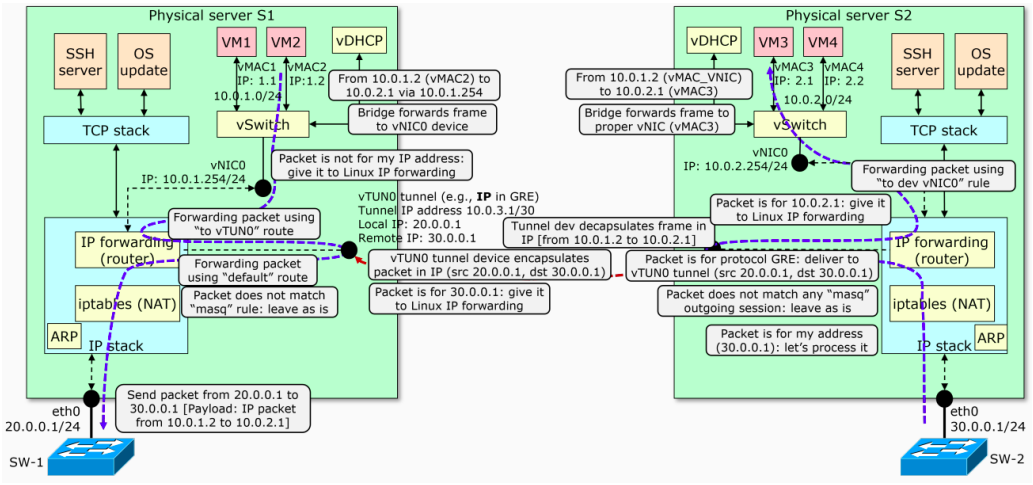
Image 4.17: Characteristics required by layer 3 connectivity

Example 4 - Example of communication using tunnels.

Let's consider the same case as before: a packet sent by VM2 to VM3:



This time the tunnel interface is connected to the Linux virtual router. Once the packet reaches that interface it is again encapsulated into the same packet as before, and then sent back to the router who forwards it outside and up to the other server.

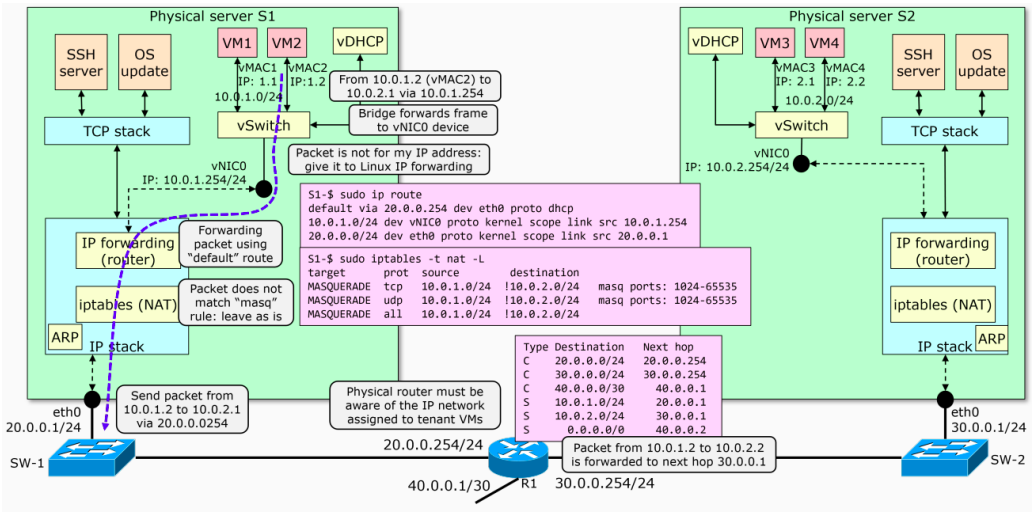


Once the packet has reached the destination server and has arrived to the other side of the tunnel, the original packet is decapsulated and sent up to VM3.

Note. VM2 and VM3 are on different private networks.

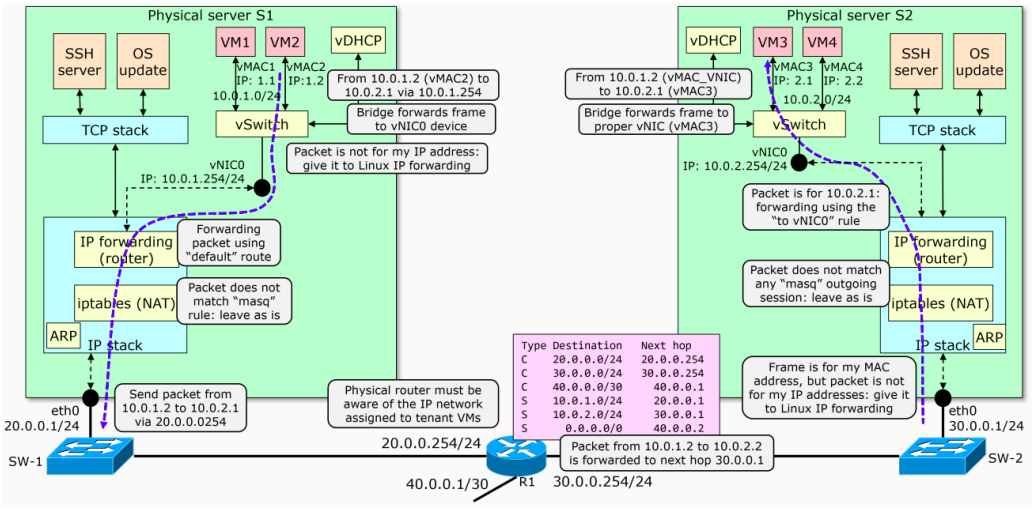
Example 5 - Example of communication using direct routing.

Let's consider the same situation as before, but with direct routing instead of tunneling:



This time, the packet is sent directly out of the server without any modifications, and it reaches router `R1`*. The router knows, because it has been configured, that the destination network* `10.0.2.0/24` *(i.e. the network to which VM3 is connected) is reachable via* `30.0.0.0/24` *network. So, the packet is forwarded out of the interface with IP* `30.0.0.254` *with next hop address set to* `30.0.0.1`*, that is the address of the second physical server.*

When the packet reaches the destination server, it can go directly up to VM3 as a normal packet.



Note. It's easy to understand the route of every packet if the behaviour of every component is considered alone.

Note. Even if we've proceded in our discussion considering inconvinient any solution that required to manage directly physical hardware, *layer 3 connectivity* through *direct routing* is often used, and is made convenient by the possibility to configure routers via a set of REST APIs.

Tunnels in practice For the sake of semplcity we've considered a single tunnel between a pair of servers, but this is obviously difficult to handle on a large scale because it would require a full mesh of connections. This is therefore the reason for which tunneling technologies such as GRE (Generic Routing Encapsulation) are often replaced by others. In particular, *VxLAN* allows providing both a trasparent full mesh solution, with a single endpoint for all the hosts belonging to the same VxLAN domain, and a way of distributing traffic across multiple parallel links through layer 4 ports.

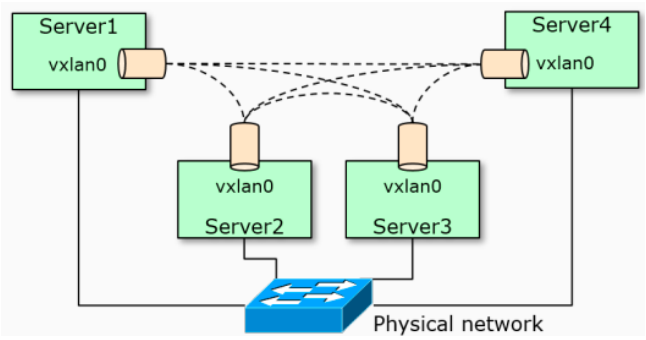


Image 4.18: *VxLAN network*

Tunneling VS direct routing *Tunneling* enables the deployment of tenant services without any interaction with the infrastructure provider and becomes useful in case of public datacenters. However, it may trigger some performance problems due to the reduced MTU on the tunnel: since packets are being encpasulated, the maximum size of the original packets must be reduced. *Direct routing* on the other hand, doesn't require changing the MTU, but the infrastructure provider has to cooperate.

