Appunti Cloud Computing

Slide A00 Introduction

Slide A01 History of Computing

NIST (*US National Institute of Standards and Technology*) **definition 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 some day be organized as ***public utility***" (e.g. water, gas, electricity).

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

Personal Computers: in late 70s, 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). LAN, WAN were introduced.

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

Parallel Processing: in 80s changed the believed idea that computing performance can be improved only by scaling vertically (by introducing higher performances hardware); parallel processing allows multiple processors (located on the same or on other machines) can work together to solve a single 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 due to the fact that each node was a single point of failure,

then **Cluster Computing** was introduced: nodes were clustered together on the same LAN (less overhead, higher reliability), each cluster was in charge of a kind of task, each cluster had its own cluster head that was in charge of distributing the load.

Grid Computing: in cluster computing each cluster head is a single point of failure, then in 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.

CONs of Grid Computing were that:

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

Hardware virtualization:

To leverage 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:

Web 2.0: where user generated content is central in the experience.

SOA (Service Oriented Architecture): a standard for IT systems software development where software components (/ software services) are interacting each other and re-used by many applications. (systems using SOA are flexible to changes).

***Nowadays**, thanks to the evolutions in networking, software developing and hardware, **utility computing is a reality**, it is cloud computing.*

Salesforce introduced the first product based on cloud computing concepts in 1999.
In 2006 Amazon introduced Elastic Cloud Computing (AWS EC2) and Simple Storage Service (AWS S3).
Microsoft introduced Azure in 2009.
The term Cloud Computing was used for the first time with its current meaning by Eric Schmidt in 2006 in a conference.

Slide A02 Cloud Computing Foundations and Business Models

[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]

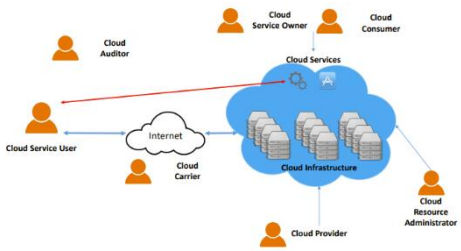
Cloud computing is a buzzword. Historically the cloud term was used to abstract the network in systems diagrams. Used both to refer to cloud services (the applications delivered) and cloud infrastructure (hardware and software in the datacentres that provide those services).

Cloud Computing 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*

Cloud services can expose either an application that is directly accessed by the users (e.g. web pages), or a service that is 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: file storage, ERP service, …

Cloud Computing roles:



- **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.
- **Cloud Consumer**: the organization (or human) that has a formal contract or arrangement with the cloud provider to use its IT resources. (Cloud consumer can sell its service to other customers)
- **Cloud Service Owner**: person or organization that create a cloud service running on the resources provided by the cloud infrastructure.
- **Cloud Resource Administrator**: responsible for administering cloud-based IT resources. Usually belongs to cloud provider organization, but can belong to cloud consumer's too.

- **Cloud Auditor**: third-party that conducts assessments on the cloud environment independently. Typically evaluates security and performances.
- **Cloud Carrier**: provides connectivity between users and cloud provider
- **Cloud service user**: the final user of the cloud service

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:

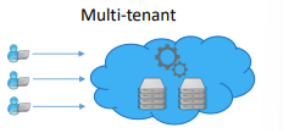
- Adopt a **utility-based** model: pay-per-use price model. Usage of resources is **metered**. Pay what you effectively get.
- Resources can be provisioned dynamically: resources must be instantiated in a short amount of time, without the need of human intervention. Resources can be provisioned not only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

Business Model: cloud providers sell IT resources, other companies buy those resources to create services that they can use internally or sell.

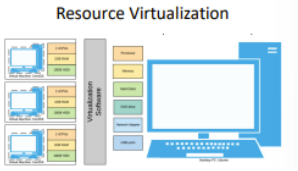
Datacenter: dedicated space to house servers and network equipment. (computing, storage, networking).

Single Tenancy: in traditional computing we have a system administrator that handle the users permissions to access the system resources. The system is installed and configured by that administrator. When the number of users increases, it becomes impossible for the administrator to handle all the requirements of all the users.

Multi-Tenancy: an architecture in which multiple consumers are served at the same time and where there is not a single administrator (single tenancy). We want that each cloud consumer have the impression to be in control of the assigned resources (multi-tenant).



Virtualization: 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 exploiting virtualization, multiple virtual copies of the real resources can be created → cloud consumers have access to the virtual copies, so that they have the illusion that they are in complete control of the resources (multi-tenancy achieved).



Hardware Virtualization (aka System Level Virtualization): allows to create a virtual representation of all hardware resources of a physical machine (a server creating a virtual server). Provides an 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.
Hypervisor (aka Virtual Machine Manager): Virtual hardware is managed and controlled by the virtualization layer. At its core we have the hypervisor, which recreates the virtualized hardware environment in which the ***guest*** operating system runs. (The real machine is called ***host***).
Many virtualization types available:

- Full virtualization
- Para-virtualization
- Operating System virtualization

Those approaches differ for abstraction levels, and amount of resources usage overhead.

In the datacenter many servers host many VMs each, and they are interconnected via LAN. VMs are created dynamically. 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 of virtualized resources can be easily monitored and measured. Cloud consumers are billed per their use proportionally.

Slide A03 Cloud Computing Advantages

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 unsafe public network)]

Traditional Computing Deployment Steps:

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

1. Buy and install hardware
2. Install and configure the operating systems on the servers, configure the network
3. Install software dependencies and deploy the service
4. Keep the system up and running by performing the following steps:
    a. Repair / substitute faulty / outdated hardware
    b. Update the OS and apply security patches
    c. 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**.

In cloud computing the above problems of configuration and maintenance are demanded to 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.

Dynamic provisioning for resources can be done automatically, 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 were costs are shared (high quantity of hardware bought → lower prices; full exploitation of resources → low waste; know how and personnel dedicated to manage the cloud infrastructure is in charge of maintaining the whole infrastructure → economy of scale).

Other advantages:

- **Minimal management responsibilities**: reduces the risks associated to managing an IT infrastructure.
- **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 grants the access to the newest techniques of management of the infrastructure, like high availability and 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: moving applications and services from local computing deployments to the cloud. Local infrastructure are dismantled.

Examples of companies that moved to 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 that have sporadic HPC needs are companies like ***Pixar***, that for short time periods need huge quantity of computing power. (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 accessed from all over the world. In order to grant reachability and low latencies, the service can not be exposed from a single location (in order to avoid network bottlenecks). There is the need to deploy different IT infrastructures in different locations.
→ cloud providers deploy their infrastructure all over the world in order 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 → General Electric is this use-case.

Global-scale content distribution: content could be of various forms (video, music, web-pages, software updates, …). Each kind have its own latency / bandwidth requirements. → King (mobile games, Candy Crush)

Challenges and risks of cloud approach:

- **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. Not always the providers solutions are interoperable. → vendor lock-in
- **Legal Issues**: data centres are placed in location of economic convenience for the cloud provider, but not always the cloud storage location is the same of 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 security of users' data stored in their infrastructure; users do not have control on cloud governance.

Slide A04 Models

Cloud Computing standardization is limited, as its technology is recently introduced. Many models to formalize it, the most appreciated is the NIST one.

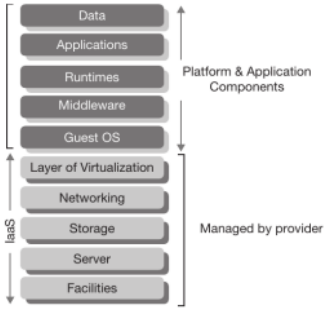
NIST (National Institute for Standard in Technology) Cloud Computing reference architecture model: defines the basic aspects and characteristics of cloud computing:

- **Essential characteristics**: the set of essential characteristics that each cloud computing system must have.
  - Broad Network Access, (infrastructure accessed from anywhere)
  - Rapid Elasticity, (rapid allocation / deallocation of resources)
  - Measured Service,
  - On-Demand Self-service, (resources allocated on request)
  - Resource Pooling. (resources are available on-demand for consumers, large enough to satisfy many users simultaneously)
- **Actors**:
  - Cloud consumer
  - Cloud provider
  - Cloud auditor
  - Cloud carrier
  - Cloud *Broker* (third party that in some cases interposes between the provider and the consumer; it manages the delivery of services from different providers to the consumer and negotiates the relationship → can contract the prices, the consumers can benefit of lower prices by the fact that the broker is requesting services to multiple providers)
- **Deployment models**: the set of way of deploying the cloud infrastructure depends on the requirements of 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, the user accesses the service remotely. In this deployment model we have the highest level of multi-tenancy; the high number of customers allows the provider to exploit economy of scale for technologies and personnel.
  - **Private**: deployed and managed by a single organization for its internal use. Access is restricted. Reside physically to the consumer organization location or in a neutral location. Its management is in the responsibility of the consumer or of a trusted external company. Used when there are specific requirements on the security of data involved. No multi-tenancy at all.
    →One-to-one relationship between consumer and provider.
  - **Community**: allow access to organizations or consumers belonging to a certain community. In the community interests or goals are shared among the participants. 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
  - **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

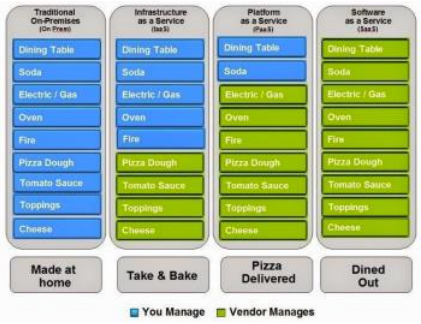
or concerns in terms of data privacy or sensitive to business related data.
The cost of migration and the cost of ownership of the infrastructure must be considered too.

- **Service Delivery Models**: different types of services that 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).
  In **conventional Computing model**: you have to take care of all the aspects: creating the physical infrastructure, installing the OS and middleware and dependencies, developing the service, manage data.
  - **IaaS (infrastructure as a service, aka Hardware as a Service)**: delivers virtualized-hardware resources to consumers, that can be used to create any computer setup (virtual machines). The consumer have to take care of installing the OS, configuring the environment, programming the app and managing data
    →examples: AWS EC2, S3, Rightscale vCloud



  - PaaS (platform as a service): the consumer can design and develop application components. All aspects of the physical infrastructure are managed by the provider. Consumer only have to develop application logic and manage the data.
    The programming environment is provided by the provider.
    → lack of flexibility, you have to use APIs exposed by the provider, hard to port apps between vendors.
    → examples of PaaS are Google App Engine, Microsoft Azure Platform, Force.com, Hadoop
  - SaaS (software as a service): the provider provides to consumers applications that are developed directly by the cloud provider. Consumers use thin clients to access them (e.g. web browser). Everything is managed by the provider, even software licensing.
    →examples: google docs, facebook, flickr, salesforce

- 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 Service Delivery Models:

- **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 fulfill 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.

Slide B-01 Virtualization Technologies

Core resources (CPU, ram, storage) *can be virtualized only if we have physical resources underlying*. *Peripherals can be completely abstract*.

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

Virtual Machine Monitor VMM / Hypervisor:
is a set of control programs that creates the environment for VMs to run. It provides access to the resources, controls and monitors VMs execution.

- **Hypervisor type 1 (bare metal approach)**: runs on bare metal, needs for hw drivers. Bare metal approach. Better performances, reduced compatibility.
  PRO: better performance compared to type 2, provides advanced features for resource and security management → administrators have higher control on host environment
  CONS: limited hardware compatibility
  Examples: Microsoft Hyper-V, VMWare ESX
- **Hypervisor type 2 (hosted approach)**: hosted approach. Runs over a host OS that handles hardware drivers.
  PRO: High compatibility.
  CONS: hypervisor does not have direct access to system resources, all the requests go through the host OS, then there is an higher overhead
  Examples: VMWare and Microsoft Virtual PC

There are different levels of virtualization:

- **Full Virtualization**: enables **unmodified versions** *of standard OSs* to run in virtual environment, lower performances. The hypervisor has to handle all the OS system calls. *Guest OS is completely isolated from physical resources*.
- **Para-Virtualization**: the guest systems must know that are in virtualized environment and you have to implement particular API calls in the guest OS to communicate with the hypervisor in order to participate in the Virtualization management task (calls differ by the specific hypervisor). → Needs modified versions of the guest OSes, closed sources os cannot be modified by the users. The hypervisor does not need system drivers since **system drivers are handled by the guest system**.
  PRO: reduced overhead compared to full virtualization, can run on many different hardware since drivers are handled by guest OSes.
  CONS: you have to modify the guest OS to adapt to the particular hypervisor. Security: the guest OS has high access to physical resources.
- **Hardware assisted Virtualization**: Intel-VT, AMD-V. Some privileged calls from the guest OSes are handled directly by the CPU. Those calls does not need to be translated by the hypervisor. Eliminates need for binary translation or Para-Virtualization. There is need of specific combinations of hardware components.
- **Operating System Level Virtualization (*without hypervisor*):** virtual servers enabled by the kernel of the OS, that is shared among many VMs. No hypervisor at all. 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.

PROs: 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.
CONs: since the kernel is shared, you can have virtual servers running different distributions, but not different OSes.

Other virtualizations: storage Virtualization, network Virtualization.

Emulation: when a system imitates another one
(i.e. a system that has an architecture is enabled through emulation to support the instruction set of some other architecture (e.g. arm - x86)).
Two main implementations:

- **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 is possible on different architectures, virtualization is not.
In regular virtualization techniques we must have the same instruction set between guest VM and System.
Emulation-based virtualization allows to run guest OS written for an architecture to run on another architecture.
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 functions).

Virtualization PROs:
Virtualization enables **server consolidation**: better use of available resources. → Reduce HW costs, simplify system administration and installation,
improve fault tolerance,
improves security (isolated environment).

Virtualization CONs:
single point of failure (if one machine fails, many VM (and then services) fail),
lower performances (85/90% of hw performance)

Slide B02 Virtualization Techniques

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 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: same instruction set in virtualization)

Multiprogramming: the process must have the impression that it has complete control on the processor, and it is the only process executed on it. Exploiting multiprogramming we can have an higher number of processes than the number of CPU physically available. Each process has its own virtual CPU (vCPU).
Virtual processors are multiplexed over time on host machine physical processors.

Virtual Processors (VCPU): each VCPU has its own CPU state, stored in memory. The CPU registers determines its state. When a **context switch** happens, the values inside the registers of the CPU is copied inside the current process VCPU state. 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:

- An **interruption mechanism** (e.g. timer, hardware signal) to implement a periodic (or event based) context switch
- A **memory protection mechanism** to isolate the processes' VCPU states stored in memory
- An automated mechanism to copy the CPU current state in VCPU state (to copy registries to memory)

Two working modes in modern CPUs:

- **system** (or kernel) level → can access all the memory.
- **user** level → limited access to memory, cannot edit VCPU states stored in memory.

Virtual Memory:

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, 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.

Memory Management Unit (MMU):

Handles address translation (from virtual to physical) in hardware. Is a specific hardware component.
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 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, some pages can be stored in secondary storage devices (e.g. hard disk).
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: 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. Once the exception (or interrupt) has been managed, the execution of the program in user space resumes.

Exceptions (aka trap): are internal, synchronous. Used to handle internal program errors (e.g. division by zero, bad address, page fault). A trap can also be a software generated interrupt used as system call that can be invoked in user space from programs.
Interrupts: used to notify the CPU of external events. (generated by hardware devices outside the CPU (e.g. keyboard button pressed)).
Interrupt Descriptor Table (IDT) (aka Interrupt Vector): a table used by the processor to link interrupts and exceptions with handlers. IDT is populated by the OS. Each handler is a function in kernel space.

Full Virtualization:

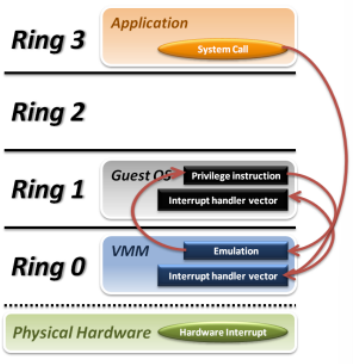
Full virtualization (aka system level virtualization) implementation is facilitated by the presence in CPU architecture of the structures adopted for multiprogramming. → since multiprogramming make each process have the impression to have complete control of CPU and RAM.
In the same way hypervisor must give VMs the impression to have full control on physical hardware (CPU, memory, I/O devices). **The hypervisor role is similar to the kernel role in multiprogramming**:

- VMM code is executed in kernel space, while 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 are not runnable directly) are found → when the context switch gives back control to the VMM, the target instruction is emulated, then the control is given back to VM.
- When many VMs are running on the same machine, VMM handles context switches in order to ensure fairness in the use of physical resources among the VMs.

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

- The guest OS VM code would also run at system level and have complete control of the host hardware
- VMM must emulate the entire processor, not only user level functions → guest OS code must have access also to privileged registers and instructions

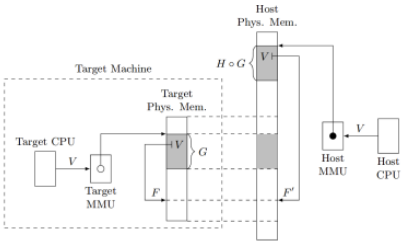
Trap and Emulate Virtualization Model:



Guest OS and guest applications are executed in user mode of the CPU. Every time an exception or an interrupt arise, there is a context switch from guest OS to the VMM. This happens also every time the guest OS tries to run a privileged instruction (i.e. when the guest OS wants to manipulate the MMU, access I/O devices, handle interrupts, …).

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 behaviour.
→ the performance are not affected significantly if the privileged instructions quantity is limited

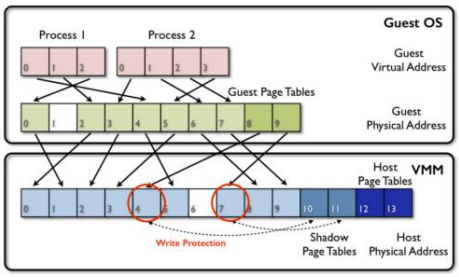
Virtual MMU:



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.

Implementing the Virtual MMU: Brute Force method: the shadow page table:



The page table is modified in order to add an additional level (the shadow page tables) that implements the H function. Shadow tables are write-protected, so that every action that tries to modify them causes a VM EXIT.
CONS: the shadow page tables introduces high overhead and requires constant context switches to the VMM for the translations.
Implementation: the VMM must trap all the possible actions from the guest OSes 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)

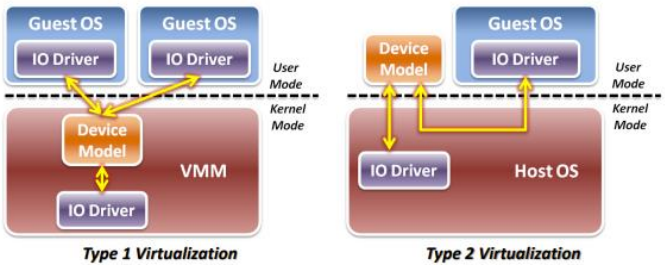
Every time this happens, the VMM takes control and updates (or add) the shadow page table.

Virtualizing I/O Devices:

I/O instructions are usually system level instructions.
In multiprogramming processes access I/O devices through the OS kernel.
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 is accessed by guest VM. The VMM is in charge of keeping the virtual representation up to date. 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:

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



Interrupt Management:
interrupts 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. Each 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.

Virtualized Interrupts:
When an interrupt for a VM arrives, the VMM has to read the IDT of the guest OS target, save registries state, point the Instruction Pointer to the first instruction of the interrupt handler code and then give back control to the VM.

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

➔ Hardware interrupts are always handled by the VMM, which eventually could make a context switch and launch the interrupt handler of an eventual target VM (software emulation of the interrupt).

QEMU:

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.
QEMU can run a VM with the same set of instructions of the VMM. In this case has an *accelerator* to speed up emulation in order to *run some of the code of the guest OS as user mode code*.

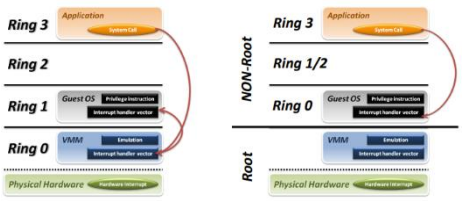
Slide B03 Hardware assisted Virtualization

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 → introduce high overhead (performance penalty)

Intel VMX (Virtual Machine Extension):
Two new operating modes: **root** and **non-root**. Those modes are orthogonal to system and user modes. We have 4 combinations. Root mode is for the VMM running on the host, non-root for guest VM.

Two new instruction for the host system to manage VMs: **VMLAUNCH** and **VMRESUME**. *Allowed only in root + system mode*. When the VM code is at special instruction or for many other reasons we have a **VMEXIT**.

This mechanism optimizes the performances, since we have lower need to emulate part of the code.
Example: the instruction INT and IRET of a VM can be executed without hypervisor intervention since the VM can go in non-root/system mode. Without this hardware support there was the need for emulation of those instructions. (the VMM would have to modify registries of vCPU manually, trigger execution of code, …).



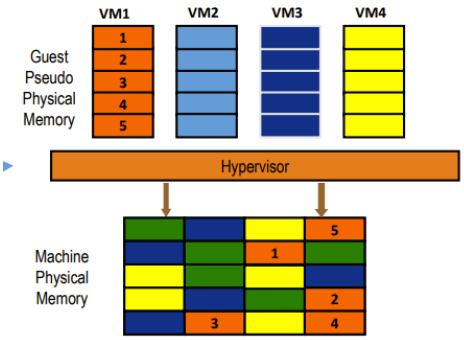
Intel VMX introduces **Virtual Machine Control Structure (VMCS)** that is a structure that contains all information about VM's state. It handles context switches between VM and VM or VM and VMM.

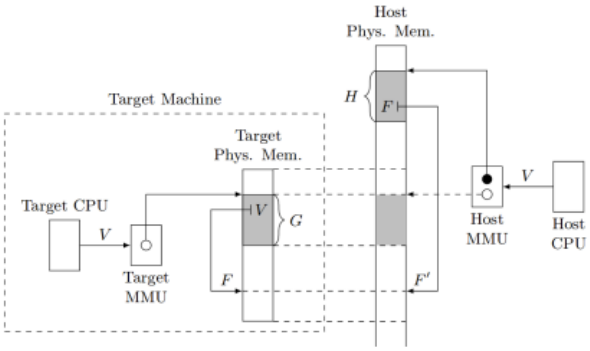
Virtual Machine Control Structure's field are:

- **Guest state** -> contains the state of the vCPU of the VM. (contains VM Instruction Pointer)
- **Host state** -> physical processor state before VM launch. Restored at VM EXIT. (contains VMM IP)
- **VM execution control** -> specifies which actions are allowed in non-root mode. **Unallowed actions triggers a VM EXIT**. Determines interrupts and I/O management (in particular for *peripheral passthrough*). There is a set of flag for I/O operations allowed and a set of flag for interrupts allowed inside VM
- **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** -> for switch from non-root to root.
- **VM exit reason** -> contains info about last VM exit. A code specifying exit reason. Is exploited by VMM to understand VM status and act accordingly.

RAM Management





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.

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

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

---

Hardware Passthrough

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**

There is the need of two functionalities:

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

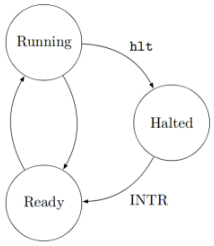
In the VMCS we have an I/O bitmap with one bit for each possible I/O address in order to completely support hardware passthrough. In this way the VMM can set the bits corresponding to the addresses of the peripheral that we want to passthrough.

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. Instead, when a device is passed-through to the VM, the interrupt should be handled directly by the VM (the CPU looks in IDT of the guest VM and executes the corresponding handler without VMM intervention).

Virtual Machine States:



A VM can be in three states: **running**, **halted** and **ready**.
A VM switches from running to halted by the ***HLT*** instruction encountered during VM code execution, while switches from halted to ready by the ***INTR*** instruction.

