Cloud Computing

Prof: Carlo Vallati

Course Layout:

1. Module I
    1. General Intro and Foundations Concepts (8 Hours)
    2. Virtualization Technologies (12 Hours)
    3. Cloud Applications (12 Hours)
    4. Cloud Platforms (18 Hours)
    5. Cloud Infrastructure and Mechanisms (10 Hours)
2. Module II
    1. Cloud Programming Models (30 Hours)

Introduction and Foundations Concepts 3
A01 Evolution towards Cloud Computing 3
A02 Technology foundation concepts 5
A03 Benefit analysis and business models examples 8
A04 Cloud NIST model, service types and deployment models 10
Virtualization Technologies 13
B01 Introduction to virtualization and basic concepts 13
B02 Multiprogramming recap & Full Virtualization 17
Full Virtualization Techniques - trap and emulate approach 19
B03 Hardware assisted virtualization 24
Lab 1 - B04 (Lab infrastructure, VPN, Full Virtualization) 31
B05 Paravirtualization and Operating System Level Virtualization 32
Paravirtualization 32
Operating System Virtualization 33
Lab 2 - B06 Docker 35
Cloud Applications 36
C01 Cloud applications - structure and architecture 36
C02 Cloud applications - design methodologies 39
Lab3 - C03 REST interfaces 43
C04 Scalable Cloud application design: Message oriented distributed systems and data replication 44
Lab4 - C05 Message Queue Systems (RabbitMQ) 50
C06: Geographically distributed applications: large-scale load balancing and eventually consistent data replication strategies 51
Cloud Platforms 55
D01-D02 Cloud computing platforms 55
OpenStack 55
Lab5 - D03 OpenStack installation using JuJu 65
D05 Lightweight cloud computing platforms for DevOps 66
Lab6 - D06 Kubernetes installation 70
Lab7 - D08 Cloud Platform Kubernetes Operations 70
Cloud Storage 72
E01 Cloud Storage and Distributed File System 72
Storage Model on Single Server 72
Distributed File Systems (DFS) 73
Appendix 78
SOAP VS REST 78
OpenStack VS Kubernetes 78

Introduction and Foundations Concepts

A01 Evolution towards Cloud Computing

Official definition by NIST of cloud computing:

"Cloud Computing is a model for enabling ***ubiquitous, convenient, on-demand*** *network access to a* ***shared pool of configurable computing resources*** *that can be rapidly provisioned and released with minimal management effort of service provider interactions"*

In 1960 McCarthy (who also coined the term Artificial Intelligence) wrote that "*Computation may someday be organized as a* ***public utility***" (e.g. water, gas, electricity).
It took, however, more than 40 years for technology to develop and mature for cloud computing: only the composition of some key technologies has enabled cloud computing.

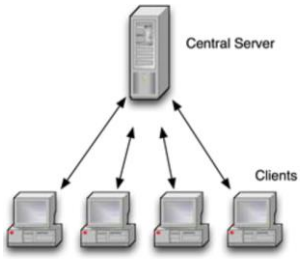
History

Mainframes: in the 1970s were the reference computing model; used mostly to automate basic data processing (e.g. payroll management); there were many "dumb" terminals connected to a mainframe (fully centralized processing environment). This centralized processing environment was a bottleneck → high queueing time for the users.

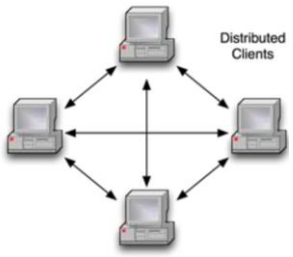
Personal Computers: in the late 70s, Personal Computers (PCs) provided processing and storage capabilities to each user. Soon replaced mainframes.

Network of PCs: introduced to allow communications between PCs of the same organization (e.g. data transfer). Each one, however, was functioning independently. Local Area Network (LAN) and Wide Area Network (WAN) were introduced during that period so that computers within the same office or located at a distance could communicate with each other.

Distributed Applications: client-server computing & peer-to-peer computing (data exchange was limited due to limits in network bandwidths).



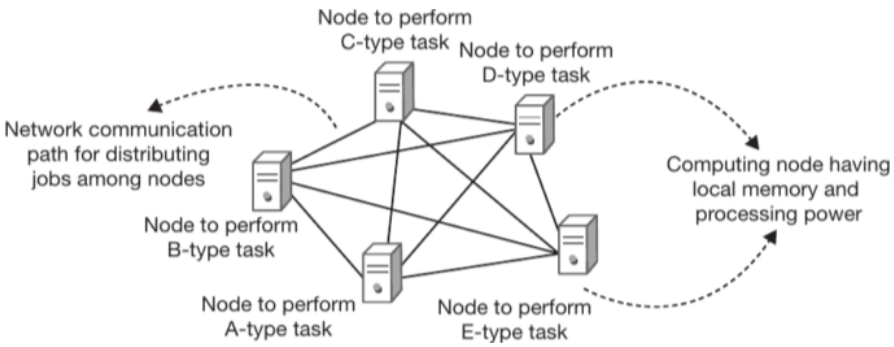
Client / Server



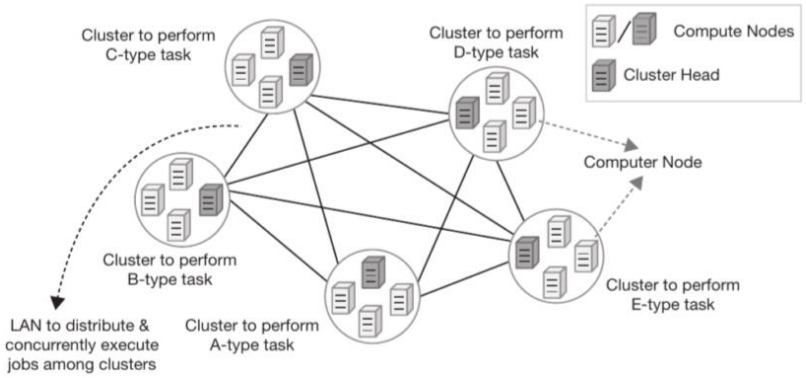
Peer to Peer

Parallel Processing: in the 80s changed the belief idea that computing performance can be improved only by scaling vertically (by introducing higher performance hardware); Parallel processing introduced the notion that multiple processors, whether located on the same PC or different PCs, could collaborate to solve a single large task.
The task has to be designed to be parallel (e.g. decomposed in a set of subtasks).

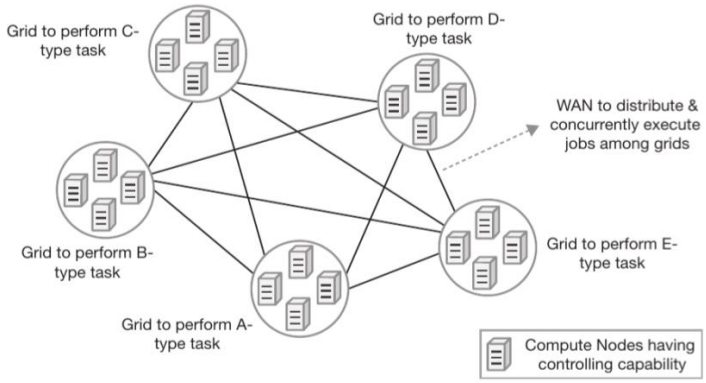
Distributed Computing Systems: systems in which applications are designed as a set of subtasks that are run on a set of nodes (/systems). Every system has its own resources. Nodes communicate to exchange data or results.
This first implementation was subjected to faults because each node was a single point of failure.



then **Cluster Computing** was introduced: groups of multiple nodes (computers) all connected to the same LAN to perform the same (or similar) tasks (less overhead, higher reliability). In this setup, if one node goes offline, there are other nodes capable of performing the same type of task. However, in distributed computing, if a node crashes, tasks of a specific type that were supposed to be executed by that node can no longer be carried out.
Each cluster had its own cluster head in charge of controlling the cluster and distributing the load.



Grid Computing: in cluster computing, each cluster head represented a single point of failure, then in the early 90s grid computing introduced the decentralization of the control functionalities from the cluster head.
Nodes of the same cluster can be deployed either on the same or on different LANs



New era begins

Grid computing provided many advantages: a scalable architecture that could offer high performance computing for complex tasks, but it had **many drawbacks**:

- Real-time scaling was not possible (more hardware has to be connected physically)
- Low level of fault tolerance: a node failure determines a task failure
- Heterogeneous hardware requires code adaptation

Hardware virtualization

To answer the Grid Computing problems, hardware virtualization was introduced: decoupling software systems from the underlying hardware allows both to deal with hardware diversity and also to achieve real-time scalability. (more details in the next chapters).

Web-based Technologies

In addition to enabling grid computing to empower the development of large computing systems, faster networks allowed users from different geographical locations to collaborate in almost real time.
The World Wide Web emerged as the key application for disseminating information and facilitating collaboration.

SOA (Service Oriented Architecture): method in which applications are developed by leveraging software components (software services) that interact with each other (modules). Systems using SOA are flexible to changes.

Nowadays, thanks to the evolutions in networking, software development, and hardware, utility computing has become a reality in the form of cloud computing. Crucial enabler for this model was hardware virtualization.

Salesforce introduced the first product based on cloud computing concepts in 1999.
The first large scale cloud computing service was commercialized by Amazon in 2006 that lunched its Elastic Cloud Computing (AWS EC2) and Simple Storage Service (AWS S3). Soon after other big players entered in the market with similar services, e.g. Microsoft Azure in 2009.

A02 Technology foundation concepts

Small recap of the chapter in italian: [ruoli nell'infrastruttura Cloud, i tipi di virtualizzazione, i tipi di scaling (horizontal e vertical), il discorso del pay per use, everything as a service, il discorso del multi tenancy]

Cloud computing is now a buzzword. Historically the cloud term was used to abstract the network in systems diagrams. Used both to refer to cloud services (the applications delivered) and cloud infrastructure (hardware and software in the datacentres that provide those services).

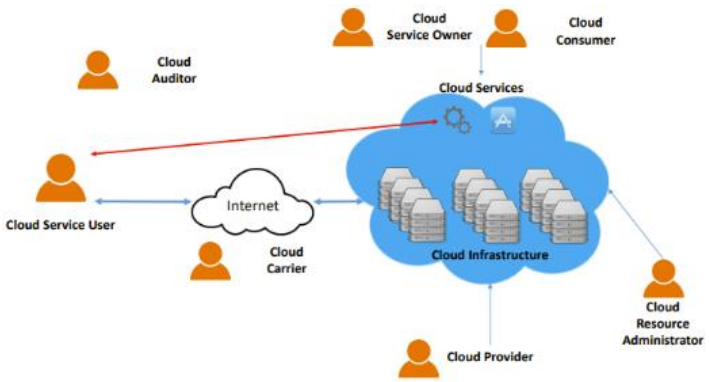
Cloud Computing formal definition: "cloud computing is a distinct IT environment designed for the purpose of remotely providing **scalable and measured** IT resources that are accessible via the Internet".

So cloud computing is an environment to offer IT resources via the internet. **What are those IT resources?** We usually will refer to the three basic resources that are part of any computer: CPU, RAM and hard drives (long term storage). Such resources provided by the Cloud Infrastructure are used to implement Cloud Services.

Cloud services can expose either an application directly accessed by the users (e.g. web pages), or a service exploited by other applications (e.g. local apps on smartphones). In the latter case the cloud service has to expose a set of programming interfaces APIs.

Everything as a Service (XaaS): the idea is to move all the complex tasks that were previously performed within our devices to cloud devices where everything is cheaper and more powerful (e.g. file storage).
The result is a minimal design for the application that basically implements only an interface towards the application logic that is now implemented on the cloud.

Cloud Computing roles



1. **Cloud provider**: the organization that provides cloud-based IT resources. Is responsible for creating and managing the infrastructure. IT resources are made available for lease by cloud consumers.
2. **Cloud Consumer**: the organization (or a person) that has a formal contract with the cloud provider to use its IT resources. (Cloud consumer can sell its service to other customers)
3. **Cloud Service Owner**: person or organization that creates a cloud service running on the resources provided by the cloud infrastructure. Usually the Cloud consumer and Cloud Service Owner are the same person or company.
4. **Cloud Resource Administrator**: responsible for administering cloud-based IT resources. Usually belongs to a cloud provide organizations on, but can belong customers too.
5. **Cloud Auditor**: third-party that conducts assessments on the cloud environment independently. Typically evaluates security and performance. The cloud consumer could ask to the cloud auditor if the resources are provided in compliance with the contract signed.
6. **Cloud Carrier**: provides connectivity between users and cloud provider
7. **Cloud service user**: the final user of the cloud service

For the NIST, there could be another actor: the **Cloud Broker**: is a third party that interposes, in some cases, between the cloud provider and the cloud consumer. Specifically, a cloud broker manages the delivery of cloud services from different providers to the consumer and negotiates the relationship.
Through the service of the broker, consumers (which interacts with the broker and not directly with the providers) can avoid the responsibility of interacting directly with the provider and can benefit from reducing the costs, by requesting services from multiple providers at the same time.

Cloud Model

Everything is based on the resources offered by the cloud provider. As highlighted in the original definition, the delivery of IT resources must be scalable and measured. Then:

- Adopt a **utility-based model**: a pay-per-use price model. Pay what you effectively get. That can be implemented only because the resources provided can be metered.
- **Resources can be provisioned dynamically**: resources must be instantiated in a short amount of time, without the need for human intervention. Resources can be provisioned not only after a human request but also under software orchestration.

Business Model: cloud computing providers sell IT resources; other companies (the cloud consumer) buy those resources to create services that they can use internally or sell to their customers.

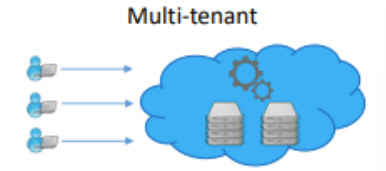
Cloud Computing Infrastructure

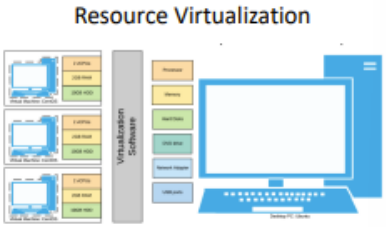
Cloud infrastructure is deployed in datacenters by the cloud provider.
Datacenter: dedicated space to house servers and network equipment. (Computing, storage, networking).
One peculiar feature of the hardware installed in a datacenter is the fact this hardware must be multi-tenant.

Single Tenancy: in a single-tenant architecture, a single instance of software and its underlying infrastructure serve a single customer (tenant). In this model, resources are not shared among multiple customers.

Multi-user

Traditional computing architectures are designed as **multi-user** systems, where a system administrator configures and manages user permissions to access system resources. Initially, the administrator installs and configures the system for all users. Subsequently, multiple users can log into this shared server to access data or run programs. Users acknowledge the shared nature of the system due to imposed limitations, including restricted instructions, the inability to execute privileged commands, and limited access to certain peripherals. Access to all system resources is prohibited for users, and only the system administrator has the authority to execute any command.





As the number of users increases, it becomes challenging for the administrator to address individual user requirements. This multi-user system paradigm was prevalent before the advent of cloud computing, enabling organizations to optimize the utilization of a single server by configuring it as a multi-user system.

Multi-tenancy

A Multi-tenant system, like a multi-user system, is shared among different users with the ultimate goal of maximizing resource utilization and investment. However, in a multi-tenant system, the fundamental concept differs.
In this setup, individuals accessing the system are labeled as "tenants" rather than mere "users." Essentially, each tenant assumes the role of an administrator with the authority to enact configuration changes and decisions. This stands in contrast to a multi-user system with a single administrator.

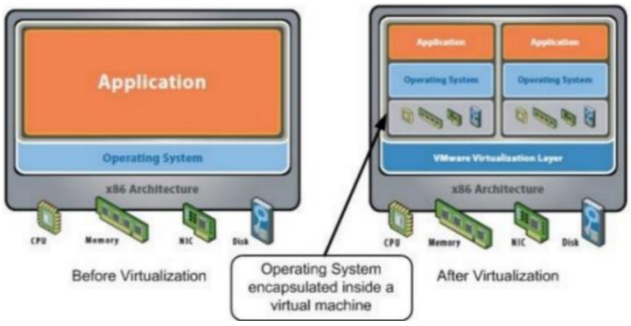
Multi-tenancy provides each cloud consumer (tenant) with the perception and experience of having complete control over the system (or, more precisely, the resources dedicated to them). Indeed, multi-tenancy aims to provide each tenant with the impression of being in control of their allocated resources within the shared system architecture.

Virtualization

Virtualization is a key enabler to create a multi-tenant infrastructure. It is a broad concept, refers to a technique that allows the creation of a virtualized version of something (e.g. IT resources, a server, a program, …).
By using virtualization, multiple virtual copies of the real resources can be created → cloud consumers have access not to the real versions but to the virtualized ones so they have the illusion that they are in complete control of the real resources (multi-tenancy achieved).

Hardware Virtualization **(aka System Level Virtualization)**

It is the most popular virtualization technology used in cloud computing. It allows the creation a virtual representation of all hardware resources of a physical machine (a server creating a virtual server). Provides a complete abstract execution environment on which a full operating system can be run. Exploits different virtualization techniques to virtualize each component of the system. Each virtual environment on which an OS is run is called Virtual Machine.



Hardware Virtualization allows multiple Operating Systems to run on the same hardware. This is possible because they access the virtualized version of each resource.

Hypervisor **(aka Virtual Machine Manager)**: Virtual hardware is managed and controlled by the virtualization layer. At its core, we have the hypervisor, which is software that recreates the virtualized hardware environment in which the **guest** operating system runs. (The real machine is called "**host**").

Different virtualization types available:

- Full virtualization
- Para-virtualization
- Operating System virtualization

Those approaches differ for level of abstraction, which results in different amount of resources usage overhead for its implementation, in terms of (real) resources employed to virtualize the resources.

Cloud Virtualized Infrastructure

In the datacenter many servers host many VMs each, and they are interconnected via LAN so VMs can communicate with external hosts (to expose its cloud services) or can communicate with other VMs (to implement complex services). Each server runs a hypervisor for the execution of VMs. VMs are created dynamically, to handle changing conditions/loads. Dynamic provisioning requirement is achieved via two scaling mechanisms:

- **Vertical scaling**: replace an IT resource with another that has higher (or lower) capacity. (or change the configuration of the same resource to offer more (or less) capacity).
  Scale up / scale down → less adopted, requires downtimes for the replacement
- **Horizontal Scaling**: allocating or releasing a certain number of IT resources, all equals each other.
  Scale out / scale down → most popular scaling mechanism in cloud computing

Metering: virtualization allows effective metering of computing resources. Access to virtualized resources can be easily monitored and measured. Cloud consumers are billed per their use proportionally.

A03 Benefit analysis and business models examples

Small recap of the chapter in italian: [il workflow di un'azienda che fa deploy locale di un servizio o su Cloud, scalability, delegare competenze e responsabilità, delegare rischio acquisto attrezzatura che diventa vecchia, non avere da riparare / sostituire hardware, non dover fare manutenzione sul sistema operativo, non c'è rischio di waste of resources. Non ci sono fixed costs. È un utility market. Higher quality of service, reliability, continuous availability, location independent, minimal software management, companies focused on their business.

Esempi di cloudification: Expedia, AirBnB (ha proprio iniziato già su aws), Pixar (per fare i montaggi è sporadic HPC (HIGH PERFORMANCE COMPUTING)), global services (Google), global data collection (General Electric), global content distribution (King Candy Crush).

Svantaggi: network cost, e network bandwidth (può essere collo di bottiglia per l'accesso a Cloud services), limited portability (vendor lock in), legal issues (geographical position of data), data Security (where data is stored, transmitted over the unsafe public network)]

Traditional Computing Deployment steps

A business that wants to deploy a service without using the cloud (/ before the cloud) had to:

- Buy and install hardware.
- Install and configure the operating systems on the servers, configure the network.
- Install software dependencies and deploy the service.
- Keep the system up and running by performing the following steps:
  - Repair / substitute faulty / outdated hardware
  - Update the OS and apply security patches.
  - Maintain the software environment.

In traditional computing, if the **business has to scale** (because of demand increase, that can even happen overnight) its resources it has to scale up the hardware and the hardware has to be powerful enough to manage the peaks of requests → need to buy more powerful hardware.
In case of **business shrinks** (temporary or permanently), the more powerful hardware resources will be **wasted**.

Cloud Computing benefits

In cloud computing the above problems of configuration and maintenance are demanded by the cloud provider, while scaling up and down can be performed without the risk of wasting resources or buying not enough resources: pay per use approach → cloud computing facilitates dynamic scalability and reduce costs both initial and operational.

- **High scalability**: Dynamic provisioning for resources can be done automatically using software automation (based on current demand requirements), allowing the cloud service to scale dynamically.
- **No Fixed Costs**: you do not have to own an infrastructure, if you want to close your service you can without penalties. No initial investment, no need for internal specialized personnel for infrastructure management.

Cloud Computing providers offer resources at very low cost by creating a large infrastructure where costs are shared (high quantity of hardware bought → lower prices; full exploitation of resources → low waste; know how and personnel dedicated to managing the cloud infrastructure is in charge of maintaining the whole infrastructure → economy of scale).

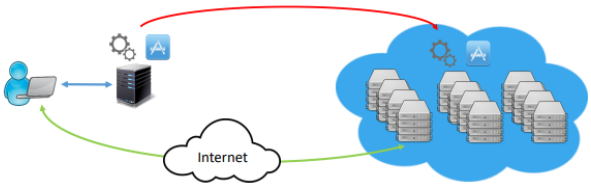
In summary, the advantages of cloud computing are similar to those of utility systems: “would you rather build your own power plant or plug into the power grid?”

Other indirect advantages:

- **Minimal management responsibilities**: reduces the risks associated with managing an IT infrastructure (even legal).
- **Higher quality of service**: the team of the cloud infrastructure is normally more expert and fully dedicated to the cloud infrastructure and QoS, while in traditional computing those teams were shared and less specialized.
- **Reliability**: the dedicated team and infrastructure grant access to the newest techniques of management of the infrastructure, like load balancing, backup management and recovery procedures. Cloud providers ensure an environment that is safe from failures and disasters.
- **Continuous Availability**: 24 x 7 service availability is ensured by employing state of the art redundancy mechanisms.
- **Minimal Software Management**: some kinds of cloud services take care of the software management too (licensing in charge of the cloud provider, updates and patches too).
- **Location independent**: accessed via internet, then are available from everywhere.
- **Companies can focus on their business**: system developers can focus on developing business logic rather than maintaining the infrastructure.

Cloudification

The benefits of cloud computing are driving a trend known as "cloudification". This term refers to the process of *moving applications and services from local computing deployments to the cloud*. Local infrastructure is dismantled to move services and applications to the cloud.



Examples of companies that moved to the cloud:

- **Expedia**: moved to AWS in 2017 → reduced infrastructure costs, minimized response latency from 700 to 50 ms.
- **AirBnB**: as a startup in 2008 started its infrastructure on AWS. This allowed it to handle its growth without problems.

Sporadic High-Performance Computing (sporadic HPC) needs: companies with sporadic HPC needs are companies like Pixar, which need huge amounts of computing power for short periods. (e.g. Pixar needs it to render a movie). Sporadic HPC companies can rent computing power for short time ranges so that they do not have to buy their infrastructure → rendering a computer animated movie for Pixar on a single machine could take between 100 and 1000 years.

Global-Scale services: are services that have to be access from all over the world to grant reachability and low latencies, the service can not be exposed from a single location (in order to avoid bottlenecks). There is a need to deploy different IT infrastructures in different locations. → cloud providers deploy their infrastructure all over the world to cover it efficiently. They offer ad-hoc solutions to replicate cloud services on different data centers.

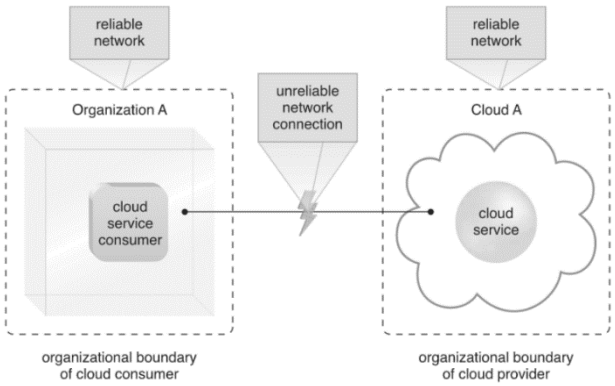
Data Collection services: in order to handle the ingestion of data from a network of sensors that is spread all over the world, cloud computing services can be exploited. (e.g. General Electric is this use-case)

Global-scale content distribution: content could be of various forms (video, music, web-pages, software updates, …). Each type has its own latency and bandwidth requirements. In this scenario, cloud providers can be leveraged for global content distribution to achieve optimal and scalable global reach. (e.g. the company King, known for mobile games like Candy Crush, serves as an example of this).

Challenges/risks

The cloud approach does present some disadvantages when compared with the traditional approach. These drawbacks pose **challenges** for cloud providers in supporting a broader range of use cases and also introduce **risks** for cloud consumers.

- **Network cost and bandwidth**: cloud providers and cloud users have to be able to access to network continuously and with enough bandwidth.
- **Limited portability**: cloud computing standardization is still limited, there is no well-recognized standard. Different cloud computing providers have different solutions that are proprietary. A service developed to run on the infrastructure of a cloud provider can hardly run on a different infrastructure. This limits the portability of cloud applications.
- **Legal Issues**: data centers are placed in a location of economic convenience for the cloud provider, but not always the cloud storage location is the same as the cloud consumer, and this could lead to privacy rules compliance problems. (e.g. some countries require that sensitive data (e.g. medical data) have to be stored in the same country of users)
- **Data security**: to reach the cloud infrastructure (usually considered trusted), data has to be transmitted over the internet (unreliable / untrusted) → many trust boundaries are involved.
  Connection security mechanisms are required; Cloud providers must ensure and grant the security of users' data stored in their infrastructure; users do not have control over cloud governance.

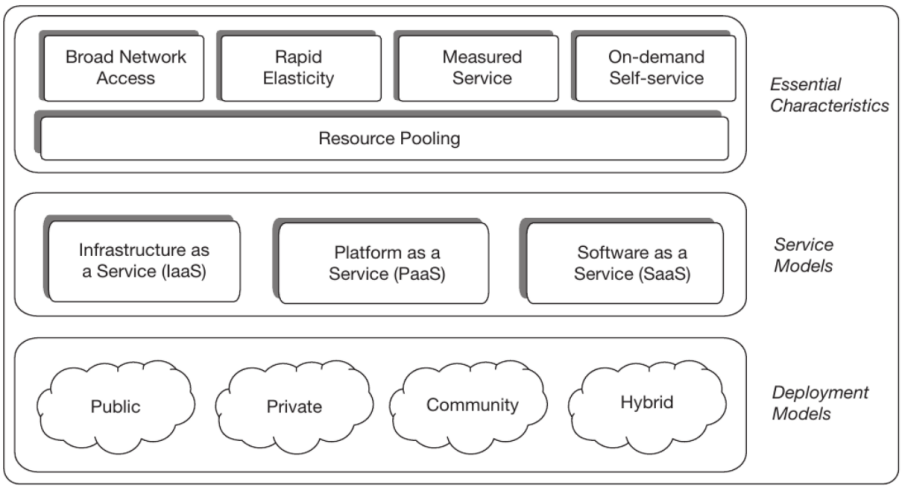


A04 Cloud NIST model, service types and deployment models

Cloud Computing standardization is limited, as its technology is recently introduced. Many models to formalize it, the most appreciated is the NIST one. It defines a model that provide a definition of cloud computing and a reference architecture that highlights all the components of a cloud system.

The **NIST** (National Institute for Standard in Technology) defines the basic aspects and characteristics of cloud computing. It focuses on what composes the cloud infrastructure and not how it is implemented. It specifies also the involved actors. In particular, it focuses on three fundamental concepts of cloud computing:

- **Essential characteristics:** the set of mandatory features that each cloud computing system must have
- **Service Models:** the set of different types of service that are provided to consumers
- **Deployment Models:** the set of different models for the deployment of the cloud infrastructure



Essential characteristics

5 essential characteristics for a cloud computing infrastructure:

1. **Broad network access** (the infrastructure must be accessed from anywhere)
2. **Rapid Elasticity** (resources are allocated/deallocated rapidly)
3. **Measured service** (the resources are measured and billed proportionally to the usage)
4. **On-demand** (resources are allocated upon request)
5. **Resource pooling** (computing resource are installed and available on demand to consumers to satisfy their needs, the resource pools must be large enough to satisfy many users simultaneously)

Deployment models

The set of ways of deploying the cloud infrastructure depends on the requirements of the consumer organization. Models are related with the location of the infrastructure with respect to the consumer organization and its access boundaries:

● **Public** (aka external cloud): infrastructure is placed off-premises and is managed by an external organization or enterprise that sell the service and the resources, the user accesses the service remotely.
In this deployment model we have the highest level of multi-tenancy: the same physical resources are shared among multiple and unrelated consumers. The high number of customers allows the provider to exploit economy of scale for technologies and personnel. This type of deployment is the most commonly used.

● **Private**: deployed and managed by a single organization for its internal use. The infrastructure is built with all the features of a cloud computing system, however, its access is restricted. Private clouds are typically located either within the consumer organization's premises or in a neutral location.
Private cloud deployments are adopted usually when the consumer wants specific control over the environment, or when there are specific requirements (e.g. sensitive data) are involved.

What is different between private clouds and traditional computer model? This is an evolution of it. The infrastructure will have virtualization, it exploit dynamic scalability and dynamic allocations of resources.

Public vs Private

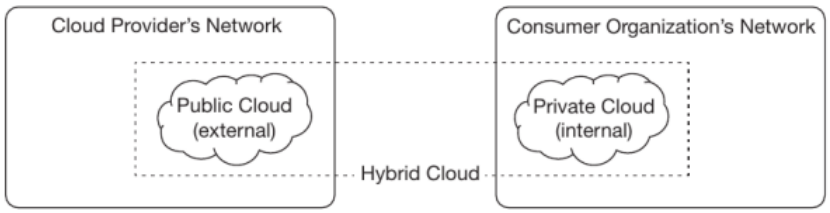
One major difference of private cloud with public cloud is that any private cloud shares **one-to-one** relationship with

consumer while a public cloud maintains **one-to-many** relationship. In the private case, the feature of multi-tenancy does not apply.

Private Cloud	*Public Cloud*
It can be both of types of on-premises and off-premises.	There cannot be any on-premises public cloud deployment.
On-premises private cloud can be delivered over the private network.	It can only be delivered over public network.
It does not support multi-tenancy feature for unrelated and external tenants.	It demonstrates multi-tenancy capability with its full ability.
The resources are for exclusive use of one consumer (generally an organization).	The resources are shared among multiple consumers.
A private cloud facility is accessible to a restricted number of people.	This facility is accessible to anyone.
This is for organizational use.	It can be used both by organization and the user.

● **Community**: consortium of companies that put together their money to create an infrastructure shared between these companies that are part of the community. Such deployments are open to members only.
Might reside on-premise or off-premises, can be either managed by community members or by some external computing vendors. Pay-per-use model can be applied, multi-tenancy among members of the community is applied.
→ are a form of generalized private cloud: provide the benefits of public clouds, with the control of a private deployment

● **Hybrid**: created by combining private or community cloud deployments with public cloud deployments. With hybrid clouds the consumer gets both low-cost computing and high level of control and privacy. Services can be deployed on the private or on the public part, or on both (usually different part of the services).



Selecting the Cloud Infrastructure Deployment Model:
For general use any public cloud is a good option, while private or community deployments are the best when the company has specific requirements or concerns in terms of data privacy or sensitive to business related data. The cost of migration and the cost of ownership of the infrastructure must be considered too.