Chapter Nr.5

Cloud storage

After going through CPU and network virtualization, it's time to talk about proper ways of handling storaged data. In the age of *cloud computing* there's a huge amount of data to store, access and secure, so novel approches to system design are necessary. To achieve the goal of cloud computation, effective data replication and appropriate storage management strategies are critical.

In the past, storage systems were designed according to a *performance-at-any-cost* philosophy, but now there's been a shift towards *reliability-at-the-lowest-possible-cost*. For example, data replication allows concurrent access to data from multiple processors and decreases the chances of data loss, but maintaining consistency among multiple copies of data records increases complexity of the management software, and could negatively affect the storage system performance as a whole, if data is frequently updated.

In this chapter we're going to explain some modern strategies adopted to manage cloud storage while also respecting performance and realibility requirements.

5.1 Preliminary definitions

Before proceding, we need to fix some concepts.

Definition 15 - Storage model.

A storage model describes the layout of a data structure in a physical storage.

Definition 16 - Data model.

A data model captures the most important logical aspects of a data structure in a database

Definition 17 - Read-write coherence.

The result of reading by a memory cell, should be the same as the most recent writing done on that cell.

Definition 18 - Before-or-after atomicity.

The result of every read or write operation is the same as if that operation has been performed completely before or after another read or write operation.

Read-write coherence and *Before-or-after atomicity* are two highly desirable properties of any *storage model*.

5.1.1 Types of storage

There are three main types of storage:

1. *Block storage*: data is managed as blocks within sectors and tracks. Can be used when storage has access to raw and unformatted hardware and is useful when both speed and efficiency are important;

2. *File storage*: data is organized as structured files which are managed through a file system. However, this doesn't work very well with large amounts of data or when there's high-demand for a particular piece of data;

3. *Object storage*: data is managed as objects. Typically, each object has a global unique identifier and holds both the data itself and some metadata. It's possible to access whole objects or blobs of data, but random accesses to data within an object is more difficult;

Block storage is often used for database because it's the ideal way of storing relational information. On the other hand, *Object storage* isn't suitable for databases, but is preferable to store content that can grow without bounds, for example, backups and archives. Finally, the *File storage* model can be easily used to create *distributed file systems* via network protocols.

Examples of *Block storage*, *File storage* and *Object storage* are, respectivelly, *Cinder*, *Google file system* and *OpenStack Swift*.

5.2 Block storage

We'll give a first quick look to this model by scratching the surface of *OpenStack Cinder*. *OpenStack Cinder* implements services and libraries to provide on-demand, self-service access to block storage resources. It provides API to interact with storage backends which is also exposed to the cloud. End users can manage their storage without knowing how it's organized and for both physical and virtual environments.

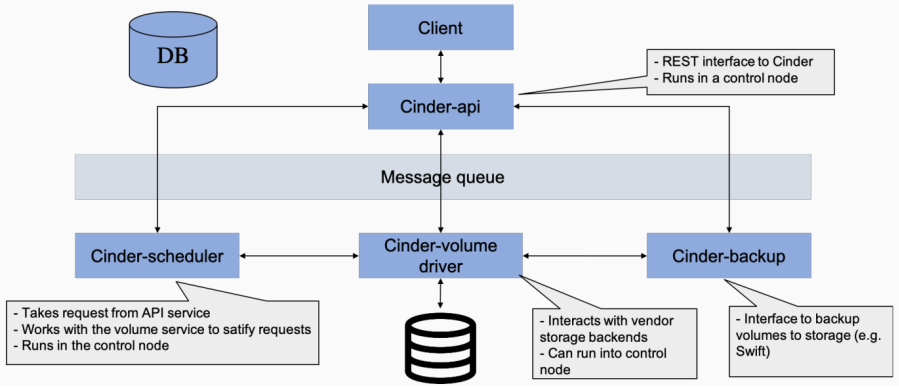


Image 5.1: *OpenStack Cinder* basic architecture

Cinder APIs allow users to create and delete volumes and snaphots (for backup pourposes). Attach or detach volumes for the available storage, clone and extend them, and so on.

5.3 Distributed file system

Again, we'll start by giving some definitions.

Definition 19 - File.

A file is a linear array of cells stored on a persistent storage device. It is seen by application as a collection of logical records, and it's stored on a physical device as a set of physical records, or blocks, whose size is dictated by the physical media.

Definition 20 - File pointer.

A file pointer is a cell used as starting point for a read or write operation.

A file has a logical organisation that reflects the *data model* from the perspective of the application, and a physical orgnisation that reflects the *storage model* and describes the manner the file is stored on a given storage media.

Definition 21 - File system.

A file system is a collection of directories and each directory provides information about a set of files. A file system controls how data is stored and retrieved.

5.3.1 Unix File System

The *Unix File System* (*UFS*) has both a layered and hierarchical design. The first allows it to separate the physical file structure from the logical one, and this becomes useful when dealing with files stored both locally and remotely. The hierarchical design, on the other hand, allows grouping of files in directories, and supports multiple levels of directories, or collections of directories and files; thus, supporting a good degree of scalability.

The metadata that *UFS* uses includes file owner, access rights, creation time, time of the last modification, file size, the structure of the file and the persistent storage device cells where data is stored. All of these are memorized in a structure called *inode*. Each *inode* holds information about individual files and directories and is kept on persistent media together with the actual data.

Going back to *UFS* layering, there are two layers:

- *Lower layer*: is about the physical organisation and is furderly separated into three sublayers:
  - *Block layer*: responsible for locating individual blocks on the physical device;
  - *File layer*: reflects the organisation of blocks into files;
  - *Inode layer*: provides metadata for files and directories;
- *Upper layer*: is about the logical orgnisation;

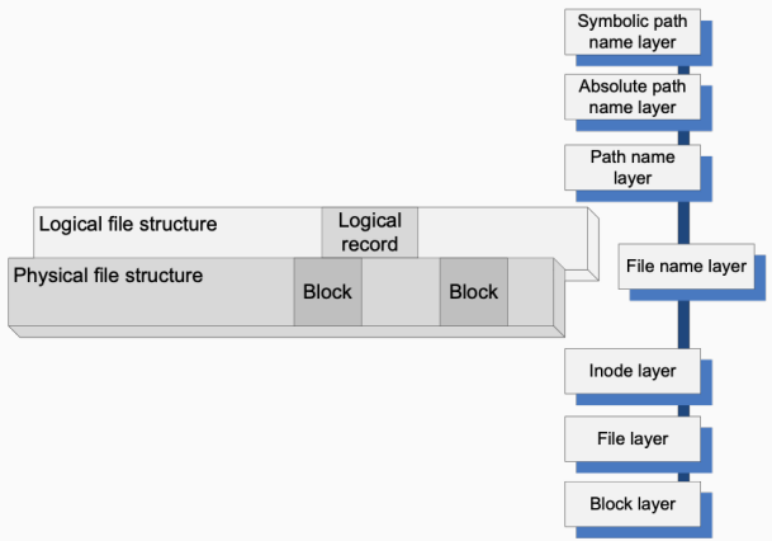


Image 5.2: *UFS* layering

5.3.2 Network File System

The *Network File System* (*NFS*) is designed to provide the same semantics as the *UFS* to ensure compatiblity with existing applications. *NFS* is based on the client-server paradigm. Clients run on the local host and interact with the server via remote procedure calls.

Each remote file is identified by a 32-bit file handler rather than a file descriptor like in *UFS*. A file handler is obtained combining a file system identification, an *inode* number and a generation number.