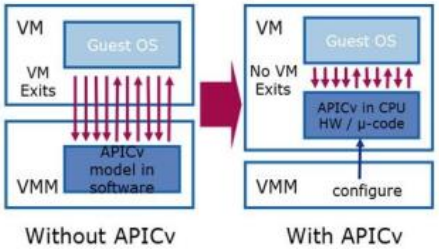
An interrupt coming from a passed through device should be handled always from the VM, beside its current state:

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

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

Posted Interrupts:

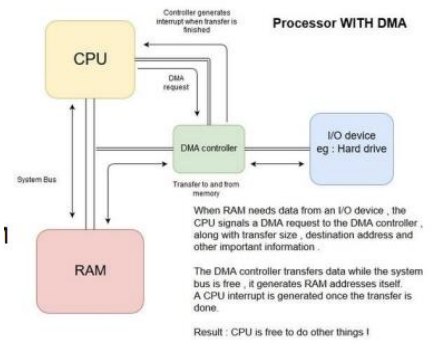
The posted interrupt mechanism allows to store interrupts signals for later notification to the VM. Two additional data for each VM:



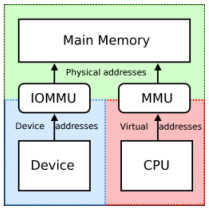
- **interrupt remapping table** (IRT). Contains the correspondence between each type of interrupt and VM (in form of PID or request to interrupt vector of a VM) → one IRT for all the VMs.
- **posted interrupt descriptor** (PID), handles asynchronous notification, contains:
    - **Posted Interrupt Request (PIR)** structure where interrupts are posted.
    - **Suppress Notification (SN) flag,** states if after posting the interrupt in the PIR the controller must notify the CPU or not (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):** 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.

DMA Devices:

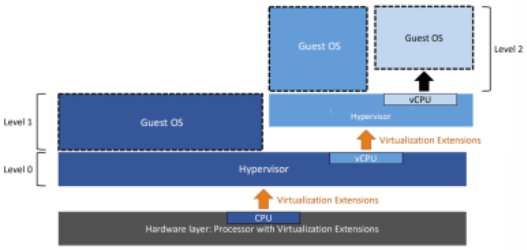


Direct Memory Access devices are initially instructed by the CPU by writing to particular addresses on where to write or read directly on physical memory.



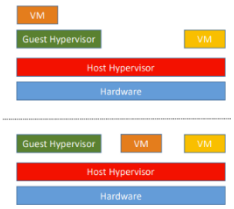
In order to translate Virtual Addresses to Physical Addresses, the DMA Devices exploits **IOMMU**, a specific MMU unit for I/O Devices. IOMMU were projected for multiprogrammed environments, can already be used for virtualization.
→Are also capable of triggering page faults interrupts if find a page that is not stored in memory.

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 to set up an IaaS over another IaaS*.

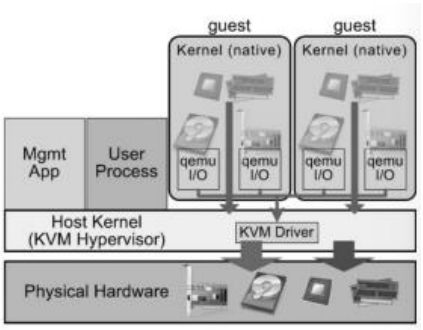
Intel and AMD only support one virtualization level.



→ Many levels can be multiplexed into a single level:

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

KVM Kernel Based Virtual Machine



A virtualization infrastructure of the **Linux kernel** added in **2007**. Runs VM in an execution environment nearly identical to physical hardware. KVM does not run as a normal program inside Linux, but in the 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 B04 (Lab infrastructure, VPN, Full Virtualization)

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

sudo apt-get install qemu-kvm libvirt-daemon-system libvirtclients bridge-utils virtinst

Libvirt: is an opensource tool for managing platform virtualization (create and configure 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: 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 different networks through a 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`.

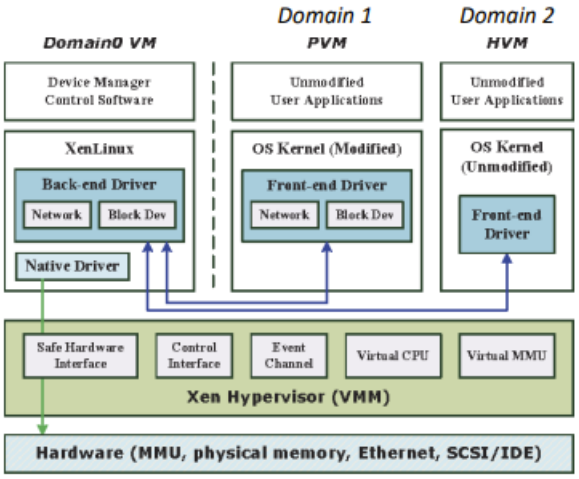
Slide B05 Lightweight Virtualization (Paravirtualization and Operating System Level Virtualization)

Paravirtualization

Modify the guest OS in order to make it aware of being virtualized. Reduces the VMM interventions, needs to edit the guest OS code.
This technique was common before the hardware assisted virtualization.
Those modifications were usually at kernel level, the guest OS application at user level were not aware of being in a virtual environment.

The guest OS kernel has to interface with some predefined VMM APIs exposed through an interface called **Virtualization Layer**. Those calls functions to the hypervisor from the guest OS kernel are called **hypercalls**.
The hypercalls are executed at system CPU level, so that those can interact directly with the hardware.

A popular paravirtualization system was ***XEN***. Released in 2003. Now XES implements both paravirtualization both hardware assisted virtualization.



XES is a small kernel that isolates the real hardware and is placed between the VMs and the hardware.
Each VM is in a domain. Domains are isolated each other. The special **domain 0** is the one in charge of creating and destroying other VMs. Has access to XES APIs. In domain 0 we have to load a Linux OS. Domains can be given direct access to some I/O devices, or they can use fully virtualized devices, or they can use paravirtual devices.

One use case for virtualization is the **safe sharing of hardware resources among different applications**. *Application isolation is not granted anymore by modern OSes*, 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:

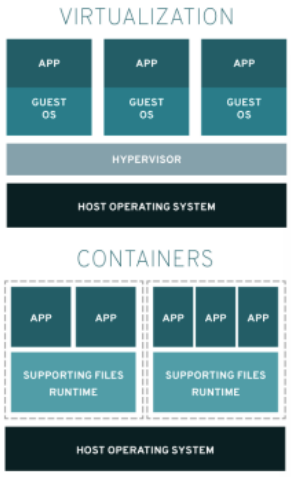
- do not require the virtualization of an entire system.
- guarantee **isolation**.
- **dynamic instantiation**.
- **Self-contained environment**.

The new approaches are the *Operating System Virtualization*, aka **Shared Kernel Approaches**.
PROS: Those reduce the overhead since there is only one kernel, and for that one machine can handle more VMs.
CONS: On the other hand, all the VMs have to share the same kernel, and not all the OSs offer virtualization solutions.

The kernel is shared among the virtual servers, that are enabled from the kernel of the physical machine itself.
Since the OS is shared, it has to implement logically distinct user spaces instances.

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

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 the containers can access may feature a subset of the physical hardware resources.

There is no guest kernel, the kernel of all the containers is shared and is the one of the OS. So the container must be made to use the same kernel as the host.

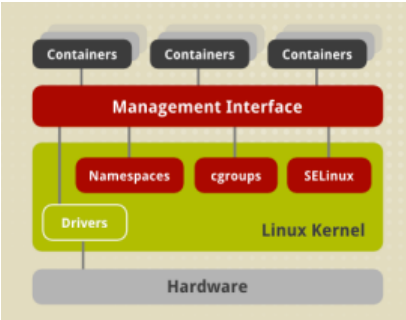
You cannot run an OS inside a container.

Containers are used for software distribution.

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, 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. The specified subdirectory becomes the root for the process. This function works for the filesystem, namespaces were made to extend this behaviour 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).

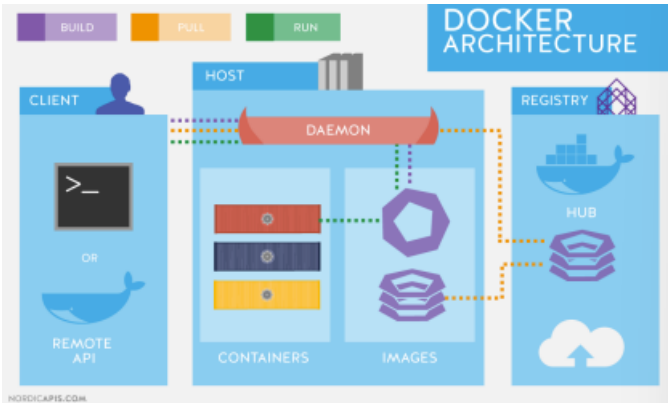
Control groups are groups of processes for which the ***resources usage is controlled and enforced***.

When a process creates a child process, its child is assigned to the same control group (control group inheritage). Each group can be restricted in the access to any resource (i.e. memory, CPU time, …).

Docker

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 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 the images from the registry.

The Docker Engine exposes an interface to users to select and install software packages.

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, but containers with their software bundle, libraries and dependencies ready to run. Cloud providers offers services based on this model, with cloud platforms projected to support it.

LAB B06 Docker

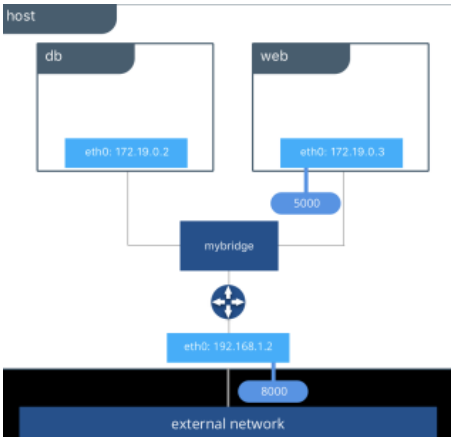
Docker: a container technology that allows to package up an application with all the parts it needs.
Containers: are created from locally available images.
Images: are created from *standard images*, which can be downloaded from *public repositories*.
Docker Daemon: controls all operations.

`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.

Slide C01 Cloud Applications: cloud application structure and architecture

RECAP OF WHAT WE KNOW OF THE CLOUD:

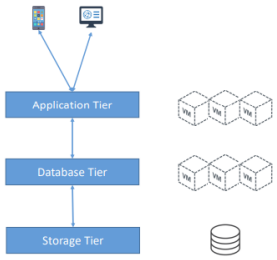
- VMs also have a virtual network adapter, through which they can communicate with other VMs via a (virtual or physical) LAN or with external hosts via the Internet.
- VMs can have access to a cloud storage, a physical or virtualized storage shared among all the VMs usually accessed via the LAN that connect the VMs.

TRADITIONAL APPLICATIONS: we have a physical server and a client that usually communicates over LAN or WAN, data is stored locally.

CLOUDIFICATION: move services, as they are, on the cloud (without changing the implementation). 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:

Are usually **implemented as multi-tier systems** -> *different functionalities split among many VMs*.
Basic example: 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.



The multi-tier system results in an architecture that is made of a **chain of interactions among servers of different tiers**.

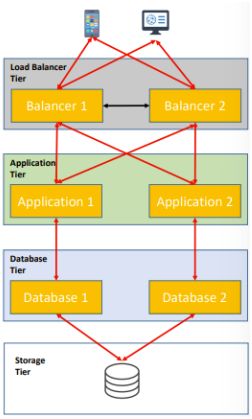
The core-tier servers communicate in response to client request to first-tier application servers. The servers are connected through a **LAN** and use an *application protocol* which can be *implemented by a **middleware** software* **to simplify the development**.

COMPUTE CLUSTERS: 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 of 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.

HIGH-AVAILABILITY CLUSTERS: (HA CLUSTERS) clusters designed to be fault tolerant and support the execution of services that require to quite always available. They operate with redundant nodes to avoid single point of failures. 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 the 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 implement this mechanism, there is the need of some kind of **replication** among the nodes of the cluster *for data, settings and configuration*.

LOAD BALANCING CLUSTERS: (LB CLUSTERS) are designed for higher resource utilization through load balancing among all the participants nodes. The main goal is scalability, so all the nodes share the workload. Requests are distributed. This is achieved by implementing another tier, the **load balancing tier**, between the application tier and the clients. This tier receives requests and forwards them to one of the VMs of the application tier (following a policy for the selection, i.e. the less loaded instance). There could be more than one load balancer in the load balancing tier. They must be equals. The application tier will

know to which database node of the application tier forward the request to respond to the client request. Load is balanced.

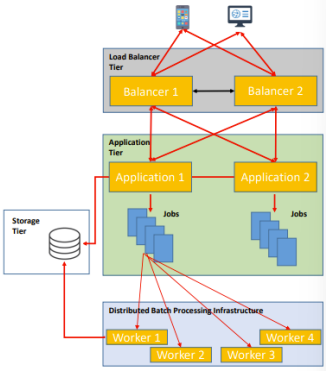


Load Balancing cluster

In order to achieve **Data Synchronization** one possibility is to implement *data replication*, but it introduces much **overhead**. Another way is to **introduce a data storage tier**, to which is demanded the complexity of handling data. This choice depends on the particular application requirements.

COMPUTE INTENSIVE CLUSTERS: (CI Clusters) designed to handle large amounts of data. Case in which the analysis on one single server would take too much time. They 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. The batch processing cluster can have access to the storage tier. Different workers can access to the data to be analysed. The application is required to provide one pointer to the data without transferring it (?Forse che una volta salvato il dataset su storage tier, non può essere spostato e/o modificato?).

LB and CI clusters can easily scale horizzontally. Also in HA clusters we can scale easily (to increase 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*.

Slide C02: Cloud Applications design methodologies

Cloud applications are made of different components interacting in order to offer an application. Expensive and complex task that requires to follow a rigorous methodology. Main requirements:

Interoperability: expose a standard interface that can be easily invoked by other components

Scalability: must easily scale horizontally dynamically and effortlessly

Composability: must be designed to run with different components (VMs are allocated and deallocated rapidly) with minimal (or, even better, without) changes.

Service Oriented Architecture (SOA)

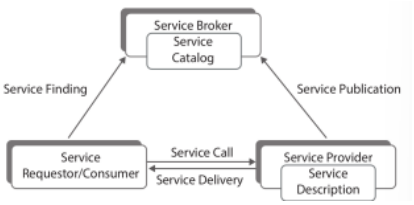
A modular approach for designing software objects. In SOA each software component is called a **service**, that must be *self contained* and *must implement specific functionalities*.
In SOA **applications are built as a collection of services** made from existing software components.
Highly modular applications 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 is implemented via **message passing**, so there is no technological requirement or dependency. There is a **message schema** that defines of you should interact with a specific component and how it must behave to the message and which messages it should handle.
Two roles in message passing: **consumer** service and **provider** service. Each service can be any role depending on the implementation.

Between consumer and provider there is the **service broker**, which is a software module in charge of giving to the consumer the information about the address of the provider and the descriptor (interface) of the provider. The broker manages the *service catalogue* (or service registry) where all available services are enlisted.



SOA Advantages:

hidden implementation → as long as the interface is the same, the service can be modified without interfering with service consumers (/other services). → **loose coupling**: 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**

Web Service: self-contained, self-describing modular application designed to be used by other software applications across the web. W3C definition: *software system designed to provide interoperable machine-to-machine interaction over a network.*
Web Services are the most common instance of SOA implementation.

Uniform Resource Identifier: <uri> ::= <scheme>":"<scheme-specific-part>

URI are also the URN (**Uniform Resource Name**), i.e.: urn:isbn:0-395-36341-1, and also URL

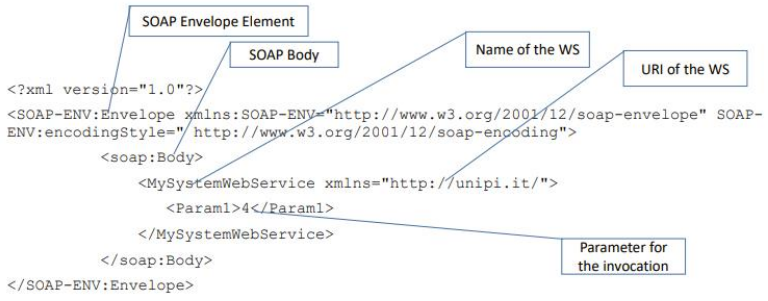
HTTP "… is an application-level protocol for distributed, collaborative, hypermedia information systems".

XML:
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 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**.
DTD and XML Schema give a schema for the grammar of the document: specify what kind of information a document can include, what attributes and elements.

Each Web Service must have an interface described in machine executable format, the **Web Services Description Language** (WSDL). The service interface in WSDL is transmitted to the broker in phase of service provider registration. Service consumer and provider interact using **Simple Object Access Protocol** (**SOAP**), that provides a *standard messaging format*.

SOAP provides a standard packaging structure for transmission of XML documents over various internet protocols. 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. 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 message architecture:
contains in machine readable format:

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

It is made of two main parts: **Service Interface Definition** (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"); the other main part is **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
Is a lightweight alternative for Web Services construction. Based on HTTP transfer protocol. 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 data content is sent through XML, but the additional markup introduced by SOAP is removed. Exploits the minimum support offered by http to implement method invocation.
Parameters can be specified inside the request (query value) or as payload of the request (data to be uploaded). The URI identifies the resource too. REST allows XML, JSON, plain text or any other data format.

JSON JavaScript Object Notation

Uses human readable text to transmit data objects. Originally defined for AJAX but is language independent. Lightweight data encoding, based on two structures:

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

CRUD: Create Read Update Delete

The major functions that are implemented in a relational database. 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.

Slide C03 LAB 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 slide there is a tutorial on how to deploy on Docker a simple flask WS generated with Swagger.

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

REST is the most adopted message passing system in modern applications due to its simplicity, by the way sometimes SOAP is still adopted in the communication between frontend and backend tiers, while in the communication between clients and frontend tier it is adopted hardly never.

Both SOAP and REST 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 (this determines **space coupling**).

CONS: Space and time coupling determine **overhead** and **complexity**.

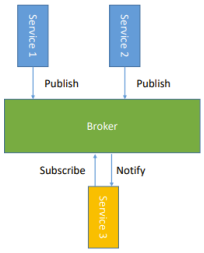
Request-Response paradigm is good for the communication between clients and cloud front-end because client implementations can tolerate space and time coupling, **while in the communication between cloud frontend and backend** this is different: since the backend clusters must assure high scalability,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

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.

Those are reached exploiting 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. (The sender does not need to meet in time with the receiver to communicate, the request can be processed asynchronously).

There are many implementations of indirect communication, but in general those are message oriented, and **bus driven**. Two kinds of intermediary: **broker** or **message queue**.

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**: pubs send data to a named channel, subs receive data from the channels they are subscribed to.
- **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

Senders put requests in a queue, that is consumed by the receiver. There could be just one queue, or one queue for each (type of) receiver. The queues are implemented by the Message Queueing System.

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 in which a receiver (aka consumer) had subscribed before (with the non-blocking receive)

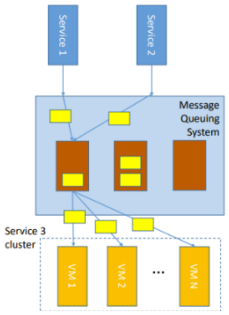
Messages are made of:

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

Queues are **FIFO**: First In First Out, but usually they support also **priorities**.
Consumers can pick messages from the queue also **according to properties** of the messages.
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:
The queues can handle the varying number of nodes in a tier, i.e. nodes from the same cluster can consume a 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 brokers and message queuing systems can be implemented in centralized or distributed manner.
Centralized: straightforward implementation, but single point of failure, lacks performances, bottleneck.
Distributed: needs data replication, more complicated, but guarantees resiliency and scalability.

Advanced Message Queuing Protocol (AMQP): one of the most widely adopted implementation.

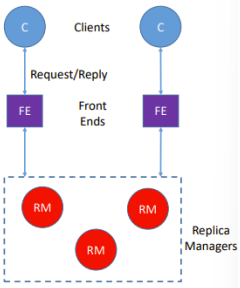
Data Replication:

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**. One solution, as already mentioned, is to adopt a Distributed Storage System (and so, another tier), but also in that case some messages among the nodes of the same tier are needed to keep the data consistent and **keep nodes coordinated**. Requirements:

- **replication transparency**: transparent to users, they should access different copies of data without being aware.

- **consistency**: replicas should be consistent 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 cannot occur (reasonable since the nodes are on the same LAN).

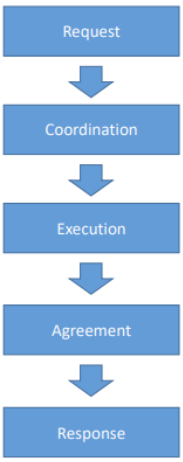
System Model:



The system data is a collection of objects.
Object: 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** (in our case a node of a tier).
To grant consistency *each replica contains a state*, not only the data.
Each replica manager accepts update operations on the data **in a recoverable way**, so that inconsistencies can be recovered.

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

Phases to handle a request:



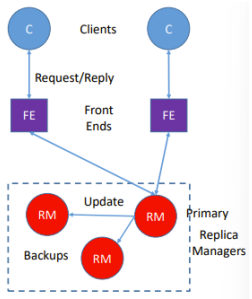
1. **Request received** from the frontend module, which in turns issues the request to one or more replica managers (of the first backend tier). The request is forwarded to other replica managers either by the frontend node or from the first replica manager depending on the implementation.
2. **Coordination**: the replica managers coordinate to prepare for the execution. The execution could be deferred to maintain consistency.
3. **Execution**: the request is executed by the replica managers. (usually executed tentively: in a way that can be rolled back).
4. **Agreement**: the replica managers reach consensus on the effects of the request: if there is an agreement on the effect of the request, the new version of the object is committed
5. **Response**: one (or more) replica manager send the response to the frontend (one or more is application dependent).

Group / Multicast Communication: it is exploited in many phases of the request handling (coordination and data exchange).
Different methods of multicast communication:

- **Reliable / Atomic Multicast**: no guarantees on ordering, the message is delivered to every correct member or to no member at all. Losses in the communication are handled. (low complexity and guarantees)
- **Total / Casual order Multicast**: no guarantees in the 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. (medium complexity and guarantees)

- **View – synchronous Communication**: with respect to each message, all members have the same view all the time. (Update a processed on all nodes, then update a committed, then update b processed on all nodes, …). (Highest complexity and guarantees)

Primary – Backup (aka Passive) **Replication**:

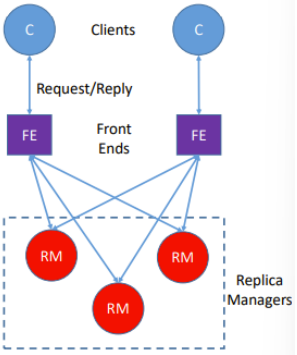


Primary – Backup Replication: one replica manager is selected as primary, while the others are backup replicas. (Guarantees fault tolerance). *Frontend tier nodes communicate only with the primary replica*, which is in charge of sending copies of the updated data to the backups. If primary fails, one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

Primary – Backup Request Handling:

1. The frontend issues the request: giving an UID to the request.
2. The primary replica manager receives the request and directly handles it (executes it), if a request with the same UID was already processed, the same response is sent back and nothing else is done. (→ no coordination needed)
3. Agreement: when the request is an update, the other backup replicas are updated. An acknowledgment is sent. ***Atomic multicast*** is sufficient.
4. The primary responds to the frontend.

Active Replication:



Replica managers are organized in groups, frontend nodes forward the request to each member of the replica group, so that each can process the request independently and reply (identically).
If one node fails, there is no impact, since all the others can reply.

Active Replication Request Handling:

1. The request is multicasted from the frontend tier node to all the replica managers, an UID is attached.
2. **Coordination**: is not 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 repicas send the response to the frontend tier 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

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 frontend instances for load balancing.