Service Delivery Models

Different types of services can be provided to cloud consumers: From IaaS to SaaS we have increasing level of abstraction. You can use one lower level to implement a service of higher level. (e.g. a IaaS to create a PaaS).
Each delivery model differs from the others based on the level of abstraction from the hardware infrastructure. Consequently, they vary in terms of the aspects that the consumer needs to take care.

In **conventional Computing model**: the consumer has to take care of all the aspects: creating the physical infrastructure, installing the OS, middleware and dependencies, developing the service, manage data.

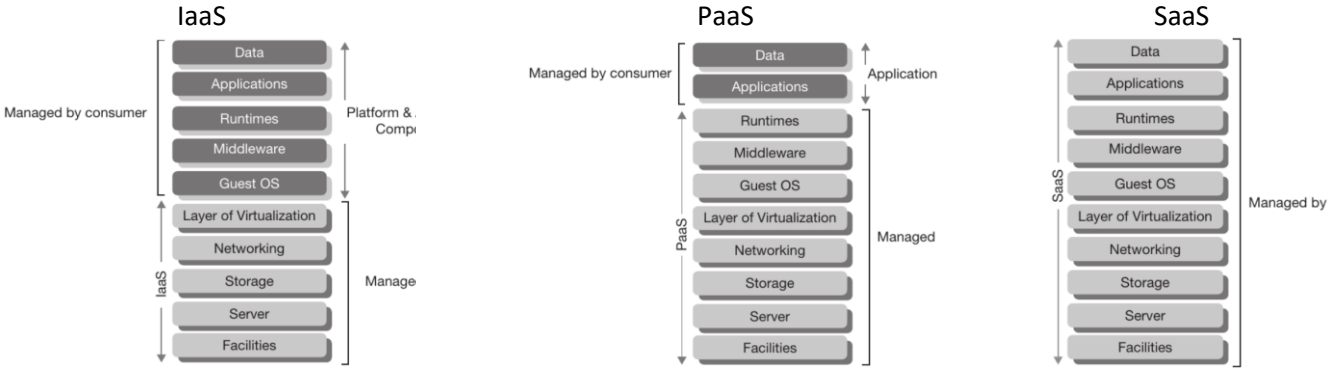
Now, let's look at the three main categories of service delivery models defined by the NIST model:

- **IaaS (Infrastructure As A Service, aka Hardware as a Service)**: delivers virtualized-hardware resources to consumers. It provides the possibility of remotely using virtual processor, memory, storage and network resources, that can be used (like the real hardware) to create any computer setup (virtual machines).
  The consumer have to take care of installing the OS on top of virtualized hardware, configuring the environment, programming the application and managing data.
  IaaS remove costs to build and maintain the physical computing infrastructure, with good control on the environment on which services are deployed.

- **PaaS (Platform As A Service)**: delivers access to a computing platform in which the consumer can design and develop the application components. All aspects, from the physical infrastructure to the installation of the OS and

a middleware/runtime platforms, are managed by the provider. Consumer only have to develop application logic and manage the data.
The programming environment is provided by the cloud provider → main **drawback**: lack of flexibility, you have to use APIs exposed by the provider and for this reason the apps are hard to port between different cloud.

- **SaaS (Software As A Service)**: the provider provides to consumers applications that are developed directly by the cloud provider. Such applications (or service) are offered to consumers via web interface to consumers.
  In this model everything is managed by the provider, from the hardware to the software development, even software licensing.



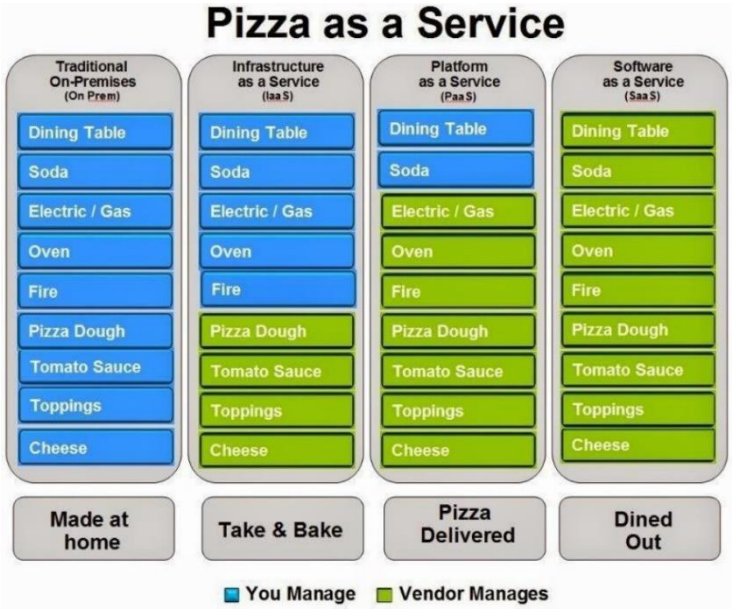
PaaS and SaaS can be implemented on top of an IaaS infrastructure.

An example to understand

Infrastructure as a Service: IaaS provides virtual machines, virtual storage, virtual infrastructure, and other hardware assets as resources that clients can provision

Platform as a Service: PaaS provides virtual machines, operating systems, applications, services, development frameworks, transactions, and control structures

Software as a Service: SaaS is a complete operating environment with applications, management, and the user interface



Other types of Cloud Services

Storage as a Service: Many cloud vendors offer independent storage services from IaaS services.
Database as a Service: exclusive cloud computing solution for database, offers a unique platform with on-demand and self-service capability where even non-DBAs can easily fulfil their requirements.
Backup as a Service: Backup is considered as a specialized service that asks for expertise and many cloud computing vendors offer backup-as-a-service (BaaS) that turns out to be most useful and cost-effective for the consumers.
Desktop as a Service (aka Hosted Desktop / virtual desktop): to provide personalized desktop environments to users independently to their device from which are accessing.

Virtualization Technologies

B01 Introduction to virtualization and basic concepts

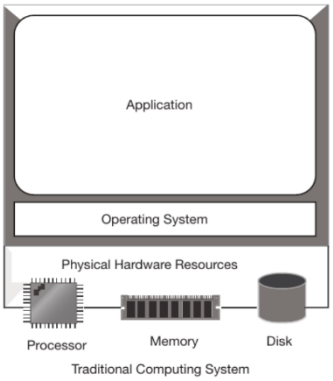
Virtualization refers to the representation of physical computing resources in simulated form through an additional software layer. This software, referred as **virtualization layer**, is installed over the physical machine to provide a virtual form of the hardware.

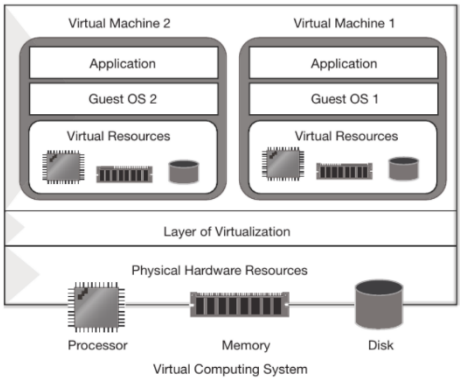
Any kind of IT resources can be virtualized: from the basic computing devices (e.g. processor, RAM, etc) to other resources like storage, network devices or peripheral (keyboard, mouse, printer, etc.). Core resources (CPU, RAM, storage) can be virtualized only if we have physical resources underlying. Peripherals can be completely abstract.
The simulated devices produced through virtualization may or may not be equal to the actual physical component, in architecture, quantity or quality, for example:

- Three virtual processor might be produced by using one physical processor
- One 32-bit virtual processor can be produced from a 64-bit physical processor

Guest system is the virtual machine, **host system** is the physical machine.

Previously we had a one-to-one relationship between operating system and physical computer.
With virtualization each guest system is independent from the others and indirectly accesses host system's hardware. It has its own operating environment.





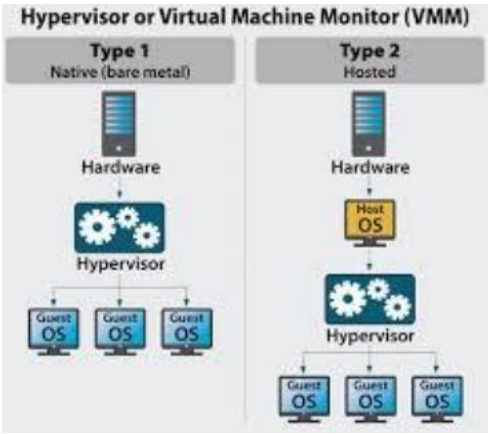
The virtualization layer is a set of control programs that creates the environments for multiple VMs on top of which guest OS can be executed, these guest OS runs on top of the virtual resources whose representation is created and managed by the virtualization layer. Every VM represents an isolated execution environment, so every VM remains independent from the other and independent from the physical hardware since every software running on top of the VM can access the physical resources only indirectly, through the virtualization layer.

Hypervisor (or Virtual Machine Monitor VMM)

The virtualization layer is a set of control programs that creates the environment for VMs to run on. It provides access to the virtual resources to the OS that is installed in any VM. This software layer is referred as the *Hypervisor* or *Virtual Machine Monitor* (VMM).
The role of the hypervisor is to abstract completely the underline physical hardware by creating this virtual representation of resources. The underline hardware environment is not accessed directly since the OS inside the VMs access only the virtual representation of the resources.

There are two different techniques of virtualization, for each technique we have a different type of hypervisor:

Hypervisor type 1 (Bare Metal Approach):
Runs on bare metal: hypervisor is directly installed over the physical machine, accesses directly the hardware of the host. Needs for hardware drivers.
PRO: better performance compared to type 2, provides advanced features for resource and security management → administrators have higher control on host environment
CONS: hypervisor is more complex because he needs to also have the drivers and software to interact with physical hardware → limited hardware compatibility (cannot run on a wide variety of hardware platform)



Hypervisor type 2 (Hosted Approach):
Runs over an host OS that handles hardware drivers.
PRO: High compatibility. Hypervisor installed interacts with physical hardware only through the functionalities offered by the host OS → eases the installation and configuration of the hypervisor
CONS: hypervisor does not have direct access to system hardware resources, all the requests go through the host OS, then there is an higher overhead

For large cloud providers managing numerous servers, bare metal is typically adopted due to its significant performance advantages. Conversely, the hosted approach is chosen when simplicity takes precedence over performance.

Further insight into the drivers

Let's recall that the drivers are specific software programs that allow the operating system (OS) and other software applications to interact with a computer's hardware. Device drivers act as an interface between the operating system and hardware components such as video cards, hard drives, and other peripheral devices.

- **Hosted Approach** (type-2 hypervisor): the host operating system manages the device drivers for the underlying hardware. This means the hypervisor relies on the host OS to access hardware, leveraging the device drivers already installed on it. This approach simplifies the installation and configuration of the hypervisor, as there's no need to worry about direct hardware compatibility or managing specific drivers for the hypervisor.
However, the downside is that all hardware access requests from the virtual machines (VMs) must pass through the host OS, potentially introducing overhead that can degrade performance.

- **Bare Metal Approach** (type-1 hypervisor): bare metal hypervisors are installed directly on the computer's hardware, operating without an underlying host operating system. In this scenario, the hypervisor must have its own device drivers to communicate directly with hardware. This allows VMs to access hardware resources more efficiently, improving overall performance compared to type-2 hypervisors.
However, because the hypervisor must directly manage all necessary device drivers, this can limit hardware compatibility to a narrower set of supported platforms. The direct management of drivers requires that the type-1 hypervisor be specifically designed and optimized for the hardware it runs on.

Different levels of virtualization

Hypervisor-based virtualization techniques can be divided into three different categories as **full virtualization**, **para-virtualization**, **hardware-assisted virtualization**.

Towards the end of the chapter, we will explore a virtualization technique that does not rely on the use of an hypervisor. This virtualization technique is known as **lightweight virtualization**, and specifically, we will examine **operating system virtualization**.

Full Virtualization

The hypervisor fully simulates or emulates the underlying hardware. This enables **unmodified versions** of standard OSs to run in virtual environment. The hypervisor has to handle all the OS system calls.
Guest OS assumes that he is running on physical resources but he is completely isolated from them.
Any OS can run on this virtualized environment unmodified because there are "no" differences with physical resources.

PRO: security is implemented at the highest level since the guest OS remains isolated from physical hardware and is completely isolated from the other VMs. OS guest has no idea that he is running in a physical environment and that no modifications are required.
CON: emulation of all resources results in a significant overhead.

Para-Virtualization

In order to reduce the overhead of full virtualization, the para-virtualization is introduced.
In para-virtualization, a portion of the virtualization management task is transferred (from the hypervisor) to the guest operating systems. The guest OS must be aware that it is operating in a virtualized environment and needs to implement specific API calls to communicate with the hypervisor, enabling its participation in the virtualization management tasks.

A modified version of the guest OS is necessary, meaning that not every OS can run on a para-virtualized platform. Modifications are based on the specific underlying hypervisor, as API variations are dictated by the specific hypervisor in use. The hypervisor does not need system drivers since **system drivers are handled by the guest system**.

In paravirtualization, the guest operating system uses a set of APIs provided by the hypervisor to interact with the underlying hardware. This means that, instead of using traditional hardware device drivers, the guest operating system uses "**paravirtualized drivers**" specifically designed to communicate with the hypervisor. These paravirtualized drivers allow the guest operating system to send I/O (Input/Output) requests to the hypervisor, which then manages the interaction with the actual physical hardware.

How It Works:

1. **Paravirtualized Interface:** The hypervisor exposes a paravirtualized interface to the guest operating systems. This interface is optimized for virtualization and enables efficient communication between the guest and the hypervisor.
2. **Paravirtualized Drivers:** The guest operating systems use paravirtualized drivers, which are aware of the virtual environment and designed to interact with the hypervisor through the paravirtualized interface. These drivers are different from standard device drivers that communicate directly with hardware.
3. **Handling I/O Requests:** When a guest operating system needs to perform an I/O operation, it uses the paravirtualized drivers to forward the request to the hypervisor. The hypervisor, in turn, handles the request using its device drivers to interact with the physical hardware.

PRO: the use of a modified OS reduces the virtualization overhead of the hypervisor compared to full virtualization.
Why? Because there is **no emulation** of instructions carried out by the hypervisor. Para-virtualization allows calls from the guest OS to directly communicate with the hypervisor.
Can run on many different hardware since drivers are handled by guest OSs.

CON: Unmodified versions of available operating systems (like Windows or Linux) are not compatible with para-virtualization hypervisors.
Security: the guest OS has high access to physical resources, if the guest OS is compromise, it could have access to the hardware, since part of the code that in full-virtualization run on hypervisor, now it runs on guest OS.

▲Be cautious: in para-virtualization, the guest OS communicates directly with the hypervisor, not with the CPU, as it will occur in Hardware-assisted virtualization.

Hardware assisted Virtualization

To improve the performance of full-virtualization, Intel-VT and AMD-V introduced direct virtualization support on processors. This allowed guest OSs to make privileged calls that are directly handled by the CPU, eliminating the need for translation/emulation by the hypervisor and reducing overhead. Consequently, binary translation or para-virtualization were no longer required.
There is need of specific combinations of hardware components.
These hardware changes bring the performance virtualization in the same order of a program running directly on top on OS on top of bare metal without virtualization and this is why para-virtualization is not used anymore.

Operating System Level Virtualization (lightweight virtualization)

Operating system virtualization, is a technique of virtualization that falls under the category of **lightweight virtualization** approaches. This form of virtualization differs from the ones we have discussed so far in that it does not rely on an hypervisor.

It's a partially redesign of the virtualization aimed at minimizing its overhead.
The idea is to implement a virtualization technique in a different manner, embedding it directly into the OS. It works in a completely different manner.

The **virtual servers** are enabled by the kernel of the OS of physical machine, that is shared among many VMs.
No hypervisor at all: all the functionalities of the hypervisor are implemented by the kernel of that OS. It is a lighter approach but you can install only VMs with same kernel.
The host OS and the kernel are in charge of creating many logically distinct user-space instances (called virtual servers) on a single instance of OS kernel. They are not VMs properly, but they conserve many of the same characteristics.

PRO: since virtual servers are more lightweight than VMs, you can support an higher number of them with respect to the number of VM you can run on the same hardware.
CON: **less flexibility** since the kernel is shared, you can have virtual servers running different distributions, but not different OSs. **Security** because if we have a glitch in the kernel, the malicious user con compromise hardware and all VMs.

Other types of virtualization

Virtualization of computing infrastructure is not only about machine or server virtualization. In order to create a complete infrastructure, we also need **network** and **storage virtualization**.

- **Network Virtualization**: network virtualization is the process of combining network resources and network functionalities into a single, (usually software-based) administrative entity called as a **virtual network**. It allows to create a virtual network for VMs to communicate. The virtual network usually exploits the real network infrastructure for data transmission
- **Storage Virtualization**: In traditional computing system, the storages have always been directly linked with the physical servers. In a virtualized system storage must be virtualized. Like other computing resources, virtualization of storage also happens through layer of software which creates logical abstraction of the pooling of physical storage devices having linked together by network. Data stored in logical (virtualized) devices ultimately get stored in some physical storage disks.

Advantages and disadvantages of virtualization

Virtualization PROs:

- This process of running multiple VMs on the same hardware is called **server consolidation** and allows to increase hardware utilization → It reduces hardware costs.
- Virtualization simplifies system installation, as a new system can be created by cloning another VM, without requiring the installation and configuration of a full system.
- It improves security as each VM is isolated from the others thanks to the additional isolation provided by the virtualization layer.

Virtualization CONs:

- Each physical machine is a single point of failure: the major benefit of virtualization is resource sharing but if one machine fails, many VM fail.
- Lower performances: virtual servers can achieve up to 85/90% of hw native performance as VMs cannot get direct access to the hardware.

Emulation

Emulation in computing is the act of making a system **imitating** another one; a system that has an architecture is enabled through emulation to support the instruction set of some other machine architecture.

Difference between virtualization and emulation? In virtualization, the architecture of the VM is the same of the physical machine in which it is performed, the same is not true for emulation.
Virtualization is a specific case of emulation, is an emulation performed to emulate a system that has the same architecture of the host system.

We consider virtualization and not emulation in cloud computing because emulation requires translation of instruction since virtually the emulated architecture is different from the physical one where it is executed and consequently the instruction set is different, so we need a binary translation.

There are two ways for implementation of emulations:

- **Binary translation** (aka recompilation): the binary code base is recompiled for the targeted platform (statically or dynamically).
- **Interpretation**: each instruction is interpreted by the emulator every time it is encountered (high overhead, but easier to implement).

Emulation-based virtualization allows to run guest OS written for an architecture to run on another architecture.

In **regular virtualization** techniques we must have the same instruction set between guest VM and host system, this assumption improves significantly the performance respect to the emulation because translate instruction is time consuming. Regular virtualization does not need of a translation layer for the execution of instructions of the virtual machine (guest OS and programs), except for some exceptions (that we will see).
Performances approaches native speed in many cases → virtualization is significantly faster than emulation.

By the way virtualization can still require emulation in some cases (to emulate some hardware or privileged instructions).

B02 Multiprogramming recap & Full Virtualization

Virtualization requirements:

- **Equivalence**: between an OS running directly on bare metal and in a VM.
- **Resource Control**: the hypervisor must have complete control of the physical resource, while the OS running in the VM must have complete control of the virtualized resource.
- **Efficiency**: the most part of instructions must be executed without hypervisor intervention in order to ensure efficiency.

Remember: the instruction set used by the virtual system is the same to that of the actual hardware system.

We note that such requirements are similar to the ones for multiprogramming.

Multiprogramming

Multiprogramming is the possibility to run more programs at the same time. The processes must have the impression that they have the complete control over the processor and that they are the only process executed on it. Through multiprogramming we emulate a machine with a higher number of processes than the number of CPUs physically available. Each process runs on its own virtual CPU (vCPU).
By executing different VCPUs frequently, we give processes the impression they are executed continuously.

Virtual Processors (VCPU): OS implements multiprogramming by creating a virtual representation of the processor, which is a data structure that contains a copy of the processor's state (its registers). So each VCPU has its own CPU state, stored in memory. When a **context switch** happens, the current values inside the registers of the CPU are copied inside the current process VCPU state, while the values of another VCPU are loaded into the registers of the real CPU. This is handled by the OS → a context switch can be triggered either by a timer (handled by the OS scheduler) or an interrupt (e.g. when a process that was waiting for a hardware input goes in READY state).

Multiprogramming hardware support required

A complete multiprogramming environment requires some hardware support:

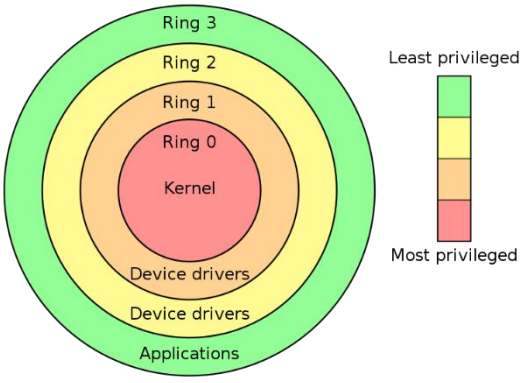
- An interruption mechanism (e.g. timer, hardware signal) to trigger a periodic context switch
- An automated mechanism to copy the state of the registries, like specific instructions (not mandatory but it helps to speed things up)
- A memory protection mechanism to isolate the processes' VCPU states stored in memory thus predicting unauthorized access to certain portions of RAM.

Processors have support for at least two different privilege levels:

- **system** (or **kernel**) level → can access all the memory
- **user** level → can access to only a subset that does not include VCPU data structures

The code of the OS is executed in system level while the code of user's processes is executed in user level.
Access to privileged memory and execution of privileged instructions from a process executed in user level is denied.

The architecture of modern CPUs is slightly more complex than this, as they have not just two levels (user and kernel) but four levels of execution privilege known as **rings**, ranging from ring 0 to ring 3. These levels are used to protect the operating system's data and functionality from unauthorized access.
Each ring has a distinct set of instructions and a different memory space that can be accessed. The level of instructions and the portion of memory that can be accessed increase as the ring ID decreases.
Ring 3 is associated with the lowest privilege and is typically assigned for running applications, up to **ring 0**, which is usually linked with kernel code as it has unrestricted access to all system resources.

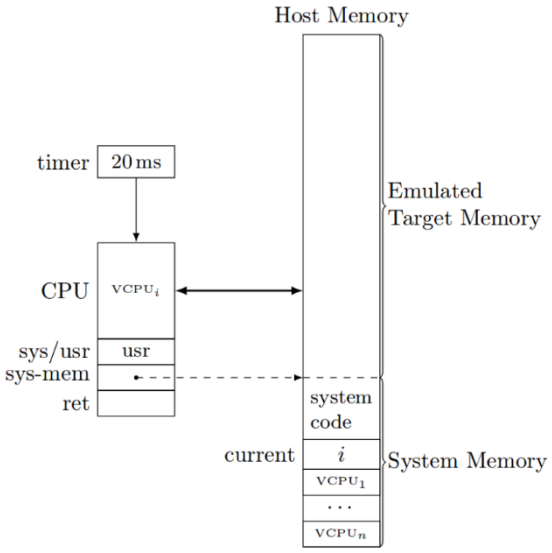


The intermediate privilege levels (**ring 1** and **ring 2**) are typically designated for the execution of device drivers, which are programs requiring some elevated privileges but not all those available in Ring 0.
Since device drivers are often external programs developed by individuals other than the operating system kernel developer, placing them in intermediate rings ensures they do not have complete privileges within the system.

This arrangement ensures that if a device driver turns out to be malicious, it won't have complete control over the system.

Protection support includes:

- A **usr/sys registry**, a flag that specifies if the CPU is currently in system or user mode
- A **sys-mem registry** that contains the starting address of the privileged memory where VCPUs data structures are stored
- A **ret register** that stores a memory address to be used when returning to user mode from kernel mode



When the timer for context switch is triggered, the OS performs the following operations:

1. Saves the state of the host CPU on the VCPU representation
2. Selects a new VCPU to run and loads its state into the registers of the host CPU
3. Executes a special instruction to jump to the (new) address stored in ret and return in user mode

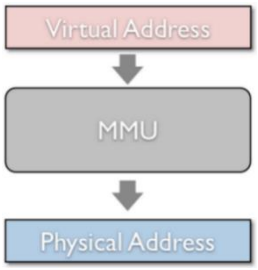
Virtual Memory

Virtual Memory is a mechanism introduced to abstract from real memory available on the system and from real addresses.

Virtual Address Space: every process has its own address space, and has the impression of having access to the whole memory. Processes can access contiguous addresses in their virtual address space. Data can be stored in physical memory in independent positions.
The management of virtual address spaces is demanded to the OS, that is in charge of allocating each required virtual address space in the physical memory.
The result is that each process has the impression that can have access to the whole physical memory.

Virtual memory is implemented through two mechanisms:

- Virtual address spaces management: is performed by the OS, that takes care of managing the virtual address spaces created for each application
- Address translation: is performed on the CPU by a specific hardware element called Memory Management Unit (MMU) that takes care of translating the virtual address in the corresponding physical address



Memory is divided into **pages**, that are blocks of equal size (e.g. 4KB).
The CPU has a special register called **PTBR** (Page Table Base Register) which points to a physical address in which the page table for the current process is stored.

Page Table: is organized into page directories, each one containing information on how to translate a portion of the virtual address into a physical address. Page table is maintained by the OS.
Page Table could be multilevel, many nested directories can be exploited to translate an address.

Segmentation: when the amount of virtual memory allocated exceeds the quantity of physical memory available , certain pages may be stored outside RAM, specifically on secondary storage devices, such as a hard disk, when they are not in use.

Page Fault: when a program tries to access a page stored in a secondary storage. The OS is forced to retrieve it and move it to RAM. Once the page is moved, the page table is updated.

Interrupts and Exceptions

Interrupts and Exceptions are used to notify the system of events that need immediate attention. Alter the normal execution of the program triggering the execution of a function in kernel space. When an exception or an interrupt occurs, the transition from user mode to kernel mode is performed and the handler of the OS associated with the exception/interrupt is executed. When the execution of the handler is completed, the regular execution resumes in user space.

Exceptions: are internal, synchronous. Used to handle internal program errors (e.g. division by zero, page fault). Another name for exception is **trap**. A trap can also be invoked by software via the instruction `INT`. This is used to implement system calls that can be invoked from programs in user space.

Interrupts: used to notify the CPU of external events generated by hardware devices outside the CPU.

Interrupt Descriptor Table (IDT) (aka Interrupt Vector): a table used by the processor to link interrupts and exceptions with **handlers**. Each handler is a function in kernel space. IDT is populated by the OS at the bootstrap.

Full Virtualization Techniques - trap and emulate approach

Full virtualization (aka system level virtualization) implementation is facilitated by the presence of structures introduced for multiprogramming in CPU architecture. In fact, similar to how multiprogramming creates the illusion for each process of having complete control over CPU and RAM, the hypervisor must likewise provide virtual machines (VMs) with the impression of full control over the physical hardware (CPU, memory, I/O devices).

As a result, the role of the VMM is similar to the role played by the OS Kernel, its functionalities, however, are far more extended than just supporting multiprogramming, due to the complexity of hiding the virtual environment to VM: a VM is a multiprogrammed system itself and hosts an OS and not just a program.

The objective is to implement the hypervisor by leveraging existing hardware mechanisms designed for multiprogramming support, creating a virtual representation of the system and its devices.
If the target and host architectures are the same, the implementation can minimize the use of emulation, which incurs significant overhead due to the need for binary translation. Most of the time, the code of the operating system or software running inside the virtual machine can be directly executed on the hardware without requiring intervention from the hypervisor (e.g. instruction emulation), to minimize performance penalties for code executed within the virtual machine.

Virtualizing CPU

The VMM adopts the same techniques adopted for creating VCPUs in a multiprogramming environment:
VMM code is executed in system space, while all the guest OS code is executed in user space.

VMM loads VCPU state of a VM in the host CPU, then let the code run until certain instructions (that couldn't be executed by the guest OS in software) are found → this will trigger a context switch that gives back control to the VMM, the target instruction is emulated. Once emulation is complete, the VMM reloads the virtual CPU onto the host CPU to continue execution. This procedure is expensive in terms of wasted time.

When many VMs are running on the same machine, the VMM could schedule a timer to trigger a context switch periodically, in order to ensure fairness in the use of physical resources among the VMs.

There are **differences between VMM and OS kernel in multiprogramming**, in particular:

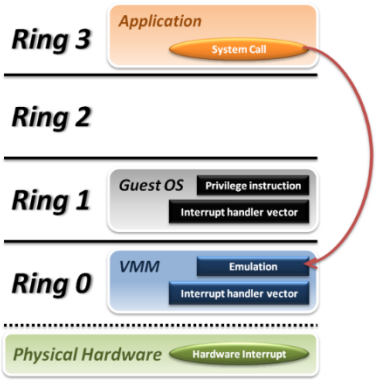
- The guest OS VM code would like to run also at system level and to have complete control of the host hardware. Consequently, VMM must emulate the entire processor, not only user level functions but all the functionalities used at system level → guest OS code must have access also to privileged registers and instructions
- In multiprogramming, every time a privileged instruction is executed in user space the process is killed by the kernel, we don't want the VMM to kill the VM but to manage (emulating it) the exception instead!

Trap and Emulate Virtualization Model

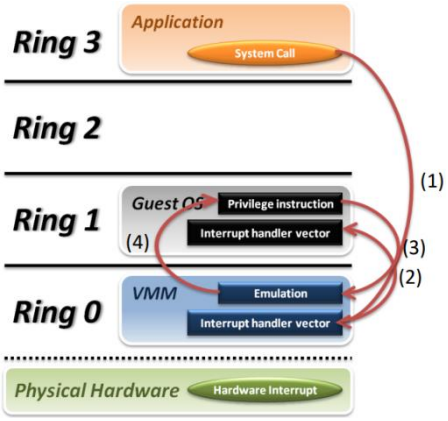
Guest OS and guest applications are both executed in user mode of the CPU, with the guest OS operating in ring 1 and the guest applications in ring 3. Since the guest OS code is executed in user space, every time a privileged instruction is executed (i.e. guest OS wants to manipulate the MMU, access I/O devices, handle interrupts, …) an exception is raised. Every time an exception or an interrupt arise, there is a context switch from guest OS to the VMM.

This virtualization model is called trap and emulate because after the context switch to the VMM, the trap code executed by the VMM employs binary translation to execute the privileged instructions of the guest OS, emulating their behavior.
The performance are not affected significantly if the privileged instructions quantity is limited.

- Simply case of privileged instruction (e.g. IN/OUT instructions):
  - The guest OS want to execute a privileged instruction
  - The host CPU generate an exception (trap) since the Guest OS is executed in the user space
  - A context switch occurs: the hypervisor save the state of the VCPU and then emulate the execution of the privileged instruction (remember, it can access to all the instruction of the CPU host)
  - The VCPU state is reloaded (as when a process is in the ready queue)
  - The program execution continue



- INT instruction case:
  - Application want to do a system call
  - The `INT` instruction triggers the execution of the handler code of the VMM
  - This code has to understand that this interrupt come from the execution of a program on the guest OS, then need to read the guest OS IDT and change the Instruction pointer of the VCPU with the value of the interrupt code in order to execute the routine X on guest OS
  - Control is returned to the guest OS (running in ring 1), allowing it to execute its routine
  - While executing the routine X, the guest OS try to execute a lot of privileged instructions: for each of them an exception on host CPU is generated → the hypervisor emulate the single instruction and then the routine (or the program when the routine terminates) continue



RECAP: In a virtualized environment, when the code inside the VM attempts to execute a privileged instruction, typically when it wants to execute kernel code of the guest operating system, this event is detected by the trap-and-emulate mechanism. In such cases, the VMM regains control and emulates the behavior of the privileged instruction.
Emulation, in this context, refers to performing all the necessary steps to replicate the system's state as if the instruction had been executed originally.
So, when the VMM regains control, it emulates the behavior of the privileged instruction and subsequently returns control to the VM, which then resumes execution immediately after the privileged instruction. At this point, the status of the VM will reflect as if the privileged instruction had been executed.