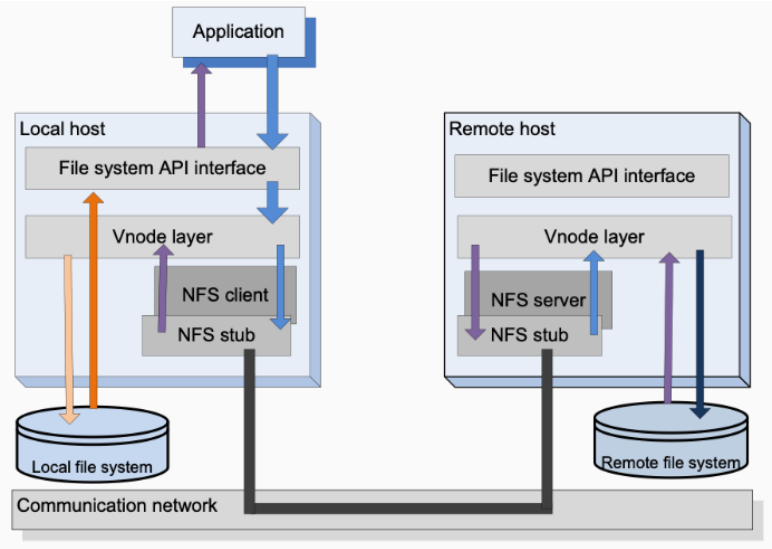


Image 5.3: *NFS* design

As the image shows, instead of an *inode layer* the *NFS* has a *vnode layer* which implements file operations in a uniform manner, regardless of whether the file is local or remote. Also, an operation targeting a local file is directed to the local file system without involving the *NFS*.

When a client interacts with the server, the client-side *NFS* packages the relevant information about the target, then sends those packaged information to the server-side *NFS*. The latter then passes the received information to the *vnode layer* of the remote host which finally directs it to the remote file system.

API	Description		API	Description
`fd`	File descriptor		`dfh`	The directory in which the file handle can be found
`fh`	File handle		`count`	Number of bytes to be transferred
`fname`	File name		`buf`	Buffer used to transfer data
`dname`	Directory name		`device`	The device in which the file system is located

Table 5.1: *NFS* APIs

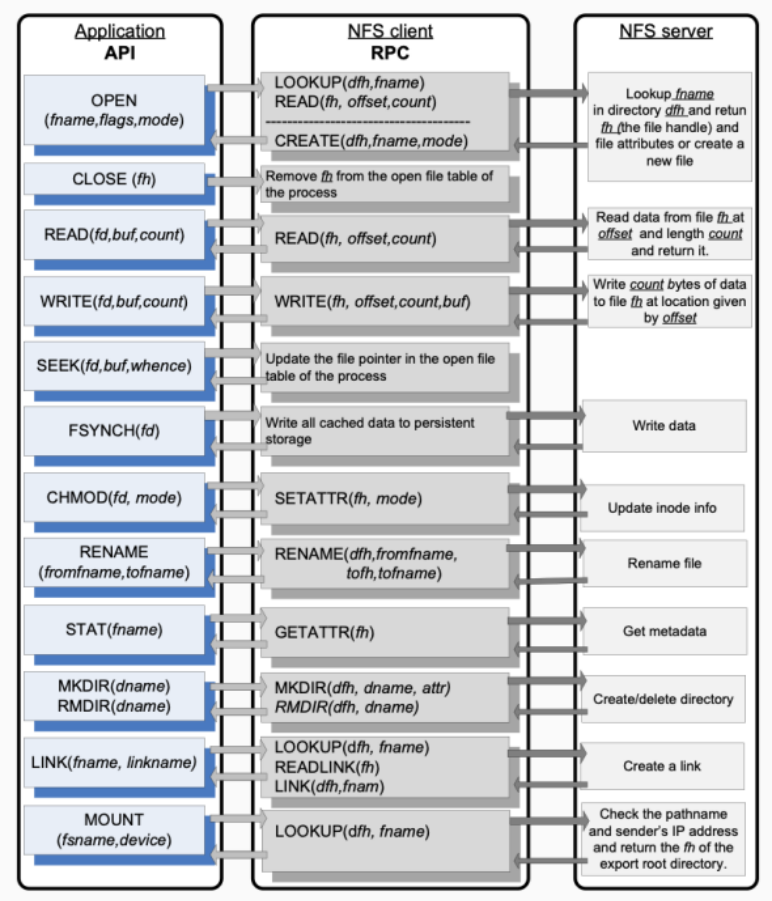


Image 5.4: *UFS* APIs and corresponding *NFS* remote procedure calls

Common design choices for distributed file system The vast majority of distributed file systems have been implemented in similar a way. That's not so surprising if we think that they all need to solve the same problems and that the possibles or best solutions to choose from are limited.

Tipically, every distributed file system is implemented in a way that once a file is closed, the server will have the newest version of it on persisten storage. Another concern is about what to do when writing on a file. The possible approches are two:

1. *Write-through*: a block is written to the disk as soon as it is available on the cache. This approch increases reliability, but it takes more time to complete each write operation;

2. *Delay in write-back*: a block is first written to cache and writing on the disk is delayed for a time in the order of tens of seconds. This speeds up writings and avoids useless writings when data is discarded before saving on disk is necessary. However, data can be lost in case of system failures;

Finally, how should the system act when multiple clients tries to access the same file at the same time? The possibilities are two:

1. *Sequential write-sharing*: a file cannot be opened simultaneously for reading and writing by several clients;

2. *Concurrent write-sharing*: multiple clients can modify the same file at the same time (in a concurrent way, of course);

5.3.3 General Parallel File System

General Parallel File System (*GPFS*) is a distributed file system developed by IBM that's designed for optimal performance of large clusters. To do so, it allows parallel I/O, meaning that multiple I/O operations can be executed concurrently.

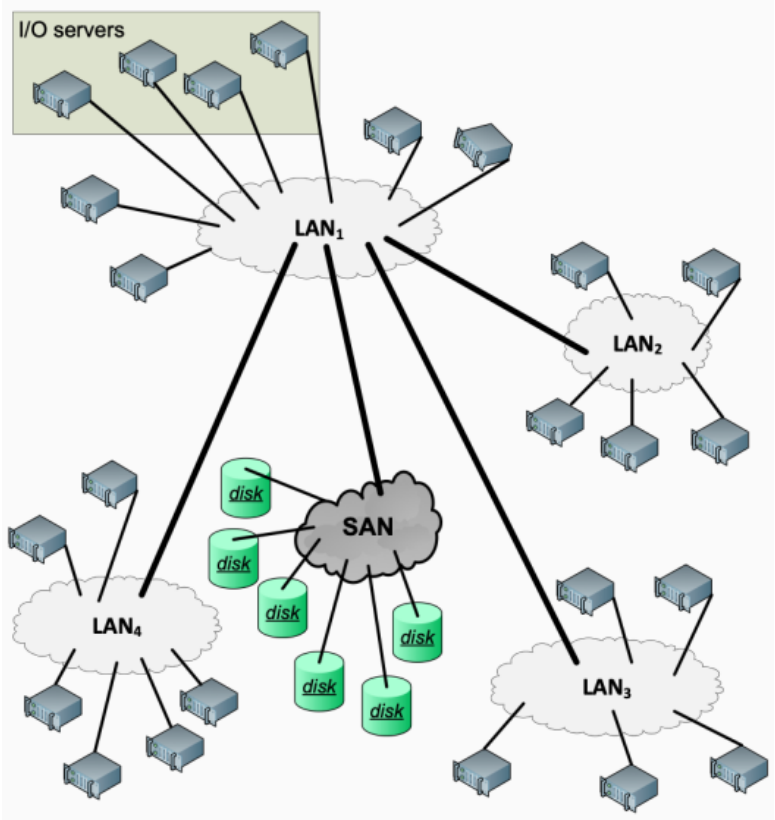


Image 5.5: *GPFS* architecture

As the image shows, every disk is interconnected via a *Storage Area Network* (*SAN*) and servers used to handle I/O are distributed in various LANs.

GPFS is a realiable file system because it can recover from system failures. To achive so, *GPFS* records all metadata updates in a write-ahead log file. Write-ahead means that updates are written to persistent storage only after log records have been written. Every I/O node manages one log file for each file system it mounts and any I/O node can initiate recovery on behalf of a failed node.