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

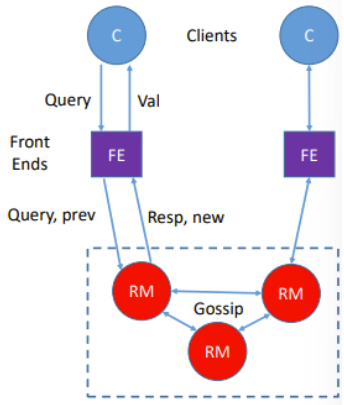
Gossip guarantees:

- 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.

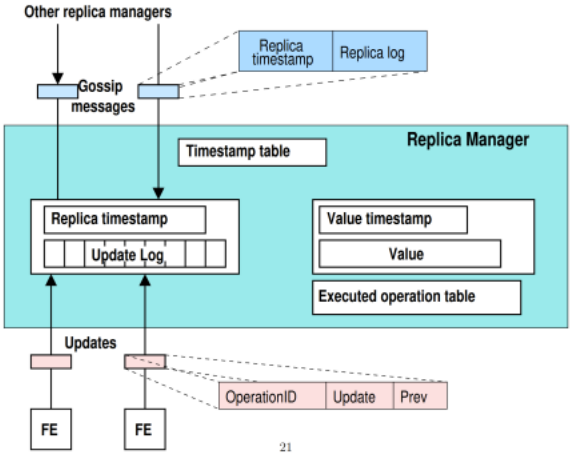
Gossip Architecture – Request Handling:

1. **Request**: the request is sent from the frontend tier node to just one replica manager
2. **Coordination**: the receiver replica manager will not process the request until the required ordering constraints are not respected. This may involve waiting for the next gossip messages.
3. **Execution**: the replica manager executes the request
4. **Agreement**: replica managers update the replicas 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, else 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 frontend node issuing the request. The clock of frontend nodes must be synchronized.



Each frontend node keeps also a timestamp array of the latest version accessed for each object.
That timestamp is sent together with each query.
The replica manager answer with the response and the timestamp of the sent version ("new").
? Each returned timestamp is merged with the previous timestamp ?
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, the request is deferred and kept pending operation in order to wait for a missing update.



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 creates an unique timestamp by incrementing the prev value received from the frontend node and then sends back the response with the updated timestamp (? Di quanto è incrementato ?). Then the replica manager adds a new entry to the local log of pending updates operations. If prev <= valueTS the

update operation is committed, else it is deferred (waiting for more recent gossip updates).

ZooKeeper:

High-available service usually exploited for data exchange among different services.
Can be used to store configuration information, a distributed directory or a naming system.
It is a replicated service, requires that a majority of nodes are not crashed to progress.
Crashed servers are automatically excluded, can re-join on recover.
It exploits a primary – backup scheme to operate and implements Zab, an atomic broadcast algorithm.
https://zookeeper.apache.org/

Slide LAB C05 Message Queue Systems (RabbitMQ)

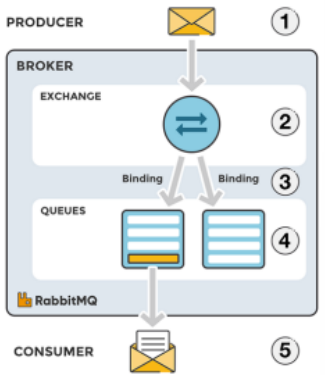
RabbitMQ: opensource message broker that implements AMQP

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).

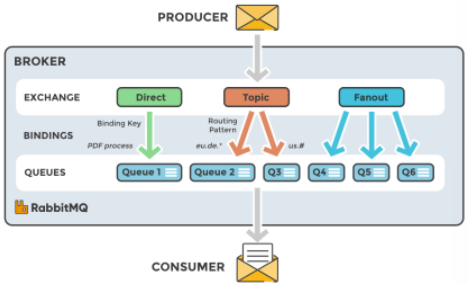
RabbitMQ messages phases:



1. The producer publishes messages to an exchange (that has a type).
2. The exchange routes the message according to message attributes, like for example the routing key. (the rules are defined by the exchange type).
3. The exchange has bindings with different queues, according to rules and bindings, it forwards the message to a given queue
4. The message stays in the queue until it is consumed (by the consumer)
5. 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)

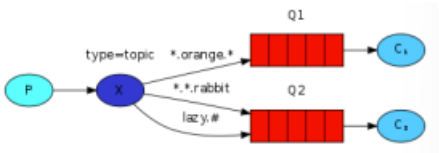
[in the slide there are the instructions on the command to deploy RabbitMQ on docker and on how to interface as a producer and as a consumer with RabbitMQ with PIKA and how to execute the connection]

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]



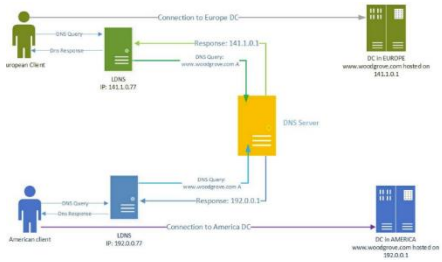
Slide C06: Geographically distributed applications: large scale load balancing and eventually consistent data replication strategies

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 datacentres all over the world)

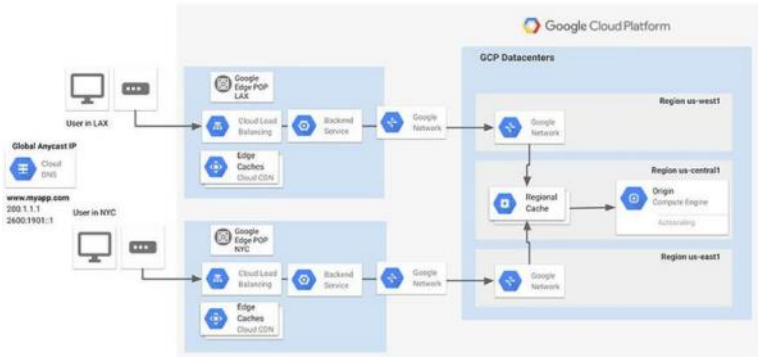
Two levels of load balancing:
Global level: to dispatch globally the requests across different datacentres (different routing policies)
Local level: to decide which VM serves one request within the datacentre (usual routing policy: scalability)

At global level the request could be routed either by the *nearest datacentre* (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 *datacentre with low load or with the least used path* (maximize throughput, not latency).

DNS for Global Load Balancing:
DNS can be exploited to introduce a first layer of load balancing; one simple solution is to s**et many A records for the service**, so that the client will pick one of them arbitrarily.
The main problem is in the fact that the **client behaviour** is **unknown** and in any case does 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).



Global routing with Virtual IP Addresses (VIPs):
A virtual IP address is not assigned to one host, but is shared among many devices. From the point of view of the user it is a regular IP address.
One service could have assigned a *set of VIPs*, and *one VIP to a geographical region*. The localized DNS service returns the VIP address for the region for any DNS request. The request directed to the VIP address of the service is **intercepted by the global load balancer of that region** (that is usually installed in the ISP network or at certain exchange points) is **then encapsulated inside another IP packet**. The new destination IP address is the one of the nearest (or least loaded) datacentre, according to the global level load balancing policy. The destination datacentre is selected by those policies by the global load balancer. The packet is then received and decapsulated by the destination datacentre, and then processed normally. ***Generic Routing Encapsulation (GRE)*** is a protocol adopted to encapsulate packets.

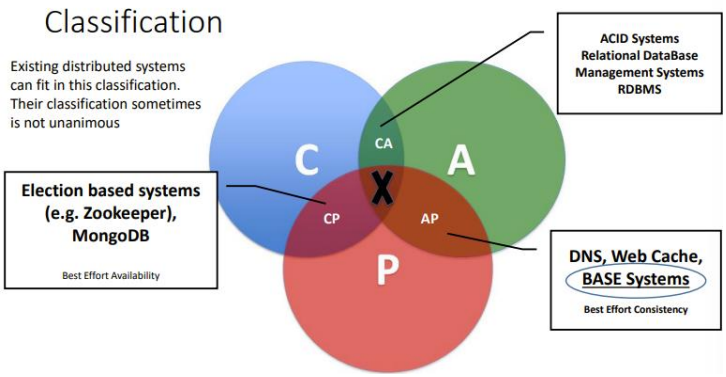


The challenge of Consistency among geographically distributed services:
it is a big challenge due to limited bandwidth, low communication reliability (partitions of the network are frequent) -> since during network partitioning the service must be given in the isolated partition, traditional data replication strategies are not suitable.

ACID systems: traditional systems (i.e. relational database) whose first focus is consistency (then availability). ACID semantic: Atomicity, Consistency, Isolation, Durability.
In ACID systems availability could be achieved by implementing replication in master-slave fashion.

In those systems partitions are considered rare, and when those occurs, the isolated nodes are considered failed.

CAP theorem (Consistency, Availability, Partition Tolerance): a distributed system can satisfy any two of these guarantees at the same time but not all three
Consistency: all nodes see the same data at the same time
Availability: failures do not prevent survivors from continuing to operate
Partition Tolerance: the system continues to operate despite network partitions.



CAP theorems states that *perfect* availability and consistency are not feasible in presence of partitions.

BASE semantic: Basically Available, Soft State and Eventual Consistency
in BASE systems consistency will be reached eventually *after a transition period* (opposes to strong consistency). Depending on the type of service this is acceptable (i.e.: a social network, not a bank system).

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 by the anti-entropy protocol.

Queries handling: both in ACID and BASE systems queries are handled basing only on the local status.

Inconsistencies: due to the deferred propagation of updates, it could happen that **two different versions** of one object could be present (due to deferred propagation, like in gossip architecture), and also that **two conflicting updates are accepted**. In this case according to a **conflict resolution mechanism policy** just one version is kept and propagated (i.e. the most recent update in terms of timestamp).

BASE semantic PROs: handle network partitions, response delay is minimal
CONS: 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

Bayou:
is an eventually consistent architecture that provides data replication and gives weak guarantees on sequential consistency.
Distinguishes between *tentative* and *committed* **updates**:
tentative updates can be undo and reapplied to reach a consistent state
committed updates 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).

Updates are committed by the primary replica manager (differs from the others just for that ability)
The order in which updates are committed is stated by the commit protocol (application requirements dependent)

The list of committed updates is exchanged through the anti-entropy protocol.

Even when a network partition arises, the disconnected nodes continue to work according to the current state of the system, and by keeping the upcoming updates as tentative updates.

Conflict detection and resolution
two mechanisms to detect and resolve conflicts that can be defined by application developers:

- **dependency check**: the developer gives a query, the expected result and the type of update -> before applying the update the system check if the result of the query is the expected one, else arise a conflict.
- **Merge procedure**: those procedures are run when a conflict is detected (merge procedures are provided by the developers)

Both those procedures are deterministic, so that every replica manager resolve conflicts in the same way.

Performance Measurement on Eventual Consistent Systems:

Those systems 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 anty-entropy protocol and **rate*** (how often it sends updates).
Probabilistically Bounded Staleness (PBS): expected time to consistency calculated by knowing the anty-entropy protocol rate.
Many systems allow to set certain system parameters to obtain a certain PBS.

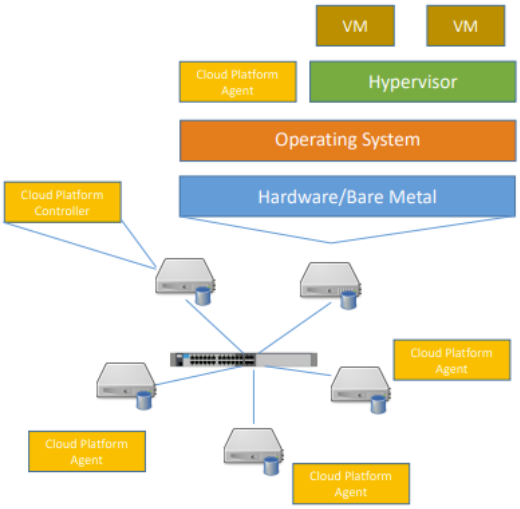
Slide D01-D02 Cloud Platforms

Cloud computing infrastructure: we exploit it to create and control a large number of VMs on a large scale. Exploits unmodified hardware. Groups of servers connected via LAN comprises the cloud infrastructure. Eventually we have a router that connects the LAN with external users. The environment of those server is the datacentre. We have one hypervisor over each physical server.

Cloud platform: physically is a pool of servers, each of them running an hypervisor, federated together. By software the cloud platform is a distributed system running over that pool of VMs (can be considered as a special cloud application). It controls all the hypervisors of the pool (in any moment states which VM is run on which server and with which resources). The cloud platform exposes a **management interface**. That interface could be exploited either by humans or by software to automate some behaviour. The cloud platform allows the implementation of **IaaS** cloud model. On top of that more complex models can be deployed (PaaS, SaaS).

General Architecture:

- LAN between servers (more than 10 Gbps)
- An operating system installed on each server (OS on bare metal)
- One hypervisor on top of the OS that will virtualize physical resources.
- **Cloud platform agent** installed on the OS that is in charge of controlling the local resources.

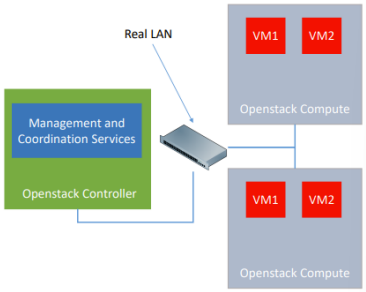


On one (or more for redundancy) server it is installed a particular instance of the cloud platform software called **cloud platform controller**. Could be installed either on a dedicated physical server or on top of an hypervisor in a VM. The cloud platform controller is in charge of controlling the overall platform by communicating with the cloud platform agents.

Cloud Platform Implementations:
commercial public platform have hidden implementations, but are all similar to the open-source alternatives. There is not a standard for cloud architecture, we will study **OpenStack**.

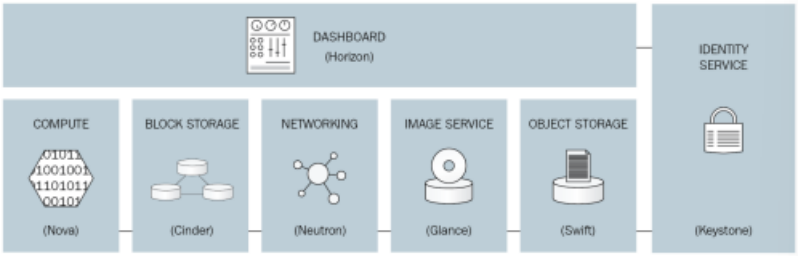
OpenStack: opensource, started in 2010 by NASA and Rackspace Hosting, supported by OpenStack foundation, made of over 500 companies. It is the main solution for private cloud computing. Adopted at CERN in Geneva and Budapest too.

OpenStack Platform: the OpenStack cloud platform agent is installed on top of the OS (that must be Linux). OpenStack works with many hypervisors. Every server on which installed OpenStack is called **OpenStack node**. The platform handles the connection of VMs among them and to the external network. The platform handles the storage too, that is shared and virtualized (virtual hard drives can be connected to VMs). VMs on different OpenStack nodes can be connected together seamlessly; *nodes are configured to form a **single OpenStack instance***, as if we have just one physical hardware on top of which VMs are set up. In one instance *at least one node is set as* **controller**, while *the others are* **compute nodes** (computation and storage resources for VMs).



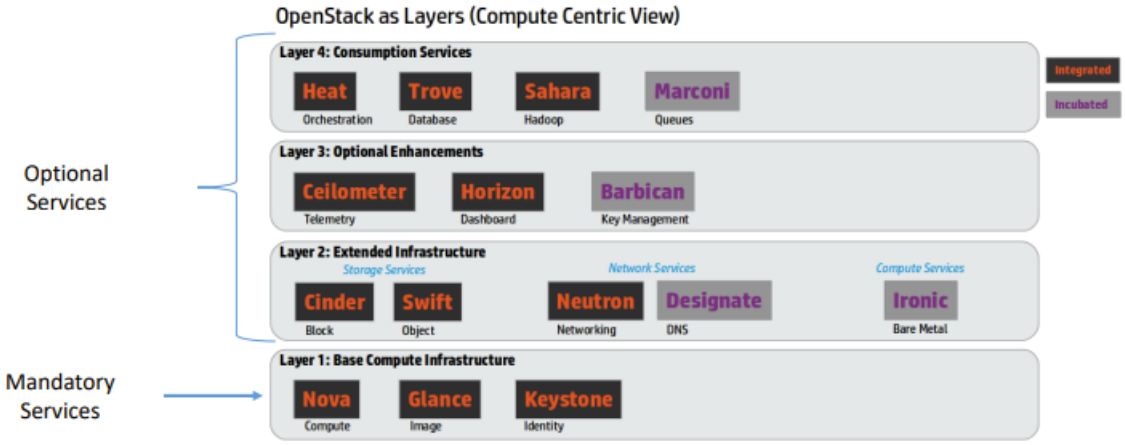
The real LAN infrastructure among physical nodes is exploited as infrastructure for the virtual networks and for the communication between OpenStack components for coordination and management.

OpenStack interface: it exposes a web dashboard, REST APIs, a command line tool.



OpenStack modularity: one service to handle each type of resource (network, storage, computing). Each service is implemented as a different and separated module. Each module can work separately. Each module exposes its APIs.
Coordination and status information among services are exchanged using a message queue system.

Apart from Core Services, that are mandatory, the others are marked as optional.



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.

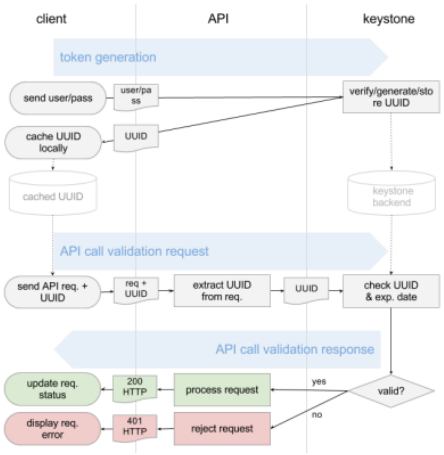
The services of the controller node leverage supporting services, like one database (e.g. MySQL) for status and configuration storage, and one message broker (e.g. RabbitMQ) for message exchange among modules.

Information Exchange among services: could be either via **Message broker**, **REST API** (perform API requests as an external program would do) or **Database** (store configurations in the database that is accessed by both). 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.

Keystone (Identity Management Component):

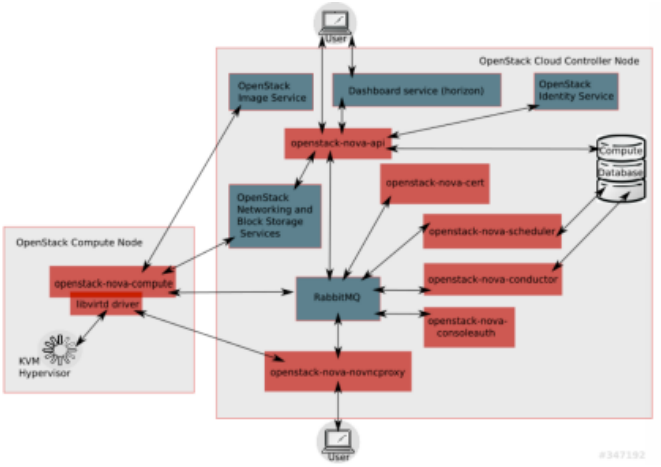
Usually installed on the controller node, used by OpenStack for authentication and authorization.
Ensures security by granting or denying access to **objects** (VMs or Virtual Networks) to different **Users**. Objects are grouped into **projects**, that are assigned to users.

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.

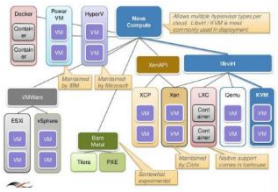


Nova (VM management):
manages the compute service: *installation and management of virtual machines*. **It interfaces with existing hypervisors** to manage VMs (compatible with many technologies, like KVM, Xen, Libvirt, QEMU, Docker, VMWare ESX, …).
It is made of two components: **Nova Controller**, installed on the controller node, and the **Nova Agent**, installed on compute nodes. The Nova Controller handles the requests of VM management coming from the users and evaluates the action to be performed and sent 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 <u>Controller</u> Subcomponents:
all of them interacts via messaging queue.

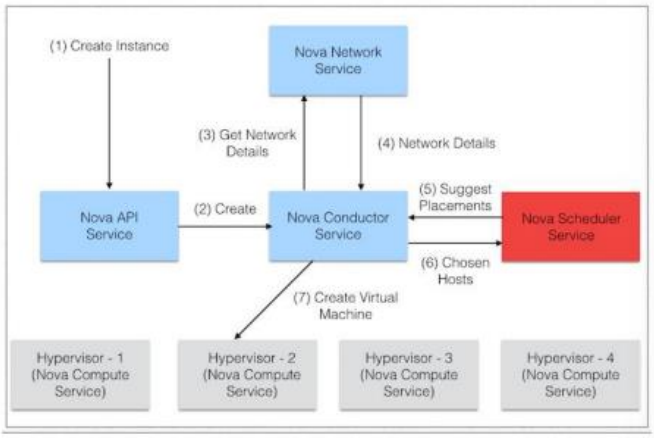


- **Nova-API**: RESTful interface used by users and other services.
- **Nova-Conductor**: manages the control operations.
- **Nova-Scheduler**: decides where a new VM will be instantiated (on which compute node) according to the available resources.
- **Nova-network**: implements basic networking services for VMs. If other network service is installed, it interfaces with it.
- **Nova-consoleauth and Nova-novncproxy**: implement a console service through which users can connect to VMs.



Nova Agent's (aka Nova Compute) only subcomponent is the **compute service**, which receives the actions from the controller and interacts with hypervisor (many drivers are maintained, one for each hypervisor; each of them exposes the same interface to the compute service).

VM Creation Workflow: request received from Nova-API -> Nova-Conductor asks the Nova-Scheduler which node will host the VM -> Nova-Conductor asks to Nova-network to configure the network for the new arriving VM -> Nova Compute of the Compute Node chosen receives the request of instantiating the VM and asks the hypervisor to do it.



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.

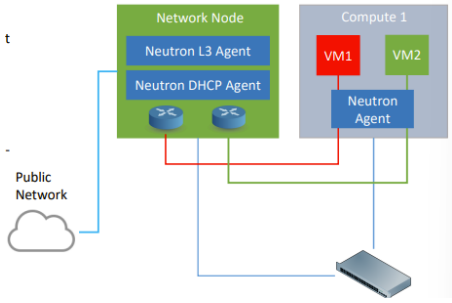
Glance (image management service):
Each VM is instantiated from an image, which includes a specific operating system pre-installed and pre-configured. (*Custom images* can be done; e.g. with a webserver preinstalled). Glance manages the VM **templates**. Glance exposes a REST API, is made of two components:

- **Glance** (stores the available images informations on DB, exposes the **API**) and
- **Glance Storage**: 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):
Many Virtual Networks must be instantiated, VMs of different tenants must not communicate. Neutron is in charge of instantiating the Virtual Networks, that could also be among VMs that are on different compute nodes. The physical network is exploited to enable the VNs.
Neutron subcomponents:

- **Neutron Server**: exposes the REST APIs, is installed on the controller, manages the virtual networks by coordinating the Neutron-agents running on compute nodes.
- **Neutron Agent**: manages the traffic coming from / going to the VMs that are on the compute node on which is installed. Dispatches data (exploiting the physical LAN network) according to the virtual networks' rules. It enforces isolation between different virtual networks.