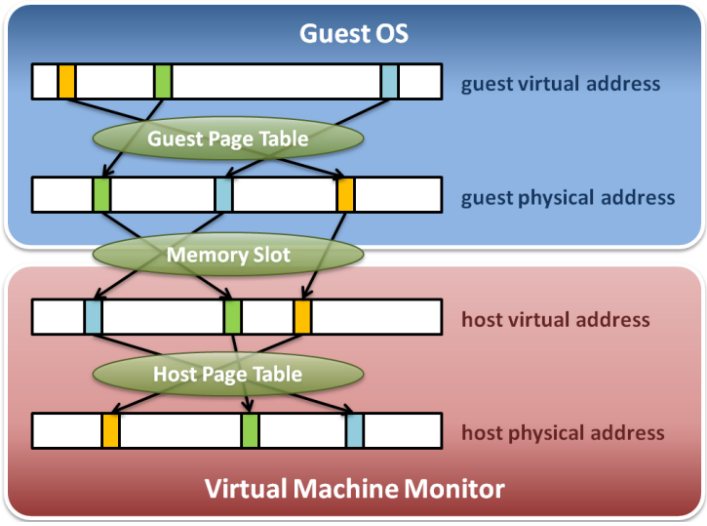
It is essential to highlight that in a virtualized system, when a VM invokes a system call, two context switches occur, contrasting with the single context switch in a non-virtualized multiprogrammed system. The first context switch happens when transitioning from the application to the VMM, and the second occurs when moving from the VMM to the kernel handler of the virtual machine (VM).

When the specified handler by the guest OS of the VM is executed, it operates under ring 1, with access to only a limited set of privileged instructions. Each time a privileged instruction is executed within this code, another context switch takes place. This additional context switch is necessary to execute a handler within the VMM that emulates the specific privileged instruction, before control is returned to the ongoing handler.

Virtualizing Physical Memory

As always the hypervisor of the guest OS has to get the impression that he has control of the RAM, however, the VMM is the only one having full control of the whole physical RAM.
The physical memory of the guest VM is typically implemented in a similar manner it is performed with multiprogramming. A part of the overall physical memory will be reserved for VMM execution.

The guest VM must have only access to the portion assigned to it and it must think that its memory starts from the physical address 0.



VMM must takes care of handling the mapping between the real RAM and the virtual physical RAM assigned to the virtual machine. The result of this process is that you have 3 address spaces (we ignore the "Host virtual space"):

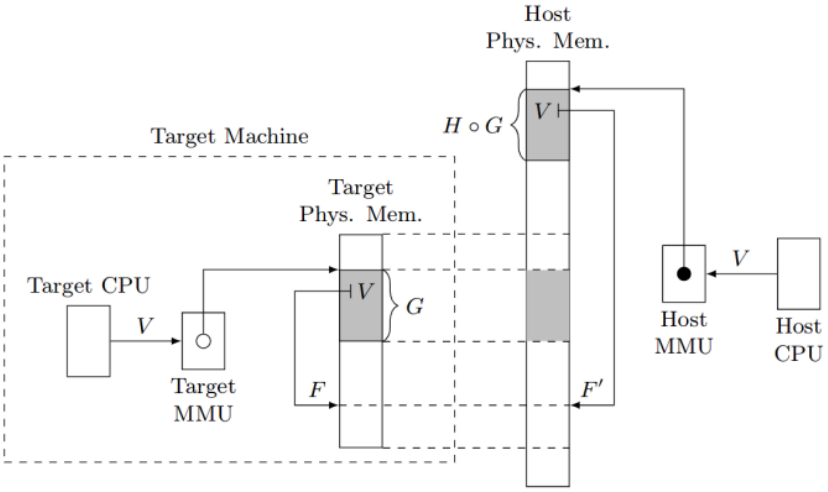
- Guest virtual address
- Guest physical address
- Host physical address, which is managed and accessible only by the hypervisor

Guest OS will have its own guest page table which regulates the translation between guest virtual and physical address, however this guest "physical" address is not real at all, it's another "virtual" address which cannot be used for accessing the physical memory, so we need a mechanism inside the hypervisor to somehow create a kind of additional translation mechanism in order to translate guest "physical" addresses (which goes to 0 to the end of the memory assigned to the VM) into a real host physical address.

Virtual MMU

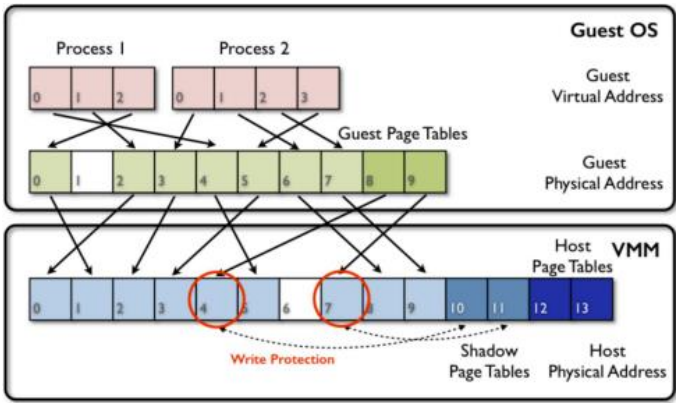
Every guest OS has its own page table. The translation function from guest virtual address to guest physical address is called G. The guest physical address is then mapped from the VMM to a host physical address by another translation function (let's call it H).

We need a MMU that transparently (with respect to the guest OS) translates every guest virtual address to a host physical address by applying the composition of G and H mappings. This role is handled by the **Virtual MMU**.



Brute Force method: shadow page table

The page table of the guest OS is modified by the VMM in order to add an additional level (the **shadow page tables**) that implements the H function, in order to obtain the direct translation: guest virtual address to host physical address, without obtaining the guest physical address as an intermediate step.



So page table are created by guest OS in order to translate guest virtual into guest physical, then VMM silently introduce additional layers into those page tables to make possible the translation of guest physical into host physical.
These page tables (combination of the page table managed by guest OS and page tables managed by VMM) are automatically read by the MMU in order to translate addresses.
However overhead is required every time the guest OS applies modification to the page table, which can be modified anytime by the guest OS and the VMM has no control over that, so every modification to the tables is intercepted and analyzed by VMM that modifies the shadow tables according to the changes.

Shadow tables are **write-protected**, so that every action that tries to modify them causes a VM EXIT.

Implementation: the VMM must trap all the possible actions from the guest OSs related to the MMU and page tables. In particular:

- Changing the PTBR (e.g. when the guest OS wants to replace the page table completely).

- Changing some page table entries in some directories (e.g. when the VM wants to change the mappings of some areas)

Every time this happens a trap is executed, the VMM takes control and updates (or add) the shadow page table.
CONS: the shadow page tables introduces high overhead and requires constant context switches to the VMM for the translations.

Virtualizing I/O Devices

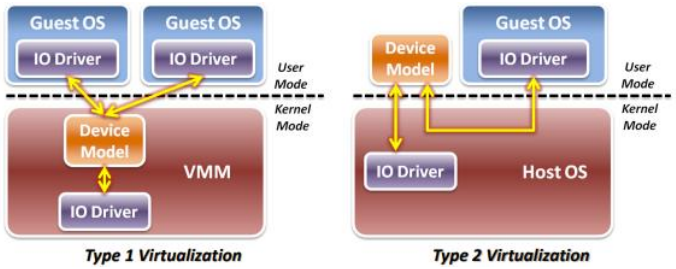
I/O instructions are usually system level instructions, so they cannot be executed in user space.
In multiprogramming processes access I/O devices through the OS kernel (system call).

In order to illude the guest OS to have the control of I/O devices, the VMM has to emulate the real hardware through a set of data structures in memory → **virtual representation** (aka **device model**) of peripherals are accessed by guest VM. The VMM is the only one who can access the real physical devices and is responsible for maintaining the virtual representation up to date. Additionally, it must ensure that the I/O instruction commands executed on the virtualized peripherals are translated into corresponding I/O instructions on the real device.

Sometimes the peripheral is completely emulated, other times the VMM reflects changes in virtual representation to a real peripheral and vice versa.

Device models can be implemented in two ways, depending on the type of hypervisor:

- **Hypervisor type 1** (bare metal): device model is implemented as part of the VMM
- **Hypervisor type 2** (hosted approach): device model runs in user space as a service (VMM is a user space service too)



Interrupt Management

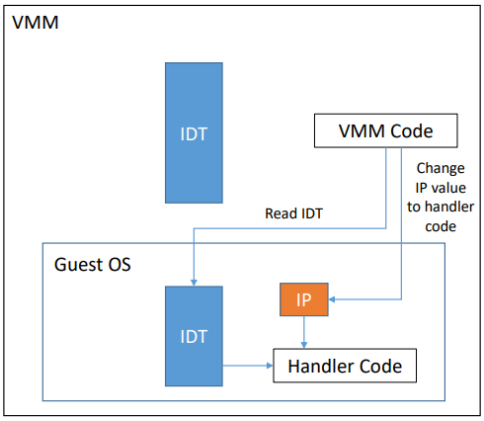
Interrupts coming from the physical devices are handled by the VMM, which has to handle them in a transparent manner with respect to the guest VMs. VMM has its own IDT (inaccessible to the guests) that the host CPU uses to handle the interrupts. Each guest VM has its own IDT too.
VMM has to handle interrupts that are generated from the virtual I/O devices emulated by VMM for the VMs.

Virtualizing Interrupts

When an interrupt for a VM arrives, the VMM has to read the IDT of the guest OS target, and perform the operations required to execute the interrupt code in the guest OS: save current state, point the Instruction Pointer (of the guest VM) to the first instruction of the interrupt handler code and then give back control to the guest VM, so that it can execute the interrupt code.

(of course the VMM has to do other operations beyond the change of the IP, this is a simplification).

Sometimes the guest vCPU could have interrupt disabled. In that case the VMM must wait until those are enabled again before emulating interrupt reception.

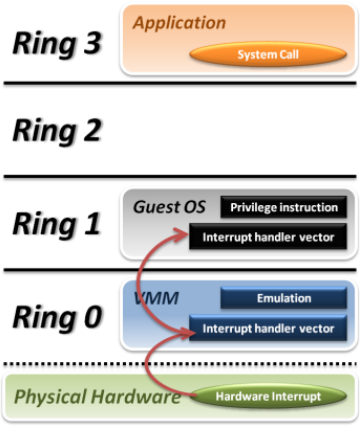


Hardware interrupts

We know that VMM is the only one accessing the real hardware. Consequentially interrupts from devices (hardware interrupts) are always handled by the VMM.

Hardware interrupt triggers the execution of the interrupt at the VMM (specified inside the IDT of the VMM), even if a VM is running. This VMM handler might initiate a context switch and launch the interrupt handler within the Guest OS of the designated target VM (software emulation of the interrupt).

(Here, we are assuming that each real device triggering these hardware interrupts is assigned to a specific VM)



QEMU

One of the hypervisor that can implement the “trap and emulate approach” and the “brute force approach” that does not exploit hardware assistance in the virtualization is QEMU.

Quick EMUlator is an open-source emulator that performs hardware virtualization. QEMU is a VMM software that emulates the machine hardware. Supports a wide set of hardware emulations.
QEMU can even run a VM with instructions set that is different from the one of the host system exploiting binary translation.

B03 Hardware assisted virtualization

CONS of classic full virtualization: since the guest OS cannot run code in system mode, we have many context switches from the VM to the VMM, which mimics the execution of a privileged instruction or other operations → introduce high overhead (performance penalty).

In order to solve this problem and improve the performance both *AMD* and *Intel* have extended their CPU with new features starting from 2006, those features are explicitly designed to support efficient implementation of VMM-Hypervisor software and minimize the performance penalties.
Intel and *AMD* extensions are equivalent; thus, we focus on the Intel extension called VMX.

Intel VMX (Virtual Machine Extension)

▲ VMX technology introduces two new operating modes: **root** and **non-root**. Those modes are orthogonal to the already existing **system** and **user** modes, so we have 4 combinations:

- root/system
- root/user
- non-root/system
- non-root/user

Root mode is for the VMM running on the host, non-root for guest VM.
The idea behind these new modes is to put hardware-controlled limitations on the set of instructions that the VM can run without hypervisor intervention in order to reduce the number of context switches between VM and VMM.

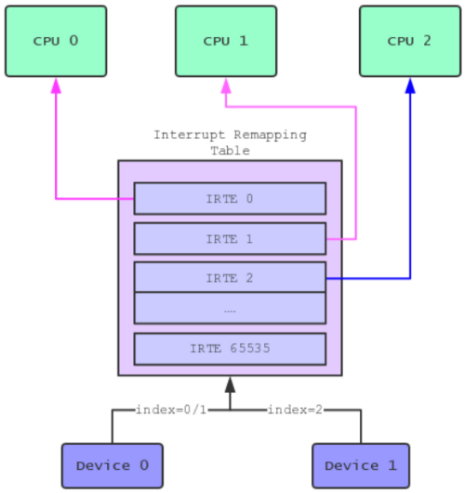
In order to implement root/non-root mode, additional hardware support is needed to let the VM to execute some privileged instructions and in order to handle directly (some) interrupts directly on real CPU, this without requiring VMM intervention for their emulation.
The VMM decide the set of privileged instruction that the VM can execute without triggering a context switch.

▲ Two new instruction for the host system to manage VMs: **VMLAUNCH** and **VMRESUME**, allowed only in root + system mode. The new hardware instructions streamline the operations that were previously manually performed by the VMM during the launch of a VM or when returning control to a VM after a context switch.
Before hardware-assisted virtualization, the VMM had to emulate registry changes, state preservation, and other operations step-by-step. However, with the introduction of hardware support, these operations can now be executed with a single instruction, leading to substantial performance enhancements for the system.

Another hardware-assisted functionality that has been introduced is the implementation of additional **VMEXIT** operations in hardware. VMEXIT represents the context switch operation from the VM to the VMM. Although this operation was already performed in hardware, the hypervisor still needed to execute a series of functions to determine the cause of the context switch. The VMM had to inspect if a privileged instruction was invoked and gather information about the current status, among other tasks. To improve efficiency, this context switch has been modified to simplify the role of the VMM.

Interrupt Management

Until now, interrupt management incurred significant overhead due to the high number of context switches required. To improve performance by reducing the frequency of context switches, a hardware mechanism called **Interrupt Remapping Table** (**IRT**) has been introduced. The IRT is an additional table implemented by the processor's management system that dynamically reroutes interrupts from hardware devices or the processor itself to their respective handlers.

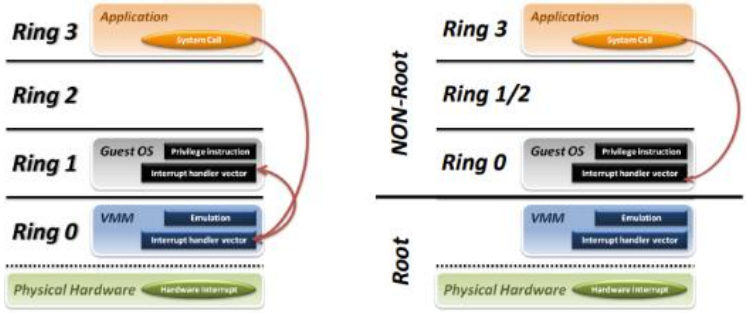


The interrupts are initially directed to the Interrupt Remapping Table (IRT), which serves as the first layer and contains entries corresponding to the potential interrupts in the system. Each entry in the IRT points to the actual table that holds the handler to be executed at that specific moment.
For every entry in the IRT, there is a pointer that directs to the Interrupt Descriptor Table (IDT) of a virtual machine if the goal is to dynamically reroute

the interrupt directly to the VM. In this case, the hardware executes the handler selected by the VM. Alternatively, the interrupt can be directed to the IDT of the hypervisor if the intention is for the hypervisor to regain control whenever that system call is invoked. It is also possible for the interrupt to point to other data structures, which will be discussed later.
CONTROLLA BENE DALLA REGISTRAZIONE SE CONTIENE anche interruzioni interne. Fai capire che Interrupt Remapping Table è sia il nome della tabella che del meccanismo hardware

Example: instructions `INT` and `IRET` within a VM, utilized by user programs running inside the VM to invoke system calls, can be executed without the need for hypervisor intervention, thanks to root and non-root modes.
The VMM has the capability to configure the hardware to delegate the execution of INT instructions to the virtual machine (VM). It achieves this by modifying the configuration of the non-root/system mode to allow the execution of INT instructions without raising an exception. The hardware recognizes this configuration and simply executes the `INT` instruction. Through the `IRT`, the `INT` can be associated directly with the handler assigned to the corresponding system call. Consequently, the hardware automatically executes the INT instruction and proceeds to execute the handler code associated with it.

Without hardware support, when the guest OS used the INT instruction, it was intercepted by the VMM, which had to perform checks to determine if the interruption originated from a VM. Subsequently, it accessed the guest OS's IDT and simulated the INT as specified there. This process involved changing the RIP of the VCPU and other operations. Then, the guest OS would execute its system call, but whenever it encountered a privileged instruction, it had to be emulated by the VMM.



Now, with hardware support, when the INT is intercepted, there is a hardware check on the IRT. If the IRT specifies that the system call can be executed by the VM, the control is passed to the VM to let it execute the system call. This results in several benefits: the VMM's IDT is not accessed at all, and most of the instructions that compose the system call are executed directly by the VM. Only a few instructions that could either compromise the isolation of a VM or require software emulation are executed by the VMM. In these cases, the hardware can trap and switch back to VMM execution.
AGGIUNGI CHE SI switched from non-root/user to non-root/system, and viceversa

Virtual Machine Control Structure (VMCS)
Pursuing the goal of simplifying the implementation of certain functions within the hypervisor and enhancing their efficiency, specific data structures have been introduced. By introducing these structures, a portion of functionalities that were previously implemented in software can now be implemented in hardware. One of these introduced data structures is the **Virtual Machine Control Structure** (**VMCS**) that is a structure that contains all information about VM's state. It handles context switches between VM and VM or VM and VMM.

For each individual running virtual machine, there is a dedicated VMCS. This means that each VM has its own VMCS.

Before hardware support the status of the VMs was stored and managed by the hypervisor, however every hypervisor had it's own structure for store those data. Now the new architecture provides new standard, and this standardization facilitates the implementation of hardware functionalities, enabling efficient and automated management of information that was previously handled by custom software solutions within the VMM.
This extension, in fact, is not only the definition of the data structure but it includes a set of instructions to manipulate such data structure automatically exploiting the hardware support.

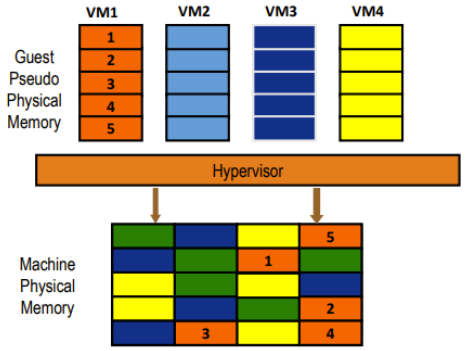
Essentially, these data structures contain the state of the virtual processor, as well as additional data needed to implement extra hardware functionalities during the context switch.

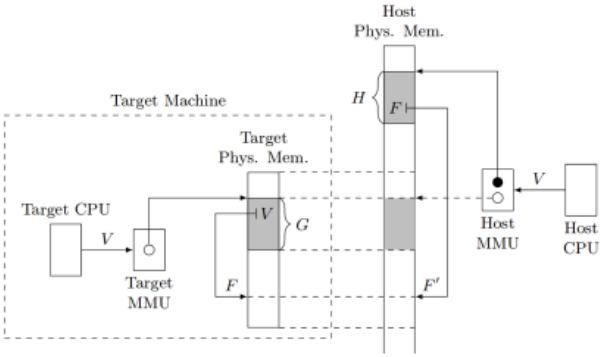
Virtual Machine Control Structure's field are:

- **Guest state**: contains the state of the vCPU of the VM when is not running due to a context switch (contains VM Instruction Pointer). The state is loaded from the structure to the processor during VM launch, while during a VM exit the state is stored back. These operation are now implemented in hardware.

- **Host state**: contains the state of the physical processor state before VM launch. Restored at VM EXIT. (contains VMM IP). These operation are now implemented in hardware.
- **VM execution control**: specifies which instructions are allowed in non-root mode through a set of flags. Unallowed actions triggers a VM EXIT. This structure can be modified only in root mode, since only the VMM can modified it.
  There is a set of flag for I/O operations allowed (exploited in particular for *peripheral passthrough*) and a set of flag for interrupts allowed inside VM:
  - a flag determines what should happen when the CPU receives an external interrupt, while running in non-root mode. Through this flag VMM can specify if the VM can handle interruptions directly or a VM exit must occur.
  - a flag determines whether some critical instructions should cause a VM exit or they can be executed by the VM.
  - another set of flags specifies if I/O operations are allowed inside the VM or a VM exit is required (if they are allowed, which address range are destinated to them).
- **VM enter control**: some options for how to set the vCPU during switch from root to non-root mode. There are fields that can be exploited by the VMM to mimic fake external interrupts to inject exceptions and faults inside the VM.
- **VM exit control**: likewise, but for switch from non-root to root.
- **VM exit reason**: contains info about last VM exit. This structure is automatically updated by the hardware when a exit event is executed with a code specifying exit reason and this will be exploited by VMM to understand VM status and act accordingly.

RAM Management





Let's talk about hardware support for RAM management. The goal of the hypervisor is to control the machine physical memory and to give access to a portion of that memory to each one of the VMs.

Composition of two memory mappings, G and H. One is the function from the virtual memory of the VM to the physical memory of the VM (guest physical address). The latter is from the physical memory of the VM to the physical memory of the hardware.

We remember that the shadow page tables have to be updated every time the guest OS performs a modification and that result in a context switch. However, what we would like to do is to modify the MMU in order to implement these 2 step translation in hardware in order to implement the 2 functions in hardware directly on the MMU (without introduce shadow tables).

Extended Page Table: The hardware of the Intel VMX has two PTBR pointers so it can read two page table: one is the page table of the VM, the other is the page table of the VMM (that implements H function). G and H mappings are applied sequentially.

PROs: no need anymore for VMEXIT every time the VM modifies the page table.
CONs: higher cost for address translation (for each translation 24 additional memory accesses required).
→ newest MMUs are equipped with cache translations to reduce the translation cost

Hardware Passthrough

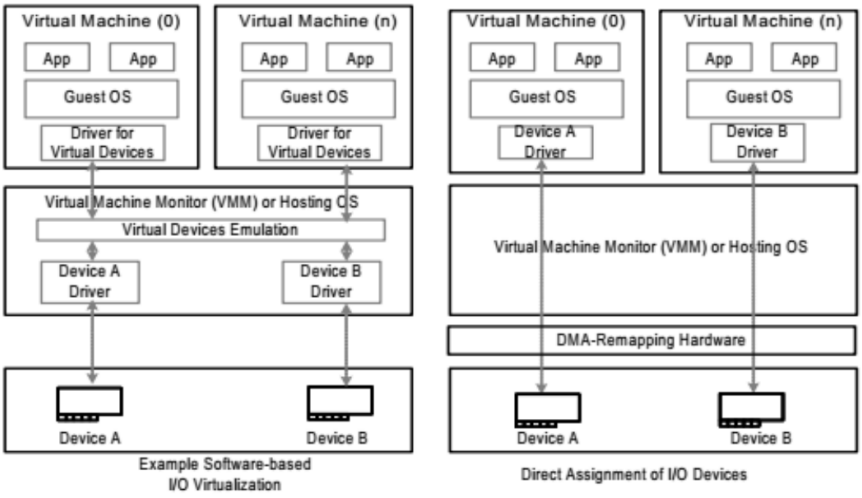
I/O emulation poses the biggest performance challenge in virtual machines: VMM has to participate in every interaction between the VM and the I/O peripherals (often emulated), thus resulting in several VM exits.

In order to reduce the number of VM exits, the solution is to give direct access to I/O device to the VM. In this way there is no need for translation or intervention of the VMM → **hardware passthrough**.
In order to implement this approach explicit hardware support is required.

This eliminates completely the need for VM exits, however, the hardware is not virtualized, and it must be dedicated to one VM and not shared.
This situation breaks the full virtualization isolation principle since it get access to a real device and not to a virtual representation of it, but, it has a significant advantage since no context switching are needed and I/O operation are speeding up.

There is the need of two functionalities:

1. **direct mapping of the I/O physical memory space to the VM memory space** (guest physical addresses), so guest OS can write directly to the registries of the I/O device.
2. **direct assignments of the interrupts linked with the peripheral**, so that the VM can handle the interrupts coming from the passed-through device without the intervention of the VMM.



I/O Mapping

In the VMCS we have an **I/O bitmap** with one bit for each possible I/O address in order to completely support hardware passthrough. In this way the VMM can set the bits corresponding to the addresses of the peripheral that we want to passthrough (to grant direct access).
When in non-root mode, if an operation in the I/O space is made, the CPU checks the I/O bitmap of the VMCS of the current VM and if the bit corresponding to the address of the current I/O operation is set the instruction is completed, else the control is given to the VMM (that will emulate the hardware operations).

Interrupts from peripherals:

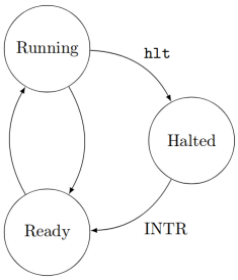
The interrupts are managed according to the device relation with the VM:

- if the device is managed by the VMM, when an interrupt arrives, a VMEXIT is fired and the interrupt is managed by the VMM.
- when a device is passed-through to the VM, the interrupt is handled directly by the VM without a VM exit: the CPU directly looks (via hardware) into the VM guest OS's IDT and executes the corresponding handler without VMM intervention.

That is implemented via the Interrupt Remapping Table.

The VM might receive an interrupt while not running, to this aim the system needs to handle the interrupts coming from the passed-through device also when the VM is not running.

Virtual Machine States



A VM can be in three states: **running**, **halted** and **ready**.
VM can switch between running and ready for VM exit or scheduled decision by the VMM scheduler. VM switches from running to halted by the `HLT` instruction encountered during VM code execution. When this occurs, the VMM intercepts the instruction and places the VM into the halted state. Resumption from the halted state is possible only if the VM receives an interrupt triggered by the `INTR` instruction (interrupt request) directed to the VM CPU.

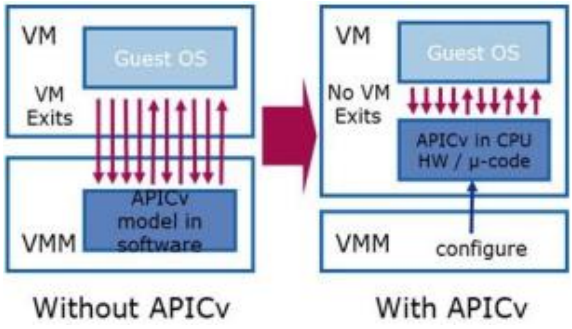
An interrupt originating from a passed-through device should always be handled by the VM. The specific actions taken will depend on the VM's current state:

1. **VM running:** the interrupt must be handled directly by the processor, jumping to the guest OS interrupt handler, without VMM intervention.
2. **VM ready:** the interrupt should be stored so that the CPU can handle it when the VM goes in running state.
3. **VM halted:** the interrupt must be stored and the VM status must be changed to ready.

To implement the above rules the **posted interrupts mechanism** is exploited.

Posted Interrupts

The posted interrupt mechanism allows to store interrupts (post) signals for later notification to the VM, without VMM intervention. This mechanism leverages, in addition to the previously discussed IRT, an additional data structure called the Posted Interrupt Descriptor (PID) for each VM.



Interrupt Remapping Table (IRT). Contains the correspondence between each type of interrupt and VM (in form of PID or request to interrupt vector of a VM) → one IRT for all the VMs.
Every time an interrupt is triggered, the IRT decides which Interrupt Vector or PID handles the interrupt.

Posted Interrupt Descriptor (PID), are used in the IRT and contains information on the status of the VM and stores the information on the interrupt notifications for asynchronous notification. One for interrupt associated with the passthrough device.
PID contains all the data structures required to implement the posted interruption mechanism:

- **Posted Interrupt Request** (**PIR**) structure in which interrupts are posted.
- **Suppress Notification** (**SN**) flag, indicates if after posting the interrupt in the PIR the controller must notify the CPU (SN=0) or not (SN=1).
  It is set to 1 (do not notify) when the VM is already in ready state.
  If the SN is 0, and the VM is in halted state, when an interrupt arrives the VM is switched to ready state (VM awakening). When the VM is running, SN is 0.
- **Notification Vector** (**NV**): pointer, points to the actual interruption vector to notify interruptions to the vCPU of the VM. When the VM is in running state, it is set to Active Notification Vector, that points to the IDT of the VM.
  → The VMM has to update the notification vector value accordingly to the VM state:
  - **VM switches to running:** SN is set 0 and NV is set to Active Notification Vector.
  - **VM switches to ready**: SN is set to 1.
  - **VM switches to halted**: SN is set to 0 and NV is set to Wakeup Notification Vector that triggers VM awakening.

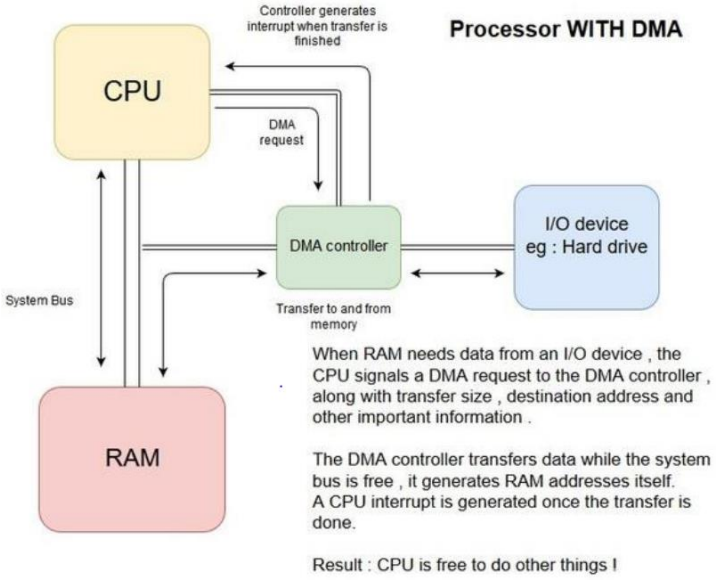
When the VMM changes the state of a VM to running, it must also look at the PIR, if there is any bit set, it must set accordingly the NV when entering the VM. This way the processor will process the interrupts that were posted while the VM was not running.

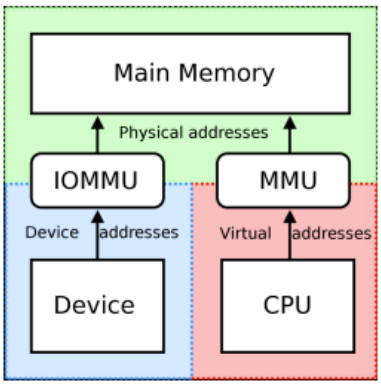
--- Please study the PDF file titled **'POSTED INTERRUPTS'** provided by the professor. ---

DMA Devices and IOMMU

DMA (Direct Memory Access) devices are I/O devices that read data directly from RAM.
These devices are initially instructed by the CPU through the writing of specific addresses, indicating where to write or read data directly in physical memory.

In order to do that for those devices an additional component is introduced and that's not related with virtualization: the **DMA controller**, it is an additional piece of hardware which is responsible of moving data from RAM to device.