Finally, *GPFS* uses data striping to allow concurrent access and improve performance. By using data striping, *GPFS* segments logically sequential data (e.g. files) so that consecutive segments are stored on different physical storage devices. The problem with this approch is that a failed disk will affect many more files. To reduce the impact of this and further improve fault tollerance, *GPFS* data files as well as metadata are replicated on two different physical disks.

5.3.4 Google File System

Google File System (*GFS*) is another distributed file system, developed by Google that uses thousands of storage devices built from inexpensive commodity components to provide PBs of storage to a large user community with various needs.

Design consideration	**Design choice**
Vast majority of files range in size from a few GBs to hundreds of TBs	Files are segmented in large *chunks*
Most common operation is to append to an existing file and random write operations to a file are extremely infrequent	Implement an atomic file append operation allowing multiple applications operating concurrently to append to the same file
Consistency model should be relaxed to simplify the system implementation but without placing an additional burden on application developers	Ensure consistency by channeling critical file operations through a *master*, a component of the cluster which controls the entire system

Other considerations made are:

- Scalability and reliability are critical features of the system, and they must be considered from the beginning, rather than at some stage of the design;
- Sequential read operations are the norm;
- Users process the data in bulk and are less concerned with the response time;

And other relevant design decisions are:

- Clusters must be built around a high-bandwidth rather than a low-latency interconnection network, so control and data flows should be separated. High-bandwidth data flow should be scheduled by pipelining data transfer over TCP connections and finally, the network topology should be exploited by sending data to the closest node in the network;
- Caching at client side should be eliminated to reduce the overhead that's necessary for maintaining consistency among cached copies;
- Efficient checkpointing and fast recovery mechanisms should be supported as well as an efficient garbace colletor;

GFS chunks So, as we said, files are segmented into large *chunks*. To be more precise, each *chunk* is large exactly 64MB. This choice was motivated by the desire to optimize performance for large files and to reduce the amount of metadata maintained by the system. Also, large *chunk* size increases the likelihood that multiple operations will be directed to the same *chunk*; thus it reduces the number of requests to locate the *chunk* and also allows applications to maintain a persistent network connection with the server where the *chunk* is located.

A *chunk* is then divided in 64KB blocks and each block holds a 32bit checksum. Each *chunk* is stored in a *UFS* and is replicated on a configurable amount of sites (default is 3 replicas). At the time of creation, each *chunk* is given a unique *chunk* handle.

GFS architecture The following image shows the typical architecture of a *GFS* implementation. Each *master* maintains state information about all system components and control a number of *chunk servers*. A *chunk server* is basically a Linux system and uses metadata provided by its *master* to communicate directly with the applications.

Note. Data and control flows have been graphically separated.

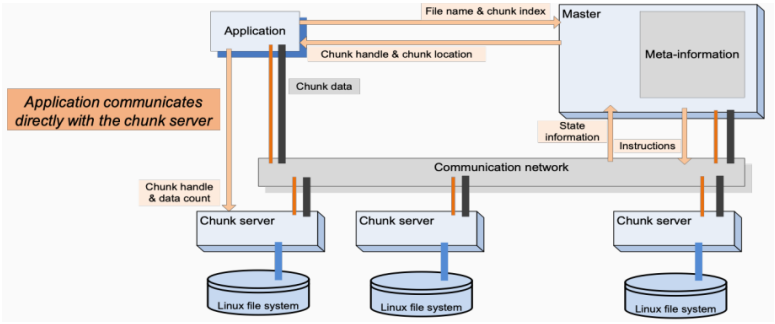


Image 5.6: *GFS* architecture

Steps for a write request Let's see the steps required to complete a generic write operation. First, the client contacts the *master* which replies with the location of the primary and secondary replicas, that is, the *master* replies with the ID of the chuck servers that hold the required *chunk*.

The *master* will also assign a lease to the primary replica. That server will have an exclusive permission to read, write and modify data of a particulare *chunk* without fear of another server changing it. When the lease expires[1]the *master* must reassign the lease to another server or the same server, depending on the needs of the system. This helps to ensure data integrity and prevents data corruption.

Then, the client sends data to all *chunk servers* holding replicas, starting from the closest to the furthest, and each one of them puts changes, aka modifications, in a least recently used buffer and sends an acknowledgement back to the client. Once the client has received all the expected acknowledgements, it sends a write request to the primary replica.

That server applies modifications, then sends write requests to all secondaries, which also apply modifications and send acknowledgements at the end. When the primary replica has received all the expected acknowledgements, it sends a final acknowledgement to the client.

Clarification about primary replicas and lease assignment The distinction between primary and secondary replicas exists just to decide which one will communicate with the client. So, the choice might change over time and is made by the *master* based on what *chunk server* is closer to the client.

When the *master* receives a request it first checks if any of the *chunk servers* already holds a lease. If any, that server will be the primary replica even if it isn't the closest to the client.

Steps for a read request A read request is simpler than a write one. As before, the client sends a read request to the *master*, which replies with primary and secondary replicas IDs. The client can then proced to contact the primary replica which responds with the required data.

5.4 Locks and consensus

Operating systems use *lock managers* to organize and serialize access to resources. *Locks* enable controlled access to shared storage and ensure atomicity of read and write operations, and sometimes, they can even provide reliable storage for loosely-coupled distributed systems.

The *lock manager*, or *leader*, can be fixed or change over time. In the latter case, the decision can be made in a sort of "democratic" way by reaching *consensus*. But before diving into that, let's talk about *locks*.

[1]The lease time is usually aroung 60 seconds

Locks categories *Locks* can be categorized by their effect and their duration. Based on the effect there are two possibilities:

- *Adivisory locks*: processes that don't hold the *lock* can still access the locked resouces by circumventing the locking mechanism;
- *Mandatory locks*: proceses that don't hold the *lock* can never access the locked object;

Based on time there are again two possibilities:

- *Fine-grained locks*: *locks* can be held for a short time. This allows processes to access locked resources more often, but generates more workload for the *lock manager*. Also, if the *leader* fails, more processes are affected;
- *Coarse-grained locks*: *locks* are held for a longer time;

Locks management systems Locking management can be delegated to clients which need to implement their own *consensus algorithm* and provide by themselves a library of functions. This is cumbersome, prone to errors, and must not be dependent on any other server.

A better approch is to implement a *locking service* and provide a library linked with a client application to support service calls. The most widely used *consensus algorithm* is the *Asynchronous PAXOS algorithm*. This approch allows to easily maintain both existing program structures and communication patterns. Then, *lock services* ca be replicated to achive high-availability (i.e. quorums), but even a single client can obtain *lock* and make progress safely. This reduces the number of servers needed for a realiable client system to make progress, but scalability might be an issue.

5.4.1 Chubby

Chubby is a *lock service* devoloped by Google and designed to be used within a loosely-coupled distributed system. So, it is thought for contexts in which there are many machines (e.g. 10K) connected by a high-speed network, and when there's a reliable, but low volume storage.

Chubby uses *coarse-grained* and *advisory locks*. Its interface allows reading and writing on whole files (no seek operation included) and notification of events (to avoid polling). As for *consensus algorithm*, *Chubby* uses the already mentioned *Asynchronous PAXOS*, with lease timers to ensure liveness.

Chubby architecture The structure is made up of two main components which communicate through remote procedure calls: a replica server and a library linked against client applications.

Each set of replica servers (typically 5) creates a *Chubby cell*. Inside each *cell*, replicas use *PAXOS* to elect a *leader* which will carry alone every read or write request. Of course, every modification is replicated among all the other replicas. If the *leader* fails, the other replicas will elect another leader when their master lease expires.

Clients can find the *leader* by sending master (leader) location requests to the replicas listed in the DNS. Non-master replicas will respond with the identity of the actual *master*, and once a client has located a *master*, it redirects all requests to it until it either ceases to respond or indicates it is no longer the *master*.

Clients communicate with the *master* using remote procedure calls provided by a library. When a *master* receives a write request, it propagates it to other replicas in the *cell* and responds to the client when the request has been accepted by a majority of the replicas. On read requests instead, the *master* can respond directly.