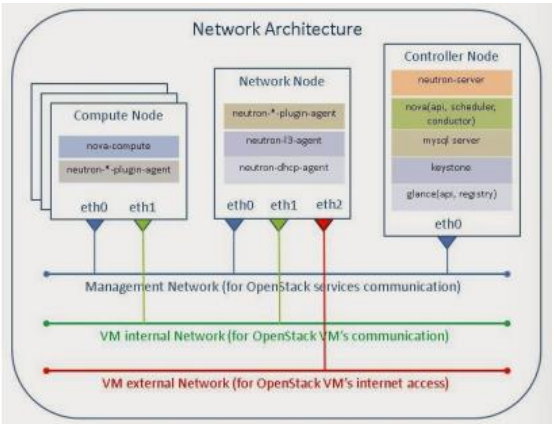
Neutron Network Node: it is a special node that is connected to the external network (the internet). Runs a particular Neutron-agent called **Neutron-L3-Agent**, that is responsible for rerouting the traffic between private virtual networks and public networks. This is accomplished by **Virtual Routers**, 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).

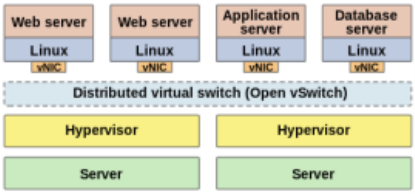
Physical Network Infrastructure:
Usually in an OpenStack infrastructure there are three different physical networks:

- **A management network**: for OpenStack services communication.
- **A VM internal network**: that connects the compute nodes each other and is exploited by Neutron for realizing the virtual local networks among VMs.
- **An external network for internet access**: usually only the Neutron-Network-Node is connected to that network.

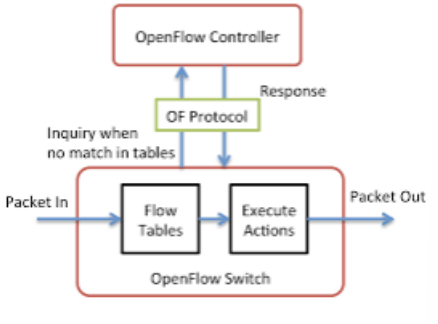


Note: OpenStack management traffic and internal VM traffic sometimes are transmitted on the same LAN
The controller node is connected only to the management network.

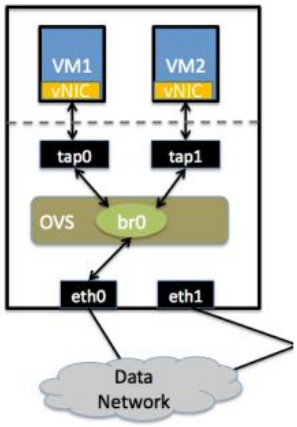
Neutron Agent: handles and forwards the traffic, is highly configurable. The latest versions exploits **OpenVSwitch** to process data between the compute node and the Network Node. OpenVSwitch *allows to rapidly reconfigure the behaviour of each agent following the directives coming from the Nova server*.



OpenVSwitch (OVS) is an open-source implementation of a distributed network virtual switch. Designed to provide a switching stack for hardware virtualization environments. Can run within (or jointly with) the hypervisor. Supports packet filtering, routing, switching and manipulation. Adopts a **centralized approach**: every node has its instance, that are managed by a **controller**. OVS adopts software defined networking paradigm.



Software Defined Networking (SDN): network rules are configured dinamically on each instance basing on directives coming from a centralized controller. Each node has a table called **flow table** that describes the features of any incoming packet (source IP, destination MAC address, …) and the action to perform when the features are matched (forward, modify, …). If for a coming packet no match is found, OVS sends a query message to the controller (so called **OpenFlow** protocol), that answers either with a new rule to be stored in the flow table, or with a specific action to be performed once.



VM side Virtual Network Implementation:
every VM has a virtual NIC (network interface controller) that is linked to another virtual NIC that is created in the host operating system called **tap**. A tap is a virtual network interface that can be linked with a software. In our case it is linked with the hypervisor, and is exploited to receive / send data traffic to the VM. Traffic coming from tap and from physical interfaces is managed by OVS.

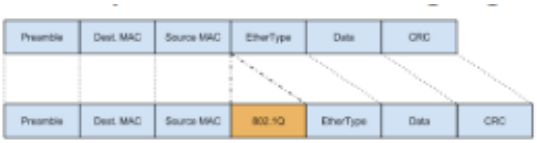
OVS on the Network Node:
Also on the Network Node it is installed an OpenVSwitch instance that is in charge of implementing L3 virtual routing functionalities to route traffic between external networks and local virtual networks.

Network Virtualization: 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

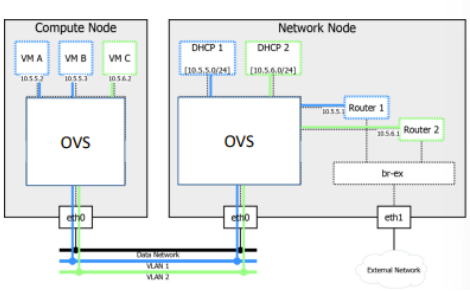
In order to dispatch correctly each packet, it must be marked with the ID of the VN to which it belongs, so that when two packets belong to the same VN can be dispatched without passing through the Network node. Different network technologies are adopted: packet tagging and packet tunnelling.

Virtual LAN: one of the most widely adopted network standard for network virtualization. IEEE 802.1Q defined to allow creation of virtual LANs on top of regular ethernet networks. This standard adds a field to the ethernet header, the **VID** (VLAN ID). 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.

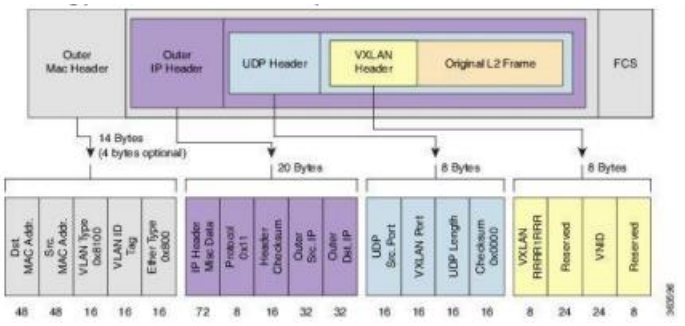
Neutron can exploit **VLAN** to implement Virtual Networks, in this case it generates a VID when a new VN is needed. The server configures the OVS so that the packets with that VID will be sent to the VMs of that VN. If that VN is also connected to external network through a router (inside the network node), then also the OVS running inside the network node is configured with the VID for that VN for the packets passing through the router of that VN.



Generic Routing Encapsulation GRE Tunnelling: alternative virtualization mechanism to VLAN.
Allows to encapsulate network IP packets inside another IP packet.
It is used to create a tunnel between two internet hosts. It adds an additional GRE header needed to specify the encapsulated payload protocol type.
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 Nova server is in charge of instantiating the GRE ID for each VN, then each compute node that contains at least one VM of a particular VN is instructed to create a GRE tunnel with each compute node in the VN.

VXLAN (virtual extensible LAN): in VLAN, VID identifiers are 12 bit long, so the maximum allowed number of VLANs is 4096. In large cloud implementations this is 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, at IP level 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 Nova server instructs the OVS instances to encapsulate packets into VXLANs tunnels according to VN instantiations.

Cinder (OpenStack component for volumes management):
Each VM has always a default volume that stores the operating system. Additional volumes can be dinamically created and attached to an instance.
Virtual hard drives are stored on 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.

iSCSI: internet protocol based storage networking standard that provides block-level access to the storage. Cinder stores the volume images in the local file system of the node on which is installed or in a cloud file system.

Cinder architecture:

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

---

Ceilometer (OpenStack 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. Cinder collects also the **telemetry** statistics that can be used to monitor the status of the system.

Ceilometer architecture:

- There is a centralized **Ceilometer-Collector** that is responsible for receiving the data from all the **Ceilometer agent** and store them in a DB (usually a NoSQL DB like MongoDB), it can eventually expose a set of *REST APIs to retrieve collected data*.
- There is also a **Ceilometer Agent** on each node that collects data from all the components and forwards them to the Ceilometer Collector.

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

API of OpenStack services: are RESTful, used both for inter-service communication and to exposes functionalities to users, users can use them to embed automation processes in external applications.

Swift (OpenStack object storage service): allows users and OpenStack modules to store data (represented as objects) in the cloud platform. (i.e. Cinder or Glance can exploit it to store volume backups or images). Swift is responsible for receiving, storing and retrieving data.
Swift usually stores data in a cloud storage to guarantee scalability. (local data storage in the compute node is allowed too).

Swift Modules:

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

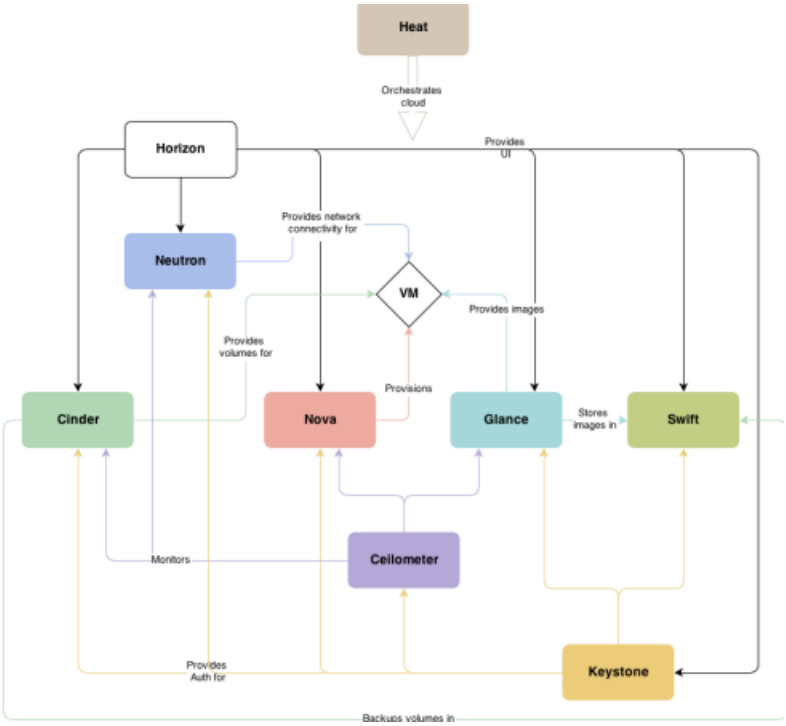
Heat (OpenStack orchestrator): manages VMs creation and destruction automatically. The automation of these processes is based on a set of rules that specify when to trigger actions on the VMs.

Example: *through Heat an autoscaling service can be created that exploits the stats collected from Ceilometer to create or destroy VMs*.

Heat modules:

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

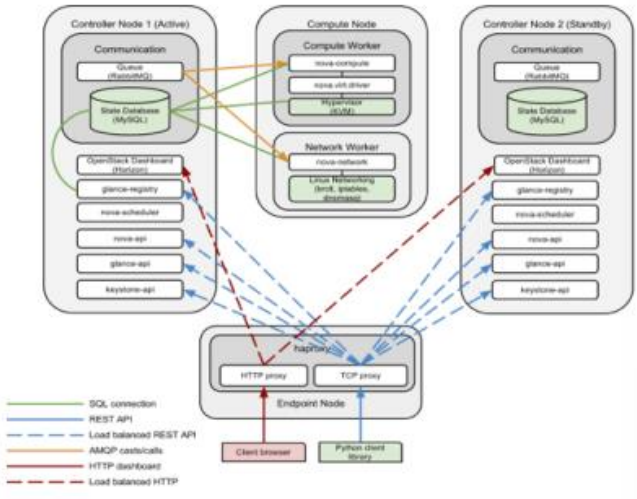
OpenStack services interactions summary:



OpenStack high availability: the standard OpenStack installation includes only one instance of each managing components. By the way OpenStack can be installed in the high availability configuration, in which multiple instances of each service can be deployed to ensure *resiliency*. The same can be done for cloud applications, so that are deployed in redundant manner to guarantee high availability. In that configuration many controller nodes are deployed for the same set of compute nodes.

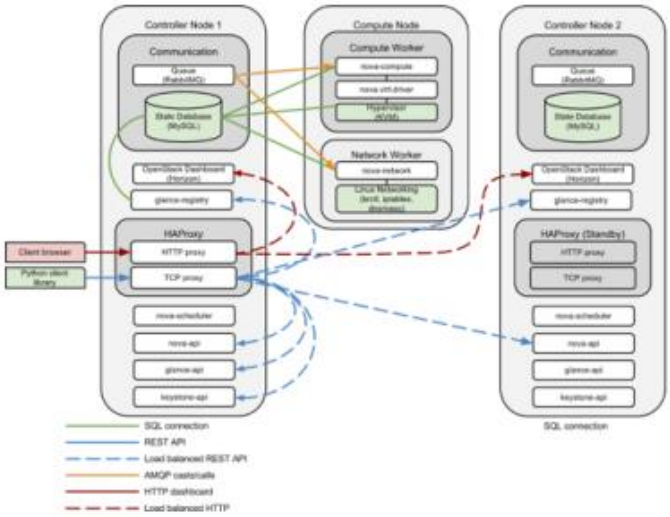
High Availability with Dedicated Node:

In this configuration the OpenStack APIs are exposed through a High Availability proxy. That proxy is responsible for receiving the requests and dispatching them to one of the controller nodes. The proxy checks when a node fails and removes it from the list of active nodes. The **State storage must be replicated across the controller nodes**.



High Availability with simple redundancy:

One of the controller nodes is selected as the master node and is in charge of handling the API requests.
The functionalities of the HA proxy are on all the controller nodes, which coordinates them and detect failures and trigger reconfigurations when needed.
The State storage must be replicated across the controller nodes.



Slide D03 LAB 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…

Slide D05 Cloud Platforms: Lightweight cloud computing platforms for DevOps

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. 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 packaged the whole system and passed it 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.
Legacy systems are nowadays disappearing, but some of them are still maintained and developed. Some of them have also been cloudified.

Software development now: big monolithic applications have been broken down in 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.

Application development process: before there was a team for application development (dev team), and another for application deployment and maintenance (operation team), nowadays there is just one team both for development and deployment and maintenance for the whole application lifecycle, called **DevOps**.

DevOps approach: the aim is to have developers more involved in running the application in production. This leads to have a better understanding of the users' needs and issues and also of the application deployment problems.
The deployment process is streamlined -> it can be performed more often -> you have developers deploying their applications by themselves without involving another team (old ops team) this approach is called **continuous development**. This approach needs the developers to have deep knowledge of the underlying infrastructure, which is not always feasible or desirable.

Usually developers and operations (aka *system administrators*) have different backgrounds, experiences and motivations. It is not always easy for a developer to learn how to take care of the infrastructure.

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

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 could differ. Any service can change arbitrarily its dependencies whenever its development team wants. This is very difficult to handle by operations team.

Consistent Application Environment:
It is hard for developers and operations to deal with the different environments in which applications are run.

In order to standardize this you can deploy applications inside **VMs** with ideal confined environment where all the libraries are preinstalled and the real hardware is abstracted.
CONS: This solution by the way is expensive in terms of computing and resources overhead introduced. There is also a VM deployment overhead.

Another solution that standardizes the environment and that has less overhead is **Lightweight Virtualization** mechanisms, like **containers**.
Containers can be standardized, with preloaded libraries and dependencies, and can be run on different machines using ad-hoc distribution systems. An example of this solution is *Docker*.

Kubernetes: a software developed by Google that allows to *easily deploy and manage containerized applications*.
Containerized Application: an application whose components (independent services) are deployed and shipped as different containers.
Kubernetes aims at simplifying the deployment and maintenance of containers by *taking care of aspects regarding scalability, failure tolerance, resource management*, …
Kubernetes aims at offering a **lightweight IaaS infrastructure** to deploy applications in a simple and effective way.
The operations team in a Kubernetes environment is required in order to maintain and run the Kubernetes infrastructure itself.

Kubernetes Core Operations:

Kubernetes is a distributed platform made of a **master node** and a varying number of **worker nodes**.
Kubernetes nodes *can be installed either directly on **physical** servers **or** on top of **VMs** of a cloud platform*.
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 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.

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; operation team do not have to take care of deploying, configuring, fixing failures; Kubernetes can achieve higher utilization of the resources available in the infrastructure.

Cluster architecture:

Master Node: implements the **control plane** (which controls the cluster). The control plane is made of many components, each of them can be installed on a single master node, so it means that can be split across multiple nodes. The components can be replicated in order to ensure high availability.

Control Plane Components:

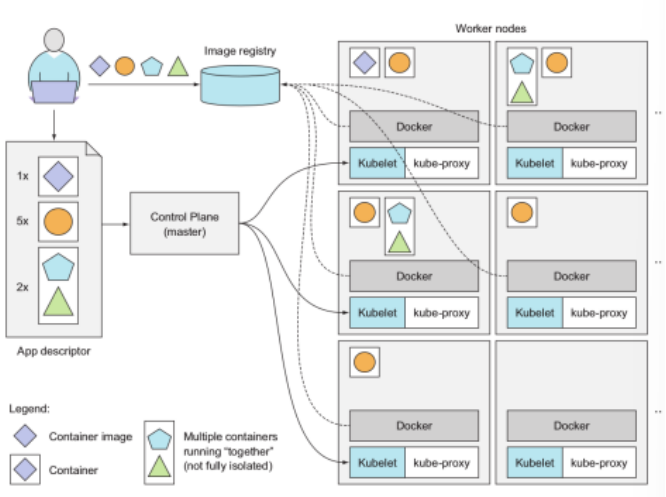
- **API server:** exposes an interface to developers.
- **Scheduler**: schedules app
- **Controller Manager**: performs cluster level functions like replicating components and handling node failures.
- **Etcd**: reliable distributed data store

Worker Nodes 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 that must be run together on the same node are grouped in so called **pods**. Containers that compose a pod must be run on the same node and must not be isolated. A pod can be made of one container too. For each pod the developer specifies the number of replicas required and the image to be used to instantiate it.



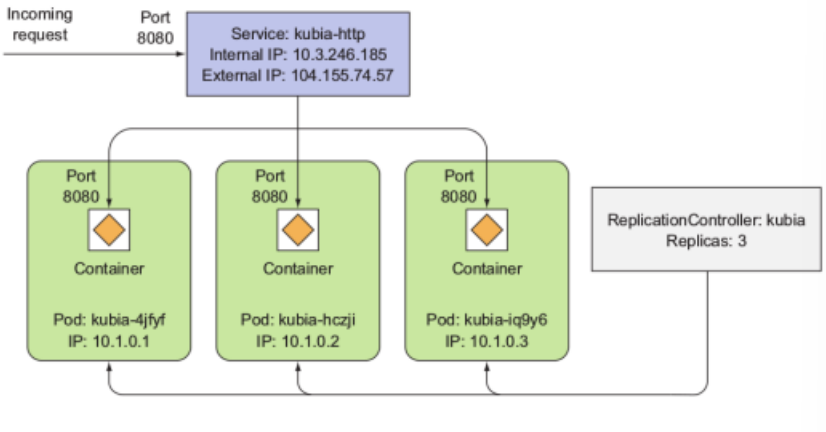
When the developer submits the descriptor to the master node, the control plane takes care of informing the scheduler to schedule the required number of replicas of pods in the available working nodes. The Kublets instance of each of the selected working nodes takes care of instructing the container runtime (usually Docker) to pull 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.
Kubernetes can even decide autonomously the optimal number of instances of a certain container based on *real time metrics like CPU load, memory consumption, query per second*, …

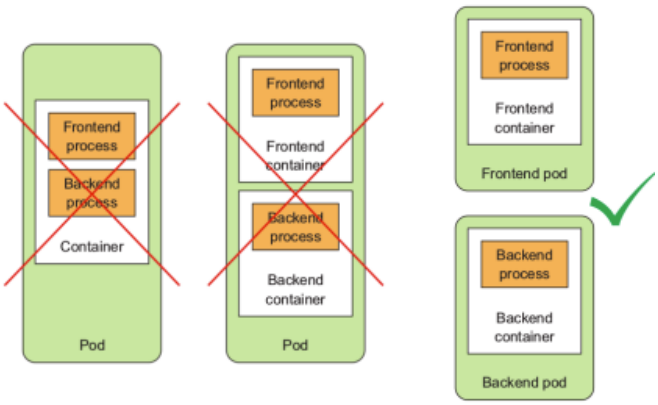
Container Migration: happens in case of failing node (or other reason). In this case Kubernetes has to take care of deploying the container on other working node, but also of forwarding the requests as the container is moved across the platform.
Usually, the platform exposes each specific service with just *one single public IP address*, regardless of the actual number of containers running it. The *Kube-proxy component takes care of connecting the containers and forwarding the traffic in a proper manner*.
Kube-proxy component is also in charge of implementing the load balancing policies forwarding the traffic in a balanced manner to ensure scalability.

Exposing Applications: pods are connected in 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 has to expose publicly to external networks, the developer has to define a service object. The service object instructs Kubernetes to associate a pod with an external IP address and port. The **service object** takes care of forwarding the requests coming to that external IP address to one of the associated pod instances.



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: 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.



Slide D07 LAB: Kubernetes installation

[refer to the slide 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), …]

Slide D08 LAB Cloud Platform Operations (hands on with Kubernetes management)

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)

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

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.

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!";
?>
```

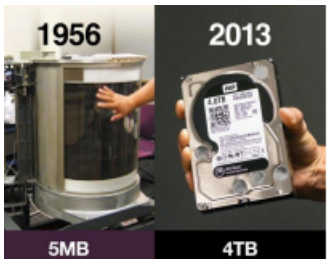
```
kubectl run --images=run-pod/v1 -it --rm load-generator --image=busybox /bin/sh
while true; do wget -q -O- http://php-apache.default.svc.cluster.local; done
```

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
```

Slide E01 Cloud Storage and Distributed File System

Storage Technology Evolution:

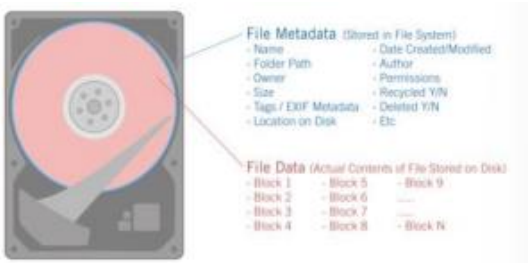


Parameter	1956	2016
Capacity	3,75MB	10TB
Average Access Time	~600ms	2,5-10 ms
Average Life Span	~2000 Hours	~22500 Hours
Price	9200$/MB	0,032$/MB
Physical Volume	1,9m³	34cm³

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

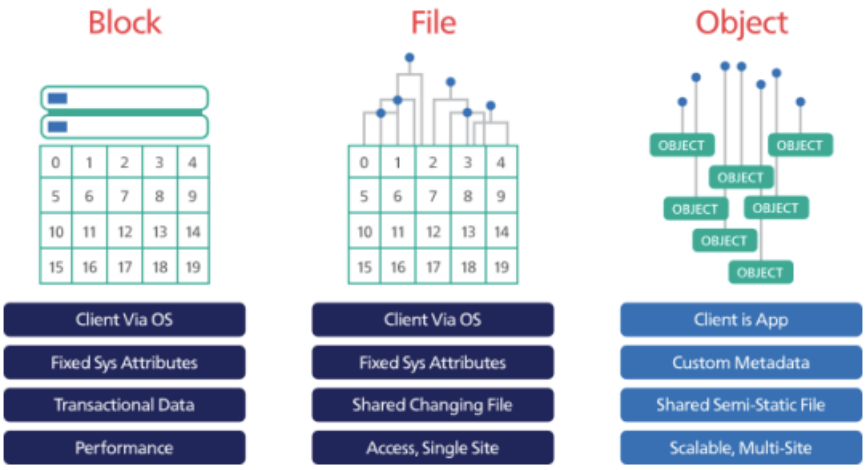
File System: organizes bocks into files and manages metadata to store files on different blocks, usually adopted by OS to store users' and system's files



Object Storage:

Alternative to block storage, new storage type. **Objects** are bundled data (i.e. a file) with the corresponding metadata.
Each object has an **unique ID**, calculated on the basis of the content and metadata. Apps 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:

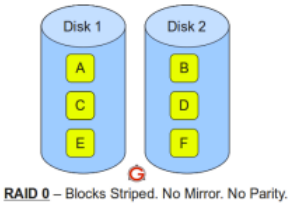
Hard drives lifespan: vary depending on it usage, about 5 years.
To prevent failures and data loss **RAID** technology is exploited.
There is a physical limitation on the maximum number of hard drives, that is given by the number of trays of the server. -> to overcome those limitations, distributed file systems are employed

RAID: Redundant Array of Independent Disks: a technology that combines multiple physical hard drives into one or more logical hard drives (virtual).
Different schemas exist: 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.