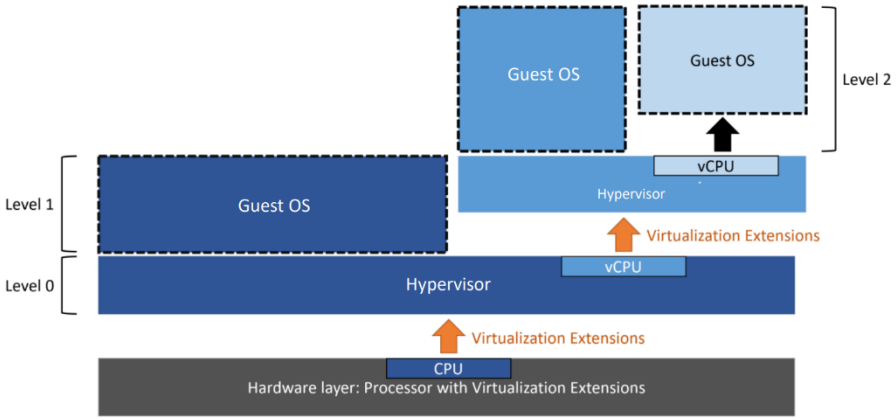


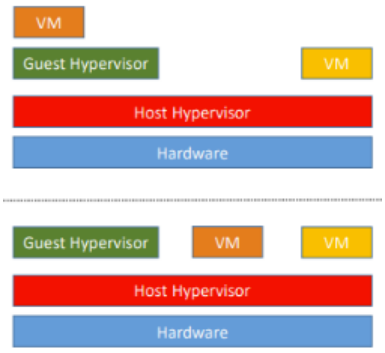
In order to translate Virtual Addresses to Physical Addresses, the DMA devices exploits **IOMMU**, a specific MMU unit for I/O Devices. IOMMU was initially projected for multiprogrammed environments that adopted virtual memory. However, it can also be leveraged for virtualization purposes → capable of triggering page faults interrupts if finds a page that is not stored in RAM.

Nested Virtualization

Is when a hypervisor runs inside a VM which is running inside a host hypervisor.

This technique is used for testing and also for production environments (since reduces the efforts for upgrading the infrastructure). Allows cloud consumer to setup their own IaaS infrastructure on top of a set of VMs running on a IaaS infrastructure.





Multi-level nested virtualization requires specific hardware support. Intel and AMD only support one virtualization level.

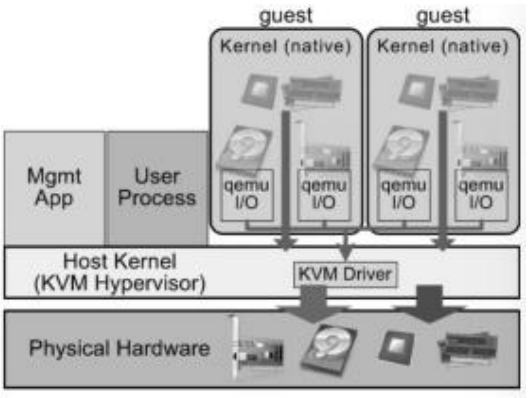
→ Multiple virtualization levels can be multiplexed into a single virtualization level. The host hypervisor emulates the guest hypervisor's access to hardware extensions for virtualization. In this scenario, the virtual machines (VMs) running on the guest hypervisor are treated similarly to other VMs operating on the host hypervisor.

Guest hypervisor access to hardware extensions for virtualization are emulated by the Host hypervisor.

KVM Kernel-Based Virtual Machine

One example of hypervisor that also implements nested virtualization is KVM (Kernel-based Virtual Machin) which is a virtualization infrastructure of the **Linux kernel** added in **2007**.

KVM allows to exploit hardware virtualization support to run VMs inside an execution environment nearly identical to the physical hardware. In fact, KVM doesn't run like a regular program in Linux; it is an integral part of the Linux kernel.



KVM runs as a driver handling the new virtualization instructions exposed by the hardware.

KVM exposes a device `/dev/kvm` that allows a user space program to setup and configure a new VM.

Kvm-userspace: is a modified version of QEMU that exploits the kvm API to run the guest VMs.

Lab 1 - B04 (Lab infrastructure, VPN, Full Virtualization)

`kvm-ok` -> ~~since we are connected to a VM, that VM cannot exploit KVM acceleration~~.

How is possible that hardware acceleration is available on our VM? The platform we have this year exploits nested virtualization and is based on KVM, so we have two hypervisor communicating each other and so somehow the VM has available hardware acceleration.

`sudo apt-get install qemu-kvm libvirt-daemon-system libvirt-clients bridge-utils virtinst`
This command will download lot of packages among which KVM and LibVirt.

Libvirt: is an opensource tool for managing platform virtualization (create and configure VMs). Is the program that we will use to create or destroy VMs.
→ KVM allows the creation of VMs, Libvirt manages them.

With Libvirt you can import preinstalled VMs images by specifying the image file.

Network Connection of VMs: the VMs created with Libvirt are added to a virtual bridge called virbr0.
By default, the interface is configured by Libvirt in order to offer NAT services to VM.

Linux Bridge: is a mechanism that allow the creating of virtual interfaces on Linux system, and those virtual interfaces can be exploited to bridge multiple network adapters so you can create a single network out of them. it's a mechanism that has nothing to do with virtualization (it was created before).
So it's a virtual interface that bridges different interfaces of a Linux system (both real and virtual ones). Is performed at layer 2, it is like connecting via software different networks through a physical switch.

Example of bridges of the host system:

`brctl show`

The system has one bridge created by LibVirt to connect VMs

The virbr0 merges two networks: vnet0 and virbr0-nic
vnet0 connects to eth0 of the VM directly
virbr0-nic is a virtual network that is used to connect all the VM hosted by the local LibVirt instance

```
root@HAJJVX8OPD7M5QO:~# brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.525400683305       yes             virbr0-nic
                                                        vnet0
```

The virtual NIC of each VM has a corresponding virtual interface on the host OS. Virtual interfaces can be bridged in order to connect VMs (creating a virtual network).
→ the bridge between VMs can be connected to a physical interface to allow data exchange with external networks.

Libvirt allows to create new network interfaces through the command `virsh net-define`.

Storage pool: Libvirt allows also to add virtual storage devices to VMs through the command `virsh vol-create-as image "NAME" SIZE`.
→ you can attach disks to VMs through
`virsh attack-disk "VMNAME" "DISK IMAGE PATH" vdb --live`.

B05 Paravirtualization and Operating System Level Virtualization

Paravirtualization

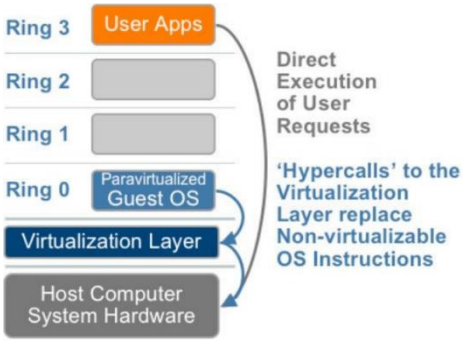
In **full virtualization**, the guest OS can run without modifications. However, instructions within the guest OS that cannot be executed directly are emulated by the VMM, resulting in frequent context switching and a noticeable overhead.

An alternative approach, known as **paravirtualization**, was introduced before the advent of hardware support. In a paravirtualized system, the VMM is implemented with the assumption that the guest OS can be modified to be aware of running inside a VM. This modification reduces the need for VMM interventions and involves editing the guest OS code. These modifications typically occurred at the kernel level, while guest OS applications at the user level remained unaware of the virtual environment.

Paravirtualization was commonly used before hardware-assisted virtualization became prevalent but is now somewhat less utilized.

Architecture & Implementation

The goal is to introduce modifications to the guest OS that prevent the OS to execute privileged instructions or implement some functionalities in a manner that does not require emulation. Such instructions or functions are substituted with the call to specific APIs exposed by the VMM through an interface named **Virtualization Layer**.



Those calls functions to the hypervisor from the guest OS kernel are called **hypercalls**. The hypercalls are executed in system mode, so that those can interact directly with the hardware.
Of course a context switching is still required, but the process is optimized since only one context switching is performed (optimize respect to the trap-emulate approach, where multiple switch are necessary).

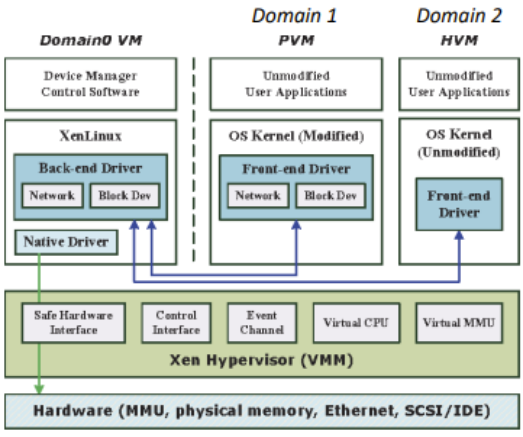
PRO: It improves significantly virtualization performance, as emulation is avoided. It does not require specific hardware support.
CON: It supports only modified version of the OS. Some closed source OSs cannot be supported.

Paravirtualization systems, e.g. XEN, were very popular at the beginning for the considerable performance increase. With the advent of hardware support for virtualization they no longer have a significant advantage over full virtualization, consequently they are not popular.

Xen

A popular paravirtualization based on paravirtualization system was **Xen**. Released in 2003. Nowadays Xen implements both paravirtualization and hardware assisted virtualization.

It was build before the introduction of hardware virtualization support, to reduce the overhead of full virtualization. It tried to find a compromise between rewriting the kernel of the OS and extensive emulation.
Xen is based on a set of modifications of the Linux kernel so you can run a Linux OS in an efficient manner in a paravirtualized manner.



Xen is a small kernel that isolates the real hardware and is placed between the VMs and the hardware.
Each VM is in a **domain**, and these domains are isolated from each other.
The special domain 0, also known as **Dom0**, is responsible for creating and destroying other VMs. Dom0 has privileged access to XEN APIs and can host any Linux operating system.

Domains can be given direct access to some I/O devices, or they can use fully virtualized devices, or they can use paravirtual devices.

Virtualization overhead → lightweight virtualization

One use case for virtualization is the **safe sharing of hardware resources among different applications**. Application isolation should be provided by the OS itself, however, recently Oss become less and less effective in ensuring isolation. So the best way to achieve this is to run each application in its own virtual environment. For this kind of uses, it can happen that we have many VMs of the same OSs running on the same hardware.
In this case multiple instances of the same kernel, libraries and dependencies will run on the same machine.

To reduce this kind of overhead new technologies of **lightweight virtualization** were introduced. The goal is to do not require the virtualization of an entire system and still guarantees the same advantages of virtualization:

- guarantee **isolation**
- **dynamic instantiation**
- **self-contained environment**

Operating System Virtualization

Operating system virtualization, is a type of virtualization that falls under the category of lightweight virtualization approaches. This form of virtualization differs from the ones we have discussed so far in that it does not rely on an hypervisor.

The new approaches are the Operating System Virtualization, aka **Shared Kernel Approaches**.
Its goal is to have a lightweight approach that minimizes the overhead of running multiple VMs with the same OS on the same system. With OS virtualization, the hypervisor is removed: **virtual servers** are enabled by the kernel of the OS of the physical machine.

The kernel of the host OS is shared among the virtual servers running over it, that are enabled from the kernel of the physical machine itself. Since the OS is shared, it has to implement logically distinct **user space** instances, meaning they can have separate file systems, environment variables, processes, etc., but share the same core hardware resource management offered by the kernel.

PRO: Those reduce the overhead since there is only one kernel, and for that one machine can handle more VMs. This is because a lot of overhead is removed: the hypervisor, many context switches and the emulation of the hardware.
CON: On the other hand, all the VMs have to share the same kernel, and not all the OSs offer virtualization solutions.

This solution is implemented by **FreeBSD Jails** and by **Linux Containers** for example.

Difference between kernel and operating system

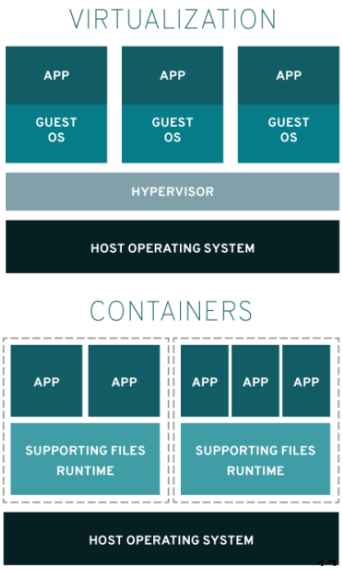
The **kernel** acts as the heart of an operating system (OS), focusing on essential tasks like memory management, process scheduling, and direct hardware communication. Essentially, it's what makes the interface between hardware and software possible. An **operating system**, includes the kernel as its core component and extends its capabilities with a broader suite of features such as graphical user interfaces (GUIs) for easier user interaction, utility programs for system maintenance, and device drivers to support a wide range of hardware, offering a comprehensive system that users and applications can interact with more intuitively and effectively.

Containers

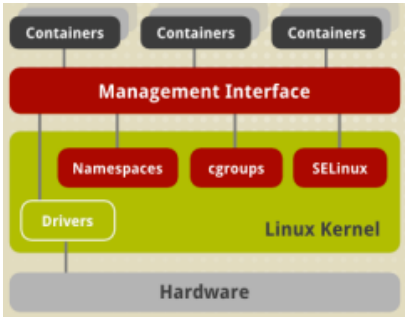
Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine. The machine that the containers can access may feature a subset of the physical hardware resources, like less memory or less CPU.

So containers are not virtual machine, they are only a virtualized environment where a set of constrains are applied to the set of processes.
There is no guest kernel running inside the container, the kernel of all the containers is shared and is the one of the host OS → processes running inside a container are normal processes running on the host kernel.
So the container must be made to use the same kernel as the host → you cannot run an arbitrary OS inside a container.

BIG PRO: there is no performance penalty in running an application inside a container compared to running on the host.



Linux Containers



Most widely adopted implementation of OS virtualization. They exploit **namespaces** and **control groups**.

Those features ensure that a container is fully isolated and can only access the resources (file system, devices, other processes) they are allowed to.

Namespaces

A way to segregate system resources so that they can be hidden from selected processes. It is an extension of an old Unix function `chroot()`, a system call that allows to specify a portion of the filesystem to which the application is confined (a subtree of the file system). The specified subdirectory becomes the root for the process. This function works for the filesystem, instead namespaces were made to extend this behavior to other resources of the system (network interfaces, PID namespaces (to hide processes)).

Control Groups

Even if namespaces prevent one process to know which other processes are running, a process can interfere in other ways, such as for example by abusing of system resources (i.e. using too much memory, too much CPU time or network bandwidth).

Control groups are groups of processes for which the **resources usage is controlled and enforced**.
They are created by root user, each process must belong to a group to which it cannot escape. When a process creates a child process, its child is assigned to the same control group (**control group inheritance**). Each group can be restricted in the access to any resource (i.e. memory RAM, CPU time, …).

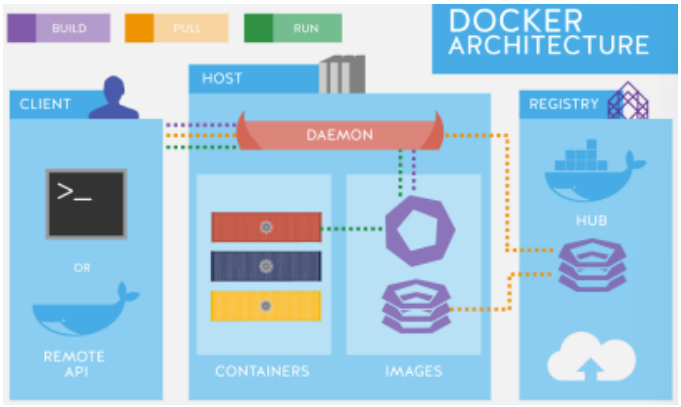
Docker

Docker is an example of a system that exploits Linux containers to create a lightweight virtualization system for software deployment or distribution. Docker exploits OS level virtualization to deliver software packages into containers.

This is very useful for software distribution since they provide an isolated bundle in which software, libraries and configuration files can be included. A Docker package is ready to run, since it does not need the installation of any dependency or library.

Docker architecture is oriented to software shipping which means easy deployment and easy installation.
Docker allows to instantiate containers on a system on the base of a Docker Image, downloaded from a centralized repository called **Docker Registry**.

The host on which we want to run the containers must have the **Docker Engine** installed, which is in charge of downloading and run the images from the registry.
The Docker Engine exposes an interface to users to select and install software packages.



Serverless computing

Serverless doesn't means that you don't need a physical server, but this refers to the absence of VMs.
In fact, containers enabled a new type of cloud computing model called **Serverless Computing** or **Microservices**. Cloud consumers do not have to create full VMs to deploy their services on the cloud, but they create containers with their software running into an environment with software dependencies and libraries pre-installed and ready to use.
Cloud providers offers services based on this model, with cloud platforms projected to support it.

Lab 2 - B06 Docker

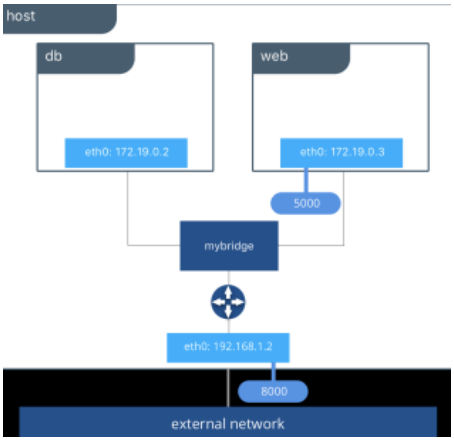
Docker: a container technology that allows to package up an application with all the parts it needs.
Containers: are created from locally available images.
Images: are created from standard images, which can be downloaded from public repositories.
Docker Daemon: controls all operations.

`docker run`,
`docker search "container"`,
`docker pull "container"`,
`docker run "container"`,
`docker ps -a`,
`docker rm "ID"` (deletes a container),
`docker images` (lists locally available images),
`docker rmi "ID"` (deletes an image that is available locally)

Dockerfile: to create a custom docker image, describes the set of customizations to be performed.

From the Dockerfile you can build the image: `docker build -t custom-dockerimage .`
Then you can run it as any image: `docker run custom-dockerimage`

Docker Networking



By default, each docker container has its virtual network.
The virtual network of Docker containers is made to allow communications among different containers. (containers on the same node can communicate via their private IP addresses).

You can bridge virtual interfaces to connect containers, or you can route the virtual interfaces to connect containers. By using port forwarding you can publish a service inside a container to the external network.
→ **EXPOSE** command in the Dockerfile can be exploited to implement that.

`docker run -p 80:80 nginx-ubuntu` (to bind host port 80 to container port 80). The docker run command does not exit, leave the terminal open. To run the container detached from the terminal use the option **-d**.
Use `docker network inspect bridge` to retrieve to the list of IPs corresponding to the running containers.

Cloud Applications

C01 Cloud applications - structure and architecture

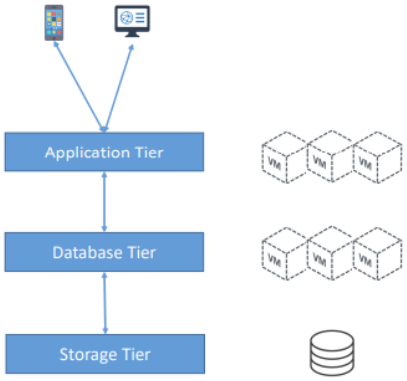
RECAP OF WHAT WE KNOW OF THE CLOUD:
Summarizing, the cloud is an infrastructure which provides a certain amount of virtual resources that are used by the cloud provider to create in general VMs. Each VM has a certain amount of virtual resources that are used to implement applications and services.
VMs also have a virtual network adapter, through which they can communicate with other VMs via a (virtual or physical) LAN or with external hosts via the Internet. VMs can have access to a cloud storage, a physical or virtualized storage shared among all the VMs usually accessed via the LAN that connect the VMs.

Traditional Applications: it's the approach used before the advent of cloud computing. In this model, a physical server hosts (runs) the service, storing data locally on its hard disks. Clients, typically communicating over a LAN or WAN, are relatively simple in this setup.

Cloudification: services have been moved, as they are, on the cloud (without changing the implementation), transferring data and services from physical hardware installed on premises to VMs in the cloud. The clients communicate with servers (that now are virtualized, executed in VMs) over the internet, same way as in traditional applications.
This process does not create true native cloud applications. Therefore, the benefit of the cloud are not fully exploited (high scalability, reliability, capacity to handle big data) -> it is needed to redesign applications.

Cloud Applications Architecture

Cloud Applications are usually implemented as **multi-tier systems** -> different functionalities split among many VMs, each one running a specific functionality.
Basic example **two-tier**: one tier for application logic, one tier for database system.
From this we can implement a **three-tier** implementation, by splitting the database tier from the data storage tier. Each tier can be formed by one or more VM.



One of the VM in the application tier receives the request from the client, process it. If it needs to retrieve some data it contacts one of the VM in the database tier. The service inside that VM retrieve the data by interacting with the storage tier.

The result in general is a multi-tier architecture in which the service is provided to the client as the result of a **chain of interactions** among servers of different tiers.
The core-tier servers communicate in response to client request to first-tier application servers.

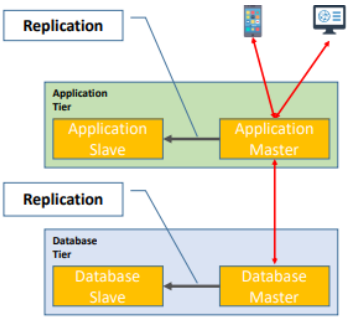
The servers are connected through a **LAN** and use an application protocol which can be implemented by a **middleware** software **to simplify the development**.

Compute Clusters

Is a set of VMs belonging to the same tier. That set is organized to perform a collective computation over a single large job or a set of single requests. The nodes within the cluster do not exploit much I/O, except for data transferring among nodes of the tier itself. They usually exploit **middleware** software for communication within tier or to other tiers.

Generally, the architecture of a cloud application is determined by its objectives and core requirements. While there are guidelines for implementing use-case-specific architectures, the structure of these systems can vary to align with the overarching purpose defined by the cloud architect. Opting for a clustered approach instead of a single VM running the software implies strategic considerations for scalability, resilience, or performance.
There are three distinct types of clusters designed to meet various operational needs:

1) High-Availability Clusters: (HA Clusters) clusters designed to be fault tolerant and support the execution of services that require to be quite always available. They operate with redundant nodes to avoid single point of failures.

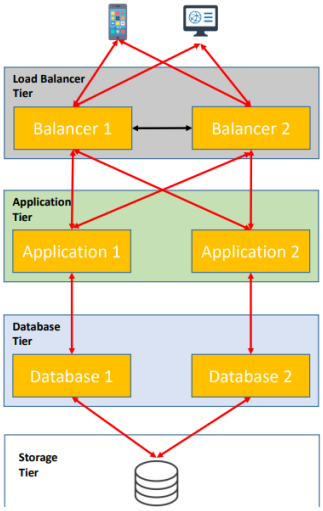


The simplest HA cluster is a cluster with only two redundant nodes that can fail over to each other. High redundancy implies high availability, so many nodes of the cluster should be capable of implementing the same operations or providing the same service.

Inside each tier usually there is the **master** node. The others are **slave** nodes. The master node answers to the requests coming from other tiers. If the master fails, there is a mechanism of **election** or a **static order** to select the new master. In order to be able to offer the same functionalities of the master, some form of replication among the nodes in the cluster is necessary for data, settings, and configuration.

CON: resource inside slave nodes are kind of wasted, because most of the time they remain unused.

2) Load Balancing Clusters: (LB Clusters) are designed for higher resource utilization.
The main goal is scalability, so all the nodes share the workload.
Incoming user requests are distributed across all node computers of the cluster in order to balance the load on each one. The result is a balanced workload among different VMs, which should lead to a higher resource utilization and higher performance, such as lower response time.



This is achieved by implementing another tier, the **load balancing tier**, between the application tier and the clients. Within this tier, incoming requests are received and subsequently directed to one of the VMs in the application tier, following a predefined policy (such as selecting the least loaded instance or using a Round Robin approach).

Within the load balancing tier, there may be multiple load balancers, and they must be identical. Clients have the flexibility to send their requests to any VM within the load balancer tier. The application tier possesses the intelligence to determine which node in the application tier (and which database node) should receive the request, ensuring a balanced distribution of the overall load.

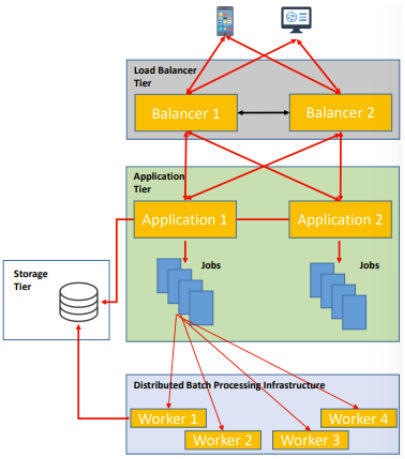
Data Synchronization

Also in this case we need to synchronize the data among the instances on the same tier VM the (each of the existing tiers). The reason for this is the same as we have for High Availability Cluster, namely, a request must be answered consistently regardless of the path it takes within the application. In other words, every component of a tier must respond in the same manner to the same request to ensure consistency. To achieve this, **data synchronization** is necessary across the VMs that belong to the same tier.

To achieve data synchronization, one option is to implement **data replication**, but it introduces significant overhead. Another approach, to avoid the replication overhead, is to introduce a **data storage tier** based on a cloud storage solution (we will discuss this further in the last part of the course), to which is demanded the complexity of managing data and ensuring that the same data is accessible from all instances.

3) Compute Intensive Clusters: (CI Clusters) designed to handle and analyze large amounts of data. Considering the amount of data, its analysis on one single server would take too much time. CI clusters adopt a "**divide et impera**" approach:

- the analysis task is divided into simpler sub-tasks (**jobs**)
- **data** is divided into smaller **chunks**, each assigned to a job
- one job and one chunk are assigned to one server for analysis (**worker**)
- the result of the worker is returned to a particular server (**collector**)
- the collector merges results from workers nodes together



CI clusters also need to implement a **load balancing tier** if they are exposed to clients (i.e. search engine). The initial dataset is huge, cannot be synchronized among all the nodes of the cluster, so a **distributed storage tier** is adopted.

To limit communication overhead, the batch processing infrastructure can have access to the storage tier: different workers can have direct access to the data to be analyzed. The application is then required to provide one pointer to the data without transferring it.

Conclusion

One of the big advantage of cloud technologies (and virtualization) is scalability. **LB** and **CI** architectures, with a load balancing tier, can easily scale horizontally by allocating or deallocating VMs in response to changes in the current traffic. In the **HA** architecture, instead, horizontal scalability can be exploited to improve redundancy.

In general the first tier is also called **frontend tier** (the one which is also in charge of communicating with the clients), while the others tier are called **backend tiers**.

C02 Cloud applications - design methodologies

In general cloud applications are distributed systems, made of different components interacting each other in order to offer an application to users. The implementation and integration of different software components for the cloud environment is a complex task that can result to be time consuming and expensive. In order to avoid these issues, a software component (part of a cloud application), should be designed with the following requirements:

- **Interoperability**: software should expose a standard interface that can be easily invoked by other components
- **Scalability**: should be ready to be integrated with a growing number of components. Must easily scale horizontally dynamically and effortlessly
- **Composability**: a cloud environment is a dynamic environment that can change rapidly, e.g. due to failure of components, creation/destruction of VMs, etc. A software should be designed in order to allow its usage with different components without (or with minimal) changes

Service Oriented Architecture (SOA)

A modular approach for designing software objects for native cloud application. In SOA each software component is called a **service**, that must be self contained and must implement a specific functionalities.
In the Service Oriented Architecture applications are built as a collection of services (or *microservices*) made from existing software components. Applications are highly modular rather than a single massive program. Services can be developed in different languages and run on heterogeneous hosts in distributed manner.

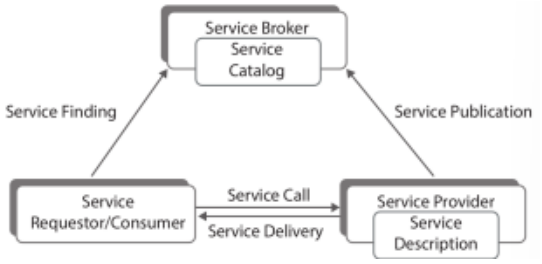
Communication between services

No technological dependency, like it was in Remote Method Invocation (RMI), where an interface is implemented remotely on the software component host. This method introduced high coupling and technology dependency.

The communication between the services is implemented via **message passing**, so there is no technological requirement or dependency, because there is a **message schema** that defines how you should interact with a specific component and how it must behave to the message and which messages it should handle.

Every message is received and then handled by the service in a manner completely hidden from the exterior.
Applications can be implemented in different languages, as they only need to implement the message schema and behave accordingly. Of course the implementations of a component can change over time, the only thing that must remain equals is the message architecture. High hiding of the internal "status".

Two roles in message passing: **consumer** service and **provider** service. A service can be a consumer, a provider, or it can be both, depending on the implementation.
In general, service providers and consumers do not interact directly with each other initially, as the consumer needs to know the address and descriptor of the provider. To bridge this gap, an additional software module, the service **broker**, is typically employed in the middle.
The service broker is responsible for managing the **service catalog** (or service registry), where all the available services are enlisted, and it provides the consumer with the necessary information about the address and descriptor of the provider.
A client service can discover a provider service only if the interface is published to the broker.



What happens is that the service broker implements a **catalogue** where all the services that are available inside the application are written. A **provider** at the beginning of his lifetime contacts the broker in order to get registered and communicate the services that he offers and the format of messages that he expects, on the other side the **consumer** who would like to trigger functionality, however, he doesn't have the address of the provider, he only has the description of the functionalities he would like to invoke. So as the first step, the consumer contacts the **service broker** in order to request that functionalities send the description of that, the service broker looks in the catalogue and returns the address of one of the providers who offer that functionality. After receiving the address, the service consumer can contact the provider directly in order to invoke that functionality by sending a request message, the provider will receive this request, will parse it and produce a response that will be sent back to the service consumer.
Of course in this approach, the address of the broker and the structure of the messages that the broker is expecting are well-known.

SOA Advantages

- **hidden implementation**: as long as the interface remains the same (the message format the server is expecting), the service can be modified without interfering with service consumers (/other services)
- **loose coupling**: thanks to separation you can plug and unplug services from an application effortlessly
- **composability**: a system can be assembled as the combination of existing services
- **stateless communication** between modules: the message passing communication ensures that each request is treated **transactionally**

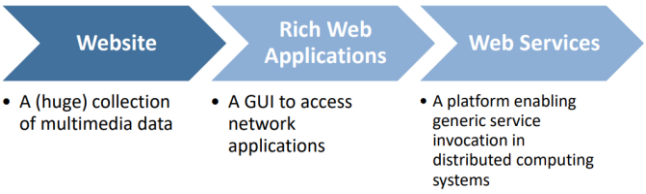
stateless + loosely coupling gives another advantage: **scalability**. This means that the components inside of a tier can be created or destroyed at runtime depending on the load of the system.
When a component is created or destructed, it must contact the broker so that the catalogue is always up to date.

Web Service

Web services is the most common example of the SOA methodology. Self-contained, self-describing modular application designed to be used by other software applications across the web.
W3C definition of web: "software system designed to provide interoperable machine-to-machine interaction over a network".

Everything started with the World Wide Web technologies, before moving to web services the web itself originally included several mechanism, in particular three of them:

- identify a resource over the internet in a unique manner
- to exchange data between entities
- a language that defines how information should be encoded to achieve interoperability



Web services are the natural evolution of the WWW and are now the core of a web application, operating below the graphical user interface. Now, let's explore how these three mechanisms, which we have discussed, were implemented.