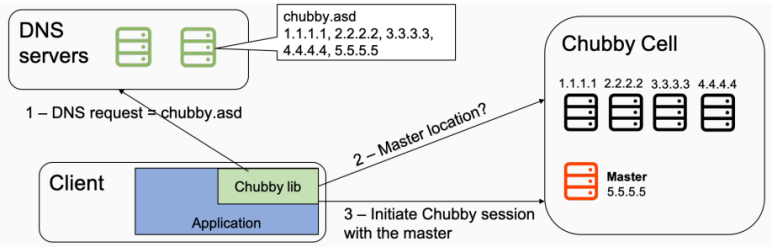


Image 5.7: *Chubby* architecture

Chubby file system *Chubby* exposes a file system interface simpler than that of *UFS*. The file system is a tree of files and directories with name components separated by /. Each directory containes a list of files and directories which are collectivelly called nodes. Each file containes a sequent of uninterpreted bytes.

The interface is simpler because there are no symbolic nor hard links, no file move, no directory times for modifications or last access, and no path-dependant permission, meaning that permissions of a file are handled by the file itself.

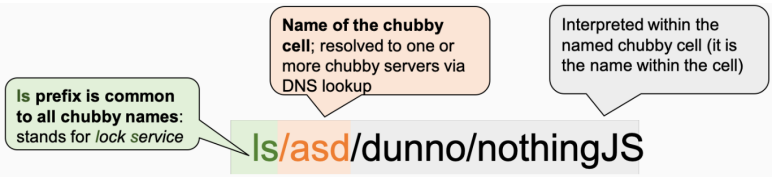


Image 5.8: *Chubby* file path

Each node can be permanent (e.g. actual files and directories) or ephemeral (e.g. temporary files) and its metadata includes three access control list used to control read, write operations and modification of the lists themselves. By opening a node, a client can obtain a handle, which is like a file descriptor in Unix, and can use it to access node internal data.

Chubby locking system Each node can act as an *advisory reader/writer lock*. So, if an application wants to get access over a shared resource and operate on it, it has to acquire a *lock*. In the case of multiple clients trying to get access to the same resource, the first one that gets the *lock* becomes the one that can use it.

As previously mentioned, *locks* are mapped to nodes and are of two kinds:

- *Exclusive mode* (*writer*): one client can hold the *lock*;
- *Shared mode* (*reader*): any number of clients can hold the *lock*;

To acquire either mode, clients must have write permission on that node. Both modes are implemented using two other kinds of *locks*:

- *Sequencer*: is used to ensure that the access order to a shared resource remains consistent. When a client acquires this *lock*, it is able to access the resource in a specific order. It also holds information about state of the *lock*;
- *Time delay*: is used to ensure that any client attempting to access the resource while a *lock* is in place will have to wait until that *lock* is released;

When using *shared mode*, clients can request access to a resource in either read or write mode. In read mode, multiple clients can access the resource concurrently, while in write mode only one client can do it.

Chubby APIs: summary So, when using *Chubby* it is possible to:

- Open/close/delete nodes;
- Read/write full content of a node;
- Set access control lists;
- Acquire/release *locks*;
- Set/get/check sequencer;

5.5 Distributed database: Google Big Table

Google Bit Table (*GBT*) is a distributed storage system for managing structured data. It's designed to scale, allowing it to handle petabytes of data across thousands of servers. It is not a relational database, but a sparse, distributed, persistent multi-dimensional sorted map (key/value store).

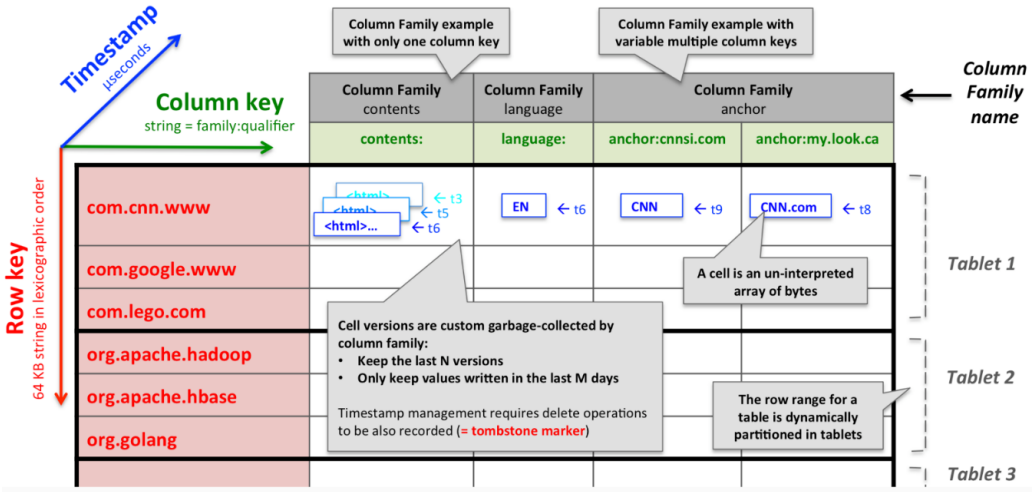
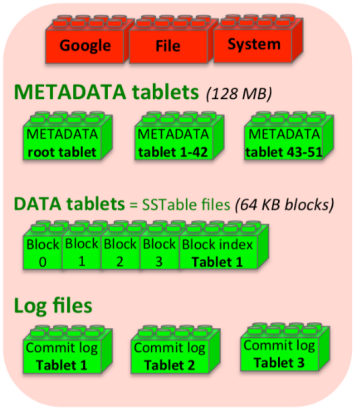


Image 5.9: *GBT data model*

5.5.1 How is GBT built?



GBT uses *GFS* to store data, metadata and *tablet logs*. Data are organized in *tablets* based on the *SSTable* format. For each *data tablet* there's a *log tablet*, and metadata of each *data tablet* are grouped in a collection of metadata *tablets*.

SSTable file format *SSTable* (*Sorted String Table*) is a file format that stores data in a persistent and ordered immutable key-value map. Each *SSTable* file contains a sequence of blocks whose size is typically 64KB.

Optionally, blocks can be mapped to memory, so that lookup and scan operatios can be performed without interacting with the disk.

At the end of the file there's a block index that's used to locate other blocks. It is loaded in memory when the *SSTable* is opened and this allows to avoid unnecessary disk interation. In

fact, when looking for something, the appropriate block can be found by performing a binary search in the in-memory index e then loading just the correct block from the disk.

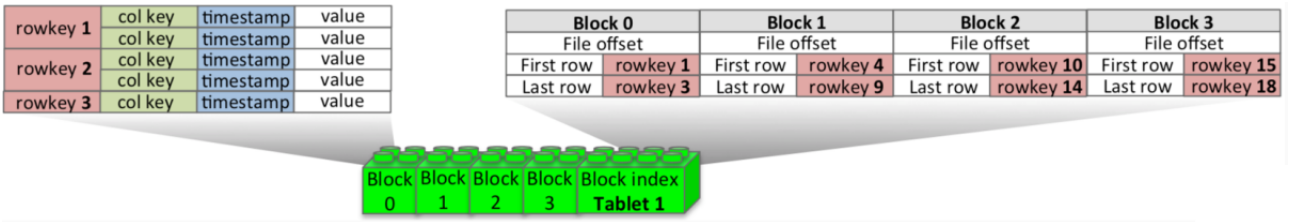
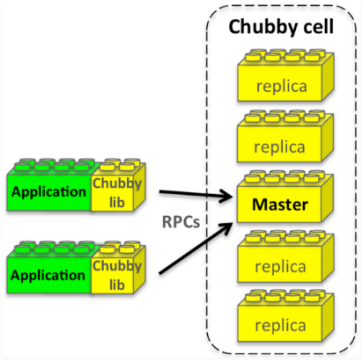


Image 5.10: *SSTable* design

Chubby as an interface for clients *GBT* uses *Chubby* as a sort of interface for client applications. In particular, since *Chubby* provides a reliable namespace that contains directories and small files, it is used to store bootstrap location of *bigtable* data (aka *root tablet*), to discover *tablet servers*, and finalize their death when necessary, and to store *bigtable* schema information (i.e. column family information for each *tablet*).
In addition to that, *Chubby* APIs allows for access control list management, and its design guarantees that there's at most one active *master* at any time in any *cell*. Since *Chubby* provides the interface to the users, if it's unavailable, *GBT* will also be unreachable.