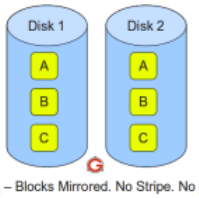
RAID Schemas:

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

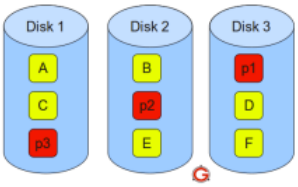
- **RAID 0**: performance RAID; blocks are stripped across physical drives
- **RAID 1**: maximum redundancy RAID; blocks are mirrored across the physical hard drives
- **RAID 5**: trade-off redundancy – performance; blocks are striped across drives, but a parity block (i.e. calculated via XOR) is added 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):

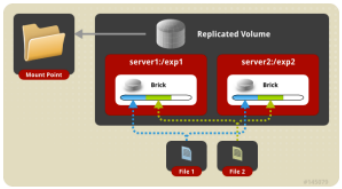
Defined to overcome the limitations of single server storage; distributes file system across many servers. Data transfer and synchronization is achieved via a local LAN. DFS can be exploited either to enlarge the capacity of a single server (by using the storage of a remote server that have) or to put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

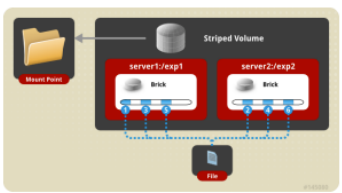
Network File System (NFS):

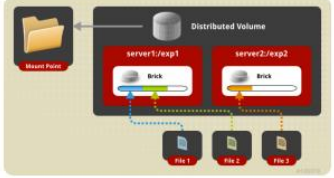
Adopts a client-server approach: a central server offers access to client to its local file system.
Defines a **synchronous** (to ensure consistency and simplify implementation) communication protocol.
The remote file system is accessed by applications running on clients in the same way of local files.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

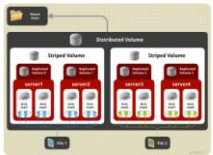
GlusterFS:

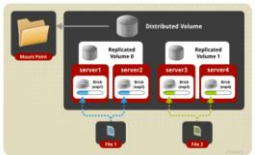
Distributed file system that pieces together storage capabilities available on each node.
Can be used locally in the same way of NFS, but here there is no distinction between clients and server. Each node can participate offering some of the local storage.
GlusterFS can be **configured** in terms of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







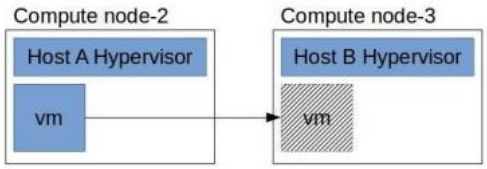




Storage for Cloud:

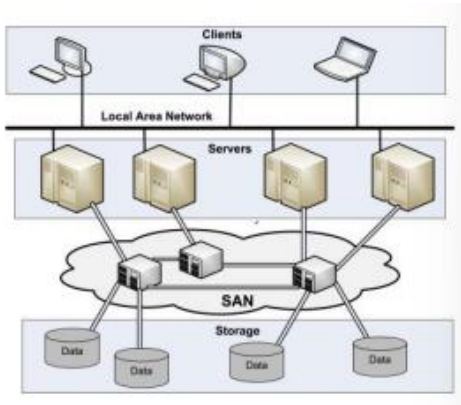
Storage requirements in cloud computing platforms are different, since the main requirement is scalability. The storage have to scale with the number of VMs, the number of volumes created, the amount of data generated by different services. (e.g. object storage Swift).
To exploit local hard drives for 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.



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.

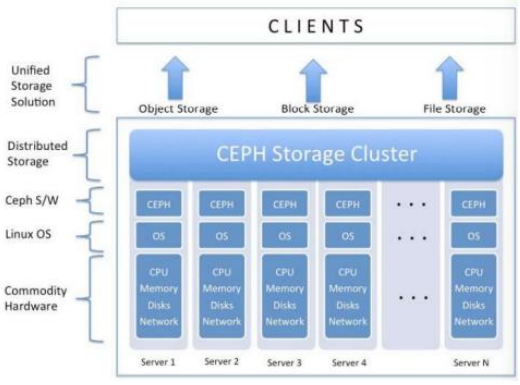
Specific protocols are adopted for the communication between servers and storage devices, i.e. iSCSI protocol, which is an IP based SCSI protocol that provides block-level access to storage devices.

Unified Storage Solutions (*alternatives to SANs*):

SAN storage devices are compex and expensive, 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: enterprise grade, robust, highly reliable storage system on commodity hardware
Main 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.

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

Ceph cluster: the regular servers cluster on which Ceph is installed.

Ceph architecture:

- **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**
- **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).

RADOS Components:
RADOS has to be distributed and is made of:

- **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:

Comprises both data and metadata that are bundled together, each couple has one unique identifier.
The identifier in unique at cluster level. There is no limit in object data size (differently from file system storage). Objects are stored by Ceph OSD in a replicated fashion. Every time Ceph receives write requests, stores data as objects. (?) OSD daemon writes data to a file in the OSD filesystem (?)

CRUSH Algorithm:

Is used both by Ceph clients and Ceph OSDs to efficiently compute information about object location.
In this way both clients and OSDs can compute the location of an object *without the need for a central lookup table*.
Compute the location of an object based on the current **cluster status**
CRUSH PROs: better data management; enables massive scale by distributing the work to all nodes in the cluster; uses intelligent data replication for resiliency

Cluster Map: represent the cluster status, can be retrieved by any MON instance, includes:

- **Monitor Map:** Current epoch, when the map was created, last time it changed
- **OSD Map**: List of pools, replica sizes, list of OSDs and their status

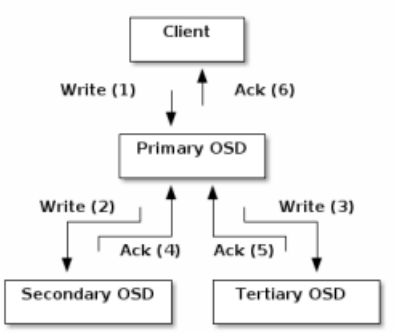
- **PG (Placement Group) Map**: details on each placement group such as PG ID, set of up nodes, set of acting nodes, state of the PG (e.g. active + clean)
- Cluster Map contains a list of storage devices, the *failure domain hierarchy* (i.e. device, host, rack, row, room, …), rules for traversing the hierarchy when storing data

Placement Group: determines the actual OSD on which an object is saved (consequently the physical hard drive, since we have one OSD for each drive)

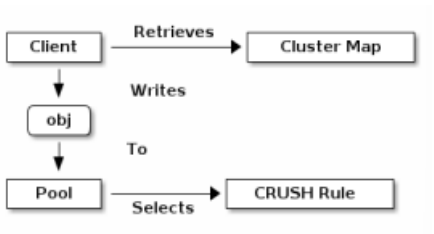
Client R/W operation:

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. 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: logical partitions for storing objects, it is a notion supported by Ceph. 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 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). Ceph allows monitors clusters. 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.

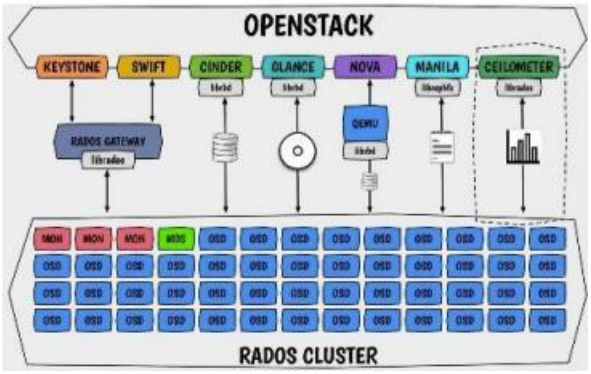
Placement Groups Computation: every pool has many placement groups, CRUSH algorithm maps PGs to OSDs dinamically depending on cluster map. The mapping is computed based on the current failure zones of the cluster, so that data can be considered safe and available even if some component fail. In addition, 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).

Prof. Tonnellotto Part

Python Crash Course

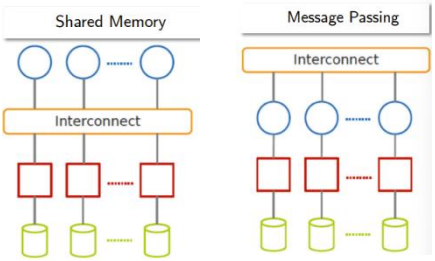
[refer to slides, an introduction to python, numpy, its .map() method]

Slide 01 Introduction

Cloud computing and *distributed programming* applications in physics, maths, science, finance… Examples of fields and quantities of data processed by many popular services and how it is increasing in the years.

Slide 02 Parallel Programming Patterns

Parallel Architectures:



- **shared memory** parallel programming model (Posix Threads, OpenMP, Compiler automatic optimizations, …)
- **message passing** programming model (Sockets, Parallel Virtual Machine (obsolete), Message Passing Interface (MPI))

Typical steps:

- **Identify what pieces** of **work can be performed concurrently.**
- **Partition** concurrent **work** onto independent processors.
- **Distribute** a program's input, output, and intermediate **data.**
- **Coordinate accesses to shared data**: avoid conflicts.
- Ensure proper order of work using **synchronization.**

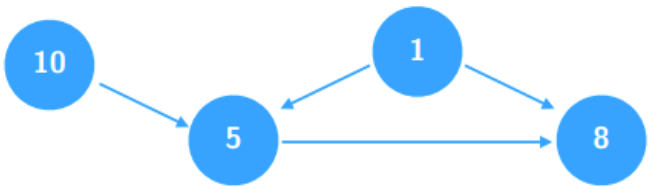
Some steps can be omitted:

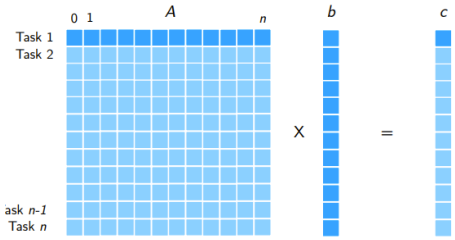
- no need to distribute data in a shared memory parallel programming model.
- In message passing parallel programming model no need to coordination on shared data (because no data shared)
- Processor partition could be done automatically.

We will study message passing model patterns. Each pattern has its own: specific structure of parallel tasks, specific techniques for partitioning / allocating data, specific structures of communication.

Task Dependency Graph *(TDG)*:

When developing a parallel algorithm the first step is to decompose the problem into task that can be executed concurrently. Many decompositions into tasks exist for one problem.
Tasks: may be of different sizes and granularities.
TDG is a Directed Acyclic Graph (DAG) in which nodes are tasks and directed edges are control dependencies among tasks. The node label is the computation weight of the task.

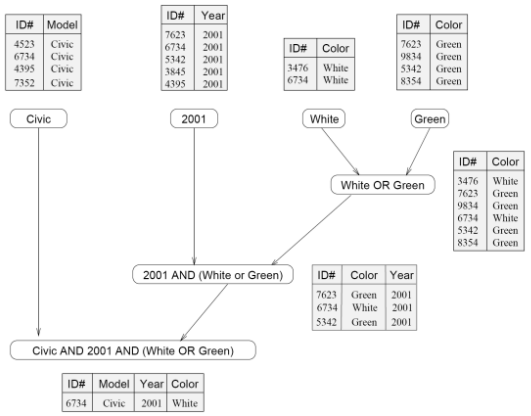


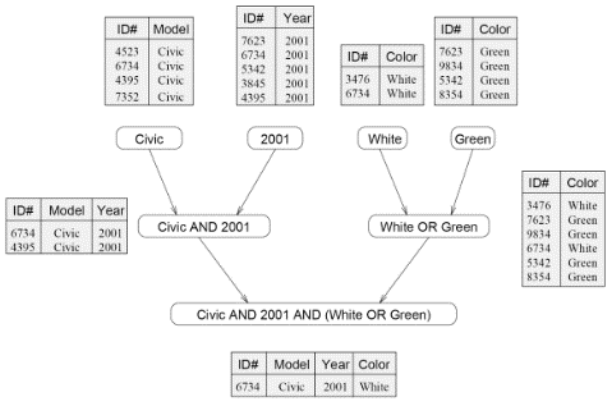


Example of program developed in concurrent way: **multiplication between a matrix (n x n) and a vector (n x 1):** we can have n tasks, all having the same computation weight, and without control dependency each other. The vector is in the shared memory among the tasks.

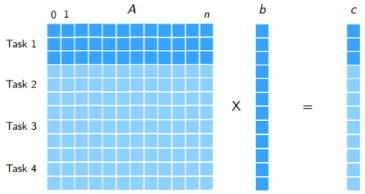
Example: Database query execution with many conditions: each WHERE condition is executed returning a result set of IDs, then the intermediate results are ANDed or ORed depending on the query. Here we have control dependencies between tasks, even though we could decide different decompositions of the problem (thanks to the additivity property of AND and OR operators).

```
SELECT * FROM T WHERE
Model = "CIVIC" AND Year = "2001" AND
(Color = "GREEN" OR Color = "WHITE")
```





Granularity of tasks: the number of tasks in which a problem is decomposed determines its granularity.
Large number of tasks for a problem: **fine-grained decomposition**.
Small number of tasks for a problem: **coarse-grained decomposition**.



Degree of concurrency: the number of tasks that can be executed in parallel
Maximum degree of concurrency: the maximum number of tasks that can be executed in parallel during execution of a program
Average degree of concurrency: average number of tasks that can be processed in parallel over the execution of the program.

The parallel system is used efficiently when the average degree of concurrency is similar to the maximum degree of concurrency.

The degree of concurrency increases when the granularity becomes fine-grained.

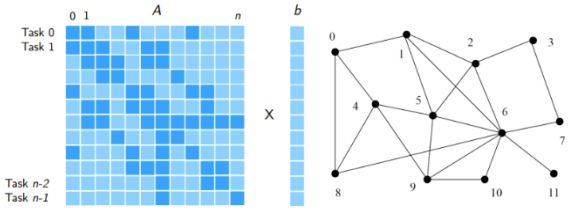
Every **directed path** in TDG (Task Dependency Graph) represents a sequence of tasks that must be processed one after the other.
The directed path in TDG **can be measured** either on the **number of tasks** or as the **sum of the weights** of the involved tasks.

Critical Path Length: the length of the longest path in a task dependency graph. Corresponds to the minimum execution time of a parallel program.

Parallel Performance Limitations: by increasing the number of tasks of the decomposition (going to fine-granularity decomposition) we can reduce the parallel time, but there *are limitations depending on the problem*. **Inherent bounds** on granularity of the computation. i.e.: in matrix – vector multiplication it relies on the number of cells of the matrix (no more than n^2 tasks).
Increasing concurrency we also **increase the amount of data that has to be exchanged** with other tasks. It results in a **communication overhead** which increases along with the increase of the parallelism degree.

Subtasks usually have to exchange data each other in order to perform the work in a decomposition.
Task Interaction Graph (TIG): The graph of tasks as nodes and data exchange (aka interactions) as edges is

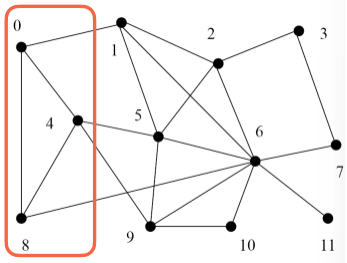
referred to as Task Interaction Graph (TIG). In TIG *edges are usually undirected*, edges have a label that is the amount of data exchanged.



i.e. in the matrix vector multiplication in case of a sparse matrix, where only non zero elements of A participate in the computation. Lets assume that we have n tasks (and so n nodes), and that the nth node has the nth element of vector b. Each node has to communicate with every node that has one of the elements of b array corresponding to an index for which its matrix cell is not empty. (i.e. row 0 has nonzero values in 0,1,4,8 and so the node 0 has to exchange messages with nodes 1,4,8).

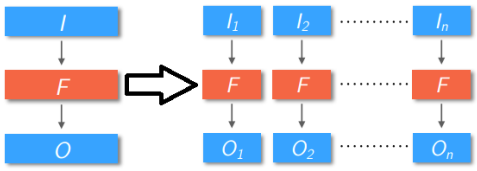
TIG and granularity: if we make a finer decomposition in terms of granularity, we introduce more overhead, on the other hand if we go to a more coarse-grained decomposition, the communication **overhead** decreases.
Computation-to-communication ratio: the ratio between the time spent by a node for computation and the time spent by that node for communication.

i.e. in matrix-vector multiplication, node 0 takes 1 time unit for computation and 3 times units for communication, the ratio is 1/3 (assuming that each computation and each communication takes 1 unit of time). If we increase the granularity (by considering tasks 0,4 and 8 as one single task), the ratio becomes more favourable: ¾.

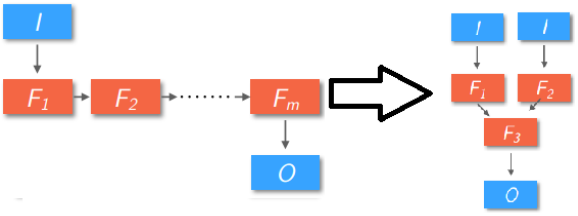


To obtain parallel tasks we can decompose either on data or on control.

Data parallelism: perform the same operation on different data items in parallel, parallelism is given by domain decomposition and grows with data size. Usually very fine-grained, facilitates very high speedups. Function F applied in parallel to many partitions of input data I. I = {I1, I2, …, In}



Task parallelism: perform different computations (tasks) at the same time. Parallelism comes from a control / **functional decomposition**. We partition on control. A function F is decomposed in the composition of m Functions (F(I) = Fm(Fm-1(…F1(I)))) (pipelined functions). From a pipeline of functions we reach a task graph where O = F(I) = F3(F2(I), F1(I)) (shorter pipeline, some functions are executed in parallel).

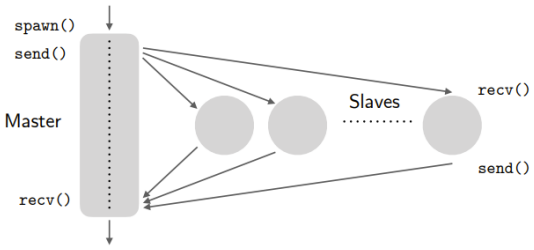


We can also obtain parallelization by combining task and data parallelism.

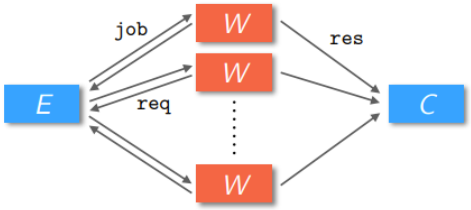
When we have a large set of completely independent sub-tasks those problems are usually solved with data parallelization, in those problem TDG are completely unconnected.

Master-Slave implementation of data parallelization with completely independent sub-tasks:
CONS: does not perform well when assigned tasks are unbalanced.

- Perform static partitioning on input data
- N slaves are created dynamically by the master (spawn())
- A partition of the input is assigned to each slave



Farm Pattern:



- Farm pattern is made of 3 logical entities:
  Emitters: are in charge of creating jobs *statically or dynamically*
  Workers: ask for jobs to execute in two possible fashions:
    - **On-demand**: requests are issued when previous jobs are completed
    - **Pre-fetching**: once the current job has been received, asks for the next one
- **Collectors**: that gather the results of the computation from the workers

Optimal number of workers:

- **n**: number of workers
- **Twork**: time to complete a task
- **Tcomm**: time spent for communication (send + receive results)
- **Bcomm**: communication bandwidth: Bcomm = 1 / Tcomm
- **Bcomp**: computation bandwidth: Bcomp = n / Twork

The optimal number of workers n is obtained when Bcomm == Bcomp. → 1 / Tcomm == n / Twork → **n == Twork / Tcomm**.

We could *use pre-fetching to overlap communication and computation*, and we can also use a single master to both emit and collect results and interpret a result as a request for the next job.

Divide and Conquer Pattern for Parallel Programming modelling:

Same properties than in sequential programming divide et impera:

- Decompose in smaller independent problems
- Use recursion

→ parallelism comes naturally from task independence

i.e. of divide et impera where **decomposition is decided dynamically at run-time**: problem: find the minimum in a list → we can split the list in many partitions depending on its size, aiming to optimize the number of partitions. Dynamic partitioning is useful especially when it is difficult to predict the cost of each task.

ISSUES: initial and final stages have small parallel degree, sub-tasks may not be balanced.

Data decomposition on output:

- Each processor produces a partition of the output → we can infer computation assignment from that
- Input data is partitioned accordingly, even though often input data is partially / totally replicated
- **Owner computes rule**: the owner of a partition of data / the assignee to a partition of output data is in charge of computing that

Problems in output partitioning:

Suppose O = {O1,O2, …, On} → in good cases we have O1 = f(I1), O2 = f(I2), …, but it *could happen that each partition of the output is a function of the whole input*
There are also cases in which the **outputs partitions have to be aggregated** by a function to obtain the final ouput. i.e. O = aggregate(O1,O2, …, On)

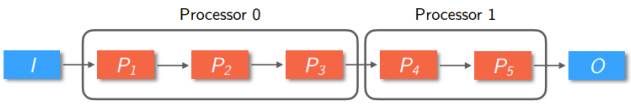
Pipeline Pattern:

In the pipeline pattern computation is partitioned into stages that are executed sequentially. Asymptotically a pipeline can achieve a speed-up equal to the number of stages.

Pipelining is **efficient when** we have:

- Independent instances of the same problem (i.e. different data, same algorithm → job parallelism (?)) data may form the input stream for the pipeline, the algorithm can be decomposed in a cascading sequence of tasks
- data parallelism: we can decompose in a sequence of tasks the algorithm processing a partition of data
- pipeline without stream: we can decompose the job in a sequence of tasks generating / working on partial results; here every stage **early feeds** the next one

Pipeline pattern issues:
The throughput of the pipeline is limited by the slowest stage → its next stages have to wait for its results (idle), the previous stage have to buffer its outputs (there could be buffering problems).
→Possible solution: **introduce a coarser grain**: assign many fast tasks or few slow tasks to a processor, even though this decreases the parallelism degree

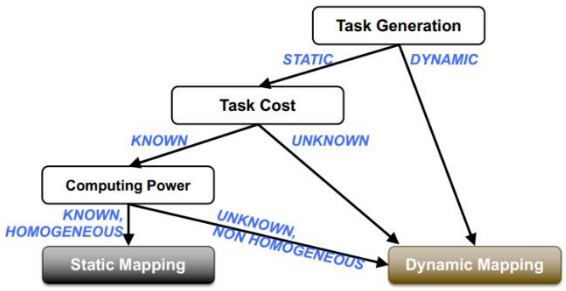


--

How to allocate tasks to processors:

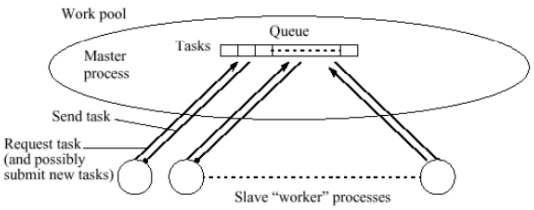
Aims of a parallel algorithm are to **reduce communications** and **balance process workload**.
Two approaches: static allocation or dynamic allocation.