URI Uniform Resource Identifier URI (to identify resources)

This is a mechanism to identify object or resources on the internet.
"is a compact string of characters for identifying an abstract or physical resource"

```
<uri> ::= <scheme>":"<scheme-specific-part>
```

URI are also the **URN** (Uniform Resource Name) and also the **URL**.

HTTP (to exchange data)

The second mechanism is a protocol for ensuring interoperable communication across different software components, the HTTP is an example of this type mechanism created for the web which allows communication between web server and client.

HTTP := "… is an application-level protocol for distributed, collaborative, hypermedia information systems".

XML (to encode data)

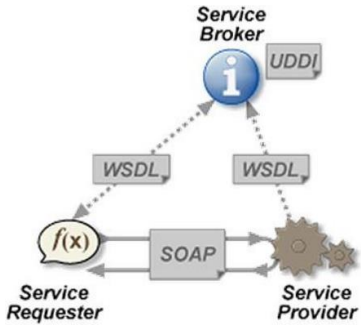
In the web the data encoding part was standardized as **HTML** which is a markup language specifically designed in order to encode data about web pages (layout, information, images, video, …). This standard was too much specific for the web, so it is not a very good encoding language for web services which requires something more general, for this reason, the eXtensible Markup Language (XML) has been introduced.

Defines a set of rules for an encoding markup language that is both human and machine readable.
XML specifies the **syntax**. XML is **extensible** because the tag set is not pre-defined.
XML is in a **hierarchical** structure. An XML **element** is a document component inside a begin tag and end tag.
The XML root element is named **XML declaration** and gives some information about the document itself. A tag is inside < and >. Inside a start tag you can find **attributes**, that are key-value pairs.

XML can only state if a document is **well formed** (if it respects the syntax). If we want also to know if the document is **valid** (/ it has a meaning) that XML document must refer a **Document Type Definition** (DTD) or an **XML Schema** (the newer version). DTD and XML Schema give a schema of the architecture of the XML of the document: specify what kind of information a document can include, what attributes and elements.

Web Services – Interaction (SOAP and WSDL)

How do we exploit all these components for creating web services according to the service-oriented architecture?
The first component we require is a component to specify the interaction between consumer, producer and broker. In order to standardize interaction with a broker the Web Services Description Language (**WSDL**) has been chosen.



Each Web Service must have an **interface** described in WSDL. Via WSDL each service provider register itself to the broker, while the service consumer uses WSDL to look up for a specific service.

After this initial phase where the consumer look up for a functionality, we have a direct interaction between the producer and the consumer and in web services that happens with another language named Simple Object Access Protocol (**SOAP**), that provides a standard messaging format.

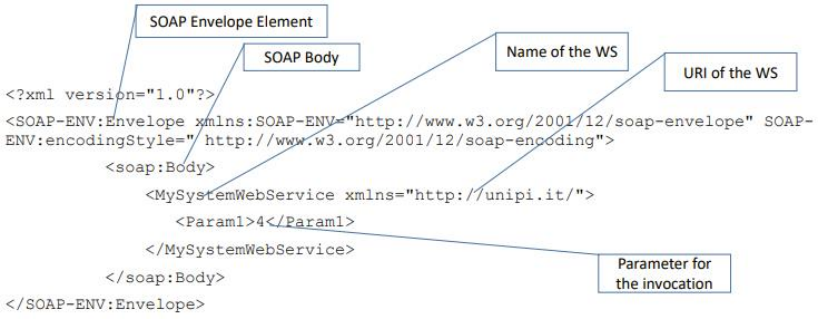
SOAP - Simple Object Access Protocol

SOAP provides a standard packaging structure for transmission of XML documents over various internet protocols (like HTTP). The structure depends on the type of the application.

A **SOAP message structure** is called **envelope** and is made of a header and a body. The **header** contains security and specific information, while the **body** includes the message payload (request or response) and a faults and errors section.

Typically, SOAP messages are bound inside HTTP application level protocol (**binding**). Usually are exploited GET and POST http methods.

With GET, only the response contains a SOAP message usually, while in POST, both request and response contain SOAP messages.



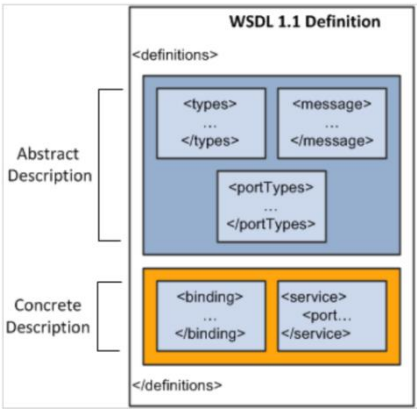
WSDL - Web Services Description Language

WSDL describes the interface and the set of operations supported by a WS in a standard format. It standardizes also the set of input and output parameters of the service operations as well as the service protocol.

WSDL message architecture

contains in machine readable format:

- what a service can do
- where the service is located
- how it can be invoked



WSDL message is an XML document and it's made of two main parts:

- **Service Interface Definition**, which defines in abstract manner the service. Comprises the type of data exchangeable (field "**types**"), the messages that can be exchanged (field "**message**"), and the operations that can be performed (field "**portType**");
- **Service Implementation Definition**, which defines in a specific manner how the service is invoked. Comprises how to transfer the message (field "**binding**", i.e. which protocol to use, like http), and how to invoke the service (field "**service**", that i.e. specifies the URI of the soap binding).

REST: Representational State Transfer

SOAP is often considered as complex protocol, as it requires the creation of the XML structure, which is mandatory for

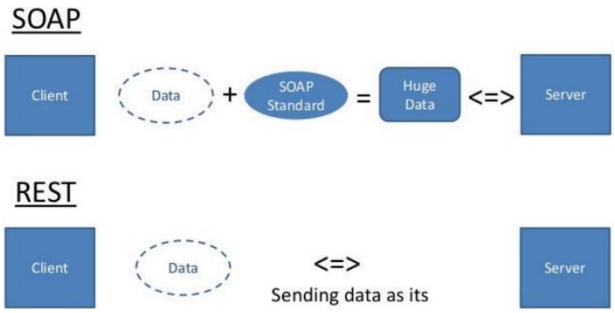
passing messages. The Representational State Transfer (**REST**) has been introduced as valid lightweight alternative for Web Services construction. Based on HTTP transfer protocol.

REST replaces both SOAP and WSDL protocols.

In REST, without WSDL or service brokers, clients find the provider's address and API details through well-documented guides, often available online, or through specifications like OpenAPI that describe the API's endpoints and operations in a standard format. There's no WSDL equivalent in REST; instead, clear documentation or interactive exploration tools help clients understand how to interact with the API.
The client sends a request using a standard HTTP method and the server answers with a response that includes a representation of the resource.

The content of data is still transmitted using **XML** as part of the **HTTP** content, but the additional markup required by SOAP is removed. Exploits the minimum support offered by http to implement method invocation.

The `GET`, `PUT`, `POST`, and `DELETE` methods constitute a minimal set of operations for retrieving, adding, modifying, and deleting data. Parameters to be included in the request, or data to be sent to the service could be included as query values in the request (parameters) or as payload (data to be uploaded). Together with an appropriate URI organization to identify resources, all the atomic operations required by a WS can be implemented.
For this reason, REST represents a lightweight alternative to SOAP, which works effectively in contexts where additional aspects beyond those manageable through HTTP are absent.



REST allows many standard formats like **XML**, JavaScript Object Notation (**JSON**) or **plain text** as well as any other agreed upon formats for data exchange.
XML is still useful to guarantee interoperability (check for example correctness of the document), but for SOAP it's too much, so **JSON** is more suitable for it.

JSON - JavaScript Object Notation

Again, XML is too much, so was introduced another encoding language called JavaScript Object Notation (**JSON**). It is a mix of text and values which is less structured than XML, it's very minimal and he doesn't have additional mechanism for the verification of the meaning.

Uses human readable text to transmit data objects. Originally defined for AJAX but is language independent. It is designed to be a lightweight data encoding, based on only two structures:

- collection of name-value pairs
- ordered list of values

What a developer need to do is to map the functionalities of his web server into a set of http invocations, every http invocation is identified by a URI and the method used of that URI. Since http has 4 main methods, on every URI 4 functionalities can be defined. As we will see in the lab, contrary to SOAP there is no standard, so the mapping of a functionality into URI+method is not something that is standardized. So every developer is free to map those, although as we will see in the lab you got some guidelines.

CRUD - Create Read Update Delete

One option to map those functionalities into URI+method, is CRUD which is a way to map functionalities into HTTP requests. In fact, the acronym CRUD refers to all of the major functions that are implemented in relational database applications: Create, Read, Update and Delete.
Each letter in the acronym can map to a SQL statement or http method. CRUD is typically used to build RESTful APIs.
CRUD principles are mapped to REST commands to comply with the goas of RESTful architecture.

Interaction is realized by **four standard actions** (**CRUD**)
- **Create** a new resource from the request data
- **Read** a resource state
- **Update** a resource state from the request data
- **Delete** a resource

HTTP methods associated to actions
- **POST**: Create a new resource from the request data
- **GET**: Read a resource
- **PUT**: Update a resource from the request data
- **DELETE**: Delete a resource

Lab3 - C03 REST interfaces

***Resource**, an object or representation of something. It has associated some data and a set of methods to operate (e.g. an Employee of a company, you can create, add or delete it)*
***Collections**, a set of resources, e.g. Employees is the collection of Employee resources*
***URL**, the Uniform Resource Locator and the path corresponding to the resource (through which the resource can be accessed)*

***Endpoints**:*

the endpoint name should reflect only the associated resource, not the action. The actions should be defined by the request method.

***Methods usages**:*

GET method is used to request data from the resource and should not produce any modification to the resource and its data
POST method is used to request the creation of a new resource, the data to be associated with the resource is included in the payload of the request
PUT method is used to request the update of the resource or the creation of a new one, the data for the update or the creation is included in the payload
DELETE method is used to request for the deletion of a resource

***Tool for designing REST APIs**: **Swagger**. -> adopt a specific language to define and describe the APIs, the language is defined within the framework of **OpenAPI**, an effort that aims at standardizing a language for REST API description. Usually the base path of a resource includes a string to identify the version of the API definition (i.e.: /v2/employees).*
https://editor.swagger.io/

***Java Servlet**: extends a web server in order to allow it to run Java code in response to an http request.*
***Apache Tomcat**: is an open-source implementation of Java Servlet*
Flask: Python framework for web services designed to produce simple REST applications.
In LAB C03 there is a tutorial on how to deploy on Docker a simple flask WS generated with Swagger.

C04 Scalable Cloud application design: Message oriented distributed systems and data replication

RECAP

Cloud applications are developed as distributed systems, whose implementation is the composition of different functionalities hosted on different VMs. Such VMs are clustered into tiers, within each tier the VMs replicate the functionalities in order to ensure high availability and scalability. Requests from clients are received by the VMs of the first tier, named frontend tier and then dispatched to the other tiers, named backend tiers.

A service oriented architecture is adopted: each VM hosts one (or more) self-contained software component, referred as service, that interact with others via **message passing**. Two different solutions have been defined for service construction, i.e. to define the way messages are exchanged between software components: SOAP and REST.

REST is the most adopted message passing system in modern applications for client/frontend communication due to its simplicity. However, **SOAP** is still adopted in the frontend-backend communication, while in the communication between client and frontend, it is hardly ever used.

Request-Reply communication

One main limitation of both SOAP and REST is that they adopt a **request-response communication protocol**.
These kinds of systems implement **synchronous communication**: the client is blocked until the reply arrives from the server (**time coupling**). This thing guarantees **reliability**: the reply is an acknowledgement of the reception of the request. It is also a **direct communication** between client and server without intermediaries (this determines **space coupling**).
CONS: Space and time coupling determine **overhead** and **complexity**.

Request-Response paradigm is good for the communication between clients and cloud front-end because client implementations can tolerate space and time coupling, while in the communication between cloud frontend and backend this is different: since the backend clusters must assure high scalability, the nodes are often allocated / deallocated, so **the coupling among nodes must be avoided** -> we need indirect communication.

Indirect communication

With **indirect communication** we want to ensure:

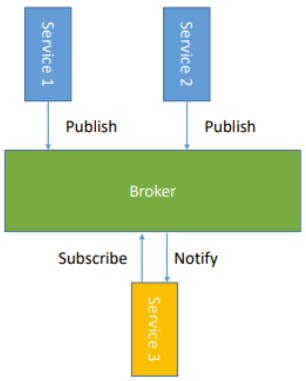
- **space uncoupling**: the sender does not need to know the identity of the receiver and vice versa.
- **time uncoupling**: sender and receiver have independent lifetimes. They do not need to exist at the same time to communicate, in this way VMs can be destroyed/created dynamically.

There are many implementations of indirect communication. In each one, source and destination does not open a direct communication channel, e.g. a TCP/UDP connection, but rather the sender injects the message into a bus.
The communication bus is implemented by an **intermediary**: requests are sent to the intermediary, that is in charge to find a receiver and to wait for the outcome (or acknowledgment) from the receiver. In this way, the sender does not need to meet in time with the receiver to communicate, the request can be processed asynchronously.

Different types of intermediary has been defined: **broker** or **message queue**.

Message Broker

A **Publish-Subscribe system** is a system where senders (called **publishers**) publish data (usually events) and receivers (**subscribers**) express interest in particular type of data. In this system the **message broker** routes the messages to the subscribers to a particular subscription, this implements a one-to-many communication.



Many types of subscription mechanism:

- **channel-based**: publishers send data to a named channel, subs receive all the data sent from the channels they are subscribed to. Channel must be defined in advance.
- **topic-based**: each published data has a set of fields, one of them denotes the topic of the data, similar to channel-based, but here topic are explicitly declared inside data.
- **type-based**: subscriptions are defined in terms of types of events and matching is defined in terms of types or subtypes of the given filter.

Message Queue

Point-to-point message exchange: senders put requests in a queue, that is consumed by the receiver. There could be just one queue for each (type of) receiver. The queues are implemented by the **Message Queueing System**.
Inside the system, different queues can be instantiated: a producer can send messages to a specific queue while a consumer can receive message from a specific one.

The allowed operations to a queue are:

- **Send**: issued by the producer to place the message in the queue.
- **Blocking receive:** issued by the consumer to wait for an appropriate message.
- **Non-blocking receive**: (aka polling) issued by the consumer to check the status of the queue, will return a message if available, a not available indication otherwise.
- **Notify**: issued by the message queueing system when a message becomes available in a queue to which a receiver (aka consumer) had subscribed earlier.
  This approach is quite similar to Publisher-Subscriber.

Messages are made of:

- **destination** the UID of the destination queue
- **metadata** a associated with the message, like the priority (if supported) and the delivery mode
- **body** of the message

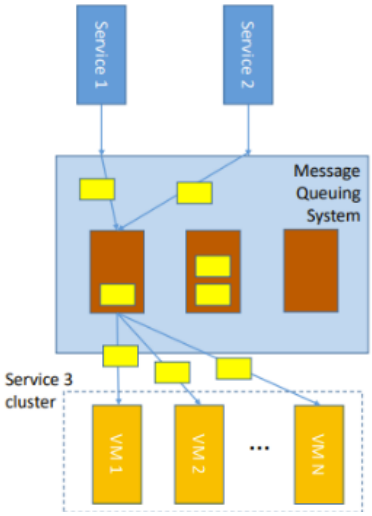
Often queues are **FIFO**: First In First Out, but usually they support also **priorities**. Consumer processes can also select messages from the queue based on properties of a message.
Messages in queues are **persistent**: queues will store messages indefinitely until they will be consumed. In order to do this the messages will also be stored to disk to assure reliable delivery.

Transformation: Arriving messages could be transformed between formats to ensure compatibility (i.e. different byte-orders, different data representations, …).

Scalability in Message Queues

Message queues are perfect for service communication at the backend of cloud applications, as they can handle the dynamicity that characterize that environment.

The queues can handle the varying number of nodes in a tier: nodes from the same cluster (in the same tier) can receive data from the same queue following some policy (load balancing, or availability). Time coupling is also managed, since consumer do not have to exist when the message is generated.



Implementations

Both message brokers and message queuing systems can be implemented in centralized or distributed manner:

- **Centralized**: the broker or the message queuing system is implemented on a single system.
  → straightforward implementation, but single point of failure, lacks performances, bottleneck.
- **Distributed**: a network of brokers/message queuing systems is adopted
  → needs data replication, more complicated, but guarantees resiliency and scalability.

The **Advanced Message Queuing Protocol** (AMQP) is one of the most widely adopted implementation for Message Queuing Systems. We will see it in lab4.

Data Replication

Since in each tier we have many nodes working on the same data, we need a way to **replicate** that set all over the cluster and keep it updated.

Generally, data replication is implemented via message exchange among the VMs of the same cluster.
When large datasets are involved, a shared storage is usually adopted: the shared storage offers a common data set that is shared among all the VMs. Even in this case, however, the exchange of the data among different VMs is only reduced as exchange of data for coordination is required.

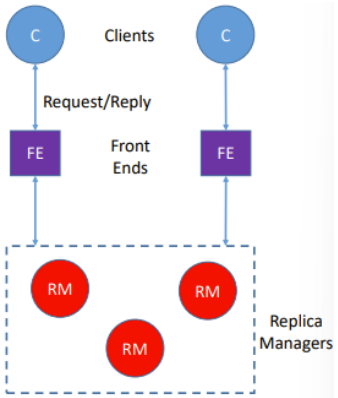
Data replication requirements:

- **replication transparency**: transparent to users, they should access different copies of data without being aware of the replication system
- **consistency**: replicas should be consistent with each other: an operation on any replica should perform the same result
- **resiliency**: from failures of VMs (at a certain extent), however, it is assumed that network partitions (outages in the communication) cannot occur, reasonable since the nodes are on the same LAN

Resiliency is crucial for data replication, in fact all the models we will discuss assume that the communication network is reliable, because every data replication strategies exploit message transmission for data synchronization and to implement coordination between the various nodes. That's why the replica managers are hosted in the same datacenter because they are connected via a extremely reliable LAN, instead if the replica managers are in different datacenters they are connected via internet which is an unreliable communication network.
For this reason, replication mechanisms that involve multiple datacenters across the world may require a distinct set of replication strategies.

System Model

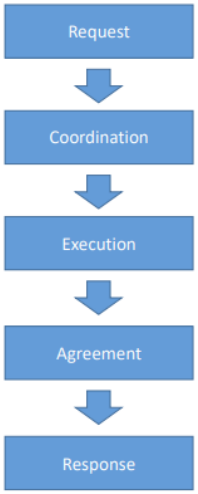
The data in the system is a collection of objects where an **object** is a file, a class or a set of records. Each **logical object** of the system is implemented as a collection of physical copies, called **replicas**.



Each replica of an object is stored in a different **replica manager**. This manager is the internal service of a node responsible for synchronizing and replicating the data across all nodes by exchanging messages with all the other replica managers.
Replica managers are installed on different component that stores an instance of the replica (in our case nodes of the same tier). To grant consistency each replica contains also the system state, not only the data.
Each replica manager accepts update operations on the data in a recoverable way, so that inconsistencies can be recovered. The clients send requests to front-end instances which forwards these requests to one or more machines that implement the back-end tier.

A request can be either a **query** (which does not modify the state of the object), or an **update** (which modifies the status of the object).

Phases to handle a request:



1. **Request:** received from the frontend module, which in turns issues the request to one or more replica managers (of the first backend tier). Depending on the implementation, the request is forwarded to other replica managers either by the frontend node or from the first replica manager.
2. **Coordination:** the replica managers coordinate to prepare for the execution of the request: they interact each other in order to understand if that specific request can be processed immediately or its execution must be deferred to maintain consistency.
3. **Execution**: the request is executed by the replica managers. (usually executed tentatively: in a way that can be rolled back).
4. **Agreement**: after the execution of the request, all the replica managers executed the request and interact with each other in order to understand if the result of the execution is the same for all the replica managers or if inconsistencies arose. If agreement is reached, the execution of the request is committed.
5. **Response**: one (or more) replica manager send the response to the frontend.

Group / Multicast Communication

Group/Multicast Communication is one of the crucial functionalities for replica managers. This communication is performed via LAN, enabling a single replica manager that receives a request to broadcast or multicast a message to the entire group of replica managers.
Group communication is utilized in multiple phases among replica managers for coordination and data exchange.

Different Multicast methods are available, each offering different levels of guarantees in terms of reliability, timed communication, and ordered communication. Among them:

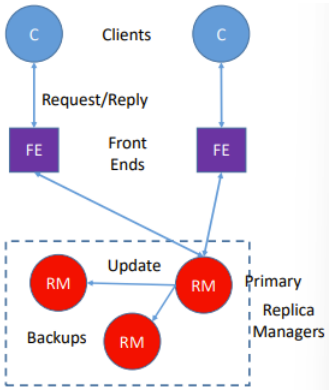
+ Complexity | + Guarantees

- **Reliable/Atomic Multicast**: this is the group communication strategy with the lowest guarantees, it only guarantees that a message is delivered to all correct members of the group or to none. No guarantees about order or delay, but losses in the communication are handled. This approach can be useful in scenarios where the order of receipt is not critical to the application's consistency or can be managed at a higher level by the application itself through other coordination mechanisms. (low complexity and guarantees)
- **Total/Casual order Multicast**: communication is reliable but it also offer some assurances in terms of ordering. No guarantees on the absolute order across different members, but every member must receive the updates in the same order. If *a* happened before *b*, then every replica manager must receive *a* then *b*. However, this mechanism does not guarantee an absolute order of the messages (absolute time of reception). It is possible that one node receives updates 1-2-3, while another node has received only update number 1 (at that moment). Thus, there is a guarantee of ordering, but this guarantee is "**per receiver**"; it does not ensure an absolute reference to the order of messages. (medium complexity and guarantees)
- **View-synchronous Communication**: with respect to each message, all members have the same view (the number of the messages received by any node node) all the time.
  If one node has received updates 1 and 2, we are sure that all the other nodes in the network received both those updates and all the nodes will receive update 2 before the first node receiving update 3 will receive that update. (highest complexity and guarantees)

Data Replication strategies

Let's see three data replication strategies.

Primary-Backup Replication (passive)

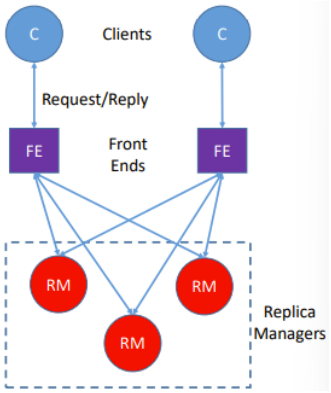


Guarantees fault tolerance. One replica manager is selected as primary, while the others are backup replicas. Frontend tier nodes communicate only with the primary replica, which is in charge of sending copies of the updated data to the backups.
If primary fails, one of the backup replicas is promoted.

Request Handling:

1. **Request**: The frontend issues the request, the request contains a UID
2. **Coordination**: The primary handles requests atomically, in the order in which they have been received. If a request with the same UID has been already processed, the same response is sent back and nothing else is done → No coordination with other replica managers is needed.
3. **Execution**: the primary executes the request and stores the response
4. **Agreement**: if the request is an update, the primary sends the updated state, the response and the UID to all the backups. The backups send an acknowledgment. **Atomic multicast** is sufficient.
5. **Response**: The primary responds to the frontend, which hands back the response to the client.

Active Replication (active)



Instead of having slaves doing nothing, we take the opposite approach: frontend nodes forward all the requests to all replica managers to execute the same request in parallel. All replies are sent to the front end.
All replica managers should reply identically, which means that if one node fails, there is no impact since all the others can reply. The frontend then sends one response to the client.

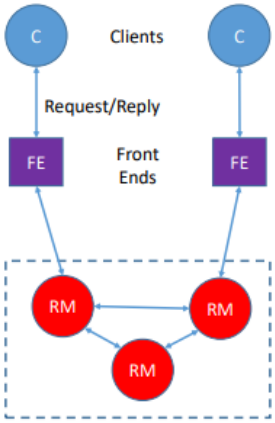
Request Handling:

1. **Request**: The request is multicasted from the frontend tier node to all the replica managers, an UID is attached.
2. **Coordination**: No coordination is needed in this phase since the request was already sent to all the replicas. The method to be adopted is **total order multicast**, so that the requests are received in the same order by all the replicas.
3. **Execution**: since all the replicas receives the updates in the same order (by total order multicast), the request is processed identically by all the replicas.
4. **Agreement**: thanks to total order multicast, there is no need for agreement.

5. **Response**: all the replicas send the response to the frontend node. Since all the response will be equal because of the total ordered algorithm for multicast, the frontend can forward the first response received.

Gossip architecture

RECAP: In **passive approach** only the primary replica is in charge of handling frontend requests and backup replica do not take part at handling users requests. (load is all on the primary).
In **active approach** the load is distributed, but in order to pursue the high availability, it is a completely redundant approach.



Gossip architecture: provide both availability and scalability and gives client access to data for as much time as possible. Updates are deferred when possible, while the control is given back to the client as soon as possible. Many replica manager are involved in handling the frontend requests in parallel maximizing utilization.

In gossip architecture replica managers exchange messages periodically in order to share with the other replicas the updates received by the clients.
Clients are assigned to specific front-end instances for load balancing.

The service provides two basic types of operations:

- **Query operations**: read-only the object
- **Update operations**: modifies the state of the object, do not read the state

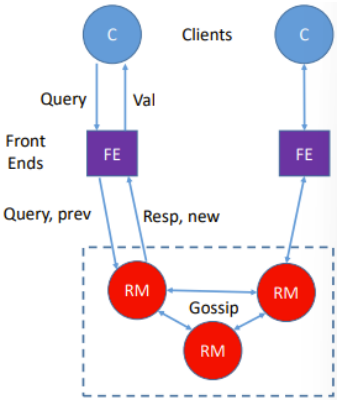
The gossip system makes two **guarantees**:

- Each client sees a **consistent** service over time (clients will always observe its own updates)
- **Sequential consistency** is guaranteed: even thought the consistency among replicas is relaxed in time to the frequency of gossip messages a client will always receive a consistent (but eventually old) information for each issued query.

Request Handling:

1. **Request**: the request is sent from the front-end tier node to just one replica manager
2. **Coordination**: the receiving replica manager will not process the incoming request until it can verify that the required ordering constraints are satisfied. This ensures that operations are applied in the correct sequence. To adhere to these constraints, the replica manager may need to wait for updates from other replica managers, which are typically disseminated through gossip messages. Beyond this exchange of gossip messages to maintain order, no additional coordination is necessary
3. **Execution**: the replica manager executes the request
4. **Agreement**: replica managers update the data by exchanging gossip messages, which contains the most recent updates they have received. The gossip messages are exchanged lazily, after some updates have been collected or after a given amount of time.
5. **Response**: if the request is a query, the response is given after execution, if it is an update the reply is given instantaneously when it is received.

Timestamps in Gossip Architecture



In order to synchronize lazily the updates, each request must have its own timestamp, assigned by the front-end node issuing the request. The clock of front-end nodes must be synchronized.

Each front-end node keeps also an array structure which contains the timestamp of the latest version accessed for each object from a specific client.
The timestamp of the last read or update known by the frontend (FE), referred to as `prev`, is sent to the replica manager along with the query. In response, the replica manager supplies a new timestamp, referred to as `new`.
This `prev` timestamp represents the most recent knowledge of the FE, while `valueTS` is the timestamp of the latest update available at the (single) replica manager receiving the request.

The `new` timestamp is used to update the internal array of the front-end. This structure, that keeps all the timestamp associated with a specific record from the point of view of a specific client, can be a local data structure (local to the

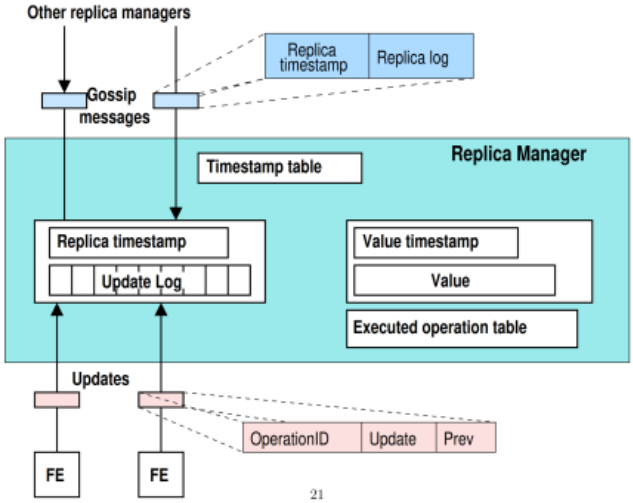
front-end) if every client receives service always from the same front-end, otherwise if the requires from the clients may be received by different front-end (which is usually the case) we have a shared structure across all the front-ends.

Processing Queries

Every time a **query** is processed, a version at least as recent as the last seen from the client must be reported. If the field `prev` of the query is more recent than the `valueTS` of the replica manager instance (`valueTS` < `prev`), the request is deferred and kept pending operation in order to wait for a missing update.
Once the query can be executed, the replica manager returns the `valueTS` of the last value update to the frontend.

Processing Updates

First the replica manager must check if the **update** has already been processed by checking the UID, if it was not a duplicate, the replica manager process the request.
An update can always be processed immediately because the update only changes the local values. So the replica manager creates a new unique timestamp for this new values, update the local data structure and then it sends back the response with the updated timestamp (`new`) to the front-end.
It also adds a new entry to the local log record of pending update operations. In this way, the node keeps track of the changes that need to be communicated to other nodes in the system.



The front-end will receive the acknowledgement, which includes the new timestamp. This information will be stored along with the corresponding client ID.

Therefore, after an update is applied to the local data structure, it is subsequently propagated to all other data managers in a 'lazy' manner over time. Even in this phase, timestamp verification is crucial because there's a possibility that a new update from a client is received, but the local data structure might have already been updated based on inputs from another client, with such updates possibly pending confirmation from other replica managers.

Consequently, the update operation is committed if `prev <= valueTS`, ensuring that updates are applied in a sequence that respects the chronological order of events. Otherwise, if the local timestamp is older (indicating pending updates that might replace the current one), the operation is deferred, awaiting more recent updates via gossip.
When a data is effectively updated, then the replica manager adds a new entry to the local log of pending updates operations.

The condition `prev <= valueTS` is used to verify that the received update is not based on a more recent state than the one currently known to the replica manager.

- If `prev <= valueTS`, the update can be considered valid because it is not based on a more recent state than the one already known. This avoids the risk of overwriting updated data with data based on a previous state.
- If `prev > valueTS`, it would mean that the frontend (FE) is sending an update based on a read or update that the replica manager has not yet received or applied, risking inconsistencies.

When a system accepts an update with prev < valueTS, there is a theoretical risk of losing intermediate updates if the system does not implement adequate mechanisms to handle this situation.
NOI PERÒ ABBIAMO GARANZIA SULL'ORDINE DI ARRIVO DEGLI UPDATE, IN QUANTO USIAMO QUANTOMENO **TOTAL ORDER MULTICAST.** Ma qui usiamo Total/Casual order Multicast oppure View-synchronous Communication?

This mechanism ensures that all updates are applied consistently, maintaining data integrity across the distributed system.