5.5.2 How is GBT implemented?

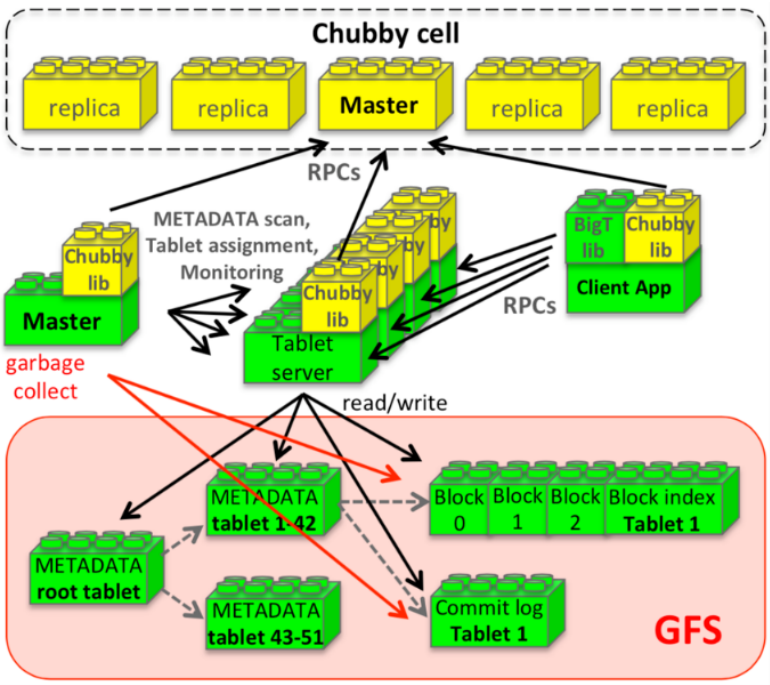


Image 5.11: *GBT* architecture

The above image shows the actual architecture of *GBT*. There's one *master server* that handles:

- Assigning *tablets* to *tablet servers*;
- Detecting the addition and expiration of *tablet servers*;

- Balancing *tablet servers* load;
- Garbage collection of files in *GFS*;
- Handling schema changes (*tablet* creation, column family creation/deletion);
- providing metadata to the *Chubby master*;

Then, there are many *tablet servers* and each of them is responsible for managing a set of *tablets*, handling read and write requests and even splitting *tablets* that have grown too large (usually around 100/200MB).

Finally, clients contact the *Chubby master*, that responds with the identity of the *tablet server* that holds the required data, and use that information to communicate directly with that server, bypassing the *GBT master*.

The API provided by *GBT* is the following:

- Add/remove *tablet servers*;
- Create/delete *tablets*;
- Create/delete column families;
- Control *tablet flags* (access control rights and metadata);
- Control columns family flags (access control rights and metadata);
- Cell value: `put(rowkey, columnkey, value)`, `get/delete(rowkey, columnkey)`;
- Lookup value for row: `has(rowkey, columnfamily)`;
- Lookup value from *tablet*: `scan(rowfilter, columnfilter, timestampfilter)`;
- Operations on a single row: `transactions(atomic read-modify-write sequence)`;

5.5.3 GBT operations

Tablet location

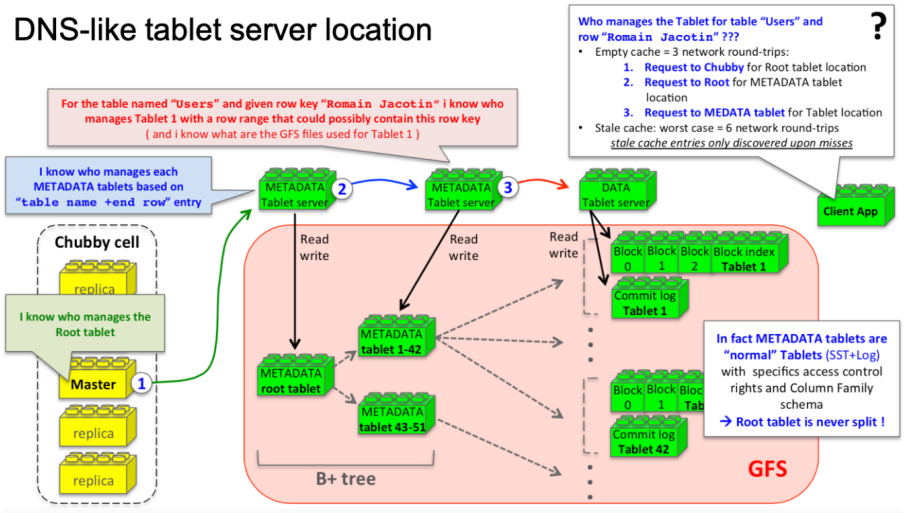


Image 5.12: *Tablet* location procedure

Tablet assignment Each *tablet* is assigned to one *tablet server* at a time. *Master* keeps tracks of the set of live *tablet servers* (via *Chubby*), current assignement of *tablets* to *tablet servers* and current unassigned *tablets*. When a *tablet* is unassigned, the master assigns it to an available *tablet server* by sending a *tablet* load request.

Tablet server discovery When a *tablet server* starts, it creates and acquires an *exclusive lock* on a uniquely-named file in a specific *Chubby* directory (`servers` directory) and the *master* monitors this directory to discover *tablet servers*. A *tablet server* stops serving its *tablets* if it loses its *exclusive Chubby lock*, and if the *Chubby* file no longer exists, then the *tablet server* will never be able to serve again, so it kills itself.

Tablet server monitoring The *master* is responsible for detecting when a *tablet server* is no longer serving its *tablets*, and for reassigning those *tablets*, and it does so by periodically asking each *tablet server* for the status of its *lock*. If a *tablet server* reports that it has lost its *lock*, or if the *master* was unable to reach a server during its last attempt, the *master* attempts to acquire the *lock* for the *Chubby* file.

If it succedes, then *Chubby* is live and the *tablet server* is dead or isolated, so the *master* deletes its server file to ensure that the *tablet server* can never serve again. Then, the *master* can put all the *tablets* that were previously assigned to that *tablet server* into the set of unassigned *tablets*.

Master startup When a *master* is started by the cluster management system, it needs to discover the current *tablet* assignments before it can change them. This is a five-steps procedure:

1. The *master* grabs a *unique master lock* in *Chubby* to prevent concurrent *master* instantiations;
2. It scans the `servers` directory in *Chubby* to find all live *tablet servers*;
3. It communicates with every live *tablet server* to discover what *tablets* are already assigned to each server;
4. It adds the *root tablet* to the set of *unassigned tablets*, if an assignment for it wasn't previuosly discovered;
5. Finally, the *master* scans the *METADATA tablet* to learn the complete set of *tablets* and eventually detect unassigned ones;

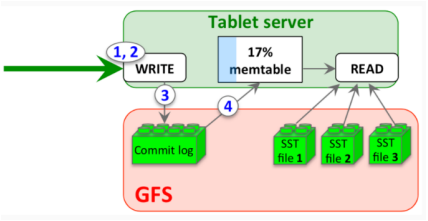
Master isolated To ensure that a *bigtable* cluster is not vulnerable to networking issues between the *master* and *Chubby*, the *master* kills itself if its *Chubby* session expires (*master* failures do not change the assignment of *tablets* to *tablet servers*).

Tablet merging/splitting The set of existing *tablets* only changes when a *tablet* is created or deleted. Two existing *tablets* can be merged to form a larger one, and a large one can be split in two smaller *tablets*. *Tablets* merging must be initiated by the *master* while *tablet* splitting can be initiated by *tablet servers*. In this case, the *tablet server* that manages the *tablet* to split, commits the split by recording information for the new *tablet* in the *METADATA tablet*. After that, the server notifies the *master*.