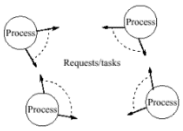
Static allocation can be exploited when interactions are not so regular, and we could group tasks to balance the load and map tasks to specific nodes in order to minimize communications.
On the other hand when tasks do not have uniform costs, have unknown dependencies and are created at run-time, both scheduling and mapping must be **dynamic**.



Centralised Load Balancing:

In centralised load balancing slaves can send new tasks (to the master), the mater has to maintain a dynamic queue of available tasks. The slave processes have to request for new tasks to process.





Distributed Load Balancing:

The aim of distributed load balancing is to remove centralisation and to favour data exchange among neighbours nodes. Two ways:

- **Push / sender initiated**: it is the worker that generates a new task that sends it to another worker
- **Pull / Receiver initiated**: when a worker is idle, it asks to other workers for a job to execute

In distributed load balancing the partner for pushing / pulling new tasks can be selected in three ways:

- **Random**
- **Global Round Robin (GRR)**: a global marker (shared among the nodes) points to the next worker; a worker that needs a partner reads the global variable and then increments it
- **Local Round Robin (LRR)**: every node has a private (un-synchronized) pointer to the next worker → PROs: no overhead due to sharing a global variable; approximate a GRR

Slide 03 Map Reduce

Programmer problems in splitting code: • How to divide code into parallel tasks? • How to distribute the code? • How to coordinate the execution? • How to load the data? • How to store the data? • What if more tasks than CPUs? • What if a CPU crashes? • What if a CPU is taking too long? • What if the CPUs are different? • What if we have a new (serial) code?

Programmer problems in splitting data: • How to split the data? • How to distribute the data? • How to collect the data? • How to merge the data? • How to coordinate the access to the data? • What if more data splits than tasks? • What if tasks need to share data splits? • What if a data split becomes unavailable? • What if we have a new input? • What if the data is big?

Typical Big Data Applications: iterate over a large number of records; extract something from each; shuffle and sort intermediate results; aggregate intermediate results; generate final output

To **scale out** (not up) → no need of supercomputers, low cost commodity hardware **CONS**: many failures
Move processing to where data is → same code that runs everywhere, less data over network but **code must be portable**
Process sequentially, no random access →write once and read many, works on huge files **CONS: poor support of standard file APIs**
Right level of abstraction → hide implementation details from app dev, few lines of code **CONS: everything must fit into the abstraction**

Object Oriented Programming: drawbacks:

- **Testability**: requires lots of mocking, extensive environmental setup, hard to maintain evolving objects
- **Complexity**: over-designing, re-use complicated and often requires refactoring, often objects do not represent the problem correctly
- **Concurrency**: objects live in shared environment, many objects in shared state environment handle concurrency poorly

---

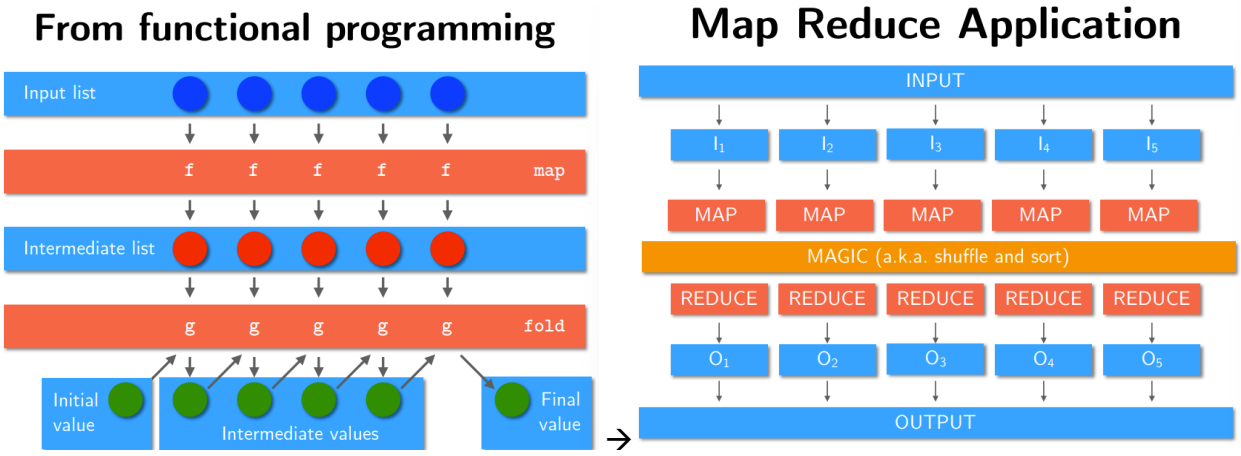
Functional Programming:

Functional programming is a mathematical approach to solving problems, is **simpler** and **less abstract**, **easy** to **reuse**, to **test** and to handle **concurrency**.
Functional programming principles are:

- **Purity**: pure functions act on their parameters only. Are not effective if not returning anything. Will always produce the same output for the given parameters (**idempotency**). Pure functions do not have **side effects**.
- **Immutability**: no variables in functional programming, all variables must be considered constants. We can mutate variables only for short living local variables, loop flow structures, state objects.
- **Higher Order functions**: in functional programming a function is a first-class object of the language. Higher order functions are also called **closures** or **anonymous functions**. A closure is a variable that stores a **function** and an **environment** (that keeps track of the variables it may refer). With a closure it is possible to access its scope even once it has been executed.
  A functional programming language supports:
  - **Construction** of new functions at run time
  - **Storing** functions in a variable
  - **Passing** functions as arguments to other functions

- **Returning** functions as values of other functions

- **Composition**: to apply a function to the result of another to obtain a third function *h = f(g(x))*
- **Currying**: process of turning a function with multiple arity in a function with less arity.
  Arity: the number of arguments a function takes.
  Example of currying: A function that takes two arguments, one from X and one from Y, and produces outputs in Z, by currying is translated into a function that takes a single argument from X and produces as outputs functions from Y to Z.



Map Reduce programming model:

Map function: from [key, value] (always 1 row) to [key, value] (0 or more rows)
receives as input one key, value pair. Produces as output a list of key-value pairs (usually 1 or more per input). It is invoked by the Mapper (function)

Reduce function: from [key, list of values] (always 1 row) to [key, value] (0 or more rows)
receives as input a [key, list of values] pair, produces as output a list of [key, value] pairs (usually 0 or 1 per input). It is invoked by the Reducer function.

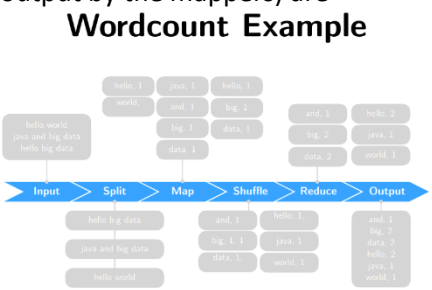
Both Map and Reduce functions are **STATELESS**.

Mappers:

Mapper should run on nodes which hold their portion of data locally (to avoid network traffic).
Many mappers run in parallel, the mappers read the [key, value] pair from HDFS.
Mapper can use or ignore the key of input.
Example: mapper that read a line of the file at a time -> the key is the byte offset of that line, the value is the line → typically the key is irrelevant.
Mapper output (if any) must be in the form of [key, value] pair.

Reducers:

There could be one single reducer for a job, but sometimes more than one.
All values associated with a particular **intermediate key** (the one given in output by the mappers) are granted to go to the **same reducer**.
Intermediate keys and value lists are passed to the reducer in **sorted key order**. (Shuffle and sort step).
The reducer outputs zero or more final key / value pairs, that are written to HDFS. Usually the reducer emits one single [key, value] pair for each input key.

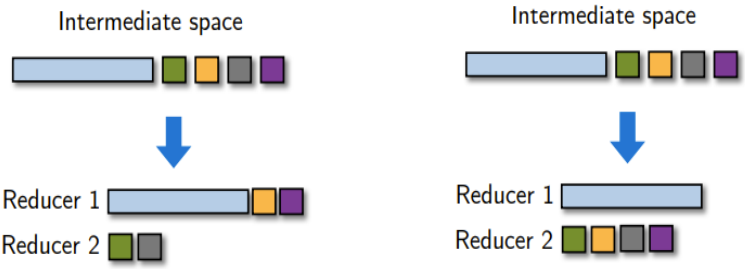


[in the slides there is the WordCount example in pseudocode]

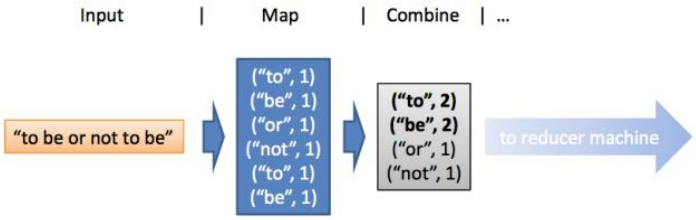
Optimizations in Map Reduce: the most important **bottleneck** is data exchange between the map and the reduce phases, because it is an all-to-all communication and depends on the intermediate data, which depends on the application code and input data.

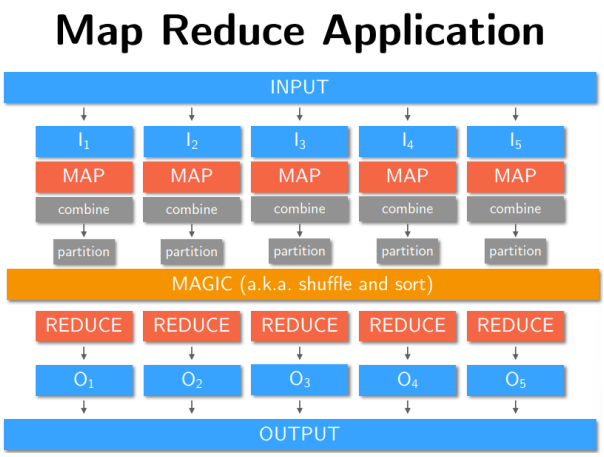
Knowing how many map function invocations per machine, how many reduce function invocations per machine it is possible to optimize the runtime by looking at the implementation details (→ this means **breaking** functional programming paradigm!)

Partitioners:
Partitioners balance key assignments to reducers. By default intermediate keys are hashed to reducers. Partitioners instead can specify the node to which an intermediate key-value pair must be copied. Partitioners are in charge of dividing the space of the keys in order to parallel reduce operations as much as possible. Partitioners only consider the keys, not the value.



Combiners:
Combiners are in charge of performing local aggregation at node level before the shuffle. Typically a combiner is a local copy of the reducer function. Aims at reducing network traffic and parallelizing reduce operations.





Map Reduce frameworks history:

2002: Apache Nutch → project for building a search engine capable of indexing 1 billion pages
2003: Google File System : a distributed file system for large data sets
2004: Googe Distributed Platform: MapReduce
2007: Yahoo new project Hadoop (open source implementation of Google MapReduce software, open source implementation of Google File Storage, called HDFS)
2008: Yahoo releases Hadoop as an open source project to Apache software foundation.

Slide 04 Hadoop

Hadoop has three working modes:

- **Standalone (aka local) mode**: no daemons, uses local file system as a substitute to HDFS; jobs will run as if there is 1 mapper and 1 reducer.
- **Pseudo-distributed mode**: many daemons on one machine (mimics the behaviour of a cluster); all daemons use HDFS protocol; there could be many mappers and many reducers (by default 1 reducer).
- **Fully-distributed mode**: on real cluster; daemons run on cluster's machines using HDFS protocol; many mapper and many reducers (by default 1 reducer).

Debugging and execution must be performed on Hadoop without the IDE. Have to deploy jar file to hdfs in order to run it.

MapReduce job: consists of input data; MapReduce program; configuration information.

Hadoop runs a job by dividing it into **tasks** → two types of tasks: map tasks and reduce tasks.

YARN (Yet Another Resource Negotiation): schedules tasks to the nodes of the cluster.
If a task fails, it will be automatically rescheduled on a different node.

Hadoop divides the input of a MapReduce job into **fixed-size pieces** called **input splits**.
Hadoop creates one map task for each split, which runs the user-defined map function for each record of the split.

[Hadoop code example on the slides]

Hadoop Types: Combiner: if used, it must be in the same form as the reduce function, because the combiner invocation is optional for Hadoop framework. So the combiner can only be used to reduce the execution time, but not to generate another type of intermediate data. Combiner is an implementation of the Reducer, its output types must be the intermediate key and value types.
Often the combiner and reduce functions are the same (in that case intermediate type and output type of the job must be the same).

Partition function: operates on intermediate pairs, returns the partition index. Usually exploits just the key of the tuple.

[Hadoop java types for setting input / output types and map, combine, … functions on the slides]

Note: due to Java type erasure, it is possible to configure an Hadoop job with incompatible types, because the configuration is not checked at compile time.

The default input format is TextInputFormat; The default mapper is the Mapper class, which writes the input key and value unchanged to the output; The default reducer is the Reducer class, which simply writes all its input to its output; The default partitioner is HashPartitioner, which hashes an intermediate key to determine which partition the key belongs in.

The number of partitions is equal to the number of reduce tasks for the job. By default one single reducer → just one partition.

The number of map tasks is equal to the number of splits → that depends on size of input and on file block's size (if in HDFS).

Serialization in Hadoop:

Serialization is the process of turning structured objects into a byte stream (for transmission over a network (inter-process communication) or for writing to persistent storage). (Deserialization is the reverse process: from byte stream to a series of structured objects).

In Hadoop inter-process communication between nodes is implemented using remote procedure calls (RPCs).

Hadoop serialization format is called Writables. (compact, not easy to extend or use on other languages). Other serialization frameworks supported are Avro, Thrift, Protobuffers.

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1–5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1–9
double	DoubleWritable	8

[in the slides example for IntWritable]

Hadoop input:

Input split is a chunk of the input that is processed by a single map task.
Each split is divided into records.
Both splits and records are logical → not required to be files (commonly they are that).

In Java class InputSplit, the object has a length in bytes and a set of storage locations (i.e. hostname strings). That object does not contain the input data, is a reference to the data.

- ➔ Storage locations are used by Hadoop to place map tasks as close as possible to split's data.
- ➔ **Size** of the splits is exploited in order to let the **bigger splits be processed first** (in order to minimize job runtime)

When a new job is created, the client running the jobs calculates the splits for the job (by getSplits()), then sends them to the Application Master, which schedules the jobs on the nodes of the cluster according to the splits locations.
Each map task (one for each node assignee) passes the split to the method createRecordReader() that returns a RecordReader for that split. The RecordReader is little more than an iterator over records, that are passed to the map function.

Setup and Cleanup Methods:

Usually Mapper and Reducer tasks have to execute some code before the invocation of their map() or reduce() methods. This setup phase consists of: initializing data structures; reading data from external files; setting parameters. The method public void setup(Context context) absolves this task. The cleanup() method does the same.

```
public class MyDriverClass
{
    public int main(String[] args) throws Exception
    {
        int value = 42;
        Configuration conf = new Configuration();
        conf.setInt ("paramname", value);
        Job job = new Job(conf);
        // ...
        return job.waitForCompletion(true);
    }
}


public class MyMapper extends Mapper
{
    public void setup(Context context)
    {
        Configuration conf = context.getConfiguration();
        int myParam = conf.getInt("paramname", 0);
        // ...
    }

    public void map...
}
```

Hadoop Tricks:

- **Limit memory footprint**: avoid storing reducer values in local lists; use static final objects; reuse Writable objects → avoid the saturation of the Garbage Collector of the nodes

- **Use a single reducer when possible**: Object fields are shared among reduce() method invocations; remind to make deep copies of value objects when needed because the framework reuses them
- You **cannot pass parameters through class statics**: use configuration parameters or use external data sources (cache services, HDFS, …)
- **Iterable object is not equal to a List object**: in Iterable the number of elements is not known; the iterable returns a reference to the current object; that current object can change state, perform deep copies when needed; be careful in allocating Lists to store iterables for the memory consumption

Slide 05 Hadoop Distributed File System (HDFS) and YARN

Hadoop Distributed File System (HDFS) features:

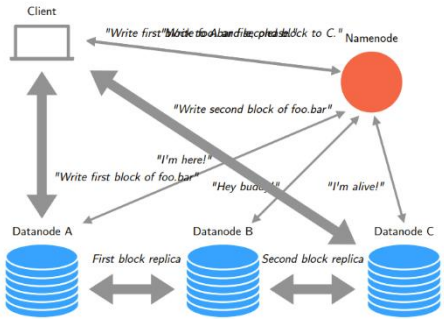
- **Highly fault tolerant**: failures occur frequently.
- **High throughput**: HDFS can consist of thousands of machines, each storing part of the file system.
- Suitable for applications with large data sets: HDFS fits well for huge files, time to read the whole file is optimized, while **does not fit well for**:
    - Low latency data access
    - Lots of small files
    - Multiple writers, arbitrary file modifications
- HDFS can be built on *commodity hardware*.
- HDFS grants *streaming access to file system data*.

HDFS files are divided into **chunks** (aka **blocks**) made of 64 / 128 megabytes.
Blocks are **replicated** in many computer nodes located on different racks (usually 3 or more).
→ Block size and degree of replication can be **set by the user.**

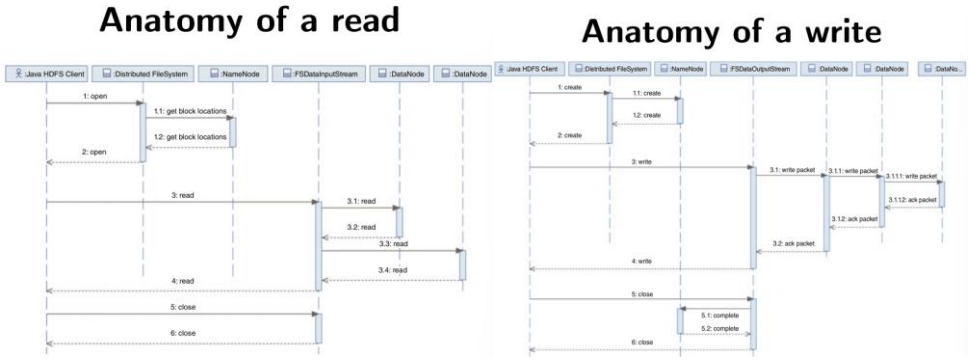
Normal file system blocks are few KBs, while disks blocks are usually 512 bytes. In HDFS blocks are 64 MB, but smaller files do not occupy the whole HDFS block. Those huge blocks minimize seek costs, transfer of blocks happens at disk transfer rate. Blocks abstraction allows to: have files larger than blocks; no need to store them on the same disk; simplify storage subsystem; fit to replica mechanism; copies are read transparently to the client.

Master node (aka Name Node): is a special file that stores for each file the ***positions of its blocks***. Master node *is often replicated* to other nodes in active stand-by (so called *shadow name nodes*).

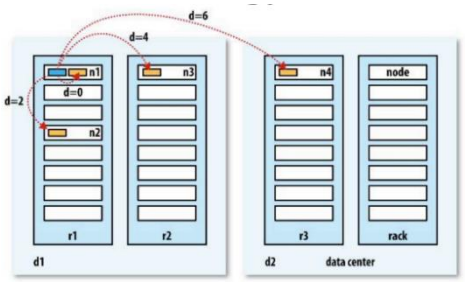
Directory (or tree) of the file system: knows where to find the master node; can be replicated; all participants to the DFS know where the directory copies are.



HDFS adopts a **Master – Slave architecture** where there is a single **name node** that manages FS namespace and regulates accesses.
The **Name Node** handles metadata, directory structure, file-to-blocks mapping, blocks' locations, access permissions.
Many Data Nodes (usually one for each node in a cluster) that store sets of blocks and manage the storage attached to their nodes. Data Nodes serve read and / or write requests, perform block creation, deletion and replication instructions from the name node.



When a new file is created from a node inside the HDFS cluster, the first replica is stored directly inside that node, otherwise the first replica node is chosen randomly.
The second replica is stored on a different rack from the first replica.
The third replica is usually on the same rack as the second, in a different node chosen randomly.
Further replicas are placed random in nodes of the cluster.
(System avoids to place many replicas on the same rack / node).



Rack topology: describes how the nodes are grouped in terms of racks.

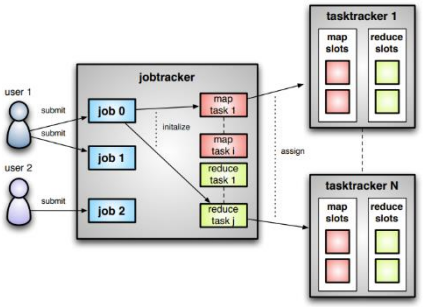
Hadoop Distributed Resource Management (adopted in Hadoop 1.0):

Definitions and components of Hadoop 1.0:

Job: unit of work that the client wants to be performed.

Task: unit of work that Hadoop schedules on a node of the cluster (map or reduce task).

Slot: processing element for tasks



Job Tracker: accepts jobs submitted by users; creates tasks; assigns map and reduce tasks to Task Trackers; monitors tasks and Task Trackers status; re-executes tasks in case of failures.

Task Tracker: runs map and reduce tasks following instructions from Job Tracker; manages the storage and transmission of intermediate output.

Limitations of Hadoop 1.0:

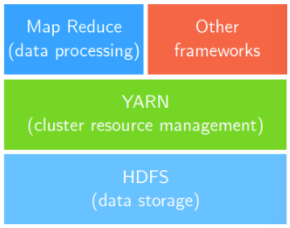
scalability has a ***bottleneck*** caused by the ***single Job Tracker*** (maximum 4000 nodes and 40000 concurrent tasks)

availability Is limited because the Job Tracker is a single point of failure, and every failure kills all queued and running jobs; jobs need to be resubmitted by users.

Resource utilization: since the *number of map and reduce slots on each Task Tracker are predefined* and limited, utilization issues can occur (when the assigned tasks are too fast or too slow)

Running non-Hadoop applications is problematic, in particular for real time analysis and running ad-hoc queries since Hadoop is **batch-driven**.

YARN is a cluster resource manger positioned between the HDFS and the Map Reduce data processing jobs of Hadoop. It was implemented since Hadoop version 2.0. YARN introduces many benefits, like:



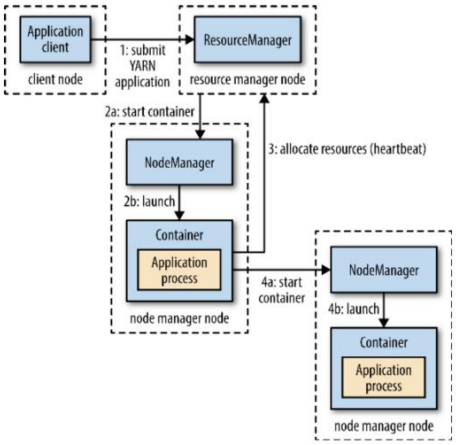
- **Scalability**: more than 10.000 nodes, 100000 concurrent jobs
- **Compatibility**: apps for Hadoop 1.0 run on YARN
- **Resource Utilization**: dynamic allocation of resources; improved use of resources.
- **Multi-tenancy**: can use both open source and proprietary data access engines; can perform real-time analysis and execute ad-hoc queries. Used in many distributed framework (i.e. Spark)

YARN Components:

Two types of long running daemons:

- **Resource manager**: one per cluster, manages the use of resources among the cluster.
- **Node manager**: one per node in the cluster, launch and monitors **containers**.

Container: executes an application specific process with a constrained set of resources (~~aka application master~~)



Resource request: is made for a set of containers, can express the amount of resources required for each container (memory and CPU) and locality constraints for the containers in that request. If the locality constraint cannot be met either the allocation is aborted, or the constraint is loosened.

YARN does not provide a communication mechanism for the parts of the application.