ZooKeeper

ZooKeeper is a high-availability service commonly utilized for data exchange among various services. It can serve as a storage solution for configuration information, a distributed directory, or a naming system.
Operating as a replicated service, ZooKeeper requires a majority of nodes to remain operational for continued functionality. In the event of server crashes, they are automatically excluded but can re-join upon recovery.
The system employs a primary-backup scheme and implements Zab, an atomic broadcast algorithm.
https://zookeeper.apache.org/

Lab4 - C05 Message Queue Systems (RabbitMQ)

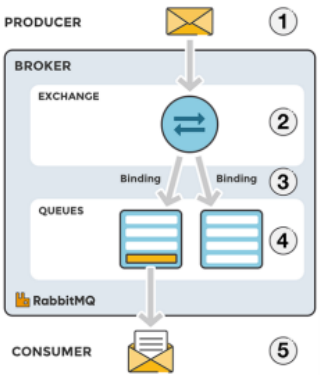
RabbitMQ: opensource message broker that implements AMQP

Many libraries implement functionalities to interface with RabbitMQ:
PIKA: a Python library to interface with RabbitMQ

the producer enqueues messages, the consumer dequeues them
RabbitMQ broker functionalities take care of routing the messages between them

RabbitMQ is composed by a set of queues, one for each consumer. Routing of the messages is made according to a set of rules (binding rules).

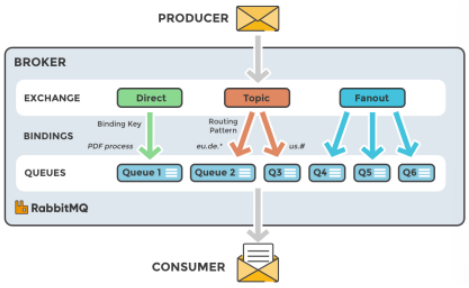
***RabbitMQ messages phases**:*



The producer publishes messages to an exchange (that has a type).
The exchange routes the message according to message attributes, like for example the routing key. (the rules are defined by the exchange type).
The exchange has bindings with different queues, according to rules and bindings, it forwards the message to a given queue
The message stays in the queue until it is consumed (by the consumer)
The consumer handles the message

Routing key: associated with each queue

***Exchange types**:*



***Direct**: message is routed to queues whose routing key exactly matches the message routing key*
***Fanout**: the exchange routes messages to all queues bound to it*
***Topic**: the exchange performs a wildcard match between the routing key and the routing pattern specified in the binding (to each queue)*

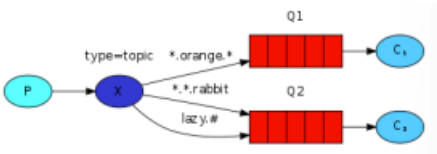
[in the there are the instructions on the command to deploy RabbitMQ on docker and on how to interface as a producer and as a consumer with RabbitMQ with PIKA and how to execute the connection]

***Work Queues** (or Task Queues): to dispatch work-intensive tasks among different workers -> in this scenario there usually is one producer and many consumers on one queue (according to the resources available one consumer or another will process a task in the queue).*

The exchange type Fanout is used for message broadcasting.

[Some examples of implementation in the slides]

***Message Exchange by Topic**: In the example, if we consider a routing key with the following pattern, .., Q1 will receive all the messages regarding oranges animals, while Q2 all messages about rabbits or lazy animals*
[in the slides there is an example of implementation]



C06: Geographically distributed applications: large-scale load balancing and eventually consistent data replication strategies

In a cloud application distributed worldwide (i.e. Google Search) the VM cluster that handle the search load (billions of queries per day) a load balancing cluster would not be sufficient, since the bottleneck would become the network to the cluster -> there is the need to spread the cluster worldwide (VMs in many datacenters all over the world).

Load balancing

Load balancing is performed at two levels:

- **Global level**: to dispatch globally the requests across different datacenters (different routing policies)
- **Local level**: to decide which VM serves one request within the datacenter (usual routing policy: scalability)

At the local level, the load balancing policy usually aims at balancing the load for scalability.
At global level the request could be routed either by the nearest datacenter (i.e. in round trip time from the client) in order to minimize the request latency; in case of huge file upload the load could be routed to a datacenter with low load or with the least used path (maximize throughput, not latency).

Global Load Balancing using DNS

DNS can be exploited to introduce a first layer of load balancing; one simple solution is to set many DNS records for the service, so that the client will pick one of them arbitrarily.
Limitation: the client behavior is unknown: records are selected by clients in a random manner, as they do not know which address is the closest.

Another solution is to exploit **localized DNS systems**: different IP addresses are returned for the same domain depending on the geographical location of the client.
CONS of DNS load balancing: DNS records take hours to be updated (coarse-grained control).

A fine-grained control can be obtained by employing **Virtual IPs**.

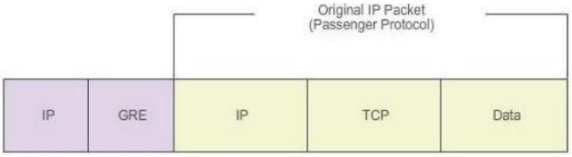
Global routing with Virtual IP Addresses (VIPs)

A virtual IP address is an IP address that is not assigned to one host, but is shared among many devices. From the point of view of the user it remains a regular IP address. One service could have assigned a set of VIPs, with each VIP corresponding to a specific geographical region.

When a DNS request is made, a **localized DNS** service responds with the VIP address associated with the corresponding region. Subsequently, the request directed to the VIP address is intercepted by the **global load balancer** of that region (typically installed in the ISP network or specific exchange points).
The intercepted request is encapsulated inside another IP packet. The new destination IP address is determined by the **global load balancer** based on proximity or lower load, according to global-level load balancing policies.

Upon reaching the destination datacenter, the encapsulated packet is received, decapsulated, and processed in the usual manner as if it were received directly by the datacenter. These operations are usually performed by the datacenter **local load-balancer**, which ensures equitable distribution of the workload among various servers or resources within the datacenter.

To establish a **tunnel** between the global load balancer and the datacenter, the **Generic Routing Encapsulation** (**GRE**) protocol is commonly adopted. GRE enables the encapsulation of an IP packet inside another IP packet. The outer IP packet contains the destination IP address of the (local) load balancer at the destination datacenter.



Geographically distributed systems

We got a problem: **how to synchronize data across different instances that are running on different datacenters?**
Data replication and synchronization is a challenge across distant sites interconnected via wide area network links due to:

- **Limited bandwidth**, which increases significantly the time required for data synchronization operation
- **Lower communication reliability**, which can result in frequent network partitions

For these two reasons, traditional replication strategies cannot be used: those data replication strategies have as assumption that network partition cannot occurs, so new data replication strategies must be adopted.

In a geographically distributed system, you must tolerate network partition: a certain region (or a datacenter) could disconnect from the others for a certain period of time, during which they must continue to offer the service.

ACID systems

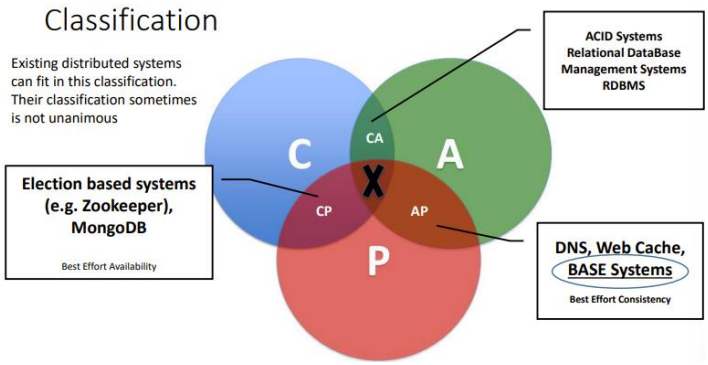
In traditional systems (i.e. relational database) whose first focus is **consistency**, data must be consistent all the time, availability is a secondary focus. So data is stored according to the ACID semantic: **A**tomicity, **C**onsistency, **I**solation, **D**urability.
In ACID systems **availability** (to tolerate failing nodes) is ensured through replication in master-slave fashion.
In those systems partitions are considered rare events, and when those occurs, the partitioned replicas are considered failed and must stop providing the service.

In a global scale in which partitions can be frequent and each node serves thousands of clients, partitioned replicas must go on and provide the service. For this reason, a different approach is needed to handle network partitions!

CAP theorem

The CAP theorem holds that a distributed system cannot simultaneously have all three of the following properties but at most two.

- **Consistency**: all nodes see the same data at the same time
- **Availability**: node failures do not prevent survivors from continuing to operate
- **Partition Tolerance**: if the system continues to operate, in all his replicas, despite network partitions



Note that network partitions are different from node failure. A network partitions is a failure of the network, instead node failure is the failure of node which impacts availability.
CAP theorems states that perfect availability and consistency are not feasible in presence of partitions.

Of course Consistency and Availability is not a "binary" decision: **AP** systems relax consistency in favor of availability (when partitions occur) – but are not always inconsistent, **CP** systems sacrifice availability for consistency (when partitions occur) - but are not unavailable.

BASE semantic: **Basically Available, Soft State and Eventual Consistency**

In BASE systems, consistency will be reached eventually (after a certain period of time), as opposed to Strong Consistency. Depending on the type of service, such an approach is acceptable (e.g., for a social network, not a banking system); however, it is guaranteed that consistency will be reached eventually after a transition period.

The main difference between ACID and BASE systems lies is in how updates are handled:

- **Updates handling**: In ACID systems updates are propagated immediately in synchronous manner; the client has to wait for the propagation to end, while in BASE systems the control is given back to the client once that the data are updated locally (with respect to the node that handled the request). The propagation in BASE systems is propagated in a deferred manner using a specific protocol, named ***anti-entropy protocol***.
- **Queries handling**: both in ACID and BASE systems queries are handled basing only on the local status.

Inconsistency

Inconsistencies may arise due to the deferred propagation of updates in BASE systems. This could result in the presence of two different versions of the same object (due to deferred propagation, like in gossip architecture), and also that two conflicting updates are accepted. Hence, a **conflict resolution mechanism** is necessary to ensure that only one version is kept and propagated (e.g. the most recent update in terms of timestamp).

It's important to note that in the gossip approach, conflicts were not possible thanks to the continuous exchange of timestamps, which is not included is BASE systems.

Advantages/Disadvantages of the BASE Approach

PRO:

- This mechanism allows to handle network partitions. If a network partition occurs, the datacenters can continue to operate. Once the partition is resolved, the **anti-entropy protocol** ensures the eventual distribution of data, leading to a consistency point.
- Response delay is minimal because client does not have to wait for the system to synchronize, avoiding a significant overhead, particularly in global-scale systems.

CON: this approach is not suitable for systems that do not tolerate conflicts

Facebook uses eventual consistency model: reduces the load and improves availability; Facebook has more than 1 billion users → huge amount of data. Eventual consistent model offers the option to reduce the load and improve availability.

Bayou architecture

Bayou architecture is a generic architecture that implements the **eventually consistent** methodology. It provides data replication for high availability with weak guarantees on **sequential consistency**: it means that at any point in time, a replication managers may have a status not consistent with the others or there can be two different replica manager with status in conflict with the another, and this is because they exchange information in an asynchronous manner.

Let's clarify the concept further…

In the Bayou architecture, the focus is on eventual consistency, meaning that systems designed with this model allow for temporary discrepancies or conflicts among distributed data replicas.
In contrast to **sequential consistency**, where the order and outcome of all operations must be immediately visible and consistent across all nodes in the system, **Bayou** accepts that, at any given moment, different replica managers may exhibit states that are not consistent with each other.
These divergences are then resolved through reconciliation mechanisms, ensuring that ultimately all replicas converge to a consistent state.

Each replica manager exchanges updates on the data using an **anti-entropy protocol** that provides updates in group: it ensures that through the continuous exchange of messages the data on all the replica managers is the same (at least in the long period).

- When an **update is received**, it is initially applied but marked as "**tentative"**: while an update is a tentative, the system may undo and reapply it as necessary to reach a consistent state (for example if there was a conflict). For every variable, there is an history in order to roll back the update.
- Instead, once an **update is committed** it remains applied.

The status of the system is given by the committed updates (ordered by their commit timestamp) followed by the tentative updates (ordered by their local reception timestamp). So is the sum of committed and tentative updates.

Primary replica manager

Bayou adopts a **primary commit scheme**, i.e. one replica manager is selected as primary and takes responsibility for committing the updates, responsible to decide when and which update will be committed (data are managed independently), then the decision is spread through other replicas, using the anti-entropy protocol.
The commit protocol can be any policy that establishes the order in which updates are applied and it can be defined specifically to suit the requirements of the application.

Even when a network partition arises, the disconnected nodes continue to work. The temporary unavailability of the primary (e.g. due to a disconnection of the replica manager) does not prevent requests to be handled by the other replica managers: updates are accepted as tentative in the order they are received and queries are answered using the current state of the system. For that period of time, the updates are not committed, so the tentative queue will be longer.

Conflicts

When an update (tentative or committed) is applied a conflict might arise, two mechanism to detect and resolve conflicts that can be defined by application developers:

- **dependency check**: a query (and the expected result) provided by the developer, to check if a set of pre-conditions are met for the execution of the update. Before applying the update, the replica manager executes the query. If the results differ with the ones provided by the developer a **conflict** is arisen.
- **merge procedure**: those procedures are run when a conflict is detected, tells the replica manager how to resolve the conflict (merge procedures are provided by the application developers)

Both those procedures are deterministic, so that every replica manager resolves the same conflict in the same manner as the others. They should not require communication between replica managers!

Performance Measurement of Eventual Consistent Systems

Eventually consistent systems are widely adopted today, even they do not provide safety but have multiple advantages in terms of latency and availability.

The most used metric for eventual consistency systems is **time**:

- Time window of inconsistency
- Time required to have an information visible on the system

Those metrics depends on the anti-entropy protocol and anti-entropy rate (how often it sends updates). Given a certain anti-entropy rate the expected time to consistency can be calculated in order to obtain a certain Probabilistically Bounded Staleness (PBS)**.**

Cloud Platforms

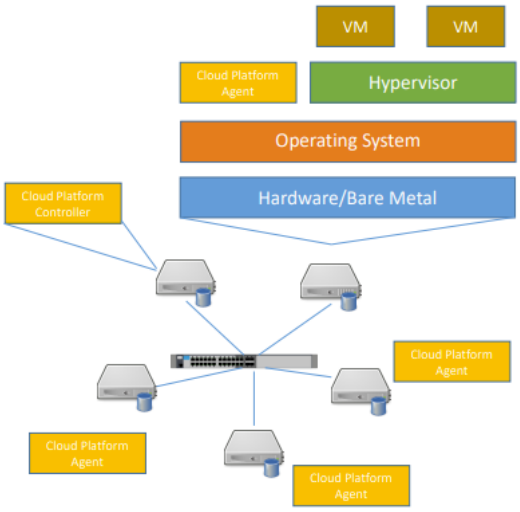
D01-D02 Cloud computing platforms

RECAP: Cloud computing infrastructure: we exploit it to create and control a large number of VMs. A powerful server can support the creation of multiple VMs running at the same time on the same hardware, exploiting unmodified hardware. Groups of servers connected via a high speed LAN comprises the **cloud infrastructure**.
Eventually we have a router that connects the LAN with external users. This group of servers and the LAN are deployed on an appropriate environment, the **datacenter**. We have one hypervisor installed on each physical server.

A pool of servers, each running a hypervisor, is federated to create a **cloud platform**: a distributed system that unifies all servers with their respective hypervisors to function as a single system. This cloud platform (a specialized type of cloud application) is installed on every server and controls the hypervisors of all the servers in the infrastructure.
It manages the instantiation/destruction of VMs on the infrastructure, i.e. at any moment it controls which WMs has to be executed on which server and with which properties.

In order to allow the management of VMs, the cloud platform exposes a **management interface** (web service or REST API). That interface could be exploited either by humans or by software to automate some behavior.
The cloud platform allows the implementation of **IaaS** cloud model as it enables the creation/destruction of VMs. On top of that more complex models can be deployed (PaaS, SaaS).

General Architecture:



- A set of servers are connected to a high speed Local Area Network (Ethernet LAN > 10Gbps)
- Each server has an operating system installed on top of the bare metal (the physical hardware)
- A hypervisor is installed on the host OS to virtualize the physical resources offered by the servers enabling the creation of VMs
- In addition to the hypervisor a cloud platform software is installed to control the local resources virtualized by the hypervisor. This software is often referred as the **agent** of the cloud platform
- In order to implement system wide management and control functionalities of the cloud platform on one (or more to ensure resiliency) server, is installed a particular instance of the cloud platform software. This software is often referred as **cloud platform controller** and can be installed on a dedicated physical server that does not have a hypervisor or it could be even installed on a virtual machine.

The cloud platform controller is a software responsible for controlling the overall platform communicating with the agents. Since it can be a single point of failure, generally there's more than one in order to manage load balancing.

Cloud Platform Implementations

Different public or private cloud platform implementation are available today. There is not a standard for cloud architecture, however, their architecture is very similar.

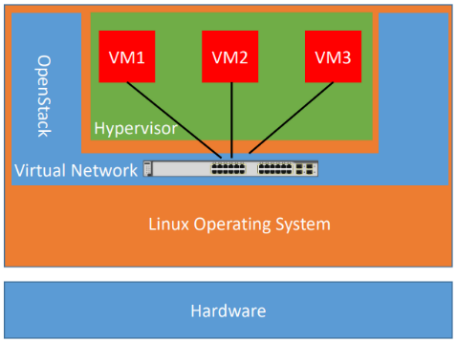
Commercial private platform have hidden implementations, but are all similar to the open-source alternatives.
In this course we will take a look at the architecture and functionalities of one cloud platform: **OpenStack**.

OpenStack

OpenStack is a free open-source software platform for IaaS cloud computing and it is the main solution adopted for private cloud computing. It's supported by the OpenStack Foundation, composed of more than 500 companies, and adopted at CERN in Geneva and Budapest.

It's not a virtualization solution, since it doesn't embed inside any hypervisor: it's realized on external hypervisor in order to implement and support virtualization and manage creation/destroy of VMs.

- The software platform is installed on each server and runs on top of the host operating system, but it can run only on the Linux OS, while it supports multiple hypervisor technologies
- Through each hypervisor, OpenStack creates/destroys the VMs
- Every server on which installed OpenStack is called **OpenStack node**
- The collection of software installed on each server, enables the creation of Cloud platforms and it eventually communicate with the hypervisor in order to implement the commands coming from users.
- The OpenStack installation interact with the local hypervisor

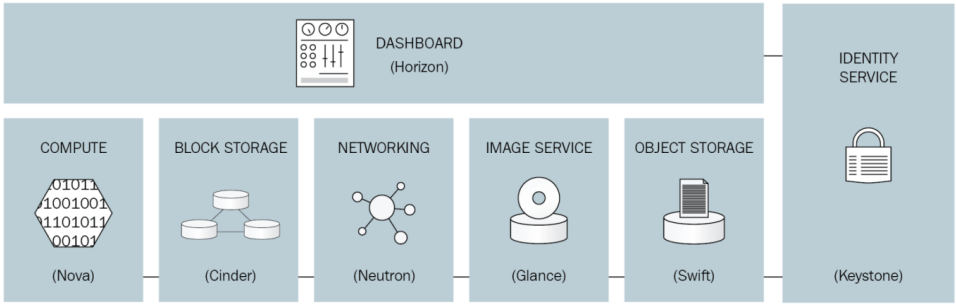


The platform manages the connection of VMs to Virtual Networks, which can be used to communicate with other VMs or with external networks.
In addition to Virtual Machines and Virtual Networks, the platform manages the Storage: this enable the creation of virtual hard drives that can be connected to virtual machines.

An openstack **instance** is a group of server on which the openstack software is installed. Those servers federate together in order to combine their resources to create VMs.
In a instance at least one node is set as **controller** and is in charge of coordinating OpenStack functions and managing the resources available to the instance, while the others are **compute** nodes that offer computation and storage resources for VMs.
The real LAN infrastructure among physical nodes is exploited as infrastructure for the Virtual Networks and for the communication between OpenStack components for coordination and management.

OpenStack Architecture

The infrastructure will expose an interface to allow users and administrators to manage virtual resources. It can be: a web dashboard, REST APIs, a command line tool.

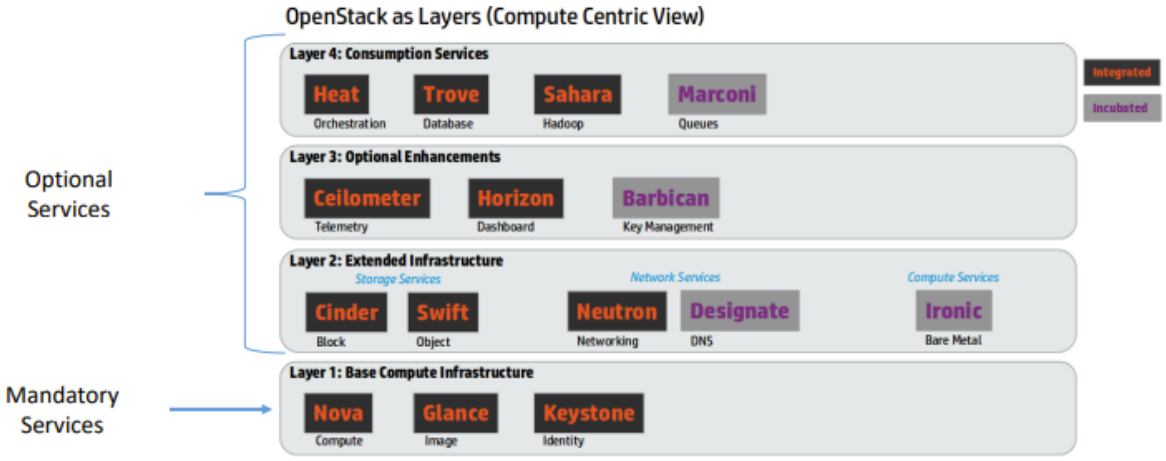


We already mentioned OpenStack is a collection of software, that are installed together in order to create a **cloud platform**. The platform itself and the software are **highly modular**. The Openstack platform is composed of a different set of modules, each one implementing a specific functionalities. These functionalities are organized into services, each one of these **services** is implemented as a separated **module**. Each module could work separately from the platform and is developed and maintained as a separate project.

Each service exposes a set of REST APIs, through this interface it can receive requests from platforms users and administrator via the web/command line interface and from other service modules.
Coordination operations and status information among services, instead, are exchanged using a message queue system.

OpenStack Services

Apart from Core Services which are **mandatory** on each installation, other services are **optional**. The set of optional services is divided in four different levels according to how crucial are the functionalities they provide (layer 2 for most important, layer 4 for rarely included into installation).



Some services must be installed on the controller node, while others on the computing nodes. Some other services need to be installed on both but with different configurations.

The services of the controller node leverage supporting services, like one Database (e.g. MySQL) for status and configuration storage, and one Message Broker (e.g. RabbitMQ) for message exchange among modules.

Information Exchange among services

Services interact each other in three different ways:

- **Message broker:** exchange a direct message via the message broker. Is usually employed to exchange data internally for coordination and synchronization, so is used internally to exchange data about the status and the synchronization of the platform across different modules.
- **REST API:** used to invoke programs, can be either exploited to receive common from external users or from other modules part of the installation.
- **Database:** data is exchanged via data entry or data requests to the database in the form of state of the platform

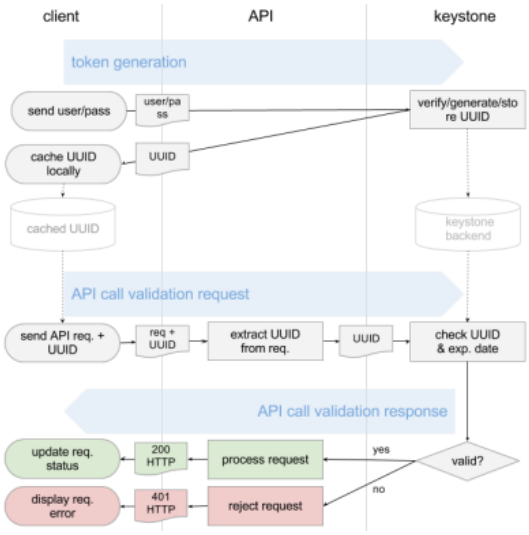
The communication between services can always happen via network, so also the services of the controller module could be spread among the nodes to balance the load.

OpenStack Services

● Keystone (Identity Management Component)

Usually installed on the controller node, used by OpenStack for authentication and authorization.
Ensures security by granting or denying access to **objects** (like VMs or Virtual Networks) to different **Users**. Objects are grouped into **projects**, authorizations can be granted per project that are assigned to an user.

Implements a token-based authorization; the token is given by Keystone after a successful login via username and password. That token is used to access all OpenStack services. Each service has to validate the token.



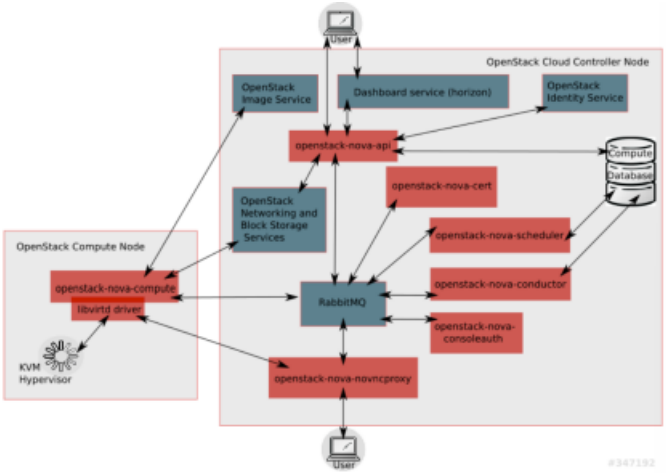
● Nova (VM management)

Nova is the core service of OpenStack, it manages the compute service: installation and management of virtual machines. Nova does not implement a new virtualization technology, but it interfaces with existing hypervisors to manage VMs (compatible with many technologies, like KVM, Xen, Libvirt, QEMU, Docker, VMWare ESX, …).

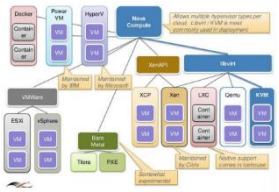
It is made of two components: **Nova Controller** installed on the controller node, and the **Nova Agent** installed on compute nodes. The Nova Controller handles the requests of VMs management coming from the users, evaluates the action to be performed and send them to the compute nodes. Nova Agents on computing nodes are responsible for receiving the actions to be performed from the controller and translating them to commands for the hypervisors.

Nova - Subcomponents

Nova module installed on the **controller node** is composed of the following sub-components.
All of them interacts via messaging queue (**RabbitMQ**).



- **Nova-API**: exposes the external REST interface used by users (and other services) to request actions about VMs
- **Nova-Conductor**: manages the control operations.
- **Nova-Scheduler**: decides where a new VM will be instantiated (on which compute node) according to the available resources.
- **Nova-Network**: implements basic networking services for VMs. If other network service is installed (e.g. Neutron), it interfaces with it.
- **Nova-Consoleauth and Nova-Novncproxy**: implement a console service through which users can connect to VMs.



On the **compute node** the Nova module is composed only of the compute service.

- **Nova-Compute service:** receives the actions from the controller and interacts with the local hypervisor (drivers for different hypervisors are maintained; each of them exposes the same interface to the compute service).

Workflow to create a VM:

1. **Nova-API** receives the request that is forwarded to **Nova-Conductor** (both are running on the controller)
2. **Nova-Conductor** forwards the request to **Nova-Scheduler**, which suggests the proper placement for the new VM (which compute node has available resources). It analysis the current status of the servers and picks a decision about which servers will host this VM
3. If the **Nova-Scheduler** allows the creation of the VM (sufficient resources are available in the system), **Nova-Conductor** forwards the request to the **Nova-Network** service to instantiate the proper network configuration. (Until now, is all on controller node)
4. As the Network is properly configured, the request is forwarded to the **Nova-Compute** service of the node selected by **Nova-Scheduler**
5. The local **Nova-Compute** interacts with the hypervisor to create the VM

Nova Scheduling Strategy

When a scheduling request arrives, the list of available nodes is filtered so that only the nodes with enough resources and proper hardware (i.e. a particular passed-through device) are kept; then that list is sorted according to the particular scheduling **policy** (that is **customizable**).
The policy can include a certain degree of **overcommitment** for RAM and CPU: more vRAM can be allocated than the physical quantity available, the same for vCPU.

overcommitment
The policy can be configured to include certain degree of over-commitment for RAM and CPU, i.e. more VCPU than the CPU available in a host are allocated (the CPU will go slower), more RAM than the one available in the system is allocated to VMs (the host will swap). Quite often overcommitment: for example, a server has 10 GB of RAM, but it's chosen to allocate 15 GBs based on statically assumption that not all 15 will be used all together by VMs. But what happen if not? There's a paging systems, of course the VMs will be slow! It's like overbooking for airplane.

● Glance (image management service)

Each VM is instantiated from an image, which includes a specific operating system pre-installed and pre-configured. (Custom images can be done; e.g. with a webserver preinstalled). Glance manages the collection of **VM templates**.
Glance is made of two sub-components:

- **Glance** (for image management): is responsible for exposing a REST API for managing the images (e.g. uploading an image, delete an image, etc) and store the list of the images available and its features on the DB

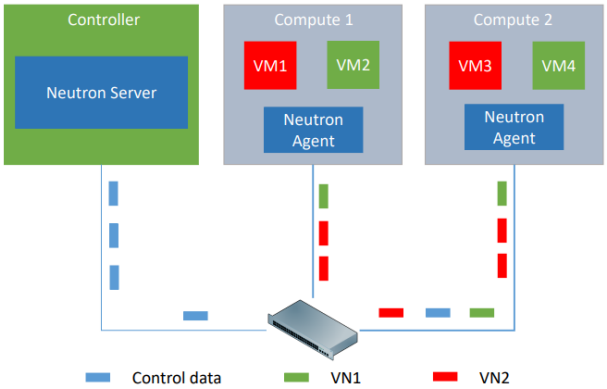
- **Glance Storage** (for storage management): is in charge of storing the images either on the local node on which Glance is installed or on a distributed file system or on a **cloud storage** (storage are covered in the next chapters).

● Neutron (network management component)

It's optional but it's always present because VMs require a **virtual network** to communicate each other.
Every VM have one virtual network adapter, which must be connected to this virtual network. Different Virtual Networks are instantiated for different VMs in order to ensure isolation among them, i.e. VMs created by different tenants cannot communicate.

Neutron is the service responsible for managing the virtual network infrastructure, it allows the creation of Virtual Networks among VMs that can run on different OpenStack compute nodes.
The Local Physical Network that interconnects Computing nodes is exploited as infrastructure to enable the virtual networks across different compute nodes.

Made of two sub-components:

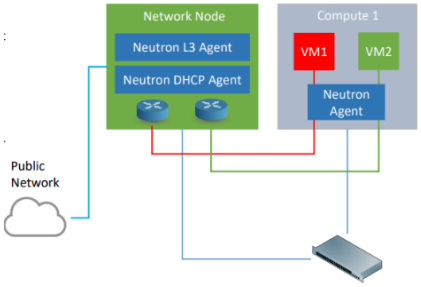


- **Neutron Server**: it's installed on the controller and it's the one responsible for managing request from users and translate those command into command transmitted to the neutron agent. It manage the Virtual Networks by coordinating the neutron-agents.
  It is also responsible for exposing the REST APIs for the management of Virtual Networks to users.
- **Neutron Agent**: is implemented on compute nodes and it's responsible for receiving commands from the server and implementing them locally. So the traffic to/from the local VMs is managed by rules and command provided by the server (and of course the user).
  Specifically, it dispatches the traffic directed_to/generated_from the local VMs in order to emulates different virtual networks that span across different compute nodes.

Virtual Networks are usually private networks. Neutron allows VMs to be connected to external networks, so VMs can connect to the Internet and vice-versa they can be accessible from the Internet.
In order to connect Virtual Networks to the Internet, a **Network Node** must be instantiated.

Neutron Network Node

It is a special node (usually the controller node) that is connected to the external network (the internet). Runs a particular Neutron-agent called **Neutron-L3-Agent**, that is responsible for rerouting the traffic between private virtual networks and public networks. This is accomplished by **Virtual Routers**, that implement traffic routing and NAT functionalities.
On the network node there is also the module named **Neutron-DHCP-Agent** that manages the network configuration assignments to VMs.