Tablet serving

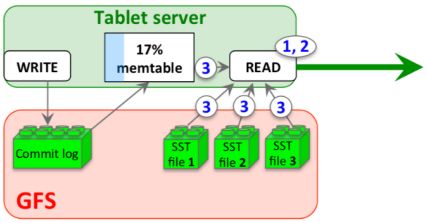
Steps for a write operation:

1. *Tablet server* checks if the request is well-formed;
2. *Tablet server* checks if sender is authorized to write by looking up the list of permitted writers in a *Chubby* file;
3. The valid mutation is written to commit log that stores redo records (commits can be grouped to increase throughtput);
4. After mutation is committed, its content is inserted into memtable (i.e. in a memory sorted buffer);



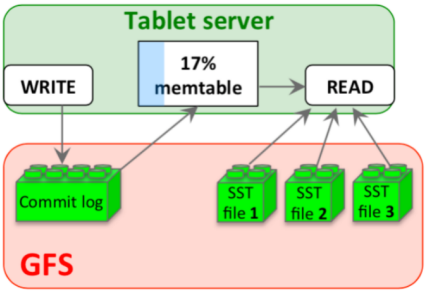
Steps for a read operation:

1. *Tablet server* checks if the request is well-formed;
2. *Tablet server* checks if sender is authorized to read by looking up the list of permitted readers in a *Chubby* file;
3. The valid read operation is executed on a merged view of the sequence of *SSTables* and the memtable;



Steps for a *tablet* recovery operation:

1. *Tablet server* reads its metadata from the *METADATA table* (lists of *SSTables* that comprise a *tablet* and a set of a redo points, which are pointers into any commit logs that may contain data for the *tablet*);
2. *Tablet server* reads the indices of the *SSTables* into memory and reconstructs the memtable by applying all the updates that have been committed since the redo points;



Tablet minor compaction When memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to a new *SSTable* and written to *GFS*. The goals of this operation are two: shrink the memory usage of the *tablet server* and reduce the amount of data that has to be read from the commit log during a recovery.

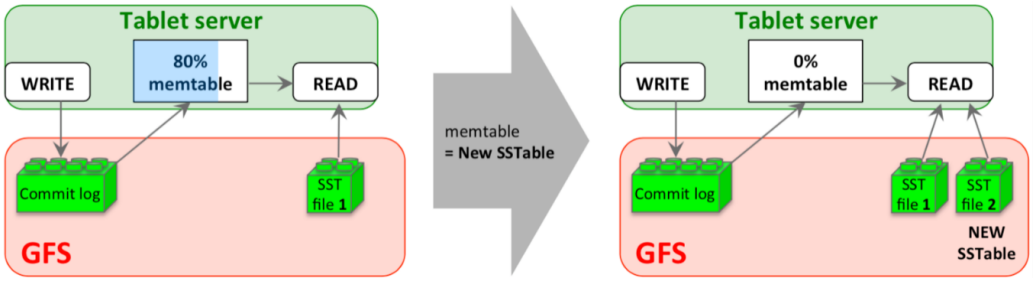


Image 5.13: *Tablet minor compaction*

Tablet merging compaction The problem of *tablet minor compaction* is that every *minor compaction* creates a new *SSTable* and in the long run there might be an inconveniently large amount of *SSTables*. The solution to this is doing a periodic merging of a few *SSTables* and the memtable.

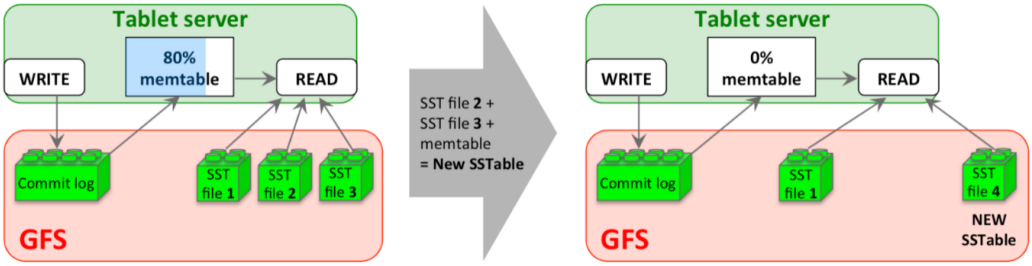


Image 5.14: *Tablet merging compaction*

Tablet major compactor It is a *merging compaction* that rewrites all *SSTables* into exactly one *SSTable* that contains no deletion information or deleted data. *Bigtable* cycles throught all of its *tablets* and regularly applies *major compaction* to them, that is to say, that it reclaims resources used by deleted data in a timely fashion.

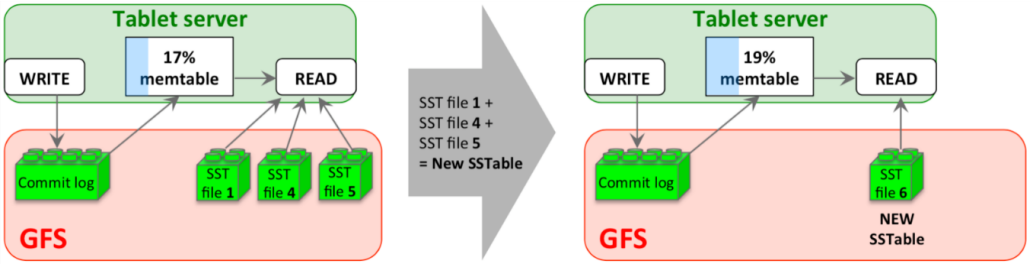


Image 5.15: *Tablet major compaction*

5.6 Distributed object storage: OpenStack Swift

OpenStack Swift (*OSS*) is a distributed object storage system designed to scale from a single machine to thousands of servers. It's optimized for *multi-tenancy* and high concurrency. It's ideal for backups, web and mobile content, and any other unstructured data that can grow without bound. Among the features of *OSS* there are consistency, because every access to an item will return its most recent value, high availability, possibility to rely on commodity hardware, authentication procedures and possibility the possibility to set up quotas, access control lists and various storage policies.

Since this is not a distributed file system, it's impossible to mount it, create a file hierarchy, create a file system from a certain state of the service, and even store objects which exedes a certain size (but this might not always be the case).

OSS hierarchycal structure Despite not allowing file hierarchies, *OSS* is itself organized in a hierarchical way. Specifically, it is made up of three levels:

1. *Account*: is the top level and is associated to every service user. It defines a namespace in which the owner can create and manage its own *containers*;

2. *Container*: is a namespace for *objects* and can be thought as a folder. Access control list, along with storage policies can be associated to each *container*;

3. *Object*: represent the actual data and is handled as an uncompressed and unencrypted format with metadata;