YARN applications lifespan depends on app type, exist three types:

- First: **one application for each user job** (i.e. Hadoop MapReduce job)
- Second: **one YARN application per each workflow or user session** made of many jobs (that could be even unrelated) (i.e. Spark jobs). →PRO: containers can be reduced between jobs (more efficient), possibility to cache intermediate results
- Third: **long running application** shared between users → this kind of app has a coordination role, i.e. an app master for launching apps in the cluster, the always on app master means that users have very low latency

YARN scheduling:

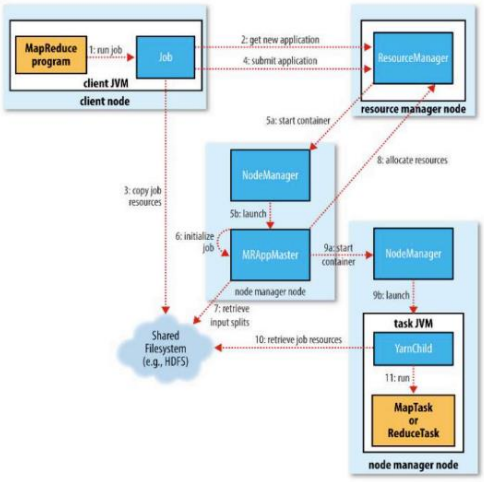
Scheduling is a NP-hard problem, there is no absolute best policy.

YARN provides a choice of configurable schedulers:

- **FIFO Scheduler**: apps placed in a single queue, are run in order of submission (first in, first out)
- **Capacity Scheduler**: has separate dedicated queues, each of them can use a given fraction of the cluster capacity. Within each queue, apps are scheduled using FIFO policy.
- **Fair Scheduler**: balances resources between all running jobs. (there is no need to set amount of capacity)

➔ **Delay Scheduling**: supported both by Capacity and Fair Scheduler → waiting a short time (few secs) can increase the possibility of allocating a container on the requested node (increases the efficiency of the cluster)

Every Node Manager in a YARN cluster periodically sends a heartbeat to the resource manager.
Heartbeats: carry information on the containers running in the node and on the amount of resources available on the node. Each heartbeat is a potential scheduling opportunity for an application to be run in a container.



Fault Tolerance:

Four macro-types of failures:

- Task level failure: can be of 3 types:
  - **user-defined code throws a runtime exception**: In this case the task JVM reports the error to its parent application master, then exits. The app master marks the task attempt as failed, then frees up the container.
  - **Sudden exit of the task's JVM**: node manager informs the application master, that marks the attempt as failed and frees up the container
  - **Hanging tasks**: happens when the application master notices that it has not received updates from a task attempt for a while, in this case the task JVM process is killed automatically and the app master marks the task attempt as failed
  - ➔ Application master will try to **reschedule failed tasks**, possibly on a node manager where it has not previously failed
- Application Master failure:
  - In this case the resource manager will detect the failure and start a new instance of the master on another container (on a node manager). The client must ask for the new app master address (this process is transparent to the user). If this happens twice for a MapReduce application, this will be not tried again, and the job will be considered failed.
- Node Manager failure:
  - Resource Manager will consider a node manager failed if **it has not sent heartbeat for at least 10 minutes**; in this case the Resource Manager will remove it from the pool of nodes for containers scheduling. Any task that was running on the failed container will be considered failed. If a node manager **fails many times**, it will be **blacklisted** by the application master.
- Resource Manager failure:
  - By default, the resource manager is a **single point of failure**.
    In order to grant high availability it is necessary to run a pair of resource managers in an **active-standby** configuration. *Info on all running apps is stored in an High Available state store* so that the standby Resource Manager can recover the apps states in case of active RM failure. That transition is handled by a **failover controller** (the default one uses leader election to ensure that there is only one active RM at a time).

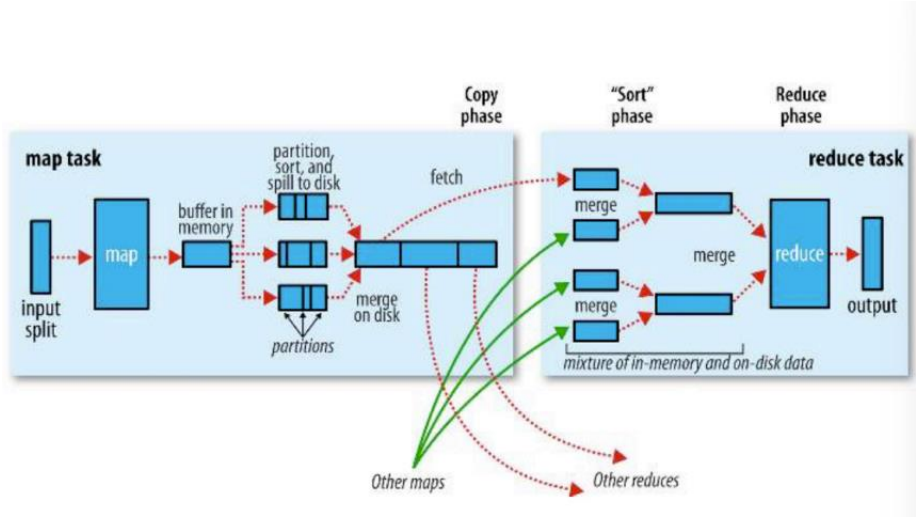
Speculative Execution in YARN:

In many clusters there are **stragglers** (i.e. slow workers), that lengthen the completion time of jobs. Stragglers can be of many types:

- Other jobs that are consuming resources on the machine of a container of the job
- Disks with soft errors that transfer data slowly
- Weird things (like CPU cache disabled at machine init)

The *solution* to reduce completion time of jobs adopted by YARN is to **schedule backup copies of the remaining in-progress tasks when the job is near to completion**. In this way the task that completes first (the original, or the backup copy) "wins". The drawback is a (small) additional resource cost, the advantage is that completion time can be reduced significantly (i.e. a sorting program without backup is 44% longer).

How Shuffle and Sort is executed (a mixture of in-memory and on-disk data in some phases):



Slide 06: Map Reduce Design Patterns

MapReduce is a framework, you have to adapt it to your application needs.
The app algorithm must be broke into filter / aggregate steps. (filtering is in Map function, aggregating is part of Reduce). Sometimes many MapReduce stages are needed. The MapReduce framework is not the solution for any distributed algorithm.
MapReduce makes sense when:

- **files** are very **large and rarely updated** and
- we need to **iterate** over all the files **to generate interesting property** out of the data

There exist *many design patterns for MapReduce*:

- Intermediate data reduction (e.g. InMapperCombiner)
- Matrix generation and multiplication
- Selection and filtering (Relational Algebra operations)
- Joining (Relational Algebra operations)
- Graph Algorithms

In-Mapper Combiner (Intermediate Data Reduction):

Intermediate Data is written locally, then transferred over the network to reducers. It arises a performance **bottleneck**. To reduce this issue we can reduce the amount of intermediate data by using Combiners (as already explained) or **In-Mapper Combiners** too.

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

PROs: total control over aggregation (how and when), that is local; guaranteed to execute (Combiners are not); efficiency control on creation of intermediate data and structures; avoid creation of unnecessary intermediate objects

CONs: breaks the functional programming paradigm (state is introduced); there could be potential ordering-dependent bugs; memory scalability bottleneck (can be solved with memory foot-printing and flushing (?))

Matrix Generation:

When we have as input N records (e.g. N documents) and we have to emit N*N outputs (e.g. for each document we have to emit N records, one for each bigram (from a predefined set), each record has to indicate the frequency of that bigram in that document).
Two solutions to this problem either emit:

- **Pairs**: generate O(N^2) data in O(1) space (e.g. the key of the emitted tuple is docID_bigramID, while the value is an integer counting the number of occurrences of that bigram in that doc)

```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            for all term u ∈ Neighbors(w) do
                Emit(pair (w, u), count 1)          ▷ Emit count for each co-occurrence
class Reducer
    method Reduce(pair p, counts [c1, c2, ...])
        s ← 0
        for all count c ∈ counts [c1, c2, ...] do
            s ← s + c                               ▷ Sum co-occurrence counts
        Emit(pair p, count s)
```

- **Stripes**: generate O(N) data in O(N) space (e.g. the key of an emitted record is the termID, while *the value is a list of tuples*, each mode of the documentID as the key and the number of occurrences in that doc of that bigram as value)

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                         ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

Selection in relation algebra: the Map function has to iterate over records and check if the condition on the current tuple t is satisfied (c(t) == true). A pair (t, ⊥) is output. The Reduce function receives a tuple in the form (t, ⊥,⊥,⊥,…⊥) as input, and outputs (t, ⊥). (? What does ⊥ means?)

Projection in relation algebra: Map steps creates a new tuple t' for each tuple record t, containing only the projected attributes, then outputs a (t', ⊥) pair. The Reduce function receives a tuple in the form (t', ⊥,⊥,⊥,…⊥) as input, and outputs (t', ⊥). (? What does ⊥ means?)

Union:
Map: for each tuple t in R, output a (t, ⊥) pair
Reduce: for each input key t, there will be 1 or 2 values equal to ⊥. Coalesce them in a single output (t, ⊥)

Intersection:
Map: for each tuple t in R, output a (t, ⊥) pair
Reduce: for each input key t, there will be 1 or 2 values equal to ⊥. If there are 2 value, coalesce them in a single output (t, ⊥) otherwise do nothing

Difference:
Map: for each tuple t in R, output (t , R) and for each tuple t in S, output (t , S)
Reduce: for each input key t, there will be 1 or 2 values. If there is 1 value equal to (t , R) output (t, ⊥), otherwise do nothing

Natural Join: (For simplicity, assume we have two relations R(A,B) and S(B,C). Find tuples that agree on the B attribute values and output them)
Map: for each tuple (a, b) from R, output (b, (R, a)) and for each tuple (b, c) from S, produce (b, (S, c))
Reduce: For each input key b, there will a list of values of the form (R, a) or (S, c). Construct all pairs and output them together with b.

Grouping and Aggregration: (For simplicity, assume we have the relation R(A,B,C) and we want to group-by A and aggregate on B, disregarding C.)
Map: for each tuple (a, b, c) from R, output (a, b). Each key a represents a group.
Reduce: apply the aggregation operator to the list of b values associated with group keyed by a, producing x. Then output (a, x)

Stage Chaining:

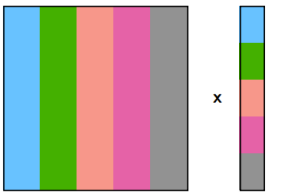
In complex applications it is useful to break them down into stages, where the output of one stage is the input for the next stage. Those intermediate outputs could be useful for reuse too.
Intermediate records can be stored in data store, forming a materialized view.
→ early stages are usually the heaviest amount of data access, so this can save a lot of work

---

Matrix – Vector Multiplication (in the case the whole Matrix does not fit in one node memory): two cases:

Vector fits in one node memory:
The matrix is stored in HDFS as a list of (i,j, mij) tuples, while the elements vj of vector v are available to all the Mappers. (The partitions can be made over the matrix rows (one row to each mapper)
Map: ((i, j), mij) pair → (i, mijvj) pair
Reduce: (i, [mi1v1, mi2v2, ..., $m_{in}v_n$]) pair → (i, mi1v1 + mi2v2 + ... + $m_{in}v_n$) pair

Vector does not fit in memory:
In this case we cannot partition on the matrix column.
We have to divide the vector in equal-sized subvectors that do fit in node memory, according to that, we have to divide the matrix in stripes → note that stripe I and subvector I are independent from others.
We can apply the previous alg orithm for each strip / subvector pair.

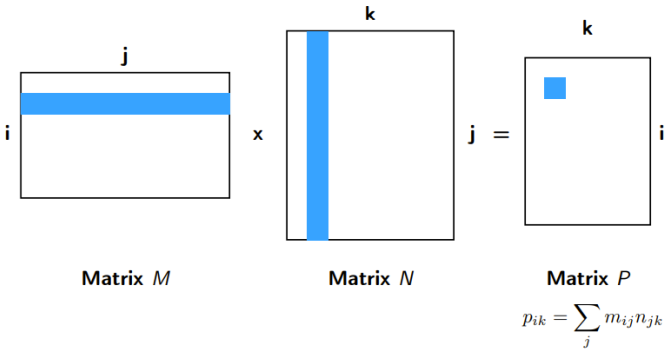


Matrix – Matrix Multiplication:

We can see a matrix as a three attributes relation (row index, column index, value) tuple.
M → (i, j, m_{ij}), N → (j, k, n_{jk})

- large matrices are often sparse (0s) we can omit that tuples.

The product of the matrices can be seen as a natural join over the attribute j followed by product computation, then followed by grouping and aggregation.



1. Start with (i, j, v) and (j, k, w)
2. Compute (i, j, k, v, w)
3. Compute (i, j, k, v * w)
4. Compute (i, k, Σ_j (v * w))

- First stage
  - **Map**: given *(i, j, m_{ij})* produce *(j, (M, i, m_{ij}))*
    given *(j, k, n_{jk})* produce *(j, (N, k, n_{jk}))*
  - **Reduce**: given *(j, [(M, i, m_{ij}), (N, k, n_{jk})])* produce *((i, k), m_{ij} x n_{jk})*
    otherwise do nothing
- Second stage
  - **Map**: identity
  - **Reduce**: produce the sum of the list of values associated with the key

Algorithm 1: The Map Function

```
for each element m_ij of M do
    produce (key, value) pairs as ((i,k),(M,j,m_ij)), for k = 1,2,3,.. up
    to the number of columns of N
for each element n_jk of N do
    produce (key, value) pairs as ((i,k),(N,j,n_jk)), for i = 1,2,3,... up
    to the number of rows of M
return Set of (key, value) pairs that each key, (i,k), has a list with
values (M,j,m_ij) and (N,j,n_jk) for all possible values of j
```

Algorithm 2: The Reduce Function

```
for each key (i,k) do
    sort values begin with M by j in list_M
    sort values begin with N by j in list_N
    multiply m_ij and n_jk for j_th value of each list
    sum up m_ij * n_jk
return (i,k), sum_{j=1} m_ij * n_jk
```

[on the slides there is also an example of execution of the algorithm]

Graphs:

G = (V,E) where G is the Graph, V is the set of vertices (aka nodes) and E is the set of edges (aka links).
Both vertices and edges may contain additional information.

Graph algorithm typically involve:

- **Computation at each node** (using node features, edges features, local link structure)

- **Propagating Computations** between nodes

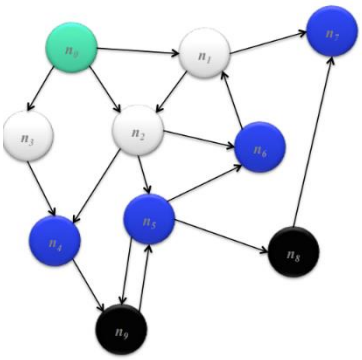
We will see how to represent Graph Data and how to implement Graph Algorithm in MapReduce.

Representing Graphs:

Representing Graphs via Adjacency Matrix:
Can be done with an Adjacency Matrix M of n x n components for a graph of n nodes (n == |V|). In the adjacency matrix m_{ij} is equal to 1 when there is a link from node i to node j.
PROs: adapt to mathematical manipulation; iterating over rows and columns corresponds to computations on outlinks and inlinks.
CONs: lots of zeros and sparse matrices → space wasting

Representing Graphs via Adjacency List:
Adjacency List is obtained by removing all zeros from adjacency matrices. (???)
PROs: compact representation, easy to compute over outlinks
CONs: difficult to compute over inlinks

Shortest Path Algorithm:
In the case of equal edge weights.



Definition: reachable: b is reachable from a if b is on adjacency list of a

Inductive solution:

1. Let assume that we start from the node s, we have DISTANCETO(s) = 0
2. For all nodes p *reachable* from s, we have DISTANCETO(p) = 1
3. For all nodes n reachable from some other set of nodes M, we have DISTANCETO(n) = 1 + min(DISTANCETO(m), for m in set M) (??)

Shortest Path Algorithm implementation:

We adopt as data representation:

- For the **key**: the **node id** *n*
- For the **value**: **(d, adjacency list)** where d is the distance of n from start and adjacency list is the list of nodes directly reachable from n

➔ **Initialization** is performed by setting the value of **d to infinity** except for the start node s

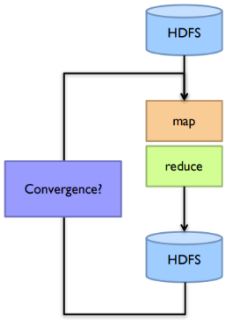
```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)
```

```
1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         dmin ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d
8:             else if d < dmin then
9:                 dmin ← d
10:        M.Distance ← dmin
11:        Emit(nid m, node M)
```

Mapper: receives the tuple (node id, (d, adjacency list)), for each node m in the adjacency list emits (m, 1 + d); additional bookkeeping is needed to keep track of actual path

Sort/Shuffle: groups distances by node id

Reducer: selects the minimum d value from the list of values received for each node id n; (additional bookkeeping is needed to keep track of actual path)



NOTE: This MapReduce algorithm has to be executed many times in order to obtain distance for all the nodes. Each iteration advances the "frontier of reached nodes" by 1 hop. After each iteration we perform a check in order to understand if there is convergence.

NOTE: In order to keep adjacency list of each node, the mapper must emit also (n, adjacency list) for each execution.

Graph Algorithm recipe in general:

- ➔ Represent graphs as adjacency lists
- ➔ Perform **local computations in mapper**
- ➔ Pass along partial results via out-links, using as key the destination node
- ➔ Perform aggregation in reducer on in-links to a node
- ➔ Iterate until convergence: controlled by external "driver"
- ➔ Pass the graph structure between iterations

PageRank Algorithm Implementation:

We have *random surfers*, that are web users that randomly clicks on links in webpages and visit other webpages. PageRank is used to estimate the rank of each webpage on the basis of some parameters.

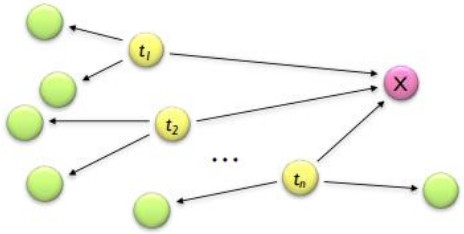
PageRank:

Let's consider:

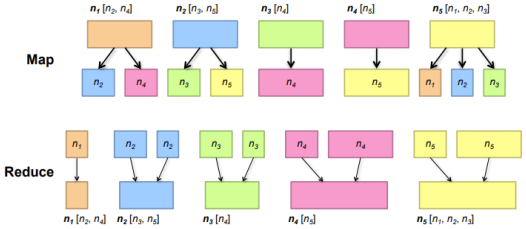
- page x with t_1, …. t_n in-links to it,
- $C(t_i)$ the probability that the in-link to page x present in page t_i is clicked by the user,
- α the probability of random jump to any page of the user from anywhere (the random surfer),
- N the total number of nodes in the graph (pages in the web)

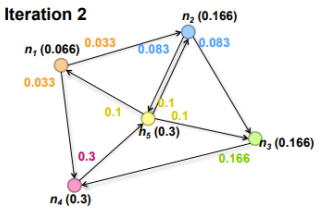
We have that:

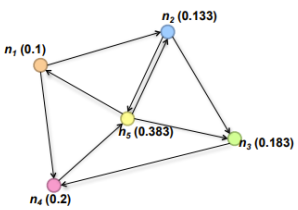
$$PR(x) = \alpha \left(\frac{1}{N}\right) + (1-\alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$



To run PageRank we have to start with seed PR(i) values (i.e. PR value equally distributed among all nodes) and observe that each page i distributes mass of PR to all pages it links to (by the other perspective, each target page adds up mass from any in-bound link to compute its new PR(i)). Then we have to iterate until the values of PR converge.







```
1: class Mapper
2:     method Map(nid n, node N)
3:         p ← N.PageRank/|N.AdjacencyList|
4:         Emit(nid n, N)                          ▷ Pass along graph structure
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, p)                      ▷ Pass PageRank mass to neighbors
```

```
1: class Reducer
2:     method Reduce(nid m, [p1, p2, ...])
3:         M ← ∅
4:         for all p ∈ counts [p1, p2, ...] do
5:             if IsNode(p) then
6:                 M ← p                           ▷ Recover graph structure
7:             else
8:                 s ← s + p                       ▷ Sums incoming PageRank contributions
9:         M.PageRank ← s
10:        Emit(nid m, node M)
```

Slide 07 MapReduce Complexity

Costs in MapReduce:

- **Map function executions costs** (given by the sum of the number of executions performed by each mapper); **is a fixed cost** (does not depend on the implementation)
- **Transmission Cost** for the shuffle and sort phase: every intermediate value is sent across the network of compute nodes. It is proportional to the number of intermediate key-value pairs generated by the mappers. It **is the Communication cost** of a MapReduce algorithm.
- **Reduce cost of execution on reducers**: is the sum of computation cost of execution of each reducer. It **is the Computation Cost** of a MapReduce algorithm.

➔ Map cost is constant and can be included in the transmission costs
➔ The output is rarely large compared to input and intermediate data
➔ We assume that intermediate keys skewness is not a problem

There is a trade-off between Computation and Communication:

Reducer size (q): its upper bound is the size of the list of values (intermediate values) that can appear associated with a key. q is the average computation cost.
If we force to have **many reducers**, each reducer will have to process a smaller list, then we will have high parallelism and the data in charge of the computation of each reducer will entirely fit in main memory (this reduces synchronization (communication and I/O) → **low wall-clock time**

Replication rate (r): is the **number of intermediate key-value tuples** produced by all the mappers **divided** by **the number of original input tuples**

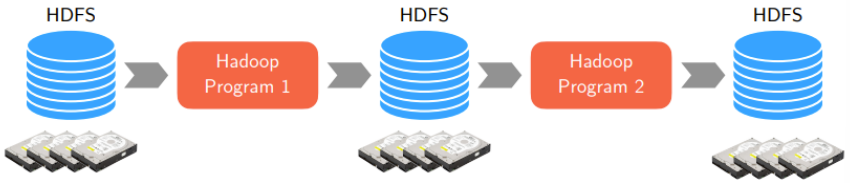
[in the slides there are some examples of q and r calculations (Natural join, Matrices Multiplication, Similarity Join (two approaches)]

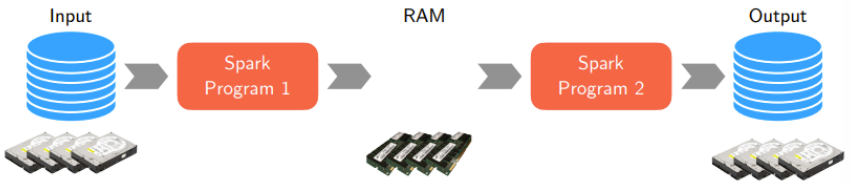
r and q are **inversely proportional** → reducing replication rate will increase reducer size:

- Extreme case: r = 1 and q = 2n: there is a single reducer doing all the comparisons
- Extreme case: r = n and q = 2: there is a reducer for each pair of inputs

Slide 08 Spark

Hadoop always stores the result of a job on HDFS, while Spark let you also keep it in RAM only, if you want to use it as input for another job.

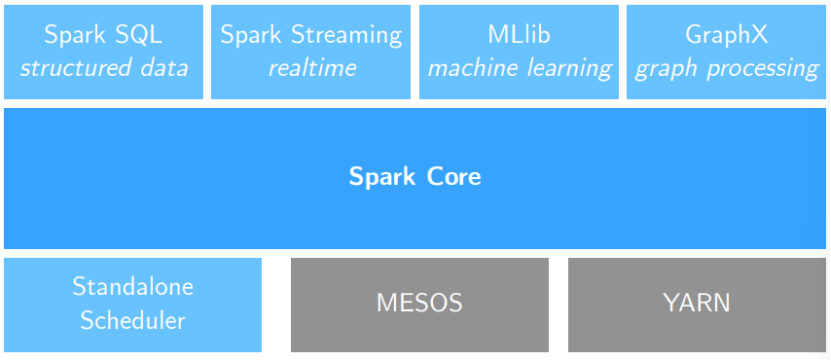




In general distributed programming consists of restricting the programming interface in order to introduce more automatism.
Jobs are expressed as graphs of high level operators.
The **system** is in charge of **choosing how** to **split** each operator into tasks and **where** to **run** each task. Parts are **run multiple times** for **fault recovery**.
The main implementation of that is MapReduce framework.

Limitations of **MapReduce**: **inefficient for multi-pass algorithms**; does not provide efficient primitives for data sharing (in fact the state between steps goes to DFS, that is slow due to replication and disk storage).
An example is the implementation of PageRank, that requires many iterations and needs to repeatedly multiply sparse matrix and vector and hashing together page adjacency lists and rank vector.
→MapReduce is simple, but can require asymptotically more communication or I/O.
MapReduce is useful as an algorithmic framework, but not useful to use directly.

Spark Modules:



Spark Core:
provides basic functionalities like:

- Task scheduling
- Memory management
- Fault recovery
- Storage system interaction

Spark Core provides a data abstraction called Resilient Distributed Dataset (RDD): it is a collection of items distributed across many compute nodes that can be manipulated in parallel.
RDD can be manipulated and built by exploiting Spark Core RDD APIs (RDD written in Scala) available for Java, Python and R.

Spark SQL: extends the Spark RDD API, enable to work with structured data, allows SQL querying.

Spark Streaming: extends the Spark RDD API, to process live streams of data

MLlib: scalable Machine Learning library, implements many distributed algorithms (features extraction classification, regression, clustering, recommendation, …)

GraphX: extends Spark RDD API for manipulation of graphs, allows graph-parallel computations; includes common graph algorithms (e.g. PageRank)

Resilient Distributed Dataset (RDD):

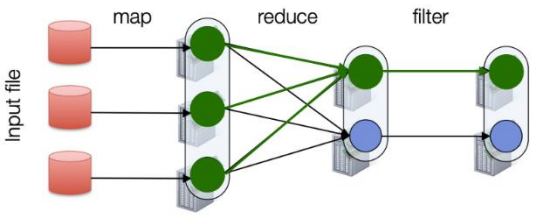
RDD is a **distributed memory abstraction**, is an **immutable collection** of objects spread across the cluster.
Every **RDD** is **divided in**to a number of **partitions**;
Partitions are atomic pieces of information, that *can* be stored on different nodes of a cluster.
RDD are built via parallel transformations (map, filter, …) and their partitioning is user controlled, and their storage type too (can be stored onto memory, disk, …).

RDD can be defined only in a way such that can **be automatically rebuilt on failure**.

RDDs track **lineage** info to rebuild lost data.

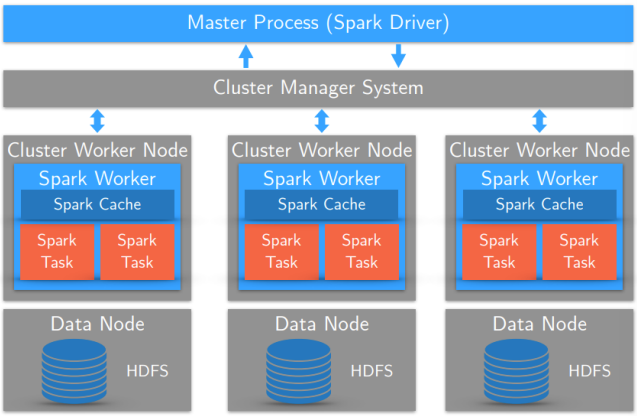
Lineage: every set of transformations applied to an RDD outputs another RDD. The graph of those RDD "states" is called Lineage Graph, and is exploited by Spark to achieve fault-tolerance.

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



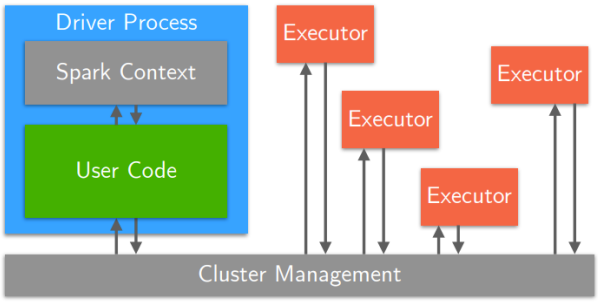
Spark Architecture:

There is a Master Process (called **Spark Driver**) that interacts with the cluster management system (e.g. YARN); the cluster management system controls the cluster worker nodes; inside each cluster worker node there is a **Spark Worker** in execution, which has its own **Spark Cache** and could have some **Spark Tasks** running.



Spark Application Architecture:

A Spark application consists of a **driver process**, and a **set of executor processes**.



Spark Driver: runs the main() function, is the heart of the Spark application, runs in a node of the cluster.
Spark Driver is responsible for:

- maintaining information about the Spark app,
- interacting with the user,
- analysing, distributing and scheduling work across the executors.

A **Spark Executor** is responsible for:

- **Executing code** assigned to it by the driver
- **Reporting the state of computation** back to the driver

In the Spark Driver process we have both the user code and the **Spark Context**.

Spark Context: represents a connection with the cluster system.
Inside the *pySpark shell* a **SparkContext** is created automatically on start and is accessible through the variable *sc*. In a Python script that includes a Spark application, you need to create it as soon as necessary.

sc = SparkContext(appName="MY-APP-NAME", master="local[*]")

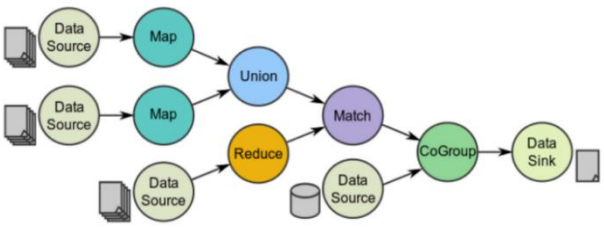
The master argument can assume many values:

- **local**: run in local mode on a single core
- **local[N]:** run in local mode on N cores
- **local[*]:** run in local mode on all cores available
- **yarn**: connect to a YARN cluster

The command line tool **spark-submit** allows to submit jobs across all cluster managers.
The tool accepts the - - master flag to specify the execution mode of the Spark application.

You can create RDD using the sparkcontext method parallelize on a variable (e.g. a list); it is possible to state the number of partitions; you can also create RDD from many external storages (disk, HDFS, AWS S3, …) and textfiles.

Spark programming model is based on **parallelizable operators**, which are higher-order functions that *execute **user-defined code*** **in parallel**.
Data flow is made of any number of data sources, operators and data sinks (which can be used as input for other flows). The job description is based on directed acyclic graphs (DAG).



RDD Operations are of two types: Transformations or Actions:

- **Transformations**: allow to build the logical plan (/data flow structure):
  Transformations are lazy (they do not compute the result right away), are computed only when an action requires a result to be returned to the driver program. Are remembered in their order for the application to the dataset

- **Actions**: trigger the computation → actions instruct Spark to compute a result from a series of transformations. There are three types of actions: to view data; to collect data to native objects of the current programming language; to write to output data sinks

---

RDD Actions:

- **Collect**: returns the elements of the RDD as an **array** to the Spark Driver
- **First**: that returns the first **value** of the RDD
- **Take**: returns the first n (n is specified in the method call) elements of the RDD in an array
  There are also the methods takeOrdered and takeSample

- **Count**: that returns the number of elements of the RDD
  CountByValue returns a dictionary where each key is one of the value appearing in the RDD and each value is the number of occurrences of that value
- **Max / Min**: returns respectively the maximum and the minimum **value** in the RDD
- **Reduce**: aggregates the elements of the RDD using a given function user-specified that must be commutative and associative, in order to be computed correctly in parallel
- **saveAsTextFile**: writes the records of the RDD in a text file (local fs, HDFS or any other Hadoop-supported file system)

Generic RDD Transformations:

- **Distinct**: removes duplicate values
- **Filter**: returns only the RDD records that match some predicate function
- **Sample**: draws a random sample from the RDD (can be with or without replacement)
- **Map and Flatmap**: apply a given function to each RDD element independently
  map transforms an RDD of length n into another RDD of length n
  flatmap allows returning 0, 1 or more elements from each function invocation (i.e. for each RDD row)
- **sortBy**: sorts an RDD
- **union**: merges RDDs
- **intersection**: perform **set intersection** of RDDs

```
words = sc.parallelize("nel mezzo del cammin di nostra vita".split(" "))
sorted_words = words.sortBy(lambda w: len(w))
print(sorted_words.collect()) # this is an action
['di', 'nel', 'del', 'vita', 'mezzo', 'cammin', 'nostra']

data1 = sc.parallelize(range(0,7))
data2 = sc.parallelize(range(3,10))
union = data1.union(data2)
print(union.collect()) # this is an action
[0, 1, 2, 3, 4, 5, 6, 3, 4, 5, 6, 7, 8, 9]

intersection = data1.intersection(data2)
print(intersection.collect()) # this is an action
[3, 4, 5, 6]
```

Key-Value RDD: A key-value RDD is made of tuples records (k,v) where k is the key and v the value.
To create a kv RDD you can either:

- Map over you current RDD to a basic key-value structure
- Use the keyBy to create a key from the current value
- Use the zip transformation to zip together two RDDs

Key-Value RDD Transformations:

- **Lookup**: receives as parameter a key, looks up in the RDD for all the records whose key is the specified one and returns the values associated in an array
- **reduceByKey**: combines all the values of the records that have the same key. Receives as input a function that is used to combine values of the same key.
- **sortByKey**: returns an RDD sorted by key
- **join**: performs an inner-join on the key between two RDDs
  Other joins: fullOuterJoin, LeftOuterJoin, RightOuterJoin, cartesian (is the cross join)

---

wordcount example:

1. Load "comedies.txt" text file into Spark.
2. Transform the lines RDD into a words RDD.
3. Transform each word into a (word, 1) pair.
4. Reduce words by key to sum up word occurrences.
5. Save results as text file.

```
text   = sc.textFile("data/comedies.txt")
words  = text.flatMap(lambda x: x.split(" "))
ones   = words.map(lambda w: ( w, 1 ))
counts = ones.reduceByKey(lambda x, y: x + y)
counts.saveAsTextFile("data/comedies_wordcount.txt")
```

```
import sys
from pyspark import SparkContext


if __name__ == "__main__":
    master = "local"
    if len(sys.argv) == 2:
        master = sys.argv[1]
    sc = SparkContext(master, "WordCount")
    lines = sc.parallelize(["pandas", "i like pandas"])
    result = lines.flatMap(lambda x: x.split(" ")).countByValue()
    for key, value in result.items():
        print("%s %i" % (key, value))
```

Bigram count:

1. Define a function extracting all bigrams from a string of words.
2. Load "comedies.txt" text file into Spark.
3. Transform the lines RDD into a bigrams RDD.
4. Transform each bigram into a (bigram, 1) pair.
5. Reduce bigrams by key to sum up bigram occurrences.
6. Save results as text file.

```
def create_bigrams(line):
    pairs = []
    words = line.lower().split()
    for i in range(len(words) -1):
        pairs.append(words[i] + "_" + words[i + 1])
    return pairs

text = sc.textFile("data/comedies.txt")
bigrams = text.flatMap(create_bigrams)
ones = bigrams.map(lambda b: ( b, 1 ))
counts = ones.reduceByKey(lambda x, y: x + y)
counts.saveAsTextFile("data/comedies_bigramscount.txt")
```

By default, each transformation on an RDD is recomputed each time an action is performed on it.
For efficiency Spark supports also persistence (or caching) of RDDs in memory between operations.

RDD Persistence: when an RDD is persisted, each node that has a partition of that RDD stores the transformed partition and reuses it in other actions on the derived dataset.
In this way future actions are much faster (up to 100x).
In order to enable persistence of modified RDD, use the method *persist()* or *cache()* on it.
Spark cache is fault-tolerant: if a partition is lost, it can be recomputed by applying the same transformations on the original partition.
This RDD caching is a key tool for iterative algorithms and fast interactive use.
The method persist() allow to specify the storage level for the RDD: MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, …
The method cache() calls the persist() method at level MEMORY_ONLY.
Which level of persistence to use: the best choice is to keep in memory only when possible. Use the disk only if the computed transformations are very heavy to be applied again. Use replicated storage only if fast fault recovery is needed.

For performance reasons use Serialization of objects for space-efficiency. (use a fast serialization library).

Spark issues:

- it is hard to handle single jobs with large global variables (e.g. a large lookup table that has to be sent to workers; a large feature vector in a ML algorithm to workers);
- it is difficult to count events that occur during job execution (e.g. counting blank records / corrupt records / …)

➔ closures are re-sent with every job
→ it is inefficient to re-send large data inside the closure to each worker
→closures are one-way: those cannot be exploited by workers for counts or any else

Those problems can be solved by exploiting **Distributed Shared Variables**.

Distributed Shared Variables:

Two types of distributed shared variables:

- **Broadcast variables**: let you save a value in all the workers and reuse it across many Spark actions without resending it to each node of the cluster
- **Accumulators**: add together data from all the tasks into a shared result

Those variables can be used inside user-defined functions (e.g. in a map function on an RDD) and have the defined special properties when running on a cluster.

Broadcast Variables:

Sends efficiently large, read-only values to all workers. Like sending a lookup table to all workers. Can be reused between many Spark operations. Allows to keep read-only variables cached on workers. Ship to each worker once, instead that with every task.

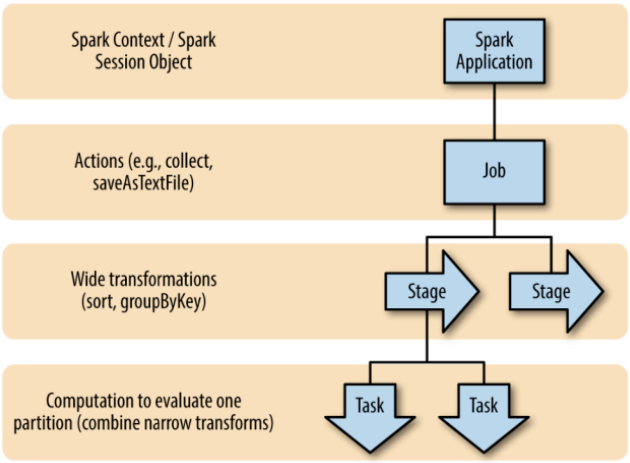
```
# At driver

broadcastVar = sc.broadcast([1, 2, 3])

# At workers (in a closure)

print(broadcastVar.value)

>> [1, 2, 3]
```

Accumulators:

Allow to update a value inside of transformations at worker level and allow to propagate that value to the Spark Driver node in an efficient and fault-tolerant way. Accumulators are added through an **associative** and **commutative** operation and can therefore be used in parallel. Their values are updated effectively only once that the RDD is actually computed.
Accumulators are **write-only** for the workers.

```
# At driver

accumulator = spark.sparkContext.accumulator(0)

# At workers (in a closure)

accumulator.add(3)

# At driver

print(accumulator.value)

>> 3
```

Anatomy of a Spark job:



Each application has its own Spark Context; each Spark action (e.g. collect method) is a job. Each *wide transformation* is a stage (e.g. sort, groupByKey). Each computation on one partition of an RDD for a transformation is a task (the smallest execution unit, represents one local computation; same code on different partitions of data).

The Spark application does not evaluate stages until a Spark action is called. (lazy evaluation).

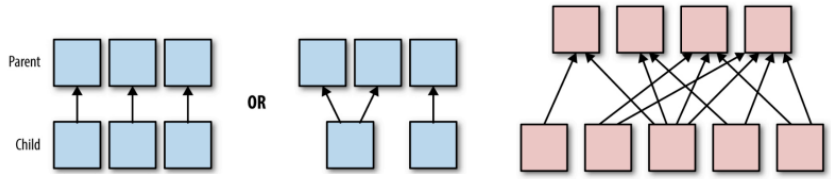
A Spark job is the highest element of Spark's execution hierarchy, it corresponds to one action; each action is called by the Spark Driver program of a Spark application.

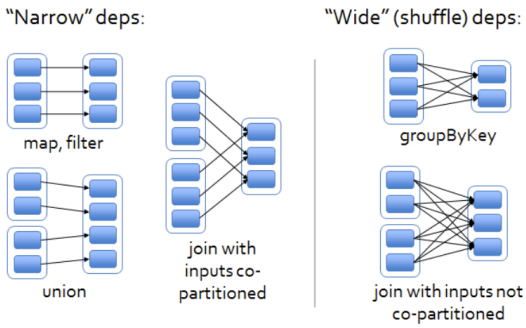
Stages are group of tasks that can be performed together.

Wide transformations define the breakdown of jobs into stages.

Wide and Narrow Transformations: are distinguished for dependencies (narrow or wide deps)

- **Narrow Transformations**: their *dependencies can be determined at design time*, independently of the values of the records in parent partitions. In order for a transformation to be called narrow, each parent partition has at most one child partition. *Can be executed on an arbitrary subset of data without any information on other partitions.*
- **Wide transformations (shuffle)**: *require the data to be partitioned in a particular way* (cannot be executed on arbitrary rows) (e.g. according to the value of their key)





Communication Costs:

Shuffling: requires to solve the *sorting problem* (considering that there is already some ordering in the data we want to sort, which starts out distributed on p machines).
→ an all-to-all communication is needed → it is not granted neither that the data will be sent only once.
The sorted dataset is stored in a distributed manner.

Distributed Sorting:

Each node sends a **uniform sample** (see next) **of its data** to a sorting driver.
That sorting driver uses the samples received from each machine *weighted by the amount of data on each machine* to compute an approximate distribution of the data that has to be sorted.
The obtained distribution is used to compute *p split points*, determined so that the number of records between two split points in the data to be sorted is approximately the same.
The sorting driver broadcasts the split points to the nodes.
Then each machine performs locally a sorting on its data.
By using the received split points, each machine maps which data goes to which machine (assigning an index to each record).
Each machine has its own index assigned, and asks to all the others for its portion of data assigned with the previous step procedure. → in this step an all-to-all communication occurs
Then each machine has to perform a ***p-way* merge** (because data comes from p different nodes) ***of sorted data sets***.

Stream: a stream **σ** is a sequence of m elements where each is an integer between 1 and n, with n < < m.

Uniform Sampling: given a stream **σ**, called f_i the number of occurrences of value *i* in the stream (with i between 1 and n). We can define a probability distribution P on i by $p_i = f_i/m$.
The goal is to output a single element sampled in accordance with the distribution P.

For example, consider the stream σ = (1, 3, 4, 5, 5, 2, 1, 1, 1, 7). In this case p1 = 4/10, p2 = 1/10, p3 = 1/10, p4 = 1/10, p5 = 2/10, p7 = 1/10. The goal would be to output a random variable in accordance to this distribution.

Naïve Sampling Algorithm:

- **Initialization**: for each value i = 1, …, n initialize a counter $f_i = 0$, set m = 0
- **Streaming**: for each element j of the stream, s_j, if s_j == i, increment f_i, always increment m = m + 1
- **Output**: calculate $p_i = f_i/m$; then choose an index i according to the probability distribution P = (p1, …, pn).

To store the **entire stream would take O(m*log(n))** space (because the entire stream is composed of m elements, and because every element can be represented on log(n) bits, since the maximum value in the stream is n).

To store the data structure for this **Naïve Sampling Algorithm** we would need n counters whose value could go between 0 and m (in the case that the stream is composed of one single value repeated m times), so we need **O(n*log(m)) space**. → this is better than the stream size since n < < m.

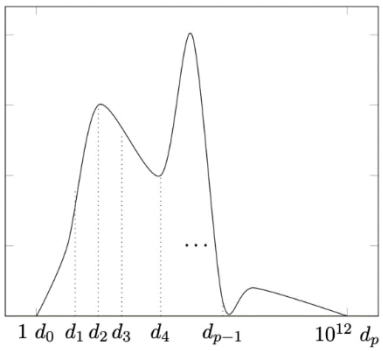
Uniform Sampling Algorithm:

- **Initialization**: set x to null, k to 0.
- **Streaming**: for each s_j element of the stream, increment k = k + 1. With probability 1/k set x = sj (the current element of the stream), else do nothing.
- **Output**: return x

We have to store one variable for x (the sampled value, that can be at most n) and one counter for k (the number of elements of the stream).
So the occupied space for the Uniform Sampling Algorithm is **O(log(n) + log(m))** in total.

Split Points Computation:

The split points could be one near the other for low values and very distant each other for high values.



Sorting (shuffle):

	Hadoop World Record	Spark 100 TB *	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

Developing Spark Applications:

1. Create input RDDs from external data or parallelize a collection in the Driver program
2. Define transformations for the RDDs (like filter() and map() methods) that will be executed lazily
3. Use the cache() method to any intermediate RDD that will be reused
4. Launch the action on each RDD (e.g. count() or collect()) to kick off the parallel execution, which will be optimized and executed by Spark

MLlib Algorithms supported by Spark:

- **classification**: logistic regression, linear SVM, naïve Bayes, least squares, classification tree
- **regression**: generalized linear models (GLMs), regression tree
- **collaborative filtering**: alternating least squares (ALS), non-negative matrix factorization (NMF)
- **clustering**: k-means||
- **decomposition**: SVD, PCA
- **optimization**: stochastic gradient descent, L-BFGS

Gradient Descent implementation in Spark

Considering separable objective functions of the form $\min_w \sum_{i=1}^{n} f_i(w)$ where w belongs to R^d and is the vector of minimizing parameters; $f_i(w)$ is the loss function applied to a training point i. (the loss function can be either linear or non-linear; can be least squares or neural network).

We assume that the number of training points (n) is large (e.g. trillions).

We assume that the number of parameters (d) can fit on a single machine's RAM (e.g. millions of parameters)

1. Start a random vector x0.
2. Then $x_{k+1} \leftarrow x_k - \alpha \sum_{i=1}^{n} \nabla F_i(x_k)$ (where (triangle)F is the gradient vector)

Implementation in Spark:

One training point is stored on a single machine.

The data points are arranged in a matrix, which is divided among machines row-by-row. The matrix is encoded in text.

Input data are not moved among machines.

```
def parsePoint(row):
 tokens = row.split(' ')
 return np.array(tokens[:-1], dtype=float), float(tokens[-1])
w = np.zeros(d)
points = sc.textFile(input).map(parsePoint).cache()
```

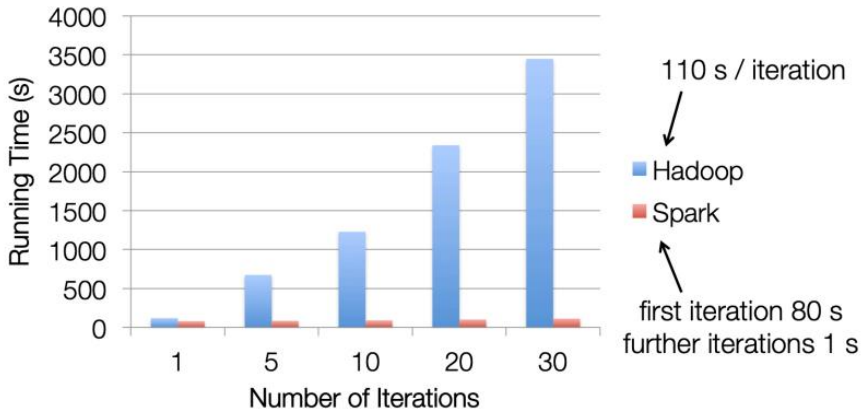
To compute the gradient, we have to perform a computation on each machine and then a summation across machines.

```
def compute_gradient(p, w):
    ...
gradient = points.map(lambda p: compute_gradient(p, w))
                 .reduce(lambda x, y: x + y)
```

(? On a single machine we can have many CPUs so we can avoid to store w for each CPU using the method broadcast() ?)

```
points = sc.textFile(input).map(parsePoint).cache()
w = np.zeros(d)
w_br = sc.broadcast(w)
for i in range(num_iterations):
    gradient = points.map(lambda p: compute_gradient(p, w_br.value))
                     .reduce(lambda x, y: x + y)
   w -= alpha * gradient
   w_br = sc.broadcast(w)
```

Results with logistic regression:



100 GB of data on 50 m1.xlarge EC2 machines