→ Public IP addresses can be assigned to VMs, also dynamically (floating IP address). An alternative is to have Virtual Routers at the edge of each Virtual Network will take care of implementing Network Address Translation (NAT).

Physical Network Infrastructure

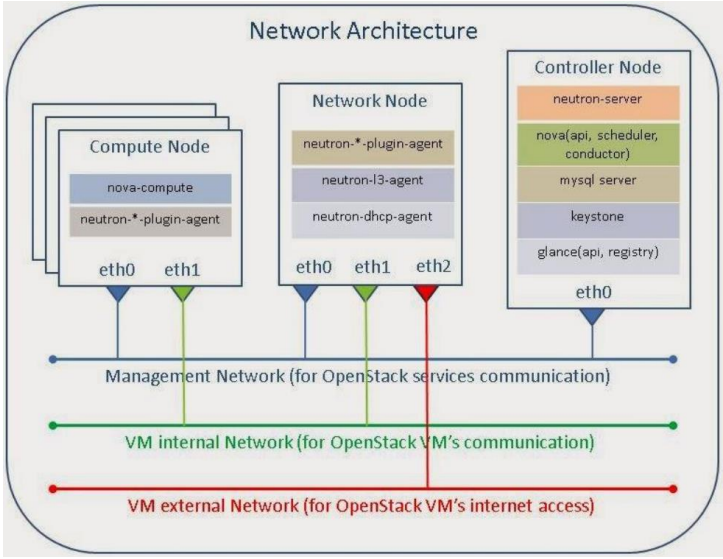
A typical OpenStack network infrastructure connected to an external network is usually configured with three different physical networks:

- **VM internal network**: every compute node should be connected to this network, which allows the dispatch of internal communication among different VMs.
- **Management network**: interconnects compute nodes with the controller nodes, networks used to exchange control communications.

These two networks, usually are implemented on the same physical network!

- **VM external network**, the one connected to the Internet. It's connected only to the network node (in the majority of the case, it's the controller node).

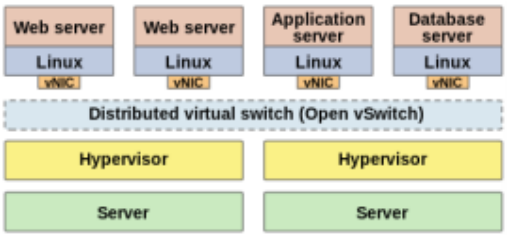
The controller node is connected only to the management network.



Neutron Agent (again)

The neutron agent is the component deployed on every computer node for handling and managing all the traffic originated by the local VMs (or received by) according to the Local Network configuration.
This Neutron Agent is highly configurable, different configurations are possible exploiting different technologies to process and forward traffic.
The latest versions exploits **OpenVSwitch** to process data between the compute node and the network node.
OpenVSwitch allows to rapidly reconfigure the behavior of each agent following the directives coming from the Nova server.

OpenVSwitch (OVS) is an open-source implementation of a distributed network virtual switch, so it mainly implements layer 2 switching functionalities.
Designed to provide a switching stack for hardware virtualization environments. Can run within (or jointly with) the hypervisor.

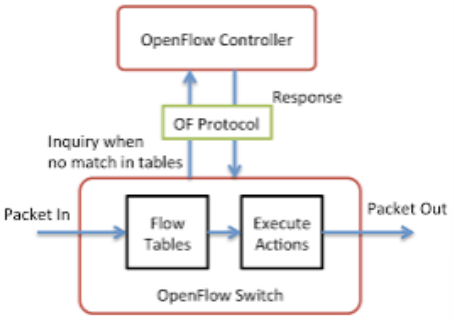


Adopts a **centralized approach:** every compute node has its own OVS instance that handle the traffic following the directives of a centralized controller. This approach is called **Software Defined Networking** (SDN).

It does not only implement layer 2 functionalities (traffic forwarding) but also they implement some traffic manipulation functionalities like packet filtering, routing, switching and manipulation.
The work of an OpenVSwitch instance is simple, they receive a directive from the controller (in our case, the Neutron Server) and it will follow the instruction received.

Software Defined Networking (SDN)

OVS adopt the SDN paradigm: allow network reconfigurable at runtime in a very short amount of time: this is useful because VMs can be created frequently and network configuration must be ready without waiting.



Every **OpenFlow Switch** that is installed on every compute node has an internal table called **flow table** that contains a set of rules received by the controller and that describes the features of any incoming packet (e.g. the source IP, the destination MAC address, etc.) and the corresponding action to be executed (e.g. forward the packet, modify the packet, etc…).

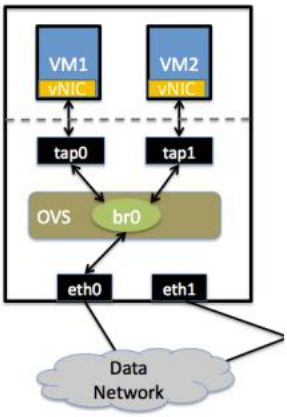
Whenever a packet is received, the table is accessed:

- if the packet has a match with one of the entry in the table the corresponding action is executed
- if no match is found, OVS sends a query message to the controller (the Neutron Server), which replies with a new rule to be added in the local table or a specific action to be performed once. The protocol that implements the communication between OVS instances and the controller is named **OpenFlow**.

With SDN, switches or routers are very stupid, they commonly implement a basic set of functions, and those functions are applied based on the rules that are received from a centralized coordinator. This approach was introduced to reprogram rapidly the configuration of the network, we do not need an admin that configures all the routers.

With SDN instead of having a distributed logic for implementing the packet forwarding and packet routing, you have a centralized entity that imposes the decision of the admin at runtime to all the switches and all the routers. So the admin has a centralized point where he can implement his policy that will be forwarded to all the switches. If the policy has to be changed, the only thing that needs to be done is to change the rules on the controller, and that change will be spread out.

How does the OVS component fit in our Neutron implementation?



Locally each compute node has an instance of OVS module which receive traffic from the VM. Every VM, from the network point of view, is characterized by a pair of virtual NIC:

- One deployed inside the VM (vNIC, whose hardware is emulated by the hypervisor)
- One deployed in the host operating system (called **tap**)

tap

A tap is a virtual network interface that can be linked with a software, in our case it is linked with the hypervisor.
This connection allows the tap to send and receive network traffic (layer 2 data frames) to and from the software. In simple terms, the tap acts as a bridge between the vNIC inside a VM and the rest of the network system that the hypervisor controls. So, when data needs to go in or out of a VM, it passes through this tap interface.
Both the data coming from these tap interfaces and the data from the actual physical network cables in the node are handled by OVS. OVS helps to organize and direct this data where it needs to go, making sure that the virtual machines talk to the right parts of the network at the right time.

When traffic flows from VM to other network, it flows from the vNIC deployed inside the VM to the one deployed on the host system (the tap): this second NIC is plugged into a OVS, which receive traffic, manage the traffic according to local rules (provided by the controller: the Neutron Server) and then forward it.
The opposite flow of traffic is quite similar.

OVS on the Network Node

When the data is directed towards an external network, it is routed to the **network node**, which necessitates the instantiation of a **virtual router**. This is implemented by installing an OVS instance on the network node, responsible for routing traffic to and from VMs and the external network.
The **Neutron L3 Agent** on the network node manages routing by configuring rules and policies for **OVS**, which then handles the actual traffic routing. This collaboration ensures efficient traffic flow between VMs and external networks in the OpenStack environment.

Network Virtualization

Is the process through which Virtual Networks for VMs are created on top of the same physical infrastructure. Virtual networks are **layer 2** networks, specifically ethernet networks.

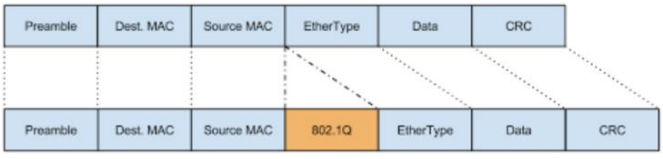
The implementation of VNs requires:

- Layer 2 data forwarding between VMs belonging to the same VN
- Isolation between VMs that are in different VNs except when there is a router connecting them at layer 3

In order to dispatch correctly each packet, it must be marked with the ID of the VN to which it belongs. Consequently, if two compute nodes host two VMs belonging to the same VN the network service should be configured to forward traffic between them without passing through the Network node. This is achieved by leveraging the various OVS instances present in the different nodes.

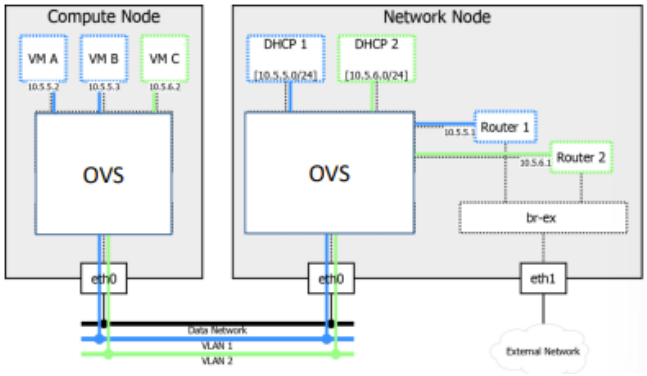
Different network technologies are adopted: **packet tagging** and **packet tunnelling**. There are three technologies that are exploited by OpenStack for this purpose:

1) Virtual LAN: one of the most widely adopted network standard for network virtualization because of its simplicity. It's defined to allow creation of virtual LANs on top of regular physical ethernet network. This standard adds a field to the



ethernet header, the **VID** (VLAN ID), that specifies the ID of the Virtual LAN to which it belongs.
A switch supporting VLAN standard, is in charge of handling the VID field, adding it for injected packets and removing it when the packet reaches the destination.
For example, broadcast frames should be delivered only to hosts belonging to the same VID to guarantee isolation.

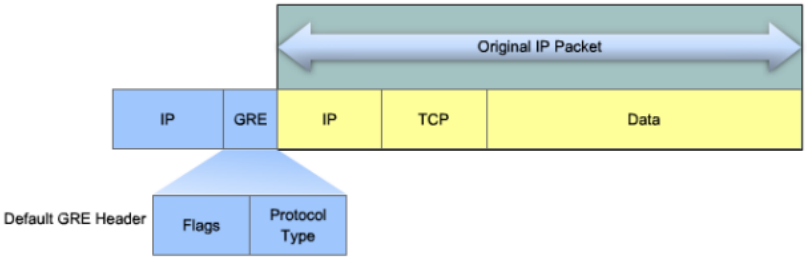
Neutron can exploit VLAN to implement VNs, in this case when a VN is created to connect two or more VMs, a VID is assigned to the VN by the Neutron Server. The server configures the **OVS** of the compute nodes so that the packets with that VID will be sent to the VMs of that VN. In case the VN is connected to the Internet via a Virtual Router, the Neutron-Server configures also the OVS running on the Network Node to tag the packets coming from/directed to the Router assigned to the VN.



2) Generic Routing Encapsulation (GRE) Tunnelling

The protocol is used to create a tunnel between two internet hosts. It achieves this by encapsulating network IP packets inside another IP packet by adding an additional GRE header to specify the type of the encapsulated protocol.

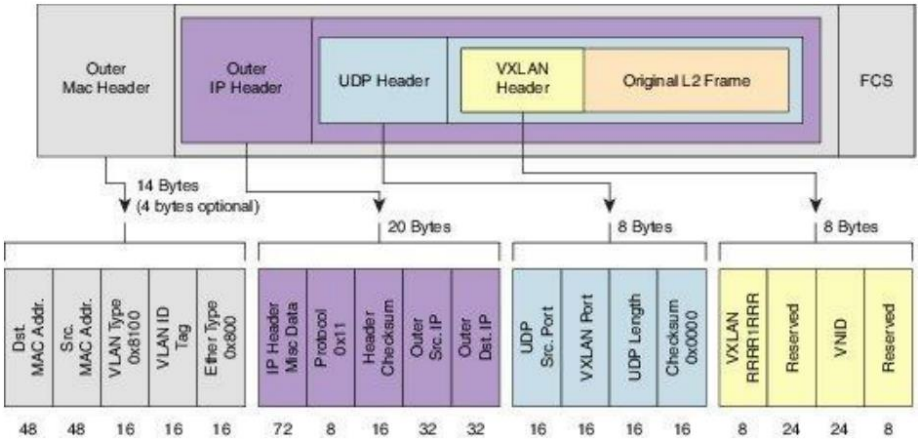
It implements VNs by creating a GRE tunnel for each pair of compute nodes (and between network node and all the compute nodes) that run VMs belonging to a specific virtual network.
Different GRE tunnel ID are used for each Virtual Network.



The Neutron server is in charge of instantiating the GRE ID for each VN. Subsequently, each compute node containing at least one VM of a particular VN is instructed to create a GRE tunnel with every other compute node in the VN, directing the **OVS instances** to create a virtual network adapter on top of the physical interface.
The GRE interface is responsible for creating the tunnel across the involved computing nodes (nodes that have at least one VM that belongs to the VN, or computing nodes and the Network Server) and encapsulate/decapsulate the traffic.

3) VXLAN (virtual extensible LAN)

VLAN has a limit, VLAN identifiers are 12 bits long, consequently the maximum number of VLANs is set to 4094: in large cloud implementations this number, even if high, can be an issue. VXLAN allows up to 16 millions of different VNs.
VXLAN is still a packet encapsulation methodology between two endpoints, called **Virtual Tunnel Endpoints** (**VTEs**).
Frames are encapsulated in **UDP** packets, while at IP level the source and destination of the VTEs are specified. A VXLAN header is included to report the ID of the Virtual LAN as two VTEs can support multiple VLANs at the same time.



Neutron exploits VXLAN tunnels in the same way in which GRE tunnels are exploited. The Nova server assigns a VXLAN ID for each VN. The Neutron server instructs the **OVS instances** to encapsulate packets into VXLANs tunnels according to VN instantiations.

The main difference among these 3 ways for creating VNs is that new fields are added at different layers of the Internet protocol stack: VLAN at layer 2, GRE at layer 3, VXLAN at layer 4.

● Cinder (component for volumes management)

Cinder is the component responsible for managing volumes. A **volume** in OpenStack nomenclature is a virtual hard drive. It could be the primary hard drive where the OS of the VM is located or it could be a secondary hard drive.

Additional volumes can be dynamically created and attached to an instance. Virtual hard drives are stored in the file system as an **image** (a file or an object). Cinder manages those images and exposes them to VMs. The virtual hard drive is exposed by the hypervisor to the VM, the hypervisor accesses the actual virtual hard drive via the iSCSI protocol.

Cinder stores the volume images in the local file system of the node on which the service is installed or in a cloud file system.

Cinder comprises the following components:

- **Cinder-API**: exposes RESTful interfaces o clients for control operations (Nova or final user)
- **Cinder-Volume**: handles directly the requests coming from the REST interface
- **Cinder-Scheduler**: selects the storage to store new drives (if multiple storages are configured on the system)
- **Cinder-Backup**: responsible for creating backup of existing volumes when requested by the users

● Ceilometer (telemetry component)

Monitors all the components of the OpenStack instance, measures the resources being used by each User.
That data can be used for billing purposes or to check the status of the system.

Architecture:
There is a centralized **Ceilometer-Collector** that is responsible for receiving the data from all the OpenStack components and store them into a DB (usually a NoSQL DB like Mongo DB).
In order to collect data from all the compute node, a **Ceilometer-Agent** is installed on each node. The agent is responsible for collecting all the notifications from all the local components and forward them to the collector.

● Horizon (web interface): dashboard usually exposed by the Controller node, allows management of all the instances aspects (creation/destruction). Includes also a set of command line tools for backend management.

● API of OpenStack services: Every OpenStack service exposes a set of **REST** APIs for inter-service interaction and to offer a range of functionalities to users. These APIs can also be utilized by users to embed automation processes into external applications.

● Swift (OpenStack object storage service)

It allows users and OpenStack modules to store data (represented as **objects**) in the cloud platform.
It's quite popular because it's offer a reliable and large-scale storage system. The system itself take care of everything, the system only require or send an object.
Data is usually eventually stored in a **cloud storage** (which we will discuss more thoroughly in the next chapter), to guarantee scalability, although local storage on the server where the service is installed is also allowed.

Other services, like for instance **Cinder** or **Glance**, can use Swift to store Volumes backups or images, respectively.

object
An object is an immutable piece of data which can be stored on platform, cannot be modified and can be retrieved from the user.

Swift is composed of two components:

- **Swift Proxy**: exposes REST interface and handles requests
- **Swift Storage Node**: is installed on each **Storage Node** (the nodes that hosts data) to actually store data on physical drive

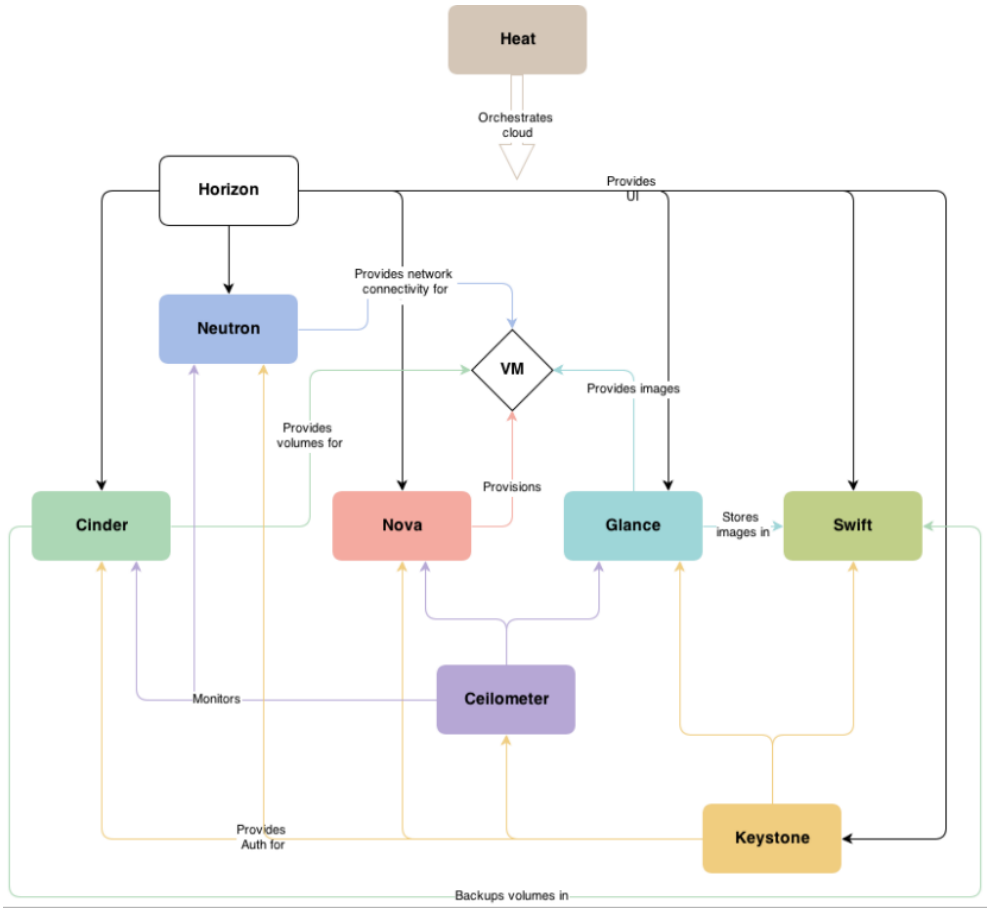
● Heat (OpenStack orchestrator)

It manages VMs creation and destruction automatically. The automation of these processes is based on a set of rules that specify when to trigger actions on the VMs.
Example: through Heat an autoscaling service can be created that exploits the stats collected from Ceilometer to create or destroy VMs according to the status of the system.

Composed of two modules:

- **Heat API**: exposes the REST interface for the users to configure the orchestrator
- **Heat Engine:** handles users' requests and implements them

OpenStack services interactions summary



In order to ensure **high availability** the management modules that are installed on the controllers node should be replicated, e.g. if you have only one instance of the Nova conductor (the coordinator for creation of VMs) and that node fails, the whole system fails. For this reason OpenStack has different high availability configuration where the modules of the controller can be replicated in the same manner the VMs implementing a tier of cloud application are replicated. For the implementation of those modules the same techniques that we saw for cloud application are replicated here.

Lab5 - D03 OpenStack installation using JuJu

[refer to slide, are quite all manual steps to deploy the components of OpenStack on a cluster of machines]

***Juju**: open-source application tool developed by Canonical to facilitate quick deployment and configuration of public and private cloud services along bare metal servers or VMs.*

The successive lesson was about OpenStack basic operations, but there are no slides about that…

D05 Lightweight cloud computing platforms for DevOps

A lightweight cloud platform is a cloud platform that exploit lightweight virtualization for providing a virtualized environment to run software like container or something similar.

Software development in the past: most of the applications were large monoliths running small number of processes (nowadays called **Legacy Systems**).
→ Slow release cycles and infrequently updated due to complex and unpractical development. Since applications were monoliths changes to a part of that required a redeployment of the whole application. With increasing complexity there was the need to scale up vertically the servers (vertical scaling by the way is not always feasible).
Developers (**dev**, which were in charge of designing and programming the application) usually package up the whole system and it is passed to the **ops** (operations) team, which is the team of system administrators in charge of managing the infrastructures (the servers) and deploying and maintaining the application.

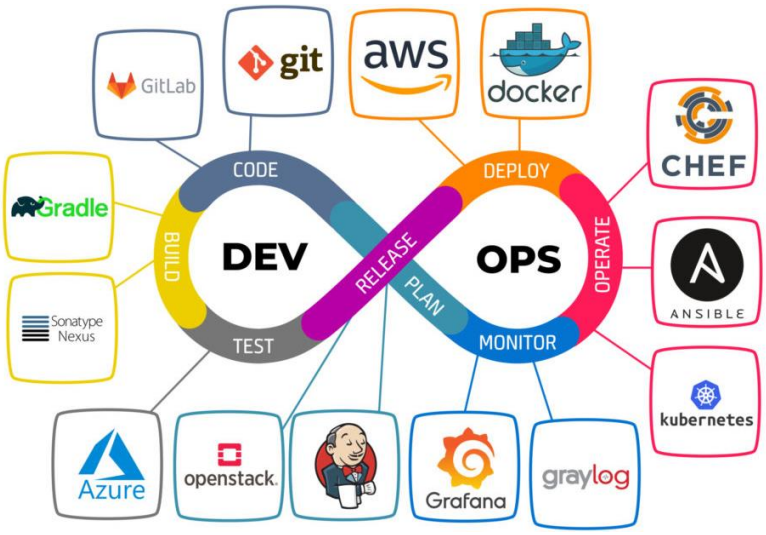
Those systems are still up and running today. If convenient they are cloudified.
When the application has to deal with an increasing load, vertical scalability is adopted, as it does not require modification to the original architecture of the application. Vertical scaling, however, has limits and it is not always feasible.

Software development now: Cloud computing radically changed the way software applications are developed and deployed. Big monolithic legacy applications are now broken down into independent components, called **services** or **microservices**. Services are decoupled each other and can be developed, deployed, updated and scaled individually. Each service (or microservice) run as an independent process and communicates with the others via well-defined interfaces.

Service independency: enables rapid application lifecycle, services communicate through synchronous protocols (SOAP, REST) or asynchronously through message exchange (message queues). If the interface remains the same, a change in one service does not affects the others. This avoid completely the cascade effect which was the main reason for the slow lifecycle of the traditional approach.

Application development process

In the past there was a team for application development (**dev team**) and a second team called "operations team" for application deployment and maintenance (**ops team**), nowadays there is just one team both for development, deployment and maintenance for the whole application lifecycle, called **DevOps**.



DevOps approach: the main advantage of this approach is to have developers more involved in running the application in production, this leads to a better understanding of both the users' needs and issues and also the problems in deploying the application. In this way the developers bring their knowledge into the operational team. Releases can be performed more often: ideally you can have developers deploying their application themselves without having the ops team involved (continuous development).

Such practice, however, requires developers to have a deep knowledge of the details of the underlying infrastructure, which is not always feasible or desirable -> **NoOps**.

NoOps: an approach where developers deploy applications without knowing the infrastructure and without having to deal with the peculiarities of deploying an application.

A solution to avoid to have these over skilled people, is to automate some of the functionalities required for the deployment of a software, so the idea is to automate the software service deployment and maintenance.
So in order to implement a NoOps or an "easy" DevOps approach there is the need for a solution that enable automatic service deployment and maintenance.

In that way you would still need an **operations team** in charge of maintaining the infrastructure, but **developers** could autonomously deploy their software easily.

Another reason why a solution to automate the deployment and configuration of components (or services) is needed is that often there are applications made of a high number of services, that would be tedious and long and error-prone to deploy them manually.

Services Dependencies Problem

Since applications are made of many services, each of them deployed (eventually) by different teams, every service will have its own set of dependencies, sometimes some of the dependencies will be shared between services (crossed dependencies), sometimes the dependency version required for the same library could differ. Any service can change arbitrarily its dependencies whenever its development team wants. This is very difficult to handle by operations team.

If the component is inside a monolithic application: it's complicated or unfeasible keep two or more different version of the same library. One desiderated solution is to have an independent and isolated environment for each single service.

Consistent Application Environment:
This isolated environment can be created through full virtualization, i.e. running the software in a VM with an ideally confined environment where all the libraries are pre-installed, and the real hardware is abstracted.
CONS: This solution is computationally expensive due to the significant overhead associated with full virtualization. Additionally, using VMs introduces overhead in terms of configuring the VM and its guest OS, involving the setup and installation of the necessary libraries.

Another solution that standardizes the environment and that has less overhead is **Lightweight Virtualization** mechanisms, like **containers** that are a cheap solution to have an isolated environment!

Containers can be used to prepare a virtualized environment in which libraries and dependencies are pre-installed and configured to run a certain application or a service, which is part of an application. In addition, containers can be easily shipped and installed on different machines using ad-hoc distribution systems. An example of this solution is **Docker**.

Kubernetes

Kubernetes is a software systems developed by Google that allows to easily deploy and manage containerized applications on top of it. So it's a platform to automate software deployment.
A **containerized application** is an application whose components (independent services) are deployed and shipped as different containers.

Kubernetes require a config file and it deploy application and all its components automatically (automate the deployment process, based on the config file). After deploying and having the application running in a infrastructure you have to maintain it, maintaining an application means: monitor the well being of the application, change configuration, react to failure, load management: automated resource management and load balance.

Kubernetes aims at offering a **lightweight IaaS infrastructure** to deploy applications in a simple and effective way.
So developers can deploy and maintain the application by themselves, but the **operations team** in a Kubernetes environment is still required in order to maintain and run the Kubernetes infrastructure itself.

Kubernetes Core Operations

Kubernetes is a distributed platform, and its overall architecture is very similar to OpenStack. It's made of at least one **master** node and a varying number of **worker** nodes.
Kubernetes nodes can be installed either directly on physical hardware servers or on a cloud platform on top of VMs. The latter is the most frequent solution: highest degree of flexibility, since infrastructure can be shared with other services.

Master node exposes an interface to developers, so that they can request to execute a specific application:

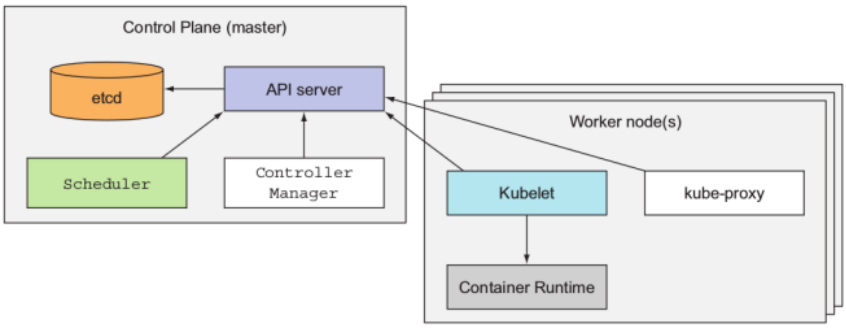
- the developer has to submit the **app descriptor**, which only includes the list of **components** that compose the app
- for each component it is specified the configuration
- the master node take care of deploying the app components on worker nodes in an automatic manner according to the description provided by the developer in a complete automatic manner

Through the interface the developer could specify that some services have to run together, so that they are deployed on the same working node (**pod**, see in next pages). The others will be spread around the cluster according to the status of each working node.
The developer has to specify the desired number of instances of each component in order to ensure scalability. The platform will take care of replicating each component and load-balancing the traffic. The platform is also in charge of monitoring the status of each component and handling failures. In case of instance (or working node) failure, Kubernetes takes care of replacing the instance with a new one.

PROs: **developers** do not have to implement infrastructure related functionalities in their application: the platform takes care of scaling, load-balancing, self-healing. Similarly, the **operations team** is relieved from the responsibilities of deploying, configuring, and fixing failures, as these functionalities are automatically managed by the Kubernetes platform. All this results in higher utilization of available resources in the infrastructure.

Cluster architecture

The **master node** implements the **control plane** which is responsible for controlling the cluster. The control plane is made of many components, each of them can be installed on a single master node, can be split across multiple nodes or they can be replicated in order to ensure high availability.



The **control plane** (master node) has the following components:

- **API server:** exposes an interface to developers
- **Controller Manager**: performs cluster level functions like replicating components and handling node failures, is also responsible for receiving requests from API server and translate them into commands
- **Scheduler**: schedules app, select which worker node will host applications
- **Etcd**: is a reliable distributed data store

The **worker nodes** instead include the following components:

- **Container runtime**: a container runtime environment (i.e. Docker) that takes care of running the containers
- **Kublet**: talks with the API server and manages the containers in the node
- **Kubernetes Service Proxy**: balances the network traffic between application components, forwards the requests when containers are migrated to other working nodes

Application Deployment

The application description provided by the developer lists the containers that composes the application. Containers are grouped together into **pods**. A pod is a group of containers that must run on the same worker nodes, in other words they must be deployed together and shouldn't be isolated (a pod can be one container).
For each pod/single container the developer specifies the configuration and the number of replicas to be deployed.

Why would I want two containers in the same node? For example 2 components communicate very frequently, and it's not convenient deploy them in different servers.

When the developer submits the descriptor to the **master node**, the **control plane** takes care of informing the scheduler to schedule the required number of replicas of pods in the available **worker nodes**. The **Kublets** instance of each of the selected worker nodes takes care of instructing the **container runtime** (usually Docker) to download the assigned containers images from the repository and run them.

Management and Maintenance

Once the application is running, Kubernetes takes care of fixing failures and monitoring the containers. During the application running, the developer can modify the number of instances of the pods; in this case the platform takes care of modifying instances at runtime.

The developer can even let Kubernetes decide autonomously the optimal number of instances of a certain container based on real time metrics like CPU load, memory consumption, query per second, …

Container Migration

The platform automatically takes care of instantiating containers and migrating them when required (e.g. when a working node fails): when a container provides a service to external client, the platform has to take care of forwarding requests as the container is moved across the platform.

Typically, the platform exposes a specific service with a single **public IP** address, regardless of the actual number of containers running it and their location. The platform, with the **kube-proxy** component, manages the connection to the containers and forwards traffic appropriately. Kube-proxy, installed on each worker node in the Kubernetes cluster, stays informed about the distribution of machines (containers) across nodes through continuous communication with the Kubernetes API server.

In addition, the component takes care of implementing load-balancing policies to forward the traffic in a balanced manner in order to ensure scalability.

Exposing Applications

Pods are connected to a local private networks. Each pod has its own **private IP** address that is not accessible from outside. Those private networks allow communication among containers running on same or different workers of the infrastructure.

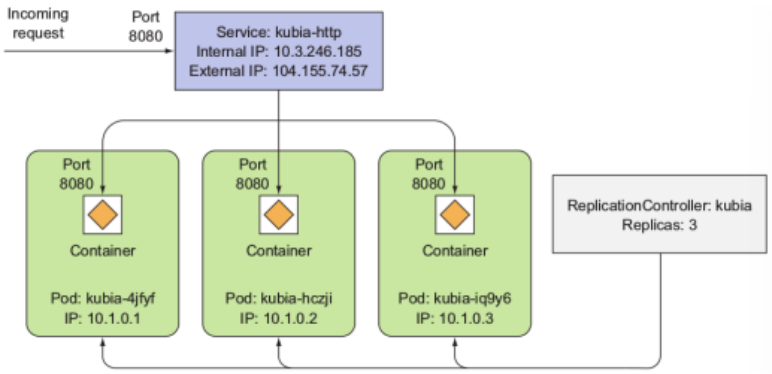
If a service needs to be accessible from external networks (e.g. because it exposes the web or the REST API interface of the application) the developer must define a **service object**. This service object instructs the platform to associate an **external IP** address and port with a pod, allowing the service exposed by the pod to be accessed from external hosts. The service object handles the task of forwarding requests from the external IP address to the local IP address of one of the associated pod instances.

> **Service Objects** in Kubernetes do not reside physically on a single worker node. Instead, they are defined within the Kubernetes API server and exist as part of the cluster configuration distributed and managed through the API server itself. When a Service Object is created, it provides an abstraction that exists at the cluster level, allowing Kubernetes components, such as kube-proxy on each worker node, to interact with this configuration to direct traffic to the appropriate pods.

Load balancing

In case of multiple instances of a pod connected to external network via a service object, the service is also in charge of load balancing the requests across the multiple instances. The platform associates to each pod a **replication controller** that takes care that the proper number of instances is running.

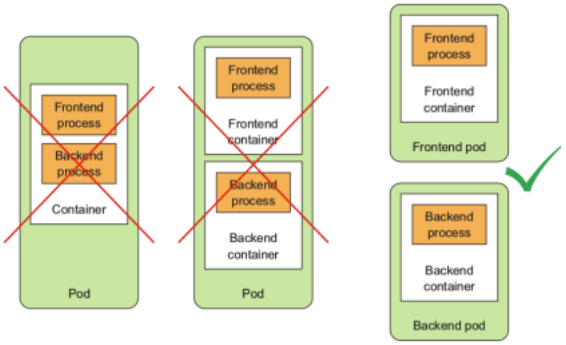
The replication controller handles also failures, and in case of detected failure creates a new pod instance.



Pod design

General rule: containers should always be in separate pods unless a specific reason requires them to be part of the same pod (and so to be deployed on the same node).

For instance an application might have latency requirement in the data exchange between two container, in this case it is convenient to group them in the same pod, so they do not communicate over the network.



Lab6 - D06 Kubernetes installation

[refer to the for the procedure, it is about installation of kubernetes with juju (using a Charmed Budle → a type of Juju installation in which everything is automated (pre-defined components and installation nodes, pre-defined configuration), minikube setup, kubectl, metallb (a load balancer), …]

Lab7 - D08 Cloud Platform Kubernetes Operations

pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
    name: helloworld
    labels:
        app: helloworld
spec:
    containers:
    - name: helloworld
      image: luksa/kubia
      ports:
      - containerPort: 8080
```

Pod metadata

Containers composing the pods

[kubectl commands to create, delete, list pods]

```
kubectl apply -f pod.yaml
kubectl get pods
kubectl delete pod "PODNAME"
```

***Deployment pod**: If you want Kubernetes also to handle some of the functions to manage the application lifecycle you need to define a Deployment.*
→in a deployment, pods can be deployed specifying also additional functionalities (e.g., the number of replicas).

```
kubectl apply -f deployment.yaml
kubectl get deployments
```

→ if you try to delete a pod that is part of a deployment, kubernetes will instatiate another pod of the same type (to respect replicas requirements)

In order to expose a service running in a container you must create a service: `kubectl expose deployment/helloworld`
→`kubectl get services` shows that a new service of type clusterIP was generated.

Since each container has its own endpoint and since the containers of a deployment can vary (and so their endpoints) we exploit ClusterIP.

deployment.*yaml*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: helloworld
spec:
  selector:
    matchLabels:
      run: helloworld
  replicas: 2
  template:
    metadata:
      labels:
        run: helloworld
    spec:
      containers:
      - name: helloworld
        image: luksa/kubia
        ports:
        - containerPort: 8080
```

Number of replicas to be instantiated

***ClusterIP**: a virtual IP is associated with a service running on many pods. Requests directed to that service are sent to the Virtual IP (and so to the service) that dispatches the request to one of the available pods.*
→ ClusterIPs are reachable only from inside the kubernetes network.

*An application (or other services) that want to contact one of the instances of a service can use the ClusterIP or the internal DNS service (**kube-dns:** `nslookup "SERVICE NAME"` returns the list of endpoints for that service).*

***NodePort**:*

A service exposed to external networks is of type NodePort.
*→when you configure a **NodePort for a service**, a port is assigned to a service, and all Kubernetes nodes of the cluster receiving a request on that port are instructed to forward the request to the clusterIP for that service.*

```
kubectl expose deployment/helloworld --type=NodePort
```
→ the external port is selected randomly

***Load Balancer**:*

Another way to expose to the external network a service. A public IP address (taken from a pool of public IP addresses assigned to the Kubernetes cluster) is assigned to the service responsible for dispatching the traffic to the replicas (pods) of that service following a load balancing policy.

[kubernetes autoscaling example in the slides]

`kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10`

The above command instruct Kubernetes to instantiate a deployment that has a target of mantaining the CPU to 50% of load and that can instantiate between 1 and 10 pods for each deployment.

autoscale.*yaml*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 2
  template:
    metadata:
      labels:
        run: helloworld
spec:
  containers:
  - name: php-apache
    image: k8s.gcr.io/hpa-example
    ports:
    - containerPort: 80
    resources:
      limits:
        cpu: "500m"
      requests:
        cpu: "200m"
```

```
kubectl apply -f autoscale.yaml
```

```
<?php
  $x = 0.0001;
  for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
  }
  echo "OK!";
?>
```

`kubectl run --images=run-pod/v1 -it --rm load-generator --image=busybox /bin/sh while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done`

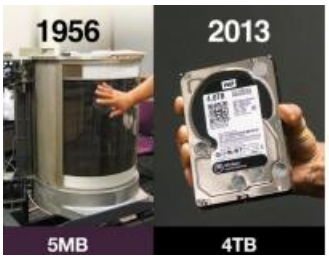
The above command launches a pod in which shell commands are executed.
This pod performs infinite requests to the local php-apache service (I think that expolits kube-dns to translate the container name to an IP). The autoscale instance of php-apache service instantiates a growing number of replicas while time passes in order to respect the CPU usage target.

```
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%    1         10        1          75s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%    1         10        1          76s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   147%/50%   1         10        3          2m9s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   147%/50%   1         10        3          2m12s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   86%/50%   1         10        3          2m45s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   38%/50%   1         10        6          3m15s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   38%/50%   1         10        6          3m20s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   33%/50%   1         10        6          3m42s
root@SWNUMUYKY62JPXH:~# kubectl get hpa
NAME         REFERENCE                     TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache   0%/50%    1         10        6          5m14s
```

Cloud Storage

E01 Cloud Storage and Distributed File System

Storage Technology Evolution:

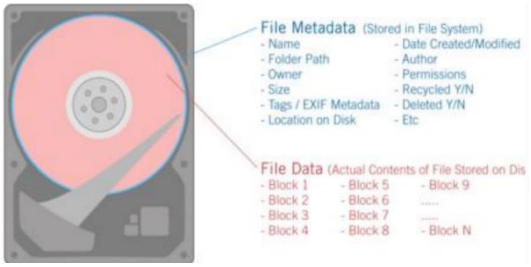


Parameter	1956	2016
Capacity	3,75MB	10TB
Average Access Time	~600ms	2,5-10 ms
Average Life Span	~2000 Hours	~22500 Hours
Price	9200$/MB	0,032$/MB
Physical Volume	1,9m^3	34cm^3

Block Storage and File System:

Block (also called physical record): sequence of bits, can be directly stored on the physical sector of the hard drive

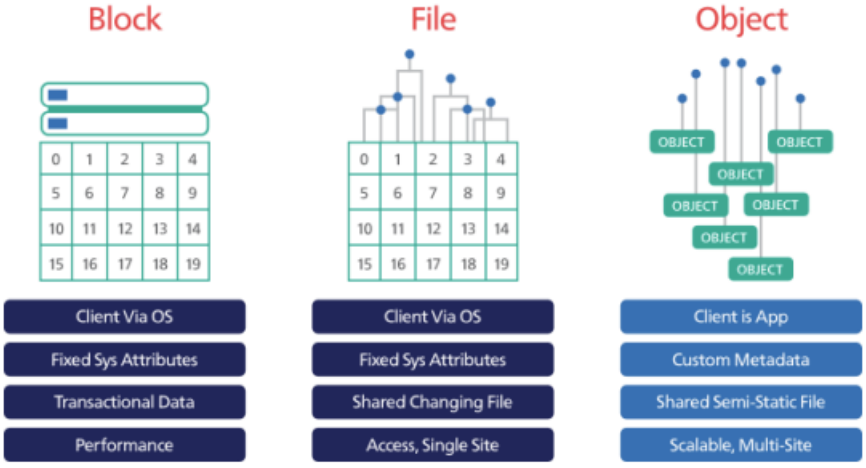
File System: organizes bocks into files and manages metadata to store files on different blocks, usually adopted by OS to store users' and system's files



Object Storage:

Alternative to block storage, new storage type has been created, the **object storage**.
Objects are bundled data (i.e. a file) with the corresponding metadata. Each object has an **unique ID**, calculated on the basis of the file content and metadata. Applications can access an object by its ID, while the set of metadata can be extended (not defined a-priori).
Every object is immutable, a change to an object produces another version, that is stored as a new object (an incremental change system can be introduced to minimize data replication).

In object storage the OS is not an intermediary in the access to stored objects, application can directly access an object by its ID.



Storage Model on Single Server

The storage of the traditional computing model is composed of one or more hard drives connected to one server. Hard Drives can fail, their lifespan depends on the load, however, on average they have an expectancy of approximately 5 years.

In order to handle failures and data loss **RAID** technology is exploited.
There is a physical limitation on the maximum number of hard drives, that is given by the number of trays of the server.
-> to overcome those limitations, distributed file systems are employed.

RAID: Redundant Array of Independent Disks

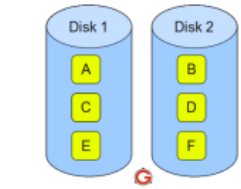
Is a technology that combines multiple physical hard drives into one or more logical hard drives (virtual).
Different RAID schemas are possible depending on the purpose: data redundancy, performance, …

RAID can be implemented either in **hardware** (by a RAID controller) or via **software**.
In **hardware** implementation, RAID functionalities are hidden from the OS, which can only access the logical hard drives.
In **software** implementation, it is the OS which manages the access to physical hard drives and the creation of the logical unit.

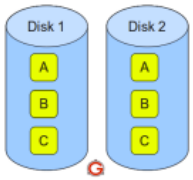
RAID Schemas:

According to the RAID schema adopted, the blocks of the logical drives are stored in the physical drives.

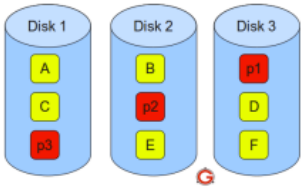
- **RAID 0**: performance RAID; blocks are striped across physical drives
- **RAID 1**: maximum redundancy RAID; blocks are mirrored across the physical hard drives for redundancy
- **RAID 5**: trade-off redundancy – performance; blocks are striped across drives, but a parity block (i.e. calculated via XOR) is added to each set of blocks to ensure fault tolerance (i.e. to recover data if a failure occurs)



RAID 0 – Blocks Striped. No Mirror. No Parity.



RAID 1 – Blocks Mirrored. No Stripe. No parity.



RAID 5 – Blocks Striped. Distributed Parity.

Distributed File Systems (DFS)

Defined to overcome the limitations of single server storage; distributes file system across many servers were defined.
Data transfer and synchronization is achieved via a local LAN. DFS can be exploited either to enlarge the capacity of a single server (by using the storage of a remote server that have) or to put together the storage of many servers. Many distributed file systems implementations: one of the first examples (still widely adopted today) is **NFS Network File System**.

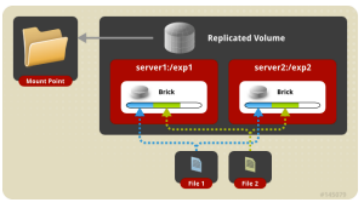
Network File System (NFS)

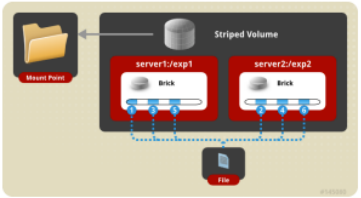
Adopts a client-server approach: a central server offers access to client to its local file system.
Defines a **synchronous** (to ensure consistency and simplify implementation) communication protocol.
The remote file system is accessed by applications running on clients in the same way of local files.
NFS functionalities are implemented at Linux kernel level, so those are hidden from applications.
CONS of NFS: centralized architecture.

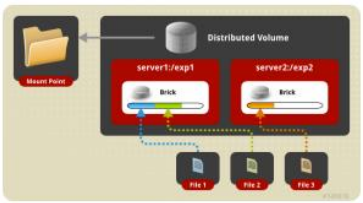
GlusterFS

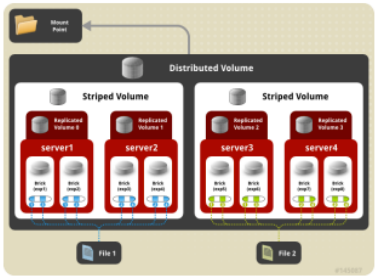
Distributed file system that pieces together storage capabilities available on each node.
Can be used locally in the same way of NFS, but here there is no distinction between clients and server. Each node can participate offering some of the local storage.

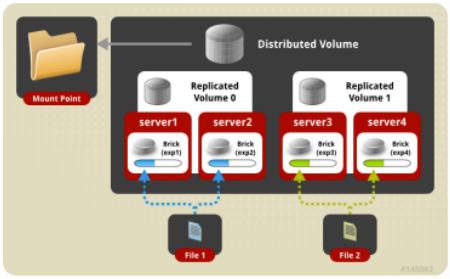
GlusterFS can be **configured** in terms of **redundancy** levels (and **replicas**). In the basic configuration there is the possibility to set up replicated volumes, distributed volumes and striped volumes.
Advanced modes allows striped replicated and distributed replicated modes.











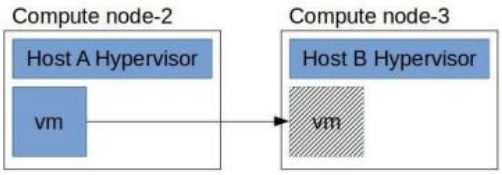
Storage for Cloud

Storage requirements of cloud computing platforms is different from the traditional computing model. Its main requirement is that the storage scales, i.e. it can scale with the amount of data that the platform has to store, for instance:

- It scales with the number of VMs that the platform has to handle
- It scales with the number and the size of the volumes created
- It scales with the amount of data the data generated by different service

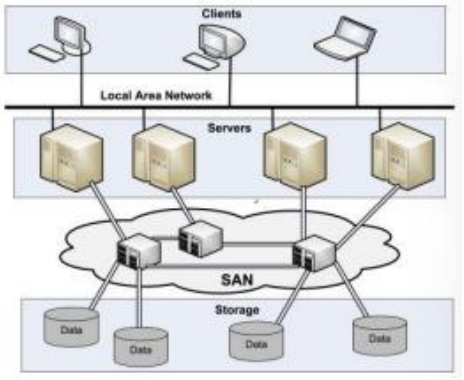
To exploit local hard drives for cloud platforms is possible (in that case the hypervisor creates virtual drives and volumes for VMs on local hard drive), but the hardware limitations of traditional computing models reduce scalability.

VM Live Migration: when a VM has to be migrated from a compute node to another, this is not feasible if the storage employed for the VM drive is on the local file system. -> in that case the VM cannot be moved while up and running. -> this is a high requested behaviour since allows to minimize down-times. Live migration requires a shared storage.



Storage Area Network (SAN)

SAN can be exploited for VM live migration, it is an high speed network made of specific high capacity storage devices (ad-hoc storage devices). Those devices are not regular servers, but in order to provide block-level storage are connected to the servers that compose the cloud platform.
That storage can be exploited by the cloud platform to create VMs virtual drives, volumes, …



SAN storage devices can exploit even tape libraries, but usually disk-based devices are used. High speed communication is accomplished via Fiber Channels.

Specific protocols are adopted for the communication between servers and storage devices, i.e. iSCSI protocol, which is an IP based SCSI protocol that provides block-level access to storage devices.

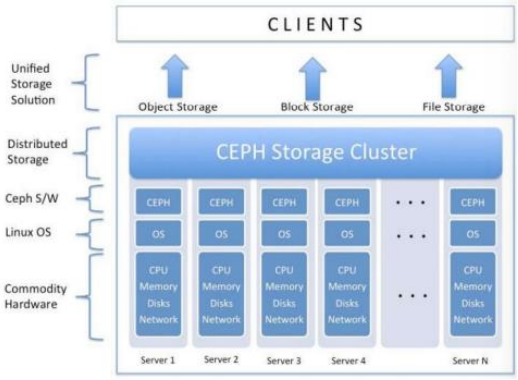
Software Storage Solution

SAN storage devices are compex and expensive, for these reasons not all cloud deployments can justify them.
So **distributed storage system** based on DFS (Distributed File Systems) gained popularity. Those distributed storage systems are based on general-purpose commodity hardware and the goal is to provide high performing, scalable, without single point of failure distributed file system. **Ceph** is one of the most popular solutions for this.

Ceph

Ceph provides an enterprise-grade, robust, and highly reliable storage system on top of commodity hardware. At the core of Ceph there are the following features:

- Every component of Ceph is **scalable**
- No single point of failure
- Solution is **software based**, open source, adaptable
- Run on **commodity hardware**
- **Self-manageable** for all that is possible



Ceph handles autonomously data replication and failure recovery, so RAID is not needed anymore.
It can be installed on regular servers to create a distributed storage system (a Ceph cluster) in which the capacity of all the hard drives of all the servers is made available

Ceph can handle object storage, block storage and file system storage.

Ceph architecture

A Ceph storage cluster is made up of several different software services. Each of these service takes care of unique Ceph functionalities and it is separated from the others.

- **Ceph clients**: interact with one of the other Ceph components, depending on the data storage type
- **RADOS (Reliable Autonomic Distributed Object Store)** is at the core of Ceph, in fact everything in Ceph is stored as objects, independently of the data type seen by the clients. Other storage types are created on top of this object storage.
  RADOS is in charge of keeping data in consistent state and reliable → performs data replication, failure detection, failure recovery, data migration and balancing across the cluster nodes.
  RADOS implements the CRASH algorithm (next pages) in order to guarantee availability, reliability, no single point of failure, self-healing
    - **LIBRADOS**: library available in many languages that exposes native interface to RADOS **object storage**

In order to offer clients storage services other than the object storage, additional services are created on top of LIBRADOS:

- **Ceph Block Device** (formerly known as **RBD: RADOS Block Device**): provides block storage. That block storage can be mapped, formatted, mounted. Equipped with enterprise block storage features such as thin provisioning and snapshots. Can be exploited to create volumes or virtual hard drives for VMs
- **Ceph File System (CFS):** POSIX-compliant, distributed file system of any size. It relies on Ceph Metadata Server (MDS) to keep track of file hierarchy and store metadata.
- **Ceph Metadata Server (MDS):** is used by CFS to store and keep track of file hierarchy and metadata. It is not needed by RADOS or Ceph block device. (Just for file system storage).

Functionalities like high availability, reliability, no single point of failure, self-healing are provided by implementing in the RADOS layer the CRASH algorithm (we will talk about it later).
The RADOS layer functionalities are implemented in a distributed manner by the services OSD and MON:

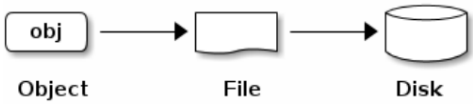
- **Ceph Object Storage Device (OSD):** it takes care of storing the actual data on the physical hard disk drive of each node of the Ceph cluster. Handles I/O operations on disks. Usually there is one OSD instance for each physical hard drive.
- **Ceph Monitor (MON):** monitors the health of the entire cluster. Store cluster state and critical cluster information (status and configuration). A set of MON instances is deployed to ensure fault tolerance.

Ceph Object

A Ceph object comprises data and metadata components that are bundled together and provided with a **globally unique identifier.** The unique identifier makes sure that there is no other object with the same object ID in the entire storage cluster, and thus guarantees object uniqueness. Unlike file-based storage, where files are limited by size, objects can be of enormous size along with variable-sized metadata.

ID	Binary Data	Metadata
1234	01010101010101001101010010 01011000010101001101010010 01011000010101001101010010	name1 value1 name2 value2 nameN valueN

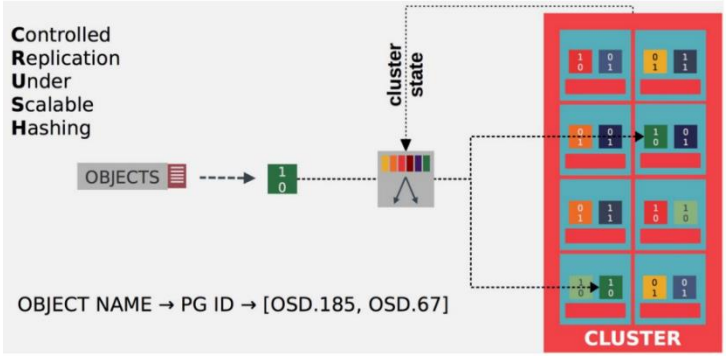
Objects are stored in **Object-based Storage Device** (**OSDs**) in a replicated fashion, which provide high availability. When the Ceph storage cluster receives data-write requests from clients, it stores the data as objects.
The OSD daemon then writes the data to a file in the OSD filesystem.



CRUSH Algorithm

The CRUSH algorithm is used by Ceph clients and Ceph OSDs to efficiently determine how to store and retrieve data by computing storage locations. In this way both clients and OSDs can compute the location of an object without the need for a central lookup table.

CRUSH PROs: better data management; enables massive scale by distributing the work to all nodes in the cluster; uses intelligent data replication to ensure resiliency.



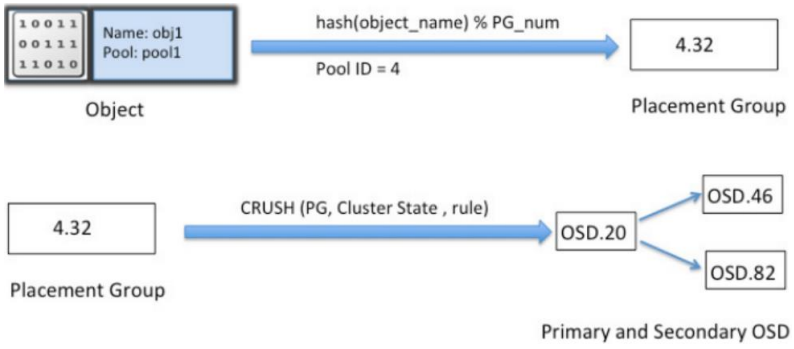
The algorithm computes the location of an object based on the current **cluster status**. The status of the cluster is represented through a Cluster Map.
The Map can be retrieved from any MON instance includes the following information:

- **Monitor Map:** contains the current epoch, when the map was created, and the last time it changed
- **OSD Map**: List of pools, replica sizes, list of OSDs and their status
- **PG (Placement Group) Map**: details on each placement group such as PG ID, set of up nodes, set of acting nodes, state of the PG (e.g. active + clean)
- Cluster Map contains a list of storage devices, the failure domain hierarchy (i.e. device, host, rack, row, room, …), rules for traversing the hierarchy when storing data

A **placement group** determines the actual OSD on which an object is saved (and consequently the physical hard drive, since we have one OSD for each drive).

Data Placement - Client R/W operation

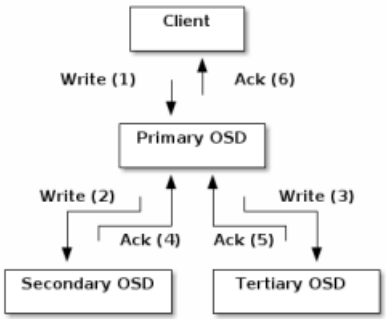
Before reading or writing data, Ceph clients have to contact one **Ceph Monitor** to obtain the most recent copy of the cluster map.
Then the client derives the placement group by applying an hash function to the object ID.
Then the CRUSH algorithm is applied to translate the placement group to the list of OSDs based on the cluster state. The resulting list of OSDs is ordered, the first one is the primary OSD, the others are the secondaries. The primary has to store the object, the secondaries are the replicas. The client sends the object to the primary OSD.



Data Replication

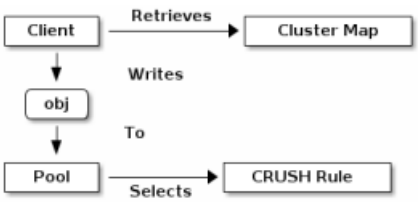
Once the primary OSD receives an object from a client, it computes the list of OSDs for that object by applying the CRUSH algorithm. Then that OSD sends the replicas to the other nodes in the list (as many OSDs as additional replicas). At that point the primary OSD can eventually respond to the client to confirm that the object was stored successfully.

This mechanism relieve Ceph clients from data replication duty, while ensuring data availability and safety.



Pools

The Ceph storage system supports the notion of 'Pools', which are logical partitions for storing objects. Different pools have different CRUSH Rules, but same cluster map. Different pools can be made to accommodate data of different applications, which could have different requirements in terms of replication as an example.



MON Redundancy

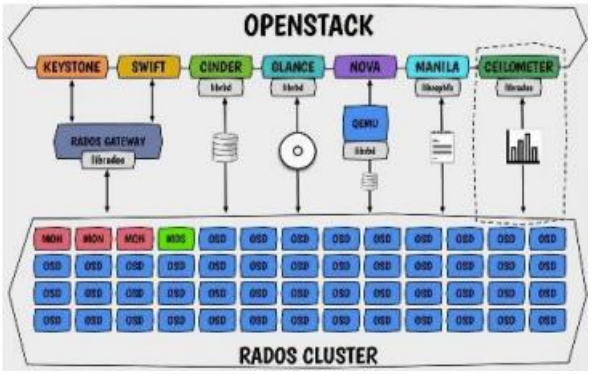
A Ceph Storage Cluster could work even with just one monitor, but this would introduce a single point of failure (CRASH algorithm could not be performed without any Monitor that gives the Cluster Map).
For added reliability and fault tolerance, Ceph supports a cluster of monitors. By the way, due to latency and other causes, it could happen that sometimes a monitor falls behind the current state of the cluster. Ceph must find an agreement among the MON instances on the current status of the cluster. Ceph exploits MON majority and the PAXOS algorithm to establish a consensus among the monitors about the current state of the cluster.

Placement Groups Computation: every pool has many placement groups, CRUSH algorithm maps PGs to OSDs dinamically depending on cluster map. The mapping is computed based on the current failure zones of the cluster, so that data can be considered safe and available even if some component fail. In addition, the mapping is made in a way to ensure balancing across OSDs in terms of disk space usages. Failure zones definition can be modified in order to include also network infrastructure architecture or power supply lines infrastructure.

Recovering and rebalancing:
Since there is a layer of indirection between Ceph OSD Daemons and Ceph clients, this gives us the possibility to grow or shrink or rebalance where Ceph storage cluster effectively stores objects dinamically.
For example, when a Ceph OSD daemon is added to a cluster, the Cluster Map is updated, then all the mappings of placement groups are updated. This changes object placement, resulting in rebalancing (movement of data across different OSDs to ensure proper balance across OSDs).

Ceph and OpenStack:
Many OpenStack services can exploit Ceph as cloud storage technology. For example:

Cinder: can exploit Ceph to store VM volumes (Ceph Block Storage is exploited)
Swift: can exploit Ceph to store its objects by using the default objects storage service of Ceph
Ceilometer: can store telemetry data in Ceph
Glance: can use Ceph to store OS templates for instance creation
Nova: can use Ceph to store VMs' virtual hard drives



Storage as a Service:

STaaS is one of the services usually offered by cloud providers. Offers access to cloud storage capabilities over the internet. This means that cloud providers offer access to their cloud storage infrastructure.
Often STaaS services adopts Object storage model and exposes a REST interface to read / store objects.
STaaS can be used for backup or long term storage of large data.

Swift OpenStack service can be used to expose a Ceph cloud storage.

AWS exposes a large set of cloud storage services, like S3 storage (similar to Swift) and AWS Glacier (for cheap long term storage).

Appendix

SOAP VS REST

Here are the main differences between SOAP and REST:

Protocol:

- SOAP: It is an XML-based protocol that defines a messaging standard for application communication.
- REST: It is not a protocol but an architectural style based on fundamental principles and constraints.

Message Format:

- SOAP: Uses **XML** for message formatting.
- REST: Can use various formats such as XML, JSON, HTML, or plain text.

Structure:

- SOAP: Requires a strict and strongly typed schema.
- REST: Is more flexible and generally less constrained by rigid schemas.

Communication Methods:

- SOAP: Primarily supports the HTTP protocol but can also use other protocols like SMTP and more.
- REST: Relies on HTTP and supports CRUD operations using HTTP methods.

Security:

- SOAP: Has security specifications like WS-Security, providing a set of security extensions for SOAP.
- REST: Security can be implemented through HTTPS and token-based authentication but lacks a specific security standard like SOAP's WS-Security.

So, in **SOAP**, you have the security of HTTP plus the security features inherent in SOAP itself. In **REST**, you only have the security of HTTP. This implies that SOAP is a more secure option.

Usability:

- SOAP: Often used in enterprise scenarios, such as corporate applications and complex web services.
- REST: Frequently preferred for web applications and less complex services, where simplicity and flexibility are more critical.

Cinder VS Swift

Cinder: Provides block storage for virtual machines.
Example: Suppose you have a virtual machine (VM) instance that needs additional storage space. With Cinder, you can create a "volume" of 100 GB and attach it to your VM. This volume acts as an additional hard drive for your VM.

Swift: Provides storage for large object data.
Example: Imagine you have an application that uploads many large images, videos, and files. With Swift, you can create "containers" to categorize and store these objects. For instance, a "Images" container might store all the images uploaded by your app.

In summary, Cinder is like an additional hard drive for your VMs, while Swift is like object storage for large amounts of data such as images, videos, and files. Both provide storage solutions for different needs in an OpenStack cloud infrastructure.

OpenStack VS Kubernetes

OpenStack and Kubernetes are two alternatives, or can they work together?
OpenStack and Kubernetes serve different purposes and, in many contexts, they can collaborate instead of being seen as direct alternatives.

- **OpenStack**
  It is an open-source cloud platform that provides Infrastructure as a Service (IaaS). It allows the management of large groups of computational, network, and storage resources in a datacenter. It is primarily used to build and manage private and public clouds, offering detailed control over the infrastructure and enabling users to deploy virtual instances and network resources on demand.

- **Kubernetes**

  It is an open-source container orchestration system designed to automate the deployment, scaling, and management of containerized applications. It is used to manage containerized applications on a cluster of physical or virtual machines, optimizing resource usage and application scalability.

OpenStack can provide the underlying infrastructure (such as network, storage, and compute services) on which Kubernetes can run. In other words, Kubernetes can orchestrate containers running on virtual machines or physical nodes managed by OpenStack. This integration allows organizations to leverage the benefits of both technologies for a comprehensive cloud-native solution.