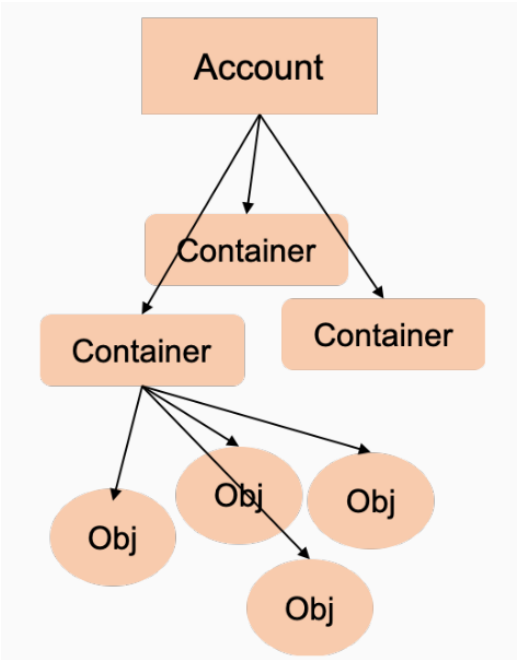


Image 5.16: *OSS* hierarchy

5.6.1 OSS architecture

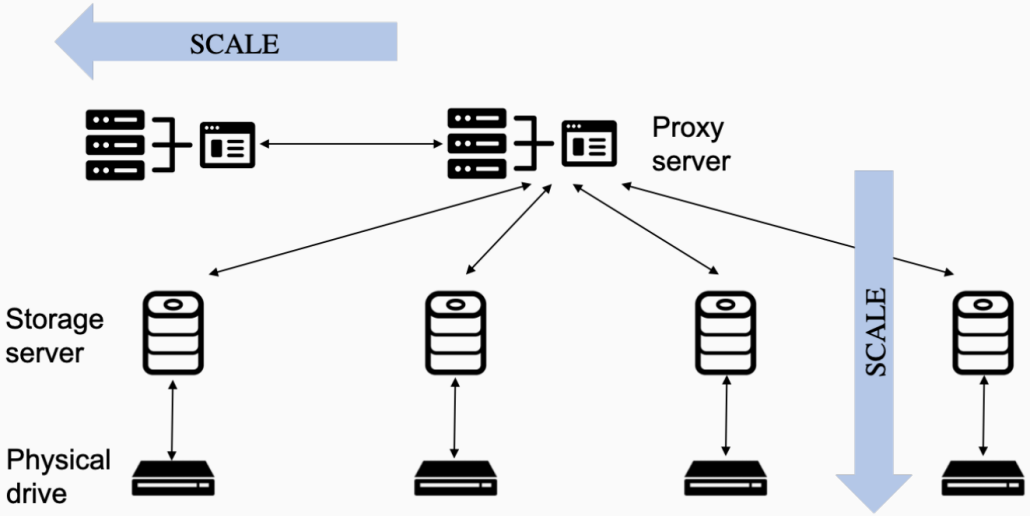


Image 5.17: *OSS* architecture

The *proxy server* is responible for keeping together the architecture. For each request it receives, it looks up the location of the *account*, *container* or *objet* in the *ring*, and routes the request accordingly. More precisely, the *proxy server* serves as the interface to the users. It's responsible to implement the *OSS* APIs, accept and answer to user requests, handle failures and make sure that the communication with *storage servers* is up and working.

Then, there are two more categories of server:

- *Storage servers*:
  - *Object servers*: servers that store, retrive and delete *objects* stored in their local devices;
  - *Container servers*: servers that handle listings of *objects*. They don't know where the *objects* are, but to what *container* they belong. Each server keeps the listing in a replicated SQLite database together with statistics about the *containers* it controls;
  - *Account servers*: they're the same as *container servers*, but keep listings of *containers* instead of *objects*;
- *Consistency servers*:
  - *Replicator*: keeps the system in a consistent state in case of temporary error conditions (e.g. drive failures or netwok outages);
  - *Updater*: when *containers* or *accounts* data cannot be immediately updated, in the case of conjestion or failures, the updates are queued locally on the file system and carried out by the *updater* when possible;
  - *Auditor*: scans local servers checking for integrity of *objects*, *containers* and *accounts*. All the errors are logged and if a corrupted file is found, that file is quarantined and will be replaced by the *replicator* with a correct replica;

OSS API Here's a list of some funtions exposed by *OSS* APIs:

- `GET`: download *objects*, list content of *containers* or *accounts*;
- `PUT`: uploads *objects*, creates *containers*, overwrites metadata headers;
- `POST`: creates *containers* if they do not exist, updates metadata (*accounts* or *containers*) and overwrites metadata (*objects*);
- `DELETE`: deletes empty *objects* and empty *containers*;
- `HEAD`: retrieves header information for *accounts*, *containers* or *objects*;

Note. *OSS* provides a REST API.

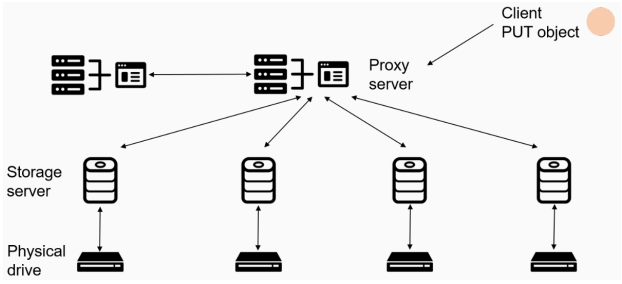
OSS rings A *ring* is a data structure in which each node is connected to its neighbours forming a loop. Each node in the *ring* is a *storage server*. *Rings* are managed by a *ring component* and is used to determine where data should be stored and even redistribute data evenly aroung nodes, according to weights based on the available capacity of each node.

A *ring* keeps a mapping of the nodes using availability zones, partitions and replicas. Each partition is replicated across the cluster (group of *storage servers* interconnected in a *ring*) according to the storage policies (by default 3 replicas per partition). The location of each partition is also stored in the mapping maintained by the *ring*. In fact, each node in the *ring* is assigned a certain number of partions, and each partition is responsible for storing a certain amount of data.

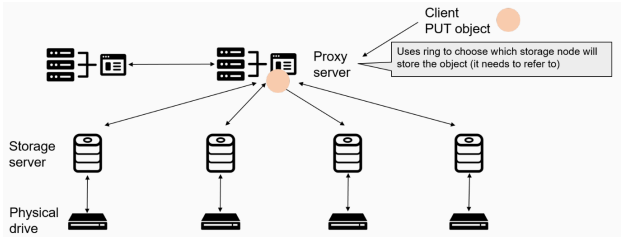
Replicas inside each partition are isolated in as many distinct regions, zones, servers and devices as possible, so as to reduce failure domains. Regions are based on their geographical location (e.g. West Europe), while zones are more abstract (e.g. city, power separtion, network separation, or any other attribute that would lessen multiple replicas being available at the same time). There might be less failure domains than replicas of a partition.

Example 6 - Steps for a PUT request.

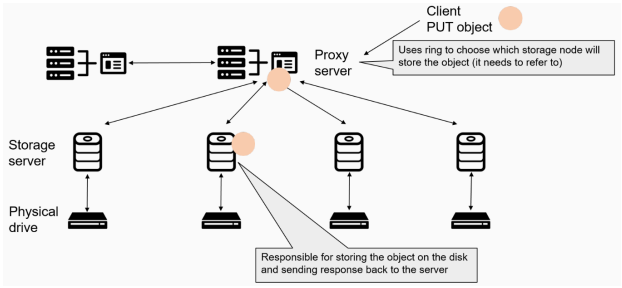
Let's see the steps needed for a simple `PUT` *request.*



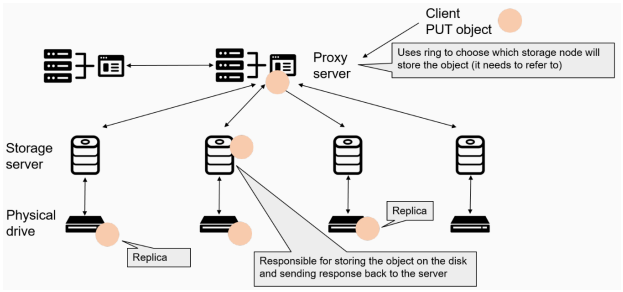
(a) Client sends the request



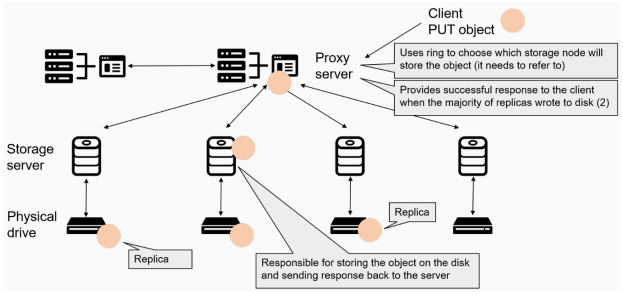
(b) *Proxy* receives the request and begins processing it



(c) A *storage server* is chosen as responsible for storing



(d) The server and its neighbours store data



(e) The *proxy* responds to the client

If the storage server that's handling the data has a failed drive, data are passed to another server. For example, it might be passed to the fourth server in the